National University of Singapore
School of Computing
Martin Henz

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

- `[]`: Empty list.
- List library: Functions for creating, accessing and processing lists.

Programs

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

statement	::=	**`const`** *name* **`=`** *expression* **`;`**	constant declaration
	|	**`function`** *name* **`(`** *parameters* **`)`** *block*	function declaration
	|	**`return`** *expression* **`;`**	return statement
	|	*if-statement*	conditional statement
	|	*statement statement*	statement sequence
	|	*block*	block statement
	|	*expression* **`;`**	expression statement
	|	ϵ	empty statement
parameters	::=	ϵ | *name* (**`,`** *name*) ...	function parameters
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
	|	*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 expression
	|	*expression* **`?`** *expression* **`:`** *expression*	conditional expression
	|	**`[]`**	primitive empty list expression
	|	**`(`** *expression* **`)`**	parenthesised expression
binary-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.
- 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.
- `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`.

[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`**.

- `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.

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	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.

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)));
}
```