The Hog Programming Language

Jason Halpern
jrh2170
Testing/Validation

Samuel Messing
sbm2158
Project Manager

Benjamin Rapaport
bar2150
System Architect

Kurry Tran
klt2127
System Integrator

Paul Tylkin
pt2302
Language Guru

May 6, 2012

Contents

Chapter 1

Introduction

1.1 Taming the Elephant

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),[1] 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.

Hog is a **data-oriented**, **high-level**, scripting language for creating MapReduce[2] 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.

1.1.1 Data-Oriented

Hog is a powerful language that allows for the efficient handling of structured, unstructured and semi-structured data. Specifically, Hog simplifies the process

[1] `http://hadoop.apache.org/`

of writing programs to handle the distributed processing of data-intensive applications. Programmers using Hog only have to express the steps for processing the data in the Map and Reduce functions without having to be concerned with relations and the constraints imposed by a traditional database schema. Hog also provides control flow structures to manipulate this data. In addition, Hog frees a programmer from having to write each step in a data processing task since many of those low-level processing details are handled by the language and the system.

Hog uses Hadoop MapReduce (TM), an open-source MapReduce framework written in Java. Hadoops run time system takes care of the details of partitioning the input data, scheduling the programs execution across machines, counteracting machine failures, and managing inter-machine communication. Hadoop also distributes data to machines and tries to collocate chunks of data with the nodes that need it, therefore maximizing data locality and giving good performance.

1.1.2 Simple

To write a simple word count program in Java using the Hadoop framework requires over 59 lines of code.[2] The same program written in Hog requires just 10 lines. The discrepancy comes from the fact that Hog takes care of the low-level details required to correctly communicate and interact with the Hadoop framework. This allows users to enhance the expressive potential of their programs, without sacrificing power. All that Hog requires a user to do is specify the location of their valid Hadoop instance, write a map function to process a segment of data, write a reduce function to combine the results, and Hog takes care of the rest.

[2] `http://hadoop.apache.org/common/docs/current/mapred_tutorial.html`

Chapter 2

Tutorial

Use your updated tutorial.

Chapter 3

Language Reference Manual

3.1 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 language can express the same computation in 15 lines.

3.1.1 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, the question is how they can 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."[5] GFS was designed with the same goals as other distributed file systems, including "performance, scalability, reliability and availability."[3] 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."[3]

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."[2] 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."[2] A programmer only has to specify the functions described above and the system handles the rest of the details. Figure 3.1.1 illustrates the execution flow of a MapReduce program.

3.1.2 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

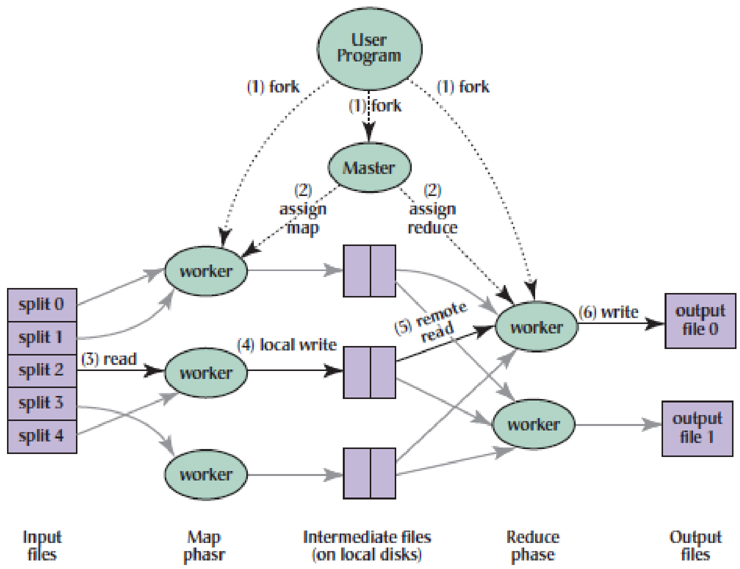

Figure 3.1: Overview of the MapReduce program, from [3].

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 only supports reading and writing plaintext files. While these limitations sacrifice the generality of the language, they promote ease of use.

Guiding Principles

The guiding principles of Hog are:

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

3.1.3 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 Java or Python), 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.

3.2 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`. When specific terms are introduced, ***emboldened, italicized font*** is used.

3.3 Program Structure

3.3.1 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 {
    .
    .
    .
 }
```

3.3.2 `@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:

> *type functionName* `(` *parameterList* `)` {
>   *expressionList*;
> }

where,

$$parameterList \rightarrow parameter \texttt{ , } parameterList \mid parameter$$

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.size() - 1; i >= 0; i--;) {
      newList.add(oldList.get(i));
    }
    return newList;
  }
}
```

User-defined functions can make reference to other user-defined functions. However, function names cannot be overloaded (i.e. it is not possible to use the same function name with a parameter list that differs in the number of arguments or argument types). Disallowing function overloading is a design choice consistent with Hog's guiding principle of simplicity.

3.3.3 `@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.

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.

Note: In the current version of the language, 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.

```
@Map (int lineNum, text line) -> (text, int) {
    # for every word on this line, emit that word and the number 1
    foreach text word in line.tokenize(" ") {
        emit(word, 1);
     }
```

```
}
```

3.3.4 `@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) {
  # initialize count to zero
  int count = 0;
  while (values.hasNext()) {
    # for every instance of '1' for this word, add to count
    count = count + values.next();
  }
  # emit the count for this particular word
  emit(word, count);
}
```

3.3.5 `@Main`

The `@Main` section defines the code that is the entry point to a Hog program. In order to run the MapReduce program defined by the user in the previous sections, `@Main` must contain a call to the system-level built-in function `mapReduce()` , which calls the `@Map` and `@Reduce` functions. Other arbitrary code can be run from the `@Main` section as well. In the current version of the language, `@Main` does not have access to the results of the MapReduce 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");
}
```

3.4 Lexical Conventions

3.4.1 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.

3.4.2 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` is 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.

3.4.3 Identifiers

A valid identifier in Hog is a sequence of contiguous letters, digits, or underscores, which are used to distinguish declared entities, such as methods, parameters, or variables from one another. A valid identifier also provide 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`.

3.4.4 Keywords

The following words are reserved for use as keywords, and may not be redefined by the programmer:

`add`	`final`	`iter`	`removeAll`
`and`	`for`	`list`	`return`
`bool`	`foreach`	`Map`	`size`
`break`	`Functions`	`mapReduce`	`sort`
`catch`	`get`	`next`	`text`
`clear`	`hadoop`	`not`	`text2int`
`contains`	`hasNext`	`or`	`text2real`
	`if`	`peek`	`throw`
`containsAll`	`in`	`print`	`tokenize`
`continue`	`instanceof`	`real`	`try`
`default`	`int`	`real2int`	`void`
`else`	`int2real`	`real2text`	`while`
`elseif`	`int2text`	`Reduce`	
`emit`	`isEmpty`	`remove`	

3.4.5 Constants

The word ***constant*** has two different meanings in Hog. It can refer to either a variable that is *fixed*, that is, once it is initialized cannot be changed, or can refer to an ***unnamed value***, such as "1.0". To declare a constant variable, use the following pattern,

`final` *type variableName* = *value*;

The following are a list of examples of unnamed values and their corresponding types:

`-1, 0, 1, 2` (all of type `int`)

`-0.12`, `3.14159`, `2.7182`, `1.41421` (all of type `real`)

`true`, `false` (all of type `bool`)

3.4.6 Text Literals

A text literal consists of a sequence of zero of more contiguous characters enclosed in double quotes, such as `"hello"`. A text literal can also contain escape characters such as `"\n"` for the new line character or `"\t"` for the tab character. A text literal has many of the same built-in 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 text literals by use of the `+` operator and are then converted into a single `text` variable. Hog implements concatenation by use of the Java `StringBuilder` (or `StringBuffer`) class and its append method. All text literals in Hog programs are implemented as instances of the `text` class, and then are mapped directly to the equivalent `String` class in Java.[1]

3.4.7 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 valid for any block nested within its enclosing block.

3.4.8 Argument Passing

Since Hog is compiled into Java, it passes arguments using call-by-value. However, similarly to Java, it is possible to imitate call-by-reference behavior.

3.4.9 Evaluation Order

Hog uses applicative order (eager) evaluation, similarly to Java.

3.5 Types

3.5.1 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 some 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

[1] Technically, `text` objects are implemented as instances of Hadoop's `Text` class, which is closely related to the Java `String` class.

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`

3.5.2 Derived Types (Collections)

There are two derived types that can be created by the programmer: `list<T>`and `set<T>`. Future versions of Hog are expected to implement other derived types, including dictionaries/hash maps, user-defined iterators, and multisets. 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. Hog supports arbitrarily nested derived types, so it is possible, for example, to have a `list`of `lists` of `lists` of `ints`.

A special derived type is `iter<T>`, which is Hog's iterator object. An `iter` object is associated with a list, and allows one traversal of the elements in the list; this is used by Hog in the `@Reduce` section of a Hog program.

3.5.3 Type Conversions

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

primitiveType`(`2*otherPrimitiveType*`)`*variableName*

Hog supports casting between the primitive types `int`, `real`, and `text`, via the built-in functions `int2real`, `int2text`, `real2int`, `real2text`, `text2int`, and `text2real`. If casting a text to an int or real results in an invalid number (e.g. `text2int("1a4")`), a run-time exception will be thrown.

3.6 Expressions

3.6.1 Operators

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,

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

the non-`real` variable would first need to be cast into a `real` before operating on them, so that both operands have the same type. So thus

```
print(a + b);
```

would throw an error, while

```
print(int2real(a) + b);
```

would print 3.0.

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†
`++`	unary	left	3	increment
`--`	unary	left	3	decrement
`-`	unary	right	3	negate

†Follows Java’s behavior: a modulus of a negative number is a negative number.

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 §**??**).

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

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. Comparisons require that the two operands be either both numeric or both boolean, and a numeric value cannot be compared to a boolean value. If the two operands are numeric but of different types, one of them must be cast so that they are of the same type. The only valid comparators that can be used with boolean expressions are `==` and `!=`. The use of a comparison operator in Hog between any two derived types will result in a run-time error.

Assignment

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

$$\textit{identifier}_1 \texttt{ = } \textit{expression} \mid \textit{identifier}_2$$

At compile time, the compiler checks that both the result of the *expression* (or $\textit{identifier}_2$) and $\textit{identifier}_1$ have the same type. If not, a compile-time `TypeMismatchException` will be thrown.

Chapter 4

Project Plan

Written by Samuel Messing (sbm2158).

4.1 Development Process

The scope of the Hog programming language was ambitious from the start. Our stated goal was to create a general-purpose scripting language which made carrying out distributed computation simple and intuitive. As such, from the beginning we were interested in ways to make the implementation of the language as simple as possible. The following goals were identified early on:

- make the build system as simple as possible,
- make the logic of our individual modules as similar as possible,
- document everything,
- use a distributed version control system,
- write verbose and informative log statements.

Focusing on these goals throughout the development enabled use to work concurrently on different aspects of the compiler and maintain a codebase that was both readable and easy to understand.

4.1.1 Simplicity of Build System

As project manager, I worked early on with both the System Architect (Ben) and the System Integrator (Kurry) to come up with a build system that was simple and easily extensible. After trying a few different options, we decided on Ant, a build system similar to Make, specialized for the Java programming language. Another advantage of going with Ant is that both JFlex and Cup, the frameworks we used to construct the lexer and parser, respectively, have native

Ant support. Identifying and implementing our build system early on enabled us to move quickly and write code that we were sure worked across all of our machines.

4.1.2 Similarity of Modules

Throughout the project, I worked very closely with the System Architect (Ben) to develop and build common data structures that could be used across all of our code. The abstract syntax tree was made in a generic enough way so that all of our different tree walks could use the exact same tree class, without having to support and debug different implementations of the same interface or abstract parent class.

Personally, I also developed our Types class, which was a static class that contained several convenience methods for handling types across the entirety of the compiler. These methods include type checking, type conversion and as well as additional functionality required for internal functionality. I set out to write the class as early as possible so that both elements of the frontend and the backend could make use of it. Simplifying and unifying how different modules handled the same information enabled everyone on the team to read each other's code qand quickly understand how it functioned.

4.1.3 Document Everything

One of the most undersold parts of Java is it's well thought out documentation schema (JavaDocs). Early on I realized that in order for us to be able to work semi-independently on different modules we would need to have a robust set of documentation. By using JavaDoc instead of regular comments, we were able to generate HTML documentation, which more clearly provides an overview of the entire architecture of our compiler, and allowed everyone on our team to work quickly and respond to updated classes appropriately.

One of the largest challeneges in this project was developing a set of node classes for our abstract syntax trees that captured the right granularity of information, without beeing too complex that handling corner cases became intractable. Our System Architect (Ben) found a great tool that generated UML diagrams for our class hierarchies, which in concert with our JavaDocs helped to make development as simple and efficient as possible.

4.1.4 Distributed Version Control

As soon as our team was formed I created a git repository on Github.com[1] for use by the team. One of the first things we discussed as a team was what workflow pattern we wanted to use throughout the course of the project. Very quickly we decided on a continuous-build pattern, where the main branch of our git repository (master) was reserved for compiling, tested, and finalized code.

[1] `http://www.github.com/smessing/Hog`

Any classes that were currently in development existed in separate branches, and were only merged into master after sufficient amount of testing. Each programmer maintained their own branch for development. If two or more programmers were working on the same class, a new, shared branch was created. By being conservative about what code was merged into the master branch, we were able to work independently, without fear that someone else's work would be interrupted by leaving our individual code in an unfinished state.

4.1.5 Verbose Logging

Another advantage of programming in Java is the robust and sophisticated logging libraries available to the programmer. Around the same time that the build system was developed, the System Architect (Ben) investigated several different logging libraries and wrote a tutorial for the rest of us on how to use it. The logging library supported several levels of log statements, FINEST, FINER, FINE, INFO, WARNING and SEVERE (from most verbose to least). We decided that FINEST and FINER were to be used strictly for debugging, while FINE was to be used to document normal behavior, at a level of detail that was concise enough for all developers to look at, but still too verbose for the user. INFO, WARNING and SEVERE were reserved for statements that the user would see. By identifying and keeping to these log levels early on, we were able to quickly identify bugs and inefficient or errant behavior.

4.2 Roles and Responsibilities

- Ben, System Architect

  Ben's major responsibilities included developing the fundamental data structures used by the compiler, working out the different elements of the compiler and how they interrelate, and developing the symbol table.

- Jason, Testing/Validation

  Jason's major responsibilities included testing all of the elements of our compiler, and working on the aspects of the compiler related to type checking, and developing the symbol table.

- Kurry, System Integrator

  Kurry's major responsibilties included developing a clean interface between Hog and Hadoop and working on the Hog wrapper program that builds, compiles and runs Hog source programs.

- Paul, Language Guru

  Paul's major responsibility was determining the syntax and semantics of our language, and developing the semantic analyzer.

- Sam, Project Manager

  As project manager, my major responsibilities included setting project deadlines, assigning work, and making sure that we met our goals. I was also responsible for developing the classes to translate Hog programs into Java programs.

4.3 Hog's Developer Style Sheet

We made use of the standard Java style guide, including such conventions as camel case, verbs for functions and method names, and hierachical object classes. For formatting, we used Eclipse's auto-format feature to keep our code looking as consistent as possible.

4.4 Project Timeline

January

Developed several potential ideas for languages. Met with Aho and decided on implementing Hog, a MapReduce language.

February

Worked on the White Paper for our language, developed both the goals of our language and the overall "feel" (simple, minimal boilerplate code, easy-to-read syntax). Started to sketch out overall compiler architecture, and decided on frameworks (JFlex for the lexer, CUP for parser, Hadoop framework for executing distributed computation, and Java as target language) and development environments (Eclipse, Git, Github, LaTeX for documentation).

March

Wrote the language reference manual and tutorial for our language. Developed the build system (Ant for compiling compiler code, Make for running the compiler on Hog source programs), implemented and tested the parser and lexer, and developed the fundamental data structures (abstract syntax tree, node classes).

April

Implemented tree walking algorithms to populate the symbol table, perform type checking, perform semantic analysis and generate Java source code. Wrote tests for the walkers.

May

Refactored code and worked on documentation. Developed more tests and worked on fixing bugs.

4.5 Project Log

January

Week of January 22nd

* Met to discuss language ideas.

Week of January 29th

* Decided on Hog, and Java as implementation language.

February

Week of Feburary 5th

* Decided on Hadoop as the framework for executing distributed computation.
* Decided on JFlex framework for implementing the lexer.
* Decided on CUP framework for implementing the parser.

Week of February 12th

* Discussed and figured out development environment (Java, Ant, Eclipse, Git).
* Started working on white paper.

Week of February 19th

* Started git repository.
* Finished white paper.

Week of February 26th

* Began the language reference manual (LRM).

March

Week of March 4th

* Started Eclipse project.
* Worked on LRM and tutorial.

Week of March 11th

* Began developing the Hog grammar.
* Worked on LRM and tutorial.
* Started working on wrapper program functionality (program that runs the Hog compiler to compile source programs).

Week of March 18th

* Finished the Hog grammar.

- Finished the tutorial and LRM.
- Began developing the lexer.

Week of March 25th

- Took the week off to study for the midterm.

April

Week of April 1st

- Worked on the lexer.
- Started developing the abstract syntax tree and node classes.
- Started developing the parser.
- Implemented developer build system.

Week of April 8th

- Developed ConsoleLexer class for development and testing of lexer.
- Developed lexer JUnit tests.
- Finished abstract syntax tree, including iterators for post- and pre-order traversals.
- Developed mock classes for testing.

Week of April 15th

- Further development/refinement of node classes.
- Developed more semantic actions for parser, mainly to construct node classes.
- Parsed our first program!

Week of April 22nd

- Developed/implemented basic type functionality.
- Further refinement of the grammar.
- Implemented logging details.
- Refinement of node classes and ASTs.
- Started tree walking algorithms (identified the visitor pattern as our common design pattern for tree walks).
- Begain developing symbol table class.

Week of April 29th

- Finished implementation of symbol table class.
- Finished type checking / symbol table population walks.
- Implemented java source generator.
- Implemented tests for walkers and parser.
- Finished compiler.

Chapter 5

Language Evolution

- Describe how the language evolved during the implementation and what steps were used to try to maintain the good attributes of the original language proposal.
- Describe the compiler tools used to create the compiler components.
- Describe what unusual libraries are used in the compiler.
- Describe what steps were taken to keep the LRM and the compiler consistent.

The initial intent was to make Hog stylistically and aesthetically resemble Python. In our first discussions about the language, we had envisioned statements being separated by line breaks and dynamic typing.

Chapter 6

Translator Architecture

To be written by Ben.

- Show the architectural block diagram of translator.
- Describe the interfaces between the modules.
- State which modules were written by which team members.

Chapter 7

Development and Run-Time Environment

To be written by Kurry.

- Describe the software development environment used to create the compiler.
- Show the makefile used to create and test the compiler during development.
- Describe the run-time environment for the compiler.

Chapter 8

Test Plan

As the tester and validator for Team Hog, I set out to create a systematic, automated set of tests at each step in the process of building a compiler. In order to make sure that each part of the design worked according to our specification, I tried to include tests that touched as many aspects of the language as possible.

I considered each of the testing phases to be a two-step process. First, create a basic set of tests with the assumption that the compiler worked as expected. These tests would touch a variety of areas of the language. These were our black box tests because they were built without the need to know what was going on under the hood. Then, the second step of the process was to attempt to break the language in as many ways as possible. These tests required an intimate knowledge of the nuances of the language and were therefore our white box tests. I tried to incorporate as many boundary cases as possible into these tests. At each phase of the testing, we uncovered various bugs and unimplemented aspects of the language that we fixed on subsequent iterations. I will briefly touch upon each phase of testing and the challenges and outcomes faced throughout the process. All of these tests are in the test package in our source code. The tests were developed using Javas JUnit development framework.

Lexer Testing (`LexerTester.java`)

In order to test the lexical analysis of Hog programs, I created a large variety of short code snippets, passed them to the lexer and made sure that the correct tokens were being returned. For example, when the string "a++" was passed to the lexer, I created tests with `assertEquals()`to make sure that the first token returned was ID and the second token returned was INCR. I started with small tests that only touched two to five token streams and built towards strings that were thirty tokens long. This phase helped us discover certain tokens that were not being returned correctly and needed to be added/modified in the lexer, such as `TEXT`, `TEXT_LITERAL`, and `UMINUS`. A sample lexer test can be seen at the end of this section.

Parser Testing (`ParserTester.java`)

This was the most challenging aspect of the testing process. Due to the limitation of built-in parsing methods, it was difficult to create an automated set of tests for the parser that tested each part of the grammar. This phase relied more heavily on manual testing than I would have preferred. We were able to run a variety of programs through the parser and focused on breaking the parser and touching as many edge cases as possible. This allowed us to uncover the bugs and produce code that was not correctly parsed. We had to modify and expand the grammar from the results of this testing. The tests that we created for the parser were the motivation for creating such specific node subclasses that captured the different details associated with each production. In addition, information that we gathered in testing the parser also allowed us to create a clean design for the symbol table, which is constructed during parsing and the first walk of our AST.

Symbol Table Testing (`SymbolTableTester.java`)

I found when I reached this phase of creating tests that there were certain details of the node classes that I needed to gain a better understanding of in order to write tests. For this reason, I worked closely with Ben and Paul in designing and implementing the Symbol Table and worked with Ben on the Symbol Table Visitor and Type Checking Visitor. In order to test the construction of the Symbol Table, we created several sample programs, created the Symbol Table from these programs and analyzed the symbol table to make sure the information was being correctly captured. In addition, we also made sure reserved words and functions were in the reserved symbol table at the root of the Symbol Table structure. There were two key issues related to creating nested scopes that were uncovered during testing and an important issue related to adding function parameters and argument lists to the symbol table. This phase also focused on making sure the correct exceptions were being thrown i.e. VariableRedefinedException, VariableUndeclaredException, etc.

Abstract Syntax Tree Testing (`AbstractSyntaxTreeTester.java`)

In order to test the AST, we created an automated set of tests that was based on the pre and post order traversals of the AST. First, we created an AST during the set up phase of the testing and made sure that we included a variety of node structures on the tree. Then, we did both a preorder walk of the tree and a postorder walk of the tree and made sure the traversals were occurring in the correct order.

Type Checking Testing (`TypesTester.java` and `TypeCheckingTester.java`)

During this walk of the AST, we did type checking and decorated the tree with the correct types. I created many of the tests for this part of the design as

Ben and I implemented functionality in the type check walk. The first part of type checking testing was to make sure the functions that we wrote around type compatibility were operating correctly. For example, we had to make sure if we visit a BiOpNode with the plus operator that the operands are both text (concatenation) or numbers (addition). Once the tests proved that these functions were all valid, we moved to implementing tests on the walk of an actual AST to make sure type decorating was occurring according to our rules. This part of the testing uncovered the fact that our IdNodes were not being decorated at all during our initial walk, so we added the functionality to the TypeCheckingVisitor to handle this.

Code Generation Testing (`CodeGeneratingTester.java`)

The goal for the code generation tests was to determine whether or not our programs were being correctly mapped to Hadoop programs written in Java. These tests focused on using the code generating visitor walk of the tree to make sure that the structure, meaning and types of our Hog programs were being captured during the transformation to Java. Besides writing tests, this step of the testing involved actually running the Hadoop programs on our local machines and on Amazon Elastic MapReduce to see if the programs would run without errors and if the results in the output files were in line with what we expected.

Testing Hog Programs

In order to prepare us for testing, I set us up on Amazon Web Services to run our programs on Amazons Elastic MapReduce platform. We upload the jar of our compiled program and the input files to Amazons S3 storage platform, then we launch the Elastic MapReduce job on a small cluster with 2 instances. The output files are stored in S3 after the processing has successfully completed. The instructions for running a Hog program on AWS are detailed in the report.

For several aspects of our implementation, we focused on a pair approach to programming and to testing. Since Sam handled a lot of the implementation regarding lexical analysis and parsing, he also worked with me to create additional tests for these phases to make sure we captured everything. In addition, he also added type tests for some additional type functions that he wrote. Kurry created some tests for the node structure since he also worked on creating ASTs. In addition, since I wanted to really understand the node structure, symbol table and visitor pattern, I worked together with Ben and Paul in designing and implementing the symbol table, designing the visitor pattern for Hog and implementing the Type Checking walk. Working on these aspects of the project enabled me to write better tests since I had a deeper understanding of these classes. I also think that writing tests helped Kurry and Sam better understand the aspects of the project that they were working on.

One of the main challenges during testing was capturing the breadth and depth of the Hog language in all of the tests. Testing, in conjunction with development, was an iterative process that required us to add and modify the testing suites as functionality was added to and removed from the language specification. As the tester and validator for this project, I believe I have developed the skills to more rigorously test software. More importantly, I learned a lot about the principles related to strong software design and software engineering during the entire process.

Sample Test from `LexerTester.java`

```
/**
* Tests for correct parsing of the postfix increment operator
*
* Specifically, ensures that Lexer produces a token stream of ID * INCR
*     for strings like "a++"
*
* @throws IOException

@Test
public void incrementSymbolTest() throws IOException {

String text = "a++";
StringReader stringReader = new StringReader(text);
Lexer lexer = new Lexer(stringReader);
List<Integer> tokenList = new ArrayList<Integer>();
Symbol token = lexer.next_token();

while (token.sym != sym.EOF) {
tokenList.add(token.sym);
token = lexer.next_token();
}

assertEquals(
"It should produce 2 tokens for the string '" + text + "'", 2, tokenList.size());
assertEquals("The first token should be a ID", sym.ID, tokenList.get(0)
.intValue());
assertEquals("The second token should be a INCR",
               sym.INCR, tokenList.get(1).intValue());
}
```

Chapter 9

Conclusions

9.1 Lessons Learned

9.1.1 Jason's Lessons

To be written by Jason.

9.1.2 Sam's Lessons

To be written by Sam.

9.1.3 Ben's Lessons

To be written by Ben.

9.1.4 Kurry's Lessons

To be wrtten by Kurry.

9.1.5 Paul's Lessons

To be written by Paul.

9.2 Advice for Other Teams

Don't take this class.

9.3 Suggestions for Instructor

Things we'd like to see more of:

- More details

- More discussion of functional languages

Appendix A

Code Listing

Include a listing of the complete source code with identification of who wrote which module of the compiler. This listing does not have to be included in the paper copy of the final report.

Bibliography

[1] Aho, A., Lam, M., Sethi, R., and J.R., U. *Compilers: Principles, Techniques, & Tools*, 2nd ed. 2007.

[2] Dean, J., and Ghemawat, S. Mapreduce: Simplified data processing on large clusters. *OSDI '04* (2004), 10.

[3] Ghemata, S., Gobioff, H., and Leung, S. The google file system. *The 19th Symposium of Operating Systems Principles* (2003), 29–43.

[4] Kernighan, B., and Ritchie, D. *The C Programming Language*, 1st ed. 1978.

[5] Patterson, D. Technical perspective: The data center is the computer. *Communications of the ACM 51*, 1 (2005), 105.