Constructing lists in Scheme

Welcome to DrRacket.

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

What went wrong?

Scheme 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)

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.

Taking lists apart in Scheme

Welcome to DrRacket.
> (define ravenclaws '(luna cho padma))
> ravenclaws
'(luna cho padma)
> (first ravenclaws)
'luna
> (rest ravenclaws)
'(cho padma)

> (define fates '(clotho lachesis atropos))
> (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

> (define two-things
  (cons '<thing1> '<thing2>))
> (first two-things)
'<thing1>
> (rest two-things)
'<thing2>

cons is inverse of first and rest

> (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)

(\texttt{caar} X) = (\texttt{car} (\texttt{car} X))
(\texttt{cadr} X) = (\texttt{car} (\texttt{cdr} X))
(\texttt{cdar} X) = (\texttt{cdr} (\texttt{car} X))
(\texttt{cddr} X) = (\texttt{cdr} (\texttt{cdr} X))

The \texttt{list} procedure

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

\begin{verbatim}
\texttt{\textgreater{} (list \texttt{1} \texttt{2})
 '(\texttt{1} \texttt{2})
\texttt{\textgreater{} (list \texttt{'a} \texttt{'b} \texttt{'c})
 '(\texttt{a} \texttt{b} \texttt{c})
\texttt{\textgreater{} (list (list \texttt{1} \texttt{2}) (list \texttt{'a} \texttt{'b} \texttt{'c}))
 '(((\texttt{1} \texttt{2}) (\texttt{a} \texttt{b} \texttt{c}))
\texttt{\textgreater{} (list)
 '())
\end{verbatim}

Cyclically permuting a list

\begin{verbatim}
\texttt{\textgreater{} (define \texttt{lst} '(\texttt{a} \texttt{b} \texttt{c}))
\texttt{\textgreater{} (define \texttt{permuted-lst} ...??...)}
\texttt{\textgreater{} \texttt{permuted-lst}
 '(\texttt{c} \texttt{a} \texttt{b})
\end{verbatim}

Cyclically permuting a list

\begin{verbatim}
\texttt{\textgreater{} (define \texttt{lst} '(\texttt{a} \texttt{b} \texttt{c}))
\texttt{\textgreater{} (define \texttt{permuted-lst} '(\texttt{c} \texttt{a} \texttt{b}))
\texttt{\textgreater{} \texttt{permuted-lst}
 '(\texttt{c} \texttt{a} \texttt{b})
\texttt{\textgreater{} Sure, but we wanted to define \texttt{permuted-lst} in terms of \texttt{lst}!}
\end{verbatim}
Cyclically permuting a list

> (define lst '(a b c))
> (define permuted-lst
  (list (third lst)
       (first lst)
       (second lst)))
> permuted-lst
'(c a b)

Shuffling two lists

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

> (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)

**The CMPU 101 diner**

**How to get the symbol eggs from the menu using first and rest?**

> (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))

> (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 first and rest?

> (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 list, quoted symbols, and '()?

> (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**?

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

How to add the symbol **sandwich** to the lunch section of the **menu**?

```scheme
> (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)))
```

**Procedural abstraction**
A pattern may occur over and over

\[ \times 8 \times 8 \]
\[ 64 \]
\[ \times 12 \times 12 \]
\[ 144 \]
\[ \times 7 \times 7 \]
\[ 49 \]
\[ \times 16 \times 16 \]
\[ 256 \]

What is the pattern?

\[ \times \underline{\text{something}} \underline{\text{something}} \]
\[ \text{something-squared} \]

Another pattern may be repeated

\[ \div (\times 22 48) 2 \]
\[ 35 \]
\[ \div (\times 91 101) 2 \]
\[ 96 \]
\[ \div (\times 3 27) 2 \]
\[ 15 \]

What is the pattern?

\[ \div (\times \underline{\text{thing1}} \underline{\text{thing2}}) 2 \]
\[ \text{average-thing1-and-thing2} \]
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
\[
> (define \textbf{square} \ (\lambda \ (x) \ (* \ x \ x)))
> (\textbf{square} \ 8) \\
64
> (\textbf{square} \ 12) \\
144
> (\textbf{square} \ 7) \\
49
\]

Defining the average procedure
\[
> (define \textbf{average} \ (\lambda \ (x \ y) \ (/ \ (+ \ x \ y) \ 2)))
> (\textbf{average} \ 22 \ 48) \\
35
> (\textbf{average} \ 91 \ 101) \\
96
> (\textbf{average} \ 3 \ 27) \\
15
\]

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

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

*A procedure*

*Of one argument “x”*

*That returns the square of x*

---

The meaning of a **lambda** expression

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

*A procedure*

*Of two arguments “x” and “y”*

*That returns the average of x and y*

---

**General form of procedure definitions**

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

Examples:

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

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

---

**Argument names carry no meaning**

These expressions define the same procedure:

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

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

These expressions define the same procedure:

\[
\begin{align*}
&\text{(define \textit{average} (\texttt{lambda} (x y) (/ (+ x y) 2)))} \\
&(\text{define \textit{average} (\texttt{lambda} (\textit{romeo} \textit{juliet}) (/ (+ \textit{romeo} \textit{juliet}) 2)))}
\end{align*}
\]

Argument names must be unique

\[
\begin{align*}
&\text{(define \textit{average} (\texttt{lambda} (x x) (/ (+ x x) 2)))}
\end{align*}
\]

This procedure definition is \textit{syntactically incorrect}.

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

A \texttt{lambda} expression has a value (just like any other expression)

The results of evaluating a lambda expression is a procedure:

\[
\begin{align*}
&>\ (\texttt{lambda} (x) (* x x)) \\
&\text{<procedure>}
\end{align*}
\]

\[
\begin{align*}
&>\ (\texttt{define \textit{square} (lambda} (x) (* x x))) \\
&\textit{square} \\
&\text{<procedure:square>}
\end{align*}
\]

Increment and decrement

\[
\begin{align*}
&>\ (\texttt{define \textit{increment} (lambda} (x) (+ x 1))) \\
&>\ (\texttt{define \textit{decrement} (lambda} (x) (- x 1))) \\
&>\ (\text{increment 0}) \\
&\text{1} \\
&>\ (\text{increment 1}) \\
&\text{2} \\
&>\ (\text{decrement 2}) \\
&\text{1} \\
&>\ (\text{decrement 1}) \\
&\text{0}
\end{align*}
\]
Double and half

\[
\begin{align*}
& \text{> (define double (lambda (x) (* x 2)))} \\
& \text{> (define half (lambda (x) (/ x 2)))} \\
& \text{> (double 1)} \\
& \text{> 2} \\
& \text{> (double 2)} \\
& \text{> 4} \\
& \text{> (half 4)} \\
& \text{> 2} \\
& \text{> (half 2)} \\
& \text{> 1}
\end{align*}
\]

The substitution model of procedure application

Example:

\[
\begin{align*}
& \text{> (define seven 7)} \\
& \text{> (define square (lambda (x) (* x x)))} \\
& \text{> (square seven)} \\
& \text{> 49}
\end{align*}
\]

The substitution model of procedure application

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

Positional association

\((\text{average 1066 2019})\)
\(((\text{lambda (x y) (/ (+ x y) 2)) 1066 2019})\)

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 Scheme expression:

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

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

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

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

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

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