CS251 Programming Languages **Handout # 8**
Prof. Lyn Turbak **January 30, 2002**
Wellesley College

Scheme Data Structures

This handout covers compound data structures in Scheme. Pairs are the usual way of composing data in Scheme. They can be used in idiomatic ways to construct lists and s-expressions (trees). Scheme also supports vectors, which are array-like data structures; see *R5RS* for details. First-class functions are another kind of compound data in Scheme; these will be covered in a later handout.

1 Pairs

In Scheme, any two values V_1 and V_2 can be glued together into a **pair** via the primitive `cons` function. Pictorially, the result of `(cons` V_1 V_2`)` is shown as a **box-and-pointer diagram**:

In the substitution model, `cons` itself is treated as a primitive function like `+` and `<=`, but `(cons` V_1 V_2`)` is treated as a new kind of value that can be the result of an evaluation. For example:

```
subst> (cons (+ 1 2) (- (* 3 4) 5))
(cons 3 (- 12 5))
(cons 3 7)

subst> (cons (* 6 7) (= 8 8))
(cons 42 #t)
```

The second example illustrates that pairs are heterogeneous data structures – i.e., their components need not have the same type.

The left and right components of a pair are extracted, respectively, by the `car` and `cdr` functions[1].

```
subst> (car (cons V1 V2))
V1

subst> (cdr (cons V1 V2))
V2
```

Arbitrarily complex data structures can be built with nested `cons`es and their components can be extracted by sequences of nested `car`s and `cdr`s. For example, consider the following box-and-pointer structure, named `stuff`:

[1]The names `car` and `cdr` are vestiges of an early Lisp implementation on the IBM 704. `car` stands for "contents of address register" and `cdr` stands for "contents of decrement register".

Give a Scheme expression that generates the data structure depicted by the above diagram:

Give expressions that extract each of the components of `stuff`:

Component	Expression
`17`	
`#t`	
`#\c`	
`'cs251`	
`(lambda (a) (* a a))`	

Notes:

- The fact that function values like `(lambda (a) (* a a))` can be stored in data structures is hallmark of functional programming and a powerful feature that we will exploit throughout this course.
- Nested combinations of `car`s and `cdr`s are so common that most Scheme implementations provide primitives that abbreviate up to four such nested applications. The abbreviations are equivalent to the following definitions:

```
(define caar (lambda (x) (car (car x))))
(define cadr (lambda (x) (car (cdr x))))
(define cdar (lambda (x) (cdr (car x))))
(define cddr (lambda (x) (cdr (cdr x))))
⋮
(define cdddar (lambda (x) (cdr (cdr (cdr (car x))))))
(define cddddr (lambda (x) (cdr (cdr (cdr (cdr x))))))
```

- Scheme interpreters usually print pairs using the "dotted pair" notation. For example, the printed representation of the pair `(cons 17 42)` is `(17 . 42)`. However, due to the list printing conventions described in the next section, the story is more complex. For instance, we might expect that the printed representation of `(cons (cons 1 2) (cons 3 4))` should be `((1 . 2) . (3 . 4))`, but it is in fact `((1 . 2) 3 . 4)`.

2 Lists

2.1 Scheme Lists Are Sequences of Pairs

In Scheme programs it is rare to use `cons` to pair two arbitrary values. Instead, it is much more common to use sequences of pairs connected by their `cdr`s to represent **linked lists** (which in

Scheme are simply called **lists**). For example, here is a list of the five values from the example of the previous section.
The sequence of pairs connected by their cdrs is known as the spine of the list.

A list is defined inductively as follows. A list is either

- An empty list (shown as the solid black circle in the above diagram).
- A pair whose `car` is the head of the list and whose `cdr` is the tail of the list.

The Scheme notation for an empty list is `'()`; the primitive predicate `null?` returns true for the empty list and false otherwise. Thus, the above list can be constructed as follows:

```
(cons 17
      (cons #t
            (cons #\c
                  (cons 'cs251
                        (cons (lambda (a) (* a a))
                              '()))))))
```

Such nested sequences of `cons`es are very difficult to read. They can be abbreviated using the primitive `list` procedure, which takes n argument values and constructs a list of those n values. For example, the above expression can be written more succinctly as follows:

```
(list 17 #t #\c 'cs251 (lambda (a) (* a a)))
```

In fact, it helps to think that `list` desugars into a nested sequence of `cons`es (it turns out that this is not quite true, but is close enough to being true that it's not harmful to think this way):

`(list` V_1 V_2 $\ldots V_n$`)` $\rightsquigarrow$ `(cons` V_1 `(cons` V_2 `...(cons` V_n `'()) ...))`

The following idioms are so common that you should commit them to memory (some Scheme systems supply these as primitives; if they don't, you can define them on your own):

```
(define first (lambda (lst) (car lst)))
(define second (lambda (lst) (cadr lst)))
(define third (lambda (lst) (caddr lst)))
(define fourth (lambda (lst) (cadddr lst)))
(define fifth (lambda (lst) (car (cddddr lst))))
(define but-first (lambda (lst) (cdr lst)))
(define but-second (lambda (lst) (cddr lst)))
(define but-third (lambda (lst) (cdddr lst)))
(define but-fourth (lambda (lst) (cddddr lst)))
(define but-fifth (lambda (lst) (cdr (cddddr lst))))
```

For example, if the example list above is named `L`, then the components can be extracted as follows:

Component	Expression	Better Expression	Best Expression
`17`	`(car L)`	`(car L)`	`(first L)`
`#t`	`(car (cdr L))`	`(cadr L)`	`(second L)`
`#\c`	`(car (cdr (cdr L)))`	`(caddr L)`	`(third L)`
`'cs251`	`(car (cdr (cdr (cdr L))))`	`(cadddr L)`	`(fourth L)`
`(lambda (a) (* a a))`	`(car (cdr (cdr (cdr (cdr L)))))`	`(car (cddddr L))`	`(fifth L)`

Most Scheme system display a list as sequence of values delimited by parenthesis. Thus, the list value produced by `(list 3 #t #\c)` is usually displayed as `(3 #t #\c)`. One way to think of this is that it is a form of the dotted-pair notation in which a dot "eats" a following open parenthesis. That is, `(3 . (#t . (#\c . ())))` becomes `(3 #t #\c)`. In most systems, symbols appear without a quotation mark, so the result of evaluating `(list 'a 'b 'c)` is printed `(a b c)`. Procedural values are printed in an ad hoc way that varies greatly from system to system.

In addition to the `list` construct, there are two other standard list-building functions.

- `(cons` *value list*`)` prepends *value* to the front of *list*. If *list* has n elements, then `(cons` *value list*`)` has $n + 1$ elements.
- `(append` $list_1$ $list_2$`)` returns a list containing all the elements of $list_1$ followed by all the elements of $list_2$. If $list_1$ has m elements and $list_2$ has n elements, then `(append` $list1_1$ $list_2$`)` has $m + n$ elements.

For example, suppose that we have the following definitions:

```
(define L1 (list 1 2 3))
(define L2 (list 4 5))
```

What are the (1) box-and-pointer diagrams and (2) printed representations of the results of the following manipulations involving these lists?

```
> (list L1 L2)
```

```
> (cons 17 L1)
```

```
> (cons L1 L2)
```

```
> (append L1 L2)
```

```
> (append L1 L1)
```

We emphasize that all the list operations we're studying now are non-destructive – they never change the contents of an existing list, but always make a new list. So when we say that "cons prepends a value to the front of a list", this is an abuse of language that really means "cons returns a list that is formed by prepending the given value to the given list." Later in the course, destructive list operators will be introduced in the context of imperative programming.

2.2 List Quotation

Scheme supports some syntactic sugar involving quotation to simplify writing complex list structures. Informally, `'`*elts* constructs the list structure whose printed representation is that of *elts*.

```
> '(1 a #t #\c)
(1 a #t #\c)

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

It turns out that `'`*elts* is itself an abbreviation for `(quote elts)`.

More formally, you can imagine that Scheme has the following syntactic sugar for quoted list structure. (This is not really the case, but it's close enough to being true that you can believe it.)

`'`$N \leadsto N$
`'`$B \leadsto B$
`'`$C \leadsto C$
`'`$R \leadsto R$
`'(`$S_1 \ldots S_n$`)` $\leadsto$ `(list` $S_1 \ldots S_n$`)`

2.3 List Recursion

The inductive definition of lists naturally leads to functions on lists that are defined recursively. Below are some representative functions, some of which we will define in class. Those that we don't get to in class you should define on your own (see me or the tutor if you need help!)

(length *list*`)`
Return the number of elements in the list *list*.

```
> (length '(a b c))
3

> (length '(17))
1

> (length '())
0
```

(**sum** *list*)
Return the sum of the elements in the list *list*.

```
> (sum '(2 4 6))
12

> (sum '())
0
```

(**from-to** *lo hi*)
Return a list of all the integers from *lo* up to *hi*, inclusive.

```
> (from-to 3 7)
(3 4 5 6 7)

> (from-to 7 3)
()
```

(**squares** *list*)
Return a list whose values are the squares of the given list of numbers *list*.

```
> (squares '(5 1 3 2))
(25 1 9 4)
```

(**evens** *list*)
Return a list containing only even numbers in the given list of numbers *list*.

```
> (evens '(5 1 4 2 7 6))
(4 2 6)

> (evens '(5 1 3 7))
()
```

(**member?** *value list*)
Determine if *value* is an element in the given list *list*. (Assume `equal?` is used to test equality.)

```
> (member? 2 '(5 1 4 2 7 6))
#t

> (member? 17 '(5 1 4 2 7 6))
()
```

(**remove** *value list*)
Remove all occurrences of *value* from the given list *list*. (Assume `equal?` is used to test equality.)

```
> (remove 3 '(1 3 2 3 3 4 3 1 2))
(1 2 4 1 2)

> (remove 3 '(1 2 4 1 2))
(1 2 4 1 2)
```

(remove-duplicates *list*)
Returns a list in which each element of appears only once. The order of elements in the resulting list is irrelevant. (Assume `equal?` is used to test equality.)

```
> (remove-duplicates '(1 3 2 3 3 4 3 1 2))
(1 3 2 4)
```

(append *list1 list2*)
Return a list that results from appending lists *list1* and *list2*.

```
> (append '(1 2 3) '(list a b c))
(1 2 3 a b c)
```

(zip *list1 list2*)
Given lists *list1* and *list2*, return a list of duples (two-element lists) containing corresponding elements from *list1* and *list2*. If the two lists differ in size, the resulting list should have the length of the shorter of the two lists.

```
> (zip '(1 2 3) '(a b c))
((1 a) (2 b) (3 b))

> (zip '(1 2) '(a b c))
((1 a) (2 b))

> (zip '(1 2 3) '(a b))
((1 a) (2 b))
```

(**reverse** *list*)
Return a list whose elements are in reverse order from the given list *list*.

```
> (reverse '(1 2 3))
(3 2 1)

> (reverse '((1 2 3) 4 (5 6)))
((5 6) 4 (1 2 3)
```

Note: there are numerous ways to define `reverse`. Try to define it both recursively and iteratively. You may find the following helper function (which corresponds to the `postpend()` method we studied in Java) function for the recursive definition:

```
(define snoc
  (lambda (lst elt)
    (if (null? lst)
        (list elt)
        (cons (car lst) (snoc (cdr lst) elt)))))
```

2.4 Scheme Lists vs. Java Lists

Scheme list operations should seem very familiar from CS111 and CS230! Indeed, the list material in these courses was patterned after Scheme list operations, so the similarity is not coincidental. Here is the correspondence between the Java list operations you've seen before and the Scheme list operations:

Java list operation	Scheme list operation
`prepend(x,L)`	`(cons x L)`
`head(L)`	`(car L)`
`tail(L)`	`(cdr L)`
`empty()`	`'()`
`IsEmpty(L)`	`(null? L)`

In CS111 and CS230, you have seen many of the recursive functions from the previous section written in Java. Two great advantages of Scheme over Java are that (1) Scheme lists are heterogeneous: a single list may contain elements of many different types) and (2) Scheme list functions are polymorphic: they work on any list, regardless of the types of its elements.

In contrast, Java lists must contain elements that are all (subtypes of) a given type, and Java list functions work only for lists whose elements are a particular type. For instance, in Java, we defined classes like `IntList` and `BoolList` for representing lists of integers and lists of booleans,

and it was necessary to write different `length()`, `append()`, `reverse()`, etc. methods for each such class even though the code for these methods never examines the elements.

One way of finessing these problems in Java is to use an `ObjectList` class in which every element is an `Object`. Then all list methods can be defined exaclty once on `ObjectList`. However, as we saw in CS111 and CS230, there are two problems with this approach: (1) Java's type system requires an explicit cast to be applied to elements extracted from an `ObjectList`; and (2) since primitive datatypes like `int`, `boolean`, etc. are not objects, they must be packaged into and unpackaged from wrapper classes like `Integer`, `Boolean`, etc. These infelicities make the Java list programs less readable and less general than their Scheme counterparts.

3 S-Expressions

A symbolic expression, or s-expression for short, is defined inductively as follows. An s-expression is either:

- a literal[2]

  (i.e., number, boolean, symbol, character, string, or empty list);

- a list of s-expressions.

An s-expression can be viewed as a tree where each list corresponds to an (unlabelled) tree node, and each subexpression corresponds to a subtree. For instance, the s-expression `'((1 2 3) 4 (5 (6 7)))` has the following tree representation:

There are many functions on trees/s-expressions. Here's a representative one:

(**flatten** *sexp*)
Return a list of the leaves of the tree *sexp* in an in-order traversal.

```
> (flatten '((1 2 3) 4 (5 (6 7)))
(1 2 3 4 5 6 7)

> (flatten 1)
(1)

> (flatten '())
()
```

[2]Many Scheme texts, such as *SICP*, refer to a literal as an atom