CS61A Notes 04 – Lists (v1.0)

Pair Up!

Introducing – the **only data structure you'll ever need** in 61A – **pairs**.

A pair is a data structure that contains two *things* – the "things" can be atomic values or even another pair. For example, you can represent a point as `(x . y)`, and a date as `(July . 1)`. Note the Scheme representation of a pair; the pair is enclosed in parentheses, and separated by a single period.

Note that there's an operator called `pair?` that tests whether something is a pair or not. For example, `(pair? (cons 3 4))` is `#t`, while `(pair? 10)` is `#f`.

You've read about "`cons`", "`car`", and "`cdr`":
`cons` – takes in two parameters and constructs a pair out of them. So `(cons 3 4)` will return `(3 . 4)`
`car` – takes in a pair and returns the first part of the pair. So `(car (cons 3 4))` will return `3`.
`cdr` – takes in a pair and returns the second part of the pair. So `(cdr (cons 3 4))` will return `4`.

These, believe it or not, will be all we'll ever need to build complex data structures in this course.

QUESTIONS: What do the following evaluate to?

```
(define u (cons 2 3))  (define w (cons 5 6))  (define x (cons u w))
(define y (cons w x))  (define z (cons 3 y))
```

1. **u, w, x, y, z (write them out in Scheme's notation)**

2. **(car y)**

3. **(car (car y))**

4. **(cdr (car (cdr (cdr z))))**

5. **(+ (cdr (car y)) (cdr (car (cdr z))))**

6. **(cons z u)**

7. **(cons (car (cdr y)) (cons (car (car x)) (car (car (cdr z)))))**

Then Came Lists

The super-cool definition of "list": **a list is either an empty list, or a pair whose `car` is an element of the list and whose `cdr` is another list.** Note the recursive definition – a list is a pair that contains a list! So then how does it end? Wouldn't there be an infinite number of list? Not so: **an empty list, called "nil" and denoted `'()` is a list containing no elements**. And so it is, that every list ends with the empty list. To test whether a list is empty, you can use the `null?` operator on a list.

So, to make a list of elements `2`, `3`, `4`, we do this:

```
(define x (cons 2 (cons 3 (cons 4 '() ) ) ))
```

So x will be then represented as:

```
(2 . (3 . (4 . '() ) ) )
```

Now, that looks a bit ugly, so Scheme, the nice friendly language that it is, sugar-coats the notation a bit so you get:

```
(2 3 4)
```

It's a bit annoying to write so many `cons` to define x. So Scheme, the mushy-gushy language that it is, provides an operator `list` that takes in elements and returns them in a list. So we can also define `x` this way:

```
(define x (list 2 3 4))
```

Note: `(car x)` is `2`, `(cdr x)` is `(3 4)`, and `(car (cdr x))` is `3`! Well, it's a bit tiresome to write `(car (cdr x))` to get the second element of `x`. So Scheme, again the huggable lovable language that it is, provides a nifty short hand: `(cadr x)`. This reads *cader*, and means "take the `car` of the `cdr` of". Similarly, you can use `(caddr x)` – *caderder* – to take the `car` of the `cdr` of the `cdr` of `x`, which is `4`. You can mix and match the '`a`' and '`d`' between the '`c`' and '`r`' to get the desired element of the list (up to a certain length).

You can also append two lists together. `append` takes in any number of lists and outputs a list containing those lists concatenated together. So `(append (list 3 4) (list 5 6))` returns `(3 4 5 6)`.

Don't You Mean `sentence`?

Oh stop grumbling. A "sentence" is actually a special kind of "list" – more specifically, a "sentence" is a *flat list* – a list without any sublists – whose elements can only be words or numbers. The operators of sentence – `first`, `butfirst`, `se`, etc. – are also much more forgiving in its domain than their list counterparts. For example, here's a list of equivalences:

```
first <==> car: (first '(1 2 3 4)) = (car '(1 2 3 4)) = 1
butfirst <==> cdr: (butfirst '(1 2 3 4)) = (cdr '(1 2 3 4)) = (2 3 4)
empty? <==> null?: (empty? '()) = (null? '()) = #t
sentence <==> list: (se 1 2 3 4) = (list 1 2 3 4) = (1 2 3 4)
sentence <==> cons: (se 1 '(2 3 4)) = (cons 1 '(2 3 4)) = (1 2 3 4)
sentence <==> append: (se '(1 2) '(3 4)) = (append '(1 2) '(3 4)) = (1 2 3 4)
count <==> length: (count '(3 4 1)) = (length '(3 4 1)) = 3
every <==> map: (every square '(1 2)) = (map square '(1 2)) = (1 4)
keep <==> filter: (keep number? '(2 k 4)) = (filter number? '(2 k 4)) = (2 4)
```

Note that while `se` can be used for any combination of single-elements and sentences to make another sentence, `list`, `cons` and `append` are a bit more subtle, and what you pass in as arguments really matters.. For example,

```
(se '(1 2) '(3 4)) = (1 2 3 4) != (list '(1 2) '(3 4)) = ((1 2) (3 4))
(se '(1 2) 3) = (1 2 3) != (cons '(1 2) 3) = ((1 2) . 3)
   != (list '(1 2) 3) = ((1 2) 3) != (append '(1 2) 3) = Error: 3 not a list!
```

And so on. You must be more careful with what you pass into the list operators! What do you get for all that trouble? Power – you can put *anything* into lists, not just words and numbers. You will now be able to construct deep lists – lists that contain sublists, which allows you to represent all sorts of cool things. You can also store exotic things like procedures in a list. The possibilities are endless!

QUESTIONS:

1. **Define a procedure `list-4` that takes in 4 elements and outputs a list equivalent to one created by calling list.**

2. **Define a procedure `length` that takes in a list and returns the number of elements within the list.**

3. **Define a procedure `list?` that takes in something and returns `#t` if it's a list, `#f` otherwise.**

4. **Define `append` for two lists.**

5. **Suppose we have `x` bound to a mysterious element. All we know is this:**
```
(list? x) ==> #t
(pair? x) ==> #f
```
What is `x`?

6. **Add in procedure calls to get the desired results. The blanks don't need to have anything:**

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

(               '(cs61a is)           `cool                  )
      ==> (cs61a is cool)

(               '(back to)            '(save the universe)   )
      ==> ((back to) save the universe)

(               '(I keep the wolf)    '((from the door))     )
      ==> ((I keep the wolf) from the door)
```

7. **Define a procedure `(insert-after item mark ls)` which inserts `item` after `mark` in `ls`.**

(Slightly) Harder Lists

1. **Define a procedure `(depth ls)` that calculates how maximum levels of sublists there are in `ls`. For example,**
   ```
   (depth '(1 2 3 4)) ==> 1
   (depth '(1 2 (3 4) 5)) ==> 2
   (depth '(1 2 (3 4 5 (6 7) 8) 9 (10 11) 12)) ==> 3
   ```
   Remember that there's a procedure called `max` that takes in two numbers and returns the greater of the two.

2. **Define a procedure `(remove item ls)` that takes in a list and returns a new list with `item` removed from `ls`.**

3. **Define a procedure `(unique-elements ls)` that takes in a list and returns a new list without duplicates. You've already done this with `remove-dups`, and it used to do this:**
   ```
   (remove-dups '(3 5 6 3 3 5 9 8)) ==> (6 3 5 9 8)
   ```
   where the *last* occurrence of an element is kept. We'd like to keep the *first* occurrences:
   ```
   (unique-elements '(3 5 6 3 3 5 9 8)) ==> (3 5 6 9 8)
   ```
   Try doing it without using `member?`. You might want to use `remove` above.

4. **Define a procedure `(count-of item ls)` that returns how many times a given item occurs in a given list; it could also be in a sublist. So,**
   ```
   (count-of 'a '(a b c a a (b d a c (a e) a) b (a))) ==> 7
   ```

5. **Define a procedure `(interleave ls1 ls2)` that takes in two lists and returns one list with elements from both lists interleaved. So,**
   `(interleave '(a b c d) (1 2 3 4 5 6 7)) ==> (a 1 b 2 c 3 d 4 5 6 7)`

6. **Write a procedure `(apply-procs procs args)` that takes in a list of single-argument procedures and a list of arguments. It then applies each procedure in `procs` to each element in `args` in order. It returns a list of results. For example,**
```
(apply-procs (list square double +1) '(1 2 3 4))
  ==> (3 9 19 33)
```

Expression Lists

Here's something interesting: a Scheme expression is really just a list! For example, `(+ 2 3)` is just a list whose first element is the symbol `+`, second element the number `2`, and third element the number `3`. Similarly, `(define x (* 4 5))` is just a list whose first element is the symbol "`define`", second element the symbol "`x`", and third element the sublist `(* 4 5)`. Given that insight, let's start manipulating more lists!

QUESTIONS

1. **Define a procedure `(eval-plus exp)` that takes in a valid Scheme expression consisting only of + and numbers, and evaluates it to the correct value. Assume that + always only gets two arguments. For example,**
```
(eval-plus 3) ==> 3
(eval-plus '(+ 3 4)) ==> 7
(eval-plus '(+ 10 (+ 3 2)) ==> 15
```

2. **(HARD!) Define `(eval-plus exp)` again, but let + take any number of arguments.**

3. **We'd like some easy way of creating a `lambda` expression. Write `(make-lambda args body)` that takes in the argument list and the body of a procedure, and produces the corresponding `lambda` expression. For example,**
```
(make-lambda '(x y) '(+ x (* y x))) ==> (lambda (x y) (* y x))
```

4. **Recall that there are two ways of defining procedures: the "real" way, and the sugar-coated way. Write a procedure `(unsugar def)` that takes in a procedure definition in sugar-coated syntax, and returns the same definition without using the syntactic sugar. For example,**
```
(unsugar '(define (square x) (* x x)))
   ==> (define square (lambda (x) (* x x)))
```

5. **Recall that a `let` expression is actually just a `lambda` expression. Write a procedure `(let->lambda exp)` that takes in a `let` expression and returns the corresponding `lambda` expression. For example,**
```
(let->lambda '(let ((x 3) (y 10)) (+ x y)))
   ==> ((lambda (x y) (+ x y)) 3 10)
```