LPL: Log Processing Language

Final Report

Eugene Kozhukalo
genek81@gmail.com
COMS 4115
05/09/2006

Table of Contents

1. WHITEPAPER........5

1.1 Introduction........5
1.2 Goals........5
1.2.1 Intuitive........5
1.2.2 Flexible........5
1.2.3 Simple yet robust........6
1.3 Features........6
1.3.1 Interpreted........6
1.3.2 Portable........6
1.3.3 High-level........6
1.4 Implementation details........7
1.4.1 Interpretation and execution........7
1.4.2 Types........7
1.4.3 Variables and scope........7
1.4.4 Control flow and operators........7
1.4.5 Comments........8
1.5 Possible enhancements........8

2. TUTORIAL........10

3. LANGUAGE REFERENCE MANUAL........13

3.1 Lexical Conventions........13
3.1.1 Tokens and Whitespace........13
3.1.2 Comments........13
3.1.3 Identifiers........13
3.1.4 Keywords........13
3.1.5 Literals........14
3.2 Types........14
3.3 Operators........14
3.3.1 Logical........14
3.3.2 Arithmetical........15
3.3.3 Relational........15
3.3.4 Other........15
3.3.5 Operator precedence table........15
3.4 Expressions........16
3.4.1 Identifiers........16
3.4.2 Function calls........16
3.4.3 Element access expressions........16
3.4.4 Arithmetic expressions........16
3.4.5 Relational expressions........16
3.4.6 Logical expressions........16
3.5 Variable declaration and assignment........17
3.6 Statements........17
3.6.1 Looping........17
3.6.2 Conditional........18
3.7 Built-in Functions........19

4. PROJECT PLAN........20

4.1 Process planning........20
4.2 Programming style guide........20
4.2.1 ANTLR........20
4.2.2 Java........21
4.3 Project timeline........21
4.4 Software development environment........22

4.4.1 Java 22
4.4.2 Eclipse 22
4.4.3 ANTLR 22
4.4.4 CVS 22
4.5 Project log 23
5. ARCHITECTURAL DESIGN 24
5.1 System components 24
5.2 Component diagram 25
6. TEST PLAN 26
6.1 Automated testing framework overview 26
6.2 Sample test file 26
6.2 Sample input script and expected output 27
6.4 Sample test run – screenshot 28
7. LESSONS LEARNED 29
8. CODE LISTING 30
8.1 Grammar 30
8.1.1 src/lpl/grammar/LPLgrammar.g 30
8.1.2 src/lpl/grammar/LPLwalker.g 36
8.2 Main classes 40
8.2.1 src/lpl/LPLMain.java 40
8.2.2 src/lpl/LPLInterpreter.java 42
8.2.2 src/lpl/LPLBuiltinFunctions.java 48
8.2.2 src/lpl/symboltable/LPLSymbolTable.java 52
8.3 Data types 55
8.3.1 src/lpl/datatypes/LPLBaseType.java 55
8.3.2 src/lpl/datatypes/LPLBoolean.java 58
8.3.3 src/lpl/datatypes/LPLCollection.java 59
8.3.4 src/lpl/datatypes/LPLDatetime.java 59
8.3.5 src/lpl/datatypes/LPLDictionary.java 61
8.3.6 src/lpl/datatypes/LPLFunction.java 63
8.3.7 src/lpl/datatypes/LPLInt.java 64
8.3.8 src/lpl/datatypes/LPLList.java 66
8.3.9 src/lpl/datatypes/LPLMapPair.java 68
8.3.10 src/lpl/datatypes/LPLString.java 68
8.3.11 src/lpl/datatypes/LPLVariable.java 70
8.3.12 src/lpl/errors/LPLBaseException.java 70
8.4 Testing 71
8.4.1 src/lpl/testing/expected/basic_loops.txt 71
8.4.2 src/lpl/testing/expected/basic_types.txt 71
8.4.3 src/lpl/testing/expected/btype_operations.txt 71
8.4.4 src/lpl/testing/expected/builtin_functions.txt 71
8.4.5 src/lpl/testing/expected/complex_types.txt 72
8.4.6 src/lpl/testing/expected/conditionals.txt 72
8.4.7 src/lpl/testing/expected/ctype_operations.txt 72
8.4.8 src/lpl/testing/expected/integration_test1.txt 72
8.4.9 src/lpl/testing/expected/print_stmt.txt 73
8.4.10 src/lpl/testing/input/basic_loops.lpl 73
8.4.11 src/lpl/testing/input/basic_types.lpl 73
8.4.12 src/lpl/testing/input/btype_operations.lpl 74
8.4.13 src/lpl/testing/input/builtin_functions.lpl 74
8.4.14 src/lpl/testing/input/complex_types.lpl 75
8.4.15 src/lpl/testing/input/conditionals.lpl 75

8.4.16 src/lpl/testing/input/ctype_operations.lpl.. 76
8.4.17 src/lpl/testing/input/integration_test1.lpl.. 76
8.4.18 src/lpl/testing/input/print_stmt.lpl.. 77
8.4.19 src/lpl/testing/tests/AllTests.java.. 77
8.4.20 src/lpl/testing/tests/BasicTypesTest.java.. 78
8.4.21 src/lpl/testing/tests/ComplexTypesTest.java.. 78
8.4.22 src/lpl/testing/tests/IntegrationTest.java.. 79
8.4.23 src/lpl/testing/tests/StatementsTest.java.. 80
8.4.24 src/lpl/testing/util/LPLRunner.java.. 81

1. Whitepaper

1.1 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. It produces 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 of the LPL project is to implement a syntax familiar to 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.

1.2 Goals

1.2.1 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 syntax and intricacies of a new language, but rather are able to quickly write scripts to perform necessary tasks in LPL.

1.2.2 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 from any type of text log files utilizing the power of regular expressions.

1.2.3 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 multi-line records (as opposed to single-line processing used in Awk), user-defined fields in records, and multi-file input.

1.3 Features

1.3.1 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.

1.3.2 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 Java Virtual Machine will be able to run LPL programs without any additional requirements.

1.3.3 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 specifically useful in log processing. All built-in functions will further enhance and tailor the capabilities of LPL to the task of log processing.

1.4 Implementation details

1.4.1 Interpretation and execution

ANTLR will be used to generate the lexer, parser, and treewalker 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.

1.4.2 Types

LPL will include 4 basic types: *int*, *string, boolean,* and *datetime*. In addition, two container types will be provided: *list* and *dictionary. 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 SimpleDateFormat classes.

1.4.3 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, 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.

1.4.4 Control flow and operators

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

1.4.5 Comments

Comments will be similar to Java's, either starting with '//' and terminating with newline, or starting with '/*' and terminating with '*/'.

1.5 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.

1.6 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:\test\test.log"); //specify log file to be parsed

sep = "...";  //define record separator
infos = {}; //dictionary to hold INFO message information
errors = []; //list to hold ERROR messages

begin (sep) { //start parse loop, separating records with sep
    //in case of INFO, add to dictionary, key is first captured
    //group(timestamp), value is message(2nd group)
    "(?s)(?i)\s*([^\s]*\s+[^\s]*).*INFO\s+([^\n\r]*).*" : {
        infos[_1] = _2;
    }
    //in case of ERROR, add whole record to list
    "(?s)(?i)\s*([^\s]*\s+[^\s]*).*ERROR\s+([^\n\r]*).*" : {
        errors += _0;
    }
} end {
    //print number of records parsed
    print("Number of records: " + _NR);
}

//initialize reference date to calculate latest INFO message datetime
latestInfoDT=datetime("Jan 1, 2006 00:00:00 am");
print("INFO messages");
print("-------------");
//initialize latest INFO message variable
latestInfo = "";

for (date in infos) { //for each key(timestamp) in infos dictionary
    dt = datetime(date, "MM/dd/yyyy HH:mm:ss"); //parse datetime
    if (dt >= latestInfoDT) { //if this datetime latest so far
        latestInfoDT = dt; //make it new latest datetime
        latestInfo = infos[date]; //set latest INFO to corresponding
                                  //message
    }
    //print out each INFO timestamp and message
    print("Date: " + date + ",  Message: " + infos[date]);
}

//print latest INFO message and total number of ERROR messages
print("");
print("Latest INFO message: " + latestInfo +
    ",  Date: " + latestInfoDT);
print("Number of errors: " + count(errors));
```

2. Tutorial

Let's take a look at the short sample program from **Section 1.6** to become familiar with LPL. Although LPL can perform other basic programming tasks, the main goal of LPL is to parse log files, extract meaningful data from them, perform manipulations on that data, and present the results in an organized fashion. Let's see how the above program accomplishes these goals.

The first line of code is an **input** command.

```
input("C:\test\test.log");
```

This command is used to tell the interpreter the location of the log file that will be opened and parsed. The test.log file we'll be using looks like this:

```
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!
```

The goal of the above program is to print out all INFO messages, determine the latest INFO message, and determine the total number of ERROR messages in this file. Let's initialize some variables:

```
sep = "...";
infos = {};
errors = [];
```

Note that there is no need to specify the type at initialization – LPL determines it automatically based on the literal value being assigned to the variable. The first variable is a string variable, since the literal value is a string. It will be used as a parse separator in the **begin/end** loop that will come later in the program. The next variable is a dictionary that will hold INFO message timestamps as keys and map them to actual messages as values. The last variable is a list to hold ERROR messages.

Next in the code is the most important part of LPL – the parse loop(**begin/end** loop). It iterates over a list of records read from the input file and separated by an optional separator string (if no string is specified, the newline character is assumed as the separator), and performs one or more matches on each record, capturing data and specifying operations to perform in case of each match. Let's take a look at its usage:

```
begin (sep) {
      "(?s)(?i)\s*([^\s]*\s+[^\s]*).*INFO\s+([^\n\r]*).*" : {
            infos[_1] = _2;
      }
      "(?s)(?i)\s*([^\s]*\s+[^\s]*).*ERROR\s+([^\n\r]*).*" : {
            errors += _0;
      }


} end {
      print("Number of records: " + _NR);
}
```

To make things more interesting, we have opted to use a separator string *sep* (which is initialized to "..."). Therefore, the parse loop will iterate 3 times over following 3 records:

- 02/02/2006 12:14:55 Main.java INFO Starting program
- \n02/02/2006 12:14:56 Main.java INFO Loading configuration file
- \n02/02/2006 12:14:57 Main.java ERROR Configuration file not found!

In the body of the loop we have two parse "cases", which are string literals or string variables preceding the colon (:). They are identical to Java Pattern class regular expressions. To find out more about the intricacies of the syntax, please visit http://java.sun.com/j2se/1.4.2/docs/api/java/util/regex/Pattern.html. For our purposes, it is sufficient to say that the first pattern matches records with INFO in them, and the second one matches records with ERROR in them. We can see that there should be two INFO matches and one ERROR match.

Note the special variables *_0, _1, _2*, and *_NR* (used in the **end** clause). These are built-in variables that exist only in the scope of a parse loop. They are used to capture data from records using capture groups (see Java Pattern documentation for more information about capture groups). *_0* corresponds to the entire match, *_1* corresponds to the first capture group, *_2* to second, etc. *_NR* contains the number of parsed records. In our program, every time an INFO message is matched, its timestamp gets captured and put into the dictionary we initialized earlier as a key, and its actual message is captured and put into the dictionary as a corresponding value. Every time an ERROR message is matched, the entire match gets put into the pre-initialized list. In the **end** clause, we print out how many records we have processed.

After the parse loop finishes, we have parsed useful data from the log file and put it in the dictionary and the list. Now we can manipulate it to accomplish our goals. First, we will print out all INFO messages and determine the latest such message in the log file. To do that, we need to compare timestamps we put in the dictionary, and the easiest way to accomplish that is to convert them to datetime type - and then compare variables of that type. Let's initialize a couple of variables before using the **for** loop:

```
latestInfoDT=datetime("Jan 1, 2006 00:00:00 am");
...
latestInfo = "";
```

The first variable will contain the date and time of the latest INFO message. We have initialized it to a timestamp that is earlier than any timestamp in the log file so that we have a reference point for initial comparison. The second variable will contain the latest INFO message text, and is initialized to an empty string. Now, we will iterate over the infos dictionary using a **for** loop:

```
for (date in infos) {
      dt = datetime(date, "MM/dd/yyyy HH:mm:ss");
      if (dt >= latestInfoDT) {
            latestInfoDT = dt;
            latestInfo = infos[date];
      }
      print("Date: " + date + ",  Message: " + infos[date]);
}
```

The **for** loop iterates over lists and dictionary keys, capturing every iteration element in a specified variable(*date*). In our case, *date* variable will contain a timestamp string stored in the *infos* dictionary. The *dt* variable will be initialized to a datetime type, which will be parsed using a specified date format string on *date* variable. The date format string corresponds to a Java SimpleDateFormat class pattern (details available at http://java.sun.com/j2se/1.4.2/docs/api/java/text/SimpleDateFormat.html). Next, we compare the new *dt* datetime variable with the *latestInfoDT* datetime variable, and if the *dt* variable is greater (later), it becomes the new *latestInfoDT* variable, and the corresponding value from the *infos* dictionary becomes the *latestInfo* variable. We also print each *date* variable and the corresponding value from *infos* dictionary within the **for** loop.

All that's left to do is print our results, and the two **print** statements do just that:

```
print("Latest INFO message: " + latestInfo +
      ",  Date: " + latestInfoDT);
print("Number of errors: " + count(errors));
```

Note the **count** command in the second print statement. It is used to count the number of elements in a list or dictionary. We use it to count the total number of ERROR messages, which corresponds to the number of elements in errors list.

This tutorial demonstrates a limited number of commands and constructs available in LPL. For comprehensive information on all aspects of the language, please read the next section (Language Reference Manual).

3. Language reference manual

3.1 Lexical Conventions

3.1.1 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.

3.1.2 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 "*/", over one or more lines.

3.1.3 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**.

3.1.4 Keywords

The following *keywords* are reserved in LPL:

begin	**remove**	**max**
end	**input**	**min**
for	**datetime**	**avg**
in	**count**	**sum**
print	**if**	**type**
add	**else**	**what**
int	**bool**	

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

3.1.5 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*.

3.2 Types

LPL has four *basic types* and two *container types*. Basic types include **int**, **string, bool** and **datetime**. Container types include **list** and **dictionary**, which are implemented similar to Python.

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 with following exceptions: strings and all other basic types may be concatenated with a + operator, and any basic type as well as another list can be added to a list with a + operator. Lists may be dereferenced only by integers. Dictionaries may contain key-value pairs of any of the basic types.

3.3 Operators

3.3.1 Logical

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

3.3.2 Arithmetical

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

3.3.3 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

3.3.4 Other

=	:	assignment operator
[]	:	list/dictionary dereference operator
()	:	function call operator

3.3.5 Operator precedence table

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

3.4 *Expressions*

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

3.4.1 Identifiers

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

3.4.2 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 ")".

3.4.3 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 ")".

3.4.4 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.

3.4.5 Relational expressions

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

3.4.6 Logical expressions

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

3.5 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 type 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 the scope of a *begin / end* loop (see below). They contain:

- parts of the current parsed record matched by groups of the current parsing string (_1 holds contents of first group, _2 of second, etc.)
- total number of records (_NR)

3.6 Statements

3.6.1 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>
        ...
}
```

3.6.2 Conditional

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

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

3.7 Built-in Functions

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

Function signature	***Description***
input(path_to_file)	specifies input log file to be parsed
print(id)	prints contents of **id**
type(id)	prints the type of **id**
what(id)	prints a detailed description of **id**, or contents of symbol table at the moment if **id** is not given
count(list, list…)	returns total number of members in all passed paramters, which can be lists or dictionaries
max(list, list…)	returns max value among all parameters, which must be lists of integers
min(list, list…)	returns min value among all parameters, which must be lists of integers
avg(list, list…)	returns a string representation of the average of all elements in all parameters, which must be lists of integers
sum(list)	returns sum of elements in all parameters, which must be lists of integers
datetime(string)	returns a datetime object either parsed from optional string created based on current date/time
int(string)	converts a string argument to integer type
bool(string)	converts a string argument to boolean type

4. Project plan

4.1 Process planning

Deadlines and completion of functionality were structured mainly in response to the course schedule. Core specification and design decisions were made during the writing of the Language Reference Manual. Several modifications to these decisions were made throughout the development period. Testing, both automated and manual, was performed concurrently with development of different functional pieces. Functional tests were written for every major part of functionality (see **6. Test plan**) and were continuously run throughout the development phase.

4.2 Programming style guide

4.2.1 ANTLR

- The colon ":" always starts at the fifth column on a separate line under the lexeme/non-terminal name, unless the rule is short.
- The choice separator "|" is always placed at the fifth column on a separate line under the last ":" or "|".
- The semicolon ";" is always placed at the fifth column on a separate line under the last ":" or "|", unless the rule is short.
- If an ANTLR rule is short and contains only one choice without any action, then it will be written next to the colon and followed by a semicolon on the same line.
- If an ANTLR rule has multiple choice or actions, the ";" is placed in a separated line at the ninth column.
- Short actions are placed at the same line as the corresponding rule, at the right half of the screen.
- Long actions start at the ninth column of the next line, under the corresponding rule. Code in actions follows the Java coding style.
- Lexeme (token) names are in uppercase and syntactic items (non-terminals) are in lowercase and may contain underscore "_".

4.2.2 Java

- Indentation of each level is 4 spaces.
- Only spaces are used (no tabs).
- The left brace ”{“ is on the same line as corresponding statement.
- The right brace “}” in in the same column as the first character of corresponding statement.
- Comma-separated argument lists include spaces after the commas.
- No spaces between function names and argument lists, and between parentheses and arguments in argument lists.
- Spaces between operators and operands for outer expressions.
- Class names start with “LPL” followed by English words in camel-case.
- Variable and method names are in camel-case starting with a lowercase letter.

4.3 Project timeline

Task	***Date Due***
Initial idea and design	01/28/06
Whitepaper	02/07/06
Core grammar/syntax specifications	03/02/06
Language Reference Manual	03/02/06
ANTLR lexer and parser grammars	03/09/06
ANTLR tree walker (core functionality)	03/16/06
Coding complete	05/07/06
Testing complete	05/07/06
Final report	05/09/06

4.4 Software development environment

4.4.1 Java

All manual coding was done in Java version 1.5.0_02. All code generated by ANTLR was also Java. Java is an object-oriented programming language developed by James Gosling and colleagues at Sun Microsystems in the early 1990s. Unlike conventional languages that are generally designed to be compiled to native code, Java is compiled to bytecode which is then run (generally using just-in-time compilation) by a Java virtual machine. The language itself borrows much syntax from C and C++ but has a simpler object model and removes the need for low-level tools like programmer-manipulable pointers. For more information on Java, visit http://java.sun.com/.

4.4.2 Eclipse

Eclipse Integrated Development Environment (IDE) was used to facilitate the development process. The version used was 3.2. Eclipse is an open source community whose projects are focused on providing a vendor-neutral open development platform and application frameworks for building software. For more information, visit http://www.eclipse.org.

4.4.3 ANTLR

ANTLR version 2.7.6.1 was used to generate lexer, parser, and treewalker Java classes. ANTLR is a parser generator that uses LL(k) parsing. ANTLR stands for "ANother Tool for Language Recognition". Given that ANTLR is in competition with LR parser generators, the alternative reading "ANT(i)-LR" may not be accidental. For more information on ANTLR, refer to http://www.antlr.org/. For convenient integration of ANTLR with Eclipse, an Eclipse plugin called ANTLR Studio was used. For information on ANTLR Studio, refer to http://www.placidsystems.com/.

4.4.4 CVS

Concurrent Versions System(CVS) was used as a version control system for the project to ensure code integrity. CVS keeps track of all work and all changes in a set of files, typically the implementation of a software project, and allows several (potentially widely separated) developers to collaborate. Eclipse IDE contains a CVS integration, so no additional installation was necessary. A free CVS account was obtained from http://www.cvsdude.org. For more information on CVS, refer to http://www.nongnu.org/cvs/.

4.5 Project log

Task	***Completion date***
Whitepaper	02/07/06
CVS setup	02/10/06
Development environment setup (Eclipse, ANTLR)	02/15/06
Language Reference Manual	03/02/06
Core lexer/parser grammar	03/06/06
Data types hierarchy	03/13/06
Basic types	03/20/06
Arithmetic and logical operators	03/25/06
Assignment operation	03/29/06
print, **type** statements for basic types	03/29/06
List type	03/31/06
Testing framework, tests for basic types	04/01/06
Dictionary type, container types tests	04/02/06
if/else statement with corresponding tests	04/02/06
for/in loop with corresponding tests	04/09/06
Comparison of complex types, boolean operators	04/11/06
continue, break statements	04/14/06
count, max, min, avg, sum statements with corresponding tests	04/15/06
List and dictionary append operation, **input** and **what** statements	04/19/06
while loop with corresponding tests	04/22/06
begin loop with integration test, **int, bool** statements	05/06/06
Coding completed	05/07/06
Final report finished	05/08/06
Final report revised, edited, and submitted	05/09/06

5. Architectural design

5.1 System components

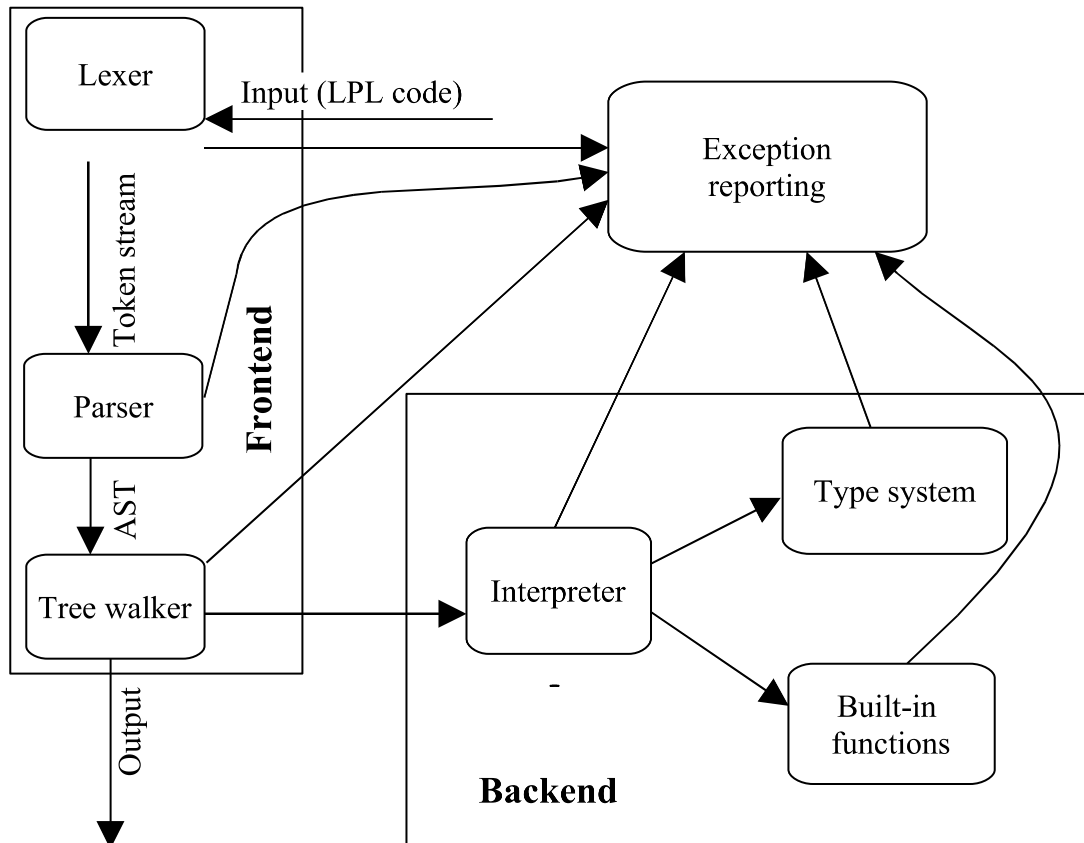
The Frontend consists of a Lexer, a Parser, and a Tree Walker. These are Java classes generated from ANTLR grammar files (LPLgrammar.g and LPLwalker.g). The Lexer takes in LPL code, removes whitespace and tokenizes the input. The token stream is input into the Parser, which analyzes the syntactic structure of the LPL program and generates an abstract syntax tree(AST). The AST is input into the Tree Walker, which performs various actions at various AST nodes, referring to the backend components as necessary.

The Backend consists of an Interpreter class, a Type System, and a Built-in Functions class. Then interpreter (LPLInterpreter.java) handles control flow for loops and conditionals, and also contains several helper functions. The Type System consists of a base type class (LPLBaseType.java), from which all other data types are derived. The data types include basic types such as integer, string, boolean, and datetime (LPLInt.java, LPLString.java, LPLBoolean.java, and LPLDatetime.java, respectively). Also, there are two container data types, list and dictionary (LPLList.java, and LPLDictionary.java), which can contain any of the basic types. The Built-in Functions class (LPLBuiltinFunctions.java) contains implementations of all LPL built-in functions.

Additional classes include LPLSymbolTable.java, which implements a symbol table to be used by the Interpreter in various scopes, and LPLMain.java, which is the main class for directing input/output and reporting errors and exceptions. Another class is LPLRunner.java, which is a utility class for executing automated tests.

5.2 Component diagram

The following diagram depicts the interaction between the components of the system:



6. Test plan

Testing was performed throughout the duration of the project. Functional testing was automated through a framework described below. Fault tolerance testing was mainly manual.

6.1 Automated testing framework overview

The automated testing framework functions in a very straightforward manner. It uses Java Unit testing framework (JUnit) to run suites of Java Unit tests. Each unit test contains several testcases which test different functionality by automatically executing short LPL scripts and comparing the output with an existing expected results database. The execution of LPL scripts and capture of output are performed by LPLRunner.java utility class, which starts the LPL interpreter in a separate Java process, executes a given LPL program file, and captures the output. The captured output represents the actual value in the testcase. The expected value comes from a database of text files. A specified text file is read in within the testcase, and its contents are compared to the actual value with an assertEquals statement.

Due to time constraints, the number of automated functional and integration tests is not as high as I would like it to be. However, existing tests cover most of basic functionality of the language, and it is very simple to create additional tests within the framework.

6.2 Sample test file

```
package lpl.testing.tests;

import java.io.IOException;

import junit.framework.TestCase;
import lpl.testing.util.LPLRunner;

public class StatementsTest extends TestCase {

   public static void main(String[] args) {
      junit.textui.TestRunner.run(StatementsTest.class);
   }

   public StatementsTest(String name) {
      super(name);
   }

   protected void setUp() throws Exception {
   }

   protected void tearDown() throws Exception {
```

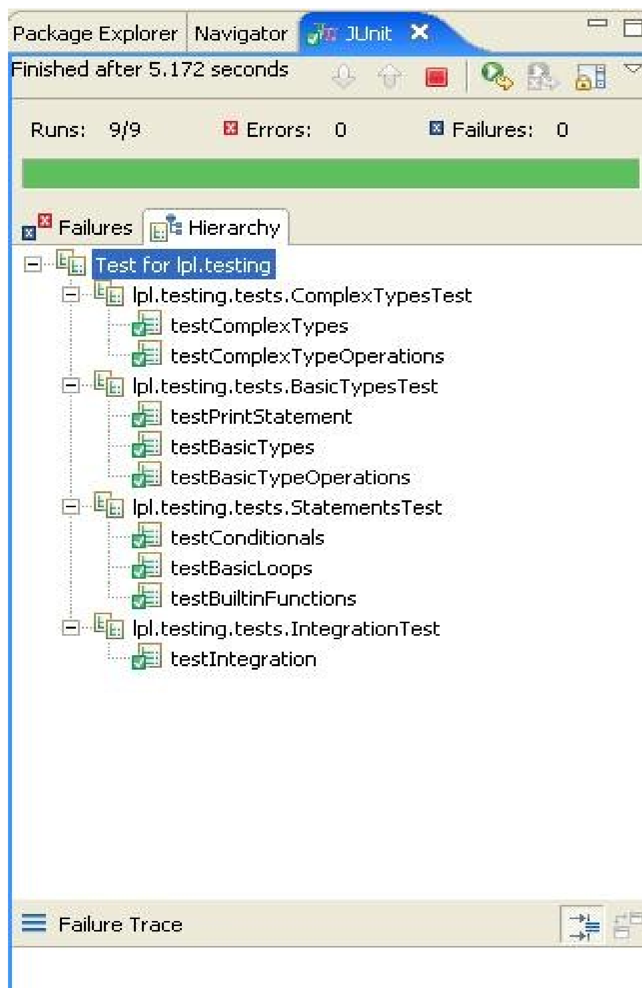
```
    }

    public void testConditionals() throws IOException {
        String expected = LPLRunner.getExpected("conditionals.txt");
        String actual = LPLRunner.runLPL("conditionals.lpl", true);
        assertEquals("Conditionals error:", expected, actual);
    }

    public void testBasicLoops() throws IOException {
        String expected = LPLRunner.getExpected("basic_loops.txt");
        String actual = LPLRunner.runLPL("basic_loops.lpl", true);
        assertEquals("Basic loops error:", expected, actual);
    }

    public void testBuiltinFunctions() throws IOException {
        String expected = LPLRunner.getExpected("builtin_functions.txt");
        String actual = LPLRunner.runLPL("builtin_functions.lpl", true);
        assertEquals("Builtin functions error:", expected, actual);
    }

}
```

6.2 Sample input script and expected output

```
/* basic_loops.lpl */

c=1;

while (c<5) {

    print(c);
    c+=1;

}

a = [1,2,3,4,5,6,7];

for (b in a) {

    if (b <= 4) {
        print(b);
    } else if (b > 4 && b < 6) {
        continue;
    } else {
        print(b);
        break;
    }

}

type(b);
```

```
/* basic_loops.txt */

c = 1
c = 2
c = 3
c = 4
b = 1
b = 2
b = 3
b = 4
b = 6
undefined-variable
```

6.4 Sample test run – screenshot



7. Lessons learned

- Good design is paramount. Planning and laying out as much of the core system functionality as possible before starting to code will greatly simplify the coding process. Determining data types, operations, and internal functions before starting to code is very important.

- Studying other language implementations can make both design and implementation simpler, as they may offer good ways to to implement similar functionality. It is also the best way to learn about tools such as ANTLR for someone who has never worked with it before. Parts of this project were based on the Mx language (© Tiantian Zhou, Hanhua Feng, Yong Man Ra, Chang Woo Lee), which made the basic infrastructure implementation quicker and easier. It also provided an excellent learning source.

- A comprehensive suite of automated tests can give the coder peace of mind when coding new functionality. Such a suite can always be run to check that the existing functionality has not been adversely affected by new code.

- Learning technology that is completely unfamiliar can be very time consuming. It is absolutely necessary to start this project as soon as possible. While I started early, I found myself spending more time than anticipated and required the full term to complete the language and report. From the start, I would advise to categorize planned features as essential and time-permitting.

- Verbal explanation can help solve mysterious errors. Explaining the purpose and functionality of code to another person, especially someone with minimal programming experience, often results in understanding a problem that has eluded resolution for some time.

8. Code listing

8.1 Grammar

8.1.1 src/lpl/grammar/LPLgrammar.g

```
 /*
 * @version $Id: LPLgrammar.g,v 1.15 2006/05/07 18:45:38 genek81 Exp $
 */

header {
   package lpl.grammar;

   import java.io.*;
}


/* LPL Lexer */

class LPLLexer extends Lexer;
options {
   k=2;
   testLiterals = false;
   exportVocab = LPL;
}

{
   public int nr_error = 0;
   public void reportError(String s) {
      super.reportError(s);
      nr_error++;
   }
   public void reportError(RecognitionException e) {
      super.reportError(e);
      nr_error++;
   }
}

protected
DIGIT   :   '0'..'9' ;

protected
LETTER   :   'a'..'z'|'A'..'Z' ;


DBL_QUOTE   :   '\"' ;
SGL_QUOTE   :   '\'' ;

EQ      :   "==" ;
```

```
NEQ       :   "!=" ;
LT        :   '<' ;
GT        :   '>' ;
GTE       :   ">=" ;
LTE       :   "<=" ;

ASSIGN    :   '=' ;
MINUS     :   '-' ;
PLUS      :   '+' ;
MULT      :   '*' ;
DIV       :   '/' ;
MOD       :   '%' ;
PLUSEQ    :   "+=" ;
MINUSEQ   :   "-=" ;
MULTEQ    :   "*=" ;
DIVEQ     :   "/=" ;
MODEQ     :   "%=" ;

LBRACKET  :   '[' ;
RBRACKET  :   ']' ;
RPAREN    :   ')' ;
LPAREN    :   '(' ;
RCURLY    :   '}' ;
LCURLY    :   '{' ;

OR        :   "||" ;
AND       :   "&&" ;
NOT       :   "!" ;

DOT       :   '.' ;
COMMA     :   ',' ;
DOLLAR    :   '$' ;
SEMI      :   ';' ;
COL       :   ':' ;


ID   options { testLiterals = true;}
   :   (LETTER|'_') (LETTER|DIGIT|'_')*
   ;


WS
   : (    ' '
   |    '\t'
   |    '\n'    { newline(); }
   |    '\r' ('\n')?    { newline(); }
   )
        {$setType(Token.SKIP);}
   ;


SL_COMMENT
   : "//"
        (~('\n'|'\r'))* (('\n'|'\r'('\n')?){newline();})?
        {$setType(Token.SKIP);}
   ;
```

```
ML_COMMENT
    : "/*" (
    options {
    greedy=false;
    generateAmbigWarnings=false;
}
    :   '\r' '\n' {newline();}
    |   '\r'  {newline();}
    |   '\n'     {newline();}
    |   ~('\n'|'\r')
    )*
    "*/"
            {$setType(Token.SKIP);}
    ;


CHAR_LITERAL
    : '\''! (ESC_CHARS|~'\'')* '\''!
    ;


STR_LITERAL
    : '\"'! (~'\"')* '\"'!
    ;


INT_LITERAL
    :    (DIGIT)+
    ;

protected
ESC_CHARS
    : '\\'
    (    'r'
    |    'n'
    |    'b'
    |    't'
    )
    ;



/* LPL Parser */

class LPLParser extends Parser;
options {
    exportVocab = LPL;
    k = 2;
    buildAST=true;
    defaultErrorHandler = true;
}

tokens {
   PROG;
   STATEMENT;
   LIST_VALS;
```

```
  DICT_VALS;
  LIST;
  DICT;
  UPLUS;
  UMINUS;
  BUILTIN_CALL;
  FOR;
  WHILE;
}

{
   public int nr_error = 0;
   public void reportError(String s) {
      super.reportError(s);
      nr_error++;
   }
   public void reportError(RecognitionException e) {
      super.reportError(e);
      nr_error++;
   }
}

program
   : (statement)* EOF!
      {#program = #([STATEMENT,"PROG"], program); }
   ;


// Statements

statement
   : for_stmt
   | begin_stmt
   | if_stmt
   | while_stmt
   | break_stmt
   | continue_stmt
   | builtin_stmt SEMI!
   | assignment
   | LCURLY! (statement)* RCURLY!
      {#statement = #([STATEMENT,"STATEMENT"], statement); }
   ;

builtin_stmt
   : ID LPAREN! list_vals RPAREN!
      {#builtin_stmt = #([BUILTIN_CALL,"BUILTIN_CALL"], builtin_stmt); }
   ;

while_stmt
   : "while"^ LPAREN! expression RPAREN! statement
   ;

break_stmt
   : "break"^ SEMI!
   ;

continue_stmt
```

```
    : "continue"^ SEMI!
    ;

begin_stmt
    : "begin"^ (parse_separator)?
        LCURLY! ( STR_LITERAL COL! statement )+ RCURLY!
        "end"! statement
    ;

protected
parse_separator
    : LPAREN ( ID|STR_LITERAL ) RPAREN!
    ;

for_stmt
    : "for"^ LPAREN! ID "in"! ID RPAREN! statement
    ;

if_stmt
    : "if"^ if_part
        ( options {greedy = true;}: "elsif"! if_part )*
        ( options {greedy = true;}: "else"! statement )?
    ;

protected
if_part
    : LPAREN! expression RPAREN! statement
    ;

assignment
    : l_value ( ASSIGN^ | PLUSEQ^ | MINUSEQ^ | MULTEQ^ | DIVEQ^ | MODEQ^ ) (expression)
SEMI!
    ;

cl_statement
    : ( statement )
    | "exit"
        { System.exit(0); }
    | EOF!
        { System.exit(0); }
    ;


//Expressions

expression
    : logic_term ( OR^ logic_term )*
    ;

logic_term
    : logic_factor ( AND^ logic_factor )*
    ;

logic_factor
    : (NOT^)? relat_expr
    ;
```

```
relat_expr
    : arith_expr ( (GTE^ | LTE^ | GT^ | LT^ | EQ^ | NEQ^) arith_expr )?
    ;

arith_expr
    : arith_term ( (PLUS^ | MINUS^) arith_term )*
    ;

arith_term
    : arith_factor ( (MULT^ | DIV^ | MOD^) arith_factor )*
    ;

arith_factor
    : PLUS! r_value
            {#arith_factor = #([UPLUS,"UPLUS"], arith_factor); }
    | MINUS! r_value
            {#arith_factor = #([UMINUS,"UMINUS"], arith_factor); }
    | r_value
    ;

r_value
    : l_value
    | literal
    | builtin_id
    | "true"
    | "false"
    | list
    | dict
    | builtin_stmt
    | LPAREN! expression RPAREN!
    ;

builtin_id
    : DOLLAR (INT_LITERAL | "NR")
    ;

l_value
    : ID^ ( LBRACKET! expression RBRACKET! )*
    ;

literal
    : INT_LITERAL
    | STR_LITERAL
    | CHAR_LITERAL
    ;

list_vals
    : expression (COMMA! expression)*
        {#list_vals = #([LIST_VALS,"LIST_VALS"], list_vals); }
        |   /*nothing */
        {#list_vals = #([LIST_VALS,"LIST_VALS"], list_vals); }
    ;

list
    : LBRACKET! list_vals RBRACKET!
        {#list = #([LIST,"LIST"], list); }
    ;
```

```
dict_vals
    : (expression COL! expression) (SEMI! (expression COL! expression))*
            {#dict_vals = #([DICT_VALS,"DICT_VALS"], dict_vals); }
        |   /*nothing */
        {#dict_vals = #([DICT_VALS,"DICT_VALS"], dict_vals); }
        ;

dict
    : LCURLY! dict_vals RCURLY!
        {#dict = #([DICT,"DICT"], dict); }
    ;
```

8.1.2 src/lpl/grammar/LPLwalker.g

```
/*
 * Partially adapted from Mx language
 *
 * @version $Id: LPLwalker.g,v 1.23 2006/05/07 18:45:38 genek81 Exp $
 */

header{
package lpl.grammar;

import java.io.*;
import java.util.*;
import lpl.datatypes.*;
import lpl.*;
}

/* LPL Tree Walker */

class LPLWalker extends TreeParser;
options{
    importVocab = LPL;
}

{
    public static LPLBaseType null_data = new LPLBaseType("<NULL>");
    public LPLInterpreter ipt = new LPLInterpreter();
}

expr returns [LPLBaseType r]
{
    LPLBaseType a, b, index = null;
    boolean isList = false, canProceed = false;
    int i = 0;
    ArrayList l=null;
    String s = null;
    String[] sx;
    r = null_data;
}
    : #(OR a=expr right_or:.)
        {
            if (a instanceof LPLBoolean)
```

```
          r = (((LPLBoolean)a).var ? a : expr(#right_or));
        else
          r = a.or(expr(#right_or));
    }
  | #(AND a=expr right_and:.)
    {
        if (a instanceof LPLBoolean)
          r = (((LPLBoolean)a).var ? expr(#right_and) : a);
        else
          r = a.and(expr(#right_and));
    }
  | #(NOT a=expr)          { r = a.not(); }
  | #(GTE a=expr b=expr)       { r = a.gte(b); }
  | #(LTE a=expr b=expr)       { r = a.lte(b); }
  | #(GT a=expr b=expr)       { r = a.gt(b); }
  | #(LT a=expr b=expr)       { r = a.lt(b); }
  | #(EQ a=expr b=expr)       { r = a.eq(b); }
  | #(NEQ a=expr b=expr)      { r = a.ne(b); }
  | #(PLUS a=expr b=expr)     { r = a.plus(b); }
  | #(MINUS a=expr b=expr)    { r = a.minus(b); }
  | #(MULT a=expr b=expr)     { r = a.mult(b); }
  | #(DIV a=expr b=expr)      { r = a.div(b); }
  | #(MOD a=expr b=expr)      { r = a.mod(b); }
  | #(UPLUS a=expr)          { r = a; }
  | #(UMINUS a=expr)         { r = a.uminus(); }
  | #(PLUSEQ a=expr b=expr)   { r = a.pluseq(b); }
  | #(MINUSEQ a=expr b=expr)  { r = a.minuseq(b); }
  | #(MULTEQ a=expr b=expr)   { r = a.multeq(b); }
  | #(DIVEQ a=expr b=expr)    { r = a.diveq(b); }
  | #(MODEQ a=expr b=expr)    { r = a.modeq(b); }
  | #(ASSIGN #(var:ID (index=expr {isList=true;})?) b=expr)
        {
        if(isList) {
          r = ipt.assign(var.getText(), index, b);
        } else {
          r = ipt.assign(var.getText(), null, b);
        }
    }

  | #(BUILTIN_CALL a=expr l=mexpr)   { r = ipt.funcInvoke(this, a, l); }
  | #(LIST l=mexpr)              { r = new LPLList(l) ; }
  | #(DICT l=mexpr)              { r = new LPLDictionary(l) ; }
  | num:INT_LITERAL              { r = ipt.getNumber(num.getText()); }
  | str:STR_LITERAL              { r = new LPLString(str.getText()); }
  | "true"                 { r = new LPLBoolean(true); }
  | "false"                { r = new LPLBoolean(false); }
  | #(id:ID                { r = ipt.getVariable(id.getText()); }
        ( a=expr           { r = ipt.getElement(r, a); }
      )*
    )
  | #("for" item_var:ID list_var:ID forbody:.)
    {
        l = ipt.forInit(item_var, list_var);
        while (ipt.forCanProceed(l, i)) {
              r = expr(#forbody);
              i++;
          ipt.forNext(item_var.getText(), l, i);
```

```
        }
        ipt.forEnd(l);
      }
   | #("while" whilecondition:. whilebody:STATEMENT)
      {
        canProceed = ipt.whileInit(expr(#whilecondition));
        while (canProceed) {
          r = expr(#whilebody);
          canProceed = ipt.whileNext(expr(#whilecondition));
        }
        ipt.whileEnd();
      }
   | #("begin" (LPAREN! (parseSepStr:STR_LITERAL|parseSepId:ID))?
        {
          String parseSeparator = null;
          HashMap actions = new HashMap();
          if (parseSepStr != null || parseSepId != null)
            parseSeparator = parseSepStr != null ?
              (String)parseSepStr.getText() :
              ipt.getVariable(parseSepId.getText()).toString();
          if (parseSeparator == null) parseSeparator = "\n";
          ipt.beginInit(parseSeparator);
        }
        (parseCaseStr:STR_LITERAL parseCaseStmt:STATEMENT {
           actions.put(parseCaseStr.getText(), parseCaseStmt);
         })+
         {
           ArrayList actionsToExecute = new ArrayList();
           while (ipt.getCanProceedBegin()) {
             actionsToExecute = ipt.beginNext(parseSeparator, actions);
             Iterator it = actionsToExecute.iterator();
             while(it.hasNext()) {
               r = expr((AST)it.next());
             }
         }
         }
         endStmt:STATEMENT {
           r = expr(#endStmt);
           ipt.beginEnd();
         })
   | #("if" a=expr thenp:. (elsep:.)?)
      {
        if (!(a instanceof LPLBoolean))
          return a.error("if: expression should be bool");
        if (((LPLBoolean)a).var)
          r = expr(#thenp);
        else if (null != elsep)
          r = expr(#elsep);
      }
   | #(STATEMENT (stmt:. { if (ipt.canProceed()) r = expr(#stmt); } )*)
   | "break"              { ipt.setBreak(); }
   | "continue"            { ipt.setContinue(); }
   ;

mexpr returns [ArrayList rl]
{
   LPLBaseType a, b;
```

```
    LPLMapPair p;
    rl = null;
}
    : #(LIST_VALS        { rl = new ArrayList(); }
          ( a=expr     { rl.add(a); }
         )*
      )
    | #(DICT_VALS              { rl = new ArrayList(); }
        (a=expr b=expr             { p = new LPLMapPair(a,b); rl.add(p); }
        )*
      )
    | a=expr            { rl = new ArrayList(); rl.add(a); }
    |  #(FOR        { rl = new ArrayList(); }
          ( s:ID a=expr { a.setName(s.getText()); rl.add(a); }
          )+
       )
    | /*nothing*/ { rl = new ArrayList(); }
    ;
```

8.2 Main classes

8.2.1 src/lpl/LPLMain.java

```
/**
 * Main class
 * Partially adapted from Mx language
 *
 * @version $Id: LPLMain.java,v 1.7 2006/04/30 04:21:23 genek81 Exp $
 */

package lpl;

import java.io.DataInputStream;
import java.io.FileInputStream;
import java.io.IOException;
import java.io.InputStream;

import lpl.datatypes.LPLBaseType;
import lpl.errors.LPLBaseException;
import lpl.grammar.LPLLexer;
import lpl.grammar.LPLParser;
import lpl.grammar.LPLWalker;
import antlr.CommonAST;
import antlr.RecognitionException;
import antlr.TokenStreamException;
import antlr.TokenStreamIOException;

class LPLMain {
    static boolean verbose = false;

    public static void execFile(String filename) {
        try {
            InputStream input = (null != filename) ?
                (InputStream) new FileInputStream(filename) :
                (InputStream) System.in;

            LPLLexer lexer = new LPLLexer(input);

            LPLParser parser = new LPLParser(lexer);

            parser.program();

            if (lexer.nr_error > 0 || parser.nr_error > 0) {
                System.err.println("Parsing errors found. Stop.");
                return;
            }

            CommonAST tree = (CommonAST)parser.getAST();

            if (verbose) {
                // Print the resulting tree out in LISP notation
```

```
            System.out.println(
               "================ tree structure ======================");
            System.out.println(tree.toStringList());
         }

         LPLWalker walker = new LPLWalker();
         // Traverse the tree created by the parser

         if (verbose)
            System.out.println(
               "================ program output ======================");

         LPLBaseType r = walker.expr( tree );

         if (verbose)
            System.out.println(
               "================ program return ======================");
         if (null != r)
            r.print();

         if (verbose) {
            System.out.println(
               "================ global variables ====================");
            walker.ipt.symt.what();
         }

      } catch(IOException e) {
         System.err.println("Error: I/O: " + e);
      } catch(RecognitionException e) {
         System.err.println("Error: Recognition: " + e);
      } catch(TokenStreamException e) {
         System.err.println("Error: Token stream: " + e);
      } catch(Exception e) {
         System.err.println("Error: " + e);
      }
   }

   public static void commandLine() {
      InputStream input = (InputStream) new DataInputStream(System.in);
      LPLWalker walker = new LPLWalker();

      for (;;) {
         try {
            while( input.available() > 0 )
               input.read();
         } catch ( IOException e ) {}

         System.out.print(">> ");
         System.out.flush();

         LPLLexer lexer = new LPLLexer(input);
         LPLParser parser = new LPLParser(lexer);

         try {
            parser.cl_statement();
            CommonAST tree = (CommonAST)parser.getAST();
            LPLBaseType r = walker.expr(tree);
```

```
            if (verbose && r != null)
               r.print();
         } catch(RecognitionException e) {
            System.err.println("Recognition exception: " + e );
         } catch(TokenStreamException e) {
            if (e instanceof TokenStreamIOException) {
               System.err.println("Token I/O exception");
               break;
            }
            System.err.println("Error: Token stream: " + e);
         } catch(LPLBaseException e) {
            System.err.println("Error: Interpretive: " + e);
            e.printStackTrace();
         } catch(RuntimeException e) {
            System.err.println("Error: Runtime: " + e);
            e.printStackTrace();
         } catch(Exception e) {
            System.err.println("Error: " + e);
            e.printStackTrace();
         }

      }
   }

   public static void main(String[] args) {

      verbose = args.length >= 1 && args[0].equals( "-v" );

      if (args.length >= 1 && args[args.length-1].charAt(0) != '-')
         execFile(args[args.length-1]);
      else
         commandLine();

      System.exit(0);
   }
}
```

8.2.2 src/lpl/LPLInterpreter.java

```
/**
 * Main interpreter class
 * Partially adapted from Mx language
 *
 * @version $Id: LPLInterpreter.java,v 1.18 2006/05/07 18:45:35 genek81 Exp $
 */

package lpl;

import java.io.BufferedReader;
import java.io.File;
import java.io.FileNotFoundException;
import java.io.FileReader;
import java.io.IOException;
import java.util.ArrayList;
import java.util.HashMap;
import java.util.Iterator;
```

```
import java.util.regex.Matcher;
import java.util.regex.Pattern;

import lpl.datatypes.LPLBaseType;
import lpl.datatypes.LPLBoolean;
import lpl.datatypes.LPLCollection;
import lpl.datatypes.LPLDictionary;
import lpl.datatypes.LPLFunction;
import lpl.datatypes.LPLInt;
import lpl.datatypes.LPLList;
import lpl.datatypes.LPLMapPair;
import lpl.datatypes.LPLString;
import lpl.datatypes.LPLVariable;
import lpl.errors.LPLBaseException;
import lpl.grammar.LPLWalker;
import lpl.symboltable.LPLSymbolTable;
import antlr.collections.AST;

public class LPLInterpreter {
   LPLSymbolTable symt;

   final static int fcNone = 0;
   final static int fcBreak = 1;
   final static int fcContinue = 2;
   final static int fcReturn = 3;

   private int control = fcNone;
   private String label;

   //for BEGIN loop
   private long _skipCount;
   private File _inputFile;
   private boolean _canProceedBegin = true;

   public LPLInterpreter() {
      symt = new LPLSymbolTable(null, null);
      registerInternal();
   }

   public static LPLBaseType getNumber(String s) {
      return new LPLInt(Integer.parseInt(s));
   }

   public LPLBaseType getVariable(String s) {
      LPLBaseType x = symt.getValue(s, true, 0);
      if (null == x)
         return new LPLVariable(s);
      return x;
   }

   public LPLBaseType getElement(LPLBaseType container, LPLBaseType index) {
      if (container instanceof LPLList) {
         if (index instanceof LPLInt)
            return ((LPLList)container).get(index);
         else
            return container.error(index, "invalid dereference type");
      } else if (container instanceof LPLDictionary) {
```

```
      return ((LPLDictionary)container).get(index);
   } else
      return container.error("cannot dereference this type");
}

public LPLBaseType rvalue(LPLBaseType a) {
   if (null == a.name)
      return a;
   return a.copy();
}

public LPLBaseType deepRvalue(LPLBaseType a) {
   if (null == a.name)
      return a;
   return a.copy();
}

public LPLBaseType assign(String varName, LPLBaseType index, LPLBaseType b) {
   LPLBaseType a = getVariable(varName);
   if ( null != a.name )
   {
      LPLBaseType x=null;
      if (index != null) {
         if (a instanceof LPLList) {
            ((LPLList)a).addAt(index, deepRvalue(b));
            x = a;
         } else if (a instanceof LPLDictionary) {
            ((LPLDictionary)a).add(index, deepRvalue (b));
            x = a;
         }
      } else {
         x = deepRvalue(b);
      }
      x.setName(a.name);
      symt.setValue(x.name, x, true, 0);
      return x;
   }

   return a.error(b, "=");
}

public LPLBaseType funcInvoke(LPLWalker walker, LPLBaseType func, ArrayList params )
   throws antlr.RecognitionException {

   // func must be an existing function
   if (!( func instanceof LPLFunction))
      return func.error("not a function");

   // Is this function an internal function?
   if (((LPLFunction)func).isInternal())
      return execInternal(((LPLFunction)func).getInternalId(), params);
   else
      return func.error("not a valid builtin function");
}

public void setBreak() {
   control = fcBreak;
```

```
}

public void setContinue() {
   control = fcContinue;
}

public void tryResetFlowControl(String loop_label) {
   if (null == label || label.equals(loop_label))
      control = fcNone;
}

public boolean whileInit(LPLBaseType result) {
   if (!(result instanceof LPLBoolean))
      throw new LPLBaseException("while: expression should be bool");
   //create a new symbol table
   symt = new LPLSymbolTable(symt, symt);
   return ((LPLBoolean)result).var;
}

public boolean whileNext(LPLBaseType result) {
   if (control == fcContinue)
      tryResetFlowControl(null);
   return ((LPLBoolean)result).var;
}

public void whileEnd() {
   if ( control == fcBreak )
      tryResetFlowControl(null);
   //remove this symbol table
   symt = symt.dynamicParent();
}

public boolean canProceed() {
   return control == fcNone;
}

public ArrayList forInit( AST item_var, AST list_var ) {
   LPLBaseType list_check = getVariable(list_var.getText());
   if (!(list_check instanceof LPLCollection)) {
      list_check.error("not a list or dictionary");
   }

   LPLBaseType item = new LPLVariable(item_var.getText());

   // create a new symbol table
   symt = new LPLSymbolTable(symt, symt);
   //set iterator value in symbol table
   LPLBaseType value = (LPLBaseType) ((LPLCollection)list_check).getVar().get(0);
   if (value instanceof LPLMapPair) value = ((LPLMapPair)value).key;
   assign(item.name, null, value);

   return ((LPLCollection)list_check).getVar();

}

public boolean forCanProceed(ArrayList list, int index) {
   if (control != fcNone)
```

```
        return false;

    return index < list.size();

}

public void forNext(String var_name, ArrayList list, int index) {

    //set iterator value in symbol table
    if (index < list.size()) {
        LPLBaseType value = (LPLBaseType) list.get(index);
        if (value instanceof LPLMapPair) value = ((LPLMapPair)value).key;
        assign(var_name, null, value);
    }

    if (control == fcContinue)
        tryResetFlowControl(((LPLBaseType)list.get(0)).name);
}

public void forEnd(ArrayList list) {
    if ( control == fcBreak )
        tryResetFlowControl(((LPLBaseType)list.get(0)).name);

    // remove this symbol table
    symt = symt.dynamicParent();
}

public void beginInit(String parseSeparator) {
    String filePath = ((LPLString)getVariable("INFILE")).var;
    if (filePath == "")
        throw new LPLBaseException("begin: no input file specified");
    _inputFile = new File(filePath);
    //create a new symbol table
    symt = new LPLSymbolTable(symt, symt);
    symt.setValue("_NR", new LPLInt(0), false, 0);
}

public ArrayList beginNext(String parseSeparator, HashMap actions) {
    String text = getTextToParse(_inputFile, parseSeparator);
    ArrayList actionsToExecute = new ArrayList();
    Iterator it = actions.keySet().iterator();
    while (it.hasNext()) {
        String parseString = (String)it.next();
        if (searchFor(parseString, text)) {
            actionsToExecute.add(actions.get(parseString));
        }
    }
    return actionsToExecute;
}

public void beginEnd() {
    _skipCount = 0;
    _canProceedBegin = true;
    symt = symt.dynamicParent();
}

private boolean searchFor(String parseString, String text) {
```

```
    text.trim();
    Pattern p = Pattern.compile(parseString);
    Matcher m = p.matcher(text);
    boolean matches = m.matches();
    if (matches) {
        symt.setValue("_0", new LPLString(m.group(0)), false, 0);
        for (int i=1; i <= m.groupCount(); i++) {
            symt.setValue("_" + i, new LPLString(m.group(i)), false, 0);
        }
    }
    return matches;
}

private String getTextToParse(File aFile, String parseSeparator) {
    StringBuffer contents = new StringBuffer();
    BufferedReader input = null;
    try {
        input = new BufferedReader(new FileReader(aFile));
        input.skip(_skipCount); //skip characters already read
        String line = null;
        while ((line = input.readLine()) != null) {
            if (parseSeparator == "\n") {
                contents.append(line);
                _skipCount++;
                break;
            }
            int psIndex = line.indexOf(parseSeparator);
            if (psIndex >= 0) {
                contents.append(line.substring(0, psIndex));
                break;
            }
            contents.append(line);
            contents.append(System.getProperty("line.separator"));
        }
        if (line == null) _canProceedBegin = false;
    } catch (FileNotFoundException ex) {
        throw new LPLBaseException("begin: input file not found");
    } catch (IOException ex) {
        throw new LPLBaseException("begin: IO Exception: " + ex);
    } finally {
        try {
            if (input != null) {
                input.close();
            }
        } catch (IOException ex) {
            throw new LPLBaseException("begin: IO Exception: " + ex);
        }
    }
     //add already read characters to skip count
    _skipCount += contents.length() + parseSeparator.length();
    //increase number of records counter variable
    ((LPLInt)symt.getValue("_NR", false, 0)).var++;
    return contents.toString();
}

public boolean getCanProceedBegin() {
    return _canProceedBegin;
```

```
    }

    public LPLBaseType execInternal(int id, ArrayList params) {
        return LPLBuiltinFunctions.run(symt, id, params);
    }

    public void registerInternal() {
        LPLBuiltinFunctions.register(symt);
    }

}
```

8.2.2 src/lpl/LPLBuiltinFunctions.java

```
/**
 * Implementation class for builtin functions
 * Partially adapted from Mx language
 *
 * @version $Id: LPLBuiltinFunctions.java,v 1.8 2006/05/07 18:45:35 genek81 Exp $
 */

package lpl;

import java.util.ArrayList;

import lpl.datatypes.LPLBaseType;
import lpl.datatypes.LPLBoolean;
import lpl.datatypes.LPLCollection;
import lpl.datatypes.LPLDatetime;
import lpl.datatypes.LPLFunction;
import lpl.datatypes.LPLInt;
import lpl.datatypes.LPLList;
import lpl.datatypes.LPLString;
import lpl.errors.LPLBaseException;
import lpl.symboltable.LPLSymbolTable;

public class LPLBuiltinFunctions {

    final static int fPrint = 0;
    final static int fInput = 1;
    final static int fType = 2;
    final static int fCount = 3;
    final static int fMax = 4;
    final static int fMin = 5;
    final static int fAvg = 6;
    final static int fSum = 7;
    final static int fDatetime = 8;
    final static int fWhat = 9;
    final static int fInt = 10;
    final static int fBool = 11;


    public static void register(LPLSymbolTable st) {

        st.put("print", new LPLFunction(null, fPrint));
        st.put("input", new LPLFunction(null, fInput));
```

```
    st.put("type", new LPLFunction(null, fType));
    st.put("count", new LPLFunction(null, fCount));
    st.put("max", new LPLFunction(null, fMax));
    st.put("min", new LPLFunction(null, fMin));
    st.put("avg", new LPLFunction(null, fAvg));
    st.put("sum", new LPLFunction(null, fSum));
    st.put("datetime", new LPLFunction(null, fDatetime));
    st.put("what", new LPLFunction(null, fWhat));
    st.put("int", new LPLFunction(null, fInt));
    st.put("bool", new LPLFunction(null, fBool));
    st.put("INFILE", new LPLString(""));
}

private static boolean isIntList(ArrayList params) {
    for (int i = 0; i < params.size(); i++)
        if (!(params.get(i) instanceof LPLInt))
            return false;
    return true;
}

private static int iterateMaxMin(ArrayList params, boolean isMax, int index) {
    String maxOrMin = isMax ? "max()" : "min()";
    if (params.get(index) instanceof LPLList) {
        int num = 0;
        LPLList currentParam = (LPLList) params.get(index);
        if (isIntList(currentParam.getVar())) {
            num = LPLInt.intValue(((LPLBaseType)currentParam.var.get(0)));
            for (int j = 1; j < currentParam.var.size(); j++) {
                int y = LPLInt.intValue(((LPLBaseType)currentParam.var.get(j)));
                if (isMax) {
                    if (y > num) num = y;
                } else {
                    if (y < num) num = y;
                }
            }
        } else {
            throw new LPLBaseException(maxOrMin + " parameters must be lists of integers");
        }
        return num;
    } else {
        throw new LPLBaseException(maxOrMin + " parameters must be lists");
    }
}

private static int getSum(LPLList currentParam) {
    int sum=0;
    for (int i=0; i<currentParam.getVar().size(); i++) {
        LPLInt currentElement = (LPLInt) currentParam.get(new LPLInt(i));
        sum += currentElement.var;
    }
    return sum;
}

public static LPLBaseType run(LPLSymbolTable st, int id, ArrayList params) {
    if (params == null) params = new ArrayList();
    switch (id) {
    case fPrint:
```

```
    for (int i = 0; i < params.size(); i++)
        ((LPLBaseType)params.get(i)).print();
    return null;

case fInput:
    if (!(params.size() == 1 && params.get(0) instanceof LPLString))
        throw new LPLBaseException("input() accepts 1 string parameter");
    st.put("INFILE", (LPLString)params.get(0));
    return null;

case fType:
    if (params.size() > 1) {
        ArrayList typeNames = new ArrayList();
        for (int i = 0; i < params.size(); i++)
            typeNames.add(((LPLBaseType)params.get(i)).typename());
        System.out.println(typeNames);
    } else {
        System.out.println(((LPLBaseType)params.get(0)).typename());
    }
    return null;

case fCount:
    if (params.size() == 0)
        throw new LPLBaseException("count() accepts 1+ parameters");
    int count = 0;
    for (int i=0; i<params.size(); i++) {
        if (params.get(i) instanceof LPLCollection) {
            count += ((LPLCollection)params.get(i)).getVar().size();
        } else {
            throw new LPLBaseException("count() parameters must be lists or dictionaries");
        }

    }
    return new LPLInt(count);

case fMax:
    if (params.size() == 0)
        throw new LPLBaseException("max() accepts 1+ parameters");
    int max = iterateMaxMin(params, true, 0);
    for (int i=1; i<params.size(); i++) {
        int temp = iterateMaxMin(params, true, i);
        max = max > temp ? max : temp;
    }
    return new LPLInt(max);


case fMin:
    if (params.size() == 0)
        throw new LPLBaseException("min() accepts 1+ parameters");
    int min = iterateMaxMin(params, false, 0);;
    for (int i=1; i<params.size(); i++) {
        int temp = iterateMaxMin(params, false, i);
        min = min < temp ? min : temp;
    }
    return new LPLInt(min);

case fAvg:
```

```
            if (params.size() == 0)
                throw new LPLBaseException("avg() accepts 1+ parameters");
            float fsum=0;
            float fcount=0;
            for (int i=0; i<params.size(); i++) {
                if (params.get(i) instanceof LPLList) {
                    LPLList currentParam = (LPLList) params.get(i);
                    if (isIntList(currentParam.getVar())) {
                        fsum += getSum(currentParam);
                        fcount += currentParam.getVar().size();
                    } else {
                        throw new LPLBaseException("avg() parameters must be lists of integers");
                    }
                } else {
                    throw new LPLBaseException("avg() parameters must be lists");
                }

            }
            return new LPLString(new Float(fsum/fcount).toString());

        case fSum:
            if (params.size() == 0)
                throw new LPLBaseException("sum() accepts 1+ parameters");
            int sum=0;
            for (int i=0; i<params.size(); i++) {
                if (params.get(i) instanceof LPLList) {
                    LPLList currentParam = (LPLList) params.get(i);
                    if (isIntList(currentParam.getVar())) {
                        sum += getSum(currentParam);
                    } else {
                        throw new LPLBaseException("sum() parameters must be lists of integers");
                    }
                } else {
                    throw new LPLBaseException("sum() parameters must be lists");
                }

            }
            return new LPLInt(sum);

        case fDatetime:
            if (params.size() > 2)
                throw new LPLBaseException("datetime() accepts 0 to 2 parameters");
            if (params.size() == 0)
                return new LPLDatetime(null, null);
            if (!(params.get(0) instanceof LPLString) || (params.size() == 2 && !(params.get(1)
instanceof LPLString)))
                throw new LPLBaseException("datetime() accepts 0 to 2 string parameters");
            if (params.size() == 1)
                return new LPLDatetime((LPLString)params.get(0), null);
            else
                return new LPLDatetime((LPLString)params.get(0), (LPLString)params.get(1));

        case fWhat:
            if (params.size() > 0) {
                for (int i = 0; i < params.size(); i++)
                    ((LPLBaseType)params.get(i)).what();
            } else
```

```
            st.what();
        return null;

    case fInt:
        if (params.size() != 1)
            throw new LPLBaseException("int() accepts 1 parameter");
        if (!(params.get(0) instanceof LPLString))
            throw new LPLBaseException("int() accepts only string parameters");
        return new LPLInt(Integer.parseInt(((LPLString)params.get(0)).var));

    case fBool:
        if (params.size() != 1)
            throw new LPLBaseException("bool() accepts 1 parameter");
        if (!(params.get(0) instanceof LPLString))
            throw new LPLBaseException("bool() accepts only string parameters");
        return new LPLBoolean(Boolean.parseBoolean(((LPLString)params.get(0)).var));

    default:
        throw new LPLBaseException("unknown internal function");
    }
  }

}
```

8.2.2 src/lpl/symboltable/LPLSymbolTable.java

```
/**
 * Symbol table class, implements a linked list of maps to store
 * identifiers and their values in various scopes
 * Partially adapted from Mx language
 *
 * @version $Id: LPLSymbolTable.java,v 1.6 2006/05/07 18:45:37 genek81 Exp $
 */

package lpl.symboltable;

import lpl.datatypes.*;

import java.io.PrintWriter;
import java.util.*;

public class LPLSymbolTable extends HashMap {
  LPLSymbolTable staticParent, dynamicParent;
  boolean readOnly;

  public LPLSymbolTable(LPLSymbolTable sparent, LPLSymbolTable dparent) {
    staticParent = sparent;
    dynamicParent = dparent;
    readOnly = false;
  }

  public void setReadOnly() {
    readOnly = true;
  }

  public final LPLSymbolTable staticParent() {
```

```
    return staticParent;
}

public final LPLSymbolTable dynamicParent() {
    return dynamicParent;
}

public final LPLSymbolTable parent(boolean is_static) {
    return is_static ? staticParent : dynamicParent;
}

public final boolean containsVar(String name) {
    return containsKey(name);
}

private final LPLSymbolTable gotoLevel(int level, boolean is_static) {
    LPLSymbolTable st = this;

    if (level < 0) {
        while (null != st.staticParent)
            st = st.parent(is_static);
    }
    else{
        for (int i=level; i>0; i--) {
            while (st.readOnly) {
                st = st.parent(is_static);
                assert st != null;
            }

            if ( null != st.parent(is_static))
                st = st.parent(is_static);
            else
                break;
        }
    }

    return st;
}

public final LPLBaseType getValue(String name, boolean is_static, int level) {
    LPLSymbolTable st = gotoLevel(level, is_static);
    Object x = st.get(name);

    while (null == x && null != st.parent(is_static)) {
        st = st.parent(is_static);
        x = st.get(name);
    }

    return (LPLBaseType) x;
}

public final void setValue(String name, LPLBaseType data,
                          boolean is_static, int level) {

    LPLSymbolTable st = gotoLevel(level, is_static);
    while (st.readOnly) {
        st = st.parent(is_static);
```

```
            assert st != null;
        }
        LPLBaseType currentVal = st.getValue(name, false, 0);
        LPLSymbolTable currentST=st, searchST = st;
        if (currentVal == null) {
            currentST.put(name, data);
        } else {
            Object x = searchST.get(name);
            while (null == x && null != searchST.parent(is_static)) {
             searchST = searchST.parent(is_static);
                x = searchST.get(name);
            }
            searchST.put(name, data);
        }
    }

    public void what(PrintWriter output) {
        for (Iterator it = values().iterator() ; it.hasNext();) {
            LPLBaseType d = ((LPLBaseType)(it.next()));
            if (!(d instanceof LPLFunction && ((LPLFunction)d).isInternal()))
                d.what(output);
        }
    }

    public void what() {
        what(new PrintWriter(System.out, true ));
    }
}
```

8.3 Data types

8.3.1 src/lpl/datatypes/LPLBaseType.java

```
/**
 * Base class for all types
 * Partially adapted from Mx language
 *
 * @version $Id: LPLBaseType.java,v 1.9 2006/04/30 04:21:22 genek81 Exp $
 */

package lpl.datatypes;

import java.io.PrintWriter;

import lpl.errors.LPLBaseException;

public class LPLBaseType
{
    public String name;

    public LPLBaseType() {
        name = null;
    }

    public LPLBaseType(String name) {
        this.name = name;
    }

    public String typename() {
        return "unknown";
    }

    public LPLBaseType copy() {
        return new LPLBaseType();
    }

    public void setName(String name) {
        this.name = name;
    }

    public LPLBaseType error(String msg) {
        throw new LPLBaseException("illegal operation: " + msg
                              + "( <" + typename() + "> "
                              + (name != null ? name : "<?>")
                              + " )");
    }

    public LPLBaseType error(LPLBaseType b, String msg) {
        if (null == b)
            return error(msg);
        throw new LPLBaseException(
```

```
        "illegal operation: " + msg
        + "( <" + typename() + "> "
        + (name != null ? name : "<?>")
        + " and "
        + "<" + b.typename() + "> "
        + (b.name != null ? b.name : "<?>")
        + " )");
}

public void print(PrintWriter w) {
    if (name != null)
        w.print(name + " = ");
    w.println(toString());
}

public void print() {
    print(new PrintWriter(System.out, true));
}

public String toString() {
    return "<undefined>";
}

public void what(PrintWriter w) {
    w.print("<" + typename() + ">  ");
    print(w);
}

public void what() {
    what(new PrintWriter(System.out, true));
}

public LPLBaseType assign(LPLBaseType b) {
    return error(b, "=");
}

public LPLBaseType uminus() {
    return error("-");
}

public LPLBaseType plus(LPLBaseType b) {
    return error(b, "+");
}

public LPLBaseType pluseq(LPLBaseType b) {
    return error(b, "+=");
}

public LPLBaseType minus(LPLBaseType b) {
    return error(b, "-");
}

public LPLBaseType minuseq(LPLBaseType b) {
    return error(b, "-=");
}

public LPLBaseType mult(LPLBaseType b) {
```

```
    return error( b, "*" );
}

public LPLBaseType multeq(LPLBaseType b) {
    return error( b, "*=" );
}

public LPLBaseType div(LPLBaseType b) {
    return error(b, "/");
}

public LPLBaseType diveq(LPLBaseType b) {
    return error(b, "/=");
}

public LPLBaseType mod(LPLBaseType b) {
    return error(b, "%");
}

public LPLBaseType modeq(LPLBaseType b) {
    return error(b, "%=");
}

public LPLBaseType gt(LPLBaseType b) {
    return error(b, ">");
}

public LPLBaseType gte(LPLBaseType b) {
    return error(b, ">=");
}

public LPLBaseType lt(LPLBaseType b) {
    return error(b, "<");
}

public LPLBaseType lte(LPLBaseType b) {
    return error( b, "<=" );
}

public LPLBaseType eq(LPLBaseType b) {
    return error(b, "==");
}

public LPLBaseType ne(LPLBaseType b) {
    return error(b, "!=");
}

public LPLBaseType and(LPLBaseType b) {
    return error( b, "&&" );
}

public LPLBaseType or(LPLBaseType b) {
    return error(b, "||");
}

public LPLBaseType not() {
    return error("!");
```

```
    }
}
```

8.3.2 src/lpl/datatypes/LPLBoolean.java

```
/**
 * Boolean data type
 * Partially adapted from Mx language
 *
 * @version $Id: LPLBoolean.java,v 1.6 2006/04/30 04:21:22 genek81 Exp $
 */

package lpl.datatypes;


public class LPLBoolean extends LPLBaseType {

    public boolean var;

    public LPLBoolean(boolean var) {
        this.var = var;
    }

    public String typename() {
        return "boolean";
    }

    public LPLBaseType copy() {
        return new LPLBoolean(var);
    }

    public String toString() {
        return Boolean.toString(var);
    }

    public boolean equals(Object o) {
        return ((LPLBoolean)o).var == var;
    }

    public LPLBaseType and(LPLBaseType b) {
        if (b instanceof LPLBoolean)
            return new LPLBoolean(var && ((LPLBoolean) b).var);
        return error(b, "and");
    }

    public LPLBaseType or(LPLBaseType b) {
        if (b instanceof LPLBoolean)
            return new LPLBoolean(var || ((LPLBoolean) b).var);
        return error(b, "or");
    }

    public LPLBaseType not() {
        return new LPLBoolean(!var);
    }

    public LPLBaseType eq(LPLBaseType b) {
```

```
        if (b instanceof LPLBoolean)
            return new LPLBoolean((var && ((LPLBoolean) b).var)
                    || (!var && !((LPLBoolean) b).var));
        return error(b, "==");
    }

    public LPLBaseType ne(LPLBaseType b) {
        if (b instanceof LPLBoolean)
            return new LPLBoolean((var && !((LPLBoolean) b).var)
                    || (!var && ((LPLBoolean) b).var));
        return error(b, "!=");
    }

}
```

8.3.3 src/lpl/datatypes/LPLCollection.java

```
/**
 * Collection interface for List and Dictionary data types
 *
 * @version $Id: LPLCollection.java,v 1.2 2006/04/30 04:21:22 genek81 Exp $
 */

package lpl.datatypes;

import java.util.ArrayList;

public interface LPLCollection {

    public ArrayList getVar();

    public LPLBaseType get(LPLBaseType item);

    public LPLBaseType remove(LPLBaseType item);

}
```

8.3.4 src/lpl/datatypes/LPLDatetime.java

```
/**
 * Datetime data type
 * A special type containing a timestamp.
 * The timestamp is parsed from a string containing time and/or date.
 *
 * @version $Id: LPLDatetime.java,v 1.7 2006/05/07 02:17:04 genek81 Exp $
 */

package lpl.datatypes;

import java.text.ParseException;
import java.text.SimpleDateFormat;
import java.util.Date;

public class LPLDatetime extends LPLBaseType {
    public Date var;
```

```
    public LPLDatetime(LPLString dt, LPLString pattern) {
        SimpleDateFormat df = (SimpleDateFormat)SimpleDateFormat.getDateTimeInstance();
        if (dt == null) {
            var = new Date();
        } else {
            if (pattern != null)
                df.applyPattern(pattern.var);
            try  {
                var = df.parse(dt.var);
            } catch (ParseException pex) {
                this.error("exception while parsing");
            }
        }
    }

    public String typename() {
        return "datetime";
    }

    public LPLBaseType copy() {
        return new LPLDatetime(new LPLString(var.toString()), new LPLString("EEE MMM dd
hh:mm:ss zzz yyyy"));
    }

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

    public LPLBaseType gt(LPLBaseType b) {
        return new LPLBoolean(var.after(((LPLDatetime)b).var));
    }

    public LPLBaseType gte(LPLBaseType b) {
        return new LPLBoolean(var.after(((LPLDatetime) b).var)
                || var.equals(((LPLDatetime) b).var));
    }

    public LPLBaseType lt(LPLBaseType b) {
        return new LPLBoolean(var.before(((LPLDatetime)b).var));
    }

    public LPLBaseType lte(LPLBaseType b) {
        return new LPLBoolean(var.before(((LPLDatetime) b).var)
                || var.equals(((LPLDatetime) b).var));
    }

    public LPLBaseType eq(LPLBaseType b) {
        return new LPLBoolean(var.equals(((LPLDatetime) b).var));
    }

    public LPLBaseType ne(LPLBaseType b) {
        return new LPLBoolean(!var.equals(((LPLDatetime) b).var));
    }
}
```

8.3.5 src/lpl/datatypes/LPLDictionary.java

```
/**
 * Dictionary data type, basically a list of pairs of basic types.
 *
 * @version $Id: LPLDictionary.java,v 1.10 2006/04/30 04:21:22 genek81 Exp $
 */

package lpl.datatypes;

import java.util.ArrayList;
import java.util.Iterator;

public class LPLDictionary extends LPLBaseType implements LPLCollection {

   public ArrayList var; //list of LPLMapPairs

   public LPLDictionary(ArrayList list) {
      this.var = list;
   }

   public String typename() {
      return "dictionary";
   }

   public LPLBaseType copy() {
      return new LPLDictionary(var);
   }

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

   public LPLBaseType add(LPLBaseType b, LPLBaseType c) {
      if (!(b instanceof LPLInt || b instanceof LPLBoolean || b instanceof LPLString))
         return error(b, "must be a basic type");
      for (int i=0; i<var.size(); i++) {
         LPLMapPair currentPair = (LPLMapPair)var.get(i);
         if (b instanceof LPLInt && currentPair.key instanceof LPLInt) {
            if (((LPLInt)currentPair.key).var == ((LPLInt)b).var)
               return (LPLBaseType)var.set(i,new LPLMapPair(b,c));
         } else if (b instanceof LPLBoolean && currentPair.key instanceof LPLBoolean) {
            if (((LPLBoolean)currentPair.key).var == ((LPLBoolean)b).var)
               return (LPLBaseType)var.set(i,new LPLMapPair(b,c));
         } else if (b instanceof LPLString && currentPair.key instanceof LPLString) {
            if (((LPLString)currentPair.key).var.equals(((LPLString)b).var))
               return (LPLBaseType)var.set(i,new LPLMapPair(b,c));
         }
      }
      LPLMapPair newPair = new LPLMapPair(b,c);
      return new LPLBoolean(var.add(newPair));
   }

   public LPLBaseType get(LPLBaseType b) {
      if (!(b instanceof LPLInt || b instanceof LPLBoolean || b instanceof LPLString))
         return error(b, "must be a basic type");
```

```
    Iterator it = var.iterator();
    while (it.hasNext()) {
        LPLMapPair currentPair = (LPLMapPair)it.next();
        if (b instanceof LPLInt && currentPair.key instanceof LPLInt) {
            if (((LPLInt)currentPair.key).var == ((LPLInt)b).var) return currentPair.value;
        } else if (b instanceof LPLBoolean && currentPair.key instanceof LPLBoolean) {
            if (((LPLBoolean)currentPair.key).var == ((LPLBoolean)b).var) return currentPair.value;
        } else if (b instanceof LPLString && currentPair.key instanceof LPLString) {
            if (((LPLString)currentPair.key).var.equals(((LPLString)b).var)) return currentPair.value;
        }
    }
    return null;
}

public LPLBaseType remove(LPLBaseType b) {
    if (!(b instanceof LPLInt || b instanceof LPLBoolean || b instanceof LPLString))
        return error(b, "must be a basic type");
    Iterator it = var.iterator();
    while (it.hasNext()) {
        LPLMapPair currentPair = (LPLMapPair)it.next();
        if (b instanceof LPLInt && currentPair.key instanceof LPLInt) {
            if (((LPLInt)currentPair.key).var == ((LPLInt)b).var) var.remove(currentPair);
        } else if (b instanceof LPLBoolean && currentPair.key instanceof LPLBoolean) {
            if (((LPLBoolean)currentPair.key).var == ((LPLBoolean)b).var) var.remove(currentPair);
        } else if (b instanceof LPLString && currentPair.key instanceof LPLString) {
            if (((LPLString)currentPair.key).var == ((LPLString)b).var) var.remove(currentPair);
        }
    }
    return null;
}

public LPLBaseType eq(LPLBaseType b) {
    return compare(b, true);
}

public LPLBaseType ne(LPLBaseType b) {
    return compare(b, false);
}

public LPLBaseType plus(LPLBaseType b) {
    ArrayList cloneVar = new ArrayList(var);
    if (!(b instanceof LPLDictionary))
        return error(b, "invalid type");
    Iterator it = ((LPLDictionary)b).var.iterator();
    while (it.hasNext()) {
        LPLMapPair currentPair = (LPLMapPair)it.next();
        if (get(currentPair.key) != null) continue;
        cloneVar.add(currentPair);
    }
    return new LPLDictionary(cloneVar);
}

private LPLBoolean compare(LPLBaseType b, boolean isEq) {
    if (var.size() != ((LPLDictionary)b).var.size())
        return new LPLBoolean(!isEq);
    for (int i=0; i<var.size(); i++) {
        LPLMapPair currentPair1 = (LPLMapPair) var.get(i);
```

```
            LPLMapPair currentPair2 = (LPLMapPair) var.get(i);
            LPLBoolean comparison = (LPLBoolean) currentPair1.eq(currentPair2);
            if (!comparison.var) return new LPLBoolean(!isEq);
        }
        return new LPLBoolean(isEq);
    }

    public ArrayList getVar() {
        return var;
    }
}
```

8.3.6 src/lpl/datatypes/LPLFunction.java

```
/**
 * Function data type, base type for builtin functions
 * Extensible to possibly implement user-defined functions later on
 * Partially adapted from Mx language
 *
 * @version $Id: LPLFunction.java,v 1.6 2006/05/07 18:45:36 genek81 Exp $
 */

package lpl.datatypes;
import java.io.PrintWriter;

import lpl.symboltable.LPLSymbolTable;

public class LPLFunction extends LPLBaseType {

    String[] args;
    LPLSymbolTable pst;
    int id;

    public LPLFunction(String name, String[] args, LPLSymbolTable pst) {
        super(name);
        this.args = args;
        this.pst = pst;
    }

    public LPLFunction(String name, int id) {
        super(name);
        this.args = null;
        this.id = id;
        pst = null;
    }

    public final boolean isInternal() {
        //TODO accurately determine if function is valid
        return true;
    }

    public final int getInternalId() {
        return id;
    }

    public String typename() {
```

```
        return "function";
    }

    public LPLBaseType copy() {
        return new LPLFunction(name, args, pst);
    }

    public void print(PrintWriter w) {
        w.println(name + " = <internal-function> #" + id);
    }

    public String[] getArgs() {
        return args;
    }

    public LPLSymbolTable getParentSymbolTable() {
        return pst;
    }

}
```

8.3.7 src/lpl/datatypes/LPLInt.java

```
/**
 * Integer data type
 * Partially adapted from Mx language
 *
 * @version $Id: LPLInt.java,v 1.6 2006/04/30 04:21:22 genek81 Exp $
 */

package lpl.datatypes;


public class LPLInt extends LPLBaseType {

    public int var;

    public LPLInt(int x) {
        var = x;
    }

    public String typename() {
        return "int";
    }

    public LPLBaseType copy() {
        return new LPLInt(var);
    }

    public static int intValue(LPLBaseType b) {
        if (b instanceof LPLString)
            return new Integer(((LPLString) b).var).intValue();
        if (b instanceof LPLInt)
            return ((LPLInt) b).var;
        b.error("cast to int");
        return 0;
```

```
}

public String toString() {
   return Integer.toString(var);
}

public boolean equals(Object o) {
   return intValue((LPLBaseType)o) == var;
}

public LPLBaseType uminus() {
   return new LPLInt(-var);
}

public LPLBaseType plus(LPLBaseType b) {
   return new LPLInt(var + intValue(b));
}

public LPLBaseType pluseq(LPLBaseType b) {
   var += intValue(b);
   return this;
}

public LPLBaseType minus(LPLBaseType b) {
   return new LPLInt(var - intValue(b));
}

public LPLBaseType minuseq(LPLBaseType b) {
   var -= intValue(b);
   return this;
}

public LPLBaseType mult(LPLBaseType b) {
   return new LPLInt(var * intValue(b));
}

public LPLBaseType multeq(LPLBaseType b) {
   var *= intValue(b);
   return this;
}

public LPLBaseType div(LPLBaseType b) {
   return new LPLInt(var / intValue(b));
}

public LPLBaseType diveq(LPLBaseType b) {
   var /= intValue(b);
   return this;
}

public LPLBaseType mod(LPLBaseType b) {
   return new LPLInt(var % intValue(b));
}

public LPLBaseType modeq(LPLBaseType b) {
   var %= intValue(b);
   return this;
```

```
    }

    public LPLBaseType gt(LPLBaseType b) {
        return new LPLBoolean(var > intValue(b));
    }

    public LPLBaseType gte(LPLBaseType b) {
        return new LPLBoolean(var >= intValue(b));
    }

    public LPLBaseType lt(LPLBaseType b) {
        return new LPLBoolean(var < intValue(b));
    }

    public LPLBaseType lte(LPLBaseType b) {
        return new LPLBoolean(var <= intValue(b));
    }

    public LPLBaseType eq(LPLBaseType b) {
        return new LPLBoolean(var == intValue(b));
    }

    public LPLBaseType ne(LPLBaseType b) {
        return new LPLBoolean(var != intValue(b));
    }

}
```

8.3.8 src/lpl/datatypes/LPLList.java

```
/**
 * List data type, an array of basic types
 *
 * @version $Id: LPLList.java,v 1.11 2006/05/07 02:17:04 genek81 Exp $
 */

package lpl.datatypes;

import java.util.ArrayList;

public class LPLList extends LPLBaseType implements LPLCollection {
    public ArrayList var;

    public LPLList(ArrayList list) {
        this.var = list;
    }

    public String typename() {
        return "list";
    }

    public LPLBaseType copy() {
        return new LPLList(var);
    }

    public String toString() {
```

```
  return var.toString();
}

public void add(LPLBaseType b) {
  var.add(b);
}

public LPLBaseType addAt(LPLBaseType index, LPLBaseType b) {
  if (index instanceof LPLInt)
    return (LPLBaseType)var.set(((LPLInt)index).var, b);
  return error(index, "index type");
}

public LPLBaseType get(LPLBaseType b) {
  if (b instanceof LPLInt)
    return (LPLBaseType)var.get(((LPLInt)b).var);
  return error(b, "dereference");
}

public LPLBaseType remove(LPLBaseType b) {
  if (b instanceof LPLInt)
    return (LPLBaseType)var.remove(((LPLInt)b).var);
  return error(b, "dereference");
}

public LPLBaseType plus(LPLBaseType b) {
  ArrayList cloneVar = new ArrayList(var);
  if (b instanceof LPLDictionary)
    return error(b, "invalid type");
  else if (b instanceof LPLList)
    cloneVar.addAll(((LPLList)b).var);
  else
    cloneVar.add(b);
  return new LPLList(cloneVar);
}

public LPLBaseType pluseq(LPLBaseType b) {
  if (b instanceof LPLDictionary)
    return error(b, "invalid type");
  else if (b instanceof LPLList)
    var.addAll(((LPLList)b).var);
  else
    var.add(b);
  return this;
}

public LPLBaseType eq(LPLBaseType b) {
  return new LPLBoolean(var.equals(((LPLList)b).var));
}

public LPLBaseType ne(LPLBaseType b) {
  return new LPLBoolean(!var.equals(((LPLList)b).var));
}

public ArrayList getVar() {
  return var;
}
```

```
}
```

8.3.9 src/lpl/datatypes/LPLMapPair.java

```
/**
 * This datatype is only used with Dictionary data type
 *
 * @version $Id: LPLMapPair.java,v 1.3 2006/04/30 04:21:22 genek81 Exp $
 */

package lpl.datatypes;


public class LPLMapPair extends LPLBaseType {
   public LPLBaseType key;
   public LPLBaseType value;

   public LPLMapPair(LPLBaseType key, LPLBaseType value) {
      this.key = key;
      this.value = value;
   }

   public String typename() {
      return "map pair";
   }

   public LPLBaseType copy() {
      return new LPLMapPair(key, value);
   }

   public String toString() {
      return key.toString() + ":" + value.toString();
   }

   public LPLBaseType eq(LPLBaseType b) {
      return new LPLBoolean(key == ((LPLMapPair)b).key && value == ((LPLMapPair)b).value);
   }

   public LPLBaseType ne(LPLBaseType b) {
      return new LPLBoolean(key != ((LPLMapPair)b).key || value != ((LPLMapPair)b).value);
   }
}
```

8.3.10 src/lpl/datatypes/LPLString.java

```
/**
 * String data type
 * Partially adapted from Mx language
 *
 * @version $Id: LPLString.java,v 1.7 2006/05/07 02:17:04 genek81 Exp $
 */

package lpl.datatypes;
```

```
public class LPLString extends LPLBaseType {

    public String var;

    public LPLString(String str) {
        this.var = str;
    }

    public String typename() {
        return "string";
    }

    public LPLBaseType copy() {
        return new LPLString(var);
    }

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

    public boolean equals(Object o) {
        return ((LPLString)o).var == var;
    }

    public LPLBaseType plus(LPLBaseType b) {
        if (b instanceof LPLString)
            return new LPLString(var.concat(((LPLString) b).var));
        else if (b instanceof LPLInt)
            return new LPLString(var.concat(Integer.toString(((LPLInt) b).var)));
        else if (b instanceof LPLBoolean)
            return new LPLString(var.concat(Boolean.toString(((LPLBoolean) b).var)));
        else if (b instanceof LPLDatetime)
            return new LPLString(var.concat(((LPLDatetime) b).toString()));
        return error(b, "+");
    }

    public LPLBaseType pluseq(LPLBaseType b) {
        if (b instanceof LPLString) {
            var = var.concat(((LPLString) b).var);
            return this;
        } else if (b instanceof LPLInt) {
            var = var.concat(Integer.toString(((LPLInt) b).var));
            return this;
        }
        return error(b, "+=");
    }

    public LPLBaseType eq(LPLBaseType b) {
        return new LPLBoolean(var == ((LPLString)b).var);
    }

    public LPLBaseType ne(LPLBaseType b) {
        return new LPLBoolean(var != ((LPLString)b).var);
    }
}
```

8.3.11 src/lpl/datatypes/LPLVariable.java

```
/**
 * Undefined variable data type
 * Partially adapted from Mx language
 *
 * @version $Id: LPLVariable.java,v 1.2 2006/04/30 04:21:22 genek81 Exp $
 */

package lpl.datatypes;
import java.io.PrintWriter;

import lpl.errors.LPLBaseException;

public class LPLVariable extends LPLBaseType {
    public LPLVariable(String name) {
        super(name);
    }

    public String typename() {
        return "undefined-variable";
    }

    public LPLBaseType copy() {
        throw new LPLBaseException("Variable " + name + " has not been defined");
    }

    public void print(PrintWriter w) {
        w.println(name + " = <undefined>");
    }
}
```

8.3.12 src/lpl/errors/LPLBaseException.java

```
/**
 * Basic exception class, may be extended in the future
 * Partially adapted from Mx language
 *
 * @version $Id: LPLBaseException.java,v 1.3 2006/04/30 04:21:24 genek81 Exp $
 */

package lpl.errors;

public class LPLBaseException extends RuntimeException {

    public LPLBaseException(String msg) {
        System.err.println("Error: " + msg);
    }

}
```

8.4 Testing

8.4.1 src/lpl/testing/expected/basic_loops.txt

```
c = 1
c = 2
c = 3
c = 4
b = 1
b = 2
b = 3
b = 4
b = 6
undefined-variable
```

8.4.2 src/lpl/testing/expected/basic_types.txt

```
int
string
boolean
a = 5
s = test
b = true
```

8.4.3 src/lpl/testing/expected/btype_operations.txt

```
c = 1
int
f = 3
true
true
false
true
h = 4
Hello world!
false
true
c = false
c = true
true
```

8.4.4 src/lpl/testing/expected/builtin_functions.txt

```
2
7
2
4
3
2
```

3
1
-3
-0.6
3
8
-3

8.4.5 src/lpl/testing/expected/complex_types.txt

```
3
4
test
true
4
test
[3, 4, test, true, 1]
1
test2
test
true
2
[1:1, 2:2, test:test, true:true, 5:test2, abc:5]
dt = Sun Apr 02 00:00:00 EST 2006
dt2 = Sun Apr 02 00:00:00 EST 2006
```

8.4.6 src/lpl/testing/expected/conditionals.txt

```
s = branch1
s = branch3
```

8.4.7 src/lpl/testing/expected/ctype_operations.txt

```
13
l = [25, 2, 3]
l2 = [25, 2, 3]
l equals l2: true
l not equal l2: false
d = [1:test, 2:true]
Hello world!!!!!...
d equals d2: true
d not equal d2: false
false
true
false
true
```

8.4.8 src/lpl/testing/expected/integration_test1.txt

```
Number of records: 3
INFO messages
-------------
Date: 02/02/2006 12:14:55,  Message: Starting program
```

Date: 02/02/2006 12:14:56, Message: Loading configuration file

Latest INFO message: Loading configuration file, Date: Thu Feb 02 00:14:56 EST 2006
Number of errors: 1

8.4.9 src/lpl/testing/expected/print_stmt.txt

Hello world!

8.4.10 src/lpl/testing/input/basic_loops.lpl

```
c=1;

while (c<5) {

    print(c);
    c+=1;

}

a = [1,2,3,4,5,6,7];

for (b in a) {

    if (b <= 4) {
        print(b);
    } else if (b > 4 && b < 6) {
        continue;
    } else {
        print(b);
        break;
    }

}

type(b);
```

8.4.11 src/lpl/testing/input/basic_types.lpl

```
a = 5;
s = "test";
b = true;
type(a);
type(s);
type(b);
print(a);
print(s);
print(b);
```

8.4.12 src/lpl/testing/input/btype_operations.lpl

```
/* integer addition and subtraction */
a = 5;
b = 3;
c = a + b - 7;
print(c);
type(c);

/* integer multiplication and division */
d = 8;
e = 3;
f = d * e / 7;
print(f);

/* integer comparison */
print(a > b);
print(b <= e);
print(a == d);
print(a != d);

/* nested expressions */
h = 1;
h += ((d - e * c) + 1) / 2;
print(h);

/* string concatenation */
str1 = "Hello";
str2 = "world!";
print(str1 + " " + str2);

/* string comparison */
print(str1 == str2);
print(str1 != str2);

/* boolean operations */
a = true;
b = false;
c = a && b;
print(c);
c = a || b;
print(c);
print(a && !b);
```

8.4.13 src/lpl/testing/input/builtin_functions.lpl

```
a = 1;
l1=[0,3];
l2=[1,2,2];
l3=["test","test"];
l4=[-1,-2,-3];
d1={"a":1;"b":2};

print(count(l1));
print(count(l1,l2,l3));
```

```
print(count(d1));
print(count(l1,d1));

print(max(l1));
print(max(l2));
print(max(l1,l2));

print(min(l2));
print(min(l2,l4));

print(avg(l1,l4));

print(sum(l1));
print(sum(l1,l2));
print(sum(l1,l4));
```

8.4.14 src/lpl/testing/input/complex_types.lpl

```
l=[3,4,"test",true];
i = 1;
print(l[0]);
print(l[1]);
print(l[2]);
print(l[3]);
print(l[i]);
print(l[i+1]);
print(l+1);

d={i:1;2:2;"test":"test";true:true;5:"test2"};
s = "test";
print(d[i]);
print(d[5]);
print(d["test"]);
print(d[true]);
print(d[i+1]);
print(d+{"abc":5});

dt=datetime("Apr 2, 2006 12:00:00 am");
print(dt);
dt2=datetime("2/4/06", "d/M/yy");
print(dt2);
```

8.4.15 src/lpl/testing/input/conditionals.lpl

```
a = 4;
s = "";

if (a > 3) {
   s = "branch1";
} else if (a == 3) {
   s = "branch2";
}

print(s);
```

```
if (a == 3 + 5) {
   s = "branch1";
} else if (a >= (3 * (5-2))) {
   s = "branch2";
} else {
   s = "branch3";
}

print(s);
```

8.4.16 src/lpl/testing/input/ctype_operations.lpl

```
/* list modification and dereferencing*/
l=[1,2,3];
l[0]=25;
print(l[0]-10-l[1]);

/* list comparison */
l2=[25,2,3];
print(l);
print(l2);
print("l equals l2: " + (l == l2));
print("l not equal l2: " + (l != l2));

/* nested expressions */

/* dictionary modification and dereferencing */
a=2;
d={1:"test";a:true};
print(d);
d[1]="Hello ";
d[a]="world!";
d[3]="!!!!";
print(d[1]+d[2]+d[3]+"...");

/* dictionary comparison */
d={1:1;"test":"Test";true:true};
d2={1:1;"test":"Test";true:true};
print("d equals d2: " + (d == d2));
print("d not equal d2: " + (d != d2));

/* datetime comparison */
dt=datetime("Apr 2, 2006 12:00:00 am");
dt2=datetime();
print(dt > dt2);
print(dt < dt2);
print(dt == dt2);
print(dt != dt2);
```

8.4.17 src/lpl/testing/input/integration_test1.lpl

```
input("I:\work\CVN\eclipse\workspace\LPL\test.log");

sep = "...";
infos = {};
```

```
errors = [];

begin (sep){
    "(?s)(?i)\s*([^\s]*\s+[^\s]*).*INFO\s+([^\n\r]*).*" : {
        infos[_1] = _2;
    }
    "(?s)(?i)\s*([^\s]*\s+[^\s]*).*ERROR\s+([^\n\r]*).*" : {
        errors += _0;
    }
} end {
    print("Number of records: " + _NR);
}

latestInfoDT=datetime("Jan 1, 2006 00:00:00 am");
print("INFO messages");
print("-------------");
latestInfo = "";

for (date in infos) {
    dt = datetime(date, "MM/dd/yyyy HH:mm:ss");
    if (dt >= latestInfoDT) {
        latestInfoDT = dt;
        latestInfo = infos[date];
    }
    print("Date: " + date + ",  Message: " + infos[date]);
}

print("");
print("Latest INFO message: " + latestInfo + ",  Date: " + latestInfoDT);
print("Number of errors: " + count(errors));
```

8.4.18 src/lpl/testing/input/print_stmt.lpl

```
print("Hello world!");
```

8.4.19 src/lpl/testing/tests/AllTests.java

```
package lpl.testing.tests;

import junit.framework.Test;
import junit.framework.TestSuite;

public class AllTests {

    public static Test suite() {
        TestSuite suite = new TestSuite("Test for lpl.testing");
        //$JUnit-BEGIN$
        suite.addTestSuite(ComplexTypesTest.class);
        suite.addTestSuite(BasicTypesTest.class);
        suite.addTestSuite(StatementsTest.class);
        suite.addTestSuite(IntegrationTest.class);
        //$JUnit-END$
        return suite;
    }
```

```
}
```

8.4.20 src/lpl/testing/tests/BasicTypesTest.java

```
package lpl.testing.tests;

import java.io.IOException;

import junit.framework.TestCase;
import lpl.testing.util.LPLRunner;

public class BasicTypesTest extends TestCase {

  public static void main(String[] args) {
    junit.textui.TestRunner.run(BasicTypesTest.class);
  }

  public BasicTypesTest(String name) {
    super(name);
  }

  protected void setUp() throws Exception {
  }

  protected void tearDown() throws Exception {
  }

  public void testPrintStatement() throws IOException {
    String expected = LPLRunner.getExpected("print_stmt.txt");
    String actual = LPLRunner.runLPL("print_stmt.lpl", true);
    assertEquals("Print statement error:", expected, actual);
  }

  public void testBasicTypes() throws IOException {
    String expected = LPLRunner.getExpected("basic_types.txt");
    String actual = LPLRunner.runLPL("basic_types.lpl", true);
    assertEquals("Basic types error:", expected, actual);
  }

  public void testBasicTypeOperations() throws IOException {
    String expected = LPLRunner.getExpected("btype_operations.txt");
    String actual = LPLRunner.runLPL("btype_operations.lpl", true);
    assertEquals("Basic type operations error:", expected, actual);
  }

}
```

8.4.21 src/lpl/testing/tests/ComplexTypesTest.java

```
package lpl.testing.tests;

import java.io.IOException;

import junit.framework.TestCase;
import lpl.testing.util.LPLRunner;
```

```
public class ComplexTypesTest extends TestCase {

   public static void main(String[] args) {
      junit.textui.TestRunner.run(ComplexTypesTest.class);
   }

   public ComplexTypesTest(String name) {
      super(name);
   }

   protected void setUp() throws Exception {
   }

   protected void tearDown() throws Exception {
   }


   public void testComplexTypes() throws IOException {
      String expected = LPLRunner.getExpected("complex_types.txt");
      String actual = LPLRunner.runLPL("complex_types.lpl", true);
      assertEquals("Complex types error:", expected, actual);
   }

   public void testComplexTypeOperations() throws IOException {
      String expected = LPLRunner.getExpected("ctype_operations.txt");
      String actual = LPLRunner.runLPL("ctype_operations.lpl", true);
      assertEquals("Complex type operations error:", expected, actual);
   }

}
```

8.4.22 src/lpl/testing/tests/IntegrationTest.java

```
package lpl.testing.tests;

import java.io.IOException;

import junit.framework.TestCase;
import lpl.testing.util.LPLRunner;

public class IntegrationTest extends TestCase {

   public static void main(String[] args) {
      junit.textui.TestRunner.run(IntegrationTest.class);
   }

   public IntegrationTest(String name) {
      super(name);
   }

   protected void setUp() throws Exception {
   }

   protected void tearDown() throws Exception {
   }
```

```
  public void testIntegration() throws IOException {
    String expected = LPLRunner.getExpected("integration_test1.txt");
    String actual = LPLRunner.runLPL("integration_test1.lpl", true);
    assertEquals("Integration test error:", expected, actual);
  }

}
```

8.4.23 src/lpl/testing/tests/StatementsTest.java

```
package lpl.testing.tests;

import java.io.IOException;

import junit.framework.TestCase;
import lpl.testing.util.LPLRunner;

public class StatementsTest extends TestCase {

  public static void main(String[] args) {
    junit.textui.TestRunner.run(StatementsTest.class);
  }

  public StatementsTest(String name) {
    super(name);
  }

  protected void setUp() throws Exception {
  }

  protected void tearDown() throws Exception {
  }


  public void testConditionals() throws IOException {
    String expected = LPLRunner.getExpected("conditionals.txt");
    String actual = LPLRunner.runLPL("conditionals.lpl", true);
    assertEquals("Conditionals error:", expected, actual);
  }

  public void testBasicLoops() throws IOException {
    String expected = LPLRunner.getExpected("basic_loops.txt");
    String actual = LPLRunner.runLPL("basic_loops.lpl", true);
    assertEquals("Basic loops error:", expected, actual);
  }

  public void testBuiltinFunctions() throws IOException {
    String expected = LPLRunner.getExpected("builtin_functions.txt");
    String actual = LPLRunner.runLPL("builtin_functions.lpl", true);
    assertEquals("Builtin functions error:", expected, actual);
  }

}
```

8.4.24 src/lpl/testing/util/LPLRunner.java

```
/**
 * Test runner utility class
 *
 * @version $Id: LPLRunner.java,v 1.3 2006/05/07 18:45:37 genek81 Exp $
 */

package lpl.testing.util;

import java.io.BufferedReader;
import java.io.File;
import java.io.FileReader;
import java.io.IOException;
import java.io.InputStreamReader;

public class LPLRunner {

    private static Runtime runtime = Runtime.getRuntime();
    private static String TEST_CLASSPATH = "-cp
.;D:\\eclipse\\plugins\\AntlrStudio_1.1.0\\antlr_2.7.6.1.jar;D:\\eclipse\\plugins\\AntlrStudio_1.1.0\\a
ntlrdebug_1.0.0.jar";
    private static String INPUT_PATH = "lpl\\testing\\input\\";
    private static String EXPECTED_PATH = "\\lpl\\testing\\expected\\";
    private static File LPL_MAIN_DIR = new
File("I:\\work\\CVN\\eclipse\\workspace\\LPL\\classes");

    public static String runLPL(String input, boolean isFile) throws IOException {
        String output = "";
        String s = null;

        if (isFile) {
            input = INPUT_PATH + input;
        }
        String command = "java "+TEST_CLASSPATH+" lpl.LPLMain "+input;
        Process p = runtime.exec(command, null, LPL_MAIN_DIR);

        BufferedReader stdInput = new BufferedReader(new
             InputStreamReader(p.getInputStream()));

        BufferedReader stdError = new BufferedReader(new
             InputStreamReader(p.getErrorStream()));

        while ((s = stdInput.readLine()) != null) {
            output += "\n"+s;
        }

        while ((s = stdError.readLine()) != null) {
            output += "\n"+s;
        }

        stdInput.close();
        stdError.close();

        return output.substring(1);
    }
```

```
    public static String getExpected(String filename) throws IOException {
        String result = "";
        String s = null;

        BufferedReader reader = new BufferedReader(new
FileReader(LPL_MAIN_DIR+EXPECTED_PATH+filename));

        while ((s = reader.readLine()) != null) {
            result += "\n"+s;
        }
        reader.close();

        return result.substring(1);
    }
}
```