Specification of Source §2—2021 edition

Martin Henz, Lee Ning Yuan, Daryl Tan

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.

1 Changes

Compared to Source §1, Source §2 has the following changes:

- `null`: primitive list expression (empty list).
- List library: Functions for creating, accessing and processing lists.

2 Syntax

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

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

Restrictions

- Return statements are only allowed in bodies of functions.
- There cannot be any newline character between **return** and *expression* in return statements.[2]

[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.

[2] Source inherits this syntactic quirk of JavaScript.

- 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

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

Disjunction

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

stands for

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

Names

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

Numbers

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

[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.

[5] In ECMAScript 2020 (9th Edition), these names are called *identifiers*.

[6] By *letter* we mean Unicode letters (L) or letter numbers (Nl).

[7] By *digit* we mean characters in the Unicode categories Nd (including the decimal digits 0, 1, 2, 3, 4, 5, 6, 7, 8, 9), Mn, Mc and Pc.

[8] By *restricted word* we mean any of: **arguments**, **await**, **break**, **case**, **catch**, **class**, **const**, **continue**, **debugger**, **default**, **delete**, **do**, **else**, **enum**, **eval**, **export**, **extends**, **false**, **finally**, **for**, **function**, **if**, **implements**, **import**, **in**, **instanceof**, **interface**, **let**, **new**, **null**, **package**, **private**, **protected**, **public**, **return**, **static**, **super**, **switch**, **this**, **throw**, **true**, **try**, **typeof**, **var**, **void**, **while**, **with**, **yield**. These are all words that cannot be used without restrictions as names in the strict mode of ECMAScript 2020.

Strings

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

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

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

Comments

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

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

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

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

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

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`.

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

- `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:
  - `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.

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>&amp;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> starting
 */
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 su
 * 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
```