Hog Language Reference

Jason Halpern
Samuel Messing
Benjamin Rapaport
Kurry Tran
Paul Tylkin

March 19, 2012

1 Introduction

As data sets have grown in size, so have the complexities of dealing with them. For instance, consider wanting to generate counts for all the words in *War and Peace* by means of distributed computation. Writing in Java and using Hadoop MapReduce (TM), a simple solution takes over 50 lines of code, as the programmer is required to specify intermediate objects not directly related to the desired computation, but required simply to get Hadoop to function properly. Our goal is to produce a language that can express the same computation in about 10 lines.

1.1 The MapReduce Framework

With the explosion in the size of datasets that companies have had to manage in recent years there are many new challenges that they face. Many companies and organizations have to handle the processing of datasets that are terabytes or even petabytes in size. The first challenge in this large-scale processing is how to make sense of all this data. More importantly, how can they process and manipulate the data in a time efficient and reliable manner. The second challenge is how they handle this across their distributed systems. Writing distributed, fault tolerant programs requires a high level of expertise and knowledge of parallel systems.

This was an obvious challenge to the company that has to process more data than any company on earth, Google. In response to this need, a group of engineers at Google developed their MapReduce framework in 2004. This high-level framework could be used for of a variety of tasks, including handling search queries, indexing crawled documents and processing logs. The software framework was developed to handle computations on massive datasets that are distributed across hundreds or even thousands of machines. The motivation behind MapReduce was to create a unified framework that abstracted away many of the low level details from programmers, so they would not have to be concerned with how the data is distributed, how the computation is parallelized and how all this is done in a fault tolerant manner. MapReduce provides fault tolerance in software rather than in hardware. MapReduce can handle both unstructured data (files) and structured data (databases), but is predominantly used with files.

The framework partitions the data across different machines, so that the computations are initially performed on smaller sets of data distributed across the cluster. Each cluster has a master node that is responsible for coordinating the efforts among the slave nodes. Each slave node sends periodic heartbeats to the master node so it can be aware of progress and failure. In the case of failure, the master node can reassign tasks to other nodes in the cluster. In conjunction with the underlying MapReduce framework created at Google, the company also had to build the distributed Google File System (GFS). This file system "allows programs to access files efficiently from any computer, so functions can be mapped everywhere." GFS was designed with the same goals

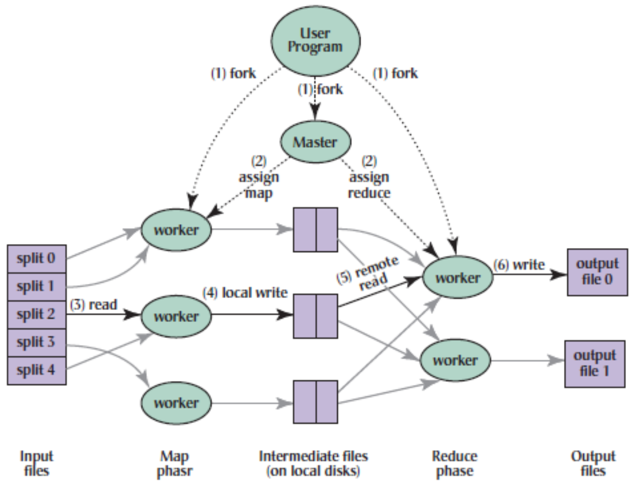

Figure 1: Overview of the Map Reduce program

as other distributed file systems, including "performance, scalability, reliability and availability." Another key aspect of the GFS design is fault tolerance and this is achieved by treating failures as normal and optimizing for "huge files that are mostly appended to and then read."

Within the framework, a programmer is responsible for writing both Map and Reduce functions. The map function is applied to all of the input data "in order to compute a set of intermediate key/value pairs." In the map step, the master node partitions the input data into smaller problems and distributes them across the worker nodes in the cluster. This step is applied in parallel to all of the input that has been partitioned across the cluster. Then, the reduce step is responsible for collecting all the processed data from the slave nodes and formatting the output. The reduce function is carried out over all the values that have the same key such that each key has a single value. which is the answer to the problem MapReduce is trying to solve. The output is done to files in the distributed file system.

The use of "a functional model with user-specified map and reduce operations allows (Google) to parallelize large computations easily and to use re-execution as the primary mechanism for fault tolerance." A programmer only has to specify the functions described above and the system handles the rest of the details. Figure 1.1 illustrates the execution flow of a MapReduce program.

1.2 The Hog Language

Hog is a **data-oriented**, **high-level**, scripting language for creating MapReduce programs. Used alongside Hadoop, Hog enables users to efficiently carry out **distributed** computation. Hadoop MapReduce is an open-source framework for carrying out distributed computation, which is especially useful for working with large data sets. While it is possible to write code to carry out computations with Hadoop directly, the framework requires users to specify low-level details that are often irrelevant to their desired goal.

By building a scripting language on top of Hadoop, we aim to simplify the process. Built around a **simple** and highly **readable** syntax, Hog will let users focus on what computations they want done, and not how they want to do them. Hog takes care of all the low-level details required to run computations on Hadoops distributed network. All a user needs to do is tell Hog the location of their valid Hadoop instance, and Hog will do the rest.

We intentionally have restricted the scope of Hog to deal with specific problems. For example, Hog's collection objects can only contain primitive types (preventing such data structures as lists of lists). Also, Hog only supports reading and writing plaintext files. While these limitations sacrifice the generality of the language, they promote ease of use.

1.2.1 Guiding Principles

The guiding principles of Hog are:

- Anyone can MapReduce
- Brevity over verbosity
- Simplicity over complexity

1.3 The (Ideal) Hog User

Hog was designed with a particular user in mind: one that has already learned the basics of programming in a different programming language (such as Python or Java), but is inexperienced enough to benefit from a highly structured framework for writing MapReduce programs. The language was designed with the goal of making learning how to write MapReduce programs as easy as possible. **@Paul: flesh out if needed**.

2 Program Structure

2.1 Overall Structure

Every Hog program consists of a single source file with a .hog extension. This source file must contain three sections: `@Map`, and `@Reduce`, and `@Main` and can also include an optional `@Functions` section. These sections must be included in the following order:

```
@Functions {
  .
```

```
  .
  .
}
@Map <type signature> {
  .
  .
  .
}
@Reduce <type signature>
  .
  .
  .
@Main {
  .
  .
  .
}
```

2.2 @Functions

At the top of every Hog program, the programmer has the option to define functions in a section called `@Functions`. Any function defined in this section can be called from any other section of the program, including `@Map`, and `@Reduce`, and can also be called from other functions defined in the `@Functions` section. The section containing the functions begins with the keyword `@Functions` on its own line, followed by the function definitions.

Function definitions have the form:

```
@Functions
return-type function-name(parameter-list) {
  exprlist
}
```

The return-type can be any valid Hog type. The rules regarding legal function-names are identical to those regarding legal variable identifiers. Each parameter in the parameter-list consists of a valid Hog type followed by the name of the parameter, which must also follow the naming rules for identifiers. Parameters in the parameter-list are separated by commas. The @Functions section ends when the next Hog section begins.

A complete example of an @Functions section:

```
@Functions
int min(int a, int b) {
  if (a < b) {
    return a
  } else {
    return b
```

```
  }
}

list<int> reverseList(list<int> oldList) {
  list<int> newList()
  for (int i = oldList.len()-1; i >= 0; i--) {
    newList.append(oldList.get(i))
  }
  return newList
}
```

Function names can be overloaded as long as the function definitions have different signatures (i.e. parameter lists different in types and/or length). Additionally, user-defined functions can make reference to other user-defined functions.

2.3 `@Map`

The map function in a MapReduce program takes as input key-value pairs, performs the appropriate calculations and procedures, and emits intermediate key-value pairs as output. Any given input pair may map to zero, one, or multiple output pairs. The `@map` section defines the code for the map function.

The `@map` header must be followed by the signature of the map function, and then the body of the map function as follows:

```
@Map (type identifier, type identifier) -> (type, type) {
  .
  .
  .
}
```

The first `type identifier` defines the ***key*** and the second defines the ***value*** of the input key-value pair to the `@map` function. The identifiers specified for the key and value can be made reference to later within the `@map` block. The `@map` signature is followed by an arrow and another key-value pair, defining the types of the output of the map function. Notice that identifiers are not specified for the output key and value (said to be *unnamed*), as these pairs are only produced at the end of the map function. Also note that any code written in the same scope after an `emit()` call will be ***unreachable***, and will cause a compile-time `UnreachableCodeException`.

The map function can include any number of calls to `emit()`, which outputs the resulting intermediate key-value pairs for use by the function defined in the `@reduce` section. The types of the values passed to the `emit()` function must agree with the signature of the output key-value pair as defined in the `@map` type signature. All output pairs from the map function are subsequently grouped by key by the framework, and passed as input to the `@reduce` function.

Currently, the only configuration available is for a file to be passed into the map function one line at a time, with the line of text being the value, and the corresponding line number as the key. This requires that the input key/value pair to the map function is of type `(int keyname,text valuename)`. Extending this to allow for other input formats is a future goal of the Hog language.

The following is an example of a complete `@Map` section for a program that counts the number of times each word appears in a set of files. The map function receives a single line of text, and for each word in the line (as delineated by whitespace), it emits the word as the key with a value of one. By emitting the word as the key, we can allow the framework to group by the word, thus calling the reduce function for every word.

2.4 `@Reduce`

The reduce function in a MapReduce program takes a list of values that share the same key, as emitted by the map function, and outputs a smaller set of values to be associated with another key. The input and output keys do not have to match, though they often do.

The setup for the reduce section is similar to the map section. However, the input value for any reduce function is always an iterator over the list of values associated with its key. The type of the key must be the same as the type of the key emitted by the map function. The iterator must be an iterator over the type of the values emitted by the map function.

```
@Reduce (type identifier, iter<type> identifier) -> (type, type) {
  .
  .
  .
}
```

As with the map function, the reduce function can emit as many key/value pairs as the user would like. Any key/value pair emitted by the reduce function is recorded in the output file.

Below is a sample at `@Reduce` section, which continues the word count example, and follows the @mapsample introduce in the previous section.

2.5 `@Main`

@BEN: Fill this in! Hog.mapReduce()

3 Lexical Conventions

3.1 Tokens

The classes of tokens include the following: identifiers, keywords, constants, string literals, operators and separators. Blanks, tabs, newlines and comments are ignored. If the input is separated into tokens up to a given character, the next token is the longest string of characters that could represent a token.

3.2 Comments

Multi-line comments are identified by the enclosing character sequences `#{` and `}#`. Anything within these enclosing characters is considered a comment, and is completely ignored by the compiler. For example,

```
int i = 0
#{ these are block
    comments and are ignored
     by the compiler }#
i++
```

In the above example, the text `these are block comments \n comments and are ignored \n by the complier` are completely ignored during compilation. Compilation goes directly from the line `int i = 0` to the line `i++`.

Single-line comments are defined to be strings of text included between a '`#`' symbol on the left-hand side an a newline character ('`\n`') on the right-hand side.

3.3 Identifiers

A valid identifier consists of sequence on contiguous leters, digits, or underscores. The first character of an identifier must not be a digit. The identifiers are case sensitive so foo is not the same identifier as Foo.

@kurry: explain what a valid identifier is

3.4 Keywords

The reserved words of Hog are a superset of the reserved words of Java, since Hadoop scripts compile into runnable Java code. The following words are reserved for use as keywords, and may not be redefined by a programmer:

`bool`	`emit`	`int`	`text`
`case`	`final`	`iter`	
`catch`	`for`	`list`	`try`
`default`	`foreach`	`map`	
`dict`	`hadoop`	`real`	`void`
`else`	`if`	`return`	
`elseif`	`instanceof`	`switch`	`while`

The experienced programmer may note that several of Java's reserved words are missing from this list. Because Hog programs compile into Java programs, at first blush it would seem that there is the potential here to create invalid java programs. The Hog compiler is smart enough to recognize when a user uses a Java reserved word as an identifier, and automatically prepends the text "hog" to avoid any collisions.

3.5 Constants

@Paul: flesh this out

3.6 String Literals

A string literal consists of a sequence of zero of more contiguous characters enclosed in double quotes, such as "hello". A string literal can also contain escape characters such as slash-n for the new line character or slash-t for the tab character.

@Kurry: flesh this out as k&r would

3.7 Variable Scope

Hog implements what is generally referred to as lexical scoping or block scope. An identifier is valid within its enclosing block. The identifier is also value for any block nested within its enclosing block.

4 Syntax Notation

In the syntax notation used throughout the Hog manual, different syntactic categories are noted by italic type, and literal words and characters are in typewriter style.

5 Types

5.1 Basic Types

The basic types of Hog include `int` (integer numbers, 64 bytes in size), `real` (floating point numbers, 64 bytes in size), `bool` (boolean values, true or false) and `text` (Strings, variable in size). Unlike most languages, Hog includes no basic character type. Instead, a programmer makes use of `text`s of size 1.

Implementation details Hogs primitive types are not so primitive. They are in fact wrappers around Hadoop classes. For instance, Hogs `int` type is a wrapper around Hadoop's `IntWritableclass`. The following lists for every primitive type in Hog the corresponding Hadoop class that the type is built on top of:

Hog Type	Enclosed Hadoop Class
int	IntWritable
real	DoubleWritable
bool	BooleanWrtiable
text	text???

5.2 Derived Types (Collections)

Derived types include `list<T>`, `set<t>`, `multiset<t>`, and `iter<t>`. The `list<T>` type is an ordered collection of objects of the same type. The `set<T>` is an unordered collection of unique objects of the same type. The `multiset<T>` is an unordered collection of objects of the same type, with duplicates allowed. The `dict<K,V>` is a collection of keyvalue pairs, where keys are all of the same type, and values are all of the same type (keys and values can be of different types from one another). The only types currently allowed within collections are primitive types, preventing such constructs as a list of lists. All collections allow for null entries.[1]

5.3 Conversions

@Paul

6 Expressions

6.1 Operators

6.1.1 Arithmetic Operators

Hog implements all of the standard arithmetic operators. All arithmetic operators are only defined for use between variables of numeric type (`int`, `real`) with the exception that the `+` operator is also defined for use between two `text` variables. In such instances, `+` is defined as concatenation. Thus, in the following,

```
text face = "face"
text book = "book"
text facebook = face + book
```

After execution, the variable `facebook` will have the value “facebook”. No other arithmetic operators are defined for use with `text` variables, and `+` is only valid if both variables are of type `text` . Otherwise, the program will result in a compile-time `TypeMismatchException`.

When an arithmetic operator is used between two numeric variables of different type, as in,

[1]Note that for `set<T>`, only one `null` entry is allowed, and for `map<K,V>`, only one `null` key is allowed.

```
int a = 1
real b = 2.0
a + b
```

the non-`real`variable will be ***coerced*** into a `real` before the evaluation of the statement, so that both operands have the same type. Therefore, the resulting type of the value of an expression involving an arithmetic operator and one or two operand of type `real` is always `real`.

If one of the operands happens to have a `null` value (for instance, if a variable is ***uninitialized***), then the resulting operation will cause a run-time `NullValueException`, and the program will crash.

Operator	**Arity**	**Associativity**	**Precedence Level**	**Behavior**
`+`	binary	left	0	addition
`-`	binary	left	0	minus
`*`	binary	left	1	multiplication
`/`	binary	left	1	division
`%`	binary	left	2 ??	mod[2]
`++`	unary	left	3	increment
`--`	unary	left	3	decrement

6.1.2 Logical Operators

The following are the logical operators implemented in Hog. Note that these operators only work with two operands of type `bool`. Attempting to use a logical operator with an object of any other type results in a compile-time exception (see §13.2).

Operator	**Arity**	**Associativity**	**Precedence Level**	**Behavior**
`or`	binary	left	0	logical or
`and`	binary	left	1	logical and
`not`	unary	right	2	negation

6.1.3 Comparators

The following are the comparators implemented in Hog (all are binary operations).

Need to say something about comparing two objects of different types, and null types. @Kurry, say something about which comparators don't work with non-numeric types

`<`	none	0	less than
`<=`	none	0	less than or equal to
`>`	none	0	greater than
`>=`	none	0	greater than or equal to
`==`	none	0	equal
`!=`	none	0	not equal

Comparison operators are used in boolean expressions and result in either a true or false being returned. Comparisons require that the two operands be either both numeric (including char) or both boolean, and a numeric value cannot be compared to a boolean value.

6.1.4 Assignment

There is one single assignment operator, '`=`'. Expressions involving the assignment operator have the following form:

```
lvalue = expression | identifier
```

At compile-time, the compiler checks that both the result of the `expression` (or `PRIMITIVE` or `DERIVED`) and `lvalue` have the same type. If not, a compile-time `TypeMistmatchException` will be thrown.

7 Declarations

While it is not specified in the grammar of Hog, like many other programming languages, a user is only allowed to use variables/functions after they have been declared. **W**hen declaring a variable, a user must include both a type and an identifier for that variable. Otherwise, an exception will be thrown at compile time.

7.1 Type Specifiers

Every variable, be it a `primitive-type` or a `derived-type` has to be assigned a type upon declaration, for instance,

```
list<int> myList
```

Declares the variable `myList` to be a `list` of `int`s. And,

```
text myText
```

Declares the variable `myText` to be of type `text`.

7.2 Declarations

7.2.1 Null Declarations

If a variable is declared but not initialized, the variable becomes a ***null reference***, which means it points to nothing, holds no data, and will fail any comparison (see §6.1) for a discussion of how `null` affects comparisons and elementary arithmetic and boolean operations).

7.2.2 Primitive-type Variable Declarations

Variables of one of the primitive types, including `int`, `real`, `text` or `bool` are declared using the following patterns:

1. `variable-type variable-name` (a null declaration)

2. `variable-type variable-name = primitive-expr` (declaration with initialization)

When the first pattern is used, we say that the variable is ***uninitialized***, and has the value `null`. When the second pattern is used, we say that the variable is ***initialized***, and has the same value as the value of the result of the `primitive-assignment-expr`. The `primitive-assignmentexpr` must return a value of the right type, or the compiler will fail citing a syntax error. The `primitiveassignmentexpr` may contain an expression involving both other variables and unnamed raw primitives (e.g. 1 or 2), an expression involving only other variables or unnamed raw primitives, or a single variable, or a single unnamed raw primitive.

7.2.3 Derived-Type Variable Declarations

Derived-type variables are declared using the following patterns:

1. `variable-type variable-name`

2. `variable-type variable-name = derived-expr`

3. `variable-type variable-name(parameter-list)`
   `parameter-list -> parameter, parameter-list | parameter`

The first two patterns operate in essentially the same way as for primitivetype variables. When the first pattern is used, we say that the variable is ***uninitialized***, and has the value `null`. If a user attempts to use any type-specific operations (for instance, `size(myList)`) on an uninitialized variable, the program will through a runtime exception (see §13 for a discussion of exceptions). When the second pattern is used, the variable is ***initialized*** to the result of the `derivedexpr`.

Because derivedtype variables often have additional structure that needs to be defined at initialization, a third pattern is provided. In this pattern, the user can specify a list of ***parameters*** to initialize the object. For instance,

```
list<int> myList(5)
```

Specifies that `myList` should be initialized with five `null` values.

7.2.4 Function Declarations

@Paul, reiterate how you define functions

8 Statements

8.1 Expression Statement

An ***expression statement*** is either an individual assignment or a function call. All consequences of a given expression take effect before the next expression is executed.

8.2 Compound Statement (Blocks)

Compound statements are defined by { and } and are used to group a sequence of statements, so that they are syntactically equivalent to a single statement.

8.3 Flow-Of-Control Statements

@Paul: think about what you want here! Maybe include example for each? The following are the ***flow-of-control*** statements included in Hog:

```
if (expression) statement

if (expression) statement else statement

if (expression) statement elif (expression statement) ..  else
statement

switch(expression) statement
```

In the above statements, the ... signifies an unlimited number of elseif statements, since there is no limit on the number of elseif statements that can appear before the final else statement. In all forms of the if statement, the expression will be evaluated as a Boolean. If the expression is a number, then any nonzero number will be considered true and zero will be treated as false. In the second statement above, when the expression in the if statement evaluates to false, then the else statement will execute. In the third statement above with if, elseif and else statements, the statement will be executed that follows the first expression evaluating to true. If none of these expressions evaluate to true, then the else statement is executed.

The switch statement causes control to transfer to a statement depending on the matching case label. There can be an unlimited number of case labels within the switch statement, so that the switch will operate as such:

```
switch(expression) {
  case consant-expression : statment
  default : statement
}
```

An expression is passed in to the switch statement and then the flow of control will fall through the switch and the expression will then be compared to the constant expression next to each case label. When the switch expression matches the expression next to a specific case, the statement for that case is executed. If the flow of control falls to the bottom of the switch without finding an equality, then the default statement will be executed. The case constants are converted to the switch expression type. There cannot be two case expressions with the same value after conversion. In addition, there can only be one default label within each switch.

The above control statements can all be nested within each other.

8.4 Iteration Statements

Iteration statements signify looping and can appear in one of the two following forms:

```
while ( expression ) statement-list

for (expression ; expression ; expression ) statement-list

foreach expression in iterable-object statement-list
```

@Ben: clean up the above, needs to be a specific example, because it's not always true in this generic a fashion.

In the above while statement, expression1 will typically represent the initialization of a variable, then at each iteration through the while loop, the current Boolean value of expression2 is evaluated. Expression2 will be a test or condition. If expression2 evaluates to true, then iteration will continue through the while loop. Expression3 normally represents an increment step and its value changes at each step through the while. Expression3 could be a part of the condition tested in expression2. The first time expression2 evaluates to false, iteration through the loop ends and drops to the code that comes after the closing bracket of the while statement.

In the above for statement, expression1 is the initialization step, expression2 is the test or condition and expression3 is the increment step. At each step through the for loop, expression2 is evaluated. When expression2 evaluates to false, iteration through the loop ends.

When the foreach starts to execute, the iteration starts at the first element in the array or list and the statement executes during every iteration. The iteration ends when the statement has been executed for each item in the array (or list) and there are no items left to iterate through.

8.4.1 Example 1

```
int i = 0
while (i < 10) {
  print(i)
  i++
}
```

8.4.2 Example 2

```
for (int i = 0; i < 10; i++) {
  print(i)
}
```

8.4.3 Example 3

```
list<int> iList()
iList = [0,1,2,3,4,5,6,7,8,9]
foreach i in iList {
  print(i)
}
```

9 Built-in Functions

Overview of built-in functions? How they are called on objects... @Paul: do this

9.1 System-level Built-ins

Hog includes a number of systemlevel builtin functions that can be called from various sections of a Hog program. The functions are:

```
void emit(key, value)
```

This function can be called from the `@Map` and `@Reduce` sections in order to communicate the results of the map and reduce functions to the Hadoop platform. The types of the key/value pairs must match those defined as the output types in the header of each section.

```
void mapReduce()
```

This function can be called from the `@Main` section in order to initiate the mapreduce job, as definied in the `@Map` and `@Reduce` sections. Any Hog program that implements mapreduce will need to call this function in `@Main`.

```
void print(toPrint)
```

This function can be called from the `@Main` section in order to print to standard output. The argument must be a primitive type.

9.2 Object-level Built-ins

The derived type objects have several built-in functions that provide additional functionality.

@ALL: introduction. **@Sam: do this**

9.2.1 `iter`

`iter` is Hog's iteration object, and supports several built-in functions that are independent of the particular type of the `iter` object. Note, as with other objects in this section, if the function has return type T, it means that the return type of this function matches the parameterized type of this object (i.e. for an `iter<int>` object, these functions have return type `int`). The built-in functions are as follows:

```
iter<T> duplicate_self()
```

This function returns a copy of the current `iter` object, with the cursor currently advanced to the current position of this cursor. The new `iter` object is independent of the previous one: advancing the cursor on the new object does not advance the cursor of the old one. **@ALL: should we support? would allow for branch-prediction-eqsue computation, may help recovering from network errors**.

```
T peek()
```

This function returns the next object (if one exists) for the owning `iter` object. A call to `peek()` returns the object without advancing the iterator's cursor, thus multiple calls to `peek()` without any intermediate function calls will all return the same value. If `peek()` is called on an iterator that has no more values, `null` is returned.

```
T next()
```

This function returns the next object (if one exists) for the owning `iter` object. A call to `next()` differs from a call to `peek()` in that the function call advances the cursor of the iterator. If `next()` is called on an iterator that has no more values, `null` is returned.

```
bool has_next()
```

This function returns `true` if the iterator object has a next object to return, and `false` otherwise. This function is equivalient to evaluating `if (iter_object.peek() == null)` .

9.2.2 `set`

`bool add(T element)`

Returns `true` if the element was successfully added to the `set`, `false` otherwise.

`void clear()`

Removes all elements from the `set` such that it is empty afterwards.

`bool contains(T element)`

Returns `true` if the `set` contains this element, `false` otherwise.

`bool isEmpty()`

Returns `true` if there are no elements in this `set`, `false` otherwise.

`iter<T> iterator()`

Returns an iterator over the elements in this `set`.

`bool remove(T element)`

Returns `true` if the element was successfully removed from the `set`, `false` otherwise (i.e. the `list` didn't contain `element`).

`int size()`

Returns the number of elements in the set.

`bool removeAll(set<T> otherSet)`

Returns `true` if all the elements in `otherSet` were successfully removed from this set.

`bool containsAll(set<T> otherSet)`

Returns `true` if all elements in `otherSet` are found in this set.

9.2.3 List

`void sort()`

Function that sorts the items in the list in lexicographical ascending order.

`int len()`

Returns an int with the number of elements in the list.

`void append(T itemToAppend)`

Appends the object passed to the end of the list. The object must be of the same type as the list, or the operation will result in a **compile-time or run-time** exception.

`T get(int index)`

Returns the item from the list at the specified index.

`iter<T> iterator()`

Returns an iterator for the objects in this list.

9.2.4 Multiset

The `multiset` object is similar in kind to the `set` object in that it is an unordered collection of objects of the same type. The key difference between `set` and `multiset` is that `multiset` allows for duplicates. The built-in functions on `multiset` are as follows:

`void append(T object)`

Adds `object` to the `multiset`.

`bool contains(T object)`

Returns `true` if this `multiset` contains at least one instance of `object`, `false` otherwise.

`int count(T object)`

The `count(T object)` method returns the number of instances of `object` in the given `multiset`. If the `multiset` does not contain the given object, then `0` is returned.

`list<T> entry_set()`

This function returns a list of all entries in the multiset, with duplicates removed. Please note: objects are added to the list in an arbitrary order, since `multiset` objects are inherently unordered. Do not ever rely on `entry_set()` providing a consistent ordering of objects.

```
bool is_empty()
```

This function returns `true` if the `multiset` contains no objects, and `false` otherwise.

```
iter<T> iter()
```

This function returns an iterator for the entries of the objects in this list, with duplicates removed. Again, the order of the objects is arbitrary.

```
bool remove_all(T object)
```

Removes all instances of `object` from the given `multiset`. Returns `true` if removal was successful, `false` otherwise (i.e. `object` was not contained in the `mutliset`).

```
bool remove_one(T object)
```

Removes one instance of `object` from the given `multiset`. Returns `true` if the removal was successful, `false` otherwise (i.e., `object` was not in the `multiset`).

9.2.5 text

The following function can be called on a `text` object:

```
list<text> tokenize(text fullText, text delimiter)
```

`tokenize()` can be called on a `text` object to tokenize it into a list of `text` objects based on the delimiter. The delimiter is not included in any of the `text` objects in the returned list.

10 System Configuration

The user must set configuration variables in the `hog.rb` build script to allow the Hog compiler to link the Hog program with the necessary jar files to run the MapReduce job. The user must also specify the job name within the Hog source file.

HADOOP_HOME: absolute path of hadoop folder

HADOOP_VERSION: hadoop version number

JAVA_HOME: absolute path of java executable

JAVAC_HOME: absolute path of javac executable