1 Extending our Interpreter

Recall our interpreter from last time.

\[
\begin{align*}
(\text{define} & \ (\text{interp} \ \text{an-rp}) \\
& \quad (\text{cond}) \\
& \quad \quad \quad [(\text{numE?} \ \text{an-rp}) \ (\text{numE-num} \ \text{an-rp})] \\
& \quad \quad \quad [(\text{addE?} \ \text{an-rp}) \ (+ \ (\text{interp} \ (\text{addE-left} \ \text{an-rp}))) \\
& \quad \quad \quad \quad (\text{interp} \ (\text{addE-right} \ \text{an-rp})))]] \\
& \quad \quad \quad [(\text{multE?} \ \text{an-rp}) \ (* \ (\text{interp} \ (\text{multE-left} \ \text{an-rp}))) \\
& \quad \quad \quad \quad (\text{interp} \ (\text{multE-right} \ \text{an-rp})))]]
\end{align*}
\]

With this interpreter (and the parser we wrote), we can interpret a program like:

\[
\text{f} \ \text{let} \ \text{f} \ x \ 3 \ \text{g} \ x \ \text{g}) \ 3
\]

\[
\text{f} \ \text{let} \ \text{f} \ x \ 4 \ \text{g} \ x \ \text{g}) \ 4
\]

\[
\text{f} \ \text{let} \ \{x \ {+ \ 1} \ 2\} \ x \ {+ \ 1} \ 2
\]

It seems kind of silly to interpret \{+ 1 2\} three times. This would be really silly if the repeated expression were complicated. The programmer should only have to write the complicated expression once. We can factor out this common code using a \textbf{let} construct, which we’ll now add to Rip.

1.1 A Test Suite for let

We will write some examples of \textbf{let} expressions in Rip:

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

\[
\text{let} \ \{x \ 3\} \ 4 \Rightarrow 4
\]

\[
\text{let} \ \{x \ {+ \ 1} \ 2\} \ x \Rightarrow 3
\]
\{\text{let} \{x \text{ 3}\} \\
\text{let} \{y \text{ 2}\} \\
\{\ast x y\}\}\}

⇒ 5

\{\text{let} \{y \text{ 3}\} \\
\text{let} \{x \ x\ x\}\}

⇒ 3

\{\text{let} \{x \text{ 3}\} \\
\text{let} \{x \text{ 5}\} x\}\}

⇒ 5

\section{Intuition to Precision}

Now that we have some intuition about what’s going on with \texttt{let}, let’s try to expose precisely how it works. We’ll start with some definitions.

\textbf{Definition 1 (binding instance of a variable)} A variable is a binding instance if it appears in the binding position of a \texttt{let} expression.

\textbf{Definition 2 (scope)} The scope of a binding instance is the text within the \texttt{let} expression in which references to the variable refer to the binding instance (adapted from Friedman, et. al., Essentials of Programming Languages).

\textbf{Definition 3 (bound instance of a variable)} A variable is bound if it is contained in the scope of a binding instance for it.

For example, in the Scheme expression \texttt{f \{let \{x \text{ 3}\} \{\ast x \text{ 1}\}\} g x g}, the first \texttt{x} is a binding instance with scope \{\ast x \text{ 1}\}, and the second \texttt{x} is a bound instance.

\subsection{Data Definitions}

We start by trying to find a good data definition for a variable.

A variable is

\begin{itemize}
  \item a symbol
\end{itemize}

If we imagine that we add symbols back into the data definition of our language, then there is a problem with this simple data definition for variable. Consider the expression \texttt{let \{x \text{ 3}\} x}. The data representation of this program would be the same as the program \texttt{let \{x \text{ 3}\} ’x}. Since we will need symbols later on, we make the following changes:

A bound instance of a variable is

\begin{itemize}
  \item a symbol
\end{itemize}
which implies the struct definition \((\text{define-struct} \ varE \ (\text{name}))\).

A \textbf{binding instance} of a variable is still

\begin{verbatim}
   \text{a symbol}
\end{verbatim}

As an example of this new type of data, the above syntax \(\text{let} \ \{x \ 3\} \ x\) is
written as data with \((\text{make-letE} \ 'x \ (\text{make-numE} \ 3) \ (\text{make-varE} \ 'x))\).

\subsection{2.2 Our Current RP Data Definition}

We have to add \texttt{let} expressions to our RP data definition. An RP is either

\begin{itemize}
  \item \((\text{make-numE} \ \text{number})\)
  \item \((\text{make-addE} \ RP \ RP)\)
  \item \((\text{make-multE} \ RP \ RP)\)
  \item \((\text{make-letE} \ \text{symbol} \ RP \ RP)\)
\end{itemize}

which implies the struct definition \((\text{define-struct} \ letE \ (\text{var-name} \ \text{val} \ \text{body}))\).

\subsection{2.3 Updating our Interpreter}

We have to add a line to account for \texttt{let} expressions.

\begin{verbatim}
(define \(\text{interp} \ \text{an-rp}\))
  \text{(cond}
    \((\text{numE?} \ \text{an-rp}) \ \text{an-rp})
    \((\text{addE?} \ \text{an-rp}) \ (\text{numE}+ \ (\text{interp} \ \text{addE-left} \ \text{an-rp}))
        \ (\text{interp} \ \text{addE-right} \ \text{an-rp})))
    \((\text{multE?} \ \text{an-rp}) \ (\text{numE}* \ (\text{interp} \ \text{multE-left} \ \text{an-rp}))
        \ (\text{interp} \ \text{multE-right} \ \text{an-rp})))
    \((\text{letE?} \ \text{an-rp}) \ \ldots \ (\text{interp} \ \text{letE-var-name} \ \text{an-rp})
        \ \ldots \ (\text{interp} \ \text{letE-val} \ \text{an-rp})
        \ \ldots \ (\text{interp} \ \text{letE-body} \ \text{an-rp}) \ \ldots \)\)
  \text{)}
\end{verbatim}

We’ve used the template to help us with some of what to fill in. It’s not immediately clear, however, what else needs to be done. Basically, we want to \textit{substitute} the bind-exp for the variable, wherever it occurs in the body. Let’s develop a rule for this substitution by first studying Scheme substitution.
3 Substitution

**Rule 1 (Substitution)** Given a let expression `(make-letE var-name val body)`, replace (in the body) every occurrence of `(make-varE name)` with `val`.

This works on

```scheme
(let {x 3}
  (+ x 1))
```

but fails on

```scheme
(let {x 3}
  (let {x 4}
    x))
```

It gives the answer 3 instead of 4.

**Rule 2 (Substitution)** Given a let expression `(make-letE var-name bind-exp body)`, replace (in the body) every occurrence of `(make-varE name)` with `bind-exp`, unless you hit another let expression binding the same variable name.

This works on

```scheme
(let {x 3}
  (let {x 4}
    x))
```

but fails on

```scheme
(let {x 3}
  (+ (let {y 5} x)
    (let {x x} x)))
```

Here we are supposed to substitute 3 for the second `x` in `let {x x} x`. Our rule, however, tells us to ignore this, and we get an undefined variable `x` error.

**Rule 3 (Substitution)** Given a let expression `(make-letE var-name bind-exp body)`, replace (in the body) every occurrence of `(make-varE name)` with `bind-exp`, unless you hit another let expression binding the same variable name. But if you hit another let expression binding the same variable name, still recursively substitute into the `bind-exp` of that sub-let expression.

Stop! This is ridiculous! We can’t use English to describe something like substitution. We have two precise languages at our disposal which are more appropriate for describing substitution—math and Scheme. Since we all know Scheme, we’ll use that.

So let’s write the function `subst`. 

---

4
3.1 Contract and Header for Subst

;; subst : RP × symbol × RP → RP
;; subst consumes a body, a variable name to substitute, and the expression to substitute in. It returns a new RP that is substituted correctly.

(define (subst body var bind-exp) ...)

3.2 Examples

(subst (make-addE (make-numE 1)
                 (make-varE 'x))
     'x
     (make-numE 3))

should return

(make-addE (make-numE 1) (make-numE 3))

Of course, all of the tricky examples we came up with should work too.

3.3 The Body of Subst

(define (subst body var val)
  (cond
   [(numE? body) body]
   [(addE? body) (make-addE (subst (addE-left body) var val)
                          (subst (addE-right body) var val))]
   [(varE? body)
    (cond
     [(eq? var (varE-name body)) val]
      [else body])]
   [(letE? body)
    (cond
     [(eq? var (letE-var-name body))
      (make-letE var
                 (make-letE body)
                 var val
                 (letE-body body))]
      [else (make-letE (letE-var-name body)
                        (subst (letE-val body) var val)
                        (letE-body body))])])

4 Back to Interp

We now can finish up the definition of interp:

(define (interp an-rp)
(*begin natural_text*)

(\begin{verbatim}
(\textbf{cond}
  \begin{enumerate}
  \item \[(\text{numE? an-rp}) (\text{numE-num an-rp})]\]
  \item \[(\text{addE? an-rp}) (+ (\text{interp (addE-left an-rp)})
    (\text{interp (addE-right an-rp)}))]\]
  \item \[(\text{multE? an-rp}) (* (\text{interp (multE-left an-rp)})
    (\text{interp (multE-right an-rp)}))]\]
  \item \[(\text{letE? an-rp}) (\text{interp
    subst (letE-body an-rp)
    (letE-var-name an-rp)
    (interp (letE-val an-rp))))\])\]
  \end{enumerate}
\end{verbatim}

\textbf{Definition 4 (free variable)} A free variable is a variable expression which has no corresponding binding instance. When free variables are evaluated in Scheme, they produce undefined identifier errors.

Consider, for example:

\begin{verbatim}
{let {x {+ y 1}
   {let {y 2}
    x}}
\end{verbatim}

The first $y$ in this expression is free. Our procedure \textit{interp} gives the correct result on this Rip program—an error saying that $y$ is undefined. Without the recursive call to \textit{interp} in the last line of the procedure \textit{interp}, the program would incorrectly return 3.

\section*{4.1 Eager and Lazy Evaluation}

Eager evaluation is a model of evaluation in which you evaluate expressions before substituting. The recursive call to \textit{interp} in the last line of \textit{interp} makes the interpreter eager.

In a lazy evaluator, you substitute first, and evaluate expressions only when you are forced to. You might think that removing the \textit{interp} call from (\textit{interp \textit{(letE-val an-rp)}) will result in a lazy interpreter. This is \textit{basically} true, but it turns out to not be that simple. We will get to this later on.

\section*{5 Oh yeah, and what about that weird example from class?}

Recall the example

\begin{verbatim}
(let ([x x]) x)
\end{verbatim}

It should now be pretty clear what the correct result is for this expression. Tracing through our interpreter, we see that the \textit{val} (which is $x$ here) is interpreted first, which produces (correctly) an undefined variable $x$ error.