Specification of Source §1—1920 edition

Martin Henz

National University of Singapore
School of Computing

March 5, 2020

The language Source is the official language of the textbook *Structure and Interpretation of Computer Programs, JavaScript Adaptation*. You have never heard of Source? No worries! It was invented just for the purpose of the book. 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. For example, Source §3 is suitable for textbook Chapter 3 and the preceeding chapters.

Programs

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, and $x \ldots$ for zero or more repetitions of x.

program	::=	*statement* ...	statement sequence
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	::=	**{** *program* **}**	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*	function definition (expr. body)
	|	(*name* | **(** *parameters* **)**) **=>** *block*	function definition (block body)
	|	*expression* **?** *expression* **:** *expression*	conditional expression
	|	**(** *expression* **)**	parenthesised expression
binary-operator	::=	**+** | **-** | ***** | **/** | **%** | **===** | **!==**	
	|	**>** | **<** | **>=** | **<=** | **&&** | **||**	binary operator
unary-operator	::=	**!** | **-**	unary operator
expressions	::=	ϵ | *expression* (**,** *expression*) ...	argument expressions

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

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

Restrictions

- Return statements are only allowed in bodies of functions.
- There cannot be any newline character between **`return`** and *expression* in return statements.
- There cannot be any newline character between (*name* | **`(`** *parameters* **`)`**) and **`=>`** in function definition expressions.
- Functions must not be called before their corresponding function declaration is evaluated.

Names

Names[2] start with _, `$` or a letter[3] and contain only _, `$`, letters or digits[4]. Reserved words[5] such as keywords 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.
In addition to names that are declared using **`const`**, **`function`**, **`=>`** (and **`let`** in Source §3 and 4), the following names refer to primitive functions and constants:

- `math_`*name*, where *name* is any name specified in the JavaScript `Math` library, see ECMAScript Specification, Section 20.2. Examples:
  - `math_PI`: Refers to the mathematical constant π,
  - `math_sqrt(n)`: Returns the square root of the *number* `n`.
- `runtime()`: Returns number of milliseconds elapsed since January 1, 1970 00:00:00 UTC
- `parse_int(s, i)`: 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`: Refer to JavaScript's undefined, NaN ("Not a Number") and Infinity values, respectively.
- `is_boolean(x)`, `is_number(x)`, `is_string(x)`, `is_function(x)`: return `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`.

[2] In ECMAScript 2018 (9th Edition), these names are called *identifiers*.
[3] By *letter* we mean Unicode letters (L) or letter numbers (Nl).
[4] 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.
[5] By *Reserved word* we mean any of: **`break`**, **`case`**, **`catch`**, **`continue`**, **`debugger`**, **`default`**, **`delete`**, **`do`**, **`else`**, **`finally`**, **`for`**, **`function`**, **`if`**, **`in`**, **`instanceof`**, **`new`**, **`return`**, **`switch`**, **`this`**, **`throw`**, **`try`**, **`typeof`**, **`var`**, **`void`**, **`while`**, **`with`**, **`class`**, **`const`**, **`enum`**, **`export`**, **`extends`**, **`import`**, **`super`**, **`implements`**, **`interface`**, **`let`**, **`package`**, **`private`**, **`protected`**, **`public`**, **`static`**, **`yield`**, **`null`**, **`true`**, **`false`**.

- `prompt(s)`: 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)`: Displays the value `x` in the console[6]; returns the argument `a`.
- `display(x, s)`: Displays the string `s`, followed by a space character, followed by the value `x` in the console[6]; returns the argument `x`.
- `error(x)`: Displays the value `x` in the console[6] with error flag. The evaluation of any call of `error` aborts the running program immediately.
- `error(x, s)`: Displays the string `s`, followed by a space character, followed by the value `x` in the console[6] with error flag. The evaluation of any call of `error` aborts the running program immediately.
- `stringify(x)`: returns a string that represents[6] the value `x`.

All Source primitive functions, except `stringify`, 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 argument.

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 of the form `'` *single-quote-characters*`'`, where *single-quote-characters* is a possibly empty sequence of characters without the character `'`,

Typing

Expressions evaluate to numbers, boolean values, strings or function values.
Only function values can be applied using the syntax:

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

The following table specifies what arguments Source's operators take and what results they return.

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

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.

Comments

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

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.

Empty block as last statement of toplevel sequence: In JavaScript, empty blocks as last statement of a sequence are apparently ignored. Thus the result of evaluating such a sequence is the result of evaluating the previous statement. Implementations of Source might stick to the more intuitive result: `undefined`. Example:

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

The result of evaluating this program can be `undefined` for implementations of Source. Note that this issue only arises at the toplevel—outside of functions.