Hahn Chong - hc2361
Fred Clark Jr. - fc2413
Rotem David - rd2499
Robert Tolda - rmt2131
Jose Rodriguez - jgr2128 - Team Leader

Proposal - PB & J (Parallel Boxes and Jam)

Table of Contents

1 Introduction
2 Types
    2.1 Primitive types
    2.2 Collection types
    2.3 Special types
3 Lexical Structure
    3.1 Comments
    3.2 Operators
        3.2.1 PB&J operator (@)
        3.2.2 Assignment operator : identifier <- expr
        3.2.3 Return operator : expr ->
    3.3 Literal Types
        3.3.1 Literal primitive types
        3.3.2 Literal data types
    3.4 Keywords
    3.5 Arrays and Maps
4 Expressions
    4.1 Operators
        4.1.1 Multiplicative Operators
            4.1.1.1 Multiplication Operator
            4.1.1.2 Division Operator
            4.1.1.3 Remainder Operator
        4.1.2 Additive Operators
            4.1.2.1 Addition Operator
            4.1.2.2 Subtraction Operator
        4.1.3 Comparison Operators
        4.1.4 Inequality Operators
        4.1.5 Logical Operators
        4.1.6 String Concatenation

5 Blocks and Statements
- 5.1 Blocks
- 5.2 Spread Statement
- 5.3 Jam Statement
- 5.4 Conditional Statement
- 5.5 While Statement
- 5.4 For Statement
- 5.5 Print Statement

6 Declarations & Identifiers
- 6.1 Function Declarations
  - 6.1.1 Master function
- 6.2 Variable and Constant Declarations
  - 6.2.1 Variable and Constant Initialization

7 Scoping
- 7.1 Global Scope
- 7.2 Block Scope

8 Execution
- 8.1 Slave daemon
- 8.2 Master Execution

1 Introduction

Distributed computing is the use of multiple autonomous computers which work in concert to achieve a desired output.

PB&J is designed for developers to distribute a job amongst multiple computers/processors in a minimal amount of code needed while still being readable.

In PB&J, developers define a program which is run on both the master and slave servers. While a portion of the program is only run on the master, the slave waits for messages from the master as to what functions to run on what data sets.

It's the responsibility of the master server to distribute the job to the slave servers. A developer making a program in PB&J can use the spread & jam statements to perform these actions.

Refer to section 8 on how execution works for masters and slaves.

2 Types

2.1 Primitive types

PB&J supports the following primitive types:

long	Basic numeric type.
double	Floating point value and is declared by adding a decimal point to an integer.
boolean	True or false expression.
string	Sequence of characters.

2.2 Collection types

PB&J also supports more complex data types, known as collection types, that follow the syntactical structure of JSON.

Array	A fixed position of sequential data types.
Map	A key/value based data structure. Where keys can be any of the defined data types.

2.3 Special types

PB&J contains the following special values:

null	A special type that can be assigned to any variable type in place of it’s allowed value range.

3 Lexical Structure

3.1 Comments

In PB&J there are only single line comments. Comments are generated one line at a time, by placing three consecutive periods (...).

Example:
…This is a comment in PB&J.
… This is also a comment in PB&J.
.. . This is not a valid comment in PB&J

3.2 Operators

3.2.1 PB&J operator (@)

Spread: @ is used to identify an explicit argument that is a collection to spread amongst the slave servers with
Jam: @ is used to identify the parameter in which to deliver the spread result.

Refer to section 5 for more details on spread and jam.

3.2.2 Assignment operator : *identifier <- expr*

The assignment operator consists of <- to represent you are injecting the left identifier to the right expression.

3.2.3 Return operator : expr ->

The return operator (->) returns the value of the expression on the left of the operator.

3.3 Literal Types

3.3.1 Literal primitive types

The following are examples of literal primitive types:

long	0 10 1
double	0.0 10.0 1.1
boolean	true, false
string	“Hello World”

3.3.2 Literal data types

DCC also supports more complex data types following the syntactical structure of JSON.

array	An example of an Array of longs: [1, 2, 3, 4]

map	An example of a map with keyed by longs (similar to the list above):
	{ 1 : 1, 2 : 2, 3 : 3, 4 : 4}
	
	An example of a map keyed by strings:
	{ “one” : 1, “two” : 2, “three” : 3, “four” : 4}

3.4 Keywords

The following keywords, formed from ASCII characters, are reserved and can not be used as identifiers:

```
jam spread long double boolean string array map null if else while for
                              print global
```

3.5 Arrays and Maps

3.5.1 Arrays

Arrays a series of random access values that have a fixed length and are accessed by a number from 0 to length - 1.
Where “$value_n$” is value in the array, “index” is a number which corresponds to an address in the array, and “size” is the wanted size of the list as a long, and a is the identifier for the array.

Create an empty array	array a <- []
Create with values	array a <- [$value_1$, $value_2$, $value_n$]
Get the value for a given index	a[index]
Add or replace a value	a[index] <- value
Get length of the array	|a|

3.5.2 Maps

Where “key_n” is a wanted key in the form of a string or long and “$value_n$” is the wanted value for that key and m is the identifier for a wanted map.

Create an empty map	map m <- {}
Create with values	map m <- { key_1 : $value_1$, key_2 : $value_2$, key_n : $value_n$}
Get the value for a given key	m{key}
Add or replace the value for a key	m{key} <- value

Remove a key and it’s value	m{key} <- null
Get all the keys as an array	m*
Get all the values as an array	m{*}
Get length of map	|m|

4 Expressions

4.1 Operators

All Mathematical Operators follow traditional order of precedence with remainder ordered on the same level as division and multiplication.

4.1.1 Multiplicative Operators

Operators * and / are known as multiplicative operators. Multiplicative operators are allowed on numeric primitive data types and show described as follows:

4.1.1.1 Multiplication Operator

The * operator is used for multiplication of all numerical primitive data types. The following is a typical format for a multiplication operator:

*expr * expr2*

Multiplication Examples:

Expression	Result
1 * 2	2
1.0 * 2.5	2.5

4.1.1.2 Division Operator

The / operator is used for division of all numerical primitive data types. The following is a typical format for a division expression:

expr / expr2

Division Examples:
Division of longs rounds towards 0.

Expression	Result
1 / 2	0
3 / 2	1
3.0 / 2.0	1.5

4.1.1.3 Remainder Operator

The % operator is used for finding the remainder of all numerical primitive types. The following is a typical format for a remainder expression:

expr % *expr2*

Remainder Examples:
Division of integers rounds towards 0.

Expression	Result
4 % 2	0
4 % 3	1
2.5 % 2.0	0.5

4.1.2 Additive Operators

The + and - are known as additive operators.

4.1.2.1 Addition Operator

The + operator is used for addition of two expressions:

expr + *expr2*

Addiction Examples:

Expression	Result
1 + 2	3
1.0 + 0.5	1.5

4.1.2.2 Subtraction Operator

The - operator is used for addition of two expressions:

expr - expr2

Subtraction Examples:

Expression	**Result**
1 - 2	-1
1.0 - 0.5	0.5

4.1.3 Comparison Operators

We are dropping the == operator as to not create confusion.

=	Structural comparison
===	Physical comparison

4.1.4 Inequality Operators

Inequality operators can be used on the numeric types: long and double.

PB&J supports the following inequality operators:

>	Greater than
>=	Greater than or equal to
<	Less than
<=	Less than or equal to

4.1.5 Logical Operators

Since our language does not contain bitwise operators, we can simplify our logical operators to one character. Operators are still short-circuit operators.

&&	And operator
||	Or operator

4.1.6 String Concatenation

The ~ (tilde) is used to concatenate data types into a string.

Examples:

Expression	*Evaluation*
4 ~ 2	“42”
“Hello “ ~ 0.0	“Hello 0.0”

5 Blocks and Statements

5.1 Blocks

The start of a block is defined by open brace ({) where { is not following the identifier for a map. The end of a block is defined by close brace (}) where } is not preceded by an unclosed hash’s {. In the case of nested blocks } closes the last opened block.
Blocks for functions and statements are defined using braces { }

5.2 Spread Statement

`spread`: *func*
The spread statement is used to distribute a collection type amongst registered slave machines. The function acts as a callback returning each time a slaving reports results, but only blocks the call for the first result. *func* is a reference to the function that runs on the spread collection.

Example usage of spread for prime factorization:

```
master(map slaves, array args){ ... Runtime argument.
      long bigPrime <- args[0];
      array searchStarts;

      ... get the place for each slave to start
      long iterations <- bigPrime / 3 / |slaves|;
      for(long m <- 0; m < |slaves|; m <- m + 1) {
            searchStarts[i] <- iterations * m;
      }

      ... spread the starting points to the slaves
      long result <- spread: factor(@searchStarts, bigPrime, iterations);
      print("Result: " ~ result);
}
```

```
long factor(array mylist, long prime, long iterations) {
      long start <- mylist[0];
      for(long i <- start; i < iterations + start; i <- i + 1) {
            if(prime % i = 0 ) { ...it is not prime
                  i ->
            }
      }
      null -> ...no evidence that it is not prime
}
```

5.3 Jam Statement

`jam`: *func*
The jam statement is used to block on a spread statement until each slave reports a result. It will then return the results of the spread or the result of the given function. *func* is an optional reference to a function, which is run on the resulting collection set.
Note that jam: always precedes a spread statement with only an optional function reference in between.

Example usage of jam for addition of a list :
```
master(map slaves, array args) {
      array myList <- [1, 2, 3, 4, 5, 6];
      long result <- jam: add(@) spread: add(@myList);
      print("Result " ~ result ~ "\n");
}

long add(array myList) {
      long result <- 0;
      for(long i = 0; i < |myList|; i <- i + 1) {
            result <- result + myList[i];
      }
      result ->
}
```

5.4 Conditional Statement

The conditional statement has two forms:
`if(`*expr*) *statement1*
`if(`*expr*) *statement1* `else` *statement2*
In both cases *expr* is a boolean expression that is evaluated first and *statement1* will be executed if *expr* evaluates true. In the second case, *statement2* is executed if expr evaluates false.

5.5 While Statement

`while(`*expr*`)` *statement*
The while statement repeatedly executes *statement* until *expr* evaluates false. *expr* is a boolean expression that is evaluated before the statement is executed.

5.4 For Statement

`for(`*expr1*; *expr2*; *expr3*`)` *statement*
The for statement first evaluates *expr1* once. *expr2* is a boolean expression that is evaluated before each iteration. *statement* is executed whenever *expr2* evaluates true and *expr3* is evaluated after each time *statement* is executed. If *expr2* evaluates false, the for statement is terminated. Any or all of the expressions may be left empty. If *expr2* is left empty, it will be evaluated as true.

5.5 Print Statement

`print`(*string*)
The print statement will print *string*. Other types will be converted to a string and printed.

6 Declarations & Identifiers

Both function declaration and variable declarations require a valid identifier. A valid identifier must start with an alphabetic unicode character followed by any sequence of alphanumeric unicode characters. An identifier may not be one of the reserved keywords. Two identifiers are the same, if and only if, they have identical unicode characters for each letter of digit.

Examples of identifiers
long a … a is a valid identifier
long b123 … b123 is a valid identifier
long 1ba … 1ba is NOT a valid identifier as it does not start with an alphabetic character

6.1 Function Declarations

A function is a body of executable code which is passed a specific number of parameters. To declare a function into a program specify a unique identifier. The identifier is used to refer to the function for the duration of the program. Function declarations must follow the following format:

[type] *identifier*(**type** param [,...]) {
        … body of function

```
}
```

When defining a function, it is optional to provide a type before the identifier to define the return type of the function. It is also acceptable to leave out the return type if the function has no return.

6.1.1 Master function

For each program to run on a master node, a function with the identifier **“master”** is required. A **“master”** function must take two parameters. The first parameter must have a map type. The map corresponds to a map of server objects. The second parameter being an array of strings contains any additional command line arguments when ran. A **“master”** function must follow the following format:

```
master(map servers, array params) {
        … body of master function
}
```

Where the identifiers, servers and params, are interchangeable.

6.2 Variable and Constant Declarations

Variables and constants are declared by a unique *identifier* with the scope (in the case of constants) and data type, respectfully, specified before the *identifier*. The scope determines where the variable or constant is accepted by the compiler. Declare a variable without the global keyword the context of a block to specify a local variable. Use the specifier *global* before the type declaration to declare a global constant which cannot be changed and is accessible by the master and all servers. The global specifier can only be used outside of the block scopes.

Note that this means the only way how to declare something as constant defines it as global and visa versa.

[specifier] type ***identifier***

6.2.1 Variable and Constant Initialization

Variables and constants in PB&J can be initialized with a given literal value or another already declared variable or constant that can be seen within the scope of the initializing variable. Any variable or constant that is not provided a value with the declaration is initialized one of the following default values:

Data Type	Default Value
long, double	0, 0.0
boolean	false

string	“” (empty string)
array	[] (empty array)
map	{} (empty map)

7 Scoping

7.1 Global Scope

The global constants are part of the global scope and can be accessed in all of the functions in the file. defined by the keyword “global” followed by the type and name of the constant.

Global Scope Usage:

```
global double newresult <- 10.0;

master(double d) { ... d is a local variable.
      print("Result: " ~ newresult); ... newresult can be accessed in
      ... the function because it’s a global variable
}
```

7.2 Block Scope

Each block defines it’s own local scope. Blocks have access to their parent block’s local variables that have already been defined, as well as global variables.

Block Scope Usage:

```
foo( double a) { ... a is a local variable.
      double b <- a;  ...b is a local variable
      if(true) {
            double d <- a; ... Legal, the if block has access
            ... to it’s parent block’s variables.
      }
      b <- d; ... Illegal - d is not in the scope of this block.
}

master(double c){ ... c is a local variable.
      c <- b + 3.0;  ... ILLEGAL - b is a local variable
      ... in the foo function and thus cannot be
      ... accessed in the master function
}
```

8 Execution

Master and slave nodes of PB&J both execute the same version of compiled code to run. Execution of a PB&J compiled application happens in two phases. The first phase is to initialize the program as a slave daemon on all available slaves. The second is to initialize the master program with the desired slaves as an argument.

8.1 Slave daemon

To initialized the slave daemon, you simply run the compiled version of your program with the -*slave* argument with an optional port number. When initialized without a port number the slave defaults to TCP port 35000.

```
slave$ programName -slave [port_number]
```

8.2 Master Execution

When a program is executed without the -slave argument then by default it'll run the **"master"** function of the program with a list of acceptable slaves

```
master$ programName ip[:PORT];ip2[:PORT];...
```