1 Reading

- Little Schemer Chapter 8
- SICP 1, 3.2

2 Higher-order Procedures

Let us define a procedure to sum integers from a to b. (See Section 1.3 of SICP)

```
(define (sum-integers a b)
  (if (> a b)
      0
      (+ a (sum-integers (+ a 1) b))))
```

```
> (sum-integers 2 5)
14
```

Let us define a procedure to sum cube of integers from a to b. (See Section 1.3 of SICP)

```
(define (cube x) (* x x x))
(define (sum-cubes a b)
  (if (> a b)
      0
      (+ (cube a) (sum-cubes (+ a 1) b))))
```

```
> (sum-cubes 2 5)
```

Let us define a procedure to compute sum of a sequence of terms in a series, which converges to pi/8 very slowly. (See Section 1.3 of SICP)

```
(define (pi-sum a b)
  (if (> a b)
      0
      (/ 1.0 (* a (+ a 2))) (pi-sum (+ a 4) b)))
```

```
> (pi-sum 2 5)
> (pi-sum 2 17)
```

All these procedures have similar structure.

```
(define (<name> a b)
  (if (> a b)
      0
      (+ (<term> a)
        (<name> (<next> a) b))))
```

Exercise 2.1 Can we define a general procedure for summation?

```
(define (sum term a next b)
  (if (> a b)
      0
      (+ (term a)
        (sum term (next a) next b))))
```

Let us redefine sum of integers in terms of sum (see below):

```
> (define (inc n) (+ n 1))
> (inc 5)
6
> (define (identity x) x)
> (identity 5)
```
> (define (sum-integers a b)
  (sum identity a inc b))
> (sum-integers 2 5)
14

Let us redefine sum of cubes in terms of sum

> (define (sum-cubes a b)
  (sum cube a inc b))
> (sum-cubes 1 10)
3025

Let us redefine pi-sum in the same way.

> (define (pi-sum a b)
  (define (pi-term x)
    (/ 1.0 (+ x (+ x 2))))
  (define (pi-next x)
    (+ x 4))
  (sum pi-term a pi-next b))
> (pi-sum 1 10)
0.372005772005772

3 Procedures as Data

In Scheme procedures can also be treated as data. Therefore, they can also be understood using exactly the same framework that we used to study other data types. For more details, read Little Schemer, Chapter 8.

- Values
  - abstract: partial mappings from a domain to a range with side-effects on store
  - external (printed form): differs with interpreters, some examples are as follows:
    * in SCM: #<CLOSURE ...> #<primitive − procedure ...>
    * in Chez Scheme: #<procedure>
    * in Dr. Scheme: #<procedure> or #<procedure:name>

- Operations
  - procedures: apply
  - special forms: lambda, ——

4 Procedure Creation: Lambda Makes Procedure Values

Examples:

(l lambda (n) (+ n 1))
(l lambda (x y) (/ (+ x y) 2))
4.1 Terminology

- Closure = environment + code
- What are formals, formal parameter, bound variable?
- What is an anonymous procedure?

Exercise 4.2 What other languages use anonymous procedures?

ML, Haskell, LISP, but these are all similar kinds. Smalltalk blocks are like this. An object in C++ and Java are related to this, as we’ll see. C# delegates are also very similar.

4.2 Use of Anonymous Procedures

```latex
> ((\texttt{lambda} \ (n) \ (+ \ n \ 1)) \ 3)
4
```

```latex
> (\texttt{map} \ (\texttt{lambda} \ (n) \ (+ \ n \ 10)) \ '(4 5 6))
(14 15 16)
```

We’ll see other uses later during this course. Most languages require procedures to be named (avoids overhead of the general case of anonymous procedures)

4.3 Naming Procedures

In Scheme a procedure (or more precisely procedure value) can be named with \texttt{define} in exactly the same way that other values can be named. This is a key benefit of treating procedures as data, which allows regularity in the language.

```latex
(define one
1)
```

```latex
(define add1
(l lambda \ (n)
 (+ \ n \ one)))
```

4.4 First-class Values

**Definition 4.1** A value is first-class if it can be named, given to and returned from procedures, and stored in data structures (just like numbers).

Examples of types with first-class values:

<table>
<thead>
<tr>
<th>type</th>
<th>FORTRAN</th>
<th>The ‘C’ Language</th>
<th>The Java Language</th>
<th>Scheme</th>
</tr>
</thead>
<tbody>
<tr>
<td>number</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>boolean</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>array</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>record</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>pointer</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>procedure</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Exercise 4.3  Are procedures first-class values in FORTRAN? C? Scheme?

We have pointers to procedures in C that can be named, given to and returned from procedures and stored in data structures, so pointers to procedures are first-class.

Exercise 4.4  Are arrays first-class values in C/C++? What about Scheme?

No. However, all values in Scheme are first-class, another instance of regularity in the language.

4.5 Uses of first-class procedures

Recall the map function from Section 4.2. It captures common pattern on lists. Similarly, other languages have sort procedures, that take ordering as a parameter. Primary use of first-class procedures is in writing such functions. To demonstrate, let us write an example procedure compose with the following behavior shown by examples.

\[
((\text{compose } \text{car } \text{cdr}) '(a b c))
\]

= b

= (car '(b c))

= (car (cdr '(a b c)))

\[
((\text{compose } \text{not } \text{null?}) '())
\]

= #f

= (not (null? '()))

\[
((\text{compose } \text{not } \text{zero?}) x)
\]

= (not (zero? x))

In general the procedure compose computes the following.

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

How are we going to write this as a Scheme procedure? Recall from the example usage shown above that (compose f g) is a procedure of one argument, x, so it must be:

(lambda (x)
  (f (g x)))

But compose itself is a procedure, of two arguments, f and g, and when called it returns this other procedure, so:

(lambda (f g)
  (lambda (x)
    (f (g x)))))

And this has to be named compose. Turns out that the procedure is very similar to the representation above.

(define compose
  (lambda (f g)
    (lambda (x)
      (f (g x))))))

Exercise 4.5  Write a procedure twice with the following behavior shown using examples of its usage.
5 Currying

After thinking about first-class procedures for a while, do you think multiple-argument procedures are essential in Scheme? Turns out that they can be emulated using single argument procedures using a programming style known as currying. It is named after Haskell Curry, a logician (although it was actually invented by Frege and Schönfinkel).

To illustrate, let us consider the following example procedure with three parameters.

\[
\text{lambda } (\text{ls1 ls2 ls3})
\]
\[
(\text{append ls1 (append ls2 ls3))}
\]

The curried form of this procedure is the following.

\[
\text{lambda } (\text{ls1})
\]
\[
(\text{lambda } (\text{ls2})
\]
\[
(\text{lambda } (\text{ls3})
\]
\[
(\text{append ls1 (append ls2 ls3))})}
\]

Note that each procedure above has only one argument. Let us look at another example of a procedure with two arguments.

\[
\text{lambda } (x y)
\]
\[
(+ x y)}
\]

The curried form of this procedure is shown below.

\[
\text{lambda } (x)
\]
\[
(\text{lambda } (y)
\]
\[
(+ x y))}
\]

5.1 Uses of Curried Procedures

GRAVITATIONAL FORCE EXAMPLE

\[
\text{define } G
\]
\[
\text{TYPE: N * m^2 / kg^2}
\]
\[
6.670e-11)
\]

\[
\text{define square}
\]
\[
\text{TYPE: } (-> (m) m^2)
\]
\[
(\text{lambda } (r) (* r r))}
\]

\[
\text{define grav-force}
\]
\[
\text{TYPE: } (-> (kg m kg) N)
\]
\[
(\text{lambda } (m1 r m2)
\]
\[
(\text{if } (\text{zero? } r) 0.0
\]
\[
(\text{if } (* G (* m1 m2)) (\text{square r)}))))}
\]

\[
\text{define grav-force-c}
\]
\[
\text{TYPE: } (-> (kg)
\]
\[
(-> (m)
Here is how you would use the gravitational force procedure.

(define mass-of-earth
  TYPE: kg
  5.96e24)

(define radius-of-earth
  TYPE: m
  6.37e6)

(define earths-force
  TYPE: (-> (grav-force-c mass-of-earth)))

(define force-at-surface
  TYPE: (-> (earths-force radius-of-earth)))

> force in N on me (force-at-surface ??)  
> force in N on 1kg (a liter of coke)  
> (force-at-surface 1)

Exercise 5.6 How would you compute forces at distance of moon’s orbit by earth?

Exercise 5.7 How would you compute forces exerted by the sun?

6 Variable Arity Procedures (1.3.3)

Definition 6.1 The arity of a procedure is the number of arguments a procedure can take.

Examples built-in to Scheme:
Special form in Scheme: (lambda x body)

(defun sum (-> (number ...) number))
(define sum
  (lambda (l)
    (if (null? l)
      0
      (+ (car l)
        (sum-of-list (cdr l)))))

Semantics
- caller’s arguments made into list
- the formal denotes a list in the body
Exercise 6.8  What in C is like this?

Exercise 6.9  Does a language have to have these?

Exercise 6.10  Does language have to allow you to write them yourself?

FOR YOU TO DO
Implement the following:

(deftype product (-> (number ...) number))
(product ) => 1
(product 3) => 3
(product 4 3) => 12
(product 7 6 5 4) => 840

(deftype list
    (forall (T) (-> (T ...) (list-of T))))
(list ) => ()
(list 'a) = (a)
(list 'a 'b) = (a b)

Exercise 6.11  Does list have to be built-in to Scheme?

7 Unrestricted Lambda and Apply

((lambda args (f args)) e1 e2 ...)
  = (f (list e1 e2 ...))

(apply f (list e1 e2 ...))
  = (f e1 e2 ...)

So, for all procedures f and lists ls:

(apply (lambda args (f args)) ls)
  = (f ls)