LPL: Log Processing Language

White Paper

Eugene Kozhukalo
genek81@gmail.com
COMS 4115
02/07/2006

LPL: Log Processing Language

Introduction

Log files are widely used to store information about program execution, errors, statistics, and other valuable data. This data is useful for system analysts, developers debugging their software, quality assurance analysts, and even software written for the purpose of mining data from log files and representing it in organized fashion, such as Webalizer (http://www.mrunix.net/webalizer). Webalizer is a web server log file analysis program written to parse Apache web server logs and produce highly detailed, easily configurable reports in HTML format. LPL's aim is similar, but with the added ability of writing a parser and extracting data for virtually any kind of log files. Essentially, one should be able to write Webalizer, or a completely different log parsing/information extraction and presentation package, in LPL. The goal is to implement a familiar syntax of Python/Java with a few Awk constructs, backed by the simple yet powerful text processing facilities of Awk. Below is a breakdown of the main goals, planned features, and implementation details for the project.

Goals

Intuitive

The language will employ intuitive and familiar constructs present in other commonly used languages such as Awk, Python, and Java. The syntax will resemble Java/Python syntax, while the semantics will be based on Awk execution model. This is to ensure that developers do not waste time learning the syntax and intricacies of the language, but are able to quickly write scripts to perform necessary tasks.

Flexible

One of the distinguishing features of the language will be the flexibility to parse virtually any type of log file. Programmers will be able to define precisely what data will be extracted utilizing the power of regular expressions.

Simple yet robust

LPL will have a limited set of types, commands, and constructs that should be easy to learn and utilize, yet sufficient for the purposes of the language. As mentioned before, the language will be similar to Awk in its structure, but will also include additional enhancements focusing on log file processing, such as multiline records (as opposed to single-line processing used in Awk), user-defined fields in records, and multi-file input.

Features

Interpreted

LPL will be interpreted, allowing for the addition of an interactive command line tool, which may be added to the language if time permits. The command line tool may be used to perform simple log processing operations or for interactive log processing. For more complex or repetitive tasks, an LPL program may be written and passed to the interpreter.

Portable

Since the interpreter will be written in Java, LPL's portability will be equivalent to that of Java. In essence, any system with an installed JVM will be able to run LPL programs without any additional requirements.

High-level

In addition to basic logical constructs, several high-level built-in functions will be provided, such as statistical/aggregation functions on lists and dictionaries. As the design of the language progresses, I anticipate discovering a necessity for certain high-level functions useful specifically in log processing. All built-in functions will further enhance and tailor the capabilities of LPL to the task of log processing.

Implementation Details

Interpretation and execution

ANTLR will be used to generate the lexer, parser, and tree walker Java classes from specified ANTLR grammar files. These generated classes, combined with backend classes (also written in Java) implementing the language functionality, will be used to translate an LPL program into Java code. The translated Java code will then be executed by the Java Virtual Machine.

Types

LPL will include 3 basic types: *int*, *float* and *string*. In addition, three user-defined types will be provided: *list*, *dictionary*, and *datetime*. *datetime* will be a flexible type, which can be initialized with a data string and an optional format specification string, enabling easy comparison of timestamps in various initial formats. The goal is to implement a simple, easy-to-use combination of Java's Date and DateFormat classes.

Variables and scope

LPL will not be strongly typed, therefore variable declarations will be similar to Python. For example, an integer could be declared as *x = 5*, a string as *s = "Hello Amber"*, a list as *l = []*, a datetime as *dt = datetime()*, etc. There will be certain special variables such as $0, $1, $NR, $NF, which will be used in the scope of a parse loop, similar to Awk's BEGIN/END loop. Since no user-defined function capabilities are planned, the scope of all declared variables will always be global, except for the special built-in variables mentioned above. This should help eliminate scoping issues from the language implementation.

Control flow and operators

Control flow statements will include *for* and *while* loops as well as the *if-then-else* statement. The syntax of these statements will be the same as or very similar to Java's. All standard logical and arithmetic operators present in Java should be implemented.

Comments

Comments will be similar to Python's, starting with a # symbol and terminating with a newline.

Possible enhancements

Time permitting, additional language extensions such as multi-format (HTML/XML/plaintext) file outputting and simple 2-D graphing may be implemented. User-defined functions and an interactive interpretation tool may also be considered as possible enhancements.

Sample Code

```
#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.append($5); #aggregate today's errors
                        };
                }

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