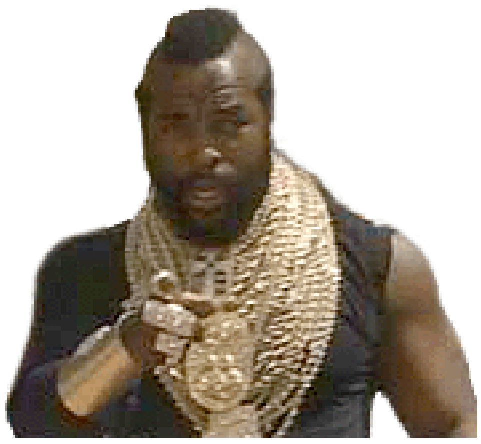


Mr. Proper

Chat Client Log File Manipulation Language

Matt Anderegg (mja105@columbia.edu)
William Blinn (wb169@columbia.edu)
Jeffrey Lin (jlin@columbia.edu)
Nabil Shahid (ns602@columbia.edu)

Table of Contents

Section 1: Introduction

Mr. Proper™: An easy to use, powerful, multi chat client supporting, log file manipulation language.

1.1 Overview

In this Internet age, people do a lot of chatting. It is not uncommon for one to amass many tens of megabytes of log files in the course of chatting over a month. Sometimes, going through and reviewing these enormously massively gigantic excessive log files can be a daunting task that will bring fear to all but the most chronic of procrastinators. Our language facilitates simple manipulation of said gargantuan log files. Such tasks as converting log files from one logger format to another will be made trivial by implementing them at the language level, hiding its complexity from users. Dealing with entire log files as primitive data types allows users of the language to manipulate logs with the ease of integers and strings.

1.1.1 Easy to use

Mr. Proper™ hides the complexity of dealing with unwieldy log files from the user, allowing them to handle them the same way that they would handle data types such as integers in other languages.

1.1.2 Powerful

Broad functionality includes analysis of conversations to determine likely personality traits of chat client users, based on characteristics such as text style, writing style, grammar style, and sentence lengths.

1.1.3 Multi chat client support

Mr. Proper™ supports log file formats for several of the major chat clients, such as Trillian, AIM, Fire, and gAIM. Converting from one log format to another is as simple as writing a few lines of code, rather than creating an entire program to parse and convert radically different logging formats.

1.1.4 Log file manipulation

At its core, Mr. Proper™ is designed to make operations that users may want to perform on their log files easy. Users can write simple powerful programs in few lines of code, without dealing with such overheads as opening files, reading individual lines, and parsing the text from them. Searching for a particular word in a particular log file is possible with a minimal amount of effort, using Mr. Proper™.

1.2 Language functionality

- Using scoring based on regular expressions (a la SpamAssassin) to determine personality traits of people with whom you speak. Loops would be used to go through multiple IM transcripts.
- Compiling lists of links that have been sent to you over IM so that you have a centralized list (a further use of regular expressions). This will be accomplished by looping through all IM transcripts for a particular chat client user.
- Converting logs from one program's format to another and the ability to support additional log formats.
- Showing the frequency of conversations based on time-stamps and using it to answer questions such as whom someone talks to most at night or what time of day a particular chat client user chats most.
- Word counting, which would make use of arrays to store most frequently used words and looping to go through several IM transcripts.
- Primitive data types for messages, which contain data about the chat client user, timestamp, date and the text of the message. There will be easy access to all of these primitives simply by opening a log file.

1.3 Why is it useful?

- Have you ever wondered whom you talk dirty to the most? Have you ever sat there at your computer and wondered how many times you said "fuck" yesterday?
- Find links that friends sent to you.
- Search logs for certain things.
- Rate the intelligence level of your friends
- Analyze conversations for personality traits and talking styles
- When you switch chat clients you can easily convert your old conversation logs into the appropriate format for your new chat client in order to preserve your conversation history without giving up consistency.
- Parse and format conversation logs to make online publishing of conversation snippets a breeze. Conversation bits may be automatically reviewed with regular expression scoring to flag questionable or deeply personal content that you might not want put online.

Section 2: Language Tutorial

2.1 Running .mrp script

MrProper is an interpreted language, so there is no need to compile .mrp programs before running them. They must, however, be piped into the MrProper executable. To execute the MrProper interpreter on a sample file test.mrp, type:

```
$ cat test.mrp | java Main
```

2.2 Layout of .mrp programs

All MrProper programs have a similar layout. Functions and variables must be declared in a specific part of the script or the program will not execute correctly. The basic format of a .mrp script is as follows:

```
function 1 {
     variables for function 1
     code for function 1
     return value for function 1
}

function 2 {
     variables for function 2
     code for function 2
     return value for function 2
}

main {
     variables for main
     code for main
}
```

All function definitions other than the main method must precede the main method. All variable declarations are done at the beginning of each function and at the beginning of the main method. The return value for every function is the last line of code in the function and no code after it will be executed. It is important to note that variables such as counters cannot be declared in loop headers, so a Java loop such as:

```
for (int i = 0; i < 100; i++)
```

must be declared as

```
int i;
for (i=1:99)
```

in a MrProper program.

2.3 Basic control structures

The MrProper language features a few basic control structures, including if/else statements, for loops, and while loops. These can be used to control the flow of the user program.

2.3.1 If/else

A basic if statement is used like this:

```
if (x == 5) {
     // execute this code;
}
```

An if else statement can also be used. It is used like this:

```
if (x == 5) {
     // execute this code;
}
else {
     // execute this code;
}
```

2.3.2 For

A for loop executes the encapsulated chunk of code repeatedly until its counter variable exceeds a given value. Note that the counter variable must be declared outside of the for loop and at the beginning of the function/main as explained above.

```
int x;
for (x = 0:1) {
     // execute code;
}
```

This sample for loop will execute the code when x = 0, increment the counter variable x, and then execute the code again for x = 1.

2.3.3 While

A while loop executes the encapsulated chunk of code repeatedly until the given condition is reached.

```
int x = 5;
int y = 0;
while (x != y) {
     y += 1; // note that MrProper does not support the ++ and --
operators
}
```

This loop will run until the value of y reaches 5, at which point it will be equal to x and the program will break out of the loop.

2.4 Functions and the main method

As explained above, all function definitions must come before the main method. Function headers include the keyword "func" as well as a return type and function name. All parameters are passed into the function with types and parameter names enclosed between parentheses. This example function takes two integer, adds them and returns the result.

```
func int addnumbers (int number1, int number2) {
     int newnumber;
     newnumber = number1 + number2;
     return newnumber;
}
```

You can now add a main method under this function definition to complete the .mrp script and make it runnable.

```
main {
     int x = 1;
     int y = 2;
     int z;
     string foo = "moo";
     z = addnumbers(x, y);
     print(z);
     print(foo);
}
```

The print command is used to simply print information to stdout (console). In this case, the numbers x and y are added together using the function addnumbers. The returned int is stored in a third variable z, which is then printed to the screen. This program will produce an output of "3" and then "moo" on a new line.

2.5 Working with logs

What makes MrProper useful and powerful is its ability to read, manipulate, and write log files. Once loaded into the program, log files can be handled just like primitive data types. Log data types are essentially a collection of message data types. Each message has associated with it a speaker name (the person who wrote the message), a timestamp, and the actual text of the message itself. Logs can either be loaded from files on disk, or can be created manually by instantiating an empty log and inserting messages into it.

To open a log, the user must know the path where the log files are stored, the screen name associated with the log files, and the format of the log file. The format of the log file is passed as an integer into the open method which loads log data into the program. An integer value of 0 specifies an AIM log, an integer value of 1 specifies a Trillian log, and an integer value of 2 specifies a Fire log.

This function opens log files for the screen name passed into it from the default path "" (assume log files are of the Trillian variety) and returns the log object.

```
func log openlog (string screenname) {
     log newlog;
     newlog = open("", screenname, 1);
     return newlog;
}
```

Once the log has been opened the user can then access individual messages from within the log to perform operations on them. The entire log can also be printed to stdout using the print command:

```
     print (newlog);
```

As an example, we can look at the following MrProper program that performs a wordcount and outputs the result.

```
func int wordcount(string word, string path) {
     log myLog = open(path, "walido2000", 1);
     int i;
     int length = logLength(myLog) - 1;
     int count = 0;
     message myMessage;
     string text;
     for(i = 0:length) {
           text = getText(getMessage(myLog, i));

           if(contains(text, word)) {
                count += 1;
           }
     }
     return count;
}

main {
     string word = "fuck";
     int count = wordcount(word, "");
     print(concat(word, " appears "));
     print(count);
     print("times");
}
```

Section 3: Language Reference Manual

3.1 Lexical conventions

3.1.1 Comments

The characters `/*` start what may be a multi-line comment terminated by `*/`, while the characters `//` start a single-line comment.

3.1.2 Identifiers

An identifier consists of letters, digits and underscore `"_"`. The first character of an identifier should be a letter or underscore. Upper and lower case letters are considered different.

3.1.3 Keywords

These identifiers are reserved as keywords:

```
for         if          else        loop        break
continue    return      exit        include     func
and         or          not         true        false
void        null
```

3.1.4 Numbers

A number consists of digits, optional decimal point `"."` and optional `e` followed by signed integer exponent. Integers and floating numbers will be distinguished.

3.1.5 Strings

A string is a sequence of characters enclosed by double quotes `"`. A double quote inside the string is represented by two consecutive double quotes.

3.1.6 Other tokens

Some symbolic characters or sequences of symbolic characters are used in the language:

```
{     }     (     )     [     ]     ,     ;
+     -     *     /     %
=     +=    -=    *=    /=    %=
>=    <=    >     <     ==    !=
```

3.2 Types

In this language we do not have any explicit type specifications. As a result, the language is not statically typed, though this language is possibly compiled to Java source code. Data types are distinguished at run time. Each variable is bounded by a type tag, which could be checked at run time.

Implemented run-time types:

- `bool` : boolean values being either true or false
- `int` : 32-bit integers
- `float` : 32-bit IEEE floating format
- `string` : string of characters
- `message` : encapsulation of a single IM message including the screen name of the sender, the time and data stamp, and the message text.
- `log` : an array of `messages`
- `logformat` : an identifier of the format of a log that tells the program how to parse a log file
- `range` : Only for temporary values (cannot be assigned to a variable)

3.3 Expression

3.3.1 Primary expressions

Primary expressions include identifiers, constants, function calls, access to array elements, and any other expressions surrounded by `(` and `)`.

Identifier
An identifier itself is a left-value expression. It will be evaluated to some values bounded to this identifier.

Constant
A constant is a right-value expression, which will be evaluated to the constant itself.

Function call
A function call consists of a function identifier, followed by a list of arguments enclosed by `(` and `)`. The list of arguments contains zero or more arguments separated by a comma `","`. Each argument is an expression. Function call is a right-value expression.

Access to array elements
This primary expression consists of an array identifier, followed by an index enclosed by `[` and `]`. This expression is itself left-value. If an array is to be initialized to a specified size, the initialization must be done in the declaration.

3.3.2 Arithmetic operators

Arithmetic operators take primary expressions as operands.

Unary arithmetic operators
Unary operators `+` and `–` can be prefixed to a expression. `+` operator returns the expression itself whereas the `–` operator returns the negative of the primary expression. They are applicable to `int` and `float` values.

e.g. `–14` or `+5`

Multiplicative operators
Binary operators `*`, `/`, and `%` indicate multiplication, division, and modulo, respectively. They are grouped left to right.
They are applicable to `int` and `float` values.

e.g. `18 * a` or `12 / 3` or `63 % 5`

Additive operators
Binary operators `+` and `–` indicate addition and subtraction, respectively. They are grouped left to right.
They are applicable to `int` and `float` values. Operator `+` is also applicable to `string` values.

e.g. `19 + 4` or `14 – b` or `moo + foo`

3.3.3 Relational operators

Binary relational operators `>=`, `<=`, `==`, `!=`, `>` and `<` indicate whether the first operand is greater than or equal to, less than or equal to, equal to, not equal to, greater than, or less than the second operand, respectively.

e.g. `a != 12` or `b == 4` or `c >= 5`

3.3.4 Logical operators

Logical operators take relational expressions as operands. Among these operators, operator not has the highest precedence, then operator and, and operator or is the lowest.
These operators are applicable to `bool` values.

Not operator
A logical expression consisting of a `not` operator followed by a relational expression returns the logical negation of this relational expression.
e.g. `not true`

And operator
Operator `and` indicates the logical and of two relational expressions. It is short-circuited for `bool` values.
e.g. `true and false` or `true and true`

Or operator
Operator `or` indicates the logical or of two relational expressions. It is short-circuited for `bool` values.
e.g. `true or false` or `false or false`

3.4 Statements

Statements are basically elements of a program. A sequence of statements will be executed sequentially, unless the flow-control statements indicate otherwise.
Note that in the following specifications, elements enclosed in < and > will be replaced by some corresponding component.

3.4.1 Statements in { and }

A group of zero or more statements can be surrounded by `{` and `}`, in which case they altogether are treated as a single statement. This is true of all function calls, including the main() function.

3.4.2 Assignments

An assignment is in this form:
```
<left-value expr> = <right-value expr>;
```
where `;` is the terminator of this assignment statement.

3.4.3 Conditional statements

A conditional statement is in this form:
```
if ( <logical expression> ) {
      <statement(s)>
}
```

or

```
if ( <logical expression> ) {
      <statement(s)>
}
else {
      <statement(s)>
}
```
If the logical expression returns `true`, the first statement is executed, otherwise the optional second statement is executed.

3.4.4 Iterative statements

Iterative statements are basically loops. There are two kinds of loops, for and while.

For statement

The `for` statement is usually used for indices of array elements. It has this form: (note that `id` means identifier)

```
for ( <id> = <range> ) {
      <statement(s)>
}
```

While statement

The `while` statement is the general iterative statement. It is in this form:

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

The expression is evaluated at the start of each loop. If the expression is evaluated to `false`, the loop will be skipped.
An infinite loop will be introduced if the expression will always evaluate to `true` and a `break` statement is not used inside the loop.

Break statement

The break statement will break the inner-most or labeled iterative statement. This statement consists of keyword break, optionally followed by a label, then followed by a `;`.

Continue statement

The continue statement will end the current iteration of the innermost or labeled iterative statement and proceed to its next iteration. This statement consists of keyword continue, optionally followed by a label, then followed by a `;`.

3.4.5 Function invocation and return

Function call

Different from other expressions, a function call followed by a `;` can be a single statement.

Return statement

The return statement is used inside the function definition body, in order to return from the function at that point. It can be followed by an optional expression, for the return value, and a `;`. The return statement must return a value of the type specified in the function definition.

3.5 Function definition

A function definition is in this form:

```
func <id> <id> ( <var-list> ) { <body> }
```

First field `id` indicates the return type of this function. Second field `id` indicates the name of this function. Field `var-list` is a list of identifiers of (formal) arguments, separated by commas `","`. The list of arguments can be empty. Field `body` is zero or more statements. The return statement should be used in the body in order to return a value of the type specified in the function definition. Functions can be overloaded.

3.6 Program Structure

This language is statically scoped.

The main part of a program is defined at the start of the language, and after that are the function definitions:

```
main {
      …
}
func void funcdef1() {
      …
}
func int funcdef2 ( String moo ) {
      …
}
```

Within the main program, variable declarations occur at the beginning before any function calls

```
main {
      log myLog;
      int i;
      string blah;

      <other statements>
}
```

Functions are allowed to have the same name as a variable.

3.7 Internal Functions

All parameters passes to function calls are required.

3.7.1 String Functions

print()

The function print simply prints a string to a standard output. The function usage is:

```
void print ( string boo )
```

Where `boo` is the string to be printed.

strLength()

The function strLength returns the integer value of the length of a string. The function usage is:

```
int strLength ( string foo )
```

Where `foo` is the string that you wish to determine the length of.

concat()

The function concat joins two strings and returns a new string as the result. The function usage is:

```
string concat ( string moo, string noo )
```

Where `moo` and `noo` are the strings to be joined.

substr()

The function substr returns a new string that is a part of the existing string. The function usage is:

```
string substr( string poo, int startIndex, int endIndex )
```

Where `poo` is the original string, Indices begin at zero. `startIndex` is the point o begin the new string, and `endIndex` is the point to end the new string at. The range is inclusive.

contains()

The function contains is a regular expression matching function. It searches one string for another string, and returns a Boolean value of true of false depending on whether

```
bool contains ( string line, string regexp )
```

Where `line` is the string that will be searched and `regexp` is the regular expression that will be looked for.

3.7.2 Log file Manipulation

createLog()

Function createLog makes a new empty log. The usage is:

```
log createLog ( int logLength, logformat format )
```

Where `logLength` is the desired length of the log in number of messages and `format` is the desired format of the log. This can be one of a few predefined types, such as AIM, Trillian, or Fire.

open()

Function open is used to open a log file and load it into the program before it can be manipulated. The usage is:

```
log open ( string logroot, string screenname, logformat
      format )
```

So the function returns a `log` datatype, which has all the log information from the file loaded into it. It takes in the name of the log file to load, and the type of log it is. This can be one of a few predefined types, such as AIM, Trillian, or Fire.

save()

After all required operations have been performed on a `Log` object, it is necessary to use the save function to save it back to disk. The save function is used as follows:

```
void save ( log logname )
```

and it writes the indicated `log` object back to disk, in the format it is currently in.

convert()

The convert function is used to transform a Log data type from one log format to another. For instance, an AIM format log file can be converted to a Trillian format log file and so on. The function usage is:

```
log convert ( logformat logtype1, logformat logtype2, log
      logname)
```

where logtype1 is the format to be converted from, and logtype2 is
the format to be converted to.

logLength()

The logLength function returns the length of a log in an integer value representing the number of messages in a log. The function usage is:

Int logLength (log myLog)

Where `myLog` is the log that you want to get the length of.

logFormat()

The function `logFormat()` takes one argument: a string corresponding to the format of the timestamp, and returns an instance of a `logformat` object. The format for the string passed in may have several parameters according to how the programmer would like the timestamp to look. The string may be any combination of white space, these parameters or other text.

- `%d` - The date of the message
- `%t` - The time of the message
- `%w` - The day of the week of the message (Monday, etc).

So, a call to `logformat` might look like this:

```
logFormat( "%d-%t" )
```

and would return a logFormat object that would print timestamps like

```
2003.10.15-14:56
```

3.7.3 Log Message Accessors

getMessageName()

Function `getMessageName()` returns as a string the Screen name associated with a given log message. It is used in the following format:

```
string getMessageName ( message myMessage )
```

Where `myMessage` is a single message that you want the name of the sender from.

getTime()

Function `getTime()` returns as an `int` the timestamp associated with a given log message. The integer returned will represent the number of seconds elapsed since 01/01/1970. It is used in the following format:

```
int getTime ( message myMessage )
```

Where `myMessage` is a single message that you want to get the timestamp from.

NOTE: If no date information is stored within the timestamp of a given log, then the time value returned will only be accurate in relation to other messages within the same log. I.e. you can compare two times within the same log relative to each other, but cannot compare with times from different log.

getText()

Function `getText()` returns as a string the text associated with a given log message. It is used in the following format:

```
string getText( int messageNumber )
```

Where `messageNumber` is a index number of a single message that you want to get the text of.

getMessage()

Function `getMessage()` returns as a message, one entry in a log. It is used in the following format:

```
message getMessage( log myLog, int messageNumber )
```

Where `myLog` is a log file that you are working with and `messageNumber` is the message entry to retrieve.

3.7.4 Log Message Manipulation

createMessage()

Function `createMessage()` creates a new empty message

```
message createMessage()
```

The function takes no input.

setMessageName()

Function `setMessageName()` modifies the screen name associated with a given log message. It is used in the following format:

```
void setMessageName( message myMessage, string name )
```

Where `myMessage` is a single message and `name` is a string containing the new desired name.

setTime()

Function `setTime()` modifies the text associated with a given log message. It is used in the following format:

```
void setName( message myMessage, int time )
```

Where `myMessage` is a single message and `time` is an `int` containing the number of seconds elapsed since 01/01/1970 for the new desired time.

setText()

Function `setText()` modifies the text associated with a given log message. It is used in the following format:

```
void setText( int messageNumber, string text )
```

Where `messageNumber` is the index of a single message and `text` is a string containing the new desired text.

setMessage()

Function setMessage() modifies a single message entry in a log. It is used in the following format:

```
setMessage( log myLog, int messageNumber,
      message myMessage )
```

Where `myLog` is a log file that you are working with, `messageNumber` is the message entry to modify, and `myMessage` is the message to be inserted into the log.

Section 4: Project Plan

4.1 Team Responsibilities

The project initially started as a group effort, with all members meeting together for early work in project planning, writing the white paper, and writing the grammar. As project progressed, each group member began specializing in tasks.

William Blinn	Team leader; walker, , and testing
Matt Anderegg	Log and message data type code
Nabil Shahid	Log and message data type code and testing
Jeffrey Lin	LRM, documentation, and report

4.2 Project Timeline

9/23/03:	White paper due
10/28/03:	LRM and grammar due
12/10/03:	Presentation
12/18/03:	Code demonstration

4.3 Project Log

Information was compiled from CVS logs and e-mail archives.

9/16/03:	Initial idea for project formed.
9/21/03:	First group meeting. Revised ideas and sent off to Professor Edwards for input.
9/23/03:	White paper submitted.
10/1/03:	First meeting with the TA.
10/9/03:	First meeting to work on the grammar.
10/13/03:	Preliminary draft of LRM written.
10/15/03:	Submitted a draft of the LRM to the TA and underwent heavy criticism.
10/19/03:	First sample programs written for testing the grammar.
10/28/03:	LRM and grammar submitted.
11/17/03:	Met with TA to demonstrate functionality of the grammar.
11/30/03:	Backend code for data type support started.
12/06/03:	Initial versions of code for data type support finished.

4.4 Programming Style

4.4.1 Antlr Coding Style

If an Antlr rule is short and contains only one choice without any action, then it will be written in one line. The colon `":"` always starts at the ninth column unless the string in front of `":"` is too long. In the one line mode, `";"` is at the end of this line.

If an Antlr rule has multiple choice or actions, the `";"` is placed in a separated line at the ninth column. `"|"` is also at ninth column. Short actions are in the same line of its rule and at the right half of the screen. Long action lines start at the thirteenth column of the next line. Code in actions follows the Java coding style.

Lexem (token) names are in upper cases and syntactic items (non-terminals) are in lower cases, which may contain the underscore `"_"`.

4.4.2 Java Coding Style

Spacing and indentation

- Indentation of each level is 4 spaces.
- The left brace `"{"` is on the same line as the statement that opens it. The corresponding right brace `"}"` occupies a full line and at the same column as the start of the statement that opened it.
- One space between a method name and `"("` but no space between arguments and their parentheses `"(" ")"`.
- Add spaces between operators and operands.
- Use spaces, not tabs.
- The then-part and else-part of an `"if"` statement must not be at the same line of `"if"` or `"else"`. Similar for `"for"` and `"while"` statement.

Names

- Class names start with "Mrp" followed by English words with the first letter of each word capitalized.
- Variable names start in lower case. Multiple-word variable names have the first letter of subsequent words capitalized.
- Method names start in lower case. Multiple-word method names have the first letter of subsequent words capitalized.

4.5 Development Environment

The lexer and parser were written in Antlr. All other code was written in Java.

4.5.1 Operating systems

Development work done using Emacs in Linux and Xcode in Mac OS X.

4.5.2 Java 1.4.2

Java is a “simple, object-oriented, network-savvy, interpreted, robust, secure, architecture neutral, portable, high-performance, multithreaded, dynamic language.” Java was chosen for our project since that is the language that all of us knew best.

4.5.3 Antlr

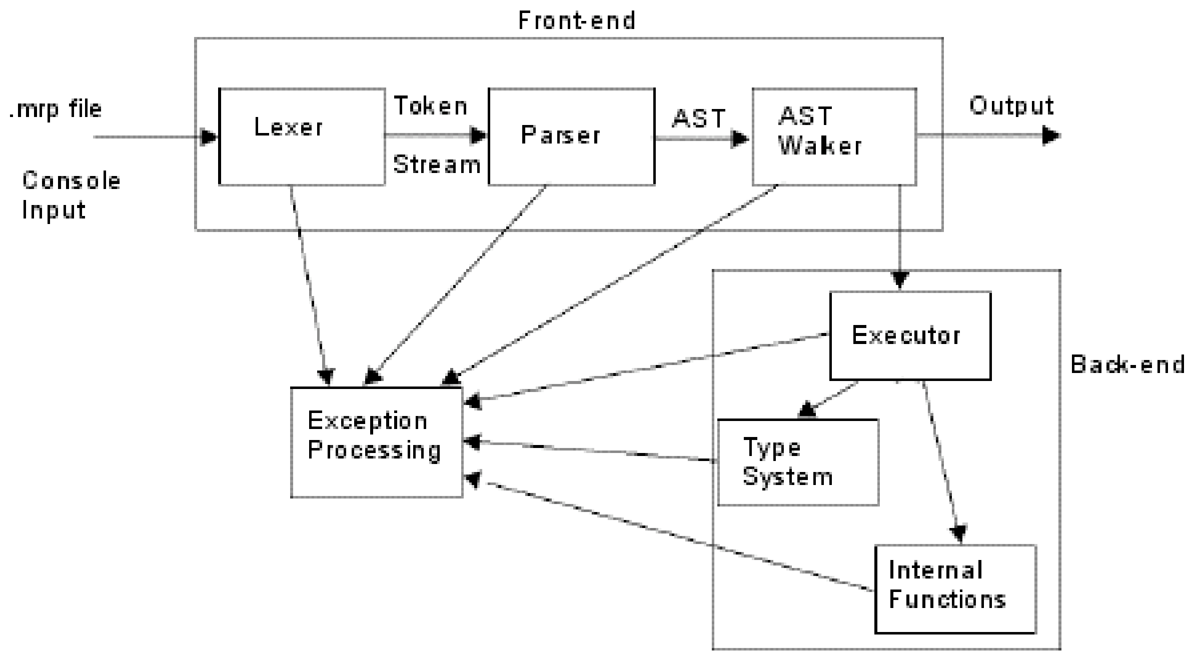
The language parser is written in Antlr, “a language tool that provides a framework for constructing recognizers, compilers, and translators from grammatical descriptions containing C++ or Java actions.”

4.5.4 CVS

Concurrent Versions System was used to keep track of code changes. The repository was kept on CRF hosts and accessed both from CLIC and remotely with SSH.

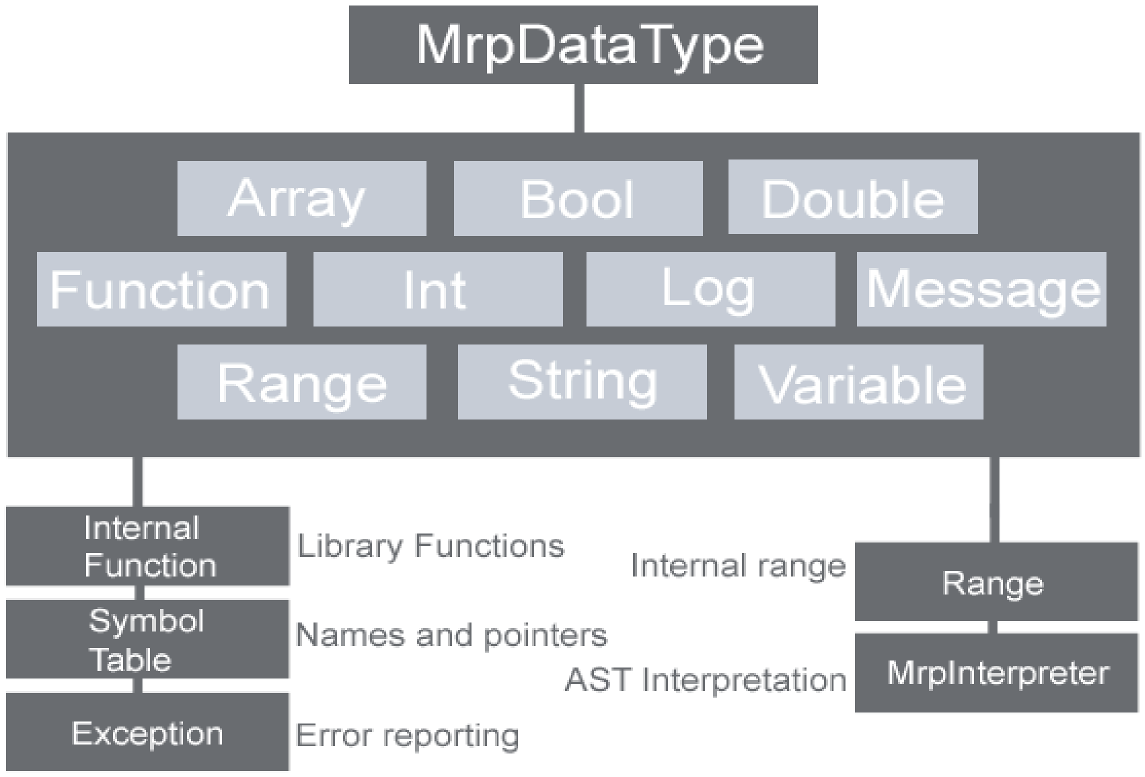
Section 5: Architectural Design

The Mrp interpreter consists of these components: a lexer that reads a program from the console and converts it to tokens, a parser that analyzes the syntactic structure of the program and converts it to an abstract syntax tree, a tree walker that traverses the abstract syntax tree and calls corresponding functions, an executor that looks up symbol tables and diverts operations to type systems, and a simple error and exception processing mechanism. The following block diagram represents this:



The lexer and parser are implemented in the Antlr source file *grammar.g* and the AST Walker is implemented in the Antlr source file *walker.g*. The source file *walker.g* makes calls to methods contained in *MrpInterpreter.java*.

The `main()` method is in *Main.java*. It is executed with input that has been piped in via the console. Very basic error reporting is handled in *MrpException.java*. The back-end consists a type system that is centralized in *MrpDataType.java* and extended by *MrpArray.java*, *MrpBool.java*, *MrpDouble.java*, *MrpFunction.java*, *MrpInt.java*, *MrpLog.java*, *MrpMessage.java*, *MrpRange.java*, *MrpString.java*, and *MrpVariable.java*. Source file *MrpRange.java* relies on source contained in *Range.java*. The classes *MrpMessage.java* and *MrpLog.java* are the central class files to the langauage and contain the predefined log types for Trillian and AIM with the methods necessary to manipulate them. The structure of the interpreter is shown in the following diagram:



Section 6: Test Plan

6.1 Goal

The goal of our language is to facilitate the manipulation of AIM logs, so testing was essential to ensure that our language operated correctly. There were different stages of testing encompassing the operation of each component individually, and the interaction of the components.

6.2 Testing Suite

The testing suite was designed to provide a quick test for all aspects of our language to ensure proper functionality. It contains the following components:

6.2.1 datatest.mrp

Test all primitive data types.

```
main {
     double d = 1.32 + 578.4365;
     print(d);
     int i = 65 * 3;
     print(i);
     string s = "MRPROPER HAS YOU";
     print(s);
     int[] a = int[3];
     a[0] = 0;
     a[1] = 1;
     a[2] = 2;
     print(a);

}
```

6.2.2 fortest.mrp

Test the functionality of `for` loops

```
main {
     print("for loop");
     for(i = 1:7) {
           print(i);
     }
}
```

6.2.3 iftest.mrp

Test the functionality of `if/else` statements.

```
main {
     int x = 3;
     if(x < 3) {
          print("x is less than three");
     }
     else {
          print("x is greater than or equal to three");
     }
}
```

6.2.4 whiletest.mrp

Test the functionality of `while` loops

```
main {
     int i = 0;
     while(i < 7) {
         print(i);
         i += 1;
     }
}
```

6.2.5 openconvertsave.mrp

Test for file manipulation functionality. Opens a log, converts it to plaintext, and then saves it.

```
main {
    log testlog = open("", "walido2000", 1);
    convert(testlog, 0);
    save(testlog, "./outlog/");
}
```

6.2.6 concatlogs.mrp

Log concatenation testing. Concatenate a log with itself and output the result.

```
main {
    log firstlog = open ("", "walido2000", 1);
    log secondlog = open ("", "walido2000", 1);
    log thirdlog = createLog(1);
    int i;
    int firstLength = logLength(firstlog);
    int secondLength = logLength(secondlog);
    int counter = 0;
    message tempMessage;

    for (i = 0:firstLength - 1) {
        tempMessage = getMessage(firstlog, i);
        setMessage(thirdlog, counter, tempMessage);
        counter += 1;
    }

    for (i = 0:secondLength - 1) {
        tempMessage = getMessage(secondlog, i);
        setMessage(thirdlog, counter, tempMessage);
        counter += 1;
    }

    convert(thirdlog, 3);
    save(thirdlog, "double.txt");
}
```

6.2.7 wordcount.mrp

Test for regular expression and word count.

```
func int wordcount(string word, string path) {
     log myLog = open(path, "walido2000", 1);
     int i;
     int length = logLength(myLog) - 1;
     //print(length);
     int count = 0;
     message myMessage;
     string text;
     for(i = 0:length) {
           text = getText(getMessage(myLog, i));
           if(contains(text, word)) {
                count += 1;
           }
     }
     return count;
}

main {
     string word = "fuck";
     int count = wordcount(word, "");
     print(concat(word, " appears "));
     print(count);
     print("times");
}
```

Section 7: Lessons Learned

7.1 Matt Anderegg

- Remember to save and compile code before running it and trying to figure out why the changes made are not apparent.
- When taking on a task as big as this, we needed to make a more proper and reasonable project plan. We ended up doing most of the work in a very short period of time whereas I feel that had we made a better plan and spread the work out over a month rather than a week it would have been much more relaxed.
- Picking a good group is imperative. Our project came out great because our group is great; we work very well together and always manage to pull through in the end.

7.2 Billy Blinn

- Always plan out the code you're going to write before starting to write.
- If you're feeling stuck and you've been attacking the same problem for two hours straight without getting anywhere, get up and take a break and think about it some more.
- Start big projects early.
- As the task becomes larger, try not to do everything together - divide up tasks to make things move along more quickly.
- Noon is too early sometimes.

7.3 Jeffrey Lin

- Trust your team members to provide sound advice and good input for you. And if there's something that you don't know, don't be embarrassed to ask your partners, no matter how stupid the question may seem.
- Create a more detailed project plan. Don’t rely on given milestones or you'll end up with way too much to do in the month of the semester.
- Use your local computer when possible, especially when doing a code demonstration. CRF servers are not very robust and should not be relied upon.
- Hangovers do not mix well with a 10am meeting.

7.4 Nabil Shahid

- Coding conventions among group members is a good thing! That way you won't be confused every time you need to work on a file that has been previously edited or written by another group member.
- No need to be overambitious about what time in the morning the group meeting should be. 10am doesn't REALLY mean 10am does it?
- CLIC gets crowded around the time O.S. assignments are due.
- Google Groups is your friend.
- Make sure you compile before you try to run (Matt...)

Appendix: Code Listing

A.1 grammar.g

```
/*
 * walker.g : the AST walker.
 */

{
import java.io.*;
import java.util.*;
}

class MrpWalker extends TreeParser;
options{
    importVocab = Mrp;
}

{
   static MrpDataType null_data = new MrpDataType( "<NULL>" );
    MrpInterpreter ipt = new MrpInterpreter();
    PrintWriter w = new PrintWriter(System.out);
}

expr returns [ MrpDataType r ]
{
    MrpDataType a, b, type; //LHS, RHS
    Vector v;
    MrpDataType[] x;
    String s = null;
    String[] sx;
    r = null_data; //result
}
        : #("or" a=expr right_or:.)
            {
                if ( a instanceof MrpBool )
                    r = ( ((MrpBool)a).var ? a : expr(#right_or) );
                else
                    r = a.or( expr(#right_or) );
            }
        | #("and" a=expr right_and:.)
            {
                if ( a instanceof MrpBool )
                    r = ( ((MrpBool)a).var ? expr(#right_and) : a );
                else
                    r = a.and( expr(#right_and) );
            }
        | #("not" a=expr)             { r = a.not(); }
        | #(GE a=expr b=expr)         { r = a.ge( b ); }
        | #(LE a=expr b=expr)         { r = a.le( 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.times( b ); }
        | #(LDV a=expr b=expr)        { r = a.lfracts( b ); }
        | #(MOD a=expr b=expr)        { r = a.modulus( b ); }
        | #(UPLUS a=expr)            { r = a; }
        | #(UMINUS a=expr)           { r = a.uminus(); }
        | #(PLUSEQ a=expr b=expr)    { r = a.add( b ); }
        | #(MINUSEQ a=expr b=expr)   { r = a.sub( b ); }
        | #(MULTEQ a=expr b=expr)    { r = a.mul( b ); }
        | #(LDVEQ a=expr b=expr)     { r = a.ldiv( b ); }
        | #(MODEQ a=expr b=expr)     { r = a.rem( b ); }
        | #(COLON (c1:. (c2:.)?)?)
            {
                r = MrpRange.create( (null==#c1) ? null : expr(#c1),
                                    (null==#c2) ? null : expr(#c2) );
            }
        | #(ASGN a=expr b=expr)      { r = ipt.assign( a, b ); }
        | #(ASSIGNMENT a=expr b=expr)      {
            System.out.println("regular assign");
            r = ipt.assign( a, b ); }
        | #(AASSIGNMENT {System.out.println("begin assigning array");}
            a=expr b=expr)
        {
            System.out.println("assigning array");
            r = ipt.arraySet( a, b );
            System.out.println("assigned array");}
        | #(FUNC_CALL a=expr x=mexpr)
        {
            a.print(w);
            w.flush();
            r = ipt.funcInvoke( this, a, x );
            System.out.println("Function " + a.name + " called");
        }
        | num:NUMBER                 {
            System.out.println("found number: " + num.getText());
            r = ipt.getNumber( num.getText() ); }
        | str:STRING                 { r = new MrpString( str.getText() ); }
        | "true"                     { r = new MrpBool( true ); }
        | "false"                    { r = new MrpBool( false ); }

        | #(DECLARATION type=expr a=expr
            { System.out.println("Found declaration");
                r = ipt.varRegister( type.name, a.name );
            System.out.println("Past declaration");}
              (b=expr                    { r = ipt.assign( r, b ); }
              )?
           )
        | #(ADECLARATION type=expr a=expr { r = ipt.varRegister( "array" + type.name,
a.name ); }
              (b=expr                     {
                    b = new MrpArray(type.name, ((MrpInt)b).var);
                    try { r = ipt.assign( r, b ); }
                                            catch(MrpException me) { ; }
                                        }//overwriting variable exception
              )?
           )
        | #(id:ID
            { System.out.println("Found id");
                r = ipt.getVariable( id.getText() ); }
             (a=expr                {
                    System.out.println("array found");
                    r = ((MrpArray)r).get(((MrpInt)a).var);
                    System.out.println("past array");
                }//array
              )?
          )
        | #("for" x=mexpr forbody:.)
            {
                MrpInt[] values = ipt.forInit( x );
                while ( ipt.forCanProceed( x, values ) )
                {
                    r = expr( #forbody );
                    ipt.forNext( x, values );
                }
                ipt.forEnd( x );
            }
        | #("if" a=expr thenp:. (elsep:.)?)
            {
                if ( !( a instanceof MrpBool ) )
                    return a.error( "if: expression should be bool" );
                if ( ((MrpBool)a).var )
                    r = expr( #thenp );
                else if ( null != elsep )
                    r = expr( #elsep );
            }
        | #(STATEMENT (stmt:.
                { System.out.println("found statement");
                    if ( ipt.canProceed() ) r = expr(#stmt); } )*)
        | #("while" whiletest:. whilebody:.
                    (whileid:ID        { s = whileid.getText(); }
                    )?
           )
            {
                while ( ipt.canProceed() )
                {
                    a = expr( #whiletest );
                    if(!((MrpBool)a).var)
                        break;
                    r = expr( #whilebody );
                    ipt.loopNext( s );
                }
                ipt.loopEnd( s );
            }
        | #("break" (breakid:ID      { s = breakid.getText(); }
                    )?
           )                         { ipt.setBreak( s ); }
        | #("continue" (contid:ID    { s = contid.getText(); }
                       )?
           )                         { ipt.setContinue( s ); }
        | #("return" ( a=expr        { r = ipt.rvalue( a ); }
                     )?
           )                         { ipt.setReturn( null ); }
        | #("func" ftype:ID fname:ID sx=vlist fbody:.)
            { ipt.funcRegister( fname.getText(), sx, ftype.getText(), #fbody ); }
        ;

mexpr returns [ MrpDataType[] rv ]
{
    MrpDataType a;
    rv = null;
    Vector v;
}
        : #(EXPR_LIST         { v = new Vector(); }
                ( a=expr      { v.add( a ); }
                )*
           )                  { rv = ipt.convertExprList( v ); }
        | a=expr              { rv = new MrpDataType[1]; rv[0] = a; }
        |  #(FOR_CON          { v = new Vector(); }
                ( s:ID a=expr { a.setName( s.getText() ); v.add(a); }
                )+
            )                 { rv = ipt.convertExprList( v ); }
        ;

vlist returns [ String[] sv ]
{
    Vector v;
    sv = null;
}
        : #(VAR_LIST     { v = new Vector(); }
              (s:ID      { v.add( s.getText() ); }
              )*
          )              { sv = ipt.convertVarList( v ); }
        ;
```

A.2 walker.g

```
/*
 * walker.g : the AST walker.
 */

{
import java.io.*;
import java.util.*;
}

class MrpWalker extends TreeParser;
options{
    importVocab = Mrp;
}

{
   static MrpDataType null_data = new MrpDataType( "<NULL>" );
    MrpInterpreter ipt = new MrpInterpreter();
    PrintWriter w = new PrintWriter(System.out);
}

expr returns [ MrpDataType r ]
{
    MrpDataType a, b, type; //LHS, RHS
    Vector v;
    MrpDataType[] x;
    String s = null;
    String[] sx;
    r = null_data; //result
}
        : #("or" a=expr right_or:.)
            {
                if ( a instanceof MrpBool )
                    r = ( ((MrpBool)a).var ? a : expr(#right_or) );
                else
                    r = a.or( expr(#right_or) );
            }
        | #("and" a=expr right_and:.)
            {
                if ( a instanceof MrpBool )
                    r = ( ((MrpBool)a).var ? expr(#right_and) : a );
                else
                    r = a.and( expr(#right_and) );
            }
        | #("not" a=expr)            { r = a.not(); }
        | #(GE a=expr b=expr)        { r = a.ge( b ); }
        | #(LE a=expr b=expr)        { r = a.le( 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.times( b ); }
        | #(LDV a=expr b=expr)       { r = a.lfracts( b ); }
        | #(MOD a=expr b=expr)       { r = a.modulus( b ); }
        | #(UPLUS a=expr)            { r = a; }
        | #(UMINUS a=expr)           { r = a.uminus(); }
        | #(PLUSEQ a=expr b=expr)    { r = a.add( b ); }
        | #(MINUSEQ a=expr b=expr)   { r = a.sub( b ); }
        | #(MULTEQ a=expr b=expr)    { r = a.mul( b ); }
        | #(LDVEQ a=expr b=expr)     { r = a.ldiv( b ); }
        | #(MODEQ a=expr b=expr)     { r = a.rem( b ); }
        | #(COLON (c1:. (c2:.)?)?)
            {
                r = MrpRange.create( (null==#c1) ? null : expr(#c1),
                                     (null==#c2) ? null : expr(#c2) );
            }
        | #(ASGN a=expr b=expr)      { r = ipt.assign( a, b ); }
        | #(ASSIGNMENT a=expr b=expr)      {
            r = ipt.assign( a, b ); }
        | #(AASSIGNMENT {System.out.println("begin assigning array");}
            a=expr b=expr)
        {
            r = ipt.arraySet( a, b );
        | #(FUNC_CALL a=expr x=mexpr)
        {
            a.print(w);
            w.flush();
            r = ipt.funcInvoke( this, a, x );
        }
        | num:NUMBER                 {
            r = ipt.getNumber( num.getText() ); }
        | str:STRING                 { r = new MrpString( str.getText() ); }
        | "true"                     { r = new MrpBool( true ); }
        | "false"                    { r = new MrpBool( false ); }

        | #(DECLARATION type=expr a=expr
            {
              r = ipt.varRegister( type.name, a.name );
            }
              (b=expr                    { r = ipt.assign( r, b ); }
              )?
           )
        | #(ADECLARATION type=expr a=expr { r = ipt.varRegister( "array" + type.name,
a.name ); }
              (b=expr                     {
                    b = new MrpArray(type.name, ((MrpInt)b).var);
                    try { r = ipt.assign( r, b ); }
```

```
                                  catch(MrpException me) { ; }
                                }//overwriting variable exception
      )?
   )
 | #(id:ID
     {
         r = ipt.getVariable( id.getText() ); }
     (a=expr                {
             r = ((MrpArray)r).get(((MrpInt)a).var);
         }//array
     )?
   )
 | #("for" x=mexpr forbody:.)
     {
         MrpInt[] values = ipt.forInit( x );
         while ( ipt.forCanProceed( x, values ) )
         {
             r = expr( #forbody );
             ipt.forNext( x, values );
         }
         ipt.forEnd( x );
     }
 | #("if" a=expr thenp:. (elsep:.)?)
     {
         if ( !( a instanceof MrpBool ) )
             return a.error( "if: expression should be bool" );
         if ( ((MrpBool)a).var )
             r = expr( #thenp );
         else if ( null != elsep )
             r = expr( #elsep );
     }
 | #(STATEMENT (stmt:.
         {
             if ( ipt.canProceed() ) r = expr(#stmt); } )*)
 | #("while" whiletest:. whilebody:.
             (whileid:ID        { s = whileid.getText(); }
             )?
    )
     {
         while ( ipt.canProceed() )
         {
             a = expr( #whiletest );
             if(!((MrpBool)a).var)
                 break;
             r = expr( #whilebody );
             ipt.loopNext( s );
         }
         ipt.loopEnd( s );
     }
 | #("break" (breakid:ID      { s = breakid.getText(); }
             )?
    )                         { ipt.setBreak( s ); }
 | #("continue" (contid:ID    { s = contid.getText(); }
                )?
    )                         { ipt.setContinue( s ); }
 | #("return" ( a=expr        { r = ipt.rvalue( a ); }
              )?
    )                         { ipt.setReturn( null ); }
 | #("func" ftype:ID fname:ID sx=vlist fbody:.)
     { ipt.funcRegister( fname.getText(), sx, ftype.getText(), #fbody ); }
```

```
        ;

mexpr returns [ MrpDataType[] rv ]
{
    MrpDataType a;
    rv = null;
    Vector v;
}
        : #(EXPR_LIST          { v = new Vector(); }
                ( a=expr       { v.add( a ); }
                )*
             )                 { rv = ipt.convertExprList( v ); }
        | a=expr               { rv = new MrpDataType[1]; rv[0] = a; }
        |  #(FOR_CON           { v = new Vector(); }
                ( s:ID a=expr { a.setName( s.getText() ); v.add(a); }
                )+
              )                { rv = ipt.convertExprList( v ); }
        ;

vlist returns [ String[] sv ]
{
    Vector v;
    sv = null;
}
        : #(VAR_LIST      { v = new Vector(); }
              (s:ID       { v.add( s.getText() ); }
              )*
           )              { sv = ipt.convertVarList( v ); }
        ;
```

A.3 MrpInterpreter.java

```
import java.util.*;
import java.io.*;
import antlr.CommonAST;
import antlr.collections.AST;
import antlr.RecognitionException;
import antlr.TokenStreamException;
import antlr.TokenStreamIOException;

/** Interpreter routines that is called directly from the tree walker.
 */
class MrpInterpreter {
    MrpSymbolTable vsymt;//variable symbol table
    MrpSymbolTable fsymt;//function symbol table

    final static int fc_none = 0;
    final static int fc_break = 1;
    final static int fc_continue = 2;
    final static int fc_return = 3;

    private int control = fc_none;
    private String label;

    private Random random = new Random();

    public MrpInterpreter() {
        vsymt = new MrpSymbolTable( null, null );
        fsymt = new MrpSymbolTable( null, null );
        registerInternal();
    }

    public MrpDataType[] convertExprList( Vector v ) {
        /* Note: expr list can be empty */
        MrpDataType[] x = new MrpDataType[v.size()];
        for ( int i=0; i<x.length; i++ )
            x[i] = (MrpDataType) v.elementAt( i );
        return x;
    }

    public static String[] convertVarList( Vector v ) {
        /* Note: var list can be empty */
        String[] sv = new String[ v.size() ];
        for ( int i=0; i<sv.length; i++ )
            sv[i] = (String) v.elementAt( i );
        return sv;
    }

    public static MrpDataType getNumber( String s ) {

      if ( s.indexOf( '.' ) >= 0
             || s.indexOf( 'e' ) >= 0 || s.indexOf( 'E' ) >= 0 )
          return new MrpDouble( Double.parseDouble( s ) );
        return new MrpInt( Integer.parseInt( s ) );
    }

    public MrpDataType getVariable( String s ) {
        // default static scoping
        MrpDataType x = vsymt.getValue( s, true, 0 );
        if ( null == x )
          if( (x = fsymt.getValue( s, true, 0 ) ) == null )
            return new MrpVariable( s );
        return x;
    }


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

    public MrpDataType assign( MrpDataType a, MrpDataType b ) {
        if ( null != a.name )
        {
            if( !validAssignment( a, b ) )
              return a.error( b, "=" );
            MrpDataType x = rvalue( b );
            x.setName( a.name );
            vsymt.setValue( x.name, x, true, 0 );  // scope?
            return x;
        }

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

    public MrpDataType arraySet( MrpDataType a, MrpDataType b ) {
      if( !validAssignment(a, b) )
          throw new MrpException("Types of a and b to not match");
      if( a instanceof MrpInt )
          ((MrpInt)a).var = ((MrpInt)b).var;
      else if(a instanceof MrpDouble)
          ((MrpDouble)a).var = ((MrpDouble)b).var;
      else if(a instanceof MrpString)
          ((MrpString)a).var = ((MrpString)b).var;
      else if(a instanceof MrpBool)
          ((MrpBool)a).var = ((MrpBool)b).var;
      else if(a instanceof MrpLog) {
          MrpLog aLog = (MrpLog)a;
          MrpLog bLog = (MrpLog)b;
          aLog.clear();
          for(int i = 0; i < bLog.logLength(); i ++)
            aLog.setMessage(i, bLog.getMessage(i));
      }
      else if(a instanceof MrpMessage) {
          System.out.println("Trying to set message in arrayset");
          ((MrpMessage)a).setText(((MrpMessage)b).getText());
          ((MrpMessage)a).setTime(((MrpMessage)b).getTime());
          ((MrpMessage)a).setMessageName(((MrpMessage)b).getMessageName());
      }
      else if(a instanceof MrpArray)
          throw new MrpException("Arrays of arrays are not allowed");
      else
          a.error(b, " = ");

      return a;
    }



    private boolean validAssignment( MrpDataType a, MrpDataType b) {
        if( a.typename().equals(b.typename() ) )
            return true;
        /*right now you must explicitly cast before assigning
        if( ( a.typename().equals( "int" ) && b.typename().equals( "double" ) )
          || ( a.typename().equals( "double" ) && b.typename().equals( "int" ) )
            return true;
        */
        return false;
    }

    public MrpDataType funcInvoke(
        MrpWalker walker,  MrpDataType func,
        MrpDataType[] params ) throws antlr.RecognitionException {

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

        // Is this function an internal function?
        // Names of formal args are not necessary for internal functions.
        if ( ((MrpFunction)func).isInternal() )
            return execInternal( ((MrpFunction)func).getInternalId(),
                                 params );

        // otherwise check numbers of actual and formal arguments
        String[] args = ((MrpFunction)func).getArgs();
        if ( args.length != (params.length * 2) )//have to match types also
            return func.error( "unmatched length of parameters" );
      for(int i = 0; i < params.length; i ++)
          if(!params[i].typename().equals(args[2 * i]))
            throw new MrpException( "Unmatched types in function. Found: " +
                          params[i].typename() + " needs " + args[2 * i]);

        // create a new symbol table
        vsymt = new MrpSymbolTable( ((MrpFunction)func).getParentSymbolTable(),
                                  vsymt );

        // assign actual parameters to formal arguments
        for ( int i=0; i<params.length; i++ )
        {
            MrpDataType d = rvalue( params[i] );
            d.setName( args[(2 * i) + 1] );
            vsymt.setValue( args[(2 * i) + 1], d, false, 0 );
        }

        // call the function body
        MrpDataType r = walker.expr( ((MrpFunction)func).getBody() );

        // no break or continue can go through the function
        if ( control == fc_break || control == fc_continue )
            throw new MrpException( "nowhere to break or continue" );

        // if a return was called
        if ( control == fc_return )
            tryResetFlowControl( ((MrpFunction)func).name );

        // remove this symbol table and return
        vsymt = vsymt.dynamicParent();

      if(!r.typename().equals(((MrpFunction)func).returnType()))
          throw new MrpException( "return types do not match" );

        return r;
    }


    public void funcRegister( String name, String[] args, String type,
                              AST body ) {
        if( fsymt.put( name, new MrpFunction( name, args, type,
                                              body, fsymt ) ) != null )
          throw new MrpException( "illegal operation: function "
                               + name + " already declared.");
    }

    public MrpDataType varRegister( String type, String name)
    {
        MrpDataType x;
        if( type.equals( "int" ) )
            x = new MrpInt( Integer.MIN_VALUE );
        else if( type.equals( "double" ) )
            x = new MrpDouble( Double.NaN );
        else if( type.equals( "string" ) )
            x = new MrpString( "" );
        else if( type.equals( "message" ) )
            x = new MrpMessage( );
        else if( type.equals( "log" ) )
            x = new MrpLog( 3 );
            /*
        else if( type.equals( "logformat" ) )
            x = new MrpLogformat( null );
            */
        else if( type.substring(0, 5).equals( "array" ) )
            x = new MrpArray( type.substring( 5 ) );
        else
            throw new MrpException ( "illegal operation: type " + type
                                  + " does not exist" );
        x.setName( name );
        if( vsymt.put( name, x ) != null)
            throw new MrpException ("illegal operation: " + name
                                  + " is already declared" );

      return x;
    }

    public void setBreak( String label ) {
        this.label = label;
        control = fc_break;
    }

    public void setContinue( String label ) {
        this.label = label;
        control = fc_continue;
    }

    public void setReturn( String label ) {
        this.label = label;
        control = fc_return;
    }

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

    public void loopNext( String loop_label ) {
        if ( control == fc_continue )
            tryResetFlowControl( loop_label );
    }

    public void loopEnd( String loop_label ) {
        if ( control == fc_break )
            tryResetFlowControl( loop_label );
    }

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

    public MrpInt[] forInit( MrpDataType[] mexpr ) {
        // very much like function call
        for ( int i=0; i<mexpr.length; i++ )
            if ( ! ( mexpr[i] instanceof MrpRange )  )
            {
                mexpr[i].error( "for: [range expected]" );
                return null;
            }

        // create a new symbol table
```

```
        vsymt = new MrpSymbolTable( vsymt, vsymt );

        MrpInt[] x = new MrpInt[mexpr.length];
        for ( int i=0; i<mexpr.length; i++ )
        {
            x[i] = new MrpInt( ((MrpRange)mexpr[i]).range.first() );
            x[i].setName( mexpr[i].name );
            vsymt.setValue( mexpr[i].name, x[i], false, 0 );
        }

        vsymt.setReadOnly();
        return x;
    }

    public boolean forCanProceed( MrpDataType[] mexpr, MrpInt[] values ) {
        if ( control != fc_none )
            return false;

        for ( int i=mexpr.length-1; ; i-- )
        {
            if ( ((MrpRange)mexpr[i]).range.contain( values[i].var ) )
                return true;
            if ( 0 == i )
                return false;

            values[i].var = ((MrpRange)mexpr[i]).range.first();
            values[i-1].var =
                ((MrpRange)mexpr[i-1]).range.next( values[i-1].var );
        }
    }

    public void forNext( MrpDataType[] mexpr, MrpInt[] values ) {

        values[ values.length-1 ].var++;
        if ( control == fc_continue )
            tryResetFlowControl( mexpr[0].name );
    }

    public void forEnd( MrpDataType[] mexpr ) {
        if ( control == fc_break )
            tryResetFlowControl( mexpr[0].name );

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

    public MrpDataType execInternal( int id, MrpDataType[] params ) {
        return MrpInternalFunction.run( fsymt, id, params );
    }

    public void registerInternal() {
        MrpInternalFunction.register( fsymt );
    }

}
```

A.4 MrpSymbolTable.java

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

```
/**
 * Symbol table class: dual parent supported: static and dynamic
 */
class MrpSymbolTable extends HashMap {
    MrpSymbolTable static_parent, dynamic_parent;
    boolean read_only;

    public MrpSymbolTable( MrpSymbolTable sparent, MrpSymbolTable dparent ) {
        static_parent = sparent;
        dynamic_parent = dparent;
        read_only = false;
    }

    public void setReadOnly() {
        read_only = true;
    }

    public final MrpSymbolTable staticParent() {
        return static_parent;
    }

    public final MrpSymbolTable dynamicParent() {
        return dynamic_parent;
    }

    public final MrpSymbolTable parent( boolean is_static ) {
        return is_static ? static_parent : dynamic_parent;
    }

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

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

        if ( level < 0 )
        {
            // global variable
            while ( null != st.static_parent )
                st = st.parent( is_static );
        }
        else
        {
            // local variable
            for ( int i=level; i>0; i-- )
            {
                while ( st.read_only )
                {
                    st = st.parent( is_static );
                }

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

    public final MrpDataType getValue( String name, boolean is_static,
                                      int level ) {
        MrpSymbolTable 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 (MrpDataType) x;
    }

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

        MrpSymbolTable st = gotoLevel( level, is_static );
        while ( st.read_only )
        {
            st = st.parent( is_static );
        }

        st.put( name, data );
    }

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

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

A.5 MrpDataType.java

```
import java.io.PrintWriter;

/**
* The base data type class (also a meta class)
 *
 * Error messages are generated here.
 *
 * Adapted from MX language.
 */
public class MrpDataType
{
    String name;   // used in hash table

    public MrpDataType() {
        name = null;
    }

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

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

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

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

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

    public MrpDataType error( MrpDataType b, String msg ) {
        if ( null == b )
            return error( msg );
        throw new MrpException(
                               "illegal operation: " + msg
                               + "( <" + typename() + "> "
                               + ( name != null ? name : "<?>" )
                               + " and "
                               + "<" + typename() + "> "
                               + ( name != null ? name : "<?>" )
                               + " )" );
    }

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

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

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

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

    public MrpDataType assign( MrpDataType b ) {
        return error( b, "=" );
```

```
}

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

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

public MrpDataType add( MrpDataType b ) {
    return error( b, "+=" );
}

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

public MrpDataType sub( MrpDataType b ) {
    return error( b, "-=" );
}

public MrpDataType times( MrpDataType b ) {
    return error( b, "*" );
}

public MrpDataType mul( MrpDataType b ) {
    return error( b, "*=" );
}

public MrpDataType lfracts( MrpDataType b ) {
    return error( b, "/" );
}

public MrpDataType ldiv( MrpDataType b ) {
    return error( b, "/=" );
}

public MrpDataType modulus( MrpDataType b ) {
    return error( b, "%" );
}

public MrpDataType rem( MrpDataType b ) {
    return error( b, "%=" );
}

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

public MrpDataType ge( MrpDataType b ) {
    return error( b, ">=" );
}

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

public MrpDataType le( MrpDataType b ) {
    return error( b, "<=" );
    }

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

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

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

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

    public MrpDataType not() {
        return error( "not" );
    }
}
```

A.6 MrpVariable.java

```
import java.io.PrintWriter;

/**
* The wrapper class for unsigned variables
 */
class MrpVariable extends MrpDataType {
    public MrpVariable( String name ) {
        super( name );
    }

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

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

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

A.7 MrpBool.java

```
import java.io.PrintWriter;

/**
* The wrapper class for boolean. Adapted from MX language.
 */
class MrpBool extends MrpDataType {
    boolean var;
    MrpBool( boolean var ) {
        this.var = var;
    }

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

    public MrpDataType copy() {
        return new MrpBool( var );
    }

    public void print( PrintWriter w ) {
        if ( name != null )
            w.print( name + " = " );
        w.println( var ? "true" : "false" );
    }


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

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

    public MrpDataType not() {
        return new MrpBool( !var );
    }

    public MrpDataType eq( MrpDataType b ) {
        // not exclusive or
        if ( b instanceof MrpBool )
            return new MrpBool( ( var && ((MrpBool)b).var )
                                || ( !var && !((MrpBool)b).var ) );
        return error( b, "==" );
    }

    public MrpDataType ne( MrpDataType b ) {
        // exclusive or
        if ( b instanceof MrpBool )
            return new MrpBool( ( var && !((MrpBool)b).var )
                                || ( !var && ((MrpBool)b).var ) );
        return error( b, "!=" );
    }
}
```

A.8 MrpDouble.java

```
import java.io.PrintWriter;

/**
* The wrapper class for double. Adapted from Mrp language.
 */
class MrpDouble extends MrpDataType {
```

```
double var;

public MrpDouble( double x ) {
    var = x;
}

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

public MrpDataType copy() {
    return new MrpDouble( var );
}

public static double doubleValue( MrpDataType b ) {
    if ( b instanceof MrpDouble )
        return ((MrpDouble)b).var;
    if ( b instanceof MrpInt )
        return (double) ((MrpInt)b).var;
    b.error( "cast to double" );
    return 0;
}

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

public MrpDataType uminus() {
    return new MrpDouble( -var );
}

public MrpDataType plus( MrpDataType b ) {
    return new MrpDouble( var + doubleValue(b) );
}

public MrpDataType add( MrpDataType b ) {
    var += doubleValue( b );
    return this;
}

public MrpDataType minus( MrpDataType b ) {
    return new MrpDouble( var - doubleValue(b) );
}

public MrpDataType sub( MrpDataType b ) {
    var -= doubleValue( b );
    return this;
}

public MrpDataType times( MrpDataType b ) {
    return new MrpDouble( var * doubleValue(b) );
}

public MrpDataType mul( MrpDataType b ) {
    var *= doubleValue( b );
    return this;
}
```

```
    public MrpDataType lfracts( MrpDataType b ) {
        return new MrpDouble( var / doubleValue(b) );
    }

    public MrpDataType rfracts( MrpDataType b ) {
        return lfracts( b );
    }

    public MrpDataType ldiv( MrpDataType b ) {
        var /= doubleValue(b);
        return this;
    }

    public MrpDataType rdiv( MrpDataType b ) {
        return ldiv( b );
    }

    public MrpDataType modulus( MrpDataType b ) {
        return new MrpDouble( var % doubleValue(b) );
    }


    public MrpDataType rem( MrpDataType b ) {
        var %= doubleValue( b );
        return this;
    }

    public MrpDataType gt( MrpDataType b ) {
        return new MrpBool( var > doubleValue(b) );
    }

    public MrpDataType ge( MrpDataType b ) {
        return new MrpBool( var >= doubleValue(b) );
    }

    public MrpDataType lt( MrpDataType b ) {
        return new MrpBool( var < doubleValue(b) );
    }

    public MrpDataType le( MrpDataType b ) {
        return new MrpBool( var <= doubleValue(b) );
    }

    public MrpDataType eq( MrpDataType b ) {
        return new MrpBool( var == doubleValue(b) );
    }

    public MrpDataType ne( MrpDataType b ) {
        return new MrpBool( var != doubleValue(b) );
    }
}
```

A.9 MrpInt.java

```
import java.io.PrintWriter;

/**
* The wrapper class of int. Adapted from MX language.
 */
class MrpInt extends MrpDataType {
```

```
int var;

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

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

public MrpDataType copy() {
    return new MrpInt( var );
}

public static int intValue( MrpDataType b ) {
    if ( b instanceof MrpDouble )
        return (int) ((MrpDouble)b).var;
    if ( b instanceof MrpInt )
        return ((MrpInt)b).var;
    b.error( "cast to int" );
    return 0;
}

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

public MrpDataType uminus() {
    return new MrpInt( -var );
}

public MrpDataType plus( MrpDataType b ) {
    if ( b instanceof MrpInt )
        return new MrpInt( var + intValue(b) );
    return new MrpDouble( var + MrpDouble.doubleValue(b) );
}

public MrpDataType add( MrpDataType b ) {
    var += intValue( b );
    return this;
}

public MrpDataType minus( MrpDataType b ) {
    if ( b instanceof MrpInt )
        return new MrpInt( var - intValue(b) );
    return new MrpDouble( var - MrpDouble.doubleValue(b) );
}

public MrpDataType sub( MrpDataType b ) {
    var -= intValue( b );
    return this;
}

public MrpDataType times( MrpDataType b ) {
    if ( b instanceof MrpInt )
        return new MrpInt( var * intValue(b) );
    return new MrpDouble( var * MrpDouble.doubleValue(b) );
}
```

```
public MrpDataType mul( MrpDataType b ) {
    var *= intValue( b );
    return this;
}

public MrpDataType lfracts( MrpDataType b ) {
    if ( b instanceof MrpInt )
        return new MrpInt( var / intValue(b) );
    return new MrpDouble( var / MrpDouble.doubleValue(b) );
}

public MrpDataType rfracts( MrpDataType b ) {
    return lfracts( b );
}

public MrpDataType ldiv( MrpDataType b ) {
    var /= intValue(b);
    return this;
}

public MrpDataType rdiv( MrpDataType b ) {
    return ldiv( b );
}

public MrpDataType modulus( MrpDataType b ) {
    if ( b instanceof MrpInt )
        return new MrpInt( var % intValue(b) );
    return new MrpDouble( var % MrpDouble.doubleValue(b) );
}


public MrpDataType rem( MrpDataType b ) {
    var %= intValue( b );
    return this;
}

public MrpDataType gt( MrpDataType b ) {
    if ( b instanceof MrpInt )
        return new MrpBool( var > intValue(b) );
    return b.lt( this );
}

public MrpDataType ge( MrpDataType b ) {
    if ( b instanceof MrpInt )
        return new MrpBool( var >= intValue(b) );
    return b.le( this );
}

public MrpDataType lt( MrpDataType b ) {
    if ( b instanceof MrpInt )
        return new MrpBool( var < intValue(b) );
    return b.gt( this );
}

public MrpDataType le( MrpDataType b ) {
    if ( b instanceof MrpInt )
        return new MrpBool( var <= intValue(b) );
    return b.ge( this );
}
```

```
    public MrpDataType eq( MrpDataType b ) {
        if ( b instanceof MrpInt )
            return new MrpBool( var == intValue(b) );
        return b.eq( this );
    }

    public MrpDataType ne( MrpDataType b ) {
        if ( b instanceof MrpInt )
            return new MrpBool( var != intValue(b) );
        return b.ne( this );
    }
}
```

A.10 MrpString.java

```
import java.io.PrintWriter;

/**
* The wrapper class for string. Adapted from MX language.
 */
class MrpString extends MrpDataType {
    String var;

    public MrpString( String str ) {
        if(str == null || str.length() < 2) {
            var = str;
        }
        else if((str.charAt(0) == '\"')
                && str.charAt(str.length() - 1) == '\"') {
            var = str.substring(1, str.length() - 1);
        }
        else {
            var = str;
        }
    }

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

    public MrpDataType copy() {
        return new MrpString( var );
    }

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

    public MrpDataType plus( MrpDataType b ) {
        if ( b instanceof MrpString )
            return new MrpString( var + ((MrpString)b).var );

        return error( b, "+" );
    }

    public MrpDataType add( MrpDataType b ) {
        if ( b instanceof MrpString )
        {
            var = var + ((MrpString)b).var;
            return this;
        }

        return error( b, "+=" );
    }
}
```

A.11 MrpArray.java

```
import java.io.PrintWriter;
import java.util.Vector;
import java.util.Iterator;

/**
* The wrapper class for array. Adapted from MX language.
 */

class MrpArray extends MrpDataType {
    MrpDataType[] array;
    String type;

    public MrpArray( String t ) {
        type = t;
    }

    public MrpArray( String t, int size ) {
        array = new MrpDataType[size];
        if(t.equals("int"))
            for(int i = 0; i < size; i ++)
                array[i] = new MrpInt(Integer.MIN_VALUE);
        else if(t.equals("double"))
            for(int i = 0; i < size; i ++)
                array[i] = new MrpDouble(Double.MIN_VALUE);
        else if(t.equals("string"))
            for(int i = 0; i < size; i ++)
                array[i] = new MrpString("");
        else if(t.equals("message"))
            for(int i = 0; i < size; i ++)
                array[i] = new MrpMessage();
        else if(t.equals("log"))
            for(int i = 0; i < size; i ++)
                array[i] = new MrpLog(3);
        else if(t.equals("bool"))
            for(int i = 0; i < size; i ++)
                array[i] = new MrpBool(true);
        type = t;
    }

    public MrpArray( MrpDataType[] newArray, String t) {
        array = newArray;
        type = t;
    }

    public String typename() {
        return type + " array";
    }
    public MrpDataType copy() {
        MrpDataType[] newArray = new MrpDataType[array.length];
        for( int i = 0; i < array.length; i ++ )
            newArray[i] = array[i].copy();
        return new MrpArray(newArray, type);
    }

    public void print( PrintWriter w ) {
        if ( name != null )
            w.println( name + " = " );
        for( int i = 0; i < array.length; i ++ )
            array[i].print( w );
    }

    public MrpDataType get( int index ) {
        if( index > array.length )
            throw new MrpException( "Illegal operation: " + index +
                                    " is larger than the size of the vector." );
        return array[index];
    }

    public void put( int index, MrpDataType x ) {
        if( index > array.length )
            throw new MrpException( "Illegal operation: " + index +
                                    " is larger than the size of the vector." );
        array[index] = x;
    }

}
```

A.12 MrpFunction.java

```
import java.io.PrintWriter;
import antlr.collections.AST;

/**
* The function data type (including internal functions)
 */
class MrpFunction extends MrpDataType {
    // we need a reference to the AST for the function entry
    String[] args;
    String returnType;
    AST body;             // body = null means an internal function.
    MrpSymbolTable pst;   // the symbol table of static parent
    int id;               // for internal functions only

    public MrpFunction( String name, String[] args, String type,
                        AST body, MrpSymbolTable pst) {
        super( name );
        this.args = args;
        returnType = type;
        this.body = body;
        this.pst = pst;
    }

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

    public final boolean isInternal() {
        return body == null;
    }

    public final int getInternalId() {
        return id;
    }

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

    public String returnType() {
        return returnType;
    }

    public MrpDataType copy() {
        return new MrpFunction( name, args, returnType, body, pst );
    }

    public void print( PrintWriter w ) {
        if ( body == null )
        {
            w.println( name + " = <internal-function> #" + id );
        }
        else
        {
            if ( name != null )
                w.print( name + " = " );
            w.print( "<function>(" );
            for ( int i=0; ; i++ )
            {
                w.print( args[i] );
                if ( i >= args.length - 1 )
                    break;
                w.print( "," );
            }
            w.println( ")" );
        }
    }

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

    public MrpSymbolTable getParentSymbolTable() {
        return pst;
    }

    public AST getBody() {
        return body;
    }
}
```

A.13 MrpInternalFunction.java

```
import java.util.Random;
import java.io.IOException;
import java.util.regex.Pattern;
import java.util.regex.Matcher;
import java.io.PrintWriter;

/** The internal functions
*/
class MrpInternalFunction {
    static Random random = new Random();

    final static int f_print = 0;
    final static int f_strlength = 1;
    final static int f_concat = 2;
    final static int f_substr = 3;
    final static int f_contains = 4;
    final static int f_createlog = 5;
    final static int f_open = 6;
    final static int f_save = 7;
    final static int f_convert = 8;
    final static int f_loglength = 9;
    final static int f_logformat = 10;
    final static int f_getname = 11;
    final static int f_gettext = 12;
    final static int f_gettime = 13;
    final static int f_createmessage = 14;
    final static int f_setname = 15;
    final static int f_settext = 16;
    final static int f_settime = 17;
    final static int f_getmessage = 18;
    final static int f_setmessage = 19;

    public static void register( MrpSymbolTable st ) {

        st.put( "print", new MrpFunction( "print", f_print ) );
        st.put( "strLength", new MrpFunction( "strLength", f_strlength ) );
        st.put( "concat", new MrpFunction( "concat", f_concat ) );
        st.put( "substr", new MrpFunction( "substr", f_substr ) );
        st.put( "contains", new MrpFunction( "contains", f_contains ) );
        st.put( "createLog", new MrpFunction( "createLog", f_createlog ) );
        st.put( "open", new MrpFunction( "open", f_open ) );
        st.put( "save", new MrpFunction( "save", f_save ) );
        st.put( "convert", new MrpFunction( "convert", f_convert ) );
        st.put( "logLength", new MrpFunction( "logLength", f_loglength ) );
        st.put( "logFormat", new MrpFunction( "logFormat", f_logformat ) );
        st.put( "getName", new MrpFunction( "getName", f_getname ) );
        st.put( "getText", new MrpFunction( "getText", f_gettext ) );
        st.put( "getTime", new MrpFunction( "getTime", f_gettime ) );
        st.put( "createMessage", new MrpFunction( "createMessage", f_createmessage ) );
        st.put( "setName", new MrpFunction( "setName", f_setname ) );
        st.put( "setText", new MrpFunction( "setName", f_settext ) );
        st.put( "setTime", new MrpFunction( "setTime", f_settime ) );
        st.put( "getMessage", new MrpFunction( "getMessage", f_getmessage ) );
        st.put( "setMessage", new MrpFunction( "setMessage", f_setmessage ) );
    }

    public static MrpDataType run( MrpSymbolTable st,
                                   int id, MrpDataType[] params ) {
        switch ( id )
        {
```

```
case f_print:
    PrintWriter stdOut = new PrintWriter(System.out);
    for ( int i=0; i<params.length; i++ )
        params[i].print(stdOut);
        stdOut.flush();
    return null;

case f_strlength:
    if( ( params.length != 1 ) || !( params[0] instanceof MrpString ) )
        throw new MrpException( "usage: strlen( string )" );
    return new MrpInt( ( ( MrpString )params[0] ).var.length() );

case f_concat://second param is checked in plus method
    if ( (params.length != 2 ) || !( params[0] instanceof MrpString ) )
        throw new MrpException( "usage: concat( string, string )" );
    return ( ( MrpString )params[0] ).plus( params[1] );

case f_substr:
    if( ( params.length != 3 ) || !( params[0] instanceof MrpString ) ||
        !( params[1] instanceof MrpInt ) || !(params[2] instanceof MrpInt ) )
        throw new MrpException( "usage: substr(string, int length, " +
                                "int offset)" ) ;
    return new MrpString( ( ( MrpString)params[0] ).var.
                          substring( ( ( MrpInt)params[1] ).var,
                                     ( ( MrpInt)params[2] ).var ) );

case f_contains:
    if( ( params.length != 2 ) || !( params[0] instanceof MrpString) ||
        !( params[1] instanceof MrpString) )
        throw new MrpException( "usage: contains(string, string)" );
    Pattern p = Pattern.compile( ( ( MrpString)params[1] ).var );
    Matcher m = p.matcher( ( ( MrpString)params[0] ).var );
    return new MrpBool( m.find() );

case f_createlog:
    if( ( params.length != 1 ) || !( params[0] instanceof MrpInt ) )
        throw new MrpException( "usage: createLog(format)" );
    return new MrpLog( ((MrpInt)params[0]).var );

case f_open:
    if( ( params.length != 3 ) || !( params[0] instanceof MrpString ) ||
        !( params[1] instanceof MrpString )  ||
        !( params[2] instanceof MrpInt ) )
        throw new MrpException( "usage: open(string logroot, " +
                                "string name, format)" );
    return new MrpLog( ((MrpString)params[0]).var,
                       ((MrpString)params[1]).var, ((MrpInt)params[2]).var);

case f_save:
    if( ( params.length != 1 ) && (params.length != 2) )
        throw new MrpException( "usage: save(log) overwrites the current " +
                                "log you opened\nor save(log, string path)");
    if( !(params[0] instanceof MrpLog ) ||
        ((params.length == 2) && !(params[1] instanceof MrpString)))
        throw new MrpException("usage: save(log) overwrites the current" +
                               " log you opened\nor save(log, string path)");
        if(params.length == 1)
            ((MrpLog)params[0]).save();
    else
        ((MrpLog)params[0]).save(((MrpString)params[1]).var);
    return null;

case f_convert:
    if( ( params.length != 2 ) ||
        !( params[0] instanceof MrpLog ) ||
        !( params[1] instanceof MrpInt ) )
        throw new MrpException ( "usage: convert(log, int logformat to)" );
    ((MrpLog)params[0]).convert(((MrpInt)params[1]).var);
    return params[0];

case f_loglength:
    if( ( params.length != 1 ) || !( params[0] instanceof MrpLog ) )
        throw new MrpException( "usage: logLength(log)" );
    return new MrpInt(((MrpLog)params[0]).logLength());


case f_logformat:
    if( ( params.length != 1 ) || !( params[0] instanceof MrpString ) )
        throw new MrpException( "usage: logformat(string)" );
    throw new MrpException ( "MrpLogformat not implemented. Please use " +
                             "an interger to represent a logformat");

case f_getname:
    if( (params.length != 1 ) || !( params[0] instanceof MrpMessage ) )
        throw new MrpException ( "usage: getMessageName(message)" );
    return new MrpString(((MrpMessage)params[0]).getMessageName());

case f_gettext:
    if( (params.length != 1 ) || !( params[0] instanceof MrpMessage ) )
        throw new MrpException ( "usage: getText(message)" );
    return new MrpString(((MrpMessage)params[0]).getText());

case f_gettime:
    if( (params.length != 1 ) || !( params[0] instanceof MrpMessage ) )
        throw new MrpException( "usage: getTime(message)" );
    return new MrpInt(((MrpMessage)params[0]).getTime());

case f_createmessage:
    if( params.length != 0 )
        throw new MrpException( "usage: createMessage()" );
    return new MrpMessage();

case f_setname:
    if( ( params.length != 2 ) || !( params[0] instanceof MrpMessage ) ||
        !( params[1] instanceof MrpString ) )
        throw new MrpException( "usage: setName(message, string)" );
    ((MrpMessage)params[0]).setMessageName(((MrpString)params[1]).var);
    return null;

case f_settext:
    if( ( params.length != 2 ) || !( params[0] instanceof MrpMessage ) ||
        !( params[1] instanceof MrpString ) )
        throw new MrpException( "usage: setText(message, string)" );
    ((MrpMessage)params[0]).setName(((MrpString)params[1]).var);
    return null;

case f_settime:
    if( ( params.length != 2 ) || !( params[0] instanceof MrpMessage ) ||
        !(params[1] instanceof MrpInt))
```

```
                throw new MrpException( "usage: setTime(message, int)");
            ((MrpMessage)params[0]).setTime(((MrpInt)params[1]).var);
            return null;


        case f_getmessage:
            if( (params.length != 2) || !(params[0] instanceof MrpLog) ||
                !(params[1] instanceof MrpInt) )
                throw new MrpException("usage: getMessage(log, int)");
            return ((MrpLog)params[0]).getMessage(((MrpInt)params[1]).var);


        case f_setmessage:
            if( ( params.length != 3 ) || !( params[0] instanceof MrpLog ) ||
                !(params[1] instanceof MrpInt) ||
                !(params[2] instanceof MrpMessage) )
                throw new MrpException( "usage: setMessage(log, int, message)");
            ((MrpLog)params[0]).setMessage(((MrpInt)params[1]).var,
                                           (MrpMessage)params[2] );
            return null;

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

}
```

A.14 MrpException.java

```
/**
 * Exception class: messages are generated in various classes
 */
class MrpException extends RuntimeException {
    MrpException( String msg ) {
        System.err.println( "Error: " + msg );
    }
}
```

A.15 MrpRange.java

```
import java.io.PrintWriter;

/**
* @version $Id: MrpRange.java,v 1.1 2003/11/30 19:41:15 jl1434 Exp $
 */
class MrpRange extends MrpDataType {
    Range range;

    MrpRange( Range range ) {
        this.range = range;
    }

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

    public MrpDataType copy() {
        return new MrpRange( range );
    }

    public static MrpDataType create( MrpDataType v1, MrpDataType v2 ) {
        int s = ( null == v1 ? 0 : MrpInt.intValue(v1) );

        if ( null != v2 )
        {
            int t = MrpInt.intValue(v2);
            if ( t >= 0 )
                return new MrpRange( new Range( s, t ) );
            return new MrpRange( new Range( s, t, 1 ) );
        }

        return new MrpRange( new Range( s, -1, 1 ) );
    }

    public static MrpDataType create( MrpDataType v1, MrpDataType v2,
                                      int stride ) {
        if ( v1 instanceof MrpInt && v2 instanceof MrpInt ) {
            if ( ((MrpInt)v2).var <= 0 )
                return v2.error( "invalid range" );
            return new MrpRange( new Range( ((MrpInt)v1).var,
                                            ((MrpInt)v2).var, stride ) );
        }

        return v1.error( v2, "cast to range components" );
    }

    public Range getRange( int n ) {
        /* Tricky short-cut: if range is empty, the "number" field stores
        * a reverse count: -1 means n-1, -2 means n-2, and so on.
        */
        if ( range.empty() )
            return new Range( range.first(), n + range.size() );

        return range;
    }

    public void print( PrintWriter w ) {
        if ( name != null )
            w.print( name + " = " );
        w.println( Integer.toString( range.first() ) + ":"
                   + range.size() + ":" + range.interval() );
    }

}
```

A.16 Range.java

```
/**
* The Range class representing a linear set of integers (usually indexes)
 *
 * Range is basically a triple (number,base,stride), where number
 * means the number of integers, base is the start integer, and
 * stride is the distance between to consecutive integers.
 *
 * Adapted from Mx language.
 */
public class Range implements Cloneable {
    private int number;
private int base;
private int stride;

/** constructor
    * @param first   the start integer of a range
    * @param last    the end integer of a range
    */
public Range( int first, int last ) {
    this.base = first;
    number = last - base + 1;
    if ( number < 0 )
    {
        number = -number;
        stride = -1;
    }
    else
        stride = 1;
}

/** constructor
    * @param first   the start integer of a range
    * @param size    the number of integers in the range
    * @param interval the distance between two consecutive integers
    */
public Range( int first, int size, int interval ) {
    this.base = first;
    this.number = size;
    this.stride = interval;
}

/** the first integer of the range
    */
public final int first() {
    return base;
}

/** the number of integers in the range
    */
public final int size() {
    return number;
}

/** the last integer of the range
    */
public final int last() {
    return base + (number-1) * stride;
}

/** the next integer of the current
    * @param n    current integer
    */
public final int next( int n ) {
    return n + stride;
}

/** the distance between two consecutive integers
    */
public final int interval() {
    return stride;
}
```

```
/** does a range contain a specified integer?
    * @param n    the specified integer
    * @return     a boolean value indicating the result
    */
public final boolean contain( int n ) {
    n -= base;
    if ( stride >= 0 )
        return n >= 0 && n < number * stride && 0 == n % stride;
    return n <= 0 && n > number * stride && 0 == (-n) % (-stride);
}

/** does a range is in bounded region
    * @param lower   the lower bound
    * @param upper   the upper bound
    * @return        a boolean value indicating the result
    */
public final boolean in( int lower, int upper ) {
    if ( stride >= 0 )
        return base >= lower && ( number - 1 ) * stride + base <= upper;
    return base <= upper && ( number - 1 ) * stride + base >= lower;
}

/** The smallest (not necessarily the first) integer in a range
    */
public int inf() {
    if ( stride >= 0 )
        return base;
    return base + (number-1) * stride;
}

/** The largest (not necessarily the last) integer in a range
    */
public int sup() {
    if ( stride <= 0 )
        return base;
    return base + (number-1) * stride;
}

/** is a range empty? (contains zero integers)
    * @return     a boolean value indicating the result
    */
public boolean empty() {
    return number <= 0;
}

/** convert a range to string for printing
    */
public String toString() {
    if ( 1 == stride )
    {
        if ( 1 == number )
            return Integer.toString( base );
        return Integer.toString( base ) + ":" + last();
    }

    return Integer.toString( base ) + ":"
    + ( stride>=0 ? "+" + stride : "-" + (-stride) )
    + ":" + number;
}
```

```
}
```

A.17 MrpLog.java

```
import java.io.*;
import java.util.*;
import java.util.regex.*;

public class MrpLog extends MrpDataType {

    private final int INITIAL = 200;
    private final int INCREMENT = 100;

    private Vector logData = new Vector(INITIAL, INCREMENT);
    private int logType;

    // Log formats:
    // 0 = AIM
    // 1 = Trillian
    // 2 = Fire
    // 3 = custom

    private int year;
    private int month;
    private int date;
    private int hours;
    private int minutes;
    private int seconds;

    private String message;
    private String sname;
    private String path;

    private boolean before12;

    private MrpMessage msg;

    public MrpLog (int logFormat) {
        logType = logFormat;
    }

    public MrpLog (Vector initLogData, int initLogFormat) {
        logType = initLogFormat;
        logData = (Vector) initLogData.clone();
    }

    public MrpLog (String logRoot, String screenName, int logFormat) {
        try {
            logType = logFormat;
            sname = screenName;

            String filename;
            String line;

            System.out.println(logRoot);
            System.out.println(screenName);

            // case when the log type is AIM
            if(logFormat == 0) {
                path = logRoot + screenName;
```

```
File[] fileList = new File(logRoot + screenName).listFiles();
if (fileList == null) {
    // no screenname exists error?
}
//for (int i = 0 ; i < 1 ; i++) {
for (int i = 0; i < fileList.length; i++) {
    filename = fileList[i].toString();
    FileReader fr = new FileReader(fileList[i]);
    BufferedReader br = new BufferedReader(fr);
    Pattern pattern;
    Matcher matcher;

    pattern = Pattern.compile("(\\d{4})-(\\d{2})-(\\d{2})");
    matcher = pattern.matcher(filename);

    if (matcher.find()) {
        year = Integer.parseInt(matcher.group(1));
        month = Integer.parseInt(matcher.group(2)) - 1;
        date = Integer.parseInt(matcher.group(3));
    }

    line = br.readLine();
    pattern = Pattern.compile("(\\d{2}):(\\d{2}):(\\d{2})");
    matcher = pattern.matcher(line);
    if (matcher.find()) {
        if (Integer.parseInt(matcher.group(1)) == 0) {
            before12 = false;
        } else {
            before12 = true;
        }
    }

    br.readLine();
    line = br.readLine();

    // yes... we are bootleg

    pattern = Pattern.compile("<HR></FONT>");
    matcher = pattern.matcher(line);

    line = matcher.replaceAll("");

    pattern = Pattern.compile("<HR>");
    matcher = pattern.matcher(line);

    line = matcher.replaceAll("<BR>");

    pattern = Pattern.compile("<BR><B></FONT>");
    matcher = pattern.matcher(line);

    // super mega html tag switch... i choose you!
    line = matcher.replaceAll("</FONT><BR><B>");

    matcher = pattern.matcher(line);

    line = matcher.replaceAll("</FONT><BR>");

    pattern = Pattern.compile("<BR>");
    matcher = pattern.matcher(line);
```

```
line = matcher.replaceAll("\n");

StringTokenizer st = new StringTokenizer(line, "\n");

while (st.hasMoreTokens()) {

    String newLine= st.nextToken();

    pattern = Pattern.compile("(signed (off|on) at \\d)|
         (Auto response from)");
    matcher = pattern.matcher(newLine);

    if (!(matcher.find())) {

        // now regexp on each token?

        pattern = Pattern.compile("<BODY BGCOLOR=\"#\\w{6}\"><B>
             <FONT COLOR=\"#\\w{6}\">(\\w*)</FONT>");

        matcher = pattern.matcher(newLine);
        if (matcher.find()) {
            screenName = matcher.group(1);
        }

        pattern = Pattern.compile("(\\d+):(\\d{2}):(\\d{2})
             (\\w{2})");
        matcher = pattern.matcher(newLine);
        if (matcher.find()) {
            hours = Integer.parseInt(matcher.group(1));
            if (matcher.group(4).equalsIgnoreCase("PM")) {
                hours = hours + 12;
            }

            if (before12 && (hours == 0)) {
                date++;
                before12 = false;
            }
            if (!before12 && (hours != 0)) {
                before12 = true;
            }

            minutes = Integer.parseInt(matcher.group(2));
            seconds = Integer.parseInt(matcher.group(3));
        }

        pattern = Pattern.compile(">:</FONT><FONT COLOR=\"#\\w{6}\">
             </FONT><FONT FACE=\"\\w*\" SIZE=\\d>(.*)</FONT>");

        matcher = pattern.matcher(newLine);

        if (matcher.find()) {
            message = matcher.group(1);
        }
        else {
            pattern = Pattern.compile(">:</FONT>
                 <FONT COLOR=\"#\\w{6}\"> (.*)</FONT>");

            matcher = pattern.matcher(newLine);

            if (matcher.find()) {
```

```
                        message = matcher.group(1);
                    }
                }
                MrpMessage msg = new MrpMessage (screenName, message, year,
                                                  month, date, hours, minutes,
                                                  seconds);

                logData.add(msg);
            }

        }
    }
}

// case when the log type is TRILLIAN
if (logFormat == 1) {

    path = logRoot + screenName + ".log";


    filename = logRoot + screenName + ".log";

    FileReader fr = new FileReader(filename);
    BufferedReader br = new BufferedReader(fr);

    Pattern pattern;
    Matcher matcher;


    //line = br.readLine();

    while ((line = br.readLine()) != null) {

        pattern = Pattern.compile("Session Start");
        matcher = pattern.matcher(line);

        if (line.length() == 0) {}

        else if (matcher.find()) {

            pattern = Pattern.compile( "(\\w{3})\\s(\\d{2})\\s(\\d{2}):
                 (\\d{2}):(\\d{2})\\s(\\d{4})");

            matcher = pattern.matcher(line);

            if (matcher.find()) {

                year = Integer.parseInt(matcher.group(6));
                month = getMonth(matcher.group(1));
                date = Integer.parseInt(matcher.group(2));

                if (Integer.parseInt(matcher.group(3)) == 0) {
                    before12 = false;
                } else {
                    before12 = true;
                }
            }
        }
        else if (line.charAt(0) == '[') {
```

```
        pattern = Pattern.compile("\\*\\*\\*");
        matcher = pattern.matcher(line);

        if (!matcher.find()) {

            // match the timestamp
            pattern = Pattern.compile("[\\d\\d:]+\\d\\d");
            matcher = pattern.matcher(line);

            if (matcher.find()) {

                StringTokenizer st =
                     new StringTokenizer(matcher.group(), ":");

                if (st.hasMoreTokens()) {
                    hours = Integer.parseInt(st.nextToken());

                    if (before12 && (hours == 0)) {
                        date++;
                        before12 = false;
                    }
                    if (!before12 && (hours != 0)) {
                        before12 = true;
                    }
                }

                if (st.hasMoreTokens()) {
                    minutes = Integer.parseInt(st.nextToken());
                }

                if (st.hasMoreTokens()) {
                    seconds = Integer.parseInt(st.nextToken());
                } else {
                    seconds = 0;
                }
            }

            // match the screen name
            pattern = Pattern.compile("\\s(\\w*):");
            matcher = pattern.matcher(line);

            if (matcher.find()) {
                //System.out.println(matcher.group());
                screenName = matcher.group(1);
            }

            // match the message content
            pattern = Pattern.compile("(:\\s)(.*)");
            matcher = pattern.matcher(line);

            if (matcher.find()) {
                message = matcher.group(2);
            }

            msg = new MrpMessage(screenName, message, year,
                                 month, date, hours,
                                 minutes, seconds);
            logData.add(msg);
        }
    }
```

```
                }
            }
            }

        catch (Exception e) {
            System.out.println("exception caught: " + e);
        }

    }

    String getDayOfWeek (int dayOfWeek) {
        if (dayOfWeek == 1) return "Sun";
        if (dayOfWeek == 2) return "Mon";
        if (dayOfWeek == 3) return "Tue";
        if (dayOfWeek == 4) return "Wed";
        if (dayOfWeek == 5) return "Thu";
        if (dayOfWeek == 6) return "Fri";
        if (dayOfWeek == 7) return "Sat";
        else return "";
    }

    String getFullDayOfWeek (int dayOfWeek) {
        if (dayOfWeek == 1) return "Sunday";
        if (dayOfWeek == 2) return "Monday";
        if (dayOfWeek == 3) return "Tuesday";
        if (dayOfWeek == 4) return "Wednesday";
        if (dayOfWeek == 5) return "Thursday";
        if (dayOfWeek == 6) return "Friday";
        if (dayOfWeek == 7) return "Saturday";
        else return "";
    }

    String getMonthName (int month) {
        if (month == 0) return "Jan";
        if (month == 1) return "Feb";
        if (month == 2) return "Mar";
        if (month == 3) return "Apr";
        if (month == 4) return "May";
        if (month == 5) return "Jun";
        if (month == 6) return "Jul";
        if (month == 7) return "Aug";
        if (month == 8) return "Sep";
        if (month == 9) return "Oct";
        if (month == 10) return "Nov";
        if (month == 11) return "Dec";
        else return "Und";
    }

    int getMonth(String month)
    {
        if (month.equalsIgnoreCase("jan")) return 0;
        if (month.equalsIgnoreCase("feb")) return 1;
        if (month.equalsIgnoreCase("mar")) return 2;
        if (month.equalsIgnoreCase("apr")) return 3;
        if (month.equalsIgnoreCase("may")) return 4;
        if (month.equalsIgnoreCase("jun")) return 5;
        if (month.equalsIgnoreCase("jul")) return 6;
        if (month.equalsIgnoreCase("aug")) return 7;
        if (month.equalsIgnoreCase("sep")) return 8;
        if (month.equalsIgnoreCase("oct")) return 9;
        if (month.equalsIgnoreCase("nov")) return 10;
        if (month.equalsIgnoreCase("dec")) return 11;
        if (month.equalsIgnoreCase("udc")) return 12;  // undecimber;
        return -1;
    }

    public int logLength() {
        return logData.size();
    }

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

    public int logFormat() {
        return logType;
    }

    public MrpMessage getMessage(int msgnum) {
        if (msgnum >= logData.size())
            throw new MrpException("invalid message number");
        return (MrpMessage)(logData.elementAt(msgnum));
    }

    public void setMessage(int msgnum, MrpMessage newmessage) {
        if (msgnum >= logData.size())
            logData.setSize(msgnum + 1);
        logData.set(msgnum, newmessage);
    }

    public MrpDataType copy() {
        return new MrpLog(logData, logType);
    }

    public void print(PrintWriter pw) {
        System.out.println(toString());
    }

    public void convert(int newFormat) {
        if (newFormat >= 0 && newFormat < 4) {
            logType = newFormat;
        }
    }

    public void clear () {
        logData.clear();
    }

    public String toString() {

        String madlong = "";

        for (int i = 0 ; i < logData.size(); i++) {
            MrpMessage tempMessage = (MrpMessage) logData.elementAt(i);
            madlong += "\n Message number: " + i + "\n"
                + tempMessage.toString() + "\n";
        }

        return madlong;
```

```
}

public void save() {
    if (path == null) {
        throw new MrpException ("trying to save unopened log");
    }

    this.save(path);
}


public void save(String outfile) {


    try {

        FileWriter fw = null;

        if (logType == 0) {
            // AIM

            String writeout = "";
            int currentDate = -1;

            String color = "";

            boolean success = (new File("./wtstc2/Egg03/").mkdirs());

            if (success)
            {
                System.out.println("dir created");
            }


            File f = new File(outfile + sname + "/");

            for (int i = 0; i < logData.size(); i++) {


                MrpMessage currentMessage = (MrpMessage)logData.get(i);

                if (currentMessage != null) {

                    if (currentMessage.getDate() != currentDate) {

                        // write previous file if not first time in loop

                        if (currentDate != -1) {
                            writeout += "</HTML><BR>";
                            fw.write(writeout);
                            fw.close();
                        }

                        currentDate = currentMessage.getDate();

                        // start new file

                        // file format: 2003-02-22 [Wednesday].html
                        String fileName = "";
```

```
// year
fileName += currentMessage.getYear() + "-";


// month
if (currentMessage.getMonth() < 10)
    fileName += "0" + (currentMessage.getMonth()
                       + 1);
else
    fileName += (currentMessage.getMonth() + 1);

fileName += "-";

// date
if (currentMessage.getDate() < 10)
    fileName += "0" + currentMessage.getDate();
else
    fileName += currentMessage.getDate();

// day name
fileName += " [";
fileName += getFullDayOfWeek(currentMessage.
                             getDayOfWeek());
fileName += "]";

File f2;

success = (new File(outfile + sname + "/" +
                    fileName +
                    ".html").createNewFile());

fw = new FileWriter(outfile + sname + "/" +
                    fileName + ".html");

// start new writeout for this file

writeout = "<-- ";

// date
if (currentMessage.getDate() < 10)
    writeout += "0" + currentMessage.getDate();
else
    writeout += currentMessage.getDate();

writeout += ":";

// hour
if (currentMessage.getHour() < 10)
    writeout += "0" + currentMessage.getHour();
else
    writeout += currentMessage.getHour();

writeout += ":";

// minute
if (currentMessage.getMinute() < 10)
    writeout += "0" + currentMessage.getMinute();
else
    writeout += currentMessage.getMinute();
```

```
                writeout += " -->\n<BASE TARGET=\"_new\">\n<HTML>";
            }  // end if

            if (currentMessage.getMessageName().equals(sname))
                color = "#0000ff";
            else
                color = "#ff0000";

            writeout += "<BODY BGCOLOR=\"#ffffff\"><B><FONT COLOR=\"" +
                color + "\">";

            writeout += currentMessage.getMessageName();

            writeout += "</FONT><FONT COLOR=\"" + color +
                "\" SIZE=1> (";

            String AMPM = "";

            if (currentMessage.getHour() > 12) {
                writeout += currentMessage.getHour() - 12;
                AMPM = " PM";
            }
            else {
                writeout += currentMessage.getHour();
                AMPM = " AM";
            }

            writeout += ":";

            // minute
            if (currentMessage.getMinute() < 10)
                writeout += "0" + currentMessage.getMinute();
            else
                writeout += currentMessage.getMinute();

            writeout += ":";

            // second
            if (currentMessage.getSecond() < 10)
                writeout += "0" + currentMessage.getSecond();
            else
                writeout += currentMessage.getSecond();

            writeout += AMPM + ")";

            writeout += "</B></FONT><FONT COLOR=\"" + color +
                "\" SIZE=3>:</FONT>";

            writeout += "<FONT COLOR=\"#000000\"> ";

            writeout += currentMessage.getText();

            writeout += "</FONT><BR>";

        } // end if message != null

    } // end for

} // end if
```

```
else if (logType == 1) {
    // trillian

    fw = new FileWriter (outfile);

    int currentDate = 0;

    for (int i = 0 ; i < logData.size(); i++) {

        MrpMessage currentMessage = (MrpMessage)logData.get(i);

        if (currentMessage.getDate() != currentDate) {
            currentDate = currentMessage.getDate();

            String writeout = "\n\nSession Start (:";

            writeout += currentMessage.getMessageName() + "): ";
            writeout += getDayOfWeek(currentMessage.getDayOfWeek())
                + " ";

            writeout += getMonthName(currentMessage.getMonth()) + " ";

            if (currentMessage.getDate() < 10)
                writeout += "0" + currentMessage.getDate() + " ";
            else
                writeout += currentMessage.getDate() + " " ;

            if (currentMessage.getHour() < 10)
                writeout += "0" + currentMessage.getHour() + ":";
            else
                writeout += currentMessage.getHour() + ":";

            if (currentMessage.getMinute() < 10)
                writeout += "0" + currentMessage.getMinute() +
                    ":";
            else
                writeout += currentMessage.getMinute() + ":";


            if (currentMessage.getSecond() < 10)
                writeout += "0" + currentMessage.getSecond() +
                    " ";
            else
                writeout += currentMessage.getSecond() + " ";


            writeout += currentMessage.getYear() + "\n";

            fw.write(writeout);


        }

        String writeout = "";

        if (currentMessage.getHour() < 10)
            writeout += "[0" + currentMessage.getHour() + ":";
        else
            writeout += "[" + currentMessage.getHour() + ":";
```

```
                if (currentMessage.getMinute() < 10)
                    writeout += "0" + currentMessage.getMinute() + "] ";
                else
                    writeout += currentMessage.getMinute() + "] ";

                writeout += currentMessage.getMessageName() +
                    ": " + currentMessage.getText() + "\n";


                fw.write(writeout);
            }

        }

        else if (logType == 2) {
        }

        // plain text transcript or custom format
        else if (logType == 3) {

            for (int i = 0 ; i < logData.size(); i++) {

                MrpMessage currentMessage = (MrpMessage)logData.get(i);

                String writeout = "";

                if (currentMessage.getHour() < 10)
                    writeout += "[0" + currentMessage.getHour() + ":";
                else
                    writeout += "[" + currentMessage.getHour() + ":";

                if (currentMessage.getMinute() < 10)
                    writeout += "0" + currentMessage.getMinute() + "] ";
                else
                    writeout += currentMessage.getMinute() + "] ";

                writeout += currentMessage.getMessageName() +
                    ": " + currentMessage.getText() + "\n";
                fw.write(writeout);

            }

        }

        fw.close();

    }

    catch (Exception e) {
        System.out.println(e);
    }
  }
}
```

A.18 MrpMessage.java

```
import java.io.PrintWriter;
import java.util.Calendar;

public class MrpMessage extends MrpDataType {
```

```
private String messagename;
private String text;
private Calendar timestamp = Calendar.getInstance();

private int year;
private int month;
private int date;
private int hours;
private int minutes;
private int seconds;

public MrpMessage() {
    this("", "", 0, 0, 0, 0, 0, 0);
};

public MrpMessage(String nme, String txt, int yr, int mth, int day,
                  int hr, int min, int sec) {
    messagename = nme;
    text = txt;

    year = yr;
    month = mth;
    date = day;
    hours = hr;
    minutes = min;
    seconds = sec;

    timestamp.set(year, month, date, hr, min, sec);
}

public MrpDataType copy() {
    return new MrpMessage(messagename, text, year, month, date,
                          hours, minutes, seconds);
}

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

public String toString() {
    return "Message: " + text + "\nWritten by: " + messagename +
         "\nAt: " + timestamp.getTime();
}

public void print(PrintWriter pw) {
    pw.println(toString());
}

public String getMessageName() { return messagename; }

public String getText() { return text; }

public int getTime() {
  return (int) (timestamp.getTimeInMillis() / 1000);
}

public int getHour() { return hours; }

public int getMinute() { return minutes; }
```

```
    public int getSecond() { return seconds; }

    public int getDate() { return date; }

    public int getDayOfWeek() { return timestamp.get(Calendar.DAY_OF_WEEK); }

    public int getYear() { return year; }

    public int getMonth() { return month; }

    void setMessageName(String nme) { messagename = nme; }

    void setText(String txt) { text = txt; }

    void setTime(int time) {
        timestamp.setTimeInMillis(time * 1000);

        year = timestamp.get(Calendar.YEAR);
        month = timestamp.get(Calendar.MONTH);
        date = timestamp.get(Calendar.DATE);
        hours = timestamp.get(Calendar.HOUR_OF_DAY);
        minutes = timestamp.get(Calendar.MINUTE);
        seconds = timestamp.get(Calendar.SECOND);
    }

}
```

A.19 Main.java

```
import java.io.*;
import antlr.CommonAST;
import antlr.collections.AST;

class Main {
    public static void main(String[] args) {
        try {
          PrintWriter stdOut = new PrintWriter(System.out);

            MrpLexer lexer = new MrpLexer(new DataInputStream(System.in));
          System.out.println("created lexer");
            MrpParser parser = new MrpParser(lexer);
          System.out.println("Created parser");
            parser.program();
          System.out.println("Parsed program");

          CommonAST t = (CommonAST) parser.getAST();
          System.out.println("Created AST");
          System.out.println(t.toStringList());
          MrpWalker walker = new MrpWalker();
          System.out.println("Created walker");
          MrpDataType r = walker.expr(t);
          System.out.println("Walked tree");
          System.out.print("value of r = ");
          r.print(stdOut);
          stdOut.flush();

        } catch(Exception e) {
            System.err.println("exception: "+e);
        }
        }
}
```