pubCrawl
Language Reference Manual
COMS W4115

Matt Dean, Sireesh Gururaja, Kevin Mangan, Alden Quimby
mtd2121, skg2123, kmm2256, adq2101

October 28, 2013

Contents

Chapter 1

Introduction

pubCrawl is a distributed systems programming language with a focus on list operations. Given some number of networked slave computers, pubCrawl helps the user automatically split work up among them requiring the user only to specify whether a function should distribute work or not. pubCrawl has been designed from the ground up to make taking advantage of the power of distributed programming as easy as possible. To this end, pubCrawl code is syntactically imperative but function calling is side effect free. This allows developers to get writing pubCrawl code quickly but also makes distributing work across slaves easy. Developers won't have to worry about shared resource locking or race conditions.

1.1 Why is it called pubCrawl?

It's Saturday and the world is your oyster, why stay put the entire night? Since 1915, the term "pub crawl" has been defined by *The New Partridge Dictionary of Slang and Unconventional* as "A drinking session that moves from one licensed premise to the next, and so on." Just as a pub crawl is all about making the most out of the diversity of drinking establishments available, pubCrawl the language is about taking advantage of the many devices the modern developer has available to him or her.

1.2 What Constitutes a Program

A pubCrawl program is simply a list of statements. The syntax of a valid statement is defined in detail in a later section.

Chapter 2

Types and Type Inference

2.1 Type Inference

Data in pubCrawl is expressed using a finite and well defined set of data types. However when writing a pubCrawl program, it is not necessary to explicitly declare types. In this sense pubCrawl is *type inferred* as opposed to *dynamically typed*. The following sections outline the valid types in pubCrawl.

2.2 Primitive Types

There are only three primitive types in pubCrawl: **booleans**, **numbers**, and **characters**.

2.2.1 Booleans

A boolean in pubCrawl is defined by the `true` and `false` keywords. Booleans are considered their own type, meaning that an expression that uses a boolean operator and a non-boolean variable will cause an error. For example, !3 will cause an error.

2.2.2 Numbers

A **number** in pubCrawl is a 64 bit piece of information representing values between 4.9E-324 and 1.7976931348623157E308. This is similar to the double class commonly found in other languages. Because there is no "int" type and all numbers in pubCrawl can have a fractional part, when using a number in a situation where having a fractional part would be inappropriate – such as when accessing a specific index of a collection – the fractional part will be ignored.

2.2.3 Characters

Much like in the C programming language, a string is not a primitive type in pubCrawl but rather a collection of characters. This allows manipulating strings to take advantage of the powerful and easy to use collection syntax and functionality found in pubCrawl. Luckily and in contrast to C, using the character primitive type will feel much like using a string in other programming languages such as Java. This is because characters can only ever exist within the context of a collection. To simplify discussion, character collections will be referred to as strings.

2.3 Non-Primitive Types

2.3.1 Collections

pubCrawl has been designed to make creating and manipulating collections as simple as possible. A collection is a list of items that maintains the order with which it was created much like an array. Items in a collection

may be of any type, although the type must remain consistent through the life of the collection. For example, attempting to put a number at the end of a collection that contains objects is not allowed.

2.3.2 Objects

Objects in pubCrawl are built using an adapted JSON syntax. Json.org writes that JSON is, "Easy for humans to read and write" but also, "Easy for machines to parse and generate." JSON notation has become the data object representation standard for networking applications and noSQL databases. pubCrawl's JSON syntax is very similar to javascript's.

2.3.3 Functions

In pubCrawl, functions are first class objects. They can be assigned to variables, and passed as arguments to other functions.

Chapter 3

Lexical Conventions

3.1 Identifiers

An **identifier** is a sequence of letters, digits, or underscores. The first character must be a letter; the underscore is not considered a letter. Upper and lower case letters are different.

3.2 Keywords

The following identifiers are reserved for use as keywords, and may not be used otherwise:

if	elif
else	for
while	return
true	false
distribute	read
print	download

3.3 Literals

Literals or **constants** are the values written in a conventional form whose value is obvious. In contrast to variables, literals do not change in value.

A **number literal**, or **number constant**, consists of an optional minus sign, one or more digits constituting the integer part of the number, and then an optional decimal point with zero or more digits consituting the fraction. If the integer part of the number will contain more than one digit, a zero cannot be the first. These are all well formed number literals in pubCrawl:

42	42.
0.42	42.4000
-42.4	0.42

These are not:

00	.68
17e-6	+3
092.5	6.7.8

A **boolean literal** represents boolean values for true and false. The two possible values are represented by `true` and `false`.

3.4 Strings as Collections

Because strings are collections of characters in pubCrawl, a number of caveats apply. For example, attempting to use the "add" built in collection function on a string literal is in some sense impossible since no character can exist on its own. Therefore what happens is one attempts to add a collection of characters to a collection of characters... this is a type mismatch! In order to perform the equivalent of "add" with strings, one must use collection concatenation. On the other hand "remove" works just fine. Declaring strings literals in pubCrawl is supported but the string is immediately converted into a collection of characters. Defining a string literal can be done with a sequence of one or more characters enclosed by single quotes. String literals can contain the ' character, but in order to represent it, and certain other characters, the following escape sequences may be used:

single quote	\'
newline	\n
horizontal tab	\t
carriage return	\r

Note that it is impossible to declare a character literal, instead it is treated as a string literal and immediately converted into a character collection of length 1.

3.5 Punctuation

Some characters are used as **punctuators**, which have their own syntactic and semantic significance. Punctuators are not operators or identifiers.

Punctuator	**Use**	**Example**
,	List seperator, function parameters	list1 = [1, 2, 3, 4, 5]
[]	List delimiter, list access, object access	list2 = []
()	Conditional parameter delimiter, expression precedence	if(bool)
{}	Statement list delimiter	if(bool) { statements }
:	List access, object property assignment	list3 = list2[0:2]
;	Statement end	mystring = 'ABC';
' '	String literal delimiter	mystring = 'ABC'

3.6 Comments

The characters /* introduce a multi-line comment, which terminates with the characters */. The characters // introduce a single line comment. Multi-line comments cannot be nested within multi-line comments, but single line comments can be nested in multi-line comments.

```
// single line comments start
/*
        multi-line comments
*/
/*
        /* this breaks */
        // this, however, works
*/
/* this will break too /* */
```

3.7 Operators

An **operator** is a token that specifies an operation on at least one operand, and yields some result.

Operator	**Use**	**Associativity**
.	Access	Left
*	Multiplication	Left
/	Division	Left
%	Modulus	Left
+	Addition	Left
-	Subtraction	Left
^	Concatenation	Left
=	Assignment	Non-associative
==	Equal to	Non-associative
!=	Not equal to	Non-associative
<	Less than	Non-associative
>	Greater than	Non-associative
<=	Less than or equal to	Non-associative'
>=	Greater than or equal to	Left

The precedence of operators is the following, from greatest to least precedence:

.
* / %
+ -
<><= >=
= !=
^
= :

Chapter 4

Syntax

4.1 Program Structure

A program in pubCrawl consists of a sequence of zero or more valid pubCrawl statements:

`statement-list-opt`

4.2 Expressions

An expression is a sequence of operators and operands that produce a value and may have side effects. Expressions have a type and a value. The order of evalution of subexpressions, and therefore the order in which side effects take place, is left to right. For example:

```
// x+y evaluated first, then a+b, then division.
(x + y) / (a + b)

// func2 evalutated first, then func3, then func1
func1(func2(), func3())
```

Operands of expressions must have compatible types.

4.2.1 Constants

The type of a constant depends on its form. It can either be a number, string or boolean, as discussed in Lexical Conventions.

4.2.2 Identifiers

An identifier designates a primitive type, collection, object, or function. The type and value of an identifier is determined by this designation. All identifiers are modifiable lvalues, so the type and value can change throughout a program. For example:

```
x = 4;
// here x has type number and value 4
x = true;
// here x has type boolean and value true
```

4.2.3 Binary Operators

Binary operators can be used with variables and constants to create complex expressions. A binary operator is of the form:

```
expression binary-operator expression
```

- Arithmetic operators

  Arithmetic operators include multiplication (*), division (/), modulus (%), addition (+), and subtraction (-). The operands to an arithmetic operator must be numbers. The type of an arithmetic operator expression is a number and the value is the result of calculating the appropriate arithmetic. For example, the multiplication operator performs multiplication on its operands.

- Relational operators

  Relational operators include less than (<), less than or equal to (<=), greater than (>), greater than or equal to (>=), equal to (==), and not equal to (!=). The operands to a relational operator must be numbers. The type of an relational operator expression is a boolean and the value is true if the relation is true. For example, the less than operator has a value of true if the left operand is strictly less than the right operand.

- Logical operators

  Logical operators include AND (&&) and OR (||). The operands to a logical operator must be booleans, and the result of the expression is also a boolean.

- String operators

  The only string operator is concatentation (^). The operands to a concatenation operator must be strings, and the result is a new string that is the concatenation of the left and right operands.

4.2.4 Parenthesized Expressions

Any expression surrounded by parentheses has the same type and value as it would without parentheses. In general, parentheses are used to alter operator precendence.

4.2.5 Function Creation

Function creation is an expression whose type is function and whose value is a reference to the newly created function. Because functions in pubCrawl are first class objects, functions can be declared anywhere having an expression would be appropriate. Because of this, functions must be stored in variables to be accessed later in the program. A function declaration is made clear with the "->" operator. On the left hand side of the "->" is the parameter declaration and on the right hand side is a block. Specifiying the result of a function is done with a return statement and void functions are not allowed. Parameter declaration is surrounded by parentheses and consists of a list of identifiers separated by commas. Optionally when only one parameter is specified, the parenthesis may be omitted. This is a function with no parameters:

```
() -> {
  statement-list-opt
  return-stmnt
}
```

Both of these functions have one parameter:

```
identifier -> {
  statement-list-opt
  return-stmnt
}
(identifier) -> {
  statement-list-opt
  return-stmnt
}
```

This function has n parameters

```
(identifier-1,identifier-2, ... ,identifier-n) -> {
  statement-list-opt
  return-stmnt
}
```

A function can have multiple return statements, however a function's return type must remain consistent over the life of the function. In other words no function can exist that sometimes returns one type, and sometimes returns a different type and such a function would fail to compile.

4.2.6 Function Call

A function call is an expression whose type and value are determined by the return type and value of the function. Calling a function executes the function and blocks program execution until the function is complete. Parameters are expressions that are separated by commas, surrounded by parenthesis and placed after the identifier representing the function. If there are no parameters, the parenthsis are still required for the function call.

```
identifier(expr1,expr2, ..., exprn)
identifier()
```

pubCrawl also supports a secondary format for function call where the function is called on an expression and that expression is passed in as the first parameter to the function. Any additonal parameters are passed in like normal. This allows functions to chain.

```
expr1.identifier(expr2, ..., exprn)
function-call.identifier(expr2, ..., exprn)
```

In the second example, the result of the first function is passed in as the first parameter to the second function.

When a function is called, the expressions passed into the function as parameters are evaluated in left to right order:

```
identifier(expr1,expr2)
```

In this example, expr1 would be evaluated first. The results of these expressions are then copied by value into the function's scope.

4.2.7 Object Creation

Object creation is an expression that has a type of object and a value of a reference to the object. An object is created as an optional list of properties surrounded by braces:

```
{ properties-opt }
```

The optional list of properties are comma separated property declarations:

```
identifier: expression
```

The identifiers in the property list must be unique for a specific object. Each property declaration assigns the value and type of the expression to the specified identifier within the context of the created object. Expressions in the property list will be evaluated in the order in which they appear.

4.2.8 Object Access

The properties of an object can be accessed via bracket notation or dot notation. The type and value of an object access expression are the type and value of the accessed property. To access via bracket notation, use a string literal to represent the identifier of the property used during creation:

```
object['identifier']
```

And to access via dot notation:

```
object.identifier
```

Note that this syntax might conflict with the secondary format for function calling in certain situations. Consider the following:

```
// myFunc compares two objects by property "value"
myFunc = (x,y) -> {
  if (x.value > y.value) {
    return 1;
  }
  elif (x.value < y.value) {
    return -1;
  }
  else {
    return 0;
  }
};

// myObj has property "myFunc" that checks if property "value" is even
myObj = {
  value: 7,
  myFunc: (x) -> {
          return x.value % 2 == 0;
  }
};

otherObj = {
  value: 4
};

// will result be 1 or true?
result = myObj.compare(otherObj);
```

In the above example, `result` will be `true`, not `0`, because object access takes precedence over secondary function calling notation. pubCrawl first looks for a property on `myObj` identified by `myFunc` of type function accepting one parameter. If such a property does not exist, pubCrawl will look for an in-scope variable `myFunc` of type function accepting two parameters. In order to call the locally defined `myFunc` function and pass `myObj` as the first parameter, the standard function calling sytax of `myFunc(myObj, otherObj)` must be used.

4.2.9 Collection Creation

Collection creation is an expression that has a type of colletion and a value of a reference to the collection. A collection is created as an optional list of expressions surrounded by brackets:

```
[ expressions-opt ]
```

The expressions in the optional list are comma separated and can be of any type, but must all have the same type.

4.2.10 Collection Access

Items in a collection are accessed using bracket notation. The expressions inside brackets must resolve to numbers. The resulting type will either be another collection, or the type of an item at the specified index. To access a particular item by it's index in a collection:

```
[ expression ]
```

To access a subset of the collection:

```
[ expression1-opt : expression2-opt ]
```

If expression1-opt is omitted, the subset will go from the beginning of the collection to the index of expression2-opt. If expression2-opt is ommited, the subset will go from the index of expression1-opt to the end of the collection. If both expressions are omitted, the collection will be copied.

4.3 Statements

A statement in pubCrawl does not produce a value and does not have a type, but can produce side effects. Statements are executed in the order in which they appear. An expression is not a valid statement in pubCrawl.

4.3.1 Assignment

Assignment statements consist of a modifiable lvalue and an expression. An lvalue is either an identifier, an object access expression, or a collection access expression that is not a subset. When an assignment statement is executed, the expression is evaluated and the result is assigned to the lvalue.

```
lvalue = expression;
```

4.3.2 Function Call

A function call statement executes the specified function and specifically ignores the return value of the function. A function call statement is a function call expression followed by a semi-colon.

```
function_call;
```

Because function calling is also an expression, the return value of a function call can be captured with an assignment statement.

4.3.3 Selection Statements

A selection statement executes a set of statments based on the value of a specific expression.

- If-elif-else

  An if-elif-else statement takes multiple boolean expression and selects one statement list to execute corresponding to the first true expression.

  ```
  if (expression-1)
  {
    statement-1-list
  }
  elif (expression-2)
  {
    statement-2-list
  }
  ...
  elif (expression-n)
  {
    statement-n-list
  }
  else
  {
    statement-else-list
  }
  ```

In the above example, if expression-1 is true, only statement-1-list will be executed, and all remaining expressions will not be evaluated. If expression-1 is false, expression-2 will be evaluated. If expression-2 is true, statement-2-list will be executed and all remaining expressions will not be evaluated. If all expressions evaluate to false, statemement-else-list will be executed. Note that all elif clauses are optional, and the else clause is optional, but must come after all elif clauses.

4.3.4 Iteration Statements

- While loop

  The while statement evaluates an expression before each execution of the body. The expression must by of type boolean, and the value of the expression typically changes in the body of the loop. If the expression is true, the loop body is executed. If the expression is false, the while statement terminates. The while statement has the following syntax:

```
while ( expression )
{
  statement-list
}
```

- For loop

  The for statement evaluates two assignments and one boolean expression, and executes the body until the expression evaluates to false.

```
for ( assignment1-opt ; expression-opt ; assignment2-opt )
{
  statement-list
}
```

  The for statement is equivalent to the following while loop:

```
assignment1-opt;
while ( expression-opt )
{
  statement-list
  assignment2-opt;
}
```

  A for statement executes as follows: assignment1-opt is evaluated once before any iteration. expression-opt is a boolean expression evaluated before each loop iteration. If this expression is true, the loop body is executed. If this expression is false, the for statement terminates. assignment2-opt is evaluated after each iteration. A jump statement inside the loop body will cause the for statement to terminate. If the optional expression is omitted, it will be treated as true, making an infinite loop.

4.3.5 Jump Statements

- Return

  A return statement is specified with the return keyword, followed by an expression and ending with a semi-colon.

```
return expression;
```

  The return statement terminates execution of the smallest containing function, or the entire program if there is no containing function, and returns control to the caller with the value and type of it's expression.

4.4 Scope

4.4.1 Lexical Scoping with Blocks

A block is a set of statements enclosed by braces. For example, the body of a for loop or an if statement is a block. An identifier appearing within a block is visible within the entire block, but can be modified by an inner block. For example, this will print 7 twice, followed by 'hey' five times:

```
x = 7;
print(x);
for (i = 0; i < 5; i = i + 1)
{
  print(x);
  x = 'hey';
}
print(x);
```

The scope of an identifier begins when it is first assigned, and ends at the end of it's smallest containing block or program.

```
a = 7;
for (i = 0; i < 5; i = i + 1)
{
  x = 'hey';
  // a and x are visible here
}
// only a is visible here
x = 'hey';
// a and x are visible here
```

4.4.2 Function Scope

Functions only have access to the identifiers in their parameter list and identifiers declared within their body.

```
x = 4;
y = (a,b,c) ->
{
  x = a + b + c; // this x does not affect outer x
  return x*5;
};
z = y(1,2,3);
/*
  x is still 4 here
  z is 30
*/
```

4.5 Distribution

The special function `distribute` is what enables programmers to leverage the processing poewr of multiple machines. Its use is essentially analogous to the `map` function (see section 6.1) - given a function on the list, it will apply the function to each element in the list, the difference being that the work of applying the function to the list is spread across multiple computers. The `distribute` function will return a list of processed values. Its usage is as follows:

```
results_list = distribute(list, function);
```

Chapter 5

Built-in Functions

pubCrawl provides some simple IO. Standard in and out, file read and write, as well as simple web downloading is all supported. Additionally, collections have built in methods to aid their manipulation.

5.1 Read

By using the read function one can read from either stdin or a file to EOF. The parameter is the name of a file, otherwise if no parameter is specified, it will read from stdin. The function will split on newline so whether referring to a file or stdin, the result of the function will be a list of strings, each one representing a line.

```
// read to EOF
input = read(); // reads from stdin until eof and returns a collection of lines
input = read('path/to/file.txt'); // returns a collection of lines from file
```

5.2 Print

By using the print function, one can print to either stdout or to a file. The first parameter must be of type string, and an optional second parameter must also be of type string. The first parameter will be written to standard out if no second parameter is passed, or to the file specified by the second parameter.

```
// write
print('Hello World'); // stdout
print('Hello World', 'path/to/file.txt');
```

5.3 Download

The download function will download either a webpage in the form of a string, or the bytes of a file.

```
//download
download('http://www.google.com'); // returns a string
download('http://images.mywebsite.com/myimage.png'); // returns bytes
```

5.3.1 Collection Modification

Built in to collections is the ability to add to the end of a collection using the "add" method. Also built in is the "remove" method which takes away an element from a specific index of a collection compressing the remaining collection much like a linked list would. Collections are zero indexed.

```
// start with 1-5
myCollection = [1,2,3,4,5];
// let's add 6
myCollection.add(6); // [1,2,3,4,5,6];
// then we can take away the element at index 3
myCollection.remove(3); //[1,2,3,5,6];
```

Chapter 6

Standard Library

6.1 Map

Usage:

```
resultList = map(list , function);
resultList = list.map(function);
```

The `map` function takes a function and a list as arguments, applies the function to each member of the list, and returns a new list with the results as elements. Since lambda functions are allowed in pubCrawl, the function with which to map can be declared inline.

6.2 Where

Usage:

```
resultList = where(list , boolean_function);
resultList = list.where(boolean_function);
```

The `where` function takes a function that returns a boolean result and a list, and applies the function to every element of the list. If the function returns `true,` the element is added to the returned list. Since lambda functions are allowed in pubCrawl, the function with which to map can be declared inline.

6.3 String/List Manipulation

pubCrawl provides a few built-in functions to aid in string manipulation. Because strings are treated as lists in pubCrawl, the functions that would be are found in this library can be used for lists as well.

6.3.1 Find

Usage:

```
number = find(string, substring);
number = string.find(substring);
```

Thhe default port that pubCrawl will use is `1099;` if the slave utility is listening on this port, the port need not be `find` function looks through a substring and returns the index at which the substring was found. If the substring was not found, the function returns `-1`.

6.3.2 Contains

Usage:

```
boolean = contains(string, substring);
boolean = string.contains(substring);
```

The `contains` function searches a string for a substring, and returns `true` if the substrng is present. Else, it returns `false`.

6.3.3 Split

Usage:

```
list = split(source, token);
list = source.split(token);
```

The `split` function returns a list containing the different parts of a string, as separated by the string given in `token`.

6.4 Range

Usage:

```
list = range(number_min, number_max);
list = range(number_max);
```

The `range` function returns a list of the integers between `number_min` and `number_max`, inclusive. If `number_min` is not specified, it is taken to be 0.

Chapter 7

Program Execution

To execute the program, the user must have the slave utility running on the slave machines. The user will then run the client program executable.

7.1 Master Setup

To run the master program, the user will run the executable generated by the compiler, using for arguments the IP addresses and port numbers of the machines being used. Alternatively, the user can specify the -f flag, and load a text file that contains all the hosts, with each host on a new line, in the format `IP_Address,port_number`. The user must specify one, but not both, of these options. Two examples are as follows:

```
master:~$ pc_exec 127.0.0.1 1100 192.168.0.1 3402

master:~\$ pc_exec -f hosts.txt
```

7.2 Slave Setup

To set up the slaves, the user will run the slave utility executable with one argument: the port number to listen on. The slave utility will then listen on that port for the client program.

```
slave:~$ pc\_slave 1337
```