WebAssembly Developer Guide

[DRAFT]

By Eamon O'Tuathail
(mailto:eamon.otuathail@clipcode.com)

Last updated: February 22, 2018

www.clipcode.net

Clipcode provides training, mentoring and consulting for teams using WebAssembly

Table of Contents

1: Introduction

Overview

The WebAssembly specification is a definition of an instruction set architecture (ISA) for a virtual CPU that runs inside a host embedder. Initially, the most widely used embedder is the modern standard web browser (no plug-ins required). Other types of embedders can run on the server (e.g. inside standard Node.js v8 – no add-ons required) or in future there could be more specialist host embedders such as a cross-platform macro-engine inside desktop, mobile or IoT applications. Google, Mozilla, Microsoft and Apple have surprisingly cooperated to come up with an agreed representation of what a low-level executable module in WebAssembly binary format should be. All have agreed to implement it in their web browsers.

The WebAssembly text format is the assembly text language that needs to be converted to the WebAssembly binary format to control the virtual CPU. WebAssembly text format is the human (well, programmer) readable representation of a WebAssembly module. It is a low-level textual description of what the module should do. Any developer familiar with x86 or ARM assembly will be somewhat aware of what is happening, though the specific syntax is different.

Just like with projects that run on x86 and ARM processors, most application developers wishing to target WebAssembly will be coding using regular C and C++ or even higher level languages for much or all of their projects, but it is still very useful they be familiar with what is happening at the assembly level. Because WebAssembly is very new, there may be an even greater need for low level knowledge among app developers, as the tooling is not as mature as with other environments and it is more likely developers will need to at least view, and maybe program, assembly language. The “View Source” feature of major browsers will be showing this text format.

Some potential uses for WebAssembly include:

- Building compilers for IDEs that run directly in a browser (even disconnected)
- Building any type of low-level tools (especially developer tools)
- Dynamic code generation in the browser (e.g. from TypeScript)
- Using the same libraries and application code on the web server and web client
- Building the absolutely fastest code

Expanding on the final point, some application usage scenarios requiring the fastest possible code include:

- financial calculations (browser-based user workbenches for investment banking and similar involving heavy CPU calculations)
- image processing (transforming images by image region or by pixel)
- cryptography (implementing modern/specialist cryptographic algorithms that are not already provided by the browser)
- input validation (where there is a wish to execute the same code on the server and client)

Useful Links

The homepage for the WebAssembly project is at:

http://webassembly.org

Its docs tab has plenty of useful documentation, starting with:

http://webassembly.org/docs/high-level-goals/

The github collection of WebAssembly repositories is:

https://github.com/WebAssembly

If you are looking for small chunks of WebAssembly source demonstrating any particular concept, look here for a comprehensive test suite:

https://github.com/WebAssembly/spec/tree/master/test/core

An interesting paper is here:

https://github.com/WebAssembly/spec/tree/master/papers

The WebAssembly Specification (draft) is

https://webassembly.github.io/spec/

The Mozilla Developer Network (MDN) has good coverage of WebAssembly:

https://developer.mozilla.org/en-US/docs/WebAssembly

A WebAssembly Reference Manual is here:

https://github.com/sunfishcode/wasm-reference-manual/blob/master/WebAssembly.md

On Twitter, keep an eye on the WebAssembly hashtag as it gives a good overview of what is happening in the world of WebAssembly:

https://twitter.com/hashtag/webassembly?lang=en

Training, Mentoring and Consulting

Clipcode provides specialist WebAssembly training, mentoring and consulting. For details, see here:

http://www.clipcode.net

WebAssembly Concepts

There are a number of concepts developers new to WebAssembly should become familiar with.

MVP And Future Versions

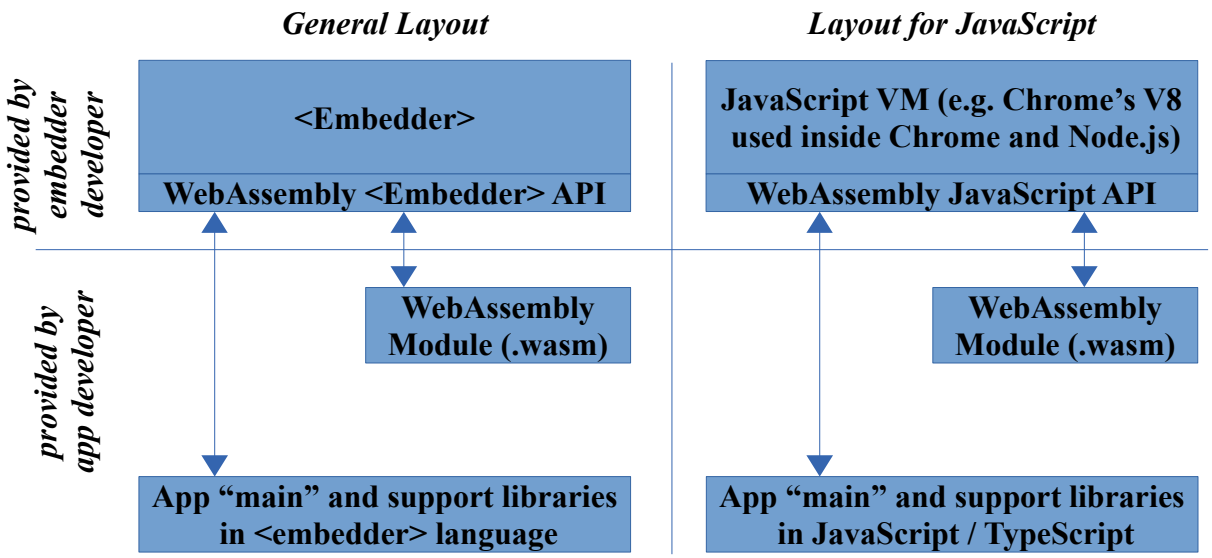
The WebAssembly Specification is being developed in an iterative manner. There was and still is no shortage of proposed ideas for what should be included in "the bytecode for the web". At some point a decision had to be made regarding what to include in the first edition and what to leave for future. The first agreed edition is known as the Minimum Viable Product (MVP). It has sufficient functionality to cater for certain workloads - specifically running hand-written assembly code, running C code and running C++ code are currently very well supported. What we discuss in this guide, what is currently supported by the shipping browsers and what is currently supported in Node v8 is this MVP version.

There many ideas for future additions and over time more are expected to be added in an evolutionary manner to the existing functionality in the MVP. Three groupings of much sought after functionality for future enhancements are:

- Threads, synchronization (atomics), SIMD, exceptions
- Allowing access to GC-managed references, thus opening up the possibility of greater integration between JavaScript and WebAssembly code and support for higher-level languages such as C# and Java
- Direct WebAssembly access to WebIDL-defined web platform APIs (e.g. the DOM, in-browser sandboxed file access)

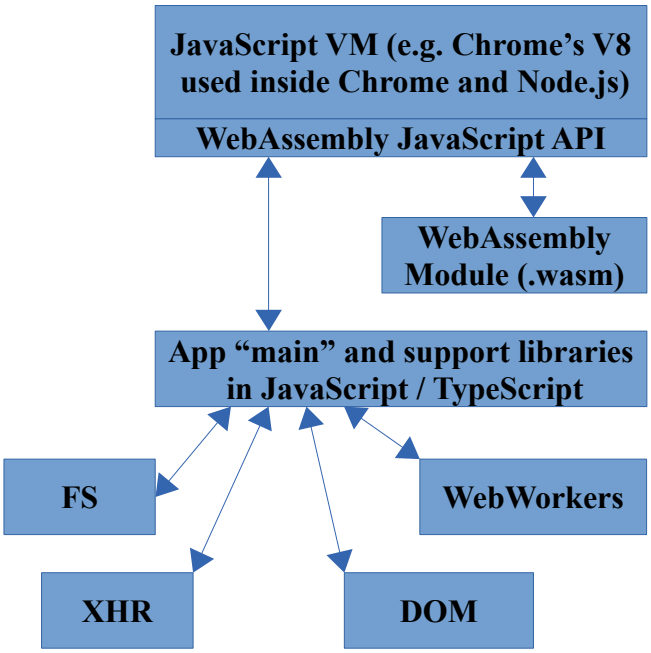
Host Embedder

WebAssembly modules executes in the context of a host embedder. The embedder is responsible for loading the module, initializing its imports, calling its exports when needed, handling traps and in general being the module's supervisor as it executes.



It is expected that for each type of embedder an API will be defined that specifies the interaction between that kind of embedder and a WebAssembly module. The most widely used embedder is for JavaScript (this is part of modern standard web browsers and part of Node v8). The WebAssembly JavaScript API describes the API between the JavaScript code and the WebAssembly module. In future other types of embedders may become available (e.g. we could imagine an embedder for C# desktop application) and each of these will require a separate API (e.g our imaginary .NET/C# embedder would have to provide a WebAssembly .NET API).

For most real world applications involving WebAssembly we expect application developers to have to write code in WebAssembly and in the language of the embedder. This is not mandatory – application developers might be able to get away with only writing Webssembly code, but from our experience most will also want to influence what happens inside the embedder. If you are an application developer writitng code to run in the web browser, it is likely you will want to interact with the DOM. In the WebAssembly MVP this can only be done from the JavaScript Embedder. Hence you are going to have to write some JavaScript/TypeScript code to work with the DOM or use a third party library that does this for you. The same applies to all the other "added-value" APIs that are available via JavaScript in a brwoser's execution context.



Though we realize the embedder has an important role to play in the bigger picture when executing WebAssembly code, in this guide we are going to focus on WebAssembly text format and so will use an open source toolkit known as the WebAssembly Binary Toolkit (WABT) to work with WebAssembly modules. To execute WebAssembly code we will use WABT's wasm-interp interpreter and we will not be using any embedder for now.

32-bit vs. 64-Bit Wasm

Currently WebAssembly is 32-bit only. In future 64-bit will be supported. The reasoning behind this design decision is that hardly any code that is executed to run in WebAssembly will need more than 4GB of memory, and so 32-bits of memory addressing information is sufficient. As the hosting scenarios for WebAssembly become more ambitious and as larger applications become more prevalent, then 64-bit WebAssembly will also become important.

Hierarchy Of Programming Languages

Programming languages may be arranged in a hierarchy:

- Scripting languages (JavaScript, TypeScript)
- Runtime-based languages (C#, Java, Swift)
- Intermediate Languages (IL, bytecode, Swift's SIL, LLVM IR)
- low-level languages (C, C++)
- Assembly language

Polyglot software developers are those who know more than one language. When recruiting a development team it is highly desirable for to recruit developers who know multiple languages from different levels, as they develop a keen sense of what happens at the different levels are what the key trade-offs are when selecting a language for a particular workload.

WebAssembly is obviously at the lowest level of this hierarchy. In the MVP low-level languages such as C and C++ can be compiled into WebAssembly.

In future updates to the WebAssembly spec, it is expected additional capabilities will be added so higher-level languages may run on a WebAssembly substrate.

Naming

The project is called WebAssembly – with capital W, capital A with no space between the two words. This means a Google search will likely match material you want (in the recent past, searching for "web assembly" (with a space) would give back lots of not so relevant topics).

Wasm module vs. compiled module

WebAssembly modules are somewhat similar in concept to ECMAScript modules. One obvious difference is the underlying representation. WebAssembly modules are binary whereas ECMAScript modules are either textual (even if minified) or compressed textual. This makes the WebAssembly module much faster to load, especially for larger modules. Another difference is that WebAssembly modules require a local compilation step, when the embedder (e.g. the web browser engine) is presented with a chance to do internal processing of the module representation. What happens during such compilation is very much browser-specific. Different browsers are implemented in vastly different ways.

When discussing modules, we use the term "module" in general, and when we wish to specifically distinguish between the bytes delivered to the embedder and the compiled representation, we use the terms "wasm module" and "compiled module". How the

first can be transformed into the second depends on the embedder. For JavaScript-based embedders, the WebAssembly JavaScript API `compile` or `instantiate` functions are what is needed. We will discuss this in detail later.

It is the job of application developers to make great web apps, it is the job of browser developers to make a great browser and the contract between each set of developers is the WebAssembly Specification and test suite.

S-expressions

WebAssembly text format uses s-expressions (symbolic expressions).

S-expressions are a very simple way of representing a tree in text format. It is a tree, not a graph (remember, every tree is a graph, but not every graph is a tree). There are a number of variations of s-expressions, what we introduce here is how WebAssembly uses them.

A tree consists of a hierarchy of nodes with one node at the root which has no parent, and all other nodes have a parent. Each node starts with an open brace and at least a node type and finishes with a close brace. A node may optionally have both or either attributes and/or sub-nodes. So our simplest s-expression is:

```
(rootnode)
```

This is the root node which in this sample we happen to state its type is rootnode. It is up to the language spec writer to specify what the valid node types are. If we were writing the language spec, we could have picked any name for the node types.

A node can contain attributes:

```
(rootnode attr0 attr1=80)
```

Here we see rootnode containing two attributes, the second of which has a value. Again it is up to the language spec writer to specify what are the valid attributes for each node type and whether they can be assigned values.

A node can contain child nodes:

```
(rootnode
  (subnode1 (subnode2))
  (subnode2)
)
```

Here we see a node of type rootnode directly containing two sub-nodes, the first of which has its own child node. It is noted the same node type (e.g. subnode2 above) can be used multiple times in a s-expression. Line breaks are not important, so we could also have written the above as:

```
( rootnode (subnode1 (subnode2)) (subnode2) )
```

Another alternative is to write the tree in some other format, such as XML (which is not used in WebAssembly – we just mention it here as an example alternative tree representation), but we see it is much more verbose (the knowledge density is lower).

```
<rootnode>
<subnode1><subnode2 /></subnode1>
```

```
<subnode2>
</rootnode>
```

While on the topic of XML, you might be wondering what the equivalent of XML Schemas (XSD) are for S-expressions. The answer is the language specification. In our case, we are interested in the WebAssembly specification so that is where we go to learn about the permitted nodes and attributes.

A node can contain both sub-nodes and attributes:

```
(rootnode
  (subnode1 (subnode1.1 attr0) attr1 attr2)
  (subnode2) attr3
)
```

Here we see a subnode of type subnode1 containing two attributes, attr1 and attr2; a subnode of type subnode1.1 containing a single attribute attr0 and a subnode of type rootnode containing an attribute of type attr3. We are just just these names for ease of identification; when designing our own language we could just as easily write this as:

```
(alpha
  (beta (foo baz) delta omega)
  (gamma) theta
)
```

Child nodes can be indexed by order of appearance in the parent. Nodes can also be named using $<name>, thus allowing them to be more easily identified from elsewhere in the tree:

```
(rootnode
  (subnode1 $name1)
  (subnode2)
)
```

In this example subnode1 can be indexed as 0 into children of its parent and subnode 2 can be indexed as 1. In addition, subnode1 is named as $name1. Naming of nodes is a good idea when they need to be referred to from elsewhere as inevitably over time additions and deletions will be made to the content of a program and hence the index values will change and will need to be updated, whereas the naming will not.

.File Formats - .wasm, wat and .wast

The file extension for the WebAssembly binary format is .wasm. You can of course use any extension you want, but pretty much everyone uses .wasm and you should too.

There are two file extension names used for the WebAssembly text format - .wat and .wast. The latter is an extension to the former. Most production assembly code will be written in .wat files. Additional test scripts and spec assertions (hence the ‘s’ in the name) and assertions can be added and such files have the .wast suffix. Note that the WABT tool to convert from either .wat/.wast to .wasm binary format is called wast2wasm (there is no wat2wasm tool).

One .wat text file gets converted to one .wasm binary module.

When an error occurs – Traps

When executing WebAssembly code detects an error (e.g. divide by zero), it throws a trap, which is not catch-able by the WebAssembly module code, instead it bubbles up and is presented to the embedder (e..g as a JavaScript exception if using the web browser embedder). If WebAssembly code wishes to manually cause a trap, it should use the `unreachable` WebAssembly instruction.

Line Comments using ;; and Block Comments using (; ..;)

There are two kinds of comments in WebAssembly text – line comments and block comments. Line comment use the `;;` symbol. Characters after this symbol until the end of a line are ignored as comments. Block comments appear as s-expression subnodes. Block comments start with `(;` and conclude with `;)` and any characters in between are considered to be comments.

Note C-style single line comments (`//`) and multi-line comments (`/* … */`) are not supported in WebAssembly but the aforementioned WebAssembly line comments and block comments provide the same functionality.

(Module) – The rootnode of a WebAssembly Module

Now that we have reviewed background topics, it is time to start looking at WebAssembly programming.

In WebAssembly a unit of execution is known as a module. Currently a module is similar to a shared library (*.so) or dynamic link library (*.DLL) in mainstream applications. The reason we state that rather than saying "a module is similar to an executable (*.exe)" is that in the MVP a WebAssembly module cannot directly make external calls (e.g. to edit the DOM in a web page, or to open a file on the local hard disk in a server app). Instead, it relies on an embedder (e.g. running inside the JavaScript VM in a web browser or Node.js) to make such calls for it. Hence is it best to think of the code that runs in the embedder (the JavaScript or TypeScript code) as conceptually being the main "executable" and what runs in WebAssembly as being a dynamically loaded shared library. In future editions of WebAssembly, it is expected that its modules will be able to directly call WebIDL-based APIs such as the DOM, so this will change, but for the MVP, the above is the case.

A WebAssembly module can export and import functions and data (memory and globals). The embedder and the executing WebAssembly module instance can interact by making functions calls into each other or sharing data via these exports and imports. So for example if one wishes to allow a WebAssembly module to add something to the DOM in a web browsing context, the WebAssembly module needs to import a function from the embedder and call it with the right parameters; then JavaScript/TypeScript code running in the embedder must contain an implementation of this function to make the right DOM calls and must use the WebAssembly JavaScript API to initialize this function identifier when instantiating the WebAssembly module.

For the MVP of the spec, there is no command line interface to the executing WebAssembly module – because there is no main:

```
int main(int args, char **argc) { .. } // does not exist in WebAssembly
```

As we said, it is best to think of a WebAssembly module as a library rather than the main application. C/C++ compilers that have a WebAssembly backend and wish to implement main need to generate some helper HTML/JavaScript when building a C/C++ project with main, and this helper code will ensure that e.g. a `printf` call in the WebAssembly code will result in a DOM call in JavaScript to add some text to a output window somewhere in the HTML page.

What is inside a WebAssembly Module?

A module contains:

- zero, one or more functions
- zero, one or more globals
- zero or one memory
- zero or one table

The maximum limitation of 1 for tables and memories may change in future, but for now in the MVP we have to work with this. Each of these may be exported or imported, how this is done will be explored in a later chapter.

Functions are the only construct where code executes. A function contains:

- zero, one or more input parameter values,
- zero, one or more local values,
- In the current spec, zero or one return value and in a future spec update, zero, one or more return values
- zero, one or more instructions

Each value has a value type, which is one of the following:

- 32-bit integer
- 32-bit float
- 64-bit integer
- 64-bit float

Globals are values whose lifetime is the lifetime of the module instance (in contrast to local values within a function, whose lifetime is the lifetime of the function call). Global values have one of the above value types. Globals can be mutable or immutable.

Memories (also called linear memories) are like heap memory. The are array buffers and it is up to the module author to decide what is placed where inside such buffers.

Tables are for function pointers and are disjoint with memories. So for C-like function pointers you would populate a slot in a table and later other code could call it. The idea behind WebAssembly tables is as a security precaution - to disallow code from changing the destination of function pointers using normal memory writes.

What is not in WebAssembly

It surprises developers new to WebAssembly that its type system is so sparse. It has a mere four types – 32-bit integers and floats along with 64-bit integers and floats. No string types? No date type? No smaller integer types (e.g. 16-bit)? There is a design

goal to keep the WebAssembly spec as succinct as possible (that is good) and everything we need can be built up from the four supported value types and their use with memory.

There is no struct or similar structure-like mechanism in WebAssembly. Implementors of a C compiler targeting WebAssembly will have a bit of work to do to implement structs. If we are manually writing WebAssembly, then we also have a little bit extra work here.

There are no object-oriented programming constructs in WebAssembly – this means no classes (and so no constructors and no inheritance), no generics, no namespacing, no properties, no events and no exceptions. All these useful mechanisms can be built on top of the capabilities that are supplied with WebAssembly. So far example, if as WebAssembly developers we wish to create the idea of class and calling a method, we could store the data for the class instance somewhere in memory, and we could pass in a integer called this as the first parameter to a function that represents the method (this is how most modern C/C++ compilers work today for most programming targets – not just WebAssembly backends). If we wish to implement a getter for a property this is really nothing more that a function, which WebAssembly does support.

There is no `#include` feature in WebAssembly text format (so no way to include the contents of one assembly file in another). hence a single assembly text file needs to represent everything that will become part of a single wasm module – in WebAssembly binary format.

In the MVP, WebAssembly modules do not have access to WebIDL-defined APIs, such as the DOM or file access.

In the MVP, WebAssembly modules cannot directly create new worker threads or post messages to existing worker threads. Note the word “directly“ in there.

2: Tooling & Getting Started

Overview

To get started exploring WebAssembly text format and see the results in action, some developer tooling is needed. The open source WebAssembly Binary Toolkit (WABT) is the set of command-line tools we need. WABT is pronounced as "wabbit", to rhythm with "rabbit"! Conceptually WABT is to WebAssembly modules as GNU Binutils is to binaries for Linux (and similar OSes).

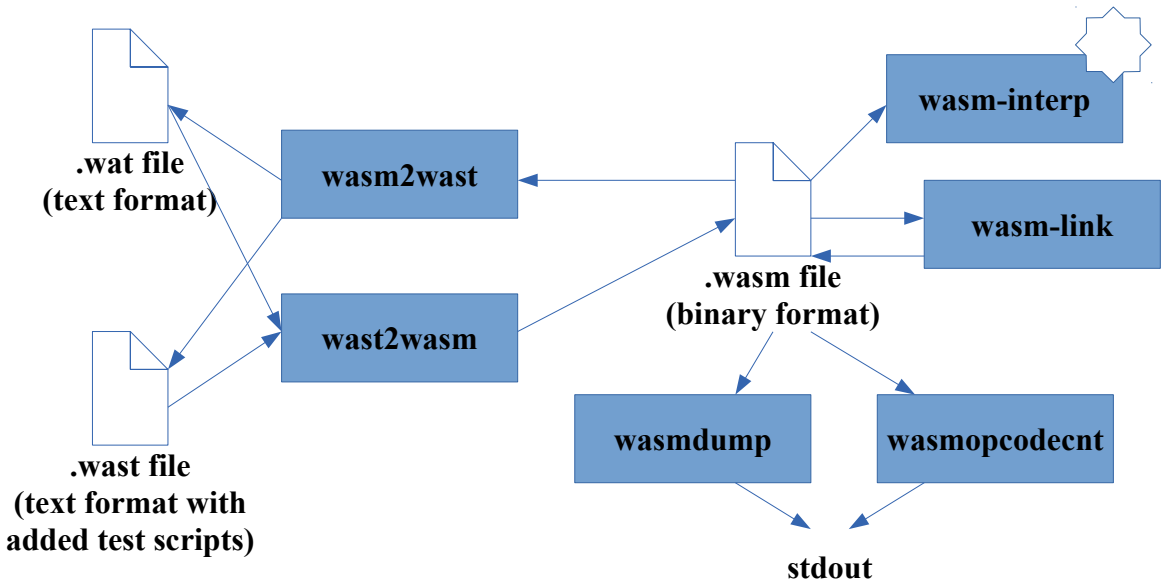
WABT Tools

WABT has functionality:

- to work with WebAssembly in text format and binary format
- to convert between these formats
- to run a validation suite
- to run a lightweight interpreter

The tools in WABT are:

- wast2wasm (an assembler)
- wasm2wast (a dissassembler)
- wasmdump (dump utility)
- wasm-interp (an interpreter)
- wasm-link (merges many .wasm files to one)
- wasmopcodecnt (counts opcode usage)
- wast-desugar (syntax cleaner)



There is also a shared library used by all of the above and also available for you to use in your own custom tools:

- wabt.lib

The bulk of the WABT functionaltiy is actually inside this library and so can be used by your custom tools.

The executables provided with WABT are really just command-line wrappers around this library. The executables accept command line arguments, validate they are correctly formatted, make calls into this library and generate appropriate output.

Building WABT

WABT is available in source code only. It can be downloaded from:

- https://github.com/WebAssembly/wabt

Its build system is based on the popular CMake and simple instructions on how to build is available on the above page.

WABT As Part Of A Bigger Picture

Custom tools

embedders

higher-level compiler toolchains (e.g. C / C++)

WABT Usage

When we wish to convert from .wat text format to .wasm binary format so we use this command syntax:

```
wast2wasm <module name>.wat -o <module-name>.wasm
```

If you do not include the -o output option, then by default no output file is generated (the .wat file is validated only). Validation means checks are made to ensure as much as possible that what is represented in the wasm file complies with the WebAssembly specification. Validation always occurs with wast2wasm, it is only the subsequent generation of the wasm binary module that required the -o switch.

We also wish to run the wasm module. Often this will run in the context of an embedder (such as a web browser or on a web server – e.g. Node.js v8) and there may need to be some interaction between the embedder and the WebAssembly module (e.g. using the WebAssembly JavaScript API). However, for now we wish to solely focus on the WebAssembly text format and get our code to run, so we will use a simple interpreter included in WABT , using this command:

```
wasm-interp <module-name>.wasm --trace
```

This will load the wasm module in an interpreter and execute the start function, if any. We will be examining the role of the start function in our chapter on WebAssembly functions – for now all we need to know is that it is automatically called when a WebAssembly module is loaded. For most of our demos in this guide we will have a start function, so the above command line is all we need. The --trace switch (notice two – characters) traces the execution of instructions, which shows us our code as it runs, which is a great help with the learning process.

wasm-interp also allows you to run all exported functions, using this command line:

```
wasm-interp <module-name>.wasm –trace –run-all-exports
```

We will be looking at exports and imports in a later chapter.

WABT deals with the WebAssembly text and binary formats at a low-level. Obviously a sophisticated <C/C++/other> compiler toolchain can perform many optimizations and offers additional capabilities, so for many application developers additional tooling will be needed. Such tooling could write the binary wasm module file directly without creating a .wat text file but it is expected that many will offer via a command-line argument the option of generating the .wat file (and if they don’t, then WABT has a disassembler, called wasm2wast).

Using wasmdump

using wasmopcodecnt

using wasm-link

Wasmlink is the only WABT tool where the format of the input file(s), of which tere may be many) and the format of the output file are all binary format.

Simple WAT Programming

The simplest module is:

```
(module)    ;; tools01.wat
```

The module itself is neither named nor versioned within the .wat file. In all our demos we add in the file name (here "tools01.wat") as a comment in the first line – this convention can of course be omitted when you write this text to a file.

To compile into wasm binary run this from the command line:

```
wast2wasm tools01.wat -o tools01.wasm
```

Execute it with:

```
wasm-interp tools01.wasm
```

Note that wast2wasm does not have a hyphen in the name, unlike wasm-interp. Also note that wast2wasm starts with wast, not wat. It works with both these types of text formats.

tools01.wasm does not do anything, so running it in the wasm interpreter will have no result. We need to add a function, and we need to export the function so it can be called externally (e.g. by wasm-interp, or by JavaScript/TypeScript code from a web page):

```
(module            ;; tools02.wat
  (func $my_main)
  (export "my_main" (func $my_main ) )
)
```

A notes on quotes – that main string in quotes moving between a document editor and a code editors – make sure you are using normal quotes. In this simple sample the my_main function has no instructions, so after the name of the function (which is optional) comes the close brace. Normally there will be a list of instructions before the closing brace.

We now compile and run with the following commands:

```
wast2wasm tools02.wat -o tools02.wasm
wasm-interp tools02.wasm –trace --run-all-exports
```

We have added two switches to the wasm-interp command line, one to enable verbose tracing and the other to execute exported functions. The trace output is:

```
>>> running export "my_exported_main":
#0.    0: V:0  | return
my_exported_main() =>
```

tools02.wat uses `$main` twice, and we can combine them, as follows:

```
(module (func (export "my_main") ) )      ;; tools03.wat
```

We will be looking at exports and imports in detail in a later chapter, so for now we need another way of beginning to execute our code in wasm-interp.

A single function may be designated as the start function, and this is automatically

called when a module is loaded:

```
(module (start $module_start) ;; tools04.wat
  (func $module_start
    ;; do something
  )
)
```

We will be using this in most of our demos.

Instructions – s-expression nodes, flat or mixed?

Up to now we have used a linear list of instructions in our .wat files. However, there is an alternative representation, which allows instructions as s-expression nodes, and nodes can be sub-nodes of other instructions.

If you wish one or more instructions to be called before another instruction, made the one or more instructions as sub-nodes. This is best shown via a demo – let's use instructions in sub-nodes:

```
(module (start $module_start) ;; tools05.wat

  (global $my_first_mutable_global (mut i32) (i32.const 1) )
  (global $my_first_immutable_global i32  (i32.const 2) )

  (func $module_start
    call $global_demo
  )

  (func $global_demo
    ;; get a global and use result to set an mutable global
   (set_global $my_first_mutable_global
                    (get_global $my_first_immutable_global))

    ;; set an mutable global using a constant
   (set_global $my_first_mutable_global (i32.const 10))
  )
)
```

It is a matter of choice which to use. When compiled to a .wasm file, the same result is generated.

WABT Command Lines

Three tools - wasm-link, wast-desugar and wasm-interp - have a hyphen in the name The other tools do not. Sometimes if you can’t seem to find the right tool from the command line, check you have spelled it correctly.

Wast2wasm

wasm2wast

wasm-interp

wasm-dump

wasm-link

wasm-opcodecnt

We’ll examine wasm-desugar a little later as we first need to cover how WebAssembly instructions can be represented.

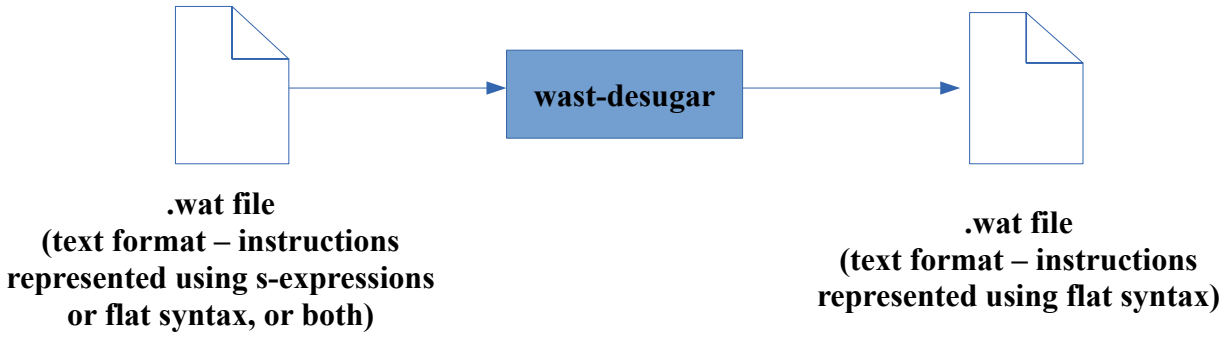
A note on linking

When developers new to WebAssembly see the name wasm-link they understandably assume, based on their previous experience with the likes of C or C++, some sort of linking phase has to occur with wasm files. That is incorrect. It might have been better off to call this tool wasm-merge, because that is actually what it does.

The wasm-link tool simply takes in two or more wasm files and merges their content.

With WebAssembly, there is no "linking" phase needed in the traditional use of that term. The nearest thing to linking is the definition of exports and imports and the population of tables (used for function pointers), all of which we will discuss in detail in later chapters.

Languages that compile into WebAssembly (e.g. C and C++) may well have their own linking phase in their toolchains, but that occurs before wasm binary file generation.



Wast-desugar is the only WABT tool where the format of the input file and the format of the output file are both text format.

3: Values

Overview

WebAssembly supports just two integer value types – i32 and i64. The reasoning behind this decision is simplicity in exchange for no loss of speed. Mathematical computation on modern physical CPUs work with 32-bit just as quickly as 16-bit or 8-bit so one is not saving any time regardless of which <64-bit integral data type to use. By omitting direct support for 16-bit and 8-bit integers, WebAssembly does not need to support instructions specifically for these value types so it reduces the number of instructions. So what happens if you have 8-bits in memory somewhere (e.g. came in from an image format that you loaded) and you need to process this 8-bit integer? WebAssembly has it covered – when loading from memory onto the value, stack, WebAssembly has load instructions that allow you to load 1/2/4 bytes and either sign-extend or zero-extend the remaining bytes to produce a 32-bit or 64-bit value in the value stack.

Stack vs. Registers for Instruction Sets

There are essentially two competing ways of designing data layout for instruction set architectures – a stack and registers. With the stack approach, instructions add and remove (push and pop) values to/from the stack (so an add instruction would need to be preceded by two instructions that pushed values onto the stack, the add instruction itself would pop the two two values from the stack, add them, and push the result back onto the stack). With the register approach, instructions can operate on named registers – so a single add instruction would need to identify two registers with the values to add, and a third register where the result is to be placed. There is ongoing debate among CPU and instruction set designers as to which approach is best. On average, with the register approach, each instruction is bigger (need to specify opcode and what registers it works with for inputs and outputs) but also fewer instructions are needed, since so much is encoded into each instruction. An instruction set architecture needs to pick one approach and it is a fundamental design decision.

WebAssembly, Java bytecode and .NET IL all use the stack approach. X86 and ARM use the register approach. More recently it seems that the stack approach slightly has the edge.

A WebAssembly implementation is free to choose how exactly it implements the stack. It may exist as an actual stack data structure or it may be synthesized by some other means. A WebAssembly implementation need only ensure it executes wasm modules according to the specification. How it does it is a internal matter (and very likely to substantially change over time as optimizations are implemented in the rapidly evolving virtual machines inside web browsers).

There are actually two stacks in WebAssembly – the value stack (for values) and the call stack (for function calls). Apart from calling functions and returning from functions, WebAssembly application code has no way of interacting with the call stack,

so in general when we talk about "the stack" in WebAssembly, we are referring to the value stack, which plays a more prominent role and which have instructions that directly affect it.

Each function call has its own value stack, which starts out empty. When a function is called with the agreed parameters, these are first placed on the value stack, followed by locals and instructions within the function can further manipulate the value stack by pushing and popping values. Once the function returns, the agreed result value must be the only thing left on the stack. This is then placed on the value stack at the call site (i.e. the calling function).

Instructions and Opcodes

Each instruction starts with an opcode and may have additional parameters. Sample opcodes are add (to add two values), gt (to determine if one value is greater than another), call (to call a function), return (to return from a function), xor (to XOR two values).

The WebAssembly spec has a description of the expected state of the value stack before and after the instruction execution and whether the instructions has parameters in addition to the opcode. For example, an add instruction takes the add opcode and has no parameters; it is processed by popping two values from the stack, adding them, and pushing the result back onto the stack.

If you were writing an interpreter to execute a WebAssembly module, there would be a large switch statement somewhere which maps the opcode to a small block of code that performs the requested operation.

Machine Code, opcodes, microcode, micro-ops

Constants

The simplest instructions relate to constants – i.e. put a constant (also called an immediate) value on the value stack. The const instruction is called const, is prefixed with a value type, and takes a single parameter. When it executes, it places that parameter value onto the value stack. So this code should put 1 on the stack:

```
(module (start $module_start) ;; val01.wat
  (func $module_start
    i32.const 1
  )
)
```

we say "should" and it actually does, but when we compile we get this error:

```
val01.wat:3:5: type stack at end of function is 1, expected 0
    i32.const 1
    ^^^^^^^^^^
```

We will see this error from time to time so it is good to understand what it is about and how to fix it. As we have seen, each function has its own value stack and at the end of the function it must be empty. In the initial version of WebAssembly a function

may have zero or one return values; in future, multiple return values are expected to be supported. In val01.wat we have not stated that the module_start function has a return value, so it defaults to none. Specifically for the start function, this must always be the case. For other functions, you could add a return value to the function signature. We are using the i32.const instruction which puts an integer value on the stack, and this is the cause of the problem at the end of the function call. The solution is to remove that value from the stack, using either the drop instruction or the return instruction.

The drop instruction simply deletes the top of stack and has no other effect.

```
(module (start $module_start) ;; val02.wat
  (func $module_start
    i32.const 1
    drop
  )
)
```

Running the generated val02.wasm file through wasmdump with the -d(isassemble) option gives us:

```
wasmdump val02.wasm -d
val02.wasm:    file format wasm 0x000001
Code Disassembly:
000018 func[0]:
 00001a: 41 01                      | i32.const 0x1
 00001c: 1a                         | drop
 00001d: 0b                         | end
```

We note it has no return instruction. (We also note wast2wasm has added an end instruction we did not have in our .wat file – this is used here to signify the end of the function, but also has other uses, as we will explore later).

When running the code in wasm-interp:

```
wasm-interp val02.wasm --trace
```

we now get

```
>>> running start function:
#0.    0: V:0  | i32.const $1
#0.    5: V:1  | drop
#0.    6: V:0  | return
```

Note we did not have a return instruction at the end of our function in val02.wat, which wasm-interp automatically added for us if not provided.

Like in a C function or C++ method, we may well wish to place the return instruction different locations within the instruction stream in a function – e.g. checking an input parameter value at the beginning of the function and returning early if an error is detected. So we can manually add return instruction. The following code has both the drop instruction and the return instruction and has the same trace output as val02.wat:

```
(module (start $module_start) ;; val03.wat
  (func $module_start
    i32.const 1
    drop
```

```
    return
  )
)
```

The wasm module now have both drop and return instructions:

```
wasmdump val03.wasm -d
val03.wasm:    file format wasm 0x000001
Code Disassembly:
000018 func[0]:
 00001a: 41 01                      | i32.const 0x1
 00001c: 1a                         | drop
 00001d: 0f                         | return
 00001e: 0b                         | end
```

When we run this module in wasm-interp we get:

```
>>> running start function:
#0.    0: V:0  | i32.const $1
#0.    5: V:1  | drop
#0.    6: V:0  | return
```

As does this demo (note the omission of the drop instruction):

```
(module (start $module_start) ;; val04.wat
  (func $module_start
    i32.const 1
    return
  )
)
```

The output of wasmdump for val04 is:

```
wasmdump val04.wasm -d
val04.wasm:    file format wasm 0x000001
Code Disassembly:
000018 func[0]:
 00001a: 41 01                      | i32.const 0x1
 00001c: 0f                         | return
 00001d: 0b                         | end
```

When executed, it gives the same as val03.wat:

```
>>> running start function:
#0.    0: V:0  | i32.const $1
#0.    5: V:1  | drop
#0.    6: V:0  | return
```

What is happening is that the wasm-interp interpreter when it detects a return instruction is smart enough to automatically drop anything that remains on the value stack, and hence manually inserting a drop is not required at the end of a function (it can be highly useful elsewhere in the function).

To wrap up our look at constants we can use all four in a demo:

```
(module (start $module_start) ;; val05.wat
  (func $module_start
    i32.const 1
    i64.const 2
    f32.const 1.4
```

```
    f64.const 2.8
   drop
   drop
   drop
   drop
   )
)
```

The trace output is:

```
>>> running start function:
#0.    0: V:0  | i32.const $1
#0.    5: V:1  | i64.const $2
#0.   14: V:2  | f32.const $1
#0.   19: V:3  | f64.const $2
#0.   28: V:4  | drop_keep $4 $0
#0.   34: V:0  | return
```

Since we have put four values on the value stack, at the end a drop_keep command has been inserted by wasm-interp to drop them from the stack. drop_keep is not a normal WebAssembly instruction, so you cannot manually add it to a function. If you wished to, you could add the drop instruction four times to manually clear the stack before the end of the function (manually adding return not needed):

```
(module (start $module_start) ;; val06.wat
  (func $module_start
    i32.const 1
    i64.const 2
    f32.const 1.4
    f64.const 2.8
  drop
  drop
  drop
  drop
  )
)
```

Its trace output is:

```
>>> running start function:
#0.    0: V:0  | i32.const $1
#0.    5: V:1  | i64.const $2
#0.   14: V:2  | f32.const $1.4
#0.   19: V:3  | f64.const $2.8
#0.   28: V:4  | drop
#0.   29: V:3  | drop
#0.   30: V:2  | drop
#0.   31: V:1  | drop
#0.   32: V:0  | return
```

You could also just drop some of the values, and depend on return to remove the remainder:

```
(module (start $module_start) ;; val07.wat
  (func $module_start
    i32.const 1
    i64.const 2
    f32.const 1.4
    f64.const 2.8
  drop
  drop
```

```
    return
   )
)
```

Its wasmdump output is:

```
wasmdump val07.wasm -d
val07.wasm:    file format wasm 0x000001
Code Disassembly:
000018 func[0]:
 00001a: 41 01                       | i32.const 0x1
 00001c: 42 02                       | i64.const 2
 00001e: 43 33 33 b3 3f              | f32.const 0x1.666666p+0
 000023: 44 66 66 66 66 66 66 06 40  | f64.const 0x1.6666666666666p+1
 00002c: 1a                          | drop
 00002d: 1a                          | drop
 00002e: 0f                          | return
 00002f: 0b                          | end
```

Its trace output is:

```
>>> running start function:
#0.    0: V:0  | i32.const $1
#0.    5: V:1  | i64.const $2
#0.   14: V:2  | f32.const $1.4
#0.   19: V:3  | f64.const $2.8
#0.   28: V:4  | drop
#0.   29: V:3  | drop
#0.   30: V:2  | drop_keep $2 $0
#0.   36: V:0  | return
```

We have two drops in val07.wat and so wasm-interp adds a call to drop_keep $2 $0 to clear the stack before return.

The curious reader will be wondering (and the curious writer too!!), what happens if a single value remains on the stack, so let's see:

```
 module (start $module_start) ;; val08.wat
  (func $module_start
    i32.const 1
    i64.const 2
    f32.const 1.4
    f64.const 2.8
   drop
   drop
   drop
   return
  )
)
```

Here we add four values to the stack and remove three and then call return. The trace result is:

```
>>> running start function:
#0.    0: V:0  | i32.const $1
#0.    5: V:1  | i64.const $2
#0.   14: V:2  | f32.const $1.4
#0.   19: V:3  | f64.const $2.8
#0.   28: V:4  | drop
#0.   29: V:3  | drop
#0.   30: V:2  | drop
```

```
#0.    31: V:1   | drop
#0.    32: V:0   | return
```

So when return is called if there are more that one value on the value stack a drop_keep command is added, whereas if there is only one, a normal drop instruction is added. There is a argument saying your code should clean up after itself and at the end of a function the value stack should be empty, so inserting return just to empty it is unnecessary. Sometimes this is right, but sometimes it is plain useful to put in a return to do the cleanup for us.

Globals

Globals are module-wide storage containers for individual values. Their type must be one of the four supported value types – i32, i64, f32, f64.

Locals and parameters are always internal to a module. In contrast , a global can be exported, and exported globals from other modules (and JavaScript) can be imported. We will look at import and export in a later chapter. Also in contrast to locals which are initialized to 0, and parameters which are initialized to the values set the the call site, globals can have initializers attached to them. Also in contrast to locals and parameters, globals can be declared as mutable or immutable.

The initial value of a global is set by an initializer expression. For an immutable global this value cannot be changed. There are additional rules associated with export/imported globals and their mutability and we will cover these in the export/import chapter.

A C programmer will be used to being able to scribble all over the all the memory in a process - which sometimes can have innovative and sometimes very disastrous consequences. With WebAssembly, the programmer does not have such uncontrolled accessibility. Each of the following storage areas are isolated from each other:

- locals & parameters,
- the value stack
- the call stack
- globals,
- memories (the heap)
- tables (for indirect function calls)

They have different instructions to interact with them programmatically and have different rules about how they operate. All mathematical computation (adds, substracts, etc.) and control flow (if else, branch, etc.) are performed based on operands placed on the value stack, so if a value from one of the other locations is needed, it first needs to be placed on the value stack. The first three cannot be exported/imported where the last three can.

In the WebAssembly text format, the global node type in the the S-Expression is a direct child of the module root node. It is an error to declare a global that is not a direct child of a module, such as trying to declare it as a child of a func node.

Globals are declared as follows:

```
(module (start $module_start) ;; val09.wat
```

```
  ;;      NAME                        MUTATABLE? TYPE   INITIALIZER

  (global $my_first_mutable_global    (mut       i32)   (i32.const 1) )
  (global $my_first_immutable_global             i32    (i32.const 2) )

  (func $module_start)

)
```

Note that the initializer is a S-Expression sub-node and that the type is an S-Expression attribute for immutables but an S-Expression sub-node for mutables (because we also need to specify that it is mutable).

Globals must be intialized via an initializer expression, such as a constant (as above) or via the value of a different imported global. When we run wasmdump with the -x or --details option on the above, we get:

```
wasmdump val09.wasm -x
val09.wasm:    file format wasm 0x000001

Section Details:

Type:
 - [0] () -> nil
Function:
 - func[0] sig=0
Global:
 - global[0] i32 mutable=1 - init i32=1
 - global[1] i32 mutable=0 - init i32=2
Start:
```

Note the different mutable setting and the effect of the initializer expression on each global.

Two instructions are provided to interact with globals:

- get_global – puts the value of the selected global on the value stack
- set_global – pops the top of value stack and uses it to set the value of the selected global

Let's see them in action:

```
(module (start $module_start) ;; val10.wat

  (global $my_first_mutable_global (mut i32) (i32.const 1) )
  (global $my_first_immutable_global i32  (i32.const 2) )

  (func $module_start
    call $global_demo
  )

  (func $global_demo
    ;; get a global and use result to set an mutable global
    get_global $my_first_immutable_global
     set_global $my_first_mutable_global

    ;; set an mutable global using a constant
    i32.const 10
```

```
      set_global $my_first_mutable_global
    )

  )
```

It is an error to omit the initializer (it does not default to 0).

It is an error to use as the initializer expression anything that is not a simple const (e.g. performing a mathematical operation such as add) or get_global on an imported global.

It is an error to try to set an immutable global.

It is an error to try to set a global with an empty value stack (since it is popped and the result used to set the global).

It is an error to try to create two globals with the same name.

4: Operators

Overview

Mathematical Instructions

https://github.com/WebAssembly/spec/blob/c11a7ad9e79d873744e4b94377261b3b526f2791/test/core/globals.wast

5: Flow Control

Overview

WebAssembly provides low-level flow control constructs which can be combined in various ways to deliver to module developers a rich collection of flow control possibilities. There are three block-like structures that contain zero or more instructions. These are blocks, loops and if-then/if-then-else and each concludes with an end instruction.

Supporting branching includes:

- un-conditional branching
- conditional branching
- jump tables

Branching inside blocks and if-then/if-then-else result in execution continuing after the block (similar to a break instruction in C). Branching inside a loop results in execution recommencing at the beginning of the loop (similar to a continue instruction in C).

If-Then and If-Then-Else

These constructs allow taking different paths through code based on a condition. The `if` instruction pops the value on the top of the stack and if it is non-zero, it executes the `then` block, otherwise, if there is an `else` block, that is executed.

In this sample we create a mutable global and then in the `start` function we put 13 on the stack, use it to set the `$my_value` local. Then just before the if we put 1 on the stack. Here we hard code this, but more often in real code it will be the result f some computation or passed in as a function parameter. The if uses this to decide whether to execute the then block, which it does in this case which updates the value of the `$my_value` local. Afterwards, we use that to set our global.

```
(module (start $module_start) ;; flow01.wat
  (global $my_global_val (mut i32) (i32.const 0))
  (func $module_start (local $my_value i32)
    i32.const 13
    set_local $my_value   ;; initialize my_value to 13
    get_local $my_value
    i32.const 1           ;; the condition (non-zero so if will execute then)
    (if
     (then
       i32.const 49
       set_local $my_value
     )
    )
    get_local $my_value    ;; expect this to be 49
     set_global $my_global_val
     return
  )
```

```
)
```

An `else` construct can also be used. In this sample we set the number of my_value depending on the top of stack before the if, which is 1 in `$my_value` should become 403.

```
(module (start $module_start) ;; flow02.wat
  (func $module_start (local $my_value i32)
    i32.const 1049
    set_local $my_value
    i32.const 1
    (if
        (then
            i32.const 403
            set_local $my_value
         )
        (else
            i32.const 101
            set_local $my_value
        )
    )
  )
)
```

Block

A block is an optionally named block of instructions whose execution may result in something being left on the stack. When branching occurs and the target is the block, then execution continues after the end of the block. To to create a block named `$first` which will place an i32 on the value stack (which we delete outside of the block) we use this code:

```
(module (start $module_start) ;; flow03.wat
  (func $module_start
    (block $first i32
      i32.const 10
    )
    drop
  )
)
```

When run, it results in this trace:

```
wasm-interp flow03.wasm --trace
>>> running start function:
#0.    0: V:0  | i32.const $10
#0.    5: V:1  | drop
#0.    6: V:0  | return
```

The closing brace (‘)’) at the end of the block (and the end of the function) gets compiled into an end instruction. When we look at this in the disassembler, we see:

```
wasmdump flow03.wasm -d
flow03.wasm:    file format wasm 0x000001
Code Disassembly:
000018 func[0]:
 00001a: 02 7f                      | block i32
 00001c: 41 0a                      |   i32.const 0xa
 00001e: 0b                         | end
 00001f: 1a                         | drop
```

```
000020: 0b                          | end
```

On its own a block does not add any additional functionality (it is just a collection of instructions and were the instruction left but the block removed, then there would be no difference). It is when branching instructions are used inside blocks that blocks become useful.

Loops

A loop on its own is also just a collection of instructions. In this sample we have a loop that pushes three integers onto the value stack and deletes each one:

```
(module (start $module_start) ;; flow04.wat
  (func $module_start
    (loop (i32.const 0) (drop) (i32.const 1) (drop)(i32.const 2) drop)
  )
)
```

Loops (like blocks) can be named and can be declared to leave a value on the value stack. In this sample we give our loop a name `$my_loop` and state it will leave an i32 on the value stack. We add three integer to the value stack but only drop two of them, so at the end of the loop there remains an i32 on the value stack, which is the declared value for the loop. Hence after the loop we call drop again.

```
(module (start $module_start)      ;; flow05.wat
 (func $module_start
   (loop $my_loop i32
      (i32.const 0) (drop)
      (i32.const 1) (drop)
      (i32.const 2)
  )
  drop
  )
)
```

Branching

Blocks and loops without branching are essentially the same concept. It is when branching is introduced that they behave differently. When a block is the target of a branch, code execution continues after the block. In contrast, when a loop is the target of a branch, code execution continues at the beginning of the loop.

Let's explore by creating our first infinite loop:

```
(module (start $module_start) ;; flow06.wat
  (func $module_start
    (loop $my_loop
      (i32.const 0) (drop)
      br $my_loop    ;; infinite loop!!
  )
  )
)
```

When you run this in the wasm-interp, be prepared to press ctrl-c to break the infinite loop. In this sample we name our loop as `$my_loop` and use this in the `br` branch instruction. Branching works based on a control flow stack and nesting of blocks/loops/if-then results in an entry being added to the control flow stack and

when end is reached, an entry is popped. When a branch instruction is detected, it is accompanied by a count of the number of entries to pop. When we use a name as in flow06.wat this is not so clear, but it is from the wasmdump:

```
wasmdump flow06.wasm -d
flow06.wasm:    file format wasm 0x000001
Code Disassembly:
000018 func[0]:
 00001a: 03 40                      | loop
 00001c: 41 00                      |   i32.const 0
 00001e: 1a                         |   drop
 00001f: 0c 00                      |   br 0
 000021: 0b                         | end
 000022: 0b                         | end
```

We can re-write our infinite loop to use integers rather than names as follows:

```
(module (start $module_start) ;; flow07.wat
  (func $module_start
    (loop
     (i32.const 0) (drop)
      br 0    ;; infinite loop!!
  )
  )
)
```

The `br 0` instruction branches to the entry on the top of our control flow stack, which is the enclosing loop. Branches to loops re-commence loop execution and hence we get our infinite loop. Running wasmdump on this gives us the exact same output as with flow.06.wasm.

To get rid of the infinite loop, let's use block rather than loop. Branching to the block will execute the instruction after the end of the block.

```
(module (start $module_start) ;; flow08.wat
  (func $module_start
    (block
     (i32.const 0) (drop)
      br 0    ;; not an infinite loop!!
  )
  )
)
```

Running this in wasm-interp gives us:

```
wasm-interp flow08.wasm --trace
>>> running start function:
#0.    0: V:0  | i32.const $0
#0.    5: V:1  | drop
#0.    6: V:0  | br @11
#0.   11: V:0  | return
```

Note the branch of @11 here.

Often we see a block and a loop construct used together with `br 0` and `br 1` in different parts of the loop. (`br_if` comes in useful here – see next).

```
(module (start $module_start) ;; flow09.wat
  (func $module_start
    (block
```

```
      (loop
       (i32.const 0)
       (drop)
         (br 1)
       )
    )
    i32.const 23232 ;; put a random instruction here, so we see in trace
    drop
    )
)
```

Here we use `br 1` so we first pop the top of the control flow stack (the loop) and target the then top of the control flow stack, which is the block and so we continue execute after the block's end. The trace shows us what is happening:

```
wasm-interp flow09.wasm --trace
>>> running start function:
#0.    0: V:0  | i32.const $0
#0.    5: V:1  | drop
#0.    6: V:0  | br @11
#0.   11: V:0  | i32.const $23232
#0.   16: V:1  | drop
#0.   17: V:0  | return
```

Often the `br` is combined with `br_if` to supply more control.

```
(module (start $module_start) ;; flow10.wat
  (func $module_start
  i32.const 100
    (block
     i32.const 200
     i32.const 0
       br_if 0
     i32.const 300
        drop
        drop
    )
    i32.const 400
    return
    )
)
```

The trace output is:

```
wasm-interp flow10.wasm --trace
>>> running start function:
#0.    0: V:0  | i32.const $100
#0.    5: V:1  | i32.const $200
#0.   10: V:2  | i32.const $0
#0.   15: V:3  | br_unless @26, 0
#0.   26: V:2  | i32.const $300
#0.   31: V:3  | drop
#0.   32: V:2  | drop
#0.   33: V:1  | i32.const $400
#0.   38: V:2  | drop_keep $2 $0
#0.   44: V:0  | return
```

The final branching instruction is `br_table`, which branches based on a table of possible targets. Note the word table here has nothing to do with the word table used with `call_indirect`. Here it is just the general meaning of "table" as in a lookup

table.

Like `br_if`, an operand is taken off the value stack. Unlike `br_if` where a single target is provided as an immediate, with `br_table` a list of targets is provided as immediates with the last one being a default. The value taken from the value stack is used as an index into the target table, with the last one used if out of bounds.

```
br_table 0 1 2 1
```

This line says pop the top of the value stack, and use it as in index into the table of immediates (that is 0 1 2) and the last is for out of bounds. So if top of stack was 6, which is out of bounds, that that is equivalent to br 1 (the last immediate). If top of stack was say 1, then that is also equivalent to 1 since that is what we have as the second element of our table.

`br_table` is useful when you have nested block structures. Like the other branching instructions, `br_table` can only branch to a parent block structure – i.e. pop the control flow stack.

The full sample code is:

```
(module (start $module_start) ;; flow10.wat
  (func $module_start
  i32.const 100
     (block
       i32.const 200
       (block
         i32.const 300
         (block
           i32.const 400
           i32.const 6       ;; branch operand (out of bounds),
                              ;; so we use last in br_table as default
           br_table 0 1 2 1
       )
       i32.const 500
       drop
       drop
     )
     i32.const 600
     drop
     drop
   )
   i32.const 700
   return
  )
)
```

After the end of each block we have an `i32.const` with a different value and this shows up in the trace. Modify the line just before the br_table instruction to see the effects of `br_table`.

6: Functions

Overview

Functions are the sole container of instructions in WebAssembly. In other words, they are the only construct that executes code. Everything that higher level languages might be working with - class static methods, class instance methods, top-level functions, events, getters and setters for properties – are represented in WebAssembly as functions.

Functions are represented by the function s-expression node type and this must be a direct child of the module s-expression node type. Instances of the function node type can directly contain param node types, local node types and a result node type. Functions also contain instructions – these can be as hierarchical sets of instruction in s-expression format or flat lists.

Functions can be exported and imported (discussed in our chapter on Export/Import). By default, functions are internal to the module.

Function Index Space And Identifying Functions

Each function that a module has access to is assigned an index, as part of the module's function index space. Imported functions are first assigned an index (starting with 0), followed by assigning an index to each function defined inside the module, from top to bottom.

Functions can optionally be named. We would recommend naming functions as it is easy to do and makes life easier for developers.

```
(func $myfuncname)
```

Functions cannot be passed as parameters to other functions. If there is a need for this functionality then pass the function's i32 index as a function parameter.

Functions that can be identified at compile time can be called in code via the call instruction. Functions can be identified at runtime are called via the call_indirect instruction (called in our chapter on tables).

The Start Function

We have stated that a module is similar to a shared library or a DLL. A DLL in particular has the concept of DllMain – a well-known function that is automatically called when the DLL is loaded. The equivalent for WebAssembly is the module start function. Unlike DllMain, with WebAssembly the start function can be called anything you want. The module developer indicates to WebAssembly which function is the start function using the start node. This can either name the function or pass in the index of the function to be called when the module is loaded. We will use the name module_start in all our examples, but you are free to pick any name you want.

```
(start $module_start)
```

or

```
(start 100)
```

The use of start is optional but if present there may only be one at most.

So for C developers, it is conceptually similar to:

```
void module_start(){ … /* do something */ }
```

For WebAssembly developers, a full demo is:

```
(module                    ;; func01.wat
  (func $module_start
    ;; do something
  )
  (start $module_start)
)
```

It is important to note that the start function is not the same as a C main function. The start function takes no parameters. The start function returns no value.

For general WebAssembly applications, the start function is a good place for module-wide initialization functionality. For our demo purposes we use it as a way of launching our code samples. ant to be using exports, start is a perfect place to use for launching our demos.

If you save func01.wat to a file you can compile and run it with:

```
wast2wasm func01.wat -o func01.wasm
wasm-interp func01.wasm –trace
```

This time the output is:

```
>>> running start function:
#0.    0: V:0  | return
```

Note we omit the `--run-all-exports` switch, since here we are running the `start` function and not an exported function.

Function Return Values

Functions cannot be nested – so func nodes in the S-expression must be a direct child node of the module node. Instructions can only appear as a list of attributes within a func node and not in any other node. A function node in the wat S-Expression starts with an open brace (‘(‘) and concludes with a close brace (‘)’) and the end instruction is not needed (indeed, not allowed, as the close brace plays that role. It we try to use end for concluding a function, such as:

```
(module (start $module_start) ;; func02.wat
  (func $module_start
  end
  )
)
```

Then wast2wasm gives us this error stating it is expecting the closing brace:

```
wast2wasm func02.wat -o func02.wasm
```

```
func02.wat:3:2: syntax error, unexpected END, expecting )
        end
 ^^^
```

The end instruction will appear in the wasm file and shows up with wasmdump and wasm-interp --trace.

In previous demos we have seen how to return from a function without having a return value (to a C or C++ programmer, these are void functions). Now let's explore how to return a value. Return values have to be one of the four value types – i32, i64, f32 or f64. In the initial version of WebAssembly, at maximum only a single return value is allowed. In future, this is expected to be relaxed and multiple return values will be supported. For now, if you really need to return multiple values, put the data somewhere in memory and return the memory address of that data. The same applies if you wish to return something other that one of the four supported return value types.

The S-Expression node type for the return value is called result (not return!!) and as an attribute it needs to have one of the four supported value types.

```
(module (start $module_start) ;; func03.wat
  (func $module_start (result i32)
    i32.const 4
    return
  )
)
```

This sample shows a function with a result of value type i32 – but a point specific to our use of the start function to launch our demos, when we compile we get this error:

```
wast2wasm func03.wat -o func03.wasm
func03.wat:1:9: start function must not return anything
(module (start $module_start)   ;; func03.wat
        ^^^^^^^^^^^^^^^^^^^^
```

One of the rules of the `start` function is that it cannot have a return value. So for us to demo a return value, we will simply add an additional function which is allowed to have a return value and call it from start.

```
(module (start $module_start) ;; func04.wat
  (func $module_start
    call $my_func
  return
  )
 (func $my_func (result i32)
    i32.const 4
  return
  )
)
```

Here we define the `my_func` function with a result of i32 and in its instruction stream we place an i32 value on the stack. When we reach the return instruction, the value on the stack becomes the result. We call `my_func` using the `call` instruction.

Wasmdump gives us:

```
wasmdump func04.wasm -d
func04.wasm:    file format wasm 0x000001
```

```
Code Disassembly:
00001d func[0]:
 00001f: 10 01                      | call 0x1
 000021: 0f                         | return
 000022: 0b                         | end
000023 func[1]:
 000025: 41 04                      | i32.const 0x4
 000027: 0f                         | return
 000028: 0b                         | end
```

Running this code with wasm-interp –trace gives us:

```
wasm-interp func04.wasm --trace
>>> running start function:
#0.    0: V:0  | call @8
#1.    8: V:0  | i32.const $4
#1.   13: V:1  | return
#0.    5: V:1  | drop
#0.    6: V:0  | return
```

In wat text format, calls to functions can either be via a name or via an index. Functions are automatically assigned a zero-based index in document order of appearance in the wat file. In the wasm binary format, indices only are used and the name is not supported. Hence in the wasmdump output above, we see call 0x1 to call our function with index 1 (i.e. the second function in our wat file). So this demo gives us the same wasmdump and wasm-interp output as func04 demo:

```
(module (start $module_start) ;; func05.wat
  (func $module_start
    call 0x1
  return
  )
 (func $my_func (result i32)
    i32.const 4
  return
  )
)
```

For anything other than trivial sample code, uses names with function calls is a good idea. At some point over the lifetime of a source file, you will insert a new function int eh middle of the set of functions or remove one, and that will mean you have to change the index used in calls elsewhere – a lot of error-prone work.

If you call a function using an index that does not exist, e.g. `call 0x2` in our sample, this error will be reported:

```
function variable out of range (max 2)
    call 0x3
```

If you call a function using a name (e.g. `call $my_other_func`) that does not match a named function, this error will be reported:

```
undefined function variable "$my_other_func"
    call $my_other_func
         ^^^^^^^^^^^^^^
```

If you have a function with a return but upon return have an empty stack:

```
(module (start $module_start) ;; func07.wat
  (func $module_start
```

```
      call $my_func
      return
    )
    (func $my_func (result i32)
      return
    )
)
```

this error will be reported:

```
wast2wasm func07.wat -o func07.wasm
func07.wat:7:2:type stack size too small at return. got 0,expected at least 1
        return
 ^^^^^^
```

A function can call another function which itself calls another function, and what is the result can be passed back up, as follows:

```
(module (start $module_start) ;; func08.wat
  (func $module_start
    call $my_func
   return
   )
 (func $my_func (result i32)
    call $my_other_func
   return
   )
(func $my_other_func (result i32)
    i32.const 4
   return
   )
)
```

A dump of func08.wasm is:

```
wasmdump func08.wasm -d
func08.wasm:    file format wasm 0x000001
Code Disassembly:
00001e func[0]:
 000020: 10 01                      | call 0x1
 000022: 0f                         | return
 000023: 0b                         | end
000024 func[1]:
 000026: 10 02                      | call 0x2
 000028: 0f                         | return
 000029: 0b                         | end
00002a func[2]:
 00002c: 41 04                      | i32.const 0x4
 00002e: 0f                         | return
 00002f: 0b                         | end
```

Running that in wasm-interp gives us:

```
wasm-interp func08.wasm --trace
>>> running start function:
#0.    0: V:0  | call @8
#1.    8: V:0  | call @15
#2.   15: V:0  | i32.const $4
#2.   20: V:1  | return
#1.   13: V:1  | return
#0.    5: V:1  | drop
#0.    6: V:0  | return
```

Function Parameters and locals

In WebAssembly text format parameters for functions can be specified using the param node type, which must be an immediate child node of a function node type. Each parameter is associated with one of the four supported values types and this needs to be specified as an attribute. Parameters are indexed and optionally can be named (we prefer naming them). The caller of a function needs to populate the stack with values that are then used when the function call is made. So in the following code:

```
(module (start $module_start) ;; func09.wat
  (func $module_start
    call $my_func
  return
  )
(func $my_func (param $my_param i64)
  return
  )
)
```

we have forgotten to populate the stack, so wast2wasm reports this error:

```
wast2wasm func09.wat -o func09.wasm
func09.wat:3:5: type stack size too small at call. got 0, expected at least 1
    call $my_func
    ^^^^^^^^^^^^^
```

The solution to this problem is of course to add the parameter value to the stack before the function call:

```
(module (start $module_start) ;; func10.wat
  (func $module_start
    i64.const 100
    call $my_func
  return
  )
  (func $my_func (param $my_param i64)
  return
  )
)
```

If there are multiple parameters defined for the function, then at the call site multiple values need to be placed on the stack – and most importantly, they must be of the correct types. Hence this code:

```
(module (start $module_start) ;; func11.wat
  (func $module_start
    f64.const 100.0123
    i32.const 100
    call $my_func
  return
  )
(func $my_func (param $my_first_param i32)  (param $my_second_param f64)
  return
  )
)
```

produces this self-explanatory error in wast2wasm:

```
wast2wasm func11.wat -o func11.wasm
```

```
func11.wat:5:5: type mismatch in call, expected i32 but got f64.
    call $my_func
    ^^^^^^^^^^^^^
func11.wat:5:5: type mismatch in call, expected f64 but got i32.
    call $my_func
    ^^^^^^^^^^^^^
```

whereas this code works:

```
(module (start $module_start) ;; func12.wat
  (func $module_start
    i32.const 100
    f64.const 100.0123
    call $my_func
  return
  )
(func $my_func (param $my_first_param i32)  (param $my_second_param f64)
  return
  )
)
```

Varargs in parameters (varidic functions) are not supported. Where this is needed, such parameters need to be placed in a Memory, and a pointer and length passed in as normal function parameters.

Function parameters are in some ways like locals (they are accessed the same way) but in other ways are different (they have a different s-expression node type). Locals are local variables (which must have a type that is one of the four value types). There can be an infinite number of locals.

Locals are defined like this:

```
(module (start $module_start) ;; func13.wat
  (func $module_start
    call $my_func
  return
  )
(func $my_func (local $my_local i32)
  return
  )
)
```

The output from wasmdump is:

```
wasmdump -d func13.wasm
func13.wasm:    file format wasm 0x000001
Code Disassembly:
000019 func[0]:
 00001b: 10 01                      | call 0x1
 00001d: 0f                         | return
 00001e: 0b                         | end
00001f func[1]:
 000023: 0f                         | return
 000024: 0b                         | end
```

So defining a local is not an instruction itself – there not additional instruction for or local. We will see shortly when getting and setting locals that then instructions are added. When we execute func13.wasm with wasm-interp we see:

```
wasm-interp func13.wasm --trace
```

```
>>> running start function:
#0.    0: V:0  | call @7
#1.    7: V:0  | alloca $1
#1.   12: V:1  | drop
#1.   13: V:0  | return
#0.    5: V:0  | return
```

The one new line to note is the alloca $1 – this is an internal wasm-interp command to allocate enough memory for the local. What follows the alloca (here $1) is the number of additional value slots needed (one is this case). If we had two locals, then $2 would be used. So if the function signature were:

```
func $my_func (local $my_int_local i32) (local $my_float_local f32)
```

then the wasm-interp trace output would have:

```
#1.    7: V:0  | alloca $2
```

Note that the storage area for locals is disjoint to Memories. Locals are initialized to zero. There are three instructions provided to interact with locals:

- get_local – pushes the current value of a local on the top of value stack
- set_local – pops the top of stack value and uses it to set a local
- tee_local – both of above

Let's start with get and set local which are both used in this demo:

```
(module (start $module_start) ;; func15.wat
  (func $module_start
    call $my_func
  return
  )
(func $my_func (local $my_local i32)
  get_local $my_local
  i32.const 100
  set_local $my_local
  return
  )
)
```

The call to get_local will put 0 on the value stack, the call to i32.const 100 will put 100 on the value stack, and the all to set_local will remove 100 from the value stack and set it to be the value of the my_local.

In the next demo we use tee_local:

```
(module (start $module_start) ;; func16.wat
  (func $module_start
    call $my_func
  return
  )
(func $my_func (local $my_local i32)
  i32.const 100
  tee_local $my_local
  return
  )
)
```

i32.const 100 puts 100 on the value stack and tee_local will leave it there and also set 100 as the value of my_local.

Because function parameters are also a kind of local, these same three instructions – get_local, set_local and tee_local – are also used to interact with function parameters. As regards indexing, for functions that have both function parameters and locals, the parameters get indexed first)starting with zero), and then the locals afterwards.

```
(module (start $module_start) ;; func17.wat
  (func $module_start
    i64.const 100
    call $my_func
  return
  )
(func $my_func (param $my_param i64)
    get_local $my_param ;; 100 is now on the value stack
  return
  )
)
```

The set_local instruction can also be used with parameters (since they are just locals) but note each function has its own stack, so parameters modified by set_local in a called function are not visible in the calling function. If you need to pass changes back to the calling function, then a return value is needed, or else have a parameter as a pointer into a Memory, and make changes to the Memory. In general, we would recommend treating parameters as read-only.

7: (Linear) Memory

As regards data storage, what we have covered so far – locals and function parameters, globals and the value stack – are good for storing discrete instances of values. For smaller modules these sometimes are all that is needed. However, in many cases modules also need to manage data storage for larger chunks of data, and this is where WebAssembly's linear memory comes in. There are no actual restrictions that small chunks of data go in locals/parameters/globals/value stack and big chunks in linear memory, but quite often in real-world applications that is what happens.

Linear memory, also simply called memory in the WebAssembly world, is the equivalent of the heap that C developers will be familiar with. Imagine you were writing an image editor in WebAssembly to run in a web browser. You would create a linear memory instance and load the contents of a JPEG image into it. Afterwards, you would decode the JPEG header and perhaps place some of its fields of interest on the value stack and then call your WebAssembly functions to work with that.

Because of its characteristics, a more accurate analogy when compared with C programming would be blocks of heap memory managed by a slab memory allocator. The concept behind a slab memory allocator is to allocate one or more large blocks of memory once, and then use a slab allocator (a custom memory manager engine) to manage each large block as as a collection of smaller chunks – some of which will be in use and others available. The smaller chunks are distributed to application code as needed, and when such chunks are no longer needed, to recycle back into a free pool of chunks. The memory managed by the slab allocator can be resized by the application and there can be many blocks of them.

In the MVP of WebAssembly, a module can work with zero or one instances of linear memory. That may be defined locally within the module instance or may be imported. The linear memory that a module instance works with (either defined locally or imported) may be exported by that module instance (we will explore exporting and importing memories in a future chapter). Since multiple modules could import the same linear memory, this allowed controlled data sharing among modules.

What we discuss here is what is currently supported. Sharing memory across threads (web workers) and synchronization (atomics) are not yet part of WebAssembly, but these are likely to appear in future. The size of linear memory may be grown, but currently may not be shrunk. More that one linear memory per module instance is likely also to be supported in future.

In WebAssembly, linear memory works in terms of pages. A page is 64 KiB (that is 64 x 1024) in size. It is very important to note that the `memory` s-expression node and the current_`memory` and `grow_memory` instructions work in units of page size, not in byte counts as this can easily trip up new WebAssembly developers.

Most physical CPUs in use today are little-endian or bi-endian (supports both little and big endian) and so WebAssembly uses little-endian byte order exclusively. X86 and x8-64 are always little-endian, ARM processors can actually be configured to be either little-endian or big-endian but most in use today are configured as little-endian. Some older CPU types (e.g. Motorola 68000) are big endian only. One practical place you might see big-endian is the network byte order of IETF protocols. However, it is the developers of web browsers and web servers themselves who mostly deal with these, but some tool developers and networking developers may need to from time to time.

Creating and initializing memory

A (linear) memory is created using the memory s-expression node. To create a memory with 1 initial page which via the `grow_memory` instruction can later be grown to a maximum of 10, use:

```
(memory 1 10)
```

To create a memory with 1 initial page without a maximum number of pages use:

```
(memory 1)
```

A memory s-expression node has either one or two attributes. So just `(memory)` is an error. The first attribute in the initial number of pages. The second attribute is the maximum number of pages. It is an error to try to grow memory passed the max size. It is permitted to have 0 as the initial page count, but before any data is stored in the memory this would need to be grown via the `grow_memory` instruction:

```
(memory 0)
```

Memories can be named, similar to functions and globals:

```
(memory $my_mem 0)
```

In the MVP of WebAssembly only a single memory is supported, so unlike globals and functions, perhaps naming of memories is currently not that needed. In future, when multiple memories will be supported, this will change.

The initial content of memory defaults to 0. It can optionally be set in one or more data segments. These are stored in a `data` section that is like a .data section in traditional binaries. Each `data` s-expression node takes an initializer expression which is the offset into the memory and a string containing the data to put into the memory. In this example we have a memory with one page (of size 64 KiB) and we set its bytes at offset 0, 100 and 1000:

```
(module  ;; mem01.wat
 (memory 1)
 (data (i32.const 0) "hello")
 (data (i32.const 100) " , ")
 (data (i32.const 1000) "world!")
)
```

We have already come across initializer expressions with globals and the same rules apply with data – the values come from const instructions or a `get_global` on an imported global (globals defined with the module instance cannot be used).

Instructions for managing memory

The use of `(memory)` and `(data)` sets up memory when the module is instantiated. Later when that module instance executes functions, two instructions can be used to manage memory.

The `current memory` instruction pushes the page count on the value stack. The `grow_memory` instruction pops the value stack and uses the to grow the number of pages in a memory. It also pushes the previous (not the new!!) page count onto the value stack. Currently no instruction is supported to shrink the number of pages.

```
(module (start $module_start) ;; mem02.wat
  (memory 16)
  (func $module_start (local $myVal i32)
     current_memory
   set_local $myVal
    (grow_memory (i32.const 47))
   set_local $myVal
   (grow_memory (i32.const 200))
   set_local $myVal
  return
  )
)
```

In this demo with start with memory size of 16 pages. We call `current_memory` to get the current number of pages. We use a local `$myVal` to store this value so that is comes up in the trace. Then we grow the memory by adding 47 extra pages to give a new memory size of 63. The previous `current_memory` size (that would be 16) is pushed onto the value stack. Then we call `grow_memory` again to add an extra 200 pages. This results in the previous memory size (63 pages) being pushed on the value stack. If we run this code in the wasm-interp we get this trace output:

```
wasm-interp mem02.wasm --trace
>>> running start function:
#0.    0: V:0  | alloca $1
#0.    5: V:1  | current_memory $0
#0.   10: V:2  | set_local $1, 16
#0.   15: V:1  | i32.const $47
#0.   20: V:2  | grow_memory $0:47
#0.   25: V:2  | set_local $1, 16
#0.   30: V:1  | i32.const $200
#0.   35: V:2  | grow_memory $0:200
#0.   40: V:2  | set_local $1, 63
#0.   45: V:1  | drop
#0.   46: V:0  | return
```

If code attempts to grow past the maximum number of pages, `grow_memory` pushes -1 on the value stack (it does not trap). The following demo creates a memory with initial number of pages set to 1 and maximum to 3. In the start function, it tries to grow memory by an extra 120 pages. Because this is beyond the maximum, -1 is placed in the value stack. Later when we call `current_memory` we see it is still at 1.

```
(module (start $module_start) ;; mem03.wat
  (memory 1 3)
  (func $module_start (local $myVal i32)
    (grow_memory (i32.const 120))
  set_local $myVal
```

```
    current_memory
    set_local $myVal
    return
   )
)
```

The trace output from this is:

```
wasm-interp mem03.wasm --trace
>>> running start function:
#0.    0: V:0  | alloca $1
#0.    5: V:1  | i32.const $120
#0.   10: V:2  | grow_memory $0:120
#0.   15: V:2  | set_local $1, 4294967295
#0.   20: V:1  | current_memory $0
#0.   25: V:2  | set_local $1, 1
#0.   30: V:1  | drop
#0.   31: V:0  | return
```

Note the 4294967295 number you see in the trace after the call to grow_memory – hmm, wonder what that is? The code in mem03.wat starts by setting the memory maximum to 3 pages so when we try to grow_memory by an extra 120 pages, then this is an error, so -1 is placed on the value stack. The trace displays this as an unsigned integer, so it wraps to 2^{32}-1 which is 4294967295. So when you see 4294967295 you are looking at an error.

The final part of handling memory is the set of instructions for loading and storing values. There are different instructions for each value type but they operate in the same manner. The MVP of WebAssembly is 32-bit so when we use the phrase "address operand" below we currently always mean a 32-bit integer value. In future, when 64-bit wasm arrives, then a 64-bit integer addressing will be needed for that.

The primary four load instructions are:

- i32.load
- i64.load
- f32.load
- f64.load

To use these, a 32-bit integer value to represent the address operand is pushed onto the value stack, the appropriate load instruction is called, and the result is placed on the value stack (e.g. i64 for i64.load or f32 for f32.load). With regards addressing, WebAssembly also supports the concept of extra optional offsets, which we will cover shortly.

The primary store instructions are:

- i32.store
- i64.store
- f32.store
- f64.store

To use these, the appropriate value is pushed onto the value stack, followed by 32-bit integer value to represent the address operand and then the store instruction is called. All this is best shown with running code:

```
(module (start $module_start) ;; mem04.wat
  (memory 1)
  (func $module_start (local $my_local i32)
    i32.const 10
    i32.const 407
    i32.store
    i32.const 10
    i32.load
    set_local $my_local
    return
  )
)
```

Here we create a memory with 1 page (of size 64 KiB). We decide for some reason we wish to write the 32-bit integer 407 to the four bytes starting at address 10. So we first load 10 onto the value stack, then load 407 and then call i32.store. Next we wish to retrieve the 32-value of the four bytes starting at address 10 (yep, we expect to find 407). So after the store (when the value stack is empty), we again place 10 on the value stack and call i32.load, which gets the 4 bytes at address 10 and put them on the value stack. For ease of tracing, we put this value in a local. Running this code gives us:

```
wasm-interp mem04.wasm --trace
>>> running start function:
#0.    0: V:0  | alloca $1
#0.    5: V:1  | i32.const $10
#0.   10: V:2  | i32.const $407
#0.   15: V:3  | i32.store $0:10+$0, 407
#0.   24: V:1  | i32.const $10
#0.   29: V:2  | i32.load $0:10+$0
#0.   38: V:2  | set_local $1, 407
#0.   43: V:1  | drop
#0.   44: V:0  | return
```

Let's zoom in on the store and load calls and how they are traced:

```
#0.   15: V:3  | i32.store $0:10+$0, 407
#0.   29: V:2  | i32.load $0:10+$0
```

We notice the +$0 for each. This represents an address offset. We have not used these so far (they are optional) and as we see they default to 0. Address offsets allow finer grain control of address. Where we give address operands we can also give offsets, and both are added to give what is called the effective address, which is actually used for load and stores. Let's explore in code:

```
(module (start $module_start) ;; mem05.wat
  (memory 1)
  (func $module_start (local $my_local i32)
    i32.const 10
    i32.const 1
    (i32.store offset=1)
    i32.const 10
    i32.load
    set_local $my_local
    return
  )
)
```

Here we wish to store a number at an offset, so we have made two small changes to the previous example – instead of using our random number 407 as our sample number to store, we use 1. the reason is to clearly show what is happening with the stored number (folks will recognize 256 is 255 + 1, whereas 255+407 is not so clear). The second change is with the call to i32.store – we add an immediate setting the offset to 1. Let's run this code in wasm-interp:

```
wasm-interp mem05.wasm --trace
>>> running start function:
#0.    0: V:0  | alloca $1
#0.    5: V:1  | i32.const $10
#0.   10: V:2  | i32.const $1
#0.   15: V:3  | i32.store $0:10+$1, 1
#0.   24: V:1  | i32.const $10
#0.   29: V:2  | i32.load $0:10+$0
#0.   38: V:2  | set_local $1, 256
#0.   43: V:1  | drop
#0.   44: V:0  | return
```

We note that the call to `i32.store` now uses 10+$1 as the effective address and stored 1 in there. This is in effect an 8-bit shift of 1, which gives us 256. Therefore when later we are setting the value in memory to local, we see 256. If instead of 1, we use a different number, let's randomly pick 19 (and we can calculate that 19*256 = 4864), then we would see this output:

```
#0.   10: V:2  | i32.const $19
#0.   15: V:3  | i32.store $0:10+$1, 19
#0.   24: V:1  | i32.const $10
#0.   29: V:2  | i32.load $0:10+$0
#0.   38: V:2  | set_local $1, 4864
```

There are a number of additional load and store instructions specifically for integers when one wishes to work with fewer bytes than is in the value type. These can with be sign-extended (making a signed number) or zero-extended (making an unsigned number).

The instructions for loading a 4-byte i32 value onto the value stack from a smaller area of memory (8-bits or 16-bits) are:

- i32.load8_s
- i32.load8_u
- i32.load16_s
- i32.load16_u

The instructions for loading a 8-byte i64 value onto the value stack from a smaller area of memory (8-bits, 16-bits or 32-bits) are:

- i64.load8_s
- i64.load8_u
- i64.load16_s
- i64.load16_u
- i64.load32_s
- i64.load32_u

The following code shows this in action:

```
(module (start $module_start) ;; mem06.wat
  (memory 1)
  (func $module_start (local $my_local i32)
    i32.const 10
  i32.const 0xff
  i32.store
  i32.const 10
  i32.load8_u
  set_local $my_local
  return
  )
)
```

We put 0xFF (that is 255) onto the value stack and store it as a 32-bit integer in memory. Then we use the lower 8-bits of that (which is still 0xFF) and load this back onto the value stack using zero-extending, so 255 goes onto the value stack and then we use this to set the value of `$m_local`. The trace output is:

```
wasm-interp mem06.wasm --trace
>>> running start function:
#0.    0: V:0  | alloca $1
#0.    5: V:1  | i32.const $10
#0.   10: V:2  | i32.const $255
#0.   15: V:3  | i32.store $0:10+$0, 255
#0.   24: V:1  | i32.const $10
#0.   29: V:2  | i32.load8_u $0:10+$0
#0.   38: V:2  | set_local $1, 255
#0.   43: V:1  | drop
#0.   44: V:0  | return
```

If we change the `i32.load8_u` to `i32.load8_s` this means we wish to sign-extend the loaded value. Because we have 0xFF (the most significant bit/signed bit is set), this changes the output:

```
#0.   38: V:2  | set_local $1, 4294967295
```

(4294967295 is 2^{32}-1).

When we wish to store a smaller number of bytes than the value on the value stack is represented by, a number of specialist store instructions can help us. The following wrap an i32 value to an i8 or i16 and stores it:

- i32.store8
- i32.store16

The following wrap an i64 value to an i8, i16 or i32 and stores it:

- i64.store8
- i64.store16
- i64.store32

The following code shows this in action:

```
(module (start $module_start) ;; mem07.wat
  (memory 1)
  (func $module_start (local $my_local i32)
    i32.const 10
  i32.const 258
  i32.store8
  i32.const 10
```

```
    i32.load
    set_local $my_local
    return
   )
)
```

We put an i32 value of 258 on the value stack (note the lowest byte of this will a value of 2). We call `i32.store8` to pop the value on the value stack and takes the lowest byte (which is 2) to create a new i32 value and store this in memory. We then load this i32 into out `$my_local` (which is now 2). The trace result is:

```
wasm-interp mem07.wasm --trace
>>> running start function:
#0.    0: V:0  | alloca $1
#0.    5: V:1  | i32.const $10
#0.   10: V:2  | i32.const $258
#0.   15: V:3  | i32.store8 $0:10+$0, 258
#0.   24: V:1  | i32.const $10
#0.   29: V:2  | i32.load $0:10+$0
#0.   38: V:2  | set_local $1, 2
#0.   43: V:1  | drop
#0.   44: V:0  | return
```

8: Tables

Overview

Tables are where artifacts are stored to which access needs to be carefully controlled. In the MVP tables are exclusively used to store function pointers. In future tables could also be used to store references to objects from high-level managed (e.g. garbage collected) languages and handles to native OS resources and perhaps other artifacts.

Why are tables needed? Why not just use linear memory? When running C code on x86-64 there is heap memory (the equivalent of WebAssembly's liner memory) but there is no equivalent of WebAssembly's tables. The WebAssembly security architecture requires that access to artifacts that have potential for damage be carefully controlled. Most web browsers and web servers run applications in separate OS processes. Inside each of these processes will be some TypeScript/JavaScript code, perhaps some WebAssembly modules, and some internal web browser logic too. The process in totality will have access to underlying OS resources (e.g. file handles). So we don't want to allow WebAssembly code to have functions calls into random areas of memory within a process, as this may be outside the bounds of the WebAssembly module instance and somewhere else in process memory. We need to be able to restrict access to what can be executed and WebAssembly tables provides an elegant solution to enable this.

Tables are constructed via the `table` node type, their contents initialized via the `elem` element node type, and used (i.e. the function executed) via the `call_indirect` instruction. With normal WebAssembly function calls (using the `call` instruction), we decide at compile time which function to execute (if we wish to execute function 47, we literally write call 47 (or call with a function name which is simply an alias for 47) into our .wat file, this gets compiled to .wasm as call 47. There is no variation about which function gets executed – it will be function 47.

When we wish to dispatch the call at runtime (think of function pointers or C++ virtual functions), then this is where tables come in. We use the call_indirect instruction to access an element in a table, and this identifies what to execute.

Tables can be imported and exported which we will cover in our chapter on Export/Import. Tables can be accessed from the TypeScript/JavaScript API, which we will discuss in the chapter dedicated to that.

The Table Node Type

Tables are created in our .wat file using the table S-Expression node type:

```
(table 4 anyfunc)
```

In the MVP only a single table is supported per module and this may be defined inside the module or imported. A table may be named, but since there only is one, this is not

that useful. In future, this will change as multiple tables are supported. A table has an element type (what does it store) along with an initial number of elements and an optional maximum number of elements. For function pointers (the only element type supported currently) the type should be set to `anyfunc`.

By default, the value of these elements are set to a sentinel, which if called, will trap. Hence the elements should be populated (see `elem` node type, discussed next).

Here we create a table of initial length 4 with no maximum defined and with element type of anyfunc:

```
(module (start $module_start) ;; tab01.wat
  (table 4 anyfunc)
  (func $module_start
  return
  )
)
```

Here we add a maximum of 10:

```
(module (start $module_start) ;; tab02.wat
  (table 4 10 anyfunc)
  (func $module_start
  return
  )
)
```

Tables on their own are just inert data storage spaces. For them to be useful, you need to populate them with function indices using the `elem` node and then when you need to execute these functions use the `call_indirect` instruction.

The elem Element Node Type

We saw with liner memory that the `data` s-expression node type is used to initialize memory. The equivalent for tables is the `elem` element node type. Excluding the `elem` node type (which is set at compile time), there is no runtime way from within the WebAssembly module (no instruction) to influence the contents of a table. All we can do with tables from within WebAssembly functions is to use `call_indirect` to execute a function represented by an index in a table. There is more dynamic control with the TypeScript/JavaScript API (see later chapter).

The `elem` node type has an offset and a list of function names or indices. The identified functions are normal functions with no special characteristics. The following shows this in action:

```
(module (start $module_start) ;; tab03.wat
  (table 2 anyfunc)
  (elem (i32.const 0) $my_add $my_sub)
  (func $module_start
  return
  )
  (func $my_add (param i32 i32) (result i32)
    get_local 0
    get_local 1
    i32.add
  )
```

```
    (func $my_sub (param i32 i32) (result i32)
    get_local 0
    get_local 1
    i32.sub
  )
)
```

We have two functions, `$my_add` and `$my_sub` and use them in the `elem` node type to populate the table from offset 0 onward. Running wasmdump -x shows us:

```
wasmdump tab03.wasm -x
tab03.wasm:     file format wasm 0x000001

Section Details:

Type:
 - [0] () -> nil
 - [1] (i32, i32) -> i32
Function:
 - func[0] sig=0
 - func[1] sig=1
 - func[2] sig=1
Table:
 - table[0] type=anyfunc initial=2
Start:
Elem:
 - segment[0] table=0
 - init i32=0
  - func[1]
  - func[2]
```

The function signature for both our `$my_add` and `$my_sub` functions are the same and so the `sig` field for both is set to 1 (in case you are curious, there are two function signatures listed under the type section – the first is for the `start` function).

The call_indirect Instruction

So far we have populated our table with function pointers and now it is time to call these functions. With direct calls we use the call instruction passing in the function index to identify the function to execute. With indirect calls, we use the `call_indirect` instruction. This needs two immediate values – the index into the table of the function pointer to execute and a description of the signature of that function.

The `anyfunc` element type we used literally means the index for any function can be placed in the table. When calling that function, we wish type checking to occur – to ensure the number of and type of each of the parameters and the result that we expect in our code exactly matches what is in the table.

The `type` node is used to describe function signatures (excluding the function name). Here we create a function signature with a name of `$my_mathematical_func`, which takes two parameters of value type i32 and with produces a result of i32:

```
(type $my_mathematical_func (func (param i32 i32) (result i32)))
```

Defining a type like this is similar to defining a function, except the function here has

no function body (no instructions). Since the function name is not part of the type, it is quite normal for multiple functions to match a particular type.

We use this type as an immediate parameter to call_indirect, which in effect means we are telling WebAssembly will wish to execute a function (whose index in the table in at the top of the stack) with this signature:

```
      i32.const 0                          ;; index into table
      call_indirect $my_mathematical_func  ;; expected function signature
```

It is an error to execute a function from the table with a mismatched function signature. Here is the sample code in full:

```
(module (start $module_start) ;; tab04.wat
  (table 2 anyfunc)
  (elem (i32.const 0) $my_add $my_sub)

  (type $my_mathematical_func (func (param i32 i32) (result i32)))

  (func $module_start
    i32.const 44
    i32.const 66
    i32.const 0                          ;; index into table
    call_indirect $my_mathematical_func  ;; expected function signature
    return
  )
  (func $my_add (param i32 i32) (result i32)
    get_local 0
    get_local 1
    i32.add
  )
  (func $my_sub (param i32 i32) (result i32)
    get_local 0
    get_local 1
    i32.sub
  )
)
```

The output from wasmdump is:

```
wasmdump tab04.wasm -x
tab04.wasm:     file format wasm 0x000001

Section Details:

Type:
 - [0] (i32, i32) -> i32
 - [1] () -> nil
Function:
 - func[0] sig=1
 - func[1] sig=0
 - func[2] sig=0
Table:
 - table[0] type=anyfunc initial=2
Start:
Elem:
 - segment[0] table=0
 - init i32=0
  - func[1]
  - func[2]
```

When we run the code in wasm-interp we get:

```
wasm-interp tab04.wasm --trace
>>> running start function:
#0.    0: V:0  | i32.const $44
#0.    5: V:1  | i32.const $66
#0.   10: V:2  | i32.const $0
#0.   15: V:3  | call_indirect $0, 0
#1.   27: V:2  | get_local $2
#1.   32: V:3  | get_local $2
#1.   37: V:4  | i32.add 44, 66
#1.   38: V:3  | drop_keep $2 $1
#1.   44: V:1  | return
#0.   24: V:1  | drop
#0.   25: V:0  | return
```

- o -

The author of this guide, Eamon O'Tuathail (mailto:eamon.otuathail@clipcode.com), presents Clipcode's one-day WebAssembly intensive training course to clients around Europe. For more details, a data sheet and to book a presentation in your office for your team, please visit http://www.clipcode.net

This might also be of interest to WebAssembly developers:

- http://www.clipcode.net/training/clipcode-webassembly-concept-library.pdf