Introduction

The language JediScript is the official language of CS1101S. You have never heard of JediScript? No wonder, because we invented it just for the purpose of this module. It is a strict sublanguage of JavaScript Version 1.8.5 and as such is fully supported by our development environment (Komodo Edit and Firefox).

It is defined in this document (and its successors); updated weekly when necessary. The missions, side quests, competitions and practical exams use the JediScript language of that week. Students will receive deductions of marks starting in Week 3 if they are using JavaScript construct that are not part of JediScript of that week.

Statements

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

\[
\begin{align*}
\langle \text{statement} \rangle & ::= \; ; \\
& \mid \text{var} \langle \text{id} \rangle = \langle \text{expression} \rangle \; ; \\
& \mid \text{if} \ (\langle \text{expression} \rangle) \ \{ \langle \text{statement} \rangle \} \ \text{else} \ \{ \langle \text{statement} \rangle \} \\
& \mid \text{function} \ (\langle \text{id} \rangle) \ (\langle \text{id-list} \rangle) \ \{ \langle \text{statement} \rangle \} \\
& \mid \text{switch} \ (\langle \text{expression} \rangle) \ \{ \\
& \quad \langle \text{switch-cases} \rangle \\
& \quad \text{default} : \ \langle \text{statement} \rangle \} \\
& \mid \langle \text{statement} \rangle \ \langle \text{statement} \rangle \\
& \mid \text{return} \ \langle \text{expression} \rangle \; ; \\
& \mid \langle \text{expression} \rangle \\
\end{align*}
\]

\[
\begin{align*}
\langle \text{id-list} \rangle & ::= \\
& \mid \langle \text{non-empty-id-list} \rangle \\
\end{align*}
\]

\[
\begin{align*}
\langle \text{non-empty-id-list} \rangle & ::= \langle \text{id} \rangle \\
& \mid \langle \text{id} \rangle , \langle \text{non-empty-id-list} \rangle \\
\end{align*}
\]
Important note: There cannot be any newline character between `return` and `(expression) ;`.

\[
\langle \text{switch-cases} \rangle ::= \\
\quad | \quad \text{case} \langle \text{expression} \rangle : \langle \text{statement} \rangle \text{ break; } \langle \text{switch-cases} \rangle
\]

\[
\langle \text{expression} \rangle ::= \langle \text{number} \rangle \\
\quad | \quad \text{true} | \text{false} \\
\quad | \quad \langle \text{string} \rangle \\
\quad | \quad \langle \text{expression} \rangle \langle \text{bin-op} \rangle \langle \text{expression} \rangle \\
\quad | \quad \langle \text{un-op} \rangle \langle \text{expression} \rangle \\
\quad | \quad \text{function} \{[\langle \text{id} \rangle] \{ \langle \text{id-list} \rangle \} \{ \langle \text{statement} \rangle \}
\quad | \quad \langle \text{id} \rangle \{ \langle \text{expr-list} \rangle \}
\quad | \quad \langle \langle \langle \langle \langle \text{expression} \rangle \rangle \rangle \rangle \rangle \langle \text{expr-list} \rangle \\
\quad | \quad \langle \text{expression} \rangle \langle \text{bin-op} \rangle \langle \text{expression} \rangle \\
\quad | \quad \langle \text{un-op} \rangle \langle \text{expression} \rangle \\
\quad | \quad \langle \langle \langle \langle \langle \text{expression} \rangle \rangle \rangle \rangle \langle \text{expr-list} \rangle \\
\quad | \quad \langle \text{expression} \rangle \langle \text{expression} \rangle ? \langle \text{expression} \rangle : \langle \text{expression} \rangle \\
\quad | \quad \langle \emptyset \rangle \\
\quad | \quad (\langle \text{expression} \rangle)
\]

\[
\langle \text{bin-op} \rangle ::= + | - | \ast | / | \% | \text{===} | \text{!==} | \text{> } | \text{< } | \text{>= } | \text{<= } | \& \& | \text{||}
\]

\[
\langle \text{un-op} \rangle ::= ! | -
\]

\[
\langle \text{expr-list} \rangle ::= \langle \text{non-empty-expr-list} \rangle
\]

\[
\langle \text{non-empty-expr-list} \rangle ::= \langle \text{expression} \rangle \\
\quad | \quad \langle \text{expression} \rangle , \langle \text{non-empty-expr-list} \rangle
\]

**Identifiers**

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

**JavaScript Builtin Functionality**

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

- `alert(string)`: Pops up a window that displays the string
- `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
pages 159 and following, and

• `Math.E`: Refers to the mathematical constant $e$.
• `Math.PI`: Refers to the mathematical constant $\pi$.
• `Math.sqrt`: Refers to the square root function.

Note that technically, `Math.<name>` is not an identifier, but more similar to an operator combination, the operator being `.`. We will learn more about this construct when learning about objects.

**List Support**

JediScript Week 6 supports all list functions defined in the library `list.js`:

• `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)`: Can only be applied to the empty list or a pair. Returns `true` if xs is the empty list, and `false` if xs is a pair.
• `is_list(x)`: Returns `true` if x is a list as defined in the lectures, and `false` otherwise.
• `list(x1, x2, ..., xn)`: Returns a list with n elements. The first element is x1, the second x2, etc.
• `length(xs)`: Returns the length of the list xs.
• `map(f, xs)`: Returns a list that results from list xs by element-wise application of f.
• `build_list(n, f)`: Makes a list with n elements by applying the unary function f to the numbers 0 to n - 1.
• `for_each(f, xs)`: Applies f to every element of the list xs, and then returns `true`.
• `list_to_string(xs)`: Returns a string that represents list xs using the `[...]` notation.
• `reverse(xs)`: Returns list xs in reverse order.
• `append(xs, ys)`: Returns a list that results from appending the list ys to the list xs.
• `member(x, xs)`: Returns first postfix sublist whose head is equal to x (===); returns `[]` if the element does not occur in the list.
• remove(x, xs): Returns a list that results from xs by removing the first item from xs that is equal (==) to x.
• removeAll(x, xs): Returns a list that results from xs by removing all items from xs that are equal (==) to x.
• filter(pred, xs): Returns a list that contains only those elements for which the one-argument function pred returns true.
• enum_list(start, end): Returns a list that enumerates numbers starting from start using a step size of 1, until the number exceeds (> end.
• list_ref(xs, n): Returns the element of list xs at position n, where the first element has index 0.
• 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 \( \text{op(1, op(2, op(3, zero)))} \).

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.

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:

\[
\begin{align*}
\text{(expression)} & \ ::= \ \text{id} \ (\text{expr-list}) \\
& \ | \ \ (\text{expression}) \ (\text{expr-list})
\end{align*}
\]
The following table specifies what arguments JediScript’s operators take and what results they return.

<table>
<thead>
<tr>
<th>operator</th>
<th>argument 1</th>
<th>argument 2</th>
<th>result</th>
</tr>
</thead>
<tbody>
<tr>
<td>+</td>
<td>number</td>
<td>number</td>
<td>number</td>
</tr>
<tr>
<td>+</td>
<td>string</td>
<td>any</td>
<td>string</td>
</tr>
<tr>
<td>+</td>
<td>any</td>
<td>string</td>
<td>string</td>
</tr>
<tr>
<td>-</td>
<td>number</td>
<td>number</td>
<td>number</td>
</tr>
<tr>
<td>*</td>
<td>number</td>
<td>number</td>
<td>number</td>
</tr>
<tr>
<td>/</td>
<td>number</td>
<td>number</td>
<td>number</td>
</tr>
<tr>
<td>%</td>
<td>number</td>
<td>number</td>
<td>number</td>
</tr>
<tr>
<td>!==</td>
<td>number</td>
<td>number</td>
<td>bool</td>
</tr>
<tr>
<td>!==</td>
<td>number</td>
<td>number</td>
<td>bool</td>
</tr>
<tr>
<td>&gt;</td>
<td>number</td>
<td>number</td>
<td>bool</td>
</tr>
<tr>
<td>&lt;</td>
<td>number</td>
<td>number</td>
<td>bool</td>
</tr>
<tr>
<td>&gt;=</td>
<td>number</td>
<td>number</td>
<td>bool</td>
</tr>
<tr>
<td>&lt;=</td>
<td>number</td>
<td>number</td>
<td>bool</td>
</tr>
<tr>
<td>&amp;&amp;</td>
<td>bool</td>
<td>bool</td>
<td>bool</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>bool</td>
</tr>
<tr>
<td>!</td>
<td>bool</td>
<td></td>
<td>bool</td>
</tr>
<tr>
<td>-</td>
<td>number</td>
<td></td>
<td>number</td>
</tr>
</tbody>
</table>

Following `if` and preceding `?`, JediScript only allows boolean expressions.

**Comments**

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