Constructing lists in Racket

Welcome to DrRacket.
> (define hufflepuffs (newt tonks cedric))
reference to an identifier before its definition: newt

What went wrong?

Racket tried to evaluate the variable newt.
We had not previously defined a variable called newt.
Nor had we defined tonks or cedric.
**quote inhibits evaluation of composite expressions**

Welcome to DrRacket.

> (quote (newt tonks cedric))
' (newt tonks cedric)

> '(newt tonks cedric)
' (newt tonks cedric)

> (define hufflepuffs
   '(newt tonks cedric))

> hufflepuffs
' (newt tonks cedric)

**Taking lists apart in Racket**

Welcome to DrRacket.

> (define ravenclaws '(luna cho padma))

> ravenclaws
'(luna cho padma)

> (first ravenclaws)
'luna

> (rest ravenclaws)
'(cho padma)

**The first and rest procedures**

(first <non-empty-list>)

The argument to first should be a non-empty list.
first returns the first item on the list.

(rest <non-empty-list>)

The argument to rest should be a non-empty list.
rest returns the new list that results from deleting the first thing
on the given list.
Welcome to DrRacket.
> (define fates '(clotho lachesis atropos))
> fates
'(clotho lachesis atropos)
> (first fates)
'clotho
> (rest fates)
'(lachesis atropos)
> (first (rest fates))
'lachesis
> (rest (rest fates))
'(atropos)
> (first (rest (rest fates)))
'atropos
> (rest (rest (rest fates)))
'()

**first and rest are inverses of cons**

Welcome to DrRacket.
> (define two-things (cons '<thing1> '<thing2>))
> (first two-things)
'<thing1>
> (rest two-things)
'<thing2>

**cons is inverse of first and rest**

Welcome to DrRacket.
> (cons (first '<thing>)
(rest '<thing>))
'<thing>

**Other names for first and rest**

first is traditionally called car.
contents of the address part of register number
rest is traditionally called cdr.
contents of the decrement part of register number
This is a legacy from the original Lisp programming language, circa 1958.
Abbreviations for combinations of car (first) and cdr (rest)

\[
\begin{align*}
(\text{caar } X) &= (\text{car } (\text{car } X)) \\
(\text{cadr } X) &= (\text{car } (\text{cdr } X)) \\
(\text{cdar } X) &= (\text{cdr } (\text{car } X)) \\
(\text{cddr } X) &= (\text{cdr } (\text{cdr } X))
\end{align*}
\]

The list procedure

Takes any number of arguments.
Forms a list out of them.

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

Cyclically permuting a list

Welcome to DrRacket.
> (define lst '(a b c))
> (define permuted-lst ...?...)
> permuted-lst
'(c a b)

Sure, but we wanted to define permuted-lst in terms of lst!
Cyclically permuting a list

Welcome to DrRacket.
> (define lst '(a b c))
> (define permuted-lst
  (list (third lst)
        (first lst)
        (second lst)))
> permuted-lst
'(c a b)

Shuffling two lists

Welcome to DrRacket.
> (define list1 '(a b))
> (define list2 '(c d))
> (define shuffle12 ...?...)
> shuffle12
'(a c b d)
Shuffling two lists

Welcome to DrRacket.
> (define list1 '(a b))
> (define list2 '(c d))
> (define shuffle12
   (list (first list1)
         (first list2)
         (first (rest list1))
         (first (rest list2))))
> shuffle12
'(a c b d)

How to get the symbol \textit{eggs} from the \textit{menu} using \textit{first} and \textit{rest}?

Welcome to DrRacket.
> (define menu
   '(((eggs bacon waffles)
      (burger soup salad)
      (spaghetti steak casserole)
      (ice-cream pie cake)
      (coffee tea milk soda)))
> (define the-eggs (first (first menu)))

How to get the symbol \textit{eggs} from the \textit{menu} using \textit{first} and \textit{rest}?

Welcome to DrRacket.
> (define menu
   '(((eggs bacon waffles)
      (burger soup salad)
      (spaghetti steak casserole)
      (ice-cream pie cake)
      (coffee tea milk soda)))
> (define the-eggs (first (first menu)))
How to get the list of desserts from the menu using \texttt{first} and \texttt{rest}?

Welcome to DrRacket.

> (define menu
  '((eggs bacon waffles)
    (burger soup salad)
    (spaghetti steak casserole)
    (ice-cream pie cake)
    (coffee tea milk soda)))

How to construct this menu using \texttt{list}, quoted symbols, and '()?

Welcome to DrRacket.

> (define menu
  (list
    (list 'eggs 'bacon 'waffles)
    (list 'burger 'soup 'salad)
    (list 'spaghetti 'steak 'casserole)
    (list 'ice-cream 'pie 'cake)
    (list 'coffee 'tea 'milk 'soda)))
How to add the symbol `sandwich` to the lunch section of the menu?

Welcome to DrRacket.

> (define menu
  '([(eggs bacon waffles)
    (burger soup salad)
    (spaghetti steak casserole)
    (ice-cream pie cake)
    (coffee tea milk soda))])

> (define new-menu
  (list
   (first menu)
   (cons 'sandwich (second menu))
   (third menu)
   (fourth menu)))
A pattern may occur over and over

Welcome to DrRacket.
> (* 8 8)
  64
> (* 12 12)
  144
> (* 7 7)
  49
> (* 16 16)
  256

What is the pattern?
> (* <something> <something>)
 <something-squared>

Another pattern may be repeated

Welcome to DrRacket.
> (/ (+ 22 48) 2)
  35
> (/ (+ 91 101) 2)
  96
> (/ (+ 3 27) 2)
  15

What is the pattern?
> (/ (+ <thing1> <thing2>) 2)
 <average-thing1-and-thing2>
**Procedural abstraction**

Captures a pattern in expressions that occur over and over.

Uses the same define mechanism that we saw earlier, along with a special notation for expressing patterns.

---

**Defining the square procedure**

Welcome to DrRacket.

```scheme
> (define square (lambda (x) (* x x)))
64
> (square 8)
64
> (square 12)
144
> (square 7)
49
```

---

**Defining the average procedure**

Welcome to DrRacket.

```scheme
> (define average
  (lambda (x y)
    (/ (+ x y) 2)))
> (average 22 48)
35
> (average 91 101)
96
> (average 3 27)
15
```

---

*Lambda expressions* are *special forms* that evaluate to *procedures.*
The meaning of a **lambda** expression

\[(\text{lambda } (x) (* x x))\]

*Of one argument “x”*

That returns the square of x

---

The meaning of a **lambda** expression

\[(\text{lambda } (x y) (/ (+ x y) 2))\]

*Of two arguments “x” and “y”*

That returns the average of x and y

---

**General form of procedure definitions**

\[
\text{(define } <\text{variable}> \text{ })(\text{lambda } (<\text{arguments}>\!) <\text{expressions}>)
\]

Examples:

\[
\text{(define square } (\text{lambda } (x) (* x x)))
\]

\[
\text{(define average } (\text{lambda } (x y) (/ (+ x y) 2)))
\]

---

**Argument names carry no meaning**

These expressions define the same procedure:

\[
\text{(define square } (\text{lambda } (x) (* x x)))
\]

\[
\text{(define square } (\text{lambda } (\text{fred}) (* \text{fred} \text{fred})))
\]
Argument names carry no meaning

These expressions define the same procedure:

```
(define average
  (lambda (x y)
    (/ (+ x y) 2)))
(define average
  (lambda (romeo juliet)
    (/ (+ romeo juliet) 2)))
```

Argument names must be unique

```
(define average
  (lambda (x x)
    (/ (+ x x) 2)))
```

This procedure definition is syntactically incorrect.

An attempt to process this definition will result in an error message.

A lambda expression has a value (just like any other expression)

The results of evaluating a lambda expression is a procedure:

```
Welcome to DrRacket.
> (lambda (x) (* x x))
#<procedure>
> (define square (lambda (x) (* x x)))
> square
#<procedure:square>
```

Increment and decrement

```
Welcome to DrRacket.
> (define increment (lambda (x) (+ x 1)))
> (define decrement (lambda (x) (- x 1)))
> (increment 0)
1
> (increment 1)
2
> (decrement 2)
1
> (decrement 1)
0
```
Double and half

Welcome to DrRacket.
> (define double (lambda (x) (* x 2)))
> (define half (lambda (x) (/ x 2)))
> (double 1)
2
> (double 2)
4
> (half 4)
2
> (half 2)
1

The substitution model of procedure application

Example:
> (define seven 7)
> (define square (lambda (x) (* x x)))
> (square seven)
49

The substitution model of procedure application

1. Start with: (square seven).
2. Evaluate variables “square” and “seven”.
   The value of square is (lambda (x) (* x x)).
   The value of seven is 7.
   Now we have: ((lambda (x) (* x x)) 7).
3. Replace x with 7 in the body of (lambda (x) (* x x)).
   Now we have: (* 7 7).
4. Evaluate (* 7 7) to get 49.

Positional association

(average 1066 2019)
((lambda (x y) (/ (+ x y) 2)) 1066 2019)

Which actual argument is substituted for x?
The one in the first position, since x is first.
Which actual argument is substituted for y?
The one in the second position, since y is second.
Each of the expressions in the body of the lambda expression is evaluated in turn, but whenever one of the symbols from argument list occurs, it will evaluate to the corresponding input.

The evaluation of symbols in the argument list does not use the global environment.

The expression \((\text{lambda} \ (x) \ x)\) evaluates to a procedure that takes one argument.

The body of the lambda expression has only one expression in this case.

Let’s call this identity function on a Racket expression:

\[(\text{lambda} \ (x) \ x) \ (+ \ 1 \ 2 \ 3)\]

This is a list with two sub-expressions. When evaluated, they yield:

\[(\text{lambda} \ (x) \ x) \rightarrow \text{a procedure}\]
\[(+ \ 1 \ 2 \ 3) \rightarrow \text{the number six}\]

\[(((\text{lambda} \ (x) \ x) \ (+ \ 1 \ 2 \ 3))\]

The result of calling this procedure on the number six is to evaluate the single expression \(x\) that appears in the body of the lambda expression.

However, since the symbol \(x\) is one of the symbols in the argument list of the lambda expression, when we evaluate \(x\), we must use the corresponding input expression (i.e., the number six).

Thus, the result of calling this procedure on the number six is simply the number six:

\[(((\text{lambda} \ (x) \ x) \ (+ \ 1 \ 2 \ 3))\]
\[6\]

The following causes DrRacket to place an entry in the Global Environment for the symbol identity-function:

\[(\text{define} \ \text{identity-function} \ (\text{lambda} \ (x) \ x))\]

Thus, we can use the symbol identity-function to refer to the new procedure:

\[(\text{identity-function} \ 32)\]
\[(\text{identity-function} \ '(1 \ 2 \ 3))\]
\[(\text{identity-function} \ (+ \ 1 \ 2 \ 3))\]
\[(\text{identity-function} \ '+'(1 \ 2 \ 3))\]
Acknowledgments

This lecture incorporates material from:

- Tom Ellman
- Luke Hunsberger