Lab: Memoization
CSC/MAT 208: Discrete Structures
Department of Computer Science
Department of Mathematics and Statistics
Grinnell College

Pairs

Here are the programmer pairs for today’s lab. Drivers are on the left.

• Danielle Williams and Anita DeWitt
• Corey Simmonds and Thu Nguyen
• Kathryn Yetter and Dennis Chan
• Ella Nicolson and Muhammad Hamza
• Amanda Hinchman-Dominguez and Julia Fay
• Adam Wesely and Tyler Williams
• Devin Dooley and Adam Hudson
• Samee Zahid and Jong Hoon Bae
• Jacob Ekstrand and Lindsey Byrne
• Elizabeth Zak and Matt Jasaitis
• Michael Owusu and Kevin Connors
• Marcel Champagne and David Chang
• Linda Oyolu and Madeleine Goldman

Exercises

In this lab, we’ll study a technique for storing values of a function as they are computed and reusing them instead of recomputing them when recomputation would lead to gross inefficiencies.

Procedures with Static Storage

In purely functional programming, the procedures that we write invariably return the same values when given the same arguments. For many applications, including most procedures for counting combinatorial objects, this is a useful property that makes our procedures easier to test, easier to maintain, and easier to prove correct.
However, Scheme allows a kind of procedure that retains some information between calls, in locations that can be modified through side effects during the execution of the procedure. These procedures, which are said to have static storage, can return different values on different calls.

Here is a simple example: a procedure that takes no arguments and returns the number of times it has been invoked.

(define invocation-count
  (let ((counter 0))
    (lambda ()
      (set! counter (+ counter 1))
      counter)))

Normally, the body of a procedure definition is a lambda-expression. In a procedure with static storage, however, the body is typically a let-expression. The local variables introduced and initialized in the binding list are the names of the static storage locations. Although these storage locations can be accessed only from inside the body of the let-expression, they retain their contents between invocations of the procedure being defined (unlike the procedure’s parameters, if any, and unlike any bindings introduced in let-expressions inside the lambda-expression, which are all discarded as soon as control leaves their scope).

In the definition of invocation-count, the identifier counter is bound to a storage location and initialized to 0 at the time the definition is processed, before the procedure has been invoked even once. When it is invoked, only the body of the lambda-expression is executed; the enclosing let-expression is never again evaluated.

The body of the lambda-expression first increments the value of counter. The set!-expression replaces the contents of a storage location with a new value obtained by evaluating an expression — in this case, the expression (+ counter 1). The set!-expression does not return any useful value, but is executed only for its side effect. The procedure then returns the new value of the variable counter.

Exercise 0: Copy the definition of invocation-count into the Definitions window in DrRacket, click Run, and invoke the procedure several times until you’re convinced that it works.

Exercise 1: Define and test a procedure called running-totaler that takes one argument (a real number), adds it to a running total, and returns
that running total. The running total should be held in static storage and should be initialized to 0.

**How Memoization Works**

One way of defining the `choose` procedure for computing binomial coefficients is to build directly on the recursion equations

\[
\binom{n}{0} = 1, \\
\binom{n}{n} = 1, \\
\binom{n+1}{k+1} = \binom{n}{k} + \binom{n}{k+1}
\]

to get the definition

\[
\text{(define choose}
\begin{align*}
\text{(lambda (n k))}
\text{(cond ((zero? k) 1))} \\
\text{((= n k) 1))} \\
\text{((else (+ (choose (- n 1) (- k 1))}
\text{ (choose (- n 1) k))))))})
\end{align*}
\]

In many respects, this is an elegant and pleasingly direct definition. However, it has a serious efficiency problem, even for fairly small values of \(n\) and \(k\). For example, you might try to use this procedure to solve the stock combinatorics problem of how many different bridge hands (thirteen-card combinations) can be drawn from the standard fifty-two card bridge deck. It takes a long time to get the answer back, basically because the the recursion has to run all the way down to the base cases every time, so that the numbers that are ultimately being added together are all 1s. The number of calls reaching one of the base cases is thus equal to the computed binomial coefficient (in this case, 635013559600), and the number of non-base-case recursive calls is only one less.

What makes this particularly annoying is that all of these calls involve only 560 different pairs of arguments — basically, \(\langle x + y, x \rangle\) for all choices of \(x\) from 0 to 39 and all choices of \(y\) from 0 to 13. Almost all of the effort of computation goes into computing the same values over and over again.
To speed this up, we can arrange to store the value of each call to \texttt{choose} in an association list before returning it. We can then rewrite \texttt{choose} so that it examines this association list before starting into its computation. If \texttt{choose} determines that it has been called before with the same arguments, it can look up the previously computed value in the association list instead of computing it again. The association list can be held in static storage.

Here’s the revised procedure:

\begin{verbatim}
(define choose
 (let ((table '()))
 (lambda (n k)
  (let ((stored-result (assoc (list n k) table)))
    (if (pair? stored-result)
    (cdr stored-result)
    (let ((computed-result
          (cond ((zero? k) 1)
                ((= n k) 1)
                (else (+ (choose (- n 1) (- k 1))
                       (choose (- n 1) k))))))
      (set! table
        (cons (cons (list n k) computed-result)
              table))
      computed-result))))
\end{verbatim}

The \texttt{assoc} procedure, which searches through an association list for a pair that has a specified car, is provided in the \texttt{(rnrs lists)} library. It returns the matching pair if it finds one, or \#f if it does not.

\textbf{Exercise 2:} In the Australian card game Five Hundred, ten cards are dealt to each player from a deck of forty-three cards. How many different Five Hundred hands are there?

\textbf{Exercise 3:} Build a similarly memoized version of the \texttt{stirling2} function, which computes Stirling numbers of the second kind, using the recursion equations

\begin{align*}
S(n, n) &= 1, \\
S(n + 1, 0) &= 0, \\
S(n + 1, k + 1) &= S(n, k) + (k + 1) \cdot S(n, k + 1)
\end{align*}

\textbf{Exercise 4:} Rewrite the \texttt{sized-integer-partition-count} function (from the lab on Stirling numbers) using memoization.
Faster Tables

If you use memoization extensively, you may find cases in which looking up stored results becomes a bottleneck, because the association list gets so long that it is inefficient to search it. In such cases, we can make memoization even faster by using a data structure called a hash table in place of the association list. There is a standard library, (rnrs hashtables), that supports this data structure.

The expression (make-hashtable equal-hash equal?) creates and returns an empty hash table, ensuring that, during search, possible keys will be compared using the equal? procedure. You would use this constructor call, for example, to initialize the static storage for the memoized procedure.

The expression (hashtable-contains? table key) finds out whether a given hash table table contains a stored value associated with a particular key key. Its value is Boolean.

The expression (hashtable-ref table key #f) looks in table for a stored value associated with key and returns that stored value if it finds one, or #f if it does not.

The expression (hashtable-set! table key value) modifies table to associate value with key, either by adding a new association or by replacing a previously existing one (which is overwritten). It is executed only for its side effect and returns no useful value.

Exercise 5: Rewrite the memoized version of choose using hash tables instead of association lists.