Java Art Chapter 8. A Compiler for Drawing Crop Circles

Some years ago I made the mistake of using biorhythms as the basis for a magazine article about dynamic imaging on the Web. I naively thought that a bit of pseudoscience would be a 'fun way' to discuss how to create on-the-fly graphics, especially since the coding only involved sine waves. I was wrong – people, who I can only kindly call "crazy-eyed loons", started coming out of the woodwork to ask my opinion on such worthy matters as Circatrigintan, Circavigintan and Circadiseptan cycles.

So it's with some trepidation that I turn to crop circles as a 'fun way' of describing the implementation of a compiler for a little programming language. To be clear: my admiration for crop circles extends only to the beauty of their design, emerging from little more than circles and straight lines. Figure 1 shows three examples: formations from Folly Barn in 2001, Tegdown Hill 2003, and Windmill Hill 2003.

Figure 1. Three Crop Circle Formations.

I am not a *cereologist*, an enthusiast who believes crop circles to be the work of extraterrestrials, manifestations of the Gaia hypothesis, or some magical mix of magnetic, radioactive and plasmic vortices. It seems extremely likely, to me at least, that every crop circle is the work of men (and occasionally women) armed with rope-and-plank contraptions called stompers.

Two good places to find out about crop circles is at the "How Stuff Works" site (http://science.howstuffworks.com/crop-circle.htm), or Wikipedia (http://en.wikipedia.org/wiki/Crop_circle).

My interest in crop circle programming was triggered by three articles by Andrew Glassner in *IEEE Computer Graphics and Applications*, running in the October 2004 to January 2005 issues. He developed a little language called Crop, which understands line and circle drawing operations, and outputs PDF diagrams. For example, the PDF output for the three crop circles in Figure 1 are shown in Figure 2.

Figure 2. Glassner's Crop Circle Drawings.

A summary of the Crop language can be found at Glassner's website, http://www.glassner.com/andrew/cg/research/crop/crop.htm. However, he doesn't explain how the language's compiler is implemented.

A little programming language, as compared to a sizeable language such as C++, is aimed at a limited problem domain, which means that it doesn't need to offer the panoply of programming constructs in mainstream languages. This makes the language easier to learn and use, and also simplifies its compiler implementation; all good reasons for using Crop in this chapter.

Glassner's language uses a postfix syntax, which involves pushing and popping operands and operations on and off a stack while they're being evaluated. I've gone for more conventional infix notation for my version of the language, with extra syntactic 'sugar' (e.g. keywords such as "let" and "draw") to make it look more familiar to Java programmers, and incidentally to make Crop programs easier to compile. These are really just surface differences – my language offers most of the same data types as Glassner's (e.g. numbers, points, lines, and circles), and operations (e.g. assignment, drawing, iteration).

I'll start by explaining my version of the Crop language through a series of examples, including code that generates the Figure 1 formations. Figure 3 shows its output.

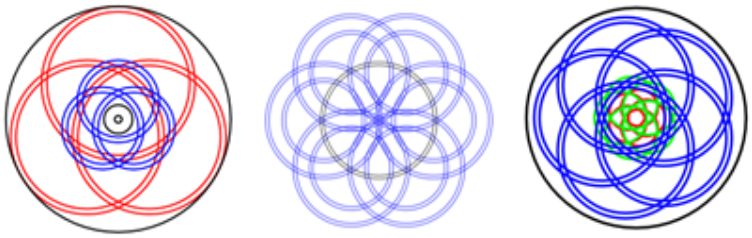


Figure 3. My Crop Circle Drawings.

Then I'll describe the three main parts of my Crop compiler, which are illustrated in Figure 4.

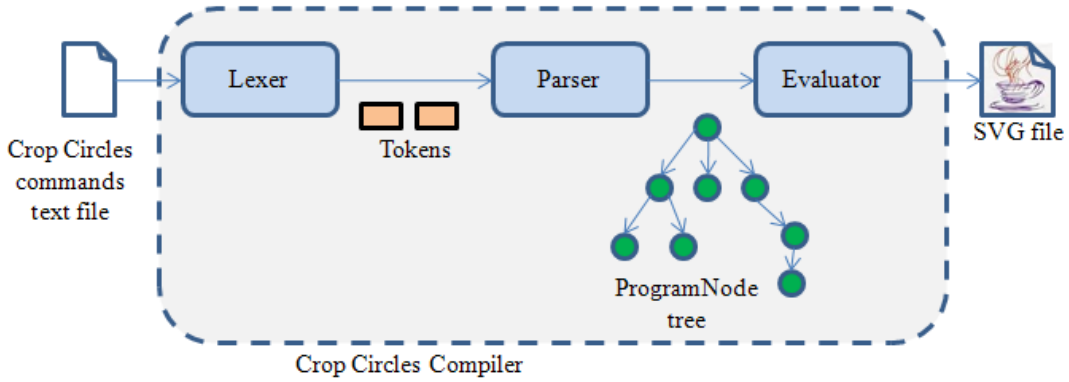


Figure 4. My Crop Language Compiler.

The Lexer reads in the text from a Crop Circles file as a stream of characters, and outputs a stream of tokens (e.g. variable names, keywords, numbers) to the Parser. The parser constructs a parse tree based on the Crop language's grammar, which is passed to the Evaluator. The evaluator employs the Apache Batik library (http://xmlgraphics.apache.org/batik/) to save Scalable Vector Graphics (SVG) drawing instructions into a file.

I call this application a compiler since the generated SVG code can be thought of as low-level instructions which will be rendered later by a browser or a SVG drawing application, such as Inkscape (http://www.inkscape.org/).

For readers familiar with compiler design, the lexer and parser are written directly in Java, without the aid of lex or yacc-style tools. Lexer employs Java's StreamTokenizer class, and Parser is a recursive descent parser that uses single token lookahead (i.e. it implements a LL(1) grammar). If those sentences just lost you, don't worry since this chapter explains all these ideas.

1. My Crop Language Explained

A Crop program can utilize four main commands: assignments using "let", a "draw" operation for circles and lines, a simple "print", and "cycle" (roughly equivalent to a while loop for circles). Variables are untyped, but there are four kinds of data: doubles, points, lines, and circles. A point is constructed from two doubles, representing an (x, y) coordinate. A line is composed from two points, and a circle is a center point and a radius.

For example, the following program draws one line and two circles:

```
draw line(origin point(30,0))     // origin is point(0,0)

let c2 = circle(point(30,0) 10)   // center and radius
draw c2

draw circle(origin 5)
```

The SVG rendering is shown in Figure 5.

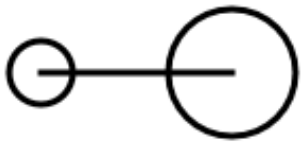


Figure 5. A Line and Two Circles

If several circles share a center, then they can be represented by a "circles" data type which has a single point and multiple radii:

```
draw circles( point(-5,-5)  10 15 20 25 30)
```

This is drawn as four concentric circles, as in Figure 6.

Figure 6. Concentric Circles.

A point can be specified as a coordinate, or with the "origin" keyword. There are also "intersect", and "turn" operations that return point results.

"intersect" takes two circle arguments and calculates their intersection. Normally the circles will intersect at two distinct coordinates, but "intersect" only returns the point which is on the left of an imaginary line going from the center of the first circle to the second. For example, the following code defines two circles that intersect at (3,4) and (-3,4):

```
let c1 = circle(origin 5)
let c2 = circle(point(6,0) 5)

draw red c1
draw blue c2

let p = intersect(c1 c2)
print "p =" p
```

Figure 7 illustrates the positioning of the circles, and the p intersection point.

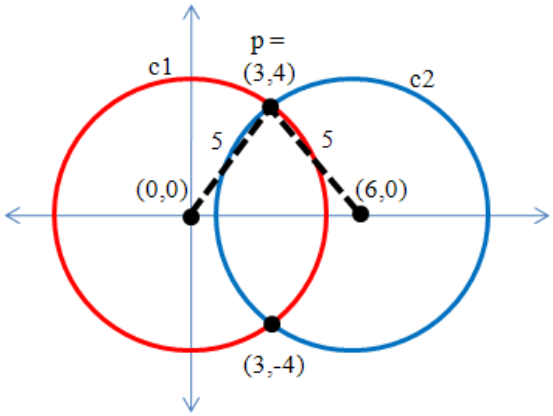


Figure 7. The Intersection of Two Circles.

The "print" command prints the value of the variable:

```
p = point(3.00, 4.00)
```

If the "right hand" point is required, then the programmer can use "intersect2" instead, and p will be assigned point(3, -4).

This example also illustrates the color argument of the "draw" operation. The possible values are: gray, red, green, blue, yellow, orange, with black being the default color.

It's possible for two circles to only intersect at a single point, or not intersect at all. In the latter case, null is returned, and an error message is printed.

"intersect" is based on Glassner's "trope" operation.

"turn" is the other point-calculating operation. It's supplied with a circle, a point on the circumference of the circle, and a counterclockwise rotation angle in degrees. The point is turned through that angle, and the resulting coordinate returned.

For example:

```
let radius = 1.41421   // sqrt(2)
let c1 = circle(origin radius)
let p1 = point(radius, 0)

let p = turn(c1 p1 45)
print "p =" p
```

This turns the point p1 anti-clockwise by 45 degrees, leaving it at (1,1):

```
p = point(1.00, 1.00)
```

Figure 8 illustrates the operation, which gives a simple answer because the circle's radius is approximately the square root of 2.

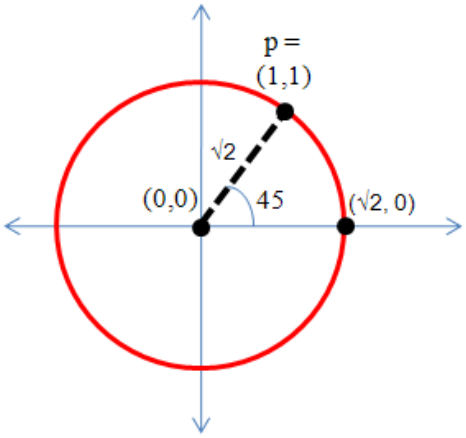


Figure 8. Turning a Point.

If the "turn" command is supplied with a starting point which is not on the circle's circumference, then the point is projected onto it by drawing an imaginary line between it and the circle's center which is extended until it touches the circle. That intersection point becomes the starting coordinate for the turn.

"turn" is based on Glassner's "pspin" operation.

Cycles

Glassner introduced a very useful looping construct, which I've utilized (under a different name). My "cycle" command is supplied with a circle, and a value representing the number of sides of a regular polygon. The user should imagine that the polygon is inscribed inside the circle, with its first vertex corresponding to the circle's intersection with the x-axis. Within the "cycle"s code block, "let", "draw", "print", and nested "cycle" commands can refer to this point as vertex_0. The other vertices of the polygon, progressing in a counterclockwise direction, are labeled as vertex_1, vertex_2, vertex_3, and so on. This is shown in Figure 9 for the case of a 6-sided regular polygon.

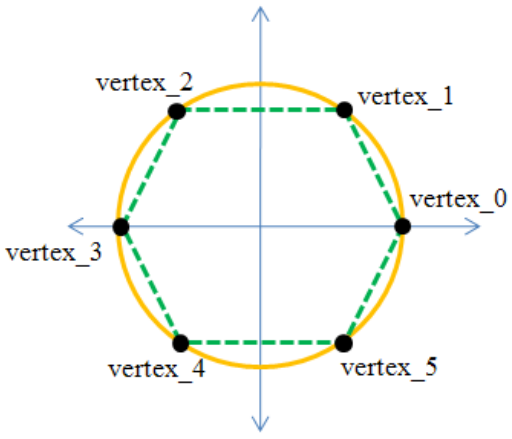


Figure 9. Cycle using a 6-sided Polygon.

The "cycle" command is a loop in the sense that it will execute its body block repeatedly, once for each corner of the polygon. As it iterates, the vertex labels (vertex_0, vertex_1, etc.) are moved counterclockwise. For instance, on the second iteration, vertex_0 will be at the old vertex_1 position, vertex_1 at the old vertex_2 position, and so on, as shown in Figure 10.

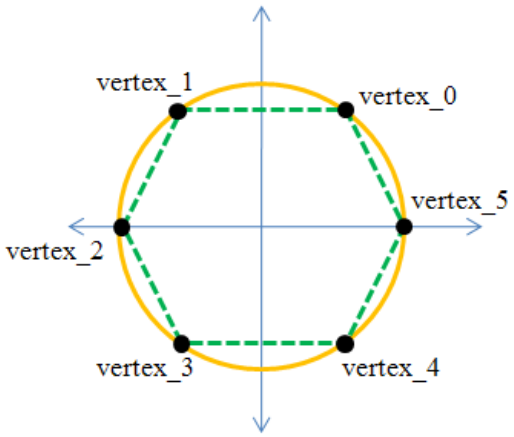


Figure 10. The Second Iteration of the 6-sided Cycle.

A counter recording the current iteration value can be accessed via the loopCounter variable. Initially it will be 0, and increments with each iteration.

The following program illustrates some of these ideas:

```
let c1 = circle( origin 10)
cycle c1 6 {     // a 6-sided cycle
   print "===============loopCounter=" loopCounter
   print "vertex_0=" vertex_0
   print "vertex_1=" vertex_1
}
```

It produces the following output:

```
===============loopCounter= 0.00
vertex_0= point(10.00, 0.00)
vertex_1= point(5.00, 8.66)
===============loopCounter= 1.00
vertex_0= point(5.00, 8.66)
vertex_1= point(-5.00, 8.66)
===============loopCounter= 2.00
vertex_0= point(-5.00, 8.66)
vertex_1= point(-10.00, 0.00)
===============loopCounter= 3.00
vertex_0= point(-10.00, 0.00)
vertex_1= point(-5.00, -8.66)
===============loopCounter= 4.00
vertex_0= point(-5.00, -8.66)
vertex_1= point(5.00, -8.66)
===============loopCounter= 5.00
vertex_0= point(5.00, -8.66)
```

```
vertex_1= point(10.00, 0.00)
```

A more typical example uses "vertex" values to draw circles around the perimeter of another circle:

```
let c1 = circle(origin 10)
draw red c1
cycle c1 6 {     // a 6-sided cycle
   draw blue circle(vertex_0 5)
}
```

Figure 11 shows the resulting output.

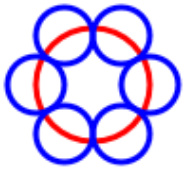


Figure 11. Red and Blue Circles.

A "cycle" command can be nested inside another "cycle", and it's possible for the inner cycle to refer to the vertices of the outer cycle by adding "^" to the end of the vertex label names.

The following code extends the previous example so that three black lines are drawn from each blue circle to their centers on the red circle.

```
let c1 = circle(origin 10)
draw red c1
cycle c1 6 {     // a 6-sided cycle
   let innerC = circle(vertex_0 5)
   draw blue innerC
   cycle innerC 3 {     // a 3-sided cycle
     draw line(vertex_0 vertex_0^)   // use of "^"
   }
}
```

The inner cycle refers to vertex_0 of the outer cycle by using vertex_0^.

Figure 12 shows the output.

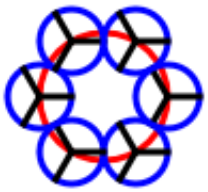


Figure 12. Red and Blue Circles with Black Lines.

By default, the n-sided polygon inside a "cycle" circle positions its first vertex on the x-axis. This starting point can be moved counterclockwise around the circle by supplying a degree value, or "%" which rotates the polygon by 180/sides degrees. This value for "%" turns the polygon so its vertices are midway between the old vertex positions.

The following cycle draws lines between the vertices inside the circle, showing that the polygon's corners have been rotated counterclockwise because of the "%".

```
let c1 = circle(origin 20)
draw red c1
cycle c1 6 % {    // turn the vertices with %
   draw line(vertex_0 vertex_1)
}
```

Figure 13 shows the hexagon and circle.

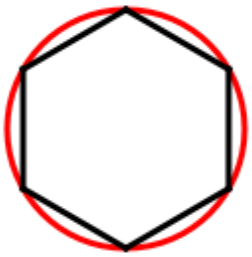


Figure 13. Hexagon Inside a Circle.

2. Crop Circle Examples

I began with Glassner's crop circle examples: Folly Barn in 2001, Tegdown Hill 2003, and Windmill Hill 2003 (Figures 1, 2, and 3). How are these implemented in my version of the Crop language?

Folly Barn is shown again in Figure 14.

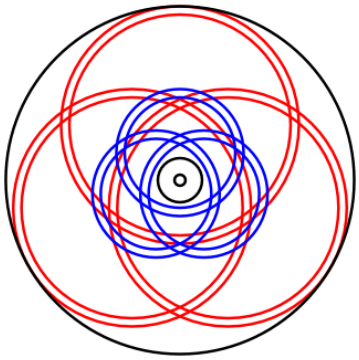


Figure 14. Folly Barn 2001.

There are three black circles, six red ones (three pairs), and six blue (another three pairs). The red and blue circles are created using two cycles of 3-sided polygons:

```
cycle circle(origin 20) 3 -90 {   // first 3-sided cycle
  draw red circles(vertex_0 40 43)
}

cycle circle(origin 10) 3 -90 {   // second 3-sided cycle
  draw blue circles(vertex_0 20 23)
}

draw circles(origin 8 63 2)   // three black circles
```

Tegdown Hill is shown in Figure 15.

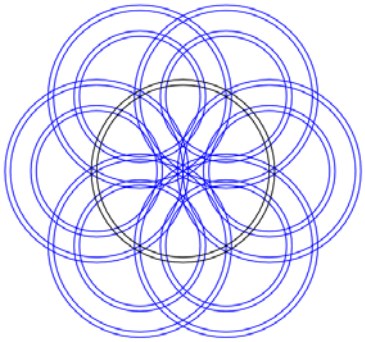


Figure 15. Tegdown Hill 2003.

There are two black circles in the middle, and six groups of four blue circles spaced around one of the black circle's perimeter. This is implemented as:

```
let a = 80
let b = 85

cycle circle(origin a) 6 {   // 6-sided cycle
   draw blue circles (vertex_0  a b 56 61)   // four blues per vertex
}

draw circles (origin a b)  // two black circles
```

Windmill Hill is shown in Figure 16.

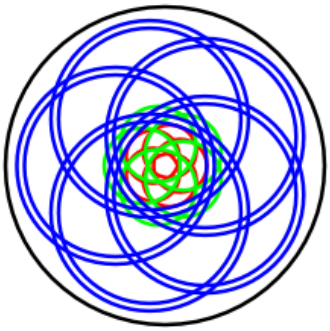


Figure 16. Windmill Hill 2003

The green and red circles in the center overlap, so it's hard to see that there's five each of them. Further out, there are five blue circle pairs, surrounded by a single black circle. The code consists of three 5-sided cycles for the green, red, and blue circles, and a single black circle.

```
draw circle(origin 42)        // black circle

cycle circle(origin 4) 5 {    // first 5-sided cycle
  draw red circle(vertex_0 7)    // red
}

cycle circle(origin 6) 5 % {  // second 5-sided cycle
  draw green circle(vertex_0  10)   // green
}

cycle circle(origin 13) 5 % {  // third 5-sided cycle
  draw blue circles(vertex_0 24 26)  // blue circle pair
}
```

3. Grammars and Parse Trees

The main purpose of this chapter is to use the Crop language as an example for discussing compiler design and implementation.

Compilers and interpreters are complicated beasts, but their main components (the lexical analyzer, syntactic analyzer, and evaluator; see Figure 4) can be greatly simplified by being based on a language *grammar*. In fact, there are compiler-generation tools which can semi-automatically convert a grammar into these components (e.g. lex and yacc in days of yore (http://dinosaur.compilertools.net/), and ANTLR more recently (http://www.antlr.org/)). I won't be using ANTLR here, since Crop is so simple (as are most domain-specific languages) that the translation of its grammar into code can be done by hand.

You may remember grammars from school, where you learnt that "the gerund is identical in form to the present participle (ending in -ing) and can behave as a verb within a clause (so that it may be modified by an adverb or have an object), but the clause as a whole (sometimes consisting of only one word, the gerund itself) acts as a noun within the larger sentence." (a quote from Wikipedia). You may also have

analyzed a sentence by writing it as a parse tree, such as the tree for "I gave Jim the card" in Figure 17.

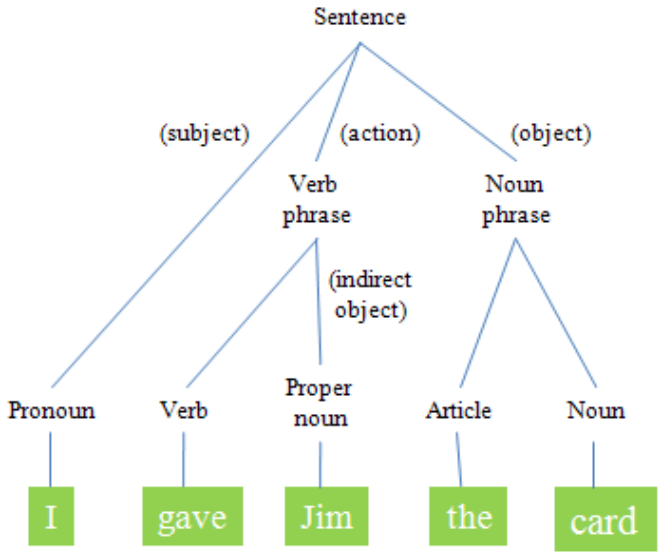


Figure 17. A Parse Tree for an English Sentence.

The sad truth is that natural languages (e.g. English) have very complex grammar rules which, in any case, rarely seem to cover the important things needed for actually using the language. The good news is that programming languages have very much simpler grammars, which do cover almost everything required for writing a program.

A grammar consists of four parts:

terminal symbols, or tokens. These are the words, such as "I", "gave", and "Jim", which appear as the leaves of the parse tree in Figure 17.

nonterminal symbols, or syntactic categories. In Figure 17, these are the grammar terms, such as Sentence, Verb_phrase, Article, which appear in the midst of the tree.

grammar rules, or productions, or rewrite rules. These explain how a nonterminal can be rewritten to other nonterminals and terminals.
e.g. Noun_phrase => Article Noun

the starting nonterminal. This is the syntactic category which appears at the top of the parse tree (e.g. Sentence in Figure 17).

The following grammar lets us construct simple arithmetic expressions involving single digits and '+' and '-' (e.g. 5 + 6 - 2):

Terminals = {+, -, 0, 1, 2, 3, 4, 5, 6, 7, 8, 9}

Nonterminals = {L, D}

Rules = { L => L + D | L – D | D
D => 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9
}

L is the starting nonterminal.

The "|" is read as "or", and allows a nonterminal to be rewritten in different ways. For instance, D can be replaced with 0 or 1 or 2, or any digit up to 9.

Starting with L (the starting nonterminal), we can perform a series of rewrites (called a *derivation*) which expands to our desired expression 5 + 6 – 2:

L => L - D
=> L + D – D
=> D + D – D
=> 5 + D – D
=> 5 + 6 – D
=> 5 + 6 – 2

At each rewrite step, there's usually a choice of applicable rules. For instance, the initial rewrite of L to L-D is correct, but it's also possible to expand L to L+D or to D, both of which lead to dead-ends. One of the arts of grammar writing is to make these choice points as simple as possible, so the resulting compiler will also be simple, and therefore fast. A common strategy is to choose a rule based on looking only at the current input token; grammars with this pleasant property are called LL(1) context-free grammars.

Derivations can be hard to follow as they get longer. For example, in the derivation above, is the '-' associated with the '6' or '5 + 6'? Such questions are easier to answer if the derivation is written as a parse tree, as in Figure 17b.

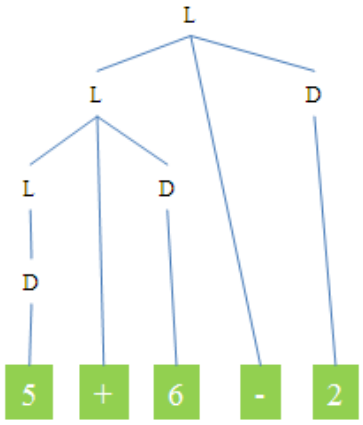


Figure 17b. A Parse Tree for "5 + 6 – 2".

The parse tree shows the ordering between the rewrites that make up the derivation. For instance, the '-' operator is applied to '5 + 6' since that subterm is rewritten from a single L nonterminal. Parse tree data structures are used by a compiler to store grammar information about the program being analyzed.

There are four main kinds of grammar, of increasing expressive power:

regular grammars

context-free grammars

context-sensitive grammars

unrestricted grammars

They vary in the kinds of rewrite rules they allow, and also in their ease of translation into compiler code. Regular grammars are the fastest, but don't have enough power to deal with common programming constructs, such as nested blocks, balanced braces, and complex if statements. Context-free grammars (CFGs) are powerful enough, so we'll use them from now on.

To make the grammar notation a little more compact, I'll introduce three more grammar operators: "[…]", "*", and "+". "[…]" around a sequence of grammar symbols indicates that they are optional. For example:

D => e [F] g H

means that D can be expanded to "e F g H" or "e g H" (the F is optional).

The "*" operator after a grammar symbol means that the symbol may be used 0, 1, or more times as required. For instance:

D => e F* g H

means that D can be expanded to "e g H", or "e F g H", or "e F F g H", and so on.

"+" is similar to "*" but means that its symbol must be used 1 or more times; the symbol must appear at least once.

ε is a useful operand, which means "nothing" or "no termininal". This can be used to make a nonterminal 'disappear', by rewriting it to nothing:

D => ε

4. The Crop Grammar

The context-free grammar for the Crop language is given below:

Terminals = all lowercase words, strings, and symbols in single quotes

Nonterminals = all uppercase words

Starting nonterminal = `PROGRAM`

Rules:

```
PROGRAM =>  COMMAND+
COMMAND  =>  LET |  DRAW  | CYCLE  | PRINT


LET  =>  let  ID  '='  ( EXPR | SHAPE )
DRAW =>  draw  [COLOR]  ( SHAPE | ID )
COLOR  =>  gray |  red |  green |  blue |  yellow |  orange
CYCLE  =>  cycle  (CIRCLE | ID)  EXPR(sides)  [ (EXPR(angle) | '%') ]
           '{'  COMMAND+  '}'
PRINT => print  STRING  ( EXPR | SHAPE )


EXPR  =>  TERM  (  ('*'|'/')  TERM )*
TERM  =>  FACTOR  (  ('+'|'-')  FACTOR )*
```



```
FACTOR  => NUMBER  | POINT  | VERTEX  | LOOPCOUNTER  |
           pi  | ID  |  '(' EXPR ')'


POINT  =>  point  '('  EXPR(x)  ','  EXPR(y)  ')' |   origin |
           INTERSECT  |   TURN
INTERSECT  =>   (intersect | intersect2)
                '('  (CIRCLE | ID)   (CIRCLE  ID)   ')'
TURN  =>  turn  '('  (CIRCLE | ID)
                     EXPR(point on circle)   EXPR(angle)  ')'
VERTEX  =>  vertex  '_'  (NUMBER  | ID)   '^'*
LOOPCOUNTER  =>  loopCounter   '^'*


SHAPE =>  LINE  |  CIRCLE  |  CIRCLES
LINE  =>  line  '('  EXPR(point)   EXPR(point)  ')'
CIRCLE  =>  circle   '(' EXPR(point)   EXPR(radius) ')'
CIRCLES  =>  circles   '('  EXPR(point)  EXPR(radius)+ ' )'
```

Unquoted brackets are used to emphasize the scope of operators. For instance, in

```
TERM  =>  FACTOR ( ('+'|'-') FACTOR )*
```

the "|" operator applies to '+' and '-', while the "*" is applied to the or'd pair and the second FACTOR.

Some of the nonterminals are followed by words in brackets. These indicate a semantic limitation on the nonterminal. For instance, EXPR(point) means that EXPR must eventually rewrite to a point, whereas EXPR can normally be a wider range of things, such as a number, arithmetic expression, or variable name.

A sharp-eyed reader will notice that several of the nonterminals are not defined, namely ID, STRING, and NUMBER. They are implemented as language tokens by the lexical analyzer (see later), along with keywords such as "point" and symbols such as '/'. I'll assume their existence without defining them in the grammar. They'll all be defined as tokens in the Lexer class below.

The grammar states that a program is a series of commands, which can be "let"s, "draw"s, "cycle"s, or "print"s. EXPR represents an expression which can utilize '*', '/', '+', '-' , brackets, IDs, numbers, the "vertex" and "loopCounter" variables, and points. A point can be defined using the "point" notation, or as the result of an "intersect", "intersect2", or "turn" operations.

The Crop grammar can be employed in the same way as the earlier expressions grammar. For example, the parse tree for the following Crop program:

```
let c1 = circle(origin 10)
draw c1
```

is shown in Figure 18.

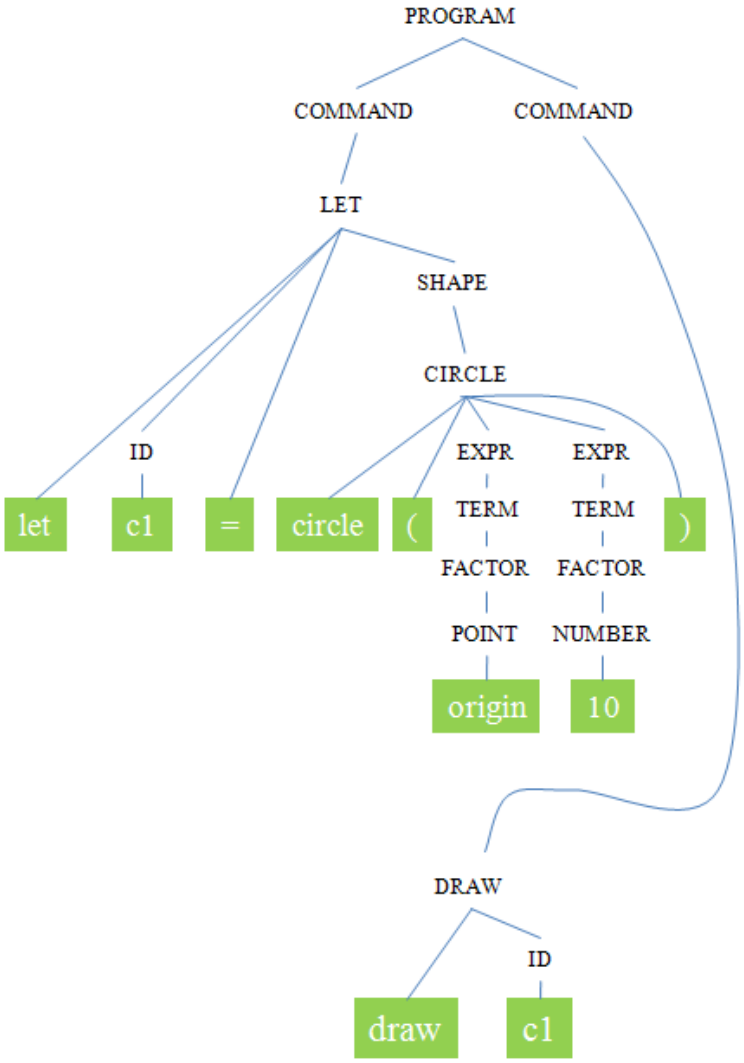


Figure 18. Parse Tree for a Crop Program.

5. Lexical Analysis

The first stage of any compiler is lexical analysis which converts a stream of characters read from an input file into the tokens (words, numbers, and symbols) used by the grammar as terminals. Our lexical analyzer, Lexer, will pass Token objects to the syntax analyzer for checking against the language grammar. As part of this checking, a parse tree will be generated.

Java contains a StreamTokenizer class which makes writing a lexical analyzer quite easy, *provided* the token types aren't too complex. StreamTokenizer can recognize identifiers, numbers, quoted strings, and ignores various comment styles. But it has some limitations:

StreamTokenizer doesn't come with a Token class, so we have to implement one.

There's no mechanism for distinguishing between ordinary words (e.g. variable names), and keywords (such as "draw" in the Crop language).

Only doubles are supported by the number tokenization.

Only single character symbols can be easily recognized (e.g. '=' and '+', but not "==" and "++" without some extra work).

Only Java-style comments are handled (i.e./* … */ and //), and there's a surprising default comment style which usually needs to be turned off.

StreamTokenizer doesn't consider '_' to be a word letter (which is often a shame when recognizing variable names), but this behavior can be changed.

My Lexer class starts by connecting a StreamTokenizer object to an input file, and defining the comments styles.

```
// global
private StreamTokenizer tokenizer = null;


public Lexer(String fnm)
{
  try {
    tokenizer = new StreamTokenizer( new FileReader(fnm));
    tokenizer.ordinaryChar('/');
    tokenizer.slashStarComments(true);
    tokenizer.slashSlashComments(true);
  }
  catch(IOException e)
  {  System.out.println("Could not open " + fnm);
     System.exit(1);
  }
}  // end of Lexer()
```

'/' is the default comment character in StreamTokenizer, which results in the parser ignoring all characters after '/' on the same line. I disable that by changing '/' into an ordinary character, and then switch on the /* … */ (slashstar) and // (slashslash) comment styles. Comments are ignored, which means no tokens are generated for their text.

After the StreamTokenizer object has been connected to the input stream, the StreamTokenizer.nextToken() method can retrieve a token from the stream. Either a single character is returned, or one of the constants: StreamTokenizer.TT_WORD, StreamTokenizer.TT_NUMBER, or StreamTokenizer.TT_EOF. If the token value is TT_WORD, then StreamTokenizer's sval variable will contain the complete word. If the token value is TT_NUMBER, then StreamTokenizer's nval variable will contain the token's numerical value as a double.

My Lexer class reads a token inside its scan() method and returns the token data as a Token object (I'll explain the Token class in a moment).

```
public Token scan()
// in my Lexer class
{
  int lineNo = -1;
  try {
    tokenizer.nextToken();
    lineNo = tokenizer.lineno();
  }
```



```
  catch (IOException e)
  {  System.out.println(e);
     return null;
  }

  switch (tokenizer.ttype) {
    case StreamTokenizer.TT_EOF:
      return new Token(Token.EOF, "EOF", -1, lineNo);

    case StreamTokenizer.TT_NUMBER:
      double val = tokenizer.nval;
      return new Token(Token.NUMBER, ""+val, val, lineNo);

    case StreamTokenizer.TT_WORD:    // keyword or identifier
      String word = tokenizer.sval;
      int keyPosn = Token.isKeyword(word);
      if (keyPosn >= 0)
        return new Token(Token.KEYWORD, word, keyPosn, lineNo);
      else
        return new Token(Token.NAME, word, -1, lineNo);

    case '\"':   // deal with a string
      String str = tokenizer.sval;
      return new Token(Token.STRING, str, -1, lineNo);

    default:    // deal with a symbol
      int type = tokenizer.ttype;
      if (Token.isSymbol((char)type))
        return new Token(Token.SYMBOL, ""+(char)type, type, lineNo);
      else {
        System.out.println("Unknown symbol: \'" +
                     (char)type + "\' on line " + lineNo);
      return null;
    }
  }
}  // end of scan()
```

The call to StreamTokenizer.nextToken() is followed by a switch statement which creates a different Token object depending on the token type. The switch has branches which deal with the end-of-file (StreamTokenizer.TT_EOF), numbers, words (distinguishing between identifiers and keywords), strings, and symbols (e.g. characters such as '+').

There's no need for scan() to deal with comments, since StreamTokenizer automatically ignores them. Also, StreamTokenizer treats all control characters, such as escape and backspace, as whitespace, which are also ignored.

5.1. Representing a Token

The recommended object-oriented approach for implementing tokens is to define an abstract Token class, and subclass it in appropriate ways to define classes for identifiers, keywords, numbers, and so on.

I haven't done that. Instead the six types of token in the Crop language are stored as instances of a single Token class. Token objects are distinguished by six Token type constants: Token.NAME, Token.KEYWORD, Token.NUMBER, Token.SYMBOL, Token.STRING and Token.EOF.

A Token object stores four values: its token type constant, the token text, its numerical value (only really useful for numbers), and the line number where the token was found in the input file (which is employed when reporting errors).

Apart from token type constants (which are integers), the Token class also stores textual names for the types, and lists of the Crop keywords and symbols:

```
// token types (encoded as ints)
public final static int NAME = 0;
public final static int KEYWORD = 1;
public final static int NUMBER = 2;
public final static int SYMBOL = 3;
public final static int STRING = 4;
public final static int EOF = 5;

private static final String[] TOKEN_NAMES = new String[] {
  "name", "keyword", "number", "symbol", "string", "eof"
          // same order as the token type constant integers
};

private static final String[] KEYWORDS = new String[] {
  "point", "origin", "intersect", "intersect2", "turn",
  "line", "circle", "circles", "let", "draw", "cycle",
  "print", "vertex", "loopCounter", "pi",
  "gray", "red", "green", "blue", "yellow", "orange"
};

private static final char[] SYMBOLS = new char[] {
  '{', '}', '=', '+', '-', '*', '/', '(', ',', ')', '_', '^', '%'
};
```

This data is accessible via static methods in Token:

```
public static int isKeyword(String value)
{
  for(int i=0; i < KEYWORDS.length; i++)
    if (value.equals(KEYWORDS[i]))
      return i;
  return -1;
}  // end of isKeyword()


public static boolean isSymbol(char value)
{
  for(char ch : SYMBOLS)
    if (value == ch)
      return true;
  return false;
}  // end of isSymbol()


public static String getTypeName(int type)
{
  if ((type >= 0) && (type < TOKEN_NAMES.length))
    return TOKEN_NAMES[type];
  return "unknown";
}
```


This coding approach separates grammatical aspects of the Crop language (e.g. its keywords and symbols) from the more general tokenization task carried out by the Lexer class. This means that if a new keyword needs to be added to the language, only the KEYWORDS[] array in Token has to be modified.

The Token constructor stores four values for each token:

```
// globals
private int type;     // token type constant
private String text;
private double value;
private int lineNo;


public Token(int t, String str, double val, int no)
{ type = t;
  text = str;
  value = val;
  lineNo = no;
}
```

5.2. Testing the Lexical Analyzer

It's quite easy to build a test-rig for the Lexer and Token classes:

```
// global
private static final String EXAMPS_DIR = "examples/";


public static void main(String args[])
{
  if (args.length != 1) {
    System.out.println("Supply a filename in " + EXAMPS_DIR);
    System.exit(0);
  }

  Lexer lexer = new Lexer(EXAMPS_DIR+args[0]);
  Token tok;
  do {
    tok = lexer.scan();
    System.out.println(tok);
  } while (tok.getType() != Token.EOF);
} // end of main()
```

main() repeatedly calls Lexer.scan() to return tokens, which are printed to stdout. For example, the tokenization of:

```
// example
let c1 = circle( origin 20)
```

produces the output:

```
< keyword, "let", 8.0, line 4 >
< name, "c1", -1.0, line 4 >
< symbol, "=", =, line 4 >
< keyword, "circle", 6.0, line 4 >
```



```
< symbol, "(", (, line 4 >
< keyword, "origin", 1.0, line 4 >
< number, "20.0", 20.0, line 4 >
< symbol, ")", ), line 4 >
< eof, "EOF", -1.0, line 5 >
```

The four outputs for each token are: the type constant name, the token's text, the token's value (or keyword index), and its line number.

6. Syntax Analysis

The next stage of the compiler is the conversion of the tokens supplied by the lexical analyzer into a parse tree (like the one in Figure 18).

The Crop parse tree uses Token information where possible. For example, there's no need to have NUMBER and ID nonterminals in the tree since the Token objects are typed – all number tokens have the type Token.NUMBER, and all ID tokens are typed as Token.NAME.

Syntax analyzers can build a parse tree either top-down (from the start symbol to the terminals) or bottom-up (from the terminals to the start symbol). My analyzer is top-down, implemented using the *recursive descent technique*. This makes it quite simple to map grammar rules into parsing code, so long as the grammar is LL(1) (i.e. a rule choice is based only on looking at the current input token sent from the lexical analyzer).

The mapping from grammar rules to analyzer code is done in two stages, as shown in Figure 19.

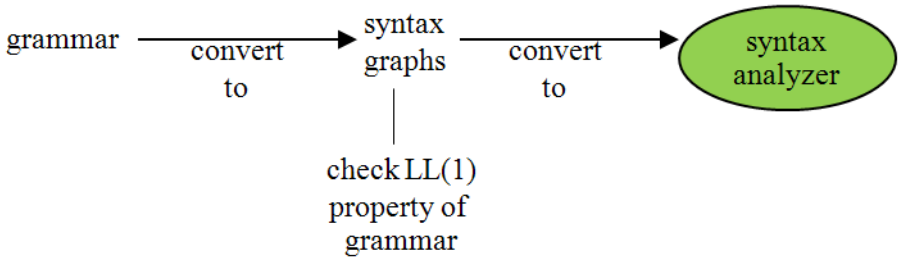


Figure 19. From Grammar to Syntax Analyzer.

The conversion of the grammar into syntax graphs, lets me utilize a relatively simple graphical way to check if the grammar is LL(1). The graphs are then converted into a collection of Java methods which carry out the analysis (see Figure 20).

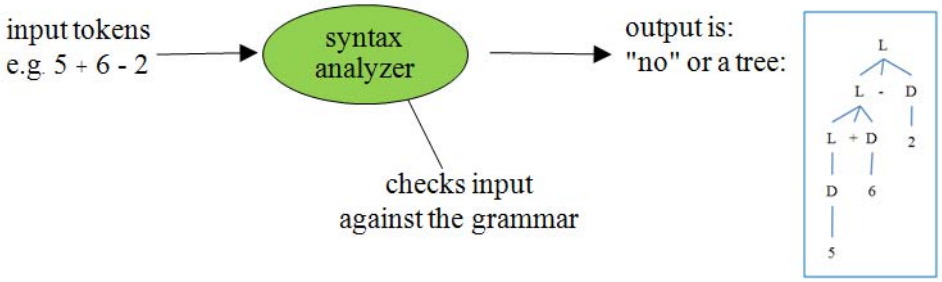


Figure 20. Syntax Analyzer in Action.

My first version of the analyzer only *checks* the input tokens, without building a tree. This makes it a bit simpler to explain the mapping of grammar to code. Then I'll add the parse tree data structures.

6.1. What are Syntax Graphs?

Syntax graphs are a visual way of representing grammar rules, which makes it easier to see whether the grammar is LL(1), at least for simple languages. The more usual LL(1) checking technique is to analyze a grammar by generating FIRST and FOLLOW sets of terminals for each nonterminals, but that's too technical for this article. If you want details, consult any good compiler textbook, such as *Compilers: Principles, Techniques, and Tools* by Aho, Lam, Sethi, and Ullman, Addison Wesley 2006.

As an example of syntax graph generation, consider the following grammar:

Terminals = {x, +, (,)}

Nonterminals = {A, B, C}

Rules = { `A => x | '( B ')'`
`B => A C`
`C => ( '+' A)*`
}

A is the starting nonterminal.

The grammar can generate strings such as "(x)" and "(x+x+x)".

A series of mappings (explained below) convert the grammar rule for each nonterminal into a syntax graph. The three nonterminals in the above grammar are mapped into the three graphs shown in Figure 21.

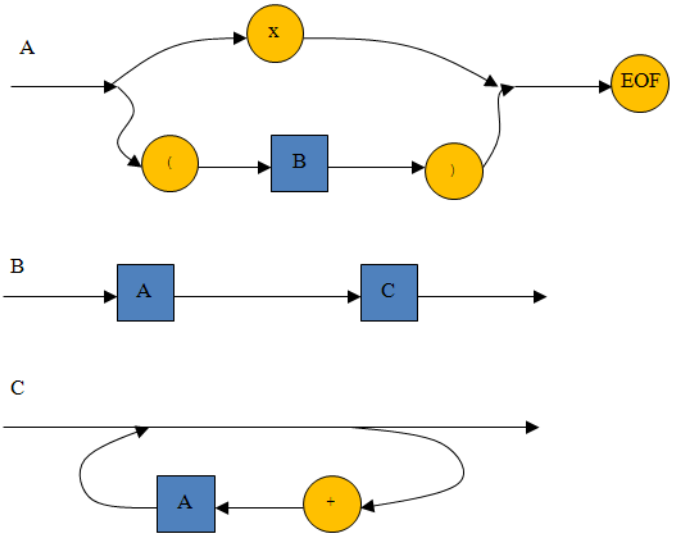


Figure 21. Three Syntax Graphs for the Grammar.

Syntax graphs can be 'executed' against a stream of input tokens. Consider a sequence of tokens, such as '(', 'x', ')'. The first token (the '(') is read in by the analyzer and the graph for the start symbol (A) is executed; this involves moving across the A graph, left-to-right, until the EOF circle is reached.

The A graph begins with a branching (choice) point, and the choice of which way to go must only depend on the current token if the grammar is LL(1). Looking along the two paths, there should be terminals (circles), which indicate which input tokens can be processed. The bottom branch starts with a '(' circle, and so that path is followed.

At a circle, the current input token is matched against the terminal inside the circle. If they are the same then the next token is read from the input sequence (i.e. the 'x'). The graph now comes to a 'B' box, which is treated as a procedure call – graph execution jumps to the start of the B syntax graph, and returns to the A graph when the traversal of B is completed.

Inside the B graph, execution jumps to a new instance of the A graph, whose top branch is followed to consume the 'x' token, and the next token (the ')') is read in. The A graph ends, returning to the B graph, which immediately calls the C graph.

Execution moves left-to-right across the C graph, and comes to another choice point – should execution move straight ahead and leave the C graph, or loop around to the '+' terminal? If the grammar is LL(1), the choice must only depend on looking along the two branches, to see which branch can consume the current token (the ')'). This time, it looks like there's a problem because the C graph has no circle along its horizontal path. In fact, it does, but this only becomes apparent if the three graphs in Figure 21 are merged into a single large A graph shown in Figure 22. This is done by replacing the B and C boxes by their corresponding graphs.

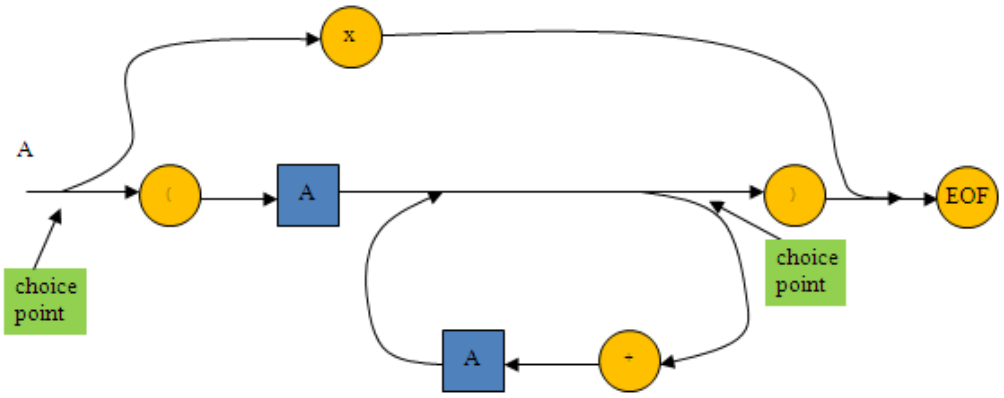


Figure 22. A Single 'A' Syntax Graph.

Figure 22 shows that the grammar has two choice points, and both are LL(1) – each path out of a choice point begins with a terminal, which allows a branch to be chosen based on the current input token alone.

In the example, execution has reached the right-hand choice point of Figure 22, and the current input token is ')'. Entering the loop will lead to failure since that branch expects a '+'. Instead, execution moves directly across the graph, the ')' is consumed, and the EOF circle matches the end of input.

6.2. From Grammar Rules to Syntax Graphs

Grammar rules are converted to syntax graphs using nine mappings, explained below.

Each nonterminal is assumed to have a single rule associated with it. If a nonterminal has several rules, then they can be combined into one by using the '|' operator. For example:

A => x and A => '(' B ')'

can be rewritten as:

A => x | '(' B ')'

Each nonterminal rule is converted to a graph, as in Figure 23.

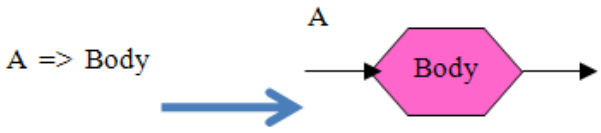


Figure 23. Rule as a Graph.

The hexagon denotes that the grammar notation in Body has still to be mapped.

A slight variation of this translation occurs if the nonterminal is the start symbol. In that case, the resulting graph ends with a EOF circle, as in Figure 23b.

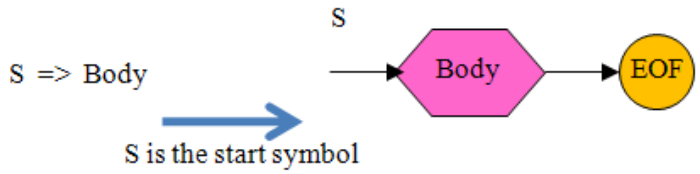


Figure 23b. Start Symbol Rule as a Graph.

A sequence of terminals and nonterminals in the rule body becomes a path in the graph, as in Figure 24.

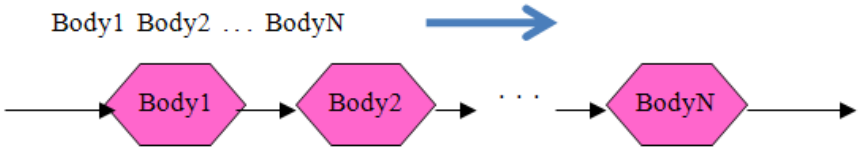


Figure 24. Rule Body Sequence as a Graph.

A terminal in the rule body becomes a circle, while a nonterminal is represented by a box (Figure 25).

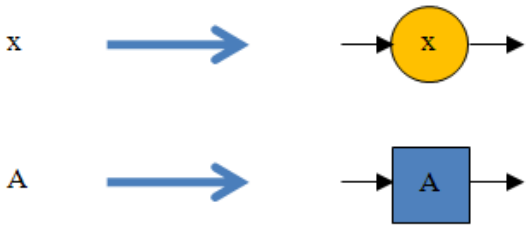


Figure 25. Rule Body Terminal and Nonterminal as Graphs.

A rule body involving '|' (the "or" operator) is mapped to a branching graph involving a choice point (Figure 26).

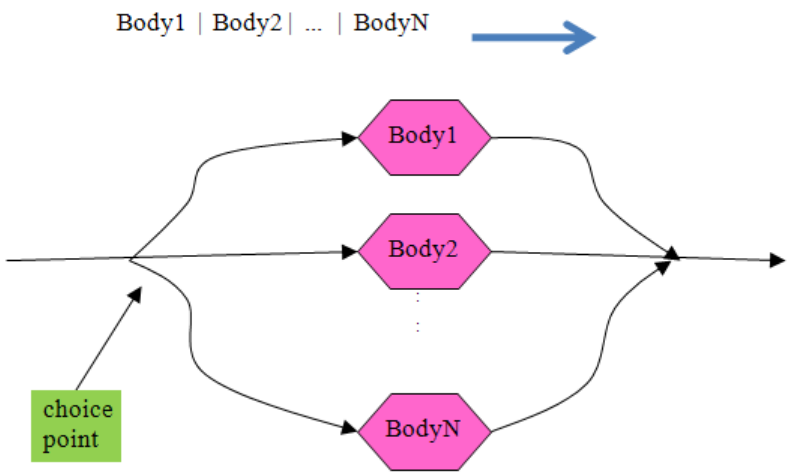


Figure 26. Rule Body '|'s as a Graph.

If the grammar is LL(1), then the hexagons along each branch will eventually be rewritten to subgraphs starting with a circle (a terminal). Also, each terminal will be different, as for example in Figure 27.

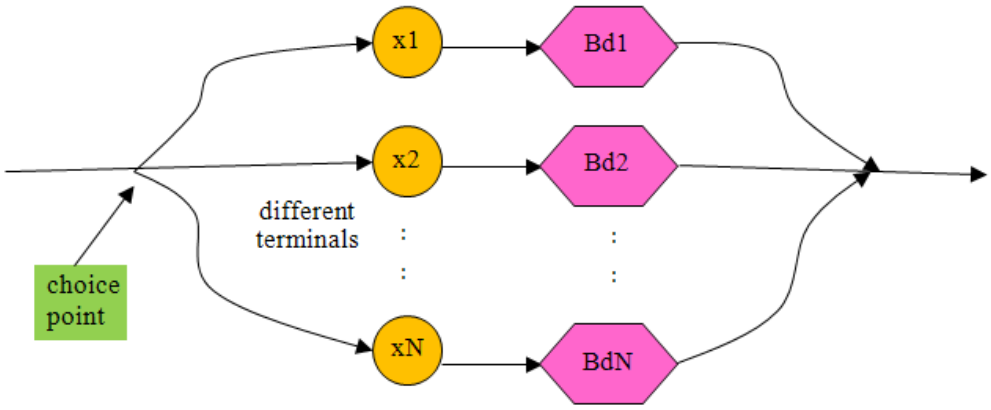


Figure 27. A LL(1) Expansion of a '|' Graph.

The expansion may have a different shape, as in Figure 27b, but the important property is that every choice point branch starts with a distinct circle.

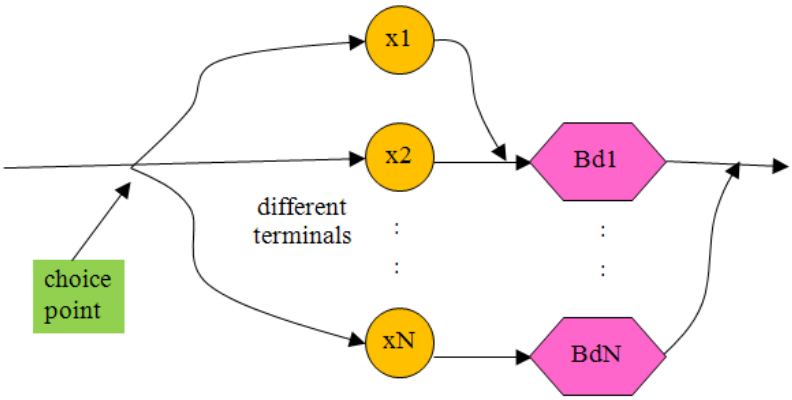


Figure 27b. Another LL(1) Expansion of a '|' Graph.

A rule body involving '[…]' (the "optional" operator) is mapped to a branching graph (Figure 28).

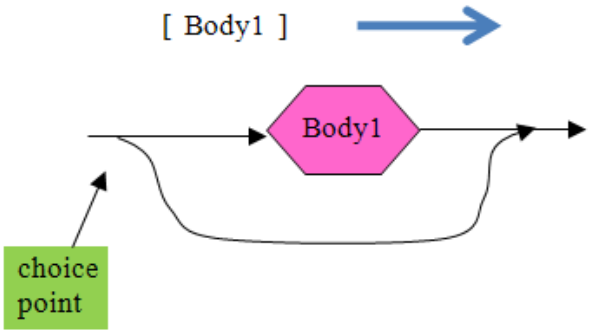


Figure 28. Rule Body […] as a Graph.

If the grammar is LL(1), then the Figure 28 graph must eventually be expanded so that the choice point leads to unique terminals along each branch, as for example in Figure 29.

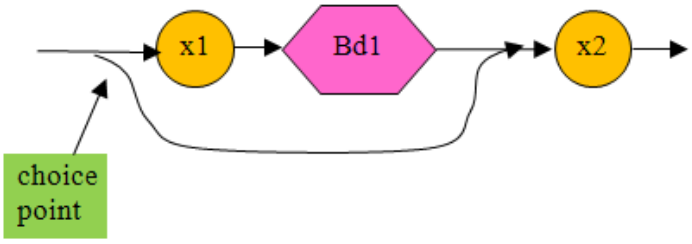


Figure 29. A LL(1) Expansion of the '[…]' Graph.

A rule body involving '*' (the "0 or more" operator) is mapped to a looping graph with a branch point (Figure 30).

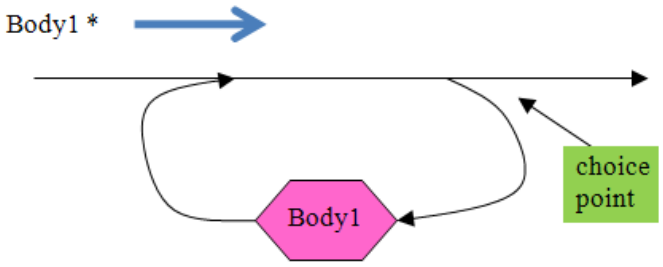


Figure 30. Rule Body '*' to Graph.

If the grammar is LL(1), then the Figure 30 graph must eventually be expanded so that the choice point leads to unique terminals along each branch, as for example in Figure 31.

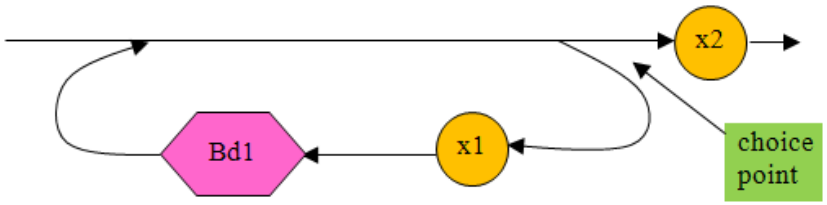


Figure 30. A LL(1) Expansion of the '*' Graph.

A rule body involving '+' (the "1 or more" operator) is also mapped to a looping graph with a choice point (Figure 31).

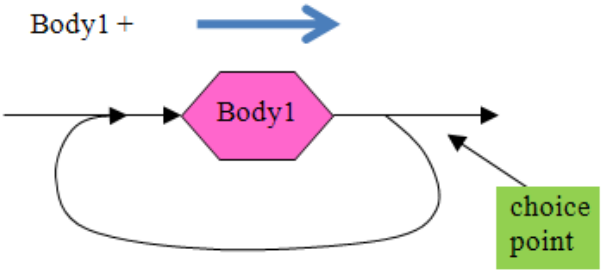


Figure 31. Rule Body '+' as a Graph.

If the grammar is LL(1), then the Figure 31 graph must eventually be expanded so that the choice point leads to unique terminals along each branch, as in Figure 32.

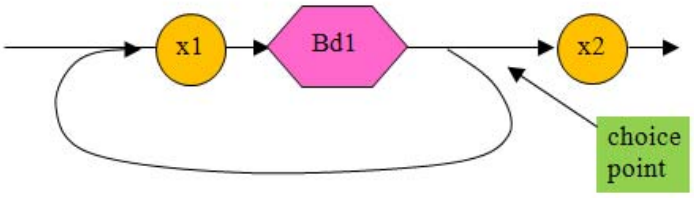


Figure 32. A LL(1) Expansion of the '+' Graph.

If the resulting syntax graphs for the grammar are not LL(1), then the translation of the graphs into code cannot proceed. In that case, the programmer has to return to the grammar, and modify it. There are a number of techniques, such as left recursion elimination and left factoring which can help. They are described in all good compiler texts, such as *Compilers: Principles, Techniques, and Tools* by Aho, Lam, Sethi, and Ullman. I've also included some informal tips on grammar design at the end of this chapter.

6.3. From Graphs to Code

Once the syntax graphs (e.g. see Figure 21) have been generated, and their choice points checked for the LL(1) property, another series of transformations convert the graph components into code. I'll generate Java code here, but since the mappings only use imperative features, such as assignment, if-statements, and while-loops, it's just as easy to output C, C++, or similar.

Each graph becomes a function. For example, consider the G graph in Figure 33.

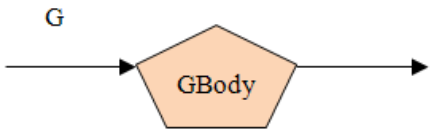


Figure 33. The G Syntax Graph.

It becomes:

```
private void G()
{  /* the code generated by transforming the graph GBody */
}
```

The pentagon in Figure 33 means that the graph GBody still has to be converted.

The graph for a start symbol, such as S in Figure 34, is treated a little differently.

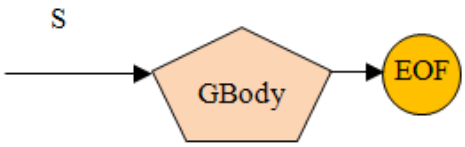


Figure 34. The Start Symbol S Syntax Graph.

A S() function is created, in a similar way to G(), but it only contains the code for the main part of the graph, excluding the EOF circle:

```
private void S()
{  /* the code generated by transforming the graph GBody */
}
```

The EOF circle is processed in the top-level parse() method, after S() has been called and returned:

```
// global
private Lexer lexer;
private Token token;    // the current input token

public Parser(String fnm)
{  lexer = new Lexer(fnm); }


public void parse()
{
  next();
  S();
  if (token.getType() == Token.EOF)
    System.out.println("Parsed succeeded");
  else
    System.out.println("Parsed failed");
}
```

Parser's constructor initializes the lexical analyzer, and parse() begins by retrieving a token from it via the next() method:

```
private void next()
{
  token = lexer.scan();
  if (token == null) {
    System.out.println("Missing token on line " + lexer.getLineNo());
    System.exit(1);
  }
}  // end of next()
```

next() gets a token from the lexer with Lexer.scan(), and stores it in the global token variable.

If the S() function is sufficiently simple, the programmer may choose to merge the parse() and S() functions, as I'll do in Crop's syntax analyzer.

A sequence of elements in a graph, as in Figure 35, becomes a series of statements inside the function representing the graph.

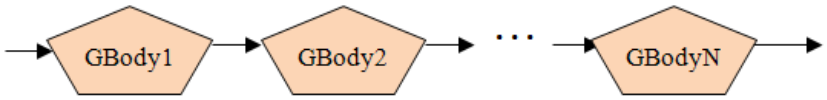


Figure 35. Sequence inside a Graph.

The sequence becomes the code:

```
{
  // transformation of GBody1 ;
  // transformation of GBody2 ;
            :
  // transformation of GBodyN ;
}
```

A nonterminal box for graph A becomes a call to the function A() (see Figure 36).

Figure 36. Converting a Nonterminal Box to a Function Call.

A terminal circle for 'x' becomes a call to the built-in function match() (see Figure 37).

Figure 37. Converting a Terminal Circle to a match() Call.

match() does three things – it checks the current input token against its argument, and if they match then another token is obtained from the lexical analyzer. If the match fails, then an error() method is called:

```
private Token match(String text)
// match using text
```



```
{
  Token t = null;
  if (token.isString(text)) {
    t = token;
    next();
  }
  else
    error("; expecting \"" + text + "\"");
  return t;
}  // end of match()


private void error(String msg)
// after an error, give up!
{
  System.out.println("Incorrect token \"" + token.getText() +
                     "\" on line " + token.getLineNo() + msg);
  System.exit(1);
}  // end of error()
```

This version of error() is a bit unforgiving: once a message has been printed, the syntax analyzer simply exits. There are better solutions, involving recovering from the error and continuing with the analysis, but my error() has the advantage of being easy to implement.

There's a second version of match(), which examines the input token type. This is useful when testing if a token is a number or identifier.

```
private Token match(int type)
// match using token type
{
  Token t = null;
  if (token.getType() == type) {
    t = token;
    next();
  }
  else
    error("; expecting \"" + Token.getTypeName(type) + "\"");
  return t;
}  // end of match()
```

The next(), match(), and error() functions remain the same even with different grammars.

Graph Branching

A graph branch will look like Figure 38.

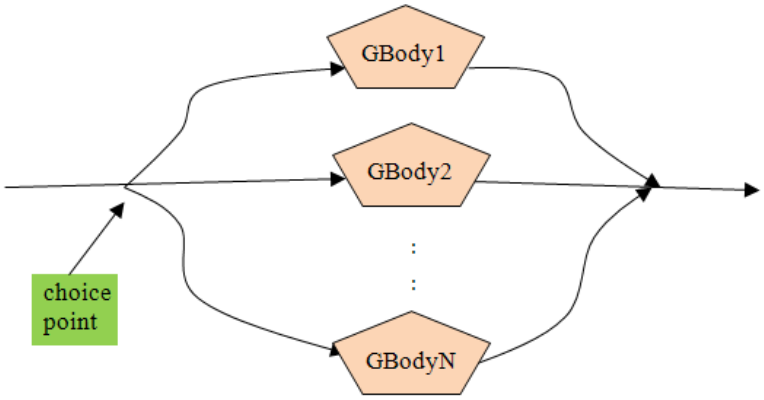


Figure 38. Branching in a Syntax Graph.

Because of the graph's LL(1) property, all the branches start with a unique terminal circle, for example as in Figure 39.

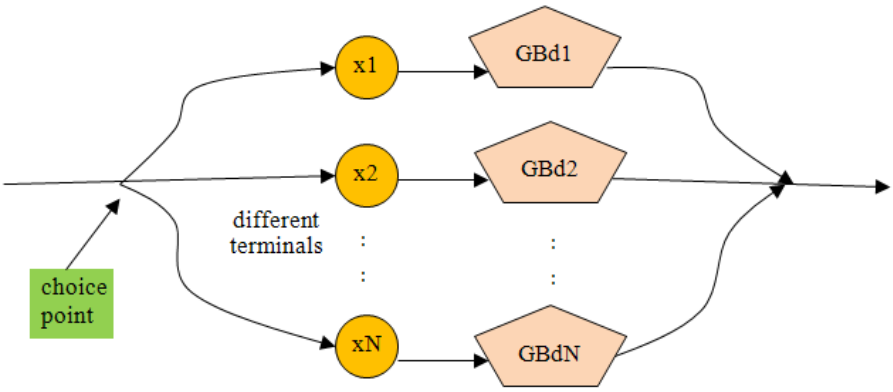


Figure 39. Expanded Figure 38 Branch Graph.

This makes it possible to translate the graph into a multiway if-statement:

```
if (token.isString("x1")) {
  match("x1");
  // transformation of GBd1;
}
else if (token.isString("x2")) {
  match("x2");
  // transformation of GBd2;
}
else if ...
      :
else if (token.isString("xN")) {
  match("xN");
  // transformation of GBdN;
}
else
  error();
```


Note the call to error() if the input token is not one of the allowed terminals.

The branching shape may be more complex than Figure 39. For example, Figure 40 has two branches leading into the GBd1 subgraph.

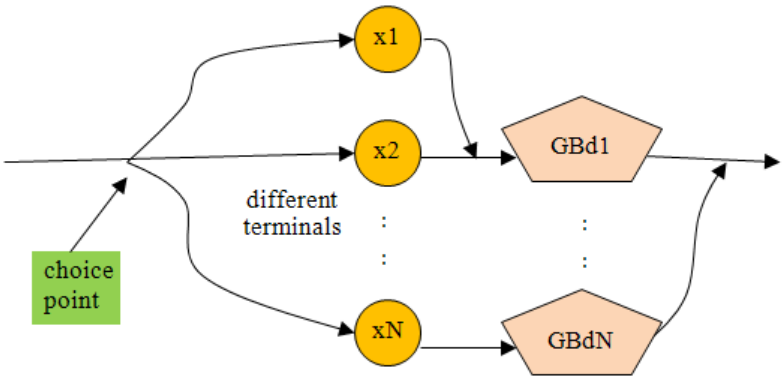


Figure 40. A More Complex Set of Expanded Branches.

This is still simple to translate, since each circle is a unique terminal.

```
if (token.isString("x1")) {
  match("x1");
  // transformation of GBd1;
}
else if (token.isString("x2")) {
  match("x2");
  // transformation of GBd1;   // two ways of getting to GBd1
}
else if ...
      :
else if (token.isString("xN")) {
  match("xN");
  // transformation of GBdN;
}
else
  error();
```

The branching graph generated by '[…]' is shown in Figure 41.

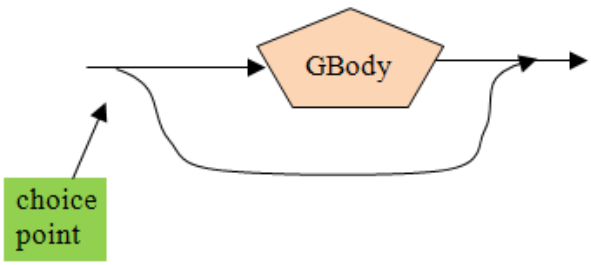


Figure 41. Branch Graph for the […] Operator.

The graph's LL(1) property means that the branches can be distinguished by their unique terminal circles, as in Figure 42

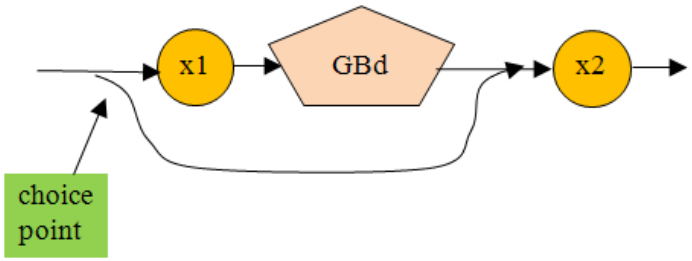


Figure 42. Expanded Figure 41 Branch Graph.

This allows an if-test to decide which branch to evaluate:

```
if (token.isString("x1") {
  match("x1");
  // transformation of GBd;
}
// else skip past the "x1" and subgraph
match("x2");
```

It's also possible to implement the branch by testing for the "x2" terminal that appears after the GBd subgraph:

```
if (!token.isString("x2") {   // use negation
  match("x1");
  // transformation of GBd;
}
// else skip past the "x1" and subgraph
match("x2");
```

The coding choice depends on the complexity of the branching surrounding the GBd subgraph.

Graph Looping

Graph loops are created when the * and + operators are employed in a grammar.

A graph derived from the * operator will look like Figure 43.

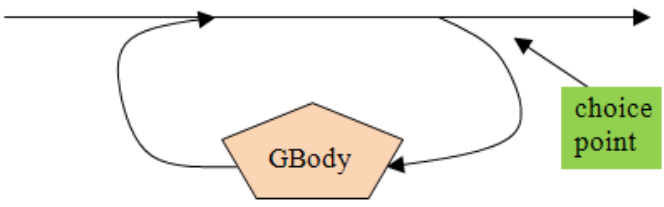


Figure 43. Looping Graph Based on the * Operator.

When this is expanded, the branch point has a unique terminal along the looping branch, (e.g. Figure 44), which can be converted into a while-test.

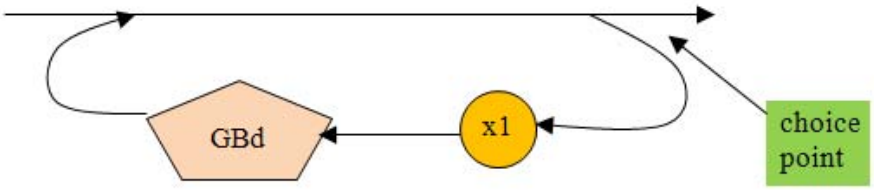


Figure 44. Expanded Figure 43 Branch Graph.

The conversion of Figure 43:

```
while (token.isString("x1") {
  match("x1");
  // transformation of GBd;
}
```

It's also possible to use the terminal that appears along the horizontal arm of the choice point to control the loop. The choice depends on the complexity of the branching.

A graph derived from the + operator will look like Figure 45.

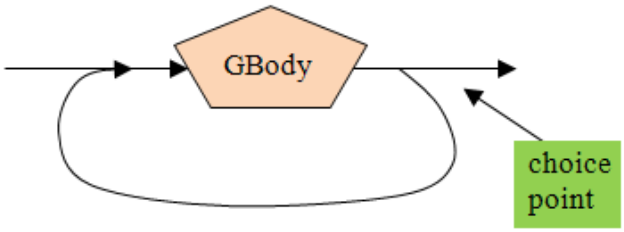


Figure 45. Looping Graph Based on the + Operator.

When this is expanded, the branch point has a unique terminal along the looping branch, (e.g. Figure 46), which can be converted to a do-while-test.

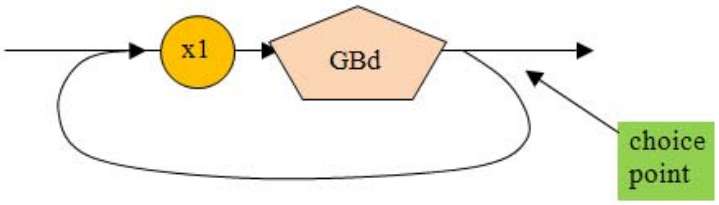


Figure 46. Expanded Figure 45 Branch Graph.

The conversion of Figure 46:

```
do {
  match("x1");
  // transformation of GBd;
}
while (token.isString("x1");
```

It's also possible to use the terminal that appears along the horizontal arm of the choice point to control the loop. The choice depends on the complexity of the branching.

6.4. Converting the A, B, C Graphs to Code

The syntax graphs generated in my earlier example are shown again in Figure 47.

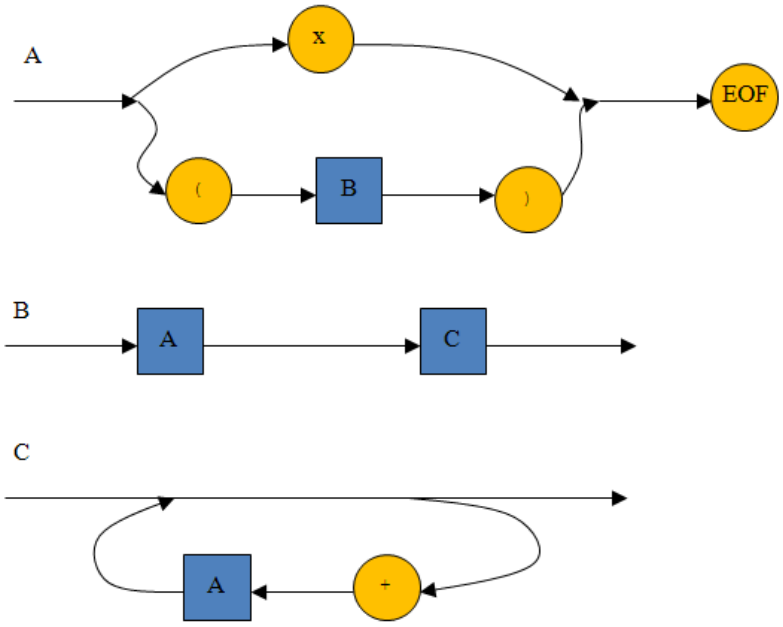


Figure 47. The A, B, and C Syntax Graphs.

Using the code generation rules just described, three methods are generated:


```
private void A()
{
  if (token.isString("x")
    match("x");
  else if (token.isString("(")) {
    match("(");
    B();
    match(")");
  }
  else
    error();
}  // end of A()


private void B()
{ A();
  C();
}


private void C()
{ while (token.isString("+")) {
    match("+");
    A();
  }
}
```

Since A() represents the start symbol, then the top-level parse() method becomes:

```
public void parse()
{
  next();
  A();
  if (token.getType() == Token.EOF)
    System.out.println("Parsed succeeded");
  else
    System.out.println("Parsed failed");
}
```

7. A Syntax Analyzer for the Crop Language

I'll carry out the same translations for Crop as for the previous language: first the Crop grammar is converted into syntax graphs to allow it to be checked for the LL(1) property. Then I'll translate the graphs into code. Initially, the code will only recognize input tokens, and I'll add parse tree building functionality in a separate stage.

The Crop grammar is quite large, so I'll restrict myself mainly to the top-level rules for the "let", "draw", "cycle", and "print" commands. A brief reminder of the grammar:

```
PROGRAM => COMMAND+
COMMAND => LET | DRAW | CYCLE | PRINT
```



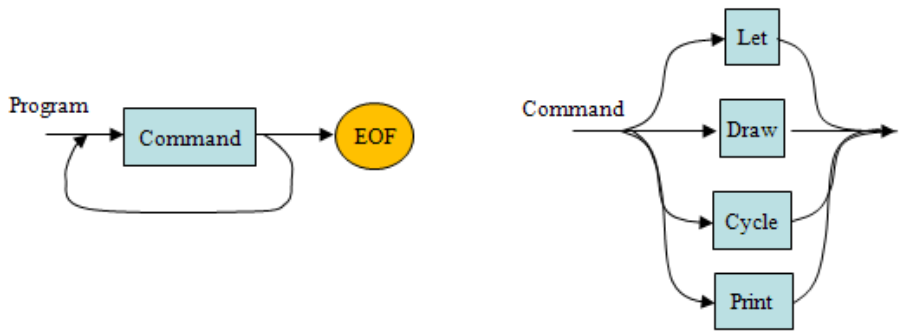
```
LET => let ID '=' ( EXPR | SHAPE )
DRAW => draw [COLOR] ( SHAPE | ID )
COLOR => gray | red | green | blue | yellow | orange
CYCLE => cycle (CIRCLE | ID) EXPR(sides) [ (EXPR(angle) | '%') ]
          '{' COMMAND+ '}'
PRINT => print STRING ( EXPR | SHAPE )


EXPR => TERM ( ('*'|'/') TERM )*
TERM => FACTOR ( ('+'|'-') FACTOR )*
FACTOR => NUMBER | POINT | VERTEX | LOOPCOUNTER |
            pi | ID | '(' EXPR ')'


POINT => point '(' EXPR(x) ',' EXPR(y) ')' | origin |
          INTERSECT | TURN
INTERSECT => (intersect | intersect2) '(' (CIRCLE | ID) (CIRCLE ID) ')'
TURN => turn '(' (CIRCLE | ID) EXPR(point on circle) EXPR(angle) ')'
VERTEX => vertex '_' (NUMBER | ID) '^'*
LOOPCOUNTER => loopCounter '^'*


SHAPE => LINE | CIRCLE | CIRCLES
LINE => line '(' EXPR(point) EXPR(point) ')'
CIRCLE => circle '(' EXPR(point) EXPR(radius) ')'
CIRCLES => circles '(' EXPR(point) EXPR(radius)+ ' )'
```

The syntax graphs for PROGRAM, COMMAND, LET, DRAW, CYCLE, and PRINT are shown in Figure 48.



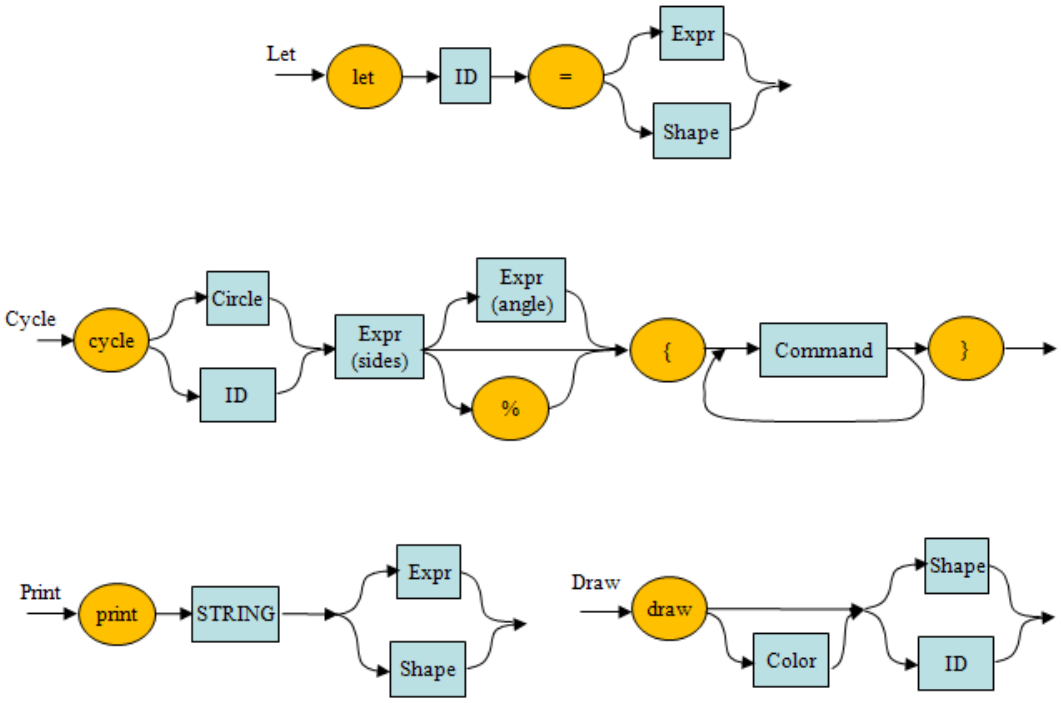


Figure 48. Syntax Graphs for Some of the Crop Language.

The aim of this stage is to check that all the choice points lead to unique terminals.

Straight away, there seems to be a problem with the Program graph, since the loop for the Command nonterminal does not start with a terminal circle. The Command graph also looks non-LL(1) since the branches for the commands (Let, Draw, Cycle, and Print) don't start with terminal circles.

Actually, Program and Command *are* LL(1), but this only becomes clear when several subgraphs are combined, resulting in Figure 49.

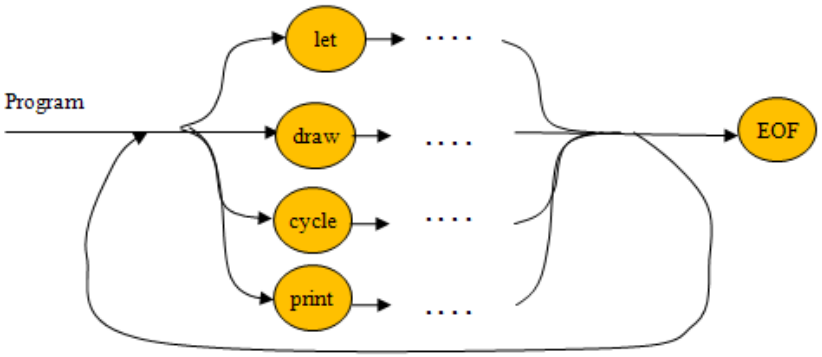


Figure 49. Part of the Combined Syntax Graph.

The Program loop can utilize the EOF terminal to control its iterations, while a multiway branch checks for the terminals which begin the four types of command.

The resulting Program and Command methods are:

```
public void parse()
// PROGRAM ::=  COMMAND+
{
  next();
  do {
    command();
  } while (token.getType() != Token.EOF);
     // use negation since EOF ends the loop
  System.out.println("Parsed succeeded");
}  // end of parse()


private void command()
// COMMAND  ::=  LET | DRAW | CYCLE | PRINT
{
  if (token.isString("let"))
    let();
  else if (token.isString("draw"))
    draw();
  else if (token.isString("cycle"))
    cycle();
  else if (token.isString("print"))
    print();
  else
    error(" in command syntax");
}  // end of command()
```

Note that I've combined the Program and top-level parse() methods, since the Program rule is so simple.

The Let syntax graph (Figure 49b) has a choice point after the assignment symbol ('='), which can lead to an expression or a shape. It's necessary to expand the Expr and Shape graphs to see if they start with unique terminals.

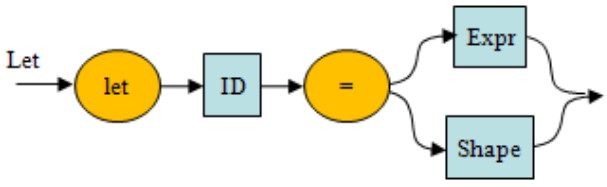


Figure 49b. The Let Syntax Graph.

An expression starts with a term, which begins with a factor, which can be seven different things, as shown in Figure 50.

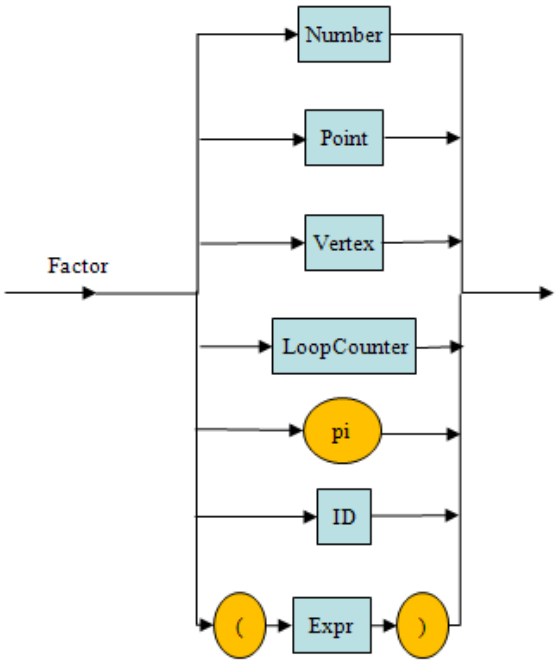


Figure 50. The Factor Syntax Graph.

Most of the branches in Factor start with unique terminals, which becomes apparent when the Point, Vertex, and LoopCounter nonterminals are replaced by their syntax graphs, as in Figure 51.

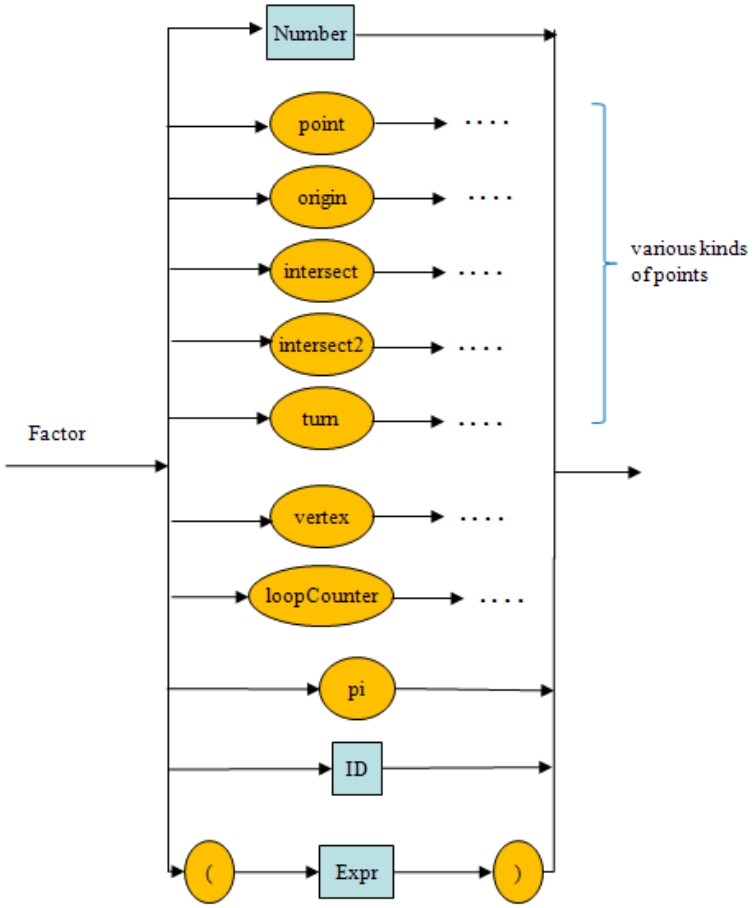


Figure 51. The Expanded Syntax Graph.

There are still two paths in Figure 51 which don't start with terminal circles – the Number and ID nonterminal boxes. However, NUMBER, ID and some other nonterminals such as STRING, are encoded as tokens, distinguishable by token types. For example, a NUMBER will be a Token object with the type Token.NUMBER, while ID is a Token with the type Token.NAME. Therefore, the parser can tell them apart from other terminals by examining their token types.

The Let syntax graph also utilizes the Shape nonterminal, which is also a little tricky to check for LL(1) (see Figure 52).

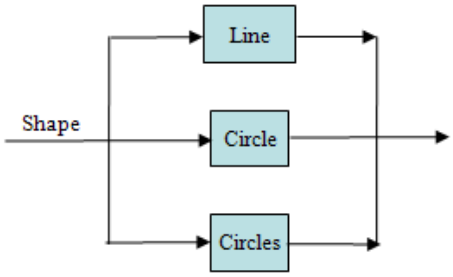


Figure 52. The Shape Syntax Graph.

The Shape graph has to be expanded with the subgraphs for Line, Circle, and Circles, which results in a graph like the one in Figure 53.

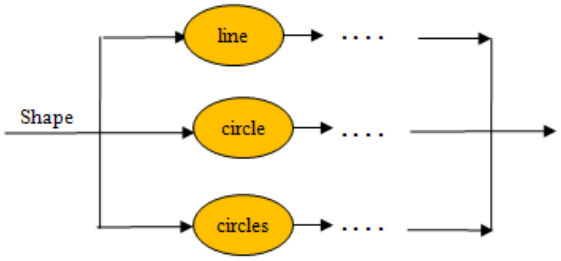


Figure 53. The Expanded Shape Syntax Graph.

The Line, Circle, and Circles graphs start with unique terminals ("line", "circle", and "circles"), and so the choice point in Shape is LL(1).

In summary, the Expr subgraph and Shape subgraphs do start with unique tokens, and so the choice point in the Let Syntax graph (shown once again in Figure 54) is LL(1).

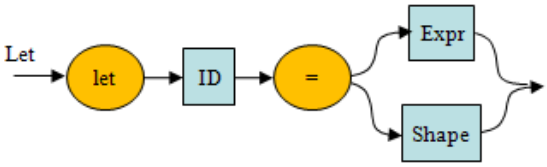


Figure 54. The Let Syntax Graph.

The resulting let() method is:

```
private void let()
// LET  ::=  let ID '=' ( EXPR | SHAPE )
{
  match("let");
  match("=");
  if (isShape(token))
    shape();
  else
    expr();
}  // end of let()


private boolean isShape(Token token)
// test for the possible tokens that start a shape
{
  if ( token.isString("line") || token.isString("circle") ||
       token.isString("circles"))
    return true;
  return false;
```



```
}  // end of isShape()
```

I've included an isShape() method for testing for Shape terminals (i.e.. "line", "circle", and "circles", as shown in Figure 53). I should also have a similar isExpr() method for testing for terminals that can start an Expr (i.e. those terminals shown in Figure 51). If the current token fails both the isShape() and isExpr() tests then let() should report an error.

However, I'm going to live dangerously instead. In let(), I've assumed that if the current token isn't for a shape (i.e. that isShape() returns false) then the token must start an expression, and so let() calls expr() without further testing. If I'm wrong then an error will still be reported, but somewhere inside expr().

This coding 'trickery' is part of the art of compiler writing. Often a parser can be speeded up by leaving out some of the tests implied by the translation of the syntax graphs.

8. Building a Parse Tree

Once the recognition parts of the syntax analyzer have been implemented and tested, the code has to be augmented to build a parse tree. The parser will pass this tree to the evaluator (the third component of the compiler shown in Figure 4).

If the language is very simple – typically one without loops, local variables, and functions – then it may be possible to skip tree creation, and augment the syntax analyzer directly with the evaluation code. In effect, this combines the syntax analyzer and evaluator into a single component.

A standard example of this combined analyzer/evaluator approach is a compiler for arithmetic expressions (e.g. x = 3 + 5; print x + 4;).

The Crop language is a little more powerful than an expressions language because of its "cycle" command for repeated evaluation. Therefore we have to keep the syntax analyzer and evaluator separate, and link them with a parse tree.

The parse tree consists of nodes holding information about the various nonterminals and terminals in the tree. How do we decided on the node types, and the data stored in the different nodes?

The language grammar can help answer these questions. A simple way to generate the node data structures is to create one node type for each nonterminal. The data fields in the nonterminal nodes can be dictated by the grammar rules for those nonterminals, which specify the nonterminals and terminals they employ.

For example, consider the grammar rules for Program, Command, and the four types of command (Let, Draw, Cycle, and Print):

```
PROGRAM =>  COMMAND+
COMMAND  =>  LET |  DRAW  | CYCLE  | PRINT

LET  =>  let  ID  '='  ( EXPR | SHAPE )
DRAW => draw  [COLOR]  ( SHAPE | ID )
```



```
CYCLE  =>  cycle  (CIRCLE | ID)  EXPR(sides)   [ (EXPR(angle) | '%') ]
             '{'  COMMAND+  '}'
PRINT => print  STRING  ( EXPR | SHAPE )
```

The six nonterminals on the left of the rules are mapped to the following six 'node' classes:

```
class ProgramNode {
// PROGRAM => COMMAND+
   ArrayList<CommandNode> cmds;
   ProgramNode(ArrayList<CommandNode> cs) {cmds = cs;}
}

abstract class CommandNode {}


class LetCmdNode extends CommandNode {
// LET  =>  let ID '=' ( EXPR | SHAPE )
   String id; ExprNode expr; ShapeNode shape;

   LetCmdNode(String i, ExprNode e, ShapeNode s)
   {id=i; expr=e; shape=s;}
}


class DrawCmdNode extends CommandNode {
// DRAW => draw [COLOR] ( SHAPE | ID )
   String color; ShapeNode shape; String id;

   DrawCmdNode(String col, ShapeNode s,String i)
   {color=col; shape=s; id=i;}
}


class CycleCmdNode extends CommandNode {
// CYCLE  =>  cycle  (CIRCLE|ID) EXPR(sides)  (EXPR(angle)|'%')
//              '{' COMMAND+ '}'
   CircleNode circle; String id; ExprNode sides; ExprNode angle;
   boolean halfRot; ArrayList<CommandNode> cmds;

   CycleCmdNode(CircleNode c, String i, ExprNode s, ExprNode a,
                  boolean hr, ArrayList<CommandNode> cs)
   {circle=c; id=i; sides=s; angle=a; halfRot=hr; cmds=cs;}
}


class PrintCmdNode extends CommandNode {
// PRINT  =>  print STRING ( EXPR | SHAPE )
   String printStr; ExprNode expr; ShapeNode shape;

   PrintCmdNode(String str, ExprNode e, ShapeNode s)
   {printStr=str; expr=e; shape=s;}
}
```

Each nonterminal becomes a 'node' class, with a constructor for initializing its data, and public fields so its data can be accessed directly. The choice of data fields

corresponds to the grammar elements that appear in the nonterminal's rule body. For example,

```
PRINT  =>  print STRING ( EXPR | SHAPE )
```

is converted into a PrintCmdNode class with three fields for the string, expression and shape. The data fields for expression and shape are their corresponding 'node' classes, ExprNode and ShapeNode.

Usually there's no need to include fields for the rule body terminals (e.g. the "print" keyword in the Print rule). If terminals do need to be represented, typically an existing Java type is sufficient, such as String or boolean. For instance, in CycleCmdNode above, the presence or absence of the "%" terminal is represented by the halfRot boolean being set to true or false.

If a nonterminal is defined in terms of a single rule using only '|' to distinguish between different values (e.g. the Command rule), then it can be translated into an empty abstract class (e.g. CommandNode). The rule's possible body values (e.g. Let, Draw) can be converted into CommandNode subclasses (e.g. LetCmdNode, DrawCmdNode).

If the operators '*' or '+' are employed with a nonterminal (e.g. Command+ in the Program rule) then the corresponding data field can be represented by an ArrayList of that nonterminal's node class (e.g. ArrayList<CommandNode>).

Each node class only has a single constructor method. Often it is also useful to include a print() method, which can be utilized when debugging the parse tree during code development.

A more object-oriented approach to parse tree data structure creation is to define a general Node class from which all the other node classes inherit. The main benefit of this is to simplify the use of polymorphic data structures, such as ArrayList<Node>, to hold any kind of 'node' object. I haven't done that, but since all Java classes have the Object class as a common ancestor, I can still create polymorphic structures by casting all the nodes to Object.

Using the Node Data Structures

Each of the parse methods developed previously can now be augmented to return its corresponding 'node' object as a result. For example, the parse(), command() and let() methods become:

```
public ProgramNode parse()
// PROGRAM =>  COMMAND+
{
  ArrayList<CommandNode> cmds = new ArrayList<CommandNode>();
  next();
  do {
    cmds.add( command() );
  } while (token.getType() != Token.EOF);
  // System.out.println("Parsed succeeded");

  return new ProgramNode(cmds);
}  // end of parse()
```



```
private CommandNode command()
// COMMAND  =>  LET | DRAW | CYCLE | PRINT
{
  if (token.isString("let"))
    return let();
  else if (token.isString("draw"))
    return draw();
  else if (token.isString("cycle"))
    return cycle();
  else if (token.isString("print"))
    return print();
  else {
    error(" in command syntax");
    return null;
  }
}  // end of command()


private LetCmdNode let()
// LET  =>  let ID '=' ( EXPR | SHAPE )
{
  match("let");
  Token t = match(Token.NAME);
  match("=");

  ShapeNode shape = null;
  ExprNode expr = null;
  if (isShape(token))
    shape = shape();
  else
    expr = expr();
  return new LetCmdNode(t.getText(), expr, shape);
}  // end of let()
```

The first versions of parse(), command(), and let() had a void return type; these new versions return ProgramNode, CommandNode, and LetCmdNode respectively.

command() may return a LetCmdNode, DrawCmdNode, CycleCmdNode, or PrintCmdNode object, which is cast to CommandNode, allowing parse() to store it in a polymorphic ArrayList.

The let() method builds a LetCmdNode object which has fields for both an expression and a shape. Since only one of these will have a value, the other field must be set to null.

9. The Evaluator

The evaluator travels over the syntax analyzer's parse tree generating the program's output (a SVG file). Since the evaluator visits every node and examines every field, the methods used by the evaluator will be closely related to the node classes. In general, each node class will require its own 'eval' method.

For example, the top-level node class, ProgramNode is:

```
class ProgramNode {
```



```
// PROGRAM => COMMAND+
   ArrayList<CommandNode> cmds;
   ProgramNode(ArrayList<CommandNode> cs) {cmds = cs;}
}
```

As a result, the Evaluator's top-level public method, genImage(), must iterate over the ProgramNode's ArrayList:

```
// global
private SVGGraphics2D drawG2D;

public SVGGraphics2D genImage(ProgramNode pNode)
{
  for(CommandNode cmd: pNode.cmds)
    evalCommand(cmd);
  return drawG2D;
}  // end of genImage()
```

A CommandNode can be a LetCmdNode, DrawCmdNode, CycleCmdNode, or PrintCmdNode object, which dictates a branching test to decide how to process the node:

```
private void evalCommand(CommandNode cmd)
{
  if (cmd instanceof LetCmdNode)
    evalLet((LetCmdNode)cmd);
  else if (cmd instanceof DrawCmdNode)
    evalDraw((DrawCmdNode)cmd);
  else if (cmd instanceof CycleCmdNode)
    evalCycle((CycleCmdNode)cmd);
  else if (cmd instanceof PrintCmdNode)
    evalPrint((PrintCmdNode)cmd);
}  // end of evalCommand()
```

The four node classes are processed by four different 'eval' methods.

9.1. Dealing with Assignment

The execution of a "let" command requires a *symbol table* to store the program's variables and their current values. The tricky aspect of assignment is that there are different types of data in Crop: doubles, points, lines, circles, and lists of circles. The simplest way of dealing with this variety is to store all of them as Object instances in the symbol table.

The LetCmdNode class has three fields:

```
class LetCmdNode extends CommandNode {
// LET  =>  let ID '=' ( EXPR | SHAPE )
   String id; ExprNode expr; ShapeNode shape;

   LetCmdNode(String i, ExprNode e, ShapeNode s)
   {id=i; expr=e; shape=s;}
}
```


Only one of the expr and shape fields will have a value, while the other will be null.

```
// global
private HashMap<String,Object> dict;   // symbol table for IDs


private void evalLet(LetCmdNode cmd)
{
  Object obj = null;   // id may be a shape or an expression
  if (cmd.shape != null)
    obj = evalShape(cmd.shape);
  else if (cmd.expr != null)
    obj = evalExpr(cmd.expr);
  else
    System.out.println("No value for let");

  dict.put(cmd.id, obj);
}  // end of evalLet()
```

evalShape() and evalExpr() deal with the evaluation of the Shape and Expr nodes, returning an Object instance, which is used to update the dict HashMap.

9.2. Evaluating a Shape

Shape evaluation needs to disambiguate the three ShapeNode subclasses (LineNode, CircleNode, and CCirclesNode).

```
private Object evalShape(ShapeNode s)
{
  if (s instanceof LineNode) {
    // LineNode data: ExprNode point1; ExprNode point2;
    LineNode line = (LineNode)s;
    return new Line2D.Double( evalPoint(line.point1),
                              evalPoint(line.point2) );
  }
  else if (s instanceof CircleNode)
    return evalCircleNode( (CircleNode)s );
  else if (s instanceof CirclesNode)
    return evalCircles( (CirclesNode)s );
  else
    System.out.println("Unknown shape type");

  return null;
}  // end of evalShape()
```

The reason for having three shape types follows from the grammar definition for Shape:

```
SHAPE =>  LINE  |  CIRCLE  |  CIRCLES
```

The three node types must be converted into graphical forms understood by Java 2D. LineNode is mapped to Java's Line2D.Double type, and its component points to Point2D.Double. CircleNode is converted to my own circle type, CCircle, and CirclesNode becomes a list of CCircle objects.

```
private CCircle evalCircleNode(CircleNode circle)
```



```
/* CircleNode data:
     ExprNode point; ExprNode radius;   */
{
  Point2D.Double pt = evalPoint(circle.point);
  double radius = evalDouble(circle.radius);
  return new CCircle(pt, radius);
}  // end of evalCircleNode()


private ArrayList<CCircle> evalCircles(CirclesNode circles)
/* CirclesNode data:
     ExprNode point; ArrayList<ExprNode> radii;  */
{
  ArrayList<CCircle> crcs = new ArrayList<CCircle>();

  Point2D.Double pt = evalPoint(circles.point);
  for(ExprNode radius : circles.radii)
    crcs.add( new CCircle(pt, evalDouble(radius) ) );
  return crcs;
}  // end of evalCircles()
```

CCircle encapsulates several circle capabilities. Figure 55 shows its class diagram, listing all the public and private data and methods.

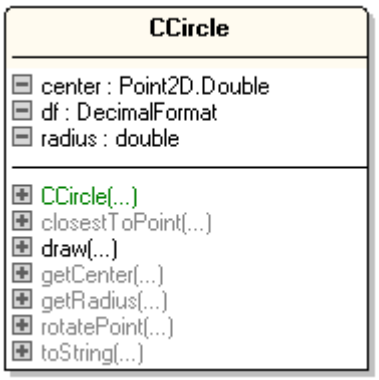


Figure 55. The CCircle Class Diagram.

CCircle stores the circle's center and radius, get methods, toString(), and draw() for rendering the circle. The closestToPoint() and rotatePoint() functions support circle geometry processing: closestToPoint() finds the closest circumference point to a supplied point argument, while rotatePoint() rotates a circumference point counterclockwise around the circle's center through a specified angle.

CCircle's draw() uses Java's Arc2D.Double to render a circle:

```
// CCircle globals
private Point2D.Double center;
private double radius;


public void draw(Graphics2D g2d)
{
  Arc2D.Double circle = new Arc2D.Double();
```



```
  circle.setArcByCenter(center.x, center.y, radius,
                0, 360, Arc2D.OPEN);   // a 360-degree arc
  g2d.draw(circle);
}  // end of draw()
```

The g2d graphics context passed to draw() writes to a SVG file, using the Batik library which I'll explain below.

9.3. Evaluating a Point

Converting a PointNode into a Point2D.Double is complicated by the four ways a point can be specified in the Crop language. This is reflected by the grammar rule for Point:

```
  POINT  =>  point '(' EXPR(x) ',' EXPR(y) ')' |  origin |
             INTERSECT  |  TURN
```

The PointNode class is abstract, with four subclasses: APointNode, OriginNode, IntersectNode, and TurnNode. evalPointNode() must deal with these four possible types:

```
private Point2D.Double evalPointNode(PointNode point)
{
  Point2D.Double pt = null;
  if (point instanceof APointNode) {
    // APointNode data: ExprNode x; ExprNode y;
    APointNode apt = (APointNode)point;
    pt = new Point2D.Double(evalDouble(apt.x), evalDouble(apt.y));
  }
  else if (point instanceof OriginNode)  // no data in OriginNode
    pt = new Point2D.Double(0,0);
  else if (point instanceof IntersectNode)
    pt = evalIntersect((IntersectNode)point);
  else if (point instanceof TurnNode)
    pt = evalTurn((TurnNode)point);
  return pt;
}  // end of evalPointNode()
```

9.4. Evaluating an Expression

Expression evaluation is a bit arduous because of the presence of several types of data: doubles, points, lines, and circles. The problem is to decide which operators (+, -, *, and /) are permissible with which types. The grammar rules aren't that helpful in this case since they don't include that kind of data type information:

```
EXPR  =>  TERM ( ('*'|'/') TERM )*
TERM  =>  FACTOR ( ('+'|'-') FACTOR )*
```

All the operators are useable with doubles, but points can only be added and subtracted. A double can be multiplied to a point, and divided into one. No operations can be performed on lines and circles.

These restrictions could be enforced in the syntax analyzer, when the parse tree is being built. One advantage of that would be the early reporting of typing problems. However, I've decided to do the checking as an expression is evaluated since it means the syntax analyzer is a little less complicated.

The node version of Expr is actually two classes:

```
class ExprNode {
// EXPR  ::=  TERM ( ('*'|'/') TERM )*
   TermNode term; ArrayList<ExprRestNode> terms;
   ExprNode(TermNode t, ArrayList<ExprRestNode> ts)
   {term=t; terms=ts;}
}


class ExprRestNode {
// ( ('*'|'/') TERM )
   String op; TermNode term;
   ExprRestNode(String o, TermNode t) {op=o; term=t;}
}
```

The (('*'|'/') TERM)* element is easiest to represent as an ArrayList of a new type, ExprRestNode. The ArrayList models the * operator, and ('*'|'/') TERM becomes two data fields inside ExprRestNode.

evalExpr() evaluates its component terms and applies the operators (* or /) between them with applyExpr() calls:

```
private Object evalExpr(ExprNode expr)
{
  Object o1 = evalTerm(expr.term);
  if (expr.terms.size() > 0)   // * may mean 0
    for(ExprRestNode er : expr.terms) {
      Object o2 = evalTerm(er.term);
      o1 = applyExpr(o1, er.op, o2);
    }
  return o1;
  }  // end of evalExpr()
```

applyExpr() enforces for the restrictions on '*' and '/' usage, by checking the arguments types:

```
private Object applyExpr(Object o1, String op, Object o2)
// apply * and / but only if the operand types allow it
{
  if ((o1 instanceof Double) && (o2 instanceof Double)) {
    double v1 = ((Double)o1).doubleValue();
    double v2 = ((Double)o2).doubleValue();
    if (op.equals("*"))
      o1 = new Double(v1*v2);
    else  // must be "/"
      o1 = new Double(v1/v2);
  }
  else if ((o1 instanceof Double) &&
           (o2 instanceof Point2D.Double)) {
    double v = ((Double)o1).doubleValue();
```



```
    Point2D.Double p = (Point2D.Double)o2;
    if (op.equals("*"))
      o1 = new Point2D.Double(v*p.x, v*p.y);
    else  // must be "/"
      o1 = new Point2D.Double(v/p.x, v/p.y);
  }
  else if ((o1 instanceof Point2D.Double) &&
           (o2 instanceof Double)) {
    Point2D.Double p = (Point2D.Double)o1;
    double v = ((Double)o2).doubleValue();
    if (op.equals("*"))
      o1 = new Point2D.Double(p.x*v, p.y*v);
    else  // must be "/"
      o1 = new Point2D.Double(p.x/v, p.y/v);
  }
  else
    System.out.println("Operand type mismatch for " + op);

  return o1;
}  // end of applyExpr()
```

The node classes for Term are similarly constructed to ExprNode, by making use of a second node class to deal with the * operator:

```
class TermNode {
// TERM  ::=  FACTOR ( ('+'|'-') FACTOR )*
   FactorNode factor; ArrayList<TermRestNode> facts;
   TermNode(FactorNode f, ArrayList<TermRestNode> fs)
   {factor=f; facts=fs;}
}

class TermRestNode {
// ('+'|'-') FACTOR
   String op; FactorNode factor;
   TermRestNode(String o, FactorNode f) {op=o; factor=f;}
}
```

evalTerm() is similarly constructed to evalExpr() except that it processes a list of factors using applyTerm() to check the type restrictions on '+' and '-'.

```
private Object evalTerm(TermNode term)
{
  Object o1 = evalFactor(term.factor);
  if (term.facts.size() > 0)    // * may mean 0
    for(TermRestNode tr : term.facts) {
      Object o2 = evalFactor(tr.factor);
      o1 = applyTerm(o1, tr.op, o2);
    }
  return o1;
}  // end of evalTerm()
```

Factor evaluation is complicated by the range of possible operands allowed by the grammar, and especially the presence of the "vertex" and "loopCounter" variables are used by Crop cycles (which I'll defer explaining until the next section).

```
FACTOR  => NUMBER | POINT | VERTEX | LOOPCOUNTER | pi |
```



```
          ID | '(' EXPR ')'
```

The FactorNode type contains fields for all of these grammar possibilities:

```
class FactorNode
{
  boolean isNumber; double number; PointNode point;
  VertexNode vertex; LoopCounterNode loopCounter;
  String id; ExprNode expr;

  FactorNode(boolean isN, double n, PointNode pt, VertexNode v,
                      LoopCounterNode lc, String i, ExprNode e)
  {
    isNumber = isN; number = n; point = pt; vertex = v;
    loopCounter = lc; id = i; expr = e;
  }
}  // end of FactorNode class
```

evalFactor() must examine all the fields to determine which one contains data.

```
private Object evalFactor(FactorNode f)
{
  Object result = null;
  if (f.isNumber)
    result = new Double(f.number);
  else if (f.point != null)
    result = evalPointNode(f.point);
  else if (f.vertex != null) {
    int vertIdx = getVertexIndex(f.vertex.number, f.vertex.id);
    result = getVertex(vertIdx, f.vertex.ups);
  }
  else if (f.loopCounter != null)
    result = getLoopCounter(f.loopCounter.ups);
  else if (f.id != null)
    result = lookupID(f.id);
  else if (f.expr != null)
    result = evalExpr(f.expr);

  return result;
}  // end of evalFactor()
```

If the ID field is being utilized, then the symbol table must be accessed to retrieve the ID's associated value.

```
// global
private HashMap<String,Object> dict;   // symbol table for IDs


private Object lookupID(String id)
// access an ID's Object
{
  Object obj = null;
  obj = dict.get(id);
  if (obj == null) {
    System.out.println(id + " not defined");
    System.exit(1);
  }
```



```
  return obj;
}  // end of lookupID()
```

9.5. Executing Cycles

The most novel part of the Crop language is the cycle mechanism (which is due to Andrew Glassner, not me, I hasten to add). The "cycle" command is supplied with a circle, and the number of sides of a regular polygon conceptually inscribed within the circle. Commands performed inside the cycle can refer to the polygon's vertices using vertex_0, vertex_1, and so on. On each iteration of the cycle, these vertex labels are moved counterclockwise around the polygon, and the loopCounter variable is incremented. A cycle example:

```
let c1 = circle(origin 10)
draw red c1
cycle c1 6 {
   draw blue circle(vertex_0 5)
}
```

"^" is a notational extension to the vertex_ (and loopCounter) labels which allows a nested cycle to refer to vertices (and the loop counter) in the enclosing cycle.

The “cycle” command's implementation in Evaluator utilizes a RegPolygon class, which stores the coordinates of the inscribed polygon, and the ongoing loopCounter value.

The RegPolygon constructor is passed circle details (as a CCircle object), and the number of polygon sides, and calculates the polygon's coordinates.

```
// globals in RegPolygon class
private Point2D.Double center;  // for circle being iterated over
private double radius;

private double rotAngle;   // rotation of first polygon coord
private int numSides;      // number of polygon sides

private Point2D.Double coords[];  // the polygon's coords
private int loopCounter = 0;


public RegPolygon(CCircle c, double a, int n)
{
  center = c.getCenter();
  radius = c.getRadius();
  rotAngle = a; numSides = n;

  if (radius <= 0) {
    System.out.println("Radius must be > 0");
    radius = 2;
  }
  if (numSides < 2) {
    System.out.println("No. of sides must be >= 2");
    numSides = 2;
  }
```



```
  coords = new Point2D.Double[numSides];
  double cornerAngle = 2*Math.PI/numSides;   // angle of each corner

  // fill in the polygon's coords
  double currAngle = rotAngle;
  for (int i=0; i < numSides; i++) {
    coords[i] =
       new Point2D.Double( center.x + radius*Math.cos(currAngle),
                           center.y + radius*Math.sin(currAngle) );
    currAngle += cornerAngle;
  }
} // end of RegPolygon()
```

The first polygon vertex (vertex_0) is located on the x-axis by default, but this can be modified by supplying a starting rotation angle (rotAngle).

A vertex is accessed using the getVertex() method along with an index (e.g. 0, 1, 2). This index is combined with the current loopCounter value to access the correct polygon corner for the current cycle iteration.

```
public Point2D.Double getVertex(int idx)
{
  if ((idx < 0) || (idx >= numSides)) {
    System.out.println("Vertex index out of range");
    idx = 0;
  }
  int coordIdx = (idx + loopCounter) % numSides;
  return coords[coordIdx];
}  // end of getVertex()
```

A RegPolygon object is sufficient for representing a single cycle, but cycles can be nested. This means that Evaluator must maintain a *list* of RegPolygons which grows and shrinks as cycles are created inside each other, and finish.

```
// globals in Evaluator
private ArrayList<RegPolygon> cycles;   // for storing nested cycles
private int numCycles = 0;

// in Evaluator()
cycles = new ArrayList<RegPolygon>();
```

A new RegPolygon object is created and added to the cycles list when a cycle starts being evaluated by evalCycle():

```
private void evalCycle(CycleCmdNode ccmd)
{
  RegPolygon regPoly = buildPoly(ccmd);  // create RegPolygon
  cycles.add(regPoly);    // store in cycles list
  numCycles++;

  int numSides = regPoly.getNumSides();
  for (int i=0; i < numSides; i++) {  // iterate through poly sides
    for(CommandNode cmd : ccmd.cmds)
      // execute all cycle cmds for each side
      evalCommand(cmd);
    regPoly.incrLoopCounter();
```



```
  }

  numCycles--;   // finished with this cycle
  cycles.remove(numCycles);    // remove RegPolygon for this cycle
}  // end of evalCycle()
```

When the cycle finishes at the end of evalCycle(), its RegPolygon object is removed from the cycles list.

buildPoly() creates a RegPolygon object by utilizing the cycle details stored in the CycleCmdNode object.

The Cycle grammar rule is:

```
CYCLE  ::=  cycle  (CIRCLE|ID) EXPR(sides)  (EXPR(angle)|'%')
                  '{' COMMAND+ '}'
```

The rule body includes five nonterminals (a circle, ID, two expressions, and a sequence of commands) and a '%' terminal, which are mirrored by six fields inside CycleCmdNode:

```
class CycleCmdNode extends CommandNode
{
  CircleNode circle; String id;
  ExprNode sides; ExprNode angle;
  boolean halfRot; ArrayList<CommandNode> cmds;

  CycleCmdNode(CircleNode c, String i, ExprNode s, ExprNode a,
                         boolean hr, ArrayList<CommandNode> cs)
  {circle=c; id=i; sides=s; angle=a; halfRot=hr; cmds=cs;}
}
```

The presence or absence of '%' in a “cycle” command is represented by setting the halfRot boolean in CycleCmdNode.

buildPoly() examines these CycleCmdNode fields in order to initialize its RegPolygon object. A RegPolygon object must be passed details about the circle, its starting rotation angle, and the number of polygon sides.

```
private RegPolygon buildPoly(CycleCmdNode cmd)
{
  CCircle ccircle = getCircleVal(cmd.circle, cmd.id);

  Double d = (Double) evalExpr(cmd.sides);
  int numSides = (int)(d.doubleValue());

  double angle = 0;     // angle is in radians
  if (cmd.halfRot)   // using "%"
    angle = Math.PI/numSides;
  else if (cmd.angle != null)
    // will be null if there's no angle in the command
    angle = Math.toRadians(
               ((Double)evalExpr(cmd.angle)).doubleValue() );

  return new RegPolygon(ccircle, angle, numSides);
```



```
}  // end of buildPoly()
```

Using a RegPolygon to Access a Cycle

A cycle is accessed using the "vertex_" and loopCounter labels. The "vertex_" label is followed by an index (e.g. 0, 1, 2), and both labels may be followed by '^'s to denote the enclosing (or higher) cycle. For example, the first vertex of a cycle is retrieved in the following code:

```
let c1 = circle(origin 10)
draw red c1
cycle c1 6 {
   draw blue circle(vertex_0 5)
}
```

Grammatically, "vertex_" and loopCounter occur inside Factor, as defined by the rule:

```
FACTOR  => NUMBER | POINT | VERTEX | LOOPCOUNTER | pi |
           ID | '(' EXPR ')'
```

Vertex and LoopCounter are defined as:

```
VERTEX  =>  vertex '_' (NUMBER|ID) '^'*
LOOPCOUNTER  =>  loopCounter '^'*
```

Note that any number of '^' symbols can follow the labels.

These rules are mapped to the node classes:

```
class VertexNode {
  double number; String id; int ups;
  VertexNode(double n, String s, int u)
  {number=n; id=s; ups=u;}
}


class LoopCounterNode {
  int ups;
  LoopCounterNode(int u) {ups=u;}
}
```

The ups fields in VertexNode and LoopCounterNode are the number of '^' symbols (which may be 0).

VertexNode and LoopCounterNode are processed in evalFactor():

```
private Object evalFactor(FactorNode f)
{
  Object result = null;
     // types of Factor ...
     //   :
  else if (f.vertex != null) {  // VertexNode
```



```
    int vertIdx = getVertexIndex(f.vertex.number, f.vertex.id);
    result = getVertex(vertIdx, f.vertex.ups);
  }
  else if (f.loopCounter != null) // LoopCounterNode
    result = getLoopCounter(f.loopCounter.ups);
  else if
    // ... other types of Factor

  return result;
}  // end of evalFactor()
```

getVertexIndex() retrieves the vertex index, which may either come directly from a number field, or need to be accessed via an ID in the symbol table.

```
private int getVertexIndex(double number, String id)
{
  if (id == null)
    return (int)number;
  else  { // use ID instead
    Object result = lookupID(id);
    if (!(result instanceof Double)) {
      System.out.println("vertex id " + id +
                    " is not a number; using 0");
      return 0;
    }
    else
      return (int)((Double)result).doubleValue();
  }
}  // end of getVertexIndex()
```

Back in evalFactor(), the vertex value is accessed with getVertex(), which is passed the index and the number of ancestor levels. For example, getVertex(1, 0) means vertex_1 in the current cycle, while getVertex(2, 1) denotes vertex_2 in the enclosing cycle.

```
// globals
private ArrayList<RegPolygon> cycles;
private int numCycles = 0;


private Point2D.Double getVertex(int vertIdx, int ancestor)
// access a specified vertex for this cycle, or an ancestor cycle
{
  int cycleRef = numCycles - 1 - ancestor;
  if (cycleRef < 0) {
    System.out.println("No cycle ancestor found");
    return new Point2D.Double(0, 0);
  }
  RegPolygon poly = cycles.get(cycleRef);
  return poly.getVertex(vertIdx);
}  // end of getVertex()
```


9.6. Drawing

The evaluation of a Crop program comes down to two graphics operations: drawing colored lines and circles. My first evaluator drew into a BufferedImage object and saved it to a PNG file, which looked less than impressive due to pixelization. Fortunately, there's an easy fix because Crop pictures only involve geometric shapes. In that case, vector graphics can be employed.

I utilize the Batik library (http://xmlgraphics.apache.org/batik/), which can generate, manipulate and transcode images in the Scalable Vector Graphics (SVG) format. It's possible for my Java code to keep on using Java 2D drawing operations, but with a specialized SVGGraphics2D graphics object which generates SVG content instead of drawing to a BufferedImage. The approach is explained in detail at http://xmlgraphics.apache.org/batik/using/svg-generator.html

Inside the revised version of Evaluator, a global SVGGraphics2D object is created by initDrawing(), which is directed to build a DOM tree representing the SVG contents:

```
// globals
private static final int WIDTH = 400;    // image dimensions
private static final int HEIGHT = 400;

private SVGGraphics2D drawG2D;


private void initDrawing()
{
  // Get a DOMImplementation
  DOMImplementation domImpl =
        GenericDOMImplementation.getDOMImplementation();

  // Create an instance of org.w3c.dom.Document
  String svgNS = "http://www.w3.org/2000/svg";
  Document document = domImpl.createDocument(svgNS, "svg", null);

  // Create an instance of the SVG Generator
  drawG2D = new SVGGraphics2D(document);

  // request highest-quality rendering
  RenderingHints hints =
      new RenderingHints(RenderingHints.KEY_ANTIALIASING,
                         RenderingHints.VALUE_ANTIALIAS_ON);
  hints.put(RenderingHints.KEY_RENDERING,
            RenderingHints.VALUE_RENDER_QUALITY);
  drawG2D.setRenderingHints(hints);

  // white background, black text
  Rectangle2D.Double bgRect =
       new Rectangle2D.Double(0, 0, WIDTH, HEIGHT);
  drawG2D.setPaint(Color.WHITE);
  drawG2D.fill(bgRect);
  drawG2D.setPaint(Color.BLACK);

  // move the origin to the center of the image
  drawG2D.translate(WIDTH/2, HEIGHT/2);
  }  // end of initDrawing()
```


The background of the SVG image is set to be a white WIDTH by HEIGHT rectangle, and drawing starts at its center.

The RenderingHints object has no effect on the SVG output since it doesn't use anti-aliasing, but I left in the code in case the output format is changed to raster graphics in the future.

Drawing is carried out by Crop's "draw" command, whose grammar rule is:

```
DRAW => draw [COLOR] ( SHAPE | ID )
```

The corresponding DrawCmdNode class is:

```
class DrawCmdNode extends CommandNode
{
  String color; ShapeNode shape; String id;

  DrawCmdNode(String col, ShapeNode s,String i)
  {color=col; shape=s; id=i;}
}
```

A DrawCmdNode node is processed by evalDraw():

```
private void evalDraw(DrawCmdNode cmd)
{
  Object obj;
  if (cmd.shape != null)
    obj = evalShape(cmd.shape);  // directly access a shape
  else   // must be an ID
    obj = lookupID(cmd.id);      // access shape via an ID
  doDraw(obj, getColor(cmd.color) );
}  // end of evalDraw()
```

A shape (e.g. a line, circle, or circles list) can be drawn directly, or retrieved via an ID and then drawn.

doDraw() carries out a different drawing task depending on which type of shape has been passed to it.

```
private void doDraw(Object obj, Color color)
// an object can be a line, circle, or list of circles
{
  drawG2D.setPaint(color);

  if (obj instanceof Line2D.Double)    // a line
    drawG2D.draw((Line2D.Double)obj);
  else if (obj instanceof CCircle)     // a single circle
    doDrawCircle((CCircle)obj);
  else if(obj instanceof ArrayList<?>) {   // an ArrayList
    @SuppressWarnings("unchecked")
    ArrayList<Object> objList = (ArrayList<Object>) obj;
    Object firstObj = objList.get(0);    // test first elem
    if (firstObj instanceof CCircle) {   // is a circle list
      for(Object objCircle : objList)
        doDrawCircle((CCircle)objCircle);
    }
    else
      System.out.println("draw shape is an unknown list type");
```



```
  }
  else
    System.out.println("draw shape is an unknown type");
}  // end of doDraw()
```

A line is drawn using the Graphics2D draw() operation, while a circle is rendered using the CCircle.draw() method:

```
private void doDrawCircle(CCircle circle)
{  circle.draw(drawG2D);  }
```

If doDraw() has to render a series of circles, then it's Object argument must be cast to an ArrayList of CCircles, which triggers a cast warning by Java at run-time. I switch off the warning message with the @SuppressWarnings annotation.

10. Bringing the Compiler Parts Together

Figure 4 shows the Lexer, Parser, and Evaluator components of the compiler. They are connected together inside the CropCircles class:

```
// global in CropCircles class
private static final String EXAMPS_DIR = "examples/";


public static void main(String args[])
{
  if (args.length != 1) {
    System.out.println("Supply a filename in " + EXAMPS_DIR);
    System.exit(0);
  }

  Parser p = new Parser(EXAMPS_DIR + args[0]);
  ProgramNode pNode = p.parse();

  Evaluator eval = new Evaluator();
  SVGGraphics2D svgGenerator = eval.genImage(pNode);

  String imFnm = outSVG(args[0]);
  saveImage(svgGenerator, EXAMPS_DIR + imFnm);
} // end of main()
```

saveImage() takes the SVGGraphics2D object, which points to the DOM tree of SVG drawing instructions, and writes the tree into a ".svg" file:

```
private static void saveImage(SVGGraphics2D svgGenerator, String fnm)
{
  try {
    OutputStream os = new FileOutputStream(new File(fnm));
    Writer out = new OutputStreamWriter(os, "UTF-8");
    svgGenerator.stream(out, true);  // use css
    os.flush();
    os.close();
    System.out.println("Saved crop circles to " + fnm);
```



```
  }
  catch (Exception e)
  {  System.out.println(e); }
}  // end of saveImage()
```

The resulting SVG file can be rendered by most browsers, or a SVG drawing application such as Inkscape (http://www.inkscape.org/), or opened as a text file.

11. Some Tips for Writing a Grammar that is LL(1)

It's obviously very important that the language's grammar is LL(1), otherwise it won't be possible to translate its rules into simple parsing code. Now that we've finished Crop, I can use it to illustrate a few design tips which should make it more likely that your grammar will be LL(1).

Start each unique *command* with a unique terminal:

e.g.
```
let x = 5
print y
```

Start each unique *data structure* with a unique terminal:

e.g.
```
x = point 2 3
x = line p1 p2
```

Expressions involving *, /, +, and - should be processed using some variant of the Crop Expr rules:

```
EXPR  =>  TERM  (  ('*'|'/')  TERM )*
TERM  =>  FACTOR  (  ('+'|'-')  FACTOR )*
FACTOR  => NUMBER  | ID  |  '(' EXPR ')'
```

If you need to add extra operators to expressions, then add them to Factor. Start each new operator with a unique terminal:

```
FACTOR  => NUMBER  | ID  |  '(' EXPR ')'  |  sqrt  EXPR
```

If your language needs to specify multiple elements, use the * or + operators, not recursive rules:

```
COMMANDS => CMD +                              // good design
COMMANDS => COMMANDS CMD   |   CMD            // bad design
```

If you must use recursion, make sure it occurs at the end of a rule body, even if that means adding extra rules:

```
COMMANDS => COMMANDS CMD   |   CMD            // bad design
```

```
COMMANDS  => CMD  MORE_CMDS                    // better design
MORE_CMDS => CMD  MORE_CMDS  |  ε
```

Multiple arguments of functions/operators may be easier to parse if they're surrounded by braces, with commas between them:

```
e.g.    foo x y z                 // may be hard to parse
        foo ( x, y, z )           // may be easier to parse
```

It may be useful to separate commands with extra syntax, such as a ';'

```
e.g.    x = 5   print x           // may be hard to parse
        x = 5 ;  print x ;        // may be easier to parse
```

The grammar rule would be:

COMMANDS => (CMD ';') +