WebAssembly
from the Ground Up

From hand crafting bytecodes to a real compiler for a toy programming language

Mariano Guerra & Patrick Dubroy

Introduction

We're pretty excited about WebAssembly.

I mean, of course we are. Who would decide to write a book about something they're not excited about?

There's certainly been a lot of hype. Since the day that WebAssembly was announced, people have been making some pretty wild claims about it. Things like:

- "Add WebAssembly, get performance."
- "It's going to replace JavaScript!"
- "WebAssembly is the future of computing!"

Of course, none of these are really true (unless you add a lot of caveats). But we do think WebAssembly is a legitimately transformative technology that's worth learning about. You probably do too. That's why you're here, right?

When we started writing this book, we had some conversations with potential readers. Something kept coming up: many of them told us that they'd heard lots *about* WebAssembly, but they still didn't really understand what it *is*.

By the end of this chapter, you should have a good understanding of what WebAssembly is, what makes it special, and what it's good for. In the rest of the book, we'll take a much more hands-on approach.

What *is* WebAssembly?

WebAssembly (or *Wasm* for short) is a low-level bytecode format originally designed for the web. It's mainly intended as a compilation target, allowing languages like C++, Go, and Rust to be executed in the browser at near-native speed.

Bytecode is a term for portable, low-level code. The name comes from the fact that individual instructions are often encoded with a single byte. Probably the most well-known example is the Java bytecode format.

The basic idea is this: rather than producing machine code for a specific architecture (like ARM or x86), your C++ or Rust compiler can instead produce WebAssembly bytecode. Later, a browser will download the bytecode, and translate it to native machine code on the user's device.

Much like machine code, Wasm bytecode is a compact, binary representation of low-level instructions. The majority of the instructions are basic operations on numeric values (e.g., `add`, `abs`, `xor`). There are also instructions for reading from and writing to memory, and some basic control-flow instructions (`call`, `loop`, `if`).

In case you're wondering, it's pronouced *wah-zum*. Depending on your accent, it might rhyme with *has 'em* or *awesome*.

But, unlike machine code, these instructions don't target any specific CPU. They're meant to be portable and close enough to real CPU instructions to be translated as fast and efficiently as possible.

So, one way to describe WebAssembly is: a low-level instruction set for a virtual CPU. The bulk of the spec is about defining these instructions (172 of them, in version 1.0) and their meaning (the *semantics*). It also specifies how instructions are packaged into a *module*, a kind of shared library of functions that can be called from the host language (e.g. JavaScript).

Why is it a big deal?

Unlike previous attempts to bring native code to the web, like — ActiveX, Native Client, or asm.js — WebAssembly is an open standard that's implemented in all the major browsers. That alone makes it a huge deal.

But there's more to it than that. This isn't the place for a deep technical comparison, but each of those previous solutions had some significant shortcomings that WebAssembly has improved on. The designers of WebAssembly managed to find a solution for low-level code that has four key properties: it's **safe**, **fast**, **portable**, and **compact**.

First, and perhaps most important, WebAssembly is **safe by default**. Its design makes it easy to verify that a module doesn't do anything it's not allowed to do. So untrusted modules can safely be executed in the same address space as other code.

Second, it's **fast**. As we already mentioned, WebAssembly bytecode isn't executed directly. But it's designed to be simple and fast to compile to native code. It's also easy to validate, and the safety features add relatively little overhead at runtime.

What sets WebAssembly apart is that its bytecode format is both **portable** and **compact**. Once your code is compiled to a Wasm module, it will run on any platform or hardware architecture. And its compact representation helps prevent large download times.

And, though it was originally designed for the web, one of the goals of WebAssembly was to be useful outside the browser too. It turns out that the properties that make it great for the web also
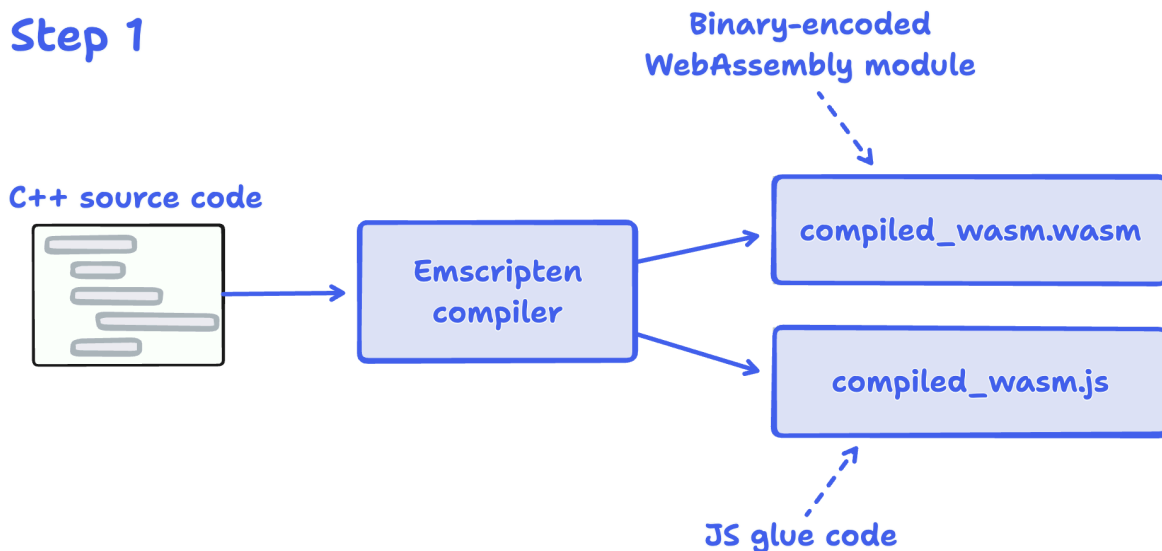
make it shine in some non-browser use cases.

For example, the fact that Wasm is safe, fast, and portable makes it a great foundation for cloud computing. Think about it: a cloud provider is interested in running customer code with as little overhead as possible, without compromising security. And the fast startup times, as compared to bringing up a full virtual machine, are great for customers and end users.

All this to say, we think WebAssembly is pretty great.
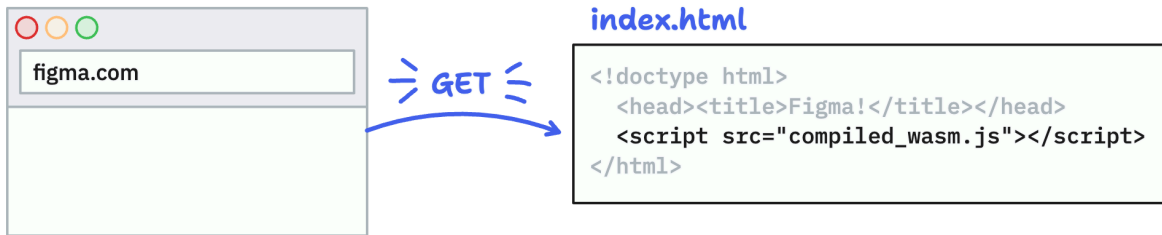
A day in the life of a Wasm module

To better understand how everything fits together, it might help to look at a concrete example. Figma is probably one of the most well-known web applications that uses WebAssembly. Let's look at what the typical flow might look like.

The first step is that an engineer at Figma writes some C++ code, and compiles it with a compiler like Emscripten. That produces a binary-encoded WebAssembly module (usually with a `.wasm` extension), as well as some JavaScript support code. A `.wasm` file on its own is not directly executable — it's more like a shared library. In order to actually execute the WebAssembly, you need at least a small amount of JavaScript code.

Emscripten is an open source compiler toolchain for compiling C and C++ to WebAssembly. Its main tool, `emcc`, can be used as a drop-in replacement for C/C++ compilers like `gcc` and `clang`.

Step 2

The next step happens when a user visits figma.com in their web browser. Their browser first downloads the HTML, which includes a link to the `compiled_wasm.js` script, so it downloads that file as well.

So what's in that file? At minimum, it will contain something like this:

```
compiled_wasm.js

const source = fetch('compiled_wasm.wasm');
const {instance} = await WebAssembly.instantiateStreaming(source);
instance.exports.renderFigma();
```
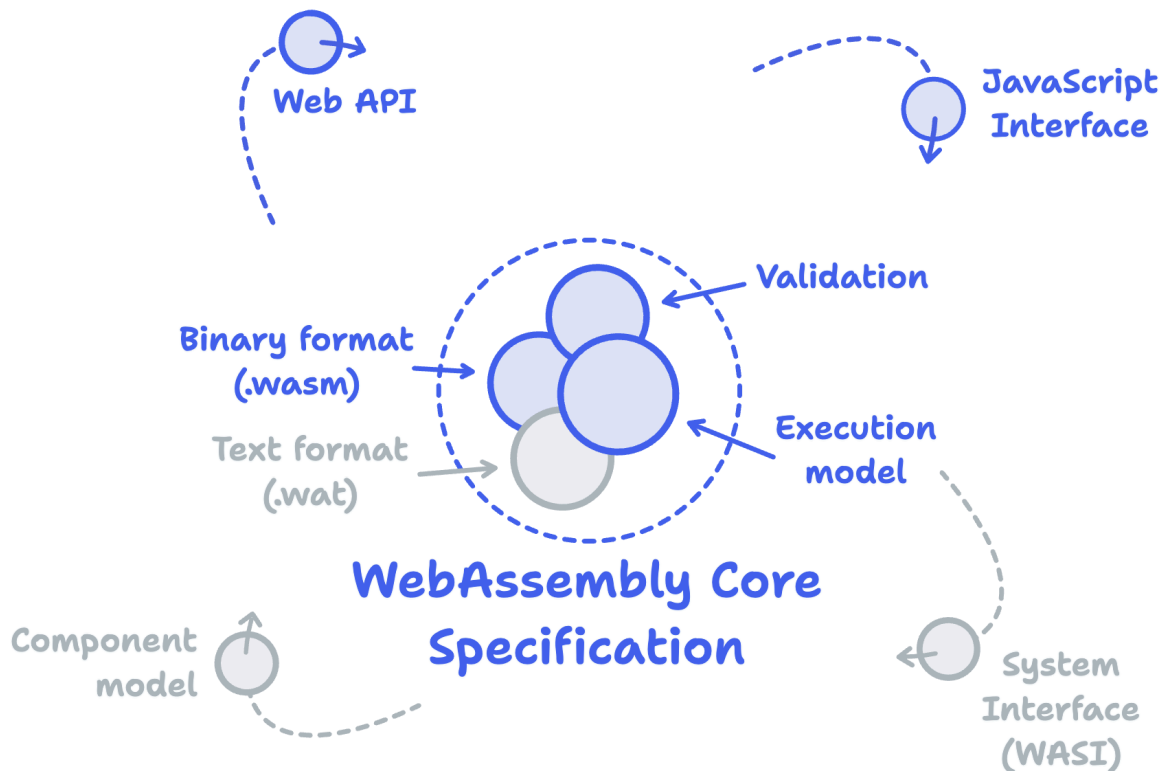
Here's what this code does:

- It fetches the binary WebAssembly module.
- Then, it calls `instantiateStreaming`, which verifies, compiles, and instantiates the the module.
- Finally, it calls one of the module's exported functions. Remember, a Wasm module is not directly executable. From the outside, it's very much like a regular JavaScript module: it exports some functions, which we can choose to call (or not).

You can, however, specify a *start function* to initialize the state of the module when it's instantiated.

A lay of the land

Now that we've talked about a concrete example, it's helpful to zoom out a bit.

One of the reasons that WebAssembly can be a bit confusing for newcomers is that it's not really a single thing — it's really a collection of different technologies and formats.

An atomic model of WebAssembly. Things in gray are not covered in this book.

At the center of the WebAssembly universe is the WebAssembly Core Specification. It defines several things:

- The execution model — the instruction set and a formal definition of how the instructions are interpreted.
- The binary module format (`.wasm`) — the primary way of packaging and distributing WebAssembly.
- The validation rules — what it means for a module to be well-formed.
- A textual format (`.wat`), which offers a human-readable way of defining modules.

This book covers the first three topics: the execution model, the binary module format, and the validation rules. Most of the focus is on the instruction set and execution model, since we think these are the most interesting aspects. We also touch on validation, because that's a critical aspect of the WebAssembly safety model.

We do not cover the textual format. This is arguably not really a "core" aspect at all, since most producers (e.g. compilers) and consumers (e.g. browsers) don't need to support it.

The WebAssembly Core Specification defines an "inside view" of a Wasm implementation, but there are many things it doesn't define:

- How are modules instantiated?
- How are imports provided, and how are exports accessed?

These kinds of details are provided by the *host environment*, and are covered by different specifications. For example, web browsers and JavaScript runtimes like Node.js implement the WebAssembly JavaScript Interface, as well as the WebAssembly Web API. Since this book uses JavaScript, we'll cover large parts of those specifications as well.

For non-browser runtimes, there are efforts to specify APIs that cover things like filesystems, networking, etc., as well as a component model. This is known as the WebAssembly System Interface, or WASI. Due to the large scope, and the fact that the proposal is still in flux, it's not covered in this book.

About this book

To really understand what WebAssembly is and what makes it special, you need to understand the low-level details. In this book, we use a hands-on approach to teach you the core of WebAssembly: the instruction set and the module format. Since WebAssembly is primarily a compilation target, we think the best way to learn the details is by writing a compiler. Really.

In this book, you'll create a compiler that compiles a simple programming language down to WebAssembly. The focus is on WebAssembly, *not* the finer details of parsing — so the compiler is built with Ohm, a user-friendly parsing toolkit.

All of the code presented in the book is written in JavaScript. One of the reasons is that it's a language that most developers are at least familiar with. So the code is hopefully understandable, even if it's not your language of choice. We do try to stick to a subset of the language and avoid some newer and lesser-known features.

All of the code you need is included in the text, along with instructions for running it. Nothing is left "as an exercise for the reader". We expect that most readers will build up the code step-by-step, in the order presented in the book, but full code for each chapter is also available on GitHub.

About the code

This book is very much centered around code. So, before you dive into the technical content, it's worth understanding our thought process and how the code is intended to be used.

You'll get the most out of the book if you have your text editor or IDE open at the same time. Our intention is that most readers will type out (or copy and paste) the code and run it themselves.
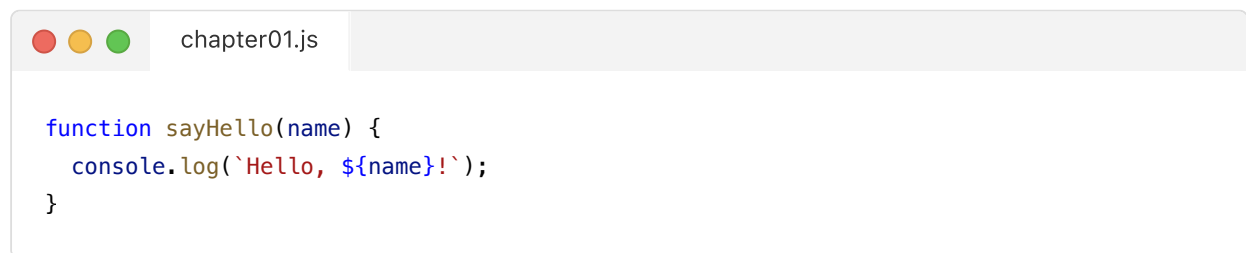
All the code in the book is open source and provided under the MIT License. That means that you're also free to modify the code and generally do whatever you'd like with it — as long as you respect the terms of the license, of course.

Structure and mechanics

In each chapter, we build up the source code for a single source file. The files are named like so: chapter01.js, chapter02.js, etc.

Conceptually, the functions and variables that we create belong to a single namespace which we build up over the course of the book. Each source file depends on the source from the previous chapter. So, the chapters end with exports, and the following chapter begins by importing anything it needs from the previous chapter.

Below is an example of a code snippet. Notice that it has a tab with the chapter filename (chapter01.js). If you're reading the online version of the book, you'll also see a copy button (⧉) in the top right corner.

```chapter01.js
function sayHello(name) {
  console.log(`Hello, ${name}!`);
}
```

There are some code snippets that are just used for explanatory purposes, and aren't intended to be added to the module for that chapter. You can recognize these blocks by the fact that they *don't* have a tab with the chapter filename.

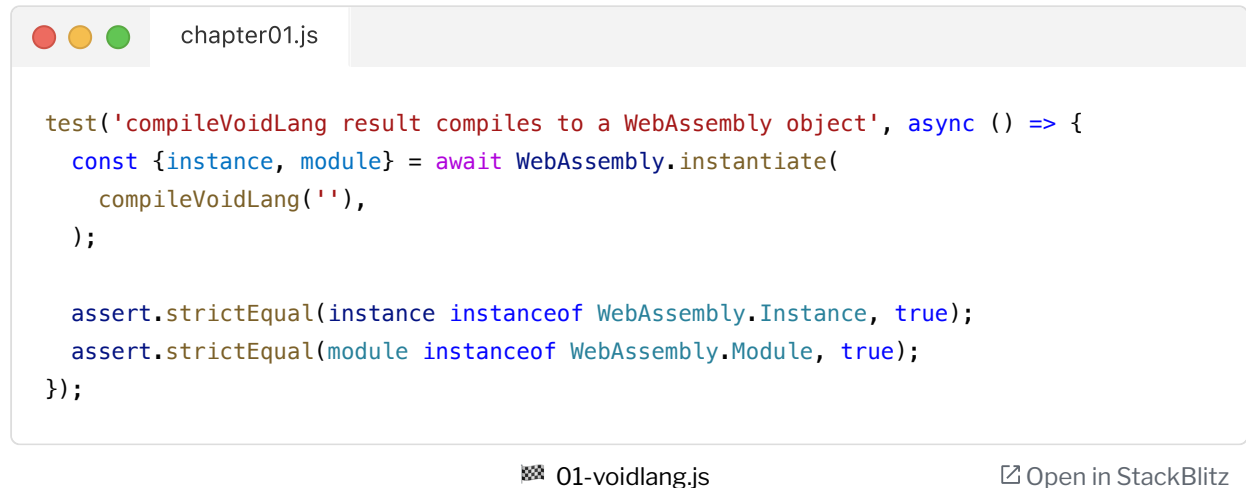Here's an example of a snippet that's not intended to be copied into the chapter source file:

```
console.log(int32ToBytes(42));
```

Checkpoints

In each chapter, there are a number of *checkpoints*. A checkpoint is a point where the code is in a consistent state. If you are following along by cutting and pasting or typing out the code, a checkpoint is a good spot to verify that your code parses, that the tests pass, etc.

Here's an example of a checkpoint:

```
● ● ●        chapter01.js

test('compileVoidLang result compiles to a WebAssembly object', async () => {
  const {instance, module} = await WebAssembly.instantiate(
    compileVoidLang(''),
  );

  assert.strictEqual(instance instanceof WebAssembly.Instance, true);
  assert.strictEqual(module instanceof WebAssembly.Module, true);
});
```

▨ 01-voidlang.js   ⎘ Open in StackBlitz

When you encounter a checkpoint, you can use the "Copy up to here" button to copy all the relevant code up to that point.

The code for each checkpoint can also be found in a public GitHub repo for the book. You can also click on the "Open in StackBlitz" link to open the code in StackBlitz, an online IDE. From there, you can compare your code to the official version, run the tests, etc.

Running the code

All of the code in the book has been tested with Node.js. If you'd like to run the code locally, you'll first need to ensure you have a recent version (>= 18) of Node installed.

We recommend starting with the project template that we provide. You can set up your project by running the following command:

```
npm create wasmgroundup@latest
```

If you don't have Node installed, you can always use the "Open in StackBlitz" links to open the code on StackBlitz, which provides a virtualized Node environment.

Testing

In JavaScript, you typically put a module's tests in a separate source file. For example, a module named `geometry.js` might have its tests in a file named `geometry.test.js`.

Other languages — including Python, Rust, and OCaml — provide support for embedding tests in the same source file as the code being tested. We like that approach, because it keeps things nice and simple. So that's what we'll be doing in this book.

However, if we just naïvely write our implementation and tests in the same module, we'll run into a small problem. Each chapter imports from the previous one, and because of that, running:

```
node chapter02.js
```

...will execute *all* of the tests, including the ones in `chapter01.js`.

To address this, we've written a small helper to ensure that tests are only run for the script that was specified on the command line. It's called `makeTestFn` (short for "make test function"), and we'll use it in each chapter like this:

```
chapter01.js

const test = makeTestFn(import.meta.url);
```

- In Node, `import.meta.url` is a `file:` URL giving the absolute path to the current module.

If you're starting with the project template that we mentioned in the previous section, there's nothing more to do — `makeTestFn` is already included in `chapter01.js`.

In case you're not starting with the template, the implementation is below. There's no need to study it closely, but we've provided some explanation in case you're curious.

```
chapter01.js

import assert from 'node:assert';
import {basename} from 'node:path';
import process from 'node:process';
import {default as nodeTest} from 'node:test';
import {fileURLToPath} from 'node:url';

function makeTestFn(url) {
  const filename = fileURLToPath(url);
```

```
    // Return a function with the same interface as Node's `test` function.
    return (name, ...args) => {
      // Only register the test if the current module is on the command line.
      // All other tests are ignored.
      if (process.argv[1] === filename) {
        // Add the chapter name to the test description.
        const chapterName = basename(filename, '.js');
        nodeTest(`[${chapterName}] ${name}`, ...args);
      }
    };
  }
```

In the rest of the book, we'll use this helper when writing tests, using:

```
const test = makeTestFn(import.meta.url);
```

at the top of each source file, rather than doing this:

```
import test from 'node:test';
```

. . .

Alright! With those preliminaries behind us, it's time to jump in to WebAssembly.

Minimum Viable Compiler

As you learned in the Introduction, WebAssembly can be described in multiple ways, depending on the perspective we take. But its most tangible representation is a file with the `.wasm` extension: a module encoded in its binary format.

Ultimately, our goal in this book is to build a small compiler that compiles a simple programming language to WebAssembly. To do that, we'll need to know how to produce a valid Wasm module.

In this chapter, we're going to take a close look at the WebAssembly module binary format. We'll write some JavaScript that generates a minimal module, and use parts of the WebAssembly JavaScript API to load the module and call functions defined in it.

The binary module format is defined in the WebAssembly Core Specification. You'll notice that we back up many of our explanations with reference to the relevant part of the spec. One of our goals with this book is to convince you that the spec is a valuable resource that's worth getting familiar with.

🛠️ Setup

To get started, make sure you have the following at the top of `chapter01.js`:

chapter01.js

```js
import assert from 'node:assert';
import {basename} from 'node:path';
import process from 'node:process';
import {default as nodeTest} from 'node:test';
import {fileURLToPath} from 'node:url';

function makeTestFn(url) {
  const filename = fileURLToPath(url);
  // Return a function with the same interface as Node's `test` function.
  return (name, ...args) => {
    // Only register the test if the current module is on the command line.
    // All other tests are ignored.
    if (process.argv[1] === filename) {
      // Add the chapter name to the test description.
      const chapterName = basename(filename, '.js');
      nodeTest(`[${chapterName}] ${name}`, ...args);
    }
  };
}
```

```
const test = makeTestFn(import.meta.url);
```

This snippet is explained in About the Code, so you may want to read that if you
haven't already.

To make sure that everything is working, try adding a basic test:

```
chapter01.js

test('setup', () => {
  assert(true);
});
```

You can run the test by executing the script with Node:

```
node chapter01.js
```

Notice that we didn't need to pass the `--test` option to Node. That option tells Node to
automatically search for test files. However, when you provide the script name on the command
line, it will always execute tests contained within that file even if the `--test` option was not used.

You should see that the test is passing:

```
✔ [chapter01] setup (4.007875ms)
ℹ tests 1
ℹ suites 0
ℹ pass 1
ℹ fail 0
ℹ cancelled 0
ℹ skipped 0
ℹ todo 0
ℹ duration_ms 7.984167
```

Getting started

Building a compiler sounds like a huge and complex task, but if you decompose it into small and
incremental steps, you'll see that it's not all that hard. And on top of that, it can be pretty fun.

As with any other project, one of the hardest things is figuring out where to start. A good way to get started is to ask — what is the minimum viable WebAssembly compiler? Or more generally — what's the simplest compiler?

The simplest compiler is probably one that compiles the simplest programming language, but then, what's the simplest programming language?

If we see a compiler as a function that takes text (source code) as input and returns a runnable program as output, then the simplest programming language is a language that only takes an empty string as input and generates a valid program that does nothing.

Let's call that programming language *Void Lang*. Its compiler will be a very simple WebAssembly compiler generating the simplest possible WebAssembly module.

Handling binary data in JavaScript

The most common way of handling binary data in JavaScript is *typed arrays*. These APIs provide array-like interfaces for reading and writing an underlying chunk of memory.

We recommend reading the MDN docs on typed arrays if you're not already familiar with these APIs.

A `Uint8Array` lets us manipulate the individual bytes much like an array of numbers. *Uint* is short for "unsigned integer", and the *8* means that each entry is 8 bits (one byte). Here's an example:

```
const arr = new Uint8Array(2); // An array of two bytes.
arr[0] = 5;
arr[1] = 7;
console.log(arr[0] + arr[1]); // Prints '12'
```

A byte can only hold one of 256 (2^8) possible values, which means that every entry must be in the range 0–255. What happens if we try to assign a larger number?

```
const arr = new Uint8Array(2);
arr[0] = 300;
console.log(arr[0]); // Prints '44'
```

Ok, we'll need to be careful about that. 😬

Hexadecimal

Sometimes it's helpful to use the hexadecimal (base-16) representation for binary data. In hexadecimal (or *hex*), a byte can always be represented by two characters — from 0 (`00`) to 255 (`ff`). This property makes it convenient for fixed-width layouts in a monospace typeface. For example, here's an array of 8 bytes:

```
const arr = Uint8Array.from([0, 1, 2, 3, 255, 254, 253, 252]);
```

Using hex, we could show the contents of the array like this:

```
00 01 02 03 ff fe fd fc
```

It's also common to use hex to imply that the value shouldn't be interpreted as a number. For example, an ASCII newline character can be represented in different ways:

- `10` in decimal
- `0x0a` or `0xa` in hexadecimal

In this case, the hexadecimal representation can help make it clear that the *meaning* of the byte is not "the number 10".

Okay! Now we're ready to jump in and implement Void Lang.

Our first compiler

Let's create a function that:

- Takes the code for a program as input.
- "Parses" it by checking that it's a valid program, in this case the empty string.
- Returns an array of bytes which encodes a minimal WebAssembly module.

```
                    chapter01.js

function compileVoidLang(code) {
  if (code !== '') {
    throw new Error(`Expected empty code, got: "${code}"`);
  }
  return Uint8Array.from([0, 97, 115, 109, 1, 0, 0, 0]);
```

```
  }
```

But wait, what are those eight bytes (`[0, 97, 115, 109, 1, 0, 0, 0]`) that we're putting into the `Uint8Array` ? Don't worry, we'll get to that in a moment.

First, let's verify that we are indeed producing a valid Wasm module. We'll use the WebAssembly JavaScript API to *instantiate* the module. Much like a class in object-oriented programming, a WebAssembly module is a kind of template which can be used to create multiple *module instances*.

```
● ● ●        chapter01.js

test('compileVoidLang result compiles to a WebAssembly object', async () => {
  const {instance, module} = await WebAssembly.instantiate(
    compileVoidLang(''),
  );

  assert.strictEqual(instance instanceof WebAssembly.Instance, true);
  assert.strictEqual(module instanceof WebAssembly.Module, true);
});
```

▨ 01-voidlang.js   ⬈ Open in StackBlitz

This is the first code checkpoint in the book (notice the ▨ icon). Try clicking "Open in StackBlitz" to see the code and run the tests. For more information, see About the code.

- We pass the module bytes to `instantiate` , and it returns both a module instance *and* an object representing the compiled module.

In this book, we generally only care about the `instance` . The compiled module is useful if you want to create more instances without compiling and validating the bytecode all over again, but we won't be doing that.

Well — congratulations, you just implemented your first WebAssembly compiler!

From raw bytes to meaningful fragments

Let's return to those eight bytes (`[0, 97, 115, 109, 1, 0, 0, 0]`) we used in `compileVoidLang` . To understand their meaning, we need to learn about the first two values in every WebAssembly module binary:

- The module "magic number"
- The module version

Many file formats start with a *magic number*: a predefined sequence of bytes used to identify the format. The magic number for a WebAssembly module is the string `"\0asm"` encoded as UTF-8*. That is, it consists of the following four bytes:

---

* Text encoding is a rabbit hole far too deep for this book; it's enough to know that in both ASCII and UTF-8, each letter in the English alphabet is encoded as a single byte.

- `0x00`
- `0x61` ("a")
- `0x73` ("s")
- `0x6d` ("m")

Since UTF-8 is used for all strings that appear in a WebAssembly module, it's worth writing a small helper for converting JavaScript strings to UTF-8:

```
chapter01.js

function stringToBytes(s) {
  const bytes = new TextEncoder().encode(s);
  return Array.from(bytes);
}
```

- We use the `TextEncoder` interface, which is a built-in Node and browser API for converting strings to UTF-8.
- `TextEncoder.encode` returns a `Uint8Array`. However, it's more convenient if all our helpers return regular arrays, and we convert to a `Uint8Array` only at the end. So we convert the result to an Array before returning.

Now we can write ourselves a small helper function to produce the magic number:

```
chapter01.js

function magic() {
  // [0x00, 0x61, 0x73, 0x6d]
  return stringToBytes('\0asm');
}
```

- `\0` is the JavaScript escape sequence for the `NUL` character (character code 0).

The magic number is followed by a 32-bit integer indicating which version of the WebAssembly module version is being used. Currently, the only valid version number is `1`, but this may change in the future. Here's what the official specification says:

> *The version of the WebAssembly binary format may increase in the future if backward-incompatible changes have to be made to the format. However, such changes are expected to occur very infrequently, if ever. The binary format is intended to be forward-compatible, such that future extensions can be made without incrementing its version.*
>
> **§ 5.5.15 Modules**

In any binary data format, if you want to encode integer values greater than 255 — that is, values that take up more than one byte — you need to define the *byte order*. WebAssembly uses what's known as *little-endian* ordering, which means the bytes appear in the opposite order of what you might expect: `01 00 00 00` . So the smallest part of the number (the least sigificant byte, or the "little end") comes first.

If you've never heard of byte ordering, don't worry — we'll cover it in more detail later in the book.

Just as we did with the magic number, let's add a helper for the version number:

```
chapter01.js

function version() {
  return [0x01, 0x00, 0x00, 0x00];
}
```

Now we can refactor the `compileVoidLang` function to use descriptively-named functions rather than an array of seemingly-meaningless numbers:

```
chapter01.js

function compileVoidLang(code) {
  if (code !== '') {
    throw new Error(`Expected empty code, got: "${code}"`);
  }
  return Uint8Array.from([...magic(), ...version()]);
}
```

- Since both `magic` and `version` return an array, we use JavaScript's *spread syntax* (the `...` operator) to "unpack" those return values into a single, flat array.

As we construct more complex modules, it will be easy to forget the `...` somewhere. Instead, we can use the built-in `flat` method on the array to recursively flatten the entire contents:

```js
function compileVoidLang(code) {
  if (code !== '') {
    throw new Error(`Expected empty code, got: "${code}"`);
  }
  const bytes = [magic(), version()].flat(Infinity);
  return Uint8Array.from(bytes);
}
```

- The `flat()` method takes an optional parameter specifying how deep a nested array structure should be flattened. We specify a depth of `Infinity` to ensure that the final result is a flat array of numbers.

As we'd hope when we're refactoring, we get the same result, but now it's easier to understand the meaning of each part.

Doing nothing but with an empty function

Void Lang is a good start, but you may be thinking "that's not even a program!" And you're right — the smallest viable WebAssembly module doesn't have something we can *run*. The closest to running we can do is to create a module instance.

Let's extend our compiler to emit a module with a function we can call. The smallest step forward is to compile a function that does nothing: a function that takes no arguments, does nothing and returns nothing.

Let's call our new language *Nop Lang*: a language whose only valid program is an empty string and compiles to a module that has a function called `main` that does nothing and returns nothing.

Since JavaScript functions always return a value, when calling a WebAssembly function that returns nothing from JavaScript it will return `undefined`. Even though this is the smallest step possible, it requires the introduction of four new WebAssembly module sections.

After the *preamble* (magic number and version number) a WebAssembly module consists of zero or more sections. In other words, every section is optional — an omitted section is treated as if the section was there, but empty. In Void Lang we had no functions, so we could omit every section.

To us humans, a function may be a thing defined in a single place, but WebAssembly expects a function definition to be split into three sections:

1. The *type section* contains the function's type signature. It's about saving space: if multiple functions share the same signature, they can all refer to the same entry in this section.

2. The *function section* is where the function is "declared". Inside the module, the function is referred to by its index into this section — for example, when calling or exporting the function.

3. The *code section* contains the body of the function with the executable instructions.

Yes, a function is split into (at least) three parts. Writing compilers is mostly about turning code from a human readable format into a computer readable format, and WebAssembly wants functions split into multiple sections.

The split between the code section and the function section is to enable parallel and streaming compilation. It means that the host knows *all* of the functions and their signatures up front, and it can start compiling the first function's code as soon as it's received.

Notice that we didn't mention where the name of the function goes! This is because internally, WebAssembly only refers to functions by *index*. The function section and the code section must be the same length, so "function #2" appears at index 2 in both sections. We only give names to functions when we export them in the module's public API.

Section basics

The sections in a WebAssembly module have a common structure, consisting of three parts:

1. The *section identifier* is a single byte that specifies which section is being defined. For example, the type section has the id 1 , and the function section has the id 3 .

2. A *size* field, which represents the size of the contents (i.e., the section payload) in bytes.

3. The *contents* of the section. The structure of this can vary based on the type of section, but most of them are structured as a list of zero or more entries.

To save space, all integers in a WebAssembly module are encoded with a *variable-length encoding*. This means that the algorithm will try to encode the value using only one byte, and if it doesn't fit it will try two, then three, and so on. (The version number which appears in the preamble is the only exception; it always takes four bytes.)

The UTF-8 text encoding is an example of a variable-length encoding.

The encoding algorithm WebAssembly uses is called *LEB128*, short for "Little Endian Base 128". It can encode up to 128 (2^7) unique values per byte. So, for unsigned integers, any value from 0 to 127 can fit in a single byte; and for signed integers, -64 to 63.

We'll avoid writing a full implementation of LEB128 right now. But there's a trick we can take advantage of: for non-negative integers that fit in a single byte, no encoding is necessary! Since we are creating small and fixed modules, that restriction is fine for now.

You can learn the full details of LEB128 in the (optional) deep dive on Variable-Length Integer Encoding.

Now that you've learned a bit more about sections, we can now hand craft a module with the sections required to define a function:

```js
const nopLangBytes = [
  magic(),
  version(),

  // ----- type section -----

  1, // Section identifier
  4, // Section size in bytes
  1, // Number of entries that follow

  // type section – entry 0
  0x60, // Type `function`
  0, // Empty vector of parameters
  0, // Empty vector of return values

  // ----- function section -----

  3, // Section identifier
  2, // Section size in bytes
  1, // Number of entries that follow
```

```
    // function section — entry 0
    0, // Index of the type section entry

    // ----- code section -----

    10, // Section identifier
    4, // Section size in bytes
    1, // Number of entries that follow

    // code section — entry 0
    2, // Entry size in bytes
    0, // Empty vector of local variables
    11, // `end` instruction
  ].flat(Infinity);
```

Let's verify that it is indeed a valid module:

```
test('hand crafted module with a function', async () => {
  const {instance, module} = await WebAssembly.instantiate(
      Uint8Array.from(nopLangBytes),
  );
  assert.strictEqual(instance instanceof WebAssembly.Instance, true);
  assert.strictEqual(module instanceof WebAssembly.Module, true);
});
```

A small but important detail is that the sections in a module must appear in ascending order by id. For example, the type section (whose id is 1) must appear before the function section (whose id is 3). This is done to make it easier for WebAssembly runtimes to decode and compile modules in one pass.

You may have noticed that we skipped one section identifier (we went from 1 to 3). This is because the section with id 2 is the import section, and we aren't importing anything yet.

Adding more structure

The "module as a flat array of bytes" may have been useful to get started but it's not a sustainable way to write a compiler. Our goal in this section is to build up a small library of functions for constructing WebAssembly modules, which will be used in the rest of the book.

As much as possible, these functions will match the naming and structure that's used in the official WebAssembly specification. The names are sometimes a bit terse, but by doing it this way, we can have a very thin layer over the actual module format and avoid introducing any new, idiosyncratic abstractions.

First let's write some stubs for LEB128-encoded integers. As we explained earlier, we can avoid doing any actual encoding if we only accept positive values that fit in a single byte. Here's a function for unsigned integers:

```javascript
function u32(v) {
  assert(v >= 0, `Value is negative: ${v}`);
  if (v < 128) {
    return [v];
  } else {
    throw new Error('Not implemented');
  }
}
```

- It takes a number as an argument and returns an array of bytes which is the unsigned LEB128 encoding of that number.
- For now, we only accept integers that can be trivially encoded in a single byte and fail otherwise.

Let's implement a similar function for signed integers:

```javascript
function i32(v) {
  if (v >= 0 || v < 64) {
    return [v];
  } else {
    throw new Error('Not implemented');
  }
}
```

- Notice that for signed integers, the range of values that can be trivially encoded in a single byte is smaller than for unsigned integers.

As we described above, sections share a common structure. Here's how that structure is defined in the spec:

> *Each section consists of*
>
> - *a one-byte section id,*
> - *the u32 size of the contents, in bytes,*

> - *the actual contents, whose structure is dependent on the section id.*
>
> **§ 5.5.2 Sections**

Let's define the `section` function:

```
                                    chapter01.js

function section(id, contents) {
  const sizeInBytes = contents.flat(Infinity).length;
  return [id, u32(sizeInBytes), contents];
}
```

- The `contents` argument is a possibly nested array of bytes, so we flatten it before getting the length.

For most sections, the contents are a (possibly empty) list of entries. In the binary format, these are known as *vectors*, or *vec* for short. Here's how they're defined in the spec:

> *Vectors are encoded with their u32 length followed by the encoding of their element sequence.*
>
> **§ 5.1.3 Vectors**

Let's implement a function for encoding vectors, again following the naming from the spec:

```
                                    chapter01.js

function vec(elements) {
  return [u32(elements.length), elements];
}
```

Let's introduce more functions by following the spec, first for the type section:

```
                                    chapter01.js

const SECTION_ID_TYPE = 1;

function functype(paramTypes, resultTypes) {
  return [0x60, vec(paramTypes), vec(resultTypes)];
}
```

```
function typesec(functypes) {
  return section(SECTION_ID_TYPE, vec(functypes));
}
```

- `typesec` is defined as a section with section id 1 that contains a vector of function types.
- A function type (`functype`) is defined as the byte `0x60` followed by a vector of parameter types and a vector of result types.

Defined in § 5.5.4 Type Section and § 5.3.3 Function Types, respectively.

Now the definitions for the function section:

```
chapter01.js

const SECTION_ID_FUNCTION = 3;

const typeidx = (x) => u32(x);

function funcsec(typeidxs) {
  return section(SECTION_ID_FUNCTION, vec(typeidxs));
}
```

- `funcsec` is defined as a section with section id 3 that contains a vector of type indices.
- A type index (`typeidx`) is simply a LEB128-encoded `u32` value.

Remember, the purpose of the function section is to declare all of the functions that are contained in the module. But, it's a very minimal declaration — it only specifies the type signature of the function (via an index into the type section).

Function declarations don't contain a name, because inside a module, functions are referred to by their index into the function section. The function bodies appear in the code section, in the same order as in the function section.

Defined in § 5.5.6 Function Section and § 5.5.1 Indices, respectively.

Here are the definitions for the code section:

```
chapter01.js

const SECTION_ID_CODE = 10;

function code(func) {
  const sizeInBytes = func.flat(Infinity).length;
```

```
    return [u32(sizeInBytes), func];
}

function func(locals, body) {
  return [vec(locals), body];
}

function codesec(codes) {
  return section(SECTION_ID_CODE, vec(codes));
}
```

- The code section (`codesec`) has the id 10 and contains a vector of code entries.
- A `code` entry consists of:
  - the size of the function code in bytes
  - the actual function code.
- A function's code (`func`) consists of:
  - the declaration of *locals* (local variables), as a vector. We'll define `locals` later, once it's actually required.
  - the function body, which is a sequence of instructions terminated by the `end` instruction.

Defined in [§ 5.5.13 Code Section.](#)

Hey, we've encountered our first instruction! As you might expect from the name, the `end` instruction doesn't do much — it just marks the end of *something*. It's always the last instruction in a function body. (You'll learn about some of its other uses later in the book.)

Each instruction is encoded by an *opcode*, which is a single byte identifying the instruction. Let's create a constant named `instr` (short for *instructions*) to map from instruction name to opcode:

chapter01.js

```
const instr = {
  end: 0x0b,
};
```

- The only instruction we need right now is `end` , but we'll be adding many more in future chapters.

With our new functions, we can write a Nop Lang compiler in a way that's both more readable and follows the spec more closely:

```js
function compileNopLang(source) {
  if (source !== '') {
    throw new Error(`Expected empty code, got: "${source}"`);
  }
  const mod = [
    magic(),
    version(),
    typesec([functype([], [])]),
    funcsec([typeidx(0)]),
    codesec([code(func([], [instr.end]))]),
  ];
  return Uint8Array.from(mod.flat(Infinity));
}
```

- We start with the preamble, followed by the type section containing one entry for a function with no arguments and no return values.

  Remember, this entry is a type signature that could be shared by multiple functions — it doesn't necessarily represent an individual function.

- Then comes the function section with a single entry, which means that this module has a single function and its type signature is found at index 0 in the type section.
- Finally, we have the code section with a single entry for a function with no local variables, and a body with only the `end` instruction.

Let's test our new implementation:

```js
test('compileNopLang compiles to a wasm module', async () => {
  const {instance, module} = await WebAssembly.instantiate(
    compileNopLang(''),
  );

  assert.strictEqual(instance instanceof WebAssembly.Instance, true);
  assert.strictEqual(module instanceof WebAssembly.Module, true);
});
```

We're almost there! But, we still have a WebAssembly module that does nothing. Even when a function is defined, it's not automatically exported, which means we can only call it from within the module.

It may feel like a lot of work to write all these functions, just to do something relatively simple. The good news is, we're almost done — just a few more things to define, and we won't have to worry about the module format for the next few chapters.

To make our function accessible from the outside, we'll add the function to the export section. But first, we need to define some export-specific functions.

```js
const SECTION_ID_EXPORT = 7;

function name(s) {
  return vec(stringToBytes(s));
}

function export_(nm, exportdesc) {
  return [name(nm), exportdesc];
}

function exportsec(exports) {
  return section(SECTION_ID_EXPORT, vec(exports));
}

const funcidx = (x) => u32(x);

const exportdesc = {
  func(idx) {
    return [0x00, funcidx(idx)];
  },
};
```

- The export section (`exportsec`) has the id 7 and contains a vector of *export* entries.
- An export entry (`export_`) consists of:
  - a *name*
  - an *export description*.

Unfortunately, we can't just name it `export` because that's a reserved keyword in JavaScript.

- A `name` is encoded as a vector of bytes containing the UTF-8 character sequence.
- Finally, an export description (`exportdesc`) consists of:

- one byte indicating the export type. For functions, it's `0x00` .
- the index of the element to export, encoded as a `u32` .

Phew! That's a lot of work to export a function, isn't it? Of course, the WebAssembly module format isn't really optimized to be written by hand. It's only those poor compiler writers who have to deal with it.

One last thing — to avoid having to specify the preamble each time, let's create a function called `module` that does the work for us:

```
chapter01.js

function module(sections) {
  return [magic(), version(), sections];
}
```

Alright, now we can update the `compileNopLang` function:

```
chapter01.js

function compileNopLang(source) {
  if (source !== '') {
    throw new Error(`Expected empty code, got: "${source}"`);
  }

  const mod = module([
    typesec([functype([], [])]),
    funcsec([typeidx(0)]),
    exportsec([export_('main', exportdesc.func(0))]),
    codesec([code(func([], [instr.end]))]),
  ]);
  return Uint8Array.from(mod.flat(Infinity));
}
```

▨ 02-noplang.js   ⬈ Open in StackBlitz

- We added the export section with a single export entry, exporting the function at index 0 under the name `main` . Note that the name `main` has no special meaning in WebAssembly; we could have named it anything.

Let's try it:

```
chapter01.js

test('compileNopLang result compiles to a wasm module', async () => {
  const {instance} = await WebAssembly.instantiate(compileNopLang(''));
```

```
    assert.strictEqual(instance.exports.main(), undefined);
    assert.throws(() => compileNopLang('42'));
});
```

- Since the function is now exported, we can call it from JavaScript.

Yay, it works! Now we have enough knowledge about WebAssembly to start creating a language that does something a bit more interesting.
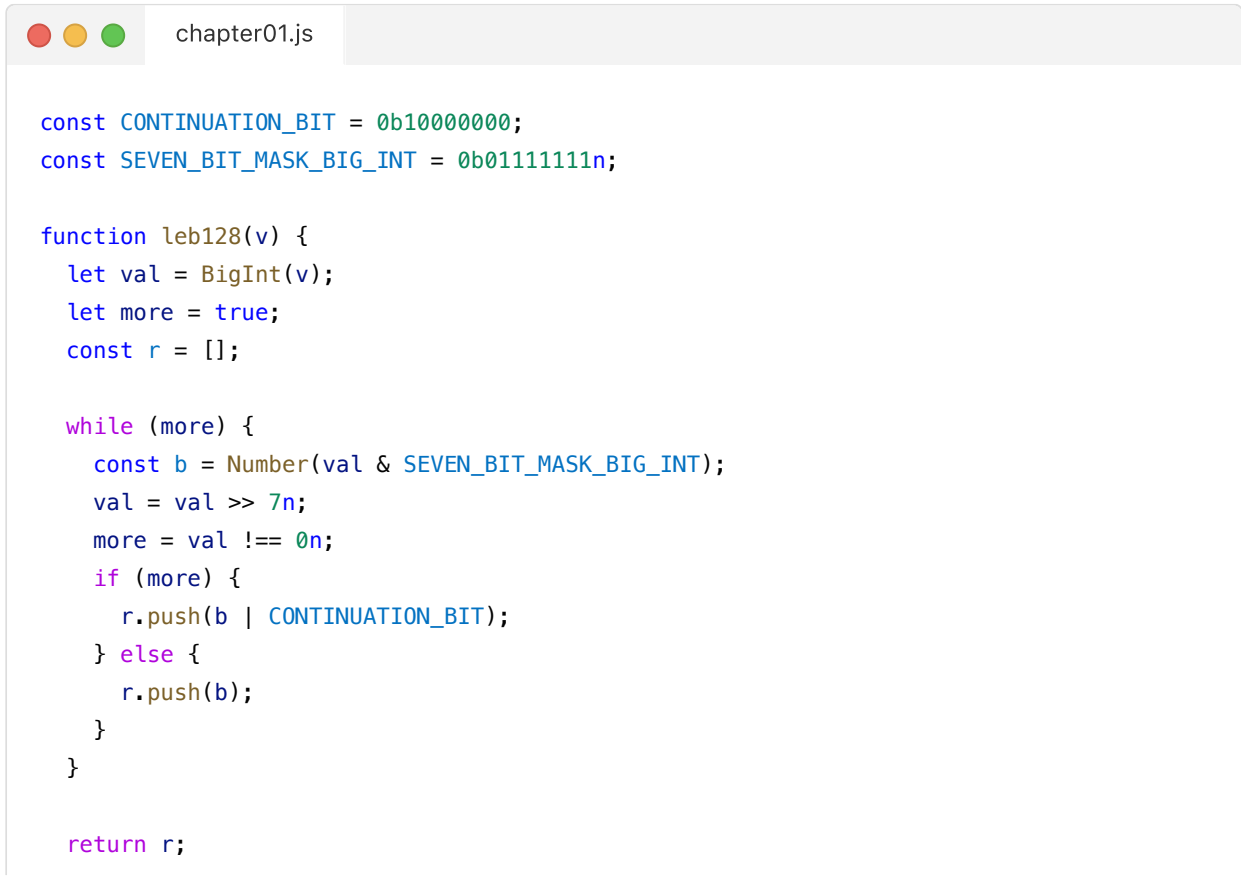
· · ·

Loose ends

Before we wrap up this chapter, we need to tie up a few loose ends.

Integer encoding

A complete implementation of integer encoding is found in the deep dive on Variable-Length Integer Encoding. Like the other deep dive chapters, it's entirely optional, and you're free to complete it at any time.

But many of the helper functions we've written in this chapter depend on `u32`, so before we go, let's replace the naïve implementation with a proper one:

chapter01.js

```js
const CONTINUATION_BIT = 0b10000000;
const SEVEN_BIT_MASK_BIG_INT = 0b01111111n;

function leb128(v) {
  let val = BigInt(v);
  let more = true;
  const r = [];

  while (more) {
    const b = Number(val & SEVEN_BIT_MASK_BIG_INT);
    val = val >> 7n;
    more = val !== 0n;
    if (more) {
      r.push(b | CONTINUATION_BIT);
    } else {
      r.push(b);
    }
  }

  return r;
```

```
}

const MIN_U32 = 0;
const MAX_U32 = 2 ** 32 - 1;

function u32(v) {
  if (v < MIN_U32 || v > MAX_U32) {
    throw Error(`Value out of range for u32: ${v}`);
  }

  return leb128(v);
}
```
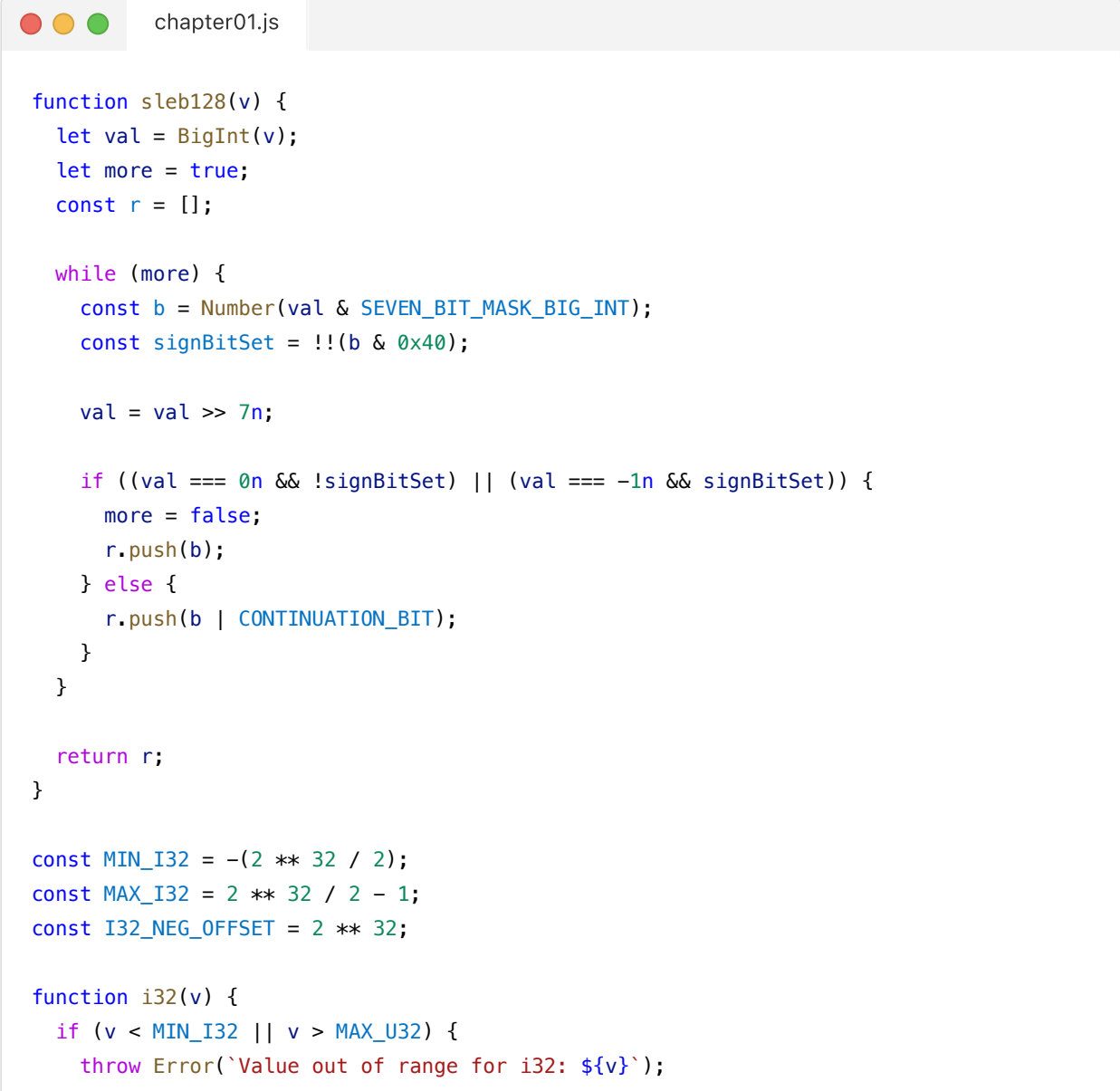
- Unlike all the other code in the book, we suggest you take this "as is" and don't worry about the details for now.

We'll do the same for `i32`:

```
chapter01.js

function sleb128(v) {
  let val = BigInt(v);
  let more = true;
  const r = [];

  while (more) {
    const b = Number(val & SEVEN_BIT_MASK_BIG_INT);
    const signBitSet = !!(b & 0x40);

    val = val >> 7n;

    if ((val === 0n && !signBitSet) || (val === -1n && signBitSet)) {
      more = false;
      r.push(b);
    } else {
      r.push(b | CONTINUATION_BIT);
    }
  }

  return r;
}

const MIN_I32 = -(2 ** 32 / 2);
const MAX_I32 = 2 ** 32 / 2 - 1;
const I32_NEG_OFFSET = 2 ** 32;

function i32(v) {
  if (v < MIN_I32 || v > MAX_U32) {
    throw Error(`Value out of range for i32: ${v}`);
```

```
  }

  if (v > MAX_I32) {
    return sleb128(v - I32_NEG_OFFSET);
  }

  return sleb128(v);
}
```

· · ·

With those cleanups out of the way, let's wrap up our work in this chapter. Nice job — you've learned the basics of the binary module format, and laid the groundwork for the compiler you'll be building out in the rest of the book.

Exports

In the next chapter, we'll be using many of the helpers that we defined above. To make that possible, we need to export them:

chapter01.js

```
export {
  code,
  codesec,
  export_,
  exportdesc,
  exportsec,
  func,
  funcidx,
  funcsec,
  functype,
  i32,
  instr,
  magic,
  makeTestFn,
  module,
  name,
  section,
  SECTION_ID_CODE,
  SECTION_ID_EXPORT,
  SECTION_ID_FUNCTION,
  SECTION_ID_TYPE,
  stringToBytes,
  typeidx,
```

```
    typesec,
    u32,
    vec,
    version,
};
```

chapter01.js

Open in StackBlitz