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 / Proposal

This document introduces MOHAWK, yet another language for processing tabular data. Similar to its highly renowned predecessor, AWK, MOHAWK processes formatted text files with each row representing a record of white space delimited fields. MOHAWK is stream-oriented text processor, reading input one record at a time while directing results to standard output. MOHAWK programs can perform numeric calculations, text-based search and extraction, or tabular transformations, among other things.

MOHAWK can most accurately be described as AWK after walking into a sketchy rock club in the late 70s and stumbling out sometime in the early 80s with a few less brain-cells, a few new tattoos, and a nose-ring. That is, it starts with some of the basic functionality of AWK, loses some of its features, trades in its crusty image, then adds a couple pieces functionality.

MOHAWK starts with the solid AWK foundation of basic I/O, mathematical operators, conditionals, loops, functions, and regular expressions and builds upon it with several layers of novel functionality. First, MOHAWK gives the programmer the ability to link meaningful variable names to field identifiers (e.g., "price" instead of "$2"). MOHAWK also introduces new scoping rules to provide cleaner variable access more familiar to java and C/C++ programmers. As MOHAWK's most significant achievement, it has finally done away with AWK's meaningless and AWKward keywords and replaced them with concise, elegant, and symbolically meaningful IRC style smilies. No longer will programmers have to type out clumsy words such as "`print`", "`if`", and "`do`" for they have been replaced with the stylish "`:-O`", "`:-/`", and "`>^O`". Finally, in order to accentuate MOHAWK's imperative nature, all statements end with a bang ('!').

The name MOHAWK is derived from the team members' surnames with an intentional nod to AWK. Although Kernighan noted that "naming a language after its authors ... shows a certain poverty of imagination", we felt that the name (and the process of naming it) should stay true to its roots.

1.1 Feature Set

MOHAWK supports the following AWK features:

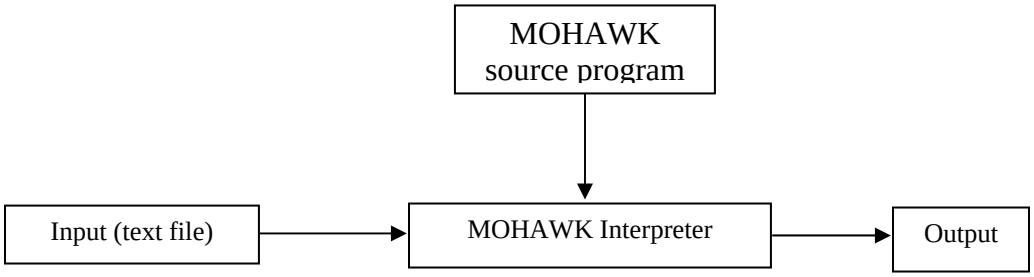
- Read a data set from a text file, parsing records and fields
- Add, multiply, subtract, divide and reorder fields
- Declare variables for storing global and local data
- Use global constants to get metadata like the number of fields
- Print to stdout
- Use constructs such as conditionals and loops
- Perform pattern matching using regular expressions
- Generate formatted reports from the original data to stdout

In addition, MOHAWK will add the following new features:

- Labeling fields with meaningful identifiers
- New variable scoping rules
- Keywords are replaced with smilies

1.2 Architecture

Like AWK, MOHAWK is an interpreted language, as opposed to a compiled language like C. The primary components of the MOHAWK interpreter are the lexer, parser, and tree walker, the last of which actually executes the MOHAWK code.



1.3 Syntax

MOHAWK source programs use a syntax that is very similar to that found in AWK, with a few adjustments. All MOHAWK programs are composed of pattern and action statements along with optional function definitions. Actions perform all of the actual data processing and output formatting. Patterns specify when to perform the corresponding action.

Patterns are:

- "`-:-|`" (forward MOHAWK)
    - Specifies an action to be executed before any input is read
    - All variables declared in the top level of this action make up the global namespace
- "`|-:-`" (reverse MOHAWK)
    - Specifies an action that runs after the entire text file has been read and processed
- Any expression that results in a numeric, boolean, or string value
    - Specifies an action that is to run only when the current record results in a non-zero value when tested against this expression
- <Nothing>
    - No pattern specifies actions that are executed on every record

Actions are a sequence of:

- Math, boolean, and string expressions
- Conditional statements
- Function calls
- Input control commands such as "`:-<`" (get a new line of text).
- Output commands such as "`:-O`" (print a line of text)

Following are some of the distinguishing features of the MOHAWK syntax:

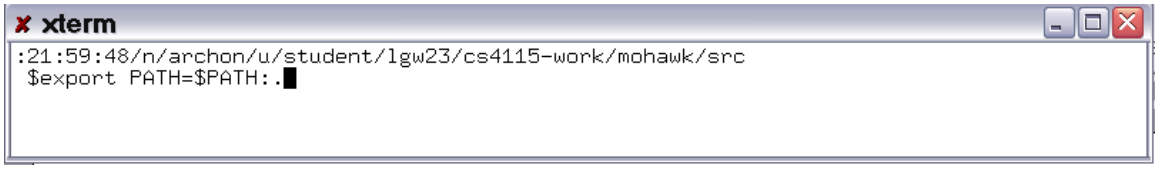
- MOHAWK uses '!' as a statement terminator instead of the more common ';'
- Like AWK, there are no data types: the same variable can represent a character, string, integer, etc.
- Java-like comments
- Most keywords are IRC style smilies

2 Tutorial

2.1 Getting Started – Simple Program Execution

2.1.1 PATH

This section is a quick guide to using the MOHAWK programming language. To begin using Mohawk, first make sure that the executable, **mohawk**, is in your PATH. Running **mohawk** invokes a script that will call the necessary antlr libraries and java executable files.



2.1.2 Writing your first Mohawk program

The Mohawk language is interpreted, and running code written in Mohawk will require that the programmer create a Mohawk program file. Conventionally, these files end with the **.oi** extension.

When running **mohawk,** the first argument to the command should be a **.oi** file, containing a Mohawk program. The second argument should be either a single data file, or should be excluded. The following images will demonstrate the format of these files and how they are run.

In the first example, the words “Hello World” are printed as part of the begin block of the program. The begin block, denoted by the **-:-|** symbol and contained in the brackets that follow this symbol, executes all statements to be run regardless of data input and assigns global variables that may be used in other program blocks. The brackets following the **-:-|** must be balanced, but may contain nested brackets within them with local scope rules. Similarly, the **|-:-** serves as the ending keyword. The print line function, :-O takes a string argument with parentheses, or empty parentheses. Statement terminators are “!” instead of the common “;”.

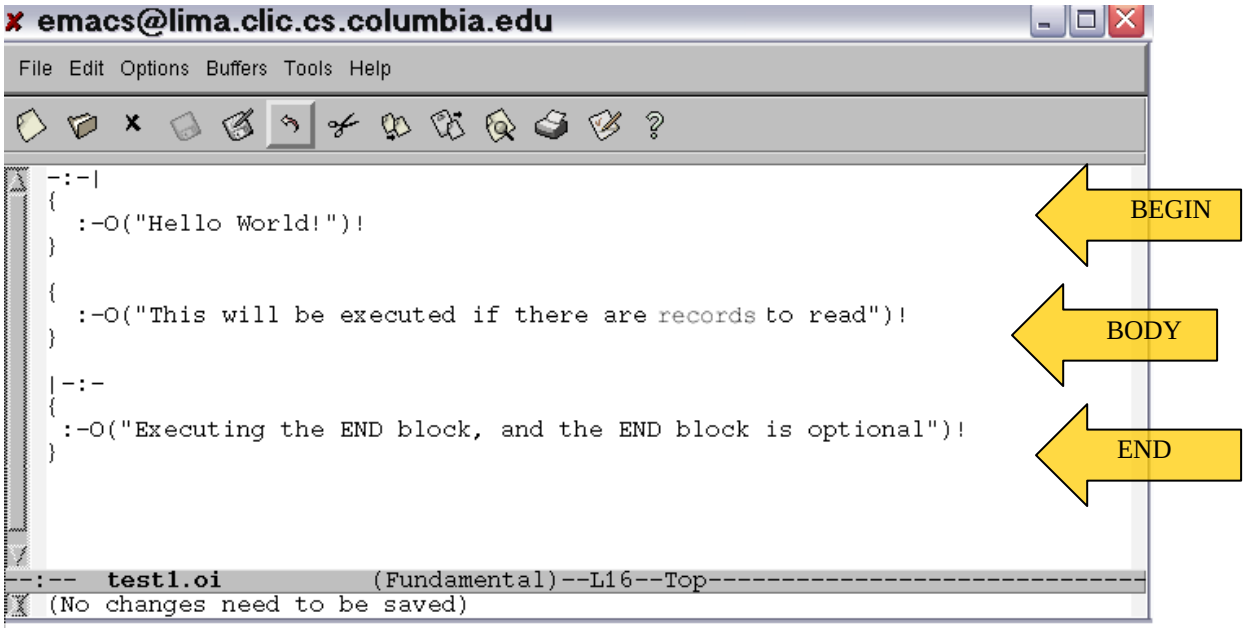


Figure 1: The requisite "Hello World" program

When we run the following Mohawk file with no data file argument, the body is not executed and the following output is generated to standard out:

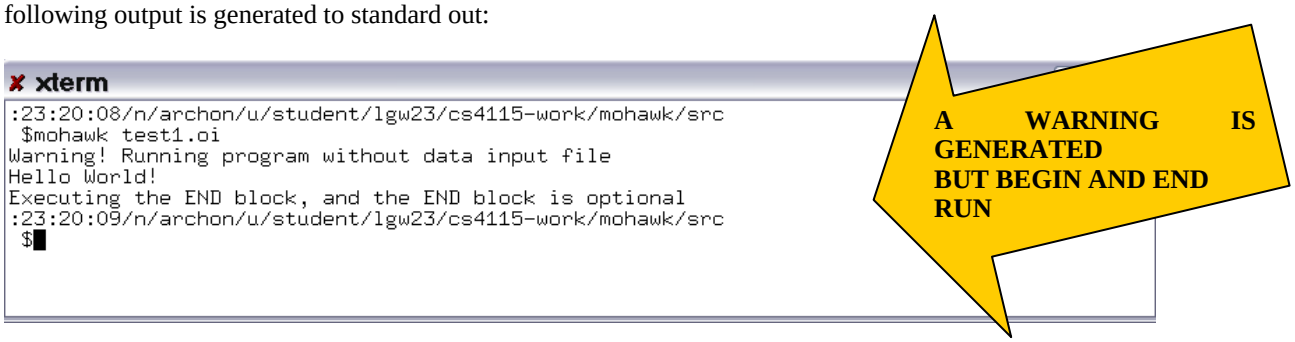


Figure 2: Output of the "HelloWorld" program run with no datafile input

A single Mohawk program can have several BEGIN and END blocks, the bodies between each would be executed for each record in the data file.

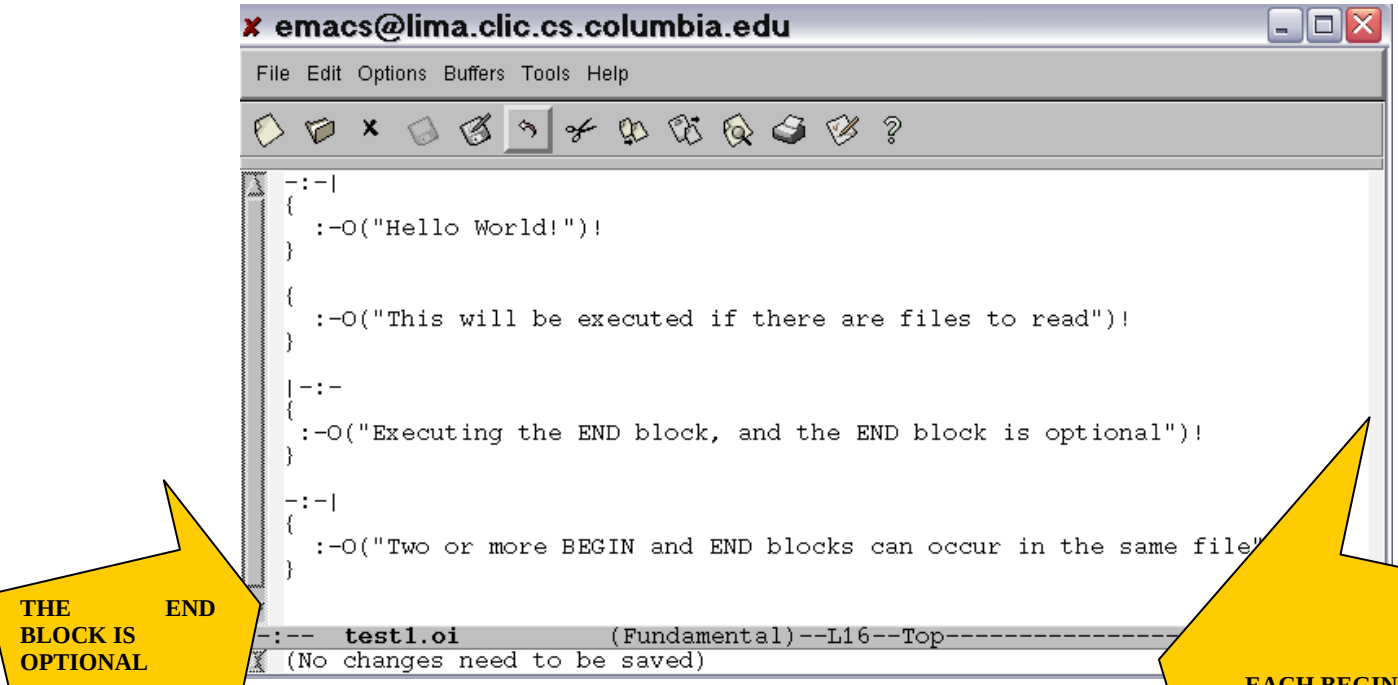


Figure 3:Two or more BEGIN/END blocks can exist in a single program file

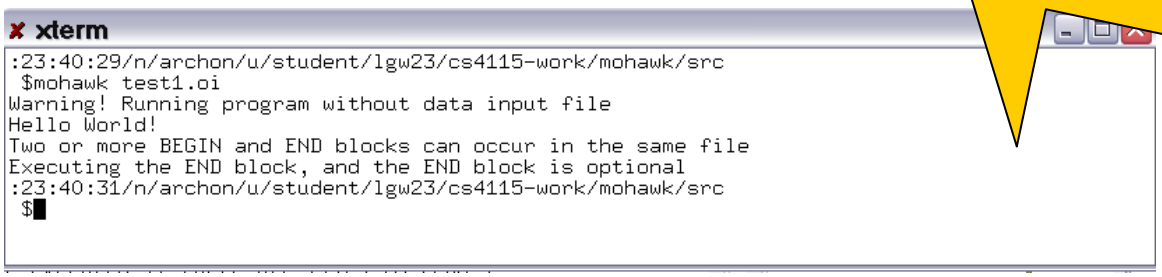


Figure 4: Output for the multiple BEGIN blocks is executed. Note the order. BEGIN blocks are always executed first, in sequential order.

2.2 Using Data Files with Mohawk Programs

2.2.1 Data file format

The Mohawk interpreter accepts data filename arguments and processes these data files as part of program execution.

The following data file, data.txt, contains records, each beginning with a name (Chris, Lauren, Ryan, or Joeng). When processed by a Mohawk program, each of these records will be read by the program body, executing the entire program body for each record. Mohawk separates fields by white space and records by newline indicators, so fields in data files should be separated by white space and records by newlines.
Each iteration of the program body would assign the field numbers $1, $2, $3 and $4 to each data text field of the current record in this example. For example, the first record has fields that correspond to:
$1: Chris $2: 9 $3: 7 $4 255.00

.

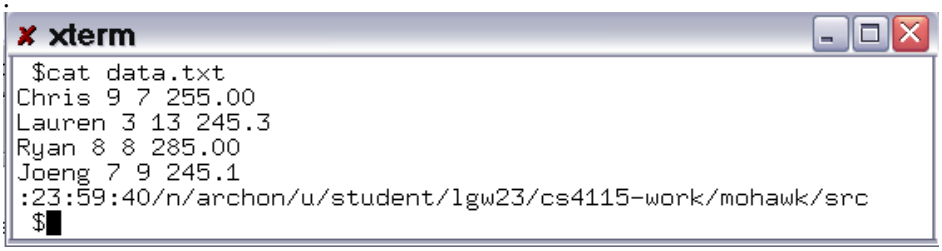


Figure 5: An example data file containing 4 records with 4 fields in each

The following Mohawk program will process these records, using $1, $2, $3, and $4 to correspond to fields of each record in the data file. This file assigns the variable **totalhours** to the first two numeric fields of each record. It then multiplies this sum by the third numeric field and adds a fourth field to the output of each record.

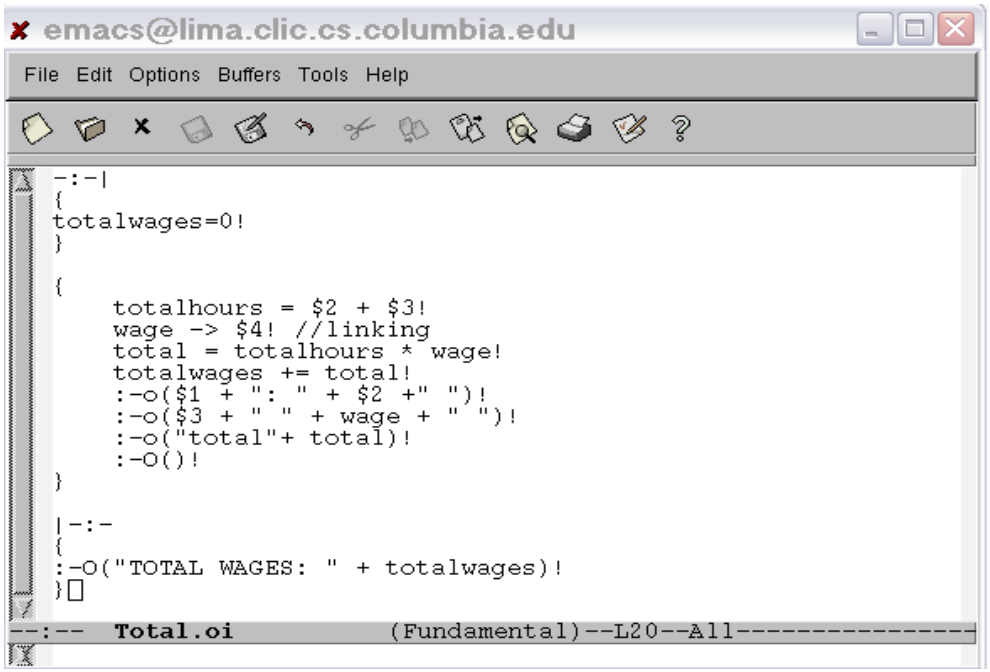


Figure 6: Example Mohawk program which reads records, performs calculations and adds a new field. This file also introduces the linking operator, used to link the variable "wages" to $4, assigning an alias to $4. One additional print *function, print rather than* print line, *is introduced here as :-o*

To run the Mohawk program above, **Total.oi** on the example file, supply the file name as the second argument to the Mohawk command, where the .oi source file is the first argument as in our first example.

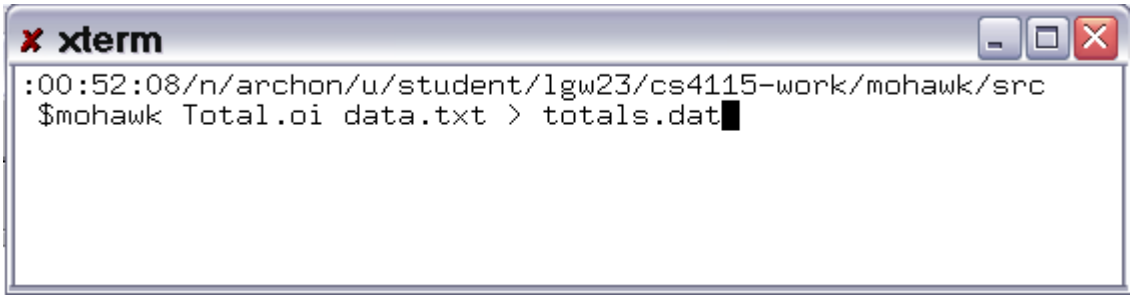


Figure 7: mohawk is run with source code Total.oi with input data file data.txt and its output is routed to the totals.dat file

The result of the last program run can be seen in totals.dat:

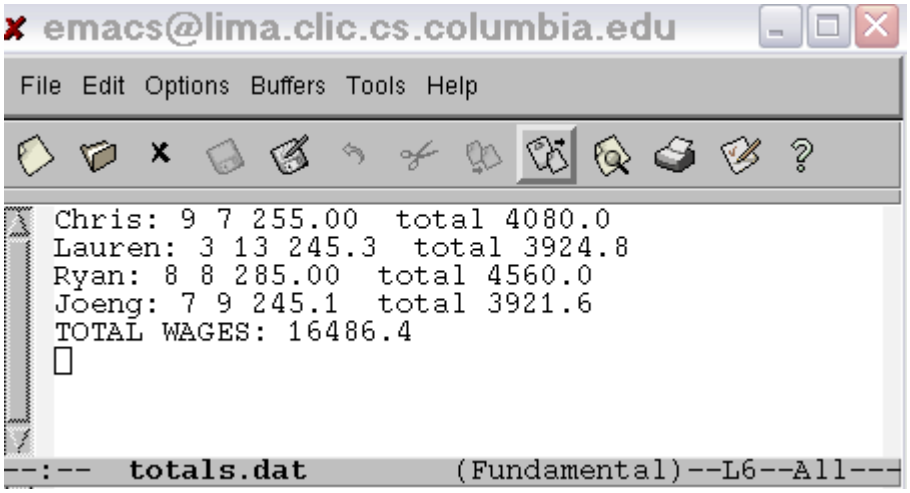


Figure 8: The results of the Total.oi program run on data.txt

2.2.2 Using record and field information

Like AWK, Mohawk supplies commonly used information, like the total number of records in the file and the total number of fields for each record, in an easy to access variable #R and #F respectively.

Using these features and some other, the Totals program can be enhanced to give averages, and process other computaions. Using the same initial data input file, data.txt, the following enhanced Mohawk program computes and reports averages.

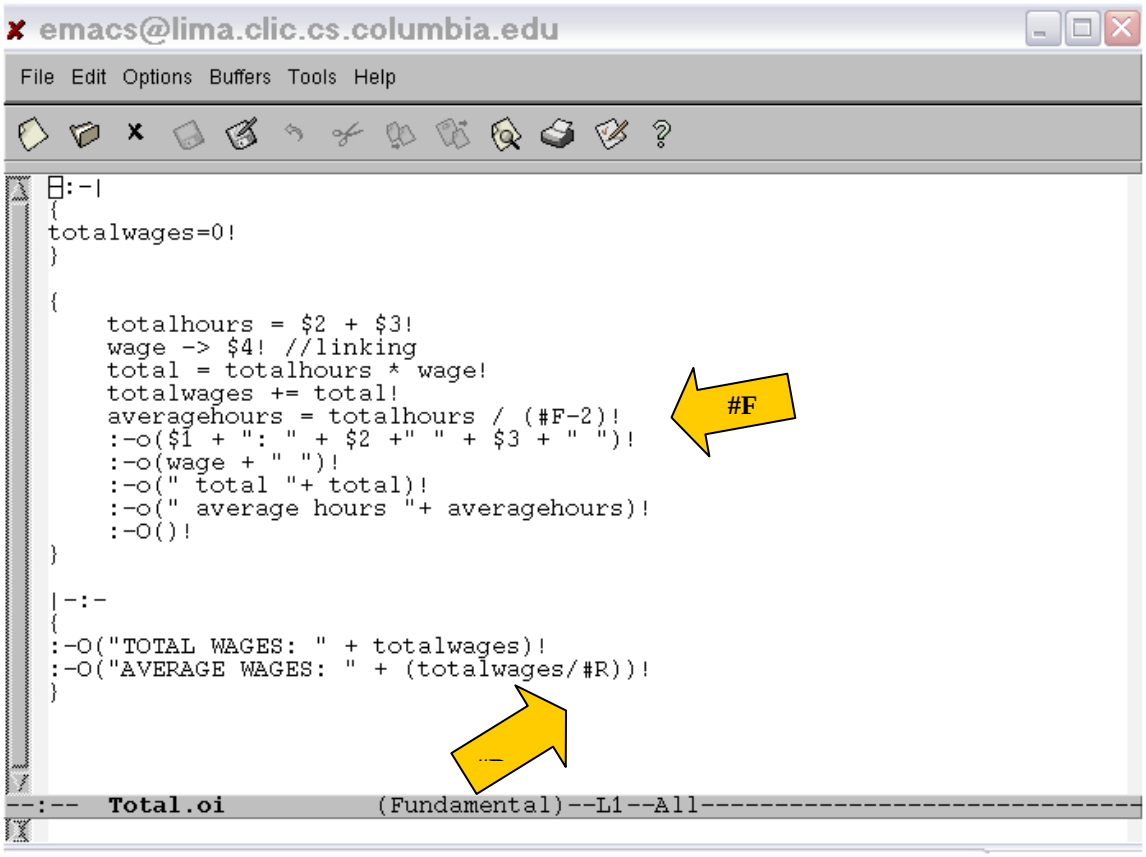


Figure 9: The enhanced Total.oi program makes use of #R and #F

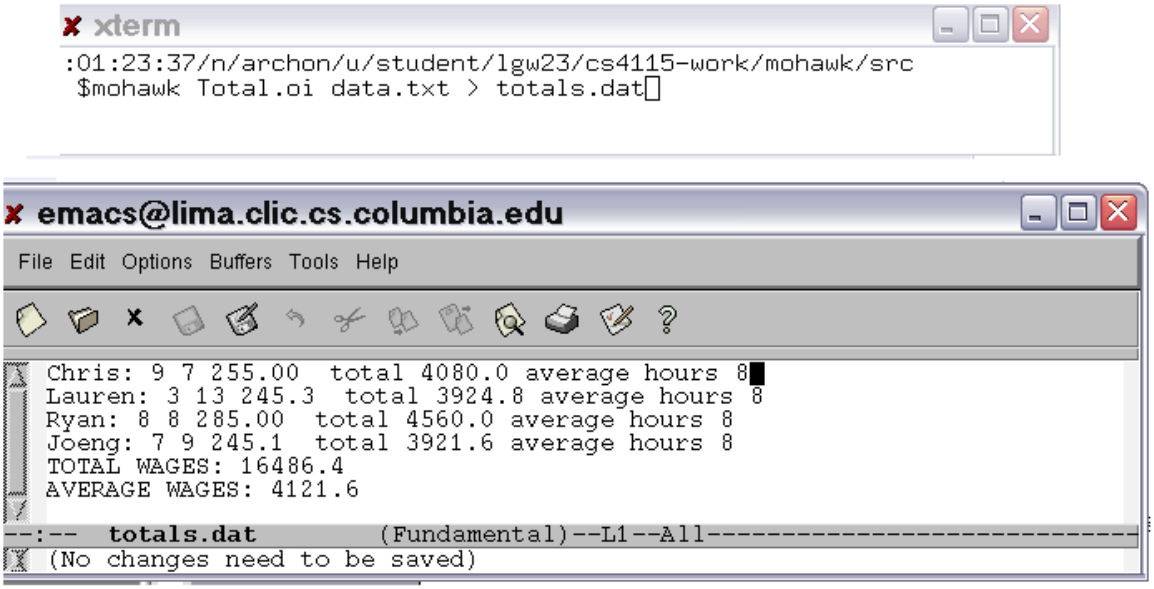


Figure 10: The output reflects the enhancements

2.2.3 More record processing: control statements and regular expression matches

The following example uses regular expressions to test a pattern containing “Ryan”. If “Ryan” or “ryan” is found in the name field, $1, the record is skipped using the next keyword **:-|**

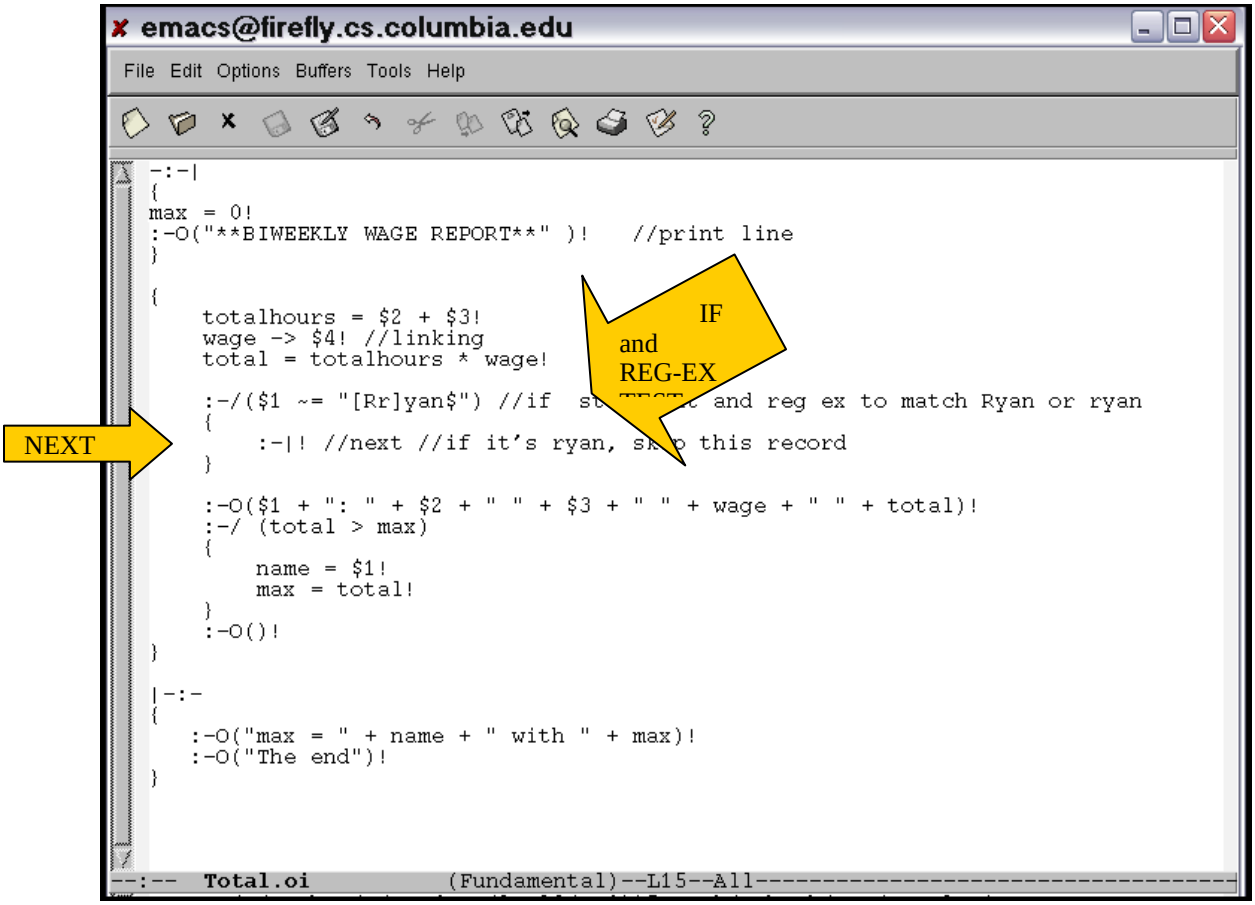


Figure 11: The if logical statement :-/ is introduced, as well as a simple regular expression match. The control statement :-| or next, is also introduced

2.3 Creating and calling functions in Mohawk

2.3.1 Function creation, loop construction, control flow within loops

Functions can be created in Mohawk using the keyword >-) followed by the function name and the parameter list, as follows. This example program uses recursion and a Mohawk implementation of Euclid's algorithm to find the greatest common denominator. The body of the program uses a global, max, which is assigned to 20. This global variable is used to control a for loop, which generates numbers in multiples of 4, then finds the GCD of these generated numbers, and the current field value.

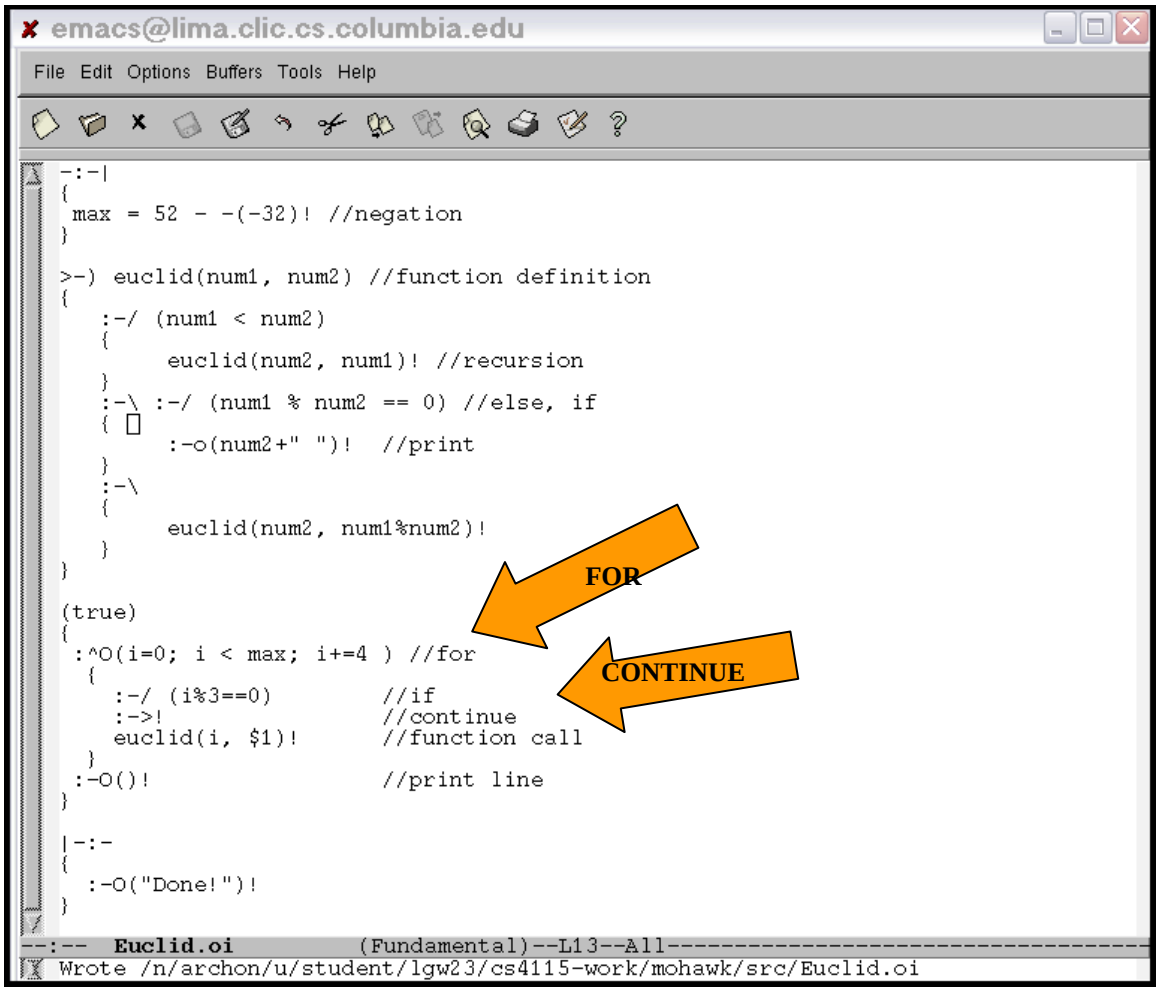


Figure 12: The GCDs are printed for each record in the output file. As a second test, the continue statement, :-> checks for multiples of 3 within the loop, and skips the rest of the statements in the loop to move on to the next iteration when the condition is true

The following sample data files reflect the input and output of the above Mohawk program when run on the input file and piped to the output file.

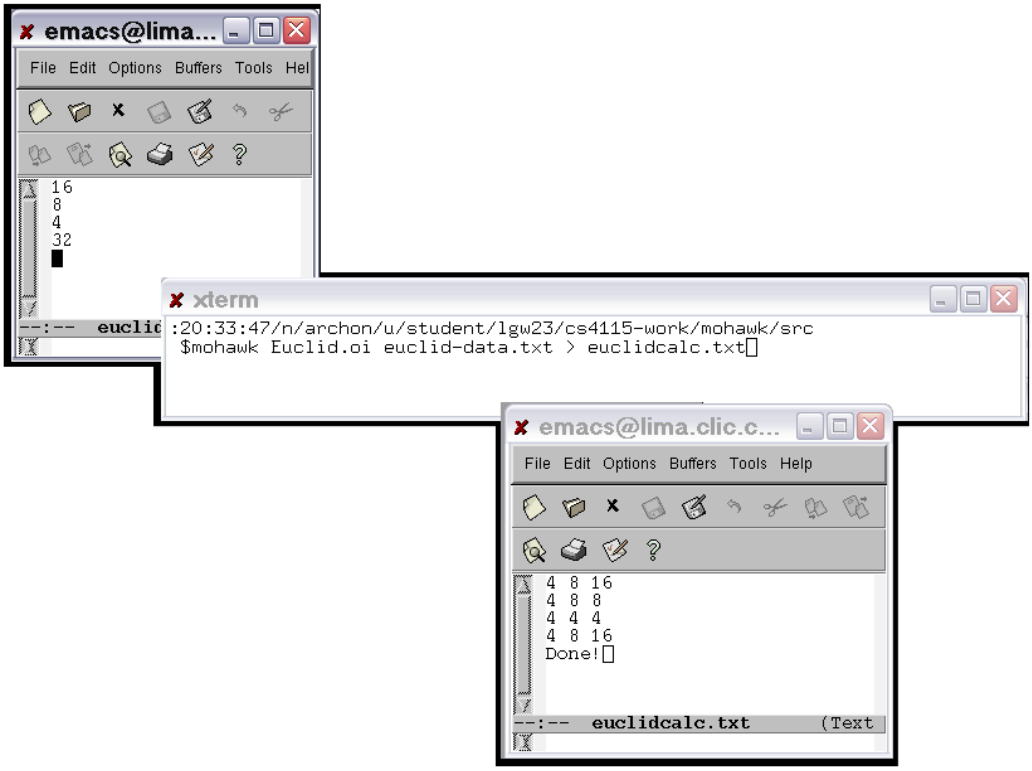


Figure 13: The results of the Euclid.oi file given the euclid-data.txt input file

3 Language Reference Manual

3.1 Introduction

This section 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 section also provides a sample program, a quick reference guide, and the complete grammar in ANTLR notation.

Conventions used

In this section, 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. Code is presented in `Courier New`, and placeholders in the code are presented in *`italics`*.

3.2 Language Construction & Execution

3.2.1 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.

Please note that, unlike in AWK, the *pattern* cannot be a regular expression surrounded by slashes; it must be a logical expression.

3.2.2 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-action statements (if any exist) are executed, starting at the top.

2. For each record in the input file, all patterns (except for the begin and end patterns) 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. If there are no records in the input file, or the input file is not specified, these pattern-action statements will not be executed.

3. Finally, the statements in the "`|-:-`" pattern-action statements (if any exist) are executed, starting at the top.

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.

In the case in which no *program-file* is provided, or the provided *program-file* name refers to a file that does not exist, MOHAWK will continue to prompt the user to enter a valid name for the *program-file*. Once a valid name is entered (i.e., the name of a file that exists), the user is then prompted to enter the name of the *input-file*, if desired, or the user can enter "n" to indicate that there is no *input-file*. When prompted for an *input-file*, if the user enters the name of a file that does not exist, MOHAWK will display a warning and execute the program with no data file.

In the case in which the *input-file* name provided as an argument to MOHAWK refers to a file that does not exist, MOHAWK will continue to prompt the user to enter a valid name for the data file; the user can enter "n" to indicate that there is no *input-file*. Once a valid name is entered (i.e., the name of a file that exists), MOHAWK executes the program as described above.

3.2.3 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:

```
>-) 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.

3.3 Lexical Conventions

3.3.1 Tokens

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

3.3.2 Line Separators/Terminators

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

3.3.3 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.

3.3.4 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.

3.3.5 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.

3.3.6 Keywords

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

```
:-/             :->             :^~
:-\             :-|             >-)
:-<             -:-|
:^0             |-:-            true
:-(             :^P             false
>^0
```

3.3.7 Operators

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

```
+         +=        &&        >         ++
-         -=        ||        <         --
*         *=        ~         >=        .
/         /=                  <=        ~=
%         =                   ==
^         ->                  !=
```

3.3.8 System Variables

MOHAWK defines two system variables that are available to programmers anywhere in the program.

The system variable "#R" can be used to refer to the total number of records in the input file.

The system variable "#F" refers to the number of fields in the current record. If used in a "begin" block, its value is zero; if used in an end block, its value is the number of fields in the **last** record of the data input file. 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 there are no records in the input file, or no input file is used, both #R and #F will be zero throughout the program.

3.3.9 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 the 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).

3.4 Datatypes

Like its predecessor AWK, MOHAWK is a typeless language. That is, variables and record fields may be floating point numbers, integers, strings, or all three. 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.

3.4.1 Initialization

There are no explicit variable declarations in MOHAWK; variables are created when they are initialized. Initialization is performed by the following:

```
name = expression !
```

name: the name of the variable (which must be a valid identifier, as explained in section 3.3.5)
expression: a string or numeric literal, the keyword “true” or “false”, the name of another variable, the name of a field variable, or a valid MOHAWK expression

MOHAWK does not, in fact, require that variables be initialized. However, if an uninitialized variable is used in a MOHAWK expression, a warning message will be displayed. Uninitialized variables have the numeric value 0 and the string value “” (the null, or empty, string) and are treated like any other variables when used with the different types of operators.

3.4.2 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} $\text{ExponentPart}_{opt}$
. Digits $\text{ExponentPart}_{opt}$
Digits ExponentPart

ExponentPart:
ExponentIndicator SignedInteger

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

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.

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 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.

3.4.3 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 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.

Non-numeric strings are considered to have a numeric representation of 0, except for the string "true", which is considered to be a 1.

---

[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)

3.4.4 Booleans

There is no primitive Boolean datatype in MOHAWK; the Boolean value is represented internally as a string. 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 (unless the string representation is the string "false"), and logical "false" if the string is empty/null.

3.4.5 Scope

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

Global variables must be defined in a "`-:-|`", 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 "linking" 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 "`|-:-`" (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. New blocks of code can be created anywhere within a MOHAWK program by using the left and right braces, and variables defined in a blocks will only be in scope for that block. Similarly, local variables initialized within a function are in scope only for that function.

3.4.6 Namespaces

Namespaces are abstract containers that are filled by unique names for a particular namespace. This means that variables of the same namespace cannot share the same name. There are two sets of namespaces in MOHAWK: one for functions and one for variable names.

All functions in a MOHAWK program must have unique names, regardless of the number of arguments they have (i.e., there is no notion of function overloading). If a function is declared with the same name as a function that has already been declared, MOHAWK will display a warning message and only the first function declaration will be used.

Variable names are used only within the scope in which they are defined, thus local variables in different scopes can have the same name. Similarly, parameters defined in function declarations can also use the same name as local variables being used elsewhere in the program.

3.5 Operators

3.5.1 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, except for the string "true", which is treated as the integer value 1.

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

Precedence Level	Operator	Usage	Description
1	`++`	`a++`	Increments the value of `a` by 1
1	`--`	`a--`	Decrements the value of `a` by 1
2	`- (unary)`	`-a`	Returns the negation of the value of `a`
3	`*`	`a * b`	Returns the product of `a` and `b`
3	`/`	`a / b`	Returns the quotient of `a` and `b`
3	`%`	`a % b`	Returns the modulus of `a` and `b`
4	`^`	`a ^ b`	Returns the value of `a` raised to the power of `b`
5	`+`	`a + b`	Returns the sum of `a` and `b`
5	`-`	`a – b`	Returns the difference of `a` and `b`

It is worth noting here that the increment and decrement operators are **not** treated as post-increment and post-decrement like they are in C/C++ or Java; in MOHAWK, they are evaluated as normal expressions, and the incrementing or decrementing of the value happens according to the order of precedence defined above. Also, these two operators can **only** be used with variables, not numeric literals.

Note that, because MOHAWK does not have explicit datatypes, it is possible to use these operators with mixed operands (float and int, for instance) or even non-numeric operands. The operators follow these rules for handling these situations:

Negation

- If the operand represents an integer, the return value will be an integer.
- If the operand represents a float, the return value will be a float.
- Otherwise, the return value is 0.

Multiplication

- If the operands both represent integers, the return value is an integer.
- If one operand represents a float and the other represents an integer, or both are floats, the return value is a float.
- If either operand represents a non-numeric string, the return value is 0.

Division

- If the first operand (numerator) represents a non-numeric string and the second operand (denominator) represents a number, the return value is zero.
- If the second operand (denominator) is zero or a non-numeric string, the return value will be "Infinity" or "-Infinity", according to the sign of the first operand. If the first operand is also zero (or is a non-numeric string), the return value will be "Infinity".
- If the operands both represent integers, the return value is an integer that represents the floor of the quotient.
- If one operand represents a float and the other represents an integer, or both are floats, the return value is a float.

Modulus

- If the first operand represents a non-numeric string and the second operand represents a number, the return value is zero.
- If the second operand (denominator) is zero or a non-numeric string, the return value will be "Infinity" or "-Infinity", according to the sign of the first operand. If the first operand is also zero (or is a non-numeric string), the return value will be "Infinity".
- If the operands both represent integers, the return value is an integer.
- If one operand represents a float and the other represents an integer, or both are floats, the return value is a float. Please note that there are known issues with rounding errors occurring when using float values and this operation.

Exponent

- If an operand represents a non-numeric string, it is treated as a zero.
- If the second operand is zero or a non-numeric string, the return value will be 1 (either a float or an integer depending on the type of the first operand).
- The return value will be of type float, unless both operands represent integers, in which case the return type will be an integer.

Addition

- If the operands both represent integers, the return value is an integer.
- If one operand represents a float and the other represents an integer, or both are floats, the return value is a float.
- Otherwise, the return value is the concatenation of the string representations of the two operands.

Subtraction

- If the operands both represent integers, the return value is an integer.
- If one operand represents a float and the other represents an integer, or both are floats, the return value is a float.
- If one operand represents a non-numeric string and the other represents an integer or a float, the return value is an integer or a float (accordingly) and the non-numeric string is represented as 0.
- If both operands are non-numeric strings, the return value is 0.

3.5.2 String Operators

The string operator 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
`.`	`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 operands are non-numeric strings. This operator is provided here merely as a convenience to the programmer.

3.5.3 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 will treat non-numeric strings as having a value zero, as defined in section 3.5.1 above.

3.5.4 Comparison Operators

Comparisons are performed on any literal or variable. If a variable or string literal has a string value that is numeric, then its numeric value will 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 operands is a non-numeric string, the operands will be compared as strings, using the Java rules for string comparison.

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`

3.5.5 Logical Operators

As described above, there are no Boolean datatypes in MOHAWK, but variables and expressions can have logical values (see section 3.4.4). 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

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

3.6 Regular Expressions

3.6.1 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.

3.6.2 Context

An example of the boolean regular expression context:

```
:-/ ($1 ~= "REGEX")
{
  :-O ("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.

As noted above, the operands to the ~= operator can be variables, field variables, or string literals.

3.6.3 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. Brackets **must** contain at least one character.

[Tt]his is a lot of fun

will match occurrences of "This" and "this".

- 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:

"This is a lot of fun" ~= "[a-z_0-9A-Z]"

returns **true** as the first character, 'T' 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

3.6.4 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

3.6.5 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

3.6.6 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”

3.7 Expressions

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

3.7.1 Simple Expressions

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

3.7.2 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.

3.7.3 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.

3.7.4 Logical Expressions

A logical expression is a relational expression, followed by zero or more sets of a logical operator (see above) and another relational expression.

3.7.5 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.

3.8 Statements

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

3.8.1 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.

3.8.2 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 ")":

function (*expression*) !

3.8.3 Print Functions

The current version of MOHAWK supports the following internal functions for printing.

`:-O`
Description: Prints a value to standard output, followed by a newline character.
This function prints out the string, data, and values of variables.
Syntax: `:-O` (string)!
Example: `:-O` ("This is my value" + value)!

`:-o`
Description: Prints a value to standard output, without a newline character.
This function prints out the string, data, and values of variables.
Syntax: `:-o` (string)!
Example: `:-o` ("This is my value" + value)!

3.8.4 User-defined Functions

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

```
>-) function_name (parameter-list) { statements }
```

function_name: The name of the function, which must be a sequence of any number of letters, digits, and underscores characters; function names cannot start with a number and must start with a letter or underscore.
parameter-list: A comma-separated list of valid MOHAWK identifiers; these identifiers will only be in scope within this function definition.
statements: The MOHAWK statements to be executed when this function is called.

The following is an example of user-defined function.

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

3.8.5 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.

3.8.6 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:

```
:^~ (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
:^~ (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.

3.8.7 Control Statements

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

Keyword	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 and move to the “end” blocks
:-|	next	finish processing the current input record and move on to the next one, starting with the first pattern-action statement
:-<	getline	Set $0 from the next input record

Break
The “:-(” keyword is similar to “break” in Java. It can only be used inside a for, while, or do/while loop; if it appears outside a loop, a compile-time error will occur. This keyword stops the loop from executing, and causes the program to go to the next statement following the loop, if one exists.

Continue
The “:->” keyword is similar to “continue” in Java. It can only be used inside a for, while, or do/while loop; if it appears outside a loop, a compile-time error will occur. This keyword causes the program to skip over the rest of the statements in the loop and re-evaluate the conditional expression to decide whether the loop should continue.

Exit
The "`:^P`" keyword is similar to "exit" in AWK. This keyword can be given an optional exit code as an argument in parentheses. If no argument is provided, the exit code is 0.

This keyword can be used anywhere in the program, according to the following:

- If called from a "begin" block, the program will stop processing the current and all other "begin" blocks. If the exit code evaluates to zero, the program will then execute the "end" block(s), if any exist, and then terminate. If the exit code is non-zero, the program will terminate immediately.
- If called from an "end" block, the program will terminate immediately.
- If called from any other pattern-action statement, the program will stop processing the pattern-action statement(s) for this and all other data input records. If the exit code is zero, the program will then execute the "end" block(s), if any exist, and then terminate. If the exit code is non-zero, the program will then terminate immediately.

Next
The "`:-|`" keyword is similar to "next" in AWK. It tells MOHAWK to stop processing the current data input record and move onto the next one. When this keyword is encountered, all statements in the current and subsequent pattern-action statements are skipped for the current data input record, the next record is read, and the pattern-action statements start again from the top of the program. If the current data input record happens to be the last data input record, this keyword will cause the "end" block(s) to be evaluated, if any exist.

If this keyword is used in a "begin" block, a warning message is displayed and the program stops processing the "begin" blocks and moves to the pattern-action statements for the data input records. If this keyword is used in an "end" block, a warning message is displayed and the program terminates immediately.

Getline
The "`:-<`" keyword is similar to "getline" in AWK. It tells MOHAWK to load the next data input record and continue processing as normal. If the current data input record happens to be the last data input record, this keyword will produce a warning message and will continue processing the current record.

If this keyword is used in a "begin" block, no action will be taken. If this keyword is used in an "end" block, a warning message is displayed and the program continues as normal.

3.8.8 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 "}". As mentioned above, any variables defined in a block are only in scope for that block.

3.9 Error Handling

There are typically two types of errors in a MOHAWK program: syntax (compile-time) errors and run-time errors. Any of these errors would cause the program to terminate.

3.9.1 Syntax Errors

MOHAWK attempts to find all syntax errors before starting the execution of the program. When MOHAWK finds an error in the syntax of the program, it will print out the error to the screen to inform the user of the syntax error. It will then continue to look for more syntax errors, stopping when it either gets to the end of the program or when 100 errors have been detected.

Error messages include the following:

- Line number of the error
- Character number of the line on which the error occurred
- Error description

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

```
Oi! Oi! Oi! line 6:8: unexpected token: }. Perhaps missing a '!'?
```

Here, the error has occurred at character 8 of line 6. In this situation, the statement terminator "!" is most likely missing. Please note that in MOHAWK, there are many instances in which one syntax error causes multiple error messages to appear. Fixing the first error may cause the others to go away.

3.9.2 Runtime Errors and Warnings

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 errors. Common runtime errors such as dividing by zero, using uninitialized variables, and calling functions with the wrong number of arguments can easily crash programs in most languages, but in MOHAWK the program will continue to run. However, in these situations a warning message will be displayed.

There are, however, some situations in which a MOHAWK program will terminate with a runtime error. One such situation is when a developer attempts to use the "`:-|`" (next) keyword in an "end" block. Also, if any internal errors occur during the execution of the program (such as running out of memory, stack overflow errors, etc.), the error will be reported and the program will terminate.

3.9.3 Exit Status

MOHAWK programs that run to completion exit with "normal" status. However, in the situations where the "`:^P`" (exit) keyword is used to terminate the program, the exit status will be 0 if the keyword is given a status zero (or no status at all), and the exit status will be 1 if the keyword is given a non-zero status. If a program terminates because of a runtime error, its exit status will also be 1.

3.10 Quick Reference Guide

In case you're having trouble with all those smiley faces:

Keyword	Translation
`:-/`	if
`:-\`	else
`:-\ :-/`	else if
`:^0`	for
`:^~`	while
`>^0`	do
`:-(`	break
`:->`	continue
`:^P`	exit
`:-|`	next
`:-<`	getline
`:-0`	println
`:-o`	print
`>-)`	function
`-:-|`	begin
`|-:-`	end

3.11 Sample Program

Datafile:

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

Program:

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

// declare other global 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 += total!
}

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

Output:

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

3.12 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 : ":-<" ;
```

3.13 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/

4 Project Plan

4.1 Processes Used

The development process for MOHAWK can be broken down to sequences of steps. The team members were responsible as a group to plan and scope out the entire project with detailed specifications. The process itself followed general soft-eng waterfall practice, began with planning out project responsibilities, general features of the language, group meeting times. The specifications itself ranged from language design choices, version control system, testing procedure, language features and syntax choices. Throughout the process of development, testing was also done at various milestones on varying levels of testing. The following is more detailed outline of the process of our project.

- Planning:
    - Create specification
    - Create software architecture
    - Develop code in multiple streams (frontend and backend)
    - Merge the code
    - Testing
    - Bug fixes
    - Feature freeze
    - Regression test
    - Bug fixes again
    - Code freeze
- Specification:
    - Shared document that listed known issues
    - Fixed bugs
    - Open bugs
    - To-do list
- Development:
    - Developed the Lexer and the Parser *(Responsibility: Ryan and Lauren)*
    - Implemented the Tree Walker, component architecture for backend *(Responsibility: Chris)*
    - Operator Implementation *(Responsibility: Chris)*
    - Symbol Table *(Responsibility: Ryan)*
    - Exceptions and MohawkMain implementation *(Responsibility: Lauren)*
    - Fixed Bugs *(Responsibility: Ryan and Lauren)*
- Testing: *(Responsibility: Joeng)*
    - Unit Testing
    - Parser Testing
    - Functional Testing

4.2 Programming Style Guide

This section describes the programming style agreed upon and used by the developers. It is primarily based on the Java Code Conventions (http://java.sun.com/docs/codeconv/html/CodeConvTOC.doc.html).

4.2.1 Naming

Classes, methods, and variables should all have meaningful names (except for loop indices and other temporary values) and be named as follows:

- Class names should start with capital letters, should contain only letters (no numbers or other punctuation), and should capitalize letters starting a new word in the name, e.g. MohawkFunctionHandler

- Method and variable names should start with lower case letters, should contain only letters (no numbers or other punctuation), and should capitalize letters starting a new word in the name, e.g. getFields or globalSymbolTable

For simplicity, no packages will be used, but all class files should begin with the word "Mohawk".

4.2.2 Indentation and Spacing

Each level of indentation should be four spaces. Tabs (8 spaces) can be used for deeper levels.

Whitespace (blank lines) should be used to demarcate different sections of code within the same method, or to separate methods in the same class.

Curly braces should appear on the line following a conditional or control statement, or following the beginning of a method, like this:

```
if (records != null)
{
    ...
}
```

All statements should fit onto one line, i.e. they should not need to continue to a second line.

4.2.3 Comments

Each class must have comments at the top of the class file (after the import statements) describing what the class does.

Each method must have comments above it describing what the method does, in terms of the input it takes and the output it returns. Javadoc comments are not necessary.

Developers should use inline comments as much as possible, using the // style comments to describe blocks of code.

4.2.4 Error Handling

Developers must ensure that arguments passed to methods are valid, i.e. that they are within range or not null. The beginning part of a method should check the validity, and display an appropriate warning message if the program can continue, and an error message if the program must terminate.

All warning and error messages must be written to System.err (not System.out); classes that extend MohawkOperator can use the "error" method. Warning messages are to be displayed if the program will continue and should start with "Warning!", followed by a meaningful message. Error messages are to be displayed if the program will terminate and should start with "Error!", followed by a meaningful message.

If an unexpected exception is caught during the flow of the program, a warning or error message should be writing to System.err, along with the value returned from the getMessage method of the exception.

4.2.5 Other Concerns

Because most of the methods in the backend of MOHAWK are static, the member variables of the classes are also typically static. Therefore, before attempting to use one of these member variables, the developer must check to see that it is not null, and initialize it if it is.

Where possible, developers should use lazy evaluation and not call a method before its return value needs to be used. Similarly, developers should try to avoid calling the same method twice if its return value can be reused.

4.3 Project Timeline

Following are the dates of the key activities and deliverables in the development of the MOHAWK programming language:

Date	Activity/Deliverables
September 27	Deliver initial proposal
October 4	Agree on architecture, including technology decisions (development platform, source control, etc.) and coding conventions
October 6	Agree on MOHAWK syntax; Start writing language reference manual
October 10	Start development of lexer/parser with ANTLR
October 20	Deliver language reference manual; Start development of tree walker and back-end
October 31	Start creation of test plan
November 7	Start testing of lexer/parser
November 14	Start testing of tree walker and back-end
November 28	Complete development of lexer/parser; Start writing final report
December 8	Complete development of tree walker and back-end
December 13	Complete all unit testing; Feature freeze; Start regression testing
December 20	Code freeze; Deliver final report

4.4 Roles and Responsibilities

The MOHAWK team consisted of one team member as the primary lead for documentation and another member for testing. There were two team members responsible for front-end development, including parser, lexer, and semantic analysis) and back-end components. Additionally, one team member took the role of team leader to assign tasks, coordinate meeting times, led meeting discussions with the TA, and helping with sections of the project which need attention. The roles and responsibilities of each team members were broken up in the following manner.

Chris Murphy
- Team Leader
- Architecture Design
- Back-end Components
- Documentation

Joeng Kim
- Test plan
- Unit and Functional Testing
- Document Editor

Lauren Wilcox
- Grammar
- MohawkMain
- Exceptions and Error handling
- Bug Fixing
- Maintained CVS repository

Ryan Overbeck
- Grammar
- MohawkParser
- MohawkLexer
- Symbol Table

4.5 Development Environment

Mohawk Parser and Lexer was developed using ANTLR 2.7.2. The components of Mohawk which were Java dependent were developed Java SDK 1.4.2.

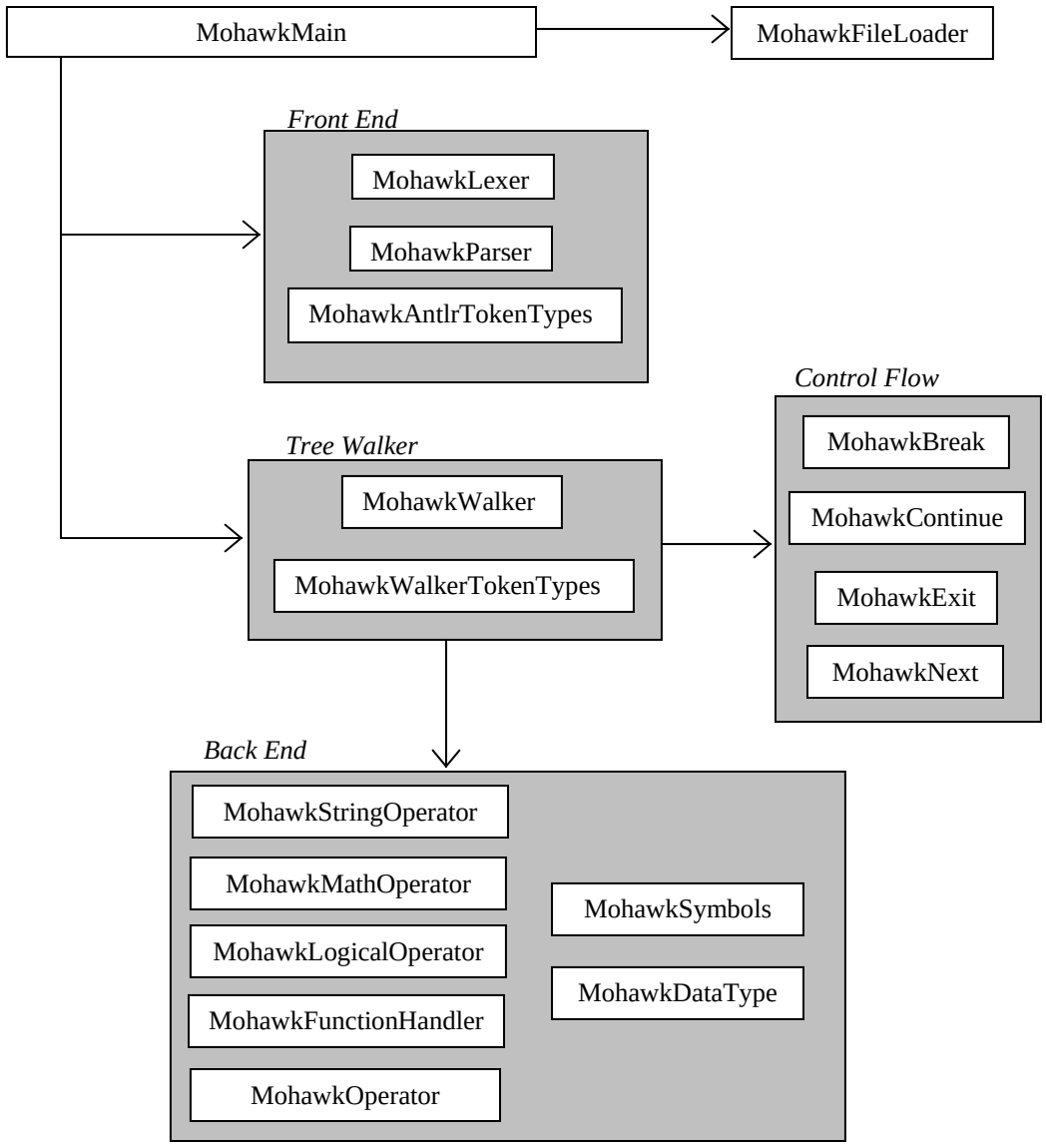
All – CVS, Java
Chris – NetBeans on WinXP, emacs on Linux
Lauren – NetBeans on WinXP, emacs on Linux
Ryan – emacs on Linux
Joeng – vi editor on Linux, JUnit library for testing

4.6 Project Log

09/08: Form group
09/13: Discuss ideas for language; decide to use interpreted language design
09/20: Agree to have AWK-like language; come up with name MOHAWK
09/27: Deliver proposal/whitepaper
10/04: First meeting with Chris Conway
10/06: The infamous trip to Amsterdam Café where we decided to use smiley faces
10/11: Meeting with Chris Conway; Ryan and Lauren work on first version of lexer/parser
10/13: Team meeting to work on LRM
10/16: Deliver first draft of LRM to Chris Conway
10/18: Meeting with Chris Conway to discuss draft of LRM; Test Plans also discussed
10/20: Deliver LRM
10/23: Chris tries to implement MOHAWK in one day
10/24: Unit test suite completed using JUnit
10/25: Receive feedback on LRM from Chris Conway; no meeting; Status report sent to Chris Conway
10/27: MOHAWK Parse completed
10/29: More MOHAWK development; MOHAWK Lexer completed
11/01: Team meeting; Meeting with Chris Conway
11/06: Team meeting to discuss scoping issues
11/10: Parser Tester completed
11/14: Team meeting to discuss exception handling
11/17: Team meeting
11/22: Meeting with Chris Conway
11/25: Functional Tester completed
11/29: Team meeting to discuss break, next implementation
12/06: Team meeting to discuss code freeze, bug fixing, final report
12/13: Team meeting to review outstanding issues, discuss final report
12/15: Team meeting to discuss presentation and final report

5 Architectural Design

This section describes the architecture of the components of the MOHAWK programming language.

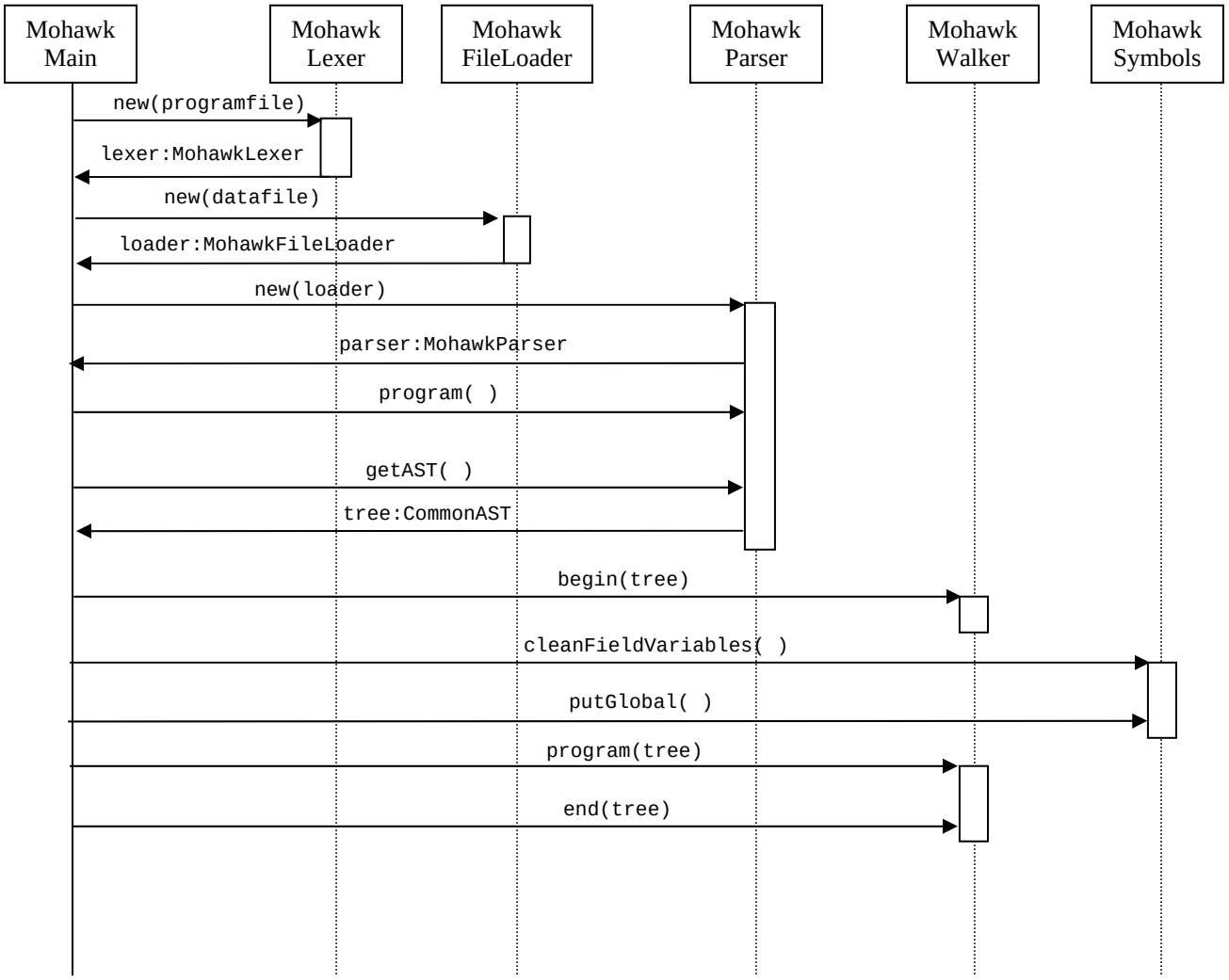


5.1 MohawkMain

This is the entry point into the Java program that runs the MOHAWK interpreter. The "main" method of this class is responsible for the following:

1. Read the name of the program file and the data file (if one exists) from the command line
2. Create a MohawkLexer from the program file
3. Use the MohawkFileLoader to read the data file and parse it into records and fields
4. Create a MohawkParser from the MohawkLexer
5. Use the MohawkParser to parse the program file, ensure that it is syntactically correct, and create an abstract syntax tree (AST)
6. Use the MohawkWalker to walk the AST and evaluate and execute the statements and expressions in the "begin" blocks.
7. For each record in the data file:
    a. Call methods to refresh update the symbol tables MohawkSymbols component
    b. Use MohawkWalker to evaluate and execute the pattern-action statements
8. Use the MohawkWalker to walk the AST and evaluate and execute the statements and expressions in the "end" blocks.

The sequence diagram for MohawkMain.main is as follows:



In the case in which no arguments are provided on the command line, or the provided program name refers to a file that does not exist, the MohawkMain.main method is responsible for prompting the user to enter a valid name for the program file. Once a valid name is entered (i.e., the name of a file that exists), the user is then prompted to enter the name of a data file, if desired. The method then goes into the normal flow described above.

In the case in which the provided data file name refers to a file that does not exist, the MohawkMain.main method is responsible for prompting the user to enter a valid name for the data file. Once a valid name is entered (i.e., the name of a file that exists), the method then goes into the normal flow described above. However, if the user chooses not to enter a data file name, or none is provided as a command line argument, then the main method executes the normal flow but *does not* call MohawkWalker.program.

This component was developed by Lauren and Chris.

5.2 MohawkFileLoader

This class loads the data file and is responsible for keeping track of all the records and their fields. It does so by exposing the following methods:

- **constructor**: When the object is created, its constructor is passed the name of the file containing all of the data records. The constructor uses the Scanner class to read the file one line at a time (the constructor throws a FileNotFoundException if the file does not exist), and the lines are stored as strings in an ArrayList. Lastly, the size of the ArrayList (i.e., the number of records) is set as the global variable "#R".
- **getRecord**: When MohawkMain needs to access a record in its entirety, it does so by calling this method and passes it the index of the record to be retrieved. The record is read from the ArrayList and is returned as a String.
- **getFields**: When MohawkMain needs to access the fields of a record, it does so by calling this method and passes it the index of the record to be retrieved. The record is read from the ArrayList is tokenized as a String (with whitespace separating the tokens), and a String array is returned.
- **numRecords**: If MohawkMain needs to know the number of records, it can call this method.

It should be noted that, unlike many of the operator classes in section 5.6, which implement static methods, the methods on MohawkFileLoader are non-static, in case a developer wants to extend MOHAWK and allow for the loading of multiple files. This component was developed by Chris and Lauren.

5.3 Front End

The classes listed here are all generated by ANTLR from the MohawkParser.g file. The methods are called by MohawkMain when the MOHAWK program is being analyzed lexically and syntactically. These components were developed by Ryan and Lauren.

5.3.1 MohawkLexer

The constructor is called by MohawkMain and is passed the name and location of the program file. The object is then passed to the MohawkParser, which does most of the "real" work; this class itself just represents a TokenStream.

5.3.2 MohawkParser

The constructor is called by MohawkMain and is passed the instance of MohawkLexer. The constructor then creates an instance of ASTFactory, which will be used to generate the actual AST.

5.3.3 MohawkAntlrTokenTypes

This is just an interface that defines a number of constants used in the MohawkParser.

5.4 Tree Walker

The classes listed here are generated by ANTLR from the MohawkWalker.g file. The methods are called by MohawkMain when the MOHAWK program is being executed. These components were developed by Chris, Ryan, and Lauren.

5.4.1 MohawkWalker

The principle function of the MohawkWalker is to take the AST created by the MohawkParser and walk through it, calling the appropriate methods of the back-end components as it goes along. As such, there are three main entry points into the MohawkWalker:

- **begin:** This method is called by MohawkMain when it wants to execute all the "begin" blocks of the program. It walks the tree, ignoring any statement that is not a "begin" block or a function definition, and then executes the code in the "begin" blocks or in registering a function.
- **end:** This method is called by MohawkMain when it wants to execute all the "end" blocks of the program. It walks the tree, executing the code in the "end" blocks and ignoring everything else.
- **program:** This method is called once for each record in the data input file. It ignores "begin" and "end" blocks and function definitions, and for each pattern-action statement it encounters, it evaluates the "pattern" expression and then executes the statements in the corresponding curly braces if that expression is true.

The MohawkWalker also relies heavily on the **expr** method, which is used to execute and evaluate all of the expressions that the AST represents. This method is called from the "begin", "end", and "program" methods. For each node in the AST, there is a corresponding entry in this method, and it calls the appropriate backend methods as necessary.

5.4.2 MohawkWalkerTokenTypes

This is just an interface that defines a number of constants used in the MohawkWalker.

5.5 Exceptions

The exceptions created for MOHAWK are for implementing some of the features that involve breaking the normal control flow of a loop or of processing the records from the data input file. All of the exceptions extend RuntimeException, so that they can be used within the ANTLR-generated code without any customization. These components were developed by Lauren.

5.5.1 MohawkBreak

Used for implementing the ":-(" (break) keyword. This exception is thrown when the keyword is encountered by MohawkWalker.expr, and it is caught by the code that implements the for, while, or do/while loop. When this exception is caught, the loop is broken. If this exception is thrown by code that is not in a loop, however, it is caught by MohawkMain and an error is reported.

5.5.2 MohawkContinue

Used for implementing the ":->" (continue) keyword. This exception is thrown when the keyword is encountered by MohawkWalker.expr, and it is caught by the code that implements the for, while, or do/while loop. When this exception is caught, the loop continues. If this exception is thrown by code that is not in a loop, however, it is caught by MohawkMain and an error is reported.

5.5.3 MohawkNext

Used for implementing the ":-|" (next) keyword. This exception is thrown when the keyword is encountered by MohawkWalker.expr, and it is caught by MohawkMain. It will then call "continue" in the

loop that is implementing MohawkWalker.begin (the "begin" blocks), MohawkWalker.program (i.e. running the pattern-action statements for all of the records from the data input file) or MohawkWalker.end (the "end" blocks). If this exception is thrown in a "begin" or "end" block, an error message is reported; otherwise, MohawkMain moves on to the next input record and continues.

5.5.4 MohawkExit

Used for implementing the "`:^P`" (exit) keyword. This exception is thrown when the keyword is encountered by MohawkWalker.expr, and it is caught by MohawkMain. If the exception is thrown in a "begin" or "end" block, then the program will terminate; if the exception's "exitStatus" property is 0, then System.exit(0) is called, otherwise System.exit(1) is called. If the exception is thrown while running the pattern-action statements for the records in the data input file, it will terminate the program if the exception's "exitStatus" property is not 0, but break the loop processing those input records otherwise.

5.6 Back End

These classes implement most of the keywords and operators in the MOHAWK programming language. These components were developed by Chris (except for MohawkSymbols, which was developed by Ryan).

5.6.1 MohawkDataType

All data elements – literals, constants, and variables – are represented by MohawkDataType objects. Because MOHAWK does not have distinct data types per se, an instance of MohawkDataType must be able to be used as an integer, a float, a string, or a Boolean. The real "value" of the MohawkDataType is stored in a String member variable called "strValue". To access that value in different ways, the following methods are used:

- **intValue**: calls Integer.parseInt(strValue) and returns an int if the value is an integer, and throws a NumberFormatException otherwise
- **floatValue**: calls Float.parseFloat(strValue) and returns an float if the value is float or an integer, and throws a NumberFormatException otherwise
- **booleanValue**: returns true if the value is the string "true" or the string is numeric (float or int) and is greater than zero; returns false otherwise

5.6.2 MohawkSymbols

This component is responsible for maintaining the symbol table, i.e. the mapping of symbol (variable) names to the MohawkDataType objects they represent. The component has a LinkedList of Hashtables for storing the local symbol tables; each Hashtable in the LinkedList represents a new scope. It also maintains a dedicated Hashtable for the global symbols, and yet another Hashtable for the "links", which map variable names to field variables read from the data input file. The component exposes the following interfaces as static methods:

- **push:** This method is called when a new scope is created, e.g. whenever a left curly brace is encountered or a function is called. It creates a new Hashtable at the head of the LinkedList.
- **pop:** This method is called when leaving a scope, e.g. when a right curly brace is encountered. It removes the Hashtable that is at the head of the LinkedList.
- **branch:** When a function is called, rather than creating a new Hashtable at the head of the LinkedList (like with the pop method), this method creates a new LinkedList and puts *that* at the head of the LinkedList. That way, the function can have its own levels of scope and handle the arguments that are passed to it.
- **endBranch:** This method is to be called when leaving a function. It removes the function-specific LinkedList at the head of the LinkedList of local symbols.
- **putGlobal:** This method is given a name and a MohawkDataType and adds the symbol to the global symbol table.

- **put:** Whenever a value is stored in a variable, this method will be called. It first looks through the local symbol tables (using the "getNoWarn" method) to see if the symbol already exists, then it checks the global symbol table. If the symbol does not exist anywhere in the existing symbol tables, it is created in the top level. The corresponding MohawkDayaType's string value is then set to whatever is being "put" in it.
- **get:** This method is given a symbol name and returns a MohawkDataType. It first looks through the local symbol tables to see if the symbol already exists, then it checks the global symbol table. If the symbol does not exist anywhere in the existing symbol tables, it is created in the top level, but a warning message is displayed indicating that the symbol did not previously exist. The corresponding MohawkDayaType is then returned.
- **getNoWarn:** This is the same as "get" but does not display a warning when the symbol does not exist in the symbol table. It is used primarily by the "put" method.
- **link**: MOHAWK provides programmers with the ability to link a name to a field variable, so that they can refer to the name rather than the less-meaningful "$2", for instance. This method takes two Strings – the link name and the field variable – and links them in a Hashtable.
- **exists:** Given a symbol name, this method simply returns true if the symbol currently exists anywhere in the symbol tables, and false otherwise.
- **cleanFieldVariables:** This method is used by MohawkMain to clean up all of the field variables for the current record. This method is called before loading the field variables from a new record, in order to ensure that any old values are not left around.

5.6.3 MohawkOperator

This is the base class for the operator classes. It simply implements an "error" method that implementing classes can use to write to System.err.

5.6.4 MohawkLogicalOperator

This class implements static methods for performing the following comparison and logical operations (see Section 3.5.4 for more information):

- **equals**: compares two MohawkDataTypes and returns a MohawkDataType with value "true" if they are numerically equal or if their string values are equal; returns a MohawkDataType with value "false" otherwise
- **notEquals**: compares two MohawkDataTypes and returns a MohawkDataType with value "true" if they are not numerically equal or if their string values are not equal; returns a MohawkDataType with value "false" otherwise
- **lessThan**: compares two MohawkDataTypes and returns a MohawkDataType with value "true" if the first argument is numerically less than the second or if their string value of the first is less than that of the second; returns a MohawkDataType with value "false" otherwise
- **greaterThan**: compares two MohawkDataTypes and returns a MohawkDataType with value "true" if the first argument is numerically greater than the second or if their string value of the first is greater than that of the second; returns a MohawkDataType with value "false" otherwise
- **lessThanOrEquals**: compares two MohawkDataTypes and returns a MohawkDataType with value "true" if the first argument is numerically less than or equal to the second or if their string value of the first is less than or equal to that of the second; returns a MohawkDataType with value "false" otherwise
- **greaterThanOrEquals**: compares two MohawkDataTypes and returns a MohawkDataType with value "true" if the first argument is numerically greater than or equal to the second or if their string value of the first is greater than or equal to that of the second; returns a MohawkDataType with value "false" otherwise
- **match**: compares two MohawkDataTypes and returns a MohawkDataType with value "true" if the first argument matches (according to the Java rules for matching) the regular expression represented by the second; returns a MohawkDataType with value "false" otherwise

- **or**: takes two MohawkDataTypes and returns returns a MohawkDataType with value "true" if either one represents a Boolean true (based on the MohawkDataType.booleanValue method) – this method does **not** use short-circuiting
- **and**: takes two MohawkDataTypes and returns a MohawkDataType with value "true" if both represent a Boolean true (based on the MohawkDataType.booleanValue method) – this method does **not** use short-circuiting
- **not**: takes a MohawkDataType and returns a MohawkDataType with a Boolean value that is the logical opposite of that of the argument

5.6.5 MohawkMathOperator

This class implements static methods for performing the following mathematical operations (see Section 3.5.1 for more information):

- **add**: takes two MohawkDataTypes and returns a MohawkDataType whose value is the sum of the numerical representations of the two arguments
- **subtract**: takes two MohawkDataTypes and returns a MohawkDataType whose value is the difference of the numerical representations of the two arguments
- **multiply**: takes two MohawkDataTypes and returns a MohawkDataType whose value is the product of the numerical representations of the two arguments
- **divide**: takes two MohawkDataTypes and returns a MohawkDataType whose value is the quotient of the numerical representations of the two arguments
- **mod**: takes two MohawkDataTypes and returns a MohawkDataType whose value is the modulus of the numerical representations of the two arguments
- **negate**: takes a MohawkDataType and returns a MohawkDataType whose value is the negative of the numerical value of the argument
- **exp**: takes two MohawkDataTypes and returns a MohawkDataType whose value is equal to the numerical representation of the first argument raised to the power of the numerical representation of the second

5.6.6 MohawkStringOperator

This class implements static methods for performing the following string operations (see Section 3.5.2 for more information):

- **concatenate**: takes two MohawkDataTypes and returns a MohawkDataType whose value is the concatenation of the string representations of the two arguments

5.6.7 MohawkFunctionHandler

This class implements static methods for registering and calling MOHAWK functions:

1. **register:** When a function is declared, this method is called with the name of the function, its parameters (a Vector) and the body (an AST) as arguments. A MohawkFunction object is created and the function is stored in a Hashtable with the name as the key. If a function with that name is already registered, a warning message is displayed.
2. **call**: When a function is called from a MOHAWK program, this method is called with a reference to the MohawkWalker, the name of the function, and the arguments (a Vector) as arguments. The MohawkFunction is retrieved from the Hashtable (if it doesn't exist, a warning message is shown) and the arguments are mapped to the parameter names (from the MohawkFunction object) in the symbol table. Then the body of the function is executed using the MohawkWalker.exprs method.

6 Test Plan

This section describes the different tests that were performed during the development of the MOHAWK programming language.

Testing MOHAWK consisted of four major components.

1. Unit Testing
2. Parser Testing
3. Program Testing
4. Regular Expression Test

6.1 Unit Testing

Unit testing the MOHAWK program was performed using JUNIT library to develop a test suite which makes adding new test cases and provides ease of regression testing in the future. Unit testing was broken up into various variable operation/manipulation testing to validate expected results. Each unit test is detailed as follows.

1. Math Operator Test

The mathematical operator test consisted of applying operators listed in sections 3.5.1 and 3.5.3.

Each math operator test was applied to a set of variables testing combinations of math operators on various data types. For example, starting with the general case of adding two integers together and working to possibilities of exponent arithmetic on an integer and a float. The results of these test cases have been validated with the expected results found in section 3.5.1.

Example

a = 1!
b = 2.0!

(a + b) should yield “3.0f”
(a * b) should yield “2.0f”
(a % b) should yield “1.0f”
(a ^ b) should yield “1”

2. Logical Operator Test

The logical operator test consisted of applying operators listed in section 3.5.4 and 3.5.5.

These tests include comparison of various data type variables with logical operators such as, = =, *!*=, >, <, >=, <=, ~, &&, | |.

Each test was conducted comparing values of variables with different types to assert validity in the results. Integer, float, and string variables were compared to itself and each other to compare expected results. The results expected from the logical operator test are the following.

Note:

- For each of the following, = =, !=, &&, | |, the identity comparison was set to the same value. Ex. int, int comparison for = = are the same values.
- ~ operator was tested with the first variable.
- **int1** = 2, **int2** = 6, **flt1** = 2.0f, **flt2** = 6.0f, **str1** = “cat”, **str2** = “dog”

	= =	!=	>	<	>=	<=	~	&&	||
int1, int2	True	False	False	True	False	True	False	True	True
int1, flt1	True	False	False	False	True	True	False	True	True
int1, str1	False	True	True	False	True	False	False	True	True
flt1, flt2	True	False	False	False	True	True	False	True	True
flt1, str1	False	True	True	False	True	False	False	True	True
str1, str2	True	False	False	False	True	True	False	True	True

To test the logical operators, several other in depth tests were done to test the Boolean operators. Since strings with explicit values of "true" and "false" are Booleans and non-zero numeric value also equals true, these conditions were tested with the *Negation*, *And*, and *Or* operators for correct execution. (Refer to Test Suite for details)

3. String Operator Test

The string operator test consisted of applying operators listed in section 3.5.2.

The string operators (equals, not equals, concatenate) were tested on a string variable with various other variable types such as integer, float and another string.

Example

```
a = 1!
b = 2.0!
c = cat!

/* concatenate */
(a . c) should yield "1cat"
(b . c) should yield "2.0cat"
(c . c) should yield "catcat"

/* equals */
(c = = c) should yield "true"
(a = = b) should yield "false"

/* not equals */
(a ! = b) should yield "true"
(c ! = c) should yield "false"
```

4. Regular Expression Test

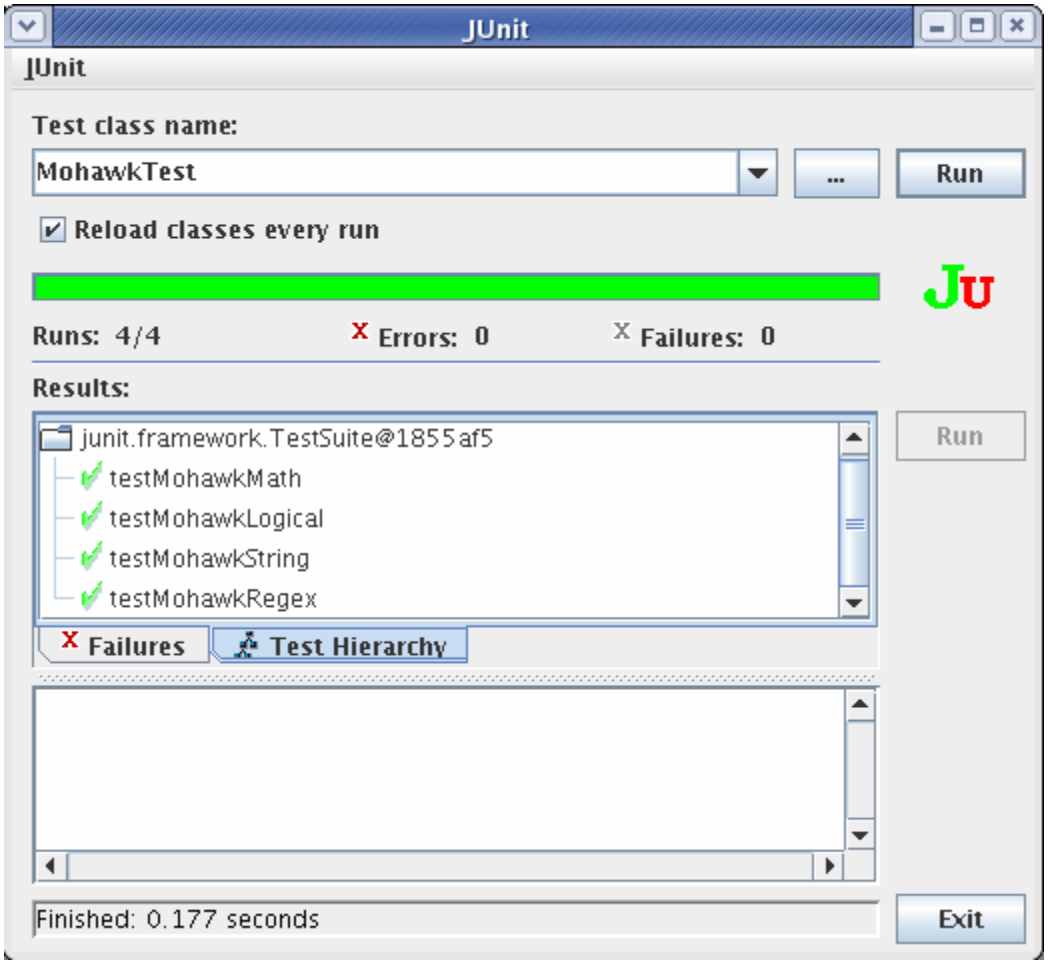
Regular expression testing was done by setting one string value to various regular expressions. The testing regarding regular expression was done a bit loosely since MOHAWK uses the regular expression API provided by the java library. Thus only couple of cases have been tested to verify the results, and any specific case is assumed to be handled by the API.

Similar to logical operator testing the following cases are examples of the RegEx testing.

OriginalString = "Chris and Becky are nice TAs"

RegEx:

"[Cc]hris and Becky are nice TAs" → ***TRUE***
"[a-z_0-9A-Z]hris and Becky are nice TAs" → ***TRUE***
"[^] [^]ris and Becky are nice TAs → ***TRUE***



6.2 Parser Testing

Testing the parser was done by breaking up the test scenarios into multiple statement groups. Language statements were grouped in the following manner.

- Statement
  - Mathematical Operators
  - Assignment Operators
- Conditional
  - Comparison Statements
  - Logical Operators
- Function
  - User created functions
- Control
  - Break
  - Next
  - Continue
  - Exit
- Loop
  - For
  - While
  - DoWhile

To test the parser, sample programs broken up into scenarios listed above, were used to generate an Abstract Syntax Tree. Using previously well tested sample cases, the expected outputs were compared with the actual tree produced by the parser. Although limited by its requirement of continuous changes required in the expected tree, it was useful to perform some regression testing through this method. This allowed us to test more effectively any modification made to the language itself in future testing.

```
Statement Test PASSED!
Conditional Test PASSED!
Loop Test PASSED!
Control Test PASSED!
Function Test PASSED!

Parser Test DONE!
[jk2438@athens TreeTest]$
```

6.3 Program Testing

Program testing was conducted to validate runtime output produced by MOHAWK. Although unit and parse test determine correctness in segments of the language, a general test of the user level output is required. Program testing takes in typical user case MOHAWK program used throughout semester and validates the expected output. This portion of the testing in essence determines the completeness in the language since it's what the user sees.

7 Lessons Learned

Christian Murphy

I imagine that many students use this opportunity to rue the fact that they didn't start development early enough, didn't have a good project plan, didn't have a well-defined software architecture, didn't do enough testing, didn't use source control, etc. However, we did all that and the project went extremely smoothly, so I would consider those to be "lessons reaffirmed" rather than "lessons learned". What I did learn in this project, though, is much more specific to language design than to running a project. I learned that many of the things we as programmers take for granted – like operator precedence, variable scoping, and datatype manipulation – require careful planning and extremely thorough testing in order to ensure that all possible cases are handled and that there is consistency across the different parts of the language. I also learned the importance of creating a syntax that allows developers using the language to balance the general concepts that they are comfortable with (like loops and conditionals) with tokens that convey and capture what is specific to the domain in which the language is best suited.

Joeng Kim

The practical application of the concepts learned in this course to create our own language was a great experience. I have learned the importance of testing on many levels to test correctness of a language throughout this project. The plan and structure of a solid test plan helped us to ensure that the new changes made to the language would not only work correctly in it, but also would not contain unexpected side effects. Although the full effect of the test suite was not applicable until the first iteration of our language features, the continuous changes and updates to the test suite has taught me how to better anticipate test scenarios for future development.

Ryan Overbeck

Current parsing / lexing theory extends seamlessly into practice making language implementation, using tools like ANTLR, easier than I would have thought. Overusing non-numeric and alpha characters in the language leads to slower programming. I learned that it is impossible to stick to your first definition of a language. There will always be parts in the original specification that interfere in unpredictable ways. Some major design decisions are often surprisingly ad-hoc. Sometimes just to keep the parser generator from complaining.

Lauren Wilcox

I learned a lot about the trade-offs that arise when designing a language to fit particular tasks. I learned how to prioritize to accommodate the specific tasks our language might perform and the flexibility that the programmer needs. For example, since the language is interpreted, we chose to use flexible types, since the tasks the programmer will perform are not worth the burden of considering strict rules for typing when they code.

I also learned that good language design incorporates many interesting considerations, from the user experience to the load on the CPU. My team considered several points between the human and the machine. We thought about syntax and the cognitive load requires in learning and using it to code in our language. For example, we explored whether our keyword changes and additions would be worth the learning time to use them. Some keywords might be harder to differentiate from others, and some "error prevention" can be introduced in the syntax and structure of the language by making symbols, logical blocks, and statements unambiguous and differentiable. We also explored how errors should be reported, and how warnings versus errors should be generated and handled in a way that is efficient based on the tasks and functions of the language.

We made decisions that we thought would save time during execution. For example, when comparing two of our Mohawk data types, the order in which we compare these corresponds to an analysis of the most

probable types we expect them to represent. We save complicated procedure calls for the time when they are absolutely necessary. We considered decisions like whether or not to short circuit, the worth of static semantics checking, how global types are handled and how scoping rules should work.

I learned that it paid off to be realistic about the time commitment involved and increase my opportunities for learning by choosing a language that wouldn't rely on extraneous libraries and features but instead be simple enough to allow the whole team to review and discuss fundamental language decisions and how to approach the code in a clever way rather than spending most of our time hooking up and relying on libraries that may or may not work as we expect them to.

Another thing I learned is that it was useful to participate in different aspects of the development, not just an assigned role. Familiarizing myself with the lexer/parser and helping with code generation helped me understand a larger view of the overall project and how decisions in one area affect design and functionality in another. Helping with documentation helped me to grasp the importance of the decisions we made and think more carefully about our reasons behind them. Testing the limits of our language helped me fix bugs in my code.

8 Appendix

8.1 Lexer/Parser (MohawkParser.g)

MohawkParser.g
head: 1.6
---
revision 1.6
date: 2005/12/13 19:09:54; author: rso2102; state: Exp; lines: +1 -1
fixed :-o() (with empty "()") so that it doesn't crash.
.
---
revision 1.5
date: 2005/12/12 23:46:32; author: rso2102; state: Exp; lines: +2 -58
Exit can have empty parens.
---
revision 1.4
date: 2005/12/12 23:19:59; author: rso2102; state: Exp; lines: +60 -12
Made break and continue into parse errors (when outside a loop)
---
revision 1.3
date: 2005/12/12 21:13:22; author: rso2102; state: Exp; lines: +10 -15
Fixed a few bugs.
---
revision 1.2
date: 2005/12/12 00:43:26; author: lgw23; state: Exp; lines: +1 -1
implemented exit with error status (LGW)
---
revision 1.1
date: 2005/12/10 20:49:07; author: cdm6; state: Exp;
recreated

```
/* Lexer for Mohawk
 */
class MohawkLexer extends Lexer;

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

protected ALPHA
options {
  paraphrase = "an alphabetic character or '_'";
}
  : 'a'..'z' | 'A'..'Z' | '_' ;
```

```
protected DIGIT
options {
  paraphrase = "a digit (0 - 9)";
}
  : '0' .. '9' ;

WS
options {
  paraphrase = "a whitespace";
}
  : (' ' | '\t' )
     { $setType(Token.SKIP); } ;

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

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

NAME
options {
  paraphrase = "an identifier";
  testLiterals = true;
}
    : ALPHA (ALPHA|DIGIT)* ;

NUMBER
options {
  paraphrase = "a number";
}
  : (DIGIT)+ ('.' (DIGIT)*)?
          (('E'|'e') ('+'|'-')? (DIGIT)+)? ;

STRING
options {
  paraphrase = "a string";
}
  : '"'!
         ( ~('"' | '\n') | ('"'! '"') )*
         '"'!;
/*
REGEXP : '['!
    ( ~( '[' | ']' | '\n' ) | (( '['! '[' ) | ( ']'! ']' )) )*
         ']'!;
*/
```

```
LCURLY
options {
  paraphrase = "'{'";
}
  : '{' ;
RCURLY
options {
  paraphrase = "'}'";
}
  : '}' ;
BANG
options {
  paraphrase = "'!'";
}
  : '!' ;
LPAREN
options {
  paraphrase = "'('";
}
  : '(' ;
RPAREN
options {
  paraphrase = "')'";
}
  : ')' ;

ASSIGN
options {
  paraphrase = "'='";
}
  : '=' ;
INCR
options {
  paraphrase = "'++'";
}
  : "++" ;
DECR
options {
  paraphrase = "'--'";
}
  : "--" ;
PLUSEQ
options {
  paraphrase = "'+='";
}
  : "+=" ;
MINUSEQ
options {
  paraphrase = "'-='";
}
  : "-=" ;
DIVEQ
options {
  paraphrase = "'/='";
}
  : "/=" ;
MULTEQ
```

```
options {
  paraphrase = "'*='";
}
  : "*=" ;
LINK
options {
  paraphrase = "'->'";
}
  : "->" ;


EQ
options {
  paraphrase = "'=='";
}
  : "==";
NEQ
options {
  paraphrase = "'!='";
}
  : "!=" ;
GE
options {
  paraphrase = "'>='";
}
  : ">=" ;
LE
options {
  paraphrase = "'<='";
}
  : "<=" ;
GT
options {
  paraphrase = "'>'";
}
  : ">" ;
LT
options {
  paraphrase = "'<'";
}
  : "<" ;
OR
options {
  paraphrase = "'||'";
}
  : "||";
AND
options {
  paraphrase = "'&&'";
}
  : "&&";
NOT
options {
  paraphrase = "'~'";
}
  : "~";
```

```
PLUS
options {
  paraphrase = "'+'";
}
  : '+' ;
MINUS
options {
  paraphrase = "'-'";
}
  : '-' ;
MULT
options {
  paraphrase = "'*'";
}
  : '*' ;
DIV
options {
  paraphrase = "'/'";
}
  : '/' ;
MOD
options {
  paraphrase = "'%'";
}
  : '%' ;
EXP
options {
  paraphrase = "'^'";
}
  : '^' ;

// PRINT : ":-O" ;
// BREAK : ":-(" ;

STRCAT
options {
  paraphrase = "'.'";
}
  : ".";
REGEQ
options {
  paraphrase = "'~='";
}
  : "~=";

COMMA
options {
  paraphrase = "','";
}
  : ',' ;
COLON
options {
  paraphrase = "':'";
}
  : ':' ;
SEMI
options {
```

```
  paraphrase = "';'";
}
  : ';' ;
BACKSLASH
options {
  paraphrase = "'\\'";
}
  : '\\' ;
SMILEY options { testLiterals = true; }
       : ( ';' | ':' | '|' | '>' ) /* possible eyes */
         ( '-' | '^' ) /* noses */
         ( ')' | '(' | 'O' | 'o' | 'D' | 'P' | '>' | '<' | '|' | '/' |
'~' | '\\' | ('/' '\\')) ; /* mouths */

BEGIN
options {
  paraphrase = "'-:-|'";
}
  : "-:-|" ;
END
options {
  paraphrase = "'|-:-'";
}
  : "|-:-" ;

FIELD_VAR
options {
  paraphrase = "'$'";
}
  : '$' ( DIGIT )+ ;

GLOBAL_VAR
options {
  paraphrase = "'#'";
}
  : '#' (ALPHA) ;


/* Parser for Mohawk.
 * Notes:
 *   Will probably want to separate string expressions from numeric
 *   expressions.  For now, they are all together.
 */
{ /* define parser exceptions */
  class MohawkParserException extends Exception {
    MohawkParserException() { super(); }
    MohawkParserException( String message ) { super(message); }
  }
}
class MohawkParser extends Parser;

options {
  k = 3;
  buildAST = true;
  ASTLabelType = "CommonASTWithLines";
  exportVocab = MohawkAntlr;
  defaultErrorHandler=false;
```

```
}

tokens {
   STMT;
   PRINT_CALL;
   ACTION;
   BEGIN_ACTION;
   IF_STMT;
   VAR_LIST;
   EXPR_LIST;
   FUNC_CALL;
   BREAK_STMT;
   NEGATE;
}

{
  int numErrors=0;
  int numWarnings=0;
  protected   void   MyDefaultHandler(   ANTLRException   e   )   throws
MohawkParserException {
    numErrors++;
    if ( numErrors > 100 ) {
      MohawkParserException  ex  =  new  MohawkParserException("Too  many
errors.  Aborting.");
      throw ex;
    }
  }
}

program throws MohawkParserException
  : ( rule | function_definition )*  EOF!
        ;
exception
catch [NoViableAltException e] {
  System.err.println("Oi! Oi! Oi!: "+e);
  consume();
  MyDefaultHandler( e );
} catch [ANTLRException e] {
  System.err.println("Oi! Oi! Oi!: "+e);
  MyDefaultHandler( e );
}

rule throws MohawkParserException
  : pattern_action_stmt
     ;
exception
catch [NoViableAltException e] {
  System.err.println("Oi! Oi! Oi!: "+e);
  consume();
  MyDefaultHandler( e );
} catch [ANTLRException e] {
  System.err.println("Oi! Oi! Oi!: "+e);
  MyDefaultHandler( e );
}

pattern_action_stmt throws MohawkParserException
  :
```

```
          ( begin_pattern ( LCURLY begin_action RCURLY )) |
            ( pattern ( BANG! | ( LCURLY action RCURLY ))) |
            ( LCURLY action RCURLY )
         ;
exception
catch [NoViableAltException e] {
  System.err.println("Oi! Oi! Oi!: "+e);
  consume();
  MyDefaultHandler( e );
} catch [ANTLRException e] {
  System.err.println("Oi! Oi! Oi!: "+e);
  MyDefaultHandler( e );
}

action throws MohawkParserException
  : ( stmt )*
        { #action = #([ACTION, "ACTION"], action); }
       ;
exception
catch [NoViableAltException e] {
  System.err.println("Oi! Oi! Oi!: "+e);
  consume();
  MyDefaultHandler( e );
} catch [ANTLRException e] {
  System.err.println("Oi! Oi! Oi!: "+e);
  MyDefaultHandler( e );
}

begin_pattern throws MohawkParserException
  : BEGIN^
     ;
exception
catch [ANTLRException e] {
  System.err.println("Oi! Oi! Oi!: "+e);
  MyDefaultHandler( e );
}

begin_action throws MohawkParserException
  : ( stmt )*
        {    #begin_action    =    #([BEGIN_ACTION,    "BEGIN_ACTION"],
begin_action); }
        ;
exception
catch [NoViableAltException e] {
  System.err.println("Oi! Oi! Oi!: "+e);
  consume();
  MyDefaultHandler( e );
} catch [ANTLRException e] {
  System.err.println("Oi! Oi! Oi!: "+e);
  MyDefaultHandler( e );
}

pattern throws MohawkParserException
  : ( END^ | expr )
        ;
exception
catch [NoViableAltException e] {
```

```
  System.err.println("Oi! Oi! Oi!: "+e);
  consume();
  MyDefaultHandler( e );
} catch [ANTLRException e] {
  System.err.println("Oi! Oi! Oi!: "+e);
  MyDefaultHandler( e );
}


/* Expressions */
/* Ryan, need to add range patterns */
expr throws MohawkParserException
  : logic_term ( OR^ logic_term )*
    ;
exception
catch [NoViableAltException e] {
  System.err.println("Oi! Oi! Oi!: "+e);
  consume();
  MyDefaultHandler( e );
} catch [ANTLRException e] {
  System.err.println("Oi! Oi! Oi!: "+e);
  MyDefaultHandler( e );
}

logic_term throws MohawkParserException
  : logic_factor ( AND^ logic_factor )*
    ;
exception
catch [NoViableAltException e] {
  System.err.println("Oi! Oi! Oi!: "+e);
  consume();
  MyDefaultHandler( e );
} catch [ANTLRException e] {
  System.err.println("Oi! Oi! Oi!: "+e);
  MyDefaultHandler( e );
}

logic_factor throws MohawkParserException
  : ( NOT^ )? rel_expr
    ;
exception
catch [NoViableAltException e] {
  System.err.println("Oi! Oi! Oi!: "+e);
  consume();
  MyDefaultHandler( e );
} catch [ANTLRException e] {
  System.err.println("Oi! Oi! Oi!: "+e);
  MyDefaultHandler( e );
}

neg_expr throws MohawkParserException
  : MINUS! arith_factor { #neg_expr = #([NEGATE,"NEGATE"], neg_expr);}
;
//create new node NEGATE of type NEGATE
//take everything from neg_expr and plant it as children of this node
exception
catch [NoViableAltException e] {
```

```
  System.err.println("Oi! Oi! Oi!: "+e);
  consume();
  MyDefaultHandler( e );
} catch [ANTLRException e] {
  System.err.println("Oi! Oi! Oi!: "+e);
  MyDefaultHandler( e );
}


rel_expr throws MohawkParserException
  : arith_expr
        ((  GE^  |  LE^  |  GT^  |  LT^  |  EQ^  |  NEQ^  |  streq_word  |
strne_word | REGEQ^ ) arith_expr )?
      ;
exception
catch [NoViableAltException e] {
  System.err.println("Oi! Oi! Oi!: "+e+". Perhaps missing a '!'?" );
 /* consume(); */
  MyDefaultHandler( e );
} catch [ANTLRException e] {
  System.err.println("Oi! Oi! Oi!: "+e);
  MyDefaultHandler( e );
}

arith_expr throws MohawkParserException
  : arith_term
        (( PLUS^ | MINUS^ | STRCAT^ ) arith_term )*
      ;
exception
catch [NoViableAltException e] {
  System.err.println("Oi! Oi! Oi!: "+e);
  consume();
  MyDefaultHandler( e );
} catch [ANTLRException e] {
  System.err.println("Oi! Oi! Oi!: "+e);
  MyDefaultHandler( e );
}

arith_term throws MohawkParserException
  : arith_factor
        (( MULT^ | MOD^ | DIV^ | EXP^ ) arith_factor )*
      ;
exception
catch [NoViableAltException e] {
  System.err.println("Oi! Oi! Oi!: "+e);
  consume();
  MyDefaultHandler( e );
} catch [ANTLRException e] {
  System.err.println("Oi! Oi! Oi!: "+e);
  MyDefaultHandler( e );
}

arith_factor throws MohawkParserException
  : r_value | l_value (INCR^ | DECR^) | neg_expr
      ;
exception
catch [NoViableAltException e] {
```

```
  System.err.println("Oi! Oi! Oi!: "+e);
  consume();
  MyDefaultHandler( e );
} catch [ANTLRException e] {
  System.err.println("Oi! Oi! Oi!: "+e);
  MyDefaultHandler( e );
}

r_value throws MohawkParserException
  : l_value | NUMBER | STRING | "true" | "false" | ( function_call ) |
GLOBAL_VAR |
         ( LPAREN! expr RPAREN! )
      ;
exception
catch [NoViableAltException e] {
  System.err.println("Oi! Oi! Oi!: "+e);
  consume();
  MyDefaultHandler( e );
} catch [ANTLRException e] {
  System.err.println("Oi! Oi! Oi!: "+e);
  MyDefaultHandler( e );
}

l_value throws MohawkParserException
  : NAME | FIELD_VAR
      ;
exception
catch [NoViableAltException e] {
  System.err.println("Oi! Oi! Oi!: "+e);
  consume();
  MyDefaultHandler( e );
} catch [ANTLRException e] {
  System.err.println("Oi! Oi! Oi!: "+e);
  MyDefaultHandler( e );
}

/* Statements */
stmt throws MohawkParserException
  : bang_stmt |
    do_stmt |
    for_stmt |
    if_stmt |
    while_stmt |
    (( LCURLY! ( stmt )* RCURLY! )
       { #stmt = #([STMT, "STMT"], stmt); } ) |
    (":-(" BANG!
       {
         RecognitionException ex = new RecognitionException(": ':-(' is
not  allowed  outside  of  a  loop.",  null,  LT(-1).getLine(),  LT(-
1).getColumn());
         if ( ex != null ) throw ex;
       }
     ) |
     (":->"  BANG!
       {
```

```
         RecognitionException ex = new RecognitionException(": ':->' is
not  allowed  outside  of  a  loop.",  null,  LT(-1).getLine(),  LT(-
1).getColumn());
         if ( ex != null ) throw ex;
       }
     )
    ;
exception
catch [NoViableAltException e] {
  System.err.println("Oi! Oi! Oi!: "+e);
  consume();
  MyDefaultHandler( e );
} catch [ANTLRException e] {
  System.err.println("Oi! Oi! Oi!: "+e);
  MyDefaultHandler( e );
}

/* Same as stmt, but also allows break and continue */
loop_stmt throws MohawkParserException
  : ( break_stmt | continue_stmt ) BANG! |
    bang_stmt |
    do_stmt |
    for_stmt |
    loop_if_stmt |
    while_stmt |
    ( LCURLY! ( loop_stmt )* RCURLY! )
      { #loop_stmt = #([STMT, "STMT"], loop_stmt); }
;
exception
catch [NoViableAltException e] {
  System.err.println("Oi! Oi! Oi!: "+e);
  consume();
  MyDefaultHandler( e );
} catch [ANTLRException e] {
  System.err.println("Oi! Oi! Oi!: "+e);
  MyDefaultHandler( e );
}


/* Statements which need to be followed by a BANG */
bang_stmt throws MohawkParserException
  : (
      exit_stmt |
      function_call |
      asgn_stmt |
      print_function_call |
      next_stmt |
      getline_stmt )
      BANG!
;
exception
catch [NoViableAltException e] {
  System.err.println("Oi! Oi! Oi!: "+e);
  consume();
  MyDefaultHandler( e );
} catch [ANTLRException e] {
  System.err.println("Oi! Oi! Oi!: "+e);
```

```
  MyDefaultHandler( e );
}


break_stmt throws MohawkParserException
  : ":-("^
      ;
exception
catch [ANTLRException e] {
  System.err.println("Oi! Oi! Oi!: "+e);
  MyDefaultHandler( e );
}

continue_stmt throws MohawkParserException
  : ":->"^
      ;
exception
catch [ANTLRException e] {
  System.err.println("Oi! Oi! Oi!: "+e);
  MyDefaultHandler( e );
}

exit_stmt throws MohawkParserException
  : ":^P"^ ( (expr)? | ( LPAREN! RPAREN! ))
      ;
exception
catch [NoViableAltException e] {
  System.err.println("Oi! Oi! Oi!: "+e);
  consume();
  MyDefaultHandler( e );
} catch [ANTLRException e] {
  System.err.println("Oi! Oi! Oi!: "+e);
  MyDefaultHandler( e );
}

next_stmt throws MohawkParserException
  : ":-|"^
        ;
exception
catch [ANTLRException e] {
  System.err.println("Oi! Oi! Oi!: "+e);
  MyDefaultHandler( e );
}

getline_stmt throws MohawkParserException
  : ":-<"^
        ;
exception
catch [ANTLRException e] {
  System.err.println("Oi! Oi! Oi!: "+e);
  MyDefaultHandler( e );
}

for_stmt throws MohawkParserException
  : ":^O"^ for_range ( loop_stmt | BANG!)
        ;
exception
```

```
catch [NoViableAltException e] {
  System.err.println("Oi! Oi! Oi!: "+e);
  consume();
  MyDefaultHandler( e );
} catch [ANTLRException e] {
  System.err.println("Oi! Oi! Oi!: "+e);
  MyDefaultHandler( e );
}

for_range throws MohawkParserException
  : LPAREN
            ( (asgn_stmt)? SEMI expr SEMI (asgn_stmt)? )
            RPAREN
        ;
exception
catch [NoViableAltException e] {
  System.err.println("Oi! Oi! Oi!: "+e);
  consume();
  MyDefaultHandler( e );
} catch [ANTLRException e] {
  System.err.println("Oi! Oi! Oi!: "+e);
  MyDefaultHandler( e );
}

if_stmt throws MohawkParserException
  : ":-/"^ LPAREN! expr RPAREN! ( stmt | BANG! )
          ( options {greedy=true;} : ":-\\" ( stmt | BANG! ))?
        ;
exception
catch [NoViableAltException e] {
  System.err.println("Oi! Oi! Oi!: "+e);
  consume();
  MyDefaultHandler( e );
} catch [ANTLRException e] {
  System.err.println("Oi! Oi! Oi!: "+e);
  MyDefaultHandler( e );
}


loop_if_stmt throws MohawkParserException
  : ":-/"^ LPAREN! expr RPAREN! ( loop_stmt | BANG! )
          ( options {greedy=true;} : ":-\\" ( loop_stmt | BANG! ))?
        ;
exception
catch [NoViableAltException e] {
  System.err.println("Oi! Oi! Oi!: "+e);
  consume();
  MyDefaultHandler( e );
} catch [ANTLRException e] {
  System.err.println("Oi! Oi! Oi!: "+e);
  MyDefaultHandler( e );
}

do_stmt throws MohawkParserException
  : ">^O"^ loop_stmt ":^~"! LPAREN! expr RPAREN!
        ;
exception
```

```
catch [NoViableAltException e] {
  System.err.println("Oi! Oi! Oi!: "+e);
  consume();
  MyDefaultHandler( e );
} catch [ANTLRException e] {
  System.err.println("Oi! Oi! Oi!: "+e);
  MyDefaultHandler( e );
}

while_stmt throws MohawkParserException
  : ":^~"^ LPAREN! expr RPAREN! ( loop_stmt | BANG! )
      ;
exception
catch [NoViableAltException e] {
  System.err.println("Oi! Oi! Oi!: "+e);
  consume();
  MyDefaultHandler( e );
} catch [ANTLRException e] {
  System.err.println("Oi! Oi! Oi!: "+e);
  MyDefaultHandler( e );
}

function_call throws MohawkParserException
  : NAME LPAREN! expr_list RPAREN!
        { #function_call = #([FUNC_CALL,"FUNC_CALL"], function_call);
}
        ;
exception
catch [NoViableAltException e] {
  System.err.println("Oi! Oi! Oi!: "+e);
  consume();
  MyDefaultHandler( e );
} catch [ANTLRException e] {
  System.err.println("Oi! Oi! Oi!: "+e);
  MyDefaultHandler( e );
}

print_function_call throws MohawkParserException
  :
               ":-O"^ LPAREN! (expr)? RPAREN! | ":-o"^ LPAREN! (expr)?
RPAREN!
           {   /*#print_function_call   =   #([PRINT_CALL,"PRINT_CALL"],
print_function_call); */}
        ;
exception
catch [NoViableAltException e] {
  System.err.println("Oi! Oi! Oi!: "+e);
  consume();
  MyDefaultHandler( e );
} catch [ANTLRException e] {
  System.err.println("Oi! Oi! Oi!: "+e);
  MyDefaultHandler( e );
}


asgn_stmt throws MohawkParserException
  : l_value
```

```
    ((( ASSIGN^ | PLUSEQ^ | MINUSEQ^ | MULTEQ^ | DIVEQ^ | MODEQ^ |
LINK^ ) expr ) |
        INCR^ | DECR^)
        ;
exception
catch [NoViableAltException e] {
  System.err.println("Oi! Oi! Oi!: "+e);
  consume();
  MyDefaultHandler( e );
} catch [ANTLRException e] {
  System.err.println("Oi! Oi! Oi!: "+e);
  MyDefaultHandler( e );
}

function_definition throws MohawkParserException
  : ">-)"^ NAME LPAREN! var_list RPAREN!
            LCURLY! ( stmt )* RCURLY!
        {      /*#function_definition      =      #([FUNCTION_DEFINITION,
"FUNCTION_DEFINITION"], function_definition); */}
        ;
exception
catch [NoViableAltException e] {
  System.err.println("Oi! Oi! Oi!: "+e);
  consume();
  MyDefaultHandler( e );
} catch [ANTLRException e] {
  System.err.println("Oi! Oi! Oi!: "+e);
  MyDefaultHandler( e );
}

var_list throws MohawkParserException
  :    (l_value  (  COMMA!  l_value  )*)    {  #var_list  =  #([VAR_LIST,
"VAR_LIST"], var_list); }
            | { #var_list = #([VAR_LIST, "VAR_LIST"], var_list); }
        ;
exception
catch [NoViableAltException e] {
  System.err.println("Oi! Oi! Oi!: "+e);
  consume();
  MyDefaultHandler( e );
} catch [ANTLRException e] {
  System.err.println("Oi! Oi! Oi!: "+e);
  MyDefaultHandler( e );
}

expr_list throws MohawkParserException
  :  expr  (  COMMA!  expr  )*  {#expr_list  =  #([EXPR_LIST,"EXPR_LIST"],
expr_list); }
          |   {#expr_list = #([EXPR_LIST,"EXPR_LIST"], expr_list); }
          ;
exception
catch [NoViableAltException e] {
  System.err.println("Oi! Oi! Oi!: "+e);
  consume();
  MyDefaultHandler( e );
} catch [ANTLRException e] {
  System.err.println("Oi! Oi! Oi!: "+e);
```

```
  MyDefaultHandler( e );
}

/* 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 : "function" ;
print_word : ":-O" ;
nr_word : "#R" ;
nf_word : "#F" ;
streq_word : "eq" ;
strne_word : "ne" ;
getline_word : ":-<" ;
```

8.2 Tree Walker (MohawkWalker.g)

Working file: MohawkWalker.g
head: 1.32
---
revision 1.32
date: 2005/12/13 19:09:54; author: rso2102; state: Exp; lines: +4 -1
fixed :-o() (with empty "()") so that it doesn't crash.
---
revision 1.31
date: 2005/12/13 18:39:08; author: rso2102; state: Exp; lines: +7 -0
added a push/pop for the symbol table in for, while, and do loops.
---
revision 1.30
date: 2005/12/12 21:13:22; author: rso2102; state: Exp; lines: +20 -16
Fixed a few bugs.
---
revision 1.29
date: 2005/12/12 00:43:26; author: lgw23; state: Exp; lines: +5 -6
implemented exit with error status (LGW)
---
revision 1.28
date: 2005/12/11 20:25:37; author: lgw23; state: Exp; lines: +22 -8
Added some run-time error handling, mainly with respect to break, continue, exit, next in places other than where we would expect. Some of it is still up for discussion.
---
revision 1.27
date: 2005/12/10 20:19:31; author: cdm6; state: Exp; lines: +2 -1
Added the :-o symbol for print and changed :-O back to println
---
revision 1.26
date: 2005/12/10 20:12:54; author: cdm6; state: Exp; lines: +1 -1
changed :-O from println to print
---

revision 1.25
date: 2005/12/02 18:05:14; author: lgw23; state: Exp; lines: +1 -0
changed (-:- to |-:-
and -:-) to -:-|
oi oi oi negation works now

changed test files to reflect new begin/end

added a one-liner called mohawk to run mohawk
---
revision 1.24
date: 2005/12/02 16:44:48; author: rso2102; state: Exp; lines: +26 -4
no message
---
revision 1.23
date: 2005/12/02 02:33:28; author: lgw23; state: Exp; lines: +1 -5
*** empty log message ***
---
revision 1.22
date: 2005/12/01 21:06:44; author: lgw23; state: Exp; lines: +1 -0
took out digit negation since tokenizing neg number as it's own token leads to problems. Implemented negation on variables, it words. Negation takes precedence.
ie: -a*5

gives (*(-a) 5)

will fix digit negation next

got variable negation rto
---
revision 1.21
date: 2005/11/29 02:26:30; author: lgw23; state: Exp; lines: +8 -3
We can now use negetive numbers, but unary negation on a variable doesn't
work yet. So b = -a for any int or float a doesn't work yet -:-(

---
revision 1.20
date: 2005/11/28 18:17:45; author: lgw23; state: Exp; lines: +22 -27
*** empty log message ***
---
revision 1.19
date: 2005/11/28 05:50:06; author: lgw23; state: Exp; lines: +62 -48

that first continue implementation was buggy, this is better. LGW
---
revision 1.18
date: 2005/11/28 01:05:13; author: lgw23; state: Exp; lines: +50 -33

we now have continue and break, might need to add them to a few more types of
loops. It's not elegant, I'll have to look at how to do it better. -L
---
revision 1.17
date: 2005/11/22 15:19:10; author: rso2102; state: Exp; lines: +10 -2
Fixed Symbol tables to correctly branch and push/pop.
Top level of Begin actions is now global scope.
Need to decide what to do with variables declared inside the if, for, etc. statements,
I think we should push/pop twice, once for when if,.. is called second when a block statement is found
.

---
revision 1.16
date: 2005/11/20 22:00:10; author: lgw23; state: Exp; lines: +37 -20
*** empty log message ***
---
revision 1.15
date: 2005/11/17 23:47:19; author: cdm6; state: Exp; lines: +1 -1
made while -> :^~
.
---
revision 1.14
date: 2005/11/17 22:25:15; author: cdm6; state: Exp; lines: +2 -1
I guess I misunderstood the difference between NEXT and GETLINE.... so now GETLINE works but NEXT doesn't
---
revision 1.13
date: 2005/11/17 22:01:03; author: cdm6; state: Exp; lines: +1 -1
Modifications needed for implementing "NEXT"
---
revision 1.12
date: 2005/11/12 17:09:16; author: rso2102; state: Exp; lines: +208 -195
Made statements and functions have their own levels of scoping
---
revision 1.11
date: 2005/11/04 03:54:24; author: cdm6; state: Exp; lines: +1 -0
Started playing around with getting unary negation to work... but didn't succeed.
---
revision 1.10
date: 2005/11/01 22:02:55; author: cdm6; state: Exp; lines: +1 -1
Changed function definition keyword to >-)
---
revision 1.9
date: 2005/11/01 18:30:51; author: cdm6; state: Exp; lines: +1 -1
Added code to handle linking... not quite tested, though
---
revision 1.8
date: 2005/11/01 18:16:59; author: cdm6; state: Exp; lines: +1 -0
Added syntax for LINK operator.
---
revision 1.7
date: 2005/10/30 19:04:58; author: cdm6; state: Exp; lines: +9 -9
Changes to get smiley faces working as keywords
---
revision 1.6
date: 2005/10/29 21:47:02; author: cdm6; state: Exp; lines: +39 -2
MohawkFunctionHandler added for registering and executing functions

Everything else modified for doing function calls
---
revision 1.5
date: 2005/10/26 03:05:09; author: cdm6; state: Exp; lines: +2 -2
Fixed a small bug with booleans in MohawkWalker.g
Tried to implement ability to map names to field variables, but could have some issues
---
revision 1.4
date: 2005/10/25 20:15:16; author: cdm6; state: Exp; lines: +4 -1
Added handlers for true & false in MohawkWalker
Changed global vars to start with a # sign and made appropriate changes

---
revision 1.3
date: 2005/10/24 03:02:29; author: cdm6; state: Exp; lines: +83 -12
Lots of updates to the parser and walker grammars.
Also modified MohawkMain to go through the data file, using MohawkFileLoader
---
revision 1.2
date: 2005/10/23 17:41:35; author: cdm6; state: Exp; lines: +25 -8
Modified the grammar a little bit to get the print function working.
Added a bunch of stuff to MohawkDataType to make it more robust
Created Operator classes to have static methods for performing operations
Created Symbol table class
---
revision 1.1
date: 2005/10/22 17:31:29; author: cdm6; state: Exp;
Just playing around, mostly...
MohawkDataType is for storing data, but doesn't do much.
MohawkWalker is definitely work in progress... lots of issues to resolve
Made some modifications to MohawkParser for the tree walker
Added debugging to MohawkMain

```
/**************************************
 * Mohawk Tree Walker
**************************************/

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

}

class MohawkWalker extends TreeParser;
options{
   k=3;
   importVocab = MohawkAntlr;
}

{
   // any code initializations go here

}

begin
{
   MohawkDataType f;
}
: (
    (  (~(">-)"|LCURLY|BEGIN))?  LCURLY  morestuff:.  RCURLY)  |  /*  ignore
program */
    (func_def) | /* handle function definitions */
    (BEGIN LCURLY f=expr RCURLY) /* handle begins */
  )*
;
```

```
/** BUG: The "pattern" gets evaluated even though we don't want it to
**/
end
{
   MohawkDataType f;
}
: (
    (  (~(">-)"|LCURLY|END))?  LCURLY  morestuff:.  RCURLY)  |  /*  ignore
program */
    (null_func_def) | /* ignore function defs */
    (END LCURLY f=expr RCURLY) /* handle end */ )*
;

program
{
   MohawkDataType f;
}
: (
    ((BEGIN|END) LCURLY stuff:. RCURLY) |  /* ignore begin/end */
    (f=pattern  LCURLY  temp:.  RCURLY      /*  handle  pattern  action
statements */
      { if (f == null || f.booleanValue()) exprs(#temp); }
    ) |
    (null_func_def) /* ignore function defs */
    /* nothing */
  )*
;

null_func_def
: #(">-)" NAME null_param_list (body:.)?)
;

null_param_list
: #(VAR_LIST ( NAME )* )
;

func_def
{
   MohawkDataType f;
   Vector params;
}
:     #(">-)"     name:NAME     params=param_list     (body:.)?)     {
MohawkFunctionHandler.register(name.getText(), params, #body); }
;

param_list returns [Vector params]
{
   params = new java.util.Vector();
}
: #(VAR_LIST ( s:NAME { params.add(s.getText()); } )* )
;


expr_list returns [Vector params]
{
   params = new java.util.Vector();
   MohawkDataType r = new MohawkDataType();
```

```
}
: #(EXPR_LIST ( r=expr { params.add(r); } )* )
;

pattern returns [MohawkDataType r]
{
   // declare and initialize variables
   MohawkDataType a, b;
   r = null;
}
: r=expr
| /* nothing */
;
//
//any expr returns a mohawk data type r
expr returns [MohawkDataType r]
{
   // declare and initialize variables
   MohawkDataType a, b;
   MohawkDataType s = new MohawkDataType(0);
   r = new MohawkDataType();
   String id;
   Vector params;
   // System.out.println("expr" + _t);

 }
: #(ASSIGN id=identifier b=expr) { r = MohawkSymbols.put(id, b); }
|   #(PLUSEQ   id=identifier   b=expr)   {   r   =   MohawkSymbols.put(id,
MohawkMathOperator.add(MohawkSymbols.get(id), b)); }
|   #(MINUSEQ   id=identifier   b=expr)   {   r   =   MohawkSymbols.put(id,
MohawkMathOperator.subtract(MohawkSymbols.get(id), b)); }
|   #(MULTEQ   id=identifier   b=expr)   {   r   =   MohawkSymbols.put(id,
MohawkMathOperator.multiply(MohawkSymbols.get(id), b)); }
|   #(DIVEQ   id=identifier   b=expr)   {   r   =   MohawkSymbols.put(id,
MohawkMathOperator.divide(MohawkSymbols.get(id), b)); }
|          #(LINK          lname:NAME          lfv:FIELD_VAR)          {
MohawkSymbols.link(lname.getText(), lfv.getText()); }
|     #(INCR     id=identifier)     {     r     =     MohawkSymbols.put(id,
MohawkMathOperator.add(MohawkSymbols.get(id),  new  MohawkDataType(1)));
}
|     #(DECR     id=identifier)     {     r     =     MohawkSymbols.put(id,
MohawkMathOperator.subtract(MohawkSymbols.get(id),                    new
MohawkDataType(1))); }
| #(PLUS a=expr b=expr) { r = MohawkMathOperator.add(a, b); }
| #(MINUS a=expr b=expr) { r = MohawkMathOperator.subtract(a, b); }
| #(MULT a=expr b=expr) { r = MohawkMathOperator.multiply(a, b); }
| #(DIV a=expr b=expr) { r = MohawkMathOperator.divide(a, b); }
| #(MOD a=expr b=expr) { r = MohawkMathOperator.mod(a, b); }
| #(EXP a=expr b=expr) { r = MohawkMathOperator.exp(a, b); }
| #(NEGATE a=expr) { r = MohawkMathOperator.negate(a); }
| #(STRCAT a=expr b=expr) { r = MohawkStringOperator.concatenate(a, b);
}
| #(EQ a=expr b=expr) { r = MohawkLogicalOperator.equals(a, b); }
| #(NEQ a=expr b=expr) { r = MohawkLogicalOperator.notEquals(a, b); }
| #(LT a=expr b=expr) { r = MohawkLogicalOperator.lessThan(a, b); }
| #(GT a=expr b=expr) { r = MohawkLogicalOperator.greaterThan(a, b); }
```

```
| #(LE a=expr b=expr) { r = MohawkLogicalOperator.lessThanOrEquals(a,
b); }
|        #(GE        a=expr        b=expr)        {        r        =
MohawkLogicalOperator.greaterThanOrEquals(a, b); }
| #(OR a=expr b=expr) { r = MohawkLogicalOperator.or(a, b); }
| #(AND a=expr b=expr) { r = MohawkLogicalOperator.and(a, b); }
| #(NOT a=expr) { r = MohawkLogicalOperator.not(a); }
| #(REGEQ a=expr b=expr) { r = MohawkLogicalOperator.match(a, b); }
| #(STMT (stmts:.)?) {
  MohawkSymbols.push();
  r = exprs(#stmts);
  MohawkSymbols.pop();
}
| #(ACTION (actions:.)?) {
  MohawkSymbols.branch();
  MohawkSymbols.push();
  r = exprs(#actions);
  MohawkSymbols.pop();
  MohawkSymbols.endBranch();
}
| #(BEGIN_ACTION (begin_actions:.)?) {
  MohawkSymbols.branch();
  r = exprs(#begin_actions);
  MohawkSymbols.endBranch();
}
| { a = null; } #(":-O" (a=expr)?) {
  if ( a != null )
    System.out.println(a);
  else
    System.out.println();
}
| { a = null; } #(":-o" (a=expr)?) {
  if ( a != null )
    System.out.print(a);
}

| #(FUNC_CALL fname:NAME params=expr_list) {
  MohawkSymbols.branch();
  MohawkSymbols.push();
  MohawkFunctionHandler.call(this, fname.getText(), params);
  MohawkSymbols.pop();
  MohawkSymbols.endBranch();
}
| ":-(" { //System.out.println("BREAK");
            boolean fbreak=false;
        if (!fbreak){throw new MohawkBreak();}}
        //I  know  this  seems  round  about,  but  if  I  don't  have  a
conditional,
        //antlr  inserts  a  break  then  I  get  a  compilation  error  'cuz
it's unreachable

| ":->" { //System.out.println("CONTINUE");
            boolean fcontinue = false;
            //System.out.println("expr" + _t);
            //System.out.println("sibling " + _t.getNextSibling());
            if (!fcontinue){throw new MohawkContinue();}
        }
```

```
        //I know this seems round about, but if I don't have a
conditional,
        //antlr inserts a break then I get a compilation error 'cuz
it's unreachable

 | #(":^P" (s=expr)?){ //System.out.println("EXIT");
            boolean fexit=false;
            if (!fexit){throw new MohawkExit(s);} //exit like AWK does
        }
| ":-|" { //System.out.println("NEXT");
             boolean fnext=false;
        if (!fnext){throw new MohawkNext();}}

            //next - go to next record
| ":-<" { MohawkMain.getNextRecord(); }  //get line

| #(":-/" a=expr (thenp:~(":-\\"))? (":-\\" (elsep:.)?)?) //if
//a is a mohawk data type //expr is method, returns a mohawk data type
   { //:. accept token, don't eval, may or may not be else ?
        if (a != null && a.booleanValue())
        {
          if ( #thenp != null )
            r = expr(#thenp); //now evaluate the expr
        }
        else if (elsep != null)
        {
            r = expr(#elsep);
        }
   }
| #(":^~" cond1:. (stmt1:.)?)//while
   {
            MohawkSymbols.push(); // push the symbol table
            MohawkDataType temp = expr(#cond1);
            while (temp.booleanValue())
            {   try
                {
                    r = expr(#stmt1);
                    temp = expr(#cond1);
                }
                catch(MohawkContinue mc)
                {
                    r = expr(#cond1);
                }
                catch(MohawkBreak mb)
                {
                    break;
                }
            }
            MohawkSymbols.pop(); // pop the symbol table
        }
    | #(">^O" stmt2:. cond2:.)//do
        {
            MohawkSymbols.push(); // push the symbol table
            // do the stuff once
            r = exprs(#stmt2);
            // then continue if needed
            MohawkDataType temp = expr(#cond2);
```

```
            while (temp.booleanValue())
                {
                try{

                    r = expr(#stmt2);
                    temp = expr(#cond2);
                }
                catch(MohawkContinue mc)
                {
                    r = expr(#cond2);
                }
                catch(MohawkBreak mb)
                {
                    break;
                }
            }
            MohawkSymbols.pop(); // pop the symbol table
        }
    |    #(":^O"    LPAREN    (assgn1:~(SEMI))?    SEMI    expr1:.    SEMI
(assign2:~(SEMI|RPAREN))? RPAREN (stmt3:.)?)//for
        {
            // push the symbol stack
            MohawkSymbols.push();
            // first do the assignment
            if ( assgn1 != null )
              r = expr(assgn1);
            // then evaluate the expression to see if you should keep
going
            MohawkDataType temp = expr(#expr1);
            // and keep doing this while it's true
            while(temp.booleanValue())
            {
                    try {
                    r = expr(#stmt3);
                    r = expr(assign2);
                    temp = expr(#expr1);
                    }
                    catch (MohawkContinue mc)
                    {
                    r = expr(assign2);
                    temp = expr(#expr1);
                    }
                    catch (MohawkBreak mb)
                    {
                    break;
                    }

            }
            MohawkSymbols.pop(); // pop the symbol stack
        }
    | num:NUMBER { r.set(num.getText()); }
    | str:STRING { r.set(str.getText()); }
    | name:NAME  { r = MohawkSymbols.get(name.getText()); }
    | fv:FIELD_VAR { r = MohawkSymbols.get(fv.getText()); }
    | gv:GLOBAL_VAR { r = MohawkSymbols.get(gv.getText()); }
    | "true" { r.set("true"); }
    | "false" { r.set("false"); }
```

```
    ;

identifier returns [String id]
{
   // declare and initialize variables
   id = null;
}
: n:NAME { id = n.getText(); } | fv:FIELD_VAR  { id = fv.getText(); }
;


exprs returns [MohawkDataType r]
{
   // declare and initialize variables
   r = new MohawkDataType();
}
: (r = expr)*
;
```

8.3 MOHAWK Backend Code

8.3.1 CommonASTWithLines.java

```
import antlr.CommonAST;
import antlr.Token;

public class CommonASTWithLines extends CommonAST {
   private int line = 0;
   private int column = 0 ;

   public void initialize(Token tok) {
        super.initialize(tok);
        line=tok.getLine();
        column = tok.getColumn();
   }

   public int getLine(){
      return line;
   }

   public int getColumn() {
      return column;
   }
}
```

8.3.2 MohawkBreak.java

```
import java.lang.Exception;

public class MohawkBreak extends RuntimeException
{
   public MohawkBreak(String message)
   {
   }
```

```
    public MohawkBreak()
    {
    }

    public void breaknow() throws MohawkBreak
    {
       throw this;
    }
}
```

8.3.3 MohawkContinue.java

```
import java.lang.Exception;

public class MohawkContinue extends RuntimeException
{
    public MohawkContinue(String message)
    {
    }

    public MohawkContinue()
    {
    }

    public void cont() throws MohawkContinue
    {
       throw this;
    }
}
```

8.3.4 MohawkDataType.java

```
/**************************************
 * This class represents a data element in MOHAWK.
 * Data elements don't actually have a type... they can
 * be used as Strings, ints, floats, or bools interchangeably.
 */

public class MohawkDataType
{

    // for checking boolean values
    private static final String TRUE = "true";
    private static final String FALSE = "false";

    // this is pretty much the REAL value of the object
    // all the methods are based on this value
    String strValue;

    /** Default constructor */
    public MohawkDataType()
    {
       strValue = "";
    }

    /** Constructor that takes a float. */
    public MohawkDataType(float floatVal)
```

```
    {
       strValue = Float.toString(floatVal);
    }

    /** Constructor that takes an int. */
    public MohawkDataType(int intVal)
    {
       strValue = Integer.toString(intVal);
    }

    /** Constructor that takes a String. */
    public MohawkDataType(String val)
    {
       strValue = val;
    }

    /** Constructor that takes a boolean. */
    public MohawkDataType(boolean val)
    {
       if (val) strValue = TRUE;
       else strValue = FALSE;
    }


    /*
     * Sets the value of the data element by using the string
representation
     */
    public void set(String str)
    {
       strValue = str;
    }

    /*
     * Returns the value as a string
     */
    public String toString()
    {
       return strValue;
    }

    /*
     * Returns the value as an int. If the value has no integer
representation,
     * it throws a NumberFormatException.
     */
    public int intValue() throws NumberFormatException
    {
      // if it's null, then return zero
      if (strValue == null) return 0;
      else if (strValue.equals(TRUE)) return 1;

      return Integer.parseInt(strValue);
    }

    /*
```

```
    * Returns the value as a float. If the value has no float
representation,
    * it throws a NumberFormatException.
    */
   public float floatValue() throws NumberFormatException
   {
      // if it's null, then return zero
      if (strValue == null) return 0;
      else if (strValue.equals(TRUE)) return 1;
      return Float.parseFloat(strValue);
   }

   /*
    * Returns the value as a boolean.
    */
   public boolean booleanValue()
   {
      // if the value is null, it's false
      if (strValue == null) return false;

      // see if the string represents a boolean
      if (strValue.equals(TRUE))
         return true;
      else if (strValue.equals(FALSE))
         return false;

      // then, if it's a number, see if it's greater than zero
      try
      {
         float f = Float.parseFloat(strValue);
         if (f > 0) return true;
         else return false;
      }
      catch (NumberFormatException nfe) { }

      // if we got here, then it's just a plain old string
      return true;
   }


}
```

8.3.5 MohawkExit.java

```
import java.lang.Exception;

public class MohawkExit extends RuntimeException
{
   private int status = 0;

   public MohawkExit(MohawkDataType a)
   {

      try
      {
         status = a.intValue();
      }
```

```
       catch (NumberFormatException nfe)
          {
             try
                {
                   status = Integer.parseInt(a.toString());
                }
             catch (NumberFormatException nfe2)
                {
                   status = 1;
                }
          }
    }


    public MohawkExit()
    {
       status = 0;
    }

    public int getExitStatus()
    {

       return status;
    }
}
```

8.3.6 MohawkFileLoader.java

```
import java.io.*;
import java.util.StringTokenizer;
import java.util.LinkedList;
import java.util.Scanner;
import java.util.*;

/**
 * This class loads the data file and is responsible for keeping track
 * of all the records and their fields.
 */

public class MohawkFileLoader extends MohawkOperator
{
    // contains all of the records that are read from the datafile
    private List records = new ArrayList();

    /*
     * The constructor loads the file
     */
    public MohawkFileLoader(String filename) throws
FileNotFoundException
    {
       Scanner filescan = new Scanner(new File(filename));

       // read in each line and add it to the list
       while(filescan.hasNext())
       {
          records.add(filescan.nextLine());
       }
```

```
      // set the global variable to track the number of records
      int numRecords = records.size();
      MohawkSymbols.putGlobal(MohawkSymbols.NUM_RECORDS, new
MohawkDataType(numRecords));

   }

   /**
    * Gets the i-th record as an entire String, to set the $0 value.
    */
   public String getRecord(int i)
   {
      if (i < records.size())
      {
         return (String)records.get(i);
      }
      else
      {
         error("Warning! Tried to access record number " + i + " but
there are only " + records.size() + " records.");
      }
      return "";
   }

   /**
    * Gets the index-th record as an array of Strings.
    */
   public String[] getFields(int index)
   {
      if (records == null) return null;
      if (index >= records.size())
      {
         error("Warning! Tried to access record number " + index + "
but there are only " + records.size() + " records.");
         return new String[0];
      }

      // tokenize it into fields
      StringTokenizer fieldTokenizer = new
StringTokenizer((String)records.get(index), " ");
      String[] fields = new String[fieldTokenizer.countTokens()];
      for (int i = 0; i < fields.length; i++)
      {
         // add each field to the linked list
         String field = fieldTokenizer.nextToken().trim();
         fields[i] = field;
      }
      return fields;
   }

   /**
    * Returns the number of records in the input file.
    */
   public int numRecords()
   {
      return records.size();
```

```
   }

}
```

8.3.7 MohawkLogicalOperator.java

```
/**
 * This class has all of the methods for performing logical operations.
 * All of the methods are static.
 **/

public class MohawkLogicalOperator extends MohawkOperator
{

   /*
    * Compares two MohawkDataTypes and returns true if their string
values are equal.
    */
   public static MohawkDataType equals(MohawkDataType a, MohawkDataType
b)
   {
      if (a == null || b == null)
      {
        error("Warning! An argument to the equals function was null!");
         return new MohawkDataType(false);
      }
      // first, see if they're both ints
      try
      {
         int a_i = a.intValue();
         int b_i = b.intValue();
         return new MohawkDataType(a_i == b_i);
      }
      catch (NumberFormatException nfe) { }

      // if we get here, one of them isn't an int, so see if they're
floats
      try
      {
         float a_f = a.floatValue();
         float b_f = b.floatValue();
         return new MohawkDataType(a_f == b_f);
      }
      catch (NumberFormatException nfe) { }

      // hmmm... still no love... they must be strings
      return new MohawkDataType(a.toString().equals(b.toString()));
   }

   /*
    * Compares two MohawkDataTypes and returns true if their string
values are not equal.
    */
   public static MohawkDataType notEquals(MohawkDataType a,
MohawkDataType b)
   {
```

```
       if (a == null || b == null)
      {
         error("Warning! An argument to the notEquals function was
null!");
         return new MohawkDataType(false);
      }

       // first, see if they're both ints
       try
       {
          int a_i = a.intValue();
          int b_i = b.intValue();
          return new MohawkDataType(a_i != b_i);
       }
       catch (NumberFormatException nfe) { }

       // if we get here, one of them isn't an int, so see if they're
floats
       try
       {
          float a_f = a.floatValue();
          float b_f = b.floatValue();
          return new MohawkDataType(a_f != b_f);
       }
       catch (NumberFormatException nfe) { }

       // hmmm... still no love... they must be strings
       return new MohawkDataType(!(a.toString().equals(b.toString())));
   }

   /*
    * Compares two MohawkDataTypes and returns true if the first is
less than the second.
    */
   public static MohawkDataType lessThan(MohawkDataType a,
MohawkDataType b)
   {
       if (a == null || b == null)
      {
         error("Warning! An argument to the lessThan function was
null!");
         return new MohawkDataType(false);
      }

       // first, see if they're both ints
       try
       {
          int a_i = a.intValue();
          int b_i = b.intValue();
          return new MohawkDataType(a_i < b_i);
       }
       catch (NumberFormatException nfe) { }

       // if we get here, one of them isn't an int, so see if they're
floats
       try
       {
```

```
            float a_f = a.floatValue();
            float b_f = b.floatValue();
            return new MohawkDataType(a_f < b_f);
         }
         catch (NumberFormatException nfe) { }

         // hmmm... still no love... they must be strings
         int compare = a.toString().compareTo(b.toString());
         return new MohawkDataType(compare < 0);
      }

      /*
       * Compares two MohawkDataTypes and returns true if the first is
greater than the second.
       */
      public static MohawkDataType greaterThan(MohawkDataType a,
MohawkDataType b)
      {
         if (a == null || b == null)
         {
            error("Warning! An argument to the greaterThan function was
null!");
            return new MohawkDataType(false);
         }

         // first, see if they're both ints
         try
         {
            int a_i = a.intValue();
            int b_i = b.intValue();
            return new MohawkDataType(a_i > b_i);
         }
         catch (NumberFormatException nfe) { }

         // if we get here, one of them isn't an int, so see if they're
floats
         try
         {
            float a_f = a.floatValue();
            float b_f = b.floatValue();
            return new MohawkDataType(a_f > b_f);
         }
         catch (NumberFormatException nfe) { }

         // hmmm... still no love... they must be strings
         int compare = a.toString().compareTo(b.toString());
         return new MohawkDataType(compare > 0);
      }


      /*
       * Compares two MohawkDataTypes and returns true if the first is
less than
       * or equal to the second.
       */
      public static MohawkDataType lessThanOrEquals(MohawkDataType a,
MohawkDataType b)
```

```
   {
      if (a == null || b == null)
      {
         error("Warning! An argument to the lessThanOrEquals function
was null!");
         return new MohawkDataType(false);
      }

      // first, see if they're both ints
      try
      {
         int a_i = a.intValue();
         int b_i = b.intValue();
         return new MohawkDataType(a_i <= b_i);
      }
      catch (NumberFormatException nfe) { }

      // if we get here, one of them isn't an int, so see if they're
floats
      try
      {
         float a_f = a.floatValue();
         float b_f = b.floatValue();
         return new MohawkDataType(a_f <= b_f);
      }
      catch (NumberFormatException nfe) { }

      // hmmm... still no love... they must be strings
      int compare = a.toString().compareTo(b.toString());
      return new MohawkDataType(compare <= 0);
   }

   /*
    * Compares two MohawkDataTypes and returns true if the first is
greater than
    * or equal to the second.
    */
   public static MohawkDataType greaterThanOrEquals(MohawkDataType a,
MohawkDataType b)
   {
      if (a == null || b == null)
      {
         error("Warning! An argument to the greaterThanOrEquals
function was null!");
         return new MohawkDataType(false);
      }

      // first, see if they're both ints
      try
      {
         int a_i = a.intValue();
         int b_i = b.intValue();
         return new MohawkDataType(a_i >= b_i);
      }
      catch (NumberFormatException nfe) { }
```

```
      // if we get here, one of them isn't an int, so see if they're
floats
      try
      {
         float a_f = a.floatValue();
         float b_f = b.floatValue();
         return new MohawkDataType(a_f >= b_f);
      }
      catch (NumberFormatException nfe) { }

      // hmmm... still no love... they must be strings
      int compare = a.toString().compareTo(b.toString());
      return new MohawkDataType(compare >= 0);
   }

   /*
    * Compares two MohawkDataTypes and returns true if the first one
matches
    * the regular expression given by the second.
    */
   public static MohawkDataType match(MohawkDataType a, MohawkDataType
b)
   {
      if (a == null || b == null)
      {
         error("Warning! An argument to the match function was null!");
         return new MohawkDataType(false);
      }

      if (a.toString().matches(b.toString())) return new
MohawkDataType(true);
      else return new MohawkDataType(false);
   }

   /*
    * Compares two MohawkDataTypes and returns true if either one
represents
    * a boolean true.
    */
   public static MohawkDataType or(MohawkDataType a, MohawkDataType b)
   {
      if (a == null || b == null)
      {
         error("Warning! An argument to the or function was null!");
         return new MohawkDataType(false);
      }

      return new MohawkDataType(a.booleanValue() || b.booleanValue());
   }


   /*
    * Compares two MohawkDataTypes and returns true if both represent
    * a boolean true.
    */
   public static MohawkDataType and(MohawkDataType a, MohawkDataType b)
   {
```

```
       if (a == null || b == null)
      {
         error("Warning! An argument to the and function was null!");
         return new MohawkDataType(false);
      }

       return new MohawkDataType(a.booleanValue() && b.booleanValue());
   }

   /*
    * Returns the logical opposite of the boolean value held in the
MohawkDataType.
    */
   public static MohawkDataType not(MohawkDataType a)
   {
       if (a == null)
      {
         error("Warning! The argument to the not function was null!");
         return new MohawkDataType(false);
      }

       return new MohawkDataType(!a.booleanValue());
   }

}
```

8.3.8 MohawkMain.java

```
import java.io.*;
import antlr.CommonAST;
import java.util.LinkedList;
import java.util.Scanner;
import java.util.InputMismatchException;
/**
 * To do:
 * separate out logic for naming field vars in the begin block
 * logic for programs w/o begin blocks, w/o end blocks, etc.
 **/

public class MohawkMain {

   private static MohawkFileLoader loader;
   private static int currentRecordIndex;


   public static void main(String[] args) {
      MohawkLexer lexer = null;
      MohawkParser parser = null;
      CommonAST tree = null;
      MohawkWalker walker = null;
       try {
          // find the source code for the program

            String filename = args[0];
            lexer = new MohawkLexer(new FileInputStream(filename));

      } catch(Exception e) {
```

```
          System.out.println("\n\nusage: ./mohawk programfile
[ filename ]");
         boolean f = true;
         Scanner scanner = new Scanner(System.in);
         while (f)
            {
               System.out.println("\nEnter program filename to run or
'q' to quit");

               String filename = scanner.next();
                  if (filename.equalsIgnoreCase("q")){ System.exit(1);}
               System.out.println("\nEnter datafile to run or 'n' to
skip");

               try{

               if (scanner.hasNext())
                  {
                     String datafile = scanner.next();
                     if (datafile.equalsIgnoreCase("n") )
                        {
                           System.out.println("Running without input
file");
                           loader = null;
                        }
                     else
                        {
                           try{
                              // creating the new file loader
automagically reads the file
                              loader = new MohawkFileLoader(datafile);
                              f = false;
                           }
                           catch(FileNotFoundException exc)
                              {
                                 System.err.println("Error! Unable to
locate input file");

                              }
                           catch(NullPointerException npe)
                              {
                                 System.err.println("Error! There are
problems with " + datafile);

                              }
                        }
                  }

               lexer = new MohawkLexer(new FileInputStream(filename));
               f= false;
               }
               catch(NullPointerException npe)
                  {
                     System.err.println("Error! That file is not a
valid mohawk program file");

                  }
```

```
               catch(FileNotFoundException fnfe)
                  {
                     System.err.println("\nError! The mohawk program
file " + filename + " was not found");

                  }
            }
       }

      // if we made it here, we must have a program file
      // initialize the global variables
      MohawkSymbols.putGlobal(MohawkSymbols.NUM_RECORDS, new
MohawkDataType(0));
      MohawkSymbols.putGlobal(MohawkSymbols.NUM_FIELDS, new
MohawkDataType(0));
      // get the name of the data file
      if (args.length > 1)
         {
            String datafile = args[1];

<<<<<<< MohawkMain.java
      // get the name of the data file
      if (args.length > 1)
       {
          String filename = args[1];
          System.out.println("Reading data from " + filename);
          // creating the new file loader automagically reads the file
          loader = new MohawkFileLoader(filename);
      }

=======
            // creating the new file loader automagically reads the
file
            try{
               // creating the new file loader automagically reads the
file
               loader = new MohawkFileLoader(datafile);
            }
            catch(FileNotFoundException fnfe)
               {
                  System.err.println("\nError! Datafile " + datafile +"
not found");
                  Scanner scanner = new Scanner(System.in);
                  boolean f = true;
                  while (f)
                     {
                        System.out.println("\nEnter datafile to run or
'n' to skip");

                        try{
                           if (scanner.hasNext())
                              {
                                 datafile = scanner.next();
                                 if (datafile.equalsIgnoreCase("n") )
                                    {
                                       f = false;
                                    }
```

```
                                    else
                                       {
                                          try{
                                             // creating the new file
loader automagically reads the file
                                             loader = new
MohawkFileLoader(datafile);
                                             f = false;
                                          }
                                          catch(FileNotFoundException exc)
                                             {
                                                System.out.println("Error!
Unable to locate input file");
                                                f = true;
                                                loader = null;
                                             }
                                          catch(NullPointerException npe)
                                             {
                                                System.out.println("Error!
There are problems with " + datafile);
                                                f = true;
                                                loader = null;
                                             }
                                       }
                                 }

                           }
                           catch(Exception e)
                              {
                                 loader = null;
                              }
                        }

                  }
            }
         else
            {
               System.err.println("Warning! Running program without data
input file");
            }
>>>>>>> 1.27
         // parse the code
         parser = new MohawkParser(lexer);
         parser.setASTNodeType("CommonASTWithLines");
          // exceptions handled internally by the parser
         try {
            parser.program();
         } catch(Throwable t) {
             //System.err.println("Oi! Oi! Oi!: "+e);
            System.err.println(t.getMessage());
         }

         if ( parser.numErrors > 0 ) {
            System.err.println("Error! Fix above "+parser.numErrors+"
error(s) first.");
            System.exit(1);
         }
```

```
      tree = (CommonAST)parser.getAST();
      walker = new MohawkWalker();
      try {
         /*
           walk the code UNCOMMENT ME TO DEBUG
          */
         //System.out.println(tree.toStringList());
       } catch(Throwable t) {
         //System.err.println("oi! oi! oi!: "+e);
         //e.printStackTrace();
         System.err.println(t.getMessage());
      }
       try {
         // only do this if the file loader got created
         if (loader != null)
         {
             // set the number of fields variable #F to zero
           MohawkSymbols.putGlobal(MohawkSymbols.NUM_FIELDS, new
MohawkDataType(0));
           // set the flag where it's possible to create mappings to
field variables
           // MohawkSymbols.SET_FIELD_VARS = true;
           // do the begin portion
           try
               {
                  walker.begin(tree);
               }
           catch(MohawkContinue mc)
               {
                  System.err.println("Warning! Illegal use of
\"continue\" \":->\" keyword in begin block.\n" ); //??now what

               }
           catch(MohawkBreak mb)
               {
                  System.err.println("Warning! Illegal use of \"break\"
\":-(\" keyword in begin block.\n" ); //??now what
               }

           catch(MohawkNext mn)
               {
                  System.err.println("Warning! Illegal use of \"next\"
\":-|\" keyword in begin block.\n" ); //??now what
               }

           catch(MohawkExit me)
               {
                  int status;
                  //like AWK: When an exit statement is executed from a
BEGIN rule,
                 //the program stops processing everything immediately.
                  //No input records are read. However, if an END rule
is present,
                  //as part of executing the exit statement, the END
rule is executed
                  try
```

```
                    {
                       status = me.getExitStatus();
                       if (status == 0)
                           {
                              walker.end(tree);
                              System.exit(0);
                           }
                       else
                           {
                              System.err.println("Exiting with non-zero
status");
                              System.exit(1);

                           }
                    }
                 catch(MohawkExit me2)
                    {
                       if (me2.getExitStatus()==0)
                          {System.exit(0);}
                       else
                           {
                              System.err.println("Exiting with non-zero
status");
                              System.exit(1);
                           }
                    }
                 catch(MohawkNext mn)
                    {
                       System.err.println("Error! Illegal use of
\"next\" \":-|\" keyword in end block. Exiting.\n" );
                       System.exit(1);
                    }
                 catch(MohawkContinue mc)
                    {
                       System.err.println("Error! Illegal use of
\"continue\" \":->\" keyword in end block. Exiting.\n" );
                       System.exit(1);
                    }
                 catch(MohawkBreak mb)
                    {
                       System.err.println("Error! Illegal use of
\"break\" \":-(\" keyword in end block. Exiting.\n" );
                       System.exit(1);
                    }
                 catch(Throwable t)
                    {
                       System.err.println("Error! An internal error
occurred: " + t.getMessage());
                       System.exit(0);
                    }
              }


           // MohawkSymbols.SET_FIELD_VARS = false;

           // get all the records
```

```
            for (currentRecordIndex = 0; currentRecordIndex <
loader.numRecords(); currentRecordIndex++)
            {
                // get the current record
                try
                   {
                       getRecord();
                       walker.program(tree);
                   }

                catch(MohawkExit me)
                   {
                       int status = me.getExitStatus();
                       if (status == 0)
                          {
                             break; //this will leave the for loop and
continue to END execution
                          }
                       else
                          {
                             System.err.println("Exiting with non-zero
status");
                             System.exit(1);

                          }

                       //like AWK: an exit statement with status 0 that
is not part of a BEGIN or END rule
                       //stops the execution of any further automatic
rules for the current record,
                       //skips reading any remaining input records, and
executes the END rule if there is one.
                       //non-zero just exits, doesn't execute end
                   }

                catch(MohawkNext mn)
                   {
                       continue;//this is the appropriate context and
action to execute when encountering :-|
                   }
                catch(MohawkContinue mc)
                   {
                       System.err.println("Warning! Illegal use of
\"continue\" \":->\" keyword in program body.\n" );
                       // System.out.println("Executing end statement");
                   }

                catch(MohawkBreak mb)
                   {
                       System.err.println("Warning! Illegal use of
\"break\" \":-(\" keyword. Exiting.\n" );
                       // System.out.println("Executing end statement");
                   }

                catch(Throwable t)
                   {
```

```
                    System.err.println("Error! An internal error
occurred: " + t.getMessage());
                    System.exit(1);
                }
             }


             MohawkSymbols.cleanFieldVariables();


            try
               {
                  walker.end(tree);
               }
            catch(MohawkExit me)
               {
                  int status = me.getExitStatus();
                  //If exit is used as part of an END rule, it causes
the program to stop immediately

                  if (status == 0)
                     {
                        System.exit(0);
                     }
                  else
                     {
                        System.err.println("Exiting mohawk with non-
zero status -:-|");
                        System.exit(1);
                     }
               }

            catch(MohawkNext mn)
               {
                  System.err.println("Error! Illegal use of \"next\"
\":-|\" keyword in end block. Exiting." );
               }
            catch(MohawkContinue mc)
               {
                  System.err.println("Error! Illegal use of
\"continue\" \":->\" keyword in end block. Exiting." );
               }

            catch(MohawkBreak mb)
               {
                  System.err.println("Error! Illegal use of \"break\"
\":-(\" keyword in end block. Exiting." );

               }
         }



          else //no datafile given
            {
               try
                  {
```

```
                    walker.begin(tree);
                 }
              catch(MohawkContinue mc)
                 {
                    System.err.println("Warning! Illegal use of
\"continue\" \":->\" keyword in begin block.\n" ); //??now what

                 }
              catch(MohawkBreak mb)
                 {
                    System.err.println("Warning! Illegal use of
\"break\" \":-(\" keyword in begin block.\n" ); //??now what
                 }

              catch(MohawkNext mn)
                 {
                    System.err.println("Warning! Illegal use of
\"next\" \":-|\" keyword in begin block.\n" ); //??now what
                 }

              catch(MohawkExit me)
                 {
                    //like AWK: When an exit statement with 0 status
is executed from a BEGIN rule,
                    //the program stops processing everything
immediately.
                    //No input records are read. However, if an END
rule is present,
                    //as part of executing the exit statement, the END
rule is executed
                    //non-zero status just exits without executing end
block
                    int status =  me.getExitStatus();
                    try
                       {
                           if (status == 0)
                              {

                                 walker.end(tree);
                                 System.exit(0);
                              }
                           else
                              {
                                 System.err.println("Exiting with non-
zero status");
                                 System.exit(1);
                              }
                       }
                    catch(MohawkExit me2)
                       {

                          status = me2.getExitStatus();
                          if (status == 0)
                             {
                                System.exit(0);
                             }
                          else
```

```
                                {
                                   System.err.println("Exiting with non-
zero exit status");
                                   System.exit(1);
                                }
                          }
                       catch(MohawkNext mn)
                          {
                             System.err.println("Error! Illegal use of
\"next\" \":-|\" keyword in end block. Exiting.\n" );
                             System.exit(1);
                          }
                       catch(MohawkContinue mc)
                          {
                             System.err.println("Error! Illegal use of
\"continue\" \":->\" keyword in end block. Exiting.\n" );
                             System.exit(1);
                          }
                       catch(MohawkBreak mb)
                          {
                             System.err.println("Error! Illegal use of
\"break\" \":-(\" keyword in end block. Exiting.\n" );
                             System.exit(1);
                          }
                       catch(Throwable t)
                          {
                             //  System.out.println("Exiting mohawk -:-
|\n");
                             System.err.println("Error! An internal error
occurred: " + t.getMessage());
                             System.exit(1);

                          }
                    }

                 try
                    {
                    walker.end(tree);
                    }
                 catch(MohawkExit me)
                    {   int status = me.getExitStatus();
                        //If exit is used as part of an END rule, it
causes the program to stop immediately
                    if (status == 0 )
                        {
                           System.exit(0);
                        }
                    else
                        {
                           System.err.println("Exiting mohawk with non-
zero status");
                           System.exit(1);
                        }

                    }

                 catch(MohawkNext mn)
```

```
                  {
                     System.err.println("Error! Illegal use of \"next\"
\":-|\" keyword in end block. Exiting." );
                  }
               catch(MohawkContinue mc)
                  {
                     System.err.println("Error! Illegal use of
\"continue\" \":->\" keyword in end block. Exiting." );
                  }

               catch(MohawkBreak mb)
                  {
                     System.err.println("Error! Illegal use of
\"break\" \":-(\" keyword in end block. Exiting." );

                  }
            }
            }
      catch(Throwable t) {
         // System.err.println("oi! oi! oi! 4: "+e);
         //e.printStackTrace();
         System.err.println("Error! An internal error occurred " +
t.getMessage());
         System.exit(1);
      }
   }

   /**
    * This method is used to fetch the next record from the input file
and
    * set the variables accordingly
    */
   private static void getRecord()
   {
      // clean up the field variables
       MohawkSymbols.cleanFieldVariables();

       // get the next record
       String record = loader.getRecord(currentRecordIndex);
       MohawkSymbols.putGlobal("$0", new MohawkDataType(record));

       // get its fields
       String[] fields = loader.getFields(currentRecordIndex);
       for (int j = 0; j < fields.length; j++)
       {
          // write each field into the symbol table
           MohawkSymbols.putGlobal("$" + (j+1), new
MohawkDataType(fields[j]));
           // System.out.println("set $" + (j+1) + " to " + fields[j]);
       }

      // set the number of fields variable #F
      MohawkSymbols.putGlobal(MohawkSymbols.NUM_FIELDS, new
MohawkDataType(fields.length));
   }

   /**
```

```
    * This method is called when the "NEXT" keyword is encountered in a
MOHAWK program.
    * It increments the currentRecordIndex so that we're pointing to
the next record,
    * and then fetches it from the FileLoader.
    */
   public static void getNextRecord()
   {
      // only get the next record if we're not on the last one
      if (currentRecordIndex < loader.numRecords() - 1)
      {
         currentRecordIndex++;
         getRecord();
      }
      else
      {
         System.err.println("Warning! Attempt to get the next record
when already on the last one!");
      }
   }
}
```

8.3.9 MohawkMathOperator.java

```
/**
 * This class contains all of the methods for math operations in MOHAWK.
 * All of the methods are static.
 */

public class MohawkMathOperator extends MohawkOperator
{

   /** Creates a new instance of MohawkMathOperator */
   public MohawkMathOperator()
   {
   }

   /**
    * Returns a MohawkDataType which is the sum of the two arguments.
    * If the arguments both represent ints, the return value is an int.
    * If one argument represents a float and the other represents an
int,
    * or both are floats, the return value is a float.
    * Otherwise, the return value is a String.
    */
   public static MohawkDataType add(MohawkDataType a, MohawkDataType b)
   {
      if (a == null || b == null)
      {
         error("Warning! An operand to the add function was null!");
         return null;
      }

      // first, see if they're both ints
      try
      {
         int a_i = a.intValue();
```

```
            int b_i = b.intValue();
            return new MohawkDataType(a_i + b_i);
         }
         catch (NumberFormatException nfe) { }

         // if we get here, one of them isn't an int, so see if they're
floats
        // the trick being that floatValue will return a float if the
        // object actually returns an int
         try
         {
            float a_f = a.floatValue();
            float b_f = b.floatValue();
            return new MohawkDataType(a_f + b_f);
         }
         catch (NumberFormatException nfe) {

        }

         // hmmm... still no love... they must be strings
         return new MohawkDataType(a.toString().concat(b.toString()));
   }



   /**
    * Returns a MohawkDataType which is the difference of the two
arguments.
    * If the arguments both represent ints, the return value is an int.
    * If one argument represents a float and the other represents an
int,
    * or both are floats, the return value is a float.
    * If one argument represents a String and the other represents an
int
    * or a float, the return value is an int or a float (accordingly)
and
    * the String is represented as 0.
    * If both arguments are Strings, the return value is 0.
    *
    */
   public static MohawkDataType subtract(MohawkDataType a,
MohawkDataType b)
   {
      if (a == null || b == null)
      {
         error("Warning! An operand to the subtract function was
null!");
         return null;
      }
      boolean a_int = false, b_int = false, a_float = false, b_float =
false;
      int a_i = 0, b_i = 0;
      float a_f = 0, b_f = 0;
       // first, see if a is an int
       try
       {
          a_i = a.intValue();
```

```
         a_int = true;
      }
       catch (NumberFormatException nfe) { }

      // now see if b is an int
      try
      {
          b_i = b.intValue();
         b_int = true;
       }
       catch (NumberFormatException nfe) { }

      // if a isn't an int, it's either a float or a string
      if (!a_int)
      {
         try
         {
            a_f = a.floatValue();
            a_float = true;
         }
         catch (NumberFormatException nfe) { }
      }

      // if b isn't an int, it's either a float or a string
      if (!b_int)
      {
         try
         {
            b_f = b.floatValue();
            b_float = true;
         }
         catch (NumberFormatException nfe) { }
      }

      // now let's figure out the result
      if (a_int && b_int) return new MohawkDataType(a_i - b_i);
     else if (a_float && b_float) return new MohawkDataType(a_f - b_f);
      else if (a_int && b_float) return new MohawkDataType((float)(a_i
- b_f));
      else if (a_float && b_int) return new MohawkDataType((float)(a_f
- b_i));
      else if (a_int) return new MohawkDataType(a_i);
      else if (b_int) return new MohawkDataType(-b_i);
      else if (a_float) return new MohawkDataType(a_f);
      else if (b_float) return new MohawkDataType(-b_f);
      else return new MohawkDataType(0);

   }


   /**
    * Returns a MohawkDataType which is the product of the two
arguments.
    * If the arguments both represent ints, the return value is an int.
    * If one argument represents a float and the other represents an
int,
    * or both are floats, the return value is a float.
```

```
    * If either argument represents a String, the return value is 0.
    *
    */
   public static MohawkDataType multiply(MohawkDataType a,
MohawkDataType b)
   {
      if (a == null || b == null)
      {
         error("Warning! An operand to the multiply function was
null!");
         return null;
      }
      boolean a_int = false, b_int = false, a_float = false, b_float =
false;
      int a_i = 0, b_i = 0;
      float a_f = 0, b_f = 0;
       // first, see if a is an int
       try
       {
          a_i = a.intValue();
         a_int = true;
      }
       catch (NumberFormatException nfe) { }

      // now see if b is an int
      try
      {
          b_i = b.intValue();
         b_int = true;
       }
       catch (NumberFormatException nfe) { }

      // if a isn't an int, it's either a float or a string
      if (!a_int)
      {
         try
         {
            a_f = a.floatValue();
            a_float = true;
         }
         catch (NumberFormatException nfe) { }
      }

      // if b isn't an int, it's either a float or a string
      if (!b_int)
      {
         try
         {
            b_f = b.floatValue();
            b_float = true;
         }
         catch (NumberFormatException nfe) { }
      }

      // now let's figure out the result
      if (a_int && b_int) return new MohawkDataType(a_i * b_i);
     else if (a_float && b_float) return new MohawkDataType(a_f * b_f);
```

```
      else if (a_int && b_float) return new MohawkDataType((float)(a_i
* b_f));
      else if (a_float && b_int) return new MohawkDataType((float)(a_f
* b_i));
      else if(a.toString().equalsIgnoreCase("dog"))
         {
            if (b.toString().equalsIgnoreCase("cat"))
            {
               return new MohawkDataType("mouse -:>");
            }
         }
      else if(a.toString().equalsIgnoreCase("cat"))
         {
            if (b.toString().equalsIgnoreCase("dog"))
               {
                  return new MohawkDataType("mouse -:>");
               }
         }
      else if(a.toString().equalsIgnoreCase("chris"))
         {
            if (b.toString().equalsIgnoreCase("ryan"))
               {
                  return new MohawkDataType("MO");
               }
         }
      else if(a.toString().equalsIgnoreCase("MO"))
         {
            if (b.toString().equalsIgnoreCase("lauren"))
               {
                  return new MohawkDataType("MOHA");
               }
         }
      else if(a.toString().equalsIgnoreCase("MOHA"))
         {
            if (b.toString().equalsIgnoreCase("joeng"))
               {
                  return new MohawkDataType("MOHAWK!! -;-)");
               }
         }



      return new MohawkDataType(0);

   }



   /**
    * Returns a MohawkDataType which is the quotient of the two
arguments.
    * If an argument represents a String (ie, is non-numeric), it is
    * treated as a zero.
    * If the second argument is zero or a non-numeric String, the
return
    * value will be Float.POSITIVE_INFINITY or Float.NEGATIVE_INFINITY
```

```
    * according to the sign of the first argument. If the first
argument
    * is zero, the return value will be Float.POSITIVE_INFINITY.
    * If the arguments both represent ints, the return value is an int.
    * If one argument represents a float and the other represents an
int,
    * or both are floats, the return value is a float.
    *
    */
   public static MohawkDataType divide(MohawkDataType a, MohawkDataType
b)
   {
       if (a == null || b == null)
      {
         error("Warning! An operand to the divide function was null!");
         return null;
      }
      boolean a_int = false, b_int = false, a_float = false, b_float =
false;
      int a_i = 0, b_i = 0;
      float a_f = 0, b_f = 0;
       // first, see if a is an int
       try
       {
          a_i = a.intValue();
         a_int = true;
      }
       catch (NumberFormatException nfe) { }

      // now see if b is an int
      try
      {
          b_i = b.intValue();
         b_int = true;
       }
       catch (NumberFormatException nfe) { }

      // if a isn't an int, it's either a float or a string
      if (!a_int)
      {
         try
         {
            a_f = a.floatValue();
            a_float = true;
         }
         catch (NumberFormatException nfe) { }
      }

      // if b isn't an int, it's either a float or a string
      if (!b_int)
      {
         try
         {
            b_f = b.floatValue();
            b_float = true;
         }
         catch (NumberFormatException nfe) { }
```

```
      }


      // now let's figure out the result
      if (b_float && (b.floatValue() == Float.POSITIVE_INFINITY ||
b.floatValue() == Float.NEGATIVE_INFINITY))
      {
         if (a_float) return new MohawkDataType((float)0.0);
         else return new MohawkDataType(0);
      }
      else if ((b_int && b_i == 0) || (b_float && b_f == 0) || (!b_int
&& !b_float))
      {
         if ((a_int && a_i >= 0) || (a_float && a_f >= 0) || (!a_int
&& !a_float))
         {
            return new MohawkDataType(Float.POSITIVE_INFINITY);
         }
         else
         {
            return new MohawkDataType(Float.NEGATIVE_INFINITY);
         }
      }
      if (a_int && b_int) return new MohawkDataType(a_i/b_i);
      else if (a_float && b_float) return new MohawkDataType(a_f/b_f);
      else if (a_int && b_float) return new
MohawkDataType((float)(a_i/b_f));
      else if (a_float && b_int) return new
MohawkDataType((float)(a_f/b_i));
      else return new MohawkDataType(0);

   }



   /**
    * Returns a MohawkDataType which is the remainder of dividing the
    * first argument by the second.
    * If an argument represents a String (ie, is non-numeric), it is
    * treated as a zero.
    * If the second argument is zero or a non-numeric String, the
return
    * value will be Float.POSITIVE_INFINITY or Float.NEGATIVE_INFINITY
    * according to the sign of the first argument. If the first
argument
    * is zero, the return value will be Float.POSITIVE_INFINITY.
    * If the arguments both represent ints, the return value is an int.
    * If one argument represents a float and the other represents an
int,
    * or both are floats, the return value is a float.
    *
    */
   public static MohawkDataType mod(MohawkDataType a, MohawkDataType b)
   {
      if (a == null || b == null)
      {
         error("Warning! An operand to the mod function was null!");
```

```
            return null;
        }
        boolean a_int = false, b_int = false, a_float = false, b_float =
false;
        int a_i = 0, b_i = 0;
        float a_f = 0, b_f = 0;
         // first, see if a is an int
         try
         {
             a_i = a.intValue();
            a_int = true;
        }
         catch (NumberFormatException nfe) { }

        // now see if b is an int
        try
        {
             b_i = b.intValue();
            b_int = true;
         }
         catch (NumberFormatException nfe) { }

        // if a isn't an int, it's either a float or a string
        if (!a_int)
        {
            try
            {
               a_f = a.floatValue();
               a_float = true;
            }
            catch (NumberFormatException nfe) { }
        }

        // if b isn't an int, it's either a float or a string
        if (!b_int)
        {
            try
            {
               b_f = b.floatValue();
               b_float = true;
            }
            catch (NumberFormatException nfe) { }
        }


        // now let's figure out the result
        if ((b_int && b_i == 0) || (b_float && b_f == 0) || (!b_int
&& !b_float))
        {
            if ((a_int && a_i >= 0) || (a_float && a_f >= 0) || (!a_int
&& !a_float))
            {
               return new MohawkDataType(Float.POSITIVE_INFINITY);
            }
            else
            {
               return new MohawkDataType(Float.NEGATIVE_INFINITY);
```

```
         }
      }
      if (a_int && b_int) return new MohawkDataType(a_i%b_i);
      else if (a_float && b_float) return new MohawkDataType(a_f%b_f);
      else if (a_int && b_float) return new
MohawkDataType((float)(a_i%b_f));
      else if (a_float && b_int) return new
MohawkDataType((float)(a_f%b_i));
      else return new MohawkDataType(0);

   }



   /**
    * Returns a MohawkDataType which represents the negation of the
value
    * represented by the argument.
    * If the argument represents an int, the return value will be an
int.
    * If the argument represents a float, the return value will be an
float.
    * Otherwise, the return value is 0.
    */
   public static MohawkDataType negate(MohawkDataType b)
   {
      if (b == null)
      {
         error("Warning! Unary negation on null reference");
         return null;
      }
      boolean b_int = false, b_float = false;
      int b_i = 0;
      float b_f = 0;

      try
      {
          b_i = b.intValue();
         b_int = true;
       }
       catch (NumberFormatException nfe) { }


      if (!b_int)
      {
         try
         {
            b_f = b.floatValue();
            b_float = true;
         }
         catch (NumberFormatException nfe) { }
      }

      // now let's figure out the result
      if (b_int) return new MohawkDataType(0 - b_i);
      else if (b_float) return new MohawkDataType(0 - b_f);
      else return new MohawkDataType(0);
```

```
   }


   /**
    * Returns a MohawkDataType which is the value of the first argument
    * raised to the power of the second.
    * If an argument represents a String (ie, is non-numeric), it is
    * treated as a zero.
    * If the second argument is zero or a non-numeric String, the
return
    * value will be 1.
    * The return value will be of type float, unless both arguments
represent
    * integers, in which case the return type will be an int.
    *
    */
   public static MohawkDataType exp(MohawkDataType a, MohawkDataType b)
   {
      if (a == null || b == null)
      {
         error("Warning! An operand to the exp function was null!");
         return null;
      }
      boolean a_int = false, b_int = false, a_float = false, b_float =
false;
      int a_i = 0, b_i = 0;
      float a_f = 0, b_f = 0;
       // first, see if a is an int
       try
       {
          a_i = a.intValue();
         a_int = true;
      }
       catch (NumberFormatException nfe) { }

      // now see if b is an int
      try
      {
          b_i = b.intValue();
         b_int = true;
       }
       catch (NumberFormatException nfe) { }

      // if a isn't an int, it's either a float or a string
      if (!a_int)
      {
         try
         {
            a_f = a.floatValue();
            a_float = true;
         }
         catch (NumberFormatException nfe) { }
      }

      // if b isn't an int, it's either a float or a string
      if (!b_int)
      {
```

```
         try
         {
            b_f = b.floatValue();
            b_float = true;
         }
         catch (NumberFormatException nfe) { }
      }

      // now let's figure out the result
      if (a_int && b_int)
      {
         if (b_i > 0) return new MohawkDataType((int)Math.pow(a_i,
b_i));
         else return new MohawkDataType((float)Math.pow(a_i, b_i));
      }
      else if (a_float && b_float) return new
MohawkDataType((float)Math.pow(a_f, b_f));
      else if (a_int && b_float) return new
MohawkDataType((float)Math.pow(a_i,b_f));
      else if (a_float && b_int) return new
MohawkDataType((float)Math.pow(a_f,b_i));
      else if (a_int) return new MohawkDataType(1);
      else if (b_int) return new MohawkDataType(0);
      else if (a_float) return new MohawkDataType((float)1.0);
      else if (b_float) return new MohawkDataType(0);
      else return new MohawkDataType(1);


   }

}
```

8.3.10 MohawkNext.java

```
import java.lang.Exception;

public class MohawkNext extends RuntimeException
{
   public MohawkNext(String message)
   {
   }

   public MohawkNext()
   {
   }

   public void cont() throws MohawkNext
   {
      throw this;
   }
}
```

8.3.11 MohawkOperator.java

```
/**
 * This is the base class for all the other operators.
 */
```

```
public class MohawkOperator
{
   /**
    * Prints the error message to System.err.println
    */
   protected static void error(String msg)
   {
      System.err.println(msg);
   }
}
```

8.3.12 MohawkParser.java

```
BitSet(mk_tokenSet_7());
      private static final long[] mk_tokenSet_8() {
            long[] data = { -2303727548841853438L, 397311L, 0L, 0L};
            return data;
      }
      public static final BitSet _tokenSet_8 = new
BitSet(mk_tokenSet_8());

      }
```

8.3.13 MohawkStringOperator.java

```
/**
 * This class contains the methods for String operations in MOHAWK.
 * All methods are static.
 */

public class MohawkStringOperator extends MohawkOperator
{


   /*
    * Concatenates two strings and returns a new MohawkDataType
containing the concatenation.
    */
   public static MohawkDataType concatenate(MohawkDataType a,
MohawkDataType b)
   {
      if (a == null || b == null)
      {
         error("Warning! An operand to the concatenate function was
null!");
         return new MohawkDataType();
      }

       // default to the empty string
       String a_str = "", b_str = "";

       // figure out the string values
       if (a != null)
       {
          a_str = a.toString();
       }
```

```
       if (b != null)
       {
          b_str = b.toString();
       }

       return new MohawkDataType(a_str + b_str);
   }

}
```

8.3.14 MohawkSymbols.java

```
import java.util.Hashtable;
import java.util.LinkedList;
import java.util.ListIterator;

/**
 * Mohawk's Symbol table.
 * Mohawk keeps the local symbols separate from the global symbols.
 * Global symbols are stored in a single hash table.
 * Local symbols are stored in a list of lists of hash tables to
 *   support local function scope and nested within functions and
 *   actions.
 **/

public class MohawkSymbols
{
   // the mapping of symbol names to their values
   // key: String representation of the name
   // value: MohawkDataType object
   private static Hashtable<String,MohawkDataType> globalSymbolTable;
   // A list of lists of hashtables of symbols
   private static LinkedList<LinkedList <Hashtable
<String,MohawkDataType> > > localSymbolTables;

   // naming of field variables
   // key: String of the new name
   // value: String of the field var name, eg $1
   private static Hashtable fieldVarAliases;

   public static final String NUM_RECORDS = "#R";
   public static final String NUM_FIELDS = "#F";
   // public static boolean SET_FIELD_VARS = false;

   /**
    * Allocate the symbol table data structures
    */
   private static void initSymbolTable()
   {
      globalSymbolTable = new Hashtable();
      localSymbolTables = new LinkedList();
      fieldVarAliases = new Hashtable();
   }
   /**
    *  push Symbol table ( when entering a new nested scope level )
    */
   public static void push()
```

```
   {
      if ( localSymbolTables == null ) initSymbolTable();

      LinkedList head = (LinkedList)localSymbolTables.getFirst();
      head.addFirst( new Hashtable() );
   }
   /**
    *  pop Symbol table  ( when leaving a scope level )
    */
   public static void pop()
   {
      if ( localSymbolTables == null ) initSymbolTable();

      LinkedList head = (LinkedList)localSymbolTables.getFirst();
      head.removeFirst();
   }
   /**
    * Branch the symbol table (when entering a function call)
    */
   public static void branch()
   {
      if ( localSymbolTables == null ) initSymbolTable();
      localSymbolTables.addFirst( new LinkedList() );
   }
   /**
    * End a symbol table branch ( when exitting a function call )
    */
   public static void endBranch()
   {
      if ( localSymbolTables == null ) initSymbolTable();
      localSymbolTables.removeFirst();
   }

   /*
    * puts a value in the global symbol table
    */
   public static MohawkDataType putGlobal(String name, MohawkDataType r)
   {
       // create the table if it doesn't already exist
       if (localSymbolTables == null) initSymbolTable();
      globalSymbolTable.put( name, r );
      return r;
   }


   /**
    * Helper method for legacy purposes
    */
   public static MohawkDataType put(String name, MohawkDataType r)
   {
      return put(name, r, false);
   }

   /*
    * puts a value in the symbol table (try local then global)
    * if isFunctionCall is true, this will NOT put it in the global
table
```

```
    */
   public static MohawkDataType put(String name, MohawkDataType r,
boolean isFunctionCall)
   {
      // create the table if it doesn't already exist
      if (localSymbolTables == null) initSymbolTable();

      MohawkDataType lvalue = getNoWarn(name, isFunctionCall);
      lvalue.strValue = r.strValue;

      return lvalue;
   }

   /*
    * retrieves the value from the symbol table
    * if a symbol is requested that is not yet in the symbol table,
    * then this will create a new one with value "0" and put it in the
table
    */
   public static MohawkDataType get(String name)
   {
      // create the table if it doesn't already exist
      if (localSymbolTables == null) initSymbolTable();

      // first, check to see if there is a link to a field variable
      String fieldVarName = (String)(fieldVarAliases.get(name));
      // if it's not null, then this is the actual name we want
      if (fieldVarName != null) name = fieldVarName;

      MohawkDataType returnVal = null;
      // get the value from the symbol table
      if ( localSymbolTables.size() > 0 &&
localSymbolTables.getFirst().size() > 0 )
      {
         LinkedList<Hashtable <String,MohawkDataType> > head =
(LinkedList)localSymbolTables.getFirst();
         ListIterator<Hashtable<String,MohawkDataType> >
symbolTableIterator = head.listIterator(0);
         Hashtable<String,MohawkDataType> symbolTable;
         while( symbolTableIterator.hasNext() && returnVal == null )
         {
            symbolTable = symbolTableIterator.next();
            returnVal = (MohawkDataType)symbolTable.get( name );
         }
      }


      if ( returnVal == null )
      {
         returnVal = globalSymbolTable.get( name );
         //System.out.println("Got " + name + " from global symbol
table");
      }

      // if it doesn't exist in the symbol table, we need to handle it
here
      if (returnVal == null)
```

```
      {
          System.err.println("Warning! Symbol " + name + " not found --
defaulting to null/empty string");
          returnVal = new MohawkDataType("");
         if ( localSymbolTables.size() > 0 &&
localSymbolTables.getFirst().size() > 0 )
         {
            localSymbolTables.getFirst().getFirst().put( name,
returnVal );
            //System.out.println("Put " + name + " into local symbol
table");
         }
         else
         {
            putGlobal( name, returnVal );
            //System.out.println("Put " + name + " into global symbol
table");
         }
      }

      return returnVal;
   }


   /*
    * Same as above but doesn't print a warning if it
    * can't find the symbol.
    */
   public static MohawkDataType getNoWarn(String name, boolean
isFunctionCall)
   {
      // create the table if it doesn't already exist
      if (localSymbolTables == null) initSymbolTable();

      // first, check to see if there is a link to a field variable
      String fieldVarName = (String)(fieldVarAliases.get(name));
      // if it's not null, then this is the actual name we want
      if (fieldVarName != null) name = fieldVarName;

      MohawkDataType returnVal = null;
      // get the value from the symbol table
      if ( localSymbolTables.size() > 0 &&
localSymbolTables.getFirst().size() > 0 )
      {
         LinkedList<Hashtable <String,MohawkDataType> > head =
(LinkedList)localSymbolTables.getFirst();
         ListIterator<Hashtable<String,MohawkDataType> >
symbolTableIterator = head.listIterator(0);
         Hashtable<String,MohawkDataType> symbolTable;
         while( symbolTableIterator.hasNext() && returnVal == null )
         {
            symbolTable = symbolTableIterator.next();
            returnVal = (MohawkDataType)symbolTable.get( name );
         }
      }

      if ( returnVal == null && !isFunctionCall)
```

```
      {
         returnVal = globalSymbolTable.get( name );
         //System.out.println("Getting " + name + " from global symbol
table");
      }

       // if it doesn't exist in the symbol table, we need to handle it
here
       if (returnVal == null)
       {
          returnVal = new MohawkDataType("");
         if ( localSymbolTables.size() > 0 &&
localSymbolTables.getFirst().size() > 0 )
         {
            localSymbolTables.getFirst().getFirst().put( name,
returnVal );
            //System.out.println("Putting " + name + " in local symbol
table");
         }
         else
         {
            putGlobal( name, returnVal );
            //System.out.println("Putting " + name + " in global symbol
table");
         }
       }

       return returnVal;
   }


   /*
    * Helper method for legacy purposes
    */
   public static MohawkDataType getNoWarn(String name)
   {
      return getNoWarn(name, false);
   }

   /**
    * This method links a name to a fieldVar. The name then becomes a
pointer to it,
    * ie they are completely linked.
    */
   public static void link(String name, String fieldVar)
   {
      if (fieldVarAliases == null) initSymbolTable();

      // maybe have some check to see if it's already been entered?
      fieldVarAliases.put(name, fieldVar);
   }


   /*
    * This method returns true if the "name" is currently in use as a
variable
    */
```

```
   public static boolean exists(String name)
   {
      MohawkDataType lvalue = getNoWarn( name );
      return ( lvalue != null );
   }

   /*
    * goes through the symbol tables and deletes all the entries
starting with "$"
    * used when there is a new record being read and we don't want to
retain old values
    */
   public static void cleanFieldVariables()
   {
      if (localSymbolTables == null) return;

      Object[] keys = globalSymbolTable.keySet().toArray();
      for (int i = 0; i < keys.length; i++)
      {
         String key = keys[i].toString();
          if (key.startsWith("$") && !key.equals(NUM_RECORDS)
&& !key.equals(NUM_FIELDS))
         {
            globalSymbolTable.remove(keys[i]);
         }
      }
   }
}
```

8.4 Mohawk Test Code

JUnit Test Code

8.4.1 MohawkTest.java

```
import java.util.*;
import junit.framework.*;

public class MohawkTest extends TestCase {

      public static Test suite() {
            TestSuite suite = new TestSuite();
            suite.addTest(new MohawkMathTest("testMohawkMath"));
            suite.addTest(new MohawkLogicalTest("testMohawkLogical"));
            suite.addTest(new MohawkStringTest("testMohawkString"));
            suite.addTest(new MohawkRegexTest("testMohawkRegex"));
            return suite;
      }

      public static void main (String [] args) {
            junit.textui.TestRunner.run(suite());
      }
      public MohawkTest (String value) {
            super.setName(value);
      }
```

```
}
```

8.4.2 MohawkLogicalTest.java

```
import java.util.*;
import junit.framework.*;

public class MohawkLogicalTest extends TestCase {
	public MohawkDataType int1, int2, float1, float2, string1,
string2, boolTrue, boolFalse;
	public MohawkLogicalOperator testOperator;
	public TestCase test;

	public void runTest() {
		testEquals();
		testNotEquals();
		testGreaterThan();
		testGreaterThanOrEquals();
		testLessThan();
		testLessThanOrEquals();
		testMatch();
		testOr();
		testAnd();
		testNot();
	}

	public void testEquals() {
		assertEquals((new MohawkDataType(false)).toString(),
testOperator.equals(int1, int2).toString());
		assertEquals((new MohawkDataType(true)).toString(),
testOperator.equals(int1, int1).toString());
		assertEquals((new MohawkDataType(false)).toString(),
testOperator.equals(float1, float2).toString());
		assertEquals((new MohawkDataType(true)).toString(),
testOperator.equals(float1, float1).toString());
		assertEquals((new MohawkDataType(false)).toString(),
testOperator.equals(string1, string2).toString());
		assertEquals((new MohawkDataType(true)).toString(),
testOperator.equals(string1, string1).toString());
		assertEquals((new MohawkDataType(true)).toString(),
testOperator.equals(int1, float1).toString());
		assertEquals((new MohawkDataType(false)).toString(),
testOperator.equals(int1, string1).toString());
		assertEquals((new MohawkDataType(false)).toString(),
testOperator.equals(float1, string1).toString());
	}

	public void testNotEquals() {
		assertEquals((new MohawkDataType(true)).toString(),
testOperator.notEquals(int1, int2).toString());
		assertEquals((new MohawkDataType(false)).toString(),
testOperator.notEquals(int1, int1).toString());
		assertEquals((new MohawkDataType(true)).toString(),
testOperator.notEquals(float1, float2).toString());
		assertEquals((new MohawkDataType(false)).toString(),
testOperator.notEquals(float1, float1).toString());
```

```
            assertEquals((new         MohawkDataType(true)).toString(),
testOperator.notEquals(string1, string2).toString());
            assertEquals((new        MohawkDataType(false)).toString(),
testOperator.notEquals(string1, string1).toString());
            assertEquals((new        MohawkDataType(false)).toString(),
testOperator.notEquals(int1, float1).toString());
            assertEquals((new         MohawkDataType(true)).toString(),
testOperator.notEquals(int1, string1).toString());
            assertEquals((new         MohawkDataType(true)).toString(),
testOperator.notEquals(float1, string1).toString());
      }

      public void testGreaterThan() {
            assertEquals((new        MohawkDataType(false)).toString(),
testOperator.greaterThan(int1, int2).toString());
            assertEquals((new        MohawkDataType(false)).toString(),
testOperator.greaterThan(int1, int1).toString());
            assertEquals((new        MohawkDataType(false)).toString(),
testOperator.greaterThan(float1, float2).toString());
            assertEquals((new        MohawkDataType(false)).toString(),
testOperator.greaterThan(float1, float1).toString());
            assertEquals((new        MohawkDataType(false)).toString(),
testOperator.greaterThan(string1, string2).toString());
            assertEquals((new        MohawkDataType(false)).toString(),
testOperator.greaterThan(string1, string1).toString());
            assertEquals((new        MohawkDataType(false)).toString(),
testOperator.greaterThan(int1, float1).toString());
            assertEquals((new        MohawkDataType(false)).toString(),
testOperator.greaterThan(int1, string1).toString());
            assertEquals((new        MohawkDataType(false)).toString(),
testOperator.greaterThan(float1, string1).toString());
      }

      public void testGreaterThanOrEquals() {
            assertEquals((new        MohawkDataType(false)).toString(),
testOperator.greaterThanOrEquals(int1, int2).toString());
            assertEquals((new         MohawkDataType(true)).toString(),
testOperator.greaterThanOrEquals(int1, int1).toString());
            assertEquals((new        MohawkDataType(false)).toString(),
testOperator.greaterThanOrEquals(float1, float2).toString());
            assertEquals((new         MohawkDataType(true)).toString(),
testOperator.greaterThanOrEquals(float1, float1).toString());
            assertEquals((new        MohawkDataType(false)).toString(),
testOperator.greaterThanOrEquals(string1, string2).toString());
            assertEquals((new         MohawkDataType(true)).toString(),
testOperator.greaterThanOrEquals(string1, string1).toString());
            assertEquals((new         MohawkDataType(true)).toString(),
testOperator.greaterThanOrEquals(int1, float1).toString());
            assertEquals((new        MohawkDataType(false)).toString(),
testOperator.greaterThanOrEquals(int1, string1).toString());
            assertEquals((new        MohawkDataType(false)).toString(),
testOperator.greaterThanOrEquals(float1, string1).toString());
      }

      public void testLessThan() {
            assertEquals((new         MohawkDataType(true)).toString(),
testOperator.lessThan(int1, int2).toString());
```

```
            assertEquals((new       MohawkDataType(false)).toString(),
testOperator.lessThan(int1, int1).toString());
            assertEquals((new        MohawkDataType(true)).toString(),
testOperator.lessThan(float1, float2).toString());
            assertEquals((new       MohawkDataType(false)).toString(),
testOperator.lessThan(float1, float1).toString());
            assertEquals((new        MohawkDataType(true)).toString(),
testOperator.lessThan(string1, string2).toString());
            assertEquals((new       MohawkDataType(false)).toString(),
testOperator.lessThan(string1, string1).toString());
            assertEquals((new       MohawkDataType(false)).toString(),
testOperator.lessThan(int1, float1).toString());
            assertEquals((new        MohawkDataType(true)).toString(),
testOperator.lessThan(int1, string1).toString());
            assertEquals((new        MohawkDataType(true)).toString(),
testOperator.lessThan(float1, string1).toString());
      }

      public void testLessThanOrEquals() {
            assertEquals((new        MohawkDataType(true)).toString(),
testOperator.lessThanOrEquals(int1, int2).toString());
            assertEquals((new        MohawkDataType(true)).toString(),
testOperator.lessThanOrEquals(int1, int1).toString());
            assertEquals((new        MohawkDataType(true)).toString(),
testOperator.lessThanOrEquals(float1, float2).toString());
            assertEquals((new        MohawkDataType(true)).toString(),
testOperator.lessThanOrEquals(float1, float1).toString());
            assertEquals((new        MohawkDataType(true)).toString(),
testOperator.lessThanOrEquals(string1, string2).toString());
            assertEquals((new        MohawkDataType(true)).toString(),
testOperator.lessThanOrEquals(string1, string1).toString());
            assertEquals((new        MohawkDataType(true)).toString(),
testOperator.lessThanOrEquals(int1, float1).toString());
            assertEquals((new        MohawkDataType(true)).toString(),
testOperator.lessThanOrEquals(int1, string1).toString());
            assertEquals((new        MohawkDataType(true)).toString(),
testOperator.lessThanOrEquals(float1, string1).toString());
      }

      public void testMatch() {
            assertEquals((new       MohawkDataType(false)).toString(),
testOperator.match(int1, int2).toString());
            assertEquals((new        MohawkDataType(true)).toString(),
testOperator.match(int1, int1).toString());
            assertEquals((new       MohawkDataType(false)).toString(),
testOperator.match(float1, float2).toString());
            assertEquals((new        MohawkDataType(true)).toString(),
testOperator.match(float1, float1).toString());
            assertEquals((new       MohawkDataType(false)).toString(),
testOperator.match(string1, string2).toString());
            assertEquals((new        MohawkDataType(true)).toString(),
testOperator.match(string1, string1).toString());
            assertEquals((new       MohawkDataType(false)).toString(),
testOperator.match(int1, float1).toString());
            assertEquals((new       MohawkDataType(false)).toString(),
testOperator.match(int1, string1).toString());
```

```
            assertEquals((new       MohawkDataType(false)).toString(),
testOperator.match(float1, string1).toString());
      }

      public void testOr() {
            assertEquals((new        MohawkDataType(true)).toString(),
testOperator.or(int1, int2).toString());
            assertEquals((new        MohawkDataType(true)).toString(),
testOperator.or(int1, int1).toString());
            assertEquals((new        MohawkDataType(true)).toString(),
testOperator.or(float1, float2).toString());
            assertEquals((new        MohawkDataType(true)).toString(),
testOperator.or(float1, float1).toString());
            assertEquals((new        MohawkDataType(true)).toString(),
testOperator.or(string1, string2).toString());
            assertEquals((new        MohawkDataType(true)).toString(),
testOperator.or(string1, string1).toString());
            assertEquals((new        MohawkDataType(true)).toString(),
testOperator.or(int1, float1).toString());
            assertEquals((new        MohawkDataType(true)).toString(),
testOperator.or(int1, string1).toString());
            assertEquals((new        MohawkDataType(true)).toString(),
testOperator.or(float1, string1).toString());
      }

      public void testAnd() {
            assertEquals((new        MohawkDataType(true)).toString(),
testOperator.and(int1, int2).toString());
            assertEquals((new        MohawkDataType(true)).toString(),
testOperator.and(int1, int1).toString());
            assertEquals((new        MohawkDataType(true)).toString(),
testOperator.and(float1, float2).toString());
            assertEquals((new        MohawkDataType(true)).toString(),
testOperator.and(float1, float1).toString());
            assertEquals((new        MohawkDataType(true)).toString(),
testOperator.and(string1, string2).toString());
            assertEquals((new        MohawkDataType(true)).toString(),
testOperator.and(string1, string1).toString());
            assertEquals((new        MohawkDataType(true)).toString(),
testOperator.and(int1, float1).toString());
            assertEquals((new        MohawkDataType(true)).toString(),
testOperator.and(int1, string1).toString());
            assertEquals((new        MohawkDataType(true)).toString(),
testOperator.and(float1, string1).toString());
      }

      public void testNot() {
            assertEquals((new       MohawkDataType(false)).toString(),
testOperator.not(int1).toString());
            assertEquals((new       MohawkDataType(false)).toString(),
testOperator.not(float1).toString());
            assertEquals((new       MohawkDataType(false)).toString(),
testOperator.not(string1).toString());
            assertEquals((new       MohawkDataType(false)).toString(),
testOperator.not(boolTrue).toString());
            assertEquals((new        MohawkDataType(true)).toString(),
testOperator.not(boolFalse).toString());
```

```
	}

	public void testMohawkLogical() {
	}

	protected void setUp() {
		testOperator = new MohawkLogicalOperator();
		int1 = new MohawkDataType(2);
		int2 = new MohawkDataType(6);
		float1 = new MohawkDataType(2.0f);
		float2 = new MohawkDataType(6.0f);
		string1 = new MohawkDataType("cat");
		string2 = new MohawkDataType("dog");
		boolTrue = new MohawkDataType(true);
		boolFalse = new MohawkDataType(false);

	}

	protected void tearDown() {
	}

	public MohawkLogicalTest (String value) {
		super.setName(value);
	}
}
```

8.4.3 MohawkMathTest.java

```
//package MohawkTest;

import java.util.*;
import junit.framework.*;

public class MohawkMathTest extends TestCase {
      public MohawkDataType int1, int2, float1, float2, string1,
string2;
      public MohawkMathOperator testOperator;
      public TestCase test;

      public void testMohawkMath() {
      }

       public void runTest() {
            testAdd();
            testSubtract();
            testMultiply();
            testDivide();
            testMod();
            testExp();
       }

      public void testAdd() {
            assertEquals((new MohawkDataType(8)).toString(),
testOperator.add(int1, int2).toString());
```

```
            assertEquals((new MohawkDataType(4.0f)).toString(),
testOperator.add(int1, float1).toString());
            assertEquals((new MohawkDataType(8.0f)).toString(),
testOperator.add(float1, float2).toString());
            assertEquals((new MohawkDataType("2.0cat")).toString(),
testOperator.add(float1, string1).toString());
            assertEquals((new MohawkDataType("catdog")).toString(),
testOperator.add(string1, string2).toString());
            assertEquals((new MohawkDataType("cat2")).toString(),
testOperator.add(string1, int1).toString());
      }

      public void testSubtract() {
            assertEquals((new MohawkDataType(-4)).toString(),
testOperator.subtract(int1, int2).toString());
            assertEquals((new MohawkDataType(0.0f)).toString(),
testOperator.subtract(int1, float1).toString());
            assertEquals((new MohawkDataType(-4.0f)).toString(),
testOperator.subtract(float1, float2).toString());
            assertEquals((new MohawkDataType(2.0f)).toString(),
testOperator.subtract(float1, string1).toString());
            assertEquals((new MohawkDataType(0)).toString(),
testOperator.subtract(string1, string2).toString());
            assertEquals((new MohawkDataType(-2)).toString(),
testOperator.subtract(string1, int1).toString());
      }

      public void testMultiply() {
            assertEquals((new MohawkDataType(12)).toString(),
testOperator.multiply(int1, int2).toString());
            assertEquals((new MohawkDataType(4.0f)).toString(),
testOperator.multiply(int1, float1).toString());
            assertEquals((new MohawkDataType(12.0f)).toString(),
testOperator.multiply(float1, float2).toString());
            assertEquals((new MohawkDataType(0)).toString(),
testOperator.multiply(float1, string1).toString());
            assertEquals((new MohawkDataType(0)).toString(),
testOperator.multiply(string1, string2).toString());
            assertEquals((new MohawkDataType(0)).toString(),
testOperator.multiply(string1, int1).toString());
      }

      public void testDivide() {
            assertEquals((new MohawkDataType(0)).toString(),
testOperator.divide(int1, int2).toString());
            //assertEquals((new MohawkDataType("1")).toString(),
testOperator.divide(int1, float1).toString());
           assertEquals((new  MohawkDataType(0.3333333333f)).toString(),
testOperator.divide(float1, float2).toString());
            assertEquals((new
MohawkDataType(Float.POSITIVE_INFINITY)).toString(),
testOperator.divide(float1, string1).toString());
            assertEquals((new
MohawkDataType(Float.POSITIVE_INFINITY)).toString(),
testOperator.divide(string1, string2).toString());
            assertEquals((new MohawkDataType(0)).toString(),
testOperator.divide(string1, int1).toString());
```

```
	}
	public void testMod() {
		assertEquals((new MohawkDataType(2)).toString(),
testOperator.mod(int1, int2).toString());
		assertEquals((new MohawkDataType(0.0f)).toString(),
testOperator.mod(int1, float1).toString());
		assertEquals((new MohawkDataType(2.0f)).toString(),
testOperator.mod(float1, float2).toString());
		assertEquals((new
MohawkDataType(Float.POSITIVE_INFINITY)).toString(),
testOperator.mod(float1, string1).toString());
		assertEquals((new
MohawkDataType(Float.POSITIVE_INFINITY)).toString(),
testOperator.mod(string1, string2).toString());
		assertEquals((new MohawkDataType(0)).toString(),
testOperator.mod(string1, int1).toString());
	}
	public void testExp() {
		assertEquals((new MohawkDataType(64)).toString(),
testOperator.exp(int1, int2).toString());
		assertEquals((new MohawkDataType(4)).toString(),
testOperator.exp(int1, float1).toString());
		assertEquals((new MohawkDataType(64.0f)).toString(),
testOperator.exp(float1, float2).toString());
		//assertEquals((new MohawkDataType(1)).toString(),
testOperator.exp(float1, string1).toString());
		assertEquals((new MohawkDataType(1)).toString(),
testOperator.exp(string1, string2).toString());
		assertEquals((new MohawkDataType(0)).toString(),
testOperator.exp(string1, int1).toString());
	}

	protected void setUp() {
		testOperator = new MohawkMathOperator();
		int1 = new MohawkDataType(2);
		int2 = new MohawkDataType(6);
		float1 = new MohawkDataType(2.0f);
		float2 = new MohawkDataType(6.0f);
		string1 = new MohawkDataType("cat");
		string2 = new MohawkDataType("dog");
	}

	protected void tearDown() {
	}

	public MohawkMathTest (String value) {
		super.setName(value);
	}
}
```

8.4.4 MohawkStringTest.java

```
//package MohawkTest;

import java.util.*;
import junit.framework.*;
```

```
public class MohawkStringTest extends TestCase {
      public MohawkDataType int1, int2, float1, float2, string1,
string2;
      public MohawkStringOperator testOperator;
      public TestCase test;

      public void testMohawkString() {
      }

       public void runTest() {
            testConcatenate();
       }

      public void testConcatenate() {
            assertEquals((new MohawkDataType(26)).toString(),
testOperator.concatenate(int1, int2).toString());
            assertEquals((new MohawkDataType(22.0f)).toString(),
testOperator.concatenate(int1, float1).toString());
            assertEquals((new MohawkDataType("2.06.0")).toString(),
testOperator.concatenate(float1, float2).toString());
            assertEquals((new MohawkDataType("2.0cat")).toString(),
testOperator.concatenate(float1, string1).toString());
            assertEquals((new MohawkDataType("catdog")).toString(),
testOperator.concatenate(string1, string2).toString());
            assertEquals((new MohawkDataType("cat2")).toString(),
testOperator.concatenate(string1, int1).toString());
      }

       protected void setUp() {
            testOperator = new MohawkStringOperator();
              int1 = new MohawkDataType(2);
              int2 = new MohawkDataType(6);
              float1 = new MohawkDataType(2.0f);
              float2 = new MohawkDataType(6.0f);
            string1 = new MohawkDataType("cat");
            string2 = new MohawkDataType("dog");
       }

       protected void tearDown() {
       }

       public MohawkStringTest (String value) {
              super.setName(value);
       }
}
```

8.4.5 MohawkRegexTest.java

```
import java.util.*;
import junit.framework.*;

public class MohawkRegexTest extends TestCase {
      public MohawkDataType string1, string2;
      public MohawkLogicalOperator testOperator;
      public TestCase test;

      public void testMohawkRegex() {
```

```
	}

	 public void runTest() {
		testConcatenate();
	 }

	public void testConcatenate() {
		assertEquals((new MohawkDataType("true")).toString(),
	MohawkLogicalOperator.match(string1, string1).toString());
		assertEquals((new MohawkDataType("true")).toString(),
	MohawkLogicalOperator.match(string1, string2).toString());
		assertEquals((new MohawkDataType("true")).toString(),
	MohawkLogicalOperator.match(string1, string3).toString());
		assertEquals((new MohawkDataType("true")).toString(),
	MohawkLogicalOperator.match(string1, string3).toString());

	}

	 protected void setUp() {
		string1 = new MohawkDataType("cat");
		string2 = new MohawkDataType("[c]at");
		string3 = new MohawkDataType("ca[^ ]");
		string4 = new MohawkDataType("[a-z_0-9A-Z]at");
	 }

	 protected void tearDown() {
	 }

	 public MohawkRegexTest (String value) {
		super.setName(value);
	 }
}
```

Parser Test Code

8.4.6 MohawkTestParser.java

```
import java.io.*;
import antlr.CommonAST;
import java.util.LinkedList;

/**
 * To do:
 * separate out logic for naming field vars in the begin block
 * logic for programs w/o begin blocks, w/o end blocks, etc.
 **/

public class MohawkTestParser {

   private static String StatementExpected =
" -:-| { ( BEGIN_ACTION ( = a 1 ) ( = b 2 ) ) } { ( ACTION ( = a 1 )
( = b 2 ) ( = a ( + a b ) ) ( :-O a ) ( = a ( - a b ) ) ( :-O a ) ( = a
( * a b ) ) ( :-O a ) ( = a ( / a b ) ) ( :-O a ) ( = a ( ^ a b ) )
( :-O a ) ( = a ( % a b ) ) ( :-O a ) ( -- a ) ( :-O a ) ( ++ b ) ) }
|-:- { ( ACTION ( :-O b ) ) }";
```

```
   private static String ConditionalExpected =
" -:-| { BEGIN_ACTION } ( >-) myfunc ( VAR_LIST a ) ( :-O ( + a 1 ) ) )
{ ( ACTION ( = a 2 ) ( = b 1 ) ( :-/ ( > a b ) ( :-O Correct ) ( :-O
Incorrect ) ) ( :-/ ( >= a b ) ( :-O Correct ) ( :-O Incorrect ) ) ( =
a 2 ) ( = b 2 ) ( :-/ ( >= a b ) ( :-O Correct ) ( :-O Incorrect ) )
( = a 1 ) ( = b 2 ) ( :-/ ( < a b ) ( :-O Correct ) ( :-O Incorrect ) )
( :-/ ( <= a b ) ( :-O Correct ) ( :-O Incorrect ) ) ( = a 2 ) ( = b 2 )
( :-/ ( <= a b ) ( :-O Correct ) ( :-O Incorrect ) ) ( = a 5 ) ( = b 5 )
( :-/ ( == a b ) ( :-O Correct ) ( :-O Incorrect ) ) ( :-/ ( != a b )
( :-O Incorrect ) ( :-O Correct ) ) ( = a 0 ) ( = b 0 ) ( :-/ ( || a b )
( :-O Incorrect ) ( :-O Correct ) ) ( = a 0 ) ( = b 1 ) ( :-/ ( || a b )
( :-O Correct ) ( :-O Incorrect ) ) ( = a 1 ) ( = b 1 ) ( :-/ ( || a b )
( :-O Correct ) ( :-O Incorrect ) ) ( = a 1 ) ( = b 1 ) ( :-/ ( && a b )
( :-O Correct ) ( :-O Incorrect ) ) ( = a 1 ) ( = b 0 ) ( :-/ ( && a b )
( :-O Incorrect ) ( :-O Correct ) ) ( = a 1 ) ( :-/ ( ~ ( == a a ) )
( :-O Incorrect ) ( :-O Correct ) ) ) } |-:- { ACTION }";

   private static String LoopExpected =
" -:-| { ( BEGIN_ACTION ( = counter 0 ) ) } { ( ACTION ( :^O ( =
counter 0 ) ( < counter 10 ) ( ++ counter ) ( STMT ( = counter
counter ) ) ) ( :^~ ( > counter 0 ) ( STMT ( -- counter ) ) ) ( >^O
( STMT ( ++ counter ) ) ( < counter 10 ) ) ( = counter 0 ) ( = inner 0 )
( :^~ ( < counter 5 ) ( STMT ( :^O ( = inner 0 ) ( < inner 3 ) ( ++
inner ) ( STMT ( :-O counter ) ) ) ( ++ counter ) ) ) ) } |-:-
{ ( ACTION ( = counter counter ) ) }";

   private static String ControlExpected =
" -:-| { BEGIN_ACTION } { ( ACTION ( = counter 0 ) ( :^~ ( < counter
100 ) ( STMT ( ++ counter ) ( :-/ ( > counter 10 ) ( STMT :-( ) ) ( :-O
counter ) ) ) ( = counter 0 ) ( :^~ ( < counter 10 ) ( STMT ( ++
counter ) ( :-/ ( % counter 2 ) ( STMT :-> ) ) ( :-O counter ) ) ) ( =
counter 0 ) ( :^~ ( < counter 10 ) ( STMT ( ++ counter ) ( :-/ ( %
counter 2 ) ( STMT :-| ) ) ( :-O counter ) ) ) :^P ) } |-:- { ACTION }";

   private static String FunctionExpected =
" -:-| { ( BEGIN_ACTION ( :-O Start ) ( = b 0 ) ) } ( >-) myfunc1
( VAR_LIST a ) ( :-O ( + a 1 ) ) ) ( >-) myfunc2 ( VAR_LIST b ) ( :-O
b ) ) { ( ACTION ( :-O New ) ( FUNC_CALL myfunc1 ( EXPR_LIST 10 ) )
( FUNC_CALL myfunc2 ( EXPR_LIST 5 ) ) ) } |-:- { ( ACTION ( :-O
Finish ) ) }";

   public static void main(String[] args) {
         runTest("Statement", "StatementTest.oi", StatementExpected);
         runTest("Conditional", "ConditionalTest.oi",
ConditionalExpected);
         runTest("Loop", "LoopTest.oi", LoopExpected);
         runTest("Control", "ControlTest.oi", ControlExpected);
         runTest("Function", "FunctionTest.oi", FunctionExpected);

         System.out.println();
         System.err.println("Parser Test DONE!");
   }

   public static void runTest (String testname, String filename, String
expected) {
      try {
          // find the source code for the program
```

```
            MohawkLexer lexer;
            lexer = new MohawkLexer(new FileInputStream(filename));

             // parse the code
             MohawkParser parser = new MohawkParser(lexer);
            parser.setASTNodeType("CommonASTWithLines");
             parser.program();

             CommonAST tree = (CommonAST)parser.getAST();
             MohawkWalker walker = new MohawkWalker();
            tree.toStringList();

             if (!expected.equals(tree.toStringList()))
               System.out.println(testname + " Test FAILED!");
            else
               System.out.println(testname + " Test PASSED!");

         } catch(Exception e) {
             System.err.println("oi! oi! oi! : " + e);
            e.printStackTrace();
         }
      }

}
```

8.4.7 ConditionalTest.oi

```
-:-|
{
}

>-) myfunc(a) { :-O(a + 1)! }

{
       /*
GT : ">" ;
LT : "<" ;
GE : ">=" ;
LE : "<=" ;
EQ : "==";
NEQ : "!=" ;
OR : "||";
AND : "&&";
NOT : "~";
       */

       a = 2!
       b = 1!
       :-/ (a > b)
             :-O("Correct")!
       :-\
             :-O("Incorrect")!

       :-/ (a >= b)
             :-O("Correct")!
       :-\
             :-O("Incorrect")!
```

```
a = 2!
b = 2!
:-/ (a >= b)
      :-O("Correct")!
:-\
      :-O("Incorrect")!

a = 1!
b = 2!
:-/ (a < b)
      :-O("Correct")!
:-\
      :-O("Incorrect")!


:-/ (a <= b)
      :-O("Correct")!
:-\
      :-O("Incorrect")!

a = 2!
b = 2!
:-/ (a <= b)
      :-O("Correct")!
:-\
      :-O("Incorrect")!

a = 5!
b = 5!

:-/ (a == b)
      :-O("Correct")!
:-\
      :-O("Incorrect")!

:-/ (a != b)
      :-O("Incorrect")!
:-\
      :-O("Correct")!

a = 0!
b = 0!
:-/ (a || b)
      :-O("Incorrect")!
:-\
      :-O("Correct")!

a = 0!
b = 1!
:-/ (a || b)
      :-O("Correct")!
:-\
      :-O("Incorrect")!

a = 1!
b = 1!
```

```
      :-/ (a || b)
            :-O("Correct")!
      :-\
            :-O("Incorrect")!

      a = 1!
      b = 1!
      :-/ (a && b)
            :-O("Correct")!
      :-\
            :-O("Incorrect")!

      a = 1!
      b = 0!
      :-/ (a && b)
            :-O("Incorrect")!
      :-\
            :-O("Correct")!

      a = 1!
      :-/ (~a == a)
            :-O("Incorrect")!
      :-\
            :-O("Correct")!
}

|-:-
{
}
```

8.4.8 FunctionTest.oi

```
-:-|
{
  :-O("Start")!
  b = 0!
}

>-) myfunc1(a) { :-O(a + 1)! }

>-) myfunc2(b) {
      :-O(b)!
      myfunc1(50)!
}

{
  :-O("New")!
  myfunc1(10)!
  myfunc2(5)!
}

|-:-
{
  :-O("Finish")!
}
```

8.4.9 StatementTest.oi

```
-:-|
{
  a = 1!
  b = 2!
}

{
      a = 1!
      b = 2!
      a = a + b!
      :-O(a)!
      a = a - b!
      :-O(a)!
      a = a * b!
      :-O(a)!
      a = a / b!
      :-O(a)!

      a = a ^ b!
      :-O(a)!
      a = a % b!
      :-O(a)!

      a--!
      :-O(a)!
      b++!
}

|-:-
{
      :-O(b)!
}
```

8.4.10 ControlTest.oi

```
-:-|
{
}

{
      counter = 0!
      // Break Test
      :^~ (counter < 100)
      {
            counter++!
            :-/ (counter > 10) {
                  :-(!
            }
            :-O(counter)!
      }

      counter = 0!
      // Continue Test
      :^~ (counter < 10)
```

```
      {
            counter++!
            :-/ (counter % 2) {
                  :->!
            }
            :-O(counter)!
      }

      // Next Test
      counter = 0!
      :^~ (counter < 10)
      {
            counter++!
            :-/ (counter % 2) {
                  :-|!
            }
            :-O(counter)!
      }
      // Exit Test
      :^P!
}

|-:-
{
}
```

8.4.11 LoopTest.oi

```
-:-|
{
  counter = 0!
}

{
      // For loop
      :^O(counter = 0; counter < 10; counter++ )
      {
            counter=counter!
      }

      // While loop
      :^~ (counter > 0)
      {
            counter--!
      }

      // DoWhile loop
      >^O
      {
            counter++!
      } :^~ (counter < 10)

      // Nested loop
      counter = 0!
      inner = 0!
      :^~ (counter < 5)
      {
```

```
            :^O(inner=0; inner < 3; inner++ )
            {
                  :-O(counter)!
            }
            counter++!
      }

}

|-:-
{
      counter = counter!
}
```

Functional Test Code

8.4.12 MohawkTestProgram.java

```
import java.io.*;
import antlr.CommonAST;
import java.util.LinkedList;


public class MohawkTestProgram {

   public static void main(String[] args) {
runTest("Statement", "StatementTest.oi", StatementExpected);
         runTest("Conditional", "ConditionalTest.oi",
         ConditionalExpected);
         runTest("Loop", "LoopTest.oi", LoopExpected);
         runTest("Control", "ControlTest.oi", ControlExpected);
         runTest("Function", "FunctionTest.oi", FunctionExpected);


         System.out.println("Program Test DONE!");

   }

   public static void runTest (String testname, String filename, String
expected) {
      try {
            // Running Program to Test
            String s = "java -classpath ../../../lib/antlr-
            2.7.5.jar:../:../../: MohawkMain " + filename + " > " +
            filename + ".output";

            Runtime rt = Runtime.getRuntime();
            Process p = rt.exec(s);
      } catch(Exception e) {
          System.err.println("oi! oi! oi! : " + e);
         e.printStackTrace();
      }
   }

}
```

8.4.13 ConditionalTest.oi

Refer to ConditionalTest.oi in section 8.4

8.4.14 LoopTest.oi

Refer to LoopTest.oi in section 8.4

8.4.15 ControlTest.oi

Refer to ControlTest.oi in section 8.4

8.4.16 FunctionTest.oi

Refer to FunctionTest.oi in section 8.4

8.5 Mohawk Tutorial Code

8.5.1 Euclid.oi

```
-:-|
{
 max = 52 - -(-32)! //negation
}

>-) euclid(num1, num2) //function definition
{
   :-/ (num1 < num2)
   {
      euclid(num2, num1)! //recursion
   }
   :-\ :-/ (num1 % num2 == 0) //else, if
   {
        :-o(num2+" ")!  //print
   }
   :-\
   {
      euclid(num2, num1%num2)!
   }
}

(true)
{
 :^O(i=0; i < max; i+=4 ) //for
  {
    :-/ (i%3==0)  //if
    :->!          //continue
    euclid(i, $1)!      //function call
  }
 :-O()!           //print line
}

|-:-
{
```

```
  :-O("Done!")!
}
```

8.5.2 Total.oi

```
-:-|
{
totalwages=0!
}

{
    totalhours = $2 + $3!
    wage -> $4!   //linking
    total = totalhours * wage!
    totalwages += total!
    :-o($1 + ": " + $2 +" " + $3 + " ")!
    :-o(wage + " ")!
    :-o(" total "+ total)!
    :-O()!
}

|-:-
{
:-O("TOTAL WAGES: " + totalwages)!
:-O("AVER " + totalwages/#R)!
}
```