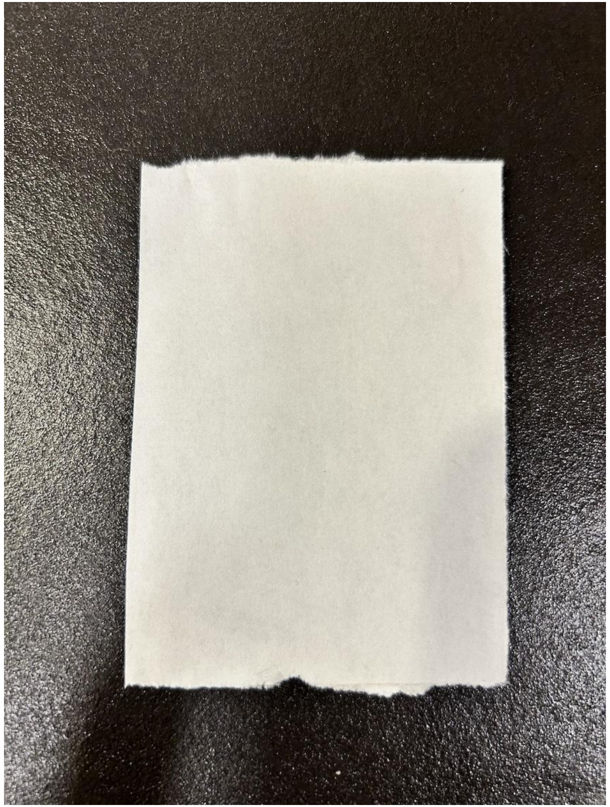
Lists

CS135 Lecture 05

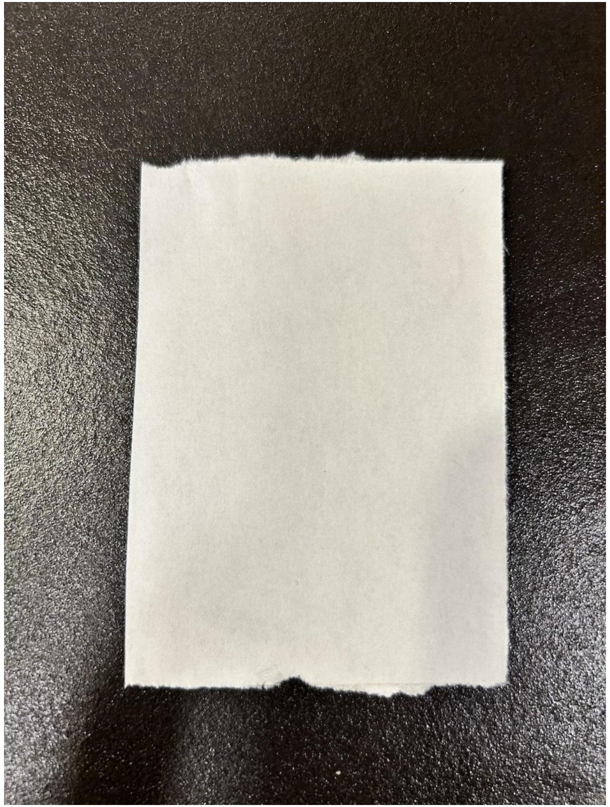
L05.0 List values and expressions

This is an empty list



Since it's empty, we don't know what kind of list it is. It might be a grocery list. It might be a list of things to do.

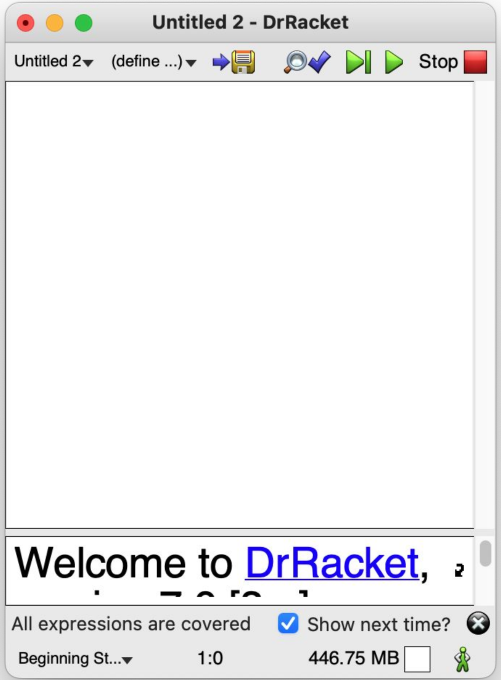
Having an empty list is not the same as having no list



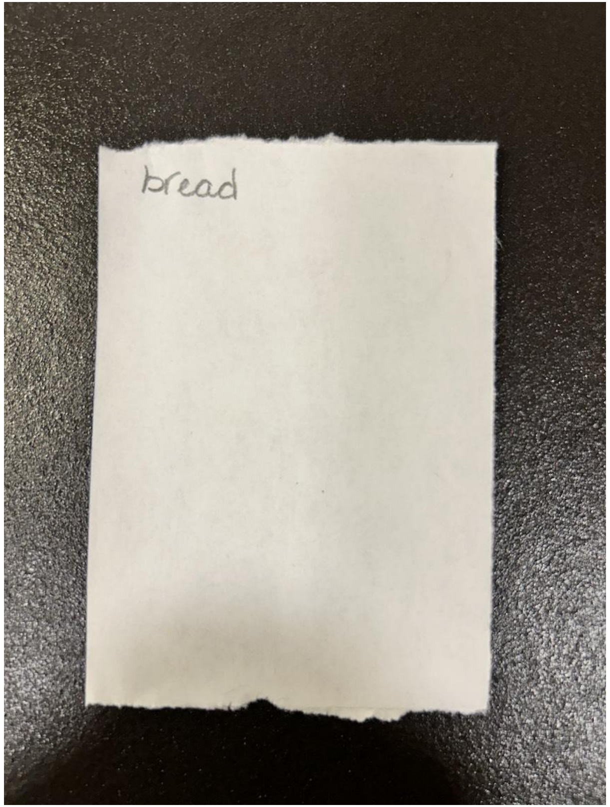
This is an empty list in Racket

Since it's empty, we don't know what kind of list it is. It might be list of **`Int`**. It might be a list of **`Sym`**.

Having an empty list is not the same as having no list

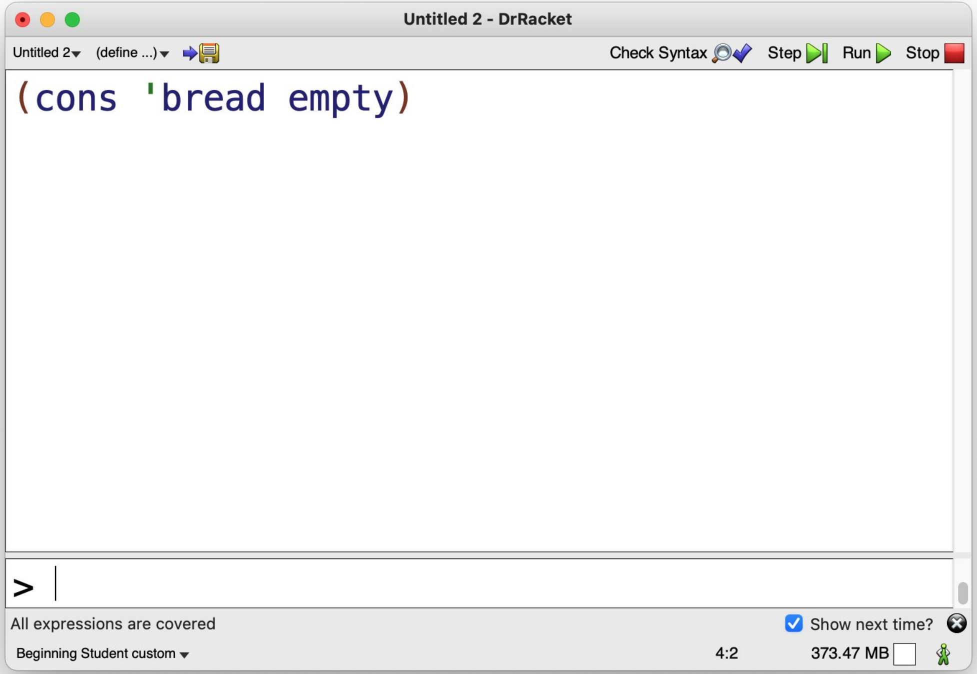


Let's add an item to our list



Looks like a grocery list.

Let's add an item to our Racket list with `cons`

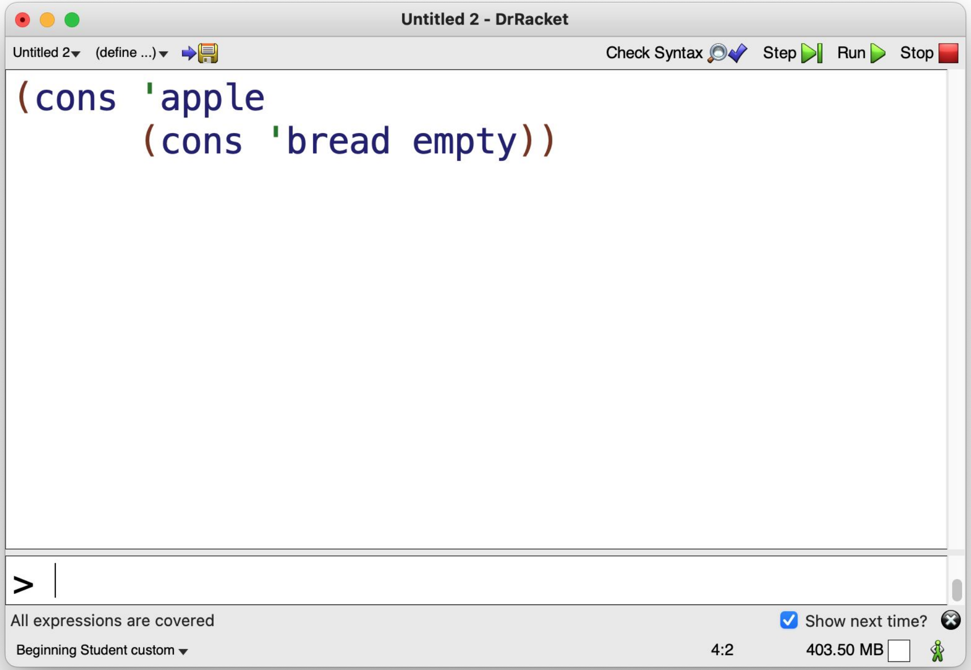


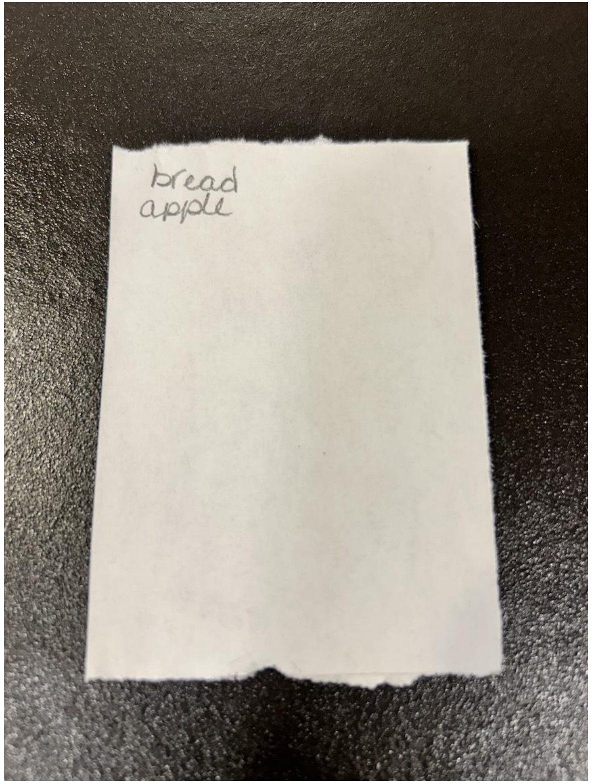
Looks like a list of `Sym`.

`cons` constructs a list by adding an item to the front of another list, e.g. `empty`.

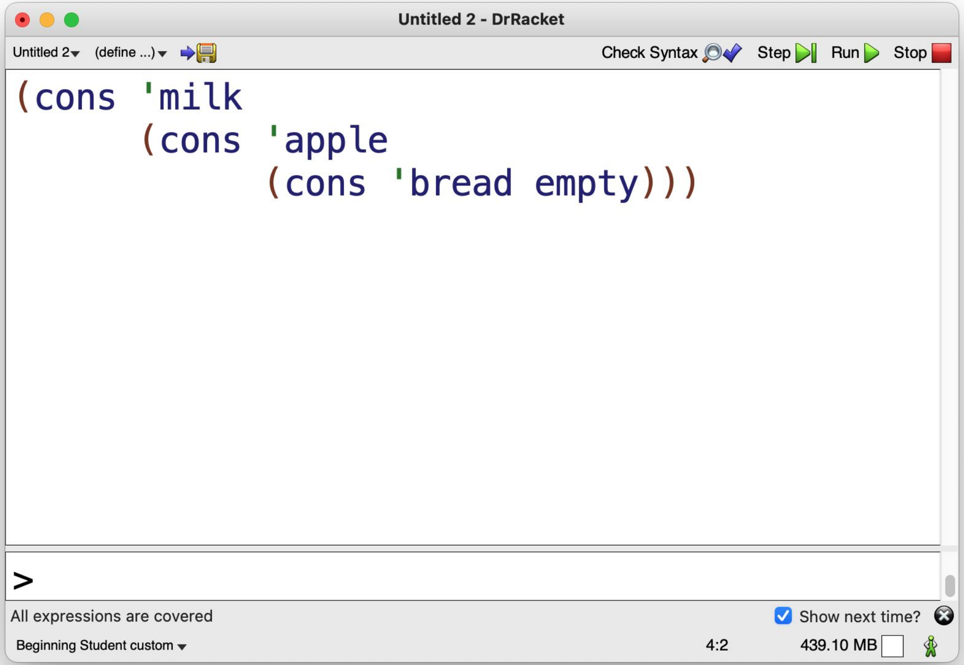
`cons` can be confusing because it can be viewed as a function or a way of representing the resulting value.

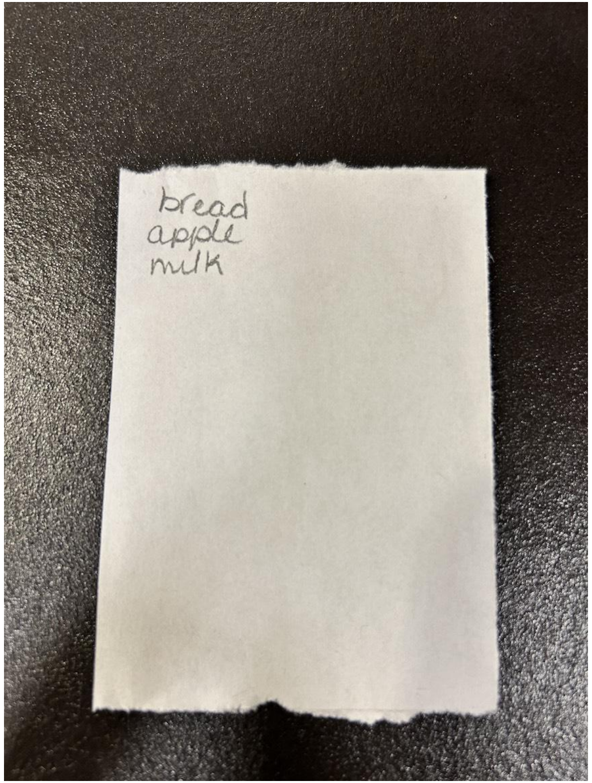
Let's add another item to our list



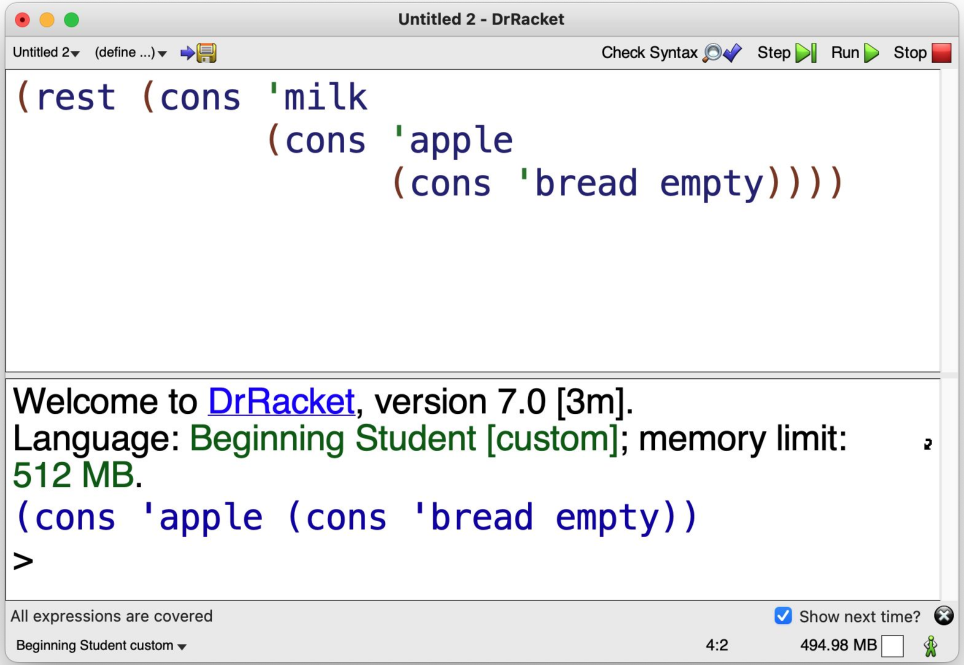


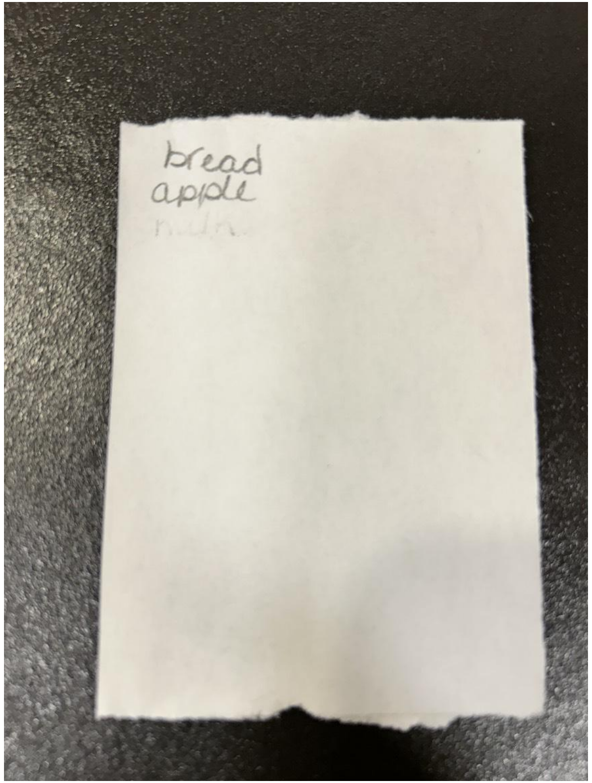
Let’s add a third item



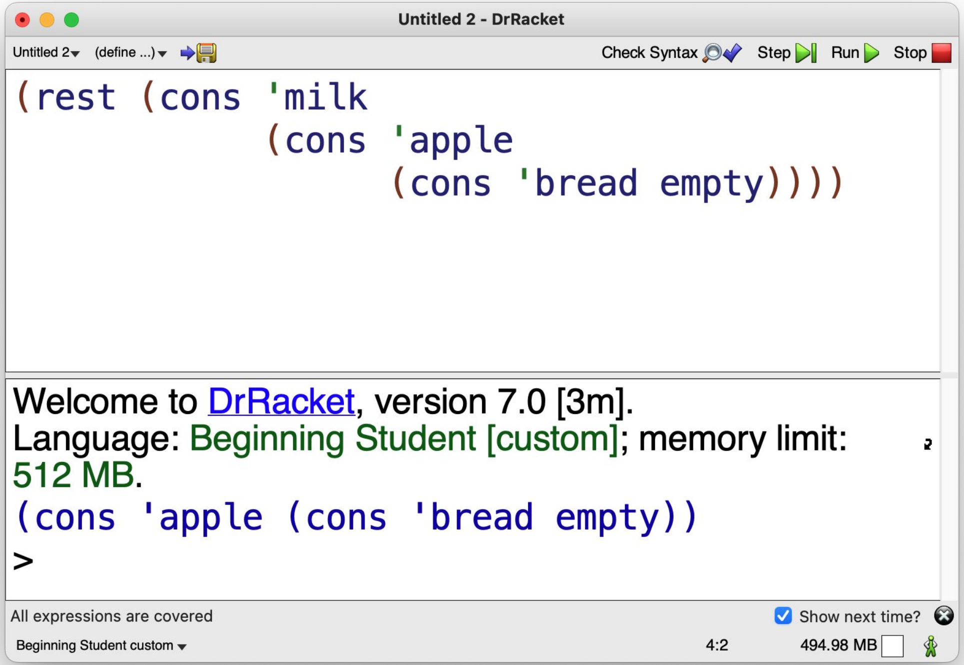


We have milk in the fridge. Let’s erase it.





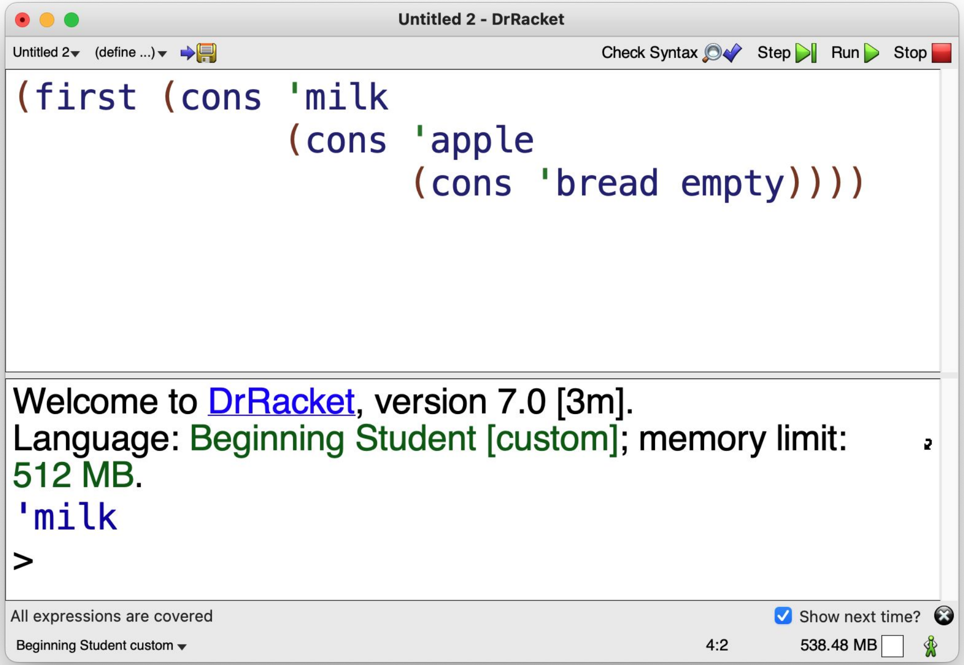
The `rest` function



`rest` is a racket function that consumes a list and produces that list with the first item removed.

It is an error to apply **`rest`** to the **`empty`** list.

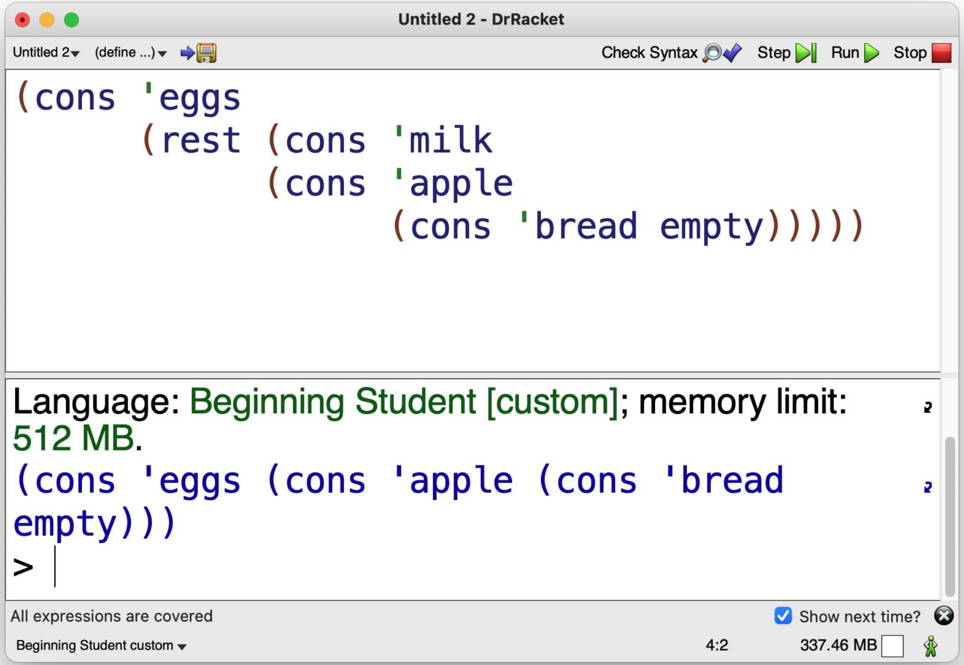
The **`first`** function

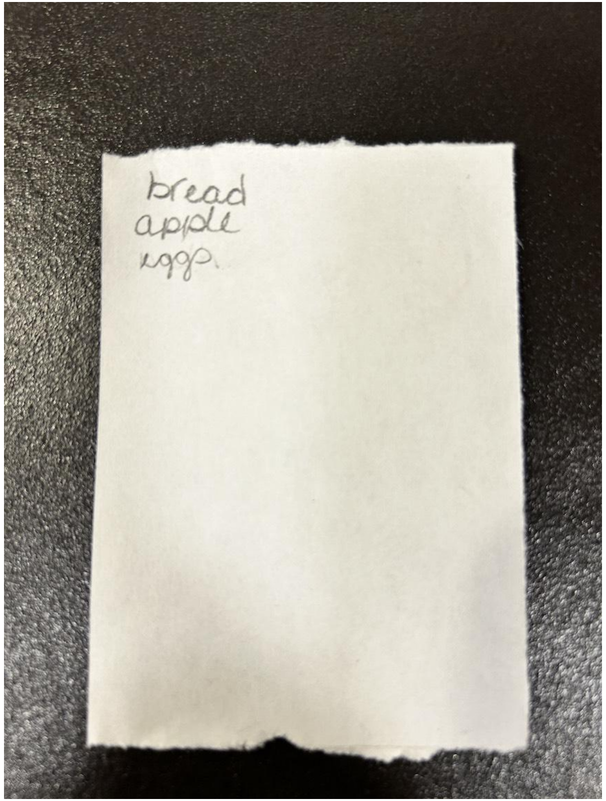


`first` is a racket function that consumes a list and produces the first item of that list.

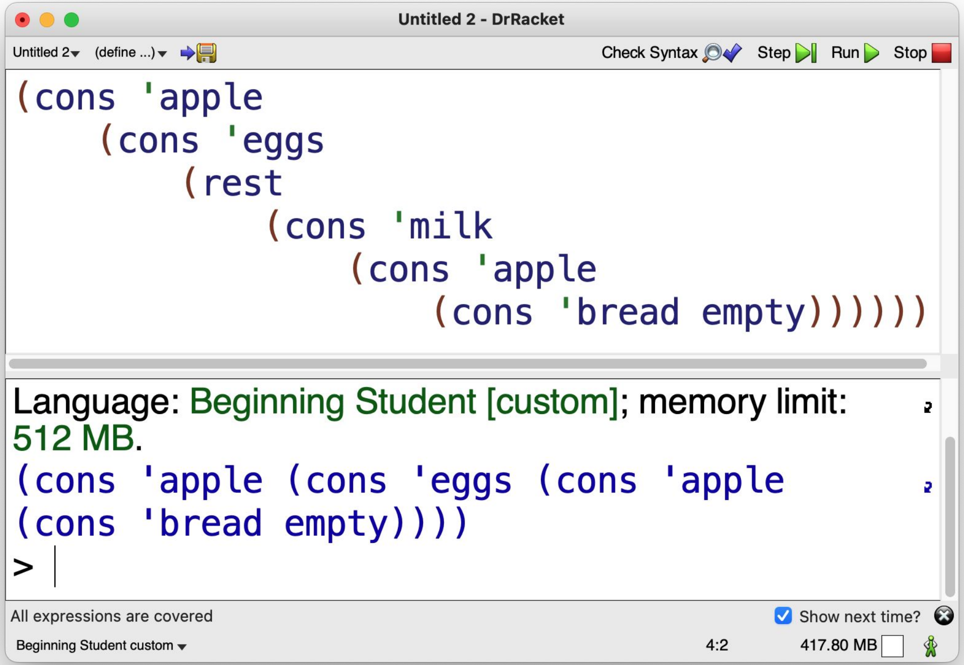
It is an error to apply **`first`** to the **`empty`** list.

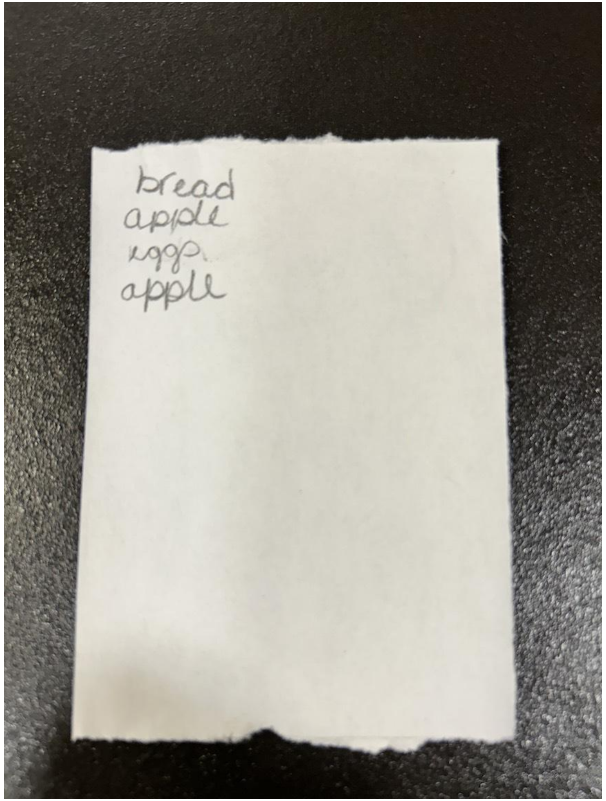
Add some eggs



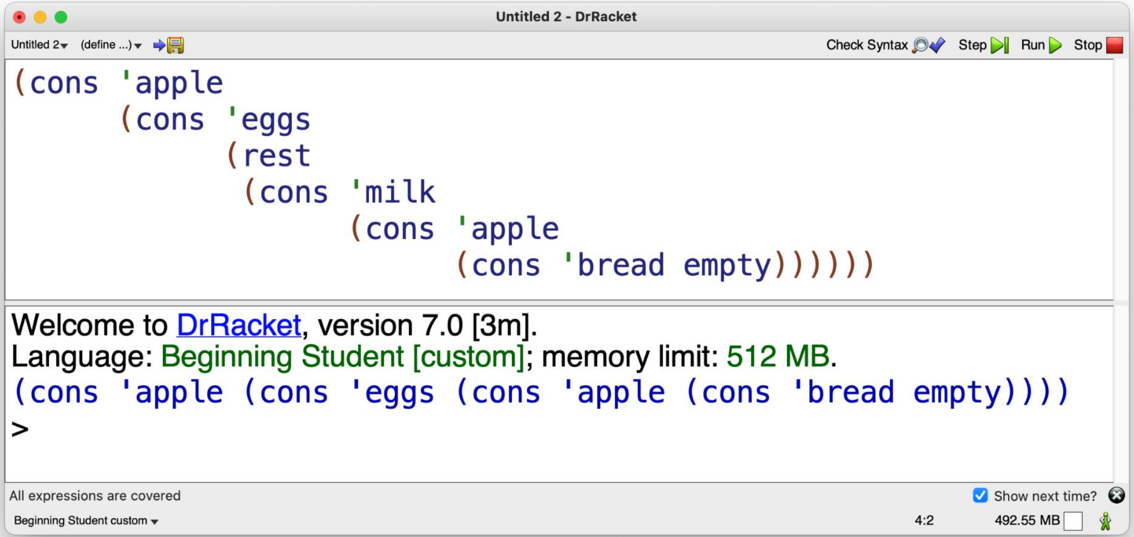


Another apple





Lists are values

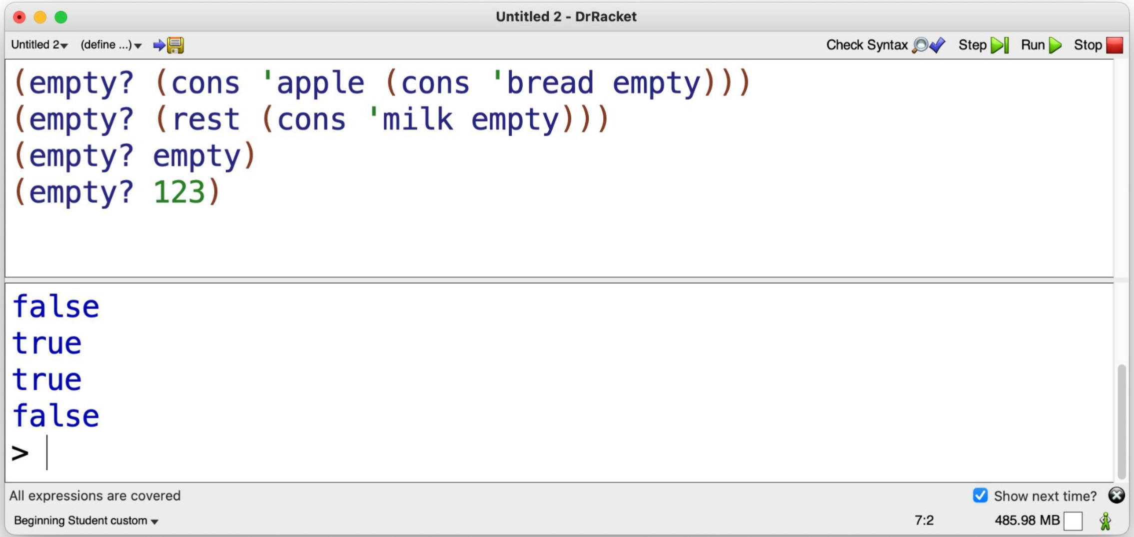


The definitions pane on the top contains an expression (because of `rest`).

The interactions pane on the bottom contains a value.

Lists are the central data structure we use in CS135.

Testing for the empty list with **`empty?`**



`empty?` consumes <u>any value</u> and produces **`true`** only if it is the empty list

List of Racket list operations

`cons`	Constructs a list from a value and a list by adding the value to the front.
`first`	Consumes a non-empty list and produces the first value in that list.
`rest`	Consumes a non-empty list and produces a list with the first value removed.
`empty?`	Consumes **`Any`** and produces **`true`** only if the value is **`empty`**.
`list?`	Consumes **`Any`** and produces **`true`** only if the value is a list.
`cons?`	Consumes **`Any`** and produces **`true`** only if the value is a non-empty list.

We use **`Any`** in contracts to mean a value of any type.

L05.1 Composite data

Composite data

Now that we have lists, we can create data types that are more than just a single number or symbol, i.e., *composite* data types

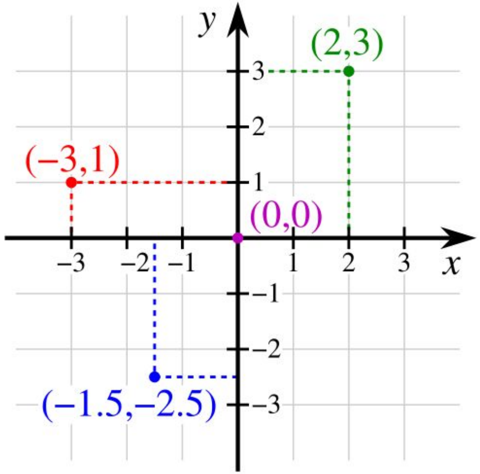
For example, we could use a list of two Num to represent a point in the Cartesian coordinate system: (*x*, *y*).

We represent the point (-3, 1) as:

```
(cons -3 (cons 1 empty))
```

More generally, we represent the point (*x*, *y*) as:

```
(cons x (cons y empty))
```



Distance from the origin

We want to write a function the computes the distance from a point (*x*, *y*) to the origin (0, 0).

This is the first draft of our **purpose**.

From our math classes we know that the distance from (*x*, *y*) to (0, 0) is $\sqrt{x^2 + y^2}$.

In thinking of **examples**, we want some with *x* positive, some with *x* negative, some *y* positive, etc., as well as some with *x* and/or *y* zero.

$(3, -1) \Rightarrow \sqrt{3^2 + -1^2} \approx 3.1622$

$(6, 0) \Rightarrow \sqrt{6^2 + 0^2} = 6$

$(-3, 4) \Rightarrow \sqrt{-3^2 + 4^2} = 5$

$(0, 0) \Rightarrow \sqrt{0^2 + 0^2} = 0$

Header

Let’s give a name to our function.

One possibility is **`distance-to-origin`**, which is accurate but long, with lots of typing. Too long can be confusing in a larger context with lots of functions.

On the other hand, **`d0`**, is short but too cryptic. Let’s err on the side of long.

```
(define (distance-to-origin point)...)
```

Contract

Our function consumes a point and produces a **`Num`**.

A point (*x*, *y*) is represented as a list with two elements giving the **contract**:

```
;; distance-to-origin: (cons Num (cons Num empty)) -> Num
```

We can now finalize our **purpose** as:

```
;; Calculate the distance from a point to the origin.
```

Body

At this point, we understand our problem fairly well, and we have a good idea how the data will be represented in Racket.

For this simple example, it's straightforward to translate the math directly into Racket.

```
(define (distance-to-origin point)
  (sqrt (+
         (sqr (first point)) ; get x from the point
         (sqr (first (rest point))) ; get y from the point
         )))
```

We've added some comments since accessing *x* and *y* seems confusing.

Putting it all together

```
;; Calculate the distance from a point to the origin.
;; distance-to-origin: (cons Num (cons Num empty)) -> Num
(define (distance-to-origin point)
  (sqrt (+
         (sqr (first point)) ; get x from the point
         (sqr (first (rest point))) ; get y from the point
         )))
(check-expect (distance-to-origin (cons -3 (cons 4 empty))) 5)
(check-within
  (distance-to-origin (cons 3 (cons -1 empty))) 3.1622 0.001)
(check-expect (distance-to-origin (cons 6 (cons 0 empty))) 6)
(check-expect (distance-to-origin (cons 0 (cons 0 empty))) 0)
```

L05.2 Data definitions

Data types

We use various types in our contracts to help document the behaviour of our functions.

These types include **`Sym`**, **`Nat`**, **`Rat`**, etc.

The contract for **`distance-to-origin`** may be hard to understand because the data type it consumes is composite.

```
;; distance-to-origin: (cons Num (cons Num empty)) -> Num
```

Data definitions

We can use a *data definition* to give a name to a composite data type.

```
;; a Point is a (x,y) point in the Cartesian coordinate system
;; a Point is a (cons Num (cons Num empty))
```

With this data definition, we can simplify our contract for `distance-to-origin`.

```
;; Calculate the distance from a point to the origin.
;; distance-to-origin: Point -> Num
```

Using helper functions

To make things more understandable, we can write “helper functions” to create a Point (called a “constructor”) and to access its components (called “accessor functions”).

```
;; a Point is a (x,y) point in the Cartesian coordinate system
;; a Point is a (cons Num (cons Num empty)
;;
;; mk-point consumes an x and y coordinate and produces a Point
;; mk-point: Num Num -> Point
(define (mk-point x y) (cons x (cons y empty)))

;; get the x coordinate from a Point
;; get-y: Point -> Num
(define (get-x point) (first point))

;; get the x coordinate from a Point
;; get-y: Point -> Num
(define (get-y point) (first (rest point)))
```

Using helper functions

```
;; Calculate the distance from a point to the origin.
;; distance-to-origin: Point -> Num
(define (distance-to-origin point)
  (sqrt (+ (sqr (get-x point)) (sqr (get-y point)))))

(check-expect (distance-to-origin (mk-point 3 4)) 5)
(check-within (distance-to-origin (mk-point 3 -1)) 3.1622 0.001)
(check-expect (distance-to-origin (mk-point 6 0)) 6)
(check-expect (distance-to-origin (mk-point 0 0)) 0)
```

A note on structures

DrRacket supports a feature called “structures”, which are composite data types similar to the lists in the previous slides. You may see structures mentioned in DrRacket documentation and in previous iterations of CS135.

On the one hand, structures do all the work of creating helper functions. Defining a structure automatically creates functions to assess its components.

On the other hand, lists are much more powerful than structures. Anything you can do with a structure, you can do with a list of fixed size.

In CS135, we have only one composite data type, the list. Almost. As you will see next lecture, data definitions for lists can be recursive, allowing us to work with composite data of arbitrary size.

Data definitions

We can also create data definitions to give names to sets of symbols a function might produce or consume.

```
;; an Outerwear is (anyof 'jacket 'sweater 'shirt)

;; what-to-wear: Num -> Outerwear
(define (what-to-wear temperature)
  (cond [(< temperature 8) 'jacket]
        [(< temperature 16) 'sweater]
        [else 'shirt]))
```

L05.3 🂡🂱🃁🃑

A playing card as a composite data type

The elements of a playing card are called **suits** and **ranks**:

- **Suits** are the categories: ♠ Spades, ♥ Hearts, ♦ Diamonds, ♣ Clubs.
- **Ranks** are the values: A (Ace), 2–10, J (Jack), Q (Queen), K (King).
- We will ignore the jokers for now.

Representing suits and ranks

We can represent suits with symbols:

```
;; A Suit is (anyof 'spade 'heart 'diamond 'club)
```

We can represent ranks with a combination of numbers and symbols:

```
;; A Rank is (anyof 2 3 4 5 6 7 8 9 10 'jack 'queen 'king 'ace)
```

A playing card (**`Card`**) combines a **`Suit`** and a **`Rank`** in a list with two elements:

```
(cons 'heart (cons 6 empty))
(cons 'club (cons 'king empty))
(cons 'diamond (cons 'ace empty))
```

A data definition for playing cards

```
;; A Card is a playing card with a Suit and a Rank
;; A Card is a (cons Suit (cons Rank empty))
(define (mk-card suit rank) (cons suit (cons rank empty)))

;; Get the Suit of a Card
;; get-suit: Card -> Suit
(define (get-suit card) (first card))

;; Get the Rank of a Card
;; get-rank: Card -> Rank
(define (get-rank card) (first (rest Card)))
```

A predicate to determine if a card is a “face card”

A “face card” is a Jack, Queen, or King of any suit.

```
;; Determine if a Card is a face card
;; face-card?: Card -> Bool
(define (face-card? card)
  (or ...
```

We need to test if two ranks are the same.
A rank can be either a symbol or a number.

```
(check-expect (face-card? (mk-card 'heart 6)) false)
(check-expect (face-card? (mk-card 'club 'king)) true)
(check-expect (face-card? (mk-card 'diamond 'ace)) false)
```

Are two ranks the same?

```
;; A Rank is (anyof 2 3 4 5 6 7 8 9 10 'jack 'queen 'king 'ace)

;; Are two ranks the same?
;; rank=?: Rank Rank -> Bool
(define (rank=? rank0 rank1)
  (or (and (symbol? rank0) (symbol? rank1) (symbol=? rank0 rank1))
      (and (number? rank0) (number? rank1) (= rank0 rank1))))

(check-expect (rank=? 'ace 'ace) true)
(check-expect (rank=? 2 2) true)
(check-expect (rank=? 'king 10) false)
```

The contract guarantees that each argument will be a `Rank` (and not `-1.5` or `'blue`)

A predicate to determine if a card is a “face card”

Now we can use **`rank=?`** to test the rank of the card:

```
;; predicate to determine if a Card is a face card
;; face-card?: Card -> Bool
(define (face-card? card)
  (or (rank=? (get-rank card) 'jack)
      (rank=? (get-rank card) 'queen)
      (rank=? (get-rank card) 'king)))

(check-expect (face-card? (mk-card 'heart 6)) false)
(check-expect (face-card? (mk-card 'club 'king)) true)
(check-expect (face-card? (mk-card 'diamond 'ace)) false)
```

Lecture 05 Summary

L05: You should know

- How to create and manipulate lists with **`cons`**, **`first`**, **`rest`**, **`empty`**, **`empty?`**, **`list?`**, and **`cons?`**
- How to write data definitions and helper functions for composite data types using lists.
- How to apply the design recipe to create functions that work with composite data types using lists.

L05: Allowed constructs

Newly allowed constructs:
`Any cons cons? empty empty? first list? rest`

Previously allowed constructs:
`( ) [ ] + - * / = < > <= >= ;`
`abs acos and asin atan boolean? check-expect check-within`
`cond cos define e else exp expt false inexact? integer? log`
`max min not number? or pi quotient rational? remainder sin`
`sqr sqrt sub1 symbol? symbol=? tan true zero?`
`anyof Bool Int Nat Num Rat Sym`

Recursion **must** follow the Rules for Recursion (first version)