**COMS 4115 Programming Languages and Translators**
Ryan DeCosmo (rd2680)
Olesya Medvedeva (oam2113)
Jyhyun Song (js4390)
Charis Lam (cl3257)

# Language Reference Manual: MiniMap (MML)

## 1. Introduction

MiniMap Language (MML) is a distributed text processing language that aims to provide the user with an interface for programming in parallel with its standard library (this is demonstrated in the sample code at the end). MML draws from Scala so it is type-safe-- a priority for a small parallel programming system.

The eventual goal [1]with MML is to be able to create an interface that uses a CPU master to harness the GPU as worker nodes. This structure would allow any user with a GPU to have a mini Hadoop cluster to process text quickly and in parallel.

## 2. Lexical conventions

A. Identifiers

- consist of a string of ASCII letters, digits, underscores
- is case-sensitive
- first character must be a letter
- identifier classification based on type (details to follow)

B. Keywords

```
auto, break, catch, char, chunk, const, continue,
default, do, double, else, enum, extern, float, for,
goto, if, int, long, main, map, match,case, register,
return, short, signed, sizeof, static, struct, try,
object, extends, var, val, typedef, union, unsigned,
void, volatile, while, finally, _, : , = , => ,<-, ->
```

[1] Perhaps outside of the scope of this class, but a stretch goal.

C. Delimiters/Separators and Whitespace:
    a. Tokens may be separated by whitespace characters and/or comments.
    b. Both single and multiline comments are supported:
        i. `// the body of the single line comment`
        ii. `/* the body of the multiline comment */`
    c. Other delimiters:
        i. Delimiter characters
            1. Terminate statements: `;`
            2. Separating elements: `,`
            3. Chars: `‘’`
            4. Strings: `“”`
            5. Access operator: `.`
        ii. Parentheses
            1. Tuples, lists: `()`
            2. Arrays: `[]`
            3. Blocks: `{}`
        iii. Newline characters: `nl {nl}`

**3. Data types**

- all data types are immutable and may be un-initialized at declaration

A. Primitives

| Data Type & Description |
| --- |
| **Byte**<br>8 bit signed value. Range from -128 to 127 |
| **Short**<br>16 bit signed value. Range -32768 to 32767 |
| **Int**<br>32 bit signed value. Range -2147483648 to 2147483647 |
| **Long**<br>64 bit signedvalue. Range -9223372036854775808 to 9223372036854775807 |
| **Float**<br>32 bit IEEE 754 single-precision float |
| **Double**<br>64 bit IEEE 754 double-precision float |
| **Char**<br>16 bit unsigned Unicode character. Range from U+0000 to U+FFFF |
| **String**<br>A sequence of Chars |
| **Boolean**<br>Either the literal true or the literal false |
| **Null**<br>null or empty reference |

B. Other primitives (as of now) with the built-in functions:
    a. Arrays:
    - store a fixed-size sequential collection of elements of the same type
    - declaration syntax:

```
var z:Array[data type] = new Array[data type](number of elements)
or
var z = new Array[data type](number of elements)
```

    - declaration with values syntax:

```
var z = Array(expr, expr, expr)
or
z(0) = expr;
z(1) = expr;
z(2) = expr;
```

    b. Tuples:
    - unlike an array or list, a tuple can hold variables of different types
    - tuples are indexed at 1(for example, `Tuple._1` returns the first element in the tuple)
    - syntax: `‘(’ [exprs] ‘)’`

**4. Variables**

- MML mostly uses immutable values defined as `val = x`
- creating a variable introduces a mutable variable of type T
- variable definition syntax: `var x = new T()`,
- with MML, variables should be used minimally, for example when creating a new job configuration:

```
//the backend for processing a distributed text file
var conf = new JobConfiguration()
//path to desired output location for the processed file
val fileOut = FileOutputFormat.setOutputPath(args(1))
```

**5. Expressions**

A. Assignment:

- syntax: x = e

- if x is mutable, then the assignment changes the current value of x to be the result of evaluating the expression e. The type of e is expected to conform to the type of x.

- if x is immutable, then a new x is created with the value e and the old instance of x becomes inaccessible

B. Operators: listed in the order of precedence

| Category | Operator |
|---|---|
| Postfix | () [] |
| Unary | ! ~ |
| Multiplicative | * / % |
| Additive | + - |
| Shift | >> >>> << |
| Relational | > >= < <= |
| Equality | == != |
| Bitwise AND | & |
| Bitwise XOR | ^ |
| Bitwise OR | \| |
| Logical AND | && |
| Logical OR | \|\| |
| Assignment | = += -= *= /= %= >>= <<= &= ^= \|= |

**6. Statements**

A. Control Flow Statements

- if expression in if-clause evaluates to true, execute statements
- if expression evaluates to false, execute statements in else-clause
- syntax:

```
if (boolean expression) {
     //List of statements
} else {
     //List of statements
}
```

B. Loop Statements

a. While

- continues to execute list of statements in the block as long as boolean expression evaluates to true
- syntax:

```
while (boolean expression) {
     //List of statements
}
```

b. Do-While

- list of statements execute at least one time then boolean condition is checked at the bottom of the loop
- syntax:

```
do {
     //List of statements
}
while (boolean expression);
```

c. For

- list of statements in the loop are executed until condition in expression is no longer true
- syntax:

```
for (expression) {
     //List of statements
}
```

d. Break

- immediately terminates block and program control resumes at the next statement following the block
- syntax:

```
for (expression) {
     //List of statements
     break;
     //List of statements
}
```

C. Pattern Matching

- checks the value against the patterns of a sequence of alternatives,
- each starts with keyword ‘`case`’
- if pattern matches, executes expression after `=>` symbol
- syntax:

```
x match {
     case 0 => "zero"
     case 1 => "one"
     case 2 => "two"
     case _ => "many"
}
```

**6 Functions**

- function declaration syntax:

```
def functionName (list of parameters):[return type]
```

- function definition syntax:

```
def functionName (list of parameters):[return type]={
     function body;
     return [expr];
}
```

- parameters: a list of zero or more variables separated by commas
- return types: optional; any valid data type
- function call syntax:

```
functionName(list of parameters);
```

## 7. Scope

- defines whether variables, functions and classes are accessible at a given point of the program
- there are no globally accessible variables in our program
- functions have global scope and must be declared with unique identifiers
- scope for blocks (ie. body of if statements and loops) are defined by curly braces: {}
- identifiers declared within curly braces { } have a local scope and are not accessible from outside

## 8. MML Standard Library

A. FileTypeIndicator
   a. CSV
      i. Splits based on ‘,’
   b. TextFile
      i. Splits based on "\r|\n"
   c. CustomDelimiter
      i. Splits based on the user defined delimiter

B. MiniMapReduce Interface
   a. Splitter(FileTypeIndicator):String
      i. The splitter decides how to split up a text file; either predefined or user defined. If the user chooses TextFile or CSV, we will split on predefined criteria
      ii. If the user chooses to make it a UDF, then they can specify the delimiter they wish to split on.
   b. Mapper(inputToken:String):UDF-MapReturnType
      i. This method takes in the output of the splitter method and performs an operation on the token. It then returns a user defined type
   c. Reducer(inputTokens[MapReturnType]):String
      i. This method takes in an aggregated portion of the Mapper’s output. It then performs a secondary operation on the output and transforms it into a String to be written to file.

C. String class with appropriate methods as in scala

D. Files I/O class
    a. FilePath(String)
        i. Takes a string and transforms it into a file path
    b. Open(FilePath)
        i. Takes a file path and opens the specified file
        ii. If file not found, throws an IOException
    c. Write()
        i. Writes some information to a file
        ii. If this fails, throws an IOException
    d. Close()
        i. Closes a file safely
        ii. If this fails, throws an IOException

E. Lists
    - a list is a linked list of elements, unlike an array, size does not need to be known beforehand.
    a. `List.size()`
        i. Returns the size of the List
    b. `List.get()`
        i. Gets element in the list by index
            1. I.e. List.get(1) - returns element 1
    c. `List.foreach()`
        i. Iterates through each element of a list to apply a function or operation
    d. `List.toArray()`
        i. Creates an array of size List.size() and copies the elements of the list to that array
    e. `List.remove(index)`
        i. Removes the element of a list at the specified index

F. Maps:
- a map is a collection of key/value pairs. A value is retrieved by giving its key.
    a. `get(Key)`
        i. Returns the value associated with the key
    b. `getOrElse(Key,"Default If not found")`
        i. If there is no value associated with the key, returns the default value
    c. `put(k,v)`
        i. Puts a value (v) into the map and associates it with the key (k)
    d. `remove(k,v)`
        i. Removes the specified key value pair from the map

**9. Sample Codes (hello, world & parallelization)**

This is a program that finds if a word is in a line of text, and if it is, writes that line to a file

```
/*
* The purpose of this code is to define a sample program for the MiniMapLanguage
* We have incorporated ideas from the Map Reduce programming paradigm with a Scala like language
* In doing so we hope to achieve a language designed to do small batch processing on text files
* The stretch goal is to have the worker nodes live on the GPU to increase speed for users
*/

object SampleMiniMap extends MiniMapReduce {

    //here user can explicitly define resulting tuple types of their map and reduce functions
    type mapReturnType = (Int,String)
    type reduceReturnType = (TextFile)
```

```
        //the splitter takes in a file type and splits the file into jobs for the nodes
        def Splitter(typeOfFile:FileTypeIndicator):String {

                //File types: TextFile, CSV, CustomDelimiter("|") <-note the custom
                delimiter function takes in a custom delimiter and splits a file on that
                if(typeOfFile == FileTypeIndicator.TextFile){
                        //the tokenize string function is built into MML and splits the file
                        based on a regex
                        tokenizeString("\r|\n")
                } else {
                        "Operation Undefined"
                }
        }

        //we will take in a TextFile for this example (alternative: CSV file)
        def Mapper(inputToken:String):mapReturnType
                //first, the text file is presplit by the delimiter or file type
                //then, the single line of text is checked on a worker node for the target
                string
                if(inputToken.contains("TheWordImLookingFor")){
                        //if word is found, return key and the line number (here the key
                        also acts as a count)
                        return (1,inputToken)
                } else {
                        //if word not found, return key, count and “not found” message
                        return (0,"NotFound")
                }
        }
 }

def Reducer(inputTokens:[mapReturnType]):File {
        //reducer receives a small batch of mapped lines (aggregated by the Job in Main)
        //reducer then checks the returned mapper Key Value pairs
        for(token<-inputTokens){
                //if key >0, mapReturnType is successful
```

```
            if(token._1 > 0){
                  //if success, add the line to output file
                  File.Write(Line)
            } else {
                  //in this case, if unsuccessful, do nothing
            }
      }
}

def main(args: Array[String]) {
      //path to input file / name
      val fileIn  = FileInputFormat.setInputPaths(args(0))
      //path to desired output location for the processed file
      val fileOut = FileOutputFormat.setOutputPath(args(1))

      //the backend for processing a distributed text file
      var conf = new JobConfiguration()
      //name of the job
      conf.setJobName("SampleMiniMap")
      //let the job know location of the file that needs to be split
      conf.setInputFile(fileIn)
      //let the job know where to accumulate
      conf.setOutPutFile(fileOut)
      //optional line for setting the batch size for each reducer
      conf.setBatchFactor()
      //pass the splitting method defined above
      conf.setSplitter(Splitter(_))
      //pass the mapper function defined above
      conf.setMapper(Mapper(_))
      //pass the reducer function defined above
      conf.setReducer(Reducer(_))
      //pass the inputFormat (in this case Text File)
      conf.setInputFormat(Type.TextFile)
      //pass the output format
      conf.setOutputFormat(Type.TextFile)
```

```
        //the actual job client class runs the job itself
        JobClient.runJob(conf);
}
}
```

```
/*
* The purpose of this code is to define a sample program for the MiniMapLanguage
* We have incorporated ideas from the Map Reduce programming paradigm with a Scala like language
* In doing so we hope to achieve a language designed to do small batch processing on text files
* The stretch goal is to have the worker nodes live on the GPU to increase speed for users
* */

object SampleMiniMap extends MiniMapReduce {

  //here a user can explicitly defines the tuple types their map and reduce functions will produce
  type mapReturnType = (Int,String)
  type reduceReturnType = (TextFile)


  //The splitter takes in a file type and splits the file for the job to dish out to nodes
  def Splitter(typeOfFile:FileTypeIndicator):String {
    //File types: TextFile, CSV, CustomDelimiter("|") <-note the custom delimiter function takes in a custom delimiter and splits a file on that
    if(typeOfFile == FileTypeIndicator.TextFile){
      //the tokenize string function is built into the language and will splut the file based on a regular expression
      tokenizeString("\r|\n")
    } else {
      return "Operation Undefined"
    }
  }


  //in our string processing language we will take in a TextFile for this example ( example alternative: CSV file)
  def Mapper(inputToken:String):mapReturnType {
    //The text file is presplit by the delimiter or file type
    //then the single line of text is checked on a worker node for the target string
    if(inputToken.contains("TheWordImLookingFor")){
      //if we find the word we are looking for we return the key which also acts as a count here + the line we found it in
      return (1,inputToken)
    else{
      //if we do not find the line we return the key / count + the message not found
      return (0,"NotFound")
    }
  }

  }

  def Reducer(inputTokens:[mapReturnType]):File {
    //the reducer receives a small batch of mapped lines (this is aggregated by the Job in Main)
    //the reducer then checks through those Key Value pairs we returned in our mapper
    for(token<-inputTokens){
      //we check if the mapReturnType is a successful line by checking if the key is > 0
      if(token._1 > 0)
        //if it is we add the line in question to our output file
        File.Write(Line)
      else{
        //if we didnt find what we are looking for, we do nothing in this case
      }
    }

  }
```

```
56
57    def main(args: Array[String]) {
58      //we set a path to the input file / name
59      val fileIn  = FileInputFormat.setInputPaths(args(0))
60      //we set a path to the output of where we want our processed file to be written
61      val fileOut = FileOutputFormat.setOutputPath(args(1))
62      //this is the backend for processing a distributed text file
63      var conf = new JobConfiguration()
64      //we set the name of the job
65      conf.setJobName("SampleMiniMap")
66      //let the job know where the file we need to split is
67      conf.setInputFile(fileIn)
68      //let the job know where to accumulate
69      conf.setOutPutFile(fileOut)
70      //this might be an optional thing where we can set the batch size for each reducer
71      conf.setBatchFactor()
72      //we pass the splitting method defined above
73      conf.setSplitter(Splitter(_))
74      //we pass the mapper function defined above
75      conf.setMapper(Mapper(_))
76      //we pass the reducer function defined above
77      conf.setReducer(Reducer(_))
78      //we pass the inputFormat (in this case Text File)
79      conf.setInputFormat(Type.TextFile)
80      //we pass the output format
81      conf.setOutputFormat(Type.TextFile)
82
83      //the actual job client class runs the job itself
84      JobClient.runJob(conf);
85
86    }
87
88
89  }
```