Linguist Parsing System

Table of Contents

Chapter 1: Introduction

The Linguist Parsing System (LPS) supplies the C# programmer with many of the facilities of the popular Unix utilities Lex and Yacc. However, LPS has a quite different interface, will handle a superset of Yacc's grammars, and will return all valid parses of its input when the grammar is ambiguous.

Instead of embedding programming statements in the grammar specification, the programmer will construct and then invoke the LPS parser from her C# program. LPS will return all valid parses of the input as parse trees (one at a time, via an iterator). The application program then operates on each parse tree to do a translation, generate machine code, invoke an interpreter, or similar functions.

You can use LPS to parse program source files, data files, and natural language text. Any grammar that can be expressed in BNF, even ambiguous ones, can be given to LPS. These include (but are not limited to) LALR(k), LL(k), and LR(k).

There are five components to LPS. First is the `LexRuleBuilder` which takes a text file and makes the lexer's rules. The second is the `ParseRuleBuilder` which makes the rules (productions) for the parser. Third is the `Lexer` (a lexical analyzer), which takes an input file of text to be parsed, builds tokens, and passes these to the fourth component: the `Parser`. The first three of these components can be replaced by your own objects.

All four components interact in the same executable. There are no intermediate files produced and no C# programs are compiled by you (or LPS). However it is possible to save and restore some of the objects constructed by LPS. This allows more rapid parsing when the same grammar is to be used against multiple inputs, but is entirely optional.

The last component is a test facility. This allows text (or a file) to be input and the resulting parse tree to be displayed. Because the LPS system does not generate a C# program, testing can be rapid and easy. No recompilation is required when your grammar changes.

The software is free for you to download, copy, modify and use for any purpose. However, no warranty is given as to its accuracy or suitability. It is written entirely in C#, version 2, using Microsoft's Visual Studio 2005.

Your comments, suggestions, and especially bug reports are welcome. Contact fredm73@hotmail.com.

Chapter 2: Quick Start

You will need to download the C# projects that support LPS. In the Linguist folder is a project called TestLinguist. One of the executables therein is LinguistTest.exe which is the test facility mentioned in the introduction. You should execute that (by itself or from the debugger). You will see the following:

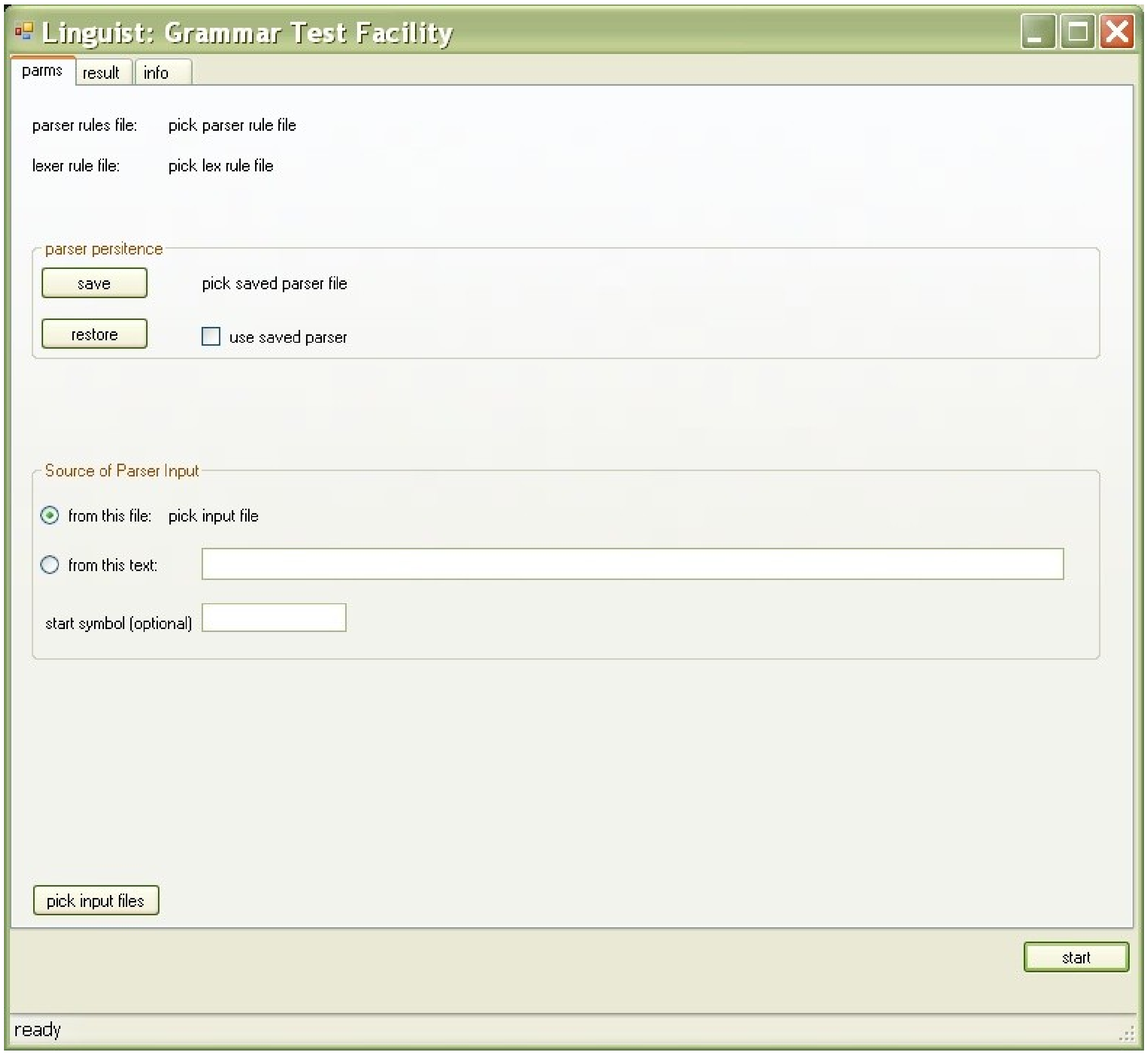
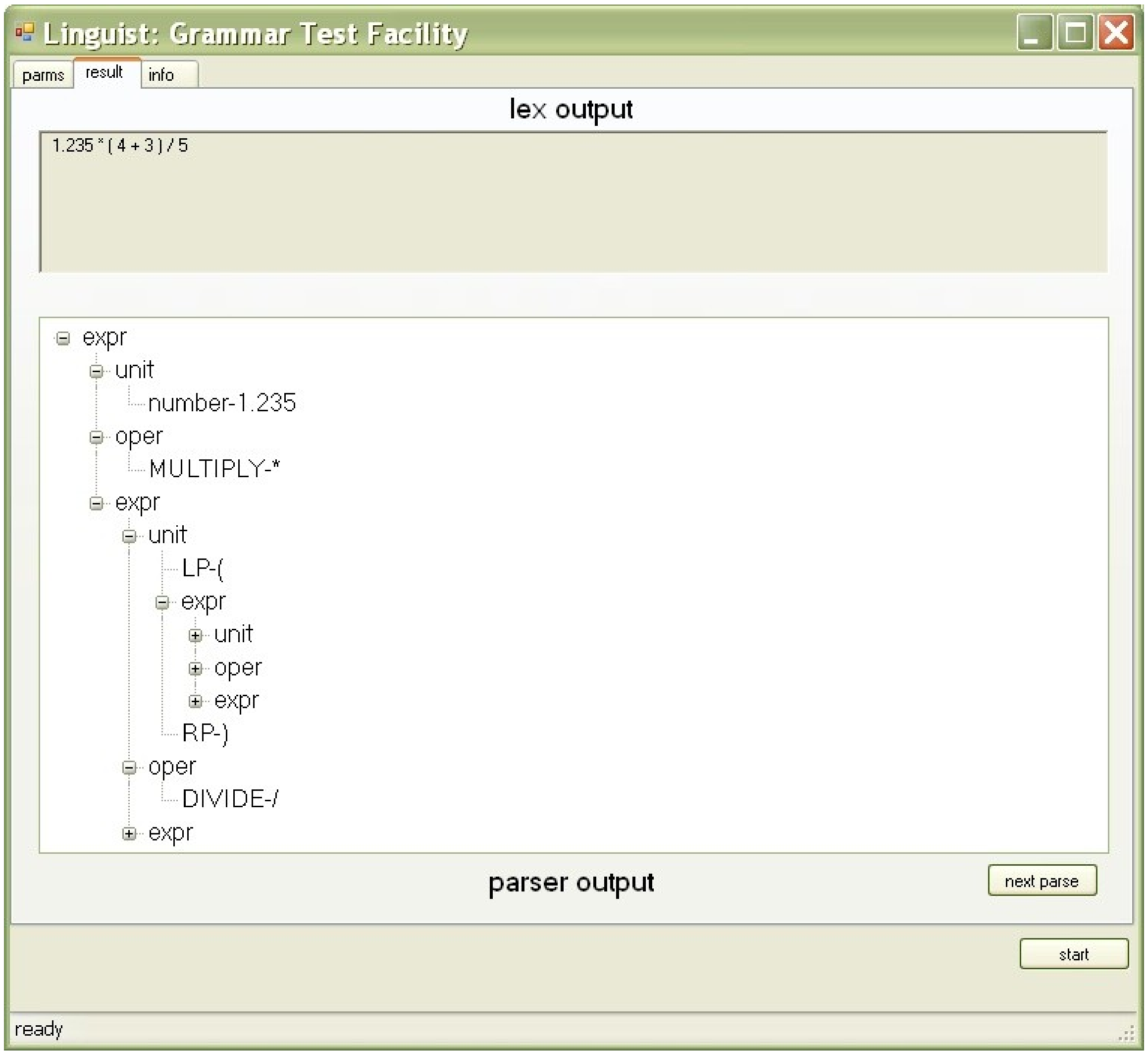


Figure 1

This program lets you test your grammar (the rules for the lexical analyzer – `Lexer`, as well as the grammar rules). If you wish to parse a string, click on the *from this text* button and fill in the text box. When you click on *start*, you will be asked for the `Lexer`'s rule file and the `Parser`'s rule file, and (optionally) a file for input to the parser.

If all goes well, you will see the results in the *result* tab and messages from the parser in the *info* tab, as below:



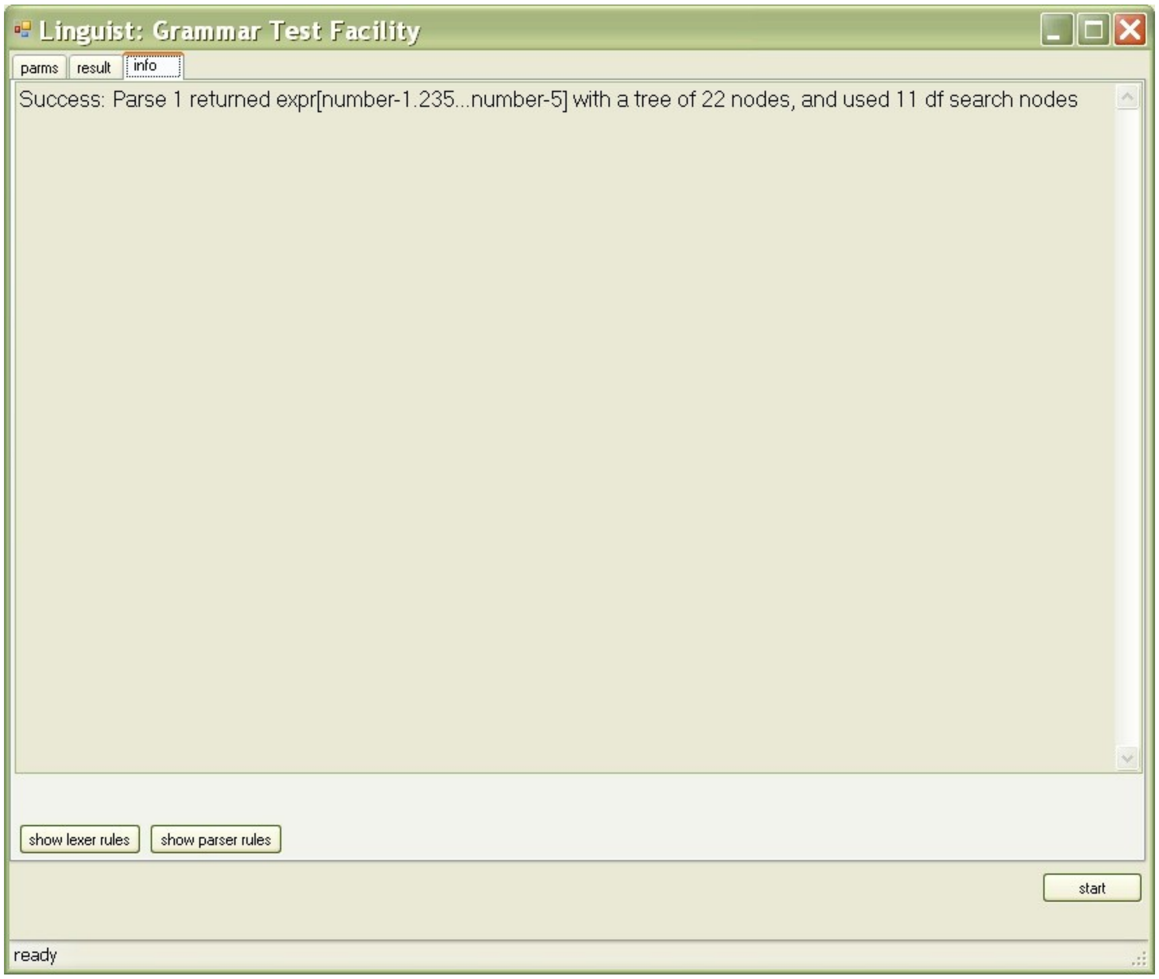
```
1.235 * ( 4 + 3 ) / 5
```

```
⊟ expr
    ⊟ unit
        number-1.235
    ⊟ oper
        MULTIPLY-*
    ⊟ expr
        ⊟ unit
            LP-(
            ⊟ expr
                ⊞ unit
                ⊞ oper
                ⊞ expr
            RP-)
        ⊟ oper
            DIVIDE-/
        ⊞ expr
```

Figure 2

What the `Lexer` saw and the parse tree constructed by the `Parser` are given in this window. If you click on *next parse* it will return other parses (if the grammar is ambiguous).

The *info* tab is shown in the next Figure.



Success: Parse 1 returned expr[number-1.235...number-5] with a tree of 22 nodes, and used 11 df search nodes

Figure 3

If the parser found errors they would be given here. Additionally, if more than one parse exists (and you asked for each with the *nextParse* button) it will explain where shift/reduce or reduce/reduce conflicts occurred.

If you examine the `Lexer`'s rule file (*lexRule.txt* for this application) you will see:

```
number = [\d]*[\.]?[\d]+
LP = \(
RP = \)

MULTIPLY = \*
SUBTRACT = \-
DIVIDE = \/
ADD = \+
```

(The line numbers are not actually in the file). The name of a `Lexer` token is given first, followed by a regular expression. These names are used in defining the `Parser`'s rules.

The `Parser`'s rule file (the grammar) is *parseRule.txt* and contains (without the line numbers):

```
expr: unit                    {1}
 |
 unit oper expr               {2}
 ;

unit: number                  {3}
 |
 LP expr RP                   {4}
 ;

oper:   MULTIPLY | ADD | SUBTRACT | DIVIDE;
```

If you are familiar with "yacc" this format (called *BNF*) should be familiar. However, for Linguist, the text inside the braces (**{ }**) is not program code, but either an uninterpreted name, or the name of a class defined in your application. These names are optional. If you trace the parse tree in Figure 1 and compare it to the input and the `Parser`/`Lexer` rules, you should be able to see what it going on.

The C# program above is available for testing your grammar. However, the actual application interface to the parser is entirely programmatic, via classes and method calls, from your application into the Linguist library.

Chapter 3: Lexer Rule Format

The `LexRuleBuilder` expects an input file to define the rules for the `Lexer`. For this discussion, we will call the file the *lexerRuleFile*. Here is a sample (the line numbers are not part of the file):

```
1   comment = \/\*  \*\/
2   ! = #.*$
3
4   NUMBER= [\d]*[\.]?[\d]+
5   LP = \(
6   RP = \)
7
8   MULTIPLY = \*
9   SUBTRACT = \-
10  DIVIDE = \/
11  ‘+’ = \+
```

In the discussion below, we will call the parser's input (i.e. text to be parsed) the *piFile*. The `LexRuleBuilder` reads its *lexerRuleFile* to gather the rules together and you pass them to the `Lexer`'s constructor. Then, when the parse begins, the `Lexer` reads the *piFile* to form tokens, which it passes to the `Parser`, one at a time.

The basic format of the *lexerRuleFile* is a *name*, followed by **=**, followed by a *regular expression*, as in lines 4-10. When the `Lexer` matches the regular expression while scanning the *piFile*, it forms a token with that name, and includes the actual input (the *text*) that the regular expression matched.

The regular expression is any string (it is not quoted) that is acceptable to the `RegEx` constructor in C#. Some characters have special meaning to `RegEx` and you must escape them if you do not wish that meaning (as in lines 19-22).

The names you give are also used in the `Parser` rules. This is how the `Parser` matches the tokens produced by the `Lexer` against its rules.

The name must have one of two forms. The first is alpha-numerics, plus the underscore (_) as in lines 4-10. The second form is a single character (not escaped), surrounded by single quotes, as in line 11. The second form can make the `Parser` rules a little clearer but is entirely optional.

The `Lexer` uses the rules *in order* to match the input in the *piFile*. Hence you must make sure that the most specific rules are ahead of the more general, so that the former can find a match (otherwise, the more general would find a match and the more specific would not even be tried).

The `Lexer` reads one line of the *piFile* at a time, and will not match text across lines in the *piFile*. This means you *cannot* define a `Lexer` rule that will match across the input lines (but you can use the end of line anchor ($) in regular expressions to match the end of the line). Furthermore, the `Lexer` removes leading white space before the next match, so that you cannot match on leading blanks or tabs either.

To get around the multiple line capture problem, we introduce another format for a `Lexer` rule, as shown in line 1. Here, you will provide a second regular expression. The two are used to match a beginning and ending delimiter. Our `Lexer` will capture all text, including and between the delimiters, and bundle it up into one token (here named *comment*). So in the rule at line 1, we are capturing multi-line comments, between the /* and */.

There is one more convention. You can tell the `Lexer` not to deliver input to the `Parser`. To do this, you use a !, instead of the alpha-numeric name. `Lexer` will match *piFile* text using the regular expression provided with the !. It will consume the text and continue to the next match before forming a token for the `Parser`. So, if we wanted to skip the comments defined in line 1, we could have used the ! directive instead of the name *comment*. An example is on line 2, where we are matching on a # and skipping the text until the end of line.

There are many websites that will provide you with "cookbook" regular expressions. Be aware that C# has its own conventions for use in `RegEx` and you must follow those.

Chapter 4: Parser Rule Input

A sample of the `Parser`'s rule file follows (the line numbers are not part of the file):

```
1   expr: unit                    {1}
2    |
3    unit oper expr               {rule2.2}
4    ;
5
6   unit: number                  {3}
7    |
8    LP expr RP                   {4}
9    ;
10  EMPTY ;
11
12  oper:   DIVIDE;
13  oper:   MULTIPLY | ADD | SUBTRACT;
```

The typical rule definition is displayed on line 12. The rule's "head" there is "oper". This is followed by a colon, then the tail of the rule. Here there is only one symbol in the tail, "DIVIDE".

Each rule head is called a non-terminal, and must not match the names of tokens delivered from the `Lexer`. These latter are called "terminals", and their names can occur in a rule's tail. For example, DIVIDE is a token defined in the `Lexer` (see the previous chapter) and so is a terminal. The tail of a rule is a combination of terminals and non-terminals, as in line 8.

A rule must end with a semi-colon.

When two rules have the same head, as in lines 6-8 and 13, you can specify the head just once, and separate the tails with a '|'. This is called an *alternation*. Use of this shorthand is entirely optional.

A rule can have no tail, as in line 10. However, these must be on a line by themselves (i.e. not in an alternation).

The other construct in a rule is the *rule name*, specified by enclosing it in braces. It is optional. The rule name takes two forms. The first is an alpha-numeric plus underscore. This name is carried forward to the `Production` object that is constructed from the rule definition. It becomes the `Production.name` field. Your application can key on this name when you process the parse tree (see the chapter on sample applications for details). Examples of this type of name are on lines 1, 6, and 8.

The second form of the rule name consists of two alpha-numeric-plus-underscore names, separated by a period, as in *XXX.YYY*. The *YYY* part has the same function as before: it just serves as the name of the rule. The first part, *XXX*, is the name of a class you define in your application. That class must be a subclass of `Production.` LPS will construct an instance of type *XXX* when it processes the rule. Your subclass can then override virtual functions in `Production` to make application logic more cogent. More details are given in a subsequent chapter.

The rules in the parser's rule file can be given in any order. However, if a "start symbol" is not supplied, LPS will use the head of the first rule in the grammar. The "start symbol" is defined as the head of some rule that the parser is to start with. It will find a parse tree with that symbol as its root.

Chapter 4: Sample Applications

INTRODUCTION

We will present a simple calculator application with three different solutions. By studying these you will learn the interface to LPS and techniques for solving your parser problems.

All three make use of the same lexer rules file (*calc1Lex.txt*):

```
1   ! = \/\*\*\/
2   ! = \/\/.*$
3
4   number = \d+(\.\d+)?(E([\-\+]?)\d+)?
5   LP = \(
6   RP = \)
7
8   MULTIPLY = \*
9   SUBTRACT = \-
10  DIVIDE = \/
11  ADD = \+
```

The first line says ignore any text between '/*' and '*/'. Line 2 says ignore, till the end of line, any text that starts with '//'.

Line 4 defines a `number` to be any text acceptable to `double.parse`. The remaining lines should be easy to figure out (read the chapter on the lexer rule input if you have doubts).

The grammar definition is found in file *calc1Parse.txt*:

```
expr: unit                          {1}
 |
 unit oper expr                     {2}
 ;

unit: number                        {3}
 |
 LP expr RP                         {4}
 ;

oper:    MULTIPLY | ADD | SUBTRACT | DIVIDE;
```

You should have no trouble interpreting this file (read the chapter on the parser rule format).

All the applications start by calling a `setup` method:

```
public List<string> setup(string parseRuleFile, string lexRuleFile,
            bool inputFromText, string inputToParse, string startSym)
        {
            try
            {
                LexRuleBuilder lexBuilder = new LexRuleBuilder(lexRuleFile);
                List<LexRule> lexRules = lexBuilder.getRules();
                lexer = new Lexer(inputToParse, lexRules, inputFromText);
9               lexBuilder.Dispose();
10
11              ParseRuleBuilder ruleBuilder = new
    ParseRuleBuilder(parseRuleFile);
12              List<Production> parseRules = ruleBuilder.getGrammar();
13              msgs.AddRange(ruleBuilder.msgs);
14
15              if (ruleBuilder.msgs.Count == 0)
16                  parser = new Linguist.Parser(parseRules, lexer,
    startSym);
17          }
18
19          catch (Exception ex)
20          {
21              msgs.Add(ex.Message);
22              appFailed = true;
23              rulesFailed = true;
24          }
25
26          return msgs;        //setup is OK
27      }
```

Except where it is obvious, the classes mentioned in the code are included in the Linguist library.

Lines 6-8 set up a `lexer`. Lines 11-16 build the parser. The parameter, `startSym` in line 16 is optional. If not supplied, the parser uses the head of the first rule in the grammar as the start symbol. You should catch exceptions and deal with message output from the "building" phase in some way (as shown in lines 19-26).

The logic in this routine has the goal of making the parser at line 16. If you wish to use the lexer and builders supplied by LPS, you should write similar logic. Optionally, you can find some other way to get the parser and lexer rules. You can even use your own lexer, although the restrictions on this object are somewhat subtle. You will have to study the `Lexer` code for more information.

After the parser is set up, all the applications call a `getTrees` method as follows:

```
if (!getTrees())
     return false;
```

That method calls the parser to obtain all the parse trees. For this grammar there will be but one parse, but it is good practice to fetch all the trees (unless you know your grammar is unambiguous, or only want the first parse). That method is:

```
bool getTrees()
        {
            int parseCount = 0;

            foreach (ParseTree pTree in parser.allParses())
            {
                parseCount++;

                parseTree.Add(pTree.clone());


                if (parseCount == 2)
                {
```

```
14                  msgs.Add("CA01: multiple parses exist");
15              }
16
17              msgs.AddRange(parser.parseInfo);
18          }
19
20          msgs.AddRange(parser.parseInfo);
21
22          if (parseCount == 0)
23          {
24              appFailed = true;
25              return false;
26          }
27
28          return true;
29      }
```

Line 5 shows how to use the iterator supplied by LPS. Inside the loop (lines 5-18), the parser will supply a parse tree. If there is no parse, messages are placed in `parser.parseInfo`, as are informative messages when there is a good parse. You can test either `parseCount`, or the Boolean `parser.noParse`, to see if you have good parses (line 22). Note in line 9 that the parse tree is cloned. This must be done across calls to the iterator since the parser adjusts the first parse tree in obtaining the second parse.

Note how messages are collected from the parser. After each good parse, the parser clears old messages from `parseInfo` and adds new ones pertaining to the latest parse. That is why the messages are collected at line 17 inside the loop. If no parses occurred the loop will not be executed even once. That is why messages must also be collected at line 20. Even when good parses occurred, the messages at line 20 will contain a summary of the total parses.

Once the parse tree list is obtained, each application processes the parse tree a bit differently. We discuss the three methods next.

<u>TOP DOWN</u>

This method is the simplest but the least general. It uses a top-down, recursive, technique for processing the tree. Here is the code:

```
foreach (ParseTree pTree in this.parseTree)
            {
                double theAnswer = evaluateTopDown(pTree);
                answers.Add(theAnswer.ToString());
            }
```

The method `evaluateTopDown` is:

```
private double evaluateTopDown(ParseTree pTree)
        {
            if (pTree.symbol.isTerminal)
                return Double.Parse(((Token)(pTree.symbol)).text);

            switch (pTree.prodn.name)
            {
```

```
8                   case "1":   //expr -> unit
9                       return evaluateTopDown(pTree.children[0]);
10
11                  case "2":   //expr -> unit oper expr
12
13                      ParseTree oper = pTree.children[1].children[0];
14                      //oper -> MULTIPLY | ADD | SUBTRACT | DIVIDE
15
16                      switch (((Token)oper.symbol).text)
17                      {
18                          case "*":
19                              return evaluateTopDown(pTree.children[0]) *
20                                  evaluateTopDown(pTree.children[2]);
21
22                          case "/":
23                              double denom =
    evaluateTopDown(pTree.children[2]);
24                              if (denom == 0.0)
25                                  throw new ApplicationException(
26                                      "0 divide");
27                              return evaluateTopDown(pTree.children[0]) /
28                                  denom;
29
30                          case "-":
31
32                              return evaluateTopDown(pTree.children[0]) -
33                                  evaluateTopDown(pTree.children[2]);
34
35                          case "+":
36                              return evaluateTopDown(pTree.children[0]) +
37                                  evaluateTopDown(pTree.children[2]);
38                          default:
39                              throw new ApplicationException(
40                                  "oper not understood: logic error");
41
42                      }
43
44                  case "3":       //unit -> number
45                      return evaluateTopDown(pTree.children[0]);
46
47                  case "4":       //unit -> LP expr RP
48                      return evaluateTopDown(pTree.children[1]);
49
50                  default:
51                      throw new ApplicationException(
52                                  "rule name not understood: logic error");
53              }
54          }
```

This method uses the node in the parse tree (each of which is also a parse tree) to obtain a `double`, and return it to the calling method (which is this method itself except for the first call).

Lines 3-4 expect that the only call to this method, when the parse tree represents a terminal, will be the string representation of a `double`. If you review `calc1Lex.txt`, you will see that this is the terminal defined at line 4. When a `Token` object is formed, the input that was matched is stored in the `text` field. When we look at the code that follows line 4 we will see why lines 3-4 cannot receive the other `Tokens` (e.g. LP, RP).

The rest of the code in this method is a switch statement, where the cases are determined by the name of the rule that lead to the parse tree. In the grammar file, *calc1Parse.txt* we attached the name to each rule we were interested in, between the '{' and '}'.

The definition of the `ParseTree` object is:

```
public class ParseTree
    {
        public ParseTree parent;    //for bottom up processing of the parse tree
        internal ItemSet state;    //used in the state table to get the next
                                   //state.
        public Decoration decoration = null;  //set by user application when
                                              //processing the pTree.

        public Production prodn;  //used to resolve this branch of the tree
                                   //null means this is a terminal node of the tree

        public ParseElement symbol;    //matches head of the prodn,
                                        //or is a token (i.e. a leaf )
        int lowLevelCode = -1;
        public List<ParseTree> children;   //matches tail of the prodn
```

The `prodn` is the parse rule from the rule file. The `symbol` is the head of that production, the `children` are the parse trees for each tail symbol. Hence you can match the tail of the production in the *calc1Parse.txt* file with the `children`, in the same order. So, at line 8, in method `evaluateTopDown`, the only 'child', at `children[0]`, corresponds to the `unit` in rule 1. We invoke this function, recursively, to evaluate that 'unit' parse, which will return a `double`, which we return from this routine to its caller.

Note that the function "bottoms out" at the `Tokens` that are the textual representation of the `doubles` (numbers) in the input stream.

At line 11 we are dealing with rule 2. Its tail has 3 symbols. The second one is the operation. The rule that deals with operations does not have a name (we did not include the '{}' in its definition). We get at the `token` that represents the operation in line 13 and operate on it via the switch statement at line 16. Note that at line 13 the child accessed is the second one (subscript is 1). The only child of that tree is the actual token we are interested in. The input is stored in `Token.text` (as was the number we accessed in line 4).

The other numbers we are interested in are in the first and third symbols in rule "2"s tail. To fetch them, we call `evaluateTopDown` on each, and combine them based on the operation.

You should be able to understand how we process rules "3" and "4" in the subsequent code.

<u>BOTTOM UP</u>

The top down function is fairly simple because we only had to pass a single number from each recursive call. More complicated situations can be handled "bottom up". Here we will "decorate" the nodes in the parse tree. The `Decoration` object we used is defined:

```
1   public class CalcValue : Decoration
2           {
3               public double number;
4
5               public CalcValue(double num, ParseTree tree): base(tree)
6               {
7                   number = num;
8               }
9
10
11              public CalcValue(double num1, double num2, Token oper,
    ParseTree tree):
12                  base(tree)
13              {
14                  switch (oper.name)
15                  {
16                      case "number":
17                          number = double.Parse(oper.text);
18                          break;
19                      case "MULTIPLY":
20                          number = num1 * num2;
21                          break;
22                      case "SUBTRACT":
23                          number = num1 - num2;
24                          break;
25                      case "DIVIDE":
26                          if (num2 != 0.0)
27                              number = num1 / num2;
28                          else throw new ApplicationException("0 divide");
29                          break;
30                      case "ADD":
31                          number = num1 + num2;
32                          break;
33                      default:
34                          throw new ApplicationException("token not
    understood " +
35                              oper.name);
36                  }
37              }
38          }
```

It must be a subclass of `Decoration`. This application stores but one item in it, at line 3. The two constructors will be called as we process the parse tree.

```
public bool bottomUp(bool inputFromText, string inputToParse)
        {
            if (!base.reset(inputFromText, inputToParse))
                return false;

            if (!getTrees())
                return false;

            foreach (ParseTree pTree in this.parseTree)
10              {
11                  double theAnswer = evaluateBottomUp(pTree);
12                  answers.Add(theAnswer.ToString());
13              }
14              return true;
15          }
```

This function is like `topDown`, except that it calls `evaluateBottomUp` for the parse tree (at line 11). This is displayed next.

```
1   private double evaluateBottomUp(ParseTree pTree)
2           {
3               foreach (List<ParseTree> aRow in pTree.bottomUp())
4               {
5                   foreach (ParseTree rowTree in aRow)
6                   {
7                       Token t = rowTree.symbol as Token;
8                       if (t != null)
9                       {
10                          if (t.name == "number")
11                              rowTree.decoration = new CalcValue(
12                                  double.Parse(t.text), rowTree);
13
14                          continue;
15                      }
16
17                      switch (rowTree.prodn.name)
18                      {
19                          case "1":       //expr -> unit
20                          case "3":       //unit -> number
21                              rowTree.decoration =
    rowTree.children[0].decoration;
22                              break;
23                          case "2":       //expr -> unit oper expr
24                              rowTree.decoration = new CalcValue(
25                                  ((CalcValue)
    (rowTree.children[0].decoration)).number,
26                                  ((CalcValue)
    (rowTree.children[2].decoration)).number,
27                                  (Token)
    (rowTree.children[1].children[0].symbol),
28                                  rowTree);
29                              break;
30                          case "4":       //unit -> LP expr RP
31                              rowTree.decoration =
    rowTree.children[1].decoration;
32                              break;
33                          case null:     //oper -> MULTIPLY | ADD | SUBTRACT
    | DIVIDE
34                              break;
35                          default:
36                              throw new ApplicationException(
37                                  "rule name not understood " +
38                                  rowTree.prodn.name);
39                      }
40                  }
41              }
42
43                  //we have added decorations all the way up to the root:
44              return ((CalcValue)(pTree.decoration)).number;
45          }
```

At line 3 we use the `bottomUp` iterator supplied by `ParseTree`. This returns the leaf nodes, then the trees whose children are all leaves, and so forth until we reach `aRow` that has but one element, representing the start symbol of the parse. So, you can be assured that all the children of a `parseTree` have been processed before that `parseTree` is given back by the iterator. At lines 7-15 we deal with all the `Tokens` that represent numbers. For each, we "decorate" the parse tree with a `CalcValue` that represents that number.

The rest of the code is similar: it builds a `CalcValue` for each `parseTree` we are interested in. Note that sometimes all we have to do is carry the previous decoration forward, as in rules "1", "3", "4". Rule "2" is the exception where we have to construct a new decoration by combining two numbers. We elected to have the constructor make the combination for us. At line 33 we have unnamed rules and we make no decoration for those.

In some ways, this method is simpler than `topDown` since it does not involve recursion. It is more flexible since we can have different decoration subclasses for different rules (not required in this application).

The next application builds on this, and is a little more "object oriented".

<u>SUBCLASS</u>

Here we alter the parser's rule file, *calc2Parse.txt*:

```
expr: unit                        {Rule1.1}
 |
 unit oper expr                   {Rule2.2}
 ;

unit: number                      {Rule3.3}
 |
 LP expr RP                       {Rule4.4}
 ;

oper:    MULTIPLY | ADD | SUBTRACT | DIVIDE;
```

The only difference is that we are using the second format for the rule name: xxx.yyy. Here, "yyy" is just carried into the `Production.name` field. The first part of the name, however, designates a user subclass of class `Production`. Thus **Rule1**, **Rule2**, **Rule3**, **Rule4** are all subclasses, defined by the application.

Class `Production` has some useful virtual functions you can override as you wish: `decorate`, `evaluate`, and `emit`. Each takes one parameter, a `ParseTree`. Our application method, `subclass`, is like the others except it calls `evaluateSubclass` at line 11.

```
1  internal bool subclass(bool inputFromText, string inputToParse)
2          {
3              if (!base.reset(inputFromText, inputToParse))
4                  return false;
5
6              if (!getTrees())
7                  return false;
8
9              foreach (ParseTree pTree in this.parseTree)
10             {
11                 double theAnswer = evaluateSubclass(pTree);
12                 answers.Add(theAnswer.ToString());
13             }
14             return true;
15         }
```

The designer has made four subclasses, one for each rule she is interested in, and each given the name that corresponds to the designation in the parser's grammar (rule) file, *calc2Parse.txt.* She has elected to override the decorate function to achieve the same effect as the previous method, `bottomUp`. By placing the rule specific code in a subclass representing that rule, this design is arguably clearer. Should each subclass require many "helper" methods, and should the `Decoration` subclasses differ for each rule this separation might be very helpful.

Here are the four classes defined (we get along with the single decoration class, `CalcValue`, as before).

```
[Serializable]
        public class Rule1: Production   //expr -> unit
        {
            public Rule1(Production prodn)
                : base(prodn)
            {
            }

            public override void decorate(ParseTree tree)
            {
                tree.decoration = tree.children[0].decoration;
            }
        }

        [Serializable]
        public class Rule2 : Production   //expr -> unit oper expr
        {
            public Rule2(Production prodn)
                : base(prodn)
            {
            }

            public override void decorate(ParseTree tree)
            {
                Token tok = (Token)(tree.children[1].children[0].symbol);
                tree.decoration = new CalcValue(
                    ((CalcValue)(tree.children[0].decoration)).number,
                    ((CalcValue)(tree.children[2].decoration)).number,
                    tok, tree);
            }
        }

        [Serializable]
        public class Rule3 : Production   //unit -> number
        {
            public Rule3(Production prodn)
                : base(prodn)
            {
            }

            public override void decorate(ParseTree tree)
            {
                Token tok = (Token)(tree.children[0].symbol);
                double num = double.Parse(tok.text);
                tree.decoration = new CalcValue(num, tree);
            }
        }

        [Serializable]
        public class Rule4 : Production   //unit -> LP expr RP
        {
            public Rule4(Production prodn)
                : base(prodn)
            {
            }

            public override void decorate(ParseTree tree)
            {
                tree.decoration = tree.children[1].decoration;
            }
        }
```

Each class has been marked `[Serializable]` so that they can be saved if/when we wish to save the parser (see the chapter on using the test facility). Each subclass must contain a constructor of the format shown. You pass the parameter back to the superclass in the constructor, as shown by the `base` parameter. When rules are constructed by the rule builder, LPS will invoke your constructor with the appropriate rule (in parameter `prodn`).

There is a restriction: all the subclasses must be defined in the same assembly that invokes the `ParseRuleBuilder.getGrammar` (or you can pass the name of the assembly that contains the subclasses as a parameter). Take a look at that method, and the `ParseRuleBuilder.factory` method to see how all this works.

If you study the decorate functions you will see that they build up a decoration as did the previous application we studied, `bottomUp`.

The method, `evaluateSubclass` is simpler than `evaluateBottomUp`:

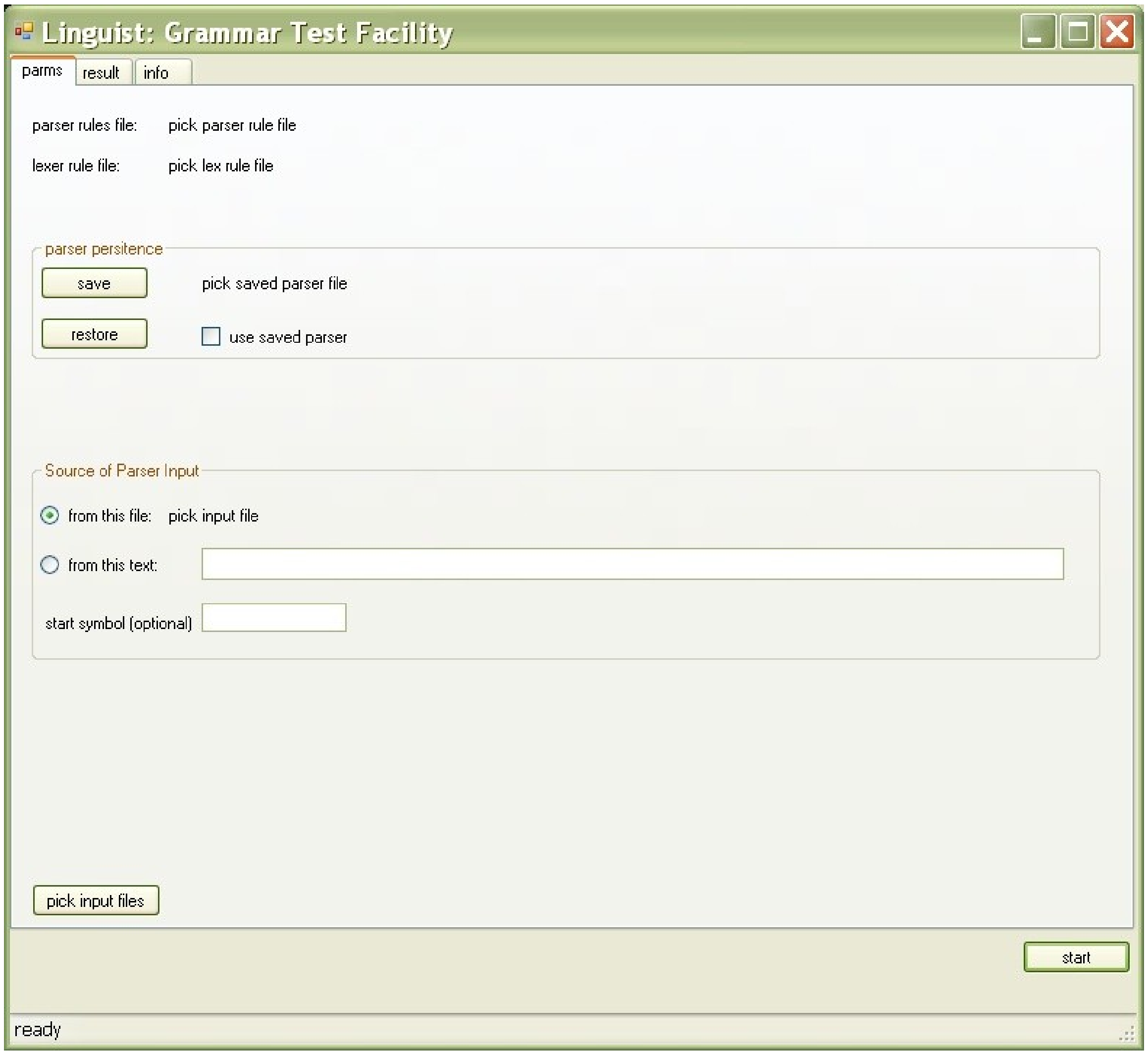
```
1   private double evaluateSubclass(ParseTree pTree)
2           {
3               foreach (List<ParseTree> aRow in pTree.bottomUp())
4               {
5                   foreach (ParseTree aTree in aRow)
6                   {
7                       if (aTree.symbol.name == "oper" ||
    aTree.symbol.isTerminal)
8                           continue;
9                       aTree.prodn.decorate(aTree);
10                  }
11              }
12              return ((CalcValue)(pTree.decoration)).number;
13          }
```

We use the same bottom up iterator in line 3. Lines 7-8 skip over nodes that represent productions (parse rules) we are not interested in. The appropriate virtual function is resolved in line 9.

We have segregated the logic by the class of rule that has been placed in the `parseTree`. Since these are user classes some generality should result. Of course you can interpose your own subclass of `Production` in order to subclass that, with your own definition of new virtual functions. Because the `ParseTree` holds an object of class `Production`, you will have to cast it to your subclass to make use of your own virtual functions.

Chapter 6: Using the Test Facility

Here is the parms tab page. You use this to set up a test of your grammar:



If you click on start, you will be asked to identify the parser's rule file and the lexer's rule file. If you have not checked "from this text", you will also be asked for the file to parse. You can change these files at any time by clicking on "pick input files".

The start symbol is optional. If it is not supplied, the head of the first rule in the grammar is used as the start symbol.

Once a parser is built (via clicking on "start") you can save the parser with the "save". That will prompt you for a file name. Conversely, clicking on "restore" will ask you for a parser file to load. However, unless the "use saved parser" is checked, the test facility will use the files mentioned above. By checking and unchecking the "use saved parser" box, you can toggle between building a new parser with the input file specified, and using an old one.

The saved parser contains the rules for the lexer and the parser that were used to build the saved parser. The start symbol is also saved and can not be overridden on a saved parser. The “parms” page will display whatever files are being used when you click on “start”.

The “start symbol” field is *very* useful for testing your grammar a “rule at a time”. If you put in the head of some rule, and fill in the “from this text” input, you can see if pieces of your grammar work. This is quite handy since testing the entire grammar at once can be difficult. As mentioned above, you cannot supply a start symbol to a save/restored parser since this is embedded in the tables that are saved.

If you open the grammar files and input file in separate windows you can rapidly run the test facility, discover the error, correct and save the appropriate file, and then click on `start` to quickly retest.

Saving and restoring a parser is entirely optional. It might be slightly faster to parse with a saved parser since there is no need to read the lexer’s and parser’s rule files, and convert these to internal form. Additionally, certain tables built by the parser are saved with the parser and so need not be constructed when the parser starts its work. Shipping a saved parser with your application will also eliminate the need to ship the rule files.

The other tabs, “result” and “info” were explained in the “quick start” chapter. It is worth repeating that a thorough testing of your grammar should exercise the “next parse” button. The only way the parser has of detecting whether there are multiple parses for a given input is for you to request each parse. This is how you can determine if your grammar is ambiguous. When multiple parses exist, the “info” page will explain where the ambiguity lies. Then you can adjust your grammar rules to make your grammar unambiguous (if you wish).

Chapter 7: Application Interface

LEXRULEBUILDER

This object reads a .txt file of lexer rules and returns a list of these rules. This list is suitable as one of the arguments to the Lexer's constructor.

```
public LexRuleBuilder(string inputFile)
          {
               ruleFile = inputFile;
          }
```

You just supply the file path as the parameter. Then you invoke:

```
public List<LexRule> getRules()
```

which returns the rules to you. `LexRuleBuilder` throws exceptions when something goes wrong, so you should be prepared to catch them. The message included therein should be helpful.

`LexRuleBuilder` tries to clean up after itself, but you should call `Dispose` on it when you are done, in case it failed to do so.

PARSERULEBUILDER

This object reads a file of parser rules ("productions") and converts them to internal form.

```
public ParseRuleBuilder(string inputFile)
```

is the construtor. You pass it the path to the parser's rules. To get the list of rules, you call either

```
public List<Production> getGrammar()
```

or

```
public List<Production> getGrammar(Assembly assem)
```

The optional `Assembly` parameter specifies the assembly that contains all of your subclasses of `Production`. As explained in the chapter on the parser's rule format, you can specify a subclass along with the parser's rules. Then, `ParseRuleBuilder` will construct a rule of your own subclass. You use this to override some virtual functions in class `Production`. This can help you define logic for processing the parse trees returned by the parser. See the chapter on sample applications for details.

The mechanism for building parser rules of your own subclass is contained in a virtual function:

```
public virtual Production factory(Production myRule)
```

This can be overridden in your own sublcass of `ParseRuleBuilder`, if you desire to design such. While we do not expect you will do this, it is included for your convenience.

The `getGrammar` method is what you use to obtain the list of `Production`s that you will pass to the parser.

LEXER

The constructor for the lexer is:

```
public Lexer(string inputSource, List<LexRule> rules, bool inputFromText)
```

The first parameter is a string that is either a file (path) name to the parser's input, or else some actual text to be parsed. The third parameter will be `true`, if the latter. The second parameter is the list of lexer's rules, normally obtained from the `LexRuleBuilder`, as described above. But you can use an alternate method to get the lexer's rules if you invent one.

The lexer will clean up after itself, but you should calll `Dispose` on it when you are done.

If you want to restart the lexer at the start of its input, you can call `reset`. Then you would normally call `reset` on the `Parser`, passing it the restarted lexer (see below).

You can write your own lexer. This is not easy since you must conform to the `ILexer` interface, and will have to understand how our `Lexer` works. But the parser was designed to work with any object that conforms to the specification.

Once you have build a `Lexer`, you are ready to construct the `Parser`.

PARSER

The constructors are:

```
public Parser(List<Production> rules, ILexer lexer)

public Parser(List<Production> rules, ILexer lexer, string
startSymbolName)
```

If you do not supply a `startSymbolName`, the parser will use the head of the first rule in its grammar. The list of rules is the output of the `ParseRuleBuilder`, the `ILexer` is from the `Lexer` constructor.

The parser will clean up after itself, but you should call `Dispose` when you are done with it.

Like the lexer, the parser can be reset via reset:

```
public void reset(ILexer lexer)
```

You should pass it a reset lexer (as explained above).

The two methods to actually parse the input are:

```
public IEnumerable<ParseTree> allParses()

public ParseTree parse()
```

The first form is preferred, as it lets you retrieve all parses in a loop. If you only want the first parse you can use the second form.

Here is sample code to build a parser, via the other components of LPS:

```
public List<string> setup(string parseRuleFile, string lexRuleFile,

            bool inputFromText, string inputToParse, string startSym)
```

```
18      {
19        try
20        {
21          LexRuleBuilder lexBuilder = new
    LexRuleBuilder(lexRuleFile);
22          List<LexRule> lexRules = lexBuilder.getRules();
23          lexer = new Lexer(inputToParse, lexRules, inputFromText);
24          lexBuilder.Dispose();
25
26          ParseRuleBuilder ruleBuilder = new
    ParseRuleBuilder(parseRuleFile);
27          List<Production> parseRules = ruleBuilder.getGrammar();
28          msgs.AddRange(ruleBuilder.msgs);
29
30          if (ruleBuilder.msgs.Count == 0)
31            parser = new Linguist.Parser(parseRules, lexer,
    startSym);
32        }
33
34        catch (Exception ex)
35        {
36          msgs.Add(ex.Message);
37          appFailed = true;
38          rulesFailed = true;
39        }
40
41        return msgs;        //setup is OK
42      }
```

The parser is built at line 31.

Here is sample application code to retrieve all the parses:

```
bool getTrees()
    {
      int parseCount = 0;

      foreach (ParseTree pTree in parser.allParses())
      {
        parseCount++;

        parseTree.Add(pTree.clone());


        if (parseCount == 2)
        {
          msgs.Add("CA01: multiple parses exist");
        }

        msgs.AddRange(parser.parseInfo);
      }

      msgs.AddRange(parser.parseInfo);

      if (parser.badParce)
      {
        appFailed = true;
        return false;
      }

28              return true;
29          }
```

Note how the parses are saved (cloned) at line 9. Note how messages are collected for each parse at line 17, and also outside the iteration at line 20. The latter will either explain why the parse failed, or be a summary message for all the parsers. You can determine that the input did not match the grammar (i.e. no parse possible) when `badParce` is true (tested at line 22).

Normally, the parser will not throw exceptions (accept for unanticipated faults). We intended that all communication to the calling program be through the `parseInfo` and `badParce` fields.

If you wish to save the parser, you use:

```
public void store(string fileName)
```

You pass the file name as a parameter (we suggest using the extension ".par" for saved parser files). The corresponding method to restore a saved parser is:

```
public static Parser restore(string fileName)
```

You would invoke this instead of the constructors mentioned above. Note that the start symbol is fixed when you save the parser. Restoring the parser also restores the lexer. The restored parser is ready to go, but you need to call reset on it and its lexer in order to establish the input to be parser. Here is sample code to do that:

```
parser = Parser.restore(savedParserFile);
                    lexer = (Lexer)parser.lexer;
                    lexer.reset(parseMe, fromText.Checked);
                    parser.reset(lexer);
```

The lexer `reset` parameters are explained above.

PARSETREE

The `ParseTree` is the object returned for each parse, by the parser. You operate on this as demonstrated in the sample applications chapter. The fields you will be interested in are:

```
        public Decoration decoration = null;  //set by user application

        public Production prodn;  //used to resolve this branch
                                  //null means this is a terminal node

        public ParseElement symbol;    //matches head of the prodn or
                                       //is a Token
        public List<ParseTree> children;   //matches tail of the prodn
```

Use of the `Decoration` object is explained in the sample applications. The `prodn` field is the parser rule that this node of a parser tree represents. The `symbol` is either the head of that `prodn`, or else is a `Token` (in which case the `prodn` will be null). The `children` will be `null` if we have a `Token`, or else will be the tail of the rule in the `prodn`. Each of the `children` is also a parse tree, so you can see how the entire tree is built up from nodes, each of which is also a `parseTree`.

There are some helpful methods defined in this class:

```
public IEnumerable<List<ParseTree>> bottomUp()
```

This iterator will deliver the nodes in the parse tree from bottom to top. It insures that no tree is returned unless its `children` have been returned in a prior call.

```
public ParseTree clone()
```

When you receive a `parseTree` from the parser you need to either process it or, if you intend to call the parser for another parse, save it with the `clone` method. The parser makes use of, and changes, the previous `parseTree` when constructing the next one.

```
public List<ParseTree> leafNodes()
public List<Token> leafTokens()
```

The first function returns all of the nodes that are at the bottom of the `parseTree`. Each of these nodes contain a `Token`. You can use the second method to retrieve these directly.

The following method counts all the nodes in the `parseTree`:

```
public int totalNodes
```

<u>TOKEN</u>

This object represents the input to the parser. It is matched to the terminals in the parser rules. The fields available to you are:

```
public String text;
public int lineNumber;
public Regex regex
```

The `text` are the characters in the input that were matched to get the `Token`. The `regex` is the regular expression that the `text` matched. The `lineNumber` is that of the input file at which the text was found.

Chapter 8: Principles of Operation

There are two basic kinds of parsers: top-down and bottom-up. The former are based on a depth first search (see for example, http://www.frontiernet.net/~fredm/dps/Contents.htm, Chapter 3). Their main defect is the inability to handle left recursive rules (they will loop on such). Although any grammar can be rewritten to eliminate left recursive rules, the resulting form is sometimes unnatural and hard to understand.

Bottom-up parsers (sometimes called "shift-reduce" parsers) usually analyze the grammar and build a set of tables, before parsing starts. This static analysis forces them to make decisions about the ambiguity of the grammar before they see the actual input. Because of the limitations of this analysis they can tag a grammar as ambiguous even if it is not. Furthermore, during a parse they are (typically) limited to looking at only the next input token before a deterministic decision must be made. For some grammars, outside the scope of bottom-up parsers, more than one input token is required to disambiguate the situation.

It should be remarked that ambiguity is necessary in some grammars (e.g. natural languages), and we do not want the parser to stop, but to return all the valid parses of the input. Then these can be analyzed outside the parser, in a larger context, to pick the correct parse.

LPS uses a bottom-up technique, called LR parsing to determine valid (acceptable) input. By itself, the LR machine would determine the next action based on the last token received from the lexer. That action would be either a "shift" (add the token to a stack), or a "reduce" (remove the "tail' of a rule at the top of the stack, and replace it with the rule's head). When an LR machine could legitimately do two or more actions, it would normally report an error and stop. In LPS, however, these alternatives are given to a depth first search DFS). Some will be rejected as the parse proceeds because they do not result in a valid parse. Others will lead to a parse and are given back to the user by LPS.

Thus LPS can parse grammars that LR (and similar systems) would reject as ambiguous. It should handle almost any context free grammar.

LR systems have been extended to SLR and LALR systems, which build more elaborate tables (using "follow sets" or "look-ahead" sets). The effect of this extension is to reduce the number of false conflicts reported, and thereby enables these parsers to handle more grammars.

In LPS we do not use the LALR extension to the LR machine. We lose no functionality thereby (the DFS will eliminate the false alternatives anyway), and our logic is considerably simpler and the tables take much less space. The disadvantage is that the parser runs a bit slower (but the tables take less time to build), since more alternatives must be explored by the DFS. Experimentation suggests that this tradeoff is reasonable.

Because LPS uses a DFS, it is difficult to determine when an error actually exists. We expect some of the paths to result in incomplete parses (because the LR machine delivers these to LPS), so we cannot report an error when this occurs. Only when the DFS is completely exhausted, and we are not at end of file, can we be assured an error exists.

LPS will track each error as it occurs, but only the one that consumed the most tokens from the input will be reported back to the user.

A similar difficulty occurs when trying to determine if a grammar is ambiguous. An LR/LALR system can report out a (possibly false) ambiguity during the first (and only) parse it constructs. LPS must try a second parse before it knows that the grammar is ambiguous. This places a burden on the user to ask for each (or at least the second) parse in order to make the determination.

Because of the DFS, LPS requires that the lexer be able to “back up”, when we backtrack. This is handled by maintaining links in the tokens. The lexer must set these links when it returns a Token to the parser. Each node in the DFS keeps the last token it asked for. It hands this to the lexer when the “next” Token from the input is required. Hence backtracking across a DFS node effectively resets the lexer appropriately.

The current version of LPS stops upon the first error: there is no recovery and no attempt to parse the rest of the input. Various alternatives are under consideration but are postponed until more experience is obtained. It is thought that the interactive nature of the test facility should make error correction in the grammar or input relatively fast and easy.

The save and restore feature of LPS will allow the LR tables, as well as the internal forms of the lexer and parser rules to be stored on disk. If you are using LPS in an interactive application, you might find that using a saved parser is a bit faster than building one up each time you want to parse the user’s input.

Chapter 9: English

The ***English*** project uses LPS to parse English sentences. After a parse tree is obtained, it is converted into an object that represents the meaning of the sentence. The project is in development, but a brief description is included here to explain a special consideration when using LPS to parse natural language.

The fact is that multiple parses will be possible from many, if not most, single English sentences. One reason for this is that a given string might represent different words of different parts of speech (*POS*). For example, the string “flies” can be a verb (e.g. “He flies the plane”), or a plural noun (e.g. “The flies landed on the table”). Because we want individual strings (single word text) to be terminals to the parser (e.g. NOUN, VERB, ADJ), this presents a problem for the way a usual lexer would work with a parser. Commonly, the parser would ask the lexer for the terminal and the lexer would have to commit to one type of terminal for the input (either a noun or a verb, but not both: the parser wants a definite tag).

Our parser was designed to handle this problem as follows: as a parse is being developed, the parser asks the lexer for the next token. The parser then asks if the token is of a certain type (e.g. Is it a noun?). As different parses are developed (because other rules are tested), the lexer will be invoked (on the same “next” token) asking if the token is of a different type.

The interface between the parser and the lexer at this point is `Token.symbolMatch.` For the ***English*** system we have subclassed `Token` and overridden `symbolMatch` in the subclass. The overridden method will look the word up in a lexicon and determine if (one of its many) POS can match what the parser is testing for. In this way, the same word can assume different POS's as the parser tests for them (presumably when trying to shift against different rules). Without such a mechanism, we would have to put the entire lexicon in the lexer defintion file which is hardly possible.

As explained in the *Principles of Operation*, we use a DFS to test the different parsing alternatives and this allows us to back up in the input stream to retest a word for different POS's.

Use of this interface also allows us to fetch other useful information from the lexicon, such as synonyms and definitions, and to determine the base form of the word (e.g. Translate “flies” to either “verb-to fly” or “noun, plural, of fly”). You may find other ways to use this interface, making your lexer much more powerful than could be done only with the lexer's rule file.

If you use Linguist to parse grammars whose input strings can represent but one kind of terminal, the `Token.symbolMatch` would not need to be overridden and the defaults will work as expected.