Although that is already more readable than the soup of bits, it is still rather unpleasant. It might help to use names instead of numbers for the instructions and memory locations.

```
 Set “total” to 0.  Set “count” to 1. [loop]  Set
“compare” to “count”.  Subtract 11 from “compare”.
If “compare” is zero, continue at [end].  Add
“count” to “total”.  Add 1 to “count”.  Continue at
[loop]. [end]  Output “total”.
```

Can you see how the program works at this point? The first two lines give two memory locations their starting values: `total` will be used to build up the result of the computation, and `count` will keep track of the number that we are currently looking at. The lines using `compare` are probably the weirdest ones. The program wants to see whether `count` is equal to 11 in order to decide whether it can stop running. Because our hypothetical machine is rather primitive, it can only test whether a number is zero and make a decision (or jump) based on that. So it uses the memory location labeled `compare` to compute the value of `count - 11` and makes a decision based on that value. The next two lines add the value of `count` to the result and increment `count`by 1 every time the program has decided that `count` is not 11 yet.

Here is the same program in JavaScript:

```
var total = 0, count = 1; while (count <= 10) {
total += count;   count += 1; } console.log(total);
// → 55
```

This version gives us a few more improvements. Most importantly, there is no need to specify the way we want the program to jump back and forth anymore. The `while` language construct takes care of that. It continues executing the block (wrapped in braces) below it as long as the condition it was given holds. That condition is `count <= 10`, which means “`count` is less