1 *Inductive Sets of Data*

1.1 Recursively Specified Data

Exercise 1.1 Write inductive definitions of the following sets. Write each definition in all three styles (top-down, bottom-up, and rules of inference). Using your rules, show the derivation of some sample elements of each set.

1. $\{3n+2 \mid n \in N\}$
2. $\{2n+3m+1 \mid n,m \in N\}$
3. $\{(n,2n+1) \mid n \in N\}$
4. $\{(n,n^2) \mid n \in N\}$ Do not mention squaring in your rules. As a hint, remember the equation $(n+1)^2 = n^2+2n+1$.

Solution:

note: $\Downarrow$ is **top-down**, $\Uparrow$ is **bottom-up**, $\Rightarrow$ is **rules of inference**.

1. $\Downarrow$ $x=2$, or $x-3 \in S$
   $\Uparrow$ $2 \in S$, and if $x \in S$, then $x+3 \in S$
   $\Rightarrow$ $2 \in S$ $\quad \dfrac{x \in S}{x+3 \in S}$

2. $\Downarrow$ $x=1$, either or both $[x-2 \in S, x-3 \in S]$
   $\Uparrow$ $1 \in S$, and if $x \in S$, either or both $[x+2 \in S, x+3 \in S]$
   $\Rightarrow$ $1 \in S$ $\quad \dfrac{x \in S}{\text{either or both}[x+2 \in S, x+3 \in S]}$

3. $\Downarrow$ $x=0, y=1$, or $(x-1,y-2) \in S$
   $\Uparrow$ $(0,1) \in S$, and if $(x,y) \in S$ then $(x+1,y+2) \in S$
   $\Rightarrow$ $(0,1) \in S$ $\quad \dfrac{(x,y) \in S}{(x+1,y+2) \in S}$

4. $\Downarrow$ $x=0, y=0$, or $(x-1,y-x) \in S$
   $\Uparrow$ $(0,0) \in S$, and if $(x,y) \in S$ then $(x+1,y+2x+1) \in S$
   $\Rightarrow$ $(0,0) \in S$ $\quad \dfrac{(x,y) \in S}{(x+1,y+2x+1) \in S}$

Exercise 1.2 What sets are defined by the following pairs of rules? Explain why.

1. $(0,1) \in S$ $\quad \dfrac{(n,k) \in S}{(n+1,k+7) \in S}$
2. $(0,1) \in S$ $\quad \dfrac{(n,k) \in S}{(n+1,2k) \in S}$
3. $(0,0,1) \in S$ $\quad \dfrac{(n,i,j) \in S}{(n+1,j,i+j) \in S}$
4. $(0,1,0) \in S$ $\quad \dfrac{(n,i,j) \in S}{(n+1,i+2,i+j) \in S}$

Solution:

1. $\{(n,7n+1) \mid n \in N\}$
2. $\{(n,2^n) \mid n \in N\}$
3. $\{(n,fib(n),fib(n+1)) \mid n \in N\}$
4. $\{(n,2n+1,n^2) \mid n \in N\}$

Exercise 1.3 Find a set T of natural numbers such that $0 \in T$, and whenever $n \in T$, then $n+3 \in T$, but $T \neq S$, where S is the set defined in definition 1.1.2.

Solution:

$T = N$

Exercise 1.4 Write a derivation from *List-of-Int* to `(-7 . (3 . (14 . ())))`.

Solution:

List-of-Int
⇒ `(Int . List-of-Int)`
⇒ `(-7 . List-of-Int)`
⇒ `(-7 . (Int . List-of-Int))`
⇒ `(-7 . (3 . List-of-Int))`
⇒ `(-7 . (3 . (Int . List-of-Int)))`
⇒ `(-7 . (3 . (14 . List-of-Int)))`
⇒ `(-7 . (3 . (14 . ())))`

Exercise 1.5 Prove that if $e \in LcExp$, then there are the same number of left and right parentheses in e.

Solution:

1. *Identifier* has 0 parenthes.

2. Other *LcExp* have matched pair parentheses.

1.2 Deriving Recursive Programs

Exercise 1.6 If we reversed the order of the tests in nth-element, what would go wrong?

Solution:

Jump over empty list check, `car` may operate a empty list.

Exercise 1.7 The error message from `nth-element` is uninformative. Rewrite `nth-element` so that it produces a more informative error message, such as "`(a b c)` does not have 8 elements."

Solution:

See source code file in "section-1.2/1.7.rkt".

Exercise 1.8 In the definition of `remove-first`, if the last line were replaced by `(remove-first s (cdr los))`, what function would the resulting procedure compute? Give the contract, including the usage statement, for the revised procedure.

Solution:

procedure: `(remove-first s ls )`
Symbol List
return: *List*
specification: Remove the first element same as `s` and before elements, return remained list.
define:

```
(define remove-first
  (lambda (s ls)
    (cond
      [(null? ls) '()]
      [(eq? s (car ls)) (cdr ls)]
      [else
       (remove-first s (cdr ls))])))
```

Exercise 1.9 Define `remove`, which is like `remove-first`, except that it removes all occurrences of a given symbol from a list of symbols, not just the first.

Solution:

See source code file in "section-1.2/1.9.rkt".

Exercise 1.10 We typically use "or" to mean "inclusive or." What other meanings can "or" have?

Solution:

either, otherwise.

Exercise 1.11 In the last line of `subst-in-s-exp`, the recursion is on `sexp` and not a smaller substructure. Why is the recursion guaranteed to halt?

Solution:

Main recursion in `subst`, whatever `subst-in-s-exp` call, they will halt in `subst`.

Exercise 1.12 Eliminate the one call to `subst-in-s-exp` in `subst` by replacing it by its definition and simplifying the resulting procedure. The result will be a version of `subst` that does not need `subst-in-s-exp`. This technique is called inlining, and is used by optimizing compilers.

Solution:

See source code file in "section-1.2/1.12.rkt".

Exercise 1.13 In our example, we began by eliminating the Kleene star in the grammar for `S-list`. Write `subst` following the original grammar by using `map`.

Solution:

See source code file in "section-1.2/1.13.rkt".

1.3 Auxiliary Procedures and Context Arguments

Exercise 1.14 Given the assumption $0 \leqslant n < length(v)$, prove that `partial-vector-sum` is correct.

Solution:

Initialization: Cause $0 \leqslant n <$ length(v), first add the nth-integer v_n of `v` to recusive chain. $(+\ v_n\ (\textit{partial-vector-sum}\ v\ (-\ n\ 1)))$

Maintenance:
$(+\ v_n\ (\textit{partial-vector-sum}\ v\ (-\ n\ 1)))$
$(+\ v_n\ (+\ v_{n-1}\ (\textit{partial-vector-sum}\ v\ (-\ n\ 2))))$
$(+\ v_n\ (+\ v_{n-1}\ (+\ v_{n-2}\ (\textit{partial-vector-sum}\ v\ (-\ n\ 3)))))$
$\vdots$

Termination: When $n = 0$, return v_0, get the result of recusive chain:
$(+\ v_n\ (+\ v_{n-1}\ (+\ v_{n-2}\ (+\ v_{n-3} \ldots\ v_0\ ))))$

1.4 Exercises

Exercise 1.15 `(duple n x)` returns a list containing n copies of `x`.

```
> (duple 2 3)
  (3 3)
> (duple 4 '(ha ha))
  ((ha ha) (ha ha) (ha ha) (ha ha))
> (duple 0 '(blah))
  ()
```

Solution:

See source code file in "section-1.4/1.15.rkt".

Exercise 1.16 `(invert lst)`, where lst is a list of 2-lists (lists of length two), returns a list with each 2-list reversed.

```
> (invert '((a 1) (a 2) (1 b) (2 b)))
  ((1 a) (2 a) (b 1) (b 2))
```

Solution:

See source code file in "section-1.4/1.16.rkt".

Exercise 1.17 `(down lst)` wraps parentheses around each top-level element of `lst`.

```
> (down '(1 2 3))
  ((1) (2) (3))
> (down '((a) (fine) (idea)))
  (((a)) ((fine)) ((idea)))
> (down '(a (more (complicated)) object))
  ((a) ((more (complicated))) (object))
```

Solution:

See source code file in "section-1.4/1.17.rkt".

Exercise 1.18 `(swapper s1 s2 slist)` returns a list the same as `slist`, but with all occurrences of `s1` replaced by `s2` and all occurrences of `s2` replaced by `s1`.

```
> (swapper 'a 'd '(a b c d))
  (d b c a)
> (swapper 'a 'd '(a d () c d))
  (d a () c a)
> (swapper 'x 'y '((x) y (z (x))))
  ((y) x (z (y)))
```

Solution:

See source code file in "section-1.4/1.18.rkt".

Exercise 1.19 `(list-set lst n x)` returns a list like `lst`, except that the n-th element, using zero-based indexing, is `x`.

```
> (list-set '(a b c d) 2 '(1 2))
  (a b (1 2) d)
> (list-ref (list-set '(a b c d) 3 '(1 5 10)) 3)
  (1 5 10)
```

Solution:

See source code file in "section-1.4/1.19.rkt".

Exercise 1.20 `(count-occurrences s slist)` returns the number of occurrences of `s` in `slist`.

```
> (count-occurrences 'x '((f x) y (((x z) x))))
  3
> (count-occurrences 'x '((f x) y (((x z) () x))))
  3
> (count-occurrences 'w '((f x) y (((x z) x))))
  0
```

Solution:

See source code file in "section-1.4/1.20.rkt".

Exercise 1.21 `(product sos1 sos2)`, where `sos1` and `sos2` are each a list of symbols without repetitions, returns a list of 2-lists that represents the Cartesian product of `sos1` and `sos2`. The 2-lists may appear in any order.

```
> (product '(a b c) '(x y))
  ((a x) (a y) (b x) (b y) (c x) (c y))
```

Solution:

See source code file in "section-1.4/1.21.rkt".

Exercise 1.22 `(filter-in pred lst)` returns the list of those elements in `lst` that satisfy the predicate pred.

```
> (filter-in number? '(a 2 (1 3) b 7))
  (2 7)
> (filter-in symbol? '(a (b c) 17 foo))
  (a foo)
```

Solution:

See source code file in "section-1.4/1.22.rkt".

Exercise 1.23 `(list-index pred lst)` returns the 0-based position of the first element of `lst` that satisfies the predicate pred. If no element of lst satisfies the predicate, then `list-index` returns `#f`.

```
> (list-index number? '(a 2 (1 3) b 7))
  1
> (list-index symbol? '(a (b c) 17 foo))
  0
> (list-index symbol? '(1 2 (a b) 3))
  #f
```

Solution:

See source code file in "section-1.4/1.23.rkt".

Exercise 1.24 `(every? pred lst)` returns `#f` if any element of `lst` fails to satisfy `pred`, and returns `#t` otherwise.

```
> (every? number? '(a b c 3 e))
  #f
> (every? number? '(1 2 3 5 4))
  #t
```

Solution:

See source code file in "section-1.4/1.24.rkt".

Exercise 1.25 `(exists? pred lst)` returns `#t` if any element of `lst` satisfies `pred`, and returns `#f` otherwise.

```
> (exists? number? '(a b c 3 e))
  #t
> (exists? number? '(a b c d e))
  #f
```

Solution:

See source code file in "section-1.4/1.25.rkt".

Exercise 1.26 `(up lst)` removes a pair of parentheses from each top-level element of `lst`. If a top-level element is not a list, it is included in the result, as is. The value of `(up (down lst))` is equivalent to `lst`, but `(down (up lst))` is not necessarily lst. (See exercise 1.17.)

```
> (up '((1 2) (3 4)))
  (1 2 3 4)
> (up '((x (y)) z))
  (x (y) z)
```

Solution:

See source code file in "section-1.4/1.26.rkt".

Exercise 1.27 `(flatten slist)` returns a list of the symbols contained in `slist` in the order in which they occur when `slist` is printed. Intuitively, `flatten` removes all the inner parentheses from its argument.

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

Solution:

See source code file in "section-1.4/1.27.rkt".

Exercise 1.28 `(merge loi1 loi2)`, where `loi1` and `loi2` are lists of integers that are sorted in ascending order, returns a sorted list of all the integers in `loi1` and `loi2`.

```
> (merge '(1 4) '(1 2 8))
  (1 1 2 4 8)
> (merge '(35 62 81 90 91) '(3 83 85 90))
  (3 35 62 81 83 85 90 90 91)
```

Solution:

See source code file in "section-1.4/1.28.rkt".

Exercise 1.29 `(sort loi)` returns a list of the elements of `loi` in ascending order.

```
> (sort '(8 2 5 2 3))
  (2 2 3 5 8)
```

Solution:

See source code file in "section-1.4/1.29.rkt".

Exercise 1.30 `(sort/predicate pred loi)` returns a list of elements sorted by the predicate.

```
> (sort/predicate < '(8 2 5 2 3))
  (2 2 3 5 8)
> (sort/predicate > '(8 2 5 2 3))
  (8 5 3 2 2)
```

Solution:

See source code file in "section-1.4/1.30.rkt".

Exercise 1.31 Write the following procedures for calculating on a bintree (definition 1.1.7): `leaf` and `interior-node`, which build bintrees, `leaf?`, which tests whether a bintree is a leaf, and `lson`, `rson`, and `contents-of`, which extract the components of a node. `contents-of` should work on both leaves and interior nodes.

Solution:

See source code file in "section-1.4/1.31.rkt".

Exercise 1.32 Write a procedure `double-tree` that takes a bintree, as represented in definition 1.1.7, and produces another bintree like the original, but with all the integers in the leaves doubled.

Solution:

See source code file in "section-1.4/1.32.rkt".

Exercise 1.33 Write a procedure `mark-leaves-with-red-depth` that takes a bintree (definition 1.1.7), and produces a bintree of the same shape as the original, except that in the new tree, each leaf contains the integer of nodes between it and the root that contain the symbol `red`. For example, the expression

```
(mark-leaves-with-red-depth
 (interior-node 'red
                (interior-node 'bar
                               (leaf 1)
                               (leaf 1))
                (interior-node 'red
                               (leaf 2)
                               (interior-node 'quux
                                              (leaf 2)
                                              (leaf 2)))))
```

which is written using the procedures defined in exercise 1.31, should return the bintree

```
(red (bar 1 1)
     (red 2
          (quux 2 2)))
```

Solution:

See source code file in "section-1.4/1.33.rkt".

Exercise 1.34 Write a procedure `path` that takes an integer `n` and a binary search tree `bst` (page 10) that contains the integer n, and returns a list of `lefts` and `rights` showing how to find the node containing `n`. If `n` is found at the root, it returns the empty list.

```
(path 17 '(14 (7 () (12 () ()))
              (26 (20 (17 () ())
                      ())
                  (31 () ()))))
(right left left)
```

Solution:

See source code file in "section-1.4/1.34.rkt".

Exercise 1.35 Write a procedure `number-leaves` that takes a bintree, and produces a bintree like the original, except the contents of the leaves are numbered starting from 0. For example,

```
(number-leaves
 (interior-node 'foo
                (interior-node 'bar
                               (leaf 26)
                               (leaf 12))
                (interior-node 'baz
                               (leaf 11)
                               (interior-node 'quux
                                              (leaf 117)
                                              (leaf 14)))
```

should return

```
(foo (bar 0 1)
     (baz 2
          (quux 3 4)))
```

Solution:

See source code file in "section-1.4/1.35.rkt".

Exercise 1.36 Write a procedure `g` such that `number-elements` from page 23 could be defined as

```
(define number-elements
  (lambda (lst)
    (if (null? lst) '()
        (g (list 0 (car lst)) (number-elements (cdr lst))))))
```

Solution:

See source code file in "section-1.4/1.36.rkt".

2 *Data Abstraction*

2.1 Specifying Data via Interfaces

Exercise 2.1 Implement the four required operations for bigits. Then use your implementation to calculate the factorial of 10. How does the execution time vary as this argument changes? How does the execution time vary as the base changes? Explain why.

Solution:

See source code file in "section-2.1/2.1.rkt".

Exercise 2.2 Analyze each of these proposed representations critically. To what extent do they succeed or fail in satisfying the specification of the data type?

Solution:

They can be safely extented to *Integer*, may be fail in *Real*, however, if allow add some other characters, it will succeed in *Real*, also *Complex*.

Exercise 2.3 Define a representation of all the integers (negative and nonnegative) as diff-trees, where a diff-tree is a list defined by the grammar

Diff-tree ::= `(one)` | `(diff` *Diff-tree Diff-tree*`)`

The list `(one)` represents 1. If t_1 represents n_1 and t_2 represents n_2, then `(diff` t_1 t_2`)` is a representation of $n_1 - n_2$.

So both `(one)` and `(diff (one) (diff (one) (one)))` are representations of 1; `(diff (diff (one) (one)) (one))` is a representation of -1.

1. Show that every number has infinitely many representations in this system.
2. Turn this representation of the integers into an implementation by writing `zero`, `is-zero?`, `successor`, and `predecessor`, as specified on page 32, except that now the negative integers are also represented. Your procedures should take as input any of the multiple legal representations of an integer in this scheme. For example, if your `successor` procedure is given any of the infinitely many legal representations of 1, it should produce one of the legal representations of 2. It is permissible for different legal representations of 1 to yield different legal representations of 2.

3. Write a procedure `diff-tree-plus` that does addition in this representation. Your procedure should be optimized for the diff-tree representation, and should do its work in a constant amount of time (independent of the size of its inputs). In particular, it should not be recursive.

Solution:

1. 0, 1 have infinitely many representations, so other integers have infinitely representations.
2. See source code file in "section-2.1/2.3.rkt".
3. See source code file in "section-2.1/2.3.rkt".

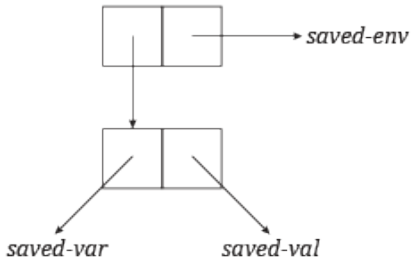
2.2 Representation Strategies for Data Types

Exercise 2.4 Consider the data type of *stacks* of values, with an interface consisting of the procedures `empty-stack`, `push`, `pop`, `top`, and `empty-stack?`. Write a specification for these operations in the style of the example above. Which operations are constructors and which are observers?

Solution:

`(empty-stack)`	= $\lceil\emptyset\rceil$	*constructor*
`(push val` $\lceil S\rceil$`)`	= push `val` into $\lceil S\rceil$	*constructor*
`(pop` $\lceil S\rceil$`)`	= pop top value from $\lceil S\rceil$	*constructor* and *observer*
`(top` $\lceil S\rceil$`)`	= get top value from $\lceil S\rceil$	*observer*
`(empty-stack?` $\lceil S\rceil$`)`	= $\lceil S\rceil =? \lceil\emptyset\rceil$	*observer*

Exercise 2.5 We can use any data structure for representing environments, if we can distinguish empty environments from non-empty ones, and in which one can extract the pieces of a non-empty environment. Implement environments using a representation in which the empty environment is represented as the empty list, and in which extend-env builds an environment that looks like



This is called an a-list or association-list representation.

Solution:

See source code file in "section-2.2/2.5.rkt".

Exercise 2.6 Invent at least three different representations of the environment interface and implement them.

Solution:

See source code file in "section-2.2/2.6.rkt".

Exercise 2.7 Rewrite `apply-env` in figure 2.1 to give a more informative error message.

Solution:

See source code file in "section-2.2/2.7.rkt".

Exercise 2.8 Add to the environment interface an observer called `empty-env?` and implement it using the a-list representation.

Solution:

See source code file in "section-2.2/2.5.rkt".

Exercise 2.9 Add to the environment interface an observer called `has-binding?` that takes an environment env and a variable s and tests to see if s has an associated value in env. Implement it using the a-list representation.

Solution:

See source code file in "section-2.2/2.9.rkt".

Exercise 2.10 Add to the environment interface a constructor `extend-env*`, and implement it using the a-list representation. This constructor takes a list of variables, a list of values of the same length, and an environment, and is specified by

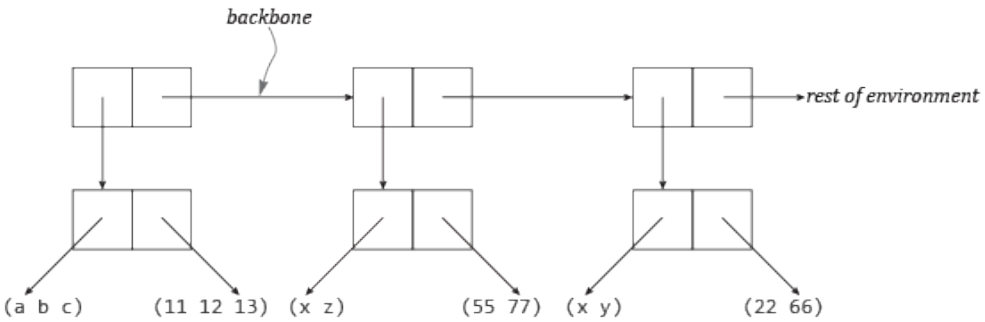
$$\texttt{(extend-env*}\ \texttt{(}var_1\ \ldots\ var_k\texttt{)}\ \texttt{(}val_1\ \ldots\ val_k\texttt{)}\ \lceil f \rceil = \lceil g \rceil,$$

$$\text{where } g(var) = \begin{cases} val_1 & \text{if } var = var_i \text{ for some } i \text{ such tha } 1 \leqslant i \leqslant k \\ f(var) & \text{otherwise} \end{cases}$$

Solution:

See source code file in "section-2.2/2.5.rkt".

Exercise 2.11 A naive implementation of `extend-env*` from the preceding exercise requires time proportional to k to run. It is possible to represent environments so that `extend-env*` requires only constant time: represent the empty environment by the empty list, and represent a non-empty environment by the data structure. Such an environment might look like



This is called the *ribcage* representation. The environment is represented as a list of pairs called *ribs*; each left rib is a list of variables and each right rib is the corresponding list of values.

Implement the environment interface, including `extend-env*`, in this representation.

Solution:

See source code file in "section-2.2/2.11.rkt".

Exercise 2.12 Implement the stack data type of exercise 2.4 using a procedural representation.

Solution:

See source code file in "section-2.2/2.12.rkt".

Exercise 2.13 Extend the procedural representation to implement `empty-env?` by representing the environment by a list of two procedures: one that returns the value associated with a variable, as before, and one that returns whether or not the environment is empty.

Solution:

See source code file in "section-2.2/2.13.rkt".

Exercise 2.14 Extend the representation of the preceding exercise to include a third procedure that implements `has-binding?` (see exercise 2.9).

Solution:

See source code file in "section-2.2/2.13.rkt".

2.3 Interfaces for Recursive Data Types

Exercise 2.15 Implement the lambda-calculus expression interface for the representation specified by the grammar above.

Solution:

See source code file in "section-2.3/2.15.rkt".

Exercise 2.16 Modify the implementation to use a representation in which there are no parentheses around the bound variable in a lambda expression.

Solution:

See source code file in "section-2.3/2.16.rkt".

Exercise 2.17 Invent at least two other representations of the data type of lambda-calculus expressions and implement them.

Solution:

Too lazy to write code.

1. Defining custom datatype, maybe this is the best for dealing with many different terms.
2. Using list, just like the book doing in this section.
3. Procedure, possible, but it is hard to extend.

Exercise 2.18 We usually represent a sequence of values as a list. In this representation, it is easy to move from one element in a sequence to the next, but it is hard to move from one element to the preceding one without the help of context arguments. Implement non-empty bidirectional sequences of integers, as suggested by the grammar

$$\textit{NodeInSequence} \quad ::= \quad (\textit{Int}\ \textit{Listof}(\textit{Int}) \quad \textit{Listof}(\textit{Int}))$$

The first list of numbers is the elements of the sequence preceding the current one, in reverse order, and the second list is the elements of the sequence after the current one. For example, `(6 (5 4 3 2 1) (7 8 9))` represents the list `(1 2 3 4 5 6 7 8 9)`, with the focus on the element 6.

In this representation, implement the procedure `number->sequence`, which takes a number and produces a sequence consisting of exactly that number. Also implement `current-element`, `move-to-left`, `move-to-right`, `insert-to-left`, `insert-to-right`, `at-left-end?`, and `at-right-end?`.

For example:

```
> (number->sequence 7)
  (7 () ())
> (current-element '(6 (5 4 3 2 1) (7 8 9)))
  6
> (move-to-left '(6 (5 4 3 2 1) (7 8 9)))
  (5 (4 3 2 1) (6 7 8 9))
> (move-to-right '(6 (5 4 3 2 1) (7 8 9)))
  (7 (6 5 4 3 2 1) (8 9))
> (insert-to-left 13 '(6 (5 4 3 2 1) (7 8 9)))
  (6 (13 5 4 3 2 1) (7 8 9))
> (insert-to-right 13 '(6 (5 4 3 2 1) (7 8 9)))
  (6 (5 4 3 2 1) (13 7 8 9))
```

The procedure `move-to-right` should fail if its argument is at the right end of the sequence, and the procedure `move-to-left` should fail if its argument is at the left end of the sequence.

Solution:

See source code file in "section-2.3/2.18.rkt".

Exercise 2.19 A binary tree with empty leaves and with interior nodes labeledwith integers could be represented using the grammar

$$\textit{Bintree} ::= () \mid (\textit{Int}\ \textit{Bintree}\ \textit{Bintree})$$

In this representation, implement the procedure `number->bintree`, which takes a number and produces a binary tree consisting of a single node containing that number. Also implement `current-element`, `move-to-left-son`, `move-to-rightson`, `at-leaf?`, `insert-to-left`, and `insert-to-right`. For example,

```
> (number->bintree 13)
  (13 () ())
> (define t1 (insert-to-right 14
               (insert-to-left 12
                 (number->bintree 13))))
> (at-leaf? (move-to-right (move-to-left t1)))
  #t
> (insert-to-left 15 t1)
  (13
    (15
      (12 () ())
      ())
    (14 () ()))
```

Solution:

See source code file in "section-2.3/2.19.rkt".

Exercise 2.20 In the representation of binary trees in exercise 2.19 it is easy to move from a parent node to one of its sons, but it is impossible to move from a son to its parent without the help of context arguments. Extend the representation of lists in exercise 2.18 to represent nodes in a binary tree. As a hint, consider representing the portion of the tree above the current node by a reversed list, as in exercise 2.18.

In this representation, implement the procedures from exercise 2.19. Also implement `move-up`, `at-root?`, and `at-leaf?`.

Solution:

See source code file in "section-2.3/2.20.rkt".

2.4 A Tool for Defining Recursive Data Types

Exercise 2.21 Implement the data type of environments, as in section 2.2.2, using `define-datatype`. Then include `has-binding?` of exercise 2.9.

Solution:

See source code file in "section-2.4/2.21.rkt".

Exercise 2.22 Using `define-datatype`, implement the stack data type of exercise 2.4.

Solution:

See source code file in "section-2.4/2.22.rkt".

Exercise 2.23 The definition of `lc-exp` ignores the condition in definition 1.1.8 that says "*Identifier* is any symbol other than `lambda`." Modify the definition of `identifier?` to capture this condition. As a hint, remember that any predicate can be used in `define-datatype`, even ones you define.

Solution:

See source code file in "section-2.4/2.23.rkt".

Exercise 2.24 Here is a definition of binary trees using `define-datatype`.

```
(define-datatype bintree bintree?
  (leaf-node
   (num integer?))
  (interior-node
   (key symbol?)
   (left bintree?)
   (right bintree?)))
```

Implement a `bintree-to-list` procedure for binary trees, so that `(bintree-to-list (interior-node 'a (leaf-node 3) (leaf-node 4)))` returns the list

```
(interior-node
 a
 (leaf-node 3)
 (leaf-node 4))
```

Solution:

See source code file in "section-2.4/2.24.rkt".

Exercise 2.25 Use `cases` to write `max-interior`, which takes a binary tree of integers (as in the preceding exercise) with at least one interior node and returns the symbol associated with an interior node with a maximal leaf sum.

```
> (define tree-1
    (interior-node 'foo (leaf-node 2) (leaf-node 3)))
> (define tree-2
    (interior-node 'bar (leaf-node -1) tree-1))
> (define tree-3
    (interior-node 'baz tree-2 (leaf-node 1)))
> (max-interior tree-2)
  foo
  (max-interior tree-3)
  baz
```

The last invocation of `max-interior` might also have returned `foo`, since both the `foo` and `baz` nodes have a leaf sum of 5.

Solution:

See source code file in "section-2.4/2.24.rkt".

Exercise 2.26 Here is another version of exercise 1.33. Consider a set of trees given by the following grammar:

$$\begin{aligned} \textit{Red-blue-tree} &::= \textit{Red-blue-subtree} \\ \textit{Red-blue-subtree} &::= \texttt{(red-node } \textit{Red-blue-subtree} \quad \textit{Red-blue-subtree}\texttt{)} \\ &::= \texttt{(blue-node } \{\textit{Red-blue-subtree}\}^*\texttt{)} \\ &::= \texttt{(leaf-node } \textit{Int}\texttt{)} \end{aligned}$$

Write an equivalent definition using `define-datatype`, and use the resulting interface to write a procedure that takes a tree and builds a tree of the same shape, except that each leaf node is replaced by a leaf node that contains the number of red nodes on the path between it and the root.

Solution:

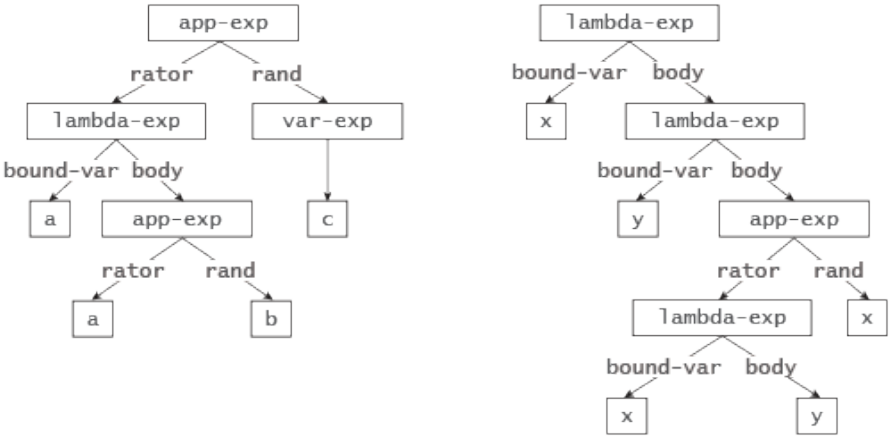
See source code file in "section-2.4/2.26.rkt".

2.5 Abstract Syntax and Its Representation

Exercise 2.27 Draw the abstract syntax tree for the lambda calculus expressions

```
((lambda (a) (a b)) c)
(lambda (x)
  (lambda (y)
    ((lambda (x)
       (x y))
     x)))
```

Solution:



Exercise 2.28 Write an unparser that converts the abstract syntax of an lc-exp into a string that matches the second grammar in this section (page 52).

Solution:

See source code file in "section-2.5/2.28.rkt".

Exercise 2.29 Where a Kleene star or plus (page 7) is used in concrete syntax, it is most convenient to use a list of associated subtrees when constructing an abstract syntax tree. For example, if the grammar for lambda-calculus expressions had been

$$
\begin{aligned}
\textit{Lc-exp} &::= \textit{Identifier} \\
&\qquad \boxed{\texttt{var-exp (var)}} \\
&::= \texttt{(lambda (}\{\textit{identifier}\}^*\texttt{)}\ \textit{Lc-exp}\texttt{)} \\
&\qquad \boxed{\texttt{lambda-exp (bound-vars body)}} \\
&::= \texttt{(}\textit{Lc-exp}\quad \{\textit{Lc-exp}\}^*\texttt{)} \\
&\qquad \boxed{\texttt{app-exp (rator rands)}}
\end{aligned}
$$

then the predicate for the `bound-vars` field could be `(list-of identifier?)`, and the predicate for the rands field could be `(list-of lc-exp?)`. Write a `define-datatype` and a parser for this grammar that works in this way.

Solution:

See source code file in "section-2.5/2.29.rkt".

Exercise 2.30 The procedure parse-expression as defined above is fragile: it does not detect several possible syntactic errors, such as `(a b c)`, and aborts with inappropriate error messages for other expressions, such as `(lambda)`. Modify it so that it is robust, accepting any s-exp and issuing an appropriate error message if the s-exp does not represent a lambda-calculus expression.

Solution:

See source code file in "section-2.5/2.29.rkt".

Exercise 2.31 Sometimes it is useful to specify a concrete syntax as a sequence of symbols and integers, surrounded by parentheses. For example, one might define the set of *prefix lists* by

$$
\begin{aligned}
\textit{Prefix-list} &::= \texttt{(}\textit{Prefix-exp}\texttt{)} \\
\textit{Prefix-exp} &::= \textit{Int} \\
&::= \texttt{-}\quad \textit{Prefix-exp}\quad \textit{Prefix-exp}
\end{aligned}
$$

so that `(- - 3 2 - 4 - 12 7)` is a legal prefix list. This is sometimes called *Polish prefix notation*, after its inventor, Jan ï£¡ukasiewicz. Write a parser to convert a prefixlist to the abstract syntax

```
(define-datatype prefix-exp prefix-exp?
  (const-exp
   (num integer?))
  (diff-exp
   (operand1 prefix-exp?)
   (operand2 prefix-exp?)))
```

so that the example above produces the same abstract syntax tree as the sequence ofconstructors

```
(diff-exp
 (diff-exp
  (const-exp 3)
  (const-exp 2))
 (diff-exp
  (const-exp 4)
  (diff-exp
   (const-exp 12)
   (const-exp 7))))
```

As a hint, consider writing a procedure that takes a list and produces a `prefix-exp` and the list of leftover list elements.

Solution:

See source code file in "section-2.5/2.31.rkt".

3 *Expressions*

3.1 Specification and Implementation Strategy

3.2 LET: A Simple Language

Exercise 3.1 In figure 3.3, list all the places where we used the fact that $\lfloor\lceil n\rceil\rfloor = n$.

Solution:

Any `const-exp` apply to other expression.

Exercise 3.2 Give an expressed value $val \in ExpVal$ for which $\lceil\lfloor val\rfloor\rceil \neq val$.

Solution:

If there is a strict type system, then no such expressed value. Otherwise, there exists. For example, 0 also can represent boolean false, other numbers represent boolean true, set initial $val =$ `(num-val 5)`$=\lceil 5\rceil$, $\lfloor 5\rfloor =$ `(expval->num` val`)`$=5$, `(bool-val 5)`$=\lceil \#t\rceil \neq \lceil 5\rceil$.

Exercise 3.3 Why is subtraction a better choice than addition for our single arithmetic operation?

Solution:

Addition products larger result, and easily identify which values in current domain.

Exercise 3.4 Write out the derivation of figure 3.4 as a derivation tree in the style of the one on page 5.

Solution:

Elided.

Exercise 3.5 Write out the derivation of figure 3.5 as a derivation tree in the style of the one on page 5.

Solution:

Elided.

Exercise 3.6 Extend the language by adding a new operator `minus` that takes one argument, `n`, and returns `-n`. For example, the value of `minus(-(minus(5),9))` should be 14.

Solution:

See source code files in folder "section-3.2".

Exercise 3.7 Extend the language by adding operators for addition, multiplication, and integer quotient.

Solution:

See source code files in folder "section-3.2".

Exercise 3.8 Add a numeric equality predicate `equal?` and numeric order predicates `greater?` and `less?` to the set of operations in the defined language.

Solution:

See source code files in folder "section-3.2".

Exercise 3.9 Add list processing operations to the language, including `cons`, `car`, `cdr`, `null?` and `emptylist`. A list should be able to contain any expressed value, including another list. Give the definitions of the expressed and denoted values of the language, as in section 3.2.2. For example,

```
let x = 4
in cons(x,
        cons(cons(-(x,1),
                  emptylist),
             emptylist))
```

should return an expressed value that represents the list `(4 (3))`.

Solution:

See source code files in folder "section-3.2".

Exercise 3.10 Add an operation `list` to the language. This operation should take any number of arguments, and return an expressed value containing the list of their values. For example,

```
let x = 4
in list(x, -(x,1), -(x,3))
```

should return an expressed value that represents the list `(4 3 1)`.

Solution:

See source code files in folder "section-3.2".

Exercise 3.11 In a real language, one might have many operators such as those in the preceding exercises. Rearrange the code in the interpreter so that it is easy to add new operators.

Solution:

See source code files in folder "section-3.2".

Exercise 3.12 Add to the defined language a facility that adds a cond expression. Use the grammar

$$\textit{Expression} \quad ::= \quad \texttt{cond} \quad \{\textit{Expression} \quad \texttt{==>} \quad \textit{Expression}\}^* \quad \texttt{end}$$

In this expression, the expressions on the left-hand sides of the ==>'s are evaluated in order until one of them returns a true value. Then the value of the entire expression is the value of the corresponding right-hand expression. If none of the tests succeeds, the expression should report an error.

Solution:

See source code files in folder "section-3.2".

Exercise 3.13 Change the values of the language so that integers are the only expressed values. Modify `if` so that the value 0 is treated as false and all other values are treated as true. Modify the predicates accordingly.

Solution:

```
(define num-pred?
  (lambda (n)
    (if (= n 0)
        #f
        #t)))

(define value-of
  (lambda (exp env)
    (cases expression exp
      ...
      [if-exp
       (exp1 exp2 exp3)
       (let ([val1 (value-of exp1 env)])
         (if (num-pred? (expval->num val1))
             (value-of exp2 env)
             (value-of exp3 env)))]
      ...
      )))
```

Exercise 3.14 As an alternative to the preceding exercise, add a new nonterminal Bool-exp of boolean expressions to the language. Change the production for conditional expressions to say

Expression ::= `if` *Bool-exp* `then` *Expression* `else` *Expression*

Write suitable productions for Bool-exp and implement `value-of-bool-exp`. Where do the predicates of exercise 3.8 wind up in this organization?

Solution:

See source code files in folder “section-3.2”.

Exercise 3.15 Extend the language by adding a new operation `print` that takes one argument, prints it, and returns the integer 1. Why is this operation not expressible in our specification framework?

Solution:

`print` not returns a value.

See source code files in folder “section-3.2”.

Exercise 3.16 Extend the language so that a `let` declaration can declare an arbitrary number of variables, using the grammar

Expression ::= `let` {*Identifier* = *Expression*}* `in` *Expression*

As in Scheme's `let`, each of the right-hand sides is evaluated in the current environment, and the body is evaluated with each new variable bound to the value of its associated right-hand side. For example,

```
let x = 30
in let x = -(x,1)
       y = -(x,2)
   in -(x,y)
```

should evaluate to 1.

Solution:

See source code files in folder "section-3.2".

Exercise 3.17 Extend the language with a `let*` expression that works like Scheme's `let*`, so that

```
let x = 30
in let* x = -(x,1) y = -(x,2)
   in -(x,y)
```

should evaluate to 2.

Solution:

See source code files in folder "section-3.2".

Exercise 3.18 Add an expression to the defined language:

Expression ::= `unpack` {*Identifier*}* = *Expression* `in` *Expression*

so that `unpack x y z = lst in ...` binds `x`, `y`, and `z` to the elements of `lst` if `lst` is a list of exactly three elements, and reports an error otherwise. For example, the value of

```
let u = 7
in unpack x y = cons(u, (cons(3, emptylist))
   in -(x,y)
```

should be 4.

Solution:

See source code files in folder "section-3.2".

3.3 PROC: A Language with Procedures

Exercise 3.19 In many languages, procedures must be created and named at the same time. Modify the language of this section to have this property by replacing the `proc` expression with a `letproc` expression.

Solution:

Same as `let name = proc (var, ...) body`, except not including keyword `proc` and = symbol, for example,

```
letproce name(var, ...) proc-body
in let-body
```

Exercise 3.20 In PROC, procedures have only one argument, but one can get the effect of multiple argument procedures by using procedures that return other procedures. For example, one might write code like

```
let f = proc (x) proc (y) ...
in ((f 3) 4)
```

This trick is called *Currying*, and the procedure is said to be *Curried*. Write a Curried procedure that takes two arguments and returns their sum. You can write $x+y$ in our language by writing $-(x,-(0,y))$.

Solution:

```
let f = proc (x) proc (y) -(x,-(0,y))
in ((f 3) 4)
```

Exercise 3.21 Extend the language of this section to include procedures with multiple arguments and calls with multiple operands, as suggested by the grammar

$$\begin{array}{lll} \textit{Expression} & ::= & \texttt{proc (}\{\textit{Identifier}\}\texttt{)*(,)}\ \ \textit{Expression} \\ \textit{Expression} & ::= & \texttt{(}\textit{Expression}\ \ \{\textit{Expression}\}\texttt{*)} \end{array}$$

Solution:

See source code files in folder "section-3.3".

Exercise 3.22 The concrete syntax of this section uses different syntax for a built-in operation, such as difference, from a procedure call. Modify the concrete syntax so that the user of this language need not know which operations are built-in and which are defined procedures. This exercise may range from very easy to hard, depending on the parsing technology being used.

Solution:

We can define procedures, just need to ensure that all basic functions are built-in before doing this, so we can write any procedure into inital environment.

$$\begin{array}{lll} \textit{Procedure} & ::= & \texttt{+(}\textit{Number, Number}\texttt{)} \\ \textit{Procedure} & ::= & \texttt{-(}\textit{Number, Number}\texttt{)} \\ & & \vdots \end{array}$$

Exercise 3.23 What is the value of the following PROC program?

```
let makemult = proc (maker)
                proc (x)
                 if zero?(x)
                 then 0
                 else -(((maker maker) -(x,1)), -4)
in let times4 = proc (x) ((makemult makemult) x)
   in (times4 3)
```

Use the tricks of this program to write a procedure for factorial in PROC. As a hint, remember that you can use Currying (exercise 3.20) to define a two-argument procedure `times`.

Solution:

```
let f1 = proc (maker)
          proc (a)
           proc (b)
            if zero?(a)
            then 0
            else -((((maker maker) -(a,1)) b), -(0,b))
in let f2 = proc (maker)
             proc (n)
              if zero?(n)
              then 1
              else (((f1 f1) n)
                    ((maker maker) -(n,1)))
   in let f3 = (f2 f2)
      in (f3 7)
```

Exercise 3.24 Use the tricks of the program above to write the pair of mutually recursive procedures, `odd` and `even`, as in exercise 3.32.

Solution:

```
let* even = proc (maker)
             proc (n)
              if zero?(n)
              then 1
              else ((odd maker) -(n,1))
     odd  = proc (maker)
             proc (n)
              if zero?(n)
              then 0
              else ((even maker) -(n,1))
in let even? = (even odd)
       odd?  = (odd even)
   in (even? 7)
```

Exercise 3.25 The tricks of the previous exercises can be generalized to show that we can define any recursive procedure in PROC. Consider the following bit of code:

```
let makerec = proc (f)
               let d = proc (x)
                        proc (z) ((f (x x)) z)
               in proc (n) ((f (d d)) n)
in let maketimes4 = proc (f)
                     proc (x)
                      if zero?(x)
                      then 0
                      else -((f -(x,1)), -4)
   in let times4 = (makerec maketimes4)
      in (times4 3)
```

Show that it returns 12.

Solution:

```
(times4 3)
((makerec maketimes4) 3)
((maketimes4 (d d)) 3)
((maketimes4 |proc (z) ((maketimes4 (d d)) z)|) 3)
(|proc (x) ... else -((|proc (z) ((maketimes4 (d d)) z)| -(x,1)),
-4)|
  3)
if zero?(3) ... else -(((maketimes4 (d d)) 2), -4)
-(((maketimes4 (d d)) 2), -4)
-(((maketimes4 |proc (z) ((maketimes4 (d d)) z)|) 2), -4)
-((|proc (x) ... else -((|proc (z) ((maketimes4 (d d) z)| -(x,1)),
-4)|
  2),-4)
-(if zero?(2) ... else -(((maketimes4 (d d) 1), -4), -4)
-(-(((maketimes4 (d d)) 1), -4), -4)
...
-(-(-(((maketimes4 (d d)) 0), -4), -4), -4)
-(-(-(0, -4), -4), -4)
12
```

Exercise 3.26 In our data-structure representation of procedures, we have kept the entire environment in the closure. But of course all we need are the bindings for the free variables. Modify the representation of procedures to retain only the free variables.

Solution:

See source code files in folder “section-3.3”.

Exercise 3.27 Add a new kind of procedure called a `traceproc` to the language. A `traceproc` works exactly like a `proc`, except that it prints a trace message on entry and on exit.

Solution:

See source code files in folder "section-3.3".

Exercise 3.28 *Dynamic binding* (*or dynamic scoping*) is an alternative design for procedures, in which the procedure body is evaluated in an environment obtained by extending the environment at the point of call. For example in

```
let a = 3
in let p = proc (x) -(x,a)
       a = 5
   in -(a, (p 2))
```

the `a` in the procedure body would be bound to `5`, not `3`. Modify the language to use dynamic binding. Do this twice, once using a procedural representation for procedures, and once using a data-structure representation.

Solution:

Just change the code in `func` constructor, remove environment. In `apply-func`, use the current environment.

```
(define-datatype function function?
  (func
   [vars (list-of identifier?)]
   [body expression?]))

(define apply-func
  (lambda (f args env)
    (cases function f
      [func
       (vars body)
       (value-of body (extend-env* vars args env))])))
```

Exercise 3.29 Unfortunately, programs that use dynamic binding may be exceptionally difficult to understand. For example, under lexical binding, consistently renaming the bound variables of a procedure can never change the behavior of a program: we can even remove all variables and replace them by their lexical addresses, as in section 3.6. But under dynamic binding, this transformation is unsafe.

For example, under dynamic binding, the procedure `proc (z) a` returns the value of the variable `a` in its caller's environment. Thus, the program

```
let a = 3
in let p = proc (z) a
   in let f = proc (x) (p 0)
      in let a = 5
         in (f 2)
```

returns `5`, since `a`'s value at the call site is 5. What if `f`'s formal parameter were `a`?

Solution:

2

3.4 LETREC: A Language with Recursive Procedures

Exercise 3.30 What is the purpose of the call to `proc-val` on the next-to-last line of `apply-env`?

Solution:

We can't directly set a cycle link between in envrionment and proc-val, only make a proc-val whlie calling `apply-env`.

Exercise 3.31 Extend the language above to allow the declaration of a recursive procedure of possibly many arguments, as in exercise 3.21.

Solution:

See source code files in folder “section-3.4”.

Exercise 3.32 Extend the language above to allow the declaration of any number of mututally recursive unary procedures, for example:

```
letrec even(x) = if zero?(x) then 1 else (odd -(x,1))
       odd(x) = if zero?(x) then 0 else (even -(x,1))
in (odd 13)
```

Solution:

See source code files in folder “section-3.4”.

Exercise 3.33 Extend the language above to allow the declaration of any number of mutually recursive procedures, each of possibly many arguments, as in exercise 3.21.

Solution:

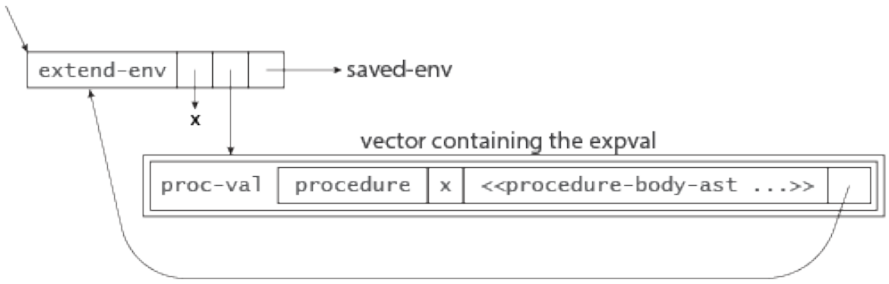
See source code files in folder “section-3.4”.

Exercise 3.34 Implement extend-env-rec in the procedural representationof environments from section 2.2.3.

Solution:

See source code file in "section-3.4/3.34.rkt".

Exercise 3.35 The representations we have seen so far are inefficient, because they build a new closure every time the procedure is retrieved. But the closure is the same every time. We can build the closures only once, by putting the value in a vector of length 1 and building an explicit circular structure, like



Here's the code to build this data structure.

```
(define extend-env-rec
  (lambda (p-name b-var body saved-env)
    (let ((vec (make-vector 1)))
      (let ((new-env (extend-env p-name vec saved-env)))
        (vector-set! vec 0
                     (proc-val (procedure b-var body new-env)))
        new-env))))
```

Complete the implementation of this representation by modifying the definitions of the environment data type and `apply-env` accordingly. Be sure that `apply-env` always returns an expressed value.

Solution:

See source code files in folder "section-3.4".

Exercise 3.36 Extend this implementation to handle the language from exercise 3.32.

Solution:

See source code files in folder "section-3.4".

Exercise 3.37 With dynamic binding (exercise 3.28), recursive procedures may be bound by `let`; no special mechanism is necessary for recursion. This is of historical interest; in the early years of programming language design other approaches to recursion, such as those discussed in section 3.4, were not widely understood. To demonstrate recursion via dynamic binding, test the program

```
let fact = proc (n) add1(n)
in let fact = proc (n)
               if zero?(n)
               then 1
               else *(n, (fact -(n,1)))
   in (fact 5)
```

using both lexical and dynamic binding. Write the mutually recursive procedures `even` and `odd` as in section 3.4 in the defined language with dynamic binding.

Solution:

Too lazy to write code.

Lexical binding returns 120. Dynamic binding returns 25.

3.5 Scoping and Binding of Variables

3.6 Eliminating Variable Names

3.7 Implementing Lexical Addressing

Exercise 3.38 Extend the lexical address translator and interpreter to handle `cond` from exercise 3.12.

Solution:

See source code files in folder "section-3.7".

Exercise 3.39 Extend the lexical address translator and interpreter to handle `pack` and `unpack` from exercise 3.18.

Solution:

This need adding a new expressed list value for unpacking multi values. Too lazy to do.

Exercise 3.40 Extend the lexical address translator and interpreter to handle `letrec`. Do this by modifying the context argument to `translation-of` so that it keeps track of not only the name of each bound variable, but also whether it was bound by `letrec` or not. For a reference to a variable that was bound by a `letrec`, generate a new kind of reference, called a `nameless-letrec-var-exp`. You can then continue to use the nameless environment representation above, and the interpreter can do the right thing with a `nameless-letrec-var-exp`.

Solution:

I don't think that need add a `nameless-letrec-var-exp`.

Exercise 3.41 Modify the lexical address translator and interpreter to handle `let` expressions, procedures, and procedure calls with multiple arguments, as in exercise 3.21. Do this using a nameless version of the ribcage representation of environments (exercise 2.11). For this representation, the lexical address will consist of two non-negative integers: the lexical depth, to indicate the number of contours crossed, as before; and a position, to indicate the position of the variable in the declaration.

Solution:

Elided.

Exercise 3.42 Modify the lexical address translator and interpreter to use the trimmed representation of procedures from exercise 3.26. For this, you will need to translate the body of the procedure not `(extend-senv var senv)`, but in a new static environment that tells exactly where each variable will be kept in the trimmed representation.

Solution:

See source code files in folder "section-3.7", main in file "section-3.7/trim-func.rkt".

Exercise 3.43 The translator can do more than just keep track of the names of variables. For example, consider the program

```
let x = 3
in let f = proc (y) -(y,x)
   in (f 13)
```

Here we can tell statically that at the procedure call, `f` will be bound to a procedure whose body is `-(y,x)`, where `x` has the same value that it had at the procedure creation site. Therefore we could avoid looking up `f` in the environment entirely. Extend the translator to keep track of "known procedures" and generate code that avoids an environment lookup at the call of such a procedure.

Solution:

See source code files in folder "section-3.7".

Exercise 3.44 In the preceding example, the only use of `f` is as a known procedure. Therefore the procedure built by the expression `proc (y) -(y,x)` is never used. Modify the translator so that such a procedure is never constructed.

Solution:

See source code files in folder "section-3.7".

4 *State*

4.1 Computational Effects

4.2 EXPLICIT-REFS: A Language with Explicit References

Exercise 4.1 What would have happened had the program been instead

```
let g = proc (dummy)
         let counter = newref(0)
         in begin
             setref(counter, -(deref(counter), -1));
             deref(counter)
            end
in let a = (g 11)
   in let b = (g 11)
      in -(a,b)
```

Solution:

a = 1, b = 1, return 0.

Exercise 4.2 Write down the specification for a `zero?-exp`.

Solution:

$$\frac{\texttt{(value-of } exp_1 \quad \rho \quad \sigma_0) = (val_1, \sigma_1)}{\texttt{(value-of (zero?-exp } exp_1) \ \rho \quad \sigma_0)} = \begin{cases} (\lceil \texttt{\#t} \rceil, \sigma_1) & \text{if } \texttt{(expval->num } val_1) = 0 \\ (\lceil \texttt{\#f} \rceil, \sigma_1) & \text{if } \texttt{(expval->num } val_1) \neq 0 \end{cases}$$

Exercise 4.3 Write down the specification for a `call-exp`.

Solution:

$$\frac{\begin{array}{c}\texttt{(value-of } rator \quad \rho \quad \sigma_0) = (val_1, \sigma_1) \\ \texttt{(value-of } rand \quad \rho \quad \sigma_1) = (val_2, \sigma_2)\end{array}}{\begin{array}{l}\texttt{(value-of (call-exp } rator \quad rand) \ \rho \quad \sigma_0) \\ = \texttt{(apply-procedure } val_1 \quad val_2 \quad \sigma_2)\end{array}}$$

Exercise 4.4 Write down the specification for a `begin` expresssion.

Expression ::= `begin` *Expression* {; *Expression*}* `end`

A `begin` expression may contain one or more subexpressions separated by semicolons. These are evaluated in order and the value of the last is returned.

Solution:

$$\texttt{(value-of (begin-exp } exps* \;\; exp\texttt{)} \;\; \rho \;\; \sigma_0\texttt{)}$$
$$= \texttt{(value-of } exp \;\; \rho \;\; \sigma_*\texttt{)}$$

Exercise 4.5 Write down the specification for `list` (exercise 3.10).

Solution:

$$\texttt{(value-of } exps* \;\; \rho \;\; \sigma_0\texttt{)}$$
$$= \texttt{((list-val (list } val_*\texttt{))}, \sigma_*\texttt{)}$$

Exercise 4.6 Modify the rule given above so that a `setref-exp` returns the value of the right-hand side.

Solution:

$$\frac{\texttt{(value-of } exp_1 \;\; \rho \;\; \sigma_0\texttt{)} = (l, \sigma_1) \qquad \texttt{(value-of } exp_2 \;\; \rho \;\; \sigma_1\texttt{)} = (val, \sigma_2)}{\texttt{(value-of (setref-exp } exp_1 \;\; exp_2\texttt{)} \;\; \rho \;\; \sigma_0\texttt{)} = (\lceil val \rceil, [l = val]\sigma_2)}$$

Exercise 4.7 Modify the rule given above so that a `setref-exp` returns the old contents of the location.

Solution:

$$\frac{\texttt{(value-of } exp_1 \;\; \rho \;\; \sigma_0\texttt{)} = (l, \sigma_1) \qquad \texttt{(value-of (deref-exp } l\texttt{)} \;\; \rho \;\; \sigma_1\texttt{)} = (val_1, \sigma_2) \qquad \texttt{(value-of } exp_2 \;\; \rho \;\; \sigma_2\texttt{)} = (val_2, \sigma_3)}{\texttt{(value-of (setref-exp } exp_1 \;\; exp_2\texttt{)} \;\; \rho \;\; \sigma_0\texttt{)} = (\lceil val_1 \rceil, [l = val_2]\sigma_3)}$$

Exercise 4.8 Show exactly where in our implementation of the store these operations take linear time rather than constant time.

Solution:

`setref!` takes $\Theta(n)$ time, `deref` depends on where `list-ref` takes $\Theta(n)$ or $\Theta(1)$ time.

Exercise 4.9 Implement the store in constant time by representing it as a Scheme vector. What is lost by using this representation?

Solution:

Every `newref` operation will copy entire old store to new store.

Exercise 4.10 Implement the `begin` expression as specified in exercise 4.4.

Solution:

See source code files in folder "section-4.2".

Exercise 4.11 Implement `list` from exercise 4.5.

Solution:

See source code files in folder "section-4.2".

Exercise 4.12 Our understanding of the store, as expressed in this interpreter, depends on the meaning of effects in Scheme. In particular, it depends on us knowing when these effects take place in a Scheme program. We can avoid this dependency by writing an interpreter that more closely mimics the specification. In this interpreter, `value-of` would return both a value and a store, just as in the specification. A fragment of this interpreter appears in figure 4.6. We call this a store-passing interpreter.

Extend this interpreter to cover all of the language EXPLICIT-REFS. Every procedure that might modify the store returns not just its usual value but also a new store. These are packaged in a data type called `answer`. Complete this definition of `value-of`.

Solution:

See source code files in folder "section-4.2/4.12~4.13".

Exercise 4.13 Extend the interpreter of the preceding exercise to have procedures of multiple arguments.

Solution:

See source code files in folder "section-4.2/4.12~4.13".

note: I might have found a mistake in the book. In the code fragment of figure 4.6, `var-exp` should return `(apply-store store (apply-env env var))`, not an `answer` type. Just like `expval` type, `answer` type constructor only should appear where returning expressed value, `var-exp` returns the value it denoted, that is an `answer` type stored in `store`.

4.3 IMPLICIT-REFS: A Language with Implicit References

Exercise 4.14 Write the rule for `let`.

Solution:

$$\frac{\texttt{(value-of}\ exp\quad \rho\quad \sigma_0\texttt{)} = (val, \sigma_1)}{\begin{array}{l}\texttt{(value-of (let-exp}\ var\quad exp\quad body\texttt{)}\ \rho\quad \sigma_0\texttt{)} \\ = \texttt{(value-of}\ body\quad [var=l]\rho\quad [l=val]\sigma\texttt{)}\end{array}}$$

Exercise 4.15 In figure 4.8, why are variables in the environment bound to plain integers rather than expressed values, as in figure 4.5?

Solution:

Implicit references.

Exercise 4.16 Now that variables are mutable, we can build recursive procedures by assignment. For example

```
letrec times4(x) = if zero?(x)
                   then 0
                   else -((times4 -(x,1)), -4)
in (times4 3)
```

can be replaced by

```
let times4 = 0
in begin
    set times4(x) = proc (x)
                     if zero?(x)
                     then 0
                     else -((times4 -(x,1)), -4);
    (times4 3)
   end
```

Trace this by hand and verify that this translation works.

Solution:

See source file in "section-4.3/test.rkt".

Exercise 4.17 Write the rules for and implement multiargument procedures and `let` expressions.

Solution:

See source files in folder "section-4.3".

Exercise 4.18 Write the rule for and implement multiprocedure `letrec` expressions.

Solution:

See source files in folder "section-4.3".

Exercise 4.19 Modify the implementation of multiprocedure `letrec` so that each closure is built only once, and only one location is allocated for it. This is like exercise 3.35.

Solution:

See source files in folder "section-4.3".

Exercise 4.20 In the language of this section, all variables are mutable, as they are in Scheme. Another alternative is to allow both mutable and immutable variable bindings:

$$\begin{aligned} \mathit{ExpVal} &= \mathit{Int} + \mathit{Bool} + \mathit{Proc} \\ \mathit{DenVal} &= \mathit{Ref}(\mathit{ExpVal}) + \mathit{ExpVal} \end{aligned}$$

Variable assignment should work only when the variable to be assigned to has a mutable binding. Dereferencing occurs implicitly when the denoted value is a reference.

Modify the language of this section so that `let` introduces immutable variables, as before, but mutable variables are introduced by a `letmutable` expression, with syntax given by

$$\mathit{Expression} ::= \texttt{letmutable}\ \mathit{Identifier} = \mathit{Expression}\ \texttt{in}\ \mathit{Expression}$$

Solution:

Environment has two types value, *Ref*(*ExpVal*) and *ExpVal*. *Ref*(*ExpVal*) is mutable, *ExpVal* is immutable.

Exercise 4.21 We suggested earlier the use of assignment to make a program more modular by allowing one procedure to communicate information to a distant procedure without requiring intermediate procedures to be aware of it. Very often such an assignment should only be temporary, lasting for the execution of a procedure call. Add to the language a facility for *dynamic assignment* (also called *fluid binding*) to accomplish this. Use the production

$$\mathit{Expression} ::= \texttt{setdynamic}\ \mathit{Identifier} = \mathit{Expression}\ \texttt{during}\ \mathit{Expression}$$

```
setdynamic-exp (var exp1 body)
```

The effect of the `setdynamic` expression is to assign temporarily the value of exp_1 to *var*, evaluate *body*, reassign *var* to its original value, and return the value of *body*. The variable var must already be bound. For example, in

```
let x = 11
in let p = proc (y) -(y,x)
   in -(setdynamic x = 17 during (p 22), (p 13))
```

the value of `x`, which is free in procedure `p`, is 17 in the call `(p 22)`, but is reset to 11 in the call `(p 13)`, so the value of the expression is 5 - 2 = 3.

Solution:

See source files in folder "section-4.3/4.21".

Exercise 4.22 So far our languages have been expression-oriented: the primary syntactic category of interest has been expressions and we have primarily been interested in their values. Extend the language to model the simple statement-oriented language whose specification is sketched below. Be sure to *Follow the Grammar* by writing separate procedures to handle programs, statements, and expressions.

Values. As in IMPLICIT-REFS.

Syntax. Use the following syntax:

$$
\begin{array}{lll}
\textit{Program} & ::= & \textit{Statement} \\
\textit{Statement} & ::= & \textit{Identifier} = \textit{Expression} \\
 & ::= & \texttt{print}\ \textit{Expression} \\
 & ::= & \texttt{\{}\{\textit{Statement}\}^{*(;)}\ \texttt{\}} \\
 & ::= & \texttt{if}\ \textit{Expression}\quad \textit{Statement}\quad \textit{Statement} \\
 & ::= & \texttt{while}\ \textit{Expresion}\quad \textit{Statement} \\
 & ::= & \texttt{var}\ \{\ \textit{Identifier}\ \}^{*(,)}\ \texttt{;}\ \textit{Statement}
\end{array}
$$

The nonterminal *Expression* refers to the language of expressions of IMPLICIT-REFS, perhaps with some extensions.

Semantics. A program is a statement. A statement does not return a value, but acts by modifying the store and by printing.

Assignment statements work in the usual way. A print statement evaluates its actual parameter and prints the result. The `if` statement works in the usual way. A block statement, defined in the last production for *Statement*, binds each of the declared variables to an uninitialized reference and then executes the body of the block. The scope of these bindings is the body. Write the specification for statements using assertions like

$$\texttt{(result-of}\quad \textit{stmt}\quad \rho\quad \sigma_0\texttt{)} = \sigma_1$$

Example. Here are some examples.

```
(run "var x,y; {x = 3; y = 4; print +(x,y)}")  % Example 1
7
(run "var x,y,z; {x = 3;                       % Example 2
                  y = 4;
                  z = 0;
                  while not(zero?(x))
                    {z = +(z,y); x = -(x,1)};
                  print z}")
12
(run "var x; {x = 3;                           % Example 3
              print x;
              var x; {x = 4; print x};
              print x}")
3
4
3
(run "var f,x; {f = proc(x,y) *(x,y);          % Example 4
```

Solution:

See source files in folder "section-4.3/4.22~4.24".

Exercise 4.23 Add to the language of exercise 4.22 `read` statements of the form `read` *var*. This statement reads a nonnegative integer from the input and stores it in the given variable.

Solution:

See source files in folder "section-4.3/4.22~4.24".

Exercise 4.24 A `do-while` statement is like a `while` statement, except that the test is performed *after* the execution of the body. Add `do-while` statements to the language of exercise 4.22.

Solution:

See source files in folder "section-4.3/4.22~4.24".

Exercise 4.25 Extend the block statement of the language of exercise 4.22 to allow variables to be initialized. In your solution, does the scope of a variable include the initializer for variables declared later in the same block statement?

Solution:

See source files in folder "section-4.3/4.25~4.27".

Yes, but it is used sequentially just like `let*` in Scheme.

Exercise 4.26 Extend the solution to the preceding exercise so that procedures declared in a single block are mutually recursive. Consider restricting the language so that the variable declarations in a block are followed by the procedure declarations.

Solution:

See source files in folder "section-4.3/4.25~4.27".

Exercise 4.27 Extend the language of the preceding exercise to include *subroutines*. In our usage a subroutine is like a procedure, except that it does not return a value and its body is a statement, rather than an expression. Also, add subroutine calls as a new kind of statement and extend the syntax of blocks so that they may be used to declare both procedures and subroutines. How does this affect the denoted and expressed values? What happens if a procedure is referenced in a subroutine call, or vice versa?

Solution:

See source files in folder "section-4.3/4.25~4.27".

Denoted value is still *Ref*(*ExpVal*), expressed value add a *Subproc*.

Procedure in subroutine, just return a value for the statement.

Subroutine in procedure, I think that if subroutine is *call by reference*, then it is more useful than *natural parameter passing*, it can be used for modifying the value of outer variable, not only printing something. If that, it will deviate from the consistency of IMPLICIT-REFS.

4.4 MUTABLE-PAIRS: A Language with Mutable Pairs

Exercise 4.28 Write down the specification rules for the five mutable-pair operations.

Solution:

$$\texttt{newpair-exp}\quad \frac{\texttt{(value-of } exp_1 \quad \rho \quad \sigma_0) = (val_1,\ \sigma_1) \qquad \texttt{(value-of } exp_2 \quad \rho \quad \sigma_1) = (val_2,\ \sigma_2)}{\texttt{(value-of (newpair-exp } exp_1 \quad exp_2) \ \rho \quad \sigma_0) = (\lceil pair(l_1, l_2) \rceil, [l_2 = val_2, l_1 = val_1]\sigma_2)}$$

$$\texttt{left-exp}\quad \frac{\texttt{(value-of } exp \quad \rho \quad \sigma_0) = (val,\ \sigma_1) \qquad \lfloor val \rfloor = (pair(l_1, l_2), \sigma_2)}{\texttt{(value-of (left-exp } exp) \ \rho \quad \sigma_0) = (\sigma(l_1), \sigma_2)}$$

$$\texttt{right-exp}\quad \frac{\texttt{(value-of } exp \quad \rho \quad \sigma_0) = (val,\ \sigma_1) \qquad \lfloor val \rfloor = (pair(l_1, l_2), \sigma_2)}{\texttt{(value-of (right-exp } exp) \ \rho \quad \sigma_0) = (\sigma(l_2), \sigma_2)}$$

$$\texttt{setleft-exp}\quad \frac{\texttt{(value-of } exp_1 \quad \rho \quad \sigma_0) = (val_1,\ \sigma_1) \qquad \texttt{(value-of } exp_2 \quad \rho \quad \sigma_1) = (val_2,\ \sigma_2) \qquad \lfloor val_1 \rfloor = (pair(l_1, l_2), \sigma_3)}{\texttt{(value-of (setleft-exp } exp_1 \quad exp_2) \ \rho \quad \sigma_0) = (\lceil 82 \rceil, [l_1 = val_2]\sigma_3)}$$

$$\texttt{setright-exp}\qquad \frac{\begin{array}{c}\texttt{(value-of } exp_1 \quad \rho \quad \sigma_0) = (val_1, \sigma_1) \\ \texttt{(value-of } exp_2 \quad \rho \quad \sigma_1) = (val_2, \sigma_2) \\ \lfloor val_1 \rfloor = (pair(l_1, l_2), \sigma_3)\end{array}}{\begin{array}{l}\texttt{(value-of (setright-exp } exp_1 \quad exp_2) \ \rho \quad \sigma_0) \\ = (\lceil 83 \rceil, [l_2 = val_2]\sigma_3)\end{array}}$$

Exercise 4.29 Add arrays to this language. Introduce new operators newarray, arrayref, and arrayset that create, dereference, and update arrays. This leads to

```
let a = newarry(2,-99)
    p = proc (x)
         let v = arrayref(x,1)
         in arrayset(x,1,-(v,-1))
in begin arrayset(a,1,0);
         (p a);
         (p a);
         arrayref(a,1) end
```

Here `newarray(2,-99)` is intended to build an array of size 2, with each location in the array containing -99. begin expressions are defined in exercise 4.4. Make the array indices zero-based, so an array of size 2 has indices 0 and 1.

Solution:

See source files in folder "section-4.4".

Exercise 4.30 Add to the language of exercise 4.29 a procedure `arraylength`, which returns the size of an array. Your procedure should work in constant time. Make sure that `arrayref` and `arrayset` check to make sure that their indices are within the length of the array.

Solution:

See source files in folder "section-4.4".

4.5 Parameter-Passing Variations

Exercise 4.31 Write out the specification rules for CALL-BY-REFERENCE.

Solution:

$$\frac{\begin{array}{c}\texttt{(value-of}\ rator\ \ \rho\ \ \sigma_0\texttt{)} = (f, \sigma_1) \\ \texttt{(value-of}\ rand\ \ \rho\ \ \sigma_1\texttt{)} = \begin{cases}\texttt{if}\ rand = \texttt{var-exp}, & (ref = \rho(rand), \sigma_1) \\ \texttt{else} & (ref, [ref = val]\sigma_2)\end{cases}\end{array}}{\begin{array}{l}\texttt{(value-of (call-exp}\ rator\ \ rand\texttt{)}\ \rho\ \ \sigma_0\texttt{)} \\ = \texttt{(apply-procedure}\ f\ \ ref\texttt{)}\end{array}}$$

Exercise 4.32 Extend the language CALL-BY-REFERENCE to have procedures of multiple arguments.

Solution:

See source files in folder "section-4.5".

Exercise 4.33 Extend the language CALL-BY-REFERENCE to support call-by-value procedures as well.

Solution:

See source files in folder "section-4.5".

Exercise 4.34 Add a call-by-reference version of `let`, called `letref`, to the language. Write the specification and implement it.

Solution:

See source files in folder "section-4.5".

Exercise 4.35 We can get some of the benefits of call-by-reference without leaving the call-by-value framework. Extend the language IMPLICIT-REFS by adding a new expression

$$\begin{array}{lll} \textit{Expression} & ::= & \texttt{ref}\ \textit{Identifier} \\ & & \boxed{\texttt{ref-exp (var)}} \end{array}$$

This differs from the language EXPLICIT-REFS, since references are only of variables. This allows us to write familiar programs such as `swap` within our call-by-value language. What should be the value of this expression?

```
let a = 3
in let b = 4
   in let swap = proc (x) proc (y)
                  let temp = deref(x)
                  in begin
                     setref(x,deref(y));
                     setref(y,temp)
                     end
      in begin ((swap ref a) ref b); -(a,b) end
```

Here we have used a version of let with multiple declarations (exercise 3.16). What are the expressed and denoted values of this language?

Solution:

The value fo this expression is 1.

$$\begin{aligned} ExpVal &= Int + Bool + Proc + Ref(ExpVal) \\ DenVal &= Ref(ExpVal) \end{aligned}$$

Exercise 4.36 Most languages support arrays, in which case array references are generally treated like variable references under call-by-reference. If an operand is an array reference, then the location referred to, rather than its contents, is passed to the called procedure. This allows, for example, a swap procedure to be used in commonly occurring situations in which the values in two array elements are to be exchanged. Add array operators like those of exercise 4.29 to the call-by-reference language of this section, and extend `value-of-operand` to handle this case, so that, for example, a procedure application like

```
((swap (arrayref a i)) (arrayref a j))
```

will work as expected. What should happen in the case of

```
((swap (arrayref a (arrayref a i))) (arrayref a j))
```
?

Solution:

It depends on the way `swap` implemented. If `swap` is implemented same as preceding exercise, then report a extract-expval error. `arrayref` returns a *ref-val*, however, `arrayref` expects a *arr-val* and a *num-val*.

Exercise 4.37 *Call-by-value-result* is a variation on call-by-reference. In call-by-value-result, the actual parameter must be a variable. When a parameter is passed, the formal parameter is bound to a new reference initialized to the value of the actual parameter, just as in call-by-value. The procedure body is then executed normally. When the procedure body returns, however, the value in the new reference is copied back into the reference denoted by the actual parameter. This may be more efficient than call-by-reference because it can improve memory locality. Implement call-by-value-result and write a program that produces different answers using call-by-value-result and call-by-reference.

Solution:

See source files in folder "section-4.5".

Exercise 4.38 The example below shows a variation of exercise 3.25 that works under call-by-need. Does the original program in exercise 3.25 work under call-by-need? What happens if the program below is run under call-by-value? Why?

```
let makerec = proc (f)
               let d = proc (x) (f (x x))
               in (f (d d))
in let maketimes4 = proc (f)
                     proc (x)
                      if zero?(x)
                      then 0
                      else -((f -(x,1)), -4)
   in let times4 = (makerec maketimes4)
      in (times4 3)
```

Solution:

Under call-by-need, returns 12.

Under call-by-value, didn't return. It is a infinite-loop in `(f (d d))`.

Exercise 4.39 In the absence of effects, call-by-name and call-by-need always give the same answer. Construct an example in which call-by-name and call-by-need give different answers.

Solution:

```
let x = 0
    say = proc (a, b, c)
           if a then b then c
in say(zero?(x), x, error("arise a error"))
```

Exercise 4.40 Modify `value-of-operand` so that it avoids making thunks for constants and procedures.

Solution:

See source files in folder "section-4.5/4.40~4.42".

Exercise 4.41 Write out the specification rules for call-by-name and call-by-need.

Solution:

$$\frac{\texttt{(value-of}\ rator\ \ \rho\ \ \sigma_0) = (f, \sigma_1) \qquad \texttt{(value-of}\ rand\ \ \rho\ \ \sigma_1) = \begin{cases} \texttt{if}\ rand = \texttt{var-exp}, & (ref = \rho(rand), \sigma_1) \\ \texttt{else}\ & (ref, [ref = thunk]\sigma_2) \end{cases}}{\begin{aligned} &\texttt{(value-of (call-exp}\ rator\ \ rand\texttt{)}\ \rho\ \ \sigma_0) \\ &= \texttt{(apply-procedure}\ f\ \ ref\texttt{)} \end{aligned}}$$

Exercise 4.42 Add a lazy `let` to the call-by-need interpreter.

Solution:

See source files in folder "section-4.5/4.40~4.42".

5 *Continuation-Passing Interpreters*

5.1 A Continuation-Passing Interpreter

Exercise 5.1 Implement this data type of continuations using the procedural representation.

Solution:

See source code file in "section-5.1/procedural-continuation.rkt".

Exercise 5.2 Implement this data type of continuations using a data-structure representation.

Solution:

See source code file in "section-5.1/evaluation.rkt".

Exercise 5.3 Add `let2` to this interpreter. A `let2` expression is like a `let` expression, except that it defines exactly two variables.

Solution:

It is a multideclaration `let`. See source files in folder "section-5.1".

Exercise 5.4 Add `let3` to this interpreter. A `let3` expression is like a `let` expression, except that it defines exactly three variables.

Solution:

It is a multideclaration `let`. See source files in folder "section-5.1".

Exercise 5.5 Add lists to the language, as in exercise 3.9.

Solution:

See next exercise.

Exercise 5.6 Add a `list` expression to the language, as in exercise 3.10. As a hint, consider adding two new continuation-builders, one for evaluating the first element of the list and one for evaluating the rest of the list.

Solution:

The fragment of `value-of/k`,

```
...
[list-exp
 (exps)
 (if (null? exps)
     (apply-cont cont (list-val '()))
     (value-of/k (car exps) env
                 (list-cont '() (cdr exps) env cont)))]
...
```

the fragment of `apply-cont`,

```
...
[list-cont
 (vals exps env saved-cont)
 (let ([vals (append vals (list (expval->any val)))])
   (if (null? exps)
       (apply-cont saved-cont (list-val vals))
       (value-of/k (car exps) env
                   (list-cont '() (cdr exps) env saved-cont)))))]
...
```

Exercise 5.7 Add multideclaration `let` (exercise 3.16) to this interpreter.

Solution:

See source files in folder "section-5.1".

Exercise 5.8 Add multiargument procedures (exercise 3.21) to this interpreter.

Solution:

See source files in folder "section-5.1".

Exercise 5.9 Modify this interpreter to implement the IMPLICIT-REFS language. As a hint, consider including a new continuation-builder `(set-rhs-cont env var cont)`.

Solution:

See source files in folder "section-5.1".

Exercise 5.10 Modify the solution to the previous exercise so that the environment is not kept in the continuation.

Solution:

See source files in folder "section-5.1".

Exercise 5.11 Add the `begin` expression of exercise 4.4 to the continuation-passing interpreter. Be sure that no call to `value-of` or `value-of-rands` occurs in a position that would build control context.

Solution:

See source files in folder "section-5.1".

Exercise 5.12 Instrument the interpreter of figures 5.4–5.6 to produce output similar to that of the calculation on page 150.

Solution:

Elided.

Exercise 5.13 Translate the definitions of `fact` and `fact-iter` into the LETREC language. You may add a multiplication operator to the language. Then, using the instrumented interpreter of the previous exercise, compute `(fact 4)` and `(fact-iter 4)`. Compare them to the calculations at the beginning of this chapter. Find `(* 4 (* 3 (* 2 (fact 1))))` in the trace of `(fact 4)`. What is the continuation of `apply-procedure/k` for this call of `(fact 1)`?

Solution:

See source code file in "section-5.1/test.rkt".

Exercise 5.14 The instrumentation of the preceding exercise produces voluminous output. Modify the instrumentation to track instead only the size of the largest continuation used during the calculation. We measure the size of a continuation by the number of continuation-builders employed in its construction, so the size of the largest continuation in the calculation on page 150 is 3. Then calculate the values of fact and fact-iter applied to several operands. Confirm that the size of the largest continuation used by fact grows linearly with its argument, but the size of the largest continuation used by fact-iter is a constant.

Solution:

Counting is too cumbersome for the datum defined by `define-datatype`. If there is represented by list, it will be simple.

:(

Exercise 5.15 Our continuation data type contains just the single constant, `end-cont`, and all the other continuation-builders have a single continuation argument. Implement continuations by representing them as lists, where `(end-cont)` is represented by the empty list, and each other continuation is represented by a nonempty list whose car contains a distinctive data structure (called *frame* or *activation record*) and whose cdr contains the embedded continuation. Observe that the interpreter treats these lists like a stack (of frames).

Solution:

:)

See source code in file "section-5.1/list-continuation.rkt".

Exercise 5.16 Extend the continuation-passing interpreter to the language of exercise 4.22. Pass a continuation argument to `result-of`, and make sure that no call to `result-of` occurs in a position that grows a control context. Since a statement does not return a value, distinguish between ordinary continuations and continuations for statements; the latter are usually called *command continuations*. The interface should include a procedure `apply-command-cont` that takes a command continuation and invokes it. Implement command continuations both as data structures and as zero-argument procedures.

Solution:

The *statement* takes effect same as *begin-exp*.

5.2 A Trampolined Interpreter

Exercise 5.17 Modify the trampolined interpreter to wrap `(lambda () ...)` around each call (there's only one) to `apply-procedure/k`. Does this modification require changing the contracts?

Solution:

No.

Exercise 5.18 The trampoline system in figure 5.7 uses a procedural representation of a *Bounce*. Replace this by a data structure representation.

Solution:

No matter what data-structure is represented, the inner is procedure.

Exercise 5.19 Instead of placing the `(lambda () ...)` around the body of `apply-procedure/k`, place it around the body of `apply-cont`. Modify the contracts to match this change. Does the definition of *Bounce* need to change? Then replace the procedural representation of *Bounce* with a data-structure representation, as in exercise 5.18.

Solution:

No need to change.

Exercise 5.20 In exercise 5.19, the last bounce before `trampoline` returns a *FinalAnswer* is always something like `(apply-cont (end-cont) val)`, where *val* is some *ExpVal*. Optimize your representation of bounces in exercise 5.19 to take advantage of this fact.

Solution:

```
(define apply-cont
  (lambda (cont val)
    (list cont val
          (lambda ()
            (cases continuation cont
              (... val)
              (... (value-of/k ...))
              (... (apply-cont ...))
              (... (apply-procedure/k ...)))))))

(define trampoline
  (lambda (bounce)
    (cond
      [(expval? bounce)  bounce]
      [(end-cont? (car bounce)) (cadr bounce)]
      [else (trampoline ((list-ref bounce 2)))])))
```

Exercise 5.21 Implement a trampolining interpreter in an ordinary procedural language. Use a data structure representation of the snapshots as in exercise 5.18, and replace the recursive call to `trampoline` in its own body by an ordinary `while` or other looping construct.

Solution:

```
(define trampoline
  (lambda (bounce)
    (while (procedure? bounce)
      (set! bounce (bounce)))
    bounce))
```

Exercise 5.22 One could also attempt to transcribe the environment-passing interpreters of chapter 3 in an ordinary procedural language. Such a transcription would fail in all but the simplest cases, for the same reasons as suggested above. Can the technique of trampolining be used in this situation as well?

Solution:

I think that trampoline satisfies in tail-calling chain, and there could be recursive or not.

5.3 An Imperative Interpreter

Exercise 5.23 What happens if you remove the "goto" line in one of the branches of this interpreter? Exactly how does the interpreter fail?

Solution:

It returns the return of `set!`.

Exercise 5.24 Devise examples to illustrate each of the complications mentioned above.

Solution:

Comment the "goto" line to see what happen.

Exercise 5.25 Registerize the interpreter for multiargument procedures (exercise 3.21).

Solution:

See source files in folder "section-5.3".

Exercise 5.26 Convert this interpreter to a trampoline by replacing each call to `apply-procedure/k` with `(set! pc apply-procedure/k)` and using a driver that looks like

```
(define trampoline
  (lambda (pc)
    (if pc (trampoline (pc)) val)))
```

Solution:

The main code fragment,

```
(define value-of-pgm
  (lambda (pgm)
    (cases program pgm
      [a-program
       (exp1)
       (set! exp exp1)
       (set! cont (end-cont))
       (set! env (init-env))
       (trampoline (value-of/k))])))

(define pc apply-func/k)

(define trampoline
  (lambda (pc)
    (if (procedure? pc)
        (trampoline (pc))
        val)))
```

In addition, need to set the return of *call-cont*,

```
[call-cont
 (vals rands saved-env saved-cont)
       (set! vals (append vals (list val)))
       (cond
         [(null? rands)
          (set! cont saved-cont)
          (set! val (cdr vals))
          (set! f-val (expval->func (car vals)))
          apply-func/k]
         [else
          ...
```

Exercise 5.27 Invent a language feature for which setting the `cont` variable last requires a temporary variable.

Solution:

I think that temporary variable should be used for improving the readableness of code, in most coding situation especially Scheme we don't have to use any temporary variable, but this will make code hulking and need more energy to understand.

As a example, see source code file "section-5.3/evaluation.rkt" in fragment `eval-syntax` » `let-exp` or `eval-syntax` » `letrec-exp`.

Exercise 5.28 Instrument this interpreter as in exercise 5.12. Since continuations are represented the same way, reuse that code. Verify that the imperative interpreter of this section generates *exactly* the same traces as the interpreter in exercise 5.12.

Solution:

Elided.

Exercise 5.29 Apply the transformation of this section to `fact-iter` (page 139).

Solution:

See source code in file "section-5.3/fact-iter.rkt".

Exercise 5.30 Modify the interpreter of this section so that procedures rely on dynamic binding, as in exercise 3.28. As a hint, consider transforming the interpreter of exercise 3.28 as we did in this chapter; it will differ from the interpreter of this section only for those portions of the original interpreter that are different. Instrument the interpreter as in exercise 5.28. Observe that just as there is only one continuation in the state, there is only one environment that is pushed and popped, and furthermore, it is pushed and popped in parallel with the continuation. We can conclude that dynamic bindings have *dynamic extent*: that is, a binding to a formal parameter lasts exactly until that procedure returns. This is different from lexical bindings, which can persist indefinitely if they wind up in a closure.

Solution:

Elided.

In `func` constructor parameters, remove environment. In `apply-func/k`, use the current environment variable.

Exercise 5.31 Eliminate the remaining `let` expressions in this code by using additional global registers.

Solution:

See source code files in folder "section-5.3/5.31~5.33", but this is not incomplete.

Exercise 5.32 Improve your solution to the preceding exercise by minimizing the number of global registers used. You can get away with fewer than 5. You may use no data structures other than those already used by the interpreter.

Solution:

See source code files in folder "section-5.3/5.31~5.33", but this is not incomplete.

Exercise 5.33 Translate the interpreter of this section into an imperative language. Do this twice: once using zero-argument procedure calls in the host language, and once replacing each zero-argument procedure call by a `goto`. How do thes alternatives perform as the computation gets longer?

Solution:

See source code files in folder "section-5.3/5.31~5.33", but this is not incomplete.

I think this perform is quicker, because this very similar to assembly language.

Exercise 5.34 As noted on page 157, most imperative languages make it difficult to do this translation, because they use the stack for all procedure calls, even tail calls. Furthermore, for large interpreters, the pieces of code linked by `goto`'s may be too large for some compilers to handle. Translate the interpreter of this section into an imperative language, circumventing this difficulty by using the technique of trampolining, as in exercise 5.26.

Solution:

I can't solve this exercise for now :(

5.4 Exceptions

Exercise 5.35 This implementation is inefficient, because when an exception is raised, `apply-handler` must search linearly through the continuation to find a handler. Avoid this search by making the `try-cont` continuation available directly in each continuation.

Solution:

See source code files in folder "section-5.4".

Exercise 5.36 An alternative design that also avoids the linear search in `apply-handler` is to use two continuations, a normal continuation and an exception continuation. Achieve this goal by modifying the interpreter of this section to take two continuations instead of one.

Solution:

See source code files in folder "section-5.4".

Exercise 5.37 Modify the defined language to raise an exception when a procedure is called with the wrong number of arguments.

Solution:

Host language(interpreter) does this easily, but can defined language do this?

Exercise 5.38 Modify the defined language to add a division expression. Raise an exception on division by zero.

Solution:

```
letrec quotient = proc (a, b)
                   if =(b, 0)
                   then raise("divided by zero.")
                   else if <(a, b)
                        then 0
                        else +(1, quotient(-(a,b), b))
in quotient(13, 0)
```

Exercise 5.39 So far, an exception handler can propagate the exception by reraising it, or it can return a value that becomes the value of the `try` expression. One might instead design the language to allow the computation to resume from the point at which the exception was raised. Modify the interpreter of this section to accomplish this by running the body of the handler with the continuation from the point at which the `raise` was invoked.

Solution:

Add a new production:

$$\textit{Expression} \;:=\; \texttt{resume}\ \textit{Expression}$$
$$\boxed{\texttt{resume-exp (exp)}}$$

Different from `raise-exp`, `resume-exp` returns a value of *Expression*, continues the control context. The fragment of `value-of/k`,

```
...
[resume-exp (exp1)
 (value-of/k exp1 env (resum-cont cont))]
...
```

The fragment of `apply-cont`,

```
...
[resume-cont (saved-cont)
 (apply-cont saved-cont val)]
...
```

Exercise 5.40 Give the exception handlers in the defined language the ability to either return or resume. Do this by passing the continuation from the `raise` exception as a second argument. This may require adding continuations as a new kind of expressed value. Devise suitable syntax for invoking a continuation on a value.

Solution:

See source code files in folder "section-5.4".

Exercise 5.41 We have shown how to implement exceptions using a data structure representation of continuations. We can't immediately apply the recipe of section 2.2.3 to get a procedural representation, because we now have two observers: `apply-handler` and `apply-cont`. Implement the continuations of this section as a pair of procedures: a one-argument procedure representing the action of the continuation under `apply-cont`, and a zero-argument procedure representing its action under `apply-handler`.

Solution:

See source code file in "section-5.4/procedural-cont.rkt", but not complete.

Exercise 5.42 The preceding exercise captures the continuation only when an exception is raised. Add to the language the ability to capture a continuation anywhere by adding the form `letcc` *Identifier* `in` *Expression* with the specification

```
(value-of/k (letcc var body) p cont)
= (value-of/k body (extend-env var cont p) cont)
```

Such a captured continuation may be invoked with `throw`: the expression `throw` *Expression* `to` *Expression* evaluates the two subexpressions. The second expression should return a continuation, which is applied to the value of the first expression. The current continuation of the `throw` expression is ignored.

Solution:

See source code files in folder "section-5.4".

Exercise 5.43 Modify `letcc` as defined in the preceding exercise so that the captured continuation becomes a new kind of procedure, so instead of writing `throw` *exp1* `to` *exp2*, one would write `(exp2 exp1)`.

Solution:

See source code files in folder "section-5.4".

Exercise 5.44 An alternative to `letcc` and `throw` of the preceding exercises is to add a single procedure to the language. This procedure, which in Scheme is called `call-with-current-continuation`, takes a one-argument procedure, `p`, and passes to `p` a procedure that when invoked with one argument, passes that argument to the current continuation, `cont`. We could define `call-with-current-continuation` in terms of `letcc` and `throw` as follows:

```
let call-with-current-continuation
      = proc (p)
          letcc cont
          in (p proc (v) throw v to cont)
in ...
```

Add `call-with-current-continuation` to the language. Then write a translator that takes the language with `letcc` and `throw` and translates it into the language without letcc and throw, but with `call-with-current-continuation`.

Solution:

Elided.

5.5 Threads

Exercise 5.45 Add to the language of this section a construct called `yield`. Whenever a thread executes a `yield`, it is placed on the ready queue, and the thread at the head of the ready queue is run. When the thread is resumed, it should appear as if the call to `yield` had returned the number 99.

Solution:

See source code files in folder "section-5.5".

I have changed `spawn` expression to `yield`.

Exercise 5.46 In the system of exercise 5.45, a thread may be placed on the ready queue either because its time slot has been exhausted or because it chose to yield. In the latter case, it will be restarted with a full time slice. Modify the system so that the ready queue keeps track of the remaining time slice (if any) of each thread, and restarts the thread only with the time it has remaining.

Solution:

See source code files in folder "section-5.5/scheduler.rkt » `run-next-thread`".

Exercise 5.47 What happens if we are left with two subthreads, each waiting for a mutex held by the other subthread?

Solution:

These two subthreads mutually locked.

Exercise 5.48 We have used a procedural representation of threads. Replace this by a data-structure representation.

Solution:

See source code in file "section-5.5/5.48.rkt".

Exercise 5.49 Do exercise 5.15 (continuations as a stack of frames) for THREADS.

Solution:

See source code in file "section-5.5/scheduler.rkt".

Exercise 5.50 Registerize the interpreter of this section. What is the set of mutually tail-recursive procedures that must be registerized?

Solution:

See source code in file "section-5.5/evaluation.rkt".

Exercise 5.51 We would like to be able to organize our program so that the consumer in figure 5.17 doesn't have to busy-wait. Instead, it should be able to put itself to sleep and be awakened when the producer has put a value in the buffer. Either write a program with mutexes to do this, or implement a synchronization operator that makes this possible.

Solution:

This problem can be solved by `mutex` and `wait-end`(next exercise).

1. *before, after* in main thread, orderly executed.
2. *before, after* in different subthreads, ensured spawning *before*'s subthread before *after*'s, then used `mutex`.
3. *before* in main thread, *after* in subthread, ensured spawing *after*'s subthread after *before*.
4. *before* in subthread, *after* in main thread, used `wait-end`.

Exercise 5.52 Write a program using mutexes that will be like the program in figure 5.21, except that the main thread waits for all three of the subthreads to terminate, and then returns the value of `x`.

Solution:

See source code files in folder "section-5.5".

Exercise 5.53 Modify the thread package to include thread identifiers. Each new thread is associated with a fresh thread identifier. When the child thread is spawned, it is passed its thread identifier as a value, rather than the arbitrary value 28 used in this section. The child's number is also returned to the parent as the value of the spawn expression. Instrument the interpreter to trace the creation of thread identifiers. Check to see that the ready queue contains at most one thread for each thread identifier. How can a child thread know its parent's identifier? What should be done about the thread identifier of the original program?

Solution:

See source code files in folder "section-5.5/5.33~5.56", but not complete.

Exercise 5.54 Add to the interpreter of exercise 5.53 a kill facility. The kill construct, when given a thread number, finds the corresponding thread on the ready queue or any of the waiting queues and removes it. In addition, kill should return a true value if the target thread is found and false if the thread number is not found on any queue.

Solution:

See source code files in folder "section-5.5/5.33~5.56", but not complete.

Exercise 5.55 Add to the interpreter of exercise 5.53 an interthread communication facility, in which each thread can send a value to another thread using its thread identifier. A thread can receive messages when it chooses, blocking if no message has been sent to it.

Solution:

See source code files in folder "section-5.5/5.33~5.56", but not complete.

Exercise 5.56 Modify the interpreter of exercise 5.55 so that rather than sharing a store, each thread has its own store. In such a language, mutexes can almost always be avoided. Rewrite the example of this section in this language, without using mutexes.

Solution:

See source code files in folder “section-5.5/5.33~5.56”, but not complete.

Exercise 5.57 There are lots of different synchronization mechanisms in your favorite OS book. Pick three and implement them in this framework.

Solution:

Elided.

I haven't read any OS books yet :)

Exercise 5.58 Go off with your friends and have some pizza, but make sure only one person at a time grabs a piece!

Solution:

I have a pizza only myself, but I can grab a piece by one hand at a time.

6 *Continuation-Passing Style*

6.1 Writing Programs in Continuation-Passing Style

Exercise 6.1 Consider figure 6.2 without `(set! pc fact/k)` in the definition of `fact/k` and without `(set! pc apply-cont)` in the definition of `apply-cont`. Why does the program still work?

Solution:

Calling-chain not changed.

Exercise 6.2 Prove by induction on n that for any `g`, `(fib/k n g) = (g (fib n))`.

Solution:

```
(fib/k (+ n 1) g)
= (fib/k n (lambda (v1) (fib/k (- n 1)
                               (lambda (v2) (g (+ v1 v2))))))
= ((lambda (v1) (fib/k (- n 1) (lambda (v2) (g (+ v1 v2)))))
   (fib/k n))
= (fib/k (- n 1) (lambda (v2) (g (+ (fib/k n) v2))))
= ((lambda (v2) (g (+ (fib/k n) v2)))
   (fib/k (- n 1))
=(g (+ (fib/k n) (fib/k (- n 1))))
=(g (fib (+ n 1)))
```

Exercise 6.3 Rewrite each of the following Scheme expressions in continuationpassing style. Assume that any unknown functions have also been rewritten in CPS.

1. `(lambda (x y) (p (+ 8 x) (q y)))`
2. `(lambda (x y u v) (+ 1 (f (g x y) (+ u v))))`
3. `(+ 1 (f (g x y) (+ u (h v))))`
4. `(zero? (if a (p x) (p y)))`
5. `(zero? (if (f a) (p x) (p y)))`
6. `(let ((x (let ((y 8)) (p y)))) x)`
7. `(let ((x (if a (p x) (p y)))) x)`

Solution:

1.
```
(lambda (x y cont)
  (q y (lambda (v1)
         (p (+ 8 x) v1 cont))))
```

2.
```
(lambda (x y u v cont)
  (g x y (lambda (v1)
           (f v1 (+ u v) (lambda (v2)
                           (cont (+ 1 v2)))))))
```

3.
```
(define end (lambda (val) val))

(h v (lambda (v1)
       (g x y (lambda v2)
          (f v2 (+ u v1) (lambda (v3)
                           (end (+ 1 v3)))))))
```

4.
```
(zero? (if a (p x end) (p y end)))
```

5.
```
(f a (lambda (v1)
       (zero? (if v1 (p x end) (p y end)))))
```

6.
```
(let ([y 8])
  (p y (lambda (v1)
         (let ([x v1]) (end x)))))
```

7.
```
(let ([x (if a (p x end) (p y end))]) x)
```

Exercise 6.4 Rewrite each of the following procedures in continuation-passing style. For each procedure, do this first using a data-structure representation of continuations, then with a procedural representation, and then with the inlined procedural representation. Last, write the registerized version. For each of these four versions, test to see that your implementation is tail-recursive by defining `end-cont` by

```
(apply-cont (end-cont) val)
= (begin
    (eopl:printf "End of computation.~%")
    (eopl:printf "This sentence should appear only once.~%")
    val)
```

as we did in chapter 5.

1. `remove-first` (section 1.2.3).
2. `list-sum` (section 1.3).
3. `occurs-free?` (section 1.2.4).
4. `subst` (section 1.2.5).

Solution:

See source code files in folder "section-6.1/6.4". Elided data-structuer representation.

Exercise 6.5 When we rewrite an expression in CPS, we choose an evaluation order for the procedure calls in the expression. Rewrite each of the preceding examples in CPS so that all the procedure calls are evaluated from right to left.

Solution:

See source code files in folder "section-6.1/6.5".

Exercise 6.6 How many different evaluation orders are possible for the procedure calls in `(lambda (x y) (+ (f (g x)) (h (j y))))`? For each evaluation order, write a CPS expression that calls the procedures in that order.

Solution:

About 4.

Exercise 6.7 Write out the procedural and the inlined representations for the interpreter in figures 5.4, 5.5, and 5.6.

Solution:

See source code in file "section-6.1/6.7.rkt".

Exercise 6.8 Rewrite the interpreter of section 5.4 using a procedural and inlined representation. This is challenging because we effectively have two observers, `apply-cont` and `apply-handler`. As a hint, consider modifying the recipe on page 6.1 so that we add to each procedure two extra arguments, one representing the behavior of the continuation under `apply-cont` and one representing its behavior under `apply-handler`.

Solution:

See source code in file "section-6.1/6.8.rkt".

Exercise 6.9 What property of multiplication makes this program optimization possible?

Solution:

It can accumulate returned values into a argument.

Exercise 6.10 For `list-sum`, formulate a succinct representation of the continuations, like the one for `fact/k` above.

Solution:

See source code in file "section-6.1/6.10.rkt".

6.2 Tail Form

Exercise 6.11 Complete the interpreter of figure 6.6 by writing `value-of-simple-exp`.

Solution:

See source code in file "section-6.2/evaluation.rkt".

Exercise 6.12 Determine whether each of the following expressions is simple.

1. `-((f -(x,1)),1)`
2. `(f -(-(x,y),1))`
3. `if zero?(x) then -(x,y) else -(-(x,y),1)`
4. `let x = proc (y) (y x) in -(x,3)`
5. `let f = proc (x) x in (f 3)`

Solution:

1. no
2. yes
3. yes
4. no
5. yes

Exercise 6.13 Translate each of these expressions in CPS-IN into continuationpassing style using the CPS recipe on page 200 above. Test your transformed programs by running them using the interpreter of figure 6.6. Be sure that the original and transformed versions give the same answer on each input.

1. `removeall`.

```
letrec
  removeall(n,s) =
    if null?(s)
    then emptylist
    else if number?(car(s))
         then if equal?(n,car(s))
              then (removeall n cdr(s))
              else cons(car(s),(removeall n cdr(s)))
         else cons((removeall n car(s)),
                   (removeall n cdr(s)))
```

2. `occurs-in?`.

```
letrec
  occurs-in?(n,s) =
    if null?(s)
    then 0
    else if number?(car(s))
         then if equal?(n,car(s))
              then 1
              else (occurs-in? n cdr(s))
         else if (occurs-in? n car(s))
              then 1
              else (occurs-in? n cdr(s))
```

3. `remfirst`. This uses `occurs-in?` from the preceding example.

```
letrec
  remfirst(n,s) =
    letrec
      loop(s) =
        if null?(s)
        then emptylist
        else if number?(car(s))
             then if equal?(n,car(s))
                  then cdr(s)
                  else cons(car(s),(loop cdr(s)))
             else if (occurs-in? n car(s))
                  then cons((remfirst n car(s)),cdr(s))
                  else cons(car(s),(remfirst n cdr(s)))
in (loop s)
```

4. `depth`.

```
letrec
  depth(s) =
    if null?(s)
    then 1
    else if number?(car(s))
         then (depth cdr(s))
         else if less?(add1((depth car(s))),
                       (depth cdr(s)))
              then (depth cdr(s))
              else add1((depth car(s)))
```

5. `depth-with-let`.

```
letrec
  depth(s) =
    if null?(s)
    then 1
    else if number?(car(s))
         then (depth cdr(s))
         else let dfirst = add1((depth car(s)))
                  drest = (depth cdr(s))
              in if less?(dfirst,drest)
                 then drest
                 else dfirst
```

6. `map`.

```
letrec
  map(f, l) = if null?(l)
              then emptylist
              else cons((f car(l)),(map f cdr(l)))
  square(n) = *(n,n)
in (map square list(1,2,3,4,5))
```

7. `fnlrgtn`. This procedure takes an n-list, like an s-list (page 9), but with number instead of symbols, and a number `n` and returns the first number in the list (in leftto-right order) that is greater than `n`. Once the result is found, no further elements in the list are examined. For example,

```
(fnlrgtn list(1,list(3,list(2),7,list(9)))
 6)
```

finds 7.

8. `every`. This procedure takes a predicate and a list and returns a true value if and only if the predicate holds for each list element.

```
letrec
  every(pred, l) =
    if null?(l)
    then 1
    else if (pred car(l))
         then (every pred cdr(l))
         else 0
in (every proc(n)greater?(n,5) list(6,7,8,9))
```

Solution:

Elided.

Exercise 6.14 Complete the interpreter of figure 6.6 by supplying definitions for `value-of-program` and `apply-cont`.

Solution:

See source code in file "section-6.2/evaluation.rkt".

Exercise 6.15 Observe that in the interpreter of the preceding exercise, there is only one possible value for `cont`. Use this observation to remove the `cont` argument entirely.

Solution:

See source code in file "section-6.2/evaluation.rkt".

Exercise 6.16 Registerize the interpreter of figure 6.6.

Solution:

See source code in file "section-6.2/evaluation.rkt".

Exercise 6.17 Trampoline the interpreter of figure 6.6.

Solution:

See source code in file "section-6.2/evaluation.rkt".

Exercise 6.18 Modify the grammar of CPS-OUT so that a simple `diff-exp` or `zero?-exp` can have only a constant or variable as an argument. Thus in the resulting language `value-of-simple-exp` can be made nonrecursive.

Solution:

Elided.

Exercise 6.19 Write a Scheme procedure `tail-form?` that takes the syntax tree of a program in CPS-IN, expressed in the grammar of figure 6.3, and determines whether the same string would be in tail form according to the grammar of figure 6.5.

Solution:

See source code in file "section-6.2/tail-form.rkt".

6.3 Converting to Continuation-Passing Style

Exercise 6.20 Our procedure `cps-of-exps` causes subexpressions to be evaluated from left to right. Modify `cps-of-exps` so that subexpressions are evaluated from right to left.

Solution:

```
...
(let ([pos (list-index (lambda (exp)
                         (not (simple-exp? exp)))
                       (reverse exps))])
...
```

Exercise 6.21 Modify `cps-of-call-exp` so that the operands are evaluated from left to right, followed by the operator.

Solution:

No need to modify, see source code in folder "section-6.3".

Exercise 6.22 Sometimes, when we generate $(K\ simp)$, K is already a `proc-exp`. So instead of generating

```
(proc (var1) ... simp)
```

we could generate

```
let var1 = simp
in ...
```

This leads to CPS code with the property that it never contains a subexpression of the form

```
(proc (var) exp1
 simp)
```

unless that subexpression was in the original expression.

Modify make-send-to-cont to generate this better code. When does the new rule apply?

Solution:

See source code in file "section-6.3/cps-of.rkt » `make-send-to-cont`".

Exercise 6.23 Observe that our rule for `if` makes two copies of the continuation K, so in a nested `if` the size of the transformed program can grow exponentially. Run an example to confirm this observation. Then show how this may be avoided by changing the transformation to bind a fresh variable to K.

Solution:

See source code in file "section-6.3/cps-of.rkt » `cps-of-exp` » `if-exp`".

Exercise 6.24 Add lists to the language (exercise 3.10). Remember that the arguments to a list are not in tail position.

Solution:

Elided.

Exercise 6.25 Extend CPS-IN so that a `let` expression can declare an arbitrary number of variables (exercise 3.16).

Solution:

See soure code in file "section-6.3/cps-of.rkt » `cps-of-exp` » `let-exp`".

Exercise 6.26 A continuation variable introduced by `cps-of-exps` will only occur once in the continuation. Modify `make-send-to-cont` so that instead of generating

```
let var1 = simp1
in T
```

as in exercise 6.22, it generates $T[simp1/var1]$, where the notation $E1[E2/var]$ means expression E1 with every free occurrence of the variable var replaced by E2.

Solution:

If I understand this correctly, it havd been solved in exercise 6.22.

Exercise 6.27 As it stands, `cps-of-let-exp` will generate a useless `let` expression. (Why?) Modify this procedure so that the continuation variable is the same as the `let` variable. Then if *exp1* is nonsimple,

```
(cps-of-exp <<let var1 = exp1 in exp2>> K)
= (cps-of-exp exp1 <<proc (var1) (cps-of-exp exp2 K)>>
```

Solution:

Elided.

`let-exp` and `letrec-exp` are very similar to

```
((lambda (var1 var2)
   body)
 exp1
 exp2)
```

Exercise 6.28 Food for thought: imagine a CPS transformer for Scheme programs, and imagine that you apply it to the first interpreter from chapter 3. What would the result look like?

Solution:

My interpreter interprets the code of Scheme-like, if do this correctly, no doubt, it will procduct same value.

Exercise 6.29 Consider this variant of `cps-of-exps`.

```
(define cps-of-exps
  (lambda (exps builder)
    (let cps-of-rest ((exps exps) (acc '()))
 ;; cps-of-rest : Listof(InpExp) × Listof(SimpleExp) → TfExp
      (cond
        ((null? exps) (builder (reverse acc)))
        ((inp-exp-simple? (car exps))
         (cps-of-rest
          (cdr exps)
          (cons (cps-of-simple-exp (car exps)) acc)))
        (else
         (let ((var (fresh-identifier 'var)))
           (cps-of-exp
            (car exps)
            (cps-proc-exp
             (list var)
             (cps-of-rest (cdr exps)
                          (cons
                           (cps-of-simple-exp (var-exp var))
                           acc)))))))))))
```

Why is this variant of `cps-of-exp` more efficient than the one in figure 6.8?

Solution:

No need `list-index` to test simple-exp.

Exercise 6.30 A call to `cps-of-exps` with a list of expressions of length one can be simplified as follows:

```
(cps-of-exps (list exp) builder)
= (cps-of-exp/ctx exp (lambda (simp) (builder (list simp))))
```

where

cps-of-exp/ctx : $InpExp \times (SimpleExp \to TfExp) \to TfExp$

```
(define cps-of-exp/ctx
  (lambda (exp context)
    (if (inp-exp-simple? exp)
        (context (cps-of-simple-exp exp))
        (let ((var (fresh-identifier 'var)))
          (cps-of-exp exp
                      (cps-proc-exp (list var)
                                    (context (cps-var-exp
var))))))))
```

Thus, we can simplify occurrences of `(cps-of-exps (list ...))`, since the number of arguments to list is known. Therefore the definition of, for example, `cps-of-diff-exp` could be defined with `cps-of-exp/ctx` instead of with `cps-of-exps`.

```
(define cps-of-diff-exp
  (lambda (exp1 exp2 k-exp)
    (cps-of-exp/ctx
     exp1
     (lambda (simp1)
       (cps-of-exp/ctx
        exp2
        (lambda (simp2)
          (make-send-to-cont
           k-exp
           (cps-diff-exp simp1 simp2))))))))
```

For the use of `cps-of-exps` in `cps-of-call-exp`, we can use `cps-of-exp/ctx` on the `rator`, but we still need `cps-of-exps` for the `rands`. Remove all other occurrences of `cps-of-exps` from the translator.

Solution:

Elided.

Exercise 6.31 Write a translator that takes the output of `cps-of-program` and produces an equivalent program in which all the continuations are represented by data structures, as in chapter 5. Represent data structures like those constructed using `define-datatype` as lists. Since our language does not have symbols, you can use an integer tag in the car position to distinguish the variants of a data type.

Solution:

See source code in file "section-6.3/maybe-bad.rkt".

Exercise 6.32 Write a translator like the one in exercise 6.31, except that it represents all procedures by data structures.

Solution:

See source code in file "section-6.3/maybe-bad.rkt".

Exercise 6.33 Write a translator that takes the output from exercise 6.32 and converts it to a register program like the one in figure 6.1.

Solution:

Elided.

Exercise 6.34 When we convert a program to CPS, we do more than produce a program in which the control contexts become explicit. We also choose the exact order in which the operations are done, and choose names for each intermediate result. The latter is called *sequentialization*. If we don't care about obtaining iterative behavior, we can sequentialize a program by converting it to *A-normal* form or *ANF*. Here's an example of a program in ANF.

```
(define fib/anf
  (lambda (n)
    (if (< n 2)
        1
        (let ((val1 (fib/anf (- n 1))))
          (let ((val2 (fib/anf (- n 2))))
            (+ val1 val2))))))
```

Whereas a program in CPS sequentializes computation by passing continuations that name intermediate results, a program in ANF sequentializes computation by using let expressions that name all of the intermediate results.

Retarget `cps-of-exp` so that it generates programs in ANF instead of CPS. (For conditional expressions occurring in nontail position, use the ideas in exercise 6.23.)Then, show that applying the revised `cps-of-exp` to, e.g., the definition of `fib` yields the definition of `fib/anf`. Finally, show that given an input program which is already in ANF, your translator produces the same program except for the names of bound variables.

Solution:

See source code in file "section-6.3/anf-of.rkt".

Exercise 6.35 Verify on a few examples that if the optimization of exercise 6.27 is installed, CPS-transforming the output of your ANF transformer (exercise 6.34) on a program yields the same result as CPS-transforming the program.

Solution:

See source code example in file "section-6.3/test.rkt".

6.4 Modeling Computational Effects

Exercise 6.36 Add a `begin` expression (exercise 4.4) to CPS-IN. You should not need to add anything to CPS-OUT.

Solution:

See source code in folder "section-6.4".

Exercise 6.37 Add implicit references (section 4.3) to CPS-IN. Use the same version of CPS-OUT, with explicit references, and make sure your translator inserts allocation and dereference where necessary. As a hint, recall that in the presence of implicit references, a var-exp is no longer simple, since it reads from the store.

Solution:

See source code in folder "section-6.4".

I used some different way to impletment this.

Exercise 6.38 If a variable never appears on the left-hand side of a set expression, then it is immutable, and could be treated as simple. Extend your solution to the preceding exercise so that all such variables are treated as simple.

Solution:

Elided.

Exercise 6.39 Implement `letcc` and `throw` in the CPS translator.

Solution:

See source code in folder "section-6.4".

Exercise 6.40 Implement `try/catch` and throw from section 5.4 by adding them to the CPS translator. You should not need to add anything to CPS-OUT. Instead, modify `cps-of-exp` to take two continuations: a success continuation and an error continuation.

Solution:

Elided.

7 Types

7.1 Values and Their Types

Exercise 7.1 Below is a list of closed expressions. Consider the value of each expression. For each value, what type or types does it have? Some of the values may have no type that is describable in our type language.

1. `proc (x) -(x,3)`
2. `proc (f) proc (x) -((f x), 1)`
3. `proc (x) x`
4. `proc (x) proc (y) (x y)`
5. `proc (x) (x 3)`
6. `proc (x) (x x)`
7. `proc (x) if x then 88 else 99`
8. `proc (x) proc (y) if x then y else 99`
9. ```
   (proc (p) if p then 88 else 99
     33)
   ```
10. ```
    (proc (p) if p then 88 else 99
      proc (z) z)
    ```
11. ```
    proc (f)
     proc (g)
      proc (p)
       proc (x)
        if (p (f x)) then (g 1) else -((f x),1)
    ```
12. ```
    proc (x)
     proc(p)
      proc (f)
       if (p x) then -(x,1) else (f p)
    ```
13. ```
    proc (f)
     let d = proc (x)
              proc (z)
               ((f (x x)) z)
     in proc (n)
         ((f (d d)) n)
    ```

Solution:
    ```

1. `(int -> int)`
2. `((t -> int) -> (t -> int))`
3. `(t -> t)`
4. `((t -> t) -> (t -> t))`
5. `((int -> t) -> t)`
6. `((p -> t) -> t)`, recursive in `p`, no type.
7. `(bool -> int)`
8. `(bool -> (int -> int))`
9. `(bool -> int)`, but wrong argument type `int`.
10. `(bool -> int)`, but wrong argument type `(t -> t)`.
11.
```
((t -> int)
  -> ((int -> int)
        -> ((int -> bool)
              -> (t -> int))))
```
12. `(int -> ((int -> bool) -> (((int -> bool) -> int) -> int)))`
13. `((t -> (t -> t)) -> (t -> t))`

Exercise 7.2 Are there any expressed values that have exactly two types according to definition 7.1.1?

Solution:

No.

Exercise 7.3 For the language LETREC, is it decidable whether an expressed value val is of type *t*?

Solution:

Yes.

7.2 Assigning a Type to an Expression

Exercise 7.4 Using the rules of this section, write derivations, like the one on page 5, that assign types for `proc (x) x` and `proc (x) proc (y) (x y)`. Use the rules to assign at least two types for each of these terms. Do the values of these expressions have the same types?

Solution:

```
                                x ∈ int
                          proc (x) x ∈ (int -> int)

                                x ∈ bool
                          proc (x) x ∈ (bool -> bool)

                                y ∈ int
                             (x y) ∈ t
                     proc (y) (x y) ∈ (int -> t)
               proc (x) proc (y) (x y) ∈ ((int -> t) -> t)
```

7.3 CHECKED: A Type-Checked Language

Exercise 7.5 Extend the checker to handle multiple `let` declarations, multiargument procedures, and multiple `letrec` declarations. You will need to add types of the form `(t1 * t2 * ... * tn -> t)` to handle multiargument procedures.

Solution:

See source code in folde "section-7.3".

Exercise 7.6 Extend the checker to handle assignments (section 4.3).

Solution:

See source code in folde "section-7.3".

Exercise 7.7 Change the code for checking an `if-exp` so that if the test expression is not a boolean, the other expressions are not checked. Give an expression for which the new version of the checker behaves differently from the old version.

Solution:

```
(define type-of
  (lambda (exp tenv)
    (cases expression exp
      ...
      [if-exp
       (exp1 exp2 exp3)
       (let ([t2 (type-of exp2 tenv)]
             [t3 (type-of exp3 tenv)])
         ;; no need to check test type
         (check-equal-type! t2 t3 (list exp2 exp3))
         t2)]
      ...
      )))
```

```
(define value-of
  (lambda (exp env)
    (cases expression exp
      ...
      [if-exp
       (exp1 exp2 exp3)
       (let* ([test (expval->any (value-of exp1 env))]
              [test (if (boolean? test) test #t)])
         (if test
             (value-of exp2 env)
             (value-of exp3 env)))]
      ...
      )))
```

Exercise 7.8 Add `pairof` types to the language. Say that a value is of type `pairof` t_1 `*` t_2 if and only if it is a pair consisting of a value of type t_1 and a value of type t_2. Add to the language the following productions:

$$\begin{aligned}
\textit{Type} &::= \texttt{pairof}\ \ \textit{Type}\ \ \texttt{*}\ \ \textit{Type} \\
&\quad \boxed{\texttt{pair-type (ty1 ty2)}} \\
\textit{Expression} &::= \texttt{newpair}\ \ (\textit{Expression}\ \ \texttt{,}\ \ \textit{Expression}) \\
&\quad \boxed{\texttt{pair-exp (exp1 exp2)}} \\
\textit{Expression} &::= \texttt{unpair}\ \ \textit{Identifier}\ \ \textit{Identifier}\ \ \texttt{=}\ \ \textit{Expression} \\
&\quad \texttt{in}\ \ \textit{Expression} \\
&\quad \boxed{\texttt{unpair-exp (var1 var2 exp body)}}
\end{aligned}$$

A `pair` expression creates a pair; an `unpair` expression (like exercise 3.18) binds its two variables to the two parts of the expression; the scope of these variables is `body`. The typing rules for `pair` and `unpair` are:

$$\frac{\begin{array}{c}\texttt{(type-of}\ \ e_1\ \ tenv\texttt{)} = t_1 \\ \texttt{(type-of}\ \ e_2\ \ tenv\texttt{)} = t_2\end{array}}{\texttt{(type-of (pair-exp}\ \ e_1\ \ e_2\texttt{)}\ \ tenv\texttt{)} = \texttt{pairof}\ \ t_1\ \ \texttt{*}\ \ t_2}$$

$$\frac{\begin{array}{c}\texttt{(type-of}\ \ e_{pair}\ \ tenv\texttt{)} = \texttt{(pairof}\ \ t_1\ \ t_2\texttt{)} \\ \texttt{(type-of}\ \ e_{body}\ [var_1 = t_1]\ \ [var_2 = t_2]tenv\texttt{)} = t_{body}\end{array}}{\texttt{(type-of (unpair-exp}\ \ var_1\ \ var_2\ \ e_1\ \ e_{body}\texttt{)}\ \ tenv\texttt{)} = t_{body}}$$

Extend CHECKED to implement these rules. In `type-to-external-form`, produce the list `(pairof t1 t2)` for a pair type.

Solution:

Elided.

Exercise 7.9 Add `listof` types to the language, with operations similar to those of exercise 3.9. A value is of type `listof` t if and only if it is a list and all of its elements are of type t. Extend the language with the productions

$$
\begin{array}{rcl}
\textit{Type} & ::= & \texttt{listof}\ \ \textit{Type} \\
 & & \boxed{\texttt{list-type (ty)}} \\
\textit{Expression} & ::= & \texttt{list}\ \ (\textit{Expression}\ \ \{,\textit{Expression}\}^*\ ) \\
 & & \boxed{\texttt{list-exp (exp1 exps)}} \\
\textit{Expression} & ::= & \texttt{cons}\ \ (\textit{Expression}\ \ ,\ \ \textit{Expression}) \\
 & & \boxed{\texttt{cons-exp (exp1 exp2)}} \\
\textit{Expression} & ::= & \texttt{null?}\ \ (\textit{Expression}) \\
 & & \boxed{\texttt{null-exp (exp1)}} \\
\textit{Expression} & ::= & \texttt{emptylist}\ \ \textit{Type} \\
 & & \boxed{\texttt{emptylist-exp (ty)}}
\end{array}
$$

with types given by the following four rules:

$$
\frac{\begin{array}{c}(\texttt{type-of}\ \ e_1\ \ tenv) = t \\ (\texttt{type-of}\ \ e_2\ \ tenv) = t \\ \vdots \\ (\texttt{type-of}\ \ e_n\ \ tenv) = t\end{array}}{(\texttt{type-of}\ (\texttt{list-exp}\ \ e_1\ \ (e_2\ \ \ldots\ \ e_n))\ \ tenv) = \texttt{listof}\ \ t}
$$

$$
\frac{\begin{array}{c}(\texttt{type-of}\ \ e_1\ \ tenv) = t \\ (\texttt{type-of}\ \ e_2\ \ tenv) = \texttt{listof}\ t\end{array}}{(\texttt{type-of}\ \texttt{cons}(e_1,e_2)\ \ tenv) = \texttt{listof}\ \ t}
$$

$$
\frac{(\texttt{type-of}\ \ e_1\ \ tenv) = \texttt{listof}\ t}{(\texttt{type-of}\ \texttt{null?}(e_1)\ \ tenv) = \texttt{bool}}
$$

$$
(\texttt{type-of}\ \texttt{emptylist}\ [t]\ \ tenv) = \texttt{listof}\ \ t
$$

Although `cons` is similar to pair, it has a very different typing rule.

Write similar rules for `car` and `cdr`, and extend the checker to handle these as well as the other expressions. Use a trick similar to the one in exercise 7.8 to avoid conflict with `proc-type-exp`. These rules should guarantee that `car` and `cdr` are applied to lists, but they should not guarantee that the lists be non-empty. Why would it be unreasonable for the rules to guarantee that the lists be non-empty? Why is the type parameter in `emptylist` necessary?

Solution:

If use `cons(e1, e2)` start up a list and `e2` is a `emptylist`, then `e2` contained type must be same as `e1`. Even `e2` is a `emptylist`, it must return a type to be checked.

Elided.

Recommend reading The Typed Racket Guide in Racket Documentation.

Exercise 7.10 Extend the checker to handle EXPLICIT-REFS. You will need to do the following:

- Add to the type system the types `refto` t, where t is any type. This is the type of references to locations containing a value of type t. Thus, if e is of type t, `(newref e)` is of type `refto` t.
- Add to the type system the type `void`. This is the type of the value returned by `setref`. You can't apply any operation to a value of type `void`, so it doesn't matter what value `setref` returns. This is an example of types serving as an information-hiding mechanism.
- Write down typing rules for `newref`, `deref`, and `setref`.
- Implement these rules in the checker.

Solution:

$$\frac{\texttt{(type-of}\ \ e\ \ tenv\texttt{)} = t}{\texttt{(type-of (newref}\ e\texttt{)}\ \ tenv\texttt{)} = \texttt{refto}\ t}$$

$$\frac{\texttt{(type-of}\ \ e\ \ tenv\texttt{)} = \texttt{refto}\ t}{\texttt{(type-of (deref}\ e\texttt{)}\ \ tenv\texttt{)} = t}$$

$$\frac{\begin{array}{c}\texttt{(type-of}\ \ e_1\ \ tenv\texttt{)} = \texttt{refto}\ t \\ \texttt{(type-of}\ \ e_2\ \ tenv\texttt{)} = t\end{array}}{\texttt{(type-of (setref}\ e_1\ \ e_2\texttt{)}\ \ tenv\texttt{)} = \texttt{void}}$$

Code implementing elided.

Exercise 7.11 Extend the checker to handle MUTABLE-PAIRS.

Solution:

Elided.

7.4 INFERRED: A Language with Type Inference

Exercise 7.12 Using the methods in this section, derive types for each of the expressions in exercise 7.1, or determine that no such type exists. As in the other examples of this section, assume there is a ? attached to each bound variable.

Solution:

1. `proc (x) -(x,3)` $t_0 = t_x \to t_1$
   `-(x,3)` $t_1 = \texttt{int}$
   $t_x = \texttt{int}$
   $t_0 = \texttt{int} \to \texttt{int}$

2. `proc (f) proc (x) -((f x), 1)` $t_0 = t_f \to t_1$
   `proc (x) -((f x), 1)` $t_1 = t_x \to t_2$
   `-((f x), 1)` $t_3 = \texttt{int}$
   $t_2 = \texttt{int}$
   `(f x)` $t_f = t_x \to t_3$

Equations	Substitution
$t_0 = t_f \to t_1$	
$t_1 = t_x \to t_2$	
$t_3 = \texttt{int}$	
$t_2 = \texttt{int}$	
$t_f = t_x \to t_3$	

Equations	Substitution
$t_1 = t_x \to t_2$	$t_0 = t_f \to t_1$
$t_3 = \texttt{int}$	
$t_2 = \texttt{int}$	
$t_f = t_x \to t_3$	

Equations	Substitution
$t_3 = \texttt{int}$	$t_0 = t_f \to (t_x \to t_2)$
$t_2 = \texttt{int}$	$t_1 = t_x \to t_2$
$t_f = t_x \to t_3$	

Equations	Substitution
$t_2 = \texttt{int}$	$t_0 = t_f \to (t_x \to t_2)$
$t_f = t_x \to t_3$	$t_1 = t_x \to t_2$
	$t_3 = \texttt{int}$

Equations	Substitution
$t_f = t_x \to t_3$	$t_0 = t_f \to (t_x \to \texttt{int})$
	$t_1 = t_x \to \texttt{int}$
	$t_3 = \texttt{int}$
	$t_2 = \texttt{int}$

Equations	Substitution
	$t_0 = (t_x \to \texttt{int}) \to (t_x \to \texttt{int})$
	$t_1 = t_x \to \texttt{int}$
	$t_3 = \texttt{int}$
	$t_2 = \texttt{int}$
	$t_f = t_x \to \texttt{int}$

3. **Equations** **Substitution**
$t_0 = t_x \to t_x$

4. **Equations** **Substitution**
$t_0 = (t_y \to t_2) \to (t_y \to t_2)$
$t_1 = t_y \to t_2$
$t_x = t_y \to t_2$

5. **Equations** **Substitution**
$t_0 = (\text{int} \to t_1) \to t_1$
$t_x = \text{int} \to t_1$

6. **Equations** **Substitution**
$t_0 = (t_x \to t_1) \to t_1$
$t_x = t_x \to t_1$

7. **Equations** **Substitution**
$t_0 = \text{bool} \to \text{int}$
$t_x = \text{bool}$
$t_1 = \text{int}$

8. **Equations** **Substitution**
$t_0 = \text{bool} \to (\text{int} \to \text{int})$
$t_1 = \text{int} \to \text{int}$
$t_x = \text{bool}$
$t_y = \text{int}$
$t_2 = \text{int}$

9. **Equations** **Substitution**
$t_0 = \text{int}$
$t_1 = \text{int} \to \text{int}$
$t_1 = \text{bool} \to \text{int}$
$t_p = \text{bool}$
$t_2 = \text{int}$

10. **Equations** **Substitution**
$t_0 = \text{int}$
$t_1 = \text{bool} \to \text{int}$
$t_2 = \text{bool}$
$t_3 = \text{int}$
$t_p = \text{bool}$
$t_z = \text{bool}$

11.

Equations	Substitution
$t_0 = (t_x \to \text{int}) \to ((\text{int} \to \text{int}) \to ((\text{int} \to \text{bool}) \to (t_x \to \text{int})))$	
$t_1 = (\text{int} \to \text{int}) \to ((\text{int} \to \text{bool}) \to (t_x \to \text{int}))$	
$t_2 = (\text{int} \to \text{bool}) \to (t_x \to \text{int})$	
$t_3 = t_x \to \text{int}$	
$t_4 = \text{int}$	
$t_5 = \text{int}$	
$t_6 = \text{bool}$	
$t_7 = \text{int}$	
$t_f = t_x \to \text{int}$	
$t_g = \text{int} \to \text{int}$	
$t_p = \text{int} \to \text{bool}$	

12.

Equations	Substitution
$t_0 = \text{int} \to ((\text{int} \to \text{bool}) \to (((\text{int} \to \text{bool}) \to \text{int}) \to \text{int}))$	
$t_1 = (\text{int} \to \text{bool}) \to (((\text{int} \to \text{bool}) \to \text{int}) \to \text{int})$	
$t_2 = ((\text{int} \to \text{bool}) \to \text{int}) \to \text{int}$	
$t_3 = \text{int}$	
$t_4 = \text{bool}$	
$t_p = \text{int} \to \text{bool}$	
$t_x = \text{int}$	
$t_5 = \text{int}$	
$t_f = (\text{int} \to \text{bool}) \to \text{int}$	

13. It makes my head spin.

Equations	Substitution
$t_0 = (t_5 \to t_4) \to (t_n \to t_3)$	
$t_1 = (t_x \to t_5) \to (t_z \to t_3)$	
$t_2 = t_z \to t_3$	
$t_6 = t_n \to t_3$	
$t_x = t_x \to t_5$	
$t_f = t_5 \to t_4$	

Exercise 7.13 Write down a rule for doing type inference for `let` expressions. Using your rule, derive types for each of the following expressions, or determine that no such type exists.

1. `let x = 4 in (x 3)`
2. `let f = proc (z) z in proc (x) -((f x), 1)`
3. `let p = zero?(1) in if p then 88 else 99`
4. `let p = proc (z) z in if p then 88 else 99`

Solution:

```
(let-exp var exp body) : t_var = t_exp
                         tenv[t_var]
                         t_let-exp = t_body
```

1. no type.
2. int → int.
3. int.
4. no type.

Exercise 7.14 What is wrong with this expression?

```
letrec
 ? even(odd : ?) =
    proc (x : ?)
     if zero?(x) then 1 else (odd -(x,1))
in letrec
    ? odd(x : bool) =
       if zero?(x) then 0 else ((even odd) -(x,1))
   in (odd 13)
```

Solution:

In second `letrec`, `odd`'s parameter `x` should be `(x : int)` or `(x : ?)`.

Exercise 7.15 Write down a rule for doing type inference for a `letrec` expression. Your rule should handle multiple declarations in a `letrec`. Using your rule, derive types for each of the following expressions, or determine that no such type exists:

1. ```
   letrec ? f (x : ?)
          = if zero?(x) then 0 else -((f -(x,1)), -2)
   in f
   ```
2. ```
   letrec ? even (x : ?)
             = if zero?(x) then 1 else (odd -(x,1))
          ? odd (x : ?)
             = if zero?(x) then 0 else (even -(x,1))
   in (odd 13)
   ```
3. ```
   letrec ? even (odd : ?)
             = proc (x) if zero?(x)
                        then 1
                        else (odd -(x,1))
   in letrec ? odd (x : ?) =
                if zero?(x)
                then 0
                else ((even odd) -(x,1))
      in (odd 13)
   ```

Solution:
   ```

$$\texttt{(letrec-exp bounds body)} \;:\; \begin{array}{l} t_{\text{var1}} = t_{\text{exp1}} \\ t_{\text{var2}} = t_{\text{exp2}} \\ \vdots \\ tenv[t_{\text{var}-n} \ldots t_{\text{var2}}, t_{\text{var1}}] \\ t_{\text{letrec}-\text{exp}} = t_{\text{body}} \end{array}$$

1. int → int
2. int
3. int

Exercise 7.16 Modify the grammar of INFERRED so that missing types are simply omitted, rather than marked with ?.

Solution:

There will be non-checked language.

Exercise 7.17 In our representation, `extend-subst` may do a lot of work if σ is large. Implement an alternate representation in which `extend-subst` is implemented as

```
(define extend-subst
  (lambda (subst tvar ty)
    (cons (cons tvar ty) subst)))
```

and the extra work is shifted to `apply-subst-to-type`, so that the property $t(\sigma[t_v = t`]) = (t\sigma)[t_v = t]$ is still satisfied. For this definition of `extend-subst`, is the no-occurrence invariant needed?

Solution:

Yes, it needed.

See source code in file "section-7.4/subst.rkt".

Exercise 7.18 Modify the implementation in the preceding exercise so that `apply-subst-to-type` computes the substitution for any type variable at most once.

Solution:

See source code in file "section-7.4/subst.rkt".

Exercise 7.19 We wrote: "If `ty1` is an unknown type, then the no-occurrence invariant tells us that it is not bound in the substitution." Explain in detail why this is so.

Solution:

Before this, call `apply-subst-to-type` with `ty1(old)`,where returned ruslut to a new `ty1(new)`. If `ty1(old)` is bound, then `ty1(new)` will not be an unknown type.

Exercise 7.20 Modify the unifier so that it calls `apply-subst-to-type` only on type variables, rather than on its arguments.

Solution:

Elided.

Exercise 7.21 We said the substitution is like a store. Implement the unifier, using the representation of substitutions from exercise 7.17, and keeping the substitution in a global Scheme variable, as we did in figures 4.1 and 4.2.

Solution:

Elided.

Exercise 7.22 Refine the implementation of the preceding exercise so that the binding of each type variable can be obtained in constant time.

Solution:

See source code in file "section-7.4/unifier.rkt".

Exercise 7.23 Extend the inferencer to handle pair types, as in exercise 7.8.

Solution:

Elided.

Exercise 7.24 Extend the inferencer to handle multiple let declarations, multiargument procedures, and multiple `letrec` declarations.

Solution:

See source code in folder "section-7.4".

Exercise 7.25 Extend the inferencer to handle list types, as in exercise 7.9. Modify the language to use the production

$$\textit{Expression} \quad ::= \quad \texttt{emptylist}$$

instead of

$$\textit{Expression} \quad ::= \quad \texttt{emptylist\_}\textit{Type}$$

As a hint, consider creating a type variable in place of the missing _t.

Solution:

See source code in folder "section-7.4/7.27".

Exercise 7.26 Extend the inferencer to handle EXPLICIT-REFS, as in exercise 7.10.

Solution:

Elided.

Exercise 7.27 Rewrite the inferencer so that it works in two phases. In the first phase it should generate a set of equations, and in the second phase, it should repeatedly call `unify` to solve them.

Solution:

See source code in folder "section-7.4/7.27".

Exercise 7.28 Our inferencer is very useful, but it is not powerful enough to allow the programmer to define procedures that are polymorphic, like the polymorphic primitives `pair` or `cons`, which can be used at many types. For example, our inferencer would reject the program

```
let f = proc (x : ?) x
in if (f zero?(0))
   then (f 11)
   else (f 22)
```

even though its execution is safe, because `f` is used both at type `(bool→ bool)` and at type `(int→ int)`. Since the inferencer of this section is allowed to find at most one type for `f`, it will reject this program.

For a more realistic example, one would like to write programs like

```
let
 ? map (f : ?) =
    letrec
     ? foo (x : ?) = if null?(x)
                     then emptylist
                     else cons((f car(x)),
                               ((foo f) cdr(x)))
     in foo
in letrec
    ? even (y : ?) = if zero?(y)
                     then zero?(0)
                     else if zero?(-(y,1))
                          then zero?(1)
                          else (even -(y,2))
   in pair(((map proc(x : int)-(x,1))
            cons(3,cons(5,emptylist))),
            ((map even)
             cons(3,cons(5,emptylist))))
```

This expression uses `map` twice, once producing a list of `int`s and once producing a list of `bool`s. Therefore it needs two different types for the two uses. Since the inferencer of this section will find at most one type for `map`, it will detect the clash between `int` and `bool` and reject the program.

One way to avoid this problem is to allow polymorphic values to be introduced only by `let`, and then to treat `(let-exp var e1 e2)` differently from `(call-exp (proc-exp var e2) e1)` for type-checking purposes.

Add polymorphic bindings to the inferencer by treating `(let-exp var e1 e2)` like the expression obtained by substituting `e1` for each free occurrence of var in `e2`. Then, from the point of view of the inferencer, there are many different copies of `e1` in the body of the `let`, so they can have different types, and the programs above will be accepted.

Solution:

I used `Any` type to denote one of types we defined, so it is not as exact as representation in book.

See source code in folder "section-7.4/7.27".

Exercise 7.29 The type inference algorithm suggested in the preceding exercise will analyze e1 many times, once for each of its occurrences in e2. Implement Milner's Algorithm W, which analyzes e1 only once.

Solution:

Elided.

Exercise 7.30 The interaction between polymorphism and effects is subtle. Consider a program starting

```
let p = newref(proc (x : ?) x)
in ...
```

1. Finish this program to produce a program that passes the polymorphic inferencer, but whose evaluation is not safe according to the definition at the beginning of the chapter.
2. Avoid this difficulty by restricting the right-hand side of a `let` to have no effect on the store. This is called the *value restriction*.

Solution:

Elided.

8 *Modules*

Elided.

9 *Objects and Classes*

9.1 Object-Oriented Programming

9.2 Inheritance

9.3 The Language

9.4 The Interpreter

Exercise 9.1 Implement the following using the language of this section:

1. A queue class with methods `empty?`, `enqueue`, and `dequeue`.
2. Extend the queue class with a counter that counts the number of operations that have been performed on the current queue.
3. Extend the queue class with a counter that counts the total number of operations that have been performed on all the queues in the class. As a hint, remember that you can pass a shared counter object at initialization time.

Solution:

If use Scheme's `list`, it is easy to be done. But if only uses OOP feature without any container, it yet may need with *explicit-refs* to do this.

Elided.

Exercise 9.2 Inheritance can be dangerous, because a child class can arbitrarily change the behavior of a method by overriding it. Define a class `bogus-oddeven` that inherits from `oddeven` and overrides the method even so that let `o1 = new bogus-oddeven() in send o1 odd (13)` gives the wrong answer.

Solution:

See source code in file "section-9.4/9.2.rkt".

Exercise 9.3 In figure 9.11, where are method environments shared? Where are the `field-names` lists shared?

Solution:

`object` always be shared.

Exercise 9.4 Change the representation of objects so that an *Obj* contains the class of which the object is an instance, rather than its name. What are the advantages and disadvantages of this representation compared to the one in the text?

Solution:

Advantage: just search class enviroment once only when creating object, and save all env-information into object.

Disadvantage: if the class has inherited many times, then the information saved may be huge.

Exercise 9.5 The interpreter of section 9.4 stores the superclass name of a method's host class in the lexical environment. Change the implementation so that the method stores the host class name, and retrieves the superclass name from the host name.

Solution:

See source code files in folder "section-9.4".

Exercise 9.6 Add to our language the expression `instanceof` *exp class-name*. The value of this expression should be true if and only if the object obtained by evaluating exp is an instance of class-name or of one of its subclasses.

Solution:

See source code files in folder "section-9.4".

Exercise 9.7 In our language, the environment for a method includes bindings for the field variables declared in the host class *and* its superclasses. Limit them to just the host class.

Solution:

See source code files in folder "section-9.4".

Exercise 9.8 Add to our language a new expression,

`fieldref` *obj* *field-name*

that retrieves the contents of the given field of the object. Add also

`fieldset` *obj field-name* = *exp*

which sets the given field to the value of exp.

Solution:

Elided.

Exercise 9.9 Add expressions `superfieldref` field-name and `superfieldset` *field-name* = *exp* that manipulate the fields of `self` that would otherwise be shadowed. Remember `super` is static, and always refers to the superclass of the host class.

Solution:

Elided.

Exercise 9.10 Some object-oriented languages include facilities for named-class method invocation and field references. In a named-class method invocation, one might write `named-send c1 o m1()`. This would invoke `c1`'s `m1` method on `o`, so long as `o` was an instance of `c1` or of one of its subclasses, even if `m1` were overridden in `o`'s actual class. This is a form of static method dispatch. Named-class field reference provides a similar facility for field reference. Add named-class method invocation, field reference, and field setting to the language of this section.

Solution:

See source code files in folder "section-9.4".

Exercise 9.11 Add to CLASSES the ability to specify that each method is either private and only accessible from within the host class, protected and only accessible from the host class and its descendants, or public and accessible from anywhere. Many object-oriented languages include some version of this feature.

Solution:

Elided.

Exercise 9.12 Add to CLASSES the ability to specify that each field is either private, protected, or public as in exercise 9.11.

Solution:

Elided.

Exercise 9.13 To defend against malicious subclasses like bogus-oddeven in exercise 9.2, many object-oriented languages have a facility for final methods, which may not be overridden. Add such a facility to CLASSES, so that we could write

```
class oddeven extends object
  method initialize () 1
  final method even (n)
    if zero?(n) then 1 else send self odd(-(n,1))
  final method odd (n)
    if zero?(n) then 0 else send self even(-(n,1))
```

Exercise 9.14 Another way to defend against malicious subclasses is to use some form of *static dispatch*. Modify CLASSES so that method calls to self always use the method in the host class, rather than the method in the class of the target object.

Solution:

Elided.

Exercise 9.15 Many object-oriented languages include a provision for *static* or *class* variables. Static variables associate some state with a class; all the instances of the class share this state. For example, one might write:

```
class c1 extends object
  static next-serial-number = 1
  field my-serial-number
  method get-serial-number () my-serial-number
  method initialize ()
    begin
      set my-serial-number = next-serial-number;
      set next-serial-number = +(next-serial-number,1)
    end
let o1 = new c1()
    o2 = new c1()
in list(send o1 get-serial-number(),
        send o2 get-serial-number())
```

Each new object of class `c1` receives a new consecutive serial number.

Add static variables to our language. Since static variables can appear in a method body, `apply-method` must add additional bindings in the environment it constructs. What environment should be used for the evaluation of the initializing expression for a static variable (1 in the example above)?

Solution:

Class environment rather then object environment.

```
(static class-id var-id)
```

Exercise 9.16 Object-oriented languages frequently allow *overloading* of methods. This feature allows a class to have multiple methods of the same name, provided they have distinct signatures. A method's signature is typically the method name plus the types of its parameters. Since we do not have types in CLASSES, we might overload based simply on the method name and number of parameters. For example, a class might have two `initialize` methods, one with no parameters for use when initialization with a default field value is desired, and another with one parameter for use when a particular field value is desired. Extend our interpreter to allow overloading based on the number of method parameters.

Solution:

Elided.

Exercise 9.17 As it stands, the classes in our language are defined globally. Add to CLASSES a facility for local classes, so one can write something like `letclass c = ...in e`. As a hint, consider adding the class environment as a parameter to the interpreter.

Solution:

Elided.

Exercise 9.18 The method environments produced by `merge-method-envs` can be long. Write a new version of `merge-method-envs` with the property that each method name occurs exactly once, and furthermore, it appears in the same place as its earliest declaration. For example, in figure 9.8, method `m2` should appear in the same place in the method environments of `c1`, `c2`, `c3`, and any descendant of `c3`.

Solution:

Put method's AST into store model, object reference it's adress.

See source code files in folder “section-9.4”.

Exercise 9.19 Implement lexical addressing for CLASSES. First, write a lexical-address calculator like that of section 3.7.1 for the language of this section. Then modify the implementation of environments to make them nameless, and modify `value-of` so that `apply-env` takes a lexical address instead of a symbol, as in section 3.7.2.

Solution:

Elided.

Exercise 9.20 Can anything equivalent to the optimizations of the exercise 9.19 be done for method invocations? Discuss why or why not.

Solution:

`self-call` and `super-call` can be converted to simple linear search, no recursive.

But if directly use the reference adress of method saved in store model, it will a great improvement.

Exercise 9.21 If there are many methods in a class, linear search down a list of methods can be slow. Replace it by some faster implementation. How much improvement does your implementation provide? Account for your results, either positive or negative.

Solution:

Ditto.

Exercise 9.22 In exercise 9.16, we added overloading to the language by extending the interpreter. Another way to support overloading is not to modify the interpreter, but to use a syntactic preprocessor. Write a preprocessor that changes the name of every method `m` to one of the form `m:@n`, where `n` is the number of parameters in the method declaration. It must similarly change the name in every method call, based on the number of operands. We assume that `:@` is not used by programmers in method names, but is accepted by the interpreter in method names. Compilers frequently use such a technique to implement method overloading. This is an instance of a general trick called *name mangling*.

Solution:

Elided.