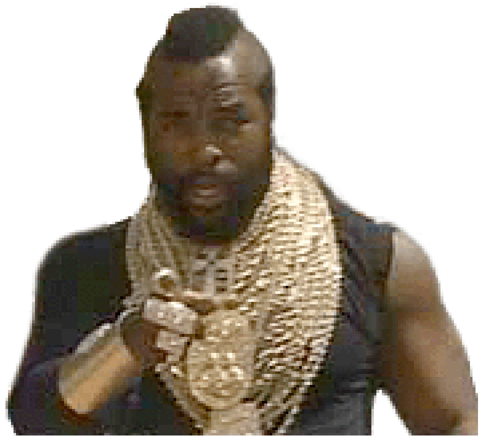

Mr. Proper

Language Reference Manual

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

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.

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 offset, int length )
```

Where `poo` is the original string, `offset` is the starting index, beginning at zero, and `length` is the number of characters to be read from the string, starting at `offset`.

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

getName()

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

```
string getName ( message myMessage )
```

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

getText()

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

```
string getText ( message myMessage )
```

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

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 `message` 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.

3.7.4 Log Message Manipulation

createMessage()

Function `createMessage()` creates a new empty message

```
message createMessage()
```

The function takes no input.

setName()

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

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

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

setText()

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

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

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

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

Sample Code

Use of the word count function

```
main
{
      string word = "mrproper";
      int count = wordCount(word);
      print(word + " appears " + count + " times in our program\n");
}

func int wordCount(string word)
{

      log myLog = open("omega1441", "/home/wb169/aim/logs/wtftc", AIM);


       int count = 0;

       int length = length(word);

      for(i=0:length)

      {
            string line = getText(log[i]);
            if(contains(line, word))
            {
                  count += 1;
            }
      }

      return count;
}
```

Grammar

```
/*
 * grammar.g : the lexer and the parser, in ANTLR grammar.
 *
 * @author William Blinn - wb169@columbia.edu
 *
 * @version $Id: grammar.g,v 1.2 2003/10/19 23:26:09 jl1434 Exp $
 */

class MrProperLexer extends Lexer;

options{
    k = 2;
    charVocabulary = '\3'..'\377';
    testLiterals = false;
    exportVocab = MrProper;
}

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

protected
ALPHA   : 'a'..'z' | 'A'..'Z' | '_' ;

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

WS      : (' ' | '\t')+          { $setType(Token.SKIP); }
        ;

NL      : ('\n' | ('\r' '\n') => '\r' '\n' | '\r')
            { $setType(Token.SKIP); newline(); }
        ;

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

LPAREN  : '(';
RPAREN  : ')';
MULT    : '*';
```

```
PLUS    : '+';
MINUS   : '-';
RDV     : '/';
MOD     : '%';
SEMI    : ';';
LBRACE  : '{';
RBRACE  : '}';
LBRK    : '[';
RBRK    : ']';
ASGN    : '=';
COMMA   : ',';
COLON   : ':';
PLUSEQ  : "+=";
MINUSEQ : "-=";
MULTEQ  : "*=";
RDVEQ   : "/=";
MODEQ   : "%=";
GE      : ">=";
LE      : "<=";
GT      : '>';
LT      : '<';
EQ      : "==";
NEQ     : "!=";

ID  options { testLiterals = true; }
        : ALPHA (ALPHA|DIGIT)*
        ;

/* NUMBER example:
 * 1, 1e, 1., 1.e10, 1.1, 1.1e10
 */

NUMBER  : (DIGIT)+ ('.' (DIGIT)*)? (('E'|'e') ('+'|'-')? (DIGIT)+)? ;

STRING  : '"'!
             (  ~('"' | '\n')
              | ('"'!'"')
             )*
          '"'!
        ;


class MrProperParser extends Parser;

options{
    k = 2;
    buildAST = true;
    exportVocab = MrProper;
}


tokens {
   STATEMENT;
   VAR_LIST;
   EXPR_LIST;
   FUNC_CALL;
   FOR_CON;
   LOOP;
   UPLUS;
   UMINUS;
}

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

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

                {System.out.println("Best program ever!");}

        ;

statement
        : ( for_stmt
        | if_stmt
            | while_stmt
        | break_stmt
        | continue_stmt
        | return_stmt
        | assignment
        | func_call_stmt
            | stmt_block
        | declaration )

        {System.out.println("statement");}

        ;

main
        : "main"^ stmt_block

            {System.out.println("main");}

        ;


stmt_block
        : LBRACE! (statement)* RBRACE!
            {#stmt_block = #([STATEMENT,"STMT_BLOCK"], stmt_block); } //ask
TA for formatting

            {System.out.println("stmt_block");}

        ;
```

```
for_stmt
        : "for"^ LPAREN! for_con RPAREN! stmt_block

            {System.out.println("for_stmt");}

        ;

for_con
        : ID ASGN! range
            {#for_con = #([FOR_CON,"FOR_CON"], for_con); }

            {System.out.println("for_con");}

        ;

if_stmt
        : "if"^ LPAREN! expression RPAREN! stmt_block
            (options {greedy = true;}: "else"! statement )?

            {System.out.println("if_stmt");}

        ;

while_stmt
            : "while"^ LPAREN! expression RPAREN!      stmt_block

            {System.out.println("while_stmt");}

            ;


break_stmt
        : "break"^ (ID)? SEMI!

            {System.out.println("break_stmt");}

        ;

continue_stmt
        : "continue"^ (ID)? SEMI!

            {System.out.println("continue_stmt");}

        ;

return_stmt
        : "return"^ (expression)? SEMI!

            {System.out.println("return_stmt");}

        ;

assignment
        : l_value ( ASGN^ | PLUSEQ^ | MINUSEQ^ | MULTEQ^ | LDVEQ^
                  | MODEQ^ | RDVEQ^
                ) expression SEMI!
            {System.out.println("assignment");}

        ;

declaration
        : ID l_value (( ASGN^ | PLUSEQ^ | MINUSEQ^ | MULTEQ^ | LDVEQ^
                    | MODEQ^ | RDVEQ^
                  ) expression )?  SEMI!

            {System.out.println("declaration");}

        ;


func_call_stmt
        : func_call SEMI!

            {System.out.println("func_call_stmt");}

        ;

func_call
        : ID LPAREN! expr_list RPAREN!
            {#func_call = #([FUNC_CALL,"FUNC_CALL"], func_call); }

            {System.out.println("func_call");}

        ;

expr_list
        : expression ( COMMA! expression )*
            {#expr_list = #([EXPR_LIST,"EXPR_LIST"], expr_list); }
            {System.out.println("expr_list");}


        |   /* nothing */
            {#expr_list = #([EXPR_LIST,"EXPR_LIST"], expr_list); }

        ;

func_def
        : "func"^ ID l_value LPAREN! var_list RPAREN! stmt_block

            {System.out.println("func_def");}

        ;

var_list
        : ID ID ( COMMA! ID ID)*
            {#var_list = #([VAR_LIST,"VAR_LIST"], var_list); }

            {System.out.println("var_list");}

        |   /* nothing */
            {#var_list = #([VAR_LIST,"VAR_LIST"], var_list); }
        ;

expression
        : logic_term ( "or"^ logic_term )*

            {System.out.println("expression");}

        ;

logic_term
        : logic_factor ( "and"^ logic_factor )*

            {System.out.println("logic_term");}

        ;

logic_factor
        : ("not"^)? relat_expr

            {System.out.println("logic_factor");}

        ;

relat_expr
        : arith_expr ( (GE^ | LE^ | GT^ | LT^ | EQ^ | NEQ^) arith_expr )?

            {System.out.println("relat_expr");}

        ;

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

            {System.out.println("arith_expr");}

        ;

arith_term
        : arith_factor ( (MULT^ | LDV^ | MOD^ | RDV^) arith_factor )*

            {System.out.println("arith_term");}

        ;

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

            {System.out.println("arith_factor");}

        ;


r_value
```

```
        : ( l_value
        | func_call
        | NUMBER
        | STRING
        | "true"
        | "false"
        | LPAREN! expression RPAREN! )

            {System.out.println("r_value");}

        ;

l_value
        : ID^ ( LBRK! expression RBRK! )? //also defines array

            {System.out.println("l_value");}

        ;

range
        : expression COLON^ expression

            {System.out.println("range");}

        ;

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

        ;
```