```javascript
// Conversion Functions

// toNumber

const toNumber = (n) => n;
const toNumber_true = () => 1;
const toNumber_false = () => 0;
const toNumber_undefined = () => 0;

// toBoolean

const toBoolean = (n) => n === 0 || n === NaN ? false : true;
const toBoolean_true = (b) => b;
const toBoolean_undefined = () => false;
```

**Figure 2.3.** JavaScript conversion functions.

### 2.2. Small-Step Operational Semantics

#### 2.2.1. Evaluation Order

In Section 2.1.2, we have carefully specified several aspects of how the expression

\[ e_1 + e_2 \]

should be evaluated. In essence, it adds two integers that result from evaluating \( e_1 \) and \( e_2 \). However, there is still at least one more semantic question that we have not specified, “Is \( e_1 \) evaluated first and then \( e_2 \) or vice versa, or are they evaluated concurrently?”

Why does this question matter? Consider the expression:

\[(2.2.1.1)\quad (\text{jsy.print}(1), \ 1) + (\text{jsy.print}(2), \ 2).\]

The `,` operator is a sequencing operator. In particular, \( e_1 , e_2 \) first evaluates \( e_1 \) to a value and then evaluates \( e_2 \) to value; the value of the whole expression is the value of \( e_2 \), while the value of \( e_1 \) is simply thrown away. Furthermore, `jsy.print(e1)` evaluates its argument to a value and then prints to the screen a representation of that value. If the left operand of `+` is evaluated first before the right operand, then the above expression (2.2.1.1) prints 1 and then 2. If the operands of `+` are evaluated in the opposite order, then 2 is printed first followed by 1. Note that the final value is 3 regardless of the evaluation order.

The evaluation order matters because the `jsy.print(e1)` expression has a *side effect*. It prints to the screen. As alluded to in Section 1.2, an expression free of side effects (i.e., is pure) has the advantage that the evaluation order cannot be observed (i.e., does not matter from the programmer's perspective). Having this property is also known as being *referentially transparent*, that is, taking an expression and replacing any of its subexpressions by the subexpression’s value cannot be observed as evaluating any differently than evaluating the expression itself. In JavaScript, our only side-effecting expression is `jsy.print(e1)`. If we remove the prints
2.2. SMALL-STEP OPERATIONAL SEMANTICS

from the above expression (2.2.1.1), then the evaluation order cannot be observed.

2.2.2. A Small-Step Operational Semantics of JAVA SCRIPTY. The big-step operational semantics given in Section 2.1.2 does give us a nice specification for implementing an interpreter, but it does leave some semantic choices like evaluation order implicit. Intuitively, it specifies what the value of an expression should be (if it exists) but not precisely the steps to get to the value.

We have already used a notation for describing a one-step evaluation relation:

\[ e \rightarrow e' \]

This notation is a judgment form stating informally, “Expression \( e \) can take one step of evaluation to expression \( e' \).” Defining this judgment allows us to more precisely state how to take one step of evaluation, that is, how to make a single reduction step. Once we know how to reduce expressions, we can evaluate an expression \( e \) by repeatedly applying reduction until reaching a value. Thus, such a definition describes an operational semantics and intuitively an interpreter for expressions \( e \). This style of operational semantics where we specify reduction steps is called a small-step operational semantics.

In contrast to Section 2.1.2, we will not extend this judgment form with value environments. Instead, we define the one-step reduction relation on closed expressions, that is, expressions without any free variables. If we require the “top-level” program to be a closed expression, then we can ensure reduction only sees closed expressions by intuitively “applying the environment” eagerly via substitution. That is, variable uses are replaced by the values to which they are bound before reduction gets to them. As an example, we will define reduction so that the following judgment holds:

\[
\text{const } x = 42; \quad x + x \rightarrow 1 + 1
\]

This choice to use substitution instead of explicit environments is orthogonal to specifying the semantics using small-step or big-step (i.e., one could use substitution with big-step or environments with small-step). Explicit environments just get a bit more unwieldy here.

First, we need to describe what action does an operation perform. For example, we want to say that the + operator adds to numbers, which we say with the following rule:

\[
\text{DoPlus}\[
\frac{n' = \text{toNumber}(v_1) + \text{toNumber}(v_2)}{v_1 + v_2 \rightarrow n'}
\]
This rule says the expression $v_1 + v_2$ reduces in one step to an integer value $n'$ that is the addition of the toNumber conversion of values $v_1$ and $v_2$. We use the meta-variables $v_1, v_2,$ and $n'$ to express constraints that particular positions in the expressions must be values or numeric values. Note that the $+$ in the conclusion is the syntactic operator, while the $+$ in the premise expresses mathematical addition of two numbers. As we discussed in Section 2.1.2, this symbol clash is rather unfortunate, but context usually allows us to determine which $+$ is which. We sometimes call this kind of rule that performs an operation a local reduction rule. We will prefix all rules for this kind of rule with DO (and so will sometimes call them Do rules).

Second, we need to describe how we find the next operation to perform. These rules will capture issues like evaluation order described informally in Section 2.2.1. To specify that $e_1 + e_2$ should be evaluated left-to-right, we use the following two rules:

$$
\frac{\text{SearchPlus}_1}{e_1 \rightarrow e'_1} \quad \frac{\text{SearchPlus}_2}{e_2 \rightarrow e'_2} \\
\frac{e_1 + e_2 \rightarrow e'_1 + e_2}{v_1 + e_2 \rightarrow v_1 + e'_2}
$$

The SearchPlus$_1$ rule states for an arbitrary expression of the form $e_1 + e_2$, if $e_1$ steps to $e'_1$, then the whole expression steps to $e'_1 + e_2$. We can view this rule as saying that we should look for an operation to perform somewhere in $e_1$. The rest of the expression $\cdot + e_2$ is a context that gets carried over untouched. The SearchPlus$_2$ rule is similar except that it applies only if the left expression is a value (i.e., $v_1 + e_2$). Together, these rules capture precisely a left-to-right evaluation order for an expression of the form $e_1 + e_2$ because (1) if $e_1$ is not a value, then only SearchPlus$_1$ could possibly apply, and (2) if $e_1$ is a value, then only SearchPlus$_2$ could possibly apply. We sometimes call this kind of rule that finds the next operation to perform a global reduction rule (or a Search rule).

Considering these three rules, there is at most one rule that applies that specifies the “next” step. If our set of inference rules defining reduction has this property, then we say that our reduction system is deterministic. In other words, there is always at most one “next” step. Determinism is a property that we could prove about certain reduction systems, which we can state formally as follows:

**Property 2.1 (Determinism).** If $e \rightarrow e'$ and $e \rightarrow e''$, then $e' = e''$. 

In general, such a proof would proceed by structural induction on the derivation of the reduction step (i.e., $e \rightarrow e'$). We do not yet discuss such proofs in detail (cf., Section 1.5.3).
In Figure 2.5, we give all of the inference rules that define the one-step evaluation relation $e \rightarrow e'$ for JavaScript. The DoNEG states that the unary operator $-$ is integer negation, while DoNOT states that $!$ is boolean negation. Observe that to "do" the operation, we require that the sub-expression under the unary operators $-$ or $!$, respectively, is a value. Contrast these rules to EVALNEG and EVALNOT in Figure 2.2. If the sub-expression under the unary operator is not a value, then instead the rule SEARCHUNARY applies telling us to look for something to reduce inside this sub-expression. The DoSEQ rules states that $,$ is used to indicate
sequencing: for \( e_1, e_2 \), first \( e_1 \) is evaluated to a value, then that value is ignored, and we continue by evaluating \( e_2 \). The \texttt{DoARITH}, \texttt{DoINEQUALITY}, and \texttt{DoEQUALITY} specify how the arithmetic, inequality, and equality operators behave, respectively. The \texttt{DoARITH} includes the case for \( + \) that we separated out in our discussion above.

We say that a short-circuit evaluation of expression is one where a value is produced before evaluating all subexpressions to values. The next four rules \texttt{DoANDTRUE}, \texttt{DoANDFALSE}, \texttt{DoORTRUE}, and \texttt{DoORFALSE} say that the boolean expressions \( e_1 \&\& e_2 \) and \( e_1 || e_2 \) may short-circuit. In particular, the rule

\[
\texttt{DoANDFALSE} \\
\texttt{false} = \text{toBoolean}(v_1) \\
\frac{v_1 \&\& e_2 }{\text{false}}
\]

says that \( v_1 \&\& e_2 \) where \( v_1 \) converts to \texttt{false} evaluates to \texttt{false} without ever evaluating \( e_2 \). The analogous rule for || is

\[
\texttt{DoORTrue} \\
\texttt{true} = \text{toBoolean}(v_1) \\
\frac{v_1 || e_2 }{\text{true}}
\]

The \texttt{DoPrint} rule

\[
\texttt{DoPrint} \\
\texttt{v_1 printed} \\
\frac{\text{jsy.print}(v_1)}{\text{undefined}}
\]

is somewhat informal. In particular, since printing is outside of our model, the“\( v_1 \) printed” in the premise of the rule is not any required condition but should be viewed as comment for when this rule is applied. What is stated is the result of a \texttt{print} is the value \texttt{undefined}.

For \( e_1 ? e_2 : e_3 \), the rules \texttt{DoIFTrue} and \texttt{DoIFFalse} specify with which expression to continue evaluation in the expected way depending on what Boolean value to which the guard converts.

The \texttt{DoConst} rule for the variable binding expression \texttt{const x = e_1 ; e_2}

\[
\texttt{DoConst} \\
\texttt{const x = v_1 ; e_2} \\
\frac{\text{e_2[v_1/x]}}{}
\]

is a bit more interesting. The expression-to-be-bound should already be a value \( v_1 \). We then proceed with \( e_2 \) with the value \( v_1 \) replacing the variable \( x \). In general, the notation \( e_1[x/e_2] \) is read as capture-avoiding substitution of expression \( e_2 \) for variable \( x \) in \( e_1 \). We describe substitution in more detail below in Section 2.2.2.1.
2.2. SMALL-STEP OPERATIONAL SEMANTICS 35

The remaining rules in Figure 2.4 describe how to find the next operation to perform (i.e., the global reduction rules). They specify that all expressions are evaluated left-to-right.

2.2.2.1. Substitution. The term capture-avoiding substitution means that we get the expression that is like $e_1$, but we have replaced all instances of variable $x$ with $e_2$ while carefully respecting static scoping (cf., Section 1.2.4). There are two thorny issues that arise.

**Shadowing:** The substitution

$$\left[ \begin{array}{l}
\text{const } a = 1; a + b
\end{array} \right]_{e_1} \left[ \begin{array}{l}
2
\end{array} \right]_{e_2} / \left[ \begin{array}{l}
x
\end{array} \right]$$

should yield (const $a = 1; a + b$). That is, only free instances of $a$ in $e_1$ should be replaced.

**Free Variable Capture:** The substitution

$$\left[ \begin{array}{l}
\text{const } a = 1; a + b
\end{array} \right]_{e_1} \left( \begin{array}{l}
(a + 2)
\end{array} \right)_{e_2} / \left[ \begin{array}{l}
x
\end{array} \right]$$

should yield something like (const $c = 1; c + (a + 2)$). In particular, the following result is wrong:

(const $a = 1; a + (a + 2)$)

because the free variable $a$ in $e_2$ gets “captured” by the const binding of $a$.

In both cases, the issues could be resolved by renaming all bound variables in $e_1$ so that there are no name conflicts with free variables in $e_2$ or $x$. In other words, it is clear what to do if $e_1$ were instead

const $c = 1; c + b$

in which case textual substitution would suffice.

The observation is that renaming bound variables should preserve the meaning of the expression, that is, the following two expressions are somehow equivalent:

(const $a = 1; a)$ $\equiv_a$ (const $b = 1; b$)

For historical reasons, this equivalence is known $\alpha$-equivalence, and the process of renaming bound variables is called $\alpha$-renaming. This observation also leads to coming up with an abstract syntax representation so that the above two expressions are represented with the same object. As an aside, one way to do this is to use variables in the meta language to represent variables in the object language. This idea is known as higher-order abstract syntax.

In DOCONST, our situation is slight more restricted than the general case discussed above. In particular, the substitution is of the form $e[v/x]$
where the replacement for $x$ has to be value. Values have no free variables, so only the shadowing issue arises.

2.2.2. Multi-Step Evaluation. We have now defined how to take one-step of evaluation. The multi-step evaluation judgment

$$e \rightarrow^* e'$$

says, “Expression $e$ can step to expression $e'$ in zero-or-more steps.” This judgment is defined using the following two rules:

\[
\begin{align*}
\text{ZERO STEPS} & : & e & \rightarrow^* e \\
\text{AT LEAST ONE STEP} & : & e \rightarrow e' & e' \rightarrow^* e'' & \Rightarrow & e \rightarrow^* e''
\end{align*}
\]

In other words, $\rightarrow^*$ is the reflexive-transitive closure of $\rightarrow$.

A property that we want is that our big-step semantics and our small-step semantics are “the same.” We can state this property formally as follows.

**Property 2.2 (Big-Step and Small-Step Equivalence).** $\vdash e \Downarrow v$ if and only if $e \rightarrow^* v$. 