Since this course is lab-based and programming-intensive, we’ll need to be quite familiar with the most commonly used features of the programming language that we all share, which is Scheme. We’ll use a subset of the language as specified by the Revised $^7$ Report on the Algorithmic Language Scheme.

Expressions

Scheme is an expression-based language, and every step in the execution of a Scheme program is the evaluation of an expression. Usually, the process of evaluation results in the computation of a single value, although Scheme also provides for cases in which evaluation results in more than one value, in no values, in an “unspecified” value, in the raising of an exception, or in some other redirection of the flow of control.

Most composite expressions in Scheme are constructed on the same pattern: a list of the (syntactic) constituents of the expression, separated by whitespace and enclosed in parentheses. I’ll use the term S-structure to refer to this pattern of construction.

Many of the constituents of Scheme expressions are identifiers. Identifiers can be used as names for storage locations (that is, as variables), as syntactic keywords, or as designators for values of Scheme’s “symbol” data type. Scheme allows identifiers to contain letters, digits, hyphens, underscores, and many other punctuation marks, provided that the initial character is not a digit, plus sign, hyphen, or full stop (but the three specific identifiers +, -, and ... are allowed as exceptions to this rule).

Here are some examples to illustrate the variety of possible identifiers: q, lambda, λ, soup, list->vector, a34kTNMs, ->-, fee-fie-fo-fum, $amount, tax%, *page-number*, foo.bar.baz, index/count, matrix@, and ~^~.

Our subset of Scheme includes expressions of seven general types: literals, variables, sequencing expressions, lambda-expressions, procedure calls, conditionals, and local binders.

(1) A literal is a constant, directly depicting its value. Scheme provides literals with special, idiosyncratic forms for Boolean values (#t and #f), numbers (numerals, such as 42, 22/7, and -1.98), Unicode characters (such as A for the Latin capital letter A), and character strings (such as "Hello, world!"). In addition, one can form a literal for a Scheme symbol by prefixing an apostrophe (pronounced “quote”) to the identifier that designates it (thus 'foo denotes the symbol foo).

The apostrophe actually indicates that the expression to which it is attached should not be evaluated, because it already depicts the result of the evaluation process. So one can use a quote to form a literal denoting a list by placing the elements of the list in an S-structure and prefixing a quote. For example, '(#t 42 #\A foo) is a literal denoting a list of four elements: the true Boolean value, the integer 42, the Latin capital letter A, and the symbol foo. As a somewhat special case, the literal '() denotes the empty list.

(2) A variable is the name of a (notional) storage location. The value of the variable is whatever is stored in the location that it names. Scheme allows any value to be stored in any location, without regard to its type. Variables have no intrinsic types of their own.

(3) A sequencing expression is an S-structure comprising the syntactic keyword begin and one or more subexpressions, which are collectively called the body of the begin-expression. When a
begin-expression is evaluated, each of its subexpressions is evaluated, in order, and whatever values the last subexpression yields become the values of the entire begin-expression. (The values of the other subexpressions are discarded, so normally those other subexpressions are evaluated only for their side effects.)

The subexpressions of a begin-expression are typically placed on separate lines and indented, thus:

```
(begin
  (display "The sum of 7 and 5 is ")
  (display (+ 7 5))
  (display ".")
  (newline))
```

4) A lambda-expression is an S-structure comprising the syntactic keyword lambda, a forms specification, and one or more subexpressions, collectively called the body. The value of a lambda-expression is always a procedure. The body of the lambda-expression is not immediately evaluated, but instead stored for possible subsequent evaluation, when and if the procedure is invoked. The forms specification contains zero or more variables (the parameters of the procedure) and defines the interface through which values supplied when the procedure is invoked are made available inside the body of the procedure.

For instance, consider the lambda-expression

```
(lambda (num)
  (* num num))
```

The forms specification is (num) and the body is (* num num).

A forms specification can have any of three forms. The most common is an S-structure comprising zero or more identifiers. In this case, when the procedure denoted by the lambda-expression is invoked, the caller must supply exactly one value for each of these identifiers, and these values are stored into newly allocated storage locations before the body of the procedure is evaluated. The lambda-expression shown above illustrates this kind of forms specification.

Alternatively, the forms specification can consist of a single variable (not enclosed in parentheses). In this case, when the procedure denoted by the lambda-expression is invoked, the caller may supply any number of values, which are collected into a list. This list is then placed in a newly allocated storage location bound to the variable before the body of the procedure is evaluated. For example:

```
(lambda arguments
  (/ (apply + arguments) (length arguments)))
```

The third kind of forms specification is a hybrid of these: an S-structure containing two or more variables, with a full stop (also set off by whitespace) between the last two variables. In this case, the caller supplies a value for each of the variables preceding the full stop and zero or more additional values. These additional values are collected into a list that becomes the value of the post-full-stop variable before the body of the procedure is evaluated. For example:

```
(lambda (first . rest)
  (display first)
  (for-each (lambda (arg)
        (display \\
space)
      (display arg))
    rest)
  (newline))
```
The subexpressions making up the body of a \texttt{lambda}-expression are typically placed on separate lines and indented, as shown above.

(5) A procedure call is an S-structure comprising one or more subexpressions. The evaluation of a procedure call begins with the evaluation of all of the subexpressions, in arbitrary order. The value of the first expression must be a procedure, which is invoked and given the values of the remaining expressions as arguments, as described above. After the parameter bindings are in place, the expressions that make up the body of the invoked procedure are evaluated, in order, as in a sequencing expression. The values of the last expression in the body of the procedure that is invoked become the values of the procedure call.

Note that a procedure call can have more than one value, if it occurs in a context where all of the values can be used—typically, in a binding specification within a \texttt{let-values} expression.

Procedure calls are normally written on single lines if possible; otherwise, line breaks are inserted between subexpressions.

Scheme’s standard libraries contain a few hundred variables that are, in effect, bound at the beginning of program execution to storage locations containing procedures provided by the creators of the implementation. Calls to these built-in procedures have the same general form as calls to user-defined procedures, and (from the programmer’s point of view) they are invoked in the same way. The built-in procedures of the base language are described in chapter 6 of the Revised\textsuperscript{7} Report on the Algorithmic Language Scheme. Here are the ones that will be particularly important in this course:

- Equivalence predicates: \texttt{eq?}, \texttt{eqv?}, \texttt{equal?}, \texttt{=}.
- Arithmetic operations on exact values: \texttt{+}, \texttt{-}, \texttt{*}, \texttt{quotient}, \texttt{remainder}, \texttt{modulo}, \texttt{abs}, \texttt{gcd}, \texttt{max}, \texttt{min}, \texttt{expt}.
- Arithmetic predicates: \texttt{<}, \texttt{>}, \texttt{<=}, \texttt{>=}, \texttt{zero?}, \texttt{negative?}, \texttt{positive?}, \texttt{odd?}, \texttt{even?}.
- The Boolean predicate \texttt{not}.
- List procedures: \texttt{cons}, \texttt{car}, \texttt{cdr}, \texttt{cadr}, \texttt{list}, \texttt{length}, \texttt{append}, \texttt{reverse}, \texttt{list-ref}.
- Character predicates: \texttt{char=?}, \texttt{char<?}, \texttt{char<=?}, \texttt{char>?}, \texttt{char>=?}.
- String procedures: \texttt{string}, \texttt{string-length}, \texttt{string-ref}, \texttt{substring}, \texttt{string-append}, \texttt{string-copy}.
- String predicates: \texttt{string=?}, \texttt{string<?}, \texttt{string<=?}, \texttt{string>?}, \texttt{string>=?}.
- Input-output and port procedures: \texttt{read-char}, \texttt{peek-char}, \texttt{read}, \texttt{read-line}, \texttt{eof-object?}, \texttt{display}, \texttt{newline}, \texttt{open-input-file}, \texttt{open-output-file}, \texttt{open-input-string}, \texttt{open-output-string}, \texttt{get-output-string}, \texttt{close-input-port}, \texttt{close-output-port}.
- Higher-order procedures: \texttt{map}, \texttt{apply}, \texttt{for-each}.
- Type classifiers: \texttt{boolean?}, \texttt{number?}, \texttt{integer?}, \texttt{exact?}, \texttt{char?}, \texttt{string?}, \texttt{symbol?}, \texttt{pair?}, \texttt{list?}, \texttt{null?}, \texttt{procedure?}.
- Type conversions: \texttt{string->number}, \texttt{number->string}, \texttt{string->list}, \texttt{list->string}, \texttt{string->symbol}, \texttt{symbol->string}, \texttt{char->integer}, \texttt{integer->char}.
- Programming aids: \texttt{error}.

The CSC 151 course introduced some additional procedures that were defined (in Scheme) by the instructors. I have collected most of these in a Scheme library called \texttt{(csc-151 collection)}. Here’s a list of its contents:

- Additional arithmetic operations on exact values: \texttt{decrement}, \texttt{double}, \texttt{increment}.
- Additional list procedures: \texttt{iota}, \texttt{index-of}, \texttt{take}, \texttt{drop}, \texttt{filter}, \texttt{reduce}, \texttt{reduce-left}, \texttt{reduce-right}, \texttt{sort}, \texttt{tally-all}, \texttt{take-random}.
- Additional input-output and port procedures: \texttt{file->chars}, \texttt{file->lines}, \texttt{file->words}, \texttt{read-word}, \texttt{read-until}, \texttt{skip-char}, \texttt{eof}.
• Additional higher-order procedures: all, any, left-section, l-s, right-section, r-s, section, comparator, o, negate, conjoin, disjoin.
• Additional character and string predicates: char-in-string?.
• Miscellaneous procedures and values: null, pi, gensym, random, exact, inexact.

(6) Scheme supports several kinds of conditional expressions. The most basic of these is the if-expression, an S-structure comprising the syntactic keyword if and three subexpressions (the test, the consequent, and the alternate).

In the evaluation of an if-expression, the test is evaluated first. This evaluation must yield a single value. Normally this value is one of the two Boolean values, the true one or the false one; if the test expression has a non-Boolean value, that value is said to be truish.

If the value of the test expression is the true Boolean value or any truish value, then the consequent is evaluated, and the resulting values are the values of the entire if-expression. If the value of the test is the false Boolean value, then the alternate is evaluated, and the resulting values are the values of the entire if-expression. Thus the values of an if-expression are specified by two cases that are distinguished by the value of the test expression.

Here’s a simple example of an if-expression:

(if (positive? residue)
  residue
  (+ residue modulus))

Sometimes one wants to choose conditionally among three or more cases, each selected by a separate condition. The cond-expression is designed for this situation. It is an S-structure comprising the syntactic keyword cond and one or more cond-clauses. Each cond-clause is itself an S-structure comprising one or more subexpressions, of which the first is the test and the others make up the action.

The evaluation of a cond-expression consists of the evaluation of the test in each cond-clause until one is found to have a value that is the true Boolean value or a truish value; along the way, the evaluation of each test must yield a single value. When such a test has been detected, the subexpressions in the action of the same cond-clause are evaluated, in order, as if they were the body of a sequencing expression, and the values of the last of those subexpressions are the values of the entire cond-expression. (If the action in the selected cond-clause does not have any subexpressions in it, then the value of the entire cond-expression is unspecified.)

If the value of each of the tests is the false Boolean value, then the value of the cond-expression is unspecified. To make it easier to avoid such cases, the last cond-clause in a cond-expression can contain the syntactic keyword else in place of a test. If such a cond-clause is reached during the evaluation of a cond-expression, the subexpressions making up its action are evaluated, and again the values of the last of these subexpressions are the values of the cond-expression.

Here’s an example of a cond-expression:

(cond ((>= score 95) 'A)
      ((>= score 88) 'B)
      ((>= score 80) 'C)
      ((>= score 70) 'D)
      (else 'F))

A third kind of conditional, the and-expression, is an S-structure comprising the syntactic keyword and and zero or more subexpressions. The evaluation of an and-expression consists in the evaluation of its subexpressions, in order, until either one of them is found to have the false Boolean value or all of them have been evaluated. In the former case, the value of the and-expression is also
the false Boolean value. In the latter case, the value of the last subexpression is the value of the entire and-expression. (If there are no subexpressions, the value of the and-expression is the true Boolean value.) Each of the evaluated subexpressions must have a single value.

For example:

```scheme
(and (integer? something)
    (not (negative? something))
    (exact? something))
```

A fourth kind of conditional, the or-expression, is a kind of dual of the and-expression. It is an S-structure comprising the syntactic keyword or and zero or more subexpressions. The evaluation of an or-expression consists in the evaluation of its subexpressions, in order, until either one of them is found to have a value that is not the false Boolean value or all of them have been evaluated. In the former case, the value of the entire or-expression is the value of the non-false subexpression. In the latter case, the value of the entire or-expression is the false Boolean value. Each of the evaluated subexpressions must have a single value.

For example:

```scheme
(or (null? ls)
    (null? (cdr ls))
    (null? (caddr ls)))
```

In conditional expressions, each subexpression after the first is typically placed on a separate line, and the subexpressions are indented so as to form a vertical column, as in the examples shown above.

(7) A local binder is an S-structure comprising one of the syntactic keywords let, let*, letrec, letrec*, let-values, and let*-values; a binding list; and a sequence of one or more subexpressions, similar to the body of a begin-expression. When a local binder is evaluated, a fresh storage location is bound to each of the variables in its binding list, and values obtained by evaluating subexpressions within the binding list are placed in those storage locations; then the subexpressions in the body are evaluated in order, as in a sequencing expression, and the values of the last subexpression become the values of the entire local binder.

The various local binders differ mainly in the mechanics of how, and in what order, values are recovered from the subexpressions inside the binding list and placed in the storage locations to which the local variables are bound.

In a let-expression, the binding list is an S-structure comprising zero or more binding specifications. Each binding specification is also an S-structure, this one comprising a variable and a subexpression called an initializer. The evaluation of the let-expression begins with the evaluation of all of the initializers, in unspecified order (so that none of them can presuppose or depend on any of the others), after which the value of each initializer is placed in the storage location bound to the corresponding variable. Then evaluation proceeds with the body of the let-expression.

Here’s an example of a let-expression:

```scheme
(let ((numerator (* 35 scale))
    (denominator (* 10 back-scale)))
    (/ numerator denominator))
```

A let*-expression contains a binding list with the same syntactic structure, but processes it differently. The binding specifications are taken up in order. As each one is reached, its initializer is evaluated and the result is placed in the storage location associated with its variable. This binding now becomes available during the evaluation of subsequent initializers. The effect is that each
binding specification creates a new scope containing the subsequent binding specifications as well as the body of the let*-expression. (In a let-expression, by contrast, there is only one new scope, and it contains only the body of the let-expression.)

For example:

(let* ((width (* font-size (string-length str)))
       (height (* width
                    (apply max (map vertical (string->list str))))))
  (draw-background-box! start-x start-y width height bg-color)
  (draw-string! start-x start-y str text-color))

A letrec-expression again contains a binding list with the same syntactic structure. This time, however, all of the new variables are bound to storage locations first, before any of the initializers are evaluated; then the initializers are evaluated, in unspecified order, in a scope that includes the new bindings. At first glance, this might seem pointless, because values are placed into the newly allocated storage locations only after all of the initializers have been evaluated. However, the initializers can be lambda-expressions, and the newly bound variables can occur in the bodies of those lambda-expressions. This makes it possible to have local names for recursive procedures.

For example:

(letrec ((sum (lambda (ls)
                 (if (null? ls)
                     0
                     (+ (car ls) (sum (cdr ls))))))
        (sum '(3 6 9 12)))

The letrec*-expression combines the features of the let*- and letrec-expressions: As in a let*-expression, each binding specification in a letrec*-expression is dealt with separately, in order, and each creates a new scope that contains subsequent binding specifications as well as the body. As in a letrec-expression, a fresh storage location is bound to the variable first, and the evaluation of the initializer is performed in a scope that includes the new binding, so that the variable’s value can be a recursive procedure.

Each initializer in a let-, let*, letrec-, or letrec*-expression must have a single value.

In a let-values-expression, each binding specification contains a formals specification instead of just a variable. The initializer, which in this case is typically an expression that has multiple values, is evaluated and the resulting values are associated with the variables that occur in the formals specification, exactly as in the invocation of a procedure. The order in which the binding specifications are processed is unspecified, and there is only one new scope, containing just the body of the let-values-expression; all of the identifiers in the binding list are bound in this scope.

For example:

(let-values (((left-weight left-height) (explore (lt tr)))
              ((right-weight right-height) (explore (rt tr))))
  (values (+ left-weight right-weight (internal-weight tr))
          (+ (max left-height right-height) 1)))

The idea is that the explore procedure will return two values, and the two identifiers in each formals specification will be bound to storage locations containing those values within the body of the let-values-expression.

The let*-values-expression combines the features of let*- and let-values-expressions. As in a let-values-expression, each binding specification in a let*-values-expression contains a
formals specification and an initializer, typically one that has multiple values. As in a let*-expression, each binding specification in a let*values-expression is dealt with separately, in order, and each creates a new scope that contains subsequent binding specifications as well as the body.

There is one final variation on the let-expression, a local binder called the named let-expression. In this construction, a variable appears between the syntactic keyword let and the binding list. When the named let-expression is evaluated, a procedure value is constructed, with the variables from the binding list as its parameters and the body of the named let-expression as its body. This procedure value is then placed in a fresh storage location bound to the extra variable. Next, the binding list is processed, just as in any other let-expression. Finally, the body of the named let-expression is evaluated, and the value of the last expression in that body becomes the value of the entire named let-expression. The feature that makes named let-expressions so useful is that the scope of the binding for the extra variable includes the body of the named let-expression, so that the implicitly defined procedure can be recursive. This makes it straightforward to create single-use recursive procedures in-line.

(let loop ((rest ls) (so-far 0))
  (if (null? rest)
      so-far
      (loop (cdr rest) (+ so-far (car rest))))
)

In a local binder, the binding specifications after the first are typically placed on separate lines and indented to form a vertical column, and the subexpressions in the body are also placed on separate lines and indented uniformly. This convention is illustrated in the preceding examples.

Definitions

When a Scheme expression has a single value, the programmer can give that value a name by embedding the expression in a definition, which is an S-structure that comprises the keyword define, the name, and the expression. Programmers conventionally place a line break after the name and indent the expression slightly:

(define sum
  (lambda addends
    (apply + addends)))

When the definition is processed, the value of the expression is placed in a storage location to which the name is bound. (In effect, the name is a variable with the value of the expression as its value.)

When the expression in a definition is a lambda-expression, one can write an equivalent definition in a “short form” that fuses the name and the formals specification and omits the keyword lambda, thus:

(define (cube number)
  (* number number number))

(define (average . arguments)
  (/ (apply + arguments) (length arguments)))

(define (write-line first . rest)
  (display first)
  (for-each (lambda (arg)
Libraries

Definitions are typically collected into thematically related groups called libraries. Syntactically, a library is an S-structure that comprises the keyword \texttt{define-library}, a library specification, an export list, an import list, and finally a begin-expression containing a sequence of zero or more forms (definitions and expressions). Here’s a short example:

$$
\begin{align*}
\text{(define-library (discrete powers)} \\
&\text{(export square cube zenzizenzic sursolid)} \\
&\text{(import (scheme base))} \\
&\text{(begin)} \\
&\text{(define (cube number)} \\
&\text{(* number number number))} \\
&\text{(define (zenzizenzic number)} \\
&\text{(square (square number))}) \\
&\text{(define (sursolid number)} \\
&\text{(* (square number) (cube number)))))
\end{align*}
$$

The first line indicates that this library is called \texttt{powers} and is included in the \texttt{discrete} collection. It exports the identifiers \texttt{square}, \texttt{cube}, \texttt{zenzizenzic}, and \texttt{sursolid} making them available for use in Scheme programs and in other libraries. This library imports definitions from the \texttt{base} library in the \texttt{scheme} collection, which includes the procedures \texttt{*} and \texttt{square}. Note that it is possible for a library to export a procedure that it has simply imported from another library, without providing a separate definition of it.

It is a good idea to put each library in its own file. In many implementations, libraries are supposed to have a distinct filetype, such as \texttt{.sls} (“Scheme library source”), but DrRacket does not recognize this filetype, so in this course we’ll use \texttt{.ss} both for libraries and for top-level Scheme programs.

DrRacket expects us to follow several other conventions about naming files as well: The part of the filename that precedes the \texttt{.ss} filetype should match the last identifier in the library specification. For example, the file containing the library shown above should be named \texttt{powers.ss}. In addition, the collection to which this library belongs should correspond to a directory containing the libraries in that collection, which should have the same name as the collection (in this case, \texttt{discrete}).

A collection can also contain other collections, just as a directory can include other directories. The two structures should match up, so that (for instance) a library specified as \texttt{(discrete utilities lists)}, meaning that it is the \texttt{lists} library in the \texttt{utilities} subcollection within the \texttt{discrete} collection, might appear in the MathLAN file system as

$$
\text{/home/spelvin/MAT208/discrete/utilities/lists.ss}
$$

Thus a library specification is an S-structure comprising one or more Scheme identifiers, naming the collection or hierarchy of collections to which the library belongs, and ending with the identifier for the library proper.
An export list is an S-structure comprising the keyword `export` and the names of some or all of the items defined in the library, as those names appear in the definitions. These names, and the values they denote, can be imported by top-level programs and by other libraries, and used there as if they were predefined.

An import list is an S-structure comprising the keyword `import` and zero or more import sets. An import set names a set of bindings — names and the values associated with them — from another library and makes them available in this library, possibly renaming them along the way. Each import set has one of the following five forms:

- A library specification. In this case, all the bindings exported by the specified library are imported.
- An S-structure comprising the keyword `only`, a library specification, and zero or more identifiers. Each of the identifiers must be one of the names exported by the specified library. Only the bindings for those names are imported.
- An S-structure comprising the keyword `except`, a library specification, and zero or more identifiers. Each of the identifiers must be one of the names exported by the specified library. All of the bindings exported by the specified library, except the ones for the listed names, are imported.
- An S-structure comprising the keyword `prefix`, a library specification, and an identifier. All of the bindings exported by the specified library are imported, but the name of each one is changed to a name formed by concatenating the identifier in the import set with the exported name.
- An S-structure comprising the keyword `rename`, a library specification, and zero or more renaming specifications. Each renaming specification is an S-structure comprising two identifiers, the first of which must be the name of a binding exported by the specified library. The bindings associated with the names in the renaming specifications are imported, but the name of each one is replaced by the other identifier in the renaming specification.

Actually, it’s possible to nest import sets (that is, an import set can occur anywhere a library specification can occur in the preceding account), but there shouldn’t be any need for us to take advantage of the additional generality.

The `begin`-expression that follows the import list usually contains only definitions, but it may contain expressions as well. The expressions are evaluated when the library is imported, for side effects only (the values of the expressions are ignored). A typical use for such expressions is to initialize a complex data structure that is defined in the library.

**Program Structure**

A top-level program in Scheme begins with an import list, exactly like the import list in a library. Importing bindings from a library makes them available inside the program. A top-level program contains an import list and zero or more forms. The bindings created by the definitions, which are typically placed before the expressions, can be used in the expressions. The execution of the program consists of the evaluation of the expressions it contains, in order, for side effects only (again, the values of the expressions are ignored).

A top-level program can use any of the identifiers that it imports from other libraries, but it does not automatically get access to identifiers that those libraries themselves import — importation is not transitive in Scheme.

Programmers should bear in mind that even essential identifiers such as `define`, `if`, `cons`, and `+` have to be imported from a library such as `(scheme base)` before they can be used in a program. Under R7RS, it makes no sense to try to write a program without importing anything.
Design Patterns

A few patterns are frequently encountered in the design and implementation of short procedures in Scheme and are probably familiar to you from CSC 151.

A Scheme procedure that takes a natural number as one of its arguments often follows the pattern of recursion over natural numbers. In this pattern, the procedure begins by distinguishing the base case, in which the argument is zero and the value to be returned can be computed directly, from the recursive case, in which the argument is positive and the value is computed by issuing a recursive call and modifying or transforming the result of that recursive call in some way.

A typical example is make-list, which takes two arguments, of which the first (len) is a natural number and the second (fill) can be any Scheme value. The make-list procedure constructs and returns a list of length len in which each element is fill. It could be defined as follows:

\[
\text{(define (make-list len fill)}
  \text{(if (zero? len)}
    \text{'()}
    \text{(cons fill (make-list (- len 1)))))}
\]

(The Revised\textsuperscript{7} Report on the Algorithmic Language Scheme provides a more general version of make-list as a built-in procedure.)

A Scheme procedure that takes a list as one of its arguments often follows the pattern of recursion over lists. In this pattern, the procedure begins by distinguishing the base case, in which the argument is the empty list and the value to be returned can be computed directly, from the recursive case, in which the argument is a non-empty list that can be separated into a car and a cdr, and the value is computed by applying the procedure recursively to the cdr of the given list and modifying or transforming the result of that recursive call in some way, often involving the car.

For example, the partition-by-parity procedure takes one argument, a list of exact integers, and returns two values: a list of the even integers from the given list and a list of the odd integers from the given list. It could be defined as follows:

\[
\text{(define (partition-by-parity ls)}
  \text{(if (null? ls)}
    \text{(values '() '())}
    \text{(let-values (((cdr-evens cdr-odds) (partition-by-parity (cdr ls)))))}
    \text{(if (even? (car ls))}
      \text{(values (cons (car ls) cdr-evens) cdr-odds)}
      \text{(values cdr-evens (cons (car ls) cdr-odds)))})
\]

In defining recursive procedures in Scheme, one frequently finds that the best way to write the recursive kernel of the procedure does not match the best external interface (or the one that is prescribed or demanded by the designers or users of a larger software system of which the procedure is a component). One common way to solve this problem is to use a helper procedure.

Here’s a simple example of the problem. We want to implement a procedure—let’s call it adjacent-sums—that takes as its argument a list of two or more real numbers and returns a list of the real numbers that are the sums of two adjacent elements of the given list. For example:

\[
\begin{align*}
> \text{(adjacent-sums (list 1 2 3 4 5 6))} \\
& (3 5 7 9 11) \\
> \text{(adjacent-sums (list -12 12 -38 46 -8 42 17))} \\
& (0 -26 8 38 34 59) \\
> \text{(adjacent-sums (list 5 3))} \\
& (8)
\end{align*}
\]
The recursive kernel for this procedure requires two arguments: (a) the most recently encountered element of the list and (b) the part of the list that has not yet been processed. This kernel can go into a helper procedure, like this:

```
(define (adjacent-sums-helper elm rest)
  (let* ((next-elm (car rest))
         (new-sum (+ elm next-elm)))
    (if (null? (cdr rest))
      (list new-sum)
      (cons new-sum (adjacent-sums-helper next-elm (cdr rest))))))
```

To provide the one-argument interface shown in the sample interactions above, the outer procedure can simply invoke the helper, splitting the list it is given into the pieces required for the launch:

```
(define (adjacent-sums ls)
  (adjacent-sums-helper (car ls) (cdr ls)))
```

I call this pattern “husk-and-kernel programming,” with the interface procedure as the protective husk and enables the kernel to do its job most effectively without having to accommodate directly to the demands of the outside world.

Using either a named-let-expression (or a letrec-expression), one can actually pack the recursive kernel inside the husk in a concise and readable way:

```
(define (adjacent-sums ls)
  (let helper ((elm (car ls))
               (rest (cdr ls)))
    (let* ((next-elm (car rest))
           (new-sum (+ elm (car rest))))
      (if (null? (cdr rest))
        (list new-sum)
        (cons new-sum (helper next-elm (cdr rest))))))
```

**Tail Call Optimization**

Another common reason for using the husk-and-kernel pattern is to take advantage of tail call optimization, which is the technical term for a specific performance guarantee that Scheme provides to programmers.

In most programming languages, including C and Java, developers are sometimes reluctant to use recursion, even when it is clearly appropriate, because of a concern about the way recursive procedure calls use storage in a computer’s random-access memory. In those languages, every procedure call causes the run-time system to allocate a chunk of storage called an activation record, to keep track of the values of the parameters of the procedure and its local variables. The activation record is freed and recycled once the procedure call has finished executing and its value has been returned to the caller.

If a recursive procedure calls itself repeatedly until the base case is reached, an activation record is allocated for each call, but none of those activation records can be freed until the base case returns a value. It is possible for a long sequence of recursive calls to exhaust all of the memory that the run-time system has set aside for activation records. In that event, the program will crash.

If a recursive procedure calls itself repeatedly until the base case is reached, an activation record is allocated for each call, but none of those activation records can be freed until the base case returns a value. It is possible for a long sequence of recursive calls to exhaust all of the memory that the run-time system has set aside for activation records. In that event, the program will crash.

In many situations, however, a procedure A doesn’t actually do any computation after it receives the result of a call to a procedure B but instead simply returns the result the call to B as its own result. (That’s a “tail call” — the call to B comes at the tail end of the execution of A.) A recursive procedure, in particular, can often be written in such a way that any recursive call that it makes is the very last step in its execution.
When a Scheme processor is executing procedure \( A \) and encounters a call to procedure \( B \) that it recognizes as a tail call, it does not allocate a new activation record for the call to \( B \), but instead reuses the activation record for \( A \). After all, \( A \) has no more computation to do! The only thing \( A \) has left to do is to return the value that \( B \) gives it to its own caller. Under tail call optimization, \( B \) bypasses the middle man. It takes over \( A \)'s old activation record and arranges to return the value it computes directly to \( A \)'s caller.

As a result, recursion through tail calls doesn’t use any storage at all in Scheme. Scheme programmers who are concerned about the prospect of running out of memory for activation records can usually arrange their recursive procedures so that they use tail calls.

For example, all of the recursive procedures that we have examined so far in this section can be rewritten to use tail calls. The process often involves adding a parameter or two (“accumulators”) to the kernel procedure to transmit the results of the computation at each level of recursion to the next level (since no further computation can be performed after a tail call returns). Here’s how the preceding procedures would look when rewritten to use tail calls:

```scheme
(define (make-list len fill)
  (let kernel ((remaining len)
               (so-far '()))
    (if (zero? remaining)
      so-far
      (kernel (- remaining 1) (cons fill so-far))))

(define (partition-by-parity ls)
  (let kernel ((rest ls)
               (evens '())
               (odds '()))
    (if (null? rest)
      (values (reverse evens) (reverse odds))
      (if (even? (car rest))
        (kernel (cdr rest) (cons (car rest) evens) odds)
        (kernel (cdr rest) evens (cons (car rest) odds))))))

(define (adjacent-sums ls)
  (let kernel ((elm (car ls))
               (rest (cdr ls))
               (so-far '()))
    (let ((new-so-far (cons (+ elm (car rest)) so-far)))
      (if (null? (cdr rest))
        (reverse new-so-far)
        (kernel (car rest) (cdr rest) new-so-far)))))
```

Since all of the computations now have to precede the recursive call, the order in which the elements of a list are constructed and prepended to the ultimate result in these versions of the procedures is the reverse of the order in which they were prepended in the straight recursions above, which did not use tail calls. In \( \text{make-list} \), this does not affect the result, but in the other two definitions we need to invoke \texttt{reverse} before completing the base case in order to restore the intuitively correct element order.