

Getting Started with LPG

LPG (a.k.a. JikesPG) is a LR parser generator that you can use to build automatically lexers and parsers for LALR(k) languages. Lexing and parsing actions may be written in almost any programming language including Java, C and C++ , although here we only consider actions and parsers written in Java. The LPG system is much like LEX and YACC in its use of LR parsing, but it is also like ANTLR in its overall structure and its approach to lexical analysis. We hope that LPG will be a highly usable and efficient replacement for ANTLR, CUP and similar systems.

Constructing a parser is generally quite a complicated process in which many detailed issues must be addressed. Of course, a parser generator, such as LPG, is intended to simplify and partially automate the task, but the workings and use of generator itself are difficult to explain and understand. In this document we explain the use of LPG using an example lexer and parser for a little expression grammar called **LEG**. We do not explain in detail the various options for LPG , the various files it generates and its runtime. Understanding will come, we hope, from the examples.

The Little Expression Grammar (LEG) Example

Suppose you want to build a lexer and parser for a simple C-like expression language, which we will call LEG. The LEG language is specified by a grammar written in standard BNF. With LPG you specify the grammar rules and associated processing in a text file with a “.g” extension, although any extension may be used (an alternative is “.lpg”). You also specify the lexical analyzer in a similar way.

We have included the LEG example in this distribution. You may want to compile and run it before delving into the details. We’ll do this from the command line.

Running the LEG Example

First run LPG on the lexer grammar file, **leglexer.g** . Here is the command line output (but we have suppressed some of the options and statistics).

```
C:\legexample\leg>lpg leglexer.g
```

```
Options in effect for leglexer.g:
```

```
ACTION-BLOCK= ("LegLexer.java", "/.", ". /")
FILE-PREFIX="LegLexer"
LALR=5 NOLIST PACKAGE="leg" PARSE-TABLE="lpg.*" PREFIX="Char_"
PRS-FILE="LegLexerprs.java" SYM-FILE="LegLexersym.java"
TABLE=JAVA
```

```
leglexer.g is LALR(5).
```

```
Number of Terminals: 73
```

```
Number of Nonterminals: 28
```

```

Number of Productions: 170
Number of Items: 378
Number of States: 30
Number of look-ahead states: 18
Number of Shift actions: 80
Number of Goto actions: 14
Number of Shift/Reduce actions: 427
Number of Goto/Reduce actions: 50
Number of Reduce actions: 1068
Number of Shift-Reduce conflicts: 0
Number of Reduce-Reduce conflicts: 0

```

. . .

Based on the options given in **leglexer.g** grammar file, LPG has generated three Java files. The names of these files are prefixed with the string given by the “file-prefix” option, which here is **LegLexer**. The “sym-file”, **LegLexersym.java**, is an interface defining the lexer’s terminal symbols (characters and character ranges). The “prs-file”, **LegLexerprs.java**, is a class defining the tables the LPG parser will use. The file **LegLexer.java**, specified in the “action-block” option, is the class defining the actions taken when rules are reduced. For a lexer, the actions generate the tokens to be parsed. The actions are Java statements enclosed with the brackets “/.” and “./”.

The “package” option identifies the Java package for Leg (which is **leg**) and this package declaration is placed in every Java file LPG generates. The “prs-table” option, “lpg.*” tells LPG that in the “prs-file” this package (here the LPG runtime) must be imported. The “table” option indicates programming language that the parse tables are to be accessed from (in our example it is Java).

Finally, there are various statistics about the grammar and the tables used to parse it. In particular, notice that this grammar is confirmed to be LALR(5). We discuss below the important “template” and “export-terminals” options.

Next run LPG on the Leg parser, **legparser.g**. The (somewhat modified) output is:

```
C:\legexample\leg>lpg legparser.g
```

Options in effect for legparser.g:

```

ACTION-BLOCK=("LegParser.java", "/.", "./")
BACKTRACK LALR=1 TABLE=JAVA
SERIALIZE
DAT-FILE="LegParserdcl.data" DAT-DIRECTORY="../bin"
FILE-PREFIX="LegParser"
PACKAGE="leg" PARSE-TABLE="lpg.*"
PRS-FILE="LegParserprs.java"
SYM-FILE="LegParsersym.java"

```

```
*** Shift/reduce conflict on "LEFT_BRACKET" with rule 21
```

```
legparser.g is not LALR(1).
```

```
Number of Terminals: 21
Number of Nonterminals: 9
Number of Productions: 23
Number of Shift-Reduce conflicts: 1
Number of Reduce-Reduce conflicts: 0
```

For the Leg parser the same Java “prs”, “sym” and “action” files are produced (but using the “file prefix” **LegParser**). The “backtrack” option signals that the LPG backtracking parser will try all possible alternatives when there are conflicts. The “serialize” option tells LPG to serialize the parse tables to the “dat-file” (with extension “data”) instead of including them as initialization in the “prs-file”. The Leg parser data file is **LegParserdcl.data** and LPG places it in the “dat-directory”, which is our bin directory, so it can be accessed on the class path.

The Leg parser grammar has a conflict that cannot be resolved using just one token of look-ahead. In a certain state the parser cannot determine whether to shift or reduce the “[“. The backtracking parser will try both possibilities in turn and choose the first one that allows the parse to advance through the rest of the input or signal an error if neither does.

Next compile the Leg example Java files which will place the class and data files in the bin directory.

```
C:\legexample\leg>javac -d ../bin
                    -classpath ../bin;../bin/lpg.jar *.java
```

Finally, execute the main program on the input file **test.leg**.

```
C:\legexample\leg>type test.leg
(1)  = expression;
(2)  if ( expression ) then
(3)      expression = 1;
(4)  else expression = 0;
(5)  end if;
    . . .
(10) while expression do
(11) whi le expression do
(12)     if expression then
(13)         break;
(14)     else
(15)     end if;
```

```
C:\legexample\leg>java -cp ../bin;../bin/lpg.jar leg.Main test.leg
test.leg:1:2:1:2: "variable"inserted before this token
test.leg:11:1:11:6: "WHILE"formed from merged tokens
test.leg:11:1:28:42: "END WHILE ;"inserted to complete scope
test.leg:10:1:28:42: "END WHILE ;"inserted to complete scope
```

The test file has syntax errors so we see the error messages produced by the diagnosing parser. There are options for dumping the tokens and showing the AST defined in the **Option** class which you can explore. You may want to try test2.leg, a correct example, and dump its AST. The command (and part of the output) is:

```

C:\legexample\leg>java -cp ../bin;../bin/lpg.jar leg.Main -da test2.leg
****Begin Parse
#1 (Block):  #4 #15 #23 #24 #26 #33 #38 #43 #48 #51 #54 #58 #69
#4 (AssignmentStatement):  #2 = #3
#2 (VariableExpression):  variable
#3 (VariableExpression):  expression
#15 (IfStatement):  if #6 then #7 else #11 end if ;
#6 (ParenthesizedExpression):  ( #5 )
#5 (VariableExpression):  expression
#7 (Block):  #10
#10 (AssignmentStatement):  #8 = #9
#8 (VariableExpression):  expression
#9 (ConstantExpression):  1
#11 (Block):  #14
#14 (AssignmentStatement):  #12 = #13
#12 (VariableExpression):  expression
#13 (ConstantExpression):  0
#23 (WhileStatement):  while #16 17 end while ;
#16 (VariableExpression):  expression
. . .

```

The LEG Parser

Grammar Rules

The grammar file has several “sections”, the most important of which is designated “\$Rules”. Here is the \$Rules section for LEG:

```

$Rules
    block ::= $empty
    block ::= block statement
    statement ::= variable = expression ;
    statement ::= IF expression THEN block ELSE block END IF ;
    statement ::= WHILE expression DO block END WHILE ;
    statement ::= BREAK ;
    statement ::= array_declaration ;
    array_declaration ::= IDENTIFIER [ ]
    array_declaration ::= array_declaration [ ]
    expression ::= term
    expression ::= expression + term
    expression ::= expression - term
    term ::= factor
    term ::= term / factor
    term ::= term * factor
    factor ::= variable
    factor ::= CONSTANT
    factor ::= ( expression )
    variable ::= IDENTIFIER
    variable ::= variable [ expression ]
$End

```

The rules contain “terminal” and “nonterminal” symbols. The symbols “block” and “factor” are examples of nonterminal symbols, while “WHILE” and “;” are terminal symbols. The symbol “\$empty” is special; it means the rule is an empty rule – i.e., has no right hand side. Here each rule is stated separately. But you can use alternatives – for example:

```
term ::= factor
      | term / factor
      | term * factor
```

There is no special notation for grouping, e.g. “(A | B | C)”, repetition, e.g. “*” or “+”, or optional choice, e.g., “?”. However, these conveniences can be easily expressed with additional rules.

Terminal and Non-terminal Symbols

In LPG symbols (terminal and non-terminal) are represented internally as integers. Terminal symbols represent “tokens” that are produced by scanning or lexically analyzing an input text source. (How tokens may be produced by a LPG generated parser is described below). Non-terminal symbols, such as “expression”, represent syntactic classes. A special file, called the “sym” file will be generated by LPG that contains the mapping of terminal symbols to integers. Here is what the Java LEG sym file looks like:

```
package leg;

interface LegParsersym {
    public final static int
        TK_ASSIGN = 13,
        TK_LEFT_BRACKET = 7,
        TK_RIGHT_BRACKET = 9,
        TK_SEMICOLON = 1,
        TK_PLUS = 2,
        TK_MINUS = 3,
        TK_DIVIDE = 10,
        TK_STAR = 11,
        TK_LEFT_PARENTHESIS = 14,
        TK_RIGHT_PARENTHESIS = 15,
        TK_IDENTIFIER = 4,
        TK_CONSTANT = 16,
        TK_EOF_SYMBOL = 17,
        TK_IF = 5,
        TK_THEN = 18,
        TK_ELSE = 19,
        TK_END = 12,
        TK_WHILE = 6,
        TK_DO = 20,
        TK_BREAK = 8,
        TK_ERROR_SYMBOL = 21,

        NUM_TOKENS = 21;
}
```

First notice that LPG has generated the Java interface “**LegParsersym**” declared in the package “leg”. Two LPG options are needed to make this happen, the package option and the file prefix option. These are specified at the beginning of the grammar file as follows:

```
%options package=leg
%options file_prefix=LegParser
%options prefix=TK_
```

Since this is the “sym” file, the interface name and the file name are the same: “**LegParsersym**”. Also shown here is the prefix option, used to prefix terminal symbol names as explained below.

Next note that the symbols for special characters, such as “;” have been changed and all symbols have been given the prefix “**TK_**”. You want to name the symbols in such a way that they can be used as Java program identifiers, so special characters and Java reserved words, such as “break”, must not be used to name symbols. LPG gives you two ways to avoid trouble when naming terminal symbols. First, there is the prefix option that lets you define a prefix (also a suffix) for the symbols that appear in the sym file. Second, in the section in which terminal symbols are declared, “**\$Terminals**” (or in another section called “**\$Alias**”) you can associate a terminal symbol name with another name or sequence of characters. For example, the **\$Terminal** section of the LEG grammar is:

```
$Terminals
    SEMICOLON ::= ;
    ASSIGN ::= =
    LEFT_BRACKET ::= [
    RIGHT_BRACKET ::= ]
    PLUS ::= +
    MINUS ::= -
    DIVIDE ::= /
    STAR ::= *
    LEFT_PARENTHESIS ::= (
    RIGHT_PARENTHESIS ::= )
    IDENTIFIER
    CONSTANT
    IF THEN ELSE END WHILE DO BREAK
    ERROR_SYMBOL
    EOF_SYMBOL
$End
```

Hopefully it is clear, for example, that “SEMICOLON” is the terminal symbol and “;” is the alias. In a rule the alias symbol can be used in place of the terminal symbol. Even non-terminal symbols can have aliases. You can have more than one alias for a given symbol. You could put additional terminal symbol aliases (and non-terminal symbol aliases) in the **\$Alias** section as follows:

```
$Alias
```

```

    eostmt ::= SEMICOLON
    identifier ::= IDENTIFIER
    expr ::= expression
$End

```

(By the way, it probably helps readability to enclose special characters with single quotes, e.g., writing `'['` instead of the square bracket alone).

Rule Actions

A LPG parser recognizes a syntactic class (represented by a non-terminal symbol) by parsing the text for the right-hand side (rhs) of a rule defining that class and “reducing” the rhs to the left-hand side (lhs) of that rule. The user can specify a “rule action” (sometimes called a “semantic” or “syntactic” action) to be executed whenever the rule is reduced. Rule actions typically produce an “abstract” representation of the source syntax. Often this abstract syntax is expressed as a tree and so is called an “Abstract Syntax Tree” or **AST** for short. In fact, the AST can be whatever you want it to be, and, indeed, it can be completely absent. If you use LPG for a parser that does lexical analysis, as we shall see below, your parser’s AST will not be a tree but rather a list of tokens. Here is an example of how the AST is specified in the LEG grammar:

```

block ::= $empty
    /.$BeginAction
        AstBlock block = new AstBlock();
        $setSym1(block);
    $EndAction
    ./

block ::= block statement
    /.$BeginAction
        AstBlock block = (AstBlock) $getSym(1);
        block.statement.add($getSym(2));
    $EndAction
    ./

statement ::= variable = expression ;
    /.$BeginAction
        AstAssignmentStatement assignment =
            new AstAssignmentStatement();
        assignment.lhs = $getSym(1);
        assignment.equal = $getToken(2);
        assignment.rhs = $getSym(3);
        assignment.semicolon = $getToken(4);

        $setSym1(assignment);
    $EndAction
    ./

```

A “**block**” consists of a sequence of “**statements**” (possibly empty) and its AST consists essentially of a list of “**statement**” AST’s. Notice that “**block**” is the “Start” symbol of the LEG grammar, so what the parser returns (assuming no errors) is the AST for the input “block”. You may specify the start symbol in a “**\$Start**” section or as the lhs of the

first rule in the **\$Rule** section. Also illustrated above is the AST generation for the LEG assignment statement. The class “**AstAssignmentStatement**” extends the “**Ast**” class, just as “**AstBlock**” does.

When specifying a rule action, you need to separate its text from the text of the rule. This is done with a pair of delimiters you are free to choose (well almost). In the LEG example the action delimiters are “/.” And “./”. These delimiters are specified using the “**action**” option. For the LEG parser this option is:

```
%options action=("LegParser.java", "/.", "./")
```

In addition to the action delimiters, the “**action**” option specifies the file containing the rule action method (procedure or function) that the code for each action will be associated with or folded into. For the LEG parser, this file defines the Java class, LegParser, which implements the rule action method. This class and its specification is somewhat complicated and will be described below.

To help specify the AST generating actions, we use two macros, **\$BeginAction** and **\$EndAction**. These are defined in a special section called **\$Define**. Here is the **\$Define** section for the LEG example:

```
$Header
/.
    //
    // Rule $rule_number:  $rule_text
    //./

$DefaultAction
/. $Header
    case $rule_number: ./

$BeginAction
/. $DefaultAction
    {./

$EndAction
/.    }
    break; ./

$NoAction
/. $Header
    case $rule_number:
    break; ./

$BeginActions
/.
    public void ruleAction( int ruleNumber)
    {
        switch(ruleNumber)
        {
./
```



```

$EndActions
/.
        default:
            break;
    }
    return;
}

$setSym1 /.btParser.setSym1./
$getSym /. (Ast)btParser.getSym./
$getToken /.btParser.getToken./

./

```

The last two macros, **\$BeginActions** and **\$EndActions**, are used to define the beginning and end of the **ruleAction** method. The method consists of a switch statement with a case for each rule with actions. The **\$BeginActions** and **\$EndActions** macros essentially bracket the action with a case alternative. The **\$Header** macro provides the rule number and text to document the action in the generated Java file. The use of the symbol access macros, **\$setSym1**, **\$getSym**, and **\$getToken**, is explained below. There are also several LPG predefined macros that facilitate references to rule properties. In this example we see the macros **\$rule_number** and **\$rule_text**.

Here is the Java code LPG generates for the actions shown above:

```

public void ruleAction( int ruleNumber)
{
    switch(ruleNumber)
    {
        . . .

//
// Rule 3:  block ::=
//
        case 3:
        {
            AstBlock block = new AstBlock();
            btParser.setSym1(block);
        }
        break;

//
// Rule 4:  block ::= block statement
//
        case 4:
        {
            AstBlock block = (AstBlock) (Ast)btParser.getSym(1);
            block.statement.add((Ast)btParser.getSym(2));
        }
        break;

//
// Rule 5:  statement ::= variable ASSIGN expression SEMICOLON
//

```

```

        case 5:
        {
            AstAssignmentStatement assignment = new
                AstAssignmentStatement();
            assignment.lhs = (Ast)btParser.getSym(1);
            assignment.equal = btParser.getToken(2);
            assignment.rhs = (Ast)btParser.getSym(3);
            assignment.semicolon = btParser.getToken(4);

            btParser.setSym1 (assignment);
        }
        break;

        . . .

        default:
        break;
    }
    return;
}

```

When the parser recognizes the right hand side of a rule, the parser calls the rule action method giving it the rule number as argument.

While parsing the input tokens, LPG maintains two stacks: a “symbol” stack, for non-terminal symbol AST, and a “token” stack for the tokens not yet fully reduced. A parser “shift” action pushes the current token onto the token stack, while a rule reduce action effectively replaces the AST generated for the rule’s right hand side non-terminals with the AST for the left hand side (i.e., the rule itself). The LPG parser (in our example “**btParser**”) has methods to get symbols and tokens (**getSym** and **getToken**) from these stacks and to set the symbol stack with the generated AST (**setSym1**). The macros, **\$setSym1**, **\$getSym**, and **\$getToken**, enable you to use these methods without identifying the specific LPG parser being used (there are actually three of them).

The symbols in the right hand side of a rule are numbered from left to right starting with one. If the *i*th symbol is a terminal symbol, its token is accessed by **getToken(i)**. If the *i*th symbol is a non-terminal symbol, its AST is accessed as `getSym(i)` and its leftmost token **getToken(i)**. To set the AST for the rule, use the method **setSym1(astObject)**. The **getToken** method returns an **int** value while the **getSym1** method returns an instance of **Object**. The argument of **setSym1** is also an instance of **Object**. This generality is necessary since the AST class cannot be predefined (when you get a symbol, you may need to cast it to your AST type).

The Action Class

The rule actions described above are contained in an “action class”, which in the LEG example is called **LegParser**. The purpose of this class is to implement the LPG **RuleAction** interface (which specifies the **ruleAction** method) and a **parser** method that invokes an LPG parser on the token stream. The rule actions may be implemented by a single method (as in the LEG example) or by an array of action methods (one for each rule) defined in inner classes and accessed through an array of instances (we’ll not

discuss this approach here). The action class is (incompletely) specified in a **\$Headers** section. Here is the LEG action class:

```
$Headers
/.
    package leg;

    import lpg.*;
    import java.util.ArrayList;

    public class LegParser implements RuleAction
    {
        LexStream prsStream;
        ParseTable prs;
        BacktrackingParser btParser;

        public LegParser(LexStream prsStream)
        {
            this.prsStream = prsStream;
            this.prs = new LegParserprs();
        }

        public Ast parser()
        {
            try
            {
                btParser = new BacktrackingParser(
                    (TokenStream)prsStream, prs,
                    (RuleAction)this);
            }
            catch (BadParseSymFileException e)
            {
                prsStream.reportError(0,
                    "BadParseSymFileException");
                return null;
            }
            try
            {
                return (Ast) btParser.parse();
            }
            catch (BadParseException e)
            {
                prsStream.reset(e.error_token);
                DiagnoseParser diagnoseParser = new
                    DiagnoseParser(prsStream, prs);
                diagnoseParser.diagnose(e.error_token);
            }
            return null;
        }
    }
./

$End
```

The **LegParser** class implements the LPG interface **RuleAction** whose **ruleAction** method is called by the LPG parsers. The specification of this class in the **\$Headers**

section is incomplete since the body of the **ruleAction** method is woven throughout the rules. As shown above, two macros, **\$BeginActions** and **\$EndActions**, are defined to delimit the start and end of this method. The **\$BeginActions** macro is called at the start of the **\$Rules** section and the **\$EndActions** macro is placed in the **\$Trailers** section (within which the **LegParser** class is completed).

The **LegParser** class has three attributes needed for parsing. The token stream (here an instance of the class **LexStream**), the parse table class (containing the tables and constants needed for parsing the token stream) and the LPG parser itself (in this case the LPG **BacktrackingParser**). The **LegParser** constructor is passed the token stream and it creates an instance of the LPG generated parse table class, **LegParserprs**, that contains the various parse tables, constants and access methods, thereby implementing the LPG **ParseTable** interface. The method **parser** creates the backtracking parser and invokes it. If the **parser** encounters a syntax error, it throws a **BadParseException**. The parser method catches this exception and calls the LPG **DiagnoseParser** to find all syntax errors in the input. If the diagnosing parser is called, no actions are executed and hence no AST is generated.

The Token Stream

The input to the LPG parser is an array (or array list) of tokens. The token array is produced by scanning and tokenizing an input array of characters. The LPG parsers assume that the token array contains **all** of the tokens. This facilitates look-ahead determination and syntactic error recovery. You can modify the deterministic parser to get tokens on demand, but we do not recommend doing that. To the LPG parser a token is simply an index into the token array. As for the token itself, there is only one attribute that is relevant to parsing: the token **kind**. The kind is the identity of the token as a terminal symbol. LPG associates with each terminal symbol its token kind, which is written out to the “**sym**” interface file (in our example, **LegParsersym.java**).

Typically (though not always) you will create a token class containing the attributes you need for parsing and semantic analysis. In addition to the “**kind**” attribute, token location attributes, which specify input source and placement, are often necessary as well, especially for error reporting. In the LEG example the Token class is written as follows:

```
class Token {
    int kind = 0;
    int startOffset = 0;
    int endOffset = 0;

    Token() {}
    Token(int startOffset, int endOffset, int kind)
    {
        this.startOffset = startOffset;
        this.endOffset = endOffset;
        this.kind = kind;
    }
    public int getKind()
    {
        return kind;
    }
}
```

```

    }
    public int getStartOffset()
    {
        return startOffset;
    }
    public int getEndOffset()
    {
        return endOffset;
    }
    public String getValue(char[] inputChars)
    {
        int len = endOffset - startOffset + 1;
        return new String(inputChars, startOffset, len);
    }
    ...
}

```

There are only three attributes in this class: the kind, the starting offset in the file (or character array) and the ending offset. You might want to include a file attribute as well. We did not do so because we think it better to place such information in a “token stream” class. The LEG token stream class is called **LexStream**. It implements the LPG Java interface, **TokenStream**, that specifies the operations on the token array the parser requires, such as **getToken()**, **getKind(i)**, **getNext()**, **getPrevious()**, etc. In addition, **LexStream** has methods to create and build the token array list and access token attributes through a token array index. Here is part of the **LexStream** declaration:

```

class LexStream implements TokenStream
{
    private int index = 0;
    private int len = 0;
    List tokens;
    CharStream charStream;

    LexStream(CharStream charStream)
    {
        this.charStream = charStream;
        tokens = new ArrayList();
        addBadToken();
    }
    void addToken(Token token)
    {
        tokens.add(token);
    }
    . . .
    String getTokenText( int i )
    {
        Token t = (Token)tokens.get(i);
        return t.getValue(charStream.inputChars);
    }
    . . .
    int getTokenLength( int i )
    {
        Token t = (Token)tokens.get(i);
        return t.getEndOffset() - t.getStartOffset() + 1;
    }
}

```

```

int getLineNumberOfTokenAt(int i)
{
    Token t = (Token)tokens.get(i);
    return charStream.getLineNumberOfCharAt(t.getEndOffset());
}
int getColumnOfTokenAt(int i)
{
    Token t = (Token)tokens.get(i);
    return charStream.getColumnOfCharAt(t.getStartOffset());
}
Token getTokenAt(int i) { return (Token)tokens.get(i); }
. . .
}

```

This class has methods for adding a token to the array list and for getting various token attributes, such as its text, length and line and column position in the character stream.

Thus, we need another class, a character stream class, that accesses the array of characters from which the tokens are drawn. Before discussing this class, you should note that these stream classes are specified by the user – they are not part of the LPG runtime. LPG has two interfaces the user must implement: the **TokenStream** interface and the **RuleAction** interface. In the future we may provide the LPG runtime with token and character stream implementations as well as a factory to create instances.

The Character Stream

The array of tokens is generated by scanning (or lexing) the input character stream. The approach taken here is to read the entire input file into a character array and pass it to a scanner class (which we call a character stream). Traditionally, scanners for LR parsers tokenize the input using either hand written methods or a table driven deterministic finite automaton (DFA). For LL parser generator systems the scanner is sometimes specified with an LL grammar so that parsing and lexing become indistinguishable – the parser terminal symbols are tokens while the lexer's are characters. Like these LL systems, LPG allows you to specify your lexer with an LR grammar. You write your lexer in much the same way as you write your parser – a lexer is just a parser with characters as tokens. The AST produced by such a lexer-parser is the token array list for your parser. The LEG character stream class is called **CharStream**. Here is a sample of its contents:

```

class CharStream implements TokenStream, LegLexersym,
                           LegLexerTokenKindMap, ParseErrorCodes
{
    private int index = -1;
    private int len = 0;
    char[] inputChars;
    int line = -1;
    int[] lineOffsets;
    int newLength = 0;
    final static int INITIAL = 4000;
    Option option;

    CharStream() {}
}

```

```

CharStream(Option option)
{
    this.option = option;
    inputChars = option.getInputChars();
    len = inputChars.length;
    index = -1;
    inputChars[len - 2] = '\n';
    inputChars[len - 1] = '\uffff';
    setLineOffset(-1);
}

. . .

// Methods that implement the TokenStream Interface

public int getToken()
{
    index = next(index);
    char c = inputChars[index];
    if (c > 32) { return index; }
    if (c == 10) { setLineOffset(index); return index; }
    for (;;)index = next(index))
    {
        c = inputChars[next(index)];
        if (c > 32 || c == 10) break;
    }
    return index;
}

public int getKind( int i )
{
    char c = inputChars[i];

    if (c < 128) return tokenKind[c];
    else if (c == '\uffff') return Char_EOF;
    else return Char_AfterASCII;
}

. . .
}

```

The **CharStream** class implements several interfaces, most of which simply give access to constants. Like the **LexStream** class, **CharStream** implements the LPG **TokenStream** interface. This is because an LPG parser does the lexical analysis. You can see that **CharStream** has required **TokenStream** methods, such as **getToken()** and **getKind(i)**. Note also that a token is just an index into the character array **inputChars**.

The **LegLexerSym** interface provides the values for the lexical tokens (actually characters and character ranges) that are scanned. Here is a sample of that interface:

```

interface LegLexersym {
    public final static int
        Char_CtlCharNotNL = 1,
        Char_Blank = 2,
        Char_NL = 3,

```

```

Char_a = 17,
Char_b = 36,
Char_c = 37,
. . .
Char_0 = 26,
Char_1 = 27,
Char_2 = 28,
Char_3 = 29,
. . .
Char_Plus = 7,
Char_Minus = 8,
Char_Slash = 5,
Char_Star = 6,
. . .
Char_AfterASCII = 51,
Char_EOF = 73,
. . .
}

```

Notice that all control characters (except line feed) are represented by a single token as are all characters above the ASCII range. Again, the **int** values for the tokens are assigned by LPG in a way that optimizes the size of the parse tables.

The **LegLexerTokenKindMap** interface defines the mapping of character values to token values. A small sample shows how this is done:

```

interface LegLexerTokenKindMap extends LegLexersym
{
    public final static int tokenKind[] =
    {
        . . .
        Char_CtlCharNotNL,
        Char_CtlCharNotNL,
        Char_NL,
        Char_CtlCharNotNL,
        Char_CtlCharNotNL,
        . . .
        Char_Blank,
        Char_Exclamation,
        . . .
        Char_z,
        Char_LeftBrace,
        Char_VerticalBar,
        Char_RightBrace,
        Char_Tilde,
        Char_AfterASCII
    };
}

```

The map is an array indexed by the first 128 ASCII characters. The `getKind(i)` method uses this array to classify the character as a token for the LPG parser.

The **CharStream** class needs to handle lexical errors. It does this by implementing the **reportError** method specified in the LPG **TokenStream** interface. LPG lexers and parsers report errors (lexical or syntactical) via the **reportError** method. The

ParseErrorCodes interface enumerates the various kinds of errors and descriptive text so that an appropriate message may be printed.

The LEG Lexer

The LEG lexer is actually an LPG parser much like the LEG parser but with some important differences. First, it uses a **CharStream** for character input and produces a **LexStream** containing an array list of lexed tokens as output. The LEG lexer uses a slightly modified LPG parser because it is difficult to scan more than one token at a time (especially key words which can be prefixes of identifiers) and because the desired error recovery for a lexer is simply to skip (and report) unrecognized characters. We have a special LPG parser for lexical analysis, the `LexParser`.

The LEG lexer has grammar rules for specifying tokens for the LEG parser. You should peruse **leglexer.g** in detail, but here we illustrate some of the rules for identifiers and keywords.

```
Identifier ::= Ident
            /.$DefaultAction
              makeToken($getToken(1), $getSym(1), $_IDENTIFIER);
            $EndAction
          ./
```

```
Ident ::= Letter
        /.$DefaultAction
          $setSym1($getToken(1));
        $EndAction
      ./
    |   Ident Letter
    /.$DefaultAction
      $setSym1($getToken(2));
    $EndAction
  ./
    |   Ident Digit
    /.$DefaultAction
      $setSym1($getToken(2));
    $EndAction
  ./
```

```
ReservedWord ::= b r e a k
               /.$DefaultAction
                 keyWord($rule_size, $_BREAK);
               $EndAction
             ./
    |   d o
    /.$DefaultAction
      keyWord($rule_size, $_DO);
    $EndAction
  ./
```

. . .

Here are the two methods, **makeToken** and **keyWord**, used in these rules:

```
void makeToken(int startOffset, int endOffset, int kind)
{
    Token t = new Token(startOffset, endOffset, kind);
    $prs_stream.addToken(t);
    printValue(t);
}

void keyWord(int size, int kind)
{
    Token t = new Token($getToken(1), $getToken(size), kind);
    t.setStartOffset($getToken(1));
    $prs_stream.addToken(t);
    printValue(t);
}
```

Both methods create a token (using the starting and ending offsets and kind) and add it to the **LexStream**'s tokens list (note that the token stream is named by the macro **\$prs_stream**). (The **printValue** method prints out the value of the token under a debugging option.)

The rules for **Identifier** and **ReservedWord** illustrate how an LPG lexer uses the “sym” stack. In the LPG **LexParser** the “sym” stack holds integers, not Objects and no “AST” is defined or returned. The integer our actions place on the sym stack (using the method **setSym1(i)**) is the location of the rightmost character reduced to that left hand side. Thus, invoking **getSym(i)** in a rule action retrieves the index in the character stream of the rightmost character scanned for that non-terminal symbol. You may recall that invoking **getToken(i)** on a non-terminal symbol obtains the index of the leftmost character scanned for that symbol. The right hand side of the **Identifier** rule is the single non-terminal **Ident**. Its start offset is given by **\$getToken(1)** and its end offset by **\$getSym(1)**.

The same is done for the reserved words. However, in this case, the right hand side consists in spelling out the letters of the word (whose characters are just tokens for this lexer). Although it is easy enough to get the starting offset of the reserved word, we need some help from LPG to get the ending offset. LPG has a predefined macro, **\$rulesize**, that gives the size (or length) of the right hand side of the current rule (the one in whose action block the macro is used). Thus, **\$getToken(\$rulesize)** gets the index of the last terminal symbol in a rule, which, in this example, is the end offset of the reserved word.

It is important also to understand how we obtain the token kind. The LEG parser needs the token kind in order to parse the token stream. We could get access to the token kind in the following way. We run LPG on the LEG parser grammar which defines all the terminal symbols – IDENTIFIER, PLUS, BREAK, etc. – and make available to the LEG lexer the “sym” interface, **LegParsersym**, which contains all these symbols prefixed with “TK_” with their integer values. Then the lexer actions could name the token kind directly. For example, the lexer non-terminal symbol “**Identifier**” represents the parser terminal symbol “**TK_IDENTIFIER**”.

While this approach works, it is obviously not desirable since it is not easily maintainable and very error prone. We have added to LPG an “export/import” capability somewhat analogous to that found in ANTLR. A grammar file may have a **\$Export** section in which the terminal symbols to be exported to another grammar are enumerated. In addition, there is an option “**export_terminals**” that specifies a “sym” file for the exported terminals. The LEG lexer exports all of the terminal symbols that will be used in the LEG parser. The export_terminals option tells LPG to create a Java “sym” interface for the exported symbols using the specified prefix (if any) and assign values to the symbol names in the order of appearance in the export section. In the LEG lexer this option is stated as follows:

```
%options export_terminals=("LegParsersym.java", "TK_")
```

The LEG lexer has the following export section,

```
$Export
    ASSIGN  LEFT_BRACKET  RIGHT_BRACKET  SEMICOLON
    PLUS    MINUS    DIVIDE    STAR
    LEFT_PARENTHESIS  RIGHT_PARENTHESIS
    IDENTIFIER  CONSTANT  EOF_SYMBOL
    IF THEN ELSE END WHILE DO BREAK
    IDENTIFIER
$End
```

for which the following interface is generated:

```
interface LegParsersym {
    public final static int
        TK_ASSIGN = 1,
        TK_LEFT_BRACKET = 2,
        TK_RIGHT_BRACKET = 3,
        TK_SEMICOLON = 4,
        TK_MINUS = 6,
        TK_DIVIDE = 7,
        TK_STAR = 8,
        TK_LEFT_PARENTHESIS = 9,
        TK_RIGHT_PARENTHESIS = 10,
        TK_IDENTIFIER = 11,
        TK_CONSTANT = 12,
        TK_EOF_SYMBOL = 13,
        TK_IF = 14,
        TK_THEN = 15,
        TK_ELSE = 16,
        TK_END = 17,
        TK_WHILE = 18,
        TK_DO = 19,
        TK_BREAK = 20,
        TK_IDENTIFIER = 21,

        NUM_TOKENS = 21;
}
```

```

    public final static boolean isValidForParser = false;
}

```

This is not a sym file the LEG parser can use, because the LPG values for terminal symbols are determined by the parse table compression algorithm. Thus, there is a boolean constant **isValidForParser** to indicate that this sym interface is invalid.

Having names for the exported symbols enables you to assign kinds to the lexed tokens. Unfortunately, using the prefix (“TK_” for the LEG example) makes the actions less readable and can lead to errors should this prefix not agree with the one specified in the parser file. To remedy this situation LPG introduces a macro for each exported symbol. The form is simply “_<terminal>” and you reference the symbol with “\$ _<terminal>”. Thus, **\$_IDENTIFIER** and **\$_BREAK** represent the kind of **TK_IDENTIFIER** and **TK_BREAK** respectively. This makes references to exported names independent of the prefix.

Just as the LEG lexer exports terminal symbols, the LEG parser imports them. The LEG parser has the following option to import its terminals from the LEG lexer:

```
%options import_terminals=LegLexer.g
```

Notice that the lexer grammar file is indicated, not the symbol file, **LegParsersym.java**. LPG reads the imported grammar file and takes all the terminals it exports as its own. The importing grammar may include these (and other) terminal symbols in a **\$Terminals** section. However, you should ensure that it specifies the same symbol file and prefix as the imported grammar file. When LPG generates the parser symbol file using the name specified in the imported grammar file (**LegParsersym.java**), it includes the imported token names (prefixed as specified in the imported file) but with their correct values. The **isValidForParser** flag will be set **true**. The backtracking or deterministic parser constructor throws the **BadParseSymFileException** if this flag is **false**. This guarantees that the symbol file generated by the parser is used when compiling parser and lexer together.

When terminal symbols are imported into a grammar file (as, for example, the exported terminals of **LegLexer.g** are imported by **LegParser.g**), LPG effectively includes the imported grammar file and processes it before processing the importing file. Thus, the command,

```
C:\legexample\leg>lpg legparser.g
```

processes first the lexer and then the parser grammar files.

The lexer’s action class, **LegLexer**, is somewhat different from the parser’s class, **LegParser**. Here is part of the Java declaration:

```

public class LegLexer implements RuleAction, LegParsersym
{

```

```

CharStream lexStream;
LexStream prsStream;
ParseTable prs;
LexParser lexParser;

public LegLexer(CharStream lexStream)
{
    this.lexStream = lexStream;
    this.prsStream = new LexStream(lexStream);
    this.prs = new LegLexerprs();
    this.lexParser = new LexParser((TokenStream)lexStream,
                                   prs, (RuleAction)this);
}

public LexStream lexer()
{
    lexParser.parseCharacters();
    prsStream.addEofToken(TK_EOF_SYMBOL);
    return prsStream;
}

. . .

```

Notice that the constructor requires a **CharStream** and creates a **LexParser** passing it a **TokenStream**, a **ParseTable** and a **RuleAction**. The **lexer** method, invoked from the main or driver class, parses the character stream into a **LexStream** which is returned to the driver.

Templates

No doubt you noticed that producing a parser or lexer involves specifying quite a bit more than the symbols, the grammar rules and the rule actions. In addition to various LPG options, you need a **\$Define** section and a **\$Header** section, both of which are similar for each parser or lexer you write. To simplify your task LPG offers several predefined grammar file **templates**. Using the **template** option you essentially include the specified template file at the beginning of your grammar file. These are the presently available template files for Java written parsers:

- **btParserTemplate.g**
- **btParserTemplateA.g**
- **dtParserTemplate.g**
- **dtParserTemplateA.g**
- **LexerTemplate.g**
- **LexerTemplateA.g**

Here “bt” denotes the backtracking parser and “dt” the deterministic parser. The lexer template uses the lex parser. The letter “A” suffix indicates an alternative approach to specifying rule actions. Instead of calling one action method consisting of a select statement with a case alternative for each rule, an array of action class instances, one for each rule and containing a single method, is indexed by the rule number and the rule

action for that instance is invoked. LPG defined macros simplify the writing actions that define a class with an action method and creating an instance of the class in the array of action instances. In fact, with these templates you write you actions the same way, regardless whether you use the “select” or the “class instance” style. The alternative approach does not appear to be as efficient as the standard one. However, for grammars with many rule actions you may need it since Java places a 64K restriction on compiled object size.

You can modify these templates to suit the needs of your project.

The LEG lexer uses **LexerTemplate.g**. This is what that template contains:

```
-- An LPG Lexer Template Using lpg.jar
--
-- An instance of this template must have a $Export section and the
-- export_terminals option
--
-- Macros that must be defined in an instance of this template
--
--      $package_declaration
--      $import_classes
--      $action_class
--      $lex_stream_class
--      $prs_stream_class
--      $eof_token

%Options escape=$,table=java,margin=8,nobacktrack
%options action=("*.*java", "/*.", "/*.")
%options ParseTable=lpg.*

$Notice /*.$copyright./

$Define

    $copyright
    /*.
    /*****
    *
    *  COPYRIGHT:
    *      (C)  COPYRIGHT IBM CORPORATION 2002
    *
    *  The source code for this program is not published or otherwise divested of
    *  its trade secrets, irrespective of what has been deposited with the U.S.
    *  Copyright Office.
    *
    *  Source File Name: %W%
    *  Version: %I%, %G%
    *
    *  Descriptive Name:
    *
    *  Function:
    *
    *  Change Activity:
    *
```

```

*****/
./
--
-- Macros that may be needed in an instance of this template
--

$setSym1 /.lexParser.setSym1./
$getSym /.lexParser.getSym./
$getToken /.lexParser.getToken./
$lex_stream /.lexStream./
$prs_stream /.prsStream./

$Header
/.
    //
    // Rule $rule_number:  $rule_text
    //./

$DefaultAction
/. $Header
    case $rule_number:
    { ./

$BeginAction
/. $DefaultAction./

$EndAction
/.
    }
    break; ./

--
-- This macro is used to reset the parser's state stack
-- after a Token has been reduced.
$ResetStackAction
/. $Header
    case $rule_number:
        lexParser.resetStateStack();
        break; ./

$NoAction
/. $Header
    case $rule_number:
        break; ./

$BeginActions
/.
    public void ruleAction( int ruleNumber)
    {
        switch(ruleNumber)
        {
./

$EndActions
/.
    default:
        break;
    }

```

```

        return;
    }
./

$End

$Headers
/.
$copyright

$package_declaration

$import_classes
import lpg.*;

public class $action_class implements RuleAction, $exp_type
{
    $lex_stream_class lexStream;
    $prs_stream_class prsStream;
    ParseTable prs;
    LexParser lexParser;

    public $action_class($lex_stream_class lexStream)
    {
        this.$lex_stream = lexStream;
        this.prsStream = new $prs_stream_class(lexStream);
        this.prs = new $prs_type();
        this.lexParser = new LexParser((TokenStream)lexStream,
                                       prs, (RuleAction)this);
    }

    public $prs_stream_class lexer()
    {
        lexParser.parseCharacters();
        prsStream.addEofToken($eof_token);
        return prsStream;
    }
./

$End

$Rules
/.$BeginActions./

$End

$Trailers
/.
    $EndActions
./

$End

```


The template contains a copyright notice that, thanks to the **\$Notice** section, will be placed at the head of each file LPG generates. You may wish to edit or remove the **\$copyright** macro. Notice that **\$copyright** is the first macro invoked in the **\$Header** section of the template (to ensure that it appears in the action file). If you are not using templates you may obtain the same effect by including a **\$Notice** section in your grammar file. This section has zero or more action blocks. Moreover you can have more than one **\$Notice** section.

In **LegLexer.g** this template is specified as follows:

```
%options template=LexerTemplate.g
```

The result are abbreviated **\$Define** and **\$Headers** sections. In fact, we only need to define the macros used in the lexer template and specify additional methods for use in LegLexer actions. Notice, that the various sections can appear in any order.

The LEG parser uses **btParserTemplate.g**. This is some of what the template contains:

```
--
-- An LPG Parser Template Using lpg.jar
--
-- In a parser using this template, define the following macros:
--     $package_declaration
--     $import_classes
--     $action_class
--     $prs_stream_class
--     $ast_class
--
%Options escape=$,table=java,margin=8,backtrack
%options action=("*.java", "/*.", "/*.")
%options ParseTable=lpg.*

$Define

    $Header
    . . . (same as the lexer template above)
--
-- Macros that may be needed in a parser using this template
--
$setSym1 /.btParser.setSym1./
$getSym /.($ast_class)btParser.getSym./
$getToken /.btParser.getToken./
$prs_stream /.prsStream./

$End

$Headers
/.
$package_declaration
$import_classes
import lpg.*;

public class $action_class implements RuleAction
{
```

```

        $prs_stream_class prsStream;
        ParseTable prs;
        BacktrackingParser btParser;

        public $action_class($prs_stream_class prsStream)
        {
            . . . (already shown above)
        }

        public $ast_class parser()
        {
            . . . (already shown above)
        }
    ./

$End

```

. . .
In LegParser.g this template is specified as follows:

```
%options template=btParserTemplate.g
```

Using the template simplifies the **\$Define** section and eliminates the **\$Headers** and **\$Trailers** sections.

Putting the Pieces Together

The LEG example has a number of components. These are:

- The LEG grammar files (**LegLexer.g** and **LegParser.g**)
- The LPG template files (**LexerTemplate.g** and **btParserTemplate.g**)
- The LPG runtime (**lpg.jar**)
- The various AST classes (too many to enumerate here)
- The token stream classes (**CharStream** and **LexStream**)
- The Token class (**Token**)
- The Option class (**Option**)
- The main program (**Main** class)

The main program puts these parts together by determining the options, reading the input file, creating the character stream, invoking the lexer and finally invoking the parser. Here essentially is what it does:

```

public static void main(String[] args)
{
    Option option;
    LegLexer legLexer;
    LegParser legParser;
    CharStream charStream;
    LexStream lexStream;
    Ast root;

```

```

try
{
    option = new Option(args);
    int rlen = option.readInputChars();
    charStream = new CharStream(option);
    legLexer = new LegLexer(charStream);
    lexStream = legLexer.lexer();
    legParser = new LegParser(lexStream);
    root = legParser.parser();
    return;
}
catch (Exception e)
{
    System.err.println(e.getMessage());
    e.printStackTrace();
}
}

```

We hope that with the LEG example as a model you can build LPG lexers and parsers for your own applications.