Specification of Source §1 WebAssembly—2021 edition

Bernard Teo Zhi Yi

National University of Singapore
School of Computing

October 14, 2021

The language Source is the official language of the textbook *Structure and Interpretation of Computer Programs*, JavaScript Adaptation. Source is a sublanguage of ECMAScript 2018 (9th Edition) and defined in the documents titled "Source §x", where x refers to the respective textbook chapter.
Source §1 WebAssembly is the dialect of Source §1 used in Sourceror, an experimental compiler from Source §1 to WebAssembly. This dialect is mostly similar to Source §1.

1 Changes compared to Vanilla Source §1

Language

All language features of Source §1 are supported except for proper tail calls.
Source §1 WebAssembly additionally supports proper modules – it is possible to import external modules from the web, and such modules are also written in Source §1 WebAssembly. The import and export syntax follow that of ECMAScript modules.

Libraries

The standard libraries (MISC and MATH) are supported with explicit import statements. There are two differences: Source §1 WebAssembly does not support true varargs, so functions like `math_max()` only support up to two arguments; and stringifying a function does not output the literal function body.
Other self-hosted libraries may also be used as long as they are implemented in Source §1 WebAssembly.

Error Handling

Compilation errors produce error messages with line numbers, just like in Source §1. Runtime type errors currently produce error messages without any location information.

2 Syntax

A Source program is a *program*, defined using Backus-Naur Form[1] as follows:

[1] We adopt Henry Ledgard's BNF variant that he described in *A human engineered variant of BNF*, ACM SIGPLAN Notices, Volume 15 Issue 10, October 1980, Pages 57-62. In our grammars, we use **bold** font for keywords, *italics* for syntactic variables, ϵ for nothing, $x \mid y$ for x or y, [x] for an optional x, x... for zero or more repetitions of x, and (x) for clarifying the structure of BNF expressions.

program	::=	*import-directive* ... *statement* ...	program
import-directive	::=	**import** **{** *import-names* **}** **from** *string* **;**	import directive
import-names	::=	ϵ | *import-name* (**,** *import-name*) ...	import name declarations
import-name	::=	*name* | *name* **as** *name*	import name declaration
export-directive	::=	**export** **{** *export-names* **}** **;**	export directive
export-names	::=	ϵ | *export-name* (**,** *export-name*) ...	export name declarations
export-name	::=	*name* | *name* **as** *name*	export name declaration
statement	::=	**const** *name* **=** *expression* **;**	constant declaration
	|	**function** *name* **(** *parameters* **)** *block*	function declaration
	|	**return** *expression* **;**	return statement
	|	*if-statement*	conditional statement
	|	*block*	block statement
	|	*expression* **;**	expression statement
parameters	::=	ϵ | *name* (**,** *name*) ...	function parameters
if-statement	::=	**if** **(** *expression* **)** *block*	
		else (*block* | *if-statement*)	conditional statement
block	::=	**{** *statement* ... **}**	block statement
expression	::=	*number*	primitive number expression
	|	**true** | **false**	primitive boolean expression
	|	*string*	primitive string expression
	|	*name*	name expression
	|	*expression binary-operator expression*	binary operator combination
	|	*unary-operator expression*	unary operator combination
	|	*expression* **(** *expressions* **)**	function application
	|	(*name* | **(** *parameters* **)**) **=>** *expression*	lambda expression (expr. body)
	|	(*name* | **(** *parameters* **)**) **=>** *block*	lambda expression (block body)
	|	*expression* **?** *expression* **:** *expression*	conditional expression
	|	**(** *expression* **)**	parenthesised expression
binary-operator	::=	**+** | **-** | ***** | **/** | **%** | **===** | **!==**	
	|	**>** | **<** | **>=** | **<=** | **&&** | **||**	binary operator
unary-operator	::=	**!** | **-**	unary operator
expressions	::=	ϵ | *expression* (**,** *expression*) ...	argument expressions

Restrictions

- Return statements are only allowed in bodies of functions.
- There cannot be any newline character between **`return`** and *expression* in return statements.[2]
- There cannot be any newline character between (*name* | **`(`** *parameters* **`)`**) and **`=>`** in function definition expressions.[3]
- Implementations of Source are allowed to treat function declaration as syntactic sugar for constant declaration.[4] Source programmers need to make sure that functions are not called before their corresponding function declaration is evaluated.

Import directives

Import directives allow programs to import values from modules and bind them to names, whose scope is the entire program in which the import directive occurs. Import directives can only appear at the top-level. All names that appear in import directives must be distinct, and must also be distinct from all top-level variables.
The import graph must not contain cycles.

Importable module types

The module being imported must either be a Source §1 WebAssembly module (the usual kind of module) or a Source Imports module.

Import filenames

The module name can be an absolute URL (e.g. `https://www.example.com/my_modules/module.source`) or a relative URL (e.g. `std/misc.source`). If the file extension is ".source", it may be omitted.
An absolute URL will fetch the module from the specified URL.
A relative URL will fetch the module from an implementation-defined location.

Export directives

Export directives allow programs to export values from modules and bind them to names, so that they can be imported by other modules. Export directives can only appear at the top-level, and hence can only export top-level variables.

Binary boolean operators

Conjunction

$$\textit{expression}_1 \texttt{ \&\& } \textit{expression}_2$$

stands for

$$\textit{expression}_1 \texttt{ ? } \textit{expression}_2 \texttt{ : false}$$

Disjunction

$$\textit{expression}_1 \texttt{ || } \textit{expression}_2$$

stands for

$$\textit{expression}_1 \texttt{ ? true : } \textit{expression}_2$$

[2] Source inherits this syntactic quirk of JavaScript.
[3] ditto
[4] ECMAScript prescribes "hoisting" of function declarations to the beginning of the surrounding block. Programs that rely on this feature will run fine in JavaScript but might encounter a runtime error "Cannot access name before initialization" in a Source implementation.

Names

Names[5] start with _, `$` or a letter[6] and contain only _, `$`, letters or digits[7]. Restricted words[8] are not allowed as names.
Valid names are `x`, `_45`, `$$` and π, but always keep in mind that programming is communicating and that the familiarity of the audience with the characters used in names is an important aspect of program readability.

Numbers

We use decimal notation for numbers, with an optional decimal dot. "Scientific notation" (multiplying the number with 10^x) is indicated with the letter `e`, followed by the exponent x. Examples for numbers are `5432`, `-5432.109`, and `-43.21e-45`.

Strings

Strings are of the form `"`*double-quote-characters*`"`, where *double-quote-characters* is a possibly empty sequence of characters without the character `"` and without the newline character, of the form `'`*single-quote-characters*`'`, where *single-quote-characters* is a possibly empty sequence of characters without the character `'` and without the newline character, and of the form `` `backquote-characters` ``, where *backquote-characters* is a possibly empty sequence of characters without the character `` ` ``. Note that newline characters are allowed as *backquote-characters*.
The following characters can be represented in strings as given:

- horizontal tab: `\t`
- vertical tab: `\v`
- nul char: `\0`
- backspace: `\b`
- form feed: `\f`
- newline: `\n`
- carriage return: `\r`
- single quote: `\'`
- double quote: `\"`
- backslash: `\\`

Unicode characters can be used in strings using `\u` followed by the hexadecimal representation of the unicode character, for example `'\uD83D\uDC04'`.

Comments

In Source, any sequence of characters between "`/*`" and the next "`*/`" is ignored.
After "`//`" any characters until the next newline character is ignored.

---

[5] In ECMAScript 2020 (9th Edition), these names are called *identifiers*.
[6] By *letter* we mean Unicode letters (L) or letter numbers (Nl).
[7] By *digit* we mean characters in the Unicode categories Nd (including the decimal digits 0, 1, 2, 3, 4, 5, 6, 7, 8, 9), Mn, Mc and Pc.
[8] By *restricted word* we mean any of: **`arguments`**, **`await`**, **`break`**, **`case`**, **`catch`**, **`class`**, **`const`**, **`continue`**, **`debugger`**, **`default`**, **`delete`**, **`do`**, **`else`**, **`enum`**, **`eval`**, **`export`**, **`extends`**, **`false`**, **`finally`**, **`for`**, **`function`**, **`if`**, **`implements`**, **`import`**, **`in`**, **`instanceof`**, **`interface`**, **`let`**, **`new`**, **`null`**, **`package`**, **`private`**, **`protected`**, **`public`**, **`return`**, **`static`**, **`super`**, **`switch`**, **`this`**, **`throw`**, **`true`**, **`try`**, **`typeof`**, **`var`**, **`void`**, **`while`**, **`with`**, **`yield`**. These are all words that cannot be used without restrictions as names in the strict mode of ECMAScript 2020.

3 Function Attributes

This feature may only be used in modules other than the main module.
Function declarations may be annotated with attributes. Attributes are a key-value map that modify the function declaration that it is applied to. The exact syntax of attributes are unstable, but their semantics are unlikely to change. The following attributes are recognised:

- `direct`: This attribute key should be specified without a value. If present, the function declaration it is applied to shall not capture any non-toplevel variables, and shall behave as if it is declared as the first statement of the scope that it is in. Such functions are called *direct* functions. Implementations are encouraged to emit direct calls instead of indirect calls.
- `constraint`: The value shall be a map that specifies the signature of the function declaration that it is applied to (for example `x:number,y:number` specifies that the parameters `x` and `y` are both numbers). If a parameter is not mentioned in this attribute value, the type defaults to `any`, meaning that there is no constraint on the parameter value. A function declaration with a `constraint` attribute must also be a direct function. Implementations are required to raise a runtime error if such a function is called with wrong actual argument types.

Runtime overloading

Multiple declarations of direct functions with the same name are allowed, subject to the rules described in this section.
At runtime, a call expression is said to *match* a direct function if the call expression has the same number of arguments as the direct function, and the actual argument values of the call expression have types that fit into each parameter type of the direct function.
A direct function `f` is said to *shadow* a direct function `g` if `f` and `g` have the same name and every call expression that matches `g` also matches `f`.
Two direct functions `f` and `g` with the same name, where `f` appears earlier than `g`, are allowed to coexist only if, either:

- `f` and `g` are not in the same scope (and hence would not have been subject to the usual ECMAScript repeated-declaration rules), or
- `g` does not shadow `f`.

Overload resolution

At runtime, when a call expression is encountered that attempts to call a function `f`, the following algorithm is used to determine the overload to call:

1. If there is a variable `f` accessible from the current scope, the normal behaviour of ECMAScript is used - we assert that `f` is indeed a function, and then try to call it.
2. Otherwise, we do the following:
   (a) Collect all direct function declarations of `f` into a list, ordered so that earlier (i.e. further) declarations come before later (i.e. nearer) declarations. Declarations in imports are also collected and ordered in a manner such that declarations from the file being imported come before declarations from the file containing the import statement. The import ordering rule applies transitively to chains of imports. Where multiple orderings satisfy the above constraints, implementations may use any valid ordering.
   (b) If there is at least one matching function declaration, the latest matching declaration is used. Otherwise, an error is raised.

A name that refers to a possibly overloaded direct function may appear in an expression or statement at any position (called the *bind* site) where a lambda expression would have been valid. Such a value behaves as if it is a non-direct function, except that when it is invoked, the actual arguments at the call site are matched against the function declarations visible at the bind site, using the overload resolution rules above.

The exact implementation details of binding direct functions is left unspecified. Implementations are encouraged to perform overload resolution at the call site in $O(n)$ time, where n is the number of function declarations in the list; and construct the overloaded function value at the bind site in $O(1)$ time.

4 Dynamic Type Checking

Expressions evaluate to numbers, boolean values, strings or function values. Implementations of Source generate error messages when unexpected values are used as follows.
Only function values can be applied using the syntax:

expression ::= *name* **(** *expressions* **)**

For compound functions, implementations need to check that the number of *expressions* matches the number of parameters.
The following table specifies what arguments Source's operators take and what results they return. Implementations need to check the types of arguments and generate an error message when the types do not match.

operator	argument 1	argument 2	result
`+`	number	number	number
`+`	string	string	string
`-`	number	number	number
`*`	number	number	number
`/`	number	number	number
`%`	number	number	number
`===`	number	number	bool
`===`	string	string	bool
`!==`	number	number	bool
`!==`	string	string	bool
`>`	number	number	bool
`>`	string	string	bool
`<`	number	number	bool
`<`	string	string	bool
`>=`	number	number	bool
`>=`	string	string	bool
`<=`	number	number	bool
`<=`	string	string	bool
`&&`	bool	any	any
`||`	bool	any	any
`!`	bool		bool
`-`	number		number

Preceding `?` and following `if`, Source only allows boolean expressions.

5 Source Imports Modules

Source Imports modules provide a foreign function interface from Source to the host environment.
A Source Imports module declares functions that are implemented in the host environment, allowing Source programs to use them. It is satisfies the following syntax:

- The first line must be exactly `@SourceImports`.
- Subsequent lines are either empty, or are *foreign-import-statement*s that satisfy the following syntax

foreign-import-statement	::=	*exported-name host-namespace host-entity return-type param-types*	FFI import statement
exported-name	::=	*name*	exported name
host-namespace	::=	*name*	host namespace name
host-entity	::=	*name*	host entity name
return-type	::=	**undefined** | **number** | **string**	return type
param-types	::=	*param-type*...	export name declarations
param-type	::=	**number** | **string**	param type

Each *foreign-import-statement* declares the signature of a host-implemented function identified by the pair (*host-namespace*, *host-entity*), and exports it from the current Source Imports module as *exported-name* as if it was a direct function.
Overloading behaves in the same way as direct functions.
The list of host-implemented functions available to Source §1 WebAssembly is implementation-defined, and behaviour is often dependent on the host environment.

6 Standard Libraries

The following libraries are always available in this language.
Names must be imported explicitly before being used. The MISC library can be imported as "std/misc" and the MATH library can be imported as "std/math".
For example, `import { get_time } from "std/misc";` will import the `get_time()` function.

MISC Library

The following names are provided by the MISC library:

- `get_time()`: *primitive*, returns number of milliseconds elapsed since January 1, 1970 00:00:00 UTC
- `parse_int(s, i)`: *primitive*, interprets the *string* `s` as an integer, using the positive integer `i` as radix, and returns the respective value, see ECMAScript Specification, Section 18.2.5.
- `undefined`, `NaN`, `Infinity`: *primitive*, refer to JavaScript's undefined, NaN ("Not a Number") and Infinity values, respectively.
- `is_boolean(x)`, `is_number(x)`, `is_string(x)`, `is_undefined(x)`, `is_function(x)`: *primitive*, returns `true` if the type of `x` matches the function name and `false` if it does not. Following JavaScript, we specify that `is_number` returns `true` for `NaN` and `Infinity`.
- `prompt(s)`: *primitive*, pops up a window that displays the *string* `s`, provides an input line for the user to enter a text, a "Cancel" button and an "OK" button. The call of `prompt` suspends execution of the program until one of the two buttons is pressed. If the "OK" button is pressed, `prompt` returns the entered text as a string. If the "Cancel" button is pressed, `prompt` returns a non-string value.
- `display(x)`: *primitive*, displays the value `x` in the console[9]; returns the argument `a`.
- `display(x, s)`: *primitive*, displays the string `s`, followed by a space character, followed by the value `x` in the console[9]; returns the argument `x`.
- `error(x)`: *primitive*, displays the value `x` in the console[9] with error flag. The evaluation of any call of `error` aborts the running program immediately.

[9]The notation used for the display of values is consistent with JSON, but also displays `undefined` and function objects.

- `error(x, s)`: *primitive*, displays the string `s`, followed by a space character, followed by the value `x` in the console[9] with error flag. The evaluation of any call of `error` aborts the running program immediately.
- `stringify(x)`: *primitive*, returns a string that represents[9] the value `x`.

All library functions can be assumed to run in $O(1)$ time, except `display`, `error` and `stringify`, which run in $O(n)$ time, where n is the size (number of components such as pairs) of their first argument.

MATH Library

The following names are provided by the MATH library:

- `math_`*name*, where *name* is any name specified in the JavaScript `Math` library, see ECMAScript Specification, Section 20.2. Examples:
  - `math_PI`: *primitive*, refers to the mathematical constant π,
  - `math_sqrt(n)`: *primitive*, returns the square root of the *number* `n`.

All functions can be assumed to run in $O(1)$ time and are considered *primitive*.

Deviations from JavaScript

We intend the Source language to be a conservative extension of JavaScript: Every correct Source program should behave *exactly* the same using a Source implementation, as it does using a JavaScript implementation. We assume, of course, that suitable libraries are used by the JavaScript implementation, to account for the predefined names of each Source language. This section lists some exceptions where we think a Source implementation should be allowed to deviate from the JavaScript specification, for the sake of internal consistency and esthetics.

Evaluation result of programs: JavaScript statically distinguishes between *value-producing* and *non-value-producing statements*. All declarations are non-value-producing, and all expression statements, conditional statements and assignments are value-producing. A block is value-producing if its body statement is value-producing, and then its value is the value of its body statement. A sequence is value-producing if any of its component statements is value-producing, and then its value is the value of its *last* value-producing component statement. The value of an expression statement is the value of the expression. The value of a conditional statement is the value of the branch that gets executed, or the value `undefined` if that branch is not value-producing. The value of an assignment is the value of the expression to the right of its `=` sign. Finally, if the whole program is not value-producing, its value is the value `undefined`.

Example 1:

```
1;
{
  // empty block
}
```

The result of evaluating this program in JavaScript is 1.

Example 2:

```
1;
{
  if (true) {} else {}
}
```

The result of evaluating this program in JavaScript is `undefined`.

Implementations of Source are currently allowed to opt for a simpler scheme.

Hoisting of function declarations: In JavaScript, function declarations are "hoisted" (automagically moved) to the beginning of the block in which they appear. This means that applications of functions that are declared with function declaration statements never fail because the name is not yet assigned to their function value. The specification of Source does not include this hoisting; in Source, function declaration can be seen as syntactic sugar for constant declaration and lambda expression. As a consequence, application of functions declared with function declaration may fail in Source if the name that appears as function expression is not yet assigned to the function value it is supposed to refer to.