

Using the LALR Parser Generator

LPG Generator Overview

The LPG system consists of two components: a generator and a language specific runtime. Here we discuss primarily the generator component.

The generator, lpg.exe, is a C++ program that takes as input a “grammar” file and produces parsing “tables”, normally expressed as constants and arrays in an application language (e.g. Java), that together with a parser (provided by the runtime) allow parsing or scanning of an input string that conforms to the language specified by the grammar. A rule may have “actions”, that is, statements in the application programming language that are executed whenever it is “reduced”, that is, when its right-hand side is parsed. Rule actions may be used to generate an intermediate representation of the input. Often this representation mirrors closely the syntax and is called an Abstract Syntax Tree (AST). The LPG generator can generate an AST automatically without user specified actions.

Input Grammar

The input to the LPG generator consists primarily of language syntax rules expressed in BNF. There is usually more input than just the grammar rules. One must specify “options” to indicate how LPG should process the grammar. The “actions” must also be specified as well as code used with the actions. To facilitate this, the input file is divided into “sections”, which we describe in detail below. Here is an example of a grammar file, **ExprParser.g** (the extension “.g” is suggested, but any other, or none, may be used):

```
%options
ast_directory=./ExprAst,automatic_ast=toplevel,var=nt,visitor=default
%options programming_language=java
%options package=expr1
%options template=dtParserTemplated.g
%options import_terminals=ExprLexer.g

$Terminals
    IntegerLiteral
    PLUS ::= +
    MULTIPLY ::= *
    LPAREN ::= (
    RPAREN ::= )
$end

$Rules
    E ::= E + T
        | T
    T ::= T * F
        | F
    F ::= IntegerLiteral
    F$ParenExpr ::= ( E )
$End
```

The options are always first. The first option indicates automatic AST generation with sub-options to indicate where to put the generated AST classes and interfaces, how to treat terminal symbols and which kind of visitor to use. The second option specifies the application programming language; it is Java which LPG will use for generating the actions, the AST and the parsing tables. The next option, “**package**”, specifies the Java package name for the generated code. The “**template**” option refers to a template file that describes additional options and code needed to simplify the generation of the parser. Finally, there is an “**import_terminals**” option that specifies the scanner (or lexer) that breaks the input into the “tokens” or terminal symbols to be parsed. The various LPG options (there are many) are described in another document.

Next there are two “sections”, each indicated by the “escape symbol” (“\$”) followed by a keyword. The first is “\$**Terminals**” and this section consists simply of a list of the terminal symbols. The other section is “\$**Rules**” which specifies the syntax in BNF rules. The non-terminal and terminal symbols in a rule may be named. The name “\$**ParenExpr**” in the above rule, “**F\$ParenExpr ::= (E)**”, names the generated AST rule class.

Generated Output

The LPG generator always produces a console output listing of the options used, error diagnostics and various grammar and state machine properties. If the language option is used, then parsing tables (in text or program format) are generated along with action code. For the simple expression parser (for which the language is Java) the following files are generated:

- **ExprParser.java** – the “action” file: a class with methods for invoking the runtime parser and the syntactic actions
- **ExprParsersym.java** – the “sym” file: an interface defining the terminal symbol representation
- **ExprParserprs.java** – the “prs” file: a class defining the state machine or parse tables used by the runtime parser

Since the automatic AST generation option is used, the AST classes and interfaces are also generated (typically within the action class, but not in this example).

Language Specific Runtime

LPG runtime support includes input and token stream management as well as stack based parsers. The Java runtime has two stream classes, called **LexStream** and **PrsStream**. The **LexStream** assumes the input to be parsed (or scanned) is a character array. There are methods to iterate through the array and to keep track of locations (offsets, line and column). The scanner (or “lexer” as we prefer to name it), whether it is LPG provided or hand coded, reads characters, classifies them and produces an array list of “tokens” (called a “token stream”) to be parsed.

An LPG lexer action class is typically an extension of **LexStream** that tokenizes the input character array according to grammar rules using a special runtime parser, **LexParser**. The generated token list is contained in a token stream class called **PrsStream**. An LPG

parser action class is normally an extension of `PrsStream` that parses the tokens according to syntax rules using one of the provided runtime parsers, the `DeterministicParser` or the `BacktrackingParser`. Lexers and parsers can detect and report errors. An LPG parser will normally raise an exception and terminate when the first error is encountered. However, the user may instruct the parser to invoke the `DiagnoseParser` to parse the entire input to diagnose all syntax errors.

To simplify building lexers and parsers, the LPG Java runtime also contains “templates” describing the user’s action class. A template is not a real class (or code) but a parameterized class fragment. Parameterization is achieved by macro definitions (much like the “define” macros in C). The simple expression example shown above uses a template, `dtParserTemplateD.g` to define its action class and method of invoking actions.

Finally, the LPG Java runtime contains some useful code sections (called “`$Header`” sections) that may be included in a lexer grammar file.

The LPG runtime is very language specific. At this time only the Java runtime has all of the features described here. For C/C++ there is only a deterministic parser and prs and sym files. We hope to expand the C/C++ runtime in the future and bring it to the same level as the Java runtime.

Putting the Pieces Together

Let’s use the simple expression grammar example to illustrate the process of building an LPG lexer and parser.

The Main Program

First we show a driver program that will invoke the lexer and parser in succession and “visit” the automatically generated AST.

```
package expr1;
import expr1.ExprAst.*;

public class Main
{
    public static void main(String[] args)
    {
        Option option;
        ExprLexer Expr_lexer;
        ExprParser Expr_parser;
        Ast ast;

        try
        {
            option = new Option(args);
            option.readInputChars();

            Expr_lexer = // Create the lexer
                new ExprLexer(option.getInputChars(), option.getFileName());
```

```

Expr_parser = // Create the parser
              new ExprParser(Expr_lexer);

// Lex the stream to produce the token stream
Expr_lexer.lex(Expr_parser);

// Parse the token stream to produce an AST
ast = Expr_parser.parser();
if (ast != null)
{
    Integer result = (Integer)ast.accept(new ExprResultVisitor());
    System.out.println("The value is : " + result.intValue());
}
return;
}
catch (Exception e)
{
    System.err.println(e.getMessage());
    e.printStackTrace();
}
}
}

```

There are several things to notice about this driver. First, the AST classes are in a separate imported package because the parser uses the `ast_directory` option. Next, an option class (not shown here) is used to read the input file so that the input character array and file name can be passed to the lexer constructor. A lexer and parser are created with the latter taking the former as an argument. The lexer is a subclass of `LexStream` and the parser is a subclass of `PrsStream`. These two objects are actually linked together when the “lexer” method is invoked. The result of lexing is the list of tokens which is known to the parser. The call on the “parser” method does the actual parsing and produces an AST object which is then “visited” to evaluate the expression. The visitor approach to traversing the AST will not be fully discussed in this document.

The Lexer

Next we examine the lexer. This is the grammar file:

```

%Options la=2
%options package=expr1
%options template=LexerTemplated.g
%options export_terminals=("ExprParsersym.java", "TK_")

$Export
IntegerLiteral PLUS MULTIPLY LPAREN RPAREN
$End

$Headers
/.
    final void makeToken(int kind)
    {
        int startOffset = getLeftSpan(), endOffset = getRightSpan();
        makeToken(startOffset, endOffset, kind);
    }

    public final int getKind(int i) // Classify character at ith location
    {

```

```

        char c = (i >= getStreamLength() ? '\uffff' : getCharValue(i));
        return (c < 33 ? Char_WSChar :
                c >= '0' && c <= '9' ? Char_Digit :
                c == '+' ? Char_Plus :
                c == '*' ? Char_Star :
                c == '(' ? Char_LeftParen :
                c == ')' ? Char_RightParen :
                c == '\uffff' ? Char_EOF :
                Char_Unused);
    }
    ./
$End

$Terminals
    WSChar Digit Unused EOF

    Plus      ::= '+'
    Star      ::= '*'
    LeftParen  ::= '('
    RightParen ::= ')'
$End

$Start
    Token
$End

$Rules
    Token ::= IntegerLiteral
        /.$BeginAction
            makeToken($_IntegerLiteral);
        $EndAction
    ./
    Token ::= '+'
        /.$BeginAction
            makeToken($_PLUS);
        $EndAction
    ./
    Token ::= '*'
        /.$BeginAction
            makeToken($_MULTIPLY);
        $EndAction
    ./
    Token ::= '('
        /.$BeginAction
            makeToken($_LPAREN);
        $EndAction
    ./
    Token ::= ')'
        /.$BeginAction
            makeToken($_RPAREN);
        $EndAction
    ./
    Token ::= WS -- White Space is scanned and ignored

    IntegerLiteral -> Digit
                    | IntegerLiteral Digit

    WS -> WSChar
        | WS WSChar
$End

```

The Options

Lexer generation is guided by the `LexerTemplated` template. This template has additional LPG options. These include:

```
%Options programming_language=java,margin=4
%options action=(*.java, "/.", "./")
%options ParseTable=lpg.lpgjavaruntime.ParseTable
%Options prefix=Char_
```

The “**action**” option indicates that the action class is named with the grammar file prefix and uses “/.” and “./” as action code delimiters. The “**ParseTable**” option indicates the location of the Java `ParseTable` interface (which specifies the methods the runtime parsers use to access the parse constants and table arrays). This option must be specified for every LPG lexer or parser. The “**prefix**” option specifies the prefix applied to terminal symbol names to ensure there will be no conflicts with Java keywords. It is required that terminal symbols be acceptable Java identifiers (since they just name integer constants in the symbol file) so the prefix may be necessary. Finally, the “**export_terminals**” option indicates the symbol file for the exported symbols and their prefix. A parser that imports this lexer need not use the same symbol file, but it must use the same prefix. It is recommended that the prefix “**TK_**” be used for exported symbols.

There are other important parts to the template that are discussed below.

The Export Section

The first section in this example lexer is an “**export**” section. The export section is signaled by the “**\$Export**” heading and ended by “**\$End**” or another section header. We recommend that all sections be terminated by a “**\$End**”. We note here that sections may appear in any order and a given section may appear more than once (multiple instances being concatenated together). The export section lists the terminal symbols that are exported to a parser. An export symbol file is created for these symbols (with the name and prefix specified in the “**export**” option, if given, or with the lexer grammar file prefix suffixed with “**ext**” and no token prefix, if there is no export option).

The Headers Section

The “**headers**” section provides additional action class methods that are not provided in the template. For most LPG lexers the two methods shown here are needed and one of them, the `getKind` method cannot easily be incorporated into a general purpose template. There are “**include**” files in the LPG distribution that may obviate the need for a special “**headers**” section in the lexer grammar file.

The first method in this section, `makeToken`, actually creates a token and adds it to the array list of tokens. To do this it invokes the corresponding method in `LexStream` passing it the starting and ending indexes of the token in the input character array and the token kind. The `LexStream` method, `makeToken`, refers to method with the same name in `PrsStream` and it is this method that constructs the token and adds it to the list. A `Token` class is defined in the LPG Java runtime. A token instance has offsets and a “**kind**”. The

token kind is just an integer that is defined in the symbol file. The runtime parser uses the token kind to determine its next move.

The other method, `getKind`, gets the character kind of the input character being lexed (as in the parser it gets the token kind of the token being parsed). In a lexer the “tokens” are the input characters. This method, then, classifies the input character as a lexer terminal symbol. This example is simple so if-then-else style logic is used. In more complex cases, one would use an array to map at least the basic ASCII characters to their kinds.

The Terminals Section

The next section is the “terminals” section, signaled by “`$Terminals`”. Note that the names are legal Java identifiers, but some of them, `plus` and `star`, for example, are often denoted by operator characters. Thus, in the rule defining the token `plus` we use the symbol ‘+’ which is an “alias” for the terminal symbol `plus`. This association can be specified using an “alias” section or by defining it in the “terminals” section as we have done above.

The Start Section

The “Start” section specifies the root or starting non-terminal. Rules that are not derivable from the start symbol are useless and have no effect.

In a lexer the start symbol stands for the class of tokens to be scanned. Usually tokens are simple and largely independent of each other. This is certainly the case in our example where the only recursive rules are for recognizing `IntegerLiteral` and white space. Thus, we use a special runtime parser, `LexParser`, in most lexers. The `LexParser` is simply a token recognition loop that resets to the start state as each token is recognized – it behaves just like a hand written scanner. This means that the start symbol describes the class of tokens, not a list of tokens. It is possible (and sometimes necessary) to use an actual parser (such as the `DeterministicParser`) and recognize a token list. Lexer construction will be discussed in a separate document on lexer construction.

The Rules Section

This section provides the token recognition rules for expressions. The tokens are the operators, parentheses, integer literals and white space. One should note how the actions are specified. No action is specified for white space so it is simply ignored. For the other tokens the action is to construct an exported token using the `makeToken` method described above. Note that to pass the exported token kind argument a special predefined macro, `$_`, is prefixed to the exported token name. The macro is replaced by the prefix specified for the exported symbol file, `tk_`, in our case.

Note also the macros `$BeginAction` and `$EndAction` surrounding the action code. These macros are defined in the lexer template and are specific to the implementation of rule actions. This is discussed in detail below.

The Action Class

The “lexer” from the user’s point of view is the object performing the token recognition actions. It is therefore necessary, but somewhat complicated, to define the action class. To facilitate this LPG provides lexer templates. Our example uses `LexerTemplated`. The template has a “headers” section that defines most (but not all, as noted below) of the action class and associated methods as follows:

```
$Headers
/.
public class $action_type
    extends $super_stream_class
    implements $exp_type, $sym_type, RuleAction$additional_interfaces
{
    private static ParseTable prs = new $prs_type();
    private $prs_stream_class prsStream;
    private LexParser lexParser = new LexParser(this, prs, this);

    public $prs_stream_class getPrsStream() { return prsStream; }

    public int getToken(int i) { return lexParser.getToken(i); }

    public int getRhsFirstTokenIndex(int i)
    { return lexParser.getFirstToken(i); }

    public int getRhsLastTokenIndex(int i)
    { return lexParser.getLastToken(i); }

    public int getLeftSpan() { return lexParser.getFirstToken(); }

    public int getRightSpan() { return lexParser.getLastToken(); }

    public $action_type(String filename, int tab)
        throws java.io.IOException
    { super(filename, tab); }

    public $action_type(char[] input_chars, String filename, int tab)
    { super(input_chars, filename, tab); }

    public $action_type(char[] input_chars, String filename)
    { this(input_chars, filename, 1); }

    public $action_type() {}

    public String[] orderedExportedSymbols()
    { return $exp_type.orderedTerminalSymbols; }

    public LexStream getLexStream() { return (LexStream) this; }

    public void lexer($prs_stream_class prsStream)
    { lexer(null, prsStream); }

    public void lexer(Monitor monitor, $prs_stream_class prsStream)
    {
        if (getInputChars() == null)
            throw new NullPointerException("LexStream not initialized");
        this.prsStream = prsStream;
        prsStream.makeToken(0, 0, 0); // first entry has a “bad” token
        lexParser.parseCharacters(monitor); // Lex the input characters
        int i = getStreamIndex();
        prsStream.makeToken(i, i, $eof_token); // last has end of file
    }
}
```

```

        prsStream.setStreamLength(prsStream.getSize());
    }
    return;
}
./
$End

```

This headers section is difficult to read (much less to understand) because of the number of macros it uses. Rather than delve into these macros let us see the Java class LPG has generated:

```

public class ExprLexer extends LpgLexStream
    implements ExprParsersym, ExprLexersym, RuleAction
{
    private static ParseTable prs = new ExprLexerprs();
    private PrsStream prsStream;
    private LexParser lexParser = new LexParser(this, prs, this);

    public PrsStream getPrsStream() { return prsStream; }

    public int getToken(int i) { return lexParser.getToken(i); }

    public int getRhsFirstTokenIndex(int i)
    { return lexParser.getFirstToken(i); }

    public int getRhsLastTokenIndex(int i)
    { return lexParser.getLastToken(i); }

    public int getLeftSpan() { return lexParser.getFirstToken(); }
    public int getRightSpan() { return lexParser.getLastToken(); }

    public ExprLexer(String filename, int tab) throws java.io.IOException
    { super(filename, tab); }

    public ExprLexer(char[] input_chars, String filename, int tab)
    { super(input_chars, filename, tab); }

    public ExprLexer(char[] input_chars, String filename)
    { this(input_chars, filename, 1); }

    public ExprLexer() {}

    public String[] orderedExportedSymbols()
    { return ExprParsersym.orderedTerminalSymbols; }

    public LexStream getLexStream() { return (LexStream) this; }

    public void lexer(PrsStream prsStream)
    { lexer(null, prsStream); }

    public void lexer(Monitor monitor, PrsStream prsStream)
    { if (getInputChars() == null)
        throw new NullPointerException("LexStream was not initialized");

        this.prsStream = prsStream;
        prsStream.makeToken(0, 0, 0); // first entry has a "bad" token
        lexParser.parseCharacters(monitor); // Lex the input characters

        int i = getStreamIndex();
        prsStream.makeToken(i, i, TK_EOF_TOKEN); // last has end of file
        prsStream.setStreamLength(prsStream.getSize());
    }
}

```

```

    return;
}

```

As we mentioned before the lexer action class, **ExprLexer**, extends **LexStream** (actually it extends **LpgLexStream**, which is an abstract subclass of **LexStream** that forces the method **getKind** to be defined). The class also implements three interfaces, **ExprParsersym**, **ExprLexersym** and **RuleAction**. The first two are the symbol interfaces for the lexer terminal symbols and the exported terminal symbols. The last is the rule action interface signifying that the action class implements **ruleAction** method used by the runtime parser (**LexParser** in this case) to invoke the appropriate action code whenever a rule is reduced (we describe this method below).

This class has three constructors with which you specify the input file, the character array and the tab value. The input file name may be given as the empty string, signifying that there isn't one. This name appears in default error messages and the parsing application can access it through the **LexStream** method **getFileName**. The tab value is used to ensure token starting and ending columns are correctly located in messages. It is not necessary to specify an input character array when creating a lexer object; one might create the lexer once and use it many times giving each time a new or modified character array using methods that are provided in **LexStream**.

The action class needs access to a parser and a parse stream. The parser needs access to the parse tables the lexer will use. Thus, three private variables are declared:

```

private static ParseTable prs = new ExprLexerprs();
private PrsStream prsStream;
private LexParser lexParser = new LexParser(this, prs, this);

```

The first is for the parse table class, **ExprLexerprs**, a class which implements the **ParseTable** interface. In order to create the token stream, a **PrsStream** is declared; this will be an argument to the **lexer** method. Finally, the runtime parser used is the **LexParser**. Note that its constructor takes **TokenStream**, **ParseTable** and **RuleAction** instances. The action class itself is the token stream (since it extends **LexStream**) as well as the rule action (since it implements **RuleAction**).

The **lexer** method calls the **LexParser** to parse the input character array and build the list of tokens. A **PrsStream**, which will hold the list of exported tokens, is the principal argument and is typically an instance of the parser's action class (since that class normally extends **PrsStream**). There is also a **Monitor** argument which is usually "null" unless one wants to limit the time allocated to lexing. Note: an unused ("zero") token starts the list and an end-of-file token ends the list. A place holder at the beginning of the list is needed by the parser.

We see several methods that may be invoked by the actions themselves: methods to get the **LexStream** and the **PrsStream** (in order to access their methods), and methods to get the span of terminals and rule right-hand sides in the input character array. We remarked above that the headers section in the lexer grammar file defined a **makeToken** method

that referenced the `makeToken` method of `LexStream`, which in turn referenced the one in `PrsStream`. This method could have referenced the `PrsStream` method directly as `prsStream.makeToken(...)`. The `getLeftSpan` and `getRightSpan` methods are used to find the left-most and right-most character locations spanned by a rule. In fact we see these methods used in the `makeToken` method discussed above. The other location methods are quite useful and are documented elsewhere.

The Rule Action Method

Finally, we must explain the connection between the runtime parser (the `LexParser`) and the user written rule actions. The parser invokes the `ruleAction` method, defined by the `RuleAction` interface and implemented by the action class, whenever a rule is reduced (that is, whenever its right-hand side has been parsed) passing the LPG assigned rule number as argument. LPG provides two rule action implementations in the templates: one uses a select statement and the other defines a class for each rule containing a method to be executed when the rule is reduced. The select implementation is simpler and for Java the more efficient one (however, for C++ the other is more efficient) so it is illustrated here. Simply by changing the template one can switch transparently from one to the other.

In the lexer Java code we find `ruleAction` as the last method listed. We shall first examine it and then relate it back to the template and the grammar file actions.

```
public void ruleAction( int ruleNumber)
{
    switch(ruleNumber) {
        // Rule 1:  Token ::= IntegerLiteral
        case 1: {
            makeToken(TK_IntegerLiteral);
            break;
        }
        // Rule 2:  Token ::= +
        case 2: {
            makeToken(TK_PLUS);
            break;
        }
        // Rule 3:  Token ::= *
        case 3: {
            makeToken(TK_MULTIPLY);
            break;
        }
        // Rule 4:  Token ::= (
        case 4: {
            makeToken(TK_LPAREN);
            break;
        }
        // Rule 5:  Token ::= )
        case 5: {
            makeToken(TK_RPAREN);
            break;
        }
    }
}
```

```

        default:
            break;
    }
    return;
}

```

The code is self explanatory, but how did it get there? In the template there is a rules section with a special macro **\$BeginActions**. This macro is defined in the “**\$Define**” section of the template as follows:

```

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

```

Since the template’s rules section is the first rules section the ruleAction method is started after the template’s headers code but before LPG emits any user defined actions. After all actions have been emitted, LPG emits the template’s “**\$Trailers**” section. This section has the macro **\$EndActions** which completes the ruleAction method. The trailers section also completes the action class declaration with a closing brace. The **\$EndActions** method is defined as follows:

```

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

```

The rules and trailers sections in the template are as follows:

```

$Rules
    /.$BeginActions./
$End

$Trailers
    /.
        $EndActions
    }
    ./
$End

```

Finally the actions themselves use two macros: **\$BeginAction** and **\$EndAction**. These are defined as follows:

```

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

$BeginAction /.$DefaultAction./

```

```

$EndAction
/.          break;
           }./

```

The `$DefaultAction` macro specifies a new case alternative.

The Parser

The parser, as noted above, is specified by the following grammar file:

```

%options ast_directory=./ExprAst,automatic_ast=toplevel,var=nt,visitor=default
%options programming_language=java
%options package=expr1
%options template=dtParserTemplated.g
%options import_terminals=ExprLexer.g

$Terminals
  IntegerLiteral
  PLUS ::= +
  MULTIPLY ::= *
  LPAREN ::= (
  RPAREN ::= )
$end

$Rules
  E ::= E + T
      | T
  T ::= T * F
      | F
  F ::= IntegerLiteral
  F$ParenExpr ::= ( E )
$End

```

The parser imports its terminal symbols from the lexer grammar file. In fact, when LPG compiles this parser grammar file, it also compiles the imported lexer grammar file. It is not necessary to import symbols from the lexer. Often the same lexer will be used by several different parsers, so having only one instance of the lexer is advantageous. If terminal symbols are not imported, then it is necessary to have a “terminals” section. In our example the terminal symbol section is redundant and can be omitted. We do not have a “start” section, so the first left-hand side symbol, `E`, is by default the start symbol. The AST is automatically generated so have no need to write actions (but we will examine the actions LPG generates).

The Parser Template

The template used here is the “deterministic” parsing template, `dtParserTemplated`. The LPG runtime has a traditional “deterministic” LALR parser, and a “backtracking” parser. The deterministic parser supports any amount of look-ahead, the amount of which is specified by the “`look_ahead`” option (the default is 1), while the backtracking parser uses only one token look-ahead but can backtrack on errors and pursue alternative parsing.

The Action Class

The deterministic parser template specifies the action class in much the same way as the lexer template does. Here is the start of the action class declaration (the declaration order has been changed to simplify our explanation):

```
public class $action_type extends PrsStream
    implements RuleAction$additional_interfaces
{
    private static ParseTable prs = new $prs_type();
    private DeterministicParser dtParser;

    public String[] orderedTerminalSymbols()
    { return $sym_type.orderedTerminalSymbols; }

    public $action_type(LexStream lexStream)
    {
        super(lexStream);

        try
        {
            super.remapTerminalSymbols(
                orderedTerminalSymbols(), $prs_type.EOFT_SYMBOL);
        }
        catch (NullExportedSymbolsException e) {
        }
        catch (NullTerminalSymbolsException e) {
        }
        catch (UnimplementedTerminalsException e)
        {
            java.util.ArrayList unimplemented_symbols = e.getSymbols();
            System.out.println
                ("The Lexer will not scan the following token(s):");
            for (int i = 0; i < unimplemented_symbols.size(); i++)
            {
                Integer id = (Integer) unimplemented_symbols.get(i);
                System.out.println
                    ("    " + $sym_type.orderedTerminalSymbols[id.intValue()]);
            }
            System.out.println();
        }
        catch (UndefinedEofSymbolException e)
        {
            throw new Error(new UndefinedEofSymbolException
                ("The Lexer does not implement the Eof symbol " +
                 $sym_type.orderedTerminalSymbols[$prs_type.EOFT_SYMBOL]));
        }
    }
}
```

Notice that the action class is named by the macro, `$action_type`, which is by default the file prefix (`ExprParser` in our case). The action class extends `PrsStream` and implements the `RuleAction` interface (just as we saw above with the lexer). There may be additional interfaces implemented by methods the user defines so the template provides a macro for specifying them.

A parse table instance (of the class defined in the “prs” file) is declared using the macro `$prs_type`, so that the tables need only be loaded once even the parse method (shown

below) is called many times. A private variable references the runtime deterministic parser.

There is one constructor for the class and it takes a `LexStream`, as its argument. As noted above in our discussion of the main program, the `LexStream` will be the lexer itself. The action class extends `PrsStream` whose constructor is called to link the `LexStream` and `PrsStream` together. The constructor then checks the validity of the terminal symbol file (given by the macro `$sym_type`) and remaps the terminal symbol values assigned by the lexer to the values defined in the parser's symbol file. In this way the parser will always use the correct symbol value, even when the lexer is independently constructed. It is possible that the parser has some terminal symbols that are not exported by the lexer and thus will never become tokens. This is not an error. The parser continues, though an exception is thrown and the symbols that cannot be scanned are listed on the console.

The Parser Method

Next we consider the “parser” methods. There is essentially one method but four different ways to invoke it:

```
public $ast_class parser()
{ return parser(null, 0); }

public $ast_class parser(Monitor monitor)
{ return parser(monitor, 0); }

public $ast_class parser(int error_repair_count)
{ return parser(null, error_repair_count); }

public $ast_class parser(Monitor monitor, int error_repair_count)
{
    try
    {
        dtParser = new DeterministicParser
            (monitor, (TokenStream)this, prs, (RuleAction)this);
    }
    catch (NotDeterministicParseTableException e)
    {
        throw new Error(new NotDeterministicParseTableException
            ("Regenerate $prs_type.java with -NOBACKTRACK option"));
    }
    catch (BadParseSymFileException e)
    {
        throw new Error(new BadParseSymFileException
            ("Bad Parser Symbol File" +
            " -- $sym_type.java. Regenerate $prs_type.java"));
    }

    try
    {
        return ($ast_class) dtParser.parse();
    }
    catch (BadParseException e)
    {
        reset(e.error_token); // point to error token
        DiagnoseParser diagnoseParser = new DiagnoseParser(this, prs);
        diagnoseParser.diagnose(e.error_token);
    }

    return null;
}
```

```
public DeterministicParser getParser() { return dtParser; }
```

The parser takes as arguments a “monitor” and an “error_repair_count”. The monitor gives you the ability to limit parser execution time. **Monitor** is actually an interface in the LPG Java runtime with a single Boolean method **isCancelled**, which the user may implement. It is not further discussed in this document. No monitor is specified by a “**null**” argument. The error count argument is used with “error productions”. These allow the parser to skip over specified syntactic constructs, for example “statement”, that contain errors without terminating the parse. The error count indicates how many times an erroneous construct may be skipped. At the present time the use of error productions is only supported by the backtracking parser (though it may be added to the deterministic parser in the future) and so this argument is simply ignored.

The body of the parser method creates a **DeterministicParser** giving it the **monitor**, the **TokenStream**, which is “**this**” instance of the action class (a subclass of **PrsStream**), the **ParseTable**, **prs**, and “**this**” as an instance of a class implementing the **RuleAction** interface. The parser checks that the parse tables are tables for a deterministic (not backtracking) parser and that the symbol file is valid. When LPG compiles a lexer that exports tokens, an export symbol file is generated. LPG always generates a symbol file for the parser’s terminal symbols. If these files are shared (that is, are the same file), then if the lexer is compiled after the parser has been compiled (to fix a lexer bug, say), the symbol file is no longer be valid and the **BadParseSymFileException** will be thrown.

Finally the deterministic parser’s **parse** method is called. Should a syntax error occur, the **BadParseException** exception is thrown and the diagnosing parser is called to find all syntax errors in the input token list. If there is an error, no syntactic actions are executed and hence no AST can be generated. Essentially the parser stops on the first error. If one prefers not to diagnose all errors, the template can be modified to take some other action, such as printing out the error token and quitting.

Token Access and Location Methods

The template defines several methods for accessing and locating tokens. These methods are used in the syntactic actions. Recall that the LPG Java runtime contains a **Token** class and an **IToken** interface. A token has a “kind” (a number LPG assigns to identify the token to the parser), start and end offsets (beginning and ending indexes of the token in the input character array), the index of the token in the token array list and an “adjunct” array index (while scanning tokens, comments or other text associated with a token may be placed in a token like array – there is a method, **makeAdjunct**, in **PrsStream** to construct the adjunct). There are methods in the **Token** class for obtaining this location information.

These are the methods provided in the template:

```
private void setResult(Object object) { dtParser.setSym1(object); }

public Object getRhsSym(int i) { return dtParser.getSym(i); }
```

```

public int getRhsTokenIndex(int i) { return dtParser.getToken(i); }
public IToken getRhsIToken(int i)
{ return super.getIToken(getRhsTokenIndex(i)); }

public int getRhsFirstTokenIndex(int i)
{ return dtParser.getFirstToken(i); }

public IToken getRhsFirstIToken(int i)
{ return super.getIToken(getRhsFirstTokenIndex(i)); }

public int getRhsLastTokenIndex(int i)
{ return dtParser.getLastToken(i); }

public IToken getRhsLastIToken(int i)
{ return super.getIToken(getRhsLastTokenIndex(i)); }

public int getLeftSpan() { return dtParser.getFirstToken(); }

public IToken getLeftIToken(){ return super.getIToken(getLeftSpan()); }

public int getRightSpan() { return dtParser.getLastToken(); }

public IToken getRightIToken()
{ return super.getIToken(getRightSpan()); }

```

The first method, `setResult` is used to place the result of the rule action on the runtime parser's "symbol" stack to represent the value of the left-hand side symbol. This result is then available when a rule with this symbol on its right-hand side is subsequently reduced and can be accessed using the method `getRhsSym`.

For example, here is what is generated for some of the rules of our expression grammar:

```

// Rule 3:  T ::= T * F
case 3: {
    setResult(
        new T(getLeftIToken(), getRightIToken(),
              (IT)getRhsSym(1),
              (IF)getRhsSym(3))
    );
    break;
}
// Rule 5:  F ::= IntegerLiteral
case 5: {
    setResult(new F(getRhsIToken(1)));
    break;
}

```

For rule 5 the result is a new instance of the class `F`. In rule 3, `F` is the third right-hand side symbol and the action for this rule accesses its value with the call `getRhsSym(3)`. The action for rule 3 creates an instance of `T` as its result. The method calls, `getLeftIToken()` and `getRightIToken()`, passed to the constructor in rule 3, retrieve the starting input character array index of the leftmost token and the ending index of the rightmost token spanned by the rule. These two operations are normally sufficient to obtain location information. Sometimes the right-hand side is a token, as in rule 5. In this case the generated AST passes to the rule class constructor the actual token itself using the method `getRhsIToken`.

Miscellaneous Methods

There are a few additional useful methods defined in the action class. There are methods to access the “error token” (which the user may choose to define), a method to get the grammar specified token name (e.g., `IntegerLiteral`, in our example) and a method to get the `PrsStream` Instance.

```
public int getRhsErrorTokenIndex(int i)
{
    int index = dtParser.getToken(i);
    IToken err = super.getIToken(index);
    return (err instanceof ErrorToken ? index : 0);
}
public ErrorToken getRhsErrorIToken(int i)
{
    int index = dtParser.getToken(i);
    IToken err = super.getIToken(index);
    return (ErrorToken) (err instanceof ErrorToken ? err : null);
}

public String getTokenKindName(int kind)
{ return $sym_type.orderedTerminalSymbols[kind]; }

public int getEOFTokenKind() { return $prs_type.EOFT_SYMBOL; }

public PrsStream getParseStream() { return (PrsStream) this; }
```

The Generated AST

For our expression example we let LPG generate the AST. This AST consists of classes representing the grammar rules and interfaces (or types) for the non-terminal symbols. For most rule reductions LPG constructs an instance of the rule class passing it right-hand side symbol values. A rule class extends one of three AST base classes and implements an interface determined by the left-hand side of the rule.

Two of the AST base classes are relevant to our example: `Ast` and `AstToken`. Rules with more than one symbol on the right-hand side generate a rule class that extends `Ast`. Rules having a single terminal symbol as their right-hand side generate a rule class that extends `AstToken`. Rules having a single non-terminal symbol as their right-hand side do not generate a rule class (however, the interface defined for their right-hand side symbol extends the interface defined for their left-hand side).

The Ast Class

Here is most of the `Ast` class (methods to get adjuncts and test for equality are omitted):

```
public abstract class Ast
{
    protected IToken leftIToken,
                  rightIToken;
    public IToken getLeftIToken() { return leftIToken; }
    public IToken getRightIToken() { return rightIToken; }
    public String toString()
    {
```

```

        PrsStream prsStream = leftIToken.getPrsStream();
        return new String(
            prsStream.getInputChars(),
            leftIToken.getStartOffset(),
            rightIToken.getEndOffset() - leftIToken.getStartOffset() + 1);
    }

    public Ast(IToken token)
    { this.leftIToken = this.rightIToken = token; }

    public Ast(IToken leftIToken, IToken rightIToken)
    {
        this.leftIToken = leftIToken;
        this.rightIToken = rightIToken;
    }
    void initialize() {}

    public abstract void accept(Visitor v);
    public abstract void accept(ArgumentVisitor v, Object o);
    public abstract Object accept(ResultVisitor v);
    public abstract Object accept(ResultArgumentVisitor v, Object o);
}

```

The attributes of the **Ast** class are left and right tokens. The constructor for a specific rule class passes to the **Ast** class constructor the left and right tokens spanned by its rule. The string of characters comprising this span can be obtained using the **toString** method. There is an **initialize** method which can be overwritten for a specific rule class by the user (we explain how this works in another document). Finally, there are four abstract methods for accepting a “visitor” to process rule class instances. This will be explained in more detail below where we discuss walking the AST.

The AstToken Class and Interface

The **AstToken** class is more like a rule class that focuses on tokens:

```

public class AstToken extends Ast implements IAstToken
{
    public AstToken(IToken token) { super(token); }

    public IToken getIToken() { return leftIToken; }

    public String toString() { return leftIToken.toString(); }

    public void accept(Visitor v) { v.visit(this); }
    public void accept(ArgumentVisitor v, Object o) { v.visit(this, o); }
    public Object accept(ResultVisitor v) { return v.visit(this); }
    public Object accept(ResultArgumentVisitor v, Object o)
    { return v.visit(this, o); }
}

```

Notice that this class extends **Ast**, as does a rule class, and implements **IAstToken**, just as a rule class implements an interface for its left-hand side. Terminal symbols (or tokens) have the same “type” or interface while non-terminal symbols generally have different types. The method **getIToken** retrieves the token and the **toString** method gives its textual content. As required by the **IAstToken** interface this class implements **accept** methods for the four kinds of visitor. The **IAstToken** interface also requires

implementation of getters for left and right tokens (to be compatible with non-terminal symbol interfaces) and is defined as follows:

```
public interface IAstToken
{
    public IToken getLeftIToken();
    public IToken getRightIToken();

    void accept(Visitor v);
    void accept(ArgumentVisitor v, Object o);
    Object accept(ResultVisitor v);
    Object accept(ResultArgumentVisitor v, Object o);
}
```

The Rule Classes and Interfaces

Now let us examine the classes and interfaces for the rules in our grammar. Consider first the rules with left-hand side symbol **F**. The first rule produces an integer literal while the second produces a parenthesized expression. The **F** interface, **IF**, is simply:

```
public interface IF extends IT, IAstToken {}
```

The first rule is the reason **F** extends **IAstToken**, while the rule, **T ::= F**, requires that **F** extend **IT**. The class for the first rule, “**F ::= IntegerLiteral**”, is (essentially) as follows:

```
public class F extends AstToken implements IF
{
    public F(IToken token) { super(token); initialize(); }

    public void accept(Visitor v) { v.visit(this); }
    public void accept(ArgumentVisitor v, Object o) { v.visit(this, o); }
    public Object accept(ResultVisitor v) { return v.visit(this); }
    public Object accept(ResultArgumentVisitor v, Object o)
    { return v.visit(this, o); }
}
```

For this rule LPG generates the action code, “**setResult(new F(getRhsIToken(1)));**”. As noted above, this creates an instance of **F** which is in fact an **AstToken**.

The class for the second rule with left-hand side **F**, “**F\$ParenExpr ::= (E)**”, is named **ParenExpr** and is defined as follows:

```
public class ParenExpr extends Ast implements IF
{
    private IE _E;

    public IE getE() { return _E; }

    public ParenExpr(IToken leftIToken, IToken rightIToken, IE _E)
    {
        super(leftIToken, rightIToken);
        this._E = _E;
        initialize();
    }
}
```

```

    public void accept(Visitor v) { v.visit(this); }
    public void accept(ArgumentVisitor v, Object o) { v.visit(this, o); }
    public Object accept(ResultVisitor v) { return v.visit(this); }
    public Object accept(ResultArgumentVisitor v, Object o)
    { return v.visit(this, o); }
}

```

This class has a single attribute, “**E**”, and a method to access it, “**getE()**”. Notice that an instance of this attribute may be an **E**, a **T** or an **F**, which is why the **IT** and **IF** interfaces extend **IE**.

LPG generates the following action code:

```

setResult(new ParenExpr(getLeftIToken(), getRightIToken(),(IE)getRhsSym(2)));

```

The left parenthesis is the left token and the right parentheses is the right token. For the second right-hand side symbol, “**E**”, previous reductions have stacked an object of type “**IE**” belonging to one of the classes **E**, **T**, **F**, or **ParenExpr**.

The only remaining rules of interest are “**E ::= E + T**” and “**T ::= T * F**”. The single productions “**E ::= T**” and “**T ::= F**” do not generate rule classes or actions but serve only to indicate subtypes of the expression type. Since these two rules are very much alike we will only consider the rule “**E ::= E + T**”.

The **IE** interface is essentially the same as the **IAstToken** interface shown above -- only the name is different. Interfaces provide strong typing for the rule class attributes.

The “**E**” rule class is as follows:

```

public class E extends Ast implements IE
{
    private IE _E;
    private IT _T;

    public IE getE() { return _E; }
    public IT getT() { return _T; }

    public E(IToken leftIToken, IToken rightIToken, IE _E, IT _T)
    {
        super(leftIToken, rightIToken);
        this._E = _E;
        this._T = _T;
        initialize();
    }

    public void accept(Visitor v) { v.visit(this); }
    public void accept(ArgumentVisitor v, Object o) { v.visit(this, o); }
    public Object accept(ResultVisitor v) { return v.visit(this); }
    public Object accept(ResultArgumentVisitor v, Object o)
    { return v.visit(this, o); }
}

```

This class has two attributes (for the expression and the term) and getters for them. The constructor is essentially the same as the one for parenthesized expression shown above. As with all rule classes there are methods to accept visitors.

The LPG generated action for this rule should not surprise us.

```
setResult(new E(getLeftIToken(), getRightIToken(),
               (IE)getRhsSym(1),
               (IT)getRhsSym(3))
);
```

Notice that the constructor is passed the token span of the rule and the expression and term instances that are to be added together. The plus operator is ignored since we have taken the option “`var=nt`”, which means that LPG should generate variables only for non-terminal symbols. If we wanted the operator token as an attribute of our class, then all we need to do is suffix the terminal symbol with a name (prefixed with a dollar sign).

The AST Visitor

The generated AST is a simplified syntax derivation tree. A tree node is an instance of a rule class and its children are its attributes that are rule instances or token instances. For example, given the input “`3 + 4 * 2`” we can visually display its AST as “`E[F[3], T[F[4], F[2]]]`”. The “`E`” node has two children, an “`F`” and a “`T`”; the “`T`” has two “`F`” children and the “`F`” nodes always have an integer literal child. (Please forgive the linear representation which encloses the children of a node in square brackets.) To process this tree we need to traverse the nodes and operate on their attributes. LPG supports the “visitor” paradigm for processing the AST. The classes and methods making up the visitor are completely independent of the AST. The visitor accesses the AST through node attribute methods and the AST communicates with the visitor through its accept methods.

The Generated Visitor Interfaces

LPG generates four visitor interfaces: `Visitor`, `ArgumentVisitor`, `ResultVisitor` and `ResultArgumentVisitor`. The simplest is the `Visitor` interface, which for our example is as follows:

```
public interface Visitor
{
    void visit(AstToken n);
    void visit(E n);
    void visit(T n);
    void visit(F n);
    void visit(ParenExpr n);
}
```

The “`visit`” method is overloaded by an argument for each AST class that is used. Each AST class has an “`accept`” method that invokes the corresponding visit method passing it “`this`” instance as argument, as we have seen above:

```
public void accept(Visitor v) { v.visit(this); }
```

Thus, while visiting one node, say “m”, should we want to visit a child node, say “n”, we simply tell the child to “accept” the visitor – “n.accept(v)”.

The **ArgumentVisitor** “visit” methods take an additional argument (an “Object”), which can be used for inherited attributes, while those of the **ResultVisitor** produce a result, which can be used for synthesized attributes. The **ResultArgumentVisitor** methods both take an argument and return a result.

The Generated Visitor Classes

In addition to the visitor interfaces, LPG generates two abstract visitor classes: the **AbstractVisitor** class that implements the **Visitor** and **ArgumentVisitor** interfaces and the **AbstractResultVisitor** class that implements the **ResultVisitor** and **ResultArgumentVisitor** interfaces.

Here is the **AbstractVisitor** class generated for our example:

```
public abstract class AbstractResultVisitor
    implements ResultVisitor, ResultArgumentVisitor
{
    public abstract Object unimplementedVisitor(String s);

    public Object visit(AstToken n)
    { return unimplementedVisitor("visit(AstToken)"); }

    public Object visit(AstToken n, Object o)
    { return unimplementedVisitor("visit(AstToken, Object)"); }

    public Object visit(E n) { return unimplementedVisitor("visit(E)"); }

    public Object visit(E n, Object o)
    { return unimplementedVisitor("visit(E, Object)"); }

    public Object visit(T n) { return unimplementedVisitor("visit(T)"); }

    public Object visit(T n, Object o)
    { return unimplementedVisitor("visit(T, Object)"); }

    public Object visit(F n) { return unimplementedVisitor("visit(F)"); }

    public Object visit(F n, Object o)
    { return unimplementedVisitor("visit(F, Object)"); }

    public Object visit(ParenExpr n)
    { return unimplementedVisitor("visit(ParenExpr)"); }

    public Object visit(ParenExpr n, Object o)
    { return unimplementedVisitor("visit(ParenExpr, Object)"); }
}
```

A visitor that extends this abstract class must implement the “unimplementedVisitor” abstract method. One implementation might simply be to ignore an unimplemented visit to a node. It might be better to provide a message or throw an exception when such a visitor is invoked.

The Expression Visitor

For our expression example we have implemented a result visitor which we call “**ExprResultVisitor**”. The purpose of the visitor is to “walk” the AST and evaluate the terms and expressions it contains.

In the main program the AST node returned from the parser accepts our visitor which evaluates the input expression.

```
Integer result = (Integer) ast.accept(new ExprResultVisitor());
```

The visitor class is as follows:

```
public class ExprResultVisitor extends AbstractResultVisitor
{
    public Object unimplementedVisitor(String s)
    {
        System.out.println(s);
        return null;
    }

    public Object visit(E expr)
    {
        Integer left = (Integer) expr.getE().accept(this),
            right = (Integer) expr.getT().accept(this);
        return new Integer(left.intValue() + right.intValue());
    }

    public Object visit(T expr)
    {
        Integer left = (Integer) expr.getT().accept(this),
            right = (Integer) expr.getF().accept(this);
        return new Integer(left.intValue() * right.intValue());
    }

    public Object visit(F expr)
    {
        return new Integer(expr.toString());
    }

    public Object visit(ParenExpr expr)
    {
        return (Integer) expr.getE().accept(this);
    }
}
```

To understand how this visitor works, let us follow its execution on the sample tree, “**E[F[3], T[F[4], F[2]]]**”, shown above. The top node “**E**”, which is visited by the accept call in the main program, has two children: an “**F**” node and a “**T**” node. The “**E**” accept method calls the “**visit(E expr)**” method of our visitor. This method retrieves the “**IE**” and the “**IT**” attributes, visits them in turn and adds the resulting values.

The “**IE**” attribute is actually an “**F**” instance so it is the “**visit(F expr)**” method that evaluates the “**IE**” child. The visitor converts the token text of “**F**” to an **Integer** object. Thus, an **Integer(3)** is returned.

The “**IT**” attribute is a “**T**” instance, so the “`visit(T expr)`” method is invoked to visit it. This method is quite similar to the “`visit(E expr)`” method, but the attributes are of type “**IT**” and “**IF**”, respectively, and the operation is multiplication. The “**IT**” attribute is an “**F**” instance which will be visited (as we saw above) and produce the value `Integer(4)`. The “**IF**” attribute is also an “**F**” instance and when visited produces the value `Integer(2)`. These two values are multiplied together to obtain an `Integer(8)` value. This is added to the `Integer(3)` value of the “**F**” instance to obtain the main program `result`, `Integer(11)`.

Conclusion

We have illustrated the use of LPG for syntactical and lexical analysis through a simple expression grammar example. We have shown the workings of the LPG generator and many features of its Java runtime. In particular, LPG options, grammar input, table output, lexer support, parser support and action class support (including automatic AST generation) have been described in sufficient (albeit incomplete) detail to enable you to build real parsing tools.