Appendix - JavaScript basics

Introduction to JavaScript

JavaScipt is a high-level, untyped programming language commonly used in web development. WebPPL uses a functional subset of JavaScript, and some basic uses will be reviewed below.

JavaScript: The Good Parts (http://bdcampbell.net/javascript/book/javascript_the_good_parts.pdf) is an excellent introduction to the language. Online tutorials can be found here (http://www.w3schools.com/js/), there (https://www.javascript.com), and elsewhere (https://www.codeschool.com/learn/javascript).

You can do basic arithmetical operations:

```
3 + 3
```

run ▼

The `+` symbol is also used to concatenate strings:

```
"My favorite food is " + "pizza"
```

run ▼

Numeric variables will automatically modified into strings during concatenation:

```
3 + " is my favorite number"
```

run ▼

Boolean variables will be automatically changed into numbers when added (`false` becomes 0 and `true` becomes 1)

```
true + true
```

run ▼

Equality can be checked using `==` and `===`. `===` is a stricter comparison which cares about the type of variable (e.g., string, numeric, boolean).

```
x
```

```
print(3 == 3)
```

```
print("3" == 3)
```

```
print("3" === 3)
```

```

```

```
print("Booleans can equal numbers when you don't care about type.")
```

```
print(true == 1)
```

```
print(true === 1)
```

run ▼

A summary of comparison and logical operators can be found here (http://www.w3schools.com/js/js_comparisons.asp)

Mathematical functions and constants

JavaScript has a built-in "Math" object with properties and methods for mathematical constants and functions. ("Objects" will be described in more detail below.)

For example, to write: 3^232

```

Math.pow(3,2) // 3^2
```

run ▼

The area of a 12 inch pizza.

```

var radius = 12 / 2
```

```
Math.round(Math.PI*Math.pow(radius, 2)) + " square inches"
```

run ▼

A full list of the functions and constants can be found here (https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Math).

Building More Complex Programs

Consider the following complex expression built out of Boolean operators `||` (*or*) and `&&` (*and*):

```

true && (true || false)
```

run ▼

This expression has an *operator*—the function `&&`—and *arguments*—`true` on the left and `(true || false)` on the right. The latter argument itself is a *subexpression* consisting of a different operator—the function `||`—and different arguments—`true` and `false`. When reasoning about

the evaluation of a program, it is best to think of evaluating the subexpressions first, then substituting their return value into the larger expression. In this example, we first evaluate the expression `true || false`, returning true. After we substitute this into the larger expression, we have `true && true`, against returning true.

As a slightly more complex example, consider:

```
// this line is a comment

if (1 == 2)  {     // the condition of "if"

  100              // the consequent ("then")

} else {

  (true || false)  // the alternative ("else")

}
```

run ▼

This expression is composed of an `if` conditional that evaluates the first expression (a test here of whether `1` equals `2`) then evaluates the second expression if the first is true or otherwise evaluates the third expression. The operator `if` is strictly not a function, because it does not evaluate all of its arguments, but instead *short-circuits* evaluating only the second or third. It has a value like any other function. (We have also used comments here: anything after a `//` is ignored when evaluating.)

JavaScript has a very useful and common shorthand for `if` statements: it is called the “ternary” operator, using a question mark `?` and colon `:` to demarcate the three components.

The syntax is: `condition ? consequent : alternative`

```
(1 == 2) ?    // the condition of "if"

  100 :       // the consequent ("then")

  (true || false)  // the alternative ("else")
```

run ▼

Ternary statements can be strung together to create multiple different conditions

```
(1 == 2) ? 100 :

(2 == 3) ? 200 :

(3 == 4) ? 300 :
```

```
(3 == 3) ? 400 :
          500
```

run ▼

Note the particular indentation style used above (called “pretty-printing”). To clarify the structure of a function call, the arguments can split up across different lines and can aid readability:

```
(3 * (
  (2 * 4) + (3 + 5)
)) +
  (
  (10 - 7) + 6
)
```

run ▼

The online editor will automatically pretty-print for you. You can re-indent according to this style by selecting some lines and pressing the TAB key.

We often want to name objects in our programs so that they can be reused. This can be done with the `var` statement. `var` looks like this:

```
var variableName = expression
```

`variableName` is a *symbol* that is bound to the value that `expression` evaluates to. When variables themselves are evaluated they return the value that they have been bound to:

```
var someVariable = 3 // assign the value 3 to the variable someVariable
someVariable // when this is evaluated it looks up and returns the value 3
```

run ▼

Assignment of variables requires use of `var`

```
someVariable = 3
someVariable
```

run ▼

Multiple variable can be assigned in the same line using a `,`. To declare the end a line in JavaScript, use a `;`. In WebPPL as in standard JavaScript, the use of `;` is optional, but can be useful for readability.

```
var x = 3, y = 2;

y
```

run ▼

Arrays and objects

There are several special kinds of values in JavaScript. One kind of special value is an *array*: a sequence of other values.

```
["this", "is", "an", "array"]
```

run ▼

Arrays can be indexed using `[index]` Note: indexing starts at 0).

```
var myArray = ["this", "is", "my", "array"]

myArray[1]
```

run ▼

The length can be computed using `.length`

```
var myArray = ["this", "is", "my", "array"]

myArray.length
```

run ▼

You can grab subsets of the array using `.slice(being, end)`.

```
var myArray = ["this", "is", "my", "array"]

myArray.slice(1,3)
```

run ▼

If you don't put an `end`, it will default to the end.

```
var myArray = ["this", "is", "my", "array"]

myArray.slice(1)
```

run ▼

Arrays can be concatenated together, forming new arrays.

```
var myFirstArray = ["this", "is"]

var mySecondArray = ["my", "array"]



myFirstArray.concat(mySecondArray)
```

run ▼

`.concat` can take multiple arguments, concatenating together multiple arrays or values simultaneously.

Other helpful methods:

```
var myArray = ["this", "is", "my", "array"]

print( myArray.join(" _ ") )

print( myArray.toString() )

print( myArray.indexOf("my") )
```

run ▼

A list of all the properties of arrays can be found here (https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Array/of). Use caution with these. Not all JavaScript methods are supported on arrays in WebPPL. Some of these JavaScript methods will have their own WebPPL version. A list of the WebPPL functions for arrays can be found here (http://docs.webppl.org/en/master/functions/arrays.html).

In addition, the WebPPL language has available a JavaScript library useful for dealing with arrays and objects, called Underscore. Underscore functions can be accessed using the `_.` prefix. A full list of functions available from Underscore can be found here (http://underscorejs.org/). Note: underscore functions that take others functions (called predicates in underscore) as arguments are not available in WebPPL (e.g., `_.map`, `_.filter`, ...).

Here is one example, for reshaping arrays:

```
_.zip(['frodo', 'gandalf', 'gimli'], ["hobbit", "wizard", "dwarf"], [true, false, false])
```

run ▼

In real life, you encounter objects. Objects have properties. Properties can be accessed using `.property` or `["property"]` syntax.

```

```

```
var bilbo = { firstName: "Bilbo", lastName: "Baggins" }

print( bilbo.lastName )

print( bilbo["lastName"] )
```

run ▼

The latter is useful when the property is itself a variable

```
var bilbo = { firstName: "Bilbo", lastName: "Baggins"}

var prop = "lastName"

bilbo[prop]

// try: bilbo.prop
```

run ▼

Objects in JavaScript are very useful for structured data (like a dictionary in Python, or a dataframe in R). Underscore has several helpful functions for interacting with objects

```
var bilbo = {
  firstName: "Bilbo",
  lastName: "Baggins",
  race: "hobbit",
  age: 111,
  ringbearer: true
}


_.keys(bilbo)
```

run ▼

Building Functions: `function`

The power of programming languages as a model of computation comes from the ability to make new functions. To do so, we use the `function` primitive. For example, we can construct a function that doubles any number it is applied to:

```
var double = function(x) {
  return x + x
}

double(3)
```

run ▼

In WebPPL, the use of the `return` keyword is optional. By default, WebPPL will return the last line of the function. We use the `return` keyword for explicitness and clarity.

The general form of a function expression is: `function(arguments){ body }`. The first sub-expression of the function, the arguments, is a list of symbols that tells us what the inputs to the function will be called; the second sub-expression, the body, tells us what to do with these inputs. The value which results from a function is called a *compound procedure*. When a compound procedure is applied to input values (e.g. when `double` was applied to `3`) we imagine identifying (also called *binding*) the argument variables with these inputs, then evaluating the body.

In functional programming, we can build procedures that manipulate any kind of value—even other procedures. Here we define a function `twice` which takes a procedure and returns a new procedure that applies the original twice:

```
var double = function(x) { return x + x }

var twice = function(f) {
  return function(x) {
    return f(f(x))
  }
}

var twiceDouble = twice(double)
```

```
twiceDouble(3)
```

```
// same as: twice(double)(3)
```

run ▼

When functions take other functions as arguments, that is called a higher-order function

Higher-Order Functions

Higher-order functions can be used to represent common patterns of computation. Several such higher-order functions are provided in WebPPL.

`map` is a higher-order function that takes a procedure and applies it to each element of a list. For instance we could use map to test whether each element of a list of numbers is greater than zero:

```
map(function(x){
  return x > 0
}, [1, -3, 2, 0])
```

run ▼

The `map` higher-order function can also be used to map a function of more than one argument over multiple lists, element by element. For example, here is the MATLAB “dot-star” function (or `.*`) written using `map2`, which maps over 2 lists at the same time:

```
var dotStar = function(v1, v2){
  return map2(
    function(x,y){ return x * y },
    v1, v2)
}
```

```
dotStar([1,2,3], [4,5,6])
```

run ▼

`repeat` is a built-in function that takes another function as an argument. It repeats it how many ever times you want:

```
var g = function(){ return 8 }
```

```
repeat(100, g)
```

run ▼

Test your knowledge: Exercises (/exercises/appendix-js-basics.html)

Next chapter: 21. Appendix - Useful distribtuions (/chapters/appendix-useful-distributions.html)