### Introduction

Functions are a key component of programming languages that allow you to abstract a collection of programming statements for re-use. You can think of them as self-contained prefabricated sections of code, ready to be used many times!

Go ahead and make a new copy of the *Perlenspiel Template* folder in your *PGF Projects* folder, and name it *Experiment 4 - Functions*. Open the *game.js* file and get ready to program!

### An Example Function

Before we can begin examining what makes up a function, let’s go ahead and create a simple one! Find the section in *game.js* labeled with the comment // Put your functions after this line and add this code below it:

```javascript
/////////////////////////////////////////////////////////////////////
// Put your function definitions after this line.

function SayHello() {
    PS.debug("Hello!");
}
```

Note that this code section will be placed right before `PS.init`.

This is a function with the name “`SayHello`”. Whenever this function runs, the line of code inside the curly brackets will occur, and print “Hello!” to the debug box. When will this function run? We need to add what’s called a function call, which is where the function is “invoked” or caused to happen.

To see an example of a function call, add this code inside of `PS.init`:

```javascript
PS.init = function () {
    "use strict";

    // change to the dimensions you want
    PS.gridSize ( 8, 8 );

    // Put any other init code here
    SayHello();
};
```
If you test your program, it will print out “Hello!” in the debug box, just as advertised! If you call `SayHello()` multiple times, like this:

```javascript
// Put any other init code here
SayHello();
SayHello();
SayHello();
SayHello();
```

The word “Hello!” will print the same number of times in the debug box. Wait, there’s a bug, though. Here is what happens:

![Debug output showing multiple lines](image)

Looks like we forgot to include the "\n" when using `PS.debug`, so all of the messages showed up on the same line! Luckily, since our code to actually print out “Hello!” is in a function, we can fix it in one place, and it will automatically be fixed for all of the times that `SayHello()` is called. Here is the fix:

```javascript
function SayHello()
{
    PS.debug("Hello!\n");
}
```

Go ahead and test this – what an improvement! This is an example of how abstraction can save time when you need to tweak the code later, because the real business of the code is all in one spot.
**Function Declarations**

So, what is going on in that function declaration? Let’s label the different parts with comments:

```javascript
// This is the start of the function declaration for SayHello.
function SayHello() {  
  // This line is the function prototype.  
  // It consists of the function keyword, the function  
  // identifier "SayHello", and the parameter list "()"

  // This { indicates the beginning of the function body.
  // This is the function body.
  // Code between these curly brackets will run whenever the function is called.
  // The code will run in-order, top to bottom, and then return to where the
  // function call happened.
  PS.debug("Hello!\n");

  // This tells the program that the function body (and the entire declaration) is
  // over.
  // This is the end of the function declaration
}
```

That’s a lot of pieces! It’ll become clearer with more examples, however. Here are some more function declarations you can add to your program. Write them after the `SayHello()` declaration, before `PS.init`.

```javascript
// This function adds two numbers together, then prints the result
function AddNumbersAndPrint() {
  var x = 5;
  var y = 9;
  var result = x + y;
  PS.debug("Result: " + result + "\n");
}

// This function does nothing.
function DoNothing() {
  // Do nothing!
}

// This function prints "Hi!" two times.
function SayHiTwoTimes() {
  PS.debug("Hi!\n");
  PS.debug("Hi!\n");
}

// This function calls SayHello() two times
function SayHelloTwice() {
  // You can call functions from inside of other functions!
  SayHello();
  SayHello();
}

// This will make the bead at coordinate x=3, y=2 an orange square with a yellow $ inside.
function DrawIntoABead() {
  PS.color(3, 2, PS.COLOR_ORANGE);
  PS.glyph(3, 2, "$");  
  PS.glyphColor(3, 2, PS.COLOR_YELLOW);
}
```
Now that these functions have been declared, you can call them from within `PS.init`. How do you do that? Let’s find out!

**Function Calls**

A function call has the following syntax:

```c
// This is a function call.
// All function calls start with the Identifier (or name) of the function.
// Then, add a pair of parentheses, ()
// Finally, put a semicolon ; at the end.
SayHello(); // --> "Hello!"
```

This code is inside of `PS.init` so that it will run automatically when the program starts. You can go ahead and add a function call for each of your function declarations, so that we can see what they look like, and what they do:

```c
// Put any other init code here
SayHello(); // --> "Hello!"
AddNumbersAndPrint(); // --> "Result: 14"
DoNothing(); // (Doesn't print anything)
SayHiTwoTimes(); // --> "Hi!" --> "Hi!"
SayHelloTwice(); // --> "Hello!" --> "Hello!"
DrawIntoABead(); // (Draws one of the beads)
```
When you run it, this is what will show up in Perlespiel:

These are some pretty basic functions, and they don’t seem too useful yet. In order to get the most out of them, we’ll actually need to add a new component to functions. The next section is my favorite part!

**Function Parameters**

Until now, we’ve left the parentheses empty, and our functions are lonely, since we can’t send information into them from elsewhere in the program. All we can do is call the functions and let them do what they were designed to do. How can we improve this? Let’s add some parameters to our functions!

Most of the function declarations that we wrote earlier can be made more powerful by the addition of parameters. Let’s start with the function `SayHello()`. Here’s what it looks like now:

```javascript
// Prints out "Hello!"
function SayHello()
{
    PS.debug("Hello!\n");
}
```
One improvement we could make to this function is for it to say Hello to someone specific! We can pass in the name of the person as a parameter:

```javascript
// Prints out "Hello, 'name'!"
function SayHello(name) { // Added the parameter with the identifier "name"
    // Now you can use "name" as a variable anywhere in the function!
    PS.debug("Hello, " + name + "!\n");
}
```

Since we added this parameter, we need to go and change the function call so that we can “pass in” the value that becomes `name`. Find the line of code in `PS.init` that calls `SayHello()` and modify it to include your name as a string inside the parentheses:

```javascript
SayHello("Mark"); // --> "Hello, Mark!"
```

When you test the program, it’ll say hello to you! There are a couple weird lines in the debug box, though. This is because we didn’t update the other places where `SayHello()` is called. If you call a function without filling out its parameters, they become “undefined”. Let’s modify the `SayHelloTwice()` function to fix this problem:

```javascript
// This function calls SayHello() two times
function SayHelloTwice() { // You can call functions from inside of other functions!
    SayHello("Fred");
    SayHello("George");
}
```

Now, the “undefined” text in the debug box will be replaced with the proper names. What if we want to use multiple parameters to pass in both names, instead of using “Fred” and “George” all the time? Try this:

```javascript
// This function calls SayHello() two times
function SayHelloTwice(firstName, secondName) { // You can call functions from inside of other functions!
    SayHello(firstName);
    SayHello(secondName);
}
```

Here, we have two parameters – note that they are separated by a comma, and each one has a unique identifier. You can actually have many parameters, and that’s why this section of code is called a “parameter list”. Each parameter in the list needs to be separated with a comma in order for JavaScript to understand it.

Don’t forget to modify the function call as well so that “undefined” doesn’t get passed in to the function:

```javascript
SayHelloTwice("Will", "Carlton"); // --> "Hello, Will! --> "Hello, Carlton!"
```

When calling a function with multiple parameters, you include the actual values for those parameters in the same order that they are in the declaration. Try running the program and make sure it works.
Writing into the debug box isn’t very fun, so let’s work on the `DrawIntoABead()` function next. Here’s what we are starting with:

```json
// This will make the bead at coordinate x=3, y=2 an orange square with a yellow $ inside.
function DrawIntoABead()
{
    PS.color(3, 2, PS.COLOR_ORANGE);
    PS.glyph(3, 2, "$");
    PS.glyphColor(3, 2, PS.COLOR_YELLOW);
}
```

This function does three things to the bead at the coordinate (3,2). As you may have guessed, the coordinates would make good candidates for parameters! Here are the logical steps you could use:

```json
// This will make the bead at coordinate x=3, y=2 an orange square with a yellow $ inside.
function DrawIntoABead()
{
    var x = 3;
    var y = 2;
    PS.color(x, y, PS.COLOR_ORANGE);
    PS.glyph(x, y, "$");
    PS.glyphColor(x, y, PS.COLOR_YELLOW);
}
```

First, we’ve abstracted the calls into `BeadColor` et al. with the variables x and y, which are set to the values 3 and 2, respectively. (Yes, `BeadColor` is actually a function! Now that we know about parameters and function calls, you can start understanding all of Perlenspiel’s built-in functions.)

Next, we can move the x and y variables into the parameter list. Remember to remove the `var` keyword:

```json
// This will make the bead at coordinate (x, y) an orange square with a yellow $ inside.
function DrawIntoABead(x, y)
{
    PS.color(x, y, PS.COLOR_ORANGE);
    PS.glyph(x, y, "$");
    PS.glyphColor(x, y, PS.COLOR_YELLOW);
}
```

Good, now we need to update the function call in `PS.init`:

```json
DrawIntoABead(3, 2);  // (Draws one of the beads)
```
Since this is a function with parameters, we can actually call it a bunch of times with different coordinates, giving us an easy way to do a bunch of drawing:

```plaintext
DrawIntoABead(4, 2);
DrawIntoABead(3, 2);
DrawIntoABead(1, 2);
DrawIntoABead(0, 3);
DrawIntoABead(3, 4);
DrawIntoABead(1, 4);
DrawIntoABead(4, 5);
DrawIntoABead(3, 6);
DrawIntoABead(1, 6);
DrawIntoABead(0, 6);
```

// Draw a line down column 2, starting on row 1
for(var i = 1; i < 8; i++)
{
    DrawIntoABead(2, i);
}

Try running it and see what happens! It should get your Scrooge McDuck senses tingling.

Brilliant! Note that you can change the colors and glyph in the body of `DrawIntoABead()` to change the colors and glyphs in this entire drawing, all at once. Try it out!
Nesting Function Calls

Let’s make some more modifications to this program to make it more interactive. One cool thing you can do is program it so that when you click somewhere, it draws a dollar sign. How do you do this? At a high level, you need to make a function that draws a dollar sign at a variable \((x, y)\) coordinate. Then, inside of `PS.touch`, that function will be called. Let’s take a look at some code!

First, let’s make an empty function that takes the parameters \(x\) and \(y\):

```javascript
// This function will draw a large dollar sign, using the location (x, y) as the upper left hand corner.
function DrawDollarAtLocation(x, y)
{
    // Draw a dollar sign here
}
```

Next, we need to move the dollar-sign drawing code from `PS.init` to the function body of `DrawDollarAtLocation`. This means we will actually have a function which consists primarily of calls to another function. This is fairly common in programming.

```javascript
// This function will draw a large dollar sign, using the location (x, y) as the upper left hand corner.
function DrawDollarAtLocation(x, y)
{
    // Draw a dollar sign here
    DrawIntoABead(4, 2);
    DrawIntoABead(3, 2);
    DrawIntoABead(1, 2);
    DrawIntoABead(0, 3);
    DrawIntoABead(3, 4);
    DrawIntoABead(1, 4);
    DrawIntoABead(4, 5);
    DrawIntoABead(3, 6);
    DrawIntoABead(1, 6);
    DrawIntoABead(0, 6);

    // Draw a line down column 2, starting on row 1
    for(var i = 1; i < 8; i++)
    {
        DrawIntoABead(2, i);
    }
}
```

Make sure those lines of code are removed from `PS.init`. Test the program to make sure nothing shows up in the grid any more. Once you’ve done this, add a `DrawDollarAtLocation(0, 0)` function call in `PS.init` after the status text is set:

```javascript
// Put any other init code here
DrawDollarAtLocation(0, 0);
```

This won’t quite work yet, since the variables for the \(x\) and \(y\) coordinate aren’t hooked into the drawing routine. You will need to make some substitutions for it to work.
First, subtract one from every ‘y’ coordinate in the function, so that it will start all the way at the top of the grid:

```javascript
// This function will draw a large dollar sign, using the location (x, y) as the upper left hand corner.
function DrawDollarAtLocation(x, y)
{
    // Draw a dollar sign here
    DrawIntoABead(x+4, y+1);
    DrawIntoABead(x+3, y+1);
    DrawIntoABead(x+1, y+1);
    DrawIntoABead(x+0, y+2);
    DrawIntoABead(x+3, y+3);
    DrawIntoABead(x+1, y+3);
    DrawIntoABead(x+4, y+4);
    DrawIntoABead(x+3, y+5);
    DrawIntoABead(x+1, y+5);
    DrawIntoABead(x+0, y+5);

    // Draw a line down column 2, starting on row 1
    for(var i = 0; i < 7; i++)
    {
        DrawIntoABead(x+2, y+i);
    }
}
```

Next, before every ‘x’ coordinate in a DrawIntoABead function call add the code x+1, and before every ‘y’ coordinate on the same lines, add the code y+1. This will allow you to adjust the upper left coordinate by changing the parameters when you call this function.

```javascript
// This function will draw a large dollar sign, using the location (x, y) as the upper left hand corner.
function DrawDollarAtLocation(x, y)
{
    // Draw a dollar sign here
    DrawIntoABead(x+4, y+1);
    DrawIntoABead(x+3, y+1);
    DrawIntoABead(x+1, y+1);
    DrawIntoABead(x+0, y+2);
    DrawIntoABead(x+3, y+3);
    DrawIntoABead(x+1, y+3);
    DrawIntoABead(x+4, y+4);
    DrawIntoABead(x+3, y+5);
    DrawIntoABead(x+1, y+5);
    DrawIntoABead(x+0, y+5);

    // Draw a line down column 2, starting on row 1
    for(var i = 0; i < 7; i++)
    {
        DrawIntoABead(x+2, y+i);
    }
}
```

Alright, the code looks ready. Test it to make sure it still draws in the upper left hand corner. If it works, good job! Otherwise, double check your code to make sure it matches the examples.
Once that is working, change the parameters in `PS.init` to be `(2, 1):

```javascript
// Put any other init code here
DrawDollarAtLocation(2, 1);
```

If you test this, it should have moved the dollar sign so that it starts two over and one down.

![Diagram](image)

**Event Functions**

We made it all this way without talking much about all of the `PS.whatever` code sections in Perlenspiel’s `game.js`! These are called event functions, and they are automatically called by the Perlenspiel engine whenever the appropriate event occurs. We have already been using `PS.init` quite a bit, and we know as well that `PS.init` occurs once when the page loads.

There are a lot more events than this, as you can see just by looking through game.js. There is a comprehensive list and reference here on the [http://perlenspiel.org/](http://perlenspiel.org/) website under **API ➔ Events**. Here is a list of them that you can click on to view in your browser:

- `PS.init` (system, options)
- `PS.touch` (x, y, data, options)
- `PS.release` (x, y, data, options)
- `PS.enter` (x, y, data, options)
- `PS.exit` (x, y, data, options)
- `PS.exitGrid` (options)
- `PS.keyDown` (key, shift, ctrl, options)
- `PS.keyUp` (key, shift, ctrl, options)
- `PS.input` (device, options)

The one we are going to look at right now is `PS.touch`, which has four parameters: `x`, `y`, `data`, and `options`. We are going to focus on `x` and `y` right now, since they are generally much more important to know about.
Whenever you click on a bead in Perlenspiel, the `PS.touch` event function will be called, and the coordinates of the clicked bead will be passed in as the variables `x` and `y`. If you click on the upper left bead, `x` and `y` will both be equal to zero.

It’s easier to illustrate using these functions with an example. Go into the `PS.touch` event, and write function call to `DrawDollarAtLocation` there, using `x` and `y` as parameters:

```javascript
PS.touch = function (x, y, data, options) {
  "use strict";
  // put code here for bead clicks
  DrawDollarAtLocation(x, y);
};
```

Go ahead and test this. When you click somewhere, it will draw a dollar sign at that location. There isn’t much room, though, so let’s make the grid bigger. Change the `PS.gridSize` call in `PS.init` from `(8, 8)` to `(24, 24).

When you test it, click around on the grid and see if it works. You may spot a couple problems (see the picture to the right):

- When you click so that part of the dollar sign is outside of the grid, it will cause an error! Another less-pressing issue is that the grid starts with a dollar sign already in it, but you can fix that easily by removing the `DrawDollarAtLocation` function call from inside of `PS.init`. The out-of-bounds error is more difficult to fix – we will need to add some code to prevent dollar signs from being drawn where they will go off the edge.

The easiest way to prevent that behavior is to use an if-statement to verify the `x` and `y` coordinate of the dollar sign before actually drawing it. The dollar sign is 5 beads wide and 7 beads tall, so we can use those numbers in our math to calculate if a location is a good spot to draw or not. To facilitate this, we will also want to convert the `PS.gridSize` parameters to variables, so that we can reference them in multiple places. Create two variables at global scope, named `WORLD_W` and `WORLD_H`:

```javascript
var WORLD_W = 24;
var WORLD_H = 18;
```

You can use any numbers you’d like, as long as they are large enough to hold one of the dollar signs, and no larger than 32. Next, substitute these variables for the values in the `GridSize` function call:

```javascript
PS.gridSize ( WORLD_W, WORLD_H );
```
Alright, we’re ready to add the bounds checking to `DrawDollarAtLocation`. First, we need to make sure that when you click, none of the dollar sign will go off the right side of the screen. The math for this is pretty simple – take the clicked x-coordinate, add the width of the dollar sign, and compare it to the width of the grid. If the first two added together are less than or equal to the world width, then it’s in bounds. Tie up the if-statement and move all of the drawing code into it.

```javascript
// This function will draw a large dollar sign, using the location (x, y) as the upper left hand corner.
function DrawDollarAtLocation(x, y)
{
    var dollarWidth = 5;
    if( x + dollarWidth <= WORLD_W )
    {
        // Draw a dollar sign here
        DrawIntoABead(x+4, y+1);
        DrawIntoABead(x+3, y+1);
        DrawIntoABead(x+1, y+1);
        DrawIntoABead(x+0, y+2);
        DrawIntoABead(x+3, y+3);
        DrawIntoABead(x+1, y+3);
        DrawIntoABead(x+4, y+4);
        DrawIntoABead(x+3, y+5);
        DrawIntoABead(x+1, y+5);
        DrawIntoABead(x+0, y+5);

        // Draw a line down column 2, starting on row 1
        for(var i = 0; i < 7; i++)
        {
            DrawIntoABead(x+2, y+i);
        }
    }
}
```

This will take care of the x-coordinate, but we need similar code for y. Remember that the height of the dollar is 7 beads, and we have a variable named `WORLD_H`. 
// This function will draw a large dollar sign, using the location (x, y) as the upper left
hand corner.

function DrawDollarAtLocation(x, y)
{
    var dollarWidth = 5;
    var dollarHeight = 7;
    if( x + dollarWidth <= WORLD_W )
    {
        if( y + dollarHeight <= WORLD_H )
        {
            // Draw a dollar sign here
            DrawIntoABead(x+4, y+1);
            DrawIntoABead(x+3, y+1);
            DrawIntoABead(x+1, y+1);
            DrawIntoABead(x+0, y+2);
            DrawIntoABead(x+3, y+3);
            DrawIntoABead(x+1, y+3);
            DrawIntoABead(x+4, y+4);
            DrawIntoABead(x+3, y+5);
            DrawIntoABead(x+1, y+5);
            DrawIntoABead(x+0, y+5);

            // Draw a line down column 2, starting on row 1
            for(var i = 0; i < 7; i++)
            {
                DrawIntoABead(x+2, y+i);
            }
        }
    }
}

That will take care of it! You can also combine both if-statements together into one by using the & operator:

```
if( (x + dollarWidth <= WORLD_W) && (y + dollarHeight <= WORLD_H) )
{
    ...
}
```

If you go this route, remember to remove the second if-statement’s curly brackets and other bits and pieces.

We are done with this function, so take a well-deserved rest. That was a real piece of work! However, if you
want to know even more crazy things you can do with functions, read onward.

**Return Statements**

A return statement allows you to do two unique things with a function:

1. Exit the function early
2. Give a result value back to whoever called the function

By exiting a function early, you can stop running a function at any time when you’ve determined that you don’t
need to run the rest of the code.

Returning a value to the caller is definitely the more interesting of the two, and is also really valuable to learn,
since it lets you simplify math, write better utility functions, and send data to where it’s needed.
Here is a quick example function that you can add into your program:

```javascript
// Adds two numbers and returns the result
function Add(x, y)
{
    var result = x + y;
    return result;
}
```

This function takes two parameters, x and y, adds them together, and returns the result. How is it used? Check out this modified `AddNumbersAndPrint` function:

```javascript
// This function adds two numbers together, then prints the result
function AddNumbersAndPrint()
{
    var x = 5;
    var y = 9;
    PS.debug("Result: " + Add(x,y) + "\n");
}
```

When this function is called from `PS.init`, it will then call `Add(x, y)`, which performs the actual addition. Then, it returns the result back to `AddNumbersAndPrint`, in the exact spot that it needs to print out the result!

Here are some more examples that you can add into `PS.init`:

```javascript
// Function return examples
var halfPerimiter = Add(WORLD_W, WORLD_H);
var perimiter = Add(halfPerimiter, halfPerimiter);
PS.debug("The world perimiter is " + perimiter + " beads.\n");
```

Here, the result of adding `WORLD_W` and `WORLD_H` together is stored in a new variable, `halfPerimiter`. Then, `halfPerimiter` is added to itself to result in the `perimeter`. Finally, `perimeter` is printed out to the debug box.

This is kind of a trivial example, so let’s write a function that does something that we can’t already do with the `+` operator:

```javascript
// Calculate the power
function Power(number, pow)
{
    var result = 1;
    for( var i = 0; i < pow; i++ )
    {
        result = result * number;
    }
    return result;
}
```

Ah, the classic `Power` function. In regular math, this is indicated by the `^` operator, but that doesn’t work in JavaScript. That’s why we wrote our own! What it does is multiply `number` repeatedly until result is just the right size. Then it returns result.
The variables `number` and `pow` are kind of like variable declarations, because the variable names don’t show up before this, but since they are in the parameter list of a function, they do not need the `var` keyword. When calling the function, the parameters you pass in will show up inside the function as those two variables – they are assigned the values that are given in the function call.

Here’s what a call to it will look like, in `PS.init`:

```groovy
// Function call and return example: Power
var number = 3;
var power = 4;
var toThePower = Power(number, power);
PS.debug(""+number+" ^ " + power + " == " + toThePower);
```

Groovy.

It should print out the number 81, which is 3 to the 4th power. Right now it only works for powers of 1 or greater, but we can fix that. See the next page.
// Calculate a number taken to the power

function Power(number, pow) {
    if (pow == 0) {
        return 1;
    } else if (pow > 0) {
        var result = 1;
        for (var i = 0; i < pow; i++) {
            result = result * number;
        }
        return result;
    } else // negative power
    {
        var result = 1;
        for (var i = 0; i > pow; i--) // Time to go backwards...
        {
            result = result / number;
        }
        return result;
    }
}

Phew, that’s a lot of code. Powers are actually really complicated, once you take negative exponents into account. Luckily, if we ever find a bug in this code, we only need to fix it in one place: The Power function definition.

We’ll see more practical applications of these sorts of functions later on in this class!

**What do you use Functions for?**

Functions are used for a lot of reasons, due to their ability to encapsulate code that performs a particular task into its own section of the code. Any time that you have an equation or piece of functionality that repeats in your game, you can move that calculation or behavior to live inside of a function. This works even if each use of the behavior is slightly different, since we can use function parameters to send the important differences to the function body.

Consider defining a new function whenever:

- You write the exact same line of code multiple times
- You are performing the same calculation multiple times
- You need to mentally separate out a piece of code into its own, distinct area

**Conclusion**

When programming, you will likely find functions to be one of the most powerful tools available to you. While it’s possible to write an entire game without defining new functions in Perlenspiel, it can be a difficult task.