Concepts of Programming Language Design

Abstract Machines Exercises

Liam O'Connor-Davis
Gabriele Keller

November 6, 2024

1. **Decision Machines**: Suppose we have a language of nested brackets N (where ϵ is the empty string):

$$\frac{}{\epsilon\ \boldsymbol{N}} \quad (N\text{-}1)$$

$$\frac{e\ \boldsymbol{N}}{\texttt{(}e\texttt{)}\ \boldsymbol{N}} \quad (N\text{-}2)$$

$$\frac{e\ \boldsymbol{N}}{\langle e\rangle\ \boldsymbol{N}} \quad (N\text{-}3)$$

$$\frac{e\ \boldsymbol{N}}{\texttt{[}e\texttt{]}\ \boldsymbol{N}} \quad (N\text{-}4)$$

Note that `()()` is not a string in this language.

We developed a simple abstract machine to check if strings are in this language. We set the states for the machine to be simply strings. Initial states are all non-empty strings, and the final state is the empty string. Then, our state transition relation is:

$$\frac{}{\texttt{(}e\texttt{)} \mapsto e} \quad (M\text{-}1)$$

$$\frac{}{\texttt{[}e\texttt{]} \mapsto e} \quad (M\text{-}2)$$

$$\frac{}{\langle e\rangle \mapsto e} \quad (M\text{-}3)$$

(a) A machine recognises a language if any machine in the state corresponding to a string S will eventually reach a final state if and only if the string S is in the language.

i. Show that the string $\texttt{([}\langle\rangle\texttt{])}$ is in the language N, and show that our machine reaches a final state given the same string, i.e: $\texttt{([}\langle\rangle\texttt{])} \overset{!}{\mapsto} \epsilon$.

Solution: The string is in the language, as shown:

$$\dfrac{\dfrac{\dfrac{\dfrac{}{\epsilon\ \boldsymbol{N}}N_1}{\langle\rangle\ \boldsymbol{N}}N_3}{\texttt{[}\langle\rangle\texttt{]}\ \boldsymbol{N}}N_4}{\texttt{([}\langle\rangle\texttt{])}\ \boldsymbol{N}}N_2$$

The machine derivation is simply:

$$\begin{array}{lll} & \texttt{([}\langle\rangle\texttt{])} & \\ \mapsto & \texttt{[}\langle\rangle\texttt{]} & (M_1) \\ \mapsto & \langle\rangle & (M_2) \\ \mapsto & \epsilon & (M_3) \end{array}$$

ii. Show that the string `[]()[]` is not in the language juN, and show that our machine reaches a stuck state given the same string, i.e, there exists some stuck state$s$ such that `[]()[]` $\overset{\star}{\mapsto}$ s

Solution: If we attempt to derive $[]()[]\ \boldsymbol{N}$:

$$\dfrac{\dfrac{???}{]()[\ \boldsymbol{N}}}{[]()[]\ \boldsymbol{N}}N_4$$

We get the subgoal $]()[\ \boldsymbol{N}$, which is false, as all strings in N are either ϵ or begin with an opening bracket. Hence, as the rules are unambiguous, there is no other way to derive $[]()[]\ \boldsymbol{N}$ and hence it is not in N. Similarly, our machine derivation:

$$\begin{array}{lll} & []()[] & \\ \mapsto &]()[& (M_2) \\ \mapsto & ??? & \end{array}$$

We end up in the state $]()[$, which is a stuck state, as there are no transitions from a state that begins with a closing bracket.

iii. **Possibly difficult** Prove that the machine recognises the language N, that is:

α) $s\ \boldsymbol{N} \implies s \overset{!}{\mapsto} \epsilon$. Remember that $\overset{!}{\mapsto}$ is just $\overset{\star}{\mapsto}$, where the second state is a final state. $\overset{\star}{\mapsto}$ of course being the reflexive transitive closure of $\mapsto$, that is:

$$\dfrac{}{s \overset{\star}{\mapsto} s}\textsc{Refl}^* \qquad \dfrac{s_1 \mapsto s_2 \quad s_2 \overset{\star}{\mapsto} s_3}{s_1 \overset{\star}{\mapsto} s_3}\textsc{Trans}^*$$

Solution:

Base case: Where $s = \epsilon$, we must show $\epsilon \mapsto s\epsilon$. We can show this using the reflexivity rule:

$$\dfrac{}{\epsilon \overset{\star}{\mapsto} \epsilon}\textsc{Refl}^*$$

Inductive cases: Where $s = (s')$, with the inductive hypothesis that $s'\ \boldsymbol{N} \implies s' \mapsto s\epsilon$, we must show that, if $(s')\ \boldsymbol{N}$, then $(s') \mapsto s\epsilon$. By inversion of rule N_2, on our assumption $(s')\ \boldsymbol{N}$, we can deduce that $s'\ \boldsymbol{N}(*)$. Then, we simply derive our goal as follows:

$$\dfrac{\dfrac{}{(s') \mapsto s'}M_1 \qquad \dfrac{\dfrac{}{s'\ \boldsymbol{N}}(*)}{s' \mapsto s\epsilon}\text{I.H}}{(s') \mapsto s\epsilon}\textsc{Trans}^*$$

The other inductive cases are extremely similar. □

β) $s \overset{!}{\mapsto} \epsilon \implies s\ \boldsymbol{N}$

Solution:

Proof. Here we use induction over the number of steps in the machine's execution, which is the same as induction over the definition of $\mapsto s$.
Base case: Where the length of the execution is zero - i.e, we are already in a final state. The only final state is ϵ, and hence our proof goal is just $\epsilon\ \boldsymbol{N}$, which is already known from rule N_1.

Inductive case: Where our state s executes in one step to s' ($s \mapsto s'$), and $s' \mapsto s\epsilon$ $(*)$. We have the inductive hypothesis $s' \mapsto s\epsilon \implies s'\ \boldsymbol{N}$. We must show that $s\ \boldsymbol{N}$. We proceed by case distinction on s. Seeing as $s \mapsto s'$, s must be one of (s') (by rule M_1), $[s']$ (by rule M_2), or $\langle s' \rangle$ (by rule M_3). All three cases are nearly identical, so we will deal with just the first case, where $s = (s')$.

$$\dfrac{\dfrac{\dfrac{}{s' \mapsto s\epsilon}(*)}{s'\ \boldsymbol{N}}\text{I.H}}{(s')\ \boldsymbol{N}}N_2$$

□

(b) Suppose that we were unable to efficiently read from both the beginning and end of the string simultaneously (For example, if a tape or a linked list is used to represent the string). This makes our original machine highly inefficient, as each state transition must examine the end of a string for a closing bracket.

We develop a new, stack-based machine that attempts to solve this problem. Our stack consists of three symbols, $\mathtt{P}, \mathtt{A}$, and $\mathtt{B}$, one for each type of bracket. The states of the machine are of the form $s \mid e$, where s is a stack and e is a string. Our initial states are all states with an empty stack and a non-empty string, i.e: $\circ \mid e$, our final state is $\circ \mid \epsilon$, and our state transitions are as follows:

$$\dfrac{}{s \mid \mathtt{(}e \mapsto \mathtt{P} \rhd s \mid e}S_1 \qquad \dfrac{}{s \mid \langle e \mapsto \mathtt{A} \rhd s \mid e}S_2 \qquad \dfrac{}{s \mid \mathtt{[}e \mapsto \mathtt{B} \rhd s \mid e}S_3$$

$$\dfrac{}{\mathtt{P} \rhd s \mid \mathtt{)}e \mapsto s \mid e}S_4 \qquad \dfrac{}{\mathtt{A} \rhd s \mid \rangle e \mapsto s \mid e}S_5 \qquad \dfrac{}{\mathtt{B} \rhd s \mid \mathtt{]}e \mapsto s \mid e}S_6$$

i. Show the execution of the new stack machine given the start state $\circ \mid \mathtt{[(}\langle\rangle\mathtt{)]}$.

Solution: The machine execution proceeds as follows:

$$\begin{array}{lll}
 & \circ \mid [(\langle\rangle)] & \\
\mapsto & \mathtt{B} \rhd \circ \mid (\langle\rangle)] & (S_3) \\
\mapsto & \mathtt{P} \rhd \mathtt{B} \rhd \circ \mid \langle\rangle)] & (S_1) \\
\mapsto & \mathtt{A} \rhd \mathtt{P} \rhd \mathtt{B} \rhd \circ \mid \rangle)] & (S_2) \\
\mapsto & \mathtt{P} \rhd \mathtt{B} \rhd \circ \mid)] & (S_5) \\
\mapsto & B \rhd \circ \mid] & (S_4) \\
\mapsto & \circ \mid \epsilon & (S_6)
\end{array}$$

ii. Does the new machine recognise N?

α) **Difficult**: Prove or disprove that $s\ \boldsymbol{N} \implies \circ \mid s \overset{!}{\mapsto} \circ \mid \epsilon$ for all strings s. Hint: You may find it useful to prove the following lemma:

$$\dfrac{s_1 \overset{\star}{\mapsto} s_2 \quad s_2 \mapsto s_3}{s_1 \overset{\star}{\mapsto} s_3}\textsc{Lemma}$$

Also, you may need to generalise your proof goal to a broader claim.

Solution:

Proof of Lemma. We will prove the lemma provided above first, as it will come in handy. We proceed by induction on the size of the execution $s_1 \mapsto s s_2$, and must show that, given $s_2 \mapsto s_3$ ($\dagger$), that $s_1 \mapsto s_3$.
Base case: $s_1 \overset{0}{\mapsto} s_2$, i.e $s_1 = s_2$. We must therefore show that $s_2 \mapsto s s_3$:

$$\dfrac{\dfrac{}{s_2 \mapsto s_3}(\dagger) \qquad \dfrac{}{s_3 \mapsto s s_3}\textsc{Refl}^*}{s_2 \mapsto s s_3}\textsc{Trans}^*$$

Inductive case: When $s_1 \mapsto s_1'$ $(*)$, and $s_1' \mapsto s s_2$ $(**)$, and we have the inductive hypothesis:

$$\dfrac{s_1' \mapsto s s_2 \qquad s_2 \mapsto s_3}{s_1' \mapsto s s_3}\text{I.H}$$

Then, we simply derive the proof goal:

$$\dfrac{\dfrac{}{s_1 \mapsto s_1'}(*) \qquad \dfrac{\dfrac{}{s_1' \mapsto s s_2}(**) \qquad \dfrac{}{s_2 \mapsto s_3}(\dagger)}{s_1' \mapsto s s_3}\text{I.H}}{s_1 \mapsto s_3}\textsc{Trans}^*$$

□

Proof of main theorem. Now that we have proven the lemma, we must now prove that $s\ \boldsymbol{N} \implies \circ \,|\, s \overset{!}{\mapsto} \circ \,|\, \epsilon$. We will generalise this proof goal to make the stronger claim that $s\ \boldsymbol{N} \implies t \,|\, sr \mapsto st \,|\, r$ for any stack t and remainder string r. Note that this trivially implies our original proof goal by setting t to $\circ$ and r to ϵ.
Base case: Where $s = \epsilon$, we must therefore show that $t \,|\, r \mapsto st \,|\, r$, trivially shown by rule $\textsc{Refl}^*$.
Inductive case: $s = (s')$, where we have the inductive hypothesis: $s'\ \boldsymbol{N} \implies t' \,|\, s'r' \mapsto st' \,|\, r'$, for any t' and r'. We must show that, assuming $(s')\ \boldsymbol{N}$, $t|(s')r \mapsto st|r$ for all t, r. Note that by inversion of rule N_2 on our assumption, we know $s'\ \boldsymbol{N}\,(*)$

$$\dfrac{\dfrac{}{t \,|\, (s')r \mapsto \mathtt{P} \triangleright t \,|\, s')r}S_1 \qquad \dfrac{\dfrac{\dfrac{}{s'\ \boldsymbol{N}}(*)}{\mathtt{P} \triangleright t \,|\, s')r \mapsto s\mathtt{P} \triangleright t \,|\,)r}\text{I.H}^1 \qquad \dfrac{}{\mathtt{P} \triangleright t \,|\,)r \mapsto t \,|\, r}S_4}{\mathtt{P} \triangleright t \,|\, s')r \mapsto st \,|\, r}\textsc{Lemma}}{t \,|\, (s')r \mapsto st \,|\, r}\textsc{Trans}^*$$

The other inductive cases are extremely similar. □
^{1}: The application of the I.H rule here sets t' to be $\mathtt{P} \triangleright t$ and r' to be $)r$.

β) Prove or disprove that $\circ \,|\, s \overset{!}{\mapsto} \circ \,|\, \epsilon \implies s\ \boldsymbol{N}$

Solution:

Counterexample. We will disprove this by way of a counterexample. It is already established that $()()$ is not in N. We will show that $\circ \,|\, ()() \mapsto s \circ \,|\, \epsilon$ and thus there is no way that $\circ \,|\, s \overset{!}{\mapsto} \circ \,|\, \epsilon$ could imply $s\ \boldsymbol{N}$.
The machine execution is as follows:

$$
\begin{array}{lll}
 & \circ \mid ()() & \\
\mapsto & \mathtt{P} \rhd \circ \mid)() & (S_1) \\
\mapsto & \circ \mid () & (S_4) \\
\mapsto & \mathtt{P} \rhd \circ \mid) & (S_1) \\
\mapsto & \circ \mid \epsilon & (S_4)
\end{array}
$$
□

iii. If your answer to the previous question was no, amend the structure of the stack machine so that it does recognise N (efficiently). Explain your answer.

Solution: The problem with the existing machine is that it recognises any amount of strings in N placed next to each other. A string in N consists of a sequence of opening brackets, followed by a sequence of closing brackets. Once a closing bracket has been observed by the machine, it should not see any opening brackets. To fix this, we modify the state such that there are two modes, pushing ($\succ$), and popping ($\prec$). The machine starts in pushing mode, i.e: $\circ \succ s$ for some string s, and now we have two terminating states: $\circ \succ \epsilon$ and $\circ \prec \epsilon$. Our transition rules are updated as follows:

$$
\frac{}{s \succ (e \mapsto \mathtt{P} \rhd s \succ e} \qquad \frac{}{s \succ \langle e \mapsto \mathtt{A} \rhd s \succ e} \qquad \frac{}{s \succ [e \mapsto \mathtt{B} \rhd s \succ e}
$$

$$
\frac{}{\mathtt{P} \rhd s \succ)e \mapsto \mathtt{P} \rhd s \prec)e} \qquad \frac{}{\mathtt{A} \rhd s \succ \rangle e \mapsto \mathtt{A} \rhd s \prec \rangle e} \qquad \frac{}{\mathtt{B} \rhd s \succ]e \mapsto \mathtt{B} \rhd s \prec]e}
$$

$$
\frac{}{\mathtt{P} \rhd s \prec)e \mapsto s \prec e} \qquad \frac{}{\mathtt{A} \rhd s \prec \rangle e \mapsto s \prec e} \qquad \frac{}{\mathtt{B} \rhd s \prec]e \mapsto s \prec e}
$$

As there are no rules to go from popping to pushing mode, the machine cannot push a symbol after one has been popped, and hence the machine recognises N.

2. **Computing Machines**: Abstract machines are not just used for decision problems (yes/no answers), they can also be used to compute results. Can you think of a machine to compute binary addition?

   (a) Formalise such a machine.

   Hint: Think about the algorithm you would use when adding up large binary numbers on paper.

Solution: The machine's states are of the form:

$$
\left.\begin{matrix} n_1 \\ n_2 \end{matrix}\right\} s \ \langle\!\langle \mathtt{c} \rangle\!\rangle
$$

Where s, n_1 and n_2 are strings of binary digits, and c is a single carry bit. n_1 and n_2 are also padded with zeros so as to be the same length.

Initial states are all states where s is empty and the carry bit is *zero*:

$$
\left.\begin{matrix} n_1 \\ n_2 \end{matrix}\right\} \epsilon \ \langle\!\langle \mathtt{0} \rangle\!\rangle
$$

Final states are all states where n_1 and n_2 are empty and the carry bit is *zero*:

$$
\left.\begin{matrix} \epsilon \\ \epsilon \end{matrix}\right\} s \ \langle\!\langle \mathtt{0} \rangle\!\rangle
$$

The transition rules work as follows:

$$\dfrac{}{\left.\begin{matrix} n_1\texttt{0} \\ n_2\texttt{0}\end{matrix}\right\} s \;\; \langle\!\langle \texttt{0} \rangle\!\rangle \mapsto \left.\begin{matrix} n_1 \\ n_2\end{matrix}\right\} \texttt{0}s \;\; \langle\!\langle \texttt{0}\rangle\!\rangle} B_1 \qquad \dfrac{}{\left.\begin{matrix} n_1\texttt{0} \\ n_2\texttt{1}\end{matrix}\right\} s \;\; \langle\!\langle \texttt{0} \rangle\!\rangle \mapsto \left.\begin{matrix} n_1 \\ n_2\end{matrix}\right\} \texttt{1}s \;\; \langle\!\langle \texttt{0}\rangle\!\rangle} B_2$$

$$\dfrac{}{\left.\begin{matrix} n_1\texttt{1} \\ n_2\texttt{0}\end{matrix}\right\} s \;\; \langle\!\langle \texttt{0} \rangle\!\rangle \mapsto \left.\begin{matrix} n_1 \\ n_2\end{matrix}\right\} \texttt{1}s \;\; \langle\!\langle \texttt{0}\rangle\!\rangle} B_3 \qquad \dfrac{}{\left.\begin{matrix} n_1\texttt{1} \\ n_2\texttt{1}\end{matrix}\right\} s \;\; \langle\!\langle \texttt{0} \rangle\!\rangle \mapsto \left.\begin{matrix} n_1 \\ n_2\end{matrix}\right\} \texttt{0}s \;\; \langle\!\langle \texttt{1}\rangle\!\rangle} B_4$$

$$\dfrac{}{\left.\begin{matrix} n_1\texttt{0} \\ n_2\texttt{0}\end{matrix}\right\} s \;\; \langle\!\langle \texttt{1} \rangle\!\rangle \mapsto \left.\begin{matrix} n_1 \\ n_2\end{matrix}\right\} \texttt{1}s \;\; \langle\!\langle \texttt{0}\rangle\!\rangle} B_1c \qquad \dfrac{}{\left.\begin{matrix} n_1\texttt{0} \\ n_2\texttt{1}\end{matrix}\right\} s \;\; \langle\!\langle \texttt{1} \rangle\!\rangle \mapsto \left.\begin{matrix} n_1 \\ n_2\end{matrix}\right\} \texttt{0}s \;\; \langle\!\langle \texttt{1}\rangle\!\rangle} B_2c$$

$$\dfrac{}{\left.\begin{matrix} n_1\texttt{1} \\ n_2\texttt{0}\end{matrix}\right\} s \;\; \langle\!\langle \texttt{1} \rangle\!\rangle \mapsto \left.\begin{matrix} n_1 \\ n_2\end{matrix}\right\} \texttt{0}s \;\; \langle\!\langle \texttt{1}\rangle\!\rangle} B_3c \qquad \dfrac{}{\left.\begin{matrix} n_1\texttt{1} \\ n_2\texttt{1}\end{matrix}\right\} s \;\; \langle\!\langle \texttt{1} \rangle\!\rangle \mapsto \left.\begin{matrix} n_1 \\ n_2\end{matrix}\right\} \texttt{1}s \;\; \langle\!\langle \texttt{1}\rangle\!\rangle} B_4c$$

$$\dfrac{}{\left.\begin{matrix} \epsilon \\ \epsilon\end{matrix}\right\} s \;\; \langle\!\langle \texttt{1} \rangle\!\rangle \mapsto \left.\begin{matrix} \epsilon \\ \epsilon\end{matrix}\right\} \texttt{1}s \;\; \langle\!\langle \texttt{0}\rangle\!\rangle} B_{\text{overflow}}$$

(b) Compute the result `110 + 1010` with your machine. Show each execution step.

Solution:
The result is 10000, as shown below:

$$\begin{array}{lll}
 & \left.\begin{matrix} \texttt{0110} \\ \texttt{1010}\end{matrix}\right\} \epsilon \;\; \langle\!\langle \texttt{0} \rangle\!\rangle & \\
\mapsto & \left.\begin{matrix} \texttt{011} \\ \texttt{101}\end{matrix}\right\} 0 \;\; \langle\!\langle \texttt{0} \rangle\!\rangle & (B_1) \\
\mapsto & \left.\begin{matrix} \texttt{01} \\ \texttt{10}\end{matrix}\right\} 00 \;\; \langle\!\langle \texttt{1} \rangle\!\rangle & (B_4) \\
\mapsto & \left.\begin{matrix} \texttt{0} \\ \texttt{1}\end{matrix}\right\} 000 \;\; \langle\!\langle \texttt{1} \rangle\!\rangle & (B_3c) \\
\mapsto & \left.\begin{matrix} \epsilon \\ \epsilon\end{matrix}\right\} 0000 \;\; \langle\!\langle \texttt{1} \rangle\!\rangle & (B_2c) \\
\mapsto & \left.\begin{matrix} \epsilon \\ \epsilon\end{matrix}\right\} 10000 \;\; \langle\!\langle \texttt{0} \rangle\!\rangle & (B_{\text{overflow}})
\end{array}$$

3. **Evaluation Machines**: We can also use abstract machines to express the operational semantics of the λ-calculus. As abstract higher-order syntax representation of λ-terms we chose the following:

$$\dfrac{x \textit{ is a var}}{x \;\; \lambda\textit{-term}} \qquad \dfrac{t \;\; \lambda\textit{-term}}{(\texttt{Function } x.t) \;\; \lambda\textit{-term}} \qquad \dfrac{t_1 \;\; \lambda\textit{-term} \quad t_2 \;\; \lambda\textit{-term}}{(\texttt{Apply } t_1 \; t_2) \;\; \lambda\textit{-term}}$$

$$\dfrac{}{(\texttt{Function } x.t) \Downarrow \langle\!\langle x.t \rangle\!\rangle} \textsc{Lambda} \qquad \dfrac{e_1 \Downarrow \langle\!\langle x.t \rangle\!\rangle \quad e_2 \Downarrow e_2' \quad t[x := e_2'] \Downarrow r}{(\texttt{Apply } e1 \; e2) \Downarrow r} \textsc{Apply}$$

Note: You may notice that this language only has functions! Turns out, you can encode any kind of data as a function, and that this is sufficient for a turing-complete programming language. This language was invented by Alonzo Church as an alternative solution to the Entscheidungsproblem, and is called the lambda calculus.

(a) Develop a structural operational ("small step") semantics for this language.

i. Include a rule for function literals, if necessary.

Solution:

$$\frac{}{\texttt{(Function } x.t\texttt{)} \mapsto \langle\!\langle x.t \rangle\!\rangle}\textsc{Lambda}$$

ii. Include three rules for function application. Assume the function expression is evaluated before the argument expression. Note that this language does not include recursion.

Solution:

$$\frac{t_1 \mapsto t_1'}{\texttt{(Apply } t_1\ t_2\texttt{)} \mapsto \texttt{(Apply } t_1'\ t_2\texttt{)}}\textsc{Apply}_1 \qquad \frac{t_2 \mapsto t_2'}{\texttt{(Apply } \langle\!\langle x.t \rangle\!\rangle\ t_2\texttt{)} \mapsto \texttt{(Apply } \langle\!\langle x.t \rangle\!\rangle\ t_2'\texttt{)}}\textsc{Apply}_2$$

$$\frac{}{\texttt{(Apply } \langle\!\langle x.t \rangle\!\rangle\ \langle\!\langle a.t' \rangle\!\rangle\texttt{)} \mapsto t[x := \langle\!\langle a.t' \rangle\!\rangle]}\textsc{Apply}_3$$

(b) Now define an abstract machine which eliminates recursion from the meta-level of the semantics to include an explicit stack, a la the C Machine.

i. Define a suitable stack formalism.

Solution:

$$\frac{}{\circ\ \textbf{\textit{Stack}}} \qquad \frac{x\ \textbf{\textit{Frame}} \quad s\ \textbf{\textit{Stack}}}{x \triangleright s\ \textbf{\textit{Stack}}}$$

Where a *Frame* is simply either `(Apply` $\square$ x`)` or `(Apply` x $\square$ `)` for some x.

ii. Define the set of states Q, the set of initial states $I \subseteq Q$, and the set of final states $F \subseteq Q$.

Solution: The set of states consists of: either a value or an expression, and a stack:

$$\frac{s\ \textbf{\textit{Stack}} \quad e\ \textbf{\textit{Expr}}}{s \mid e \in Q} \qquad \frac{s\ \textbf{\textit{Stack}} \quad e\ \textbf{\textit{Expr}}}{\textit{todo}}$$

Initial states are an expression with an empty stack:

$$\frac{e\ \textbf{\textit{Expr}}}{\textit{todo}}$$

Final states are a function value with an empty stack

iii. Define a state transition rule for function literals

Solution:

$$\frac{}{s \mid \texttt{(Function } x.t\texttt{)} \mapsto s \mid \langle\!\langle x.t \rangle\!\rangle}$$

iv. Include three rules for function application, using capture-avoiding substitution as a built-in machine operation.

Solution:

$$\frac{}{s \mid \texttt{(Apply}\ e_1\ e_2\texttt{)} \mapsto \texttt{(Apply}\ \Box\ e_2\texttt{)} \triangleright s \mid e_1}$$

$$\frac{}{\texttt{(Apply}\ \Box\ e_2\texttt{)} \triangleright s \mid \langle\!\langle x.t \rangle\!\rangle \mapsto \texttt{(Apply}\ \langle\!\langle x.t \rangle\!\rangle\ \Box\texttt{)} \triangleright s \mid e_2}$$

$$\frac{}{\texttt{(Apply}\ \langle\!\langle x.t \rangle\!\rangle\ \Box\texttt{)} \triangleright s \mid \langle\!\langle a.b \rangle\!\rangle \mapsto s \mid t[x := \langle\!\langle a.b \rangle\!\rangle]}$$

(c) Now suppose that we want to eliminate substitution from our machine. Extend the semantics to include environments, a la the E Machine. Recall than an environment is commonly defined as:

$$\frac{}{\bullet\ \textit{Env}} \qquad \frac{x\ \textit{Ident} \quad t\ \textit{Expr} \quad \Gamma\ \textit{Env}}{x \leftarrow y; \Gamma\ \textit{Env}}$$

i. Revise your definition of the state sets Q, I and F, and of the stack.

Solution: Our stack can now also include environments:

$$\frac{s\ \textit{Stack} \quad \Gamma\ \textit{Env}}{\Gamma \triangleright s\ \textit{Stack}}$$

Our state now also includes a current environment, of the form $s \mid \Gamma \mid e$, where s is a Stack, Γ is an environment and e is an expression.
I and F are unchanged except that they include the empty environment.

ii. Revise your transition rule for function literals. Note that these function literals should produce closures which capture the environment at their definition.

Solution:

$$\frac{}{s \mid \Gamma \mid \texttt{(Function}\ x.t\texttt{)} \mapsto s \mid \Gamma \mid \langle\!\langle \Gamma, x.t \rangle\!\rangle}$$

iii. Revise your rules for function application

Solution: All the rules are essentially unchanged (preserving the environment), except for the final application rule:

$$\frac{}{\texttt{(Apply}\ \langle\!\langle \Gamma, x.t \rangle\!\rangle, \Box\texttt{)} \triangleright s \mid \Delta \mid \langle\!\langle E, a.b \rangle\!\rangle \mapsto \Delta \triangleright s \mid x \leftarrow \langle\!\langle E, a.b \rangle\!\rangle; \Gamma \mid t}$$

iv. Include any additional rules necessary to complete the definition, such as variable lookup.

Solution: Variable Lookup:

$$\frac{}{s \mid x \leftarrow t; \Gamma \mid x \mapsto s \mid x \leftarrow t; \Gamma \mid t}$$

Popping environments from the stack, back into the current environment:

$$\frac{}{\Gamma \triangleright s \mid \Delta \mid \langle\!\langle E, x.t \rangle\!\rangle \mapsto s \mid \Gamma \mid \langle\!\langle E, x.t \rangle\!\rangle}$$

v. Give an example of an expression in this language which requires closures in order to evaluate correctly. Explain your answer.

Solution: A simple example is:

(`Apply` (`Apply` (`Function` x.(`Function` y.(`Apply` x y))) (`Function` $a.a$)) (`Function` $b.b$))

Evaluating the outer application will cause the inner application to be evaluated first, where x is bound to $\langle\!\langle a.a \rangle\!\rangle$. Without closures, the inner application will return the function
$\langle\!\langle y.(\texttt{Apply}\ x\ y) \rangle\!\rangle$ back beyond the stack frame where x is in scope resulting in a free variable inside the function. When the outer application is finally evaluated, one would end up encountering x free in the program, and be in a stuck state. With closures, however, the environment containing the binding for x is captured in the returned function and x will not be found free.

4. **Stack Machines**: In this question, we will examine a machine that is quite similar to a type of machine used in virtual machines, such as the JVM, called a stack machine. Imagine an arithmetic expression language with the following big step semantics:

$$\frac{x \in \mathbb{Z}}{(\texttt{Num}\ x) \Downarrow x}\textsc{Num} \qquad \frac{x \Downarrow x' \quad y \Downarrow y'}{(\texttt{Plus}\ x\ y) \Downarrow x' + y'}\textsc{Plus} \qquad \frac{x \Downarrow x' \quad y \Downarrow y'}{(\texttt{Times}\ x\ y) \Downarrow x' \times y'}\textsc{Times}$$

We have a machine, called the J Machine, that's capable of performing these operations, however it works by using a stack to store operands and accumulate results. For example, 4 * (2 + 3) would be the following program in the J Machine's bytecode: `(Val 4)`; `(Val 2)`; `(Val 3)`; `add`; `Times`. Each `Val` instruction pushes a value to the stack, and each operation instruction pops two values off, and pushes the result of the operation.

Formally, the J Machine is specified as follows: The machine consists of three instructions:

$$\frac{x \in \mathbb{Z}}{(\texttt{Val}\ x)\ \textit{\textbf{Inst}}} \qquad \frac{}{\texttt{Plus}\ \textit{\textbf{Inst}}} \qquad \frac{}{\texttt{Times}\ \textit{\textbf{Inst}}}$$

The state of the machine consists of a list of instructions, called a Program, and a stack of integers:

$$\frac{}{\texttt{Halt}\ \textit{\textbf{Program}}} \qquad \frac{i\ \textit{\textbf{Inst}} \quad p\ \textit{\textbf{Program}}}{i; p\ \textit{\textbf{Program}}}$$

$$\frac{}{\circ\ \textit{\textbf{Stack}}} \qquad \frac{x \in \mathbb{Z} \quad s\ \textit{\textbf{Stack}}}{x \triangleright s\ \textit{\textbf{Stack}}}$$

They are presented in the form $s \mid p$ where s is a stack and p is program. The initial state consists of the empty stack and any nonempty program p i.e, $\circ \mid p$. The final state consists of a stack with merely one element r (the result of the computation), and the empty program, i.e, $r \triangleright \circ \mid \texttt{Halt}$.

The state transition rules are as follows:

$$\frac{}{s \mid (\texttt{Val}\ x); p \mapsto x \triangleright s \mid p}J_1 \qquad \frac{}{y \triangleright x \triangleright s \mid \texttt{Plus}; p \mapsto x + y \triangleright s \mid p}J_2 \qquad \frac{}{y \triangleright x \triangleright s \mid \texttt{Times}; p \mapsto x \times y \triangleright s \mid p}J_3$$

(a) Translate the expression `(Plus (Times (Num -1) (Num 7)) (Num 7))` into a J Machine program, and write down each step the J Machine would take to execute this program.

Solution: The program is: `(Val -1)`; `(Val 7)`; `Times`; `(Val 7)`; `Plus`; `Halt`.

Execution is as follows:

$$\begin{array}{lll}
 & \circ \mid \texttt{(Val -1)};\texttt{(Val 7)};\texttt{Times};\texttt{(Val 7)};\texttt{Plus};\texttt{Halt} & \\
\mapsto & \text{-1} \triangleright \circ \mid \texttt{(Val 7)};\texttt{Times};\texttt{(Val 7)};\texttt{Plus};\texttt{Halt} & (J_1) \\
\mapsto & 7 \triangleright \text{-1} \triangleright \circ \mid \texttt{Times};\texttt{(Val 7)};\texttt{Plus};\texttt{Halt} & (J_1) \\
\mapsto & \text{-7} \triangleright \circ \mid \texttt{(Val 7)};\texttt{Plus};\texttt{Halt} & (J_3) \\
\mapsto & 7 \triangleright \text{-7} \triangleright \circ \mid \texttt{Plus};\texttt{Halt} & (J_1) \\
\mapsto & 0 \triangleright \circ \mid \texttt{Halt} & (J_2)
\end{array}$$

(b) **Possibly difficult** Formalise (using inference rules) a "compilation" relation $\curlyvee : \mathit{Expr} \times \mathit{Program}$ which translates expressions in the arithmetic language to the semantically equivalent J Machine bytecode.

Solution:

$$\frac{}{\texttt{(Num } n\texttt{)} \curlyvee \texttt{(Val } n\texttt{)};\texttt{Halt}}\,\textsc{Num}_J$$

$$\frac{n \curlyvee n';\texttt{Halt} \quad m \curlyvee m';\texttt{Halt}}{\texttt{(Plus } n, m\texttt{)} \curlyvee n';m';\texttt{Plus};\texttt{Halt}}\,\textsc{Plus}_J \qquad \frac{n \curlyvee n';\texttt{Halt} \quad m \curlyvee m';\texttt{Halt}}{\texttt{(Times } n, m\texttt{)} \curlyvee n';m';\texttt{Times};\texttt{Halt}}\,\textsc{Times}_J$$

(c) **Definitely difficult**. Suppose we wanted to add a `Let` construct to add variables to our arithmetic language, using environments as shown:

$$\frac{i \in \mathbb{Z}}{\Gamma \vdash \texttt{(Num } i\texttt{)} \Downarrow i}\,\textsc{Num} \qquad \frac{\Gamma \vdash x \Downarrow x' \quad \Gamma \vdash y \Downarrow y'}{\Gamma \vdash \texttt{(Plus } x\ y\texttt{)} \Downarrow x' + y'}\,\textsc{Plus} \qquad \frac{\Gamma \vdash x \Downarrow x' \quad \Gamma \vdash y \Downarrow y'}{\Gamma \vdash \texttt{(Times } x\ y\texttt{)} \Downarrow x' \times y'}\,\textsc{Times}$$

$$\frac{\Gamma \vdash e_1 \Downarrow v_1 \quad \Gamma \cup \{x \leftarrow v_1\} \vdash e_2 \Downarrow v_2}{\Gamma \vdash \texttt{(Let } x\ e_1\ e_2\texttt{)} \Downarrow v_2}\,\textsc{Let} \qquad \frac{x \leftarrow v \in \Gamma}{\Gamma \vdash \texttt{(Var } x\texttt{)} \Downarrow v}\,\textsc{Var}$$

Extend the J Machine to support this construct, and expand your $\curlyvee$ relation to include the correct translation. Don't forget to deal with name shadowing by exploiting stacks.

Solution: We extend the state definition of the states in the machine to include an additional stack of environments, called scopes, notated as $z \mid s \mid p$, where z is the integer stack and s is the scope stack. The initial states now look like this:

$$\frac{}{\circ \mid \{\} \mid p}$$

That is, they start with the empty environment sitting at the bottom of the scope stack. Similarly, final states also have the empty environment only on their scope stack.

We introduce three new instructions, `Scope`, `Descope`, and `Var`, which have the following semantics:

`(Scope x)` pushes a new environment to the scope stack. The new environment is the same as the old environment except it includes a new binding[1] from the name x to the value on the top of the value stack. The value stack is also popped.

`(Descope x)` simply pops the scope stack. `(Var x)` pushes the value of a variable to the value stack. The value is determined by looking in the topmost scope environment.

$$\frac{}{v \triangleright s \mid \Gamma \triangleright \zeta \mid \texttt{(Scope } x\texttt{)};p \mapsto s \mid \Gamma \cup \{x \leftarrow v\} \triangleright \zeta \mid p}\,J_4 \qquad \frac{}{s \mid \Gamma \triangleright \zeta \mid \texttt{Descope};p \mapsto s \mid \zeta \mid p}\,J_5$$

$$\frac{}{s \mid \{x \leftarrow v\} \cup \Gamma \triangleright \zeta \mid \texttt{(Var } x\texttt{)}; p \mapsto v \triangleright s \mid \{x \leftarrow v\} \cup \Gamma \triangleright \zeta \mid p} J_6$$

[1]: Because each environment is a superset of the last, pointer magic could be used here to make this efficient in practice.

As for the compilation relation, we translate `Let` as follows:

$$\frac{e_1 \curlyvee e_1'; \texttt{Halt} \quad e_2 \curlyvee e_2'; \texttt{Halt}}{\texttt{(Let } x\ e_1\ e_2\texttt{)} \curlyvee e_1'; \texttt{(Scope } x\texttt{)}; e_2'; \texttt{Descope}; \texttt{Halt}} \textsc{Let}_J$$

And, for variable lookup, it's quite simple:

$$\frac{}{\texttt{(Var } x\texttt{)} \curlyvee \texttt{(Var } x\texttt{)}} \textsc{Var}_J$$

Note: This is basically how the JVM bytecode works (modulo some OO features).