Specification of JavaScript §1—2021 edition

Martin Henz, Lee Ning Yuan, Daryl Tan

National University of Singapore
School of Computing

September 22, 2026

The language JavaScript is the official language of the textbook *Structure and Interpretation of Computer Programs*, JavaScript Adaptation. This language is a sublanguage of ECMAScript 2018 (9th Edition) and defined in the documents titled "JavaScript §x", where x refers to the respective textbook chapter.

1 Syntax

A JavaScript 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 list
import-name	::=	*name* | *name* **as** *name*	import name
statement	::=	**const** *name* = *expression* ;	constant declaration
	|	**function** *name* (*names*) *block*	function declaration
	|	**return** *expression* ;	return statement
	|	*if-statement*	conditional statement
	|	*block*	block statement
	|	*expression* ;	expression statement
	|	**debugger** ;	breakpoint
names	::=	ϵ | *name* (, *name*)...	name list
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 binary-logical expression*	logical composition
	|	*expression* (*expressions*)	function application
	|	(*name* | (*names*)) => *expression*	lambda expression (expr. body)
	|	(*name* | (*names*)) => *block*	lambda expression (block body)
	|	*expression* ? *expression* : *expression*	conditional expression
	|	(*expression*)	parenthesised expression
binary-operator	::=	+ | - | * | / | % | === | !==	
	|	> | < | >= | <=	binary operator
unary-operator	::=	! | -	unary operator
binary-logical	::=	&& | ||	logical composition symbol
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 JavaScript are allowed to treat function declaration as syntactic sugar for constant declaration.[4] JavaScript 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 JavaScript specifications do not specify how modules are programmed.

Logical Composition

Conjunction

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

stands for

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

Disjunction

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

stands for

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

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`.

[2] This is a syntactic quirk inherited from 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 a full JavaScript implementation but might encounter a runtime error "Cannot access name before initialization" here.
[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.

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

2 Dynamic Type Checking

Expressions evaluate to numbers, boolean values, strings or function values. Implementations of JavaScript 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 JavaScript'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`, JavaScript only allows boolean expressions.

3 Standard Library

The standard library contains constants and functions that are always available in this language. The functions indicated as *primitive* are built into the language implementations. All others are considered *predeclared* and implemented using the primitive functions.
The standard library for JavaScript § 1 is documented in online documentation of JavaScript § 1.

Deviations from JavaScript

We intend this JavaScript sublanguage to be a conservative extension of JavaScript: Every correct program in this sublanguage should behave *exactly* the same using an implementation of this sublanguage, as it does using a full JavaScript implementation. We assume, of course, that suitable libraries are used by the full JavaScript implementation, to account for the predefined names of each JavaScript sublanguage.
This section lists some exceptions where we think an implementation of this sublanguage 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 this sublanguage 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 this sublanguage does not include this hoisting; here, 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 here if the name that appears as function expression is not yet assigned to the function value it is supposed to refer to.