5. "Oh My Gawd": It's Full of Stars

What is (*rember** *a* *l*)
where *a* is cup
and
l is ((coffee) cup ((tea) cup)
(and (hick)) cup)

"*rember**" is pronounced "rember-star."

((coffee) ((tea)) (and (hick))).

What is (*rember** *a* *l*)
where *a* is sauce
and
l is (((tomato sauce))
((bean) sauce)
(and ((flying)) sauce))

(((tomato))
((bean))
(and ((flying)))).

Now write *rember**†
Here is the skeleton:

```
(define rember*
  (lambda (a l)
    (cond
      ( ________  ________ )
      ( ________  ________ )
      ( ________  ________ ))))
```

```
(define rember*
  (lambda (a l)
    (cond
      ((null? l) (quote ()))
      ((atom? (car l))
       (cond
         ((eq? (car l) a)
          (rember* a (cdr l)))
         (else (cons (car l)
                 (rember* a (cdr l))))))
      (else (cons (rember* a (car l))
              (rember* a (cdr l)))))))
```

Using arguments from one of our previous examples, follow through this to see how it works. Notice that now we are recurring down the *car* of the list, instead of just the *cdr* of the list.

† " ...* " makes us think "oh my gawd."

(*lat?* *l*)
where
l is (((tomato sauce))
((bean) sauce)
(and ((flying)) sauce))

#f.

Is (*car l*) an atom
where
  l is (((tomato sauce))
    ((bean) sauce)
    (and ((flying)) sauce))

No.

What is (*insertR* new old l*)
where
  new is roast
  old is chuck
and
  l is ((how much (wood))
    could
    ((a (wood) chuck))
    (((chuck)))
    (if (a) ((wood chuck)))
    could chuck wood)

((how much (wood))
 could
 ((a (wood) chuck roast))
 (((chuck roast)))
 (if (a) ((wood chuck roast)))
 could chuck roast wood).

Now write the function *insertR** which inserts the atom *new* to the right of *old* regardless of where *old* occurs.

```
(define insertR*
  (lambda (new old l)
    (cond
      ( ________  ________ )
      ( ________  ________ )
      ( ________  ________ ))))
```

```
(define insertR*
  (lambda (new old l)
    (cond
      ((null? l) (quote ()))
      ((atom? (car l))
       (cond
         ((eq? (car l) old)
          (cons old
            (cons new
              (insertR* new old
                (cdr l)))))
         (else (cons (car l)
                 (insertR* new old
                   (cdr l))))))
      (else (cons (insertR* new old
                    (car l))
              (insertR* new old
                (cdr l)))))))
```

How are *insertR** and *rember** similar?

Each function asks three questions.

The First Commandment

(final version)

When recurring on a list of atoms, *lat*, ask two questions about it: (*null? lat*) and else.
When recurring on a number, *n*, ask two questions about it: (*zero? n*) and else.
When recurring on a list of S-expressions, *l*, ask three question about it: (*null? l*), (*atom?* (*car l*)), and else.

How are *insertR** and *rember** similar?

Each function recurs on the *car* of its argument when it finds out that the argument's *car* is a list.

How are *rember** and *multirember* different?

The function *multirember* does not recur with the *car*. The function *rember** recurs with the *car* as well as with the *cdr*. It recurs with the *car* when it finds out that the *car* is a list.

How are *insertR** and *rember** similar?

They both recur with the *car*, whenever the *car* is a list, as well as with the *cdr*.

How are all *-functions similar?

They all ask three questions and recur with the *car* as well as with the *cdr*, whenever the *car* is a list.

Why?

Because all *-functions work on lists that are either
— empty,
— an atom *cons*ed onto a list, or
— a list *cons*ed onto a list.

The Fourth Commandment

(final version)

Always change at least one argument while recurring. When recurring on a list of atoms, *lat*, use (*cdr lat*). When recurring on a number, *n*, use (*sub1 n*). And when recurring on a list of S-expressions, *l*, use (*car l*) and (*cdr l*) if neither (*null? l*) nor (*atom?* (*car l*)) are true.

It must be changed to be closer to termination. The changing argument must be tested in the termination condition:

when using *cdr*, test termination with *null?* and
when using *sub1*, test termination with *zero?*.

(*occursomething a l*)
where
  a is banana
and
  l is ((banana)
    (split ((((banana ice)))
        (cream (banana))
        sherbet))
    (banana)
    (bread)
    (banana brandy))
	5.

What is a better name for *occursomething*	*occur**.

Write *occur**

```
(define occur*
  (lambda (a l)
    (cond
      ( ________  ________ )
      ( ________  ________ )
      ( ________  ________ ))))
```

```
(define occur*
  (lambda (a l)
    (cond
      ((null? l) 0)
      ((atom? (car l))
       (cond
         ((eq? (car l) a)
          (add1 (occur* a (cdr l))))
         (else (occur* a (cdr l)))))
      (else (+ (occur* a (car l))
               (occur* a (cdr l)))))))
```

(*subst** *new old l*)
where
new is orange
old is banana
and
l is

```
((banana)
 (split ((((banana ice)))
         (cream (banana))
         sherbet))
 (banana)
 (bread)
 (banana brandy))
```

```
((orange)
 (split ((((orange ice)))
         (cream (orange))
         sherbet))
 (orange)
 (bread)
 (orange brandy)).
```

Write *subst**

```
(define subst*
  (lambda (new old l)
    (cond
      ( ________  ________ )
      ( ________  ________ )
      ( ________  ________ ))))
```

```
(define subst*
  (lambda (new old l)
    (cond
      ((null? l) (quote ()))
      ((atom? (car l))
       (cond
         ((eq? (car l) old)
          (cons new
            (subst* new old (cdr l))))
         (else (cons (car l)
                 (subst* new old
                   (cdr l))))))
      (else
        (cons (subst* new old (car l))
          (subst* new old (cdr l)))))))
```

What is (*insertL* new old l*)
where
  new is pecker
  old is chuck
and
  l is ((how much (wood))
    could
    ((a (wood) chuck))
    (((chuck)))
    (if (a) ((wood chuck)))
    could chuck wood)

((how much (wood))
 could
 ((a (wood) pecker chuck))
 (((pecker chuck)))
 (if (a) ((wood pecker chuck)))
 could pecker chuck wood).

Write *insertL**

```
(define insertL*
  (lambda (new old l)
    (cond
      ( ________  ________ )
      ( ________  ________ )
      ( ________  ________ ))))
```

```
(define insertL*
  (lambda (new old l)
    (cond
      ((null? l) (quote ()))
      ((atom? (car l))
       (cond
         ((eq? (car l) old)
          (cons new
            (cons old
              (insertL* new old
                (cdr l)))))
         (else (cons (car l)
                 (insertL* new old
                   (cdr l))))))
      (else (cons (insertL* new old
                    (car l))
              (insertL* new old
                (cdr l)))))))
```

(*member* a l*)
where *a* is chips
and
  l is ((potato) (chips ((with) fish) (chips)))

#t, because the atom chips appears in the list *l*.

Write *member**

```
(define member*
  (lambda (a l)
    (cond
      ( ________  ________ )
      ( ________  ________ )
      ( ________  ________ ))))
```

```
(define member*
  (lambda (a l)
    (cond
      ((null? l) #f)
      ((atom? (car l))
       (or (eq? (car l) a)
         (member* a (cdr l))))
      (else (or (member* a (car l))
              (member* a (cdr l)))))))
```

What is (*member** *a* *l*)
where
a is chips
and
l is ((potato) (chips ((with) fish) (chips)))

#t.

Which chips did it find?

((potato) (<u>chips</u> ((with) fish) (chips))).

What is (*leftmost* *l*)
where
l is ((potato) (chips ((with) fish) (chips)))

potato.

What is (*leftmost* *l*)
where
l is (((hot) (tuna (and))) cheese)

hot.

What is (*leftmost* *l*)
where
l is (((() four)) 17 (seventeen))

No answer.

What is (*leftmost* (**quote** ()))

No answer.

Can you describe what *leftmost* does?

Here is our description:
"The function *leftmost* finds the leftmost atom in a non-empty list of S-expressions that does not contain the empty list."

Is *leftmost* a *-function?

It works on lists of S-expressions, but it only recurs on the *car*.

Does *leftmost* need to ask questions about all three possible cases?

No, it only needs to ask two questions. We agreed that *leftmost* works on non-empty lists that don't contain empty lists.

Now see if you can write the function *leftmost*

```
(define leftmost
  (lambda (l)
    (cond
      (________ ________)
      (________ ________))))
```

```
(define leftmost
  (lambda (l)
    (cond
      ((atom? (car l)) (car l))
      (else (leftmost (car l))))))
```

Do you remember what (**or** ...) does?

(**or** ...) asks questions one at a time until it finds one that is true. Then (**or** ...) stops, making its value true. If it cannot find a true argument, the value of (**or** ...) is false.

What is
  (**and** (*atom?* (*car* *l*))
    (*eq?* (*car* *l*) *x*))
where
  x is pizza
and
  l is (mozzarella pizza)

#f.

Why is it false?

Since (**and** ...) asks (*atom?* (*car* *l*)), which is true, it then asks (*eq?* (*car* *l*) *x*), which is false; hence it is #f.

What is
  (**and** (*atom?* (*car* *l*))
    (*eq?* (*car* *l*) *x*))
where
  x is pizza
and
  l is ((mozzarella mushroom) pizza)

#f.

Why is it false?

Since (**and** ...) asks (*atom?* (*car* *l*)), and (*car* *l*) is not an atom; so it is #f.

Give an example for *x* and *l* where
  (**and** (*atom?* (*car* *l*))
    (*eq?* (*car* *l*) *x*))
is true.

Here's one:
  x is pizza
and
  l is (pizza (tastes good)).

Put in your own words what (**and** ...) does.

We put it in our words:
"(**and** ...) asks questions one at a time until it finds one whose value is false. Then (**and** ...) stops with false. If none of the expressions are false, (**and** ...) is true."

True or false: it is possible that one of the arguments of (**and** ...) and (**or** ...) is not considered?[1]

True, because (**and** ...) stops if the first argument has the value #f, and (**or** ...) stops if the first argument has the value #t.

[1] (**cond** ...) also has the property of not considering all of its arguments. Because of this property, however, neither (**and** ...) nor (**or** ...) can be **defined** as functions in terms of (**cond** ...), though both (**and** ...) and (**or** ...) can be expressed as abbreviations of (**cond** ...)-expressions:
  (**and** α β) = (**cond** (α β) (**else** #f))
and
  (**or** α β) = (**cond** (α #t) (**else** β))

(*eqlist?* *l1* *l2*)
where
  l1 is (strawberry ice cream)
and
  l2 is (strawberry ice cream)

#t.

(*eqlist? l1 l2*) where *l1* is (strawberry ice cream) and *l2* is (strawberry cream ice)	#f.
(*eqlist? l1 l2*) where *l1* is (banana ((split))) and *l2* is ((banana) (split))	#f.
(*eqlist? l1 l2*) where *l1* is (beef ((sausage)) (and (soda))) and *l2* is (beef ((salami)) (and (soda)))	#f, but almost #t.
(*eqlist? l1 l2*) where *l1* is (beef ((sausage)) (and (soda))) and *l2* is (beef ((sausage)) (and (soda)))	#t. That's better.
What is *eqlist?*	It is a function that determines if two lists are equal.
How many questions will *eqlist?* have to ask about its arguments?	Nine.
Can you explain why there are nine questions?	Here are our words: "Each argument may be either — empty, — an atom *cons*ed onto a list, or — a list *cons*ed onto a list. For example, at the same time as the first argument may be the empty list, the second argument could be the empty list or have an atom or a list in the *car* position."

Write *eqlist?* using *eqan?*

```
(define eqlist?
  (lambda (l1 l2)
    (cond
      ((and (null? l1) (null? l2)) #t)
      ((and (null? l1) (atom? (car l2)))
       #f)
      ((null? l1) #f)
      ((and (atom? (car l1)) (null? l2))
       #f)
      ((and (atom? (car l1))
         (atom? (car l2)))
       (and (eqan? (car l1) (car l2))
         (eqlist? (cdr l1) (cdr l2))))
      ((atom? (car l1)) #f)
      ((null? l2) #f)
      ((atom? (car l2)) #f)
      (else
        (and (eqlist? (car l1) (car l2))
          (eqlist? (cdr l1) (cdr l2)))))))
```

Is it okay to ask (*atom?* (*car l2*)) in the second question?

Yes, because we know that the second list cannot be empty. Otherwise the first question would have been true.

And why is the third question (*null? l1*)

At that point, we know that when the first argument is empty, the second argument is neither the empty list nor a list with an atom as the first element. If (*null? l1*) is true now, the second argument must be a list whose first element is also a list.

True or false: if the first argument is () *eqlist?* responds with #t in only one case.

True.
For (*eqlist?* (**quote** ()) *l2*) to be true, *l2* must also be the empty list.

Does this mean that the questions
(**and** (*null? l1*) (*null? l2*))
and
(**or** (*null? l1*) (*null? l2*))
suffice to determine the answer in the first three cases?

Yes. If the first question is true, *eqlist?* responds with #t; otherwise, the answer is #f.

Rewrite *eqlist?*

```
(define eqlist?
  (lambda (l1 l2)
    (cond
      ((and (null? l1) (null? l2)) #t)
      ((or (null? l1) (null? l2)) #f)
      ((and (atom? (car l1))
         (atom? (car l2)))
       (and (eqan? (car l1) (car l2))
         (eqlist? (cdr l1) (cdr l2))))
      ((or (atom? (car l1))
         (atom? (car l2)))
       #f)
      (else
        (and (eqlist? (car l1) (car l2))
          (eqlist? (cdr l1) (cdr l2)))))))
```

What is an S-expression?

An S-expression is either an atom or a (possibly empty) list of S-expressions.

How many questions does *equal?* ask to determine whether two S-expressions are the same?

Four. The first argument may be an atom or a list of S-expressions at the same time as the second argument may be an atom or a list of S-expresssions.

Write *equal?*

```
(define equal?
  (lambda (s1 s2)
    (cond
      ((and (atom? s1) (atom? s2))
       (eqan? s1 s2))
      ((atom? s1) #f)
      ((atom? s2) #f)
      (else (eqlist? s1 s2)))))
```

Why is the second question (*atom? s1*)

If it is true, we know that the first argument is an atom and the second argument is a list.

And why is the third question (*atom? s2*)

By the time we ask the third question we know that the first argument is not an atom. So all we need to know in order to distinguish between the two remaining cases is whether or not the second argument is an atom. The first argument must be a list.

Can we summarize the second question and the third question as
(**or** (*atom? s1*) (*atom? s2*))

Yes, we can!

Simplify *equal?*

```
(define equal?
  (lambda (s1 s2)
    (cond
      ((and (atom? s1) (atom? s2))
       (eqan? s1 s2))
      ((or (atom? s1) (atom? s2))
       #f)
      (else (eqlist? s1 s2)))))
```

Does *equal?* ask enough questions?

Yes.
The questions cover all four possible cases.

Now, rewrite *eqlist?* using *equal?*

```
(define eqlist?
  (lambda (l1 l2)
    (cond
      ((and (null? l1) (null? l2)) #t)
      ((or (null? l1) (null? l2)) #f)
      (else
        (and (equal? (car l1) (car l2))
          (eqlist? (cdr l1) (cdr l2)))))))
```

The Sixth Commandment

Simplify only after the function is correct.

Here is *rember* after we replace *lat* by a list *l* of S-expressions and *a* by any S-expression.

```
(define rember
  (lambda (s l)
    (cond
      ((null? l) (quote ()))
      ((atom? (car l))
       (cond
         ((equal? (car l) s) (cdr l))
         (else (cons (car l)
                 (rember s (cdr l))))))
      (else (cond
              ((equal? (car l) s) (cdr l))
              (else (cons (car l)
                      (rember s
                        (cdr l)))))))))
```

Can we simplify it?

Obviously!

```
(define rember
  (lambda (s l)
    (cond
      ((null? l) (quote ()))
      (else (cond
              ((equal? (car l) s) (cdr l))
              (else (cons (car l)
                      (rember s
                        (cdr l)))))))))
```

And how does that differ?

The function *rember* now removes the first matching S-expression *s* in *l*, instead of the first matching atom *a* in *lat*.

Is *rember* a "star" function now?

No.

Why not?

Because *rember* recurs with the *cdr* of *l* only.

Can *rember* be further simplified?

Yes, the inner (**cond** ...) asks questions that the outer (**cond** ...) could ask!

Do it!

```
(define rember
  (lambda (s l)
    (cond
      ((null? l) (quote ()))
      ((equal? (car l) s) (cdr l))
      (else (cons (car l)
              (rember s (cdr l)))))))
```

Does this new definition look simpler?

Yes, it does!

And does it work just as well?

Yes, because we knew that all the cases and all the recursions were right before we simplified.

Simplify *insertL**

We can't. Before we can ask (*eq?* (*car l*) *old*) we need to know that (*car l*) is an atom.

When functions are correct and well-designed, we can think about them easily.

And that saved us this time from getting it wrong.

Can all functions that use *eq?* and = be generalized by replacing *eq?* and = by the function *equal?*

Not quite; this won't work for *eqan?*, but will work for all others. In fact, disregarding the trivial example of *eqan?*, that is exactly what we shall assume.