1 Adding Functions to Rip

The Rip syntax for functions is \{\texttt{proc} \texttt{\{var\} \texttt{\{expr\}}}\).

We augment our data definition for a Rip program with

\((\texttt{make-procE} \texttt{symbol RP})\)

The corresponding struct definition is

\((\texttt{define-struct} \texttt{procE (arg-name body)})\).

As an example of the data, the Rip code \{\texttt{proc} \texttt{\{x\} \{+ x 1\}}\} has data representation

\((\texttt{make-procE} \texttt{\'x (make-addE (make-varE \texttt{\'x}) (make-numE 1))})\)

We add a \texttt{procE} clause to our interpreter:

\(;; \texttt{interp : RP \rightarrow numE}\)
\(;; \text{Interp reduces an RP to a numE}\)

\(\text{\texttt{define (interp expr)}}\)

\(\text{(cond}\)
\(\text{[(numE? expr) expr]}\)
\(\text{[(addE? expr) (numE+ (interp (addE-left expr)))}}\)
\(\text{\hspace{1cm} (interp (addE-right expr))]}\)
\(\text{[(letE? expr) (subst (letE-body expr)}}\)
\(\text{\hspace{1cm} (letE-arg-name expr)}\)
\(\text{\hspace{2cm} (interp (letE-val expr))]}\)
\(\text{[(varE? expr) (error \texttt{\'undefined-identifier})]}\)
\(\text{(procE? expr) \ldots (procE-arg-name expr) \ldots (procE-body expr) \ldots])\)}
How do we fill in the procE clause? In some sense, a procedure is useless without the ability to apply it to an argument. Therefore, let’s first add procedure application to Rip, and then we will come back to the procE clause.

2 Procedure Application

We add Rip syntax for procedure application.

\{\text{procE} \langle \text{expr} \rangle \}

We add \text{(make-appE procE RP)} to our data definition for RP, with corresponding structure definition \text{(define-struct appE (proc arg))}.

\{\{\text{proc \{x\} x\} 3\}\}
evaluates to 3.

\{\{\text{proc \{x\} \{+ x 1\}\} \{+ 2 3\}\}
evaluates to 6.

How do we interpret an appE? We need to substitute the application arg for the proc’s arg in the body of the proc.

\text{(subst (procE-body (appE-proc expr)) (procE-arg-name (appE-proc expr)) (interp (appE-arg expr)))}

Recall that subst returns a substituted expression, but the interpreter needs to return a numE. We therefore still need to call interp. Thus, our interpreter now looks like

\begin{verbatim}
;; interp : RP \rightarrow \text{numE}
;; Interp reduces an RP to a numE
(define (interp expr)
 (cond
 [\(\text{numE? expr}\) expr]
 [\(\text{addE? expr}\) (numE+ (interp (addE-left expr)) (interp (addE-right expr)))]
 [\(\text{letE? expr}\) (subst (letE-body expr) (letE-var-name expr) (interp (letE-val expr)))]
 [\(\text{varE? expr}\) (error 'undefined-identifier)]
 [\(\text{procE? expr}\) ??]
 [\(\text{appE? expr}\) (interp (subst (procE-body (appE-proc expr)) (procE-arg-name (appE-proc expr)) (interp (appE-arg expr)))]))
\end{verbatim}
3 Interpreting a Procedure

What do we return for \( \text{proc}E \)? When we interpret \( \{ \text{proc} \{ x \} \{ + \ x \ 1 \} \} \ 2 \) we follow the \( \text{app}E \) clause of \( \text{interp} \) first. Consider the \( \text{app}E\text{-proc} \) field of the \( \text{app}E \) term. We do not invoke \( \text{interp} \) on it because the term can only be a \( \text{proc}E \), not any \( \text{RP} \). Instead, we directly select its \( \text{proc}E\text{-arg-name} \) and \( \text{proc}E\text{-body} \) fields, which we know it must have by virtue of being a \( \text{proc}E \). We then substitute in the value of the argument expression, and \( \text{interp} \) the resulting expression.

Since we don’t invoke \( \text{interp} \) on the \( \text{app}E\text{-proc} \) field, and there is no other useful place for a procedure to appear in a Rip program, we don’t really care what \( \text{interp} \) returns when we evaluate a procedure. Indeed, the only legal Rip program in which we might interpret a procedure is when the entire program is a procedure definition, such as \( \{ \text{proc} \{ x \} \ x \} \). We choose to simply return \( \text{procedure} \) in this case, to indicate that the expression is a procedure. If a programmer accidentally uses a procedure where he intends some other value, as in \( \{ + \ 1 \ \{ \text{proc} \{ x \} \ x \} \} \), the result will be a Scheme error when the interpreter tries to add 1 to \( \text{procedure} \). For now, we will remain satisfied with this very base form of error-detection.

This interpreter is a useful model of a language like Fortran (or even Java), where the name of the function/method is always fixed. It does, however, prevent us from writing many programs we might find useful, or at least natural. Consider the Rip program

\[
\text{let} \{ x \ \{ \text{proc} \{ x \} \ x \} \}
\{ x \ 3 \}
\]

This seems perfectly reasonable to write, but our data definition disallows it. The \( x \) in the expression \( \{ x \ 3 \} \) is not a \( \text{proc}E \) as required by the data definition.

If we ignore our data definition and try this program in our interpreter anyway, what will happen? Using Rip syntax for convenience (as opposed to working with the data representations), the expression \( \{ x \ 3 \} \) will be transformed (by \( \text{subst} \)) to \( \{ \text{procedure} \ 3 \} \). This will fail in the \( \text{app}E \) clause of our interpreter.

Interestingly, if we use a lazy evaluator, the program will work! Note that this is the second time we have seen complications with eager and lazy evaluation (the first time was with free variables).

The big problem we are having is that procedures in Rip are not \textit{first-class}.

\textbf{Definition 1 (first-class)} A value is \textit{first-class} if it may be passed to and returned from functions and stored in data structures (adapted from Essentials of Programming Languages, by Friedman, et. al.).
4 Making Procedures First-Class

Procedures are not yet first-class in Rip since we can’t invoke our procedure values. The reason a program like

\[
\text{let } \{x \ {\text{proc } \{x\} \ x}\} \\
\text{x 3}
\]

doesn’t work in our eager interpreter is because the return value \$\text{PROCEDURE} gives us only information about the value (i.e., that it is a procedure). The information in the procedure is lost. What information might we need? Clearly, at least the procedure’s argument name and its body. But this is the same information already in every procE value. Therefore, let’s instead return the expression itself. That is, we update the procE clause to be

\[
\{(\text{procE} \? \text{expr}) \ \text{expr}\}
\]

Because a potentially complex expression can evaluate to a procE, we update the appE line of the data definition to be \(\text{(make-appE RP RP)}\). Previously, we had only allowed a procE in the appE-proc position. Now that general RP expressions are acceptable, procedure application where the appE-proc is not a procE will not work without a change.

The design recipe suggests that our template line looks like

\[
\{(\text{appE} \? \text{expr}) \ldots (\text{interp } \text{appE-proc} \text{expr}) \ldots (\text{interp } \text{appE-arg} \text{expr}) \ldots\}
\]

We know we need the second recursive call. Do we need the first? This depends on whether the expression in the procedure position can be complex. Indeed it can:

\[
\{\ {\text{let } \{x \ {\text{proc } \{y\} \ y}\}} \\
\text{x} \\
3\}
\]

Working at the syntactic level, our interpreter should reduce the let expression to \{proc \{y\} y\}, so that the entire program returns 3. In general, we need to interpret the expression in the procedure position of an application to reduce it (hopefully) to a procedural value. We update the appE clause of the interpreter to be

\[
\{(\text{appE} \? \text{expr}) \ (\text{interp } (\text{do-subst } (\text{interp } \text{appE-proc} \text{expr})) \ (\text{interp } \text{appE-arg} \text{expr})))\}
\]

where do-subst is defined as follows:

\[
\text{;; do-subst : procE × ?? \text{ → RP}} \\
\text{;; Produces a correctly substituted expression given the procedure and argument} \\
\text{;; (We will delay assigning a type to argV for now.)} \\
\text{(define do-subst proc argV)}
\]
(subst (procE-body proc) ;; the body of the procedure
 (procE-arg-name proc) ;; the name of the formal argument to the procedure
 argV)) ;; the value of the actual argument

(Conventional textbooks use the names “formal” and “actual” to refer to the procedure’s argument name and the expression that evaluates to the argument value.)

4.1 Acceptable Programs

We have a problem. The expression \( \{1 \ 2\} \) is now a Rip program according to our data definition, but it is nonsense code. Fixing this is tricky. Later on in the course we will explore ways of weeding out inappropriate programs. For now, let’s generally assume we are given correct ones. If a user does, accidentally or maliciously, slip us a buggy program, the underlying Scheme implementation will catch the error eventually. For instance, in the program \( \{1 \ 2\} \), our interpreter will treat 1 as a procE struct. This will signal a Scheme error.

5 Cleaning-Up Interp

5.1 The Return Value of Interp

We’ve been playing fast and loose with the return type of interp. What kinds of things do we return from interp? With a little study, it becomes clear that interp always returns an RP, but this does not capture the important point that an interpreter reduces expressions to values. The values we return are only numEs or procEs, not general RPs. Therefore we write the following data definition:

A value is either

- \((\text{make-numE} \text{ number})\)
- \((\text{make-procE} \text{ symbol RP})\)

Now, our contract for interp is

\[\text{;; interp : RP } \rightarrow \text{ value}\]
\[\text{;; interp reduces RP expressions to values}\]

Similarly, our contract for do-subst is now

\[\text{;; do-subst : procE } \times \text{ value } \rightarrow \text{ RP}\]
\[\text{;; Produces a correctly substituted expression given the procedure and argument}\]

5.2 Let

Consider let. In words, let evaluates an expression, binds it to a variable, and then evaluates another expression in the context of this binding. That same description
fits procedure application: Once it has reduced the procedure position to a procedural value, application evaluates an expression and hands it to the procedure, which binds it to a variable and then evaluates another expression in the context of this binding. This suggests that we should be able to replace `let` with a combination of procedures and their application. For example,

\[
\text{(let \{x 1\} (+ x 2))}
\]

is equivalent to

\[
\text{(\{proc \{x\} (+ x 2)\} 1)}
\]

This raises the question, which form is more fundamental? Clearly, having both procedures and application is more flexible, because it gives the programmer the choice of deciding when to perform the actions of procedure creation and procedure application; `let` bundles these two actions together and doesn’t provide an easy way of separating them. Therefore, we want to transform `let` into application. Still, you may want to keep `let` in your interpreters, because it makes some source programs easier to write. For example, it’s a lot easier and clearer to write

\[
\text{(let \{x 5\}
  \text{let \{y 4\}
    \text{let \{z 3\}
      \{+ \{+ x y\} z\}\})})
\]

than

\[
\text{(\{proc \{x\}
  \{proc \{y\}
    \{proc \{z\}
      \{+ \{+ x y\} z\}
    3\}
  4\}
5)}
\]