Specification of Source §2—1920 edition

Martin Henz

National University of Singapore
School of Computing

March 5, 2020

The language Source is the official language of the textbook *Structure and Interpretation of Computer Programs, JavaScript Adaptation*. You have never heard of Source? No worries! It was invented just for the purpose of the book. Source is a sublanguage of ECMAScript 2018 (9th Edition) and defined in the documents titled "Source §x", where x refers to the respective textbook chapter. For example, Source §3 is suitable for textbook Chapter 3 and the preceeding chapters.

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.

Programs

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

program	::=	*statement* ...	statement sequence
statement	::=	**const** *name* **=** *expression* **;**	constant declaration
	|	**function** *name* **(** *parameters* **)** *block*	function declaration
	|	**return** *expression* **;**	return statement
	|	*if-statement*	conditional statement
	|	*block*	block statement
	|	*expression* **;**	expression statement
parameters	::=	ϵ | *name* (**,** *name*) ...	function parameters
if-statement	::=	**if (** *expression* **)** *block*	
		else (*block* | *if-statement*)	conditional statement
block	::=	**{** *program* **}**	block statement
expression	::=	*number*	primitive number expression
	|	**true** | **false**	primitive boolean expression
	|	**null**	primitive list expression
	|	*string*	primitive string expression
	|	*name*	name expression
	|	*expression binary-operator expression*	binary operator combination
	|	*unary-operator expression*	unary operator combination
	|	*expression* **(** *expressions* **)**	function application
	|	(*name* | **(** *parameters* **)**) **=>** *expression*	function definition (expr. body)
	|	(*name* | **(** *parameters* **)**) **=>** *block*	function definition (block body)
	|	*expression* **?** *expression* **:** *expression*	conditional expression
	|	**(** *expression* **)**	parenthesised expression
binary-operator	::=	**+** | **-** | ***** | **/** | **%** | **===** | **!==**	
	|	**>** | **<** | **>=** | **<=** | **&&** | **||**	binary operator
unary-operator	::=	**!** | **-**	unary operator
expressions	::=	ϵ | *expression* (**,** *expression*) ...	argument expressions

Binary boolean operators

Conjunction

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

stands for

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

Disjunction

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

stands for

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

[1] We adopt Henry Ledgard's BNF variant that he described in *A human engineered variant of BNF*, ACM SIGPLAN Notices, Volume 15 Issue 10, October 1980, Pages 57-62. In our grammars, we use **bold** font for keywords, *italics* for syntactic variables, ϵ for nothing, $x \mid y$ for x or y, and $x \ldots$ for zero or more repetitions of x.

Restrictions

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

Names

Names[2] start with _, `$` or a letter[3] and contain only _, `$`, letters or digits[4]. Reserved words[5] such as keywords are not allowed as names.
Valid names are `x`, `_45`, `$$` and π, but always keep in mind that programming is communicating and that the familiarity of the audience with the characters used in names is an important aspect of program readability.
In addition to names that are declared using **`const`**, **`function`**, **`=>`** (and **`let`** in Source §3 and 4), the following names refer to primitive functions and constants:

- `math_`*name*, where *name* is any name specified in the JavaScript `Math` library, see ECMAScript Specification, Section 20.2. Examples:
  - `math_PI`: Refers to the mathematical constant π,
  - `math_sqrt(n)`: Returns the square root of the *number* `n`.
- `runtime()`: Returns number of milliseconds elapsed since January 1, 1970 00:00:00 UTC
- `parse_int(s, i)`: interprets the *string* `s` as an integer, using the positive integer `i` as radix, and returns the respective value, see ECMAScript Specification, Section 18.2.5.
- `undefined`, `NaN`, `Infinity`: Refer to JavaScript's undefined, NaN ("Not a Number") and Infinity values, respectively.
- `is_boolean(x)`, `is_number(x)`, `is_string(x)`, `is_function(x)`: return `true` if the type of `x` matches the function name and `false` if it does not. Following JavaScript, we specify that `is_number` returns `true` for `NaN` and `Infinity`.
- `prompt(s)`: Pops up a window that displays the *string* `s`, provides an input line for the user to enter a text, a "Cancel" button and an "OK" button. The call of `prompt` suspends execution of the program until one of the two buttons is pressed. If the "OK" button is pressed, `prompt` returns the entered text as a string. If the "Cancel" button is pressed, `prompt` returns a non-string value.
- `display(x)`: Displays the value `x` in the console[6]; returns the argument `a`.
- `display(x, s)`: Displays the string `s`, followed by a space character, followed by the value `x` in the console[6]; returns the argument `x`.
- `error(x)`: Displays the value `x` in the console[6] with error flag. The evaluation of any call of `error` aborts the running program immediately.

[2] In ECMAScript 2018 (9th Edition), these names are called *identifiers*.
[3] By *letter* we mean Unicode letters (L) or letter numbers (Nl).
[4] By *digit* we mean characters in the Unicode categories Nd (including the decimal digits 0, 1, 2, 3, 4, 5, 6, 7, 8, 9), Mn, Mc and Pc.
[5] By *Reserved word* we mean any of: **`break`**, **`case`**, **`catch`**, **`continue`**, **`debugger`**, **`default`**, **`delete`**, **`do`**, **`else`**, **`finally`**, **`for`**, **`function`**, **`if`**, **`in`**, **`instanceof`**, **`new`**, **`return`**, **`switch`**, **`this`**, **`throw`**, **`try`**, **`typeof`**, **`var`**, **`void`**, **`while`**, **`with`**, **`class`**, **`const`**, **`enum`**, **`export`**, **`extends`**, **`import`**, **`super`**, **`implements`**, **`interface`**, **`let`**, **`package`**, **`private`**, **`protected`**, **`public`**, **`static`**, **`yield`**, **`null`**, **`true`**, **`false`**.
[6] The notation used for the display of values is consistent with JSON, but also displays `undefined` and function objects.

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

All Source primitive functions, except `stringify`, can be assumed to run in $O(1)$ time, except `display`, `error` and `stringify`, which run in $O(n)$ time, where n is the size (number of components such as pairs) of their argument.

List Support

The following list processing functions are supported:

- `pair(x, y)`: *primitive*, makes a pair from `x` and `y`.
- `is_pair(x)`: *primitive*, returns `true` if `x` is a pair and `false` otherwise.
- `head(x)`: *primitive*, returns the head (first component) of the pair `x`.
- `tail(x)`: *primitive*, returns the tail (second component) of the pair `x`.
- `is_null(xs)`: *primitive*, returns `true` if `xs` is the empty list `null`, and `false` otherwise.
- `is_list(x)`: Returns `true` if `x` is a list as defined in the lectures, and `false` otherwise. Iterative process; time: $O(n)$, space: $O(1)$, 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: $O(n)$, space: $O(n)$, since the constructed list data structure consists of n pairs, each of which takes up a constant amount of space.
- `draw_data(x)`: *primitive*, visualizes `x` in a separate drawing area in the Source Academy using a box-and-pointer diagram; time, space: $O(n)$, where n is the number of data structures such as pairs in `x`.
- `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: $O(n)$, where n is the number of pairs in `x`.
- `length(xs)`: Returns the length of the list `xs`. Iterative process; time: $O(n)$, space: $O(1)$, where n is the length of `xs`.
- `map(f, xs)`: Returns a list that results from list `xs` by element-wise application of `f`. Recursive process; time: $O(n)$, space: $O(n)$, where n is the length of `xs`.
- `build_list(n, f)`: Makes a list with `n` elements by applying the unary function `f` to the numbers 0 to `n - 1`. Recursive process; time: $O(n)$, space: $O(n)$.
- `for_each(f, xs)`: Applies `f` to every element of the list `xs`, and then returns `true`. Iterative process; time: $O(n)$, space: $O(1)$, 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: $O(n)$, space: $O(n)$, where n is the length of `xs`. The process is iterative, but consumes space $O(n)$ because of the result list.
- `append(xs, ys)`: Returns a list that results from appending the list `ys` to the list `xs`. Recursive process; time: $O(n)$, space: $O(n)$, 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: $O(n)$, space: $O(1)$, 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`. Recursive process; time: $O(n)$, space: $O(n)$, 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`. Recursive process; time: $O(n)$, space: $O(n)$, 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`. Recursive process; time: $O(n)$, space: $O(n)$, 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`. Recursive process; time: $O(n)$, space: $O(n)$, where n is the length of `xs`.
- `list_ref(xs, n)`: Returns the element of list `xs` at position `n`, where the first element has index 0. Iterative process; time: $O(n)$, space: $O(1)$, where n is the length of `xs`.
- `accumulate(op, initial, xs)`: Applies binary function `op` to the elements of `xs` from right-to-left order, first applying `op` 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(op,zero,list(1,2,3))` results in `op(1, op(2, op(3, zero)))`. Recursive process; time: $O(n)$, space: $O(n)$, where n is the length of `xs`, assuming `op` takes constant time.

Numbers

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

Strings

Strings are of the form `"`*double-quote-characters*`"`, where *double-quote-characters* is a possibly empty sequence of characters without the character `"`, and of the form `'` *single-quote-characters*`'`, where *single-quote-characters* is a possibly empty sequence of characters without the character `'`,

Typing

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

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

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

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

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

Comments

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

Deviations from JavaScript

We intend the Source language to be a conservative extension of JavaScript: Every correct Source program should behave *exactly* the same using a Source implementation, as it does using a JavaScript implementation. We assume, of course, that suitable libraries are used by the JavaScript implementation, to account for the predefined names of each Source language.
This section lists some exceptions where we think a Source implementation should be allowed to deviate from the JavaScript specification, for the sake of internal consistency and esthetics.

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

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

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

Appendix: List library

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

```
// list.js START


/**
 * makes a pair whose head (first component) is <CODE>x</CODE>
 * and whose tail (second component) is <CODE>y</CODE>.
 * @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) {}

/**
 * returns <CODE>true</CODE> if <CODE>x</CODE> is a
 * pair and false otherwise.
 * @param {value} x - given value
 * @returns {boolean} whether <CODE>x</CODE> is a pair
 */
function is_pair(x) {}

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

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

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

/**
 * 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>O(n)</CODE>, space: <CODE>O(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>.
 * recurses down the list and checks that it ends with the empty list null
 * @param {value} xs - given candidate
 * @returns whether {xs} is a list
 */
function is_list(xs) {
    return is_null(xs) || (is_pair(xs) && is_list(tail(xs)));
}

/**
```

```
 * 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.
 * @param {value} value1,value2,...,value_n - given values
 * @returns {list} list containing all values
 */
function list(value1, value2, ...values ) {}

/**
 * visualizes <CODE>x</CODE> in a separate drawing
 * area in the Source Academy using a box-and-pointer diagram; time, space:
 * O(n), where n is the number of data structures such as
 * pairs in <CODE>x</CODE>.
 * @param {value} x - given value
 * @returns {value} given <CODE>x</CODE>
 */
function draw_data(x) {}

/**
 * Returns <CODE>true</CODE> if both
 * have the same structure with respect to <CODE>pair</CODE>,
 * and the same numbers, boolean values, functions or empty list
 * at corresponding leave positions (places that are not themselves pairs),
 * and <CODE>false</CODE> otherwise; time, space:
 * <CODE>O(n)</CODE>, where <CODE>n</CODE> is the number of pairs in
 * <CODE>x</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(x, y) {
    return (is_pair(x) && is_pair(y))
        ? (equal(head(x), head(y)) &&
            equal(tail(x), tail(y)))
        : x === y;
}

/**
 * Returns the length of the list
 * <CODE>xs</CODE>.
 * Iterative process; time: <CODE>O(n)</CODE>, space:
 * <CODE>O(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) {
    function iter(ys, acc) {
       return is_null(ys)
           ? acc
           : iter(tail(ys), acc + 1);
    }
    return iter(xs, 0);
}

/**
 * Returns a list that results from list
 * <CODE>xs</CODE> by element-wise application of unary function <CODE>f</CODE>.
 * Recursive process; time: <CODE>O(n)</CODE>,
 * space: <CODE>O(n)</CODE>, where <CODE>n</CODE> is the length of <CODE>xs</CODE>.
 * <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 is_null(xs)
        ? null
        : pair(f(head(xs)), map(f, tail(xs)));
}

/**
 * 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 non-negative integer.
 * Recursive process; time: <CODE>O(n)</CODE>, space: <CODE>O(n)</CODE>.
 * @param {number} n - given non-negative integer
 * @param {function} f - unary function
 * @returns {list} resulting list
 */
function build_list(n, f) {
    function build(i, f, already_built) {
        return i < 0
            ? already_built
            : build(i - 1, f, pair(f(i),
                already_built));
    }
    return build(n - 1, f, null);
}

/**
 * Applies unary function <CODE>f</CODE> to every
 * element of the list <CODE>xs</CODE>.
 * Iterative process; time: <CODE>O(n)</CODE>, space: <CODE>O(1)</CODE>,
 * 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(f, xs) {
    if (is_null(xs)) {
        return true;
    } else {
        f(head(xs));
        return for_each(f, tail(xs));
    }
}

/**
 * Returns a string that represents
 * list <CODE>xs</CODE> using the text-based box-and-pointer notation
 * <CODE>[...]</CODE>.
 * @param {list} xs - given list
 * @returns {string} <CODE>xs</CODE> converted to string
 */
function list_to_string(xs) {
    return is_null(xs)
```

```
        ? "null"
        : is_pair(xs)
            ? "[" + list_to_string(head(xs)) + "," +
                list_to_string(tail(xs)) + "]"
            : stringify(xs);
}

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

/**
 * Returns a list that results from
 * appending the list <CODE>ys</CODE> to the list <CODE>xs</CODE>.
 * Recursive process; time: <CODE>O(n)</CODE>, space:
 * <CODE>O(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 is_null(xs)
        ? ys
        : pair(head(xs),
            append(tail(xs), ys));
}

/**
 * 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>O(n)</CODE>,
 * space: <CODE>O(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. Recursive process;
 * time: <CODE>O(n)</CODE>, space: <CODE>O(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 is_null(xs)
        ? null
        : v === head(xs)
            ? tail(xs)
            : pair(head(xs),
                remove(v, tail(xs)));
}

/**
 * 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.
 * Recursive process;
 * time: <CODE>O(n)</CODE>, space: <CODE>O(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 is_null(xs)
        ? null
        : v === head(xs)
            ? remove_all(v, tail(xs))
            : pair(head(xs),
                remove_all(v, tail(xs)));
}

/**
 * Returns a list that contains
 * only those elements for which the one-argument function
 * <CODE>pred</CODE>
 * returns <CODE>true</CODE>.
 * Recursive process;
 * time: <CODE>O(n)</CODE>, space: <CODE>O(n)</CODE>,
 * 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 is_null(xs)
```

```
        ? xs
        : pred(head(xs))
            ? pair(head(xs),
                filter(pred, tail(xs)))
            : filter(pred, tail(xs));
}

/**
 * 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>.
 * Recursive process;
 * time: <CODE>O(n)</CODE>, space: <CODE>O(n)</CODE>,
 * where <CODE>n</CODE> is the length of <CODE>xs</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 start > end
        ? null
        : pair(start,
            enum_list(start + 1, end));
}

/**
 * Returns the element
 * of list <CODE>xs</CODE> at position <CODE>n</CODE>,
 * where the first element has index 0.
 * Iterative process;
 * time: <CODE>O(n)</CODE>, space: <CODE>O(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>.
 * Recursive process;
 * time: <CODE>O(n)</CODE>, space: <CODE>O(n)</CODE>,
 * where <CODE>n</CODE> is the length of <CODE>xs</CODE>
 * assuming <CODE>f</CODE> takes constant time.
 * @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 is_null(xs)
        ? initial
        : f(head(xs),
            accumulate(f, initial, tail(xs)));
}

//
// list.js END
```