



WebAssembly Specification

Release 1.0

WebAssembly Community Group

Apr 11, 2017

1	Introduction	1
1.1	Introduction	1
1.1.1	Design Goals	1
1.1.2	Scope	2
1.2	Overview	2
1.2.1	Concepts	2
1.2.2	Semantic Phases	3
2	Structure	5
2.1	Conventions	5
2.1.1	Grammar	5
2.1.2	Auxiliary Notation	5
2.2	Values	6
2.2.1	Bytes	6
2.2.2	Integers	6
2.2.3	Floating-Point	6
2.2.4	Vectors	6
2.2.5	Names	7
2.3	Types	7
2.3.1	Value Types	7
2.3.2	Result Types	7
2.3.3	Function Types	8
2.3.4	Limits	8
2.3.5	Memory Types	8
2.3.6	Table Types	8
2.3.7	Global Types	8
2.3.8	External Types	9
2.4	Instructions	9
2.4.1	Numeric Instructions	10
2.4.2	Parametric Instructions	10
2.4.3	Variable Instructions	10
2.4.4	Memory Instructions	11
2.4.5	Control Instructions	11
2.4.6	Expressions	12
2.5	Modules	12
2.5.1	Indices	13
2.5.2	Types	13
2.5.3	Functions	13
2.5.4	Tables	14
2.5.5	Memories	14
2.5.6	Globals	14
2.5.7	Element Segments	15

2.5.8	Data Segments	15
2.5.9	Start Function	15
2.5.10	Exports	15
2.5.11	Imports	16
3	Validation	17
3.1	Conventions	17
3.1.1	Contexts	17
3.1.2	Textual Notation	18
3.1.3	Formal Notation	18
3.2	Instructions	19
3.2.1	Numeric Instructions	20
3.2.2	Parametric Instructions	21
3.2.3	Variable Instructions	21
3.2.4	Memory Instructions	22
3.2.5	Control Instructions	23
3.2.6	Instruction Sequences	26
3.2.7	Expressions	26
3.3	Modules	27
3.3.1	Auxiliary Rules	27
3.3.2	Functions	27
3.3.3	Tables	28
3.3.4	Memories	28
3.3.5	Globals	28
3.3.6	Element Segments	28
3.3.7	Data Segments	29
3.3.8	Start Function	29
3.3.9	Exports	29
3.3.10	Imports	30
3.3.11	Modules	31
4	Instantiation	33
5	Execution	35
6	Binary Format	37
6.1	Conventions	37
6.1.1	Grammar	37
6.1.2	Auxiliary Notation	38
6.2	Values	38
6.2.1	Bytes	38
6.2.2	Integers	38
6.2.3	Floating-Point	39
6.2.4	Vectors	39
6.2.5	Names	39
6.3	Types	40
6.3.1	Value Types	40
6.3.2	Result Types	40
6.3.3	Function Types	40
6.3.4	Limits	40
6.3.5	Memory Types	40
6.3.6	Table Types	41
6.3.7	Global Types	41
6.4	Instructions	41
6.4.1	Control Instructions	41
6.4.2	Parametric Instructions	42
6.4.3	Variable Instructions	42
6.4.4	Memory Instructions	42
6.4.5	Numeric Instructions	43

6.4.6	Expressions	45
6.5	Modules	45
6.5.1	Indices	46
6.5.2	Sections	46
6.5.3	Custom Section	46
6.5.4	Type Section	46
6.5.5	Import Section	47
6.5.6	Function Section	47
6.5.7	Table Section	47
6.5.8	Memory Section	47
6.5.9	Global Section	47
6.5.10	Export Section	48
6.5.11	Start Section	48
6.5.12	Element Section	48
6.5.13	Code Section	48
6.5.14	Data Section	49
6.5.15	Modules	49
7	Appendix: Formal Properties	51
8	Appendix: Validation Algorithm	53
9	Appendix: Text Format	55
10	Appendix: Name Section	57
11	Index of Instructions	59
	Index	63

Introduction

1.1 Introduction

WebAssembly (abbreviated Wasm²) 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](https://www.w3.org/community/webassembly/)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-safe³, 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](http://ieeexplore.ieee.org/document/4610935/)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 execution. 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.

2.1 Conventions

WebAssembly is a programming language that does not have a concrete textual 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: `i32`, `end`.
- Nonterminal symbols are written in italic: *valtype*, *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 an optional occurrence of A . (This is a shorthand for A^n where $n \leq 1$.)

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:

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

The following notation is adopted for manipulating such records:

- $r.\text{field}$ denotes the field component of r .

2.2 Values

2.2.1 Bytes

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

$$byte ::= 0x00 \mid \dots \mid 0xFF$$

Conventions

- The meta variable b range over bytes.
- Bytes are sometimes interpreted as natural numbers $n < 256$.

2.2.2 Integers

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

$$\begin{aligned}
uint_N &::= 0 \mid 1 \mid \dots \mid 2^N - 1 \\
sint_N &::= -2^{N-1} \mid \dots \mid -1 \mid 0 \mid 1 \mid \dots \mid 2^{N-1} - 1 \\
int_N &::= uint_N \mid sint_N
\end{aligned}$$

The latter class defines *uninterpreted* integers, whose signedness interpretation can vary depending on context. In those contexts, a conversion based on 2's complement will be applied for values that are out-of-range for a chosen interpretation. 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 $n - 2^N$.

Todo

once there, link to definition of conversion

Conventions

- The meta variables m, n, i range over integers.
- Numbers may be denoted by simple arithmetics, as in the grammar above.

2.2.3 Floating-Point

Floating-point data consists of values in binary floating-point format according to the IEEE 754⁷ standard.

$$float_N ::= byte^{N/8}$$

The two possible sizes N are 32 and 64.

2.2.4 Vectors

Vectors are bounded sequences of the form A^n (or A^*), where the A -s can either be values or complex constructions. A vector can have at most $2^{32} - 1$ elements.

$$vec(A) ::= A^n \quad (n < 2^{32})$$

⁷ <http://ieeexplore.ieee.org/document/4610935/>

2.2.5 Names

Names are sequences of scalar Unicode⁸ code points.

```

name      ::= codepoint*
codepoint ::= U+0000 | ... | U+D7FF | U+E000 | ... | U+10FFFF

```

Todo

The definition of a name as an arbitrary sequence of scalar code points is too general. So would be the definition of a vector. Only names whose UTF-8 encoding is within the bounds of the maximum vector lengths must be included. How specify this?

Convention

- Code points are sometimes used interchangeably with natural numbers $n < 1114112$.

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.

```

valtype ::= i32 | i64 | f32 | 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-754⁹ standard

Conventions

- The meta variable t ranges over value types where clear from context.
- The notation $|t|$ denotes the *width* of a value type in bytes. (That is, $|i32| = |f32| = 4$ and $|i64| = |f64| = 8$.)

2.3.2 Result Types

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

```

resulttype ::= [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://www.unicode.org/versions/latest/>

⁹ <http://ieeexplore.ieee.org/document/4610935/>

2.3.3 Function Types

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

$$\text{functype} ::= [\text{vec}(\text{valtype})] \rightarrow [\text{vec}(\text{valtype})]$$

Note: In the current version of WebAssembly, the length of the result type vector of a function may be at most 1. This restriction may be removed in future versions.

2.3.4 Limits

Limits classify the size range of resizeable storage like associated with *memory types* and *table types*.

$$\text{limits} ::= \{\min \text{uint}_{32}, \max \text{uint}_{32}^?\}$$

If no maximum is given, the respective storage can grow to any size.

2.3.5 Memory Types

Memory types classify linear memories and their size range.

$$\text{memtype} ::= \text{limits}$$

The limits constrain the minimum and optionally the maximum size of a memory. The limits are given in units of page size.

2.3.6 Table Types

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

$$\begin{aligned} \text{tabletype} &::= \text{limits } \text{elemtype} \\ \text{elemtype} &::= \text{anyfunc} \end{aligned}$$

Like memories, tables are constrained by limits for their minimum and optionally maximum size. The limits 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.

2.3.7 Global Types

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

$$\begin{aligned} \text{globaltype} &::= \text{mut}^? \text{valtype} \\ \text{mut} &::= \text{const} \mid \text{mut} \end{aligned}$$

2.3.8 External Types

External types classify imports and exports and their respective types.

$$\textit{externtype} ::= \text{func } \textit{functype} \mid \text{table } \textit{tabletype} \mid \text{mem } \textit{memtype} \mid \text{global } \textit{globaltype}$$

Conventions

The following auxiliary notation is defined for sequences of external types, filtering out entries of a specific kind in an order-preserving fashion:

- $\text{funcs}(\textit{externtype}^*) = [\textit{functype} \mid \text{func } \textit{functype} \in \textit{externtype}^*]$
- $\text{tables}(\textit{externtype}^*) = [\textit{tabletype} \mid \text{table } \textit{tabletype} \in \textit{externtype}^*]$
- $\text{mems}(\textit{externtype}^*) = [\textit{memtype} \mid \text{mem } \textit{memtype} \in \textit{externtype}^*]$
- $\text{globals}(\textit{externtype}^*) = [\textit{globaltype} \mid \text{global } \textit{globaltype} \in \textit{externtype}^*]$

2.4 Instructions

WebAssembly code consists of sequences of *instructions*. Its computational model is based on a *stack machine* in that instructions manipulate values on an implicit *operand stack*, *consuming* (popping) argument values and *returning* (pushing) result values.

Note: In the current version of WebAssembly, at most one result value can be pushed by a single instruction. This restriction may be lifted in future versions.

In addition to dynamic operands from the stack, some instructions also have static *immediate* arguments, typically *indices* or type annotations, which are part of the instruction itself.

Some instructions are *structured* in that they bracket nested sequences of instructions.

The following sections group instructions into a number of different categories.

2.4.1 Numeric Instructions

Numeric instructions provide basic operations over numeric values of specific type. These operations closely match respective operations available in hardware.

```

nn, mm ::= 32 | 64
sx      ::= u | s
instr   ::= inn.const intnn | fnn.const floatnn |
             inn.eqz |
             inn.eq | inn.ne | inn.ltsx | inn.gtsx | inn.lesx | inn.gesx |
             fnn.eq | fnn.ne | fnn.lt | fnn.gt | fnn.le | fnn.ge |
             inn.clz | inn.ctz | inn.popcnt |
             inn.add | inn.sub | inn.mul | inn.divsx | inn.remsx |
             inn.and | inn.or | inn.xor |
             inn.shl | inn.shrsx | inn.rotl | inn.rotr |
             fnn.abs | fnn.neg | fnn.sqrt |
             fnn.ceil | fnn.floor | fnn.trunc | fnn.nearest |
             fnn.add | fnn.sub | fnn.mul | fnn.div |
             fnn.min | fnn.max | fnn.copysign |
             i32.wrap/i64 | i64.extendsx/i32 | inn.truncsx/fmm |
             f32.demote/f64 | f64.promote/f32 | fnn.convertsx/imm |
             inn.reinterpret/fnn | fnn.reinterpret/inn

```

Numeric instructions are divided by *value type*. For each type, several subcategories can be distinguished:

- *Constants*: return a static constant.
- *Unary Operators*: consume one operand and produce one result of the respective type.
- *Binary Operators*: consume two operands and produce one result of the respective type.
- *Tests*: consume one operand of the respective type and produce a Boolean result.
- *Comparisons*: consume two operands of the respective type and produce a Boolean result.
- *Conversions*: consume a value of one type and produce a result of another (the source type of the conversion is the one after the “/”).

Some integer instructions come in two flavours, where a signedness annotation *sx* distinguishes whether the operands are to be interpreted as *unsigned* or *signed* integers. For the other integer instructions, the use of 2’s complement for the signed interpretation means that they behave the same regardless of signedness.

2.4.2 Parametric Instructions

Instructions in this group can operate on operands of any *value type*.

```

instr ::= ... |
          drop |
          select

```

The *drop* operator simply throws away a single operand.

The *select* operator selects one of its first two operands based on whether its third operand is zero or not.

2.4.3 Variable Instructions

Variable instructions are concerned with the access to *local* or *global* variables.

```

instr ::= ... |
          get_local localidx |
          set_local localidx |
          tee_local localidx |
          get_global globalidx |
          set_global globalidx |

```


These instructions get or set the values of variables, respectively. The `tee_local` instruction is like `set_local` but also returns its argument.

2.4.4 Memory Instructions

Instructions in this group are concerned with linear memory.

```

memarg ::= {offset uint32, align uint32}
instr   ::= ... |
              inn.load memarg | fnn.load memarg |
              inn.store memarg | fnn.store memarg |
              inn.load8_sx memarg | inn.load16_sx memarg | i64.load32_sx memarg |
              inn.store8 memarg | inn.store16 memarg | i64.store32 memarg |
              current_memory |
              grow_memory

```

Memory is accessed with `load` and `store` instructions for the different *value types*. They all take a *memory immediate* *memarg* that contains an address *offset* and an *alignment* hint. Integer loads and stores can optionally specify a *storage size* that is smaller than the width of the respective value type. In the case of loads, a sign extension mode *sx* is then required to select appropriate behavior.

The static address offset is added to the dynamic address operand, yielding a 33 bit *effective address* that is the zero-based index at which the memory is accessed. All values are read and written in *little endian*¹⁰ byte order. A trap results if any of the accessed memory bytes lies outside the address range implied by the memory's current size.

Note: Future version of WebAssembly might provide memory instructions with 64 bit address ranges.

The `current_memory` instruction returns the current size of a memory. The `grow_memory` instruction grows memory by a given delta and returns the previous size, or `-1` if enough memory cannot be allocated. Both instructions operate in units of page size.

Note: In the current version of WebAssembly, all memory instructions implicitly operate on *memory index* 0. This restriction may be lifted in future versions.

The precise semantics of memory instructions is described in the Instruction section.

2.4.5 Control Instructions

Instructions in this group affect the flow of control.

```

instr ::= ... |
          nop |
          unreachable |
          block resulttype instr* end |
          loop resulttype instr* end |
          if resulttype instr* else instr* end |
          br labelidx |
          br_if labelidx |
          br_table vec(labelidx) labelidx |
          return |
          call funcidx |
          call_indirect typeidx

```

The `nop` instruction does nothing.

¹⁰ <https://en.wikipedia.org/wiki/Endianness#Little-endian>

The `unreachable` instruction causes an unconditional trap.

The `block`, `loop` and `if` instructions are *structured* instructions. They bracket nested sequences of instructions, called *blocks*, terminated with, or separated by, `end` or `else` pseudo-instructions. As the grammar prescribes, they must be well-nested. A structured instruction can produce a value as described by the annotated *result type*.

Each structured control instruction introduces an implicit *label*. Labels are targets for branch instructions that reference them with *label indices*. Unlike with other index spaces, indexing of labels is relative by nesting depth, that is, label 0 refers to the innermost structured control instruction enclosing the referring branch instruction, while increasing indices refer to those farther out. Consequently, labels can only be referenced from *within* the associated structured control instruction. This also implies that branches can only be directed outwards, “breaking” from the block of the control construct they target. The exact effect depends on that control construct. In case of `block` or `if` it is a *forward jump*, resuming execution after the matching `end`. In case of `loop` it is a *backward jump* to the beginning of the loop.

Note: This enforces *structured control flow*. Intuitively, a branch targeting a `block` or `if` behaves like a break statement, while a branch targeting a `loop` behaves like a continue statement.

Branch instructions come in several flavors: `br` performs an unconditional branch, `br_if` performs a conditional branch, and `br_table` performs an indirect branch through an operand indexing into the label vector that is an immediate to the instruction, or to a default target if the operand is out of bounds. The `return` instruction is a shortcut for an unconditional branch to the outermost block, which implicitly is the body of the current function. Taking a branch *unwinds* the operand stack up to the height where the targeted structured control instruction was entered. However, forward branches that target a control instruction with a non-empty result type consume a matching operand first and push it back on the operand stack after unwinding, as a result for the terminated instruction.

The `call` instruction invokes another function, consuming the necessary arguments from the stack and returning the result values of the call. The `call_indirect` instruction calls a function indirectly through an operand indexing into a *table*. Since tables may contain function elements of heterogeneous type *anyfunc*, the callee is dynamically checked against the function type indexed by the instruction’s immediate, and the call aborted with a trap if it does not match.

Note: In the current version of WebAssembly, `call_indirect` implicitly operates on *table index* 0. This restriction may be lifted in future versions.

2.4.6 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.

$$expr ::= instr^* end$$

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

2.5 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*.

```

module ::= {
  types vec(functype),
  funcs vec(func),
  tables vec(table),
  mems vec(mem),
  globals vec(global),
  elem vec(elem),
  data vec(data),
  start start?,
  imports vec(import),
  exports vec(export) }

```

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

2.5.1 Indices

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

```

typeid    ::= uint32
funcidx   ::= uint32
tableidx  ::= uint32
memidx    ::= uint32
globalidx ::= uint32
localidx  ::= uint32
labelidx  ::= uint32

```

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 variables *x*, *y* ranges over indices in any of the other index spaces.

2.5.2 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.

2.5.3 Functions

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

```

func ::= {type typeid, locals vec(valtype), body 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 upon termination must produce 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*.

2.5.4 Tables

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

$$table ::= \{type\ tabletype\}$$

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.

2.5.5 Memories

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

$$mem ::= \{type\ memtype\}$$

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.

2.5.6 Globals

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

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

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*.

2.5.7 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.

$$elem ::= \{ \text{table } tableidx, \text{offset } expr, \text{init } vec(funcidx) \}$$

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.

2.5.8 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.

$$data ::= \{ \text{mem } memidx, \text{offset } expr, \text{init } vec(byte) \}$$

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.

2.5.9 Start Function

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

$$start ::= \{ \text{func } funcidx \}$$

2.5.10 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.

$$\begin{aligned} export & ::= \{ \text{name } name, \text{desc } exportdesc \} \\ exportdesc & ::= \text{func } funcidx \mid \\ & \quad \text{table } tableidx \mid \\ & \quad \text{mem } memidx \mid \\ & \quad \text{global } globalidx \end{aligned}$$

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.

2.5.11 Imports

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

```
import      ::= {module name, name name, desc importdesc}  
importdesc ::= func typeid |  
                  table tabletype |  
                  mem memtype |  
                  global globaltype
```

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.

Validation

3.1 Conventions

Validation checks that a WebAssembly module is well-formed. Only valid modules can be instantiated.

Validity is defined by a *type system* over the *abstract syntax* of both instructions and modules. For each piece of abstract syntax, there is a typing rule that specifies the constraints that apply to it. All 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: The prose and formal rules are equivalent, so that understanding of the formal notation is *not* required to read this specification. The formalism offers a more concise description in notation that is used widely in programming languages semantics and is readily amenable to mathematical proof.

In both cases, the rules are formulated in a *declarative* manner. That is, they only formulate the constraints, they do not define an algorithm. A sound and complete algorithm for type-checking instruction sequences according to this specification is provided in the appendix.

3.1.1 Contexts

Validity of an individual definition is specified relative to a *context*, which collects relevant information about the surrounding *module* and other definitions in scope:

- *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.
- *Tables*: the list of tables declared in the current module, represented by their table type.
- *Memories*: the list of memories declared in the current module, represented by their 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.

In other words, a context contains a sequence of suitable types for each *index space*, describing each defined entry in that space. Locals and labels are only used for validating *instructions* in *function bodies*, and are left empty elsewhere. The label stack is the only part of the context that changes as validation of an instruction sequence proceeds.

It is convenient to define contexts as *records* C with abstract syntax:

$$C ::= \{ \begin{array}{ll} \text{types} & \text{functype}^*, \\ \text{funcs} & \text{functype}^*, \\ \text{tables} & \text{tabletype}^*, \\ \text{mems} & \text{memtype}^*, \\ \text{globals} & \text{globaltype}^*, \\ \text{locals} & \text{valtype}^*, \\ \text{labels} & \text{resulttype}^* \end{array} \}$$

Note: The fields of a context are not defined as *vectors*, since their lengths are not bounded by the maximum vector size.

In addition to field access $C.\text{field}$ the following notation is adopted for manipulating contexts:

- When spelling out a context, empty fields are omitted.
- $C, \text{field } A^*$ denotes the same context as C but with the elements A^* prepended to its field component sequence.

Note: This notation is defined to *prepend* not *append*. It is only used in situations where the original $C.\text{field}$ is either empty or field is labels. In the latter case adding to the front is desired because the *label index* space is indexed relatively, that is, in reverse order of addition.

3.1.2 Textual Notation

Validation is specified by stylised rules for each relevant part of the *abstract syntax*. The rules not only state constraints defining when a phrase is valid, they also classify it with a type. A phrase A is said to be “valid with type T ”, if all constraints expressed by the respective rules are met. The form of T depends on what A is.

Note: For example, if A is a *function*, then T is a *function type*; for an A that is a *global*, T is a *global type*; and so on.

The rules implicitly assume a given *context* C . In some places, this context is locally extended to a context C' with additional entries. The formulation “Under context C' , ... *statement* ...” is adopted to express that the following statement must apply under the assumptions embodied in the extended context.

3.1.3 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.¹¹

The proposition that a phrase A has a respective type T is written $A : T$. In general, however, typing is dependent on the context C . To express this explicitly, the complete form is a *judgement* $C \vdash A : T$, which says that $A : T$ holds under the assumptions encoded in C .

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

$$\frac{\text{premise}_1 \quad \text{premise}_2 \quad \dots \quad \text{premise}_n}{\text{conclusion}}$$

¹¹ For example: Benjamin Pierce. *Types and Programming Languages*. The MIT Press 2002

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. The conclusion always is a judgment $C \vdash A : T$, and there is one respective rule for each relevant construct A of the abstract syntax.

Note: For example, the typing rule for the *instruction* `i32.add` can be given as an axiom:

$$\overline{C \vdash \text{i32.add} : [\text{i32 } \text{i32}] \rightarrow [\text{i32}]}$$

The instruction is always valid with type $[\text{i32 } \text{i32}] \rightarrow [\text{i32}]$ (saying that it consumes two `i32` values and produces one), independent from any side conditions.

An instruction like `get_local` can be typed as follows:

$$\frac{C.\text{local}[x] = t}{C \vdash \text{get_local } x : [] \rightarrow [t]}$$

Here, the premise enforces that the immediate *local index* x exists in the context. The instruction produces a value of its respective type t (and does not consume any values). If $C.\text{local}[x]$ does not exist then the premise does not hold, and the instruction is ill-typed.

Finally, a *structured* instruction requires a recursive rule, where the premise is itself a typing judgement:

$$\frac{C, \text{label } [t^?] \vdash \text{instr}^* : [] \rightarrow [t^?]}{C \vdash \text{block } [t^?] \text{ instr}^* \text{ end} : [] \rightarrow [t^?]}$$

A `block` instruction is only valid when the instruction sequence in its body is. Moreover, the result type must match the block's annotation $t^?$. If so, then the `block` instruction has the same type as the body. Inside the body an additional label of the same type is available, which is expressed by locally extending the context C with the additional label information for the premise.

3.2 Instructions

Instructions are classified by *function types* $[t_1^*] \rightarrow [t_2^*]$ that describe how they manipulate the operand stack. The types describe the required input stack with argument values of types t_1^* that an instruction pops off and the provided output stack with result values of types t_2^* that it pushes back.

Note: For example, the instruction `i32.add` has type $[\text{i32 } \text{i32}] \rightarrow [\text{i32}]$, consuming two `i32` values and producing one.

Typing extends to *instruction sequences* instr^* . Such a sequence has a *function types* $[t_1^*] \rightarrow [t_2^*]$ if the accumulative effect of executing the instructions is consuming values of types t_1^* off the operand stack and pushing new values of types t_2^* . For some instructions, the typing rules do not fully constrain the type, and therefor allow for multiple types. Such instructions are called *polymorphic*. Two degrees of polymorphism can be distinguished:

- *value-polymorphic*: the *value type* t of one or several individual operands is unconstrained. That is the case for all *parametric instructions* like `drop` and `select`.
- *stack-polymorphic*: the entire (or most of the) *function type* $[t_1^*] \rightarrow [t_2^*]$ of the instruction is unconstrained. That is the case for all *control instructions* that perform an *unconditional control transfer*, such as `unreachable`, `br`, `br_table`, and `return`.

In both cases, the unconstrained types or type sequences can be chosen arbitrarily, as long as they meet the constraints imposed for the surrounding parts of the program.

Note: For example, the `select` instruction is valid with type $[t \text{ } t \text{ } \text{i32}] \rightarrow [t]$, for any possible *value type* t . Consequently, both instruction sequences

(`i32.const 1`) (`i32.const 2`) (`i32.const 3`) `select`

and

(f64.const 1.0) (f64.const 2.0) (i32.const 3) select

are valid, with t in the typing of `select` being instantiated to `i32` or `f64`, respectively.

The `unreachable` instruction is valid with type $[t_1^*] \rightarrow [t_2^*]$ for any possible sequences of value types t_1^* and t_2^* . Consequently,

unreachable i32.add

is valid by assuming type $[] \rightarrow [i32\ i32]$ for the `unreachable` instruction. In contrast,

unreachable (i64.const 0) i32.add

is invalid, because there is no possible type to pick for the `unreachable` instruction that would make the sequence well-typed.

3.2.1 Numeric Instructions

In this section, the following grammar shorthands are adopted:

```

unop  ::= clz | ctz | popcnt | abs | neg | sqrt | ceil | floor | trunc | nearest
binop ::= add | sub | mul | div | div_sx | rem_sx | min | max | copysign |
        and | or | xor | shl | shr_sx | rotl | rotr
testop ::= eqz
relop  ::= eq | ne | lt | gt | le | ge | lt_sx | gt_sx | le_sx | ge_sx
cvtop  ::= wrap | extend_sx | trunc_sx | convert_sx | demote | promote | reinterpret

```

t .const c

- The instruction is valid with type $[] \rightarrow [t]$.

$$\overline{C \vdash t.\text{const } c : [] \rightarrow [t]}$$

t .unop

- The instruction is valid with type $[t] \rightarrow [t]$.

$$\overline{C \vdash t.\text{unop} : [t] \rightarrow [t]}$$

t .binop

- The instruction is valid with type $[t\ t] \rightarrow [t]$.

$$\overline{C \vdash t.\text{binop} : [t\ t] \rightarrow [t]}$$

t .testop

- The instruction is valid with type $[t] \rightarrow [i32]$.

$$\overline{C \vdash t.\text{testop} : [t] \rightarrow [i32]}$$

t.relop

- The instruction is valid with type $[t\ t] \rightarrow [i32]$.

$$\overline{C \vdash t.relop : [t\ t] \rightarrow [i32]}$$

t₂.cvtop/t₁

- The instruction is valid with type $[t_1] \rightarrow [t_2]$.

$$\overline{C \vdash t_2.cvtop/t_1 : [t_1] \rightarrow [t_2]}$$

3.2.2 Parametric Instructions

drop

- The instruction is valid with type $[t] \rightarrow []$, for any *value type* *t*.

$$\overline{C \vdash \text{drop} : [t] \rightarrow []}$$

select

- The instruction is valid with type $[t\ t\ i32] \rightarrow [t]$, for any *value type* *t*.

$$\overline{C \vdash \text{select} : [t\ t\ i32] \rightarrow [t]}$$

Note: Both *drop* and *select* are *value-polymorphic* instructions.

3.2.3 Variable Instructions

get_local x

- The local $C.\text{locals}[x]$ must be defined in the context.
- Let *t* be the *value type* $C.\text{locals}[x]$.
- Then the instruction is valid with type $[] \rightarrow [t]$.

$$\frac{C.\text{locals}[x] = t}{C \vdash \text{get_local } x : [] \rightarrow [t]}$$

set_local x

- The local $C.\text{locals}[x]$ must be defined in the context.
- Let *t* be the *value type* $C.\text{locals}[x]$.
- Then the instruction is valid with type $[t] \rightarrow []$.

$$\frac{C.\text{locals}[x] = t}{C \vdash \text{set_local } x : [t] \rightarrow []}$$

`tee_local x`

- The local $C.\text{locals}[x]$ must be defined in the context.
- Let t be the *value type* $C.\text{locals}[x]$.
- Then the instruction is valid with type $[t] \rightarrow [t]$.

$$\frac{C.\text{locals}[x] = t}{C \vdash \text{tee_local } x : [t] \rightarrow [t]}$$

`get_global x`

- The global $C.\text{globals}[x]$ must be defined in the context.
- Let *mut* t be the *value type* $C.\text{locals}[x]$.
- Then the instruction is valid with type $[] \rightarrow [t]$.

$$\frac{C.\text{globals}[x] = \text{mut } t}{C \vdash \text{get_global } x : [] \rightarrow [t]}$$

`set_global x`

- The global $C.\text{globals}[x]$ must be defined in the context.
- Let *mut* t be the *global type* $C.\text{globals}[x]$.
- The mutability *mut* must be *mut*.
- Then the instruction is valid with type $[t] \rightarrow []$.

$$\frac{C.\text{globals}[x] = \text{mut } t}{C \vdash \text{set_global } x : [t] \rightarrow []}$$

3.2.4 Memory Instructions

`t.load memarg`

- The memory $C.\text{mems}[0]$ must be defined in the context.
- The alignment $2^{\text{memarg.align}}$ must not be larger than the *width* of t .
- Then the instruction is valid with type $[i32] \rightarrow [t]$.

$$\frac{C.\text{mems}[0] = \text{memtype} \quad 2^{\text{memarg.align}} \leq |t|}{C \vdash t.\text{load memarg} : [i32] \rightarrow [t]}$$

`t.loadN_sx memarg`

- The memory $C.\text{mems}[0]$ must be defined in the context.
- The alignment $2^{\text{memarg.align}}$ must not be larger than N .
- Then the instruction is valid with type $[i32] \rightarrow [t]$.

$$\frac{C.\text{mems}[0] = \text{memtype} \quad 2^{\text{memarg.align}} \leq N}{C \vdash t.\text{loadN_sx memarg} : [i32] \rightarrow [t]}$$

t.store memarg

- The memory $C.\text{mems}[0]$ must be defined in the context.
- The alignment $2^{\text{memarg.align}}$ must not be larger than the *width* of t .
- Then the instruction is valid with type $[i32\ t] \rightarrow []$.

$$\frac{C.\text{mems}[0] = \text{memtype} \quad 2^{\text{memarg.align}} \leq |t|}{C \vdash t.\text{store memarg} : [i32\ t] \rightarrow []}$$

t.storeN memarg

- The memory $C.\text{mems}[0]$ must be defined in the context.
- The alignment $2^{\text{memarg.align}}$ must not be larger than N .
- Then the instruction is valid with type $[i32\ t] \rightarrow []$.

$$\frac{C.\text{mems}[0] = \text{memtype} \quad 2^{\text{memarg.align}} \leq N}{C \vdash t.\text{storeN memarg} : [i32\ t] \rightarrow []}$$

current_memory

- The memory $C.\text{mems}[0]$ must be defined in the context.
- Then the instruction is valid with type $[] \rightarrow [i32]$.

$$\frac{C.\text{mems}[0] = \text{memtype}}{C \vdash \text{current_memory} : [] \rightarrow [i32]}$$

grow_memory

- The memory $C.\text{mems}[0]$ must be defined in the context.
- Then the instruction is valid with type $[i32] \rightarrow [i32]$.

$$\frac{C.\text{mems}[0] = \text{memtype}}{C \vdash \text{grow_memory} : [i32] \rightarrow [i32]}$$

3.2.5 Control Instructions*nop*

- The instruction is valid with type $[] \rightarrow []$.

$$\overline{C \vdash \text{nop} : [] \rightarrow []}$$

unreachable

- The instruction is valid with type $[t_1^*] \rightarrow [t_2^*]$, for any sequences of *value types* t_1^* and t_2^* .

$$\overline{C \vdash \text{unreachable} : [t_1^*] \rightarrow [t_2^*]}$$

Note: The *unreachable* instruction is *stack-polymorphic*.

block $[t^?]$ *instr** end

- Let C' be the same *context* as C , but with the *result type* $[t^?]$ prepended to the labels vector.
- Under context C' , the instruction sequence *instr** must be *valid* with type $[] \rightarrow [t^?]$.
- Then the compound instruction is valid with type $[] \rightarrow [t^?]$.

$$\frac{C, \text{labels } [t^?] \vdash \text{instr}^* : [] \rightarrow [t^?]}{C \vdash \text{block } [t^?] \text{ instr}^* \text{ end} : [] \rightarrow [t^?]}$$

Note: The fact that the nested instruction sequence *instr** must have type $[] \rightarrow [t^?]$ implies that it cannot access operands that have been pushed on the stack before the block was entered. This may be generalized in future versions of WebAssembly.

loop $[t^?]$ *instr** end

- Let C' be the same *context* as C , but with the empty *result type* $[]$ prepended to the labels vector.
- Under context C' , the instruction sequence *instr** must be *valid* with type $[] \rightarrow [t^?]$.
- Then the compound instruction is valid with type $[] \rightarrow [t^?]$.

$$\frac{C, \text{labels } [] \vdash \text{instr}^* : [] \rightarrow [t^?]}{C \vdash \text{loop } [t^?] \text{ instr}^* \text{ end} : [] \rightarrow [t^?]}$$

Note: The fact that the nested instruction sequence *instr** must have type $[] \rightarrow [t^?]$ implies that it cannot access operands that have been pushed on the stack before the loop was entered. This may be generalized in future versions of WebAssembly.

if $[t^?]$ *instr**₁ else *instr**₂ end

- Let C' be the same *context* as C , but with the empty *result type* $[t^?]$ prepended to the labels vector.
- Under context C' , the instruction sequence *instr**₁ must be *valid* with type $[] \rightarrow [t^?]$.
- Under context C' , the instruction sequence *instr**₂ must be *valid* with type $[] \rightarrow [t^?]$.
- Then the compound instruction is valid with type $[] \rightarrow [t^?]$.

$$\frac{C, \text{labels } [t^?] \vdash \text{instr}_1^* : [] \rightarrow [t^?] \quad C, \text{labels } [t^?] \vdash \text{instr}_2^* : [] \rightarrow [t^?]}{C \vdash \text{if } [t^?] \text{ instr}_1^* \text{ else instr}_2^* \text{ end} : [i32] \rightarrow [t^?]}$$

Note: The fact that the nested instruction sequence *instr** must have type $[] \rightarrow [t^?]$ implies that it cannot access operands that have been pushed on the stack before the conditional was entered. This may be generalized in future versions of WebAssembly.

br l

- The label $C.\text{labels}[l]$ must be defined in the context.
- Let $[t^?]$ be the *result type* $C.\text{labels}[l]$.
- Then the instruction is valid with type $[t_1^* \ t_2^?] \rightarrow [t_2^*]$, for any sequences of *value types* t_1^* and t_2^* .

$$\frac{C.\text{labels}[l] = [t^?]}{C \vdash \text{br } l : [t_1^* \ t^?] \rightarrow [t_2^*]}$$

Note: The `br` instruction is *stack-polymorphic*.

`br_if l`

- The label $C.\text{labels}[l]$ must be defined in the context.
- Let $[t^?]$ be the *result type* $C.\text{labels}[l]$.
- Then the instruction is valid with type $[t^? \text{ i32}] \rightarrow [t^?]$.

$$\frac{C.\text{labels}[l] = [t^?]}{C \vdash \text{br_if } l : [t^? \text{ i32}] \rightarrow [t^?]}$$

`br_table l* lN`

- The label $C.\text{labels}[l]$ must be defined in the context.
- Let $[t^?]$ be the *result type* $C.\text{labels}[l_N]$.
- For all l_i in l^* , the label $C.\text{labels}[l_i]$ must be defined in the context.
- For all l_i in l^* , $C.\text{labels}[l_i]$ must be $t^?$.
- Then the instruction is valid with type $[t_1^* \ t^? \text{ i32}] \rightarrow [t_2^*]$, for any sequences of *value types* t_1^* and t_2^* .

$$\frac{(C.\text{labels}[l] = [t^?])^* \quad C.\text{labels}[l_N] = [t^?]}{C \vdash \text{br_table } l^* l_N : [t_1^* \ t^? \text{ i32}] \rightarrow [t_2^*]}$$

Note: The `br_table` instruction is *stack-polymorphic*.

`return`

- The label vector $C.\text{labels}$ must not be empty in the context.
- Let $[t^?]$ be the *result type* that is the last element of $C.\text{labels}$.
- Then the instruction is valid with type $[t_1^* \ t^?] \rightarrow [t_2^*]$, for any sequences of *value types* t_1^* and t_2^* .

$$\frac{C.\text{labels}[|C.\text{labels}| - 1] = [t^?]}{C \vdash \text{return} : [t_1^* \ t^?] \rightarrow [t_2^*]}$$

Note: The `return` instruction is *stack-polymorphic*.

`call x`

- The function $C.\text{funcs}[x]$ must be defined in the context.
- Then the instruction is valid with type $C.\text{funcs}[x]$.

$$\frac{C.\text{funcs}[x] = [t_1^*] \rightarrow [t_2^*]}{C \vdash \text{call } x : [t_1^*] \rightarrow [t_2^*]}$$

call_indirect x

- The table $C.tables[0]$ must be defined in the context.
- Let *limits elemtype* be the *table type* $C.tables[0]$.
- The *element type elemtype* must be *anyfunc*.
- The type $C.types[x]$ must be defined in the context.
- Then the instruction is valid with type $C.types[x]$.

$$\frac{C.tables[0] = \text{limits anyfunc} \quad C.types[x] = [t_1^*] \rightarrow [t_2^*]}{C \vdash \text{call_indirect } x : [t_1^*] \rightarrow [t_2^*]}$$

3.2.6 Instruction Sequences

Typing of instruction sequences is defined recursively.

Empty Instruction Sequence: ϵ

- The empty instruction sequence is valid with type $[t^*] \rightarrow [t^*]$, for any sequence of *value types* t^* .

$$\overline{C \vdash \epsilon : [t^*] \rightarrow [t^*]}$$

Non-empty Instruction Sequence: $instr^* instr_N$

- The instruction sequence $instr^*$ must be valid with type $[t_1^*] \rightarrow [t_2^*]$, for some sequences of *value types* t_1^* and t_2^* .
- The instruction $instr_N$ must be valid with type $[t^*] \rightarrow [t_3^*]$, for some sequences of *value types* t^* and t_3^* .
- There must be a sequence of *value types* t_0^* , such that $t_2^* = t_0^* t^*$.
- Then the combined instruction sequence is valid with type $[t_1^*] \rightarrow [t_0^* t_3^*]$.

$$\frac{C \vdash instr^* : [t_1^*] \rightarrow [t_0^* t^*] \quad C \vdash instr_N : [t^*] \rightarrow [t_3^*]}{C \vdash instr^* instr_N : [t_1^*] \rightarrow [t_0^* t_3^*]}$$

3.2.7 Expressions

Expressions *expr* are classified by *result types* of the form $[t^?]$.

*instr^** *end*

- The instruction sequence $instr^*$ must be *valid* with type $[] \rightarrow [t^?]$, for some optional *value type* $t^?$.
- Then the expression is valid with *result type* $[t^?]$.

$$\frac{C \vdash instr^* : [] \rightarrow [t^?]}{C \vdash instr^* \text{ end} : [t^?]}$$

Constant Expressions

- In a *constant* expression *instr*^{*} *end* all instructions in *instr*^{*} must be constant.
- A constant instruction *instr* must be:
 - either of the form *t.const c*,
 - or of the form *get_global x*, in which case *C.globals[x]* must be a *global type* of the form *const t*.

$$\frac{(C \vdash \textit{instr} \textit{const})^*}{C \vdash \textit{instr} \textit{end} \textit{const}} \quad \frac{}{C \vdash t.\textit{const} c \textit{const}} \quad \frac{C.\textit{globals}[x] = \textit{const} t}{C \vdash \textit{get_global} x \textit{const}}$$

Note: The definition of constant expression may be extended in future versions of WebAssembly.

3.3 Modules

Modules are valid when all the definitions they contain are valid. To that end, each definition is classified with a suitable type.

3.3.1 Auxiliary Rules

Limits $\{\min n, \max m^?\}$

- If the maximum $m^?$ is not empty, then its value must not be smaller than n .
- Then the limit is valid.

$$\frac{(n \leq m)^?}{\vdash \{\min n, \max m^?\} \text{ok}}$$

3.3.2 Functions

Functions *func* are classified by *function types* of the form $[t_1^*] \rightarrow [t_2^?]$.

$\{\textit{type } x, \textit{locals } t^*, \textit{body } \textit{expr}\}$

- The type $C.\textit{types}[x]$ must be defined in the context.
- Let $[t_1^*] \rightarrow [t_2^?]$ be the *function type* $C.\textit{types}[x]$.
- The length of $t_2^?$ must not be larger than 1.
- Let C' be the same *context* as C , but with:
 - the locals set to the sequence of *value types* $t_1^* t^*$, concatenating parameters and locals,
 - the labels set to the singular sequence with *result type* $[t_2^?][t_1^*] \rightarrow [t_2^?]$.
- Under the context C' , the expression *expr* must be valid with type $t_2^?$.
- Then the function definition is valid with type $[t_1^*] \rightarrow [t_2^?]$.

$$\frac{C.\text{types}[x] = [t_1^*] \rightarrow [t_2^?] \quad C, \text{locals } t_1^* t^*, \text{labels } [t_2^?] \vdash \text{expr} : [t_2^?]}{C \vdash \{\text{type } x, \text{locals } t^*, \text{body } \text{expr}\} : [t_1^*] \rightarrow [t_2^?]}$$

Note: The restriction on the length of the result types t_2^* may be lifted in future versions of WebAssembly.

3.3.3 Tables

Tables *table* are classified by *table types* of the form *limits elemtype*.

{type *limits elemtype*}

- The limits *limits* must be *valid*.
- Then the table definition is valid with type *limits elemtype*.

$$\frac{\vdash \text{limits ok}}{C \vdash \{\text{type } \text{limits elemtype}\} : \text{limits elemtype}}$$

3.3.4 Memories

Memories *mem* are classified by *memory types* of the form *limits*.

{type *limits*}

- The limits *limits* must be *valid*.
- Then the memory definition is valid with type *limits*.

$$\frac{\vdash \text{limits ok}}{C \vdash \{\text{type } \text{limits}\} : \text{limits elemtype}}$$

3.3.5 Globals

Globals *global* are classified by *global types* of the form *mut t*.

{type *mut t*, init *expr*}

- The expression *expr* must be *valid* with *result type* *[t]*.
- The expression *expr* must be *constant*.
- Then the global definition is valid with type *mut t*.

$$\frac{C \vdash \text{expr} : [t] \quad C \vdash \text{expr const}}{C \vdash \{\text{type } \text{mut } t, \text{init } \text{expr}\} : \text{mut } t}$$

3.3.6 Element Segments

Element segments *elem* are not classified by a type.

$\{\text{table } x, \text{offset } \textit{expr}, \text{init } y^*\}$

- The table $C.\text{tables}[x]$ must be defined in the context.
- Let $\textit{limits elemtype}$ be the *table type* $C.\text{tables}[x]$.
- The *element type elemtype* must be *anyfunc*.
- The expression \textit{expr} must be *valid* with *result type* $[i32]$.
- The expression \textit{expr} must be *constant*.
- For each y_i in y^* , the function $C.\text{funcs}[y]$ must be defined in the context.
- Then the element segment is valid.

$$\frac{C.\text{tables}[x] = \textit{limits anyfunc} \quad C \vdash \textit{expr} : [i32] \quad C \vdash \textit{expr} \text{ const} \quad (C.\text{funcs}[y] = \textit{functype})^*}{C \vdash \{\text{table } x, \text{offset } \textit{expr}, \text{init } y^*\} \text{ ok}}$$

3.3.7 Data Segments

Data segments *data* are not classified by any type.

$\{\text{mem } x, \text{offset } \textit{expr}, \text{init } b^*\}$

- The memory $C.\text{mems}[x]$ must be defined in the context.
- The expression \textit{expr} must be *valid* with *result type* $[i32]$.
- The expression \textit{expr} must be *constant*.
- Then the data segment is valid.

$$\frac{C.\text{mems}[x] = \textit{limits} \quad C \vdash \textit{expr} : [i32] \quad C \vdash \textit{expr} \text{ const}}{C \vdash \{\text{mem } x, \text{offset } \textit{expr}, \text{init } b^*\} \text{ ok}}$$

3.3.8 Start Function

Start function declarations *start* are not classified by any type.

$\{\text{func } x\}$

- The function $C.\text{funcs}[x]$ must be defined in the context.
- The type of $C.\text{funcs}[x]$ must be $[] \rightarrow []$.
- Then the start function is valid.

$$\frac{C.\text{funcs}[x] = [] \rightarrow []}{C \vdash \{\text{func } x\} \text{ ok}}$$

3.3.9 Exports

Exports *export* are classified by their export *name*. Export descriptions *exportdesc* are not classified by any type.

{name *name*, desc *exportdesc*}

- The export description *exportdesc* must be valid with type *externtype*.
- Then the export is valid with name *name*.

$$\frac{C \vdash \text{exportdesc ok}}{C \vdash \{\text{name } \textit{name}, \text{desc } \textit{exportdesc}\} : \textit{name}}$$

func *x*

- The function $C.\text{funcs}[x]$ must be defined in the context.
- Then the export description is valid.

$$\frac{C.\text{funcs}[x] = \textit{functype}}{C \vdash \text{func } x \text{ ok}}$$

table *x*

- The table $C.\text{tables}[x]$ must be defined in the context.
- Then the export description is valid.

$$\frac{C.\text{tables}[x] = \textit{tabletype}}{C \vdash \text{table } x \text{ ok}}$$

mem *x*

- The memory $C.\text{mems}[x]$ must be defined in the context.
- Then the export description is valid.

$$\frac{C.\text{mems}[x] = \textit{memtype}}{C \vdash \text{mem } x \text{ ok}}$$

global *x*

- The global $C.\text{globals}[x]$ must be defined in the context.
- Let *mut* *t* be the *global type* $C.\text{globals}[x]$.
- The mutability *mut* must be *const*.
- Then the export description is valid.

$$\frac{C.\text{globals}[x] = \text{const } t}{C \vdash \text{global } x \text{ ok}}$$

3.3.10 Imports

Imports *import* and import descriptions *importdesc* are classified by *external types*.

{module *name*₁, name *name*₂, desc *importdesc*}

- The import description *importdesc* must be valid with type *externtype*.
- Then the import is valid with type *externtype*.

$$\frac{C \vdash \text{importdesc} : \textit{externtype}}{C \vdash \{\text{module } \textit{name}_1, \text{name } \textit{name}_2, \text{desc } \textit{importdesc}\} : \textit{externtype}}$$

func x

- The function $C.\text{types}[x]$ must be defined in the context.
- Let $[t_1^*] \rightarrow [t_2^*]$ be the *function type* $C.\text{types}[x]$.
- The length of t_2^* must not be larger than 1.
- Then the import description is valid with type func $[t_1^*] \rightarrow [t_2^*]$.

$$\frac{C.\text{types}[x] = [t_1^*] \rightarrow [t_2^*]}{C \vdash \text{func } x : \text{func } [t_1^*] \rightarrow [t_2^*]}$$

Note: The restriction on the length of the result types t_2^* may be lifted in future versions of WebAssembly.

table *limits* *elemtype*

- The limits *limits* must be valid.
- Then the import description is valid with type table *limits* *elemtype*.

$$\frac{\vdash \text{limits ok}}{C \vdash \text{table } \textit{limits elemtype} : \text{table } \textit{limits elemtype}}$$

mem *limits*

- The limits *limits* must be valid.
- Then the import description is valid with type mem *limits*.

$$\frac{\vdash \text{limits ok}}{C \vdash \text{mem } \textit{limits} : \text{mem } \textit{limits}}$$

global *mut* t

- The mutability *mut* must be *const*.
- Then the import description is valid with type global t .

$$\overline{C \vdash \text{global } \textit{const } t : \text{global } \textit{const } t}$$

3.3.11 Modules

A module is entirely *closed*, that is, it only refers to definitions that appear in the module itself. Consequently, no initial *context* is required. Instead, the context C for validation of the module's content is constructed from the types of definitions in the module itself.

- Let *module* be the module to validate.
- Let C be a *context* where:
 - $C.\text{types}$ is *module*.types,
 - $C.\text{funcs}$ is $\text{funcs}(\text{externtype}_i^*)$ concatenated with functype_i^* , with the type sequences externtype_i^* and functype_i^* as determined below,
 - $C.\text{tables}$ is $\text{tables}(\text{externtype}_i^*)$ concatenated with tabletype_i^* , with the type sequences externtype_i^* and tabletype_i^* as determined below,

- $C.\text{mems}$ is $\text{mems}(\text{externtype}_i^*)$ concatenated with memtype_i^* , with the type sequences externtype_i^* and memtype_i^* as determined below,
- $C.\text{globals}$ is $\text{globals}(\text{externtype}_i^*)$ concatenated with globaltype_i^* , with the type sequences externtype_i^* and globaltype_i^* as determined below.
- $C.\text{locals}$ is empty,
- $C.\text{labels}$ is empty.
- Under the context C :
 - For each func_i in $\text{module}.\text{funcs}$, the definition func_i must be *valid* with a *function type* functype_i .
 - For each table_i in $\text{module}.\text{tables}$, the definition table_i must be *valid* with a *table type* tabletype_i .
 - For each mem_i in $\text{module}.\text{mems}$, the definition mem_i must be *valid* with a *memory type* memtype_i .
 - For each global_i in $\text{module}.\text{globals}$:
 - * Let C_i be the *context* where $C_i.\text{globals}$ is the sequence $\text{globals}(\text{externtype}_i^*)$ concatenated with $\text{globaltype}_0 \dots \text{globaltype}_{i-1}$, and all other fields are empty.
 - * Under the context C_i , the definition global_i must be *valid* with a *global type* globaltype_i .
 - For each elem_i in $\text{module}.\text{elem}$, the segment elem_i must be *valid*.
 - For each data_i in $\text{module}.\text{data}$, the segment data_i must be *valid*.
 - If $\text{module}.\text{start}$ is non-empty, then $\text{module}.\text{start}$ must be *valid*.
 - For each import_i in $\text{module}.\text{imports}$, the segment import_i must be *valid* with an *external type* externtype_i .
 - For each export_i in $\text{module}.\text{exports}$, the segment import_i must be *valid* with a *name* name_i .
- The length of $C.\text{tables}$ must not be larger than 1.
- The length of $C.\text{mems}$ must not be larger than 1.
- All export names name_i must be different.

$$\begin{array}{c}
 (C \vdash \text{func} : \text{ft})^* \quad (C \vdash \text{table} : \text{tt})^* \quad (C \vdash \text{mem} : \text{mt})^* \quad (C_i \vdash \text{global} : \text{gt})_i^* \\
 (C \vdash \text{elem} \text{ ok})^* \quad (C \vdash \text{data} \text{ ok})^* \quad (C \vdash \text{start} \text{ ok})^? \quad (C \vdash \text{import} : \text{it})^* \quad (C \vdash \text{export} : \text{name})^* \\
 \text{ift}^* = \text{funcs}(\text{it}^*) \quad \text{itt}^* = \text{tables}(\text{it}^*) \quad \text{imt}^* = \text{mems}(\text{it}^*) \quad \text{igt}^* = \text{globals}(\text{it}^*) \\
 C = \{\text{types } \text{functype}^*, \text{funcs } \text{ift}^* \text{ ft}^*, \text{tables } \text{itt}^* \text{ tt}^*, \text{mems } \text{imt}^* \text{ mt}^*, \text{globals } \text{igt}^* \text{ gt}^*\} \\
 |C.\text{tables}| \leq 1 \quad |C.\text{mems}| \leq 1 \quad \text{name}^* \text{ disjoint} \quad (C_i = \{\text{globals } [\text{igt}^* \text{ gt}^{i-1}]\})_i^* \\
 \hline
 \vdash \{\text{types } \text{functype}^*, \text{funcs } \text{func}^*, \text{tables } \text{table}^*, \text{mems } \text{mem}^*, \text{globals } \text{global}^*, \\
 \text{elem } \text{elem}^*, \text{data } \text{data}^*, \text{start } \text{start}^?, \text{imports } \text{import}^*, \text{exports } \text{export}^*\} \text{ ok}
 \end{array}$$

Note: Most definitions in a module – particularly functions – are mutually recursive. Consequently, the definition of the *context* C in this rule is recursive: it depends on the outcome of validation of the function, table, memory, and global definitions contained in the module, which itself depends on C . However, this recursion is just a specification device. All types needed to construct C can easily be determined from a simple pre-pass over the module that does not perform any actual validation.

Globals, however, are not recursive. The effect of defining the limited contexts C_i for validating the module’s globals is that their initialization expressions can only access imported and previously defined globals and nothing else.

Note: The restriction on the number of tables and memories may be lifted in future versions of WebAssembly.

Instantiation

Execution

Binary Format

6.1 Conventions

The binary format for WebAssembly modules is a dense linear *encoding* of their *abstract syntax*.¹³

The format is defined by an *attribute grammar* whose only terminal symbols are *bytes*. A byte sequence is a well-formed encoding of a module if and only if it is generated by the grammar.

Each production of this grammar has exactly one synthesized attribute: the abstract syntax that the respective byte sequence encodes. Thus, the attribute grammar implicitly defines a *decoding* function.

Except for a few exceptions, the binary grammar closely mirrors the grammar of the abstract syntax.

Note: Some phrases of abstract syntax have multiple possible encodings in the binary format. For example, numbers may be encoded as if they had optional leading zeros. Implementations of decoders must support all possible alternatives; implementations of encoders can pick any one allowed encoding.

6.1.1 Grammar

The following conventions are adopted in defining grammar rules for the binary format. They mirror the conventions used for *abstract syntax*. In order to distinguish symbols of the binary syntax from symbols of the abstract syntax, typewriter font is adopted for the former.

- Terminal symbols are *bytes* expressed in hexadecimal notation: 0x0F.
- Nonterminal symbols are written in typewriter font: `valtype`, `instr`.
- B^n is a sequence of $n \geq 0$ iterations of B .
- B^* is a possibly empty sequence of iterations of B . (This is a shorthand for A^n used where n is not relevant.)
- $B^?$ is an optional occurrence of B . (This is a shorthand for A^n where $n \leq 1$.)
- $x:B$ denotes the same language as the nonterminal B , but also binds the variable x to the attribute synthesized for B .
- Productions are written `name ::= B ⇒ A`, where A is the attribute that is synthesized for `name`, usually from attribute variables bound in B .
- Some productions are augmented by side conditions in parentheses, which restrict the applicability of the production. They provide a shorthand for a combinatorial expansion of the production into many separate cases.

¹³ Additional encoding layers – for example, introducing compression – may be defined on top of the basic representation defined here. However, such layers are outside the scope of the current specification.

Note: For example, the *binary grammar* for *value types* is given as follows:

valtype	::=	0x7F	⇒	i32
				0x7E ⇒ i64
				0x7D ⇒ f32
				0x7C ⇒ f64

Consequently, the byte 0x7F encodes the type *i32*, 0x7E encodes the type *i64*, and so forth. No other byte value is allowed as the encoding of a value type.

The *binary grammar* for *limits* is defined as follows:

limits	::=	0x00	<i>n</i> :uint ₃₂	⇒	{min <i>n</i> , max <i>ε</i> }
			0x01	<i>n</i> :uint ₃₂ <i>m</i> :uint ₃₂	⇒ {min <i>n</i> , max <i>m</i> }

That is, a limits pair is encoded as either the byte 0x00 followed by the encoding of a *uint*₃₂ value, or the byte 0x01 followed by two such encodings. The variables *n* and *m* name the attributes of the respective *uint*₃₂ nonterminals, which in this case are the actual *unsigned integers* they decode into. The attribute of the complete production then is the abstract syntax for the limit, expressed in terms of the former values.

6.1.2 Auxiliary Notation

When dealing with binary encodings the following notation is also used:

- *ε* denotes the empty byte sequence.
- $\|B\|$ is the length of the byte sequence generated from the production *B* in a derivation.

6.2 Values

6.2.1 Bytes

Bytes encode themselves.

byte	::=	0x00	⇒	0x00
				...
				0xFF ⇒ 0xFF

6.2.2 Integers

All *integers* are encoded using the *LEB128*¹⁴ variable-length integer encoding, in either unsigned or signed variant.

Unsigned integers are encoded in *unsigned LEB128*¹⁵ format. As an additional constraint, the total number of bytes encoding a value of type *uint*_{*N*} must not exceed $\text{ceil}(N/7)$ bytes.

uint _{<i>N</i>}	::=	<i>n</i> :byte	⇒	<i>n</i>	$(n < 2^7 \wedge n < 2^N)$
			<i>n</i> :byte <i>m</i> :uint _{<i>N-7</i>}	⇒ $2^7 \cdot m + (n - 2^7)$	$(n \geq 2^7 \wedge N > 7)$

Signed integers are encoded in *signed LEB128*¹⁶ 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 $\text{ceil}(N/7)$ bytes.

sint _{<i>N</i>}	::=	<i>n</i> :byte	⇒	<i>n</i>	$(n < 2^6 \wedge n < 2^{N-1})$
			<i>n</i> :byte	⇒ $n - 2^7$	$(2^6 \leq n < 2^7 \wedge n \geq 2^7 - 2^{N-1})$
			<i>n</i> :byte <i>m</i> :sint _{<i>N-7</i>}	⇒ $2^7 \cdot m + (n - 2^7)$	$(n \geq 2^7 \wedge N > 7)$

¹⁴ <https://en.wikipedia.org/wiki/LEB128>

¹⁵ https://en.wikipedia.org/wiki/LEB128#Unsigned_LEB128

¹⁶ https://en.wikipedia.org/wiki/LEB128#Signed_LEB128

Uninterpreted integers are always encoded as signed integers.

$$\text{int}_N ::= n:\text{int}_N \Rightarrow n$$

Note: While the side conditions $N > 7$ in the productions for *non-terminating* bytes restrict the length of the *uint* and *sint* encodings, “trailing zeros” are still allowed within these bounds. For example, 0x03 and 0x83 0x00 are both well-formed encodings for the value 3 as a *uint₈*. Similarly, either of 0x7e and 0xFE 0x7F and 0xFE 0xFF 0x7F are well-formed encodings of the value -2 as a *sint₁₆*.

The side conditions on the value n of *terminating* bytes further enforce that any unused bits in these bytes must be 0 for positive values and 1 for negative ones. For example, 0x83 0x10 is malformed as a *uint₈* encoding. Similarly, both 0x83 0x3E and 0xFF 0x7B are malformed as *sint₈* encodings.

6.2.3 Floating-Point

Floating point values are encoded directly by their IEEE bit pattern in *little endian*¹⁷ byte order:

$$\text{float}_N ::= b*:\text{byte}^{N/8} \Rightarrow \text{reverse}(b*)$$

Here, $\text{reverse}(b*)$ denotes the byte sequence $b*$ in reversed order.

6.2.4 Vectors

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

$$\text{vec}(B) ::= n:\text{uint}_{32} (x:B)^n \Rightarrow x^n$$

6.2.5 Names

Names are encoded like a vector of bytes containing the UTF-8¹⁸ encoding of the name’s code point sequence.

<code>name</code>	$::= n:\text{uint}_{32} (uc:\text{codepoint})^*$	$\Rightarrow uc^*$	(codepoint^*)
<code>codepoint</code>	$::= uv:\text{codeval}_N$	$\Rightarrow uv$	$(uv \geq N \wedge (uv$
<code>codeval_N</code>	$::= b_1:\text{byte}$	$\Rightarrow b_1$	$(b_1 < 0x80 \wedge N$
	$\quad \quad b_1:\text{byte} \quad b_2:\text{cont}$	$\Rightarrow 2^6 \cdot (b_1 - 0xc0) + b_2$	$(0xc0 \leq b_1 < 0$
	$\quad \quad b_1:\text{byte} \quad b_2:\text{cont} \quad b_3:\text{cont}$	$\Rightarrow 2^{12} \cdot (b_1 - 0xe0) + 2^6 \cdot b_2 + b_3$	$(0xe0 \leq b_1 < 0$
	$\quad \quad b_1:\text{byte} \quad b_2:\text{cont} \quad b_3:\text{cont} \quad b_4:\text{cont}$	$\Rightarrow 2^{18} \cdot (b_1 - 0xf0) + 2^{12} \cdot b_2 + 2^6 \cdot b_3 + b_4$	$(0xf0 \leq b_1 < 0$
<code>cont</code>	$::= b:\text{byte}$	$\Rightarrow b - 0x80$	$(b \geq 0x80)$

Note: The *size*, $|\text{codepoint}^*|$ denotes the number of bytes in the encoding of the sequence, not the number of code points.

The index N to `codeval` is the minimum value that a given byte sequence may decode into. The respective side conditions on it exclude encodings using more than the minimal number of bytes to represent a code point.

¹⁷ <https://en.wikipedia.org/wiki/Endianness#Little-endian>

¹⁸ <http://www.unicode.org/versions/latest/>

6.3 Types

6.3.1 Value Types

Value types are encoded by a single byte.

<code>valtype</code>	<code>::=</code>	<code>0x7F</code>	\Rightarrow	<code>i32</code>	
		<code> </code>	<code>0x7E</code>	\Rightarrow	<code>i64</code>
		<code> </code>	<code>0x7D</code>	\Rightarrow	<code>f32</code>
		<code> </code>	<code>0x7C</code>	\Rightarrow	<code>f64</code>

Note: In future versions of WebAssembly, value types may include types denoted by *type indices*. Thus, the binary format for types corresponds to the encodings of small negative *sint* values, so that they can coexist with (positive) type indices.

6.3.2 Result Types

The only *result types* occurring in the binary format are the types of blocks. These are encoded in special compressed form, by either the byte 0x40 indicating the empty type or as a single *value type*.

<code>blocktype</code>	<code>::=</code>	<code>0x40</code>	\Rightarrow	<code>[]</code>	
		<code> </code>	<code>t:valtype</code>	\Rightarrow	<code>[t]</code>

Note: In future versions of WebAssembly, this scheme may be extended to support multiple results or more general block types.

6.3.3 Function Types

Function types are encoded by the byte 0x60 followed by the respective *vectors* of parameter and result types.

$$\text{functype} ::= 0x60 \ t_1^*:\text{vec}(\text{valtype}) \ t_2^*:\text{vec}(\text{valtype}) \Rightarrow [t_1^*] \rightarrow [t_2^*]$$

6.3.4 Limits

Limits are encoded with a preceding flag indicating whether a maximum is present.

<code>limits</code>	<code>::=</code>	<code>0x00</code>	<code>n:uint₃₂</code>	\Rightarrow	<code>{min n, max ϵ}</code>	
		<code> </code>	<code>0x01</code>	<code>n:uint₃₂ m:uint₃₂</code>	\Rightarrow	<code>{min n, max m}</code>

6.3.5 Memory Types

Memory types are encoded with their *limits*.

$$\text{memtype} ::= \text{lim}:\text{limits} \Rightarrow \text{lim}$$

6.3.6 Table Types

Table types are encoded with their *limits* and a constant byte indicating their *element type*.

```
tabletype ::= et:elemtype lim:limits ⇒ lim et
elemtype  ::= 0x70 ⇒ anyfunc
```

6.3.7 Global Types

Global types are encoded by their *value type* and a flag for their *mutability*.

```
globaltype ::= t:valtype m:mut ⇒ m t
mut        ::= 0x00 ⇒ const
           | 0x01 ⇒ mut
```

6.4 Instructions

Instructions are encoded by *opcodes*. Each opcode is represented by a single byte, and is followed by the instruction's immediate arguments, where present. The only exception are *structured control instructions*, which consist of several opcodes bracketing their nested instruction sequences.

Note: Gaps in the byte code ranges encoding instructions are reserved for future extensions.

6.4.1 Control Instructions

Control instructions have varying encodings. For structured instructions, the nested instruction sequences are terminated with explicit opcodes for *end* and *else*.

```
instr ::= 0x00 ⇒ unreachable
       | 0x01 ⇒ nop
       | 0x02 rt:blocktype (in:instr)* 0x0B ⇒ block rt in* end
       | 0x03 rt:blocktype (in:instr)* 0x0B ⇒ loop rt in* end
       | 0x04 rt:blocktype (in:instr)* 0x0B ⇒ if rt in* else ε end
       | 0x04 rt:blocktype (in1:instr)* 0x05 (in2:instr)* 0x0B ⇒ if rt in1* else in2* end
       | 0x0C l:labelidx ⇒ br l
       | 0x0D l:labelidx ⇒ br_if l
       | 0x0E l*:vec(labelidx) lN:labelidx ⇒ br_table l* lN
       | 0x0F ⇒ return
       | 0x10 x:funcidx ⇒ call x
       | 0x11 x:typeidx ⇒ call_indirect x
```

Note: The *else* opcode 0x05 in the encoding of an *if* instruction can be omitted if the following instruction sequence is empty.

6.4.2 Parametric Instructions

Parametric instructions are represented by single byte codes.

```
instr ::= ...
      | 0x1A ⇒ drop
      | 0x1B ⇒ select
```

6.4.3 Variable Instructions

Variable instructions are represented by byte codes followed by the encoding of the respective index.

```
instr ::= ...
      | 0x20 x:localidx ⇒ get_local x
      | 0x21 x:localidx ⇒ set_local x
      | 0x22 x:localidx ⇒ tee_local x
      | 0x23 x:globalidx ⇒ get_global x
      | 0x24 x:globalidx ⇒ set_global x
```

6.4.4 Memory Instructions

Each variant of *memory instruction* is encoded with a different byte code. Loads and stores are followed by the encoding of their *memarg* immediate.

```
memarg ::= a:uint32 o:uint32 ⇒ {align a, offset o}
instr  ::= ...
      | 0x28 m:memarg ⇒ i32.load m
      | 0x29 m:memarg ⇒ i64.load m
      | 0x2A m:memarg ⇒ f32.load m
      | 0x2B m:memarg ⇒ f64.load m
      | 0x2C m:memarg ⇒ i32.load8_s m
      | 0x2D m:memarg ⇒ i32.load8_u m
      | 0x2E m:memarg ⇒ i32.load16_s m
      | 0x2F m:memarg ⇒ i32.load16_u m
      | 0x30 m:memarg ⇒ i64.load8_s m
      | 0x31 m:memarg ⇒ i64.load8_u m
      | 0x32 m:memarg ⇒ i64.load16_s m
      | 0x33 m:memarg ⇒ i64.load16_u m
      | 0x34 m:memarg ⇒ i64.load32_s m
      | 0x35 m:memarg ⇒ i64.load32_u m
      | 0x36 m:memarg ⇒ i32.store m
      | 0x37 m:memarg ⇒ i64.store m
      | 0x38 m:memarg ⇒ f32.store m
      | 0x39 m:memarg ⇒ f64.store m
      | 0x3A m:memarg ⇒ i32.store8 m
      | 0x3B m:memarg ⇒ i32.store16 m
      | 0x3C m:memarg ⇒ i64.store8 m
      | 0x3D m:memarg ⇒ i64.store16 m
      | 0x3E m:memarg ⇒ i64.store32 m
      | 0x3F 0x00 ⇒ current_memory
      | 0x40 0x00 ⇒ grow_memory
```

Note: In future versions of WebAssembly, the additional zero bytes occurring in the encoding of the `current_memory` and `grow_memory` instructions may be used to index additional memories.

6.4.5 Numeric Instructions

All variants of *numeric instructions* are represented by separate byte codes.

The `const` instructions are followed by the respective literal.

```
instr ::= ...
      | 0x41 n:int32    ⇒ i32.const n
      | 0x42 n:int64    ⇒ i64.const n
      | 0x43 z:float32 ⇒ f32.const z
      | 0x44 z:float64 ⇒ f64.const z
```

All other numeric instructions are plain opcodes without any immediates.

```
instr ::= ...
      | 0x45 ⇒ i32.eqz
      | 0x46 ⇒ i32.eq
      | 0x47 ⇒ i32.ne
      | 0x48 ⇒ i32.lt_s
      | 0x49 ⇒ i32.lt_u
      | 0x4A ⇒ i32.gt_s
      | 0x4B ⇒ i32.gt_u
      | 0x4C ⇒ i32.le_s
      | 0x4D ⇒ i32.le_u
      | 0x4E ⇒ i32.ge_s
      | 0x4F ⇒ i32.ge_u

      | 0x50 ⇒ i64.eqz
      | 0x51 ⇒ i64.eq
      | 0x52 ⇒ i64.ne
      | 0x53 ⇒ i64.lt_s
      | 0x54 ⇒ i64.lt_u
      | 0x55 ⇒ i64.gt_s
      | 0x56 ⇒ i64.gt_u
      | 0x57 ⇒ i64.le_s
      | 0x58 ⇒ i64.le_u
      | 0x59 ⇒ i64.ge_s
      | 0x5A ⇒ i64.ge_u

      | 0x5B ⇒ f32.eq
      | 0x5C ⇒ f32.ne
      | 0x5D ⇒ f32.lt
      | 0x5E ⇒ f32.gt
      | 0x5F ⇒ f32.le
      | 0x60 ⇒ f32.ge

      | 0x61 ⇒ f64.eq
      | 0x62 ⇒ f64.ne
      | 0x63 ⇒ f64.lt
      | 0x64 ⇒ f64.gt
      | 0x65 ⇒ f64.le
      | 0x66 ⇒ f64.ge
```

	0x67	⇒	i32.clz
	0x68	⇒	i32.ctz
	0x69	⇒	i32.popcnt
	0x6A	⇒	i32.add
	0x6B	⇒	i32.sub
	0x6C	⇒	i32.mul
	0x6D	⇒	i32.div_s
	0x6E	⇒	i32.div_u
	0x6F	⇒	i32.rem_s
	0x70	⇒	i32.rem_u
	0x71	⇒	i32.and
	0x72	⇒	i32.or
	0x73	⇒	i32.xor
	0x74	⇒	i32.shl
	0x75	⇒	i32.shr_s
	0x76	⇒	i32.shr_u
	0x77	⇒	i32.rotl
	0x78	⇒	i32.rotr
	0x79	⇒	i64.clz
	0x7A	⇒	i64.ctz
	0x7B	⇒	i64.popcnt
	0x7C	⇒	i64.add
	0x7D	⇒	i64.sub
	0x7E	⇒	i64.mul
	0x7F	⇒	i64.div_s
	0x80	⇒	i64.div_u
	0x81	⇒	i64.rem_s
	0x82	⇒	i64.rem_u
	0x83	⇒	i64.and
	0x84	⇒	i64.or
	0x85	⇒	i64.xor
	0x86	⇒	i64.shl
	0x87	⇒	i64.shr_s
	0x88	⇒	i64.shr_u
	0x89	⇒	i64.rotl
	0x8A	⇒	i64.rotr
	0x8B	⇒	f32.abs
	0x8C	⇒	f32.neg
	0x8D	⇒	f32.ceil
	0x8E	⇒	f32.floor
	0x8F	⇒	f32.trunc
	0x90	⇒	f32.nearest
	0x91	⇒	f32.sqrt
	0x92	⇒	f32.add
	0x93	⇒	f32.sub
	0x94	⇒	f32.mul
	0x95	⇒	f32.div
	0x96	⇒	f32.min
	0x97	⇒	f32.max
	0x98	⇒	f32.copysign

0x99	⇒	f64.abs
0x9A	⇒	f64.neg
0x9B	⇒	f64.ceil
0x9C	⇒	f64.floor
0x9D	⇒	f64.trunc
0x9E	⇒	f64.nearest
0x9F	⇒	f64.sqrt
0xA0	⇒	f64.add
0xA1	⇒	f64.sub
0xA2	⇒	f64.mul
0xA3	⇒	f64.div
0xA4	⇒	f64.min
0xA5	⇒	f64.max
0xA6	⇒	f64.copysign
0xA7	⇒	i32.wrap/i64
0xA8	⇒	i32.trunc_s/f32
0xA9	⇒	i32.trunc_u/f32
0xAA	⇒	i32.trunc_s/f64
0xAB	⇒	i32.trunc_u/f64
0xAC	⇒	i64.extend_s/i32
0xAD	⇒	i64.extend_u/i32
0xAE	⇒	i64.trunc_s/f32
0xAF	⇒	i64.trunc_u/f32
0xB0	⇒	i64.trunc_s/f64
0xB1	⇒	i64.trunc_u/f64
0xB2	⇒	f32.convert_s/i32
0xB3	⇒	f32.convert_u/i32
0xB4	⇒	f32.convert_s/i64
0xB5	⇒	f32.convert_u/i64
0xB6	⇒	f32.demote/f64
0xB7	⇒	f64.convert_s/i32
0xB8	⇒	f64.convert_u/i32
0xB9	⇒	f64.convert_s/i64
0xBA	⇒	f64.convert_u/i64
0xBB	⇒	f64.promote/f32
0xBC	⇒	i32.reinterpret/f32
0xBD	⇒	i64.reinterpret/f64
0xBE	⇒	f32.reinterpret/i32
0xBF	⇒	f64.reinterpret/i64

6.4.6 Expressions

Expressions are encoded by their instruction sequence terminated with an explicit 0x0B opcode for *end*.

`expr ::= (in:instr)* 0x0B ⇒ in* end`

6.5 Modules

The binary encoding of modules is organized into *sections*. Most sections correspond to one component of a *module* record, except that *function definitions* are split into two sections, separating their type declarations in the *function section* from their bodies in the *code section*.

Note: This separation enables *parallel* and *streaming* compilation of the functions in a module.

6.5.1 Indices

All *indices* are encoded with their respective *uint₃₂* value.

<code>typeidx</code>	<code>::=</code>	<code>x:uint₃₂</code>	<code>⇒</code>	<code>x</code>
<code>funcidx</code>	<code>::=</code>	<code>x:uint₃₂</code>	<code>⇒</code>	<code>x</code>
<code>tableidx</code>	<code>::=</code>	<code>x:uint₃₂</code>	<code>⇒</code>	<code>x</code>
<code>memidx</code>	<code>::=</code>	<code>x:uint₃₂</code>	<code>⇒</code>	<code>x</code>
<code>globalidx</code>	<code>::=</code>	<code>x:uint₃₂</code>	<code>⇒</code>	<code>x</code>
<code>localidx</code>	<code>::=</code>	<code>x:uint₃₂</code>	<code>⇒</code>	<code>x</code>
<code>labelidx</code>	<code>::=</code>	<code>l:uint₃₂</code>	<code>⇒</code>	<code>l</code>

6.5.2 Sections

Each section consists of

- a one-byte section *id*,
- the *uint₃₂* *size* of the contents in bytes,
- the actual *contents*, whose structure is depended on the section id.

Every section is optional; an omitted section is equivalent to the section being present with empty contents.

The following parameterized grammar rule defines the generic structure of a section with id *N* and contents described by the grammar B.

$$\begin{array}{lcl} \text{section}_N(B) & ::= & N:\text{byte} \quad \text{size}:\text{uint}_{32} \quad \text{cont}:B \Rightarrow \text{cont} \quad (size = ||B||) \\ & | & \epsilon \qquad \qquad \qquad \qquad \qquad \qquad \qquad \qquad \Rightarrow \epsilon \end{array}$$

For most sections, the contents B encodes a *vector*. In these cases, the empty result ϵ is interpreted as the empty vector.

Note: Other than for unknown *custom sections*, the *size* is not required for decoding, but can be used to skip sections when navigating through a binary. The module is malformed if the size does not match the length of the binary contents B.

6.5.3 Custom Section

Custom sections have the id 0. They are intended to be used for debugging information or third-party extensions, and are ignored by the WebAssembly semantics. Their contents consist of a *name* further identifying the custom section, followed by an uninterpreted sequence of bytes for custom use.

<code>customsec</code>	<code>::=</code>	<code>section₀(custom)</code>
<code>custom</code>	<code>::=</code>	<code>name byte*</code>

Note: If an implementation interprets the contents of a custom section, then errors in that contents, or the placement of the section, must not invalidate the module.

6.5.4 Type Section

The *type section* has the id 1. It decodes into a vector of *function types* that represent the types component of a *module*.

$$\text{typesec} ::= ft^*:\text{section}_1(\text{functype}^*) \Rightarrow ft^*$$

6.5.5 Import Section

The *import section* has the id 2. It decodes into a vector of *imports* that represent the imports component of a *module*.

<code>importsec</code>	<code>::=</code>	<code>im*:section₂(import*)</code>	\Rightarrow	<code>im*</code>
<code>import</code>	<code>::=</code>	<code>mod:name nm:name d:importdesc</code>	\Rightarrow	<code>{module mod, name nm, desc d}</code>
<code>importdesc</code>	<code>::=</code>	<code>0x00 x:typeidx</code>	\Rightarrow	<code>func x</code>
		<code> 0x01 tt:tabletype</code>	\Rightarrow	<code>table tt</code>
		<code> 0x02 mt:memtype</code>	\Rightarrow	<code>mem mt</code>
		<code> 0x03 gt:globaltype</code>	\Rightarrow	<code>global gt</code>

6.5.6 Function Section

The *function section* has the id 3. It decodes into a vector of *type indices* that represent the type fields of the *functions* in the *funcs* component of a *module*. The *locals* and *body* fields of the respective functions are encoded separately in the *code section*.

$$\text{funcsec} ::= x^*:\text{section}_3(\text{typeidx}^*) \Rightarrow x^*$$

6.5.7 Table Section

The *table section* has the id 4. It decodes into a vector of *tables* that represent the tables component of a *module*.

<code>tablesec</code>	<code>::=</code>	<code>tab*:section₄(table*)</code>	\Rightarrow	<code>tab*</code>
<code>table</code>	<code>::=</code>	<code>tt:tabletype</code>	\Rightarrow	<code>{type tt}</code>

6.5.8 Memory Section

The *memory section* has the id 5. It decodes into a vector of *memories* that represent the *mems* component of a *module*.

<code>memsec</code>	<code>::=</code>	<code>mem*:section₅(mem*)</code>	\Rightarrow	<code>mem*</code>
<code>mem</code>	<code>::=</code>	<code>mt:memtype</code>	\Rightarrow	<code>{type mt}</code>

6.5.9 Global Section

The *global section* has the id 6. It decodes into a vector of *globals* that represent the *globals* component of a *module*.

<code>globalsec</code>	<code>::=</code>	<code>glob*:section₆(global*)</code>	\Rightarrow	<code>glob*</code>
<code>global</code>	<code>::=</code>	<code>gt:globaltype e:expr</code>	\Rightarrow	<code>{type gt, init e}</code>

6.5.10 Export Section

The *export section* has the id 7. It decodes into a vector of *exports* that represent the exports component of a *module*.

```

exportsec ::= ex*:section7(export*) ⇒ ex*
export    ::= nm:name d:exportdesc ⇒ {name nm, desc d}
exportdesc ::= 0x00 x:funcidx      ⇒ func x
            | 0x01 x:tableidx     ⇒ table x
            | 0x02 x:memidx       ⇒ mem x
            | 0x03 x:globalidx    ⇒ global x

```

6.5.11 Start Section

The *start section* has the id 8. It decodes into an optional *start function* that represents the start component of a *module*.

```

startsec ::= st?:section8(start) ⇒ st?
start    ::= x:funcidx           ⇒ {func x}

```

6.5.12 Element Section

The *element section* has the id 9. It decodes into a vector of *element segments* that represent the elem component of a *module*.

```

elemsec ::= seg*:section9(elem*) ⇒ seg
elem    ::= x:tableidx e:expr y*:vec(funcidx) ⇒ {table x, offset e, init y*}

```

6.5.13 Code Section

The *code section* has the id 10. It decodes into a vector of *code* entries that are pairs of *value type* vectors and *expressions*. They represent the locals and body field of the *functions* in the funcs component of a *module*. The type fields of the respective functions are encoded separately in the *function section*.

The encoding of each code entry consists of

- the *uint₃₂* *size* of the function code in bytes,
- the actual *function code*, which in turn consists of
 - the declaration of *locals*,
 - the function *body* as an *expression*.

Local declarations are compressed into a vector whose entries consist of

- a *uint₃₂* *count*,
- a *value type*,

denoting *count* locals of the same value type.

```

codesec ::= code*:section10(code*) ⇒ code*
code    ::= size:uint32 code:func ⇒ code (size = ||func||)
func    ::= (t*)*:vec(locals) e:expr ⇒ concat((t*)*), e* (|concat((t*)*)| < 232)
locals  ::= n:uint32 t:valtype ⇒ tn

```

Here, *code* ranges over pairs $(valtype^*, expr)$. The meta function $\text{concat}((t^*)^*)$ denotes the sequence of types formed by concatenating all sequences t_i^* in $(t^*)^*$. Any code for which the length of the resulting sequence is out of bounds of the maximum size of a *vector* is malformed.

Note: The *size* is not needed for decoding, but like with *sections*, can be used to skip functions when navigating through a binary. The module is malformed if a size does not match the length of the respective function code.

6.5.14 Data Section

The *data section* has the id 11. It decodes into a vector of *data segments* that represent the data component of a *module*.

$$\begin{aligned} \text{datasec} &::= \text{seg}^*:\text{section}_{11}(\text{data}^*) && \Rightarrow \text{seg} \\ \text{data} &::= x:\text{memidx } e:\text{expr } b^*:\text{vec}(\text{byte}) && \Rightarrow \{\text{mem } x, \text{offset } e, \text{init } b^*\} \end{aligned}$$

6.5.15 Modules

The encoding of a *module* starts with a preamble containing a 4-byte magic number and a version field. The current version of the WebAssembly binary format is 1.

The preamble is followed by a sequence of *sections*. *Custom sections* may be inserted at any place in this sequence, while other sections must occur at most once and in the prescribed order. All sections can be empty. The lengths

of vectors produced by the (possibly empty) *function* and *code* section must match up.

```

module ::= 0x00 0x61 0x73 0x6D
          0x01 0x00 0x00 0x00
          customsec*
          functype*:typesec
          customsec*
          import*:importsec
          customsec*
          typeidxn:funcsec
          customsec*
          table*:tablesec
          customsec*
          mem*:memsec
          customsec*
          global*:globalsec
          customsec*
          export*:exportsec
          customsec*
          start?:startsec
          customsec*
          elem*:elemsec
          customsec*
          coden:codesec
          customsec*
          data*:datasec
          customsec*

```

⇒ { types *functype*^{*},
 funcs *func*ⁿ,
 tables *table*^{*},
 mems *mem*^{*},
 globals *global*^{*},
 elem *elem*^{*},
 data *data*^{*},
 start *start*[?],
 imports *import*^{*},
 exports *export*^{*} }

where for each (t_i^*, e_i) in $code^n$, $func^n[i] = \{\text{type } typeidx^n[i], \text{locals } t_i^*, \text{body } e_i\}$.

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

Appendix: Formal Properties

Todo

Describe and sketch proof (progress, preservation, uniqueness)

Appendix: Validation Algorithm

Todo

Describe algorithm, state correctness properties (soundness, completeness)

Appendix: Text Format

Todo

Describe

Appendix: Name Section

Todo

Describe

Index of Instructions

Instruction	Opcode	Type	Validation	Execution
unreachable	0x00	$[t_1^*] \rightarrow [t_2^*]$	<i>validation</i>	
nop	0x01	$[] \rightarrow []$	<i>validation</i>	
block $[t^?]$	0x02	$[] \rightarrow [t^*]$	<i>validation</i>	
loop $[t^?]$	0x03	$[] \rightarrow [t^*]$	<i>validation</i>	
if $[t^?]$	0x04	$[] \rightarrow [t^*]$	<i>validation</i>	
else	0x05			
(reserved)	0x06			
(reserved)	0x07			
(reserved)	0x08			
(reserved)	0x09			
(reserved)	0x0A			
end	0x0B			
br l	0x0C	$[t_1^* t^?] \rightarrow [t_2^*]$	<i>validation</i>	
br_if l	0x0D	$[t^? i32] \rightarrow [t^?]$	<i>validation</i>	
br_table $l^* l$	0x0E	$[t_1^* t^? i32] \rightarrow [t_2^*]$	<i>validation</i>	
return	0x0F	$[t_1^* t^?] \rightarrow [t_2^*]$	<i>validation</i>	
call x	0x10	$[t_1^*] \rightarrow [t_2^*]$	<i>validation</i>	
call_indirect x	0x11	$[t_1^* i32] \rightarrow [t_2^*]$	<i>validation</i>	
(reserved)	0x12			
(reserved)	0x13			
(reserved)	0x14			
(reserved)	0x15			
(reserved)	0x16			
(reserved)	0x17			
(reserved)	0x18			
(reserved)	0x19			
drop	0x1A	$[t] \rightarrow []$	<i>validation</i>	
select	0x1B	$[t t i32] \rightarrow [t]$	<i>validation</i>	
(reserved)	0x1C			
(reserved)	0x1D			
(reserved)	0x1E			
(reserved)	0x1F			
get_local x	0x20	$[] \rightarrow [t]$	<i>validation</i>	
set_local x	0x21	$[t] \rightarrow []$	<i>validation</i>	
tee_local x	0x22	$[t] \rightarrow [t]$	<i>validation</i>	
get_global x	0x23	$[] \rightarrow [t]$	<i>validation</i>	
set_global x	0x24	$[t] \rightarrow []$	<i>validation</i>	
(reserved)	0x25			
(reserved)	0x26			
(reserved)	0x27			

Continued on next page

Table 11.1 – continued from previous page

Instruction	Opcode	Type	Validation	Execution
i32.load <i>memarg</i>	0x28	[i32] → [i32]	<i>validation</i>	
i64.load <i>memarg</i>	0x29	[i32] → [i64]	<i>validation</i>	
f32.load <i>memarg</i>	0x2A	[i32] → [f32]	<i>validation</i>	
f64.load <i>memarg</i>	0x2B	[i32] → [f64]	<i>validation</i>	
i32.load8_s <i>memarg</i>	0x2C	[i32] → [i32]	<i>validation</i>	
i32.load8_u <i>memarg</i>	0x2D	[i32] → [i32]	<i>validation</i>	
i32.load16_s <i>memarg</i>	0x2E	[i32] → [i32]	<i>validation</i>	
i32.load16_u <i>memarg</i>	0x2F	[i32] → [i32]	<i>validation</i>	
i64.load8_s <i>memarg</i>	0x30	[i32] → [i64]	<i>validation</i>	
i64.load8_u <i>memarg</i>	0x31	[i32] → [i64]	<i>validation</i>	
i64.load16_s <i>memarg</i>	0x32	[i32] → [i64]	<i>validation</i>	
i64.load16_u <i>memarg</i>	0x33	[i32] → [i64]	<i>validation</i>	
i64.load32_s <i>memarg</i>	0x34	[i32] → [i64]	<i>validation</i>	
i64.load32_u <i>memarg</i>	0x35	[i32] → [i64]	<i>validation</i>	
i32.store <i>memarg</i>	0x36	[i32 i32] → []	<i>validation</i>	
i64.store <i>memarg</i>	0x37	[i32 i64] → []	<i>validation</i>	
f32.store <i>memarg</i>	0x38	[i32 f32] → []	<i>validation</i>	
f64.store <i>memarg</i>	0x39	[i32 f64] → []	<i>validation</i>	
i32.store8 <i>memarg</i>	0x3A	[i32 i32] → []	<i>validation</i>	
i32.store16 <i>memarg</i>	0x3B	[i32 i32] → []	<i>validation</i>	
i64.store8 <i>memarg</i>	0x3C	[i32 i64] → []	<i>validation</i>	
i64.store16 <i>memarg</i>	0x3D	[i32 i64] → []	<i>validation</i>	
i64.store32 <i>memarg</i>	0x3E	[i32 i64] → []	<i>validation</i>	
current_memory	0x3F	[] → [i32]	<i>validation</i>	
grow_memory	0x40	[i32] → [i32]	<i>validation</i>	
i32.const <i>int</i> ₃₂	0x41	[] → [i32]	<i>validation</i>	
i64.const <i>int</i> ₆₄	0x42	[] → [i64]	<i>validation</i>	
f32.const <i>float</i> ₃₂	0x43	[] → [f32]	<i>validation</i>	
f64.const <i>float</i> ₆₄	0x44	[] → [f64]	<i>validation</i>	
i32.eqz	0x45	[i32] → [i32]	<i>validation</i>	
i32.eq	0x46	[i32 i32] → [i32]	<i>validation</i>	
i32.ne	0x47	[i32 i32] → [i32]	<i>validation</i>	
i32.lt_s	0x48	[i32 i32] → [i32]	<i>validation</i>	
i32.lt_u	0x49	[i32 i32] → [i32]	<i>validation</i>	
i32.gt_s	0x4A	[i32 i32] → [i32]	<i>validation</i>	
i32.gt_u	0x4B	[i32 i32] → [i32]	<i>validation</i>	
i32.le_s	0x4C	[i32 i32] → [i32]	<i>validation</i>	
i32.le_u	0x4D	[i32 i32] → [i32]	<i>validation</i>	
i32.ge_s	0x4E	[i32 i32] → [i32]	<i>validation</i>	
i32.ge_u	0x4F	[i32 i32] → [i32]	<i>validation</i>	
i64.eqz	0x50	[i64] → [i32]	<i>validation</i>	
i64.eq	0x51	[i64 i64] → [i32]	<i>validation</i>	
i64.ne	0x52	[i64 i64] → [i32]	<i>validation</i>	
i64.lt_s	0x53	[i64 i64] → [i32]	<i>validation</i>	
i64.lt_u	0x54	[i64 i64] → [i32]	<i>validation</i>	
i64.gt_s	0x55	[i64 i64] → [i32]	<i>validation</i>	
i64.gt_u	0x56	[i64 i64] → [i32]	<i>validation</i>	
i64.le_s	0x57	[i64 i64] → [i32]	<i>validation</i>	
i64.le_u	0x58	[i64 i64] → [i32]	<i>validation</i>	
i64.ge_s	0x59	[i64 i64] → [i32]	<i>validation</i>	
i64.ge_u	0x5A	[i64 i64] → [i32]	<i>validation</i>	
f32.eq	0x5B	[f32 f32] → [i32]	<i>validation</i>	
f32.ne	0x5C	[f32 f32] → [i32]	<i>validation</i>	

Continued on next page

Table 11.1 – continued from previous page

Instruction	Opcode	Type	Validation	Execution
f32.lt	0x5D	[f32 f32] → [i32]	<i>validation</i>	
f32.gt	0x5E	[f32 f32] → [i32]	<i>validation</i>	
f32.le	0x5F	[f32 f32] → [i32]	<i>validation</i>	
f32.ge	0x60	[f32 f32] → [i32]	<i>validation</i>	
f64.eq	0x61	[f64 f64] → [i32]	<i>validation</i>	
f64.ne	0x62	[f64 f64] → [i32]	<i>validation</i>	
f64.lt	0x63	[f64 f64] → [i32]	<i>validation</i>	
f64.gt	0x64	[f64 f64] → [i32]	<i>validation</i>	
f64.le	0x65	[f64 f64] → [i32]	<i>validation</i>	
f64.ge	0x66	[f64 f64] → [i32]	<i>validation</i>	
i32.clz	0x67	[i32] → [i32]	<i>validation</i>	
i32.ctz	0x68	[i32] → [i32]	<i>validation</i>	
i32.popcnt	0x69	[i32] → [i32]	<i>validation</i>	
i32.add	0x6A	[i32 i32] → [i32]	<i>validation</i>	
i32.sub	0x6B	[i32 i32] → [i32]	<i>validation</i>	
i32.mul	0x6C	[i32 i32] → [i32]	<i>validation</i>	
i32.div_s	0x6D	[i32 i32] → [i32]	<i>validation</i>	
i32.div_u	0x6E	[i32 i32] → [i32]	<i>validation</i>	
i32.rem_s	0x6F	[i32 i32] → [i32]	<i>validation</i>	
i32.rem_u	0x70	[i32 i32] → [i32]	<i>validation</i>	
i32.and	0x71	[i32 i32] → [i32]	<i>validation</i>	
i32.or	0x72	[i32 i32] → [i32]	<i>validation</i>	
i32.xor	0x73	[i32 i32] → [i32]	<i>validation</i>	
i32.shl	0x74	[i32 i32] → [i32]	<i>validation</i>	
i32.shr_s	0x75	[i32 i32] → [i32]	<i>validation</i>	
i32.shr_u	0x76	[i32 i32] → [i32]	<i>validation</i>	
i32.rotl	0x77	[i32 i32] → [i32]	<i>validation</i>	
i32.rotr	0x78	[i32 i32] → [i32]	<i>validation</i>	
i64.clz	0x79	[i64] → [i64]	<i>validation</i>	
i64.ctz	0x7A	[i64] → [i64]	<i>validation</i>	
i64.popcnt	0x7B	[i64] → [i64]	<i>validation</i>	
i64.add	0x7C	[i64 i64] → [i64]	<i>validation</i>	
i64.sub	0x7D	[i64 i64] → [i64]	<i>validation</i>	
i64.mul	0x7E	[i64 i64] → [i64]	<i>validation</i>	
i64.div_s	0x7F	[i64 i64] → [i64]	<i>validation</i>	
i64.div_u	0x80	[i64 i64] → [i64]	<i>validation</i>	
i64.rem_s	0x81	[i64 i64] → [i64]	<i>validation</i>	
i64.rem_u	0x82	[i64 i64] → [i64]	<i>validation</i>	
i64.and	0x83	[i64 i64] → [i64]	<i>validation</i>	
i64.or	0x84	[i64 i64] → [i64]	<i>validation</i>	
i64.xor	0x85	[i64 i64] → [i64]	<i>validation</i>	
i64.shl	0x86	[i64 i64] → [i64]	<i>validation</i>	
i64.shr_s	0x87	[i64 i64] → [i64]	<i>validation</i>	
i64.shr_u	0x88	[i64 i64] → [i64]	<i>validation</i>	
i64.rotl	0x89	[i64 i64] → [i64]	<i>validation</i>	
i64.rotr	0x8A	[i64 i64] → [i64]	<i>validation</i>	
f32.abs	0x8B	[f32] → [f32]	<i>validation</i>	
f32.neg	0x8C	[f32] → [f32]	<i>validation</i>	
f32.ceil	0x8D	[f32] → [f32]	<i>validation</i>	
f32.floor	0x8E	[f32] → [f32]	<i>validation</i>	
f32.trunc	0x8F	[f32] → [f32]	<i>validation</i>	
f32.nearest	0x90	[f32] → [f32]	<i>validation</i>	
f32.sqrt	0x91	[f32] → [f32]	<i>validation</i>	

Continued on next page

Table 11.1 – continued from previous page

Instruction	Opcode	Type	Validation	Execution
f32.add	0x92	[f32 f32] → [f32]	<i>validation</i>	
f32.sub	0x93	[f32 f32] → [f32]	<i>validation</i>	
f32.mul	0x94	[f32 f32] → [f32]	<i>validation</i>	
f32.div	0x95	[f32 f32] → [f32]	<i>validation</i>	
f32.min	0x96	[f32 f32] → [f32]	<i>validation</i>	
f32.max	0x97	[f32 f32] → [f32]	<i>validation</i>	
f32.copysign	0x98	[f32 f32] → [f32]	<i>validation</i>	
f64.abs	0x99	[f64] → [f64]	<i>validation</i>	
f64.neg	0x9A	[f64] → [f64]	<i>validation</i>	
f64.ceil	0x9B	[f64] → [f64]	<i>validation</i>	
f64.floor	0x9C	[f64] → [f64]	<i>validation</i>	
f64.trunc	0x9D	[f64] → [f64]	<i>validation</i>	
f64.nearest	0x9E	[f64] → [f64]	<i>validation</i>	
f64.sqrt	0x9F	[f64] → [f64]	<i>validation</i>	
f64.add	0xA0	[f64 f64] → [f64]	<i>validation</i>	
f64.sub	0xA1	[f64 f64] → [f64]	<i>validation</i>	
f64.mul	0xA2	[f64 f64] → [f64]	<i>validation</i>	
f64.div	0xA3	[f64 f64] → [f64]	<i>validation</i>	
f64.min	0xA4	[f64 f64] → [f64]	<i>validation</i>	
f64.max	0xA5	[f64 f64] → [f64]	<i>validation</i>	
f64.copysign	0xA6	[f64 f64] → [f64]	<i>validation</i>	
i32.wrap/i64	0xA7	[i64] → [i32]	<i>validation</i>	
i32.trunc_s/f32	0xA8	[f32] → [i32]	<i>validation</i>	
i32.trunc_u/f32	0xA9	[f32] → [i32]	<i>validation</i>	
i32.trunc_s/f64	0xAA	[f64] → [i32]	<i>validation</i>	
i32.trunc_u/f64	0xAB	[f64] → [i32]	<i>validation</i>	
i64.extend_s/i32	0xAC	[i32] → [i64]	<i>validation</i>	
i64.extend_u/i32	0xAD	[i32] → [i64]	<i>validation</i>	
i64.trunc_s/f32	0xAE	[f32] → [i64]	<i>validation</i>	
i64.trunc_u/f32	0xAF	[f32] → [i64]	<i>validation</i>	
i64.trunc_s/f64	0xB0	[f64] → [i64]	<i>validation</i>	
i64.trunc_u/f64	0xB1	[f64] → [i64]	<i>validation</i>	
f32.convert_s/i32	0xB2	[i32] → [f32]	<i>validation</i>	
f32.convert_u/i32	0xB3	[i32] → [f32]	<i>validation</i>	
f32.convert_s/i64	0xB4	[i64] → [f32]	<i>validation</i>	
f32.convert_u/i64	0xB5	[i64] → [f32]	<i>validation</i>	
f32.demote/f64	0xB6	[f64] → [f32]	<i>validation</i>	
f64.convert_s/i32	0xB7	[i32] → [f64]	<i>validation</i>	
f64.convert_u/i32	0xB8	[i32] → [f64]	<i>validation</i>	
f64.convert_s/i64	0xB9	[i64] → [f64]	<i>validation</i>	
f64.convert_u/i64	0xBA	[i64] → [f64]	<i>validation</i>	
f64.promote/f32	0xBB	[f32] → [f64]	<i>validation</i>	
i32.reinterpret/f32	0xBC	[f32] → [i32]	<i>validation</i>	
i64.reinterpret/f64	0xBD	[f64] → [i64]	<i>validation</i>	
f32.reinterpret/i32	0xBE	[i32] → [f32]	<i>validation</i>	
f64.reinterpret/i64	0xBF	[i64] → [f64]	<i>validation</i>	

A

abstract syntax, **5**
 byte, **6**, **38**
 data, **15**, **29**, **49**
 element, **14**, **28**, **48**
 element type, **8**, **40**
 export, **15**, **29**, **47**
 expression, **12**, **26**, **45**
 external type, **8**
 floating-point number, **6**, **39**
 function, **13**, **27**, **47**, **48**
 function index, **13**, **45**
 function type, **7**, **40**
 global, **14**, **28**, **47**
 global index, **13**, **45**
 global type, **8**, **41**
 grammar, **5**
 import, **15**, **30**, **46**
 instruction, **9–11**, **20–23**, **41**, **42**
 integer, **6**, **38**
 label index, **13**, **45**
 limits, **8**, **27**, **40**
 local index, **13**, **45**
 memory, **14**, **28**, **47**
 memory index, **13**, **45**
 memory type, **8**, **40**
 module, **12**, **31**, **49**
 mutability, **8**, **41**
 name, **6**, **39**
 notation, **5**
 result type, **7**, **40**
 signed integer, **6**, **38**
 start function, **15**, **29**, **48**
 table, **14**, **28**, **47**
 table index, **13**, **45**
 table type, **8**, **40**
 type, **7**, **39**
 type definition, **13**, **46**
 type index, **13**, **45**
 uninterpreted integer, **6**, **38**
 unsigned integer, **6**, **38**
 value, **5**, **38**
 value type, **7**, **40**
 vector, **6**, **39**

B

binary encoding
 byte, **38**
binary format, **37**
 custom section, **46**
 data, **49**
 element, **48**
 element type, **40**
 export, **47**
 expression, **45**
 floating-point number, **39**
 function, **47**, **48**
 function index, **45**
 function type, **40**
 global, **47**
 global index, **45**
 global type, **41**
 grammar, **37**
 import, **46**
 instruction, **41**, **42**
 integer, **38**
 label index, **45**
 limits, **40**
 local index, **45**
 memory, **47**
 memory index, **45**
 memory type, **40**
 module, **49**
 mutability, **41**
 name, **39**
 notation, **37**
 result type, **40**
 section, **46**
 signed integer, **38**
 start function, **48**
 table, **47**
 table index, **45**
 table type, **40**
 type, **39**
 type index, **45**
 type section, **46**
 uninterpreted integer, **38**
 unsigned integer, **38**
 value type, **40**

- vector, 39
- binary: binary format
 - value, 38
- block, 11, 23, 41
- branch, 11, 23, 41
- byte, 6, 6, 15, 29, 37–39, 49
 - abstract syntax, 6
 - binary encoding, 38

C

- code, 9
 - section, 48
- code section, 48
- concepts, 2
- constant, 12, 26
- context, 17, 19, 31, 49
- control instruction, 11
- control instructions, 23, 41
- custom section, 46
 - binary format, 46

D

- data, 12, 14, 15, 29, 31, 49
 - abstract syntax, 15
 - binary format, 49
 - section, 49
 - segment, 15, 29, 49
 - validation, 29
- data section, 49
- decoding, 3
- design goals, 1

E

- element, 12, 14, 14, 28, 31, 48, 49
 - abstract syntax, 14
 - binary format, 48
 - section, 48
 - segment, 14, 28, 48
 - type, 8
 - validation, 28
- element section, 48
- element type, 8, 28, 40
 - abstract syntax, 8
 - binary format, 40
- embedder, 2
- execution, 3
- export, 12, 15, 29, 31, 47, 49
 - abstract syntax, 15
 - binary format, 47
 - section, 47
 - validation, 29
- export section, 47
- expression, 12, 13–15, 26–29, 45, 47–49
 - abstract syntax, 12
 - binary format, 45
 - constant, 12, 26, 45
 - validation, 26
- external

- type, 8
- external type, 8
 - abstract syntax, 8

F

- floating-point number, 6, 39
 - abstract syntax, 6
 - binary format, 39
- function, 2, 12, 13, 15, 27, 31, 47–49
 - abstract syntax, 13, 27
 - binary format, 47, 48
 - export, 15
 - import, 15
 - index, 13
 - section, 47
 - type, 7
- function index, 11, 13, 13–15, 23, 27–29, 41, 45, 47, 48
 - abstract syntax, 13
 - binary format, 45
- function section, 47
- function type, 7, 8, 12, 15, 17, 19, 27, 30, 31, 40, 46–49
 - abstract syntax, 7
 - binary format, 40

G

- global, 10, 12, 14, 15, 28, 31, 47, 49
 - abstract syntax, 14
 - binary format, 47
 - export, 15
 - import, 15
 - index, 13
 - mutability, 8
 - section, 47
 - type, 8
 - validation, 28
- global index, 10, 13, 14, 15, 21, 29, 42, 45, 47
 - abstract syntax, 13
 - binary format, 45
- global section, 47
- global type, 8, 8, 14, 15, 17, 28, 30, 41, 46, 47
 - abstract syntax, 8
 - binary format, 41
- globaltype, 17
- grammar notation, 5, 37

I

- import, 12–14, 15, 27, 30, 31, 46, 49
 - abstract syntax, 15
 - binary format, 46
 - section, 46
 - validation, 30
- import section, 46
- index, 13, 15, 29, 45, 47
 - function, 13
 - global, 13
 - label, 13
 - local, 13
 - memory, 13

- table, 13
- type, 13
- index space, **13**, 15, 17, 45
- instantiation, 3, 15
- instruction, 2, **9**, 19, 26, 41
 - abstract syntax, 9–11
 - binary format, 41, 42
 - validation, 20–23, 42
- integer, **6**, 38
 - abstract syntax, 6
 - binary format, 38
 - signed, 6
 - uninterpreted, 6
 - unsigned, 6
- invocation, 3

L

- label, **11**, 23, 41
 - index, 13
- label index, **11**, **13**, 23, 41, 45
 - abstract syntax, 13
 - binary format, 45
- limits, **8**, 8, 14, 27, 28, 40
 - abstract syntax, 8
 - binary format, 40
 - memory, 8
 - table, 8
 - validation, 27
- linear memory, 2
- local, **10**, **13**, 27, 48
 - index, 13
- local index, **10**, **13**, 13, 21, 27, 42, 45
 - abstract syntax, 13
 - binary format, 45

M

- memory, 2, 12, **14**, 15, 28, 29, 31, 47, 49
 - abstract syntax, 14
 - binary format, 47
 - data, 15, 29, 49
 - export, 15
 - import, 15
 - index, 13
 - limits, 8
 - section, 47
 - type, 8
 - validation, 28
- memory index, **11**, **13**, 14, 15, 22, 29, 42, 45, 47, 49
 - abstract syntax, 13
 - binary format, 45
- memory instruction, **11**, 22, 42
- memory section, **47**
- memory type, **8**, 8, 14, 15, 17, 28, 30, 40, 46, 47
 - abstract syntax, 8, 40
- module, 2, **12**, 31, 49
 - abstract syntax, 12
 - binary format, 49
 - validation, 31

- mutability, **8**, 14, 28, 41
 - abstract syntax, 8
 - binary format, 41
 - global, 8

N

- name, **6**, 15, 29, 30, 39, 46, 47
 - abstract syntax, 6
 - binary format, 39
- notation, 5, 37
 - abstract syntax, 5
 - binary format, 37
- numeric instruction, **9**, 20, 42

O

- opcode, **41**
- operand, 9
- operand stack, 9

P

- page size, 8, 14, 40
- parametric instruction, **10**
- parametric instructions, 21
- phases, 3
- polymorphism, 21, 23, 41
- portability, 1

R

- result
 - type, 7
- result type, **7**, 7, 11, 17, 23, 40, 41
 - abstract syntax, 7
 - binary format, 40
- resulttype, 17

S

- section, **46**, 49
 - binary format, 46
 - code, 48
 - custom, 46
 - data, 49
 - element, 48
 - export, 47
 - function, 47
 - global, 47
 - import, 46
 - memory, 47
 - start, 48
 - table, 47
 - type, 46
- signed integer, **6**, 38
 - abstract syntax, 6
 - binary format, 38
- stack machine, 9
- start function, 12, **15**, 29, 31, 48, 49
 - abstract syntax, 15
 - binary format, 48

- section, 48
- validation, 29
- start section, **48**
- structured control, **11**, 23, 41

T

- table, 2, 12, **14**, 14, 15, 28, 31, 47, 49
 - abstract syntax, 14
 - binary format, 47
 - element, 14, 28, 48
 - export, 15
 - import, 15
 - index, 13
 - limits, 8
 - section, 47
 - type, 8
 - validation, 28
- table index, **13**, 14, 15, 28, 29, 45, 47, 48
 - abstract syntax, 13
 - binary format, 45
- table section, **47**
- table type, **8**, 8, 14, 15, 17, 28, 30, 40, 46, 47
 - abstract syntax, 8
 - binary format, 40
- trap, 2
- type, **7**, 39
 - abstract syntax, 7
 - binary format, 39
 - element, 8
 - external, 8
 - function, 7
 - global, 8
 - index, 13
 - memory, 8
 - result, 7
 - section, 46
 - table, 8
 - value, 7
- type definition, 12, **13**, 31, 46, 49
 - abstract syntax, 13
- type index, **11**, **13**, 13, 15, 23, 27, 41, 45, 47, 48
 - abstract syntax, 13
 - binary format, 45
- type section, **46**
 - binary format, 46
- type system, **17**

U

- uninterpreted integer, **6**, 38
 - abstract syntax, 6
 - binary format, 38
- unsigned integer, **6**, 38
 - abstract syntax, 6
 - binary format, 38
- unwinding, **11**

V

- validation, 3, **17**

- data, 29
- element, 28
- export, 29
- expression, 26
- global, 28
- import, 30
- instruction, 20–23, 42
- limits, 27
- memory, 28
- module, 31
- start function, 29
- table, 28
- valtype, 17
- value, 2, **5**, 38
 - abstract syntax, 5
 - type, 7
- value type, **7**, 7, 8, 13, 17, 21, 27, 40, 41
 - abstract syntax, 7
 - binary format, 40
- variable instruction, **10**
- variable instructions, 21, 42
- vector, **6**, 11, 14, 15, 23, 39, 41
 - abstract syntax, 6
 - binary format, 39