Hog Language Reference

May 6, 2012

Jason Halpern	Testing/Validation
Samuel Messing	Project Manager
Benjamin Rapaport	System Architect
Kurry Tran	System Integrator
Paul Tylkin	Language Guru

Contents

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.

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."[**?**] 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."[**?**]

Within the framework, a programmer is responsible for writing both map

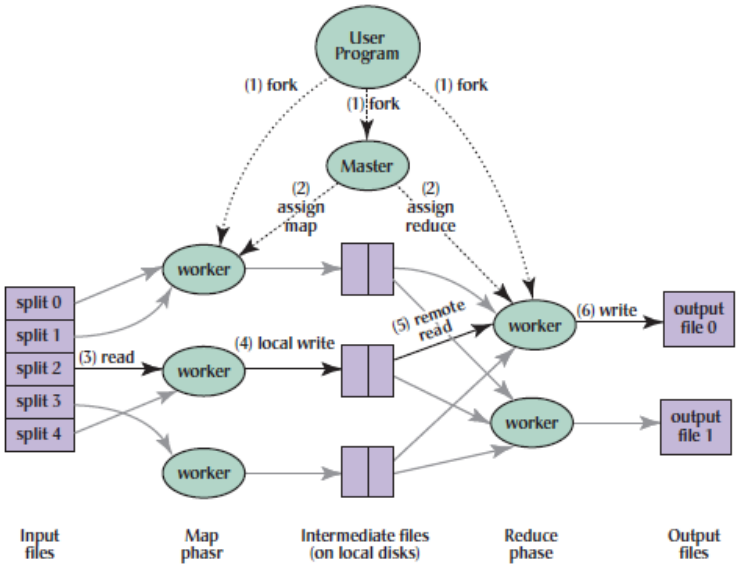

Figure 1: Overview of the MapReduce program, from [**?**].

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 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 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.

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 Program Structure

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 {
   .
   .
   .
}
```

3.2 `@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 → *parameter* `,` *parameterList* | *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 `@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);
     }
 }
```

3.4 `@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);
}
```

3.5 `@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");
}
```

4 Lexical Conventions

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.

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.

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`.

4.4 Keywords

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

```
add             bool            catch           contains

and             break           clear
```

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

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`)

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]

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.

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 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 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)

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 `list`s of `list`s of `int`s.

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.

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

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.

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,

```
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.

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).

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).

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.

6.1.4 Assignment

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

$$identifier_1 \texttt{ = } expression \mid identifier_2$$

At compile time, the compiler checks that both the result of the *expression* (or $identifier_2$) and $identifier_1$ have the same type. If not, a compile-time `TypeMismatchException` will be thrown.

7 Declarations

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, whether its type is primitive or derived, must be assigned a type upon declaration, for instance,

```
list<int> myList;
```

declares the variable `myList` to be a `list` of `int`s,

```
list<list<int>> myOtherList;
```

declares the variable `myOtherList` to be a `list` of `list`s 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 and holds no data (internally, this means that an entry has been added to Hog's symbol table with that variable name).

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 pattern:

1. *type identifier;*

When the derived type is first declared, we say that the variable is ***uninitialized***, and has the value `null`. If a user attempts to use any type-specific operations that are not meaningful (for instance, `myList.size()` on an uninitialized variable, the program will throw a runtime exception (see §13 for a discussion of exceptions)). The example code below initializes a `list`of integers and adds one element to it.

```
list<int> myList;
myList.add(5);
```

7.2.4 Function Declarations

In order to declare a function, use the following notation:

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

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

The following are the ***flow-of-control*** statements included in Hog:

`if (` *expression* `)` *statement*

`if (` *expression* `)` *statement* `else` *statement*

`if (` *expression* `)` *statement* `elseif (` *statement* `) ... else` *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 `bool`. 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.

To increase the expressive power of Hog, flow-of-control statements can also 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*

`for (` $expression_1$ `;` $expression_2$ `;` $expression_3$ `;)` *statement*

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

In the `while` pattern, the associated *statements* will be executed repeatedly until the *expression* evaluates to `false`. The *expression* is evaluated before every iteration. Please note that in a slight syntactical departure from Java, Hog requires a semicolon after the third expression (the increment step) in the `for`loop construct. Thus, an example of correct Hog syntax would be

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

In the `for` pattern, $expression_1$ is the initialization step, $expression_2$ is the test or condition and $expression_3$ is the increment step. At each step through the for loop, $expression_2$ is evaluated. When $expression_2$ evaluates to false, iteration through the loop ends.

In the `foreach` pattern, the iteration starts at the first element in the *iterable-object statement* (a statement that evaluates to an object that supports the `iterator()` function). The *statement* executes during every iteration. The iteration ends when the *statement* has been executed for each item in the iterable object and there are no items left to iterate through.

8.4.1 Example of `while`

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

8.4.2 Example of `for`

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

8.4.3 Example of `foreach`

```
# we first initialize and populate the list as follows:
list<int> iList;
for (int i = 0; i < 10; i++;) {
   iList.add(i);
}

# This is an example of using foreach
# Note that the type of the iterable must be declared.

foreach int i in iList {
  print(i);
}
```

9 Built-in Functions

Hog includes both ***system-level*** and ***object-level*** built-in functions. Here ***built-in*** means functions provided by the language itself.

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 of these functions are invoked using the following pattern:

identifier.functionName(parameterList)

Where *identifier* is the identifier for the object in question, *functionName* is the name of the function, and *parameterList* is a (possibly empty) list of parameters used to specify the behavior of the invocation.

Note: In what follows, if a 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`).

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. The built-in functions are as follows:

`bool hasNext()`

This function returns `true` if the iterator object has a next object to return, and `false` otherwise.

`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.

`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.

9.2.2 `list`

`void add(T itemToAdd)`

Adds 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.

`void clear()`

Removes all elements in this `list`.

`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.

`void sort()`

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

`int size()`

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

9.2.3 `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 containsAll(set<T> otherSet)`

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

`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`).

`bool removeAll(set<T> otherSet)`

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

`int size()`

Returns the number of elements in the set.

9.2.4 text

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

`int length()`

Returns the length (number of individual characters) of this `text`.

`text replace(text matchText, text replacementText)`

Returns a new `text` object with each sub-`text` that matches `matchText` replaced by `replacementText`. This function does **not** alter the original `text` object.

`list<text> tokenize(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

HOST where to job is rsynced to and run

LOCALMEM how much memory for java to use when running in local mode

REDUCERS the number of reduce tasks to run, set to zero for map only jobs

11 Compilation Structure

Currently, the Hog compiler is implemented as a translator into the Java programming language. The first phase of Hog compilation uses the JFlex as its lexical analyzer, which is designed to work with the Look-Ahead Left-to-Right (LALR) parser generator CUP. The lexical analyzer creates lexemes, which are logically meaningful sequences, and for each lexeme the lexical analyzer sends to the LALR parser a token of the form <`token-name, attribute-value`>. The second phase of Hog compilation uses Java CUP to create a syntax tree, which is a tree-like intermediate representation of the source program, which depicts the grammatical structure of the Hog source program.

In the last phase of compilation, the Hog semantic analyzer generates Java source code, which is then compiled into byte code by the Java compiler. Then with the Hadoop Java Archives (JARs) the bytecode is executed on the Java Virtual Machine (JVM). With the syntax tree and the information from the symbol table, the Hog compiler then checks the Hog source program to ensure semantic consistency with the language specification. The syntax tree is initially untyped, but after semantic analysis Hog types are added to the syntax tree. Hog types are represented in two ways, either a translation of a Hog type into a new Java class, or by mapping Hog types to the equivalent Java types. Mapping Hog types directly to Java types improves performance because a JVM can handle primitive types much more efficiently than objects. Also, a JVM implements optimizations for well-known types, such as String, and thus Hog is built for optimal performance.

12 Linkage and I/O

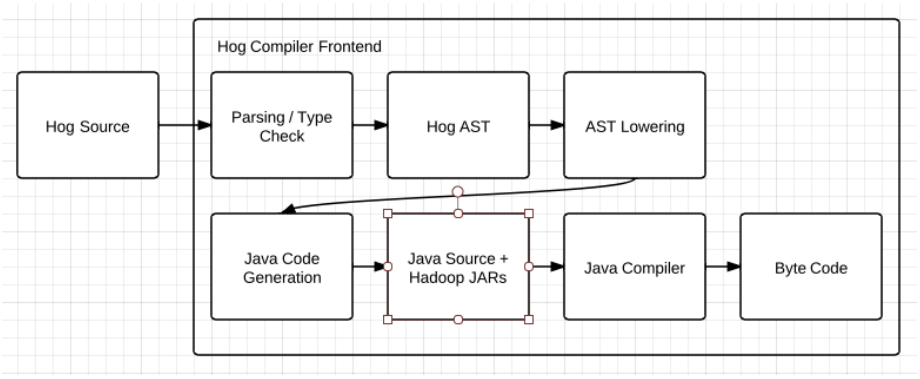

Figure 2: The overall structure of the Hog compiler.

12.1 Usage

To build and run a Hog source file there is an executable script `hog` that automates the compilation and linking steps for the user.

Usage: `hog [--hdfs|--local] job <job args>`

`--hdfs`: if job ends in '.hog' or '.java' and the file exists, link it against the hadoop JARFILE and then run it on HOST.

`--local`: run on local host.

12.2 Example

```
hog --local WordCountJob.hog --input someInputFile.txt --output ./someOutputFile.csv
```

This runs the `wordCount` job in *local* mode (i.e. not on a Hadoop cluster).

13 Exception Handling

Similar to some other programming languages (such as Java and C++), Hog uses an exception model in which an exception is thrown and can be caught by a catch block. Code should be surrounded by a try block and then any exceptions occurring within the try block will subsequently be caught by the catch block. Each try block should be associated with at least one catch block. However, there can be multiple catch blocks to handle specific types of exceptions. In addition, an optional finally block can be added. The finally block will execute in all circumstances, whether or not an exception is thrown. The structure of exception handling should be similar to this, although there can be multiple catch blocks and the finally block is optional:

```
try {
    expression;
} catch ( exception ) {
    expression;
} finally {
    expression;
}
```

The current version of the language does not support the programmer throwing exceptions, only catching them.

Because the proper behavior of a Hog program is dependent on resources outside of the language (i.e. the proper behavior of the users Hadoop software), there are more sources exceptions in Hog than most general purpose languages. These sources can be divided into two categories: ***compile-time exceptions*** and ***internal run-time exceptions***.

13.1 Compile-time Errors

The primary cause of most compile-time exceptions in Hog are semantic errors. Such errors are unrecoverable because it is impossible for the compiler to properly interpret the user program. Some compilers for other languages offer a limited amount of compile-time error correction. Because Hog programs are often designed to process gigabytes or terabytes of data at a time, the standard Hog compiler offers no compile-time error correction. The assumption is that a user would rather retool their program than risk the chance of discovering, only after hours of processing, that the compilers has incorrectly assumed what the user meant. The following are Hog compile-time exceptions:

`FunctionNotDefinedError`

Thrown when a program attempts to carry out an operations of the sort `variable.builtInFunction()` where `variable` is some variable and `builtInFunction` is a built-in function, and either `builtInFunction` cannot operate on variables of that type or `builtInFunction` is not defined as a built-in function.

`InvalidFunctionArgumentsError`

Thrown when a program calls a function with the wrong number or type of parameters. For example, if we define the function `max(int a, int b)` , this error will be thrown if the program contains a construct like `max(2,3,4)` or `max("hello", 3)` .

`TypeMismatchError`

Thrown when a program attempts to carry out an operation on a variable of the wrong type (like adding a `text` and an `int` together).

`UnreachableCodeError`

Thrown when code is included in a part of a program that will never be executed (e.g. code after a return statement that can never be reached).

13.2 Internal Run-time Exceptions

Internal runtime exceptions include such problems as I/O exceptions (i.e. a specified file is not found on either the users local file system or the associated Hadoop file system), type mismatch exceptions (i.e. a program attempts to place two elements of different types into the same list) and parsing exceptions. The following are Hog internal run-time exceptions:

`FileNotFoundException`

Thrown when the the Hog program attempts to open a non-existent file.

`FileLoadException`

Thrown when an error occurs while Hog is attempting to read a file (e.g. the file is deleted while reading).

`ArrayOutOfBoundsException`

Thrown when a program tries to access a non-valid index of a `list`.

`IncorrectArgumentException`

Thrown when a derived-type object is instantiated with invalid parameters, or a function is called with invalid parameters.

`TypeMismatchException`

Thrown when a program attempts to carry out an operation on a variable of the wrong type (like adding a `text` and an `int` together).

`NullReferenceException`

Thrown whenever the value of a variable cannot be `null` (e.g. in `myList.get(i)`, if `i` is `null`, the operation with throw a `NullPointerException`).

`ArithmeticException`

Thrown whenever an arithmetic operation is attempted on non-numeric operands.

14 Grammar

Note: The presented grammar has one minor ambiguity relating to the ***dangling-else*** problem. If the grammar is run through the parser generator `yacc`, `yacc` will identify 7 shift/reduce parsing-action conflicts. However, the ambiguity is handled by the default behavior of `yacc`, which preferences shift to reduce, associating `else` and `elseif` clauses with the closest `if` clause.

```
%token DECR INCR RETURN CONTINUE
%token TIMES  DIVIDE  MOD
%token LESS  GRTR  LESS_EQL  GRTR_EQL  DBL_EQLS  NOT_EQLS  ASSIGN
%token TEXT  BOOL  INT  REAL  VOID
%token MINUS  UMINUS  PLUS
%token ARROW  DOT
%token TEXT_LITERAL
%token ID
%token INT_CONST
%token REAL_CONST
%token BOOL_CONST
%token CASE
%token BREAK DEFAULT
%token AND OR NOT
%token WHILE FOR FOREACH IN IF ELSE ELSEIF SWITCH
%token FUNCTIONS MAIN MAP REDUCE
%token L_BRACE R_BRACE L_BRKT R_BRKT L_PAREN R_PAREN SEMICOL COL COMMA
%token LIST ITER SET
%token TRY CATCH FINALLY
%token EXCEPTION

%left MINUS PLUS
%right UMINUS
%right ELSE
%right ELSEIF
%right L_PAREN

%start Program

%%

Program
  : Functions Map Reduce Main
  ;

Functions
  : FUNCTIONS L_BRACE FunctionList R_BRACE
  | /* epsilon */
  ;

FunctionList
  : Function
  | FunctionList Function
  ;

Function
```

```
  : Type ID L_PAREN ParameterList R_PAREN L_BRACE StatementList R_BRACE
  ;

ParameterList
  : ParameterList COMMA Type ID
  | Type ID
  | /* epsilon */
  ;

Map
  : MAP SectionType L_BRACE StatementList R_BRACE
  ;

Reduce
  : REDUCE SectionType L_BRACE StatementList R_BRACE
  ;

SectionType
  : L_PAREN Type ID COMMA Type ID R_PAREN ARROW L_PAREN Type COMMA Type R_PAREN
  ;

Main
  : MAIN L_BRACE StatementList R_BRACE
  ;

StatementList
 : Statement
 | StatementList Statement
 ;

Statement
  : ExpressionStatement
  | SelectionStatement
  | IterationStatement
  | LabeledStatement
  | JumpStatement
  | DeclarationStatement
  | GuardingStatement
  | Block
  ;

GuardingStatement
   : TRY Block Finally
   | TRY Block Catches
   | TRY Block Catches Finally
   ;
```

```
Block
  : L_BRACE StatementList R_BRACE
  | L_BRACE R_BRACE
  ;

Finally
 : FINALLY Block
 ;

Catches
  : CatchHeader Block
  | Catches CatchHeader Block
  ;

CatchHeader
  : CATCH L_PAREN EXCEPTION ID R_PAREN
  ;

DeclarationStatement
  : Type ID
  | Type ID ASSIGN Expression
  ;

JumpStatement
  : CONTINUE
  | BREAK
  | RETURN ExpressionStatement
  ;

ExpressionStatement
  : SEMICOL
  | Expression SEMICOL
  ;

Expression
  : LogicalExpression
  | UnaryExpression ASSIGN Expression
  ;

LogicalExpression
  : LogicalExpression OR LogicalTerm
  | LogicalTerm
  ;

LogicalTerm
```

```
  : LogicalTerm AND EqualityExpression
  | EqualityExpression
  ;

EqualityExpression
  : RelationalExpression
  | EqualityExpression DBL_EQLS RelationalExpression
  | EqualityExpression NOT_EQLS RelationalExpression
  ;

RelationalExpression
  : AdditiveExpression
  | RelationalExpression LESS AdditiveExpression
  | RelationalExpression GRTR AdditiveExpression
  | RelationalExpression LESS_EQL AdditiveExpression
  | RelationalExpression GRTR_EQL AdditiveExpression
  ;

AdditiveExpression
  : MultiplicativeExpression
  | AdditiveExpression PLUS MultiplicativeExpression
  | AdditiveExpression MINUS MultiplicativeExpression
  ;

MultiplicativeExpression
  : CastExpression
  | MultiplicativeExpression TIMES CastExpression
  | MultiplicativeExpression DIVIDE CastExpression
  | MultiplicativeExpression MOD CastExpression
  ;

CastExpression
  : UnaryExpression
  | L_PAREN Type R_PAREN CastExpression
  ;

UnaryExpression
  : UnaryOperator CastExpression
  | PostfixExpression
  ;

UnaryOperator
  : MINUS %prec UMINUS
  | NOT
  ;
```

```
PostfixExpression
  : PrimaryExpression
  | ID DOT ID
  | ID DOT ID L_PAREN ArgumentExpressionList R_PAREN
  | ID L_PAREN ArgumentExpressionList R_PAREN
  | PostfixExpression INCR
  | PostfixExpression DECR
  ;

ArgumentExpressionList
  : Expression
  | ArgumentExpressionList COMMA Expression
  | /* epsilon */
  ;

PrimaryExpression
  : ID
  | Constant
  | L_PAREN Expression R_PAREN
  ;

Constant
  : INT_CONST
  | REAL_CONST
  | BOOL_CONST
  | TEXT_LITERAL
  ;

SelectionStatement
  : IF Expression Block ElseIfStatement ElseStatement
  | SWITCH Expression L_BRACE StatementList R_BRACE
  ;

ElseIfStatement
  : ELSEIF Expression Block ElseIfStatement
  | /* epsilon */
  ;

ElseStatement
  : ELSE Block
  | /* epsilon */
  ;

IterationStatement
  : WHILE L_PAREN Expression R_PAREN Block
  | FOR L_PAREN ForInit ForExpr ForIncr R_PAREN Block
```

```
  | FOR L_PAREN ForInit ForExpr R_PAREN Block
  | FOREACH Type ID IN Expression Block
  ;

ForInit
  : ExpressionStatements
  | DeclarationStatement SEMICOL
  ;

ForExpr
  : ExpressionStatement
  ;

ForIncr
  : ExpressionStatements
  ;

ExpressionStatements
  : ExpressionStatement
  | ExpressionStatements COMMA ExpressionStatement
  ;

LabeledStatement
  : CASE LogicalExpression COL Statement
  | DEFAULT COL Statement
  ;

Type
  : VOID
  | TEXT
  | BOOL
  | INT
  | REAL
  | DerivedType LESS Type GRTR
  ;

DerivedType
  : LIST
  | ITER
  | SET
  ;
```