

# Hog Language Tutorial

Jason Halpern  
Samuel Messing  
Benjamin Rapaport  
Kurry Tran  
Paul Tylkin

March 20, 2012

## Introduction

Hog gives users with some programming experience a gentle introduction to mapReduce, a popular programming model for distributed computation. In a Hog program, a user specifies an @Map function, which operates on key-value pairs (read from a text file), and outputs intermediate key-value pairs. The user also specifies an @Reduce function, which groups the intermediate key-value pairs by key, and outputs a final set of key-value pairs. This model of computation has been widely adopted for parallelizing large computations that might be considered "embarrassingly parallelizable."

## Program Structure

Every Hog program has four sections, defined in the following order:

**@Functions** optional section which defines functions used throughout the program.

**@Map** defines the map function that takes the input key-value pairs and outputs intermediate key-value pairs.

**@Reduce** defines the reduce function that takes a single key from the set of intermediate key-value pairs output by the map function, and all of the values associated with that key, to reduce to a final output.

**@Main** the entry point for a program which usually initiates the mapReduce program.

## Word Count

Let's assume we have thousands of large text files, and we would like to get a cross-file word count for each word that appears in any of the files. We also have a cluster of computers to help us complete this task. The following short Hog program will produce a single output file with each word and its count.

## Word Count Code

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

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

```

        int count = 0
        while (values.hasNext()) {
            count = count + values.getNext()
        }
        emit(word, count)
    }

    @Main: {
        mapReduce()
    }

```

## Word Count Explanation

The general idea of this program is that we want to read every line of text from every file, and then, grouping by word, output the number total number of times we encountered the word. Since we want to group by word, this is an indicator that the words will be the intermediate key output by the @Map function, since their values will be grouped and sent to the @Reduce function.

### @Functions

The first thing we notice is that this program does not contain an @Functions section. This section is optional, and only needs to be included if the user must write his or her own subroutines to be used elsewhere in the program.

### @Map

This section's job is to read in a line of text from a file, and simply output each word as a key with a value that indicates we have just encountered it (that will be used for summation later in the program).

The first line of this section defines the signature of the @Map function. In the current release, all Hog programs read input files one line at a time, where the file offset of the line is the key, and the text of the line is the value. That is why in this and all Hog programs, the only allowable types for the input key-value pair is (int, text). The inputs are also named in the signature, to reference them in the body of the function.

The input signature is followed by an arrow, followed by the type signature of the outputted intermediate key-value pairs. In this case, we will output each word as text and a count as an int. These values are unnamed, as they cannot be referenced in the @Map section.

In the body of the function, we split the line of text passed in as the value into words delineated by whitespace by using the built-in function `tokenize()`. We then iterate through the list of words that `tokenize()` returns using a `foreach` loop.

In the body of the foreach loop, we use the built-in function `emit()` to output a key-value pair to the framework for grouping by key. In this case, since we want to group by the word, we emit the word and the value 1, which we will use to calculate the totals in the `@Reduce` section.