National University of Singapore
School of Computing
Martin Henz

Source §4, 2018

November 4, 2018

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 2016 (7^{th} 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 §3, Source §4 has the following changes:

- literal object expressions
- dot abbreviation (see Section **Objects**)
- Source §4 adds the functions `parse`, `apply_in_underlying_javascript`, and `JSON.stringify`. Furthermore, the functions `is_boolean`, `is_number`, `is_string`, `is_function`, `is_object` and `is_array` are available, with their obvious meaning. Following JavaScript, `is_object` returns `true` when applied to arrays. For details, see Section "Interpreter Support" below.

Programs

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

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

<table>
<tr><td>statement</td><td>::=</td><td>const name = expression ;</td><td>constant declaration</td></tr>
<tr><td></td><td>|</td><td>let ;</td><td>variable declaration</td></tr>
<tr><td></td><td>|</td><td>assignment ;</td><td>variable assignment</td></tr>
<tr><td></td><td>|</td><td>expression[expression] = expression ;</td><td>array/object assignment</td></tr>
<tr><td></td><td>|</td><td>function name (parameters) block</td><td>function declaration</td></tr>
<tr><td></td><td>|</td><td>return expression ;</td><td>return statement</td></tr>
<tr><td></td><td>|</td><td>if-statement</td><td>conditional statement</td></tr>
<tr><td></td><td>|</td><td>while (expression) block</td><td>while loop</td></tr>
<tr><td></td><td>|</td><td>for ((assignment | let) ;
    expression ;
    assignment) block</td><td>for loop</td></tr>
<tr><td></td><td>|</td><td>break ;</td><td>break statement</td></tr>
<tr><td></td><td>|</td><td>continue ;</td><td>continue statement</td></tr>
<tr><td></td><td>|</td><td>statement statement</td><td>statement sequence</td></tr>
<tr><td></td><td>|</td><td>block</td><td>block statement</td></tr>
<tr><td></td><td>|</td><td>expression ;</td><td>expression statement</td></tr>
<tr><td></td><td>|</td><td>ϵ</td><td>empty statement</td></tr>
<tr><td>parameters</td><td>::=</td><td>ϵ | name (, name) ...</td><td>function parameters</td></tr>
<tr><td>if-statement</td><td>::=</td><td>if (expression) block
else (block | if-statement)</td><td>conditional statement</td></tr>
<tr><td>block</td><td>::=</td><td>{ statement }</td><td>block statement</td></tr>
<tr><td>let</td><td>::=</td><td>let name = expression</td><td>variable declaration</td></tr>
<tr><td>assignment</td><td>::=</td><td>name = expression</td><td>variable assignment</td></tr>
<tr><td>expression</td><td>::=</td><td>number</td><td>primitive number expression</td></tr>
<tr><td></td><td>|</td><td>true | false</td><td>primitive boolean expression</td></tr>
<tr><td></td><td>|</td><td>string</td><td>primitive string expression</td></tr>
<tr><td></td><td>|</td><td>name</td><td>name expression</td></tr>
<tr><td></td><td>|</td><td>expression binary-operator expression</td><td>binary operator combination</td></tr>
<tr><td></td><td>|</td><td>unary-operator expression</td><td>unary operator combination</td></tr>
<tr><td></td><td>|</td><td>expression (expressions)</td><td>function application</td></tr>
<tr><td></td><td>|</td><td>(name | (parameters)) => expression</td><td>function definition expression</td></tr>
<tr><td></td><td>|</td><td>expression ? expression : expression</td><td>conditional expression</td></tr>
<tr><td></td><td>|</td><td>[]</td><td>empty list/array expression</td></tr>
<tr><td></td><td>|</td><td>expression[expression]</td><td>array/object access</td></tr>
<tr><td></td><td>|</td><td>{ properties }</td><td>literal object expression</td></tr>
<tr><td></td><td>|</td><td>(expression)</td><td>parenthesised expression</td></tr>
<tr><td>binary-operator</td><td>::=</td><td>+ | - | * | / | % | === | !==</td><td></td></tr>
<tr><td></td><td>|</td><td>> | < | >= | <= | && | ||</td><td></td></tr>
<tr><td>unary-operator</td><td>::=</td><td>! | -</td><td></td></tr>
<tr><td>expressions</td><td>::=</td><td>ϵ | expression (, expression) ...</td><td>argument expressions</td></tr>
<tr><td>properties</td><td>::=</td><td>ϵ | property (, property) ...</td><td>object properties</td></tr>
<tr><td>property</td><td>::=</td><td>(string | name) : expression</td><td>object property</td></tr>
</table>

Binary boolean operators

Conjunction

$$expression_1 \; \texttt{\&\&} \; expression_2$$

stands for

$$expression_1 \; \texttt{?} \; expression_2 \; \texttt{:} \; \texttt{false}$$

Disjunction

$$expression_1 \; \texttt{||} \; expression_2$$

stands for

$$expression_1 \; \texttt{?} \; \texttt{true} \; \texttt{:} \; expression_2$$

Loops

while-loops

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;
    }
}
```

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
        }
    }
}
```

Restrictions

- Return statements are only allowed in bodies of functions.
- Return statements are not allowed in the bodies of while and for loops.
- 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.
- Local functions within an outer function must precede all other statements in body of the outer function.

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 builtin 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
- `display(a)`: Displays *any* value `a` in the console; returns `undefined`.
- `error(a)`: Displays *any* value `a` in the console with error flag. The evaluation of any call of `error` aborts the running program immediately.
- `prompt(s)`: Pops up a window that displays the *string* `s`, provides an input line for the user to enter a text and an "OK" button. The call of `prompt` suspends execution of the program until the "OK" button is pressed, at which point it returns the entered text as a string.
- `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.

---

[2] In ECMAScript 2016 (7th 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`**.

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

List Support

The following list processing functions are supported:

- `pair(x, y)`: *builtin*, makes a pair from `x` and `y`.
- `is_pair(x)`: *builtin*, returns `true` if `x` is a pair and `false` otherwise.
- `head(x)`: *builtin*, returns the head (first component) of the pair `x`.
- `tail(x)`: *builtin*, returns the tail (second component) of the pair `x`.
- `is_empty_list(xs)`: *builtin*, returns `true` if `xs` is the empty list, 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)`: *builtin*, 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_list(x)`: *builtin*, 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 pairs in `x`.
- `equal(x1, x2)`: Returns `true` if both have the same structure and the same numbers, boolean values, functions or empty list at corresponding leaf positions, 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.

Pair Mutators

The following pair mutator functions are supported:

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

Array Support

The following array processing function is supported:

- `array_length(x)`: *builtin*, 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`.

Stream Support

The following stream processing functions are supported:

- `stream_tail(x)`: *Builtin*, 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)`: *Builtin*, 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.
- `stream(x1, x2,..., xn)`: *Builtin*, 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`.
- `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 `[]` 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.

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 `'`,

Arrays

Arrays in Source are created using the empty array syntax:

```
let my_array = [];
```

Arrays in Source are limited to integers as keys. In statements like

```
a[i];
a[j] = v;
```

the values `i` and `j` must be integers if `a` is an array.

Objects

Object properties

Literal objects can be created using literal object expressions:

```
var my_obj = { "key 1": 1,
               "key 2": 2 };
```

As keys, only strings are allowed in Source. If the string (without quotation marks) looks like a *name*, the quotation marks around property keys can be omitted, for example:

```
var my_motorcycle = { cc: 399,
                      number_of_cylinders: 4 };
```

The syntax

```
my_obj["key 1"];
my_motorcycle["cc"] = 1249;
```

allows for object access and assignment. Here, the quotation marks are not optional, even when the string (without quotation marks) looks like a *name*.

Dot Abbreviation

In *statement* and *expression*, the syntax

expression **.** *id*

is an abbreviation for

expression **["***id***"]**

For example

```
my_motorcyle . number_of_cylinders;
my_motorcycle . cc = 1249;
```

are abbreviations for

```
my_motorcyle [ "number_of_cylinders" ];
my_motorcycle [ "cc" ] = 1249;
```

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	any	string
`+`	any	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.

Interpreter Support

- `is_number(x)`: *builtin*, returns `true` if `x` is a number, and `false` otherwise.
- `is_boolean(x)`: *builtin*, returns `true` if `x` is `true` or `false`, and `false` otherwise.
- `is_string(x)`: *builtin*, returns `true` if `x` is a string, and `false` otherwise.
- `is_function(x)`: *builtin*, returns `true` if `x` is a function, and `false` otherwise.
- `is_object(x)`: *builtin*, returns `true` if `x` is an object, and `false` otherwise. Following JavaScript, arrays are considered objects.
- `is_array(x)`: *builtin*, returns `true` if `x` is an array, and `false` otherwise. The empty array `[]`, also known as the empty list, is an array.
- `parse(x)`: *builtin*, returns the parse tree that results from parsing the string `x` as a Source program.
- `JSON.stringify(x)`: *builtin*, returns a string that represents the given JSON object `x`.
- `apply_in_underlying_javascript(f, xs)`: *builtin*, calls the function `f` with arguments `xs`. For example:

```
function times(x, y) {
   return x * y;
}
apply_in_underlying_javascript(times, list(2, 3)); // returns 6
```

Comments

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

Appendix: List library

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

```
// is_list recurses down the list and checks that it ends with the empty list []

function is_list(xs) {
    return is_empty_list(xs) || (is_pair(xs) && is_list(tail(xs)));
}

// equal computes the structural equality
// over its arguments

function equal(item1, item2){
    return (is_pair(item1) && is_pair(item2))
        ? (equal(head(item1), head(item2)) &&
           equal(tail(item1), tail(item2)))
        : (is_empty_list(item1) && is_empty_list(item2))
          || item1 === item2;
}

// returns the length of a given argument list
// assumes that the argument is a list

function length(xs) {
    return is_empty_list(xs)
        ? 0
        : 1 + length(tail(xs));
}

// map applies first arg f, assumed to be a unary function,
// to the elements of the second argument, assumed to be a list.
// f is applied element-by-element:
// map(f, [1, [2, []]]) results in [f(1), [f(2), []]]

function map(f, xs) {
    return is_empty_list(xs)
        ? []
        : pair(f(head(xs)), map(f, tail(xs)));
}

// build_list takes a non-negative integer n as first argument,
// and a function fun as second argument.
// build_list returns a list of n elements, that results from
// applying fun to the numbers from 0 to n-1.

function build_list(n, fun){
    function build(i, fun, already_built) {
        return i < 0
            ? already_built
            : build(i - 1, fun, pair(fun(i),
                                     already_built));
    }
    return build(n - 1, fun, []);
}

// for_each applies first arg fun, assumed to be a unary function,
// to the elements of the second argument, assumed to be a list.
// fun is applied element-by-element:
// for_each(fun, [1, [2, []]]) results in the calls fun(1) and fun(2).
// for_each returns true.
```

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

// to_string uses JavaScript's + to turn its argument into a string

function to_string(x) {
    return x + "";
}

// list_to_string returns a string that represents the argument list.
// It applies itself recursively on the elements of the given list.
// When it encounters a non-list, it applies toString to it.

function list_to_string(xs) {
    return is_empty_list(xs)
        ? "[]"
        : is_pair(xs)
            ? "[" + list_to_string(head(xs)) + ","+
                    list_to_string(tail(xs))      + "]"
            : to_string(xs);
}

// reverse reverses the argument, assumed to be a list

function reverse(xs) {
    function rev(original, reversed) {
        return is_empty_list(original)
            ? reversed
            : rev(tail(original),
                  pair(head(original), reversed));
    }
    return rev(xs, []);
}

// append first argument, assumed to be a list, to the second argument.
// In the result, the [] at the end of the first argument list
// is replaced by the second argument, regardless what the second
// argument consists of.

function append(xs, ys) {
    return is_empty_list(xs)
        ? ys
        : pair(head(xs),
               append(tail(xs), ys));
}

// member looks for a given first-argument element in the
// second argument, assumed to be a list. It returns the first
// postfix sublist that starts with the given element. It returns [] if the
// element does not occur in the list

function member(v, xs){
    return is_empty_list(xs)
```

```
        ? []
        : (v === head(xs))
            ? xs
            : member(v, tail(xs));
}

// removes the first occurrence of a given first-argument element
// in second-argument, assmed to be a list. Returns the original
// list if there is no occurrence.

function remove(v, xs){
    return is_empty_list(xs)
        ? []
        : v === head(xs)
            ? tail(xs)
            : pair(head(xs),
                   remove(v, tail(xs)));
}

// Similar to remove, but removes all instances of v
// instead of just the first

function remove_all(v, xs) {
    return is_empty_list(xs)
        ? []
        : v === head(xs)
            ? remove_all(v, tail(xs))
            : pair(head(xs),
                   remove_all(v, tail(xs)));
}

// filter returns the sublist of elements of the second argument
// (assumed to be a list), for which the given predicate function
// returns true.

function filter(pred, xs){
    return is_empty_list(xs)
        ? xs
        : pred(head(xs))
            ? pair(head(xs),
                   filter(pred, tail(xs)))
            : filter(pred, tail(xs));
}

// enumerates numbers starting from start, assumed to be a number,
// using a step size of 1, until the number exceeds end, assumed
// to be a number

function enum_list(start, end) {
    return start > end
        ? []
        : pair(start,
               enum_list(start + 1, end));
}

// Returns the item in xs (assumed to be a list) at index n,
// assumed to be a non-negative integer.
// Note: the first item is at position 0

function list_ref(xs, n) {
```

```
    return n === 0
        ? head(xs)
        : list_ref(tail(xs), n - 1);
}

// accumulate applies an operation op (assumed to be a binary function)
// to elements of sequence (assumed to be a list) in a right-to-left order.
// first apply op to the last element and initial, resulting in r1, then to
// the  second-last element and r1, resulting in r2, etc, and finally
// to the first element and r_n-1, where n is the length of the
// list.
// accumulate(op, zero, list(1, 2, 3)) results in
// op(1, op(2, op(3, zero)))

function accumulate(f, initial, xs) {
    return is_empty_list(xs)
        ? initial
        : f(head(xs),
             accumulate(f, initial, tail(xs)));
}
```

Appendix: Stream library

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

```
// stream.js: Supporting streams in the Scheme style, following
//            "stream discipline"
// A stream is either the empty list or a pair whose tail is
// a nullary function that returns a stream.

// list_to_stream transforms a given list to a stream
// Lazy? Yes: list_to_stream goes down the list only when forced
function list_to_stream(xs) {
    return is_empty_list(xs)
        ? []
        : pair(head(xs),
               () => list_to_stream(tail(xs)));
}

// stream_to_list transforms a given stream to a list
// Lazy? No: stream_to_list needs to force the whole stream
function stream_to_list(xs) {
    return is_empty_list(xs)
        ? []
        : pair(head(xs), stream_to_list(stream_tail(xs)));
}

// stream_length returns the length of a given argument stream
// throws an exception if the argument is not a stream
// Lazy? No: The function needs to explore the whole stream
function stream_length(xs) {
    return is_empty_list(xs)
        ? 0
        : 1 + stream_length(stream_tail(xs));
}

// stream_map applies first arg f to the elements of the second
// argument, assumed to be a stream.
// f is applied element-by-element:
// stream_map(f,list_to_stream([1,[2,[]]])) results in
// the same as list_to_stream([f(1),[f(2),[]]])
// stream_map throws an exception if the second argument is not a
// stream, and if the second argument is a non-empty stream and the
// first argument is not a function.
// Lazy? Yes: The argument stream is only explored as forced by
//            the result stream.
function stream_map(f, s) {
    return is_empty_list(s)
        ? []
        : pair(f(head(s)),
                 () => stream_map(f, stream_tail(s)));
}

// build_stream takes a non-negative integer n as first argument,
// and a function fun as second argument.
// build_list returns a stream of n elements, that results from
// applying fun to the numbers from 0 to n-1.
// Lazy? Yes: The result stream forces the applications of fun
//            for the next element
function build_stream(n, fun) {
    function build(i) {
        return i >= n
```

```
            ? []
            : pair(fun(i),
                   () => build(i + 1));
    }
    return build(0);
}

// stream_for_each applies first arg fun to the elements of the list
// passed as second argument. fun is applied element-by-element:
// for_each(fun,list_to_stream([1,[2,[]]])) results in the calls fun(1)
// and fun(2).
// stream_for_each returns true.
// stream_for_each throws an exception if the second argument is not a list,
// and if the second argument is a non-empty list and the
// first argument is not a function.
// Lazy? No: stream_for_each forces the exploration of the entire stream
function stream_for_each(fun, xs) {
    if (is_empty_list(xs)) {
        return true;
    } else {
        fun(head(xs));
        return stream_for_each(fun, stream_tail(xs));
    }
}

// stream_reverse reverses the argument stream
// stream_reverse throws an exception if the argument is not a stream.
// Lazy? No: stream_reverse forces the exploration of the entire stream
function stream_reverse(xs) {
    function rev(original, reversed) {
        return is_empty_list(original)
            ? reversed
            : rev(stream_tail(original),
                  pair(head(original), () => reversed));
    }
    return rev(xs,[]);
}

// stream_append appends first argument stream and second argument stream.
// In the result, the [] at the end of the first argument stream
// is replaced by the second argument stream
// stream_append throws an exception if the first argument is not a
// stream.
// Lazy? Yes: the result stream forces the actual append operation
function stream_append(xs, ys) {
    return is_empty_list(xs)
        ? ys
        : pair(head(xs),
               () => stream_append(stream_tail(xs), ys));
}

// stream_member looks for a given first-argument element in a given
// second argument stream. It returns the first postfix substream
// that starts with the given element. It returns [] if the
// element does not occur in the stream
// Lazy? Sort-of: stream_member forces the stream only until the element is found.
function stream_member(x, s) {
    return is_empty_list(s)
        ? []
        : head(s) === x
```

```
            ? s
            : stream_member(x, stream_tail(s));
}

// stream_remove removes the first occurrence of a given first-argument element
// in a given second-argument list. Returns the original list
// if there is no occurrence.
// Lazy? Yes: the result stream forces the construction of each next element
function stream_remove(v, xs) {
    return is_empty_list(xs)
        ? []
        : v === head(xs)
            ? stream_tail(xs)
            :  pair(head(xs),
                    () => stream_remove(v, stream_tail(xs)));
}

// stream_remove_all removes all instances of v instead of just the first.
// Lazy? Yes: the result stream forces the construction of each next element
function stream_remove_all(v, xs) {
    return is_empty_list(xs)
        ? []
        : v === head(xs)
            ? stream_remove_all(v, stream_tail(xs))
            : pair(head(xs), () => stream_remove_all(v, stream_tail(xs)));
}

// filter returns the substream of elements of given stream s
// for which the given predicate function p returns true.
// 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 p holds.
function stream_filter(p, s) {
    return is_empty_list(s)
        ? []
        : p(head(s))
            ? pair(head(s),
                   () => stream_filter(p, stream_tail(s)));
            : stream_filter(p, stream_tail(s));
}

// enumerates numbers starting from start,
// using a step size of 1, until the number
// exceeds end.
// Lazy? Yes: The result stream forces the construction of
//            each next element
function enum_stream(start, end) {
    return start > end
        ? []
        : pair(start,
               () => enum_stream(start + 1, end));
}

// integers_from constructs an infinite stream of integers
// starting at a given number n
// Lazy? Yes: The result stream forces the construction of
//            each next element
function integers_from(n) {
    return pair(n,
```

```
                () => integers_from(n + 1));
}

// eval_stream constructs the list of the first n elements
// of a given stream s
// Lazy? Sort-of: eval_stream only forces the computation of
//                the first n elements, and leaves the rest of
//                the stream untouched.
function eval_stream(s, n) {
    return n === 0
        ? []
        : pair(head(s),
               eval_stream(stream_tail(s),
                           n - 1));
}

// Returns the item in stream s at index n (the first item is at position 0)
// Lazy? Sort-of: stream_ref only forces the computation of
//                the first n elements, and leaves the rest of
//                the stream untouched.
function stream_ref(s, n) {
    return n === 0
        ? head(s)
        : stream_ref(stream_tail(s), n - 1);
}
```