LambdaJS quick reference

Marek Materzok

December 11, 2015

1 Syntax

$$
\begin{array}{rcl}
e & ::= & x \mid l \mid e\,(e,\ldots) \mid \mathbf{func}\,(x,\ldots)\,e \mid \mathit{un}\,e \mid e\;\mathit{bin}\,e \mid e\,;\,e \mid e\,;;\,e \mid \\
& & \mathbf{let}\,(x=e)\,e \mid \mathbf{rec}\,(x=e)\,e \mid \mathbf{if}\,(e)\,e\,\mathbf{else}\,e \mid \\
& & \mathbf{label}\;i : e \mid \mathbf{break}\;i\;e \mid \mathbf{throw}\;e \mid \mathbf{try}\;e\;\mathbf{catch}\,(x)\,e \mid \mathbf{try}\;e\;\mathbf{finally}\;e \mid \\
& & e\,[e\langle \mathit{pa}\rangle] \mid e\,[e\langle \mathit{pa}\rangle = e] \mid e\,[\mathbf{delete}\;e] \mid e\,[\langle \mathit{oa}\rangle] \mid e\,[\langle \mathit{oa}\rangle = e] \mid \\
& & \{[\mathit{oa} : e,\ldots]\; s : \mathit{pe},\ldots\} \\
\mathit{pe} & ::= & \{\mathbf{value} : e, \mathbf{writable} : e, \mathbf{enumerable} : e, \mathbf{configurable} : e\} \mid \\
& & \{\mathbf{getter} : e, \mathbf{setter} : e, \mathbf{enumerable} : e, \mathbf{configurable} : e\} \\
l & ::= & b \mid n \mid k \mid s \mid \mathbf{undef} \mid \mathbf{null} \mid \mathbf{empty} \\
b & ::= & \mathbf{true} \mid \mathbf{false} \\
n & ::= & \textit{IEEE floating-point numbers} \\
s & ::= & \textit{UTF-16 encoded strings} \\
k & ::= & \textit{32-bit integers} \\
\mathit{un} & ::= & \mathbf{typeof} \mid \mathbf{strlen} \mid \mathbf{is\text{-}primitive} \mid \mathbf{is\text{-}closure} \mid \mathbf{is\text{-}object} \mid \\
& & \mathbf{to\text{-}string} \mid \mathbf{to\text{-}number} \mid \mathbf{to\text{-}boolean} \mid \mathbf{to\text{-}int} \mid \mathbf{ntoc} \mid \mathbf{cton} \mid \\
& & ! \mid \sim \mid - \mid \mathbf{abs} \mid \mathbf{floor} \mid \mathbf{ceil} \mid \ldots \\
\mathit{bin} & ::= & + \mid - \mid * \mid / \mid \% \mid < \mid == \mid === \mid +_s \mid <_s \mid \& \mid \texttt{|} \mid \hat{}\, \mid << \mid >> \mid >>> \mid \\
& & \mathbf{has\text{-}own\text{-}property} \mid \mathbf{has\text{-}internal} \mid \mathbf{is\text{-}accessor} \mid \mathbf{char\text{-}at} \mid \ldots \\
\mathit{pa} & ::= & \mathbf{value} \mid \mathbf{writable} \mid \mathbf{getter} \mid \mathbf{setter} \mid \mathbf{enumerable} \mid \mathbf{configurable} \\
\mathit{oa} & ::= & \mathbf{proto} \mid \mathbf{class} \mid \mathbf{extensible} \mid \mathbf{code} \mid i
\end{array}
$$

2 Semantics

Presented in big-step style; the formalized pretty-big-step semantics in Coq can be obtained from it. Abort-handling rules (for throws and breaks) are not presented.

2.1 Additional syntax

$$
\begin{array}{rcl}
v & ::= & l \mid (\Delta; x,\ldots; e) \mid \mathit{ptr} \\
\mathit{ptr} & ::= & \textit{heap pointers} \\
r & ::= & v \mid \mathbf{throw}\;v \mid \mathbf{break}\;i\;v \\
o & ::= & \{[\mathbf{proto} : v, \mathbf{class} : s, \mathbf{extensible} : b, \mathbf{code} : v, \ldots]\; s : p, \ldots\} \\
p & ::= & \{\mathbf{value} : v, \mathbf{writable} : v, \mathbf{enumerable} : b, \mathbf{configurable} : b\} \mid \\
& & \{\mathbf{getter} : v, \mathbf{setter} : v, \mathbf{enumerable} : b, \mathbf{configurable} : b\}
\end{array}
$$

2.2 Helper functions and predicates

2.2.1 Handling empty results

The following is used in the semantic rules for the double semicolon:

$$
\begin{array}{rcll}
v + \mathbf{empty} & = & v & \\
v + v' & = & v' & v' \neq \mathbf{empty} \\
v + \mathbf{throw}\;v' & = & \mathbf{throw}\;v & \\
v + \mathbf{break}\;i\;v' & = & \mathbf{break}\;i\;(v + v') &
\end{array}
$$

2.2.2 Control flow breaking results

$$\frac{}{\text{abort}(\textbf{throw}\ v)} \qquad \frac{}{\text{abort}(\textbf{break}\ i\ v)}$$

2.2.3 Restrictions on object and property attributes

Used in semantic rules for setting attributes.

$$\frac{oa \in \{\textbf{proto}, \textbf{extensible}\}}{oa \text{ writable}} \qquad \frac{}{pa \text{ writable in } \{\textbf{configurable} : \textbf{true}, \dots\}}$$

$$\frac{pa \in \{\textbf{value}, \textbf{writable}\}}{pa \text{ writable in } \{\textbf{writable} : \textbf{true}, \dots\}} \qquad \frac{}{\textbf{null} \text{ is valid } \textbf{proto}} \qquad \frac{}{ptr \text{ is valid } \textbf{proto}}$$

$$\frac{}{b \text{ is valid } \textbf{extensible}} \qquad \frac{}{s \text{ is valid } \textbf{class}} \qquad \frac{}{\textbf{undef} \text{ is valid } \textbf{code}} \qquad \frac{}{(\Delta; x, \dots; e) \text{ is valid } \textbf{code}}$$

$$\frac{}{v \text{ is valid } i} \qquad \frac{}{v \text{ is valid } \textbf{value}} \qquad \frac{}{v \text{ is valid } \textbf{getter}} \qquad \frac{}{v \text{ is valid } \textbf{setter}} \qquad \frac{}{b \text{ is valid } \textbf{writable}}$$

$$\frac{}{b \text{ is valid } \textbf{enumerable}} \qquad \frac{}{b \text{ is valid } \textbf{configurable}}$$

2.2.4 Property attribute writing

Note: changing the attribute type does not occur often (probably only in DefineOwnProperty). I consider removing this from the semantics and adding a simpler primitive instead.

$$\begin{array}{rcll}
\{pa : v, \dots\}(pa = v') & = & \{pa : v', \dots\} & \\
\left\{\begin{array}{l}\textbf{value} : v_1 \\ \textbf{writable} : b_1 \\ \textbf{enumerable} : b_2 \\ \textbf{configurable} : b_3\end{array}\right\}(pa = v) & = & \left\{\begin{array}{l}\textbf{getter} : \textbf{undef} \\ \textbf{setter} : \textbf{undef} \\ \textbf{enumerable} : b_2 \\ \textbf{configurable} : b_3\end{array}\right\}(pa = v) & \begin{array}{l} pa \text{ is one of:} \\ \textbf{getter} \\ \textbf{setter}\end{array} \\
\left\{\begin{array}{l}\textbf{getter} : v_1 \\ \textbf{setter} : v_2 \\ \textbf{enumerable} : b_1 \\ \textbf{configurable} : b_2\end{array}\right\}(pa = v) & = & \left\{\begin{array}{l}\textbf{value} : \textbf{undef} \\ \textbf{writable} : \textbf{true} \\ \textbf{enumerable} : b_1 \\ \textbf{configurable} : b_2\end{array}\right\}(pa = v) & \begin{array}{l} pa \text{ is one of:} \\ \textbf{value} \\ \textbf{writable}\end{array}
\end{array}$$

2.3 Semantic rules

2.3.1 Variables and functions

$$\frac{}{\Delta; \Phi; l \Downarrow \Phi; l} \qquad \frac{}{\Delta; \Phi; x \Downarrow \Phi; \Delta(x)} \qquad \frac{}{\Delta; \Phi; \textbf{func}\,(x_1, \dots, x_n)\,e \Downarrow \Phi; (\Delta; x_1, \dots, x_n; e)}$$

$$\frac{\Delta; \Phi; e \Downarrow \Phi_0; (\Delta'; x_1, \dots, x_n; e') \qquad \forall i, \Delta; \Phi_{i-1}; e_i \Downarrow \Phi_i; v_i \qquad \Delta'(x_1, \dots, x_n = v_1, \dots, v_n); \Phi_n; e' \Downarrow \Phi'; r}{\Delta; \Phi; e\,(e_1, \dots, e_n) \Downarrow \Phi'; r}$$

$$\frac{\Delta; \Phi; e_1 \Downarrow \Phi'; v \qquad \Delta(x = v); \Phi'; e_2 \Downarrow \Phi''; r}{\Delta; \Phi; \textbf{let}\,(x = e_1)\,e_2 \Downarrow \Phi''; r} \qquad \frac{\Delta' = \Delta(x = (\Delta'; x_1, \dots, x_n; e)) \qquad \Delta'; \Phi; e' \Downarrow \Phi'; r}{\Delta; \Phi; \textbf{rec}\,(x = \textbf{func}\,(x_1, \dots, x_n)\,e)\,e' \Downarrow \Phi'; r}$$

2.3.2 Operators

$$\frac{\Delta; \Phi; e \Downarrow \Phi'; v \qquad un(\Phi', v) = v'}{\Delta; \Phi; un\,e \Downarrow \Phi'; v'} \qquad \frac{\Delta; \Phi; e_1 \Downarrow \Phi'; v_1 \qquad \Delta; \Phi'; e_1 \Downarrow \Phi''; v_2 \qquad bin(\Phi'', v_1, v_2) = v}{\Delta; \Phi; e_1\,bin\,e_2 \Downarrow \Phi''; v}$$

$$\frac{\Delta; \Phi; e_1 \Downarrow \Phi'; v \qquad \Delta; \Phi'; e_2 \Downarrow \Phi''; r}{\Delta; \Phi; e_1\ ;\ e_2 \Downarrow \Phi''; r} \qquad \frac{\Delta; \Phi; e_1 \Downarrow \Phi'; v \qquad \Delta; \Phi'; e_2 \Downarrow \Phi''; r \qquad r' = v + r}{\Delta; \Phi; e_1\ ;;\ e_2 \Downarrow \Phi''; r'}$$

2.3.3 Control flow

$$\frac{\Delta;\Phi;e\Downarrow\Phi';\mathbf{true}\qquad\Delta;\Phi';e_1\Downarrow\Phi'';r}{\Delta;\Phi;\mathbf{if}\,(e)\,e_1\,\mathbf{else}\,e_2\Downarrow\Phi'';r}\qquad\frac{\Delta;\Phi;e\Downarrow\Phi';\mathbf{false}\qquad\Delta;\Phi';e_2\Downarrow\Phi'';r}{\Delta;\Phi;\mathbf{if}\,(e)\,e_1\,\mathbf{else}\,e_2\Downarrow\Phi'';r}$$

$$\frac{\Delta;\Phi;e\Downarrow\Phi';\mathbf{break}\ i\ v}{\Delta;\Phi;\mathbf{label}\ i:e\Downarrow\Phi';v}\qquad\frac{\Delta;\Phi;e\Downarrow\Phi';r\qquad\forall v,r\neq\mathbf{break}\ i\ v}{\Delta;\Phi;\mathbf{label}\ i:e\Downarrow\Phi';r}\qquad\frac{\Delta;\Phi;e\Downarrow\Phi';v}{\Delta;\Phi;\mathbf{break}\ i\ e\Downarrow\Phi';\mathbf{break}\ i\ v}$$

$$\frac{\Delta;\Phi;e\Downarrow\Phi';v}{\Delta;\Phi;\mathbf{throw}\ e\Downarrow\Phi';\mathbf{throw}\ v}\qquad\frac{\Delta;\Phi;e_1\Downarrow\Phi';r\qquad\forall v,r\neq\mathbf{throw}\ v}{\Delta;\Phi;\mathbf{try}\ e_1\ \mathbf{catch}\,(x)\,e_2\Downarrow\Phi';r}$$

$$\frac{\Delta;\Phi;e_1\Downarrow\Phi';r\qquad\Delta;\Phi';e_2\Downarrow\Phi'';v}{\Delta;\Phi;\mathbf{try}\ e_1\ \mathbf{finally}\ e_2\Downarrow\Phi'';r}\qquad\frac{\Delta;\Phi;e_1\Downarrow\Phi';r\qquad\Delta;\Phi';e_2\Downarrow\Phi'';r'\qquad\text{abort}(r')}{\Delta;\Phi;\mathbf{try}\ e_1\ \mathbf{finally}\ e_2\Downarrow\Phi'';r'}$$

2.3.4 Property access

$$\frac{\Delta;\Phi;e_1\Downarrow\Phi';ptr\qquad\Delta;\Phi';e_2\Downarrow\Phi'';s\qquad\Phi''(ptr)=\{[\dots]\,s:\{pa:v\},\dots\}}{\Delta;\Phi;e_1\,[e_2\langle pa\rangle]\Downarrow v;\Phi''}$$

$$\frac{\begin{array}{c}\Delta;\Phi;e_1\Downarrow\Phi_1;ptr\\ \Delta;\Phi_1;e_2\Downarrow\Phi_2;s\qquad\Delta;\Phi_2;e_3\Downarrow\Phi_3;v\qquad\Phi_3(ptr)=o\qquad o=\{[\dots]\dots\}\qquad s\text{ not a property in }o\\ o\text{ extensible}\qquad v\text{ is valid }pa\qquad\Phi'=\Phi_3(ptr=\{[\dots]\,s:\text{defaultprop}(pa=v),\dots\})\end{array}}{\Delta;\Phi;e_1\,[e_2\langle pa\rangle=e_3]\Downarrow\Phi';v}$$

$$\frac{\begin{array}{c}\Delta;\Phi;e_1\Downarrow\Phi_1;ptr\qquad\Delta;\Phi_1;e_2\Downarrow\Phi_2;s\qquad\Delta;\Phi_2;e_3\Downarrow\Phi_3;v\qquad\Phi_3(ptr)=o\\ o=\{[\dots]\,s:p,\dots\}\qquad v\text{ is valid }pa\qquad pa\text{ writable in }p\qquad\Phi'=\Phi_3(ptr=\{[\dots]\,s:p(pa=v),\dots\})\end{array}}{\Delta;\Phi;e_1\,[e_2\langle pa\rangle=e_3]\Downarrow\Phi';v}$$

$$\frac{\begin{array}{c}\Delta;\Phi;e_1\Downarrow\Phi_1;ptr\qquad\Delta;\Phi_1;e_2\Downarrow\Phi_2;s\\ \Phi_2(ptr)=\{[\dots]\,s:\{\mathbf{configurable}:\mathbf{true},\dots\},\dots\}\qquad\Phi'=\Phi_2(ptr=\{[\dots]\dots\})\end{array}}{\Delta;\Phi;e_1\,[\mathbf{delete}\ e_2]\Downarrow v;\Phi'}$$

2.3.5 Object attributes

$$\frac{\Delta;\Phi;e\Downarrow ptr;\Phi'\qquad\Phi'(ptr)=\{[oa:v,\dots]\dots\}}{\Delta;\Phi;e\,[\langle oa\rangle]\Downarrow v;\Phi'}$$

$$\frac{\begin{array}{c}\Delta;\Phi;e_1\Downarrow\Phi_1;ptr\qquad\Delta;\Phi_1;e_2\Downarrow\Phi_2;v\qquad\Phi_2(ptr)=o\qquad o=\{[oa:v',\dots]\dots\}\\ v\text{ is valid }oa\qquad o\text{ extensible}\qquad oa\text{ writable}\qquad\Phi'=\Phi_2(ptr=\{[oa:v,\dots]\dots\})\end{array}}{\Delta;\Phi;e_1\,[\langle oa\rangle=e_2]\Downarrow\Phi';v}$$

$$\frac{\forall i,\Delta;\Phi_{i-1};e_i\Downarrow\Phi_i;v_i\qquad v_3\text{ and }v_4\text{ are bools}}{\begin{array}{c}\Delta;\Phi_0;\{\mathbf{getter}:e_1,\mathbf{setter}:e_2,\mathbf{enumerable}:e_3,\mathbf{configurable}:e_4\}\Downarrow\\ \Phi_4;\{\mathbf{getter}:v_1,\mathbf{setter}:v_2,\mathbf{enumerable}:v_3,\mathbf{configurable}:v_4\}\end{array}}$$

$$\frac{\forall i,\Delta;\Phi_{i-1};e_i\Downarrow\Phi_i;v_i\qquad v_2,\,v_3\text{ and }v_4\text{ are bools}}{\begin{array}{c}\Delta;\Phi_0;\{\mathbf{value}:e_1,\mathbf{writable}:e_2,\mathbf{enumerable}:e_3,\mathbf{configurable}:e_4\}\Downarrow\\ \Phi_4;\{\mathbf{value}:v_1,\mathbf{writable}:v_2,\mathbf{enumerable}:v_3,\mathbf{configurable}:v_4\}\end{array}}$$

2.3.6 Object creation

$$\frac{\begin{array}{c}\forall i,\Delta;\Phi_{i-1};e_i\Downarrow\Phi_i;v_1\qquad\forall i,\Delta;\Phi_{n+i-1};pe_i\Downarrow\Phi_{n+i};p_i\qquad oa_1,\dots,oa_n\text{ distinct}\qquad s1,\dots,s_m\text{ distinct}\\ \{\mathbf{proto},\mathbf{class},\mathbf{extensible},\mathbf{code}\}\subseteq\{oa_i:i\in\{1,\dots,n\}\}\qquad\forall i,v_i\text{ is valid }oa_i\qquad ptr\notin\Phi_{n+m}\end{array}}{\begin{array}{c}\Delta;\Phi;\{[oa_1:e_1,\dots,oa_n:e_n]\,s_1:pe_1,\dots,s_m:pe_m\}\Downarrow\\ \Phi_{n+m}(ptr=\{[oa_1:v_1,\dots,oa_n:v_n]\,s_1:p_1,\dots,s_m:p_m\});ptr\end{array}}$$

2.4 Operators

Please note that operators are partial functions: they can be undefined for some inputs. Operators can access the heap (for object-related operators) but cannot modify it. The "otherwise" clauses in the definitions are used when no other clause can be used.

2.4.1 typeof

Consistent with JS typeof on JS primitives (ES5 11.4.3).

$$\begin{array}{rcl}
\textbf{typeof}(\Phi, b) & = & \texttt{"boolean"} \\
\textbf{typeof}(\Phi, n) & = & \texttt{"number"} \\
\textbf{typeof}(\Phi, k) & = & \texttt{"int"} \\
\textbf{typeof}(\Phi, s) & = & \texttt{"string"} \\
\textbf{typeof}(\Phi, \textbf{undef}) & = & \texttt{"undefined"} \\
\textbf{typeof}(\Phi, \textbf{null}) & = & \texttt{"null"} \\
\textbf{typeof}(\Phi, \textbf{empty}) & = & \texttt{"empty"} \\
\textbf{typeof}(\Phi, (\Delta; x, \ldots; e)) & = & \texttt{"function"} \\
\textbf{typeof}(\Phi, ptr) & = & \texttt{"object"}
\end{array}$$

2.4.2 type testing operators

These are redundant (they can be implemented with **typeof**), but are used often and so it is useful to have them.

$$\begin{array}{rcl}
\textbf{is-primitive}(\Phi, v) & = & \textbf{true} \text{ if } v \text{ is } b, n, s, \textbf{undef}, \textbf{null} \\
\textbf{is-primitive}(\Phi, v) & = & \textbf{false} \text{ otherwise} \\
\textbf{is-closure}(\Phi, (\Delta; x, \ldots; e)) & = & \textbf{true} \\
\textbf{is-closure}(\Phi, v) & = & \textbf{false} \text{ otherwise} \\
\textbf{is-object}(\Phi, ptr) & = & \textbf{true} \\
\textbf{is-object}(\Phi, v) & = & \textbf{false} \text{ otherwise}
\end{array}$$

2.4.3 string operators

$$\begin{array}{rcl}
\textbf{strlen}(\Phi, s) & = & \textit{length of } s \\
\textbf{ntoc}(\Phi, k) & = & \textit{one character string with code } k \\
\textbf{cton}(\Phi, \texttt{"c"}) & = & \textit{character code of } c \\
\textbf{char-at}(\Phi, s, k) & = & k\textit{th character of } s \\
+_s(\Phi, s_1, s_2) & = & s_1 \textit{ appended to } s_2 \\
<_s (\Phi, s_1, s_2) & = & \textbf{true} \text{ if } s_1 \text{ precedes } s_2 \text{ lexicographically} \\
<_s (\Phi, s_1, s_2) & = & \textbf{false} \text{ otherwise}
\end{array}$$

2.4.4 number operators

Number operators work as specified in IEEE 754.

$$\begin{array}{rcl}
-(\Phi, n) & = & -n \\
\textbf{abs}(\Phi, n) & = & \text{abs } n \\
\textbf{floor}(\Phi, n) & = & \text{floor } n \\
\textbf{ceil}(\Phi, n) & = & \text{ceil } n \\
+(\Phi, n_1, n_2) & = & n_1 + n_2 \\
-(\Phi, n_1, n_2) & = & n_1 - n_2 \\
*(\Phi, n_1, n_2) & = & n_1 n_2 \\
/(\Phi, n_1, n_2) & = & n_1 / n_2 \\
\%(\Phi, n_1, n_2) & = & n_1 \text{ mod } n_2 \\
< (\Phi, n_1, n_2) & = & \textbf{true} \text{ if } n_1 < n_2 \text{ (IEEE 754)} \\
< (\Phi, n_1, n_2) & = & \textbf{false} \text{ otherwise}
\end{array}$$

2.4.5 integer operators

Bit shifts are defined as in the ECMA specification (ES5 11.7).

$$
\begin{aligned}
\sim(\Phi, k) &= \sim k \\
\&(\Phi, k_1, k_2) &= k_1 \& k_2 \\
|(\Phi, k_1, k_2) &= k_1 \,|\, k_2 \\
\hat{}(\Phi, k_1, k_2) &= k_1 \hat{} k_2 \\
<<(\Phi, k_1, k_2) &= k_1 << k_2 \\
>>(\Phi, k_1, k_2) &= k_1 >> k_2 \\
>>>(\Phi, k_1, k_2) &= k_1 >>> k_2
\end{aligned}
$$

2.4.6 to-string

Consistent with ToString on JS primitives (ES5 9.8).

$$
\begin{aligned}
\textbf{to-string}(\Phi, \textbf{true}) &= \texttt{"true"} \\
\textbf{to-string}(\Phi, \textbf{false}) &= \texttt{"false"} \\
\textbf{to-string}(\Phi, n) &= \textit{string representation of } n \\
\textbf{to-string}(\Phi, k) &= \textit{string representation of } k \\
\textbf{to-string}(\Phi, s) &= s \\
\textbf{to-string}(\Phi, \textbf{undef}) &= \texttt{"undefined"} \\
\textbf{to-string}(\Phi, \textbf{null}) &= \texttt{"null"} \\
\textbf{to-string}(\Phi, \textbf{empty}) &= \texttt{"empty"} \\
\textbf{to-string}(\Phi, (\Delta; x, \ldots; e)) &= \texttt{"closure"} \\
\textbf{to-string}(\Phi, ptr) &= \texttt{"object"}
\end{aligned}
$$

2.4.7 to-number

Consistent with ToNumber on JS primitives (ES5 9.3).

$$
\begin{aligned}
\textbf{to-number}(\Phi, \textbf{true}) &= 1 \\
\textbf{to-number}(\Phi, \textbf{false}) &= 0 \\
\textbf{to-number}(\Phi, n) &= n \\
\textbf{to-number}(\Phi, k) &= k \textit{ as floating point} \\
\textbf{to-number}(\Phi, s) &= s \textit{ parsed as number} \\
\textbf{to-number}(\Phi, \textbf{undef}) &= \text{NaN} \\
\textbf{to-number}(\Phi, \textbf{null}) &= 0 \\
\textbf{to-number}(\Phi, \textbf{empty}) &= \text{NaN} \\
\textbf{to-number}(\Phi, (\Delta; x, \ldots; e)) &= \text{NaN} \\
\textbf{to-number}(\Phi, ptr) &= \text{NaN}
\end{aligned}
$$

2.4.8 to-int

Floating point to integer conversion is carried out as in ToInt32/ToUint32 (ES5 9.5, 9.6).

$$
\begin{aligned}
\textbf{to-number}(\Phi, \textbf{true}) &= 1 \\
\textbf{to-number}(\Phi, \textbf{false}) &= 0 \\
\textbf{to-number}(\Phi, n) &= n \textit{ converted to an integer} \\
\textbf{to-number}(\Phi, k) &= k \\
\textbf{to-number}(\Phi, s) &= s \textit{ parsed as number} \\
\textbf{to-number}(\Phi, \textbf{undef}) &= 0 \\
\textbf{to-number}(\Phi, \textbf{null}) &= 0 \\
\textbf{to-number}(\Phi, \textbf{empty}) &= 0 \\
\textbf{to-number}(\Phi, (\Delta; x, \ldots; e)) &= 0 \\
\textbf{to-number}(\Phi, ptr) &= 0
\end{aligned}
$$

2.4.9 to-boolean

Consistent with ToBoolean on JS primitives (ES5 9.2).

$$
\begin{aligned}
\textbf{to-boolean}(\Phi, b) &= b\\
\textbf{to-boolean}(\Phi, n) &= \textbf{false} \text{ if } n \text{ is } 0, -0, \mathrm{NaN}\\
\textbf{to-boolean}(\Phi, n) &= \textbf{true} \text{ otherwise}\\
\textbf{to-boolean}(\Phi, 0) &= \textbf{false}\\
\textbf{to-boolean}(\Phi, k) &= \textbf{true} \text{ otherwise}\\
\textbf{to-boolean}(\Phi, \texttt{""}) &= \textbf{false}\\
\textbf{to-boolean}(\Phi, s) &= \textbf{true} \text{ otherwise}\\
\textbf{to-boolean}(\Phi, \textbf{undef}) &= \textbf{false}\\
\textbf{to-boolean}(\Phi, \textbf{null}) &= \textbf{false}\\
\textbf{to-boolean}(\Phi, \textbf{empty}) &= \textbf{false}\\
\textbf{to-boolean}(\Phi, (\Delta; x, \ldots; e)) &= \textbf{true}\\
\textbf{to-boolean}(\Phi, \mathit{ptr}) &= \textbf{true}
\end{aligned}
$$

2.4.10 strict equality

Consistent with JS strict equality on JS primitives (ES5 11.9.6).

$$
\begin{aligned}
== (\Phi, b, b) &= \textbf{true}\\
== (\Phi, n_1, n_2) &= \textbf{true} \text{ if } n_1 = n_2 \text{ (IEEE 754)}\\
== (\Phi, k, k) &= \textbf{true}\\
== (\Phi, s, s) &= \textbf{true}\\
== (\Phi, \textbf{undef}, \textbf{undef}) &= \textbf{true}\\
== (\Phi, \textbf{null}, \textbf{null}) &= \textbf{true}\\
== (\Phi, \textbf{empty}, \textbf{empty}) &= \textbf{true}\\
== (\Phi, \mathit{ptr}, \mathit{ptr}) &= \textbf{true}\\
== (\Phi, v_1, v_2) &= \textbf{false} \text{ otherwise}
\end{aligned}
$$

2.4.11 same value

Consistent with JS SameValue algorithm on JS primitives (ES5 9.12).

$$
\begin{aligned}
=== (\Phi, l, l) &= \textbf{true}\\
=== (\Phi, \mathit{ptr}, \mathit{ptr}) &= \textbf{true}\\
=== (\Phi, v_1, v_2) &= \textbf{false} \text{ otherwise}
\end{aligned}
$$

2.4.12 has-own-property

$$
\begin{aligned}
\textbf{has-own-property}(\Phi(\mathit{ptr} = \{[\ldots]\, s : p, \ldots\}), \mathit{ptr}, s) &= \textbf{true}\\
\textbf{has-own-property}(\Phi(\mathit{ptr} = o), \mathit{ptr}, s) &= \textbf{false} \text{ otherwise}
\end{aligned}
$$

2.4.13 has-internal

$$
\begin{aligned}
\textbf{has-internal}(\Phi(\mathit{ptr} = \{[i : v, \ldots]\, \ldots\}), \mathit{ptr}, \texttt{"}i\texttt{"}) &= \textbf{true}\\
\textbf{has-internal}(\Phi(\mathit{ptr} = o), \mathit{ptr}, s) &= \textbf{false} \text{ otherwise}
\end{aligned}
$$

2.4.14 is-accessor

$$
\begin{aligned}
\textbf{is-accessor}(\Phi(\mathit{ptr} = \{[\ldots]\, s : \{\textbf{getter} : v, \ldots\}, \ldots\}), \mathit{ptr}, s) &= \textbf{true}\\
\textbf{is-accessor}(\Phi(\mathit{ptr} = \{[\ldots]\, s : p, \ldots\}), \mathit{ptr}, s) &= \textbf{false} \text{ otherwise}
\end{aligned}
$$