MOHAWK Language Reference Manual

COMS W4115 Fall 2005

Joeng Kim
jk2438@columbia.edu

Chris Murphy
cdm6@columbia.edu

Ryan Overbeck
rso2102@columbia.edu

Lauren Wilcox
lgw23@columbia.edu

1 Introduction

This document describes the syntax and semantics of the MOHAWK programming language. It starts with a brief overview of the structure of a MOHAWK program, then discusses the lexical conventions in the language, and then builds up the semantics by explaining the operators, expressions, and statements that make up the language. This document also provides a sample program and the complete grammar in ANTLR notation.

Conventions used in this document

In this document, characters and strings used in the MOHAWK language are identified by double quotes; however, the double quotes themselves are not part of the string, unless otherwise noted.

2 Language Construction & Execution

Rules

A rule (or "pattern-action statement") in a MOHAWK program is structured as follows:

```
pattern { statements }
```

pattern: either the keyword "`-:-)`" (begin), the keyword "`(-:-`" (end), a MOHAWK logical expression or nothing.
statements: MOHAWK statements.

Execution

To execute a MOHAWK program, the following command is used

```
mohawk program-file input-file
```

program-file: the name of the file containing the MOHAWK program.
input-file: a data file of newline-separated records containing whitespace-separated fields.

As in awk, MOHAWK programs consist of a sequence of rules and optional function definitions.

MOHAWK programs execute as follows:

1. The statements in the "`-:-)`" pattern (if it exists) are executed. This pattern must appear before all other patterns in the program.

2. For each record in the input file, all patterns (except for the "end" pattern) are evaluated in the order they appear in the MOHAWK program, starting at the top; if a pattern evaluates to logical "true", the corresponding statements are executed. Similarly, if there is no pattern, the statements within the curly braces are executed. A pattern without any statement(s) is not valid syntax and will generate a syntax error.

3. Finally, the statements in the "`(-:-`" pattern (if it exists) are executed.

If a program only has a "`-:-)`" rule but not other patterns, no input files are processed. If a program only has a "`(-:-`" rule and no other patterns, the input *will* be read.

Function Definitions

User-defined functions, as in C and Java, must be named with valid identifier names (see below). A function definition in MOHAWK is written as follows:

```
function name (params) { statements }
```

name: the name of the function.
params: a comma-separated list of parameters (in the form of MOHAWK expressions).
statements: the MOHAWK statements that make up the function.

User-defined functions do not have return values in MOHAWK.
Function definitions must appear in the MOHAWK program before they can be used (forward declaration).
i.e. above the code that calls it in the program.

3 Lexical Conventions

Tokens

In MOHAWK, tokens are line separators/terminators, comments, identifiers, keywords, operators, literals, field variables, system variables. White space " " and tabs "\t" are ignored by MOHAWK and are merely used to separate tokens.

Line Separators/Terminators

In MOHAWK, all statements end with a bang. That is, the exclamation point "!" is used as a statement terminator (except for loops and conditional statements).

Comments

MOHAWK uses C/Java-style conventions for comments in the code. Single-line comments start with two forward slashes “//” and run to the end of the line (until “\n” or “\r” is encountered). Multiple-line comments start with a slash and asterisk “/*” and end with an asterisk and slash “*/”.

Comments cannot be nested within other comments. That is, a comment that starts with “/*” will be terminated by the first “*/” that it sees, regardless of whether there is another “/*” within that comment.

Field Variables

When a MOHAWK program runs, the pattern-action statements (except for the begin and end blocks) will be executed for each record in the input data file. Programmers can refer to the fields in each record by using the dollar sign “$” followed by the number of the field in the record, going from left to right and starting with 1. The symbol “$0” refers to the entire record.

Identifiers

Users can create as many identifiers as necessary in their program. An identifier is a sequence of any number of letters, digits, and underscores characters. Identifiers cannot start with a number and must start with a letter or underscore. Lastly, identifiers are case sensitive.

Keywords

Following are the keywords in the MOHAWK language; no identifier can have the same name as a keyword.

```
:-/         :->         while
:-\         :-|         >-)
:-/\        -:-)        INFINITY
:^O         (-:-        true
:-(         :^P         false
>^O
```

Operators

Following are the operators that are used in the MOHAWK language, primarily for mathematical and logical functions.

```
+      +=     ==     >      ++
-      -=     !=     <      --
*      *=     ~=     >=     EQ
```

```
/      /=     &&     <=     NE
%      =      ||     ^      .
~
```

System Variables

MOHAWK defines two system variables that are available to programmers anywhere in the program. The system variable “#F” refers to the number of fields in the current record; this system variable cannot be used in the “begin” or “end” patterns of the program. If it is used in a user-defined function, it refers to the number of fields in the record being used when the function is called. If the user-defined function is called from the “begin” or “end” pattern statements, it evaluates to zero.

The system variable “#R” can be used to refer to the total number of records in the input file, and can only be referred to in the “end” pattern or a user-defined function. If the user-defined function that refers to “#R” is called from any block other than the “end” pattern, it evaluates to zero.

Literals

There are three types of literals in MOHAWK: numbers, strings, and Boolean values. Numbers can be integers or floating point numbers, as described below. All string literals will start with the double-quote character and end with a double-quote character. In the event that the string must contain double-quotes, two double-quotes in a row must be used. Boolean literals can be either the keywords “true” or “false” (without the quotation marks).

4 Datatypes

Like its predecessor awk, MOHAWK is a typeless language. That is, variables and record fields may be (floating point) numbers, strings or both. Context determines how the value of a variable is interpreted. If used in a numeric expression, it will be treated as a number, if used as a string it will be treated as a string.

Uninitialized variables have the numeric value 0 and the string value “” (the null, or empty string).

Numbers

Numbers in MOHAWK can be either integers or floating point decimal numbers.

Integer numbers in MOHAWK, like in Java, range from -2^{31} to $2^{31}-1$.

Floating point numbers in MOHAWK, like in Java, range from 2^{-149} to $(2-2^{-23}) \cdot 2^{127}$. A floating point number in MOHAWK must be in the form defined by the Java standard[1]:

> A floating-point literal has the following parts: a whole-number part, a decimal point (represented by an ASCII period character), a fractional part, an exponent, and a type suffix. The exponent, if present, is indicated by the ASCII letter `e` or `E` followed by an optionally signed integer.
>
> At least one digit, in either the whole number or the fraction part, and either a decimal point, an exponent, or a float type suffix are required. All other parts are optional.
>
> *FloatingPointLiteral:*
>   *Digits* **.** *Digits*$_{opt}$ *ExponentPart*$_{opt}$
>   **.** *Digits* *ExponentPart*$_{opt}$
>   *Digits* *ExponentPart*
>
> *ExponentPart:*
>   *ExponentIndicator* *SignedInteger*
>
> *ExponentIndicator: either or not both*
>   e | E
>
> *SignedInteger:*
>   *Sign*$_{opt}$ *Digits*
>
> *Sign: either or not both*
>   + | -
>
> The largest positive finite float literal is 3.40282347e+38f. The smallest positive finite nonzero literal of type float is 1.40239846e-45f.
>
> A [runtime] error occurs if a nonzero floating-point literal is too large, so that on rounded conversion to its internal representation it becomes an IEEE 754 infinity. A program can represent infinities without producing a [runtime] error by using constant expressions such as [INFINITY].
>
> A [runtime] error occurs if a nonzero floating-point literal is too small, so that, on rounded conversion to its internal representation, it becomes a zero. A [runtime] error does not occur if a nonzero floating-point literal has a small value that, on rounded conversion to its internal representation, becomes a nonzero denormalized number.

When a MOHAWK operator has one argument as a floating point number and the other as an integer, the integer will be widened (promoted) to a floating point number, and the

[1] http://java.sun.com/docs/books/jls/second_edition/html/lexical.doc.html#230798

result (if numeric) will be a floating point number as well. This is done in accordance with the Java specification[2], despite any possible loss of precision.

Strings

The string representation of an integer variable is rather straightforward. If the number is less than zero, the first character is '-'; then the rest of the integer value is returned, taking up as many characters as needed.

However, for floating point numbers, the conversion to a string is as described in the Java 1.5 API specification[3]:

> The result is a string that represents the sign and magnitude (absolute value) of the argument. If the sign is negative, the first character of the result is '-'; if the sign is positive, no sign character appears in the result. As for the magnitude *m*:
>
> - If *m* is infinity, it is represented by the characters "Infinity"; thus, positive infinity produces the result "Infinity" and negative infinity produces the result "-Infinity".
> - If *m* is zero, it is represented by the characters "0.0"; thus, negative zero produces the result "-0.0" and positive zero produces the result "0.0".
> - If *m* is greater than or equal to 10^{-3} but less than 10^{7}, then it is represented as the integer part of *m*, in decimal form with no leading zeroes, followed by '.', followed by one or more decimal digits representing the fractional part of *m*.
> - If *m* is less than 10^{-3} or greater than or equal to 10^{7}, then it is represented in so-called "computerized scientific notation." Let *n* be the unique integer such that $10^n <= m < 10^{n+1}$; then let *a* be the mathematically exact quotient of *m* and 10^n so that $1 <= a < 10$. The magnitude is then represented as the integer part of *a*, as a single decimal digit, followed by '.', followed by decimal digits representing the fractional part of *a*, followed by the letter 'E', followed by a representation of *n* as a decimal integer.
>
> How many digits must be printed for the fractional part of *m* or *a*? There must be at least one digit to represent the fractional part, and beyond that as many, but

[2] http://java.sun.com/docs/books/jls/second_edition/html/conversions.doc.html#25214
[3] http://java.sun.com/j2se/1.5.0/docs/api/java/lang/Float.html#toString(float)

only as many, more digits as are needed to uniquely distinguish the argument value from adjacent values. That is, suppose that *x* is the exact mathematical value represented by the decimal representation produced by this method for a finite nonzero argument *f*. Then *f* must be the value nearest to *x*; or, if two values are equally close to *x*, then *f* must be one of them and the least significant bit of the significand of *f* must be **0**.

Booleans

There is no primitive Boolean datatype in MOHAWK; the Boolean value is represented internally as an integer. Thus, the logical value of a variable in MOHAWK is defined as follows:

- If the variable is numeric or if its string representation is numeric (i.e., the variable is a number or a numeric string), it is considered to be logical "true" if the numeric value is greater than zero, and logical "false" if the value is zero or is negative.

- If the identifier's string representation is non-numeric, it is considered to be logical "true" if the string is non-null, and logical "false" if the string is empty/null.

Scope

Variables in MOHAWK can have one of two types of scope: global or local.

Global variables must be defined in the "`-:-)`", or begin, block of the program. Any identifier initialized in this block will be in scope in every other block of the program. This allows programmers to define global variables up front, at the very beginning.

Typically, global variables will be used to give meaningful names to the record fields from the input file. The assignment of names to the fields is in the format

```
varname = $number!
```

where *varname* is the name of the global variable and *number* is the number of the field in the record, going from left to right and starting with 1. Whenever *varname* is encountered in any other part of the program, it will use the value of the field *number* in the record that is currently being evaluated. If no record is being evaluated at the time, *varname* will have the value 0 or "" (the empty string).

Local variables, on the other hand, are defined in the main part of the program, or in the "(-:-", or end, block of the program. These variables are statically scoped and are only accessible in the block of code (starting with a left brace "{" and ending with a right brace "}") in which they are defined.

Namespaces

Namespaces are abstract containers which filled by unique names for a particular namespace. This means that variables of the same namespace cannot share the same name.

There is only one set of namespaces in MOHAWK. These are functions and variables names which share the same namespace.

5 Operators

Mathematical Operators

The mathematical operators in MOHAWK work on the operands as follows:

- If an operand is numeric or is a numeric string (i.e., the string representation is a valid integer or floating point number as defined above), the operator acts on the numeric representation.
- If an operand is a non-numeric string, it is treated as the integer value 0, with the exception of the addition operator "+", which would return the concatenation of the string representations of the two operands.

In order of precedence, the mathematical operators are as follows:

Operator	**Usage**	**Description**
`*`	`a * b`	Returns the product of `a` and `b`
`/`	`a / b`	Returns the quotient of `a` and `b`
`%`	`a % b`	Returns the modulus of `a` and `b`
`+`	`a + b`	Returns the sum of `a` and `b`
`-`	`a – b`	Returns the difference of `a` and `b`
`^`	`a ^ b`	Returns the value of `a` raised to the power of `b`
`++`	`a++`	Increments the value of `a` by 1
`--`	`a--`	Decrements the value of `a` by 1

String Operators

These operators can be used for any variable or literal in MOHAWK; if an operand is a numeric variable or literal, its string representation will be used.

Operator	**Usage**	**Description**
`EQ`	`a EQ b`	Returns true if the string representations of `a` and `b` are equivalent

`NE`	`a NE b`	Returns true if the string representations of `a` and `b` are not equivalent
`.`	`a . b`	Returns the concatenation of `a` and `b`

As noted above, the mathematical operator "+" also returns the concatenation if either or both of the arguments are non-numeric strings.

Assignment Operators

Assignment operators in MOHAWK are used as in many other programming languages, like C and Java.

Operator	**Usage**	**Description**
`=`	`a = b`	Assigns the value of `b` to `a`
`+=`	`a += b`	Assigns the value of `a+b` to `a`
`-=`	`a -= b`	Assigns the value of `a-b` to `a`
`*=`	`a *= b`	Assigns the value of `a*b` to `a`
`/=`	`a /= b`	Assigns the value of `a/b` to `a`
`%=`	`a %= b`	Assigns the value of `a%b` to `a`

Except for the basic assignment operator "=", the other operators are only valid for numbers, numeric strings, and numeric literals. If one of these operators is used with a non-numeric string, then a runtime error will occur.

Comparison Operators

Comparisons are performed only on numbers, numeric strings, and numeric literals. If a variable or string literal has a string value that is numeric, then it can be used for comparisons. This is done so that record fields from the input file can be treated as numbers when they "look" like numbers. However, if one or both of the arguments is a non-numeric string, then a runtime error will occur (string comparisons should use the string operators defined above).

The comparison operators in MOHAWK are as follows (all with equal precedence):

Operator	**Usage**	**Description**
`==`	`a == b`	Evaluates to `true` if `a` is equal to `b`
`!=`	`a != b`	Evaluates to `true` if `a` is not equal to `b`
`>`	`a > b`	Evaluates to `true` if `a` is greater than `b`
`<`	`a < b`	Evaluates to `true` if `a` is less than `b`
`>=`	`a >= b`	Evaluates to `true` if `a` is greater than or equal to `b`
`<=`	`a <= b`	Evaluates to `true` if `a` is less than or equal to `b`

Logical Operators

As described above, there are no Boolean datatypes in MOHAWK, but variables and expressions can have logical values. In order of precedence, the logical operators in MOHAWK are as follows:

Operator	Usage	Description
`~`	`~a`	Returns the logical opposite of `a`
`&&`	`a && b`	Evaluates to `true` if `a` and `b` are both logically true
`||`	`a || b`	Evaluates to `true` if either `a` or `b` is logically true

Regular Expression Operators

The only regular expression operator in MOHAWK uses the operator "~=". This operator takes the following form

```
    varname ~= regexp
```

where *varname* is a variable and *regexp* is a valid regular expression. This operator returns a logical "true" if *varname* matches the regular expression *regexp* (which must be a valid MOHAWK string) and a logical "false" if it does not.

6 Regular Expressions

Mohawk regular expressions

Regular expressions allow the user to describe sets of strings or substrings with a single corresponding pattern rather than explicitly listing the string or substrings. Mohawk incorporates regular expressions for enhanced matching and comparison capabilities by allowing users to **evaluate boolean conditional statements** based on whether the current token or string matches or fits a specified pattern.

Context

An example of the boolean regular expression context:

```
if ($1 =~ "REGEX")
{
  :-O ("The token matches")!  //if the var matches the pattern, print "The token matches"
 }
```

Here, REGEX is a quoted string pattern composed of characters, numbers, and operators. Mohawk Reg-Ex length can be zero to maximum string buffer size. Mohawk Reg-Ex patterns are demarcated with surrounding quotation marks, like those used to define a string. Tokens or characters included in a pattern are evaluated sequentially when greater than zero.

A regular expression can be differentiated from a string literal with the =~ operator used in the evaluation of a boolean expression containing it. The expression has two syntactic possibilities:

“string” =~ “REGEX PATTERN”
$variable =~ “REGEX PATTERN”

Each of the above result in a match evaluation, which returns the boolean values “true” or “false” on the string or variable on the left with the regular expression contained in quotation marks on the right.

Regular Expression construction

Regular Expression operators

[ ]
Brackets around a pattern indicate that the match should take place on one of what occurs inside them. This can be used in logical “or” contexts to match one of the characters inside, greedily.

[Cc]hris and Becky are nice TAs

will match occurrences of “Chris” and “chris”.

- within [ ]
The dash “-“ when inside “[ ]” specifies character and numeric range patterns
The patterns “[a-z]” and “[A-Z]” represent character classes comprising all lowercase and all uppercase characters defined in the English alphabet, respectively. This way, any single lowercase character in the alphabet in the range a-z may be matched with the expression “[a-z]”. In this context, the “-“ represents “the range of characters between the starting character “a” and ending character “z”, inclusive. The other use of the dash is to specify a numerical range of single digit characters, matching the range of digits between the start digit and the ending digit, as in “[0-9]”. Character and number classes may be combined in any order and other quantifiers may be affixed before or after ranges in the same pattern, as in: “[a-z_0-9A-Z]”. Range matches are also greedy, thus using a similar example as above, the comparison:

“Chris and Becky are nice TAs” =~ “[a-z_0-9A-Z]”

returns **true** as the first character, ‘C’ matches. Ranges are also interpreted in sequential order, such that “[a-z] [0-9] [A-Z]” matches strings like “a0A” and “b1G” but not “bi1G”.

^ within []

The negation operator means negate the following pattern when used as the first character inside []. Thus "[^]" matches any character other than a space (" ")
"[^A-Z]" matches any character that is not an uppercase character.

"[A-HK-Z][A-NP-Z][A-DF-Z][A-MO-Z][A-FH-Z][^][^][^]");
Matches an upper case letter (excluding JOENG), followed three non-space characters.

\ to escape literals
Non-alphanumeric characters must be escaped when the user wishes to include these as literal characters in a pattern. This can be achieved by attaching the escape character \ before the character to be considered literal. For example, \[indicates that a bracket should be considered a literal in a pattern instead of a range operator around a pattern. The escape character\ can be escaped so as to be considered a literal forward slash as in: \\
The escape character can also indicate boundaries

\b	A word boundary
\B	A non-word boundary
\A	The beginning of the input
\G	The end of the previous match
\Z	The end of the input but for the final terminator, if any
\z	The end of the input

Quantifiers

Quantifiers can be appended to patterns to indicate occurrence information when matching, using the following syntax:

PATTERN?	once or not at all
*PATTERN**	zero or more times
PATTERN+	one or more times
X{*n*}	exactly *n* times
X{*n*,}	at least *n* times
X{*n*,*m*}	at least *n* but not more than *m* times

Logical Operators

Logical operators can be used to match on Boolean logic, using the following syntax:

PATTERN – XPATTERN - Y	Pattern X followed by Pattern Y
PATTERN - X | *PATTERN - Y*	Either Pattern X or Pattern Y

Flags

(?i) following pattern will ignore case when determining match, but does not take effect until after the instance it corresponds to appears in the pattern.
(?-i) turns off the “ignore case” option for the pattern which follows this flag.
Multiple flags may occur in a single pattern

```
Example: "Haveyouheard(?i)Mohawk(?-i)isthenew(?i)Awk"
Matches:
     “HaveyouheardMOHAWKisthenewAWK”
     “Haveyouheardmohawkisthenewawk”
```

7 Expressions

Expressions are the building blocks of any MOHAWK program. The different types of expressions can be combined to make more complicated expressions.

Simple Expressions

The simplest type of expression consists only of a single identifier, field variable, number, string, boolean (“true” or “false”), or function call.

Arithmetic Expressions

An arithmetic expression is a simple expression, followed by zero or more sets of a mathematical operator (see above) and another simple expression.

Relational Expressions

Relational expressions are used for comparisons. They consist of an arithmetic expression, followed by zero or more sets of comparison operators (see above) and another arithmetic expression.

Expressions

An expression, therefore, can be defined in MOHAWK as a relational expression (with an optional logical not “~”), followed by zero or more sets of logical operators (see above) and another relational expression. Additionally, an expression surrounded by parentheses can also be considered a simple expression.

8 Statements

Statements in MOHAWK fall into several categories. All MOHAWK statements must end with the terminator character “!” except for conditionals and loops.

Assignments

An assignment statement is used to set the value of a variable in the MOHAWK program. It consists of an identifier or field variable, one of the assignment operators (see above), and an expression.

Function Calls

There are two types of function calls in MOHAWK: internal and user-defined.
In either case, function calls start with the name of the function, a left parenthesis "(", a comma-separated list of parameters (MOHAWK expressions), and a right parenthesis ")".

i.e.   function (expression) !

Internal Functions

Internal function call is used for accessing system and environment information; all internal functions in MOHAWK uses "smileys", which are ASCII-drawn smiley faces. Smileys have eyes (";", ":", "-", or ">"), noses ("-" or "^") and mouths (")", "(", "0", "D", "P", ">", "<", "|", "\", "/").

The current version of MOHAWK supports two internal functions.

:-O

Description: Prints the current record.
This function prints out the string, data, and values of variables.
Syntax: `:-O ( string )!`
Example: `:-O ( "This is my value" + value )!`

:-<

Description: Set $0 from current input record or particular file.
This function is used to read in data.
Syntax: `:-< ()!`
Example: `:-< ()!`

User-defined Functions

User-defined functions, as in C and Java, must be named with valid identifier names (see above).

```
>-)  name (parameter-list)
     statement
```

Each parameter in the list is simply a comma-separated list of variables as arguments.
The following is an example of user-defined function.

```
>-)  myfunc (val1, val2)
{
     :-O ("print my stuff: " + val1 + val2)!
}
```

Conditionals

Similar to C/Java, MOHAWK allows for conditional statements with the “if”/”else” structure, which can work in one of three ways:

```
:-/ (expr)
    statement
```

First, `expr` is evaluated; if it is logically true, `statement` is executed.

```
:-/ (expr)
    statement1
:-\
    statement2
```

First, `expr` is evaluated. If it is logically true, then `statement1` is executed; otherwise, `statement2` is executed.

```
:-/ (expr1)
    statement1
:-/\ (expr2)
    statement2
:-/\ . . .
:-\
   statement(s)
```

First, `expr1` is evaluated; if it is logically true, then `statement1` is executed. If it is not true, then `expr2` is evaluated; if it is logically true, then `statement2` is executed. This continues for all subsequent “:-/\” statements. If no expressions in the “:-/” and “:-/\” statements are logically true and an optional “:-\” statement exists at the end, the `statement` in the corresponding block is executed.

Loops

MOHAWK supports the use of three different types of loops: “for”, “while”, and “do/while”.

As in C/Java, a “for” loop is structured in the following way:

```
:^0 (expr1; expr2; expr3)
    statement
```

First, `expr1` is evaluated. Then, if `expr2` is logically true, the `statement(s)` in the block are executed, and then `expr3` is evaluated. If `expr2` is still logically true, the `statement(s)` and `expr3` are executed until `expr2` is no longer true.

Similarly, a “while” loop is structured as follows:

```
while (expr)
    statement
```

First, *`expr`* is evaluated. If it is logically true, the *`statement(s)`* in the block are executed. If *`expr`* is still logically true, the *`statement(s)`* are executed until *`expr`* is no longer true.

Lastly, a “do/while” loop is structured as follows:

```
>^0
    statement
while (expr)
```

First, *`statement`* is executed. Then *`expr`* is evaluated; if it is logically true, then *`statement`* is executed again. If *`expr`* is still logically true, the *`statement`* is executed until *`expr`* is no longer true.

To review:

Operator	**Translation**
`:-/`	`if`
`:-\`	`else`
`:-/\`	`if else`
`:^O`	`for`
`>^O`	`do`

Control Statements

There are four MOHAWK keywords that can be used as standalone statements and are especially useful in the context of a conditional or loop statement.

Operator	**Translation**	**Description**
`:-(`	`break`	break out of the nearest enclosing loop
`:->`	`continue`	skip the rest of the loop body but continue looping
`:^P`	`exit`	terminate the processing of input records
`:-|`	`next`	finish processing the current input record and move on to the next one

Blocks

Any statement can be considered a block (or compound statement) by using a left brace “{“, a set of one or more statements, and a right brace “}”.

9 Error Handling

Error handling in MOHAWK is similar to the practices of Java programming language.

Syntax Errors

MOHAWK attempts to find all syntax errors when possible. When MOHAWK finds an error in the syntax of the program, it will print out error to the screen to inform the user of the syntax error. MOHAWK will try to display to the user messages which are most relevant and easily comprehensible.

Error messages include the following:

- Line number of the error
- Filename of the error
- Error description

The format of the error will be produced in the following manner:

```
MOHAWK says…
myProgram.mk [Line: 34] ERROR: Bad usage of smiley face!
```

Runtime Errors

Although MOHAWK attempts to resolve possible errors at compile time, errors can occur while the program runs. Rather than crashing the program with meaningless messages, MOHAWK programs attempt to continue running. The result of this is that unexpected results can occur but the program will not simply fall over due to runtime error. Common runtime errors such as divide by zero, null exception, and invalid input can easily crash programs in most languages.

MOHAWK attempts to continue the program by attempting to resolve issues when possible. Example of this situation is when errors occur due to invalid data manipulation. When a user tries to add a number to a string, the string will be evaluated as a 0. Similar situations where values can be computed as zero or null will be done if the result is a continuous program.

When errors cannot be corrected, MOHAWK will terminate the program with the line number which caused the error.

Example:

```
MOHAWK says…
myProgram.mk [Line: 59] DIED WHILE RUNNING!!!
```

10 Sample Program

Datafile:

```
bob  40
8.75
mary 43
9.65
```

Program:

```
-:-) {
// assign names to fields
name = $1!
hours = $2!
rate = $3!

// declare other variables
sum = 0!
}

/* there is no pattern so this will run for each record */
{
   overtime = 0!
   :-/ (hours > 40)
   {
      overtime = hours – 40!
      hours = 40!
   }

   // employees get time-and-a-half for overtime
   total = (hours * rate) + (overtime * rate * 1.5)!
   :-O(name + ": $" + total)! // print
   sum = sum + total!
}

(-:-
{
   :-O("Total wages: $" + total)!
}
```

Output:

```
bob: $350
mary: $425.425
Total wages: $779.425
```

11 Complete ANTLR Grammar

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

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); } ;

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

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

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

LCURLY : '{' ;
RCURLY : '}' ;
BANG : '!' ;
LPAREN : '(' ;
RPAREN : ')' ;

ASSIGN : '=' ;
INCR : "++" ;
DECR : "--" ;
PLUSEQ : "+=" ;
MINUSEQ : "-=" ;
DIVEQ : "/=" ;
MULTEQ : "*=" ;

EQ : "==";
NEQ : "!=" ;
GE : ">=" ;
LE : "<=" ;
GT : ">" ;
LT : "<" ;
```

```
OR : "||";
AND : "&&";
NOT : "~";

PLUS : '+' ;
MINUS : '-' ;
MULT : '*' ;
DIV : '/' ;
MOD : '%' ;
EXP : '^' ;

STRCAT : ".";
REGEQ : "~=";

COMMA : ',' ;

BEGIN : "-:-)" ;
END : "(-:-" ;

FIELD_VAR : '$' ( DIGIT )+ ;


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


program
      : ( rule | function_definition )*  EOF!
;

rule
      : pattern_action_stmt
;

pattern_action_stmt :
      pattern LCURLY action RCURLY
;

action :
      ( stmt )*
;

pattern : ( BEGIN | END | expr )
        ;


/* Expressions */
expr
      : logic_term ( OR logic_term )*
;

logic_term
      : logic_factor ( AND logic_factor )*
;
```

```
logic_factor
      : ( NOT )? rel_expr
;

rel_expr
      : arith_expr
         (( GE | LE | GT | LT | EQ | NEQ ) arith_expr )?
;

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

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

arith_factor
      : r_value
;

r_value
      : l_value | NUMBER | STRING | "true" | "false" | ( function_call
) |
         ( LPAREN expr RPAREN )
;

l_value
      : NAME | FIELD_VAR
;

/* Statements */
stmt
      : (( break_stmt | continue_stmt |
            exit_stmt |
            function_call |
            asgn_stmt |
            next_stmt ) BANG ) |

            do_stmt |
            for_stmt |
            if_stmt |
            while_stmt |
            ( LCURLY ( stmt )* RCURLY )
;

break_stmt
      : break_word
;

continue_stmt
      : continue_word
;
```

```
exit_stmt
      : exit_word ( expr | )
;

next_stmt
      : next_word
;

for_stmt
      : for_word for_range stmt
;

for_range
      : LPAREN !
            ( asgn_stmt SEMI expr SEMI asgn_stmt )
            RPAREN !
;


if_stmt
      : if_word LPAREN expr RPAREN stmt
      ( options {greedy=true;} : ( else_word! stmt )
      | ( else_if_stmt ))?
;

else_if_stmt
      : else_if_word LPAREN expr RPAREN stmt
      ( options {greedy=true;} : ( else_word! stmt )
      | ( else_if_stmt ))?
;

do_stmt
      : do_word stmt while_word LPAREN expr RPAREN
;

while_stmt
      : while_word LPAREN expr RPAREN stmt
;

function_call
      : NAME LPAREN! expr_list RPAREN!
;

asgn_stmt
      : l_value
      ( ASSIGN | PLUSEQ | MINUSEQ | MULTEQ | DIVEQ | MODEQ ) expr
;

function_definition
      : func_word  NAME LPREN var_list RPAREN
      LCURLY ( stmt )* RCURLY
;

var_list
      : (( l_value ( COMMA! l_value )* ) | )
;
```

```
expr_list
      :  (( expr ( COMMA! expr )*  ) | )
;


/* Some keywords */
if_word : ":-/" ;
else_word : ":-\\" ;
else_if_word : ":-/\\" ;
break_word : ":-(" ;
continue_word : ":->" ;
exit_word : ":^P" ;
next_word : ":-|" ;
for_word : ":^O" ;
do_word : ">^O" ;
while_word : "while" ;
func_word : ">-)" ;
print_word : ":-O" ;
nr_word : "#R" ;
nf_word : "#F" ;
streq_word : "EQ" ;
strne_word : "NE" ;
getline_word : ":-<" ;
```

References

http://torvalds.cs.mtsu.edu/~neal/awkcard.pdf
http://java.sun.com/j2se/1.5.0/docs/api/
http://java.sun.com/docs/books/jls/html/