

# Hog Language Reference

Jason Halpern  
Samuel Messing  
Benjamin Rapaport  
Kurry Tran  
Paul Tylkin

March 20, 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.

In response to this need, a group of engineers at Google developed the MapReduce framework in 2004. This high-level framework can be used for 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 of this is done in a fault tolerant manner.

The MapReduce framework partitions input 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 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.”



Figure 1: Overview of the Map Reduce program

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 implementation of the MapReduce framework, 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 with distributed computation and can 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. However, the user should be adept with programming concepts such as program structure, control flow (iteration and conditional operators), evaluation of boolean expressions, etc.

## 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**, **@Reduce**, and **@Main** 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 {
  type functionName(parameterList) {
    expressionList
  }
}

```

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 valuenam`). 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.

```
@Map: (int lineNum, text line) -> (text, int) {
    foreach word in line.tokenize(" ") {
        emit(word, 1)
    }
}
```

## 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, 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 `@Reduce` section, which continues the word count example, and follows the `@map` sample introduced in the previous section.

```
@Reduce: (text word, iter<int> values) -> (text, int) {
```

```

    int count = 0
}

```

## 2.5 @Main

The @Main section defines the code that is the entry point to a Hog program. In order to run the map reduce program defined by the user in the previous sections, @Main must contain a call to the system-level built-in function mapReduce(). Other arbitrary code can be run from the @Main section as well. Currently, @Main does not have access to the results of the map reduce program resulting from a call to mapReduce(). Therefore, it is quite common for the @Main section to contain the call to mapReduce() and nothing else.

Below is a sample @Main section which prints to the standard output and runs a map reduce job.

```

@Main {
    print("Starting mapReduce job.\n")
    mapReduce()
    print("mapReduce complete.\n")
}

```

## 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 compiler` 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 and a newline character ('\n') on the right-hand side.

### 3.3 Identifiers

A valid identifier in Hog is a sequence of contiguous letters, digits, or under-scores, which are used to distinguish declared entities, such as methods, parameters, or variables from one another. A valid identifier also provides a means of determining scope of an entity, and helps to determine whether the same valid identifier in another scope refers to the same entity. The first character of an identifier must not be a digit. Valid identifiers are case sensitive so foo is not the same identifier as Foo.

### 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:

@functions	default	if	real
@main	dict	in	return
@map	else	instanceof	switch
@reduce	elseif	int	text
and	emit	iter	throw
bool	final	list	try
break	for	map	void
case	foreach	not	while
catch	hadoop	or	

### 3.5 Constants

There are different kinds of constants, corresponding to each of the primitive data types in Hog. Once a variable has been declared a constant, its type and value cannot be changed. To declare a constant, use the following notation:

```
final type variableName value
```

constants can be of the following types:

```
int
real
bool
text
```

Example: `int` constants would be declared as such: `final int rate 100`

### 3.6 String Literals

consists of a sequence of zero or more contiguous characters enclosed in double quotes, such as `"hello"`. A string literal can also contain escape characters such as `"\n"` for the new line character or `"\t"` for the tab character. A string literal has a `String` type has many of the same builtin functions as the `String` class in Java. String literals are constant and their values cannot be changed after they are created. String literals can be concatenated with adjacent string literals by use of the `+` operator and are then converted into a single string. Hog implements concatenation by use of the Java `StringBuilder`(or `StringBuffer`) class and its `append` method. All string literals in Hog programs are implemented as instances of the `String` class, and then are mapped directly to the equivalent `String` class in Java.

### 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 in base 10, 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 `texts` 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 `IntWritable` class. 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
<code>int</code>	<code>IntWritable</code>
<code>real</code>	<code>DoubleWritable</code>
<code>bool</code>	<code>BooleanWritable</code>
<code>text</code>	<code>Text</code>

## 5.2 Derived Types (Collections)

Derived types include `dict<K, V>`, `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.<sup>1</sup>

The last derived type is `iter<T>`, which is Hog's iterator object. An `iter` object is associated with a collection object (of one of the types mentioned above), and allows the user to traverse the elements in the collection once.

## 5.3 Conversions

In order to cast a variable to be of a different type, use the following notation:  
(new type)variableName

Not all basic types can be cast to a different basic type. Any variable of type `int` or `real` can be cast to any of the other basic types (`int`, `real`, `bool` `text`). However, a variable of type `bool` can only be cast to `text` and a variable of type `text` cannot be cast to any of the other variable types.

# 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
```

---

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

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,

```
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 `NullPointerException`, and the program will crash.

Operator	Arity	Associativity	Precedence Level	Behavior
<code>+</code>	binary	left	0	addition
<code>-</code>	binary	left	0	minus
<code>*</code>	binary	left	1	multiplication
<code>/</code>	binary	left	1	division
<code>%</code>	binary	left	2	mod <sup>2</sup>
<code>++</code>	unary	left	3	increment
<code>--</code>	unary	left	3	decrement
<code>-</code>	unary	right	3	negate

### 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 §13.1 13.2).

Operator	Arity	Associativity	Precedence Level	Behavior
<code>or</code>	binary	left	0	logical or
<code>and</code>	binary	left	1	logical and
<code>not</code>	unary	right	2	negation

### 6.1.3 Comparators

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

Operator	Associativity	Precedence Level	Behavior
<	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

*Note:* All comparators do not work with non-numeric or non-boolean types, except for `null`. Comparisons require that the two operands be either both numeric or both boolean, and a numeric value cannot be compared to a boolean value. The only valid comparators that can be used with boolean expressions are `==` and `!=`. The use of a comparison operator in Hog between any two objects (non-numeric (including `char`), or non-boolean) will result in a compile-time exception (see §13.1 13.2). In Hog, `null == null` will evaluate to `false`. For any non-null reference value `foo`, `foo.equals(null)` will evaluate to `false`.

#### 6.1.4 Assignment

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

$$identifier_1 = expression \mid identifier_2$$

At compile-time, the compiler checks that both the result of the *expression* (or *identifier<sub>2</sub>*) and *identifier<sub>1</sub>* have the same type. If not, a compile-time `TypeMismatchException` 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. When 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 `ints`. 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. *type identifier* (uninitialized)
2. *type identifier = expression* (initialized)

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 *expression*. The *expression* must return a value of the right type, or the compiler will throw a `TypeMismatchError`. The *expression* 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. *type identifier*
2. *type identifier = expression*
3. *type identifier(parameterList)*  
*parameterList*  $\rightarrow$  *parameter*, *parameterList* | *parameter*

The first two patterns operate in essentially the same way as for primitive-type 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, `myList.size()` 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.