1 Recursion

Since our interpreter does not support letrec or define, it’s not trivial to do recursion. We can let-bind a variable to a function that takes itself as an argument, then pass in the original function every time we want to call it:

\[
\begin{align*}
\text{let } (\! \! (\text{lambda } (n) \\
\quad (\text{if } (\text{zero? } n) 1 (* \ n \ (\! \! (\! \! \text{sub1 } n))))))) \\
\quad ((\! !) 5))
\end{align*}
\]

It is fairly easy to implement let in our interpreter, but if we would prefer not to, we can always rewrite let in terms of lambda:

\[
\begin{align*}
\text{let } ((\lambda \! \! (\! \! \text{sub1 } n)))))) \]
\end{align*}
\]

We would prefer not to see the self-application of \(!\). We can move it into another function, called fix, or the Y combinator:

\[
\begin{align*}
\text{let } (\! \! (\text{lambda } (n) \\
\quad (\text{if } (\text{zero? } n) 1 (* \ n \ ((\! !) (\text{sub1 } n))))))) \\
\quad ((\! !) 5))
\end{align*}
\]

In Scheme, however, this doesn’t work so well. Because Scheme is a call-by-value language, this code will go into an infinite loop. To fix the problem, we perform a transformation called “inverse-\(\eta\)”. This version of fix will work in Scheme, and we can write factorial:

\[
\begin{align*}
\text{let } ((\lambda \! \! (\text{lambda } (f) \\
\quad ((\text{lambda } (x) (f (x x))) \\
\quad (\text{lambda } (x) (f (x x))))))))) \\
\quad ((\! !) (\text{sub1 } n)))))) \\
\quad ((\! !) 5))))
\end{align*}
\]
Inverse-\(\eta\) (pronounced "inverse eta") is a transformation that can be used to delay evaluation of an expression. In general, this transformation looks like:

\[
M \Rightarrow (\text{lambda} \ (x) \ (M \ x))
\]

Inverse-\(\eta\) is, surprisingly enough, the inverse of the \(\eta\) transformation, which has the following general form:

\[
(\text{lambda} \ (x) \ (M \ x)) \Rightarrow M
\]

The \(\eta\) transformation works provided that \(x\) does not occur free in \(M\), \(M\) does not diverge, and \(M\) takes exactly one argument. For example, the \(\lambda\)-expression:

\[
(\text{lambda} \ (x) \ (\text{add1} \ x))
\]

Can be rewritten simply as \(\text{add1}\) by the \(\eta\) rule.

## 2 letrec

Scheme’s built-in method for doing recursion, \texttt{letrec}, actually does not use \texttt{fix} at all. Instead, it uses a \texttt{let} to bind the name of the function to some arbitrary value, then uses \texttt{set!} to set the original name to the value of the right-hand-side expression. An example like:

\[
(\texttt{letrec} \ ([\ [\text{!} \ (\text{lambda} \ (n))] \ (\text{if} \ (\text{zero?} \ n) \ 1 \ (\text{!} \ (\text{sub1} \ n))))])
\]

\[
(\text{!} \ 5)
\]

Would become:

\[
(\texttt{let} \ ([\ ! \ \text{hukars}]])
\]

\[
(\text{set!} \ ! \ (\text{lambda} \ (n))
\]

\[
(\text{if} \ (\text{zero?} \ n) \ 1 \ (\text{!} \ (\text{sub1} \ n))))
\]

\[
(\text{!} \ 5)
\]

We can write a macro that expands \texttt{letrec} expressions the same way Scheme does:

\[
(\texttt{define-syntax letrec}
\]

\[
(\texttt{syntax-rules} ()
\]

\[
([\ _, \ ([\text{lhs} \ \text{rhs}] \ldots) \ \text{body} \ldots])
\]

\[
(\texttt{let} \ ([\text{lhs} \ \text{hukars}] \ldots)
\]

\[
(\texttt{set!} \ \text{lhs} \ \text{rhs} \ldots \ \text{body} \ldots))
\]

## 3 The Interpreter

We can implement a recursive \texttt{letrec} in our interpreter in yet another way. We introduce a new type of environment, the recursive environment, to hold the code for a recursive function until we are ready to commit to an environment for its closure. In the \texttt{letrec} line, we extend the environment with the binding for the new function before evaluating the function in the new environment. Our \texttt{apply-env} changes as follows:
(define apply-env
  (lambda (env s)
    (pmatch env
      [(empty) () (error ...)]
      [(extend-env ,id ,val ,env) () (if (eq? id s) val
                                 (apply-env env s))]
      [(extend-env-recursively ,name ,code ,env^) ()
                                 (make-closure name code env)]))))

This way, the function’s name gets added to the environment before the function is evaluated, so the environment in which the function is closed contains a binding for the function itself.

4 More Combinators

Combinators are higher-order functions with no free variables. Some of them have been given names; some commonly-used combinators are $S$, $K$, and $I$:

\[
S = (\text{lambda} (f) (\text{lambda} (g) (\text{lambda} (x) ((f x) (g x)))))
\]

\[
K = (\text{lambda} (x) (\text{lambda} (y) x))
\]

\[
I = (\text{lambda} (x) x)
\]

In fact, it is possible to write any program using only these three combinators. This may seem difficult to believe—especially because there are no variables—but it is true. The abstraction algorithm is an algorithm for converting programs written in the $\lambda$-calculus into programs using just these three combinators. Programs written this way are, as you might expect, impossible to read.

5 Fewer Combinators

It turns out that $I$ can be written in terms of $S$ and $K$:

\[
I = ((SK)K)
\]

In fact, it is possible to write both $S$ and $K$ in terms of another combinator. One such combinator is $X$. It is defined such that:

\[
((XX)X) = K
\]

\[
(X(XX)) = S
\]

Thus, we can write all programs with a single letter. However, we still need parentheses. Chris Okasaki’s paper explains two combinators, $P$ and $A$, that allow writing programs without parentheses. $P$ pushes the $X$ combinator onto a stack, and $A$ applies the first item on the stack to the second, popping both off and pushing the result back on. Thus, it is possible to write programs without parentheses, using only two characters.
6 If

In λ-calculus programs and in programs written using combinators, there is no such thing as if. Instead, if is emulated by applying the test to the two possible results: ((test then) other). If we define true to be (λ(x) (λ(y) x)), then ((true 1) 2) will indeed return 1. However, because Scheme is a call-by-value language, we can run into trouble when emulating if this way. Consider the following example:

(((true 1) ((λ(x) (x x)) (λ(x) (x x))))

The result should obviously be 1. In Scheme, however, this example produces an infinite loop. Even though the second argument to the test is never used, Scheme evaluates it before passing its value along. To solve this problem, we can use the inverse-η technique we saw earlier:

(((true (λ(a) (1 a))) (λ(a) ((λ(x) (x x)) (λ(x) (x x)))))))

Obviously, this code would create an error in Scheme, as 1 is not a procedure that we can apply. Remember, however, that every value in the λ-calculus is a λ-term—including numbers. Using the Church numeral representation of 1, (λ(f) (λ(x) (f x))), our code will run perfectly, and avoid the infinite loop.

Scheme requires a special form to implement if correctly. Because Scheme is a call-by-value language, the arguments to a function are evaluated before they are passed to the function. In the case of if, this is incorrect behavior. In the example above, we solved this problem by using the inverse-η transformation to delay evaluation of the arguments until after evaluation of the test. In Scheme, if simply does not evaluate the then and other expressions until after it has evaluated and considered the test expression.