National University of Singapore
School of Computing
CS1101S: Programming Methodology
Semester I, 2017/2018

Source
Week 11

Introduction

The language Source is the official language of CS1101S. You have never heard of Source? No wonder, because we invented it just for the purpose of this module. Source is a sublanguage of ECMAScript 2016 (8th Edition) and defined in the documents titled "Source Week x". More specifically, the missions, side quests, competitions, practical, midterm and final assessments use the Source language of the current week or the next week x for which a document "Source Week x" is available.

Changes

Week 10 adds the functions `parse`, `apply_in_underlying_javascript`, and `JSON.stringify`, see Section "Miscellaneous Functions" below.

Statements

A Source program is a statement. Statements are defined using Backus-Naur Form (BNF) as follows:

```
⟨statement⟩  ::=  ;
               |  var ⟨id⟩ = ⟨expression⟩ ;
               |  ⟨id⟩ = ⟨expression⟩ ;
               |  ⟨expression⟩ [ ⟨expression⟩ ] = ⟨expression⟩ ;
               |  ⟨if-statement⟩
               |  while ( ⟨expression⟩ ) { ⟨statement⟩ }
               |  for ( ⟨expression⟩ ; ⟨expression⟩ ; ⟨expression⟩ ) { ⟨statement⟩ }
               |  function ⟨id⟩ ( ⟨id-list⟩ ) { ⟨statement⟩}
               |  ⟨statement⟩ ⟨statement⟩
               |  return ⟨expression⟩ ;
               |  break ;
               |  continue ;
```

```
                       |   ⟨expression⟩ ;

       ⟨if-statement⟩ ::=  if ( ⟨expression⟩ ) { ⟨statement⟩ } else { ⟨statement⟩ }
                       |   if ( ⟨expression⟩ ) { ⟨statement⟩ } else ⟨if-statement⟩

            ⟨id-list⟩ ::=
                       |   ⟨non-empty-id-list⟩

  ⟨non-empty-id-list⟩ ::=  ⟨id⟩
                       |   ⟨id⟩ , ⟨non-empty-id-list⟩
```

Important note: There cannot be any newline character between `return` and ⟨*expression*⟩ ;.

```
           ⟨expression⟩ ::=  ⟨number⟩
                         |   true | false
                         |   ⟨string⟩
                         |   ⟨expression⟩ ⟨bin-op⟩ ⟨expression⟩
                         |   ⟨un-op⟩ ⟨expression⟩
                         |   function ( ⟨id-list⟩ ) { ⟨statement⟩}
                         |   ⟨id⟩ ( ⟨expr-list⟩ )
                         |   ( ⟨expression⟩ ) ( ⟨expr-list⟩ )
                         |   ⟨expression⟩ ? ⟨expression⟩ : ⟨expression ⟩
                         |   [ ⟨expr-list⟩ ]
                         |   ⟨expression⟩ [ ⟨expression⟩ ]
                         |   []
                         |   ( ⟨expression⟩ )

               ⟨bin-op⟩ ::=  + | - | * | / | % | === | !== | > | < | >= | <= | && | ||

                ⟨un-op⟩ ::=  ! | -

            ⟨expr-list⟩ ::=
                         |   ⟨non-empty-expr-list⟩

  ⟨non-empty-expr-list⟩ ::=  ⟨expression⟩
                         |   ⟨expression⟩ , ⟨non-empty-expr-list⟩
```

Identifiers

Variables in Source are syntactically represented by identifiers. In Source, an identifier consists of digits (0,...,9), the underline character _ and letters (a,...z,A,...Z) and begins with a letter or the underline character.

Builtin Functionality

The following identifiers can be used, in addition to identifiers that are declared using **var** and **function**:

- `undefined`: Refers to the value `undefined`
- `alert(string)`: Pops up a window that displays the string
- `display(value)`: Displays a value in the console
- `prompt(string)`: Pops up a window that displays the string and an entry space. The user can enter his own string in the entry space and press "OK". After that, `prompt` returns the string that the user entered.
- `parseInt(string)`: Interprets the given string as an integer and returns that integer.
- `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`: Refers to the square root function.

List Support

Source Week 9 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`.
- `set_head(p, x)`: Sets the head (first component) of the pair `p` to be `x`; returns `undefined`.
- `set_tail(p, x)`: Sets the tail (second component) of the pair `p` to be `x`; returns `undefined`.
- `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.

Miscellaneous Functions

- `is_number(x)`: Returns `true` if `x` is a number, and `false` otherwise.

- `equal(x, y)`: Returns `true` if `x` and `y` have the same structure (using pairs and `[]`), and corresponding leaves are `===`, and `false` otherwise.
- `array_length(x)`: 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`.
- `parse(x)`: returns the parse tree that results from parsing the string `x` as a Source program.
- `JSON.stringify(x)`: returns a string that represents the given JSON object `x`.
- `apply_in_underlying_javascript(f, xs)`: 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
```

Numbers

Examples for numbers are `5432`, `-5432.109`, and `-43.21e-45`.

Strings

Strings are of the form `"`⟨*characters*⟩`"`, where the character `"` does not appear in ⟨*characters*⟩, and of the form `'`⟨*characters*⟩`'`, where the character `'` does not appear in ⟨*characters*⟩.

Typing

Expressions evaluate to numbers, boolean values, strings or function values.

Only function values can be applied using the syntax:

$$\begin{array}{rcl} \langle \textit{expression} \rangle & ::= & \langle \textit{id} \rangle \ \texttt{(} \ \langle \textit{expr-list} \rangle \ \texttt{)} \\ & | & \texttt{(} \ \langle \textit{expression} \rangle \ \texttt{)} \ \texttt{(} \ \langle \textit{expr-list} \rangle \ \texttt{)} \end{array}$$

The following table specifies what arguments Source's operators take and what results they return. The type "e-pair" refers to the empty list `[]` or a pair.

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
===	e-pair	e-pair	bool
===	any	undefined	bool
===	undefined	any	bool
!==	number	number	bool
!==	bool	bool	bool
!==	string	string	bool
!==	function	function	bool
!==	e-pair	e-pair	bool
!==	any	undefined	bool
!==	undefined	any	bool
>	number	number	bool
<	number	number	bool
>=	number	number	bool
<=	number	number	bool
&&	bool	bool	bool
||	bool	bool	bool
!	bool		bool
-	number		number

Following **if** and preceding ?, Source only allows boolean expressions.

Arrays

Arrays in Source are created using the empty array syntax:

```
var 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.

Object-oriented Programming

Object properties

Literal objects can be created using the object creation syntax:

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

As keys, only strings are allowed in Source. The syntax

```
my_obj["key 1"];
my_obj["key 2"] = 3;
```

allows for object access and assignment.

Dot Abbreviation

In ⟨*statement*⟩ and ⟨*expression*⟩, the syntax

⟨*expression*⟩ `.` ⟨*id*⟩

stands for

⟨*expression*⟩ `[` "⟨*id*⟩" `]`

and

⟨*expression*⟩ `.` ⟨*id*⟩ `(...)`

stands for

`var newid =` ⟨*expression*⟩`;` `(newid [` "⟨*id*⟩" `]).call(newid,...)`

Creating Objects

Objects can be created using the keyword **new** followed by a function that serves as “class”.

⟨*expression*⟩	::=	[**new**] ⟨*id*⟩**(** ⟨*expr-list*⟩ **)**
	|	[**new**] **(** ⟨*expression*⟩ **)(** ⟨*expr-list*⟩ **)**

Declaring Methods

A method for a class `F` is declared using the following construct:

```
F.prototype.methodname = function(...) {...};
```

Inheritance

A class `G` can inherit a class `F` using:

```
G.Inherits(F);
```

The call of `Inherits` must happen *before* the `prototype` of the class is used, and there must only be *one* `G.Inherits` call for any class `G`.

Calling a Constructor

Within a constructor function `F`, you can call an inherited constructor function `G` using the following construct:

```
G.call(this,...)
```

Invoking a Method

Within a method of class `F`, you can call an inherited method of class `G` using the following construct:

```
G.prototype.methodname.call(this,...)
```

Comments

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

After “`//`” any characters until the next newline character is ignored.

Remarks

Variable declarations with **var** occurring inside of functions declare variables to be usable in the entire function, even if they appear in only one branch of a conditional. It is therefore good practice to declare variables only in the beginning of functions.