2

WEBASSEMBLY TEXT BASICS

In this chapter, we'll dive into the basics of WAT code. We'll write most of the code in this book in WAT, the lowest level of programming you can write for deployment to WebAssembly (although for experienced assembly coders, it might seem rather high level).

This chapter covers a lot of ground. We'll begin by showing you the two comment styles in WebAssembly. Next, we'll write the traditional hello world! application. We don't start with hello world! because working with strings from within WAT is more challenging than you might expect.

Then we'll discuss how to import data from JavaScript into our WebAssembly module using an import object. We'll look at named and unnamed global and local variables, as well as the data types that WebAssembly supports. We'll discuss the S-Expression syntax and how the `wat2wasm` compiler unpacks those S-Expressions when it compiles your code. You'll delve into conditional logic, including `if/else` statements and branch tables, and you'll learn how to use loops and blocks in conjunction with conditional logic.

By the end of this chapter, you should be able to write simple WebAssembly apps that you can execute from the command line using Node.js.

Writing the Simplest Module

Every WAT application must be a module, so we'll first look at the module syntax. We declare a module in a block, like the one in Listing 2-1.

```
(module
  ;; This is where the module code goes.
)
```

Listing 2-1: Single line WAT comment

We declare a module with the `module` keyword, and anything inside the surrounding parentheses is part of the module. To add a comment, we use two semicolons `;;` , and everything on the line that follows is a comment. WAT also has block comment syntax; you open the block comment with `(;` and close it with `;)` , as shown in Listing 2-2.

```
(module
  (;
  This is a module with a block comment.
  Like the /* and */ comments in JavaScript
  you can have as many lines as you like inside
  between the opening and closing parenthesis
  ;)
)
```

Listing 2-2: Multi-line WAT comment

Because this module doesn't do anything, we won't bother to compile it. Instead, we'll move on to writing our hello world! application.

Hello World! in WebAssembly

WAT doesn't have any native string support, so working with strings requires you to work directly with the memory as an array of character data. That memory data then must be converted into a string in JavaScript code, because manipulating strings from within JavaScript is much simpler.

When working with strings in WAT, you need to declare an array of character data that is stored within WebAssembly linear memory. Linear memory is a topic we'll discuss in detail in Chapter 5, but for now know that linear memory is similar to a memory heap in native applications, or a giant Uint8Array in JavaScript.

You'll also need to call an imported JavaScript function from WebAssembly to handle I/O operations. Unlike in a native application where the operating system usually handles I/O, in a WebAssembly module, I/O

must be handled by the embedding environment, whether that environment is a web browser, an operating system, or runtime.

Creating Our WAT Module

In this section, we'll create a simple WebAssembly module that creates a `hello world!` string in linear memory and calls JavaScript to write that string to the console. Create a new WAT file and name it *helloworld.wat*. Open that file and add the WAT code in Listing 2-3.

helloworld.wat

```
(module
  (import "env" "print_string" (func $print_string( param i32 )))
)
```

Listing 2-3: Importing a function

This code tells WebAssembly to expect the import object `env` from our embedding environment, and that, within that object we're expecting the function `print_string`. When we write our JavaScript code later, we'll create this `env` object with the `print_string` function, which will be passed to our WebAssembly module when we instantiate it.

We also set up the signature as requiring a single `i32` parameter representing the length of our string. We name this function `$print_string` so we can access it from our WAT code.

Next, we'll add an import for our memory buffer. Add the line in bold in Listing 2-4; the grayed out code indicates that it's repeated from the previous listing.

helloworld.wat

```
(module
  (import "env" "print_string" (func $print_string( param i32 )))
  (import "env" "buffer" (memory 1))
)
```

Listing 2-4: Importing a function and memory buffer

This new `import` tells our WebAssembly module that we'll be importing a memory buffer from the object `env` and the buffer will be called `buffer`. The `(memory 1)` statement indicates that the buffer will be a single page of linear memory: a *page* is the smallest chunk of memory you can allocate at one time to linear memory. In WebAssembly, a page is 64KB, which is more than we need for this module, so we need just one page. Next, in Listing 2-5, we add a few global variables to *helloworld.wat*.

helloworld.wat

```
(module
  (import "env" "print_string" (func $print_string( param i32 )))
  (import "env" "buffer" (memory 1))
❶ (global $start_string (import "env" "start_string") i32)
❷ (global $string_len i32 (i32.const 12))
)
```

Listing 2-5: Adding global variables

The first `global` ❶ variable is a number imported from our JavaScript import object; it maps to a variable with the name `env` in JavaScript (which we've yet to create). That value will be the starting memory location of our string and can be any location in our linear memory page up to the maximum 65,535. Of course, you wouldn't want to choose a value close to the end of linear memory because it would limit the length of the string you could store. If the value passed in is `0`, you can use the entire 64KB for your string. If you passed in the value `65,532`, you would only be able to use the last four bytes to store character data. If you try to write to a memory location that is greater than what was allocated, you'll get a memory error in your JavaScript console. The second global variable, `$string_len` ❷, is a constant that represents the length of the string we'll define, and we set it to `12`.

In Listing 2-6, we define our string in linear memory using a data expression.

helloworld.wat

```
(module
  (import "env" "print_string" (func $print_string( param i32 )))
  (import "env" "buffer" (memory 1))
  (global $start_string (import "env" "start_string") i32)
  (global $string_len i32 (i32.const 12))
  (data (global.get $start_string) "hello world!")
)
```

Listing 2-6: Adding a data string

We first pass the location in memory where the module will write data. The data is stored in the `$start_string` global variable that the module will import from JavaScript. The second parameter is the data string, which we define as the string `"hello world!"`.

Now we can define our `"helloworld"` function and add it to the module, as shown in Listing 2-7.

helloworld.wat

```
(module
  (import "env" "print_string" (func $print_string (param i32)))
  (import "env" "buffer" (memory 1))
  (global $start_string (import "env" "start_string") i32)
  (global $string_len i32 (i32.const 12))
  (data (global.get $start_string) "hello world!")
❶ (func (export "helloworld")
  ❷ (call $print_string (global.get $string_len))
  )
)
```

Listing 2-7: Adding a `"hello world"` function to the WebAssembly module

We define and export our function as `"helloworld"` for use in JavaScript ❶. The only thing this function does is call the imported `$print_string` ❷ function, passing it the length of the string we defined as a global. We can now compile our WebAssembly module, like so:

```
wat2wasm helloworld.wat
```

Running `wat2wasm` generates a *helloworld.wasm* module. To execute the WebAssembly module, we'll need to create a JavaScript file that executes it.

Creating the JavaScript File

Now we'll create *helloworld.js* to run our WebAssembly module. Create and open the JavaScript file in your text editor, and add the Node.js file constants as well as three variables, as shown in Listing 2-8.

helloworld.js

```
const fs = require('fs');
const bytes = fs.readFileSync(__dirname + '/helloworld.wasm');

❶ let hello_world = null; // function will be set later
❷ let start_string_index = 100; // linear memory location of string
❸ let memory = new WebAssembly.Memory ({ initial: 1 }); // linear memory
...
```

Listing 2-8: Declaring the JavaScript variables

NOTE *Notice the . . . in the last line of code in Listing 2-8. In this book, we'll use . . . to indicate that there is more code to be added to this file in the next few sections. If the . . . appears at the beginning of a block of code, it indicates that this is a continuation from code in a previous section that ended with the . . . syntax.*

The `hello_world` ❶ variable will eventually point to the `helloworld` function exported by our WebAssembly module, so we set it to `null` for the time being. The `start_string_index` ❷ variable is the starting location of our string in the linear memory array. We set it to `100` here, so as not to be close to the 64KB limit. We chose the address 100 arbitrarily. You can choose any address as long as none of the memory you're using extends past the 64KB limit.

The last variable holds the `WebAssembly.Memory` ❸ object. The number passed represents the number of pages you want to allocate. We initialize it with a size of one page by passing in `{initial: 1}` as the only parameter. You can allocate up to two gigabytes this way, but setting this value too high can result in an error if the browser is unable to find enough contiguous memory to fulfill the request.

Listing 2-9 shows the next variable we need to declare, `importObject`, which will be passed into our WebAssembly module when we instantiate it.

helloworld.js

```
...
let importObject = {
❶ env: {
  ❷ buffer: memory,
  ❸ start_string: start_string_index,
  ❹ print_string: function (str_len) {
      const bytes = new Uint8Array (memory.buffer,
        start_string_index, str_len);
      const log_string = new TextDecoder('utf8').decode(bytes);
      console.log (log_string);
    }
```

```
  }
};
...
```

Listing 2-9: Declaring the `importObject` in JavaScript

Inside our `importObject`, we add an object named `env` ❶ an abbreviation of *environment*, although you can call this object anything you like as long as it matches the name inside the WebAssembly import declaration. These are the values that will be passed into the WebAssembly module when it's instantiated. If there is any function or value from the embedding environment you want to make available to the WebAssembly module, pass them in here. The `env` object contains the memory buffer ❷ and the starting location ❸ of our string within `buffer`. The third property in `env` ❹ contains our JavaScript function, `print_string`, which our WebAssembly module will call as we instructed in Listing 2-9. This function retrieves the length of the string in our memory buffer and uses it in combination with our starting string index to create a string object. The app then displays the string object on the command line.

Additionally, we add the IIFE that asynchronously loads our WebAssembly module and then calls the `helloworld` function, as shown in Listing 2-10.

helloworld.js

```
...
( async () => {
  let obj = await
❶ WebAssembly.instantiate(new Uint8Array (bytes), importObject);
❷ ({helloworld: hello_world} = obj.instance.exports);
❸ hello_world();
})();
```

Listing 2-10: Instantiating the WebAssembly module in an asynchronous IIFE

The first line of the `async` module awaits the `WebAssembly.instantiate` ❶ function call, but unlike the simple addition example from Listing 1-1, we're passing that function the `importObject` we declared earlier. We then pull the `helloworld` function out of `obj.instance.exports` using the destructuring syntax to set the `hello_world` variable to the `obj.instance.exports` function ❷.

The last line of our IIFE calls the `hello_world` ❸ function. We enclose our arrow function in parentheses, and then add the function call parentheses to the end of our function declaration, which causes this function to execute immediately.

NOTE *We declare asynchronous code using the IIFE `async/await` syntax in an `async` function; alternatively, you can create a named `async` function that you call immediately after the function's declaration.*

Once you have the JavaScript and WebAssembly files, run the following call to `node` from the command line:

```
node helloworld.js
```

You should see the following output on the command line:

```
hello world!
```

We've built the ubiquitous hello world! application! Now that you have the hello world! application under your belt, we'll explore variables and how they work in WAT.

WAT Variables

WAT treats variables a little differently than other programming languages, so it's worth providing you with some details here. However, the browser manages local or global WAT variables in the same way it manages JavaScript variables.

WAT has four global and local variable types: i32 (32-bit integer), i64 (64-bit integer), f32 (32-bit floating-point), and f64 (64-bit floating-point). Strings and other more sophisticated data structures need to be managed directly in linear memory. We'll cover linear memory and the use of more complicated data structures in WAT in Chapter 6. For now, let's look at each variable type.

Global Variables and Type Conversion

As you might expect, you can access globals in WAT from any function, and we generally use globals as constants. *Mutable globals* can be modified after they're set and are usually frowned upon because they can introduce side effects in functions that use them. Instead, you can import global variables from JavaScript, allowing the JavaScript portion of your application to set constant values inside your module.

When importing global variables, keep in mind that, at the time of this writing, standard JavaScript number variables don't support 64-bit integer values. Numbers in JavaScript are 64-bit floating-point variables. A 64-bit floating-point variable can represent every value in a 32-bit integer, so JavaScript has no trouble making this conversion. However, you cannot represent all possible 64-bit integer values with a 64-bit floating-point value. Unfortunately, this means that you can work with 64-bit integers in WebAssembly, but if you want to send 64-bit values to JavaScript, it requires additional effort, which is beyond the scope of this book.

NOTE *The BigInt WebAssembly proposal will make support for JavaScript BigInt types available in WebAssembly. That will make it easier for JavaScript to exchange 64-bit integers with a WebAssembly module. The BigInt proposal is currently in the final stages of development by the WebAssembly Working Group.*

Another detail you must know about data types and WebAssembly and JavaScript is that JavaScript treats all numbers as 64-bit floating-point numbers. When you call a JavaScript function from WebAssembly, the JavaScript engine will perform an implicit conversion to a 64-bit float, no matter what data type you pass. However, WebAssembly will define the imported function as having a specific data type requirement. Even if you pass the same function into the WebAssembly module three times, you'll need to specify a type that the parameter passed from WebAssembly.

Let's create a module named *globals.wat* that imports three numbers from JavaScript. The WAT file in Listing 2-11 declares global variables for a 32-bit integer, a 32-bit floating-point, and a 64-bit floating-point numeric value.

globals.wat

```
(module
❶ (global $import_integer_32  (import "env" "import_i32") i32)
  (global $import_float_32    (import "env" "import_f32") f32)
  (global $import_float_64    (import "env" "import_f64") f64)

❷ (import "js" "log_i32" (func $log_i32 (param i32)))
  (import "js" "log_f32" (func $log_f32 (param f32)))
  (import "js" "log_f64" (func $log_f64 (param f64)))

  (func (export "globaltest")
  ❸ (call $log_i32 (global.get $import_integer_32))
  ❹ (call $log_f32 (global.get $import_float_32))
  ❺ (call $log_f64 (global.get $import_float_64))
  )
)
```

Listing 2-11: Importing alternative versions of the JavaScript function

We first declare the globals, including their types and import location ❶. We're also importing a log function from JavaScript. WebAssembly requires us to specify data types, so we import three functions, each with different types for the parameter: a 32-bit integer, a 32-bit float, and a 64-bit float ❷.

The variable passed into `$log_f64` is `(global.get $import_float_64)`, which tells WebAssembly that the variable we're pushing onto the stack is global. If you wanted to push a local variable called `$x` onto the stack, you would need to execute the expression `(local.get $x)`. We'll cover local variables later in this chapter.

In JavaScript, all of these functions take a dynamic variable. The JavaScript functions will be almost identical. In the function `globaltest`, we call the 32-bit integer version of the `log` function (`$log_i32`) ❸, followed by the 32-bit float (`$log_f32`) ❹ and the 64-bit float ❺. These functions will log three different messages to demonstrate the perils of moving between the native 64-bit floating-point values in JavaScript and the data types supported by WebAssembly. Before we look at the output, we need to create a JavaScript file to run our WebAssembly module. We'll start by declaring a `global_test` variable followed by a `log_message` function that will be called for each of our data types , as shown in Listing 2-12.

globals.js

```
const fs = require('fs');
const bytes = fs.readFileSync('./globals.wasm');
let global_test = null;

let importObject = {
  js: {
    log_i32: (value) => { console.log ("i32: ", value) },
    log_f32: (value) => { console.log ("f32: ", value) },
    log_f64: (value) => { console.log ("f64: ", value) },
  },
  env: {
    import_i32: 5_000_000_000, // _ is ignored in numbers in JS and WAT
    import_f32: 123.0123456789,
    import_f64: 123.0123456789,
  }
};
...
```

Listing 2-12: Setting `importObject` functions and values

In Listing 2-12, there are three different JavaScript functions passed to the WebAssembly module using `importObject`: `log_i32`, `log_f32`, and `log_f64`. Each of these functions is a wrapper around the `console.log` function. The functions pass a string as a prefix to the value from the WebAssembly module. These functions take in only a single parameter called `value`. JavaScript doesn't assign a type to the parameter in the same way WebAssembly does, so the same function could have been used three times. The only reason we didn't use the same function three times is because we wanted to change the string that prefixed the values to keep the output clear.

We chose the values in Listing 2-12 to demonstrate the limitations of each data type. We set the global variable `import_int32` to a value of `5,000,000,000`, which we pass into WebAssembly as a 32-bit integer. That value is larger than can be held by a 32-bit integer. We set the global variable `import_f32` to `123.01234567891`, which has a higher level of precision than is supported by the 32-bit floating-point variable set in our WebAssembly module. The final global variable set in the `importObject` is `import_f64`, which, unlike the previous two variables, is large enough to hold the value passed into it.

The code in Listing 2-13 instantiates our WebAssembly module and executes the `globaltest` function.

globals.js

```
...
( async () => {
  let obj = await WebAssembly.instantiate(new Uint8Array (bytes),
                                          importObject);
  ({globaltest: global_test} = obj.instance.exports);

  global_test();
})();
```

Listing 2-13: Instantiating the WebAssembly module in the asynchronous IIFE

Now that we have all our code in the JavaScript and WAT files, we can compile the WAT file into *globals.wasm* using the following `wat2wasm` call:

```
wat2wasm globals.wat
```

After compiling *globals.wasm*, we run our application using the following `node` command:

```
node globals.js
```

When you run this JavaScript file using `node`, you should see the output in Listing 2-14 logged to the console.

```
i32: 705032704
f32: 123.01234436035156
f64: 123.0123456789
```

Listing 2-14: Output logged to the console from globals.js

We passed in a value of `5,000,000,000` using our `importObject`, but our output shows a value of `705,032,704`. The reason is that a 32-bit unsigned integer has a maximum value of 4,294,967,295. If you add `1` to that number, the 32-bit integer wraps back around to a value of `0`. So if you take the 5,000,000,000 number we passed in and subtract 4,294,967,296, the result is 705,032,704. The lesson is, if you're dealing with numbers larger than a few billion, you might not be able to work with 32-bit integers. Unfortunately, as mentioned earlier, you can't pass 64-bit integers to JavaScript from WebAssembly. If you want to pass 64-bit integers to JavaScript as WebAssembly, you'll need to convert them to 64-bit floats or pass them as two 32-bit integers.

We passed a value of `123.0123456789` to our WebAssembly module, but because the 32-bit floating-point number has such limited precision, the best it can do is approximate that number, and it doesn't do a great job of it. A 32-bit floating-point number in JavaScript and WebAssembly uses 23 bits to represent the number and multiplies it by two raised to an 8-bit exponent value. All floating-point numbers are approximations, but 64-bit floating-point numbers do a much better job of those approximations. The performance differences you'll see using 32-bit versus 64-bit floating-point numbers vary with your hardware. If you want to use 32-bit floating-point numbers to improve the performance of your application, it's a good idea to know the target hardware. Some mobile devices might see a larger performance boost using 32-bit floating-point numbers.

The final message shows the 64-bit floating-point value returned to JavaScript as `f64: 123.0123456789`.

As you can see, this is the first number that remains unmodified from what we passed into the WebAssembly module. That by no means indicates that you should always use 64-bit floating-point numbers. Addition, subtraction, and multiplication typically perform three to five times faster with integers. Dividing by powers of two is also several times faster. However, division by anything but a power of two can be faster with floating-point numbers.

We'll explore these data types in more detail in Chapter 4. Now that you have a better understanding of globals and types, let's examine local variables.

Local Variables

In WebAssembly, the values stored in local variables and parameters are pushed onto the stack with the `local.get` expression. In Chapter 1, we wrote a small function that performed the addition of two parameters passed into the function that looked like Listing 2-15.

AddInt.wat

```
(module
    (func (export "AddInt")
    (param $value_1 i32) (param $value_2 i32)
    (result i32)
        local.get $value_1
        local.get $value_2
        i32.add
    )
)
```

Listing 2-15: WebAssembly module with a 32-bit integer add

Let's make a few modifications to the code. To demonstrate how we can use local variables, we'll square the value of the sum that `AddInt` returned. Create a new file named *SumSquared.wat* and add the code in Listing 2-16. The changes are called out with numbers.

SumSquared.wat

```
(module
  (func (export ❶"SumSquared")
    (param $value_1 i32) (param $value_2 i32)
    (result i32)
  ❷ (local $sum i32)

 ❸ (i32.add (local.get $value_1) (local.get $value_2))
 ❹ local.set $sum

  ❺ (i32.mul (6local.get $sum) (local.get $sum))
  )
)
```

Listing 2-16: Bit integer parameter and local variable definition

First, we change the name in the export to `SumSquared` ❶. We add a local variable called `$sum` ❷ that we'll use to store the result of the call to `i32.add` ❸. We change `i32.add` to use the S-Expression syntax. Immediately after that, we call `local.set $sum` to pop the value off the stack and set the new local variable `$sum` ❹. Then we call `i32.mul` ❺ using the S-Expression syntax, passing in the value of `$sum` for both parameters. This is done through a call to `local.get` ❻.

To test this function, create a new JavaScript file named *SumSquared.js* and add the code in Listing 2-17.

```
const fs = require('fs');
const bytes = fs.readFileSync(__dirname + '/SumSquared.wasm');
const val1 = parseInt(process.argv[2]);
const val2 = parseInt(process.argv[3]);

(async () => {
  const obj =
    await WebAssembly.instantiate(new Uint8Array (bytes));
  let sum_sq =
      obj.instance.exports.SumSquared(val1, val2);
      console.log (
        `(${val1} + ${val2}) * (${val1} + ${val2}) = ${sum_sq}`
      );
})();
```

Listing 2-17: JavaScript that executes the SumSquared WebAssembly module

Once you've created your *SumSquared.js* function, you can run it the same way you ran the *AddInt.js* file earlier, making sure to pass in two extra parameters that represent the values you want to sum and then square. The following command will add 2 and 3, and then square the result:

```
node SumSquared.js 2 3
```

The output of that run looks like this:

```
(2 + 3) * (2 + 3) = 25
```

You should now understand how to set a local variable from a value on the stack and how to add a value to the stack from a global variable. Next, let's explore how to unpack the S-Expression syntax.

Unpacking S-Expressions

So far we've been mixing the use of S-Expressions with the linear WAT syntax. However, the browser debugger doesn't keep your S-Expressions intact when you're debugging; instead, it unpacks them. Because you'll want to use your knowledge of WAT to decompile and debug WebAssembly, you'll need to understand the unpacking process. We'll walk through the process the `wat2wasm` compiler uses to unpack a short piece of WAT code. The unpacking process evaluates the expressions inside out first and then in order. It initially dives into each S-Expression looking for su bexpressions. If subexpressions exist, it evaluates the subexpressions first. If two expressions are at the same depth, it evaluates them in order. Let's look at Listing 2-18.

```
❶ (i32.mul          ;; executes 7th (last)
  ❷ (i32.add        ;; executes 3rd
    ❸ (i32.const 3) ;; executes 1st
    ❹ (i32.const 2) ;; executes 2nd
    )
  ❺ (i32.sub        ;; executes 6th
```

```
  ❻ (i32.const 9) ;; executes 4th
  ❼ (i32.const 7) ;; executes 5th
   )
)
```

Listing 2-18: Using the S-Expression syntax

First, we need to go inside our `i32.mul` expression ❶ to see if any subexpressions exist. We find two subexpressions, an `i32.add` expression ❷ and an `i32.sub` expression ❺. We look at the first of these two expressions and go inside `i32.add` ❷, evaluating `(i32.const 3)` ❸, which pushes a 32-bit integer 3 onto our stack. Because nothing is left to evaluate inside that statement, we move on to evaluate `(i32.const 2)` ❹, which pushes a 32-bit integer `2` onto the stack. Then the S-Expression executes `i32.add` ❷. The first three lines executed in the S-Expression are shown in Listing 2-19.

```
i32.const 3
i32.const 2
i32.add
```

Listing 2-19: Code from `i32.add` after it's unpacked

Now that `i32.add` is executed, the next piece to get unpacked is `i32.sub`. Similarly, the code first goes inside the S-Expression and executes the `(i32.const 9)` expression ❻ followed by the `(i32.const 7)` expression ❼. Once those two constants are pushed onto the stack, the code executes `i32.sub`. The unpacked subexpression looks like Listing 2-20.

```
i32.const 9
i32.const 7
i32.sub
```

Listing 2-20: Code from `i32.sub` after the S-Expression is unpacked

After the `i32.add` and `i32.sub` S-Expressions have been executed, the unpacked version executes the `i32.mul` command.

The fully unpacked version of the S-Expression is shown in Listing 2-21.

```
i32.const 3  ;; Stack = [3]
i32.const 2  ;; Stack = [2, 3]
i32.add      ;; 2 and 3 popped from stack, added sum of 5 pushed onto stack
[5]

i32.const 9  ;; Stack = [9,5]
i32.const 7  ;; Stack = [7,9,5]
i32.sub      ;; 7 and 9 popped off stack . 9-7=2 pushed on stack [2,5]


i32.mul      ;; 2,5 popped off stack, 2x5=10 is pushed on the stack [10]
```

Listing 2-21: Example of using the WAT stack

How the stack machine works might seem a little daunting at first, but it will feel more natural once you get accustomed to it. We recommend using S-Expressions until you're comfortable with the stack machine. The S-Expression syntax is an excellent way to ease your way into WAT if you're only familiar with higher-level languages.

Indexed Variables

WAT doesn't require you to name your variables and functions. Instead, you can use index numbers to reference functions and variables that you haven't yet named. From time to time, you might see WAT code that uses these indexed variables and functions. Sometimes this code comes from disassembly, although we've also seen people write code that looks like this occasionally.

Code that calls `local.get` followed by a number is retrieving a local variable based on the order it appears in the WebAssembly code. For example, we could have written our *AddInt.wat* file in Listing 2-21 like the code in Listing 2-22.

```
(module
    (func (export "AddInt")
  ❶ (param i32 i32)
    (result i32)
      ❷ local.get 0
      ❸ local.get 1
        i32.add
    )
)
```

Listing 2-22: Using variables

As you can see, we don't name the parameters in the `param` ❶ expression. A convenient part of this code style is that you can declare multiple parameters in a single expression by adding more types. When we call `local.get`, we need to pass in a zero indexed number to retrieve the proper parameter. The first call to `local.get` ❷ retrieves the first parameter by passing in `0` ❷. The second call to `local.get` ❸ retrieves the second parameter by passing in `1` ❸. You can also use this syntax for functions and global variables. I find this syntax difficult to read, so I won't use it in this book. However, I felt it was necessary to introduce because some debuggers use it.

Converting Between Types

JavaScript developers don't need to deal with converting between different numeric types. All numbers in JavaScript are 64-bit floating-point numbers. That simplifies coding for developers but comes at a performance cost. When you're working with WebAssembly, you need to be more familiar with your numeric data. If you need to perform numeric operations between two variables with different data types, you'll need to do some conversion. Table 2-1

provides the conversion functions you can use in WAT to convert between the different numeric data types.

Table 2-1: Numeric Type Conversion Functions

Function	Action
i32.trunc_s/f64 i32.trunc_u/f64	Convert a 64-bit float to a 32-bit integer
i32.trunc_s/f32 i32.trunc_u/f32	Convert a 32-bit float to a 32-bit integer
i32.reinterpret/f32	
i32.wrap/i64	Convert a 64-bit integer to a 32-bit integer
i64.trunc_s/f64	Convert a 64-bit float to a 64-bit integer
i64.trunc_u/f64 i64.reinterpret/f64	
i64.extend_s/i32 i64.extend_u/i32	Convert a 32-bit integer to a 64-bit integer
i64.trunc_s/f32 i64.trunc_u/f32	Convert a 32-bit float to a 64-bit integer
f32.demote/f64	Convert a 64-bit float to a 32-bit float
f32.convert_s/i32 f32.convert_u/i32 f32.reinterpret/i32	Convert a 32-bit integer to a 32-bit float
f32.convert_s/i64 f32.convert_u/i64	Convert a 64-bit integer to a 32-bit float
f64.promote/f32	Convert a 32-bit float to a 64-bit float
f64.convert_s/i32 f64.convert_u/i32	Convert a 32-bit integer to a 64-bit float
f64.convert_s/i64 f64.convert_u/i64 f64.reinterpret/i64	Convert a 64-bit integer to a 64-bit float

I omitted quite a bit of information from this table to stay focused. The `_u` and `_s` suffixes on expressions, such as `convert`, `trunc`, and `extend`, let WebAssembly know whether the integers you're working with are unsigned (cannot be negative) or signed (can be negative), respectively. A `trunc` expression truncates the fractional portion of a floating-point number when it converts it to an integer. Floating-point numbers can be promoted from an `f32` to an `f64` or demoted from an `f64` to an `f32`. Integers are simply converted to floating-point numbers. The `wrap` command puts the lower 32 bits of a 64-bit integer into an `i32`. The `reinterpret` command keeps the bits of an integer

or floating-point value the same when it reinterprets them as a different data type.

if/else Conditional Logic

One way that WAT differs from an assembly language is that it contains some higher-level control flow statements, such as `if` and `else`. WebAssembly doesn't have a boolean type; instead, it uses `i32` values to represent booleans. An `if` statement requires an `i32` to be on the top of the stack to evaluate control flow. The `if` statement evaluates any non-zero value as true and zero as false. The syntax for an `if/else` statement using S-Expressions looks like Listing 2-23.

```
;; This code is for demonstration and not part of a larger app
(if (local.get $bool_i32)
  (then
    ;; do something if $bool_i32 is not 0
    ;; nop is a "no operation" opcode.
    nop ;; I use it to stand in for code that would actually do something.
  )
  (else
    ;; do something if $bool_i32 is 0
    nop
  )
)
```

Listing 2-23: The `if/else` syntax using S-Expressions

Let's also look at what the unpacked version of the `if/else` statements look like. Unpacking an `if/else` statement might look a little different than you would expect. There is no `(then)` expression in the unpacked version. Listing 2-24 shows how the code in Listing 2-23 would look after it's unpacked.

```
;; This code is for demonstration and not part of a larger app
local.get $bool_i32

if
  ;; do something if $bool_i32 is not 0
  nop
else
  ;; do something if $bool_i32 is 0
  nop
end
```

Listing 2-24: The `if/else` statement using the linear syntax

The S-Expression `then` expression is pure syntactic sugar and doesn't exist in the unpacked version of our code. The unpacked version requires an `end` statement that doesn't exist in the S-Expression syntax.

When you're writing real programs, you use boolean logic with your `if/else` statements. Checking whether a value is zero is very limited. In JavaScript, you might have an `if` statement that looks something like this:

```
if( x > y && y < 6 )
```

To replicate this in WebAssembly, you would need to use expressions that conditionally return 32-bit integer values. Listing 2-25 shows how we would do the logic from the JavaScript `if` example if `x` and `y` were both 32-bit integers.

```
;; This code is for demonstration and not part of a larger app
(if
  (i32.and
    (i32.gt_s (local.get $x) (local.get $y) ) ;; signed greater than
    (i32.lt_s  (local.get $y) (i32.const 6) ) ;; signed less than
  )
  (then
    ;; x is greater than y and y is less than 6
    nop
  )
)
```

Listing 2-25: An `if` expression with an `i32.and` using S-Expression syntax

It looks a bit complicated in comparison. The `i32.and` expression performs a bitwise AND operation on 32-bit integers. It ends up working out because `i32.gt_s` and `i32.lt_s` both return `1` if true and `0` if false. In WebAssembly, you must keep in mind that you're using bitwise AND/OR operations; if you use an `i32.and` on a value of `2` and a value of `1`, it will result in `0` because of the way the binary AND works. You might want a logical AND instead of a binary AND, but `i32.and` is a binary AND. If you're unfamiliar with binary AND/OR operations, we discuss them in more detail in Chapter 4. In some ways, complicated `if` expressions look better when they're unpacked. Listing 2-26 shows the code in Listing 2-25 without the sugar.

```
;; This code is for demonstration and not part of a larger app
local.get $x
local.get $y
i32.gt_s      ;; pushes 1 on the stack $x > $y

local.get $y
i32.const 6
i32.lt_s      ;; pushes 1 on the stack if $y < 6

i32.and       ;; do a bitwise and on the last two values on the stack

if
  ;; x is greater than y and y is less than 6
  nop
end
```

Listing 2-26: An `if` statement with `i32.and` using stack syntax

Listing 2-27 shows there are similar expressions you can use if `$x` and `$y` are 64-bit or 32-bit floating-point numbers.

```
;; This code is for demonstration and not part of a larger app
(if
  (i32.and
❶ (f32.gt (local.get $x) (local.get $y) )
 ❷ (f32.lt  (local.get $y) (f32.const 6) )
  )
  (then
    ;; x is greater than y and y is less than 6
    nop
  )
)
```

Listing 2-27: Using `f32` comparisons but `i32.and` results

Notice that we changed `i32.gt_s` and `i32.lt_s` to `f32.gt` ❶ and `f32.lt` ❷ respectively. Many integer operations must specify whether they support negative numbers using the `_s` suffix. You don't have to do that for floating-point numbers, because all floating-point numbers are signed and have a dedicated sign bit.

There are a total of 40 comparison expressions in WebAssembly. Table 2-2 shows expressions that are useful in conjunction with the `if/else` expressions. Unless otherwise stated, these functions pop two values off the stack, compare them, and push `1` on the stack if true and `0` on the stack if false.

Table 2-2: Functions to Use with if/else

Function	Action
i32.eq i64.eq f32.eq f64.eq	Test for equality
i32.ne i64.ne f32.ne f64.ne	Not equal
i32.lt_s i32.lt_u i64.lt_s i64.lt_u f32.lt f64.lt	Less than test. The _s suffix indicates signed comparison; _u indicates unsigned.
i32.le_s i32.le_u i64.le_s i64.le_u f32.le f64.le	Less than or equal test. The _s suffix indicates signed comparison; _u indicates unsigned.

Function	Action
i32.gt_s i32.gt_u f32.gt f64.gt i64.gt_s i64.gt_u	Greater than test. The _s suffix indicates signed comparison; _u indicates unsigned.
i32.ge_s i32.ge_u i64.ge_s i64.ge_u f32.ge f64.ge	Greater than or equal test. The _s suffix indicates signed comparison; _u indicates unsigned.
i32.and i64.and	Bitwise AND
i32.or i64.or	Bitwise OR
i32.xor i64.xor	Bitwise exclusive OR
i32.eqz i64.eqz	Pop one, push 1 if the number is 0, and push 0 if it's anything else

Loops and Blocks

The branching expressions in WAT are different than branching statements you might find in an assembly language. The differences prevent the spaghetti code that comes about as the result of jumps to arbitrary locations. If you want your code to jump backward, you must put your code inside a loop. If you want your code to jump forward, you must put it inside a block. For the kind of functionality you would see in a high-level programming language, you must use the loop and block statements together. Let's explore these structures with some throwaway code examples that won't be a part of a larger app.

The block Statement

First, we'll look at the `block` expression. The block and loop statements in WAT work a bit like `goto` statements in assembly or some low-level programming languages. However, the code can only jump to the end of a `block` if it's inside that `block`. That prevents the code from arbitrarily branching to a `block` label from anywhere within your program. If the code jumps to the end of a `block`, the code that performs that jump must exist inside that `block`. Listing 2-28 shows an example.

```
;; This code is for demonstration and not part of a larger app
❶ (block $jump_to_end
  ❷ br $jump_to_end

    ;; code below the branch does not execute.  br jumps to the end of the block
  ❸ nop
  )

  ;; This is where the br statement jumps to
❹ nop
```

Listing 2-28: Declaring a `block` in WAT

The `br` ❷ statement is a branch statement that instructs the program to jump to a different location in the code. You might expect `br` to jump back to the beginning of the block where the label is defined ❶. But that isn't what happens. If you use a `br` statement within a block to jump to the block's label, it exits that block and begins to execute the code immediately outside the block ❹. That means that the code directly below the `br` statement ❸ never executes. As mentioned earlier, this code isn't meant to be used, we only wanted to demonstrate how the `block` and `br` statements work.

The way we use the `br` statement here isn't useful. Because the `br` statement always branches to the end of the labeled block, you want it inside an `if` or `else` block.

The `br_if` conditional branch in Listing 2-29 is used to branch given a condition, unlike the code in Listing 2-28.

```
;; This code is for demonstration and not part of a larger app
(block $jump_to_end
❶ local.get $should_I_branch
❷ br_if $jump_to_end

  ;; code below the branch will execute if $should_I_branch is 0
❸ nop
)

❹ nop
```

Listing 2-29: Branching to the end of the block with `br_if`

The new version of the code pushes a 32-bit integer value `$should_I_branch` onto the stack ❶. The `br_if` statement pops the top value off the stack ❷, and if that value isn't `0`, branches to the end of the `$jump_to_end` block ❹. If `$should_I_branch` is `0`, the code in the block below the `br_if` statement ❸ executes.

The loop Expression

The `block` expression always jumps to the end of the `block` on a branch. If you need to jump to the beginning of a block of code, use the `loop`

statement. Listing 2-30 shows how a WAT `loop` statement works. You might be mistaken if you think this code executes in an infinite loop.

```
;; This code is for demonstration and not part of a larger app
(loop $not_gonna_loop
  ;; this code will only execute once
❶ nop
)

;; because there is no branch in our loop, it exits the loop block at the end
❷ nop
```

Listing 2-30: A `loop` expression that doesn't loop

In fact, a `loop` expression in WAT doesn't loop on its own; it needs a branch statement located inside the loop to branch back to the beginning of the `loop` expression. A `loop` block will execute the code inside it ❶ just like a `block` expression and, without a branch, exits at the end of the block ❷.

If for any reason you want to create an infinite loop, you need to execute a `br` statement at the end of your `loop`, as shown in Listing 2-31.

```
;; This code is for demonstration and not part of a larger app
  (loop $infinite_loop
    ;; this code will execute in an infinite loop
    nop

  ❶ br $infinite_loop
  )

  ;; this code will never execute because the loop above is infinite
❷ nop
```

Listing 2-31: Branching in an infinite loop

The `br` statement ❶ always branches back to the top of the `$infinite_loop` block with every iteration. The code below the `loop` ❷ never executes.

NOTE *When you write WAT to execute in the browser, you never want to use an infinite loop because WebAssembly doesn't do your browser rendering; so you need to relinquish control back to the browser, or the browser hangs.*

Using block and loop Together

To make your loop able to break and continue, you need to use the `loop` and the `block` expressions together. Let's put together a little WebAssembly module and JavaScript app that finds factorials. The program will run a loop until we have the factorial value of the number passed into the function. That will allow us to test the `continue` and `break` functionality of our `loop` expression. Our simple loop will calculate the factorial value for each number up to some parameter value `$n` that we'll pass in from JavaScript. Then the value of `$n` factorial will be returned to JavaScript.

Create a new file named *loop.wat* and add the code in Listing 2-32.

loop.wat

```
(module
❶ (import "env" "log" (func $log (param i32 i32)))

  (func $loop_test (export "loop_test") (param $n i32)
    (result i32)

    (local $i         i32)
    (local $factorial i32)

    (local.set $factorial (i32.const 1))

  ❷ (loop $continue (block $break ;; $continue loop and $break block
    ❸ (local.set $i               ;; $i++
        (i32.add (local.get $i) (i32.const 1))
      )

    ❹ ;; value of $i factorial
      (local.set $factorial ;; $factorial = $i * $factorial
        (i32.mul (local.get $i) (local.get $factorial))
      )

      ;; call $log passing parameters $i, $factorial
    ❺ (call $log (local.get $i) (local.get $factorial))

    ❻ (br_if $break
       (i32.eq (local.get $i) (local.get $n)));;if $i==$n break from loop
    ❼ br $continue        ;; branch to top of loop
    ))

  ❽ local.get $factorial  ;; return $factorial to calling JavaScript
  )
)
```

Listing 2-32: Branching forward and backward with a `loop` and a `block`

The first expression in this module is an import of the `$log` function ❶. In a moment, we'll write this function in JavaScript and call it on every pass through our loop to log the value of `$i` factorial for each pass. We labeled the loop `$continue` ❷ and the block `$break` ❷ because branching to `$continue` will continue to execute the loop and branching to `$break` will break out of the loop. We could have done this without using the `$break` block, but we want to demonstrate how the loop can work in conjunction with a block. This allows your code to work like a `break` and a `continue` statement in a high-level programming language.

The `loop` increments `$i` ❸ and then calculates a new `$factorial` value by multiplying `$i` by the old `$factorial` value ❹. It then makes a call to log with `$i` and `$factorial` ❺. We use a `br_if` to break out of the `loop` if `$i == $n` ❻. If we don't break out of `loop`, we branch back to the top of `loop` ❼. When the loop exits, we push the value of `$factorial` onto the stack ❽ so we can return that value to the calling JavaScript.

Once you have your WAT file, compile it into a WebAssembly file using the following command:

```
wat2wasm loop.wat
```

Now we'll create a JavaScript file to execute the WebAssembly. Create a *loop.js* file and enter the code in Listing 2-33.

loop.js

```
const fs = require('fs');
const bytes = fs.readFileSync(__dirname + '/loop.wasm');
❶ const n = parseInt(process.argv[2] || "1"); // we will loop n times
let loop_test = null;

let importObject = {
  env: {
  ❷ log: function(n, factorial) { // log n factorial to output tag
      console.log(`${n}! = ${factorial}`);
    }
  }
};


( async() => {
❸ let obj = await WebAssembly.instantiate( new Uint8Array(bytes),
                                           importObject );

❹ loop_test = obj.instance.exports.loop_test;

❺ const factorial = loop_test(n); // call our loop test
❻ console.log(`result ${n}! = ${factorial}`);
❼ if (n > 12) {
    console.log(`
    ================================================================
    Factorials greater than 12 are too large for a 32-bit integer.
    ================================================================
    `)
  }
})();
```

Listing 2-33: Calling the `loop_test` from JavaScript

The `log` ❷ function, which our WAT code will call, logs a string to the console with the values of `n` ❶ and `n` factorial passed from the WAT `loop`. When we instantiate the module ❸, we pass a value of `n` to the `loop_test` ❹ function. The `loop_test` function finds the factorial as a result ❺. We then use a `console.log` ❻ call to display the value of `n` and `n` factorial. We have a check at the end to make sure the number we enter isn't greater than a value of `12` ❼, because signed 32-bit integers only support numbers up to about 2 billion. Run *loop.js* using `node` by executing the following on the command line:

```
node loop.js 10
```

Listing 2-34 shows the output you should see on the command line.

```
1! = 1
2! = 2
3! = 6
4! = 24
5! = 120
6! = 720
7! = 5040
8! = 40320
9! = 362880
10! = 3628800
result 10! = 3628800
```

Listing 2-34: Output from loop.js

Now that you know how loops work in WAT, let's look at branch tables.

Branching with br_table

Another way to use the `block` expression in WAT is in conjunction with a *`br_table` expression*, which allows you to implement a kind of `switch` statement. It's meant to provide the kind of jump table performance you get with a `switch` statement when there are a large number of branches. The `br_table` expression takes a list of blocks and an index into that list of blocks. It then breaks out of whichever block your index points to. The awkward thing about using a branch table is that the code can only break out of a block it's inside. That means you must declare all of your blocks ahead of time. Listing 2-35 shows what the WAT code looks like to build a `br_table`.

```
;; This code is for demonstration and not part of a larger app
❶ (block $block_0
  (block $block_1
  (block $block_2
  (block $block_3
  (block $block_4
  (block $block_5
❷ (br_table $block_0 $block_1 $block_2 $block_3 $block_4 $block_5
      (local.get $val)
    )
❸ ) ;; block 5
  i32.const 55
  return

  )   ;; block 4
  i32.const 44
  return

  )    ;; block 3
  i32.const 33
  return

  )    ;; block 2
  i32.const 22
```

```
return

)    ;; block 1
i32.const 11
return

)   ;; block 0
i32.const 0
return
```

Listing 2-35: Using the `br_table` syntax from within WAT

We define all the `block` expressions before the `br_table` expression ❶. So when the `br_table` expression is called ❷, it's not always completely clear where in the code it will jump. This is why we added the comments ❸ in the code indicating which `block` was ending.

The `br_table` provides some performance improvement over the use of `if` expressions when you have a large number of branches. In our testing, using the `br_table` expression wasn't worthwhile until there were about a dozen branches. Of course this will depend on the embedding environment and hardware it runs on. Even at this number of branches, the `br_table` was still slower on Chrome than `if` statements. Firefox with about a dozen branches was noticeably faster with the `br_table` expression.

Summary

In this chapter, we covered many of the WAT programming basics. After learning to create and execute a WebAssembly module in Chapter 1, you moved on to creating the traditional hello world! application in this chapter. Creating a hello world! application is a bit more advanced in WAT than in most programming languages.

After completing a few initial programs, we began looking at some of the basic features of WAT and how they differ from a traditionally high-level language like JavaScript. We explored variables and constants and how they can be pushed onto the stack using WAT commands. We discussed the S-Expression syntax and how to unpack it. We also briefly mentioned indexed local variables and functions, and introduced you to that syntax. You learned the basic branching and looping structures, and how to use them within the WAT syntax. In the next chapter, we'll explore functions and function tables in WAT, and how they interact with JavaScript and other WAT modules.