

# Specification of Webassembly Text (Source Version)—2023 edition

Kim Yongbeom

National University of Singapore  
School of Computing

November 3, 2023

# 1 Introduction

## 1.1 Background

The main motivation behind this document is to provide a specification for the WebAssembly Text format used in the Source Academy.

This differs from the official WebAssembly specifications in that this document is meant to be a specification of the WebAssembly Text features implemented in Source Academy, and that it is also meant for users to write and understand WebAssembly Text as a language, rather than to provide a industry-wide specification for WebAssembly as a whole, including runtime, binary and verification details among others.

In short, this document is meant to be a specification for the WebAssembly Text format supported and used in the Source Academy (or the `source-academy-wabt` module).

## 1.2 About WebAssembly Text

WebAssembly Text is a text format for WebAssembly modules. The design of the WebAssembly runtime and instruction set are beyond the scope of this specifcaiton, and can be read in the official WebAssembly [specification](#).

Notably, the computational model of WebAssembly is based on a stack machine, where a sequence of instructions are executed in order. Instrucitons consume values on an implicit operand stack, and push any results back onto the stack. The WebAssembly Text format is a rendering of the above syntax into S-expressions.

## 1.3 Differences between official WebAssembly Text Format

Here are documented differences between the current specifications and the official WebAssembly Text specifications.

### 1.3.1 Data & Data Count Segment

The data and data count segments in the official WebAssembly Text specifications are omitted and not support in the current iteration of the WebAssembly Text compiler.

## 2 Syntax

The following rules concern basic WebAssembly Text syntax.

### 2.1 White Space

White space is any sequence of the following: space (U+020), horizontal tab (U+009), line feed (U+00A), carriage return (U+00D) or comments. White space is ignored except as it separates tokens that would otherwise combine into a single token.

```

space ::= (U+020 | format | comment)*
format ::=
    | U+009                horizontal tab
newline ::= U+00A          line feed
    | U+00D                carriage return
    | U+00D U+00A          line feed + carriage return

```

A line comment starts with a double semicolon (;;) and continues to the end of the line, whereas a block comment is enclosed in parentheses and semicolons ((; and ;)). Block comments can be nested.

```

comment ::= linecomment | blockcomment
linecomment ::= ;; char* (newline | eof)
blockcomment ::= (; blockchar* ;)
blockchar ::= char                if char is not ; or (
    | ;                          if next char is not )
    | (                          if next char is not ;
    | blockcomment

```

### 2.2 Strings

A string is a sequence of characters encoded as UTF-8. A string must be enclosed in quotation marks and may contain any character other than ASCII44 control characters, quotation marks ("), or backslash (\), except when expressed with an escape sequence.

```

string ::= " stringelem "
stringelem ::= stringchar
stringelem ::= \ hexdigit hexdigit    2-digit hexcode
module-name ::= string
name ::= string
import-desc ::= (func id? typeuse )    function import
    | (table id? tabletype )          table import
    | (memory id? memtype )           memory import
    | (global id? globaltype )        global import

```

## 2.3 Names

A name is string.

$$\text{name} ::= \text{string}$$

## 2.4 Identifiers

Each definition can be identified by either its index or symbolic identifier. Symbolic identifiers are identifiers that start with a dollar sign (\$) followed by a name. A name is a string that does not contain a space, quotation mark, comma, semicolon or bracket.

$$\begin{aligned} \text{id} &::= \$ \text{idchar}^+ \\ \text{idchar} &::= 0 \mid \dots \mid 9 \mid \\ &\quad \mid a \mid \dots \mid z \mid \\ &\quad \mid A \mid \dots \mid Z \mid \\ &\quad \mid ! \mid \# \mid \$ \mid \% \mid \& \mid * \mid + \mid - \mid . \mid / \mid : \mid < \\ &\quad \mid = \mid > \mid ? \mid @ \mid \backslash \mid ^ \mid \_ \mid ' \mid | \end{aligned}$$

## 2.5 Types

The following are the available types in WebAssembly.

### 2.5.1 Number Types

All numbers are either 32- or 64-bit integers or floating points.

$$\begin{aligned} \text{numtype} &::= i32 \\ &\quad \mid i64 \\ &\quad \mid f32 \\ &\quad \mid f64 \end{aligned}$$

### 2.5.2 Reference Types

Reference types are first-class references to objects. `funcref` is a reference to a function. `externref` is a reference to an external object.

$$\begin{aligned} \text{reftype} &::= \text{funcref} \\ &\quad \mid \text{externref} \\ \text{heaptypes} &::= \text{func} \\ &\quad \mid \text{extern} \end{aligned}$$

### 2.5.3 Value Types

$$\begin{aligned} \text{valtype} &::= \text{numtype} \\ &\quad \mid \text{reftype} \end{aligned}$$

### 2.5.4 Function Types

The type of a function is determined by its parameters and result, where the function maps the parameter types to the result types. The parameters and result are value types.

```
functype ::= ( func param* result* )
param    ::= ( param id? valtype )
result   ::= ( result valtype )
```

### 2.5.5 Global Types

Globals refer to global variables. A global type is a value type and a mutability flag.

```
globaltype ::= mut valtype
mut        | const | var
```

## 3 Segments

Each webassembly program is a module consisting of a sequence of segments. A module collects definitions for types, functions, tables, memories and globals. In addition, it can declare imports and exports and provide initialisation in the form of data and element segments, or a start function.

```
module ::= ( module segment* ) module
segment ::= type                type segment
          | import              import segment
          | function            function segment
          | table               table segment
          | memory              memory segment
          | global              global segment
          | export              export segment
          | start               start segment
          | element             element segment
          | data                data segment
```

### 3.1 Type Segment

A type segment declares and binds identifiers to function types. The type segment is typically omitted in WebAssembly Text programs.

```
type ::= ( type id? functype )
```

## 3.2 Import Segment

We can import functions, tables, memories or globals.

```

import-section ::= (import module-name name import-desc ) import
module-name  ::= string
              name ::= string
import-desc  ::= (func id? typeuse )           function import
                | (table id? tabletype )       table import
                | (memory id? memtype )        memory import
                | (global id? globaltype )     global import

```

## 3.3 Function Segment

A function segment declares a function with an optional function identifier, parameters, return values and local variables.

```

function-section ::= ( func id? typeuse local* instr* ) function section
local           ::= ( local id? valtype )

```

### 3.3.1 Type Uses

A type use is a reference to a type definition.

```

typeuse ::= ( type typeidx ) param* result* type definition
param   ::= ( param id? valtype ) parameter declaration
result  ::= ( result valtype ) param* result* return value declaration

```

A type use can also be replaced by inline parameter and result declarations. In this case, a type index is automatically inserted.

```

param* result* ::= ( type typeidx ) param* result*

```

### 3.3.2 Function Instructions

Instructions are distinguished between plain and block instructions.

```

instr ::= plaininstr
       | blockinstr

```

### 3.3.3 Control Instructions

The block type of a block instruction is given similarly to the type definition of a function, or a single result type.

```

blocktype ::= (result)?
           | typeuse
blockinstr ::= block label blocktype (instr)* end id?
           | loop label blocktype (instr)* end id?
           | if label blocktype (instr)* else id? (instr)* end

```

Note that the `else` keyword of an `if` instruction can be omitted if the following instruction sequence is empty.

The following are the plain instructions that interact with instruction blocks.

```

plaininstr ::= ...
           | unreachable
           | nop
           | br
           | br_if
           | br_table
           | return
           | call funcidx
           | call_indirect tableidx typeuse

```

The `unreachable` instruction is a special instruction that always traps, which immediately aborts execution. It can be used to indicate unreachable code.

The `nop` instruction does nothing.

`br`, `br_if` and `br_table` are branch instructions. `br` performs an unconditional branch, `br_if` performs a conditional branch, and `br_table` performs an indirect branch through operand indexing to a table. Notably, taking a branch unwinds the operand stack up to the

branch instruction's target block.

The `return` instruction is an unconditional branch to the caller of the current function.

The `call` instruction calls a function, consuming necessary arguments from the stack and returning the result values of the call back onto the stack.

The `call_indirect` instruction invokes a function indirectly via operand indexing to a table.

### 3.3.4 Reference Instructions

```

plaininstr ::= ...
           | ref.null heaptypes
           | ref.is_null
           | ref.null func-index

```

The `ref.null` instruction produces a null value.

The `ref.is_null` instruction checks for a null value.

The `ref.func` instruction produces a reference to a given function.

### 3.3.5 Parametric Instructions

```
plaininstr ::= ...
            | drop
            | select (valtype)?
```

The `drop` instruction throws away a single operand on the stack.

The `select` operand selects one of its first two operand based on whether the third operand may be zero or not. It may include an optional value type determining the type of the operands. If the value type is excluded, the operands must be of numeric type.

### 3.3.6 Variable Instructions

Variable instructions are used to access local and global variables.

```
plaininstr ::= ...
            | local.get local-index
            | local.set local-index
            | local.tee local-index
            | global.get global-index
            | global.set global-index
```

The `local.get` instruction fetches the value of a local variable.

The `local.set` instruction writes a value to a local variable.

The `local.tee` instruction writes a value to a local variable and returns the same value.

In each function, local variable consists of the function parameters and the local variables declared in the function, and each local variable may be identified by its index. Local variables are indexed from zero, starting with the function parameters, and then spilling over to the local variables. Alternatively, local variables may also be identified by a given identifier.

The `global.get` instruction fetches the value of a global variable.

The `global.set` instruction writes a value to a global variable.

Global variables are indexed in order of their declaration, or by a given identifier.



### 3.3.7 Table Instructions

```

plaininstr ::= ...
            | table.get table-index
            | table.set table-index
            | table.size table-index
            | table.grow table-index
            | table.fill table-index
            | table.copy table-index table-index
            | table.init table-index elem-index
            | elem.drop elem-index

```

All table indices can be omitted from table instructions, and they default to zero. The `table.get` and `table.set` instructions access table elements.

The `table.size` instruction returns the current size of a table.

The `table.grow` instruction grows a table by a given number of elements. It returns the previous size of the table, or -1 if space cannot be allocated. Its second operand is an initialisation value for the newly allocated entries.

The `table.fill` instruction takes in three operands, the first being the starting table index, the second being the ending table index, and the third being the given value. It fills the table with the given value.

The `table.copy` instruction copies a range of elements from one table to another, and the `table.init` instruction initialises the contents of a table with a passive element segment. They both take in three operands - the destination index, the starting and the ending source index.

The `elem.drop` instruction drops a passive element segment, and marks it as unused.

## 3.3.8 Memory Instructions

```

memarg ::= offset align
offset ::= offset=u32
        | ~                                0 if omitted
align  ::= align=u32
        | ~                                0 if omitted
plaininstr ::= ...
            | i32.load memarg
            | i64.load memarg
            | f32.load memarg
            | f64.load memarg
            | i32.store memarg
            | i32.load8_s memarg
            | i32.load8_u memarg
            | i32.load16_s memarg
            | i32.load16_u memarg
            | i64.load8_s memarg
            | i64.load8_u memarg
            | i64.load16_s memarg
            | i64.load16_u memarg
            | i64.load32_s memarg
            | i64.load32_u memarg
            | i64.store memarg
            | f32.store memarg
            | f64.store memarg
            | i32.store8 memarg
            | i32.store16 memarg
            | i64.store8 memarg
            | i64.store16 memarg
            | i64.store32 memarg
            | memory.size
            | memory.grow
            | memory.fill
            | memory.copy

```

Instructions of the form `x.loady` loads a value of type `y` from memory and pushes it onto the operand stack as type `x`, and instructions of the form `x.storey` stores a value of type `x` to memory as type `y`.

The `memory.size` instruction returns the current size of the memory in units of pages.

The `memory.grow` instruction grows the memory by a given number of pages. It returns the previous size of the memory, or -1 if space cannot be allocated.

The `memory.fill` instruction takes in three operands, the first being the starting memory index, the second being the ending memory index, and the third being the given value. It fills the memory with the given value.

The `memory.copy` instruction copies a range of memory from one memory to another. It takes in three operands - the destination index, the starting and the ending source index.

### 3.3.9 Numeric Instructions

```
plaininstr ::= ...
            | i32.const
            | i64.const
            | f32.const
            | f64.const

            | i32.clz
            | i32.ctz
            | i32.popcnt
            | i32.add
            | i32.sub
            | i32.mul
            | i32.div_s
            | i32.div_u
            | i32.rem_s
            | i32.rem_u
            | i32.and
            | i32.or
            | i32.xor
            | i32.shl
            | i32.shr_s
            | i32.shr_u
            | i32.rotl
            | i32.rotr
```

- | i64.clz
- | i64.ctz
- | i64.popcnt
- | i64.add
- | i64.submul
- | i64.div\_s
- | i64.div\_u
- | i64.rem\_s
- | i64.rem\_u
- | i64.and
- | i64.or
- | i64.xor
- | i64.shl
- | i64.shr\_s
- | i64.shr\_u
- | i64.rotl
- | i64.rotr

- | f32.abs
- | f32.neg
- | f32.ceil
- | f32.floor
- | f32.trunc
- | f32.nearest
- | f32.sqrt
- | f32.add
- | f32.sub
- | f32.mul
- | f32.div
- | f32.min
- | f32.max
- | f32.copysign

- | f64.abs
- | f64.neg
- | f64.ceil
- | f64.floor
- | f64.trunc
- | f64.nearest
- | f64.sqrt
- | f64.add
- | f64.sub
- | f64.mul
- | f64.div
- | f64.min
- | f64.max
- | f64.copysign

- | i32.eqz
- | i32.eq
- | i32.ne
- | i32.lt\_s
- | i32.lt\_u
- | i32.gt\_s
- | i32.gt\_u
- | i32.le\_s
- | i32.le\_u
- | i32.ge\_s
- | i32.ge\_u
- | i64.eqz
- | i64.eq
- | i64.ne
- | i64.lt\_s
- | i64.lt\_u
- | i64.gt\_s
- | i64.gt\_u
- | i64.le\_s
- | i64.le\_u
- | i64.ge\_s
- | i64.ge\_u

| f32.eq  
| f32.ne  
| f32.lt  
| f32.gt  
| f32.le  
| f32.ge  
| f64.eq  
| f64.ne  
| f64.lt  
| f64.gt  
| f64.le  
| f64.ge

```
| i32.wrap_i64
| i32.trunc_f32_s
| i32.trunc_f32_u
| i32.trunc_f64_s
| i32.trunc_f64_u
| i32.trunc_sat_f32_s
| i32.trunc_sat_f32_u
| i32.trunc_sat_f64_s
| i32.trunc_sat_f64_u
| i64.extend_i32_s
| i64.extend_i32_u
| i64.trunc_f32_s
| i64.trunc_f32_u
| i64.trunc_f64_s
| i64.trunc_f64_u
| i64.trunc_sat_f32_s
| i64.trunc_sat_f32_u
| i64.trunc_sat_f64_s
| i64.trunc_sat_f64_u
| f32.convert_i32_s
| f32.convert_i32_u
| f32.convert_i64_s
| f32.convert_i64_u
| f32.demote_f64
| f64.convert_i32_s
| f64.convert_i32_u
| f64.convert_i64_s
| f64.convert_i64_u
| f64.promote_f32
| i32.reinterpret_f32
| i64.reinterpret_f64
| f32.reinterpret_i32
| f64.reinterpret_i64
```

```

| i32.extend8_s
| i32.extend16_s
| i64.extend8_s
| i64.extend16_s
| i64.extend32_s

```

The `const` instructions push a constant value onto the operand stack.

`clz`, `ctz`, and `popcnt` instructions are all unary operations.

The `clz` instruction counts the number of leading zero bits in the operand. The `ctz` instruction counts the number of trailing zero bits in the operand. The `popcnt` instruction counts the number of one bits in the operand.

The `add`, `sub`, `mul`, `div_s`, `div`, `rem`, `and`, `or`, `xor`, `shl`, `shr`, `shr`, `rotl`, and `rotr` instructions are binary numeric operations. They take in two operands and produce one result of the same type.

The `abs`, `neg`, `sqrt`, `ceil`, `floor`, `trunc`, `nearest` instructions are numeric operations that consume one operand and produce an operand of the same type.

The `eqz` instruction is a comparison that consumes one operand and produces a boolean integer result (of type `i32`).

The `eq`, `ne`, `lt`, `gt`, `le` and `ge` instructions are comparisons that consume two operands and produce a boolean integer result (of type `i32`).

Some integer instructions distinguish whether they work on signed or unsigned integers through the annotation `_s` or `_u`. .

### 3.4 Table Segment

A table is an array of values of a given reference 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 or a reference to an external host value.

```

table ::= ( table id? tabletype ) table
tabletype ::= limits reftype           table with limits capacity, reftype type.
limits ::= u32                        min
          | u32 u32                   min max

```

### 3.5 Element Segment

Element segments are segments used to initialise tables.



<code>elem ::= ( elem id<sup>?</sup> elemlist )</code>	passive element section
<code>  ( elem id<sup>?</sup> tableuse ( offset expr ) elemlist )</code>	active element section
<code>  ( elem id<sup>?</sup> declare elemlist )</code>	declarative element section
<code>elemlist ::= reftype elemexpr*</code>	
<code>elemexpr ::= ( item elemexpr )</code>	
<code>tableuse ::= ( table tableidx )</code>	

### 3.6 Memory Segment

A memory definition binds a symbolic memory identifier to a memory segment.

<code>mem ::= ( memory id<sup>?</sup> memtype )</code>	memory section
<code>memtype ::= limits</code>	memory section

### 3.7 Global Segment

<code>global ::= ( global id<sup>?</sup> globaltype expr )</code>	global segment
---	----------------

### 3.8 Export Segment

<code>export ::= ( export name exportdesc )</code>	export segment
<code>exportdesc ::= ( func funcidx )</code>	function export
<code>  ( table tableidx )</code>	table export
<code>  ( memory memidx )</code>	memory export
<code>  ( global globalidx )</code>	global export

### 3.9 Start Segment

A start function can be defined in terms of its index. Note that there is currently no good way for the return value of the start function to be retrieved by Source, so it is not recommended to be used.

<code>start ::= ( start funcidx )</code>	function start segment
--	------------------------