We have been writing Rip interpreters in Scheme in order to understand various features of programming languages. What if we want to explain our interpreter to someone who doesn’t know Scheme? It would be nice to have some common language for explaining interpreters. We already have one—math!

We are going to write out a series of rules. Using these rules, a person familiar with mathematics can evaluate any Rip program.

Let’s start out with the simplest case—numbers. If our program is a number \( n \), it just evaluates to \( n \). Hence, we write the rule

\[ n \Rightarrow n \], where \( n \) is syntax for a Rip number and \( n \) is a number

We read the \( \Rightarrow \) as “reduces to”. This rule says that any expression of the form \( n \) (i.e. a Rip number) can be reduced to the number \( n \). Let’s consider the more interesting case, addition:

\[ \{ + \ LHS \ RHS \} \Rightarrow LHS + RHS \], where LHS and RHS are Rip expressions

This rule is applicable when we have an expression that matches the pattern \( \{ + \ LHS \ RHS \} \). Note that the symbol + on the left-hand-side of the \( \Rightarrow \) is a syntactic marker in Rip, while the + on the right-hand-side is the addition operation of mathematics.

As you may have noticed, this rule is complete nonsense. The right-hand-side of the rule indicates that we use the addition operation of mathematics on Rip expressions. This is meaningless. We must modify our rule so that numerical values are added together, not Rip expressions:

\[
\frac{LHS \Rightarrow lv \quad RHS \Rightarrow rv}{\{ + \ LHS \ RHS \} \Rightarrow lv + rv} , \text{ where LHS and RHS are Rip expressions}
\]

This is called a judgment. A judgment sets up a logical implication. The pieces above the line are the “if” part of the implication, and the pieces below the line are the “then” part of the implication. We read this judgment as follows: If, in the Rip program \( \{ + \)}
LHS \text{ RHS}, the expression corresponding to LHS reduces to \( lv \) and the expression corresponding to RHS reduces to \( rv \), then \( \{+\text{ LHS RHS}\} \) reduces to \( lv + rv \).

We can use these rules to build a \textit{proof} that \( \{+ \{+ 2 \ 3\} 1\} \) reduces to 6. The following proof reads from top to bottom, using only the rules we have made so far:

\[
\begin{align*}
2 & \Rightarrow 2 \\
3 & \Rightarrow 3 \\
\{+ 2 3\} & \Rightarrow 5 \\
1 & \Rightarrow 1 \\
\{+ \{+ 2 3\} 1\} & \Rightarrow 6
\end{align*}
\]

Let’s now consider Rip procedures. They evaluate to closures, which we will represent as tuples:

\[
\{\text{proc} \{X\} \text{ BODY}\} \Rightarrow (X,\text{BODY},?)
\]

where \( X \) is a variable name and \( \text{BODY} \) is a Rip expression.

What do we put in place of the “?” in the tuple above? In our interpreter, we close over the extant environment. In order to capture the environment in the tuple, we need to track environments. We do this by adding an environment \( \xi \) to the left-hand-side of each rule. \( \xi \) is a function with contract

\[
\xi : \text{var-name} \rightarrow \text{value}
\]

We rewrite our current rules, this time incorporating \( \xi \). Note that the semi-colon in the following rules is simply a piece of syntax that groups an expression with its environment:

\[
\begin{align*}
n;\xi & \Rightarrow n \\
\text{LHS;}\xi & \Rightarrow lv \quad \text{RHS;}\xi \Rightarrow rv \\
\{+\text{ LHS RHS}\};\xi & \Rightarrow lv + rv \\
\{\text{proc} \{X\} \text{ BODY}\};\xi & \Rightarrow (X,\text{BODY},\xi)
\end{align*}
\]

Now that we keep track of environments, and since \( \xi \) is just a function, the rule for variable expressions is easy:

\[
X;\xi \Rightarrow \xi(X)
\]

The only rule that remains is procedure application. This one is a bit tricky. We need to evaluate the function expression and argument expression, then evaluate the body of the function in an extended environment (i.e. the closure environment plus the binding of the function’s argument):

\[
\begin{align*}
\text{FEXP;}\xi \Rightarrow (X,\text{BODY},\xi') \quad \text{ARGEXP;}\xi \Rightarrow av \quad \text{BODY;}\xi' [X \mapsto av] \Rightarrow appv \\
\{\text{FEXP ARGEXP}\};\xi & \Rightarrow appv
\end{align*}
\]
The notation $\xi' [X \mapsto av]$ denotes an update of environment $\xi'$: The environment $\xi' [X \mapsto av]$ is the same environment as $\xi'$ except that $X$ maps to $av$.

As a side note, this rule for procedure application nicely demonstrates the difference between static scoping and dynamic scoping: Using $\text{BODY} ; \xi [X \mapsto av]$ instead of $\text{BODY} ; \xi' [X \mapsto av]$ in the previous rule would give us dynamic scoping.

Let's use these rules to construct a proof that the Rip program

\[
\left\{ \{ \text{proc} \ (y) \ \{ + \ y \ 5 \} \} \right. \\
\{ + \ 3 \ 4 \} \}
\]

reduces to 12:

\[
\begin{align*}
\{ \text{proc} \ (y) \ \{ + \ y \ 5 \} \} ; \{} & \Rightarrow \langle y, \{ + \ y \ 5 \}, \{} \rangle \\
\{ + \ 3 \ 4 \} ; \{} & \Rightarrow 7 \\
\{ \{ \text{proc} \ (y) \ \{ + \ y \ 5 \} \ \{ + \ 3 \ 4 \} \} ; \{} & \Rightarrow 12
\end{align*}
\]

The final reduction we are trying to prove appears at the bottom of the judgment. This is a procedure application reduction. Using our rule for procedure application, we see that three conditions must be true in order to get our result. Each of these conditions is in turn proved using other reduction rules. Note that the reduction rules do not give us an algorithm for evaluating programs. Instead, they give us a language for proving what the result of a program will be.