Background for Studio Week 2

Martin Henz

National University of Singapore
School of Computing

October 21, 2025

1 Concepts covered

- Primitive values: numbers, booleans (no strings)
- Conditional expressions (no conditional statements)
- Function declarations (body is a single return statement, no lambda expressions)
- No recursion (if anyone asks, refer them to L2)

The following reference is for Avenger reference, not to be shared with students. We do not expect Avengers to cover all features in the Studio. But please not to cover anything that is not mentioned.

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`** `{` *names* `}` **`from`** *string* `;`	import directive
statement	::=	**`const`** *name* `=` *expression* `;`	constant declaration
	|	**`function`** *name* `(` *parameters* `)` `{` **`return`** *expression*; `}`	function declaration
	|	*expression* `;`	expression statement
parameters	::=	ε | *name* (`,` *name*)...	function parameters
expression	::=	*number*	primitive number expression
	|	*string*	primitive string expression
	|	**`true`** | **`false`**	primitive boolean expression
	|	*name*	name expression
	|	*expression binary-operator expression*	binary operator combination
	|	*unary-operator expression*	unary operator combination
	|	*expression binary-logical expression*	logical composition
	|	*name* `(` *expressions* `)`	function application
	|	*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

- There cannot be any newline character between **`return`** and *expression* in return statements.[2]

Logical Composition

Conjunction

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

stands for

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

Disjunction

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

stands for

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

Names

Names[3] start with _, `$` or a letter[4] and contain only _, `$`, letters or digits[5]. Restricted words[6] 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`.

Comments

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

3 Dynamic Type Checking

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

$$\textit{expression} \quad ::= \quad \textit{name}\,\texttt{(}\ \textit{expressions}\ \texttt{)}$$

For compound functions, implementations need to check that the number of *expressions* matches the number of parameters.

[2] Source inherits this syntactic quirk of JavaScript.
[3] In ECMAScript 2020 (9th Edition), these names are called *identifiers*.
[4] By *letter* we mean Unicode letters (L) or letter numbers (Nl).
[5] 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.
[6] 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.

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
`-`	number	number	number
`*`	number	number	number
`/`	number	number	number
`%`	number	number	number
`===`	number	number	bool
`!==`	number	number	bool
`>`	number	number	bool
`<`	number	number	bool
`>=`	number	number	bool
`<=`	number	number	bool
`&&`	bool	any	any
`||`	bool	any	any
`!`	bool		bool
`-`	number		number

Preceding **`?`**, Source only allows boolean expressions.

4 Standard Libraries

The following library is always available in this language.

MISC Library

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`.
- `NaN`, `Infinity`: Refer to JavaScript's NaN ("Not a Number") and Infinity values, respectively.
- `is_boolean(x)`, `is_number(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`.

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.

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 (in Studio 2: the beginning of the program, because there are no blocks yet). 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. 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.