MR

A MapReduce Programming Language

W4115 Programming Language and Translator
Prof. Stephen A. Edwards

Siyang Dai
Zhi Zhang
Zeyang Yu
Jinxiong Tan
Shuai Yuan

Table of Contents

1. Introduction to MR

1.1 Background

MapReduce (MR) is a program language whose aim is to provide a convenient and effective way for users to develop distributed applications to process a large number of data across a large number of computers. The MR programs mainly consist of two functions, the Map function and the Reduce function, which are corresponding to the two significant steps in distributed computing. The Map function is meant to take the input and partition it up into smaller sub-problems, and then distribute them to computers, while the Reduce function is supposed to collect the answers to all the sub-problems from nodes and combines them in some way to form the output. Users can perform specific computation based on their own needs by self-define the body of the Map and Reduce function.

1.2 Features

The MR programming language has several features as follows.

1.2.1 Improving the efficiency

Hadoop is an effective framework in distributed computation. Users can develop applications run by Hadoop in Java. However, users have to finish some routine tasks first, say introducing a whole bunch of Hadoop libraries and defining the classes in Java. Hence, the MR programming language is effective since it is developed to eliminate such kinds of routine works.

1.2.2 Concice structure and syntax

The MR programming language consists of two functions, the Map function and Reduce function. Such design is more structured and concise, which contributes to the efficiency and readability of the MR programs. Nevertheless, we have simplified some syntactic structures. For example, we replace the for-loop structure with the followings,

foreach word in text {...}

1.3 Mechanism

The mechanism for the MR programming language is as follows.

Firstly, Users can write the codes to solve their own problems by merely focusing on the body of the Map and Reduce function, which contributes to the efficiency of distributed programming. Secondly, The compiler takes the MR codes from users as input, and translate them into Java language. In order to effectively eliminate the routine works, the compiler can automatically introduce all the necessary libraries, and define the classes in Java. After successfully compiling the MR program, some files with “java” as their suffixes are generated. Then we can use run a script to compile the java files with the necessary Hadoop libraries. At last, we have to package the java files in a single jar file and then the program can be run in Hadoop framework.

2. Tutorial

2.1 Prerequisties

Ensure that Hadoop is installed, configured and is running.

2.2 Example: wordcount.mr

Let's take the program wordcount.mr as an example of MR application and see how it works. wordcount.mr is a simple application that counts the number of occurrences of each word in a given input set.
It works with a local-standalone Hadoop installation.

Source code

```
//wordcount.mr

#JobName = "WordCount"

//map function definition

def wordcount_map <(Int, Text) -> (Text , Int)> (offset, line): Mapper

{

        List<Text> words;

        Int one = 1;

        words = split line by " ";
        foreach Text word in words
                emit(word,  one);
}

//reduce function definition

def wordcount_reduce <(Text , Int) -> (Text, Int)> (word, counts): Reducer

{

        Int total = 0;
        foreach Int count in counts
                total = total + count;
```

```
        emit(word, total);

}
```

2.3 Usage

The steps to show how to run the program are as follows:

0. Please run the Makefile to generate the actual compiler program - “translator”

1. ./mrc sample/wordcount.mr WordCount.java ./hadoop/
The first one is the compiler script; The second one is the source file; The third one is the target file(which must be the same as the Jobname attribute in the mr source); The fourth one is the hadoop directory

2. hadoop/bin/hadoop jar WordCount.jar WordCount ./input/ output
To run it, use the second command, the first one is the hadoop executable; the second one is the parameter telling hadoop to run the following jar file; the third one is the executable jar file. the fourth one is the main class of the executable; the last two are the input directory and output directory (the output directory should not exist a priori)

2.4 Explanation

The wordcount.mr application is quite straight-forward.

The **Mapper** transforms records into intermediate records, its implementation via the `wordcount_map` method, processes one line at a time. It then splits the line into tokens separated by whitespaces, and emits a key-value pair of < <word>, one>.

```
//configuration declaration

#JobName = “WordCount”

//definition of mapping relation of input and output for the function.

def wordcount_map <(Int, Text) -> (Text , Int)> (offset, line): Mapper

{
        //define text type of the list words
        List<Text> words;

        //define the constant one
        Int one = 1;
```

```
        //split the words when whitespace is met
        words = split line by " ";

        //Use foreach structure to traverse the List words given by the expression. It iterates
        through each object in the list and execute the emit() statement to output.
        foreach Text word in words
                emit(word,  one);

}
```

The **Reducer** reduces a set of immediate values which share a key to a small set of values, its implementation, via the `wordcount_reduce` method, sums up the values, which are the occurence counts for each key (i.e. words in this example).

```
//definition of mapping relation of input and output for the function.

def wordcount_reduce <(Text , Int) -> (Text, Int)> (word, counts): Reducer

{
        //define the integer variable total and set it to zero which uses to count the word.
        Int total=0;

        //Use foreach structure to traverse and count the word to total and collect the result.
        foreach Int count in counts
                total = total + count;
        emit(word, total);

}
```

3. Language Reference Manual

1. Introduction

MapReduce is a programming to support distributed computing on large data sets on clusters of computer. The paradigm is inspired by the map and reduce functions universally used in functional programming. The MR programming language is designed specifically for MapReduce.

1.1 Concept of MapReduce

1.1.1 List Processing

Essentially, the basic idea of a MapReduce program is to convert a list of input data elements into a list of output data elements. The transformation is done in two phases: map and reduce.

1.1.2 Map

The first phase of a MapReduce program is called mapping. A list of data pairs is fed, one at a time, to a function called the Mapper, which transforms each input element individually to an output data element. Logically, a map function is defined as the following form:

$$Map(k_1,v_1) \rightarrow list(k_2,v_2)$$

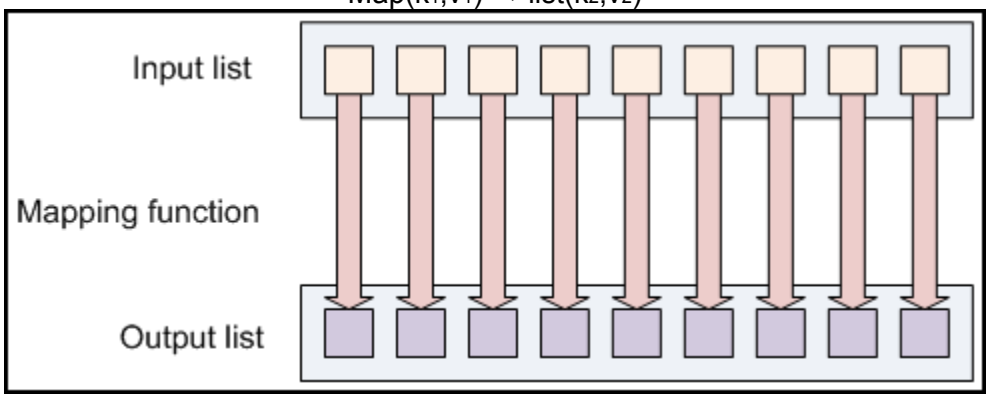


Figure 1 Map (Figure from Yahoo Developer Network)

After that, all pairs with the same key from all lists generated by map function will be grouped together, thus creating one group for each one of the different generated keys. The groups will be the input of the next phase.

1.1.3 Reduce

Reduction aggregates values together. A reduce function receives a list of values with the same key. It then combines these values together. Logically, a reduce function is defined as the following form:

$$Reduce(k_2, list\ (v_2)) \rightarrow (k_3,v_3)$$

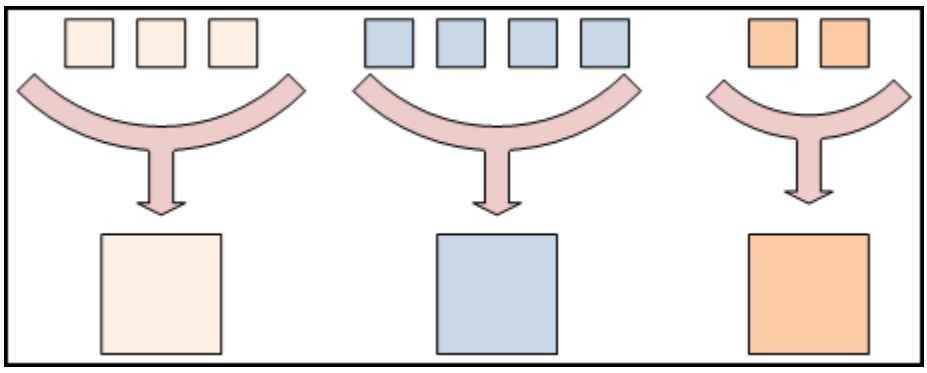


Figure 2 Reduce (Figure from Yahoo Developer Network)

As a result, we get a pair of (k,v) for each distinct key generated by map function.

1.2 Data-flow of MapReduce

Combining map and reduce, we can have the following overview for the data-flow of a MapReduce program on a cluster consisting of three nodes:

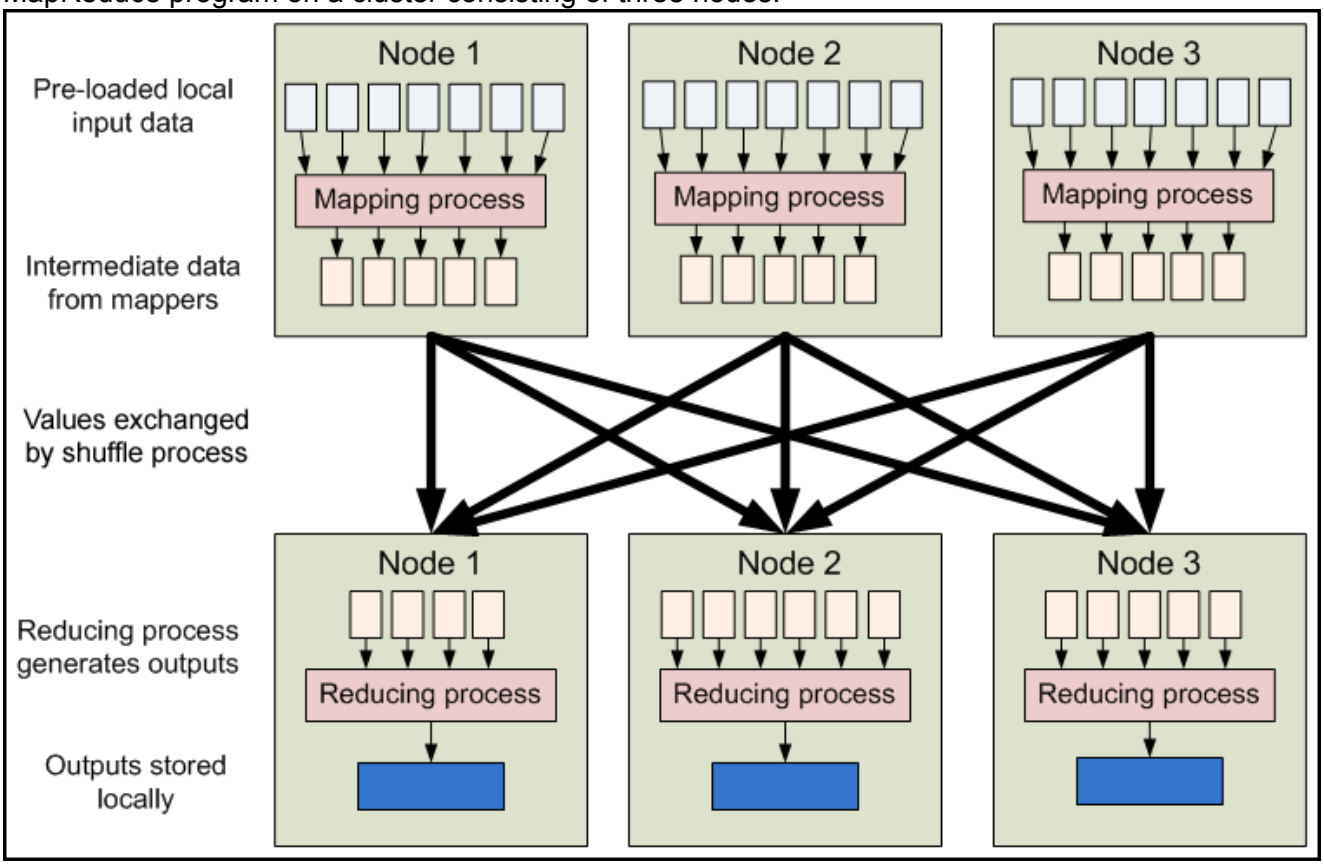


Figure 3 MapReduce (Figure from Yahoo Developer Network)

1.3 The MR Programming Language

MR is designed to support MapReduce paradigm. It hides the details of MapReduce framework from the programmers. All the programmers need to do is to define a map function and a reduce function. The program will be run according to the data-flow of MapReduce.

1.4 Input and Output of MR Program

An MR program takes two arguments from command line. The first one is the input directory. And the second one is the output directory.

1.4.1 Input

All files under the input directory are used as input files. MR treats each line of each input file as a separate record, and performs no parsing. It feeds the map function with the byte offset of the line as key and the line content as value. Therefore, for map function, k_1 is always an integer and v_1 is always one line of text.

1.4.2 Output

The output directory must not exist before the MR program runs. The MR program will create one automatically. The output of reduce function will be written to files under the output directory in form “key \t value” per line.

2. Lexical Elements

2.1 Tokens

There are five kinds of tokens in MR, i.e., literals, keywords, identifiers, operators and other separators. Blanks, newlines and comments are ignored during lexical analysis except that they separate tokens.

2.2 Constants

2.2.1 Text Constant

Text constant is a string containing a sequence of characters surrounded by a pair of double quotes, i.e. “...”. For example, “hello world!” is a Text constant. Identical Text constants are the same. All Text literal are immutable.

One thing to note is that, in MR, there is no character type. Even a single character is Text constant type which can be regarded as an extended character set.

2.2.2 Int Constant

A Int constant refers to a integer consisting of a sequence of digits. It supports signed and unsigned integers. Int constant cannot start with a 0 (digit zero). All integers are default to be decimal (base 10). For example, -15 and 2012 are valid Int constant.

2.2.3 Double Constant

In MR, a double constant refers to a floating constant which consists a integer part, a decimal point and a fraction part. In addition, it supports an ‘e’ followed by an optionally signed integer exponent. The integer part and fraction part can be one digit or a sequence of digits. Either of them can be missing, but not both. Also either the decimal point or the e and the exponent (not both) may be missing. The following are valid Double constants: 1. or 0.5e15 or .3e+3 or .2 or 1e5

2.3 Keywords

The following words are reserved as the keywords which cannot be used otherwise.

Text	**Int**	**Double**	**Boolean**	**List**
def	**if**	**else**	**foreach**	**emit**
and	**or**	**Mapper**	**Reducer**	**split**
by	**true**	**false**		

2.4 Identifiers
Identifiers are used for naming variables, parameters and functions. Identifier consists of a sequence of letters, digits and the underscore _ , but it must start with a letter. Identifier should not be the keywords listed above. It is case-sensitive.

2.5 Operators
An operator is a special token that performs an operation, such as addition or subtraction, on either one or two operands. More details will be covered in later section.

2.6 Separators
A separator separates tokens. Other separators (Blanks, newlines and comments) are ignored during lexical analysis except the following:
( ) < > { } ;

2.7 Comments
// is used to indicate the rest of the line is comment (C++/Java style comment)

3. Data Types

3.1 Int
The 64-bit Int data type can hold integer values in the range of −9,223,372,036,854,775,808 to 9,223,372,036,854,775,807.

3.2 Double
The Double type covers a range from 4.94065645841246544e-324d to 1.79769313486231570e+308d (positive or negative).

3.3 Boolean
A variable of Boolean may take on the values true and false only.

3.4 List
It is used as List<T>, i.e. List<Int> represents a list of Int values. It has unlimited size.

4. Program Structure
A MR program must exist entirely within a single source file (with a ".mr" extension). By convention, a typical MR source file must include three parts: configuration, mapper and reducer. That is,

program -> configuration-declaration mapper-definition reducer-definition

Here is an example program:

```
//wordcount.mr
#JobName = "WordCount"

//map function definition
def wordcount_map <(Int, Text) -> (Text , Int)> (offset, line): Mapper
{
```

```
        List<Text> words;
        words = split line by " ";
        foreach Text word in words {
                emit(word,  1);
        }
}
//reduce function definition
def wordcount_reduce <(Text , Int) -> (Text, Int)> (word, counts): Reducer
{
        Int total = 0;
        foreach count in counts {
                total = total + count;
        }
        emit(word, total);
}
```

4.1 Configuration

configuration-declaration -> **#***configuration-attribute* **= Text**$_{\textbf{const}}$**;**
configuration-attribute -> **JobName**
In this field, users can specify attribute JobName using a Text constant, which is started with a capital letter. (The support of specifying the number of Mapper/Reducer process will be extended in the future.)

4.2 Mapper/Reducer Definition

mapper-definition -> **def** identifier *mapping-relation parameters* **:** *function-type block*
reducer-definition -> **def** identifier *mapping-relation parameters*: *function-type block*

The keyword def explicitly indicates the following code is a function definition. identifier field is used to specify the name of function.

mapping-relation -> **<(***type-specifier*$_1$*, type-specifier*$_2$**) -> (***type-specifier*$_3$*, type-specifier*$_4$**)>**

mapping-relation defines the mapping relation of a pair of input and output for the function. The format is given as < (*type-specifier*$_1$*, type-specifier*$_2$) -> (*type-specifier*$_3$*, type-specifier*$_4$) >. For mapper, it specifies k_1, v_1 and k_2, v_2 as in Map(k_1,v_1) → list(k_2,v_2). For reducer, it specifies k_2, v_2 and k_3, v_3 as in Reduce(k_2, list (v_2)) → (k_3,v_3).

parameters -> (identifier$_1$, identifier$_2$)

Parameters refers to the identifiers that receive values passed to a function. identifier$_1$ is of type as *type-specifier*$_1$ specifies. identifier$_2$ is default to be a List of type as *type-specifier*$_2$ *specifies.*

function-type -> **Mapper**
| **Reducer**

At the end of the function declaration, it is compulsory for users to explicitly specify the function type. Exactly one mapper and one reducer is allowed and needed in one MR program.

4.3 Scope

A declared object can be visible only within a particular function. Also a declaration is not visible to declarations that came before it. A variable name cannot be referred before declared.

5. Expression

An expression consists of at least one operand and zero or more operators. The operands may be any value, including constants and variables.

5.1 Operators

The general view of the precedence and associative can be shown in the following table.

Operator	Description	Associativity
()	Parentheses	left-to-right
split by	Split a Text value by delimiter	
-	Unary Negative Operator (%prec)	
* /	Multiplication/division	left-to-right
+ -	Addition/subtraction	left-to-right
< <= > >=	Relational less than/less than or equal to Relational greater than/greater than or equal to	left-to-right
== !=	Relational is equal to/is not equal to	left-to-right
and	Logical AND	left-to-right
or	Logical OR	left-to-right
=	Assignment	

The operators on top on the table possess a higher precedence than those on the bottom of the table. The detail of the expression will be included in the following subsections with the order of precedence from high to low.

5.2 Primary Expression

expression:

literal
identifier
(expression)
unary-expression
binop-expression
split identifier by $\text{Text}_{\text{constant}}$
identifier = expression
declaration

An identifier is a primary expression, e.g., age. Its type should be specifically declared in the program before it is evaluated in an expression. A literal is a primary constant, e.g., 1, 2, 1.1, true. The type could be Boolean, Int, Double and Text. A parenthesized expression is a primary expression, e.g., (x+y). This expression allows you to group expressions together to allocate them a higher precedence. The other expressions will be explained in the following sections.

5.3 Unary Negative Operator

unary-expression:

- expression

The operand of this operation should have a type of Int or Double. This operation converts the value of the expression from a positive number to a negative number or vise versa.

5.4 Binop Operation

binop-expression:

arithmetic-expression
relational-expression
logical-expression

5.4.1 Arithmetic operators

The arithmetic operators include *, /, +, -.

arithmetic-expression:

expression ***** expression
expression **/** expression
expression **+** expression
expression **-** expression

The operands must be of type Int or Double.

The binop expression will return the arithmetic result of the operation. Operator * denotes multiplication, / denotes division, + denotes addition, and – denotes subtraction. When applying the division operation, the second operand could not be zero. Example: 11+22, 21.1*21.5

5.4.2 Relational operation

The relational operators group left-to-right.

relational-expression:

expression **<** expression
expression **>** expression
expression **<=** expression
expression **>=** expression
expression **==** expression
expression **!=** expression

The result of operations < (less), > (greater), <= (less or euqal), >= (greater or equal), == (equal to), and != (not equal to) is Boolean true/false according to the result of the boolean logic. Examples: x<y, 11>=33

5.4.3 Logical Operation

logical-expression:

expression **and** expression
expression **or** expression

The **and** operator groups left-to-right. It returns true if both its operands are evaluated to be true. Otherwise, it returns false. The **or** operator also groups from left-to-right. It returns true if either of its operands is evaluated to be true. Otherwise it returns false. Both **and** and **or** follows short-circuit evaluation, a.k.a. the second argument is only executed or evaluated if the first argument does not suffice to determine the value of the expression. Examples: (1+1) and 0, (x>2) or (x < 1)

5.5 Split Operation

expression:

split identifier by Text$_{\textbf{constant}}$

This operation will separate a Text constant or variable according to the delimiters specified by the Text constant. The Text constant represents a regular expression used as delimiter. For example, split “a-b-c” by “-” gives a list of Text [“a”, “b”, “c”] using “-” as delimiter.

5.6 Assignment Expression

expression:

identifier = expression

The value of the expression replaces that of the object referred to by identifier. The right operand is converted to the type of the left by the assignment if applicable. Examples: x = 1

5.7 Declaration Expression

expression:

declaration

declaration*:*

type-specifier **identifier**
type-specifier **identifier =** *expression*

Identifiers must be preceded by the type of the identifier.

6. Statements

statements -> statements statement | *ϵ*

Statements are a list of statement. Statements are executed in sequence, which are executed for their effect, and do not have values. Statements should not occur within the Literals. They fall into one of the following production:

statement -> *expression*;
| block
| **emit** (*expression*, *expression*);
| **if** *(expression) block* **else** *block*
| **foreach** *declaration* **in** *expression block*

6.1 Expression Statement

Most statements are expression statements, which have the following form:
statement:

*expression***;**

Usually expression statements are expressions evaluated for their side effects, such as assignments.

6.2 Block statement

statement:

{ *statements* **}**

Block statement is the compound statement surrounded by brackets. It groups a set of statements into a syntactic unit, so that the several statements can be used where one is

expected.

6.3 Emit Statement

statement:

emit (identifier, **identifier);**

The emit statement is used for output in map function and reduce function: both a key and a value must be emitted to the next list in the data flow.

6.4 Conditional Statement

statement:

if *(expression) block* **else** *block*
if *(expression) block*

Conditional statement chooses one of the two blocks to execute, based on the evaluation of the expression. If the expression is evaluated as true, the first sub-statement is evaluated. If the expression is false, the second sub-statement is executed.

6.5 Iteration Statement

statement*:*

foreach *declaration* **in** *expression block*

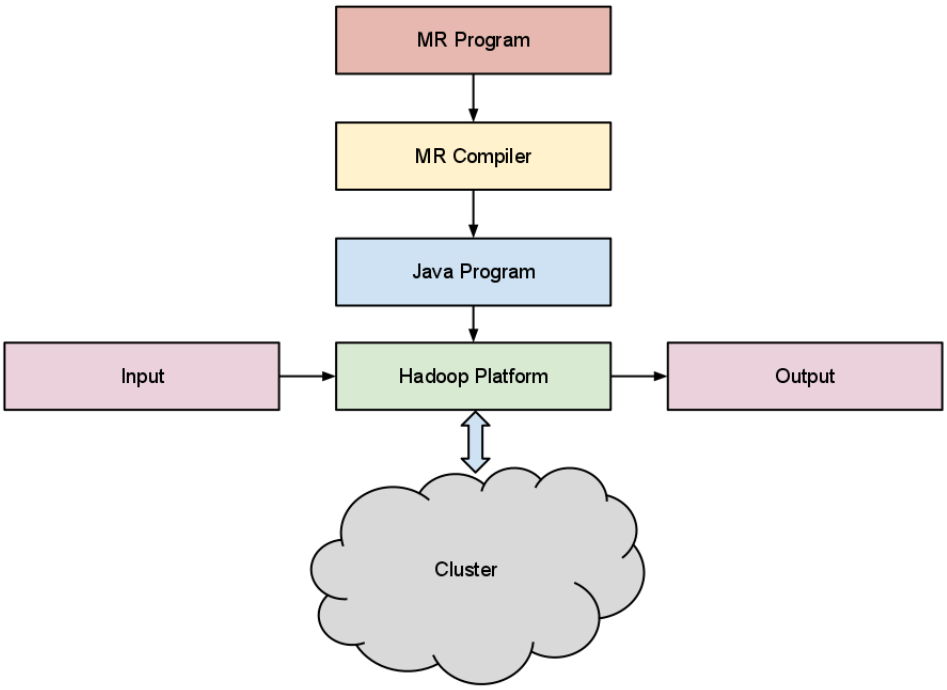
The foreach structure is used to traverse a list given by the expression. It iterates through each object in the list and execute the block statement.

7. Reference

(Concept of Mapreduce)
Hadoop Tutorial on Yahoo Developer Network, http://developer.yahoo.com/hadoop/tutorial/
Wikipedia, http://en.wikipedia.org/wiki/MapReduce

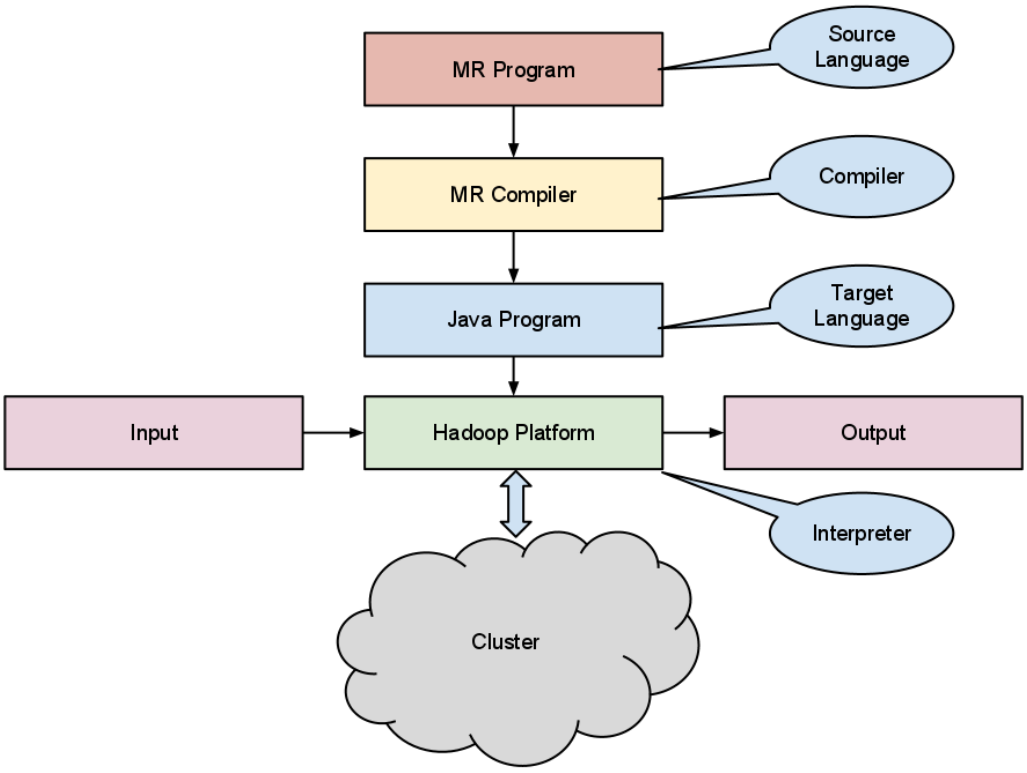
4. Architectural Design

4.1 MR Overview



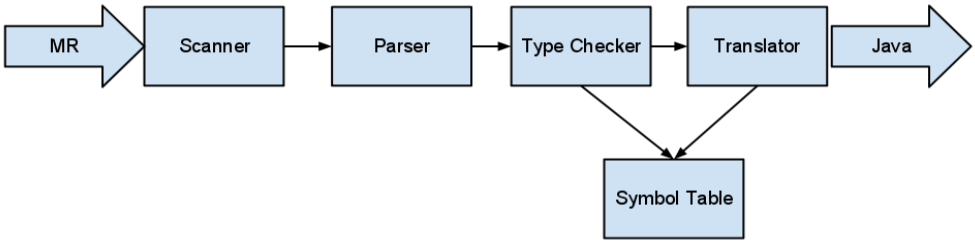
First, a file, say WordCount.mr, implementing some mapreduce algorithm, is written in MR language. The MR compiler transforms it into a Java counterpart called WordCount.java. Then the java program is fed to the hadoop framework to run on a cluster of computers.

In fact, using the compiler terminology, MR is the source language; MR compiler translates it into corresponding Java form. Java, in this case, is the target language. And the Hadoop serves as the interpreter. The relationship is shown as follows:



4.2 MR Compiler Overview

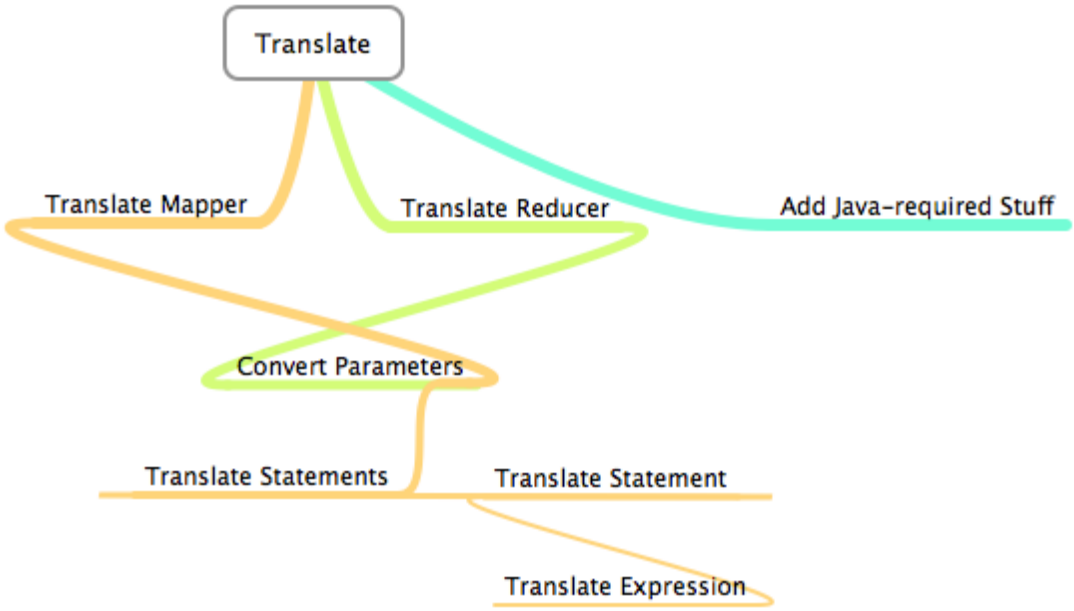
In general, a program written in MR is fed to the MR compiler, which transforms it into Java counterpart step by step. First, scanner gives the tokens sequences. Second, the parser builds up a abstract syntax tree based on the MR grammar. Third, the type checker validate the elements in the AST is legal and meaningful. Finally, the translator converts the program into Java form by walking the AST. Both type checker and translator resort to a symbol for name resolving and type conversion. The compiler structure is shown as follows:



As to the translator, it is consisted of three major parts:

1. translate the mapper
2. translate the reducer
3. add java-required stuff, i.e. main class, import etc.

Both translate mapper and translate reducer requires translate statements, which depends on translate expression. One subtlety in the translator is that we need to convert the type of the parameters before using them. The reason is we have to deal with two type systems in Java.



Parameters Type Conversion

Hadoop uses its own type system for serialization, i.e. IntWritable, DoubleWritable, Text etc.

However, these types do not support basic operations, such as addition, concatenation. In order to make use of the native support operation in Java, we have to convert all the parameters, which is default to be in hadoop types, into Java types. After that, we can perform the logic using Java operators. And finally, when we are ready to emit those values, we have to convert them back to Hadoop types. As the key solution in computer science, we solve it by adding another layer of indirection.

```
Mapper / Reducer (Parameters in Hadoop types) {
        Convert Parameters from Hadoop types into Java types
        mapper / reducer logic using Java-supported operations
        Convert values from Java types back to Hadoop types
        emit the values
}
```

5. Project Plan

Project Timeline

Milestone	Estimate Time
Language Core Features	10-02-2011
Basic Flow Design for the Project	10-11-2011
Fully Outline Grammar	10-21-2011
Complete LRM	10-27-2011
Complete Scanner Parser and AST	11-10-2011
Finish Code Generation	12-01-2011
Final Testing	12-10-2011
Complete Final Report	12-14-2011

Log

10-09-2011:

1. Analyze Hadoop features and program for Hadoop
2. Basic grammar and structure for the program
3. Sample program for the project

10-11-2011:

1. Basic flow for the program
2. Basic flow for scanner
3. Basic flow for parser

10-21-2011:

1. Build up the project on Github
2. Successful run program on Hadoop
3. Refine Grammar
4. Sample token file and sample parser tree file

10-29-2011:

1. Finish LRM
2. Code for Scanner
3. Demo code for Parser and AST is postponed because of the midterm

11-11-2011

1. Demo code for Parser and AST

2. Test case for scanner
3. Hadoop platform
4. The code for parser is postponed due to the grammar

11-13-2011

1. Grammar modified

11-27-2011

1. Scanner and parser bug fixed
2. Second test program for the compile: maximum
3. Automated script compiling java file into jar

12-04-2011

1. Type checking
2. Translation part of the compiler

12-15-2011

1. Testing for the compiler
2. Finish the final report

6. Test Plan

Checklist

Illegal Variable Definition	checked
Illegal Assignment Operation	checked
Illegal Binary Operation	checked
Ilegal Split Expression	checked
Illegal If-Expression	checked
Illegal Foreach Statement	checked
Unmatched Emit Statement	checked

6.1 Illegal Variable Defintion

Testcase1:

```
 //map function definition
 def wordcount_map(offset, line) <(Int, Text) -> (Text , Int)> : Mapper
  {
        List<Text> words = split line by " ";
        Int one = 1;
        Text tmp = 1;  //illegal vairable definition
        foreach word in words {
        emit(word, one);
        }
 }
```

Output:

Fatal error: exception Failure("illegal vdfn type")

Testcase2:

```
 //map function definition
 def wordcount_map(offset, line) <(Int, Text) -> (Text , Int)> : Mapper
  {
        List<Text> words = split line by " ";
        Int one = 1;
        Text tmp = oops;  //undefined variable oops
        foreach word in words {
        emit(word, one);
        }
 }
```

Output:

Fatal error: exception Failure("undeclared variable oops")

6.2 Illegal Assignment Operation

Testcase:

```
  //map function definition
  def wordcount_map(offset, line) <(Int, Text) -> (Text , Int)> : Mapper
  {
        List<Text> words = split line by " ";
        Int one = 1;
        Text tmp;
        tmp = 1; // illegal assign operation
        foreach word in words {
        emit(word, one);
        }
 }
```

Output:

Fatal error: exception Failure("illegal assign type")

6.3 Illegal Split Expression

Testcase:

```
  //map function definition
  def wordcount_map(offset, line) <(Int, Text) -> (Text , Int)> : Mapper
  {
        Int oops = 123;
        List<Text> words = split oops by " "; //split parameter should be a Text type
        Int one = 1;
        foreach word in words {
        emit(word, one);
        }
 }
```

Output:

Fatal error: exception Failure("expected a Text type")

6.4 Illegal Binary Operation

Testcase1:

```
  //map function definition
  def wordcount_map(offset, line) <(Int, Text) -> (Text , Int)> : Mapper
  {
        List<Text> words = split line by " ";
        Int one = 1 + "oops"; // illegal operation: Int + Text
        foreach word in words {
        emit(word, one);
        }
 }
```

Output:

Fatal error: exception Failure("illegal type of opprand")

Testcase2:

```
emit(word, one);
 //map function definition
 def wordcount_map(offset, line) <(Int, Text) -> (Text , Int)> : Mapper
  {
        List<Text> words = split line by " ";
        Int one = 35 > “a”; // illegal operation: Int *Relational Operand* Text
        foreach word in words {

        }
 }
```

Output:

Fatal error: exception Failure("illegal type of opprand")

Testcase3:

```
 //map function definition
 def wordcount_map(offset, line) <(Int, Text) -> (Text , Int)> : Mapper
  {
        List<Text> words = split line by " ";
        Int one = 0;
        Text tmp = “a”;
        if (one > tmp) // illegal relational operation
        {
                one = one + 1;
        }
        foreach word in words {
        emit(word, one);
        }
 }
```

Output:

Fatal error: exception Failure("illegal type of opprand")

6.5 Illegal If-Expression

Testcase:

```
 //map function definition
 def wordcount_map(offset, line) <(Int, Text) -> (Text , Int)> : Mapper
  {
        List<Text> words = split line by " ";
        Int one = 0;
        Text tmp = “a”;
        if (tmp) // illegal relational operation
        {
                one = one + 1;
        }
        foreach word in words {
        emit(word, one);
        }
 }
```

Output:
Fatal error: exception Failure("illegal if type")

6.6 Illegal Foreach Statement

Testcase:

```
//map function definition
def wordcount_map(offset, line) <(Int, Text) -> (Text , Int)> : Mapper
 {
        List<Text> words = split line by " ";
        Int one = 1;
        Text oops = “a”
        foreach word in oops {  //oops is not a List
        emit(word, one);
        }
 }
```

Output:
Fatal error: exception Failure("oops is not a List")

6.7 Unmatched Emit Statement

Testcase:

```
//map function definition
def wordcount_map(offset, line) <(Int, Text) -> (Text , Int)> : Mapper
 {
        List<Text> words = split line by " ";
        Int one = 1;
        Text oops = “a”
        foreach word in words {
        emit(word, oops); //emit type should correspond to (Text, Int)
        }
 }
```

Output:
Fatal error: exception Failure("illegal emit type")

7. Acknowledgement

First we would like to thank Prof. Steve for his vivid lecture and great analogies, i.e. the pac-man, which has made the tough compiler topic more accessible. Second we would like to thank Prof. Steve for his example code - the micro c compiler, which gives us a concrete idea about how to construct a compiler on our own. And finally we want to thank Yahoo Developer Network, which provides a great amount of resource about Mapreduce including the basic Mapreduce concept in our language reference manual as well as the figures.

Appendix

1. Grammar

Notation Convention:

italic = non-terminal
bold = terminal

Grammar

1. Types

type-specifier -> *atom-type-specifier*
| *list-type-specifier*
atom-type-specifier -> **Text**
| **Int**
| **Double**
| **Boolean**

list-type-specifier -> **List<***atom-type-specifier***>**

---

2. Program Structure

program -> *configuration-declaration mapper-definition reducer-definition*

configuration-declaration -> **#** *configuration-attribute* **= Text**$_{\text{const}}$**;**
configuration-attribute -> **JobName**

mapper-definition -> **def identifier** *parameters mapping-relation* **: Mapper {** *statements* **}**
reducer-definition -> **def identifier** *parameters mapping-relation* **: Reducer {** *statements* **}**

mapping-relation -> **<** (*type-specifier, type-specifier*) **->** (*type-specifier, type-specifier*) **>**
parameters -> **(identifier, identifier)**
function-type -> **Mapper**
| **Reducer**

---

3. Expression

literal -> **Text**$_{\text{constant}}$ | **Int**$_{\text{constant}}$ | **Double**$_{\text{const}}$ | **Boolean**$_{\text{const}}$

expression -> *literal*
| **identifier**
| **(**expression**)**
| **-***expression*
| *expression binop expression*
| **identifier =** *expression*
| **split identifier by Text**$_{\textbf{constant}}$
| *declaration*
binop -> **+ - * / and or < > <= >= == !=**
declaration -> *type-specifier* **identifier**
| *type-specifier* **identifier =** *expression*

4. Statement

statements -> *statements statement* | *ϵ*
statement -> *expression*;
| **emit (***expression***,** *expression***);**
| **if** *(expression) statement*
| **if** *(expression) statement* **else** *statement*
| **foreach identifier in identifier** *statement*

2. Contribution

Siyang Dai
1. Project Motivator
2. Architectural Design
3. Grammar Design
4. Parser Implementation
5. Translator Implementation

Zhi Zhang
1. Type Checking Implementation
2. Test Plan Design and Implementation
3. Demo MR program (Wordcount)
4. Sample Token File

Zeyang Yu
Design: The basic structure and grammar for the language.
Coding: parser and ast

Shuai Yuan
1. Hadoop semantics
2. Language statement grammar
3. Wordcount tutorial
4. Scanner debug

Jinxiong Tan

1. Scanner
2. compiler script

3. Lessons Learned

Siyang Dai
1. Think Recursively
For parser and translator, especially for tree structure, thinking recursively eliminates unnecessary attention to details.
2. Key - another level of indirection
Conversion between Hadoop type, Java type, MR type as mentioned in the Architectural Design section.
3. Parametric Polymorphism is a magic for writing compilers, which is much easier than using Subtype Polymorphism.

Zeyang Yu

1. Basic structure of the compiler and how each part works
2. Programming in Ocaml
3. Use LR(0) automaton to debug the parser
4. The mechanism of Hadoop
5. Team work: divide the work wisely so that the work can be done separately.

Zhi Zhang
1.Incremental Development and Testing
Abundant rules and expression need to be checked in this project, I realize that if I write the whole program before I start any testing, then this process will become very tedious and annoying. Therefore, I chose to write basic function module first and carry out the test plan early. By iteratively appending more complex function into the program, I make the process of development controllable and productive.
2. Everything is function.
It is my first time to use functional programming language to implement a big project. I have to reshape my mind in writing code, and consider everything as the input/output of a function. Recursive functions are widely used in this project. Especially in my work of type checking, the correctness of expression could only be checked by recursive function.

Shuai Yuan

The project makes me understand more about compiler. Though we have learned a lot of knowledge in the class, when we are facing to design our own language, it is a challenging work. As far as I'm concerned, the following three aspects are most important for me:
First, we should understand the principles of compiler deeply before implementing our project. In order to know much more about it, we should not only focus on the lectures related to it, but also analysis the micro C example provided by professor. It's a nice example for us to know the real world design and implement.
Secondly, in order to implement the project, Ocaml is a very important language for us. I feel

difficult to learn at first. And afterwards, I find it's really useful.
Finally, effective communication among team members is necessary for the teamwork, We can always conclude the best ideas after our discussions and meetings.

Jinxiong Tan

In this course, I have learned the following three things.
1. Ocaml and the recursive method. Before learning Ocaml, I have very little knowledge about recursion. I can only handle the problems, say transversing an array, with loops. Ocaml, in my perspective, provide me with the recursive ways to think and deal with problems; Moreover, recursion is quite helpful and effective in writing a compiler;
2. More than the knowledge of compiler itself, I have got a clearer picture of the principals of the computer, for example, what would happen to the stack when the functions are calling themselves, and how the data is stored in the memory;
3. Besides, since our projects are associated with distributed computation, I also learned how distributed computation can be performed. For example, we can implement distributed computation by programming in Java and run the Java program in Hadoop framework.

4. Code Listings

Makefile
ast.mli
scanner.mll
parser.mly
typechecker.ml
translator.ml
mrc
    |-lib
        header.mrlib
        main_template.mrlib
        mapper_template.mrlib
        reducer_template.mrlib
    |-sample
        wordcount.mr
        WordCount.java