1.4 Exercises

1.4.1 LEARNING ABOUT THE LANGUAGE

The first six exercises define some number-theoretic functions for you to program. Try to use recursion; it will be good practice for Chapter 2.

1. `(sigma` $m\ n$`)` $= m + (m+1) + \cdots + n$.

2. `(exp` $m\ n$`)` $= m^n$ $(m,\ n \geq 0)$. `(log` $m\ n$`)` = the least integer l such that $m^{l+1} > n$ $(m > 1,\ n > 0)$.

3. `(choose` $n\ k$`)` is the number of ways of selecting k items from a collection of n items, without repetitions, n and k nonnegative integers. This quantity is called a *binomial coefficient*, and is notated $\binom{n}{k}$. It can be defined as $\frac{n!}{k!(n-k)!}$, but the following identities are more helpful computationally: $\binom{n}{0} = 1$ $(n \geq 0)$, $\binom{n}{n} = 1$ $(n \geq 0)$, and $\binom{n}{k} = \binom{n-1}{k} + \binom{n-1}{k-1}$ $(n, k > 0)$.

4. `(fib` m`)` is the mth Fibonacci number. The Fibonacci numbers are defined by the identities: `(fib 0)` $= 0$, `(fib 1)` $= 1$, and for $m > 1$, `(fib` m`)` $=$ `(fib` $m-1$`)+(fib` $m-2$`)`.

5. `(prime` n`)` = true (1) if n is prime, false (0) otherwise. `(nthprime` n`)` = the nth prime number. `(sumprimes` n`)` = the sum of the first n primes. `(relprime` $m\ n$`)` = true if m and n are relatively prime (have no common divisors except 1), false otherwise.

2.7 Exercises

2.7.1 LEARNING ABOUT THE LANGUAGE

1. Code the following LISP functions:

   (a) `(count x l)` counts the number of occurrences of `x` at the top level of `l`, and `(countall x l)` counts the number of occurrences throughout `l`.

   ```
   -> (count 'a '(1 b a (c a)))
   1
   -> (countall 'a '(1 b a (c a)))
   2
   ```

(b) `(reverse l)` returns a list containing the elements of `l` in reverse order. (*Hint:* you will probably want to use the **append** function.)

```
-> (reverse '(a b (c d) e))
(e (c d) b a)
```

(c) `(twist l)` reverses the top level of `l` and recursively twists all the items in `l`.

```
-> (twist '((a (b 5)) (c d) e))
(e (d c) ((5 b) a))
```

(d) `(flatten l)` constructs a list having the same atoms as `l` in the same order but in a flat list.

```
-> (flatten '((a b) ((c d) e)))
(a b c d e)
```

(e) `(sublist l1 l2)` and `(contig-sublist l1 l2)` determine whether the elements of l1 are contained, and contiguously contained, respectively, in the same order in `l2`.

```
-> (sublist '(a b c) '(x a y b z c))
T
-> (contig-sublist '(a b c) '(x a y b z c))
()
-> (contig-sublist '(a y) '(x a y b z c))
T
```

2. Program the following set functions:

   (a) `(remove x s)` returns a set having the same elements as set `s` with element `x` removed.

   (b) `(subset s1 s2)` determines if `s1` is a subset of `s2`.

   (c) `(=set s1 s2)` determines if `s1` and `s2` are the same set.

3. Explain why `wrong-sum` (page 2.1.3) doesn't work. Then explain why it works when the last line is changed to:

   `(+ tmp (wrong-sum (cdr l))).`

4. This set of questions concerns tree traversal.

   (a) Program post-order and in-order traversal for binary trees.

   (b) Modify the pre-order and level-order traversals so that, instead of the node labels being printed, they are placed in a list, which is returned as the value of the call.

(c) Extend the pre-order and level-order traversals to trees of arbitrary degree, represented in such a way that binary trees have the same representation given them in the text. For example, `'(a b c d)` represents a ternary tree whose root is labeled with `a` and which has three children, labeled `b`, `c`, and `d`, respectively, all leaf nodes. Note that there are two ways to represent leaf nodes: `'a` and `'(a)` both represent a leaf node labeled `a`.

5. These problems relate to the relational data base example.

   (a) Program `AND-SELECT`, whose first two arguments are lists (of the same length) and which selects only those rows that have all the given values for the given attributes. For example, `(AND-SELECT '(Crime Location) '(murder London) CRIMES)` would select only rows representing murders in London.

   (b) Program `OR-SELECT`, whose first argument is an attribute name, whose second argument is a list of values, and which selects those rows which have any of the values for the given attribute.

   (c) Lift the restriction on `UNION`, `INTER`, and `DIFF` that the attributes of their two arguments must occur *in the same order*. These operations should check that their arguments have the same set of attributes and then choose an order of those attributes for the result.

   (d) `REMOVE` has the same arguments as `PROJECT`, but projects onto those attribute *not* in its first argument.

6. Modify the last version of `eval` in Section 2.4 as follows:

   (a) Add `begin` and `print`.

   (b) Add `set` and global variables.

   (c) Add local variables, as described in Exercise 12 of Chapter 1.

4.10 Exercises

4.10.1 LEARNING ABOUT THE LANGUAGE

1. Use `mapcar`, `mapc`, or `combine` to define the following functions (those not defined here were defined in Chapter 2):

   (a) `cdr*` takes the `cdr`'s of each element of a list of lists:

   ```
   -> (cdr* '((a b c) (d e) (f)))
   ((b c) (e) ())
   ```

   (b) `max*` finds the maximum of a list of nonnegative integers.

   (c) `append`.

   (d) `addtoend` adds its first argument (an arbitrary S-expression) as the last element of its second argument (a list):

   ```
   -> (addtoend 'a '(b c d))
   (b c d a)
   ```

(e) **reverse** (use **addtoend**).

(f) **insertion-sort** (you may take **insert** as given).

(g) **(mkpairfn x)** is a function which, given a list, places **x** in front of each element:

```
-> ((mkpairfn 'a) '(() (b c) (d) ((e f))))
((a) (a b c) (a d) (a (e f)))
```

2. **lex-order*** extends **lex-order** to apply to lists of differing length, as in normal alphabetical ordering:

```
-> (set alpha-order (lex-order* <))

-> (alpha-order '(4 15 7) '(4 15 7 5))
T

-> (alpha-order '(4 15 7) '(4 15 6 6))
()
```

To relate this to normal alphabetical ordering, just translate the numbers to letters: These two results say DOG $<$ DOGE and DOG $\not<$ DOFF. Program **lex-order***.

3. An alternate representation for sets in SCHEME is as Boolean-valued functions, called "characteristic functions." In particular, the null set is represented by a function that always returns *nil*, and the membership test is just application:

```
-> (set nullset (lambda (x) '()))

-> (set member? (lambda (x s) (s x)))
```

(a) Program **addelt**, **union**, **inter**, and **diff** using this new representation.

(b) Code the third approach to polymorphism (page 106) using this representation.

6. Extend the equivalence given for **letrec** in Section 4.6 to allow for multiple recursive definitions, i.e., to allow for the general form of **letrec**, as in this version of **eval**, which evaluates expressions having operations + and *, of arbitrary arity:

```
(set eval (lambda (e)
     (letrec
        ((ev (lambda (exp)
               (if (number? exp) exp
                  (if (= (car exp) '+)
                     (+/ (evlis (cdr exp)))
                     (*/ (evlis (cdr exp)))))))
         (evlis (mapc ev)))
        (ev e))))
```

(a) Give the extension on paper.

(b) Translate **eval** by hand, and run it to test the translation.

(c) Write the SCHEME functions **trans-let**, **trans-letrec**, and **trans-let*** that do the translations of each type of expression into SCHEME; for example:

```
-> (trans-letrec '(letrec ((f e1)) e2))
((lambda (f) (begin (set f e1) e2)) (quote ()))
```

(d) Add **let** and **let*** to SCHEME **eval** (Section 4.5). (Note that adding **letrec** is much more difficult, because it requires implementing **set** in SCHEME **eval**.)