National University of Singapore
School of Computing
Martin Henz

Source §1, 2018

May 18, 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 (7th 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:

Programs

A Source program is a *statement*, defined using Backus-Naur Form[2] 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.

[2] 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.

statement	::=	**const** *name* **=** *expression* **;**	constant declaration
	|	**function** *name* **(** *parameters* **)**	
		{ *statement* **}**	function declaration
	|	**return** *expression* **;**	return statement
	|	*if-statement*	conditional statement
	|	*statement statement*	statement sequence
	|	*expression* **;**	expression statement
parameters	::=	ϵ | *name* (**,** *name*) ...	function parameters
if-statement	::=	**if (** *expression* **) {** *statement* **}**	
		else (**{** *statement* **}** | *if-statement*)	conditional 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* **)**	(compound) 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

return statements

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

Names

Names[3] start with _, $ or a letter[4] and contain only _, $, letters or digits[5]. Reserved words[6] 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 therefore the familiarity of the audience with the characters used in names is an important aspect of program readability.
The following names can be used, in addition to names that are declared using **const**, **function** and **=>**:

[3] In ECMAScript 2016 (7th Edition), these names are called *identifiers*.
[4] By *letter* we mean Unicode letters (L) or letter numbers (Nl).
[5] 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.
[6] 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**.

- `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
- `error(a)`: Displays *any* value `a` in the console with error flag
- `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

Source Week 5 supports the following list processing functions:

- `pair(x, y)`: Makes a pair from `x` and `y`.
- `is_pair(x)`: Returns `true` if `x` is a pair and `false` otherwise.
- `head(x)`: Returns the head (first component) of the pair `x`.
- `tail(x)`: Returns the tail (second component) of the pair `x`.
- `is_empty_list(xs)`: 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)`: 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.
- `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.

Names

Names[7] start with _, `$` or a letter[8] and contain only _, `$`, letters or digits[9]. Reserved words[10] 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 therefore the familiarity of the audience with the characters used in names is an important aspect of program readability.
The following names can be used, in addition to names that are declared using **`const`**, **`function`** and **`=>`**:

- `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
- `error(a)`: Displays *any* value `a` in the console with error flag
- `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.

[7] In ECMAScript 2016 (7th Edition), these names are called *identifiers*.
[8] By *letter* we mean Unicode letters (L) or letter numbers (Nl).
[9] 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.
[10] 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`**.

- `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.

Numbers

We use decimal notation for numbers, with an optional decimal dot. "Scientific notation" (multiplying the number with a power of 10) is indicated with the letter `e`. 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
===	number	number	bool
===	bool	bool	bool
===	string	string	bool
===	function	function	bool
!==	number	number	bool
!==	bool	bool	bool
!==	string	string	bool
!==	function	function	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	bool	bool
||	bool	bool	bool
!	bool		bool
-	number		number

Preceding **?**, 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.