LPL: Log Processing Language

Language Reference Manual

Eugene Kozhukalo
genek81@gmail.com
COMS 4115
03/02/2006

1. Introduction

LPL is a simple, intuitive language for parsing and processing log files. It provides the user with flexibility to write a parser for and extract data from virtually any kind of log files. Syntactically, it is a mix of familiar Python/Java conventions with a few Awk constructs, which produces a combination of high-level convenience and simple yet powerful text processing facilities. ANTLR is used to specify its grammar, and Java is used to implement the grammar.

2. Lexical Conventions

2.1 Tokens and Whitespace

Parser *tokens* include identifiers, keywords, operators, literals, and other separators. Spaces, newlines and tabs constitute *white space*. The purpose of white space is to separate tokens, otherwise it is ignored.

2.2 Comments

Just like whitespace, *comments* are ignored by the parser. They are similar to Java. **Single line** comments begin with "//" and include all characters up to the end of the line. **Multi-line** comments begin with "/* and terminate with "*/".

2.3 Identifiers

An *identifier* in LPL consists of a letter character followed by any combination of letters, digits, and underscores. Uppercase and lowercase letters are considered different. Examples of identifiers: **a, abc123, test_string**.

2.4 Keywords

The following *keywords* are reserved in LPL:

begin	**elsif**	**max**
end	**input**	**min**
for	**datetime**	**avg**
in	**count**	**sum**
print	**if**	**type**
println	**else**	

true and **false** are also reserved words in LPL.

2.5 Literals

There are three types of *literals*: **integer**, **string**, and **boolean**.

An **integer** literal is a sequence of one or more digits beginning with an optional sign. An integer literal may not begin with a zero if it is more than one character long.

A **string** literal is a sequence of zero or more characters enclosed in double quotes. Single character strings may be enclosed in single quotes. Certain characters may be escaped with a backslash (\). These characters include single and double quotes, (\', \“) and whitespace characters (\n, \r, \t, \b).

A **boolean** literal is one of two reserved words, *true* or *false*.

3. Types

LPL has three *basic types* and three *user-defined types*. Basic types include **int**, **string** and **bool**. User-defined types include **list** and **dictionary**, which are implemented similar to Python, and **datetime**.

int, string and **bool** types are 32-bit integers, strings of characters, and *true*/*false* values, respectively. **list** and **dictionary** types can only contain basic types and datetime type. The **datetime** type is a special type containing a timestamp. The timestamp is parsed from a string containing time and/or date. This parsing is performed when a datetime object is initialized, and an optional regular expression may be specified to use during parsing.

Arithmetic and relational operations may only be performed on operands of the same type. Lists may be dereferenced only by integers. Dictionaries may contain key-value pairs of any of the basic and datetime types.

4. Operators

4.1 Logical

&& : logical *and* operator
|| : logical *or* operator
! : logical *not* operator

4.2 Arithmetical

+	:	binary addition operator and string/list append operator
-	:	binary subtraction operator
*	:	binary multiplication operator
/	:	binary division operator

4.3 Relational

==	:	relational *equals* operator
!=	:	relational *not equals* operator
>	:	relational *greater than* operator
>=	:	relational *greater than or equals* operator
<	:	relational *less than* operator
<=	:	relational *less than or equals* operator

4.4 Other

=	:	assignment operator
()	:	list/dictionary dereference, function call operators
.	:	method call operator

4.5 Operator precedence table

() .	**High precedence**
!	
*** /**	
+ -	
!= == < <= > >=	
&& ||	
=	**Low precedence**

5. Expressions

Expressions include identifiers, function calls, list and dictionary element access, and other expressions surrounded by “(“ and “)”

5.1 Identifiers

Identifiers are *left-value* expressions. They are bound to values resulting from evaluating *right-value* expressions.

5.2 Function calls

Function calls are *right-value* expressions. They consist of an identifier (one of reserved keywords) followed by a comma-separated list of arguments enclosed by “(“ and “)”.

5.3 Element access expressions

Element access expressions are *left-value* expressions. They consist of a list/dictionary identifier followed by an integer index enclosed in “(“ and “)”.

5.4 Arithmetic expressions

Arithmetic expressions consist of operands separated by arithmetic operators. Operands can be identifiers, integer literals, function calls, or element access expressions. Operands for “+” operator can also be string literals.

5.5 Relational expressions

Relational expressions consist of operands separated by relational operators. Operands can be identifiers, integer literals, function calls, or element access expressions.

5.6 Logical expressions

Logical expressions consist of operands separated by logical operators. Operands can be identifiers, relational expressions, or boolean literals.

6. Variable declaration and assignment

LPL, like Python, is not strongly typed. That means variable declarations are not needed. A correct type is automatically assigned at variable initialization. Variable assignments take the form of:

<left-value expression> = < expression>;

Examples:

```
i = 5;                  // i becomes an integer with a value of 5
name = “Joe”;           // name is a string with a value of “Joe”
l = ();                 // l is an empty list
d = {};                 // d is an empty dictionary
dt = datetime();        // dt is a datetime object reflecting current date and time
c = (1,2);              // c is a list containing integers 1 and 2.
ct = count(d);          // ct is an integer corresponding to number of elements in d
```

There are a few special variables defined only in scope of a *begin / end* loop (see below). They contain:

- parts of current parsed record matched by groups of current parsing string ($1 holds contents of first group, $2 of second, etc.)
- total number of records ($NR)

7. Statements

7.1 Looping

There are three *looping statements* provided. One is the **for / in** loop used to iterate over lists and dictionaries. The syntax is as follows:

```
for <element identifier> in <list/dictionary identifier> {
        <statement>
        ...
}
```

Another loop is the **while** loop. The syntax is as follows:

```
while (<expression>) {
        <statement>
        ...
}
```

Finally, there is the **begin / end** loop. It is used to iterate over records parsed from the input file. The syntax is as follows:

```
begin (<optional record parse string>) {
        <parse string 1> : {
                <statement>
                ...
        }
        <parse string 2> : {
                <statement>
                ...
        }
} end {
        <statement>
        ...
}
```

7.2 Conditional

Conditional statement is comprised of **if / elsif / else** keywords. The syntax is as follows:

```
if (<expression>) {
        <statement>
        ...
} elsif (<expression>) {
        <statement>
        ...
} else {
        <statement>
        ...
}
```

8. Built-in Functions

There are a few built-in functions to facilitate certain tasks in LPL. They are as follows:

input(path_to_file) – specifies input log file to be parsed
print(id) – prints contents of **id**
println(id) – prints contents of **id** plus newline
type(id) – prints the type of **id**

count(list) – returns number of members in **list**, which can be a list or a dictionary
max(list) – returns max value of members in **list**, which must be a list of integers
min(list) – returns min value of members in **list**, which must be a list of integers
avg(list) – returns average of members in **list**, which must be a list of integers
sum(list) – returns sum of members in **list**, which must be a list of integers

datetime(string) – returns a datetime object either parsed from optional **string** or created based on current date/time.

9. Sample Program

```
/*
  This short program parses a log file that is in following format:
 <timestamp> <java class> <message type> <message>
 For example:
 02/02/2006 12:14:55 Main.java INFO Starting program...
 02/02/2006 12:14:56 Main.java INFO Loading configuration file...
 02/02/2006 12:14:57 Main.java ERROR Configuration file not found! *#
*/

input("C:\logs\*.log") /*specify input fileset
                          input file(s) may also be passed in as command line arguments */

dateTimeRE = "\d+\/\d+\/\d+\s\d+:\d+:\d+"; //declaring a regular expression for datetime
classNameRE = "[A-Za-z]+\.java"; //declaring an RE for class name
msgTypeRE = "[A-Z]+"; //declaring an RE for message type
msgContentRE = "(\w+\s*)+"; //declaring an RE for message contents
singleLogMsgRE = "(" + dateTimeRE +
              ")\s+(" + classNameRE +
              ")\s+(" + msgTypeRE +
              ")\s+(" + msgContentRE + ")";  //declaring a single message RE

/* these regular expression variables are defined for convenience
 it will be possible to use the actual string represented by singleLogMsgRE
 inside the parse loop */

errorMsgList = (); //declaring an empty list

begin ("^.*\n") { /* parse loop, optional record pattern specified (single line here)
                     a record is also a single line by default
                     $1...$5 are matched groups defined in singleLogMsgRE */

        singleLogMsgRE: { #if a log message is found, do following actions
                          if ($2 == "Main.java" && $3 == "INFO")  {
                                print ("Time:" $1 "Message: " $5);
                          } else if (datetime($1) > datetime("02/02/06") {
                                errorMsgList = errorMsgList + $5; /* aggregate today's
                                                                     errors */
                          };
                  }

} end { //after parsing complete, do following
        println "Number of errors today: " + count(errorMsgList);
        println "Error messages:"
        println errorMsgList;
}
```