Specification of Source §3 Typed—2023 edition

Martin Henz, Jingjing Zhao

National University of Singapore
School of Computing

October 21, 2025

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 §3 Typed is a variant of Source §3 that introduces type syntax and type checking.

1 Syntax

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

program	::=	*import-directive... type-alias... 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
type-alias	::=	**type** *name*[< *name* (**,** *name*)... >] **=** *alias-type***;**	type alias declaration
statement	::=	**const** *name*[**:** *type*] **=** *expression* **;**	constant declaration
	|	*let* **;**	variable declaration
	|	**function** *name* **(** *rest-names* **)** [**:** *type*] *block*	function declaration
	|	**return** *expression* **;**	return statement
	|	*if-statement*	conditional statement
	|	**while (** *expression* **)** *block*	while loop
	|	**for (** (*expression*$_1$ | *let*)**;**	
		expression$_2$ **;**	
		expression$_3$ **)** *block*	for loop
	|	**break ;**	break statement
	|	**continue ;**	continue statement
	|	*block*	block statement
	|	*expression* **;**	expression statement
	|	**debugger ;**	breakpoint
names	::=	ϵ | *name*[**:** *type*] (**,** *name*[**:** *type*])...	name list
typed-names	::=	ϵ | *name***:** *type* (**,** *name***:** *type*)...	name list (typed)
rest-names	::=	ϵ | **. . .** *name* | *names* [**,** **. . .** *name*]	name list (rest)
if-statement	::=	**if (** *expression* **)** *block*	
		[**else** (*block* | *if-statement*)]	conditional statement
block	::=	**{** *statement...* **}**	block statement
let	::=	**let** *name*[**:** *type*] **=** *expression*	variable declaration

[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 | 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.

assignment	::=	*name* **=** *expression*	variable assignment
	|	*expression*[*expression*] **=** *expression*	array assignment
expression	::=	*number*	primitive number expression
	|	**true** | **false**	primitive boolean expression
	|	*string*	primitive string expression
	|	**null**	primitive list expression
	|	*name*	name expression
	|	*expression binary-operator expression*	binary operator combination
	|	*unary-operator expression*	unary operator combination
	|	*expression binary-logical expression*	logical composition
	|	*expression* **(** *spread-expressions* **)**	function application
	|	(*rest-names*) **=>** *expression*	lambda expression (expr. body)
	|	(*rest-names*) **=>** *block*	lambda expression (block body)
	|	*expression* **?** *expression* **:** *expression*	conditional expression
	|	*assignment*	assignment
	|	*expression*[*expression*]	array access
	|	**[** *expressions* **]**	literal array expression
	|	**(** *expression* **)**	parenthesised expression
	|	*expression* **as** *type*	as expression
binary-operator	::=	**+** | **-** | ***** | **/** | **%** | **===** | **!==**	
	|	**>** | **<** | **>=** | **<=**	binary operator
unary-operator	::=	**!** | **-**	unary operator
binary-logical	::=	**&&** | **||**	logical composition symbol
expressions	::=	ϵ | *expression* (**,** *expression*)...	element expressions
spread-expressions	::=	ϵ | *spread-expression* (**,** *spread-expression*)...	argument expressions
spread-expression	::=	*expression* | **...** *expression*	argument expression (spread)
type	::=	`number` | `boolean` | `string`	
		| `undefined` | `null` | `void` | `any`	basic type
	|	*number* | *string* | **true** | **false**	literal type
	|	*name*[< *type* (**,** *type*)... >]	type reference
	|	(*typed-names*) **=>** *type*	function type
	|	*type* **|** *type*	union type
	|	*Pair<type, type>*	pair type
	|	*List<type>*	list type
	|	*type[]*	array type
alias-type	::=	`number` | `boolean` | `string`	
		| `undefined` | `null` | `void` | `any`	basic type
	|	*number* | *string* | **true** | **false**	literal type
	|	*name*[< *alias-type* (**,** *alias-type*)... >]	type reference
	|	(*typed-names*) **=>** *alias-type*	function type
	|	*alias-type* **|** *alias-type*	union type
	|	*Pair<alias-type, alias-type>*	pair type
	|	*List<alias-type>*	list type
	|	*alias-type[]*	array type
	|	*name*	type parameter

Restrictions

- Return statements are only allowed in bodies of functions.
- There cannot be any newline character between **`return`** and *expression* in return state-

ments.[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 Source specifications do not specify how modules are programmed.

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

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

Loops

while-loops

Roughly speaking, while loops are seen as abbreviations for function applications as follows:

while (*expression*) *block*

stands for

```
function _body() { block }
_while( () => expression , _body );
```

where `_while` is defined as follows:

```
function _while(test, body) {
    if (test()) {
        body();
        _while(test, body);
    } else {
        undefined;
    }
}
```

[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.

Simple for-loops

for (*assignment*$_1$; *expression* ; *assignment*$_2$) *block*

stands for

```
assignment₁
while (expression) {
    block
    assignment₂
}
```

for-loops with loop control variable

for (**let** *name* = *expression*$_1$; *expression*$_2$; *assignment*) *block*

stands for

```
{
    let name = expression₁;
    for (name = name; expression₂; assignment) {
        const _copy_of_name = name;
        {
            const name = _copy_of_name;
            block
        }
    }
}
```

Return values, `break` and `continue`

Contrary to the simplified explanation above, **while** and **for** loops return the value of their last loop execution, or `undefined` if there is no loop execution. Evaluation of a **break** statement within a loop terminates the loop with the return value `undefined` and evaluation of a **continue** statement within a loop terminates the current loop iteration and evaluates the test.

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`.

[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'`.

Arrays

Arrays in Source are created using literal array expressions:

```
let my_array_1 = [];
let my_array_2 = [42, 71, 13];
```

Array access of the form a`[`i`]` has constant time complexity $\Theta(1)$. Array assignment the form a`[`n`] =` x has a time complexity $O(n)$.

Comments

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

2 Type System

In Source §3 Typed, the Source §3 syntax is expanded to include type syntax such as type annotations and type aliases. This allows names to be explicitly typed, and for type checks to be performed.
Support for `typeof` operations is also added to Source §3 Typed.

2.1 Type Environment

In order to keep track of the type of names in a program, we define a *type environment*, denoted by Γ. More formally, the partial function Γ from names to types expresses a context, in which a name x is associated with type $\Gamma(x)$.
We define a relation $\Gamma[x \leftarrow t]\Gamma'$ on type environments Γ, names x, types t, and type environments Γ', which constructs a type environment that behaves like the given one, except that the type of x is t. More formally, if $\Gamma[x \leftarrow t]\Gamma'$, then $\Gamma'(y)$ is t, if $y = x$ and $\Gamma(y)$ otherwise. Obviously, this uniquely identifies Γ' for a given Γ, x, and t, and thus the type environment extension relation is functional in its first three arguments.

The set of names, on which a type environment Γ is defined, is called the domain of Γ, denoted by *dom*(Γ).
For each non-overloaded primitive operator, we add a binding to our initial type environment Γ_0 as follows:

$$\begin{aligned}
&\emptyset[-_2 \leftarrow (\texttt{number}, \texttt{number}) \rightarrow \texttt{number}]\\
&\ [* \leftarrow (\texttt{number}, \texttt{number}) \rightarrow \texttt{number}]\\
&\ [/ \leftarrow (\texttt{number}, \texttt{number}) \rightarrow \texttt{number}]\\
&\ [\% \leftarrow (\texttt{number}, \texttt{number}) \rightarrow \texttt{number}]\\
&\ [\&\& \leftarrow (\texttt{boolean}, \texttt{T}) \rightarrow \texttt{boolean} \mid \texttt{T}]\\
&\ [|| \leftarrow (\texttt{boolean}, \texttt{T}) \rightarrow \texttt{boolean} \mid \texttt{T}]\\
&\ [! \leftarrow \texttt{boolean} \rightarrow \texttt{boolean}]\\
&\ [-_1 \leftarrow \texttt{number} \rightarrow \texttt{number}]\\
&\ [\texttt{typeof} \leftarrow \texttt{any} \rightarrow \texttt{string}]\Gamma_{-2}
\end{aligned}$$

The overloaded binary primitives (with the exception of $+$, the handling of which will be elaborated in Typing Relations) are handled as follows:

$$\begin{aligned}
&\Gamma_{-2}[=== \leftarrow (\texttt{any},\ \texttt{any}) \rightarrow \texttt{boolean}]\\
&\qquad [!== \leftarrow (\texttt{any},\ \texttt{any}) \rightarrow \texttt{boolean}]\\
&\qquad [> \leftarrow (\texttt{string},\ \texttt{string}) \rightarrow \texttt{boolean} \mid (\texttt{number},\ \texttt{number}) \rightarrow \texttt{boolean}]\\
&\qquad [>= \leftarrow (\texttt{string},\ \texttt{string}) \rightarrow \texttt{boolean} \mid (\texttt{number},\ \texttt{number}) \rightarrow \texttt{boolean}]\\
&\qquad [< \leftarrow (\texttt{string},\ \texttt{string}) \rightarrow \texttt{boolean} \mid (\texttt{number},\ \texttt{number}) \rightarrow \texttt{boolean}]\\
&\qquad [<= \leftarrow (\texttt{string},\ \texttt{string}) \rightarrow \texttt{boolean} \mid (\texttt{number},\ \texttt{number}) \rightarrow \texttt{boolean}]\Gamma_{-1}
\end{aligned}$$

The Source §3 standard library functions and constants have their types defined as follows:

Γ_{-1}	[	display	←	any			]
	[	error	←	any			]
	[	Infinity	←	number			]
	[	is_boolean	←	any	→	boolean	]
	[	is_function	←	any	→	boolean	]
	[	is_number	←	any	→	boolean	]
	[	is_string	←	any	→	boolean	]
	[	is_undefined	←	any	→	boolean	]
	[	math_abs	←	number	→	number	]
	[	math_acos	←	number	→	number	]
	[	math_acosh	←	number	→	number	]
	[	math_asin	←	number	→	number	]
	[	math_asinh	←	number	→	number	]
	[	math_atan	←	number	→	number	]
	[	math_atan2	←	(number, number)	→	number	]
	[	math_atanh	←	number	→	number	]
	[	math_cbrt	←	number	→	number	]
	[	math_ceil	←	number	→	number	]
	[	math_clz32	←	number	→	number	]
	[	math_cos	←	number	→	number	]
	[	math_cosh	←	number	→	number	]
	[	math_exp	←	number	→	number	]
	[	math_expm1	←	number	→	number	]
	[	math_floor	←	number	→	number	]
	[	math_fround	←	number	→	number	]
	[	math_hypot	←	any			]
	[	math_imul	←	(number, number)	→	number	]
	[	math_LN2	←	number			]
	[	math_LN10	←	number			]

[	`math_log`	←	`number`	→	`number`	]
[	`math_log1p`	←	`number`	→	`number`	]
[	`math_log2`	←	`number`	→	`number`	]
[	`math_LOG2E`	←	`number`			]
[	`math_log10`	←	`number`	→	`number`	]
[	`math_LOG10E`	←	`number`			]
[	`math_max`	←	`any`			]
[	`math_min`	←	`any`			]
[	`math_PI`	←	`number`			]
[	`math_pow`	←	(`number`, `number`)	→	`number`	]
[	`math_random`	←	()	→	`number`	]
[	`math_round`	←	`number`	→	`number`	]
[	`math_sign`	←	`number`	→	`number`	]
[	`math_sin`	←	`number`	→	`number`	]
[	`math_sinh`	←	`number`	→	`number`	]
[	`math_sqrt`	←	`number`	→	`number`	]
[	`math_SQRT1_2`	←	`number`			]
[	`math_SQRT2`	←	`number`			]
[	`math_tan`	←	`number`	→	`number`	]
[	`math_tanh`	←	`number`	→	`number`	]
[	`math_trunc`	←	`number`	→	`number`	]
[	`NaN`	←	`number`			]
[	`parse_int`	←	(`string`, `number`)	→	`number`	]
[	`prompt`	←	`string`	→	`string`	]
[	`get_time`	←	()	→	`number`	]
[	`stringify`	←	`any`	→	`string`	]
[	`undefined`	←	`undefined`			]
[	`null`	←	`null`			]
[	`pair`	←	$(T_1,\ T_2)$	→	*Pair*$< T_1,\ T_2 >$	]
[	`head`	←	*Pair*$< T_1,\ T_2 >$	→	T_1	]
[	`tail`	←	*Pair*$< T_1,\ T_2 >$	→	T_2	]
[	`list`	←	$(T_1, \ldots, T_n)$	→	*List*$< T_1 \mid \ldots \mid T_n >$	]
[	`is_pair`	←	`any`	→	`boolean`	]
[	`is_null`	←	`any`	→	`boolean`	]
[	`is_list`	←	`any`	→	`boolean`	]
[	`set_head`	←	*Pair*$< T_1,\ T_2 >$	→	`undefined`	]
[	`set_tail`	←	*Pair*$< T_1,\ T_2 >$	→	`undefined`	]
[	`is_array`	←	`any`	→	`boolean`	]
[	`array_length`	←	`any`[]	→	`number`	] Γ_0

In order to support the definition of type aliases, we define a separate *type alias environment*, denoted by Γ_{alias}. Unlike Γ, Γ_{alias} binds names to special *type functions* of the form $< T_1, \ldots, T_n > \to t$ where $T_1 \ldots T_n$ are type parameters t is the return type expressed in terms of $T_1 \ldots T_n$. $<>$ is used to differentiate type functions from function types, which are of the form $(t_1, \ldots, t_n) \to t$.
Since Γ and Γ_{alias} are separate environments, the same name x can be used for both variables and type aliases.
The initial type alias environment Γ_{alias0} is as follows:

$\Gamma_{alias-1}$	[	`Pair`	←	$< T_{head}, T_{tail} >$	→	*Pair* $< T_{head}, T_{tail} >$	]
	[	`List`	←	$< T_{elem} >$	→	*List* $< T_{elem} >$	] Γ_{alias0}

2.2 Success Types

In order for type checks to be performed in Source §3 Typed, we introduce the notion of success types.
We first define the special `any` type:

Definition 2.1 *`any` is the union of all possible types.*

Success typing in Source Typed is defined as follows:

Definition 2.2 *Type t' is a success type of type t if $\exists x (x \in t \wedge x \in t')$. Alternatively: $t \wedge t' \neq \emptyset$.*

In Source Typed, type checks are performed by checking that the actual type is a success type of the expected type. This means that type errors will be thrown if and only if a definite clash in types at runtime is detected. Given that `any` is the union of all possible types, this also means that the `any` type is guaranteed not to produce any type errors.

2.3 Typing Relations

To perform type checking on the program, typing relations are applied to every statement and expression in the program.
Names that do not have a type declared will be assumed to have the `any` type.

2.3.1 Typing Relations on Expressions

The derived type of primitive expressions is their literal type, which is an element of its corresponding basic type.

$$\frac{}{\Gamma, \Gamma_{alias} \vdash n : \textit{literal type } n} \qquad \frac{}{\Gamma, \Gamma_{alias} \vdash s : \textit{literal type } s}$$

where n denotes any literal number and s denotes any literal string.

$$\frac{}{\Gamma, \Gamma_{alias} \vdash \texttt{true} : \textit{literal type } \texttt{true}} \qquad \frac{}{\Gamma, \Gamma_{alias} \vdash \texttt{false} : \textit{literal type } \texttt{false}}$$

For names, the type must be derived from the type environment.

$$\frac{}{\Gamma, \Gamma_{alias} \vdash x : \Gamma(x)}$$

For function applications (including applications of binary and unary operators), the following two type rules are used, depending on the type of E_0.

$$\frac{\Gamma, \Gamma_{alias} \vdash E_0 : (t_1, \ldots, t_n) \to t \quad \Gamma, \Gamma_{alias} \vdash E_1 : t'_1, \ldots, \Gamma, \Gamma_{alias} \vdash E_n : t'_n \quad (\forall 1 \le i \le n)(t'_i \wedge t_i \neq \emptyset)}{\Gamma, \Gamma_{alias} \vdash E_0 \texttt{ (} E_1, \ldots, E_n \texttt{) } : t}$$

$$\frac{\Gamma, \Gamma_{alias} \vdash E_0 : \texttt{any} \quad \Gamma, \Gamma_{alias} \vdash E_1 : t'_1, \ldots, \Gamma, \Gamma_{alias} \vdash E_n : t'_n}{\Gamma, \Gamma_{alias} \vdash E_0 \texttt{ (} E_1, \ldots, E_n \texttt{) } : \texttt{any}}$$

The type of the operator must be a function type with the right number of parameters, and the type of every argument must be a success type of the corresponding parameter type of the function type. If all of the conditions are met, the type of the function application is the same as the return type of the function type that is the type of the operator. If the type of the operator is `any`, the return type will be `any`.
Applications of binary and unary operators are treated the same as function applications, with the exception of the + operator. We use the $\subseteq$ operator to indicate that a type is a subset of another type, as defined below:

- A type is a subset of type `number` if it is of type `number`, literal number type, or a union type containing any number of literal number types.
- A type is a subset of type `string` if it is of type `string`, literal string type, or a union type containing any number of literal string types.

For the + operator, the following rules are applied, in order of priority:

1. If the expression on the left side is a subset of type `number`, check that the other expression is a success type of `number`. The return type is `number`.
2. If the expression on the left side is a subset of type `string`, check that the other expression is a success type of `string`. The return type is `string`.
3. If the expression on the right side is a subset of type `number`, check that the other expression is a success type of `number`. The return type is `number`.
4. If the expression on the right side is a subset of type `string`, check that the other expression is a success type of `string`. The return type is `string`.
5. If the expression on the left side cannot be narrowed to a subset of either `number` or `string`, check that both sides are success types of `number | string`. The return type is `number | string`.

$$\frac{\Gamma, \Gamma_{alias} \vdash E_0 : t_0 \quad \Gamma, \Gamma_{alias} \vdash E_1 : t_1 \quad t_0 \subseteq \texttt{number} \quad t_1 \wedge \texttt{number} \neq \emptyset}{\Gamma, \Gamma_{alias} \vdash E_0 \texttt{ + } E_1 : \texttt{number}}$$

$$\frac{\Gamma, \Gamma_{alias} \vdash E_0 : t_0 \quad \Gamma, \Gamma_{alias} \vdash E_1 : t_1 \quad t_0 \subseteq \texttt{string} \quad t_1 \wedge \texttt{string} \neq \emptyset}{\Gamma, \Gamma_{alias} \vdash E_0 \texttt{ + } E_1 : \texttt{string}}$$

$$\frac{\Gamma, \Gamma_{alias} \vdash E_0 : t_0 \quad \Gamma, \Gamma_{alias} \vdash E_1 : t_1 \quad t_1 \subseteq \texttt{number} \quad t_0 \wedge \texttt{number} \neq \emptyset}{\Gamma, \Gamma_{alias} \vdash E_0 \texttt{ + } E_1 : \texttt{number}}$$

$$\frac{\Gamma, \Gamma_{alias} \vdash E_0 : t_0 \quad \Gamma, \Gamma_{alias} \vdash E_1 : t_1 \quad t_1 \subseteq \texttt{string} \quad t_0 \wedge \texttt{string} \neq \emptyset}{\Gamma, \Gamma_{alias} \vdash E_0 \texttt{ + } E_1 : \texttt{string}}$$

$$\frac{\Gamma, \Gamma_{alias} \vdash E_0 : t_0 \quad \Gamma, \Gamma_{alias} \vdash E_1 : t_1 \quad t_0 \wedge (\texttt{number} \mid \texttt{string}) \neq \emptyset \quad t_1 \wedge (\texttt{number} \mid \texttt{string}) \neq \emptyset}{\Gamma, \Gamma_{alias} \vdash E_0 \texttt{ + } E_1 : \texttt{number} \mid \texttt{string}}$$

For lambda expressions, we temporarily extend Γ with the declared types of all the function parameters, and check the type of the function body against the declared return type. As type syntax is optional, if type annotations are absent for any of the arguments or the return type, the type is assumed to be `any`. The type of the lambda expression is then the function type with the declared types of the parameters and the return type.

$$\frac{\Gamma[x_1 \leftarrow t_1] \cdots [x_n \leftarrow t_n] \vdash S : t' \quad t' \wedge t \neq \emptyset}{\Gamma, \Gamma_{alias} \vdash (x_1 \texttt{ : } t_1, \ldots, x_n \texttt{ : } t_n) \texttt{ : } t \texttt{ => } S : (t_1, \ldots, t_n) \to t}$$

The type of a conditional expression is the union of the type of its consequent expression and its alternate expression. The predicate expression of a conditional expression must be a success type of a boolean.

$$\frac{\Gamma, \Gamma_{alias} \vdash E_{pred} : t_{pred} \quad \Gamma, \Gamma_{alias} \vdash E_{cons} : t_{cons} \quad \Gamma, \Gamma_{alias} \vdash E_{alt} : t_{alt} \quad t_{pred} \wedge \texttt{boolean} \neq \emptyset}{\Gamma, \Gamma_{alias} \vdash E_{pred} \texttt{ ? } E_{cons} \texttt{ : } E_{alt} : t_{cons} \mid t_{alt}}$$

For as expressions, the type to cast the expression to must be a success type of the type of the expression.

$$\frac{\Gamma, \Gamma_{alias} \vdash E : t' \quad t \wedge t' \neq \emptyset}{\Gamma, \Gamma_{alias} \vdash E \texttt{ as } t : t}$$

2.3.2 Typing Relations on Assignments

For assignments, the type of the right expression must be a success type of the type of the left expression. The type of the assignment itself is the type of the right expression.

$$\frac{\Gamma, \Gamma_{alias} \vdash E_0 : t_0 \quad \Gamma, \Gamma_{alias} \vdash E_1 : t_1 \quad t_1 \wedge t_0 \neq \emptyset}{\Gamma, \Gamma_{alias} \vdash E_0 = E_1 : t_1}$$

2.3.3 Typing Relations on Statements

Sequences in the top level are handled using the following steps:

1. Type alias declarations are evaluated, which adds type aliases to Γ_{alias} to construct Γ'_{alias}.
2. The declared types of constant/variable declarations are added to Γ to construct Γ'. Note that the declaration statements themselves are yet to be checked.
3. All statements are checked using Γ' and Γ'_{alias}.
4. The type of the sequence is the type of the last value-producing statement.

In the below rule, D_n denotes constant/variable declarations of the form `const/let` $x_n : t_n = E_n$`;`. If the type annotation for x_n is absent, the declared type t_n is assumed to be `any`.

$$\frac{\begin{array}{l}\Gamma_{alias} \vdash A_1 : \Gamma_{alias1}, \ldots, \Gamma_{aliasm-1} \vdash A_m : \Gamma'_{alias} \quad \Gamma[x_1 \leftarrow t_1] \cdots [x_n \leftarrow t_n]\Gamma' \\ \Gamma', \Gamma'_{alias} \vdash D_1 : t_1, \ldots, \Gamma', \Gamma'_{alias} \vdash D_n : t_n \quad \Gamma', \Gamma'_{alias} \vdash S_1 : t'_1, \ldots, \Gamma', \Gamma'_{alias} \vdash S_p : t'_p\end{array}}{\Gamma, \Gamma_{alias} \vdash \{A_1, \ldots, A_m, D_1, \ldots, D_n, S_1, \ldots, S_p\} : t'_p, \Gamma', \Gamma'_{alias}}$$

For type alias declarations, the declared type t for type alias name T is first checked against the type environments. Any type parameters declared are temporarily added to the type alias environment when checking the type of t to ensure that the type parameters are only used within t itself. Then, the binding of T to type function $< T_1, \ldots, T_n >\rightarrow t$ is added to the type alias environment. If no type parameters are given, the type function is assumed to take in 0 type arguments.

$$\frac{\Gamma, \Gamma_{alias} \vdash t : t \quad \Gamma_{alias}[T \leftarrow <> \rightarrow t]\Gamma'_{alias}}{\Gamma, \Gamma_{alias} \vdash \texttt{type } T = t\texttt{;} : \texttt{undefined}, \Gamma'_{alias}}$$

$$\frac{\Gamma, \Gamma_{alias}[T_1 \leftarrow T_1] \ldots [T_n \leftarrow T_n] \vdash t : t \quad \Gamma_{alias}[T \leftarrow < T_1, \ldots, T_n > \rightarrow t]\Gamma'_{alias}}{\Gamma, \Gamma_{alias} \vdash \texttt{type } T < T_1, \ldots, T_n >= t\texttt{;} : \texttt{undefined}, \Gamma'_{alias}}$$

For constant and variable declarations, the declared type t is retrieved from the type environment. If the declared type is a type reference to a type alias with name T, t is obtained by applying the type arguments $t_1, \ldots, t_n$ to the type function for T, replacing all instances of type variables $T_1, \ldots, T_n$ in t with $t_1, \ldots, t_n$ respectively.

The derived type of the expression E, t_E, must be a success type of t. The type of the statement itself is `undefined`.

$$\frac{\Gamma \vdash E : t_E \quad t_E \wedge t \neq \emptyset}{\Gamma, \Gamma_{alias} \vdash \texttt{const/let}\ x\texttt{:}\ t = E\texttt{;} : \texttt{undefined}}$$

$$\frac{\Gamma_{alias}(T) < t_1, \ldots, t_n >= t \quad \Gamma \vdash E : t_E \quad t_E \wedge t \neq \emptyset}{\Gamma, \Gamma_{alias} \vdash \texttt{const/let}\ x\texttt{:}\ T < t_1, \ldots, t_n >= E\texttt{;} : \texttt{undefined}}$$

The type of return statements and expression statements is the type of the expression in the statement.

$$\frac{\Gamma, \Gamma_{alias} \vdash E : t}{\Gamma, \Gamma_{alias} \vdash \texttt{return}\ E\texttt{;} : t} \qquad \frac{\Gamma, \Gamma_{alias} \vdash E : t}{\Gamma, \Gamma_{alias} \vdash E\texttt{;} : t}$$

The type of break and continue statements is `undefined`.

$$\frac{}{\Gamma, \Gamma_{alias} \vdash \texttt{break;} : \texttt{undefined}} \qquad \frac{}{\Gamma, \Gamma_{alias} \vdash \texttt{continue;} : \texttt{undefined}}$$

For blocks, Γ is first extended temporarily to include the types of names declared in the block. Then, the component statements are checked against the extended type environment.
For function body blocks and if statement blocks, we assume that whenever there is a return statement or a conditional statement with a return statement within a block, it is the last statement in the block. (One could consider a "dead code" error otherwise.)
The type of a function body or if statement block is the type of the return statement in the block. If the block does not contain any return statements, the type is `void`, which is a special type that is used to denote the return type of a function that does not return anything, and changes to `undefined` if unioned with another type that is not `void`.
In the below rule, D_n denotes constant/variable declarations of the form `const/let` x_n`:` $t_n = E_n$`;`.
If the type annotation for x_n is absent, the declared type t_n is assumed to be `any`.

$$\frac{\Gamma[x_1 \leftarrow t_1] \cdots [x_m \leftarrow t_m]\Gamma_{temp} \quad \Gamma_{temp} \vdash D_1 : t_1, \ldots, \Gamma_{temp} \vdash D_m : t_m \quad \Gamma_{temp} \vdash S_1 : t'_1, \ldots, \Gamma_{temp} \vdash S_n : t'_n}{\Gamma, \Gamma_{alias} \vdash \{D_1, \ldots, D_m, S_1, \ldots, S_n\} : \begin{cases} t'_n & S_n \textit{ is a return statement} \\ \texttt{void} & S_n \textit{ is not a return statement} \end{cases}}$$

The type of a block that is not a function body or if statement block is the type of last value-producing statement in the block.
In the below rule, D_n denotes constant/variable declarations of the form `const/let` x_n`:` $t_n = E_n$`;`.
If the type annotation for x_n is absent, the declared type t_n is assumed to be `any`. We also assume that S_k is the last value-producing statement in the block.

$$\frac{\Gamma[x_1 \leftarrow t_1] \cdots [x_m \leftarrow t_m]\Gamma_{temp} \quad \Gamma_{temp} \vdash D_1 : t_1, \ldots, \Gamma_{temp} \vdash D_m : t_m \quad \Gamma_{temp} \vdash S_1 : t'_1, \ldots, \Gamma_{temp} \vdash S_n : t'_n}{\Gamma, \Gamma_{alias} \vdash \{D_1, \ldots, D_m, S_1, \ldots, S_n\} : t'_k}$$

The type of a conditional statement or if statement is the union of the type of its consequent statement and its alternate statement. The predicate expression of a conditional statement must be a success type of a boolean.

$$\frac{\Gamma, \Gamma_{alias} \vdash S_{pred} : t_{pred} \quad \Gamma, \Gamma_{alias} \vdash S_{cons} : t_{cons} \quad \Gamma, \Gamma_{alias} \vdash S_{alt} : t_{alt} \quad t_{pred} \wedge \texttt{boolean} \neq \emptyset}{\Gamma, \Gamma_{alias} \vdash \texttt{if}\ (S_{pred})\ S_{cons}\ \texttt{else}\ S_{alt} : t_{cons} \mid t_{alt}}$$

For `while` statements, the predicate expression must be a success type of `boolean`. The rest of the statement is treated as a block.

$$\frac{\Gamma, \Gamma_{alias} \vdash E_{pred} : t_{pred} \quad \Gamma, \Gamma_{alias} \vdash S : t \quad t_{pred} \wedge \texttt{boolean} \neq \emptyset}{\Gamma, \Gamma_{alias} \vdash \texttt{while}\ (\ E_{pred}\ )\ S : t}$$

For `for` statements, the typing is the same as that of a `while` statement, except with additional type checks for the initialization and assignment expressions.

$$\frac{\begin{array}{l}\Gamma, \Gamma_{alias} \vdash E_{init} : t_{init} \quad \Gamma, \Gamma_{alias} \vdash E_{pred} : t_{pred} \quad \Gamma, \Gamma_{alias} \vdash E_{assign} : t_{assign} \\ \Gamma, \Gamma_{alias} \vdash S : t \quad t_{pred} \wedge \texttt{boolean} \neq \emptyset\end{array}}{\Gamma, \Gamma_{alias} \vdash \texttt{for}\ (\ E_{init}\texttt{;}\ E_{pred}\texttt{;}\ E_{assign}\ )\ S : t}$$

3 Dynamic Type Checking

Expressions evaluate to numbers, boolean values, strings, arrays 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
===	any	any	bool
!==	any	any	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.
In array access **`arr[key]`**, only arrays are allowed as **`arr`** and only integers are allowed as **`key`**.
Array indices in Source are limited to integers i in the range $0 \leq i < 2^{32} - 1$.
Pairs in Source are represented by arrays with two elements. Therefore,

```
is_pair([1, 2]);
```

and

```
equal(pair(1, 2), [1, 2]);
```

evaluate to `true`.
Access of an array with an array index to which no prior assignment has been made on the array returns `undefined`.

4 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.

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.
- `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:

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

  - `math_PI`: *primitive*, refers to the mathematical constant π,
  - `math_sqrt(n)`: *primitive*, returns the square root of the *number* `n`.

All math functions can be assumed to run in $O(1)$ time and are considered *primitive*. All math functions expect numbers as arguments and return numbers. We don't specify the behavior of a math function when some arguments are not numbers.

List Support

The following list processing functions are supported:

- `pair(x, y)`: *primitive*, makes a pair from `x` and `y`; time: $\Theta(1)$, space: $\Theta((1)$.
- `is_pair(x)`: *primitive*, returns `true` if `x` is a pair and `false` otherwise; time: $\Theta(1)$, space: $\Theta((1)$.
- `head(x)`: *primitive*, returns the head (first component) of the pair `x`; time: $\Theta(1)$, space: $\Theta((1)$.
- `tail(x)`: *primitive*, returns the tail (second component) of the pair `x`; time: $\Theta(1)$, space: $\Theta((1)$.
- `is_null(xs)`: *primitive*, returns `true` if `xs` is the empty list `null`, and `false` otherwise; time: $\Theta(1)$, space: $\Theta((1)$.
- `is_list(x)`: *primitive*, returns `true` if `x` is a list as defined in the lectures, and `false` otherwise. Iterative process; time: $\Theta(n)$, space: $\Theta(1)$, where n is the length of the chain of `tail` operations that can be applied to `x`.
- `list(x1, x2,..., xn)`: *primitive*, returns a list with n elements. The first element is `x1`, the second `x2`, etc. Iterative process; time: $\Theta(n)$, space: $\Theta(n)$, since the constructed list data structure consists of n pairs, each of which takes up a constant amount of space.
- `draw_data(x1, x2,..., xn)`: *primitive*, visualizes each `x1, x2,..., xn` in a separate drawing area in the Source Academy using a box-and-pointer diagram; time, space: $\Theta(n)$, where n is the combined number of data structures such as pairs in `x1, x2,..., xn`.
- `equal(x1, x2)`: Returns `true` if both have the same structure with respect to `pair`, and the same numbers, boolean values, functions or empty list at corresponding leave positions (places that are not themselves pairs), and `false` otherwise; time, space: $\Theta(n)$, where n is the number of data structures such as pairs in `x1` and `x2`.
- `length(xs)`: Returns the length of the list `xs`. Iterative process; time: $\Theta(n)$, space: $\Theta(1)$, where n is the length of `xs`.
- `map(f, xs)`: Returns a list that results from list `xs` by element-wise application of `f`. Iterative process; time: $\Theta(n)$ (apart from `f`), space: $\Theta(n)$ (apart from `f`), where n is the length of `xs`.
- `build_list(f, n)`: Makes a list with `n` elements by applying the unary function `f` to the numbers 0 to `n - 1`. Iterative process; time: $\Theta(n)$ (apart from `f`), space: $\Theta(n)$ (apart from `f`).
- `for_each(f, xs)`: Applies `f` to every element of the list `xs`, and then returns `true`. Iterative process; time: $\Theta(n)$ (apart from `f`), space: $\Theta(1)$ (apart from `f`), where n is the length of `xs`.
- `list_to_string(xs)`: Returns a string that represents list `xs` using the text-based box-and-pointer notation `[...]`.
- `reverse(xs)`: Returns list `xs` in reverse order. Iterative process; time: $\Theta(n)$, space: $\Theta(n)$, where n is the length of `xs`. The process is iterative, but consumes space $\Theta(n)$ because of the result list.
- `append(xs, ys)`: Returns a list that results from appending the list `ys` to the list `xs`. Iterative process; time: $\Theta(n)$, space: $\Theta(n)$, where n is the length of `xs`.

- `member(x, xs)`: Returns first postfix sublist whose head is identical to `x` (===); returns `[]` if the element does not occur in the list. Iterative process; time: $\Theta(n)$, space: $\Theta(1)$, where n is the length of `xs`.
- `remove(x, xs)`: Returns a list that results from `xs` by removing the first item from `xs` that is identical (===) to `x`. Iterative process; time: $\Theta(n)$, space: $\Theta(n)$, where n is the length of `xs`.
- `remove_all(x, xs)`: Returns a list that results from `xs` by removing all items from `xs` that are identical (===) to `x`. Iterative process; time: $\Theta(n)$, space: $\Theta(n)$, where n is the length of `xs`.
- `filter(pred, xs)`: Returns a list that contains only those elements for which the one-argument function `pred` returns `true`. Iterative process; time: $\Theta(n)$ (apart from `pred`), space: $\Theta(n)$ (apart from `pred`), where n is the length of `xs`.
- `enum_list(start, end)`: Returns a list that enumerates numbers starting from `start` using a step size of 1, until the number exceeds (>) `end`. Iterative process; time: $\Theta(n)$, space: $\Theta(n)$, where n is `end` − `start`.
- `list_ref(xs, n)`: Returns the element of list `xs` at position `n`, where the first element has index 0. Iterative process; time: $\Theta(n)$ (apart from `f`), space: $\Theta(1)$ (apart from `f`), where n is the length of `xs`.
- `accumulate(f, initial, xs)`: Applies binary function `f` to the elements of `xs` from right-to-left order, first applying `f` to the last element and the value `initial`, resulting in r_1, then to the second-last element and r_1, resulting in r_2, etc, and finally to the first element and r_{n-1}, where n is the length of the list. Thus, `accumulate(f,zero,list(1,2,3))` results in `f(1, f(2, f(3, zero)))`. Iterative process; time: $\Theta(n)$, space: $\Theta(n)$, where n is the length of `xs`, assuming `f` takes constant time.

Pair Mutators

The following pair mutator functions are supported:

- `set_head(p, x)`: *primitive*, changes the pair `p` such that its head is `x`. Returns `undefined`.
- `set_tail(p, x)`: *primitive*, changes the pair `p` such that its tail is `x`. Returns `undefined`.

Array Support

The following array processing functions are supported:

- `array_length(x)`: *primitive*, returns the current length of array `x`, which is 1 plus the highest index `i` that has been used so far in an array assignment on `x`; time: $\Theta(1)$, space: $\Theta(1)$.
- `is_array(x)`: *primitive*, returns returns `true` if `x` is an array, and `false` if it is not; time: $\Theta(1)$, space: $\Theta(1)$.

Stream Support

The following stream processing functions are supported:

- `stream(x1, x2,..., xn)`: *primitive*, returns a stream with n elements. The first element is `x1`, the second `x2`, etc.
  Laziness: No: In this implementation, we generate first a complete list, and then a stream using `list_to_stream`.
- `stream_tail(x)`: Assumes that the tail (second component) of the pair `x` is a nullary function, and returns the result of applying that function.
  Laziness: Yes: `stream_tail` only forces the direct tail of a given stream, but not the rest of the stream, i.e. not the tail of the tail, etc.

- `is_stream(x)`: Returns `true` if `x` is a stream as defined in the lectures, and `false` otherwise.
  Laziness: No: `is_stream` needs to force the given stream.
- `list_to_stream(xs)`: transforms a given list to a stream.
  Laziness: Yes: `list_to_stream` goes down the list only when forced.
- `stream_to_list(s)`: transforms a given stream to a list.
  Laziness: No: `stream_to_list` needs to force the whole stream.
- `stream_length(s)`: Returns the length of the stream `s`.
  Laziness: No: The function needs to force the whole stream.
- `stream_map(f, s)`: Returns a stream that results from stream `s` by element-wise application of `f`.
  Laziness: Yes: The argument stream is only explored as forced by the result stream.
- `build_stream(n, f)`: Makes a stream with `n` elements by applying the unary function `f` to the numbers 0 to `n - 1`.
  Laziness: Yes: The result stream forces the applications of fun for the next element.
- `stream_for_each(f, s)`: Applies `f` to every element of the stream `s`, and then returns `true`.
  Laziness: No: `stream_for_each` forces the exploration of the entire stream.
- `stream_reverse(s)`: Returns finite stream `s` in reverse order. Does not terminate for infinite streams.
  Laziness: No: `stream_reverse` forces the exploration of the entire stream.
- `stream_append(xs, ys)`: Returns a stream that results from appending the stream `ys` to the stream `xs`.
  Laziness: Yes: Forcing the result stream activates the actual append operation.
- `stream_member(x, s)`: Returns first postfix substream whose head is equal to `x` (===); returns `null` if the element does not occur in the stream.
  Laziness: Sort-of: `stream_member` forces the stream only until the element is found.
- `stream_remove(x, s)`: Returns a stream that results from given stream `s` by removing the first item from `s` that is equal (===) to `x`. Returns the original list if there is no occurrence.
  Laziness: Yes: Forcing the result stream leads to construction of each next element.
- `stream_remove_all(x, s)`: Returns a stream that results from given stream `s` by removing all items from `s` that are equal (===) to `x`.
  Laziness: Yes: The result stream forces the construction of each next element.
- `stream_filter(pred, s)`: Returns a stream that contains only those elements for which the one-argument function `pred` returns `true`.
  Laziness: Yes: The result stream forces the construction of each next element. Of course, the construction of the next element needs to go down the stream until an element is found for which `pred` holds.
- `enum_stream(start, end)`: Returns a stream that enumerates numbers starting from `start` using a step size of 1, until the number exceeds (>) `end`.
  Laziness: Yes: Forcing the result stream leads to the construction of each next element.
- `integers_from(n)`: Constructs an infinite stream of integers starting at a given number `n`.
  Laziness: Yes: Forcing the result stream leads to the construction of each next element.
- `eval_stream(s, n)`: Constructs the list of the first `n` elements of a given stream `s`.
  Laziness: Sort-of: `eval_stream` only forces the computation of the first `n` elements, and leaves the rest of the stream untouched.
- `stream_ref(s, n)`: Returns the element of stream `s` at position `n`, where the first element has index 0.
  Laziness: Sort-of: `stream_ref` only forces the computation of the first `n` elements, and leaves the rest of the stream untouched.

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.

Appendix: List library

Those list library functions that are not primitive functions are pre-declared as follows:

```
// list.js START


/**
 * **primitive**; makes a pair whose head (first component) is <CODE>x</CODE>
 * and whose tail (second component) is <CODE>y</CODE>; time: <CODE>Theta(1)</CODE, spac
 * @param {value} x - given head
 * @param {value} y - given tail
 * @returns {pair} pair with <CODE>x</CODE> as head and <CODE>y</CODE> as tail.
 */
function pair(x, y) {}

/**
 *  **primitive**; returns <CODE>true</CODE> if <CODE>x</CODE> is a
 * pair and false otherwise; time: <CODE>Theta(1)</CODE, space: <CODE>Theta(1)</CODE>.
 * @param {value} x - given value
 * @returns {boolean} whether <CODE>x</CODE> is a pair
 */
function is_pair(x) {}

/**
 *  **primitive**; returns head (first component) of given pair <CODE>p</CODE>; time: <C
 * @param {pair} p - given pair
 * @returns {value} head of <CODE>p</CODE>
 */
function head(p) {}

/**
 *  **primitive**; returns tail (second component of given pair <CODE>p</CODE>; time: <C
 * @param {pair} p - given pair
 * @returns {value} tail of <CODE>p</CODE>
 */
function tail(p) {}

/**
 *  **primitive**; returns <CODE>true</CODE> if <CODE>x</CODE> is the
 * empty list <CODE>null</CODE>, and <CODE>false</CODE> otherwise; time: <CODE>Theta(1)<
 * @param {value} x - given value
 * @returns {boolean} whether <CODE>x</CODE> is <CODE>null</CODE>
 */
function is_null(x) {}

/**
 * **primitive**; returns <CODE>true</CODE> if
 * <CODE>xs</CODE> is a list as defined in the textbook, and
 * <CODE>false</CODE> otherwise. Iterative process;
 * time: <CODE>Theta(n)</CODE>, space: <CODE>Theta(1)</CODE>, where <CODE>n</CODE>
 * is the length of the
 * chain of <CODE>tail</CODE> operations that can be applied to <CODE>xs</CODE>.
 * <CODE>is_list</CODE> recurses down the list and checks that it ends with the empty li
 * @param {value} xs - given candidate
 * @returns whether {xs} is a list
 */
function is_list(xs) {}

/**
 *  **primitive**; given <CODE>n</CODE> values, returns a list of length <CODE>n</CODE>.
 * The elements of the list are the given values in the given order; time: <CODE>Theta(n
 * @param {value} value1,value2,...,value_n - given values
 * @returns {list} list containing all values
 */
function list(value1, value2, ...values) {}

/**
 * visualizes the arguments in a separate drawing
 * area in the Source Academy using box-and-pointer diagrams; time, space:
 * <CODE>Theta(n)</CODE>, where <CODE>n</CODE> is the total number of data structures su
 * pairs in the arguments.
 * @param {value} value1,value2,...,value_n - given values
 * @returns {value} given <CODE>x</CODE>
 */
function draw_data(value1, value2, ...values) {}

/**
 * Returns <CODE>true</CODE> if both
 * have the same structure with respect to <CODE>pair</CODE>,
 * and identical values at corresponding leave positions (places that are not
 * themselves pairs), and <CODE>false</CODE> otherwise. For the "identical",
 * the values need to have the same type, otherwise the result is
 * <CODE>false</CODE>. If corresponding leaves are boolean values, these values
 * need to be the same. If both are <CODE>undefined</CODE> or both are
 * <CODE>null</CODE>, the result is <CODE>true</CODE>. Otherwise they are compared
 * with <CODE>===</CODE> (using the definition of <CODE>===</CODE> in the
 * respective Source language in use).
 * Time, space:
 * <CODE>Theta(n)</CODE>, where <CODE>n</CODE> is the total number of data structures su
 * pairs in <CODE>x</CODE> and <CODE>y</CODE>.
 * @param {value} x - given value
 * @param {value} y - given value
 * @returns {boolean} whether <CODE>x</CODE> is structurally equal to <CODE>y</CODE>
 */
function equal(xs, ys) {
  return is_pair(xs)
    ? is_pair(ys) && equal(head(xs), head(ys)) && equal(tail(xs), tail(ys))
    : is_null(xs)
      ? is_null(ys)
      : is_number(xs)
        ? is_number(ys) && xs === ys
        : is_boolean(xs)
          ? is_boolean(ys) && ((xs && ys) || (!xs && !ys))
          : is_string(xs)
            ? is_string(ys) && xs === ys
            : is_undefined(xs)
              ? is_undefined(ys)
              : // we know now that xs is a function
                is_function(ys) && xs === ys
}

/**
 * Returns the length of the list
 * <CODE>xs</CODE>.
 * Iterative process; time: <CODE>Theta(n)</CODE>, space:
 * <CODE>Theta(1)</CODE>, where <CODE>n</CODE> is the length of <CODE>xs</CODE>.
 * @param {list} xs - given list
 * @returns {number} length of <CODE>xs</CODE>
 */
function length(xs) {
  return $length(xs, 0)
}
function $length(xs, acc) {
  return is_null(xs) ? acc : $length(tail(xs), acc + 1)
}

/**
 * Returns a list that results from list
 * <CODE>xs</CODE> by element-wise application of unary function <CODE>f</CODE>.
 * Iterative process; time: <CODE>Theta(n)</CODE> (apart from <CODE>f</CODE>),
 * space: <CODE>Theta(n)</CODE> (apart from <CODE>f</CODE>), where <CODE>n</CODE> is the
 * <CODE>f</CODE> is applied element-by-element:
 * <CODE>map(f, list(1, 2))</CODE> results in <CODE>list(f(1), f(2))</CODE>.
 * @param {function} f - unary
 * @param {list} xs - given list
 * @returns {list} result of mapping
 */
function map(f, xs) {
  return $map(f, xs, null)
}
function $map(f, xs, acc) {
  return is_null(xs) ? reverse(acc) : $map(f, tail(xs), pair(f(head(xs)), acc))
}

/**
 * Makes a list with <CODE>n</CODE>
 * elements by applying the unary function <CODE>f</CODE>
 * to the numbers 0 to <CODE>n - 1</CODE>, assumed to be a nonnegative integer.
 * Iterative process; time: <CODE>Theta(n)</CODE> (apart from <CODE>f</CODE>), space: <C
 * @param {function} f - unary function
 * @param {number} n - given nonnegative integer
 * @returns {list} resulting list
 */
function build_list(fun, n) {
  return $build_list(n - 1, fun, null)
}
function $build_list(i, fun, already_built) {
  return i < 0 ? already_built : $build_list(i - 1, fun, pair(fun(i), already_built))
}

/**
 * Applies unary function <CODE>f</CODE> to every
 * element of the list <CODE>xs</CODE>.
 * Iterative process; time: <CODE>Theta(n)</CODE> (apart from <CODE>f</CODE>), space: <C
 * where <CODE>n</CODE> is the length of <CODE>xs</CODE>.
 * <CODE>f</CODE> is applied element-by-element:
 * <CODE>for_each(fun, list(1, 2))</CODE> results in the calls
 * <CODE>fun(1)</CODE> and <CODE>fun(2)</CODE>.
 * @param {function} f - unary
 * @param {list} xs - given list
 * @returns {boolean} true
 */

function for_each(fun, xs) {
  if (is_null(xs)) {
    return true
  } else {
    fun(head(xs))
    return for_each(fun, tail(xs))
  }
}
```

```
/**
 * Returns a string that represents
 * list <CODE>xs</CODE> using the text-based box-and-pointer notation
 * <CODE>[...]</CODE>.
 * Iterative process; time: <CODE>Theta(n)</CODE> where <CODE>n</CODE> is the size of th
 * The process is iterative, but consumes space <CODE>O(m)</CODE>
 * because of the result string.
 * @param {list} xs - given list
 * @returns {string} <CODE>xs</CODE> converted to string
 */
function list_to_string(xs) {
  return $list_to_string(xs, x => x)
}
function $list_to_string(xs, cont) {
  return is_null(xs)
    ? cont('null')
    : is_pair(xs)
      ? $list_to_string(head(xs), x =>
          $list_to_string(tail(xs), y => cont('[' + x + ',' + y + ']'))
        )
      : cont(stringify(xs))
}

/**
 * Returns list <CODE>xs</CODE> in reverse
 * order. Iterative process; time: <CODE>Theta(n)</CODE>,
 * space: <CODE>Theta(n)</CODE>, where <CODE>n</CODE> is the length of <CODE>xs</CODE>.
 * The process is iterative, but consumes space <CODE>Theta(n)</CODE>
 * because of the result list.
 * @param {list} xs - given list
 * @returns {list} <CODE>xs</CODE> in reverse
 */
function reverse(xs) {
  return $reverse(xs, null)
}
function $reverse(original, reversed) {
  return is_null(original) ? reversed : $reverse(tail(original), pair(head(original), re
}

/**
 * Returns a list that results from
 * appending the list <CODE>ys</CODE> to the list <CODE>xs</CODE>.
 * Iterative process; time: <CODE>Theta(n)</CODE>, space:
 * <CODE>Theta(n)</CODE>, where <CODE>n</CODE> is the length of <CODE>xs</CODE>.
 * In the result, null at the end of the first argument list
 * is replaced by the second argument, regardless what the second
 * argument consists of.
 * @param {list} xs - given first list
 * @param {list} ys - given second list
 * @returns {list} result of appending <CODE>xs</CODE> and <CODE>ys</CODE>
 */
function append(xs, ys) {
  return $append(xs, ys, xs => xs)
}
function $append(xs, ys, cont) {
  return is_null(xs) ? cont(ys) : $append(tail(xs), ys, zs => cont(pair(head(xs), zs)))
}

/**
```

```
 * Returns first postfix sublist
 * whose head is identical to
 * <CODE>v</CODE> (using <CODE>===</CODE>); returns <CODE>null</CODE> if the
 * element does not occur in the list.
 * Iterative process; time: <CODE>Theta(n)</CODE>,
 * space: <CODE>Theta(1)</CODE>, where <CODE>n</CODE> is the length of <CODE>xs</CODE>.
 * @param {value} v - given value
 * @param {list} xs - given list
 * @returns {list} postfix sublist that starts with <CODE>v</CODE>
 */
function member(v, xs) {
  return is_null(xs) ? null : v === head(xs) ? xs : member(v, tail(xs))
}

/** Returns a list that results from
 * <CODE>xs</CODE> by removing the first item from <CODE>xs</CODE> that
 * is identical (<CODE>===</CODE>) to <CODE>v</CODE>.
 * Returns the original
 * list if there is no occurrence. Iterative process;
 * time: <CODE>Theta(n)</CODE>, space: <CODE>Theta(n)</CODE>, where <CODE>n</CODE>
 * is the length of <CODE>xs</CODE>.
 * @param {value} v - given value
 * @param {list} xs - given list
 * @returns {list} <CODE>xs</CODE> with first occurrence of <CODE>v</CODE> removed
 */
function remove(v, xs) {
  return $remove(v, xs, null)
}
function $remove(v, xs, acc) {
  return is_null(xs)
    ? append(reverse(acc), xs)
    : v === head(xs)
      ? append(reverse(acc), tail(xs))
      : $remove(v, tail(xs), pair(head(xs), acc))
}

/**
 * Returns a list that results from
 * <CODE>xs</CODE> by removing all items from <CODE>xs</CODE> that
 * are identical (<CODE>===</CODE>) to <CODE>v</CODE>.
 * Returns the original
 * list if there is no occurrence.
 * Iterative process;
 * time: <CODE>Theta(n)</CODE>, space: <CODE>Theta(n)</CODE>, where <CODE>n</CODE>
 * is the length of <CODE>xs</CODE>.
 * @param {value} v - given value
 * @param {list} xs - given list
 * @returns {list} <CODE>xs</CODE> with all occurrences of <CODE>v</CODE> removed
 */
function remove_all(v, xs) {
  return $remove_all(v, xs, null)
}
function $remove_all(v, xs, acc) {
  return is_null(xs)
    ? append(reverse(acc), xs)
    : v === head(xs)
      ? $remove_all(v, tail(xs), acc)
      : $remove_all(v, tail(xs), pair(head(xs), acc))
}
```

```
/**
 * Returns a list that contains
 * only those elements for which the one-argument function
 * <CODE>pred</CODE>
 * returns <CODE>true</CODE>.
 * Iterative process;
 * time: <CODE>Theta(n)</CODE> (apart from <CODE>pred</CODE>), space: <CODE>Theta(n)</CO
 * where <CODE>n</CODE> is the length of <CODE>xs</CODE>.
 * @param {function} pred - unary function returning boolean value
 * @param {list} xs - given list
 * @returns {list} list with those elements of <CODE>xs</CODE> for which <CODE>pred</COD
 */
function filter(pred, xs) {
  return $filter(pred, xs, null)
}
function $filter(pred, xs, acc) {
  return is_null(xs)
    ? reverse(acc)
    : pred(head(xs))
      ? $filter(pred, tail(xs), pair(head(xs), acc))
      : $filter(pred, tail(xs), acc)
}

/**
 * Returns a list that enumerates
 * numbers starting from <CODE>start</CODE> using a step size of 1, until
 * the number exceeds (<CODE>&gt;</CODE>) <CODE>end</CODE>.
 * Iterative process;
 * time: <CODE>Theta(n)</CODE>, space: <CODE>Theta(n)</CODE>,
 * where <CODE>n</CODE> is <CODE>end - start</CODE>.
 * @param {number} start - starting number
 * @param {number} end - ending number
 * @returns {list} list from <CODE>start</CODE> to <CODE>end</CODE>
 */
function enum_list(start, end) {
  return $enum_list(start, end, null)
}
function $enum_list(start, end, acc) {
  return start > end ? reverse(acc) : $enum_list(start + 1, end, pair(start, acc))
}

/**
 * Returns the element
 * of list <CODE>xs</CODE> at position <CODE>n</CODE>,
 * where the first element has index 0.
 * Iterative process;
 * time: <CODE>Theta(n)</CODE>, space: <CODE>Theta(1)</CODE>,
 * where <CODE>n</CODE> is the length of <CODE>xs</CODE>.
 * @param {list} xs - given list
 * @param {number} n - given position
 * @returns {value} item in <CODE>xs</CODE> at position <CODE>n</CODE>
 */
function list_ref(xs, n) {
  return n === 0 ? head(xs) : list_ref(tail(xs), n - 1)
}

/** Applies binary
 * function <CODE>f</CODE> to the elements of <CODE>xs</CODE> from
 * right-to-left order, first applying <CODE>f</CODE> to the last element
 * and the value <CODE>initial</CODE>, resulting in <CODE>r</CODE><SUB>1</SUB>,
 * then to the
 * second-last element and <CODE>r</CODE><SUB>1</SUB>, resulting in
 * <CODE>r</CODE><SUB>2</SUB>,
 * etc, and finally
 * to the first element and <CODE>r</CODE><SUB>n-1</SUB>, where
 * <CODE>n</CODE> is the length of the
 * list. Thus, <CODE>accumulate(f,zero,list(1,2,3))</CODE> results in
 * <CODE>f(1, f(2, f(3, zero)))</CODE>.
 * Iterative process;
 * time: <CODE>Theta(n)</CODE> (apart from <CODE>f</CODE>), space: <CODE>Theta(n)</CODE
 * where <CODE>n</CODE> is the length of <CODE>xs</CODE>.
 * @param {function} f - binary function
 * @param {value} initial - initial value
 * @param {list} xs - given list
 * @returns {value} result of accumulating <CODE>xs</CODE> using <CODE>f</CODE> startin
 */
function accumulate(f, initial, xs) {
  return $accumulate(f, initial, xs, x => x)
}
function $accumulate(f, initial, xs, cont) {
  return is_null(xs) ? cont(initial) : $accumulate(f, initial, tail(xs), x => cont(f(hea
}

/**
 * Optional second argument.
 * Similar to <CODE>display</CODE>, but formats well-formed lists nicely if detected;
 * time, space:
 * <CODE>Theta(n)</CODE>, where <CODE>n</CODE> is the total number of data structures s
 * pairs in <CODE>x</CODE>.
 * @param {value} xs - list structure to be displayed
 * @param {string} s to be displayed, preceding <CODE>xs</CODE>
 * @returns {value} xs, the first argument value
 */
function display_list(xs, s) {}

//

// list.js END
```

Appendix: Stream library

Those stream library functions that are not primitive functions are pre-declared as follows:

```
// stream.js START


// Supporting streams in the Scheme style, following
// "stream discipline"

/**
 * assumes that the tail (second component) of the
 * pair {x} is a nullary function, and returns the result of
 * applying that function. Throws an exception if the argument
 * is not a pair, or if the tail is not a function.
 * Laziness: Yes: {stream_tail} only forces the direct tail
 * stream, but not the rest of the stream, i.e. not the tail
 * of the tail, etc.
 * @param {Stream} xs - given stream
 * @returns {Stream} result stream (if stream discipline is used)
 */

function stream_tail(xs) {
  if (is_pair(xs)) {
    const the_tail = tail(xs)
    if (is_function(the_tail)) {
      return the_tail()
    } else {
      error(
        the_tail,
        'stream_tail(xs) expects a function as ' +
          'the tail of the argument pair xs, ' +
          'but encountered '
      )
    }
  } else {
    error(xs, 'stream_tail(xs) expects a pair as ' + 'argument xs, but encountered ')
  }
}

/**
 * Returns <CODE>true</CODE> if
 * <CODE>xs</CODE> is a stream as defined in the textbook, and
 * <CODE>false</CODE> otherwise. Iterative process.
 * Recurses down the stream and checks that it ends with the empty stream null.
 * Laziness:  No: <CODE>is_stream</CODE> needs to force the given stream.
 * @param {value} xs - given candidate
 * @returns {boolean} whether <CODE>xs</CODE> is a stream
 */

function is_stream(xs) {
  return (
    is_null(xs) ||
    (is_pair(xs) && is_function(tail(xs)) && arity(tail(xs)) === 0 && is_stream(stream_t
  )
}

/**
 * Given list <CODE>xs</CODE>, returns a stream of same length with
 * the same elements as <CODE>xs</CODE> in the same order.
 * Laziness:  Yes: <CODE>list_to_stream</CODE>
 * goes down the list only when forced.
 * @param {list} xs - given list
 * @returns {stream} stream containing all elements of <CODE>xs</CODE>
 */

function list_to_stream(xs) {
  return is_null(xs) ? null : pair(head(xs), () => list_to_stream(tail(xs)))
}

/**
 * Given stream <CODE>xs</CODE>, returns a list of same length with
 * the same elements as <CODE>xs</CODE> in the same order.
 * Laziness:  No: <CODE>stream_to_list</CODE> needs to force the whole
 * stream.
 * @param {stream} xs - stream
 * @returns {list} containing all elements of <CODE>xs</CODE>
 */

function stream_to_list(xs) {
  return is_null(xs) ? null : pair(head(xs), stream_to_list(stream_tail(xs)))
}

/**
 * Given <CODE>n</CODE> values, returns a stream of length <CODE>n</CODE>.
 * The elements of the stream are the given values in the given order.
 * Lazy? No: A
 * complete list is generated,
 * and then a stream using <CODE>list_to_stream</CODE> is generated from it.
 * @param {value} value1,value2,...,value_n - given values
 * @returns {stream} stream containing all values
 */

function stream() {
  var the_list = null
  for (var i = arguments.length - 1; i >= 0; i--) {
    the_list = pair(arguments[i], the_list)
  }
  return list_to_stream(the_list)
}

/**
 * Returns the length of the stream
 * <CODE>xs</CODE>.
 * Iterative process.
 * Lazy? No: The function needs to explore the whole stream
 * @param {stream} xs - given stream
 * @returns {number} length of <CODE>xs</CODE>
 */

function stream_length(xs) {
  return is_null(xs) ? 0 : 1 + stream_length(stream_tail(xs))
}

/**
 * Returns a stream that results from stream
 * <CODE>xs</CODE> by element-wise application
 * of unary function <CODE>f</CODE>.
 * <CODE>f</CODE> is applied element-by-element:
 * <CODE>stream_map(f, stream(1,2))</CODE> results in
 * the same as <CODE>stream(f(1),f(2))</CODE>.
 * Lazy? Yes: The argument stream is only explored as forced by
 *            the result stream.
 * @param {function} f - unary
 * @param {stream} xs - given stream
 * @returns {stream} result of mapping
 */
function stream_map(f, s) {
  return is_null(s) ? null : pair(f(head(s)), () => stream_map(f, stream_tail(s)))
}

/**
 * Makes a stream with <CODE>n</CODE>
 * elements by applying the unary function <CODE>f</CODE>
 * to the numbers 0 to <CODE>n - 1</CODE>, assumed to be a nonnegative integer.
 * Lazy? Yes: The result stream forces the application of <CODE>f</CODE>
 *           for the next element
 * @param {function} f - unary function
 * @param {number} n - given nonnegative integer
 * @returns {stream} resulting stream
 */

function build_stream(fun, n) {
  function build(i) {
    return i >= n ? null : pair(fun(i), () => build(i + 1))
  }
  return build(0)
}

/**
 * Applies unary function <CODE>f</CODE> to every
 * element of the stream <CODE>xs</CODE>.
 * Iterative process.
 * <CODE>f</CODE> is applied element-by-element:
 * <CODE>stream_for_each(f, stream(1, 2))</CODE> results in the calls
 * <CODE>f(1)</CODE> and <CODE>f(2)</CODE>.
 * Lazy? No: <CODE>stream_for_each</CODE>
 * forces the exploration of the entire stream
 * @param {function} f - unary
 * @param {stream} xs - given stream
 * @returns {boolean} true
 */

function stream_for_each(fun, xs) {
  if (is_null(xs)) {
    return true
  } else {
    fun(head(xs))
    return stream_for_each(fun, stream_tail(xs))
  }
}

/**
 * Returns stream <CODE>xs</CODE> in reverse
 * order. Iterative process.
 * The process is iterative, but consumes space <CODE>Omega(n)</CODE>
 * because of the result stream.
 * Lazy? No: <CODE>stream_reverse</CODE>
 * forces the exploration of the entire stream
 * @param {stream} xs - given stream
 * @returns {stream} <CODE>xs</CODE> in reverse
 */
```

```
function stream_reverse(xs) {
  function rev(original, reversed) {
    return is_null(original)
      ? reversed
      : rev(
          stream_tail(original),
          pair(head(original), () => reversed)
        )
  }
  return rev(xs, null)
}

/**
 * Returns a stream that results from
 * appending the stream <CODE>ys</CODE> to the stream <CODE>xs</CODE>.
 * In the result, null at the end of the first argument stream
 * is replaced by the second argument, regardless what the second
 * argument consists of.
 * Lazy? Yes: the result stream forces the actual append operation
 * @param {stream} xs - given first stream
 * @param {stream} ys - given second stream
 * @returns {stream} result of appending <CODE>xs</CODE> and <CODE>ys</CODE>
 */

function stream_append(xs, ys) {
  return is_null(xs) ? ys : pair(head(xs), () => stream_append(stream_tail(xs), ys))
}

/**
 * Returns first postfix substream
 * whose head is identical to
 * <CODE>v</CODE> (using <CODE>===</CODE>); returns <CODE>null</CODE> if the
 * element does not occur in the stream.
 * Iterative process.
 * Lazy? Sort-of: <CODE>stream_member</CODE>
 * forces the stream only until the element
 * is found.
 * @param {value} v - given value
 * @param {stream} xs - given stream
 * @returns {stream} postfix substream that starts with <CODE>v</CODE>
 */

function stream_member(x, s) {
  return is_null(s) ? null : head(s) === x ? s : stream_member(x, stream_tail(s))
}

/** Returns a stream that results from
 * <CODE>xs</CODE> by removing the first item from <CODE>xs</CODE> that
 * is identical (<CODE>===</CODE>) to <CODE>v</CODE>.
 * Returns the original
 * stream if there is no occurrence.
 * Lazy? Yes: the result stream forces the construction of each next element
 * @param {value} v - given value
 * @param {stream} xs - given stream
 * @returns {stream} <CODE>xs</CODE> with first occurrence of <CODE>v</CODE> removed
 */

function stream_remove(v, xs) {
  return is_null(xs)
```

```
    ? null
    : v === head(xs)
      ? stream_tail(xs)
      : pair(head(xs), () => stream_remove(v, stream_tail(xs)))
}

/**
 * Returns a stream that results from
 * <CODE>xs</CODE> by removing all items from <CODE>xs</CODE> that
 * are identical (<CODE>===</CODE>) to <CODE>v</CODE>.
 * Returns the original
 * stream if there is no occurrence.
 * Recursive process.
 * Lazy? Yes: the result stream forces the construction of each next
 * element
 * @param {value} v - given value
 * @param {stream} xs - given stream
 * @returns {stream} <CODE>xs</CODE> with all occurrences of <CODE>v</CODE> removed
 */

function stream_remove_all(v, xs) {
  return is_null(xs)
    ? null
    : v === head(xs)
      ? stream_remove_all(v, stream_tail(xs))
      : pair(head(xs), () => stream_remove_all(v, stream_tail(xs)))
}

/**
 * Returns a stream that contains
 * only those elements of given stream <CODE>xs</CODE>
 * for which the one-argument function
 * <CODE>pred</CODE>
 * returns <CODE>true</CODE>.
 * Lazy? Yes: The result stream forces the construction of
 *            each next element. Of course, the construction
 *            of the next element needs to go down the stream
 *            until an element is found for which <CODE>pred</CODE> holds.
 * @param {function} pred - unary function returning boolean value
 * @param {stream} xs - given stream
 * @returns {stream} stream with those elements of <CODE>xs</CODE> for which <CODE>pred<
 */

function stream_filter(p, s) {
  return is_null(s)
    ? null
    : p(head(s))
      ? pair(head(s), () => stream_filter(p, stream_tail(s)))
      : stream_filter(p, stream_tail(s))
}

/**
 * Returns a stream that enumerates
 * numbers starting from <CODE>start</CODE> using a step size of 1, until
 * the number exceeds (<CODE>&gt;</CODE>) <CODE>end</CODE>.
 * Lazy? Yes: The result stream forces the construction of
 *            each next element
 * @param {number} start - starting number
 * @param {number} end - ending number
 * @returns {stream} stream from <CODE>start</CODE> to <CODE>end</CODE>
 */

function enum_stream(start, end) {
  return start > end ? null : pair(start, () => enum_stream(start + 1, end))
}

/**
 * Returns infinite stream if integers starting
 * at given number <CODE>n</CODE> using a step size of 1.
 * Lazy? Yes: The result stream forces the construction of
 *            each next element
 * @param {number} start - starting number
 * @returns {stream} infinite stream from <CODE>n</CODE>
 */

function integers_from(n) {
  return pair(n, () => integers_from(n + 1))
}

/**
 * Constructs the list of the first <CODE>n</CODE> elements
 * of a given stream <CODE>s</CODE>
 * Lazy? Sort-of: <CODE>eval_stream</CODE> only forces the computation of
 * the first <CODE>n</CODE> elements, and leaves the rest of
 * the stream untouched.
 * @param {stream} s - given stream
 * @param {number} n - nonnegative number of elements to place in result list
 * @returns {list} result list
 */

function eval_stream(s, n) {
  function es(s, n) {
    return n === 1 ? list(head(s)) : pair(head(s), es(stream_tail(s), n - 1))
  }
  return n === 0 ? null : es(s, n)
}

/**
 * Returns the element
 * of stream <CODE>xs</CODE> at position <CODE>n</CODE>,
 * where the first element has index 0.
 * Iterative process.
 * Lazy? Sort-of: <CODE>stream_ref</CODE> only forces the computation of
 *                the first <CODE>n</CODE> elements, and leaves the rest of
 *                the stream untouched.
 * @param {stream} xs - given stream
 * @param {number} n - given position
 * @returns {value} item in <CODE>xs</CODE> at position <CODE>n</CODE>
 */

function stream_ref(s, n) {
  return n === 0 ? head(s) : stream_ref(stream_tail(s), n - 1)
}

//

// stream.js END
```