Domain Specific Language for Modular Knitting Pattern
Definitions:
Purl

Chelsea Battell
McMaster University

April 2014
Contents

1 Introduction
  1.1 Knitting Pattern Example .................................................. 1
    1.1.1 Verification Example ............................................... 4
  1.2 Purl Compiler Overview ................................................... 5
  1.3 Error Handling .............................................................. 5

2 Elements of Knitting
  2.1 Cast-On ................................................................. 7
    2.1.1 AST Node ............................................................. 7
    2.1.2 Verification .......................................................... 9
    2.1.3 HTML Generation .................................................. 10
  2.2 Pick-Up ................................................................. 10
    2.2.1 AST Node ............................................................. 10
    2.2.2 Verification .......................................................... 12
    2.2.3 HTML Generation .................................................. 13
  2.3 Row ................................................................. 13
    2.3.1 AST Node ............................................................. 13
    2.3.2 Verification .......................................................... 16
    2.3.3 HTML Generation .................................................. 18
  2.4 Row Elements ........................................................... 19
    2.4.1 AST Node ............................................................. 19
2.11.3 HTML Generation .......................................................... 54
2.12 Pattern Body ................................................................. 55
  2.12.1 AST Node ................................................................. 55
  2.12.2 HTML Generation ....................................................... 56
2.13 Row Repeats ................................................................. 56
  2.13.1 AST Node ................................................................. 56
  2.13.2 HTML Generation ....................................................... 58
2.14 Pattern Section ............................................................. 59
  2.14.1 AST Node ................................................................. 59
  2.14.2 Verification .............................................................. 61
  2.14.3 HTML Generation ....................................................... 62
2.15 Pattern Sample .............................................................. 62
  2.15.1 Sample Definition ...................................................... 63
  2.15.2 Sample Call ............................................................. 66
2.16 Root ........................................................................... 68
  2.16.1 AST Node ................................................................. 68
  2.16.2 HTML Generation ....................................................... 68
3 Lexical Analysis ................................................................. 69
  3.1 Parser .......................................................................... 73
    3.1.1 Pass 1: AST Construction ........................................... 75
    3.1.2 Pass 2: Sample Substitution ....................................... 77
    3.1.3 Pass 3: Pattern Verification ....................................... 80
4 Code Generation ............................................................... 83
  4.1 HTML ....................................................................... 83
5 Discussion ...................................................................... 88
   Appendices ......................................................................... 89
Purl is a language to be used for modular definition and verification of knitting patterns. The syntax is similar to the standard knitting pattern notation provided by the Craft Yarn Council (see [1]). Purl provides constructs not available in the standard notation to allow reuse of segments of patterns.

This report was written using the literate programming (see [3]) tool Literate Programming for Eclipse (LEP, see [5]). The compiler source code, and HTML and CSS for a web page with rudimentary IDE for Purl are “tangled” by the LEP plugin in the Eclipse IDE [2].
1

Introduction

Knitting is a process by which a strand of yarn can be turned into flexible fabric. Patterns are written to record knitting designs and techniques. Simple project patterns using only a few different stitches may be shared orally, but to knit more complicated objects it becomes necessary to document these instructions. The earliest known example of an object being knit using stitches other than the knit stitch dates from the 16th century (purl stitch and yarn over, see [7]). A popular and once standard instruction reference for knitting [4] written in 1886 lists 12 stitches. The stitch reference [9] published in the last five years lists 30 stitches basic (not all included in this project). Improvements in pattern documentation techniques and tools is both necessary for, and conducive to, increased complexity and innovation in knitting patterns.

Patterns are typically written according to the standard [1], and may include extra information, such as the number of active stitches that should be on the needles at the end of each row. This extra information guides the knitter to help avoid mistakes, but this assumes that the pattern has no errors. In the most common case (not including industrial settings), for every pattern written, a pattern designer has to experiment and perform many tedious computations to make sure their pattern is correct. Also, even though segments of a pattern may be reused in many others, they are rewritten in each new pattern.

This project implements a compiler for a language called Purl. The syntax is similar to the standard knitting pattern notation [1]. Purl has features to increase reusability of segments of knitting patterns. The compiler performs automatic verification of the number of stitches in each row and displays this guiding information in the output pattern.

1.1 Knitting Pattern Example

Below is a knitting pattern for a bag [8] in the standard notation. This pattern is broken into two sections: body and handle. Each section begins with how to add the initial stitches to the needle (cast-on for the body, pick-up stitches for the handles). The rows of the pattern are defined next, and each section ends with either a bind-off or a join. A single asterisk means to repeat stitches as directed. Two asterisks means to repeat the rows between the double asterisk and the row repeat instructions.
Body

Cast–on 8 sts circular.

Rnd 1 (RS): *K, yo, K; rep from * to end.
Rnd 2 (RS): *K; rep from * to end.
Rnd 3 (RS): (K, yo, K, yo, K) 4 times.
Rnd 4 (RS): *K; rep from * to end.
Rnd 5 (RS): (K, yo, K3, yo, K) 4 times.
Rnd 6 (RS): *K; rep from * to end.
Rnd 7 (RS): (K, yo, K5, yo, K) 4 times.
Rnd 8 (RS): *K; rep from * to end.
Rnd 9 (RS): (K, yo, K7, yo, K) 4 times.
Rnd 10 (RS): *K; rep from * to end.
Rnd 11 (RS): (K, yo, K9, yo, K) 4 times.
Rnd 12 (RS): *K; rep from * to end.
Rnd 13 (RS): (K, yo, K11, yo, K) 4 times.
Rnd 14 (RS): *K; rep from * to end.
Rnd 15 (RS): (K, yo, K13, yo, K) 4 times.
Rnd 16 (RS): *K; rep from * to end.
Rnd 17 (RS): (K, yo, K15, yo, K) 4 times.
Rnd 18 (RS): *K; rep from * to end.
Rnd 19 (RS): (K, yo, K17, yo, K) 4 times.
Rnd 20 (RS): *K; rep from * to end.
Rnd 21 (RS): (K, yo, K19, yo, K) 4 times.
Rnd 22 (RS): *K; rep from * to end.
Rnd 23 (RS): (K, yo, K21, yo, K) 4 times.
Rnd 24 (RS): *K; rep from * to end

**
Rnd 25 (RS): *k2tog, yo; rep from * to end.
Rnd 26 (RS): *K; rep from * to end.
rep from ** 30 times

**
Rnd 27 (CC) (RS): *K; rep from * to end.
Rnd 28 (CC) (RS): *P; rep from * to end.
rep from ** 4 times

Bind–off 100 sts.

Handle

Pick–up 10 sts from body top.

**
Row 1 (CC) (RS): *K; rep from * to end.
Row 2 (CC) (WS): *P; rep from * to end.
rep from ** 2 times
Row 3 (RS): K, k2tog, K4, k2tog, K.

**
Row 4 (CC) (WS): *K; rep from * to end.
Row 5 (CC) (RS): *P; rep from * to end.
rep from ** 100 times
Row 6 (WS): K, M1, K6, M1, K.

**
Row 7 (CC) (RS): *K; rep from * to end.
Row 8 (CC) (WS): *P; rep from * to end.
rep from ** 2 times
Join 10 sts to opposite side of body top.

Pattern Example 1.1: Market Bag

The program in Purl to generate this pattern is below. All of the blocks beginning with “sample” are a new construct introduced to increase reusability of parts of knitting patterns. These samples could be used in many other patterns and the definition of the market bag pattern (below the samples) becomes much simpler.

```purl
sample circle with n, max
/ n < max:
   rnd : [K, YO, K n, YO, K] 4.
   rnd : *K; to end.
   circle with n + 2, max.

sample diagonalLace with n:
   **
   rnd : *K2T, YO; to end.
   rnd : *K; to end.
   repeat n

sample garterStitchCC with n, type
/ type = 0:
   **
   row CC : *K; to end.
   row CC : *P; to end.
   repeat n
/ type = 1:
   **
```
The constructs used in this example are discussed in more detail throughout this report.

1.1.1 Verification Example

In the section for the body of the bag, the number of stitches is increasing for all but the last couple of rows. It can be difficult for a knitter to keep track of the number of stitches they should have on the needle. Rows in this pattern in the standard notation could be written with the stitch count at the end of each row as:
1.2 Purl Compiler Overview

The Purl compiler consists of a three-pass top-down parser and a back-end that generates a knitting pattern in the standard notation in HTML. The first pass constructs an abstract syntax tree according to the provided grammar and is implemented using recursive descent parsing techniques as described in [3]. The lexer is a module used by this pass. Errors and warnings reported by the first pass are only lexical and syntactic errors. Pass two traverses the syntax tree depth first, replacing all variables and constructs that are purely elements of Purl and are not expressible in the standard notation. These constructs will be discussed when exploring the individual pattern elements. In the third pass, the syntax tree is again traversed depth first. All verification occurs in this pass and errors indicate problems in the structure of the knitting pattern. A global State object is used throughout parsing to track information necessary for error reporting, such as section name, position in code, and row number in the generated pattern. It is also used in the verification pass to track the pattern orientation, width, and row index, and to update nodes with these values as necessary.

The reason for breaking up parsing into three passes is because a syntax tree representation of the pattern is much easier to manipulate and verify. A main feature of Purl is the ability to define modular and parametrized segments of patterns, through the pattern sample construct introduced by this language (see 2.15), so a second pass is used for trimming nodes representing sample calls. Also, there are some challenges in verifying a pattern. It is necessary that every row works all of the stitches of the previous row, but there are some pattern constructs which work a number of stitches that depends on the width of the current row. Since we allow modular pattern definitions and parameterized segments of patterns, this verification cannot be done in a single pass over the source language.

1.3 Error Handling

Below are explanations of many of the strategies used by the compiler for handling errors, including panic-mode recovery approximately as discussed in [3]. They are covered here to avoid redundant explanations later.

1. Whenever a character symbol is expected, there are some characters that are considered likely errors. These likely errors will generate a warning, and compilation will continue as usual.

| Expected | Likely typo |
|----------|-------------|
| ;        | ;           |
| ;        | :           |
| .        | ,           |
| ,        |             |
| **       | *           |

2. If a keyword is expected and an ident is found, then a typo is assumed. In this case a warning is created, and compilation continues.

3. Any other unexpected symbols generate an error and the lexer will scan to the end of the production (usually the period or comma symbols) and return the node to resume compilation at the next sibling. For certain stitches (as will be noted later), there is no reliable symbol delimiting siblings. In this case, the lexer will scan to the end of the parent production.
If an unexpected keyword is found, then the error message reports an invalid use of keyword.
If an unexpected ident symbol is found, then the error message reports an invalid use of ident.
If an unexpected character symbol is found, then the error message reports an invalid use of character.

<Unexpected Symbol Error 1>

```c
if (Sym.type == SymType.Ident) {
    AddMsg(MsgType.Error, node, "Invalid use of ident " + Sym.value + ".");
} else if (hasOwnValue(KeywordSym, Sym.type)) {
    AddMsg(MsgType.Error, node, "Invalid use of keyword " + Sym.value + ".");
} else if (hasOwnValue(CharSym, Sym.type)) {
    AddMsg(MsgType.Error, node, "Invalid use of \" + Sym.value + "\ character.");
}
```

**USED IN:**
- Ast Construction Pass on page 75
- Pattern Parse Pattern on page 52
- Pattern Parse Title on page 52
- Colon Separator on page 15
- Cast-On Parse Keyword on page 8
- Cast-On Parse Value on page 8
- Period Terminator on page 7
- Pick-Up Parse Keyword on page 11
- Pick-Up Parse Value on page 11
- Pick-Up Parse Origin on page 12
- Pick-Up Parse Origin on page 12
- Row Definition Parse Row Type on page 14
- Row Element Parse on page 15
- Stitch Op Parse on page 20
- Undetermined Stitch Repeat Parse Open on page 42
- Undetermined Stitch Repeat Parse Close on page 43
- Undetermined Stitch Repeat Parse To on page 43
- Undetermined Stitch Repeat Parse Instruction on page 44
- Fixed Stitch Repeat Parse Open on page 39
- Fixed Stitch Repeat Parse Close on page 39
- Compound Stitch Parse Open on page 36
- Compound Stitch Parse Close on page 36
- Bind-Off Parse BO on page 46
- Bind-Off Parse Count on page 47
- Join Parse Keyword on page 49
- Join Parse Count on page 49
- Join Parse Destination on page 50
- Join Parse Destination on page 50
- RowRepeatParse Open on page 57
- RowRepeatParse Close on page 57
- Section Parse Section on page 59
- Section Parse Title on page 60
- Section Content Parse on page 60
- Section Content Parse on page 60
- Sample Def Parse Sample on page 63
- Sample Def Parse Params on page 64
- Sample Def Parse Params on page 64
- Sample Call Parse Ident on page 66
- Sample Call Parse Params on page 66
Elements of Knitting

This section explores each element of a knitting pattern and its corresponding representation throughout compilation. This organization allows for additions of new features in a single location, and provides isolated explanations of each knitting concept.

2.1 Cast-On

To begin a knitting project, it is necessary to add stitches on to the needles. This is called a “cast-on”, and is essentially a number of loops made on one of the needles such that each loop is connected to its adjacent loops.

2.1.1 AST Node

\[
\text{co} ::= \text{CO} \langle \text{Nat} \rangle \left[ \text{`circular'} | \text{`provisional'} \right] \text{'}.
\]

The syntax for a cast-on is similar to the standard notation, but replacing cast-on with CO (also commonly used in patterns) and not requiring “sts” to be explicit.

CO 8 circular.

Purl Example 2.1: Market Bag Body Cast-On

```javascript
var CoParse = function () {
    var node = { type : NodeType.CastOn, value : 0, line : State.line };
    <<Cast-On Parse Keyword 3>>
    <<Cast-On Parse Value 4>>
};
```
The “CO” keyword is used to declare a cast-on for a pattern.

Following the cast-on keyword, a natural number is given as the number of stitches that will be added to the needle.
The default cast-on type is flat, but the grammar also allows a circular or provisional cast-on. A circular cast-on is used when knitting a circular-shaped object and beginning knitting from the center of the circle. A provisional cast-on is used if the cast-on will be removed later and the project will be worked in the opposite direction.

```javascript
if (Sym.type == KeywordSym.CastOnCirc) {
    node.coType = CoType.Circular;
    nextSym();
} else if (Sym.type == KeywordSym.CastOnProv) {
    node.coType = CoType.Provisional;
    nextSym();
} else {
    node.coType = CoType.Flat;
}
```

The end of a cast-on is marked by a period.

```javascript
if (Sym.type == CharSym.Period) {
    nextSym();
} else if (Sym.type == CharSym.Comma) {
    AddMsg(MsgType.Warning, node, "Use ". symbol at end of "+ node.type + ".");
    nextSym();
} else {
    AddMsg(MsgType.Error, node, "Missing ". symbol at end of "+ node.type + ");
    scanToSym(CharSym.Period);
    nextSym();
}
```

### 2.1.2 Verification

When a cast-on is verified, the initial side, row width, and row index are set in the State object.

```javascript
var VerifyCastOn = function(node) {
    var
```
State.side = SideType.RS;
State.width = node.value;
State.rowIndex = 1;
}

2.1.3 HTML Generation

```javascript
var WriteCo = function(node) {
  var coType = node.coType !== null && node.coType.length > 0 ? " " + node.coType : "";
  return AddElement(TagType.Div, ClassType.CastOn, "Cast-on " + node.value + " sts" +
   coType + ".");
}
```

2.2 Pick-Up

It is possible to begin knitting off of a completed knitted object by picking up stitches. This is done by using one needle to pull loops of yarn through spaces along the edge that stitches are to be picked up from, so that new active stitches are on the needle.

2.2.1 AST Node

\[
\langle pu \rangle ::= \text{'PU'} \langle \text{Nat} \rangle \text{'from'} \langle \text{String} \rangle \text{'}
\]

```
PU 10 from "Body top".
```

Purl Example 2.2: Market Bag Handle Pick-Up

```javascript
var PuParse = function() {
  var node = { type: NodeType.PickUp, value: 0, line: State.line }

<<Pick-Up Parse Keyword 10>>
<<Pick-Up Parse Value 11>>
```
The “PU” keyword is used at the beginning of a declaration to pick-up stitches.

Following the pick-up keyword, a natural number is given as the number of stitches that will be added to the needle.
Next we expect the keyword “from”, followed by a string directing where the stitches should be picked up from.

```javascript
if (Sym.type == KeywordSym.From) {
    nextSym();
} else {
    /**/ Unexpected Symbol Error 1
    scanToSym(CharSym.Period);
    nextSym();
    return node;
}

if (Sym.type == SymType.String) {
    node.origin = Sym.value;
    nextSym();
} else {
    /**/ Unexpected Symbol Error 1
    AddMsg(MsgType.Error, node, "Missing pick-up origin.");
    scanToSym(CharSym.Period);
    nextSym();
    return node;
}
```

The end of a pick-up is marked by a period.

### 2.2.2 Verification

When a pick-up is verified, the initial side, row width, and row index are set in the `State` object.

```javascript
var VerifyPickUp = function(node) {
    State.side = SideType.RS;
    State.width = node.value;
    State.rowIndex = 1;
};
```

The end of a pick-up is marked by a period.
2.2.3 HTML Generation

```javascript
var WritePu = function(node) {
    return AddElement(TagType.Div, ClassType.CastOn, "Pick-up " + node.value + " sts from " + node.origin + ".");
};
```

**USED IN:** code/codegen.js on page 83

2.3 Row

Once stitches have been added to the needles, it is possible to begin the body of the pattern. Typically knitting is worked from the left needle to the right needle. Knitting by hand may be done flat or in the round. Both of these methods are worked “horizontally”, meaning an object is created as a sequence of knit rows. Flat knitting is done with two straight needles which can be thought of as two stacks since the first stitch worked in each row is the last created in the previous row. Circular knitting with a circular needle (two needles connected with a cable) can be thought of as two queues since the first stitch worked in each row is the first stitch that was created in the previous row.

The “active” stitches are the loops on the needles. These are on the left needle at the start of a row. They are considered active because if removed from the needle, they could be pulled out from whatever stitches they support below. A stitch is worked by pulling a loop of yarn through an active stitch and then dropping that active stitch from the left needle. This previously active stitch will be anchored by the loop pulled through it, and that loop is now an active stitch on the right needle. A row will list the stitches required to work all of the active stitches from the left needle to the right needle.

2.3.1 AST Node

```
<rowDef> ::= ('row' | 'rnd') ['MC' | 'CC'] ':' <rowElem> (',', <rowElem>)* '.'
```

```javascript
var RowDefParse = function() {
    var node = { type : NodeType.Row, children : [], line : State.line };

    <<Row Definition Parse Row Type 16>>
    <<Row Definition Parse Color 17>>
    <<Colon Separator 18>>
    <<Row Definition Parse Row Elements 19>>
    <<Period Terminator 6>>
```
A row begins with the row type of either “row” or “rnd”. In flat knitting the row type “row” is used, meaning once all the stitches on the left needle have been worked (or popped), the whole object is turned over, the left and right needles are swapped, and the process can begin again (the last stitch pushed on the right needle is now the first popped off the left). In circular knitting the row type “rnd” is used, meaning once the stitches on the left needle have been worked and new stitches are on the right needle, all the stitches are moved along the cable so that they are worked in a FIFO fashion. A default row type of “row” is assumed if an unexpected symbol is found.

The body section in the market bag pattern example is knit in the round, and the handle is knit flat.

**Purl Example 2.3: Market Bag Body Row**

```
rnd: *K, YO, K; to end.
```

**Purl Example 2.4: Market Bag Handle Row**

```
row: K, K2T, K 4, K2T, K.
```

---

```java
if (Sym.type == KeywordSym.Row) {
    node.rowType = RowType.Row;
    nextSym();
} else if (Sym.type == KeywordSym.Rnd) {
    node.rowType = RowType.Rnd;
    nextSym();
} else {
    AddMsg(MsgType.Error, node, "Invalid row type specified.");
    node.rowType = RowType.Rnd;
    nextSym();
}
```
The pattern writer can optionally specify which yarn color is to be used for a row. Currently only a main color, 'MC', and a contrasting color, 'CC' are available in the language. The main color is assumed by default.

if (Sym.type == KeywordSym.ColorMain) {
    node.color = ColorType.Main;
    nextSym();
} else if (Sym.type == KeywordSym.ColorContrast) {
    node.color = ColorType.Contrast;
    nextSym();
}

A colon symbol separates the row declaration from its content.

if (Sym.type == CharSym.Colon) {
    nextSym();
} else if (Sym.type == CharSym.Semicolon) {
    var msg = "Use ":\" symbol before listing " + node.type + " elements.";
    AddMsg(MsgType.Warning, node, msg);
    nextSym();
} else {
    <<Unexpected Symbol Error 1>>
    var msg = "Missing ":\" symbol before listing " + node.type + " elements.";
    AddMsg(MsgType.Error, node, msg);
    scanToSym(CharSym.Period);
}

The elements of a row of a pattern are separated by commas. The syntax tree node for each row element is added to an array representing the children of the row.

node.children.push(RowElemParse());

while (Sym.type == CharSym.Comma) {
    nextSym();
    node.children.push(RowElemParse());
}
The end of a row definition is marked by a period.

### 2.3.2 Verification

To verify a row, first the node needs to have row index and side properties set, as these are used in pattern output.

```javascript
node.index = State.rowIndex;
node.side = State.side;
```

The row verification function uses a `rowState` object that is updated upon verification of each child of the row (stitches, compound stitches, and stitch repeats).

```javascript
var rowState = { initialWidth : State.width, workedSt : 0, stChange : 0 };
VerifyRowElemChildren(node, rowState);
```

Details on verification of specific row elements is discussed in the row elements section (see 2.4).

```javascript
var VerifyRowElemChildren = function(node, rowState_0) {
  var rowState_1 = { initialWidth : rowState_0.initialWidth, workedSt : 0, stChange : 0 };
  if (node.children != null) {
    for (var i = 0; i < node.children.length; i++) {
      VerifyRowElem(node.children[i], rowState_1);
    }
  }
  rowState_0.workedSt += rowState_1.workedSt;
};
```
The verification error that will be caught at this level is an incorrect number of worked stitches. This means that the stitches specified for this row either use less or require more stitches than exist from the previous row.

```javascript
if (rowState.workedSt != State.width) {
    var msg = rowState.workedSt + " sts worked over " + State.width + " sts.";
    AddMsg(MsgType.Verification, node, msg);
}
```

Once the children of the row have been verified, the row width of the node and the State object are updated according to the stitch change caused by this row. If the row type of this row is “row”, then the project is to be flipped at the end of the row, so the side is changed. The State.rowIndex is also incremented by one before we move to the next sibling.

```javascript
node.width = rowState.workedSt + rowState.stChange;
State.width = node.width;
if (node.rowType == RowType.Row) {
    if (State.side == SideType.RS) {
        State.side = SideType.WS;
    } else if (State.side == SideType.WS) {  
        State.side = SideType.RS;
    }
}
State.rowIndex += 1;
```

```javascript
var VerifyRow = function(node) {
    <<Verify Row Setup 20>>
    <Verify Row Worked Stitches Error 23>
    <Verify Row State Update 24>
    <Verify Row 25>
    <<Verify Row Setup 20>>
}```
2.3.3 HTML Generation

The written row includes the row type, side, color if explicitly given, row index, children, and the number of active stitches at the end of the row.

```
var WriteRow = function(node) {

    var result = [];

    result.push(OpenElement(TagType.Div, ClassType.Row));

    if (node.rowType == RowType.Row) {
        result.push("Row");
    } else if (node.rowType == RowType.Rnd) {
        result.push("Rnd");
    }

    result.push(node.index);

    if (node.color == ColorType.Main) {
        result.push("(MC)");
    } else if (node.color == ColorType.Contrast) {
        result.push("(CC)");
    }

    result.push("(" + node.side + ")\:" + ";

    var content = [];

    if (node.children != null) {
        for (var i = 0; i < node.children.length; i++) {
            content.push(WriteNode(node.children[i]));
        }
    }

    return {result: result, content: content};
};
```
2.4 Row Elements

Children of a row are grouped as row elements. The <rowElem> production is used to provide structure to the language: it is necessary to not allow nesting of undetermined stitch repeats (see 2.8), but it is desirable to allow fixed repeats (see 2.7) to have fixed stitch repeats as children. The bottom-level row elements are [basic stitch] [compound stitch] [fixed stitch repeat] and [undetermined stitch repeat].

2.4.1 AST Node

\[
\langle \text{rowElem} \rangle := \langle \text{stitchOp} \rangle \mid \langle \text{uStRep} \rangle
\]

\[
\langle \text{stitchOp} \rangle := \langle \text{fixedStRep} \rangle \mid \langle \text{compSt} \rangle \mid \langle \text{basicSt} \rangle
\]

The parse function corresponding to the <rowElem> production determine what <rowElem> the current symbol is a first symbol of, and passes up the result of the appropriate parse function. If the current symbol is a basic stitch, an open angle bracket, or open parentheses, then we need to then parse according to the <stitchOp> production. An asterisk is the only possible first symbol of an undetermined stitch repeat, so for an asterisk we parse according to the <uStRep> production.

```
var RowElemParse = function () {
  var node = {};

  if (hasOwnProperty(StitchSym, Sym.type))
    || Sym.type == CharSym.OpenAngle
  || Sym.type == CharSym.OpenBrack)
    node = StitchOpParse();
  else if (Sym.type == CharSym.Asterisk)
```
node = UStRepParse();
} else {
    <<Unexpected Symbol Error 1>>
    AddMsg(MsgType.Error, node, "Invalid row element.");
    scanToSym(CharSym.Period);
}

    return node;
};

var StitchOpParse = function() {

    var node = [];

    if (Sym.type == CharSym.OpenBrack) {
        node = FixedStRepParse();
    } else if (Sym.type == CharSym.OpenAngle) {
        node = CompStParse();
    } else if (hasOwnValue(StitchSym, Sym.type)) {
        node = BasicStParse();
    } else {
        <<Unexpected Symbol Error 1>>
        var msg = sym.value + " is not a known stitch, start of stitch repeat or compound stitch."
        AddMsg(MsgType.Error, node, msg);
        scanToSym(CharSym.Period);
    }

    return node;
};

2.4.2 Verification

Row elements are verified according to the row element node type. Details are covered in their respective sections.
```
var VerifyRowElem = function(node, rowState_0) {

    var rowState_1 = {initialWidth: rowState_0.initialWidth, workedSt: 0, stChange: 0};

    var rep = 1;
    if (node.repCount !== null) {
        VerifyExpression(node.repCount);
        if (node.repCount.value > 1) {
            rep = node.repCount.value;
        }
    }

    var num = 0;
    if (node.num !== null) {
        VerifyExpression(node.num);
        if (node.num.value > 0) {
            num = node.num.value;
        }
    }

    switch (node.type) {
        case NodeType.FixedStRep:
            <<Verify Fixed Stitch Repeat 65>>
            break;

        case NodeType.UStRep:
            <<Verify Undetermined Stitch Repeat 73>>
            break;

        case NodeType.CompSt:
            <<Verify Compound Stitch 59>>
            break;

        case NodeType.Knit:
        case NodeType.Purl:
        case NodeType.KnitTBL:
        case NodeType.PurlTBL:
        case NodeType.KnitBelow:
        case NodeType.PurlBelow:
        case NodeType.Slip:
        case NodeType.SlipKW:
        case NodeType.SlipPW:
```
2.5 Basic Stitches

As discussed in 2.3, a stitch is typically formed by pulling the trailing yarn through an active stitch on the left needle, creating a new stitch to the right needle. Any active stitches on the left needle that the right needle passes through are dropped off the left needle once the stitch is complete. Variations in the direction the right needle passes through the last active stitch and whether the yarn is in front or behind the needle allow for different stitches to be created. More complex stitches are created by pulling the trailing yarn through some other location in the fabric, or through multiple active stitches, rather than a single active stitch.

A stitch that adds more new stitches to the right needle than are removed from the left is called an increase. Similarly, a stitch that adds fewer stitches to the right needle than it drops off the left is called a decrease. We will say that the number of stitches dropped off of the left needle is the number of worked stitches, since it is the number of stitches from the previous row that have been processed. We say that the difference a stitch makes to the width is the stitch change.

Due to a desire to adhere to the standard knitting pattern and stitch notation and allow flexibility in the naming of identifiers in the language, lexing and parsing of stitches has been treated differently than the rest of the language. Some stitches have numeric parameters in the middle of the stitch notation. Consider the stitch K2T. This represents a knit 2 together stitch, but any natural number is valid. A string of this form is allowed in the structure of identifiers (e.g. K1abc is an acceptable identifier). To reserve certain strings with parameterized segments as stitches, an attempt is made to match an identifier string to regular expressions for each stitch before assuming it is an ident symbol. So the stitch type is determined by the lexer, but the specific information contained in the stitch string is then extracted in the stitch parse function.
The example below is a row from the market bag pattern. After the colon is the list of stitches to be created for this row.

```
row : K, K2T, K 4, K2T, K.
```

Purl Example 2.5: Market Bag Handle Row

### 2.5.1 AST Node

```
var stParts = Sym.value.split(/([1-9][0-9]*)/);
```

*USED IN:* Basic Stitch Parse on page 23

A stitch may be optionally followed by a natural number expression to indicate the number of times a stitch should be repeated.

```
if (Sym.type == SymType.Ident || Sym.type == SymType.Nat) {
    node.repCount = ExpressionParse();
}
```

*USED IN:* Compound Stitch Parse on page 37, Basic Stitch Parse on page 23

```
var BasicStParse = function() {
    var node = { line : State.line };

    switch (Sym.type) {
        case StitchSym.Knit:
            <<Knit 33>>
            break;
        case StitchSym.Purl:
            <<Purl 36>>
            break;
        case StitchSym.KnitTBL:


```
break;
case StitchSym.PurlTBL:
  <<PurlTBL 37>>
  break;
case StitchSym.KnitBelow:
  <<KnitBelow 35>>
  break;
case StitchSym.PurlBelow:
  <<PurlBelow 38>>
  break;
case StitchSym.Slip:
  <<Slip 39>>
  break;
case StitchSym.SlipKW:
  <<SlipKW 40>>
  break;
case StitchSym.SlipPW:
  <<SlipPW 41>>
  break;
case StitchSym.YarnOver:
  <<YarnOver 42>>
  break;
case StitchSym.KnitFB:
  <<KnitFB 43>>
  break;
case StitchSym.PurlFB:
  <<PurlFB 44>>
  break;
case StitchSym.Make:
  <<Make 45>>
  break;
case StitchSym.MakeL:
  <<MakeL 46>>
  break;
case StitchSym.MakeR:
  <<MakeR 47>>
  break;
case StitchSym.KnitTog:
  <<KnitTog 48>>
  break;
case StitchSym.PurlTog:
  <<PurlTog 49>>
  break;
case StitchSym.SSK:
break;
case StitchSym.SSP:
break;
case StitchSym.PSSO:
break;
default:
    node.workedSt = 0;
    node.stChange = 0;
    break;
}

nextSym();

return node;

}
Knit  The most ubiquitous stitch.

Needle Entry
  Top active stitch, from left

Yarn Position
  Back

Effect

<Knit 33>

node.type = NodeType.Knit;
node.workedSt = 1;
node.stChange = 0;

USED IN: Basic Stitch Parse on page 23

Knit Through Back Loop

Needle Entry
  Top active stitch, from right

Yarn Position
  Back

Effect

<KnitTBL 34>

node.type = NodeType.KnitTBL;
node.workedSt = 1;
node.stChange = 0;

USED IN: Basic Stitch Parse on page 23

Knit Below  Stitch with parameter num

Needle Entry
  num stitches below top active stitch, from right

Yarn Position
  Back

Effect

<KnitBelow 35>

node.type = NodeType.KnitBelow;
node.num = stParts[1];
node.workedSt = 1;
node.stChange = 0;

USED IN: Basic Stitch Parse on page 23
**Purl**

**Needle Entry**
Top active stitch, from right

**Yarn Position**
Front

**Effect**

```
<Purl 36>
```
node.type = NodeType.Purl;
node.workedSt = 1;
node.stChange = 0;

**USED IN:** Basic Stitch Parse on page 23

**Purl Through Back Loop**

**Needle Entry**
Top active stitch, from left

**Yarn Position**
Front

**Effect**

```
<PurlTBL 37>
```
node.type = NodeType.PurlTBL;
node.workedSt = 1;
node.stChange = 0;

**USED IN:** Basic Stitch Parse on page 23

**Purl Below** Stitch with parameter *num*

**Needle Entry**
*num* below top active stitch, from right

**Yarn Position**
Front

**Effect**

```
<PurlBelow 38>
```
node.type = NodeType.PurlBelow;
node.num = stParts[1];
node.workedSt = 1;
node.stChange = 0;

**USED IN:** Basic Stitch Parse on page 23
Slip

Needle Entry
Top active stitch, from left

Yarn Position
Back

Notes
No loop pulled through

Effect

<Slip 39>

```java
node.type = NodeType.Slip;
node.workedSt = 1;
node.stChange = 0;
```

Used in: Basic Stitch Parse on page 23

Slip Knitwise The default slip stitch

Needle Entry
Top active stitch, from left

Yarn Position
Back

Notes
No loop pulled through

Effect

<SlipKW 40>

```java
node.type = NodeType.SlipKW;
node.workedSt = 1;
node.stChange = 0;
```

Used in: Basic Stitch Parse on page 23

Slip Purlwise

Needle Entry
Top active stitch, from right

Yarn Position
Front

Notes
No loop pulled through

Effect

<SlipPW 41>

```java
node.type = NodeType.SlipPW;
node.workedSt = 1;
node.stChange = 0;
```

28
**Yarn Over**  An increase

**Yarn Position**  Back

**Instruction**  Wrap yarn counterclockwise around the right needle

**Effect**

```
<YarnOver 42>

node.type = NodeType.YarnOver;
node.workedSt = 0;
node.stChange = 1;
```

**Knit Front and Back**  An increase

**Instruction**  In the same active stitch, knit then knit through back loop

**Effect**

```
<KnitFB 43>

node.type = NodeType.KnitFB;
node.workedSt = 1;
node.stChange = 1;
```

**Purl Front and Back**  An increase

**Instruction**  In the same active stitch, purl then purl through back loop

**Effect**

```
<PurlFB 44>

node.type = NodeType.PurlFB;
node.workedSt = 1;
node.stChange = 1;
```

**Make**  An increase with parameter *num*

**Setup**  From back, left needle picks up vertical bar between top left and top right active stitches
**Instruction**
Knit the stitch picked up by the left needle, repeat *num* times

**Effect**

```
<Make 45>
node.type = NodeType.Make;
node.num = stParts[1];
node.workedSt = 0;
node.stChange = 1;
```

*Used in:* [Basic Stitch Parse on page 23](#)

**Make Left** The default “make” stitch, an increase with parameter *num*

**Setup**
From back, left needle picks up vertical bar between top left and top right active stitches

**Instruction**
Knit the stitch picked up by the left needle, repeat *num* times

**Effect**

```
<MakeL 46>
node.type = NodeType.MakeL;
node.num = stParts[1];
node.workedSt = 0;
node.stChange = 1;
```

*Used in:* [Basic Stitch Parse on page 23](#)

**Make Right** An increase with parameter *num*

**Setup**
From front, left needle picks up vertical bar between top left and top right active stitches

**Instruction**
Knit through the back loop the stitch picked up by the left needle, repeat *num* times

**Effect**

```
<MakeR 47>
node.type = NodeType.MakeR;
node.num = stParts[1];
node.workedSt = 0;
node.stChange = 1;
```

*Used in:* [Basic Stitch Parse on page 23](#)

**Knit Together** A decrease with parameter *num*
Needle Entry
Top num active stitches, from left

Yarn Position
Back

Effect

\[ \text{\textless KnitTog 48} \]

\begin{verbatim}
node.type = NodeType.KnitTog;
node.num = stParts[1];
node.workedSt = node.num;
node.stChange = (-1) * (node.num - 1);
\end{verbatim}

U\textsc{sed In}: Basic Stitch Parse on page 23

Purl Together A decrease with parameter num

Needle Entry
Top num active stitches, from right

Yarn Position
Front

Effect

\[ \text{\textless PurlTog 49} \]

\begin{verbatim}
node.type = NodeType.PurlTog;
node.num = stParts[1];
node.workedSt = node.num;
node.stChange = (-1) * (node.num - 1);
\end{verbatim}

U\textsc{sed In}: Basic Stitch Parse on page 23

Slip Slip Knit A decrease

\textbf{Instruction}
Slip knitwise then slip purlwise and knit the two slipped stitches together

Effect

\[ \text{\textless SSK 50} \]

\begin{verbatim}
node.type = NodeType.SSK;
node.workedSt = 2;
node.stChange = -1;
\end{verbatim}

U\textsc{sed In}: Basic Stitch Parse on page 23

Slip Slip Purl A decrease

\textbf{Instruction}
Slip knitwise then slip purlwise and purl the two slipped stitches together
Pass Slip Stitch Over  A decrease

Needle Entry
On right needle, second active stitch from top, from left

Yarn Position
Back

Instruction
Pass stitch over top active stitch on right needle, and off needle

Notes
Most commonly used after a slip then knit, hence passing the slipped stitch over

**Effect**

```
<SSP 51>
node.type = NodeType.SSP;
node.workedSt = 2;
node.stChange = -1;
```

2.5.2 Verification

The role of a basic stitch during the verification pass is to update the row state with the stitch change and number of worked stitches by the given stitch.

```
<Verify Basic Stitch 53>
rowState_0.workedSt += node.workedSt * rep;
rowState_0.stChange += node.stChange * rep;
```

2.5.3 HTML Generation

The writing of stitches follows the standard notation [1].
var WriteBasicStitch = function(node) {

    var result = [];
    var rep = "";
    var num = "";

    if (node.repCount != null && node.repCount.value > 1) {
        rep = node.repCount.value;
    }

    if (node.num != null) {
        if (node.num.value > 0) {
            num = node.num.value;
        } else if (node.num > 0) {
            num = node.num;
        }
    }

    switch (node.type) {

        case NodeType.Knit:
            result.push(AddElement(TagType.Span, ClassType.Stitch, "K" + rep));
            break;

        case NodeType.Purl:
            result.push(AddElement(TagType.Span, ClassType.Stitch, "P" + rep));
            break;

        case NodeType.KnitTBL:
            result.push(AddElement(TagType.Span, ClassType.Stitch, "K" + rep + " tbl"));
            break;

        case NodeType.PurlTBL:
            result.push(AddElement(TagType.Span, ClassType.Stitch, "P" + rep + " tbl"));
            break;

        case NodeType.KnitBelow:
            result.push(AddElement(TagType.Span, ClassType.Stitch, "K" + num + "B" + rep));
            break;

        case NodeType.PurlBelow:
            result.push(AddElement(TagType.Span, ClassType.Stitch, "P" + num + "B" + rep));
            break;
    }

    return result;
}
case NodeType.Slip:
    result.push(AddElement(TagType.Span, ClassType.Stitch, "sl" + rep));
    break;

case NodeType.SlipKW:
    result.push(AddElement(TagType.Span, ClassType.Stitch, "sl" + rep + "k"));
    break;

case NodeType.SlipPW:
    result.push(AddElement(TagType.Span, ClassType.Stitch, "sl" + rep + "p"));
    break;

case NodeType.YarnOver:
    result.push(AddElement(TagType.Span, ClassType.Stitch, "yo" + rep));
    break;

case NodeType.KnitFB:
    result.push(AddElement(TagType.Span, ClassType.Stitch, "KFB"));
    if (rep > 0) { result.push(rep); }
    break;

case NodeType.PurlFB:
    result.push(AddElement(TagType.Span, ClassType.Stitch, "PFB"));
    if (rep > 0) { result.push(rep); }
    break;

case NodeType.Make:
    result.push(AddElement(TagType.Span, ClassType.Stitch, "M" + num));
    if (rep > 0) { result.push(rep); }
    break;

case NodeType.MakeL:
    result.push(AddElement(TagType.Span, ClassType.Stitch, "M" + num + "L"));
    if (rep > 0) { result.push(rep); }
    break;

case NodeType.MakeR:
    result.push(AddElement(TagType.Span, ClassType.Stitch, "M" + num + "R"));
    if (rep > 0) { result.push(rep); }
    break;

case NodeType.KnitTog:
    result.push(AddElement(TagType.Span, ClassType.Stitch, "k" + num + "tog"));
    if (rep > 0) { result.push(rep); }

break;

case NodeType.PurlTog:
    result.push(AddElement(TagType.Span, ClassType.Stitch, "k" + num + "tog"));
    if (rep > 0) { result.push(rep); }
    break;

case NodeType.SSK:
    result.push(AddElement(TagType.Span, ClassType.Stitch, "ssk"));
    if (rep > 0) { result.push(rep); }
    break;

case NodeType.SSP:
    result.push(AddElement(TagType.Span, ClassType.Stitch, "ssp"));
    if (rep > 0) { result.push(rep); }
    break;

case NodeType.PSSO:
    result.push(AddElement(TagType.Span, ClassType.Stitch, "psso"));
    if (rep > 0) { result.push(rep); }
    break;

default:
    break;

return result.join(" ");
};

2.6 Compound Stitch

It is possible to work a number of stitches in a single stitch. This means that after the loop has been pulled through an active stitch, that stitch is not dropped off the left needle, but remains until the sequence of stitches have been performed.

2.6.1 AST Node
\( \text{compSt} ::= '='<textspace}\{\text{basicSt}\}\text{ ('} ',' \text{basicSt}\text{)}*'} '>{\text{expr}} \)

The below example of a compound stitch means to perform the stitches between the angle brackets in one stitch.

\[ <K, P, K> \]

Purl Example 2.6: Compound stitch

The sequence of stitches of a compound stitch is contained between angle brackets. The use of any other bracket symbols will generate a warning and allow compilation to continue. A compound is separated from its siblings by a comma, but its children are also comma-separated. This is a case where on an error it is necessary for the lexer to scan to the terminator of the parent node production before continuing parsing.

```c
if (Sym.type == CharSym.OpenAngle) {
    nextSym();
} else if (Sym.type == CharSym.OpenParen
    || Sym.type == CharSym.OpenBrace
    || Sym.type == CharSym.OpenBrack) {
    AddMsg(MsgType.Warning, node, "Use '<' symbol at start of compound stitch.");
    nextSym();
} else {
    <<Unexpected Symbol Error 1>>
    AddMsg(MsgType.Error, node, "Missing '<' symbol at start of compound stitch.");
    scanToSym(CharSym.Period);
    return node;
}
```

USED IN: Compound Stitch Parse on page 37 INCLUDED BLOCKS: 1 on page 37

```c
if (Sym.type == CharSym.CloseAngle) {
    nextSym();
} else if (Sym.type == CharSym.CloseParen
    || Sym.type == CharSym.CloseBrace
    || Sym.type == CharSym.CloseBrack) {
    AddMsg(MsgType.Warning, node, "Use '>' symbol at end of compound stitch.");
    nextSym();
} else {
    <<Unexpected Symbol Error 1>>
```

36
Between the compound stitch brackets is a comma separated list of basic stitches.

A compound stitch optionally ends with a natural number or variable indicating how many times the compound stitch should be repeated. Note that a compound stitch works one stitch for every repeat, since all stitches in the sequence are worked into a single active stitch on the left needle.

### 2.6.2 Verification
2.6.3 HTML Generation

```javascript
var WriteCompSt = function(node) {
  var result = [];
  var stitches = [];

  if (node.children != null) {
    for (var i = 0; i < node.children.length; i++) {
      stitches.push(WriteNode(node.children[i]));
    }
  }

  result.push("(" + stitches.join(" ", ",") + ")");

  if (node.repCount != null && node.repCount.value > 1) {
    result.push(node.repCount.value + " times ");
  }

  result.push("in next st");

  return result.join(" ");
};
```

2.7 Fixed Stitch Repeat

A sequence of stitches may be repeated a fixed number of times.
2.7.1 AST Node

\[
\langle \text{fixedStRep} \rangle ::= \left[ \langle \text{stitchOp} \rangle (, \langle \text{stitchOp} \rangle)^* \right] \langle \text{expr} \rangle
\]

The below example means to perform the sequence between brackets (knit then purl) three times.

\[(K, P)\ 3\]

Purl Example 2.7: Fixed repeat

The children of a fixed stitch repeat is a sequence of fixed stitch repeats, compound stitches, and basic stitches contained within parentheses. As we saw for compound stitches, we cannot confidently assume the location of the next sibling in an error situation. We again scan to the terminator of the parent node production.

\[
\text{if (Sym.type == CharSym.OpenBrack)} { \\
\text{nextSym();}
\} \text{ else if (Sym.type == CharSym.OpenAngle} \\
\text{|| Sym.type == CharSym.OpenBrace} \\
\text{|| Sym.type == CharSym.OpenParen) } \{
\text{AddMsg(MsgType.Warning, node, "Use \'[\' symbol to start fixed stitch repeat.");}
\text{nextSym();}
\} \text{ else } \\
<<\text{Unexpected Symbol Error 1>>}
\text{AddMsg(MsgType.Error, node, "Missing \'[\' symbol to start fixed stitch repeat.");}
\text{scanToSym(CharSym.Period);}
\text{return node;}
\]

\[
\text{if (Sym.type == CharSym.CloseBrack)} { \\
\text{nextSym();}
\} \text{ else if (Sym.type == CharSym.CloseAngle} \\
\text{|| Sym.type == CharSym.CloseBrace} \\
\text{|| Sym.type == CharSym.CloseParen) } \{
\text{var msg = "Use \'\]\' symbol to end fixed stitch repeat stitches.";}
\text{AddMsg(MsgType.Warning, node, msg);}
\text{nextSym();}
\} \text{ else }
\]

39
The children of a fixed stitch repeat node are comma separated.

A fixed stitch repeat must end with a natural number expression.
2.7.2 Verification

A fixed stitch repeat adds the number of worked stitches and stitch change multiplied by the specified repeat amount to these values for its parent node.

```
<Verify Fixed Stitch Repeat 65>
VerifyRowElemChildren(node, rowState_1);
rowState_0.workedSt += rowState_1.workedSt * rep;
rowState_0.stChange += rowState_1.stChange * rep;
```

USED IN: Verify Row Elem on page 20

2.7.3 HTML Generation

```
<Write HTML Fixed Stitch Repeat 66>
var WriteFixedStRep = function(node) {
  var stitches = [];
  if (node.children != null) {
    for (var i = 0; i < node.children.length; i++) {
      stitches.push(WriteNode(node.children[i]));
    }
  }
  return "[* + stitches.join(" ", ") + "] " + node.repCount.value + " times";
};
```

USED IN: code/codegen.js on page 83

2.8 Undetermined Stitch Repeat

A sequence of stitches may be repeated a number of times that depends on the current length of the row. We will call this an undetermined stitch repeat.

2.8.1 AST Node

\[
\langle uStRep \rangle : \star \langle stitchOp \rangle (',', \langle stitchOp \rangle )* ';'; 'to' (\langle last \rangle \langle expr \rangle ) | 'end'
\]
The first example below means perform the knit stitch as many times as required to get to the end of the row. The second example means perform the knit stitch to the last two stitches, then purl the last two.

\*K; to end

Purl Example 2.8: Undetermined Stitch Repeat 1

\*K; to last 2, P 2.

Purl Example 2.9: Undetermined Stitch Repeat 2

An undetermined stitch repeat must begin with an asterisk. As in compound stitches and fixed repeats, if an invalid symbol is used, the lexer scans to the terminator of the parent production to continue parsing.

if (Sym.type == CharSym.Asterisk) {
    nextSym();
} else {
    \<<Unexpected Symbol Error 1>>
    var msg = "Missing \\*\" symbol at start of undetermined stitch repeat."
    AddMsg(MsgType.Error, node, msg);
    scanToSym(CharSym.Period);
    return node;
}

Following the asterisk is a sequence of comma-separated basic stitches, compound stitches, and fixed stitch repeats.

node.children.push(StitchOpParse());

while (Sym.type == CharSym.Comma) {
    nextSym();
    node.children.push(StitchOpParse());
}

42
The sequence of children is terminated with a semicolon.

```javascript
if (Sym.type == CharSym.Semicolon) {
    nextSym();
} else if (Sym.type == CharSym.Colon) {
    var msg = "Use \';\' symbol at the end of undetermined stitch repeat stitches."
    AddMsg(MsgType.Warning, node, msg);
    nextSym();
} else {
    var msg = "Use \';\' symbol at the end of undetermined stitch repeat stitches."
    AddMsg(MsgType.Error, node, msg);
    scanToSym(CharSym.Period);
    return node;
}
```

If there is an error in the remainder of the undetermined stitch repeat, then since we are past the sequence of children we can assume that the next comma delimits the next sibling, and the next period is the terminator of the parent production.

Next, the pattern writer must specify how far along the row this repeat should be performed, beginning with the keyword “to”.

```javascript
if (Sym.type == KeywordSym.To) {
    nextSym();
} else if (Sym.type == SymType.Ident) {
    var msg = "Use \"\" + KeywordSym.To + \"\" after \"\';\\"\" for undetermined stitch repeat."
    AddMsg(MsgType.Warning, node, msg);
    nextSym();
} else {
    var msg = "Missing \"\" + KeywordSym.To + \"\" after \"\';\\"\" for undetermined stitch repeat."
    AddMsg(MsgType.Error, node, msg);
    scanToSym(CharSym.Comma || CharSym.Period);
}
```
The keyword “end” is used if the sequence should be repeated to the end of the row, and the keyword “last” is used if the sequence should be repeated to within a given number of stitches from the end of the row. For this reason, nesting of undetermined stitch repeats is not possible. An important note is that the number of stitches remaining on the left needle at the end of the last repeat must coincide exactly with the number specified here (0 for “end”).

```javascript
if (Sym.type == KeywordSym.Last) {
    nextSym();
    node.num = ExpressionParse(CharSym.Comma || CharSym.Period);
} else if (Sym.type == KeywordSym.End) {
    node.num = { type: NodeType.NatLiteral, value: 0 };
    nextSym();
} else {
    <<Unexpected Symbol Error 1>>
    var msg = "Missing repeat instructions. Expecting \"" + KeywordSym.Last + "\" or \"" + KeywordSym.End + "\"."
    AddMsg(MsgType.Error, node, msg);
    scanToSym(CharSym.Comma || CharSym.Period);
}
```

Used in: Undetermined Stitch Repeat Parse on page 44
Included Blocks: 1 on page 6

```javascript
var UStRepParse = function () {

    var node = { type: NodeType.UStRep, children: [], line: State.line };

    <<Undetermined Stitch Repeat Parse Open 67>>
    <<Undetermined Stitch Repeat Parse Children 68>>
    <<Undetermined Stitch Repeat Parse Close 69>>
    <<Undetermined Stitch Repeat Parse To 70>>
    <<Undetermined Stitch Repeat Parse Repeat Instruction 71>>

    return node;
};
```

Used in: Ast Construction Pass on page 75
Included Blocks: 67 on page 42, 68 on page 42, 69 on page 43, 70 on page 43, 71 on page 44

44
2.8.2 Verification

To verify an undetermined stitch repeat, we first determine the number of stitches the children will be repeated over. If the number of stitches worked over a single repeat does not divide this value, then there will be a number of unworked stitches equal to the division remainder at the end of the row. In this case, an error is generated. Otherwise, the row state of the parent node is updated with the number of worked stitches and stitch change caused by the undetermined stitch repeat.

```javascript
VerifyRowElemChildren(node, rowState_1);

var stToWork = rowState_1.initialWidth - rowState_0.workedSt - node.num.value;

var remainSt = stToWork % rowState_1.workedSt;
if (remainSt != 0) {
    var msg = remainSt + " st will remain after the last possible repeat."
    AddMsg(MsgType.Verification, node, msg);
} else {
    rep = (stToWork - (stToWork % rowState_1.workedSt)) / rowState_1.workedSt;
}

rowState_0.workedSt += rowState_1.workedSt * rep;
rowState_0.stChange += rowState_1.stChange * rep;
```

2.8.3 HTML Generation

```javascript
WriteUStRep = function(node) {
    var result = [];
    var stitches = [];

    if (node.children != null) {
        for (var i = 0; i < node.children.length; i++) {
            stitches.push(WriteNode(node.children[i]));
        }
    }

    result.push("*" + stitches.join("", ") + "; rep from * to");

    var rem = node.num.value;
```
if (rem == 0) {
  result.push("end");
} else if (rem == 1) {
  result.push("last " + rem + " st");
} else if (rem > 1) {
  result.push("last " + rem + " sts");
} else {
  result.push("invalid value");
}

return result.join(" ");

2.9 Bind-Off

We have considered cast-ons, rows and stitches. These elements, followed by a bind-off, are sufficient to construct a simple pattern.

2.9.1 AST Node

⟨bo⟩ ::= ‘BO’ ⟨Nat⟩ ‘.’

Purl Example 2.10: Market Bag Body Bind-off

The “BO” keyword is used to declare a bind-off for the pattern.

<Bind-Off Parse BO 75>
AddMsg(MsgType.Error, node, msg);
scanToSym(CharSym.Period);
nextSym();
return node;
}

Next, a natural number is given as the number of stitches to bind-off.

if (Sym.type == SymType.Nat) {
   node.value = Sym.value;
   nextSym();
} else {
   <<Unexpected Symbol Error 1>>
   AddMsg(MsgType.Error, node, "Bind-off count not specified.");
   scanToSym(CharSym.Period);
   nextSym();
   return node;
}

A bind-off ends with a period as terminator.

var BoParse = function() {
   var node = { type: NodeType.BindOff, value: 0, line: State.line };

   <<Bind-Off Parse BO 75>>
   <<Bind-Off Parse Count 76>>
   <<Period Terminator 6>>

   return node;
};
2.9.2 Verification

The value of a bind-off node must be equivalent to the final width of the row before it, otherwise either some stitches will still be active after completing a pattern, or else there will be too few stitches to bind-off (the former error being much more severe).

```javascript
var VerifyBindOff = function(node) {
    if (node.value != State.width) {
        var msg = "Binding off " + node.value + " sts over " + State.width + " sts.";
        AddMsg(MsgType.Verification, node, msg);
    }
};
```

2.9.3 HTML Generation

```javascript
var WriteBo = function(node) {
    var msg = "Bind-off " + node.value + " sts.";
    return AddElement(TagType.Div, ClassType.BindOff, msg);
};
```

2.10 Join

An alternative to using a bind-off to finish a pattern is to join the remaining active stitches to some other location on the same section, or another knitted object. If active stitches are to be joined to other active stitches, this is called grafting.

2.10.1 AST Node

\[\text{\langle join\rangle := \text{'Join' \langle Nat\rangle 'to' \langle String\rangle} \text{'.'}}\]
**Purl Example 2.11: Market Bag Handle Join**

The “Join” keyword is used to declare a join for the pattern.

```plaintext
if (Sym.type == KeywordSym.Join) {
    nextSym();
} else if (Sym.type == SymType.Ident) {
    var msg = "A join declaration must start with \"" + KeywordSym.Join + \\
    AddMsg(MsgType.Warning, node, msg);
} else {
    <<Unexpected Symbol Error 1>>
    var msg = "Missing \"at start of join declaration.";
    AddMsg(MsgType.Error, node, msg);
    scanToSym(CharSym.Period);
    nextSym();
    return node;
}
```

Next, a natural number is given as the number of stitches to join.

```plaintext
if (Sym.type == SymType.Nat) {
    node.value = Sym.value;
    nextSym();
} else {
    <<Unexpected Symbol Error 1>>
    AddMsg(MsgType.Error, node, "Join count not specified.");
    scanToSym(CharSym.Period);
    nextSym();
    return node;
}
```

It is necessary to state where the active stitches should be joined. This begins with the keyword “to” followed by a string with directions on the join.
if (Sym.type == KeywordSym.To) {
    nextSym();
} else {
    <<Unexpected Symbol Error 1>>
    scanToSym(CharSym.Period);
    nextSym();
    return node;
}

if (Sym.type == SymType.String) {
    node.destination = Sym.value;
    nextSym();
} else {
    <<Unexpected Symbol Error 1>>
    AddMsg(MsgType.Error, node, "Missing join destination.");
    scanToSym(CharSym.Period);
    nextSym();
    return node;
}

A join ends with a period as terminator.

var JoinParse = function() {

    var node = { type : NodeType.Join, value : 0, line : State.line };

    <<Join Parse Keyword 80>>
    <<Join Parse Count 81>>
    <<Join Parse Destination 82>>
    <<Period Terminator 6>>

    return node;
};
2.10.2 Verification

The value of a join node must be equivalent to the final width of the row before it, otherwise either some stitches will still be active after completing a pattern, or else there will be too few stitches to bind-off (the former error being much more severe).

```javascript
var VerifyJoin = function(node) {
    if (node.value != State.width) {
        var msg = "Joining " + node.value + " sts of " + State.width + " sts.";
        AddMsg(MsgType.Verification, node, msg);
    }
};
```

**USED IN:** Verification Pass on page 80

2.10.3 HTML Generation

```javascript
var WriteJoin = function(node) {
    var msg = "Join " + node.value + " sts to " + node.destination + ".";
    return AddElement(TagType.Div, ClassType.Join, msg);
};
```

**USED IN:** code/codegen.js on page 83

2.11 Pattern

Since we have now seen enough to construct a simple pattern, we will look at the syntax of a knitting pattern written in Purl.

2.11.1 AST Node

\[
\text{pattern} ::= \text{pattern'} \langle \text{String} \rangle \ ':\' (\langle \text{co} \rangle \langle \text{body} \rangle \langle \text{bo} \rangle \ | \langle \text{section} \rangle +)
\]
pattern "One Row":
CO 1.
row : K.
BO 1.

Purl Example 2.12: Simple single row pattern

A pattern definition begins with the keyword “pattern”.

```
if (Sym.type == KeywordSym.Pattern) {
    nextSym();
} else if (Sym.type == SymType.Ident) {
    var msg = "A pattern declaration must start with "." + KeywordSym.Pattern + ".";
    AddMsg(MsgType.Warning, node, msg);
    nextSym();
} else {
    <<Unexpected Symbol Error 1>>
    var msg = "Expecting "." + KeywordSym.Pattern + "." to start pattern declaration."
    AddMsg(MsgType.Error, node, msg);
    scanToSym(CharSym.Colon);
}
```

The next requirement is a string in double quotes. This is the title of the pattern. The reason for using a string rather than a single identifier is to allow greater flexibility in the naming of patterns. In this way, a pattern can have a multi-word name that can also include any reserved keywords. If an identifier is provided rather than a string, a warning is created and compilation continues. Any other symbol causes an error and the lexer scans to the next colon symbol, which is required following the pattern title.

```
if (Sym.type == SymType.String) {
    node.name = Sym.value;
    nextSym();
} else if (Sym.type == SymType.Ident) {
    node.name = Sym.value;
    var msg = "Remember to use double quotes around the name of your pattern."
    AddMsg(MsgType.Warning, node, msg);
    nextSym();
} else {
    <<Unexpected Symbol Error 1>>
    AddMsg(MsgType.Error, node, "The pattern name is not specified.");
    scanToSym(CharSym.Colon);
}```
The main content of the pattern may by a cast-on, pattern body, then bind-off for a single-section pattern, or a number of defined sections. If the symbol after the colon is the cast-on keyword "CO", then we are parsing a simple pattern. A simple pattern provides instructions for a single discrete object. In contrast, a composite pattern provides instructions to knit multiple objects to be joined to form a larger structure.

```javascript
if (Sym.type == KeywordSym.CastOn) {
    <<Pattern Content Parse 89>>
} else {
    <<Pattern Parse Composite 99>>
}
```

A simple pattern begins with a cast-on or pick-up, followed by a pattern body, and finally a bind-off or a join.

```javascript
node.start = CoParse();
node.children = BodyParse();
node.finish = BoParse();
```

```javascript
var PatternParse = function () {
    var node = { type: NodeType.Pattern, children: [], line: State.line };

    <<Pattern Parse Pattern 86>>
    <<Pattern Parse Title 87>>
    <<Colon Separator 18>>

    <<Pattern Parse Main 88>>

    return node;
};
```
2.11.2 Verification

When the `co` property is not null, we are verifying a simple pattern, so we verify the cast-on, the children, and then the bind-off. Otherwise, we have a composite pattern and we verify the children (which are pattern sections in this case).

```javascript
var VerifyPattern = function(node) {
    if (node.start !== null) {
        VerifyNode(node.start);
        VerifyChildren(node);
        VerifyNode(node.finish);
    } else {
        VerifyChildren(node);
    }
};
```

2.11.3 HTML Generation

```javascript
var WritePattern = function(node) {
    var result = [];

    result.push(OpenElement(TagType.Div, ClassType.Pattern));
    result.push(AddElement(TagType.Div, ClassType.PatternName, node.name));

    if (node.start !== null) {
        result.push(WriteNode(node.start));
        result.push(WriteBody(node));
        result.push(WriteNode(node.finish));
    } else if (node.children !== null) {
        for (var i = 0; i < node.children.length; i++) {
            result.push(WriteNode(node.children[i]));
        }
    }
};
```
2.12 Pattern Body

The body of a pattern is the main content of the pattern (everything between cast-on and bind-off). This consists of zero or more row repeats (see 2.13), and sample calls (see 2.15.2). Note that <pattern> is not the only production that uses the <body> production to create its child nodes. Row repeats and sample definitions (see 2.15.1) are other constructs that use the <body> production.

2.12.1 AST Node

\[
\langle \text{body} \rangle ::= ((\langle \text{rowDef} \rangle \mid \langle \text{rowRep} \rangle \mid \langle \text{sampleCall} \rangle))^*
\]

The body parse method builds up and returns an array of nodes which make up a pattern body.

```javascript
var BodyParse = function () {
    var bodyElems = [];
    while (Sym.type != SymType.EOF) {
        switch (Sym.type) {
            case KeywordSym.Row:
            case KeywordSym.Rnd:
                bodyElems.push(RowDefParse());
                break;

            case SymType.RowRep:
                bodyElems.push(RowRepeatParse());
                break;

            case SymType.Ident:
                bodyElems.push(SampleCallParse());
                break;

            default:
```
2.12.2 HTML Generation

```javascript
var WriteBody = function (node) {
    var result = [];
    result.push(OpenElement(TagType.Div, ClassType.Body));

    if (node.children != null) {
        for (var i = 0; i < node.children.length; i++) {
            result.push(WriteNode(node.children[i]));
        }
    }

    result.push(CloseElement(TagType.Div));

    return result.join('')
};
```

2.13 Row Repeats

If elements of a pattern body are to be repeated a number of times, rather than rewriting those elements, a row repeat may be used for more concise notation.

2.13.1 AST Node

\[
\text{(rowRepeat)} ::= \text{`**'} \text{(body)} \text{`repeat'} \text{<expr>}
\]
The body of the diagonalLace sample used by the market bag pattern uses a row repeat. This means the two rows between the double asterisk and repeat n are to be repeated n times.

\[
\begin{align*}
** \\
rd : *K2T, YO; to end. \\
rnd : *K; to end. \\
repeat n
\end{align*}
\]

Purl Example 2.13: Ribbing repeat

A row repeat begins with two asterisks.

The end of a row repeat body is marked by the “repeat” keyword. This must then be followed by a natural number or variable.
```
AddMsg(MsgType.Error, node, msg);
scanToSym(CharSym.Period);
nextSym();
```

The children of a row repeat are represented as an array of nodes. We use the same body parse method as simple patterns and pattern sections.

```
var RowRepeatParse = function() {

    var node = { type: NodeType.RowRep, children: [], line: State.line);

    node.children = BodyParse();
    node.repCount = ExpressionParse(CharSym.Period);

    return node;
};
```

2.13.2 HTML Generation

```
var WriteRowRep = function(node) {

    var result = [];

    result.push(OpenElement(TagType.Div, ClassType.RowRep));
    result.push("**");
    result.push(WriteBody(node));
    result.push("rep from ** " + node.repCount.value + " times");
    result.push(CloseElement(TagType.Div));

    return result.join("\n");
};
```
2.14 Pattern Section

If writing a pattern to create multiple discrete objects, then the pattern must be made up of a number of pattern sections. In this case, we write a composite pattern, which is a pattern with zero or more pattern sections as its children.

```javascript
while (Sym.type == KeywordSym.Section) {
    node.children.push(SectionParse());
}
```

2.14.1 AST Node

\[
\text{section} ::= \text{section} \langle \text{String} \rangle \text{'}(: (\text{co} | \text{pu}) \text{body} (\text{bo} | \text{join}))
\]

The market bag body is a section of the market bag pattern.

```plain
section "Body":
CO 8 circular.
   rnd : *K, YO, K; to end.
   rnd : *K; to end.
   circleX with 1, 23.
   diagonalLace with 30.
   garterStitchCC with 4, 1.
BO 100.
```

Purl Example 2.14: Market Bag Body

A section definition begins with the “section” keyword.
The title of the section follows, which should be a string enclosed in double quotes. As in the title of a pattern, if an identifier symbol is found instead, then a warning is generated, the name of the section is set to the identifier, and compilation can continue. Any other symbol will cause an error and the lexer will scan to the colon separator that should occur before the section content.

Parsing of the content of a section is similar to parsing the content of a simple pattern, except that picking up stitches and joining are alternatives for cast-on and bind-off, respectively.
} else {
    <<Unexpected Symbol Error 1>>
    node.start = {};
    scanToSym(CharSym.Period);
    nextSym();
}

node.children = BodyParse();

if (Sym.type == KeywordSym.BindOff) {
    node.finish = BoParse();
} else if (Sym.type == KeywordSym.Join) {
    node.finish = JoinParse();
} else {
    <<Unexpected Symbol Error 1>>
    node.finish = {};
    scanToSym(CharSym.Period);
    nextSym();
}

---

**Section Parse**

```javascript
var SectionParse = function() {
    var node = { type: NodeType.Section, line: State.line };

    <<Section Parse Section 100>>
    <<Section Parse Title 101>>
    <<Colon Separator 18>>
    <<Section Content Parse 102>>

    return node;
};
```

---

**Ast Construction Pass**

---

### 2.14.2 Verification

Pattern section verification is very similar to verification of a simple pattern, but first the `sectionName` property of the `State` object is set to the name of the current section. This is used in reporting of errors.
2.14.3 HTML Generation

```javascript
var WriteSection = function(node) {
    var result = [];
    result.push(OpenElement(TagType.Div, ClassType.Section));
    result.push(AddElement(TagType.Div, ClassType.SectionName, node.name));
    result.push(WriteNode(node.start));
    result.push(WriteBody(node));
    result.push(WriteNode(node.finish));
    result.push(CloseElement(TagType.Div));
    return result.join('');
};
```

2.15 Pattern Sample

A pattern sample may be thought of as any segment of a knitting pattern between the cast-on and bind-off. In Purl this construct allows pattern samples to be defined with natural number parameters. A pattern sample can be called from any pattern body. There is no analogous concept in the standard notation.
2.15.1 Sample Definition

In order to use a sample, it must first be defined outside of the pattern definition.

\[
\langle \text{sampleDef} \rangle ::= '\text{sample}' \langle \text{Ident} \rangle \left[\begin{array}{l}
\text{with}\ (\langle \text{Ident} \rangle \ (',\ (\langle \text{Ident} \rangle)^*)
\end{array}\right]
\langle (\text{sampleBranch} \mid ':') \langle \text{body} \rangle \rangle
\]

\[
\langle \text{sampleBranch} \rangle ::= ('|' \langle \text{condition} \rangle ':') \langle \text{body} \rangle+\]

The “circle” sample used by the market bag pattern is an example of a sample with a branch. For the two parameters n and max given, rows are added to the calling pattern only if \( n < max \).

| sample | circle with n, max |
|--------|--------------------|
|       | \( / n < max: \) |
|       | \( \text{rnd} : [K, YO, Kn, YO, K] 4. \) |
|       | \( \text{rnd} : *K; \text{ to end}. \) |
|       | \( \text{circle with } n + 2, \text{ max}. \) |

Purl Example 2.15: Circle Sample for Market Bag

A pattern sample definition begins with the “sample” keyword.

A pattern sample definition begins with the “sample” keyword.

if (Sym.type == KeywordSym.Sample) {
    nextSym();
} else if (Sym.type == SymType.Ident) {
    var msg = "A sample definition must start with \" + KeywordSym.Sample + \\
                \" at start of sample definition";
    AddMsg(MsgType.Warning, node, msg);
    nextSym();
} else {
    <<Unexpected Symbol Error 1>>
    var msg = "Missing \" + KeywordSym.Sample + \\
                \" at start of sample definition";
    AddMsg(MsgType.Error, node, msg);
    scanToSym(SymType.Ident);
}

Next is the sample identifier. Since the sample is not part of the target language, the identifier will not be seen after compilation. So there is no reason to consider aesthetic motivations in sample naming (as there is in pattern and section naming). If the current symbol is not an identifier, then an error is created and the lexer scans to the next colon symbol to continue compilation.
If a sample definition uses parameters, the parameter list is prefixed with the keyword “with”, followed by one or more identifiers separated by commas. These are added to the list of parameter names used in the sample definition.

Sample Def Parse on page 65

```javascript
if (Sym.type == SymType.Ident) {
  node.name = Sym.value;
  nextSym();
} else if (hasOwnProperty(KeywordSym, Sym.type)) {
  var msg = Sym.value + " is a reserved keyword and not a valid sample identifier.";
  AddMsg(MsgType.Error, node, msg);
  scanToSym(KeywordSym.With || CharSym.Colon);
} else {
  AddMsg(MsgType.Error, node, "Missing or invalid sample identifier.");
  scanToSym(KeywordSym.With || CharSym.Colon);
}
```

Sample Def Parse Params 108

```javascript
if (Sym.type == KeywordSym.With) {
  nextSym();

  if (Sym.type == SymType.Ident) {
    node.paramNames.pushSym(value);
    nextSym();
  } else {
    <<Unexpected Symbol Error 1>>
  }

  while (Sym.type == CharSym.Comma) {
    nextSym();

    if (Sym.type == SymType.Ident) {
      node.paramNames.pushSym(value);
      nextSym();
    } else {
      <<Unexpected Symbol Error 1>>
    }
  }
}
```

Sample Def Parse on page 65

INCLUDED BLOCKS: 1 on page 65
The sample definition node is then added to the collection of samples in the global state object of the parser so that a sample may be used in its own definition.

A sample definition can be written to use a specific branch for the sample body if the branch condition is satisfied. For a sample without branching, we expect a colon separator after the parameters, followed by the sample definition body, which is parsed in the same way as simple patterns, pattern sections, and row repeats.

```javascript
var SampleDefParse = function() {
  var node = { type: NodeType.SampleDef,
               paramNames: [],
               children: [],
               line: State.line
          };

  State.samples[node.name] = node;

  if (Sym.type == CharSym.VerticalBar) {
    <<Sample Branches Parse 110>>
  } else {
    <<Colon Separator 18>>
    node.children = BodyParse();
  }
};
```

If the sample is defined with branches, then a branch begins with ‘|’, followed by a condition, a colon separator, then the sample body.

```javascript
while (Sym.type == CharSym.VerticalBar) {
  nextSym();

  var branch = { type: NodeType.Branch };
  branch.condition = ConditionParse();
  <<Colon Separator 18>>
  branch.children = BodyParse();
```
2.15.2 Sample Call

Once a sample has been defined, it can be called from any pattern body (simple pattern, section, row repeat, sample definition). The sample example in the above section also includes a recursive sample call.

\[
\text{\texttt{(sampleCall)} ::= \texttt{\{Ident\} ['\texttt{with}\} \texttt{\{expr\}} (', \texttt{\{expr\}})+\texttt{\}}.}'
\]

\texttt{stockinetteStitch with 10.}

Purl Example 2.16: Use stockinette stitch sample

A sample call begins with an identifier for the sample to be used.

\texttt{if (Sym.type == SymType.Ident) { node.name = Sym.value; nextSym(); } else { <<Unexpected Symbol Error 1>> AddMsg(MsgType.Error, node, "Missing sample call identifier."); scanToSym(CharSym.Period); }}

\texttt{var sampleDef = State.samples[node.name];}
if (Sym.type == KeywordSym.With) {
    nextSym();

    var i;
    var required = sampleDef.paramNames.length;
    for (i = 0; i < required; i++) {

        if (i > 0) {
            if (Sym.type == CharSym.Comma) {
                nextSym();
            } else {
                //Unexpected Symbol Error 1>
            }
        }

        node.paramMap[sampleDef.paramNames[i]] = ExpressionParse();
    }

    if (i != sampleDef.paramNames.length) {
        var msg = node.name + "parameters required: " + required + ", passed: " + i + "."
        ;
        AddMsg(MsgType.Error, node, msg);
        scanToSym(CharSym.Period);
    }
}

A sample call is terminated by a period symbol.

```javascript
var SampleCallParse = function() {

    var node = { type: NodeType.SampleCall, paramMap: {}, line: State.line }

    //Sample Call Parse Ident 111>
    //Sample Call Parse Params 112>
    //Period Terminator 6>

    return node;
};
```

Var SampleCallParse = function() {

    var node = { type: NodeType.SampleCall, paramMap: {}, line: State.line }

    //Sample Call Parse Ident 111>
    //Sample Call Parse Params 112>
    //Period Terminator 6>

    return node;
};
2.16 Root

We have now looked at all available knitting pattern constructs in Purl. All that remains is to combine the

top level elements of a knitting pattern program: sample definitions and a pattern definition. The market

bag pattern given in the introduction is an example of a complete Purl program.

2.16.1 AST Node

\[
\langle \text{program} \rangle := (\langle \text{sampleDef} \rangle)^* \langle \text{pattern} \rangle
\]

In the present implementation, all sample definitions and one pattern definition must be combined in

one file, with all sample definitions above the pattern definition. This is not ideal since Purl is meant to allow

modularity and reusability of segments of knitting patterns, but it is sufficient for the initial implementation

and experimentation.

```javascript
var ProgramParse = function () {
    var program = { type: NodeType.Root, pattern: null, line: State.line }; 
    while (Sym.type == KeywordSym.Sample) {
        SampleDefParse();
    }
    program.pattern = PatternParse();
    return program;
};
```

Used in: Ast Construction Pass on page 75

2.16.2 HTML Generation

```javascript
var WriteRoot = function(node) {
    return WriteNode(node.pattern);
};
```

Used in: code/codegen.js on page 83
Lexical Analysis

The function `CreateLexer` returns an object that consists of a function to get the next symbol and a function to get the current line in the code. The input is first split into an array. `CreateLexer` has a private method `next` to update the current character, position within the array, line, and position on the line.

```javascript
var CreateLexer = function(src) {
    var srcArr = [];
    if (src != null) {
        srcArr = src.split('');
    }

    var ch = srcArr[0];
    if (srcArr.length == 0) {
        ch = SymType.EOF;
    }
    var pos = 0;
    var lineNum = 1;
    var linePos = 0;

    var next = function() {
        if (pos < srcArr.length - 1) {
            pos += 1;
            linePos += 1;
            ch = srcArr[pos];

            if (/
                .test(ch)) {
                lineNum += 1;
                linePos = 0;
```
GetLine simply returns an object with properties for line number, line position and character position.

GetLine : function() |
  return { num : lineNum, pos : linePos, charPos : pos };

GetSym determines the current token by the value of the current character, skipping any whitespace characters.
If the lexer reaches the end of the input, an end of file token is returned, and the character will no longer be updated.

```javascript
return { type : SymType.EOF, value : SymType.EOF } ;
```

If the current character is a double quote, then the symbol type is `title`, and everything until the next double quote is the value.

```javascript
next();
var str = "";
while (ch != SymType.String && ch != SymType.EOF) {
    str = str + ch;
    next();
}
if (ch == SymType.String) {
    next();
}
return { type : SymType.String, value : str } ;
```

If the current character is a number, then the symbol type is a natural number and the symbol value is the concatenation of all characters until a non-numeric character.

```javascript
var num = "";
```
\[
\text{while } (/^[0-9]$/.test(ch)) \\{
    \text{num} = \text{num} + \text{ch};\\
    \text{next();}\\
}\]

\text{return } \{ \text{type} : \text{SymType.Nat}, \text{value} : \text{num} \};

\text{USED IN: Lexer GetSym Function on page 70}

If the current character is a letter, concatenate all characters until the next non-alphanumeric character. If this string matches a reserved \text{keyword}, then return a symbol representing that keyword. If the string matches a regular expression in the \text{stitch lookup}, then return a symbol representing that stitch. Otherwise, return an \text{ident} symbol.

\text{var id = "";}

\text{while } (/^[a-zA-Z0-9]$/.test(ch)) \\{
    \text{id} = \text{id} + \text{ch};\\
    \text{next();}\\
}\]

\text{for (var idSym in KeywordSym) } \\{
    \text{if (id == KeywordSym[idSym]) } \\{
        \text{return } \{ \text{type} : \text{id}, \text{value} : \text{id} \};\\
    \}\\
}\]

\text{for (var stSym in StitchSym) } \\{
    \text{if (\text{StitchSym[stSym]}.test(id)) } \\{
        \text{return } \{ \text{type} : \text{StitchSym[stSym]}, \text{value} : \text{id} \};\\
    \}\\
}\]

\text{return } \{ \text{type} : \text{SymType.Ident}, \text{value} : \text{id} \};

\text{USED IN: Lexer GetSym Function on page 70}

If the current symbol is in the \text{char symbol lookup}, then if it is an asterisk, first check if the next character is also an asterisk. If a double asterisk, then we have a \text{row repeat} symbol, otherwise return the initial character symbol type and value.
for (var chSym in CharSym) {

    if (ch == CharSym[chSym]) {

        var result = ch;
        next();

        if (result == CharSym.Asterisk && ch == CharSym.Asterisk) {
            result = SymType.RowRep;
            next();
        } else if (result == CharSym.OpenAngle && ch == CharSym.Equal) {
            result = SymType.LessEq;
            next();
        } else if (result == CharSym.CloseAngle && ch == CharSym.Equal) {
            result = SymType.GreaterEq;
            next();
        }

        return { type : result, value : result };
    }
}

3.1 Parser

var Parser = (function() {

<<Ast Construction Pass 125>>
<<Sample Substitution Pass 126>>
<<Verification Pass 131>>

var AddMsg = function(msgType, node, msgStr) {
    var msgObj = {
        messageType : msgType,
        sectionName : State.sectionName,
        line : State.line,
        rowIndex : State.rowIndex,
        message : msgStr
    };
};
Statemessages.push(msgObj);

switch (msgType) {
    case MsgType.Error:
        node.hasErrorMsg = true;
        break;
    case MsgType.Warning:
        node.hasWarningMsg = true;
        break;
    case MsgType.Verification:
        node.hasVerificationMsg = true;
        break;
    default:
        break;
}

var State = {};

return {
    Parse : function (input) {
        State = { sectionName : null, samples : [], messages : [] };

        var ast = AstConstructionPass(input);
        console.clear();
        console.log("PASS 1:----------------------------- \n" + JSON.stringify(ast, undefined, 2));
        console.log("STATE:------------------------------- \n" + JSON.stringify(State, undefined, 2));
        SampleSubstitutionPass(ast);
        console.log("PASS 2:----------------------------- \n" + JSON.stringify(ast, undefined, 2));
        console.log("STATE:------------------------------- \n" + JSON.stringify(State, undefined, 2));
        VerificationPass(ast);
        console.log("PASS 3:----------------------------- \n" + JSON.stringify(ast, undefined, 2));
        console.log("STATE:------------------------------- \n" + JSON.stringify(State, undefined, 2));
        ast.messages = State.messages;
        State = {};

        return ast;
    }
};
3.1.1 Pass 1: AST Construction

The first pass constructs a syntax tree representing the source pattern. See [2] for details on parsing specific pattern elements.

```javascript
var AstConstructionPass = function(input) {

  var nextSym = function() {
    Sym = Lexer.GetSym();
    if (Sym.type == SymType.EOF) {
      State.line = null;
    } else {
      State.line = Lexer.GetLine();
    }
  };

  var scanToSym = function(symType) {
    while (Sym.type != symType && Sym.type != SymType.EOF) {
      nextSym();
    }
  };

  var hasOwnValue = function(obj, val) {
    for (var prop in obj) {
      if (obj.hasOwnProperty(prop) && obj[prop] === val) {
        return true;
      }
    }
    return false;
  };

  var getNatSym = function(terminatorSym) {
    var result = [];
    if (Sym.type == SymType.Nat) {
      result = [{ type: NodeType.NatLiteral, value: Sym.value }];
      nextSym();
    } else if (Sym.type == SymType.Ident) {
```
result = { type: NodeType.NatVariable, value: Sym.value };  
nextSym();
}  
else {
    <<Unexpected Symbol Error 1>>
    scanToSym(terminatorSym);
}

return result;

var Lexer = CreateLexer(input);
var Sym;

nextSym();

if (Sym.type == SymType.EOF) {
    AddMsg(MsgType.Warning, {}, "No pattern to compile :(");
    return {};
} else {
    return ProgramParse();
}

76
3.1.2 Pass 2: Sample Substitution

The second pass is responsible for replacing all sample call nodes with the children of the corresponding sample definition, and updating parameter values according to the `paramMap` object of each sample call. The AST is traversed depth first in this pass and the function `SampleSubstitutionPass` starts by traversing the children of the pattern node with an empty object as the `paramMap`.

```javascript
var SampleSubstitutionPass = function (ast) {

    // Pass 2 Traverse Children
    var TraverseChildren = function (node, paramMap) {
        if (node.children != null) {
            for (var i = 0; i < node.children.length; i++) {
                var child = node.children[i];
                switch (child.type) {
                    case NodeType.Section:

```

All row and stitch constructs require any expressions to have variables updated, and for nodes with children, the child nodes are then traversed.

```javascript
    case NodeType.Section:
```
TraverseChildren(child, paramMap);
break;

case NodeType.SampleCall:
    <<Pass 2 Replace Sample Call 128>>
    break;

case NodeType.Branch:
    UpdateCondition(child.condition, paramMap);
    if (child.condition.doBranch) {
        TraverseChildren(child, paramMap);
        node.children = child.children;
        return;
    }
    break;

case NodeType.Row:
case NodeType.RowRep:
case NodeType.FixedStRep:
case NodeType.UStRep:
case NodeType.CompSt:
    UpdateExpressions(child, paramMap);
    TraverseChildren(child, paramMap);
    break;

case NodeType.Knit:
case NodeType.Purl:
case NodeType.KnitTBL:
case NodeType.PurlTBL:
case NodeType.KnitBelow:
case NodeType.PurlBelow:
case NodeType.Slip:
case NodeType.SlipKW:
case NodeType.SlipPW:
case NodeType.YarnOver:
case NodeType.KnitFB:
case NodeType.PurlFB:
case NodeType.Make:
case NodeType.MakeL:
case NodeType.MakeR:
case NodeType.KnitTog:
case NodeType.PurlTog:
case NodeType.SSK:
case NodeType.SSP:
case NodeType.PSSO:
UpdateExpressions(child, paramMap);
break;

case NodeType.NatVariable:
  var first = node.children.slice(0, i);
  var last = node.children.slice(i + 1);
  var exprChildren = paramMap[child.value].children;
  node.children = first.concat(exprChildren.concat(last));
  i += exprChildren.length - 1;
  break;

default:
  break;
}

When traversing the children of a node, if we find a sample call then we need to remove the sample call node from the tree.

<Pass 2 Replace Sample Call 128>

var first = node.children.slice(0, i);
var last = node.children.slice(i + 1);

var sampleChildren = GetSampleCallChildren(child, paramMap);

node.children = first.concat(sampleChildren.concat(last));

i += sampleChildren.length - 1

However, before the sample call node can be replaced we must acquire a deep copy of the children of the corresponding sample definition. These children are set as the children of the sample call node, which we then traverse to update their expressions and sample calls. To avoid naming conflicts, a localMap object is created as a deep copy of the parent node's paramMap. The paramMap object of the current node must have its expressions updated according to the local map. The localMap is then updated with the current node's updated paramMap and passed on when traversing the children of the sample call.

<Pass 2 Sample Call Children 129>
```javascript
var GetSampleCallChildren = function(node, paramMap) {

    var localMap = jQuery.extend(true, {}, paramMap);

    for (var domainVal in node.paramMap) {
        TraverseChildren(node.paramMap[domainVal], localMap);
        localMap[domainVal] = node.paramMap[domainVal];
    }

    var sampleDef = jQuery.extend(true, {}, State.samples[node.name]);
    node.children = sampleDef.children;

    TraverseChildren(node, localMap);

    return node.children;
};

var UpdateExpressions = function(node, paramMap) {
    if (node.repCount != null && node.repCount.type == NodeType.Expression) {
        TraverseChildren(node.repCount, paramMap);
    }
    if (node.num != null && node.num.type == NodeType.Variable) {
        TraverseChildren(node.num, paramMap);
    }
};
```

There are only two properties that currently allow natural number expressions: repCount and num. If a node has either of these properties, then the expression is updated according to the passed paramMap.

```javascript
var VerificationPass = function(ast) {
```

### 3.1.3 Pass 3: Pattern Verification

The third pass traverses the syntax tree and determines if there are structural issues with the pattern. See for details on verification of specific pattern elements.
```
var VerifyChildren = function(node) {

    if (node.children != null) {
        for (var i = 0; i < node.children.length; i++) {
            VerifyNode(node.children[i]);
        }
    }
};

var VerifyNode = function(node) {

    State.line = node.line;

    switch (node.type) {

        case NodeType.Root:
            VerifyNode(node.pattern);
            break;
        case NodeType.Pattern:
            VerifyPattern(node);
            break;
        case NodeType.Section:
            VerifySection(node);
            break;
        case NodeType.CastOn:
            VerifyCastOn(node);
            break;
        case NodeType.PickUp:
            VerifyPickUp(node);
            break;
        case NodeType.BindOff:
            VerifyBindOff(node);
            break;
        case NodeType.Join:
```
```javascript
VerifyJoin(node);
break;
case NodeType.Row:
    VerifyRow(node);
    break;
case NodeType.RowRep:
    VerifyExpression(node.repCount);
    VerifyChildren(node);
    break;
case NodeType.Expression:
    VerifyExpression(node);
    break;
default:
    break;
}
```
4

Code Generation

4.1 HTML

The back end for the Purl compiler generates HTML code representing the pattern. Given an abstract syntax tree, ast, representing a knitting pattern, PatternTextWriterHTML.Generate(ast) returns HTML to display the knitting pattern according to the knitting pattern standard [1]. Each pattern element is usually contained within a div with a corresponding class name. There are also tags added for pattern and section names. The order in which elements of a pattern are written is the same as has been seen for all passes of the compiler. See [2] for details on code generation of specific pattern elements.

```javascript
var PatternTextWriterHTML = (function () {

<<Code Gen Types 133>>
<<Code Gen Tag Writing Functions 134>>
<<Write HTML Node 136>>
<<Write HTML Root 115>>
<<Write HTML Pattern 92>>
<<Write HTML Section 105>>
<<Write HTML Cast-On 8>>
<<Write HTML Pick-Up 14>>
<<Write HTML Bind-Off 79>>
<<Write HTML Join 85>>
<<Write HTML Body 94>>
<<Write HTML Row 26>>
<<Write HTML Basic Stitch 54>>
<<Write HTML Undetermined Stitch Repeat 74>>
<<Write HTML Fixed Stitch Repeat 66>>
<<Write HTML Compound Stitch 60>>
<<Write HTML Row Repeat 98>>

```
return {
    Generate : function(ast) {
        return WriteRoot(ast);
    }
}
});

INCLUDED BLOCKS: 133 on page 84, 134 on page 85, 136 on page 85, 115 on page 68, 92 on page 54, 105 on page 62, 8 on page 10, 14 on page 13, 79 on page 48, 85 on page 51, 94 on page 56, 26 on page 18, 54 on page 33, 74 on page 45, 66 on page 41, 60 on page 38, 98 on page 58

Only div and span tag types are used, and a class type is associated with each pattern node in the output language (used for styling).

<Code Gen Types 133>

var ClassType = {
    Pattern : "pattern",
    PatternName : "patternname",
    PatternNote : "patternnote",
    SectionName : "sectionname",
    SectionNote : "sectionnote",
    CastOn : "caston",
    CastOnNote : "castonnote",
    PickUp : "pickup",
    Body : "body",
    Row : "row",
    RowNote : "rownote",
    RowRep : "rowrepeat",
    StitchCount : "stitchcount",
    Stitch : "stitch",
    BindOff : "bindoff",
    BindOffNote : "bindoffnote",
    Join : "join",
    Error : "error",
    Warning : "warning",
    Verification : "verification"
}

var TagType = {
    Div : "div",
    Span : "span"
}

USED IN: code/codegen.js on page 83
The following functions are used by the code generation module to create the HTML tags.

```javascript
var AddElement = function(tag, classType, text) {
    return OpenElement(tag, classType) + text + CloseElement(tag);
}

var OpenElement = function(tag, classType) {
    return "<" + tag + " class="" + classType + "">";
}

var CloseElement = function(tag) {
    return "</" + tag + ">");
}
```

The function `WriteNode` first checks if a node has any messages, and if so, an error tag is added to the result.

```javascript
if (node.hasErrorMsg) {
    result = AddElement(TagType.Span, ClassType.Error, "!");
}

if (node.hasWarningMsg) {
    result = AddElement(TagType.Span, ClassType.Warning, "!");
}

if (node.hasVerificationMsg) {
    result = AddElement(TagType.Span, ClassType.Verification, "!");
}
```

The appropriate node writing function is then called based on the node type.

```javascript
var WriteNode = function(node) {
    var result = "";

    if (node == null) {
```
return result;
}

<<Mark Node Message 135>>

switch (node.type) {
    case NodeType.Root:
        result += WriteRoot(node);
        break;
    case NodeType.Pattern:
        result += WritePattern(node);
        break;
    case NodeType.Section:
        result += WriteSection(node);
        break;
    case NodeType.CastOn:
        result += WriteCo(node);
        break;
    case NodeType.PickUp:
        result += WritePu(node);
        break;
    case NodeType.BindOff:
        result += WriteBo(node);
        break;
    case NodeType.Join:
        result += WriteJoin(node);
        break;
    case NodeType.Row:
        result += WriteRow(node);
        break;
    case NodeType.RowRep:
        result += WriteRowRep(node);
        break;
    case NodeType.FixedStRep:
        result += WriteFixedStRep(node);
        break;
}
break;

case NodeType.UStRep:
    result += WriteUStRep(node);
    break;

case NodeType.CompSt:
    result += WriteCompSt(node);
    break;

case NodeType.Knit:
    case NodeType.Purl:
    case NodeType.KnitTBL:
    case NodeType.PurlTBL:
    case NodeType.KnitBelow:
    case NodeType.PurlBelow:
    case NodeType.Slip:
    case NodeType.SlipKW:
    case NodeType.SlipPW:
    case NodeType.YarnOver:
    case NodeType.KnitFB:
    case NodeType.PurlFB:
    case NodeType.Make:
    case NodeType.MakeL:
    case NodeType.MakeR:
    case NodeType.KnitTog:
    case NodeType.PurlTog:
    case NodeType.SSK:
    case NodeType.SSP:
    case NodeType.PSSO:
        result += WriteBasicStitch(node);
        break;

    default:
        break;
    }

    return result;
};
Discussion

The language Purl provides a format for reuse and efficient storage of knitting patterns, with the compiler generating an assembled and formatted pattern for use by a knitter. In its present state, this project is a good foundation for writing knitting patterns, but many desirable features are still missing. Some planned components were removed from this initial implementation due to time constraints and a desire to give each component the time and thought deserved for a proper implementation. Below are listed the postponed features and next steps of this project, in approximate order of priority.

Preconditions/Postconditions
User defined requirements are a natural component for the knitting pattern samples construct. This concept requires more research and planning than initially anticipated. This is the top priority next large feature.

Test Page The test page provided gives an idea of how the language works, but for practical purposes, the following will be necessary:

- Load pattern files for compilation
- Allow pattern compilation over multiple files (pattern samples would then be reusable in multiple patterns)
- Save feature to download target pattern in pdf format

Charts
Originally the compiler was planned to have code generation to patterns in both the standard text notation and the chart notation [1]. Charts are written and read in a style that is very different from text patterns, with column repeats as well as row repeats, and alternating rows requiring the visual reverse of the written stitch to be performed. A large amount of optimization would have been required to make a pattern that is “concise” in Purl be output in a chart that is a reasonable size.

Small Updates In a productive pattern design setting, the following would be desirable:

- Display of pattern attributes such as designer name, date, recommended yarn, and needles
- Commenting in the pattern source file
- Notes on the target pattern
Appendices
Appendix A

Extra Productions

A.1 Expressions

Expressions currently only allow for addition of natural number literals and variables.

\[
\langle expr \rangle ::= (\langle Ident \rangle \mid \langle Nat \rangle) \ (\ '+\ \langle Ident \rangle \mid \langle Nat \rangle\ )^*
\]

An expression node contains children which are objects of type \texttt{NodeType.NatLiteral} or \texttt{NodeType.NatVariable}.

```javascript
var ExpressionParse = function(terminatorSym) {
    var node = { type : NodeType.Expression, children : [] };
    if (Sym.type == SymType.Nat || Sym.type == SymType.Ident) {
        node.children.push(getNatSym(terminatorSym));

        while (Sym.type == CharSym.PlusOp) {
            nextSym();
            node.children.push(getNatSym(terminatorSym));
        }
    }
    return node;
};
```

USED IN: Ast Construction Pass on page 75
An expression is verified by adding the values of the children (all of which should be natural number literals at the verification pass), and setting the value of the expression node.

```javascript
var VerifyExpression = function (node) {
    node.value = 0;

    if (node.children != null) {
        for (var i = 0; i < node.children.length; i++) {
            node.value += parseInt(node.children[i].value, 10);
        }
    }
};
```

```javascript
var WriteExpression = function (natObj) {
    var result = []; 
    var resultVal = 0;

    if (natObj.children != null) {
        for (var i = 0; i < natObj.children.length; i++) {
            var child = natObj.children[i];
            if (child.type == NodeType.NatVar) {
                resultVal += child.value;
            } else if (child.type == NodeType.NatLit) {
                result.push(child.value);
            }
        }
    }

    result.push(resultVal);

    return result.join("+");
};
```
A.2 Condition

\[ (\text{condition}) ::= (\text{expr}) \ (\text{'='} \ | \ '<' \ | \ '<=' \ | \ '>' \ | \ '>=' ) \ (\text{expr}) \]

A condition node includes a property for the comparison operator and a node for the left and a node for the right of the operator.

```javascript
var ConditionParse = function() {
  var node = { type: NodeType.Condition };
  if (Sym.type == SymType.Nat || Sym.type == SymType.Ident) {
    node.nodeL = ExpressionParse();
  }
  switch (Sym.type) {
    case CharSym.Equal:
      node.comparison = CompareType.Equal;
      nextSym();
      break;
    case CharSym.OpenAngle:
      node.comparison = CompareType.Less;
      nextSym();
      break;
    case CharSym.CloseAngle:
      node.comparison = CompareType.Greater;
      nextSym();
      break;
    case SymType.LessEq:
      node.comparison = CompareType.LessEq;
      nextSym();
      break;
    case SymType.GreaterEq:
      node.comparison = CompareType.GreaterEq;
      nextSym();
      break;
    default:
      break;
  }
  if (Sym.type == SymType.Nat || Sym.type == SymType.Ident) {
    node.nodeR = ExpressionParse();
  }
}
```
var UpdateCondition = function (node, paramMap) {

    TraverseChildren(node.nodeL, paramMap);
    TraverseChildren(node.nodeR, paramMap);

    node.nodeL.value = 0;
    if (node.nodeL.children != null) {
        for (var i = 0; i < node.nodeL.children.length; i++) {
            node.nodeL.value += parseInt(node.nodeL.children[i].value);
        }
    }

    node.nodeR.value = 0;
    if (node.nodeR.children != null) {
        for (var i = 0; i < node.nodeR.children.length; i++) {
            node.nodeR.value += parseInt(node.nodeR.children[i].value);
        }
    }

    switch (node.comparison) {
        case CompareType.Equal:
            node.doBranch = (node.nodeL.value == node.nodeR.value);
            break;
        case CompareType.Less:
            node.doBranch = (node.nodeL.value < node.nodeR.value);
            break;
        case CompareType.LessEq:
            node.doBranch = (node.nodeL.value <= node.nodeR.value);
            break;
        case CompareType.Greater:
            node.doBranch = (node.nodeL.value > node.nodeR.value);
            break;
        case CompareType.GreaterEq:
            node.doBranch = (node.nodeL.value >= node.nodeR.value);
            break;
        default:
            node.doBranch = false;
    }

    return node;
};
break;
}

};
Appendix B

Compiler Types

Below are types used throughout the compiler.

B.1 Parser Types

```javascript
var SideType = {
    RS : "RS",
    WS : "WS"
};

var CoType = {
    Flat : "flat",
    Circular : "circular",
    Prov : "provisional"
};

var RowType = {
    Row : "row",
    Rnd : "rnd"
};

var ColorType = {
    Main : "MC",
    Contrast : "CC"
};

var YarnPosType = {
    Front : "wyif",
    95
};
```
Back = "wyib"

var NodeType = {
    Root : "Root",
    Pattern : "Pattern",
    Section : "Section",
    CastOn : "CO",
    PickUp : "PU",
    BindOff : "BO",
    Join : "Join",
    Row : "Row",
    RowRep : "RowRepeat",
    SampleDef : "SampleDef",
    SampleCall : "SampleCall",
    FixedStRep : "FixedStRep",
    UStRep : "UndeterminedStRep",
    CompSt : "CompSt",
    Knit : "K",
    Purl : "P",
    KnitTBL : "KT",
    PurlTBL : "PT",
    KnitBelow : "KBelow",
    PurlBelow : "PBelow",
    Slip : "S",
    SlipKW : "SK",
    SlipPW : "SP",
    YarnOver : "YO",
    KnitFB : "KFB",
    PurlFB : "PFB",
    Make : "M",
    MakeL : "ML",
    MakeR : "MR",
    KnitTog : "KT",
    PurlTog : "PT",
    SSK : "SSK",
    SSP : "SSP",
    PSSO : "PSSO",
    Expression : "expr",
    NatLiteral : "NatLit",
    NatVariable : "NatVar",
    Branch : "Branch"
};

var MsgType = {
}
B.2 Symbol Lookup Types

```javascript
var CharSym = {
    Comma : ',',
    Period : '.',
    Colon : ':
    Asterisk : '*',
    PlusOp : '+',
    MinusOp : '-',
    Semicolon : ';',
    OpenParen : '(',
    CloseParen : ')',
    OpenBrack : '[',
    CloseBrack : ']',
    OpenAngle : '<',
    CloseAngle : '>',
    VerticalBar : '|',
    Equal : '='
};

var SymType = {
    Nat : "nat",
    Ident : "ident",
    String : "\",
    RowRep : "**",
    LessEq : "leq",
```
GreaterEq : "geq",
EOF : "!EOF",
Unknown : "?unknown"
);

var KeywordSym = {
  Pattern : "pattern",
  CastOn : "CO",
  PickUp : "PU",
  BindOff : "BO",
  Join : "Join",
  CastOnCirc : "circular",
  CastOnProv : "provisional",
  Section : "section",
  Sample : "sample",
  From : "from",
  To : "to",
  Last : "last",
  End : "end",
  Row : "row",
  Rnd : "rnd",
  Repeat : "repeat",
  With : "with",
  YarnInFront : "wyif",
  YarnInBack : "wyib",
  ColorMain : "MC",
  ColorContrast : "CC"
};

var StitchSym = {
  Knit : /^K$/,
  Purl : /^P$/,
  KnitTBL : /^KB$/,
  PurlTBL : /^PB$/,
  KnitBelow : /^K[1-9][0-9]*B$/,
  PurlBelow : /^P[1-9][0-9]*B$/,
  Slip : /^S$/,
  SlipKW : /^SK$/,
  SlipPW : /^SP$/,
  // Increases:
  YarnOver : /^YO$/,
  KnitFB : /^KFB$/,
  PurlFB : /^PFB$/,
  Make : /^M[1-9][0-9]*$/,
  MakeL : /^M[1-9][0-9]*L$/,
MakeR : /^M\[[1-9]\[0-9\]*R$/ ,
//Decreases:
KnitTog : /^K\[[1-9]\[0-9\]*T$/ ,
PurlTog : /^P\[[1-9]\[0-9\]*T$/ ,
SSK : /^SSK$/ ,
SSP : /^SSP$/ ,
PSSO : /^PSSO$/ }
};

PARTS OF THIS BLOCK: 142 on page 95, 143 on page 97
Appendix C

Built-In Examples and Tests

A number of pattern examples and tests are provided on the web page. These tests are intended to show the variety of features available in the language but are not exhaustive.

---

<Project Display Board Pattern 144>

```
sample edging with r:
  **
  row : *K; to end.
  repeat r

sample seedStitchBordered with r:
  **
  row : K 2 , *P, K; to last 3, P, K 2.
  repeat r

pattern "Project Display Board":
CO 79.
edging with 4.
seedStitchBordered with 70.
edging with 4.
BO 79.
```

---

<Market Bag Pattern 145>

```
sample circleX with n, max
  | n < max:
  rnd : [K, YO, K n, YO, K] 4.
  rnd : *K; to end.
circleX with n + 2, max.
```

---

USED IN: Pattern Tests on page 106
sample diagonalLace with n:
**
rnd : *K2T, YO; to end.
rnd : *K; to end.
repeat n

sample garterStitchCC with n, type
| type = 0:
**
row CC : *K; to end.
row CC : *P; to end.
repeat n
| type = 1:
**
rnd CC : *K; to end.
rnd CC : *P; to end.
repeat n

pattern "Market Bag":
section "Body":
CO 8 circular.
rnd : *K, YO, K; to end.
rnd : *K; to end.
circleX with 1, 23.
diagonalLace with 30.
garterStitchCC with 4, 1.
BO 100.

section "Handle":
PU 10 from "Body top".
garterStitchCC with 2, 0.
row : K, K2T, K 4, K2T, K.
garterStitchCC with 100, 0.
row : K, M1, K 6, M1, K.
garterStitchCC with 2, 0.
Join 10 to "Body top".

<Shawl Pattern 146>

sample shawlRep with m:
row : K 2, YO, K m, YO, K, YO, K m, YO, K 2.

USED IN: Pattern Tests on page 106
row : K 2, +P; to last 2, K 2.
row : K 2, YO, P m + 2, YO, K, YO, P m + 2, YO, K 2.
row : K 2, +P; to last 2, K 2.

sample shawlBody with m
\[m > 10:\]
shawlRep with m.
\[m \leq 10:\]
shawlRep with m.
shawlBody with m + 4.

pattern "Shawl":
CO 7.
shawlBody with 1.
BO 39.

| Basic Stitches Test 147 |

pattern "Basic Sts":
CO 40.
row : K 40.
row : P 40.
row : KB 40.
row : PB 40.
row : K1B 40.
row : P1B 40.
row : S 40.
row : SK 40.
row : SP 40.
row : K2T 20.
row : P2T 10.
row : SSK 5.
row : SSP 2, P.
row : S, K, PSSO, K.
row : K, YO, K.
row : KFB 3.
row : PFB 6.
row : K, MI, K 11.
row : K, MIL, K 12.
row : K, MIR, K 13.
BO 15.
<Compound Stitch Test 148>

pattern "Compound Stitch Test":
CO 20.
row : &lt;K, P &gt;, K 19.
BO 20.

<Fixed Stitch Repeat Test 149>

pattern "Fixed Stitch Repeat Test":
CO 18.
row : [K, P, K] 6.
BO 18.

<Undetermined Stitch Repeat Test 150>

pattern "Undetermined Stitch Repeat Test":
CO 100.
row : *K, P; to end.
row : K, *P; to last 1, K.
BO 100.

<Row Repeat Test 151>

pattern "Row Repeat Test":
CO 10.
**
row : *K; to end.
repeat 2
BO 10.
<Section Test 152>

pattern "Section Test":
section "first section":
CO 20.
row : K 20.
BO 20.

section "second section":
CO 5.
row : K, P 3, K.
BO 5.

USED IN: Pattern Tests on page 106

<Sample Test 153>

sample stockinette with m, n:
++
row : K m.
repeat n

pattern "Sample Test":
CO 20.
stockinette with 20, 3.
BO 20.

USED IN: Pattern Tests on page 106

<Sample Branch Test 154>

sample sampleBranch with m, n
| m = 0:
row : K n.
| m > 0:
row : P n.

pattern "Branch Test":
CO 4.
sampleBranch with 0, 4.
sampleBranch with 1, 4.
BO 4.

USED IN: Pattern Tests on page 106
Recursive Sample Test 155

| m <= n:  |
|---------|
| row : P m, *K; to end. |
| recursiveSample with m + 1, n. |
| m > n:  |
| row : *K; to end. |

pattern "Sample Recursion":
CO 20.
recursiveSample with 1, 10.
BO 20.

USED IN: Pattern Tests on page 106

Row Type Test 156

pattern "Row Type Test":
CO 20.
rnd : K 20.
row : P 20.
BO 20.

USED IN: Pattern Tests on page 106

Color Options Test 157

pattern "Color Test":
CO 20.
row MC : K 20.
row CC : P 20.
BO 20.

USED IN: Pattern Tests on page 106

Errors Test 158

pattern "Error Test":
CO 20.
row MC : K 20.
row CC : P 19.
row : knit
BO 20.
<div id="tests">
<input type="button" value="Shawl" onclick="javascript:loadTest(test0);" />
<div id="test0" class="pattern-example">
<<Shawl Pattern 146>>
</div>

<input type="button" value="Market Bag" onclick="javascript:loadTest(test01);" />
<div id="test01" class="pattern-example">
<<Market Bag Pattern 145>>
</div>

<input type="button" value="Project Display Board" onclick="javascript:loadTest(test02);" />
<div id="test02" class="pattern-example">
<<Project Display Board Pattern 144>>
</div>

<input type="button" value="Basic Stitches" onclick="javascript:loadTest(test1);" />
<div id="test1" class="test">
<<Basic Stitches Test 147>>
</div>

<input type="button" value="Compound Stitch" onclick="javascript:loadTest(test2);" />
<div id="test2" class="test">
<<Compound Stitch Test 148>>
</div>

<input type="button" value="Fixed Stitch Rep" onclick="javascript:loadTest(test3);" />
<div id="test3" class="test">
<<Fixed Stitch Repeat Test 149>>
</div>

<input type="button" value="Undetermined Stitch Rep" onclick="javascript:loadTest(test4);" />
<div id="test4" class="test">
<<Undetermined Stitch Repeat Test 150>>
</div>

<input type="button" value="Row Repeat" onclick="javascript:loadTest(test5);" />
<div id="test5" class="test">
<<Row Repeat Test 106>>
</div>
</div>
Appendix D

Test Page DOM

<code/index.html 160>

<!DOCTYPE html>

<html>
<head>
<link rel="stylesheet" type="text/css" href="styles/main.css"/>
<link rel="stylesheet" type="text/css" href="styles/pattern.css"/>
<script type="text/javascript" src="http://code.jquery.com/jquery-2.1.0.min.js"></script>
<script type="text/javascript" src="util.js"></script>
<script type="text/javascript" src="lexer.js"></script>
<script type="text/javascript" src="parser.js"></script>
<script type="text/javascript" src="codegen.js"></script>

function loadTest(idStr) {  
  $("#pattern-entry").val($(idStr).text().trim());  
}

function showMessages(msgArr) {  

  for (var i = 0; i < msgArr.length; i++) {  
    var msgObj = msgArr[i];  
    var msg = [];

    if (msgObj.sectionName !== null && msgObj.sectionName !== "") {  
      msg.push("Section: \\
    + msgObj.sectionName + \\
    );
    }

    if (msgObj.rowIndex > 0) {  
      msg.push("Row: \\
    + msgObj.rowIndex);
    }

    showMessages(msg);  
  }

}
```javascript
var cursorPos = 0;
if (msgObj.line != null) {
    msg.push("Line: " + msgObj.line.num + ":" + msgObj.line.pos);
    cursorPos = msgObj.line.charPos;
}
msg.push(msgObj.message);

var classStr = " class=" + msgObj.messageType + "";
var valueStr = " value=" + msg.join(", ") + "";
var onclick = " onclick="javascript:moveCursor(" + cursorPos + ");";
var tagStr = "<input type="button"" + classStr + valueStr + onclick + "/>";

$("#message-display ul").append("<li>" + tagStr + "</li>");
}

function moveCursor(pos) {
    document.getElementById("pattern-entry").selectionStart = pos;
    document.getElementById("pattern-entry").selectionEnd = pos;
    $('#pattern-entry').focus();
}

$(document).ready(function () {
    $('input').click(function () {
        $('pattern-display').empty();
        $('message-display ul').empty();
        var root = Parser.Parse($('pattern-entry').val());
        if (root.messages.length > 0) {
            showMessages(root.messages);
        }
        var output = PatternTextWriterHTML.Generate(root);
        $('pattern-display').prepend(output);
    });
});
</script>
</head>

<body>
<div id="header">
    <h1>Purl</h1>
</div>
```
pattern "Example Pattern":
section "This is a section of a pattern":
CO 4.
row : P, K, P, K.
row : [P, K] 2.
**
row : *P, K; to end.
repeat 2
BO 4.
INCLUDED BLOCKS: [159 on page 106]
Appendix E

Style Sheets

E.1 Test Page Styles

```css
* {
    margin: 0;
    padding: 0;
    border: 0;
    box-sizing: border-box;
}

html, body { height: 100%; }

html {
    background-image: url("../img/knitting.png");
    padding-top: 50px;
}

#header {
    position: absolute;
    top: 0;
    left: 0;
    right: 0;
    text-align: center;
    font: 25px overlockblackit;
    height: 50px;
}
```
```css
#left-content, #right-content
{
  width: 44%;
  height: 100%;
  min-width: 150px;
  min-height: 300px;
  padding: 10px;
}

#center-content { float: left; width: 12%; height: 100%; padding: 0px 0; }
#center-content-wrapper { height: 100%; position: relative; }

#left-content { float: left; }
#right-content { float: right; }

#left-content-wrapper, #right-content-wrapper
{
  width: 100%;
  height: 100%;
  box-shadow: 0px 1px 10px 1px #888888;
}

/*Left content*/

#pattern-entry
{
  display: block;
  width: 100%;
  height: 100%;
  padding: 10px;
  background-color: #333333;
  color: #E6E6E6;
  box-shadow: inset 0px 1px 10px 1px black;
  font: 16px convergence;
  line-height: 150%;
  letter-spacing: 1px;
}

@font-face { font-family: "shadows"; src: url("../res/ShadowsIntoLightTwo-Regular.ttf"); }
@font-face { font-family: "sofia"; src: url("../res/Sofia-Regular.ttf"); }

#entry-wrapper

113
```
| position: relative;  
height: 85%;  
padding-bottom: 80px; |

#compile
|
position: absolute;  
bottom: 0;  
width: 100%; |

#compile input, #tests input
|
color: white;  
background-color: #6194FF;  
border: 2px solid #286CFC; |

#compile input:hover, #tests input:hover { background-color: #709eff; }

#compile input:active, #tests input:active
|
box-shadow: inset 0px 0px 1px 1px #286CFC; |

#compile input
|
display: table;  
margin: 0 auto;  
padding: 5px 0;  
width: 100%;  
height: 80px;  
font: 30px overlockblackit; |

#message-display
|
width: 100%;  
height: 15%;  
min-height: 60px;  
background-color: #DBDBDB;  
border: 1px solid #CCCCCC;  
overflow-y: auto; |
ul#message-list li input
|
white-space: normal;
width: 100%;
text-align: left;
|
#message-display .error-message { border: 1px solid red; }
#message-display .warning-message { border: 1px solid yellow; }
#message-display .verification-message { border: 1px solid purple; }

/*@Center content*/

#tests-wrapper { height: 100%; padding: 10px 0; }
#tests { width: 100%; height: 100%; margin: 0 auto; padding: 10px; overflow-y: scroll; box-shadow: inset 0px 0px 10px 1px #525252; }
#tests::-webkit-scrollbar { display: none; }

#tests input
|
width: 100%;
min-height: 75px;
padding: 10px;
margin-bottom: 10px;
font: 20px overlockblackit;
white-space: normal;
|
.test, .pattern-example { visibility: hidden; display: none; }

/*@Right content*/

#pattern-display
|
width: 100%;
height: 100%;
overflow-y: auto;
padding: 10px 10px 0 10px;
background-color: white;
border: 1px solid #CCCCCC;
/*Other Styles*/

#weave-shadow-top
{
    position: absolute;
    width: 100%;
    height: 10px;
    z-index: 5;
    border-bottom: 1px solid #b9b9b9;
    -webkit-box-shadow: 0px 6px 10px -3px #525252;
    box-shadow: 0px 6px 10px -3px #525252;
}

#weave-shadow-bottom
{
    position: absolute;
    width: 100%;
    height: 10px;
    bottom: 0%;
    z-index: 5;
    border-bottom: 1px solid #b9b9b9;
    -webkit-box-shadow: 0px -6px 10px -3px #525252;
    box-shadow: 0px -6px 10px -3px #525252;
}

/*Fonts*/

@font-face { font-family: "artifica"; src: url("../res/Artifika-Regular.ttf"); }
@font-face { font-family: "delius"; src: url("../res/Delius-Regular.ttf"); }
@font-face { font-family: "novacut"; src: url("../res/NovaCut.ttf"); }
@font-face { font-family: "novaslim"; src: url("../res/NovaSlim.ttf"); }
@font-face { font-family: "radley"; src: url("../res/Radley-Italic.ttf"); }
@font-face { font-family: "convergence"; src: url("../res/Convergence-Regular.ttf"); }
@font-face { font-family: "overlockreg"; src: url("../res/Overlock-Regular.ttf"); }
@font-face { font-family: "overlockblackit"; src: url("../res/Overlock-BlackItalic.ttf"); }
@font-face { font-family: "overlockitalic"; src: url("../res/Overlock-Italic.ttf"); }
@font-face { font-family: "overlockblack"; src: url("../res/Overlock-Black.ttf"); }
@font-face { font-family: "overlockbold"; src: url("../res/Overlock-Bold.ttf"); }
@font-face { font-family: "novaround"; src: url("../res/NovaRound.ttf"); }
### E.2 Target Language Styles

```css
.pattern * { font: 20px overlockreg; }

.patternname { font: 30px overlockblack; }
.sectionname { font: 24px overlockbold; margin-top: 30px; }

caston, .bindoff, .join, .pickup { margin: 20px 0; }

.body { margin: 10px 0 15px 0; }

.row { margin: 5px 0; }

.rowrepeat { margin: 20px 0; }
.rowrepeat > * { padding-left: 10px; }

.stitch { font-family: overlockit; }

.stitchcount { font: 16px overlockit; }

.verification, .error, .warning { font: bold 16px arial; float: left; clear: left; margin-right: 5px; }
.verification { color: purple; }
.error { color: red; }
.warning { color: yellow; }

.ast-display { background-color: white; clear: both; display: none; visibility: hidden; }
#ast1-display, #ast2-display { margin-bottom: 110px; }
#ast3-display { margin-bottom: 110px; }
```
Bibliography

[1] Craft yarn council standards and guidelines for crochet and knitting. [http://www.craftyarncouncil.com/standards.html](http://www.craftyarncouncil.com/standards.html) Accessed: 2014-04-05.

[2] Eclipse standard. [https://www.eclipse.org/downloads/](https://www.eclipse.org/downloads/) 2014. Version: Kepler Service Release 2, Build id: 20140224-0627.

[3] Ravi Sethi Jeffrey D. Ullman Alfred V. Aho, Monica S. Lam. *Compilers: Principles, Techniques, and Tools (2nd ed)*. Prentice Hall, 2006.

[4] Therese de Dillmont. *Encyclopedia of Needlework*. Mulhouse (Alsace), 1886. Available free from Project Gutenberg.

[5] Seffen Ernst. Literate programming for eclipse. [http://lep.sourceforge.net/](http://lep.sourceforge.net/) 2012. Version: 0.5.21.

[6] Donald E. Knuth. Literate programming. *The Computer Journal*, 1984.

[7] Richard Rutt. *A History of Hand Knitting*. Batsford, Ltd., 1987.

[8] Laura Spradin. Grrlfriend market bag. [http://www.ravelry.com/patterns/library/grrlfriend-market-bag](http://www.ravelry.com/patterns/library/grrlfriend-market-bag) 2014. Knitting pattern, Accessed: 2014-04-12.

[9] Lesley Stanfield and Melody Griffiths. *The Essential Stitch Collection*. Reader’s Digest Association, 2010.