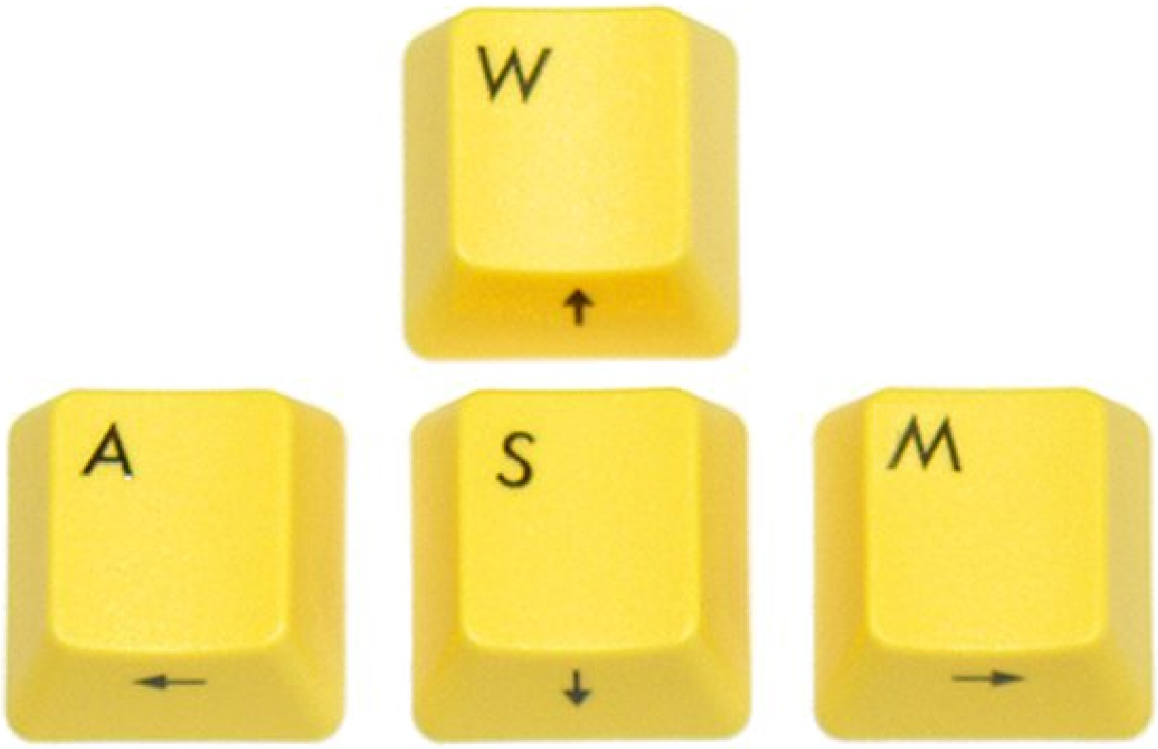

WebAssembly Specification

Release 1.0

WebAssembly Community Group

Mar 10, 2017

Contents

CHAPTER 1

Introduction

1.1 Introduction

WebAssembly (abbreviated Wasm [2]) is a *safe, portable, low-level code format* designed for efficient execution and compact representation. Its main goal is to enable high performance applications on the Web, but it does not make any Web-specific assumptions or provide Web-specific features, so can be employed in other environments as well.

WebAssembly is an open standard developed by a W3C Community Group[1] that includes representatives of all major browser vendors.

This document describes version 1.0 of the core WebAssembly standard. It is intended that it will be superseded by new incremental releases with additional features in the future.

1.1.1 Design Goals

The design goals of WebAssembly are the following:

- Fast, safe, and portable *semantics*:
  - **Fast**: executes with near native code performance, taking advantage of capabilities common to all contemporary hardware.
  - **Safe**: code is validated and executes in a memory-safe [3], sandboxed environment preventing data corruption or security breaches.
  - **Well-defined**: fully and precisely defines valid programs and their behavior in a way that is easy to reason about informally and formally.
  - **Hardware-independent**: can be compiled on all modern architectures, desktop or mobile devices and embedded systems alike.
  - **Language-independent**: does not privilege any particular language, programming model, or object model.
  - **Platform-independent**: can be embedded in browsers, run as a stand-alone VM, or integrated in other environments.
  - **Open**: programs can interoperate with their environment in a simple and universal manner.
- Efficient and portable *representation*:
  - **Compact**: a binary format that is fast to transmit by being smaller than typical text or native code formats.

[2] A contraction of "WebAssembly", not an acronym, hence not using all-caps.

[1] https://www.w3.org/community/webassembly/

[3] No program can break WebAssembly's memory model. Of course, it cannot guarantee that an unsafe language compiling to WebAssembly does not corrupt its own memory layout, e.g. inside WebAssembly's linear memory.

- **Modular**: programs can be split up in smaller parts that can be transmitted, cached, and consumed separately.
- **Efficient**: can be decoded, validated, and compiled in a fast single pass, equally with either just-in-time (JIT) or ahead-of-time (AOT) compilation.
- **Streamable**: allows decoding, validation, and compilation to begin as soon as possible, before all data has been seen.
- **Parallelizable**: allows decoding, validation, and compilation to be split into many independent parallel tasks.
- **Portable**: makes no architectural assumptions that are not broadly supported across modern hardware.

WebAssembly code is also intended to be easy to inspect and debug, especially in environments like web browsers, but such features are beyond the scope of this specification.

1.1.2 Scope

At its core, WebAssembly is a *virtual instruction set architecture (virtual ISA)*. As such, it has many use cases and can be embedded in many different environments. To encompass their variety and enable maximum reuse, the WebAssembly specification is split and layered into several documents.

This document is concerned with the core ISA layer of WebAssembly. It defines the instruction set, binary encoding, validation, and execution semantics. It does not, however, define how WebAssembly programs can interact with a specific environment they execute in, nor how they are invoked from such an environment.

Instead, this specification is complemented by additional documents defining interfaces to specific embedding environments such as the Web. These will each define a WebAssembly *application programming interface (API)* suitable for a given environment.

1.2 Overview

1.2.1 Concepts

WebAssembly encodes a low-level, assembly-like programming language. This language is structured around the following main concepts.

Values WebAssembly provides only four basic *value types*. These are integers and IEEE-754 floating point[4] numbers, each in 32 and 64 bit width. 32 bit integers also serve as Booleans and as memory addresses. The usual operations on these types are available, including the full matrix of conversions between them. There is no distinction between signed and unsigned integer types. Instead, integers are interpreted by respective operations as either unsigned or signed in 2's complement representation.

Instructions The computational model of WebAssembly is based on a *stack machine*. Code consists of sequences of *instructions* that are executed in order. Instructions manipulate values on an implicit *operand stack* [5] and fall into two main categories. Simple instructions perform basic operations on data. They pop arguments from the operand stack and push results back to it. *Control* instructions alter control flow. Control flow is *structured*, meaning it is expressed with well-nested constructs such as blocks, loops, and conditionals. Branches can only target such constructs.

Traps Under some conditions, certain instructions may produce a *trap*, which immediately aborts excecution. Traps cannot be handled by WebAssembly code, but are reported to the outside environment, where they typically can be caught.

[4] http://ieeexplore.ieee.org/document/4610935/

[5] In practice, implementations need not maintain an actual operand stack. Instead, the stack can be viewed as a set of anonymous registers that are implicitly referenced by instructions. The type system ensures that the stack height, and thus any referenced register, is always known statically.

Functions Code is organized into separate *functions*. Each function takes a sequence of values as parameters and returns a sequence of values as results. [6] Functions can call each other, including recursively, resulting in an implicit call stack that cannot be accessed directly. Functions may also declare mutable *local variables* that are usable as virtual registers.

Tables A *table* is an array of opaque values of a particular *element type*. It allows programs to select such values indirectly through a dynamic index operand. Currently, the only available element type is an untyped function reference. Thereby, a program can call functions indirectly through a dynamic index into a table. For example, this allows emulating function pointers with table indices.

Linear Memory A *linear memory* is a contiguous, mutable array of untyped bytes. Such a memory is created with an initial size but can be dynamically grown. A program can load and store values from/to a linear memory at any byte address (including unaligned). Integer loads and stores can specify a *storage size* which is smaller than the size of the respective value type. A trap occurs if access is not within the bounds of the current memory size.

Modules A WebAssembly binary takes the form of a *module* that contains definitions for functions, tables, and linear memories, as well as mutable or immutable *global variables*. Definitions can also be *imported*, specifying a module/name pair and a suitable type. Each definition can optionally be *exported* under one or more names. In addition to definitions, a module can define initialization data for its memory or table that takes the form of *segments* copied to given offsets. It can also define a *start function* that is automatically executed.

Embedder A WebAssembly implementation will typically be *embedded* into a *host* environment. This environment defines how loading of modules is initiated, how imports are provided (including host-side definitions), and how exports can be accessed. However, the details of any particular embedding are beyond the scope of this specification, and will instead be provided by complementary, environment-specific API definitions.

1.2.2 Semantic Phases

Conceptually, the semantics of WebAssembly is divided into three phases. For each part of the language, the specification specifies each of them.

Decoding WebAssembly modules are distributed in a *binary format*. *Decoding* processes that format and converts it into an internal representation of a module. In this specification, this representation is modelled by *abstract syntax*, but a real implementation could compile directly to machine code instead.

Validation A decoded module has to be *valid*. Validation checks a number of well-formedness conditions to guarantee that the module is meaningful and safe. In particular, it performs *type checking* of functions and the instruction sequences in their bodies, ensuring for example that the operand stack is used consistently.

Execution Finally, a valid module can be *executed*. Execution can be further divided into two phases:

Instantiation. An *instance* is the dynamic representation of a module, complete with its own state and execution stack. Instantiation executes the module body itself given definitions for all its imports. It initializes globals, memories and tables and invokes the module's start function if defined. It returns the instances of the module's exports.

Invocation. Once instantiated, further WebAssembly computations can be initiated by *invoking* an exported function of an instance. Given the required arguments, that executes the respective function and returns its results.

Instantiation and invocation are operations within the embedding environment.

[6] In the current version of WebAssembly, there may be at most one result value.

CHAPTER 2

Structure

2.1 Abstract Syntax

WebAssembly is a programming language that does not have a concrete syntax (other than the auxiliary text format). For conciseness, however, its structure is described in the form of an *abstract syntax*. All parts of this specification are defined in terms of this abstract syntax, including the decoding of the *binary format*.

2.1.1 Grammar

The following conventions are adopted in defining grammar rules for abstract syntax.

- Terminal symbols (atoms) are written in sans-serif: $\mathsf{i32}, \mathsf{end}$.
- Nonterminal symbols are written in italic: $\mathit{valtype}, \mathit{instr}$.
- A^n is a sequence of $n \geq 0$ iterations of A.
- A^* is a possibly empty sequence of iterations of A. (This is a shorthand for A^n used where n is not relevant.)
- A^+ is a non-empty sequence of iterations of A. (This is a shorthand for A^n where $n \geq 1$.)
- $A^?$ is an optional occurrence of A. (This is a shorthand for A^n where $n \leq 1$.)

Each non-terminal defines a syntactic class.

2.1.2 Auxiliary Notation

When dealing with syntactic constructs the following notation is also used:

- ϵ denotes the empty sequence.
- $|s|$ denotes the length of a sequence s.
- $s[i]$ denotes the i-th element of a sequence s, starting from 0.

Productions of the following form are interpreted as *records* that map a fixed set of fields field_i to values x_i, respectively:

$$r \ ::= \ \{\mathsf{field}_1\ x_1, \mathsf{field}_2\ x_2, \dots\}$$

The following notation is adopted for manipulating such records:

- $r.\mathsf{field}$ denotes the field component of r.
- $r, \mathsf{field}\ s$ denotes the same record as r but with the sequence s appended to its field component.

2.2 Values

2.2.1 Bytes

The simplest form of value are raw uninterpreted *bytes*. In the abstract they are represented as hexadecimal literals.

$$\mathit{byte} \quad ::= \quad \mathrm{0x00} \mid \dots \mid \mathrm{0xFF}$$

Conventions

- The meta variable b range over bytes.
- The meta function $\mathrm{byte}(n)$ denotes the byte representing the natural number $n < 256$.

2.2.2 Integers

Different classes of *integers* with different value ranges are distinguished by their *size* and their *signedness*.

$$\begin{array}{lll} \mathit{uint}_N & ::= & 0 \mid 1 \mid \dots \mid 2^N{-}1 \\ \mathit{sint}_N & ::= & -2^{N-1} \mid \dots \mid -1 \mid 0 \mid 1 \mid \dots \mid 2^{N-1}{-}1 \\ \mathit{int}_N & ::= & \mathit{uint}_N \mid \mathit{sint}_N \end{array}$$

The latter class defines *uninterpreted* integers, whose signedness interpretation can vary depending on context. A 2's complement interpretation is assumed for out-of-range values. That is, semantically, when interpreted as unsigned, negative values $-n$ convert to $2^N - n$, and when interpreted as signed, positive values $n \geq 2^{N-1}$ convert to $n - 2^N$.

Conventions

- The meta variables m, n, i, j, k range over unsigned integers.
- Numbers may be denoted by simple arithmetics.

2.2.3 Floating-point Numbers

Floating-point numbers are represented as binary values according to the IEEE-754[7] standard.

$$\mathit{float}_N \quad ::= \quad \mathit{byte}^{N/8}$$

The two possible sizes N are 32 and 64.

Conventions

- The meta variable z ranges over floating point values.

2.2.4 Vectors

Vectors are bracketed sequences of the form $[A^n]$ (or $[A^*]$), where the A-s can either be values or complex constructions.

$$\mathit{vec}(A) \quad ::= \quad [A^*]$$

[7] http://ieeexplore.ieee.org/document/4610935/

Conventions

- $|v|$ denotes the length of a vector v.
- $v[i]$ denotes the i-th element of a vector v, starting from 0.

2.2.5 Names

Names are vectors of bytes interpreted as character strings.

$$\mathit{name} \quad ::= \quad \mathit{vec}(\mathit{byte})$$

Todo

Unicode?

2.3 Types

2.3.1 Value Types

Value types classify the individual values that WebAssembly code can compute with and the values that a variable accepts.

$$\mathit{valtype} \quad ::= \quad \mathsf{i32} \mid \mathsf{i64} \mid \mathsf{f32} \mid \mathsf{f64}$$

The types int_{32} and int_{64} classify 32 and 64 bit integers, respectively. Integers are not inherently signed or unsigned, their interpretation is determined by individual operations.

The types float_{32} and float_{64} classify 32 and 64 bit floating points, respectively. They correspond to single and double precision floating point types as defined by the IEEE-754[8] standard

Conventions

- The meta variable t ranges over value types where clear from context.

2.3.2 Result Types

Result types classify the results of functions or blocks, which is a sequence of values.

$$\mathit{resulttype} \quad ::= \quad \mathit{valtype}^?$$

Note: In the current version of WebAssembly, at most one value is allowed as a result. However, this may be generalized to sequences of values in future versions.

---

[8] http://ieeexplore.ieee.org/document/4610935/

2.3.3 Function Types

Function types classify the signature of functions, mapping a sequence of parameters to a sequence of results.

$$functype \quad ::= \quad valtype^* \to resulttype$$

2.3.4 Memory Types

Memory types classify linear memories and their size range.

$$\begin{array}{lll} memtype & ::= & limits \\ limits & ::= & \{\mathsf{min}\ uint_{32}, \mathsf{max}\ uint_{32}{}^?\} \end{array}$$

The limits constrain the minimum and optionally the maximum size of a table. If no maximum is given, the table can grow to any size. Both values are given in units of page size.

2.3.5 Table Types

Table types classify tables over elements of *element types* within a given size range.

$$\begin{array}{lll} tabletype & ::= & limits\ elemtype \\ elemtype & ::= & \mathsf{anyfunc} \end{array}$$

Like memories, tables are constrained by limits for their minimum and optionally the maximum size. These sizes are given in numbers of entries.

The element type anyfunc is the infinite union of all *function types*. A table of that type thus contains references to functions of heterogeneous type.

Note: In future versions of WebAssembly, additional element types may be introduced.

2.3.6 Global Types

Global types classify global variables, which hold a value and can either be mutable or immutable.

$$globaltype \quad ::= \quad \mathsf{mut}^?\ valtype$$

2.3.7 External Types

External types classify imports and exports and their respective types.

$$\begin{array}{lll} externtype & ::= & \mathsf{func}\ functype\ | \\ & & \mathsf{table}\ tabletype\ | \\ & & \mathsf{mem}\ memtype\ | \\ & & \mathsf{global}\ globaltype \end{array}$$

2.4 Modules

WebAssembly programs are organized into *modules*, which are the unit of deployment, loading, and compilation. A module collects definitions for *types*, *functions*, *tables*, *memories*, and *globals*. In addition, it can declare *imports* and *exports* and provide initialization logic in the form of *data* and *element* segments or a *start function*.

$$
\begin{array}{lcll}
\mathit{module} & ::= & \{ & \mathsf{types}~\mathit{vec}(\mathit{functype}), \\
 & & & \mathsf{funcs}~\mathit{vec}(\mathit{func}), \\
 & & & \mathsf{tables}~\mathit{vec}(\mathit{table}), \\
 & & & \mathsf{mems}~\mathit{vec}(\mathit{mem}), \\
 & & & \mathsf{globals}~\mathit{vec}(\mathit{global}), \\
 & & & \mathsf{elem}~\mathit{vec}(\mathit{elemseg}), \\
 & & & \mathsf{data}~\mathit{vec}(\mathit{dataseg}), \\
 & & & \mathsf{start}~\mathit{funcidx}^?, \\
 & & & \mathsf{imports}~\mathit{vec}(\mathit{import}), \\
 & & & \mathsf{exports}~\mathit{vec}(\mathit{export}) \quad \}
\end{array}
$$

Each of the vectors – and thus the entire module – may be empty.

2.4.1 Indices

Definitions are referenced with zero-based *indices*. Each class of definition has its own *index space*, as distinguished by the following classes.

$$
\begin{array}{lcl}
\mathit{typeidx} & ::= & \mathit{uint}_{32} \\
\mathit{funcidx} & ::= & \mathit{uint}_{32} \\
\mathit{tableidx} & ::= & \mathit{uint}_{32} \\
\mathit{memidx} & ::= & \mathit{uint}_{32} \\
\mathit{globalidx} & ::= & \mathit{uint}_{32} \\
\mathit{localidx} & ::= & \mathit{uint}_{32} \\
\mathit{labelidx} & ::= & \mathit{uint}_{32}
\end{array}
$$

The index space for functions, tables, memories and globals includes respective imports declared in the same module. The indices of these imports precede the indices of other definitions in the same index space.

The index space for locals is only accessible inside a function and includes the parameters and local variables of that function, which precede the other locals.

Label indices reference block instructions inside an instruction sequence.

Conventions

- The meta variable l ranges over label indices.
- The meta variable x ranges over indices in any of the other index spaces.

2.4.2 Expressions

Function bodies, initialization values for *globals* and offsets of *element* or *data* segments are given as expressions, which are sequences of *instructions* terminated by an end marker.

$$
\mathit{expr} \quad ::= \quad \mathit{instr}^*~\mathsf{end}
$$

In some places, validation restricts expressions to be *constant*, which limits the set of allowable insructions.

2.4.3 Types

The types component of a module defines a vector of *function types*.

All function types used in a module must be defined in the type section. They are referenced by *type indices*.

Note: Future versions of WebAssembly may add additional forms of type definitions.

2.4.4 Functions

The funcs component of a module defines a vector of *functions* with the following structure:

$$\mathit{func} \quad ::= \quad \{\mathsf{type}\ \mathit{typeidx}, \mathsf{locals}\ \mathit{vec}(\mathit{valtype}), \mathsf{body}\ \mathit{expr}\}$$

The type of a function declares its signature by reference to a *type* defined in the module. The parameters of the function are referenced through 0-based *local indices* in the function's body.

The locals declare a vector of mutable local variables and their types. These variables are referenced through *local indices* in the function's body. The index of the first local is the smallest index not referencing a parameter.

The body is an *instruction* sequence that must evaluate to a stack matching the function type's *result type*.

Functions are referenced through *function indices*, starting with the smallest index not referencing a function *import*.

2.4.5 Tables

The tables component of a module defines a vector of *tables* described by their *table type*:

$$\mathit{table} \quad ::= \quad \{\mathsf{type}\ \mathit{tabletype}\}$$

A table is a vector of opaque values of a particular table *element type*. The min size in the *limits* of the table type of a definition specifies the initial size of that table, while its max, if present, restricts the size to which it can grow later.

Tables can be initialized through *element segments*.

Tables are referenced through *table indices*, starting with the smallest index not referencing a table *import*. Most constructs implicitly reference table index 0.

Note: In the current version of WebAssembly, at most one table may be defined or imported in a single module, and *all* constructs implicitly reference this table 0. This restriction may be lifted in future versions.

Tables can contain values that are not otherwise accessible – like host object references, raw OS handles, or native pointers – so that they can be accessed indirectly through an integer index. That bridges the gap between low-level, untrusted linear memory and high-level opaque handles or references. Currently, the primary purpose of tables is to emulate function pointers, which can be represented as integers indexing into a table of type anyfunc holding functions and can be called via the call_indirect instruction.

2.4.6 Memories

The mems component of a module defines a vector of *linear memories* (or *memories* for short) as described by their *memory type*:

$$\mathit{mem} \quad ::= \quad \{\mathsf{type}\ \mathit{memtype}\}$$

A memory is a vector of raw uninterpreted bytes. The min size in the *limits* of the memory type of a definition specifies the initial size of that memory, while its max, if present, restricts the size to which it can grow later. Both are in units of page size.

Memories can be initialized through *data segments*.

Memories are referenced through *memory indices*, starting with the smallest index not referencing a memory *import*. Most constructs implicitly reference memory index 0.

Note: In the current version of WebAssembly, at most one memory may be defined or imported in a single module, and *all* constructs implicitly reference this memory 0. This restriction may be lifted in future versions.

It is unspecified how embedders map this array into their process' own virtual memory. However, linear memory is sandboxed and does not alias other memory regions.

2.4.7 Globals

The globals component of a module defines a vector of *global variables* (or *globals* for short):

$$global \quad ::= \quad \{\mathsf{type}\ globaltype, \mathsf{init}\ expr\}$$

Each global stores a single value of the given *global type*. Its type also specifies whether a global is immutable or mutable. Moreover, each global is initialized with an init value given by a constant initializer *expression*.

Globals are referenced through *global indices*, starting with the smallest index not referencing a global *import*.

2.4.8 Element Segments

The initial contents of a table is uninitialized. The elem component of a module defines a vector of *element segments* that initialize a subrange of a table at a given offset from a static vector of elements.

$$elemseg \quad ::= \quad \{\mathsf{table}\ tableidx, \mathsf{offset}\ expr, \mathsf{init}\ vec(funcidx)\}$$

The offset is given by a constant *expression*.

Note: In the current version of WebAssembly, at most one table is allowed in a module. Consequently, the only valid $tableidx$ is 0.

2.4.9 Data Segments

The initial contents of a memory are zero bytes. The data component of a module defines a vector of *data segments* that initialize a range of memory at a given offset with a static vector of bytes.

$$dataseg \quad ::= \quad \{\mathsf{mem}\ memidx, \mathsf{offset}\ expr, \mathsf{init}\ vec(byte)\}$$

The offset is given by a constant *expression*.

Note: In the current version of WebAssembly, at most one memory is allowed in a module. Consequently, the only valid $memidx$ is 0.

2.4.10 Start Function

The start component of a module denotes the function index of an optional *start function* that is automatically invoked when the module is instantiated, after tables and memories have been initialized.

2.4.11 Exports

The exports component of a module defines a set of *exports* that become accessible to the host environment once the module has been instantiated.

$$\begin{array}{lll} \mathit{export} & ::= & \{\mathsf{name}\ \mathit{name}, \mathsf{desc}\ \mathit{exportdesc}\} \\ \mathit{exportdesc} & ::= & \mathsf{func}\ \mathit{funcidx}\ | \\ & & \mathsf{table}\ \mathit{tableidx}\ | \\ & & \mathsf{mem}\ \mathit{memidx}\ | \\ & & \mathsf{global}\ \mathit{globalidx} \end{array}$$

Each export is identified by a unique *name*. Exportable definitions are *functions*, *tables*, *memories*, and *globals*, which are referenced through a respective descriptor.

Note: In the current version of WebAssembly, only *immutable* globals may be exported.

2.4.12 Imports

The imports component of a module defines a set of *imports* that are required for instantiation.

$$\begin{array}{lll} \mathit{import} & ::= & \{\mathsf{module}\ \mathit{name}, \mathsf{name}\ \mathit{name}, \mathsf{desc}\ \mathit{importdesc}\} \\ \mathit{importdesc} & ::= & \mathsf{func}\ \mathit{typeidx}\ | \\ & & \mathsf{table}\ \mathit{tabletype}\ | \\ & & \mathsf{mem}\ \mathit{memtype}\ | \\ & & \mathsf{global}\ \mathit{globaltype} \end{array}$$

Each import is identified by a two-level *name* space, consisting of a module name and a unique name for an entity within that module. Importable definitions are *functions*, *tables*, *memories*, and *globals*. Each import is specified by a descriptor with a respective type that a definition provided during instantiation is required to match.

Every import defines an index in the respective index space. In each index space, the indices of imports go before the first index of any definition contained in the module itself.

Note: In the current version of WebAssembly, only *immutable* globals may be imported.

2.5 Instructions

Todo

Describe

$$\begin{array}{lll} \mathit{nn}, \mathit{mm} & ::= & 32\ |\ 64 \\ \mathit{sx} & ::= & \mathsf{u}\ |\ \mathsf{s} \\ \mathit{memop} & ::= & \{\mathsf{align}\ \mathit{uint}_{32}, \mathsf{offset}\ \mathit{uint}_{32}\} \end{array}$$

$$\begin{array}{lll}
\mathit{instr} & ::= & \mathsf{unreachable} \mid \\
&& \mathsf{nop} \mid \\
&& \mathsf{block}~\mathit{resulttype}~\mathit{instr}^*~\mathsf{end} \mid \\
&& \mathsf{loop}~\mathit{resulttype}~\mathit{instr}^*~\mathsf{end} \mid \\
&& \mathsf{if}~\mathit{resulttype}~\mathit{instr}^*~\mathsf{else}~\mathit{instr}^*~\mathsf{end} \mid \\
&& \mathsf{br}~\mathit{labelidx} \mid \\
&& \mathsf{br\_if}~\mathit{labelidx} \mid \\
&& \mathsf{br\_table}~\mathit{vec}(\mathit{labelidx})~\mathit{labelidx} \mid \\
&& \mathsf{return} \mid \\
&& \mathsf{call}~\mathit{funcidx} \mid \\
&& \mathsf{call\_indirect}~\mathit{typeidx} \mid \\
&& \mathsf{drop} \mid \\
&& \mathsf{select} \mid \\
&& \mathsf{get\_local}~\mathit{localidx} \mid \\
&& \mathsf{set\_local}~\mathit{localidx} \mid \\
&& \mathsf{tee\_local}~\mathit{localidx} \mid \\
&& \mathsf{get\_global}~\mathit{globalidx} \mid \\
&& \mathsf{set\_global}~\mathit{globalidx} \mid \\
&& \mathsf{i}\mathit{nn}\mathsf{.load}~\mathit{memop} \mid \mathsf{f}\mathit{nn}\mathsf{.load}~\mathit{memop} \mid \\
&& \mathsf{i}\mathit{nn}\mathsf{.store}~\mathit{memop} \mid \mathsf{f}\mathit{nn}\mathsf{.store}~\mathit{memop} \mid \\
&& \mathsf{i}\mathit{nn}\mathsf{.load8\_}\mathit{sx}~\mathit{memop} \mid \\
&& \mathsf{i}\mathit{nn}\mathsf{.load16\_}\mathit{sx}~\mathit{memop} \mid \\
&& \mathsf{i64.load32\_}\mathit{sx}~\mathit{memop} \mid \\
&& \mathsf{i}\mathit{nn}\mathsf{.store8}~\mathit{memop} \mid \\
&& \mathsf{i}\mathit{nn}\mathsf{.store16}~\mathit{memop} \mid \\
&& \mathsf{i64.store32}~\mathit{memop} \mid \\
&& \mathsf{i}\mathit{nn}\mathsf{.const}~\mathit{int}_{\mathit{nn}} \mid \mathsf{f}\mathit{nn}\mathsf{.const}~\mathit{float}_{\mathit{nn}} \mid \\
&& \mathsf{i}\mathit{nn}\mathsf{.eqz} \mid \\
&& \mathsf{i}\mathit{nn}\mathsf{.eq} \mid \mathsf{i}\mathit{nn}\mathsf{.ne} \mid \mathsf{i}\mathit{nn}\mathsf{.lt\_}\mathit{sx} \mid \mathsf{i}\mathit{nn}\mathsf{.gt\_}\mathit{sx} \mid \mathsf{i}\mathit{nn}\mathsf{.le\_}\mathit{sx} \mid \mathsf{i}\mathit{nn}\mathsf{.ge\_}\mathit{sx} \mid \\
&& \mathsf{f}\mathit{nn}\mathsf{.eq} \mid \mathsf{f}\mathit{nn}\mathsf{.ne} \mid \mathsf{f}\mathit{nn}\mathsf{.lt} \mid \mathsf{f}\mathit{nn}\mathsf{.gt} \mid \mathsf{f}\mathit{nn}\mathsf{.le} \mid \mathsf{f}\mathit{nn}\mathsf{.ge} \mid \\
&& \mathsf{i}\mathit{nn}\mathsf{.clz} \mid \mathsf{i}\mathit{nn}\mathsf{.ctz} \mid \mathsf{i}\mathit{nn}\mathsf{.popcnt} \mid \\
&& \mathsf{i}\mathit{nn}\mathsf{.add} \mid \mathsf{i}\mathit{nn}\mathsf{.sub} \mid \mathsf{i}\mathit{nn}\mathsf{.mul} \mid \mathsf{i}\mathit{nn}\mathsf{.div\_}\mathit{sx} \mid \mathsf{i}\mathit{nn}\mathsf{.rem\_}\mathit{sx} \mid \\
&& \mathsf{i}\mathit{nn}\mathsf{.and} \mid \mathsf{i}\mathit{nn}\mathsf{.or} \mid \mathsf{i}\mathit{nn}\mathsf{.xor} \mid \\
&& \mathsf{i}\mathit{nn}\mathsf{.shl} \mid \mathsf{i}\mathit{nn}\mathsf{.shr\_}\mathit{sx} \mid \mathsf{i}\mathit{nn}\mathsf{.rotl} \mid \mathsf{i}\mathit{nn}\mathsf{.rotr} \mid \\
&& \mathsf{f}\mathit{nn}\mathsf{.abs} \mid \mathsf{f}\mathit{nn}\mathsf{.neg} \mid \mathsf{f}\mathit{nn}\mathsf{.sqrt} \mid \\
&& \mathsf{f}\mathit{nn}\mathsf{.ceil} \mid \mathsf{f}\mathit{nn}\mathsf{.floor} \mid \mathsf{f}\mathit{nn}\mathsf{.trunc} \mid \mathsf{f}\mathit{nn}\mathsf{.nearest} \mid \\
&& \mathsf{f}\mathit{nn}\mathsf{.add} \mid \mathsf{f}\mathit{nn}\mathsf{.sub} \mid \mathsf{f}\mathit{nn}\mathsf{.mul} \mid \mathsf{f}\mathit{nn}\mathsf{.div} \mid \\
&& \mathsf{f}\mathit{nn}\mathsf{.min} \mid \mathsf{f}\mathit{nn}\mathsf{.max} \mid \mathsf{f}\mathit{nn}\mathsf{.copysign} \mid \\
&& \mathsf{i32.wrap/i64} \mid \mathsf{i64.extend\_}\mathit{sx}\mathsf{/i32} \mid \mathsf{i}\mathit{nn}\mathsf{.trunc\_}\mathit{sx}\mathsf{/f}\mathit{mm} \mid \\
&& \mathsf{f32.demote/f64} \mid \mathsf{f64.promote/f32} \mid \mathsf{f}\mathit{nn}\mathsf{.convert\_}\mathit{sx}\mathsf{/i}\mathit{mm} \mid \\
&& \mathsf{i}\mathit{nn}\mathsf{.reinterpret/f}\mathit{nn} \mid \mathsf{f}\mathit{nn}\mathsf{.reinterpret/i}\mathit{nn}
\end{array}$$

CHAPTER 3

Validation

3.1 Conventions

Validation is described *declaratively* in terms of rules specifying constraints on phrases of abstract syntax. These can be understood as typing rules.

For every construct, the validation rules are given in two equivalent forms:

1. In *prose*, describing the meaning in intuitive form.
2. In *formal notation*, describing the rule in mathematical form.

Note: Understanding of the formal notation is *not* required to read this specification, but it provides an effective means of communication and is amenable to mathematical proof.

3.1.1 Contexts

Validity of a definition is defined relative to a *context* C, which collects relevant information about the surrounding module and other definitions:

- *Types*: the list of types defined in the current module.
- *Functions*: the list of functions declared in the current module, represented by their function type.
- *Table*: the optional table declared in the current module, represented by its table type.
- *Memory*: the optional memory declared in the current module, represented by its memory type.
- *Globals*: the list of globals declared in the current module, represented by their global type.
- *Locals*: the list of locals declared in the current function (including parameters), represented by their value type.
- *Labels*: the stack of labels accessible from the current position, represented by their result type.

Locals and labels are only used for validating function bodies, and are left empty elsewhere. The label stack is the only part of the context that changes as validation of a function body proceeds.

Formally, a context can be defined as a record of the aforementioned components, described by the following grammar:

$$\begin{array}{llll}
\text{(context)} & C & ::= & \{ \mathsf{types} & \mathit{functype}^*, \\
&&& \phantom{\{}\ \mathsf{funcs} & \mathit{functype}^*, \\
&&& \phantom{\{}\ \mathsf{tables} & \mathit{tabletype}^*, \\
&&& \phantom{\{}\ \mathsf{mems} & \mathit{memtype}^*, \\
&&& \phantom{\{}\ \mathsf{globals} & \mathit{globaltype}^*, \\
&&& \phantom{\{}\ \mathsf{locals} & \mathit{valtype}^*, \\
&&& \phantom{\{}\ \mathsf{labels} & \mathit{resulttype}^* \ \}
\end{array}$$

3.1.2 Formal Notation

Note: This section gives a brief explanation of the notation for specifying typing rules formally. For the interested reader, a more thorough introduction can be found in respective text books. [9]

The proposition that a phrase x has a type t is written $x : t$. In general, however, typing is dependent on a context C. To express this, the complete form is a *judgement* $C \vdash x : t$, which says that $x : t$ holds under the assumptions encoded in C. In this document, the “$C \vdash$” part is often omitted when C either does not matter or is clear from context.

The typing rules use a standard approach for specifying formal type systems, rendering them into *deduction rules*. Every rule has the following general form:

$$\frac{premise_1 \qquad premise_2 \qquad \dots \qquad premise_n}{conclusion}$$

Such a rule is read as a big implication: if all premises hold, then the conclusion holds. Some rules have no premises; they are *axioms* whose conclusion holds unconditionally.

A judgement holds when there is a deduction rule with a matching conclusion for which all premises hold, recursively until only axioms are reached.

Note: For example, consider the following rudimentary language of expressions over numbers and Booleans:

$$expr \quad ::= \quad num \mid ident \mid expr + expr \mid expr = expr \mid \mathsf{if}\ expr\ expr\ expr$$

The typing rules for this language can be expressed as follows:

$$\frac{}{C \vdash num : \mathsf{Num}} \qquad \frac{C(ident) = t}{C \vdash ident : t}$$

$$\frac{C \vdash expr_1 : \mathsf{num} \quad C \vdash expr_2 : \mathsf{num}}{C \vdash expr_1 + expr_2 : \mathsf{num}} \qquad \frac{C \vdash expr_1 : t \quad C \vdash expr_2 : t}{C \vdash expr_1 = expr_2 : \mathsf{bool}}$$

$$\frac{C \vdash expr_1 : \mathsf{bool} \qquad C \vdash expr_2 : t \qquad C \vdash expr_3 : t}{C \vdash \mathsf{if}\ expr_1\ expr_2\ expr_3 : t}$$

There is one rule for each syntactic construct, defining the typing of that construct. The rule for numbers is an axiom. The rule for identifiers refers to the context C to look up its type t (in this type system, the context is simply a mapping from identifiers to types). The other rules have premises deducing the types of subexpressions. In the rule for addition, both their types must be Num. In the rule for comparison, their type can be any t, but must be the same t for both operands. In the rule for the conditional, the first operand must be a Boolean, the others again can have any type but must be consistent.

For example, the type of the expression “if x y (y + 1)” can be derived by recursively applying the typing rules. Under a context where $C(\mathrm{x}) = \mathsf{bool}$ and $C(\mathrm{y}) = \mathsf{num}$ they will deduce this expression to have type Num. Under a context where, for example, $C(\mathrm{x}) = \mathsf{num}$, no such derivation exists, and the expression would be ill-typed. Similarly, if $C(\mathrm{x})$ is not defined, that is, x is unbound.

[9] For example: Benjamin Pierce. Types and Programming Languages. The MIT Press 2002

CHAPTER 4

Instantiation

CHAPTER 5

Execution

CHAPTER 6

Instructions

6.1 Overview

Todo

Describe

6.2 Control Instructions

6.2.1 Block Instructions

6.2.2 Branch Instructions

6.2.3 Call Instructions

6.2.4 Miscellaneous Control Instructions

6.3 Variable Instructions

6.4 Parametric Instructions

6.5 Numeric Instructions

6.5.1 Numeric Instructions

6.5.2 Integer Test Instructions

6.5.3 Integer Comparison Instructions

6.5.4 Floating Point Comparison Instructions

6.5.5 Unary Integer Instructions

6.5.6 Unary Floating Point Instructions

6.5.7 Binary Integer Instructions

6.5.8 Binary Floating Point Instructions

6.5.9 Conversion Instructions

6.5.10 Reinterpretation Instructions

6.6 Memory Instructions

6.6.1 Load Instructions

6.6.2 Memory Instructions

6.6.3 Store Instructions

6.7 Instruction Sequences

Todo

Describe

CHAPTER 7

Binary Format

7.1 Conventions

7.1.1 Encoding

The binary format for WebAssembly modules is defined by *encoding* functions that map abstract syntax to sequences of raw bytes. [11]

Some syntactic phrases have multiple possible encodings. For example, numbers may be encoded as if they had optional leading zeros. Consequently, encoding functions are in fact defined to yield a (non-empty) *set* of possible byte sequences for each phrase. Implementations of encoders producing WebAssembly binaries can pick any encoding contained in these sets.

7.1.2 Decoding

The defined encoding functions are all invertible in the sense that every possible byte sequence is the encoding of at most one syntactic phrase. Consequently, *decoding* is defined implicitly as the respective inverse function.

Where multiple encodings are possible, a decoder must accept all of them. Conversely, a decoder must reject all inputs that are not a possible encoding for any phrase.

7.1.3 Notation

The following notation is adopted in defining binary encoding functions.

- The meta variable b is used to range over byte values.
- A hexadecimal constant like $\mathrm{0x08}$ or $\mathrm{0xFF}$ denotes the respective byte value as a singleton set.
- $\mathrm{byte}(n)$ denotes the byte value of the natural number n (for $0 \leq n < 256$).
- $[\![x]\!]_t$ denotes the set of encodings of x of syntactic class t.

Note: For example, $[\![10]\!]_{uint_8}$ denotes the set of encodings of the $uint_8$ value 10, which is $\{\mathrm{0x0A}, \mathrm{0x8A}\ \mathrm{0x00}\}$.

- The concatenation $A\ B$ denotes the set of all encodings that are concatenations of an element from encoding A with an element from encoding B.

[11] Additional encoding layers – for example, introducing compression – may be defined on top the basic representation defined here. However, such layers are outside the scope of the current specification.

Note: For example, the concatenation $[\![10]\!]_{uint_8}\ [\![13]\!]_{uint_8}$ yields the set $\{$0x0A 0x0D, 0x0A 0x8D 0x00, 0x8A 0x00 0x0D, 0x8A 0x00 0x8D 0x00$\}$ of possible encodings.

- The definition of encoding functions is given in clause form:

$$\begin{array}{lcll}[\![x]\!]_t & = & \mathit{byte\ sequence\ encoding}\ x & (\text{side condition}) \\ [\![y]\!]_t & = & \mathit{byte\ sequence\ encoding}\ y & (\text{side condition})\end{array}$$

The encoding for a single syntactic class t may be defined by multiple clauses covering different cases. When clauses overlap, either is applicable, expressing multiple possible encodings. The result of the encoding function then is the union of all applicable clauses.

Note: For example, the encoding of $uint_8$ values in length-bound LEB128[12] format could be given as follows:

$$\begin{array}{lcll}[\![n]\!]_{uint_8} & = & \mathrm{byte}(n) & (n < 128) \\ [\![m \cdot 128 + n]\!]_{uint_8} & = & \mathrm{byte}(n+128)\ \mathrm{byte}(m) & \end{array}$$

For values smaller than 128, both clauses apply, resulting in a set of two possible encodings as sampled in the previous note.

7.2 Values

7.2.1 Integers

All integer numbers are encoded using the LEB128[13] variable-length integer encoding, in either unsigned or signed variant.

Unsigned integers are encoded in unsigned LEB128[14] format. As an additional constraint, the total number of bytes encoding a value of type $uint_N$ must not exceed $\mathrm{ceil}(N/7)$ bytes.

$$\begin{array}{lcll}[\![n]\!]_{uint_N} & = & \mathrm{byte}(n) & (n < 128) \\ [\![m \cdot 128 + n]\!]_{uint_N} & = & \mathrm{byte}(n+128)\ [\![m]\!]_{uint_{N-7}} & (n < 128 \wedge N > 7)\end{array}$$

Note: In the case of a value less than 128, both rules apply, allowing for optional padding bytes.

Signed integers are encoded in signed LEB128[15] format, which uses a 2's complement representation. As an additional constraint, the total number of bytes encoding a value of type $sint_N$ must not exceed $\mathrm{ceil}(N/7)$ bytes.

$$\begin{array}{lcll}[\![n]\!]_{sint_N} & = & \mathrm{byte}(n) & (0 \le n < 64) \\ [\![-n]\!]_{sint_N} & = & \mathrm{byte}(128-n) & (0 < n \le 64) \\ [\![\pm m \cdot 128 + n]\!]_{sint_N} & = & \mathrm{byte}(n+128)\ [\![\pm m]\!]_{sint_{N-7}} & (n < 128 \wedge N > 7)\end{array}$$

Note: In the case of a non-negative value less than 64, both the first and second rule apply, allowing for optional padding bytes. In the case of a negative value greater than or equal to -64, both the second and third rule apply, allowing for optional padding bytes.

[12] https://en.wikipedia.org/wiki/LEB128
[13] https://en.wikipedia.org/wiki/LEB128
[14] https://en.wikipedia.org/wiki/LEB128#Unsigned_LEB128
[15] https://en.wikipedia.org/wiki/LEB128#Signed_LEB128

Uninterpreted integers are encoded as signed, assuming a 2's complement interpretation.

$$\begin{array}{lcll}
[\![n]\!]_{int_N} & = & [\![n]\!]_{sint_N} & (-2^{N-1} \leq n < 2^{N-1}) \\
[\![n]\!]_{int_N} & = & [\![n - 2^N]\!]_{sint_N} & (n \geq 2^{N-1})
\end{array}$$

7.2.2 Floating-point Numbers

Floating point values are encoded directly by their IEEE bit pattern in little endian[16] byte order:

$$[\![z]\!]_{float_N} \quad = \quad z$$

7.2.3 Vectors

Vectors are encoded with their length followed by the encoding of their element sequence.

$$[\![[x^n]]\!]_{vec(\_)} \quad = \quad [\![n]\!]_{uint_{32}}\ [\![x]\!]^n$$

7.2.4 Strings

Strings are encoded directly as a vector of bytes.

$$\begin{array}{lcl}
[\![[b^*]]\!]_{string} & = & [\![[b^*]]\!] \\
[\![s]\!]_{name} & = & [\![s]\!]_{string}
\end{array}$$

7.3 Types

7.3.1 Value Types

$$\begin{array}{lcl}
[\![\mathsf{i32}]\!]_{valtype} & = & \mathrm{0x7F} \\
[\![\mathsf{i64}]\!]_{valtype} & = & \mathrm{0x7E} \\
[\![\mathsf{f32}]\!]_{valtype} & = & \mathrm{0x7D} \\
[\![\mathsf{f64}]\!]_{valtype} & = & \mathrm{0x7C}
\end{array}$$

Note: These bytes correspond to the encodings of small negative sX values. This scheme is so that types can coexist in a single space with (positive) indices into the type section, which may be relevant for future extensions Gaps in this scheme are also reserved for future extensions.

7.3.2 Result Types

$$\begin{array}{lcl}
[\![\epsilon]\!]_{resulttype} & = & \mathrm{0x40} \\
[\![t]\!]_{resulttype} & = & [\![t]\!]_{valtype}
\end{array}$$

[16] https://en.wikipedia.org/wiki/Endianness#Little-endian

7.3.3 Function Types

Function types are encoded by the byte $\mathrm{0x60}$ followed by the vector of parameter types and the vector of result types.

$$[\![t_1^* \to t_2^?]\!]_{\mathit{functype}} = \mathrm{0x60}\ [\![[t_1^*]]\!]\ [\![[t_2^?]]\!]$$

Note: For future extensibility, the result is encoded as a vector.

7.3.4 Memory Types

$$\begin{aligned}
[\![\mathit{limits}]\!]_{\mathit{memtype}} &= [\![\mathit{limits}]\!] \\[1em]
[\![\{\mathsf{min}\ n, \mathsf{max}\ \epsilon\}]\!]_{\mathit{limits}} &= \mathrm{0x00}\ [\![n]\!]_{\mathit{uint}_{32}} \\
[\![\{\mathsf{min}\ n, \mathsf{max}\ m\}]\!]_{\mathit{limits}} &= \mathrm{0x01}\ [\![n]\!]_{\mathit{uint}_{32}}\ [\![m]\!]_{\mathit{uint}_{32}}
\end{aligned}$$

7.3.5 Table Types

$$\begin{aligned}
[\![\mathit{limits}\ \mathit{elemtype}]\!]_{\mathit{tabletype}} &= [\![\mathit{elemtype}]\!]\ [\![\mathit{limits}]\!] \\
[\![\mathsf{anyfunc}]\!]_{\mathit{elemtype}} &= \mathrm{0x70}
\end{aligned}$$

7.3.6 Global Types

$$\begin{aligned}
[\![\mathit{mutability}\ t]\!]_{\mathit{globaltype}} &= [\![t]\!]_{\mathit{valtype}}\ [\![\mathit{mutability}]\!] \\
[\![\epsilon]\!]_{\mathit{mutability}} &= \mathrm{0x00} \\
[\![\mathsf{mut}]\!]_{\mathit{mutability}} &= \mathrm{0x01}
\end{aligned}$$

CHAPTER 8

Appendix: Formal Properties

Todo

Describe and sketch proof (progress, preservation, uniqueness)

CHAPTER 9

Appendix: Validation Algorithm

Todo

Describe algorithm, state correctness properties (soundness, completeness)

CHAPTER 10

Appendix: Text Format

Todo

Describe

CHAPTER 11

Appendix: Name Section

Todo

Describe