Procedural abstractions

- So far, we have talked about primitive data elements and done various levels of abstraction using procedures only
  - This is a key capability in being able to recognize and implement common behaviors
- The ability to combine data elements will further extend our ability to model the world

Compound data

- In order to build compound structures we need a way to combine elements and refer to them as a single blob
- We can write a lambda expression that combines one or more expressions
- We can write a cons expression that ties two data elements together

(cons a b)

- Takes a and b as args, returns a compound data object that contains a and b as its parts
- We can extract the two parts with accessor functions car and cdr ("could-er")

(car (cons 3 4))
(cdr (cons 3 4))

(car (cons 3 4))
(cdr (cons 3 4))

http://www.cs.washington.edu/education/courses/413/04au/
car and cdr

\(\text{define } a \ (\text{cons } 1 \ 2)\)

\(\text{define } b \ (\text{cons } a \ 3)\)

\(\text{car } (\text{car } b)\)

\(\text{cdr } (\text{car } b)\)

\(\text{cdr } b\)

\(\text{cdr } (\text{cdr } c)\)

\(\text{define } c \ (\text{cons } (\text{cons } 1 \ 2) \ (\text{cons } 3 \ 4))\)

\(\text{car } (\text{car } c)\)

\(\text{cdr } (\text{car } c)\)

\(\text{car } (\text{cdr } c)\)

\(\text{cdr } (\text{cdr } c)\)

\(\text{pair? predicate}\)

\(\text{pair? } z\) is true if \(z\) is a pair

\(\text{define } c \ (\text{cons } (\text{cons } 1 \ 2) \ (\text{cons } 3 \ 4))\)

\(\text{pair? } c\)

\(\text{pair? } (\text{car } c)\)

\(\text{pair? } (\text{cdr } c)\)

\(\text{pair? } (\text{car } (\text{car } c))\)

\(\text{pair? } (\text{car } (\text{cdr } c))\)

\(\text{pair? } (\text{cdr } (\text{car } c))\)

\(\text{pair? } (\text{cdr } (\text{cdr } c))\)

\(\text{nil}\)

\(\text{define } d \ (\text{cons } 1 \ '())\)

\(\text{null? } (\text{car } d)\)

\(\text{null? } (\text{cdr } d)\)

\(\text{null? } (\text{car } (\text{car } d))\)

\(\text{null? } (\text{car } (\text{cdr } d))\)

\(\text{What do we really know about pairs?}\)

\(\text{An Application Programming Interface (API)}\)

\(\text{cons} - \text{constructor}\)

\(\text{car}, \text{cdr} - \text{accessor functions}\)

\(\text{We may think we know how they are stored}\)

\(\text{box-and-pointer drawings}\)

\(\text{pointers to pointer blocks ...}\)

\(\text{But if we can stay at the API level, the separation between layers of implementation can stay clean which is a "good thing"\)
Can we implement cons/car/cdr?

- If we focus on the behaviors that are defined, what do we actually need to do?
  - \((\text{cons } a \ b)\)

- \((\text{car } \text{something})\)

- \((\text{cdr } \text{something})\)

We tend to think of the \text{something} returned by \text{cons} as a structured data variable of some sort.

However, the only actual requirement on \text{something} is that we can recover \(a\) and \(b\) from it using procedures named \text{car} and \text{cdr}.

How about we use a procedure definition for \text{something} ...

Procedural representation of pairs

\begin{verbatim}
(define (cons x y)
  (lambda (m) (m x y)))

(define (car z)
  (z (lambda (p q) p)))

(define (cdr z)
  (z (lambda (p q) q)))
\end{verbatim}

Procedural cons and car

\begin{verbatim}
(define (cons x y)
  (lambda (m) (m x y)))

(define (car z)
  (z (lambda (p q) p)))

(define (cdr z)
  (z (lambda (p q) q)))
\end{verbatim}

Lexical closure

- Take another look at the definition of \text{cons}

\begin{verbatim}
(define (cons x y)
  (lambda (m) (m x y)))
\end{verbatim}

- Where did the values of \(x\) and \(y\) come from?

- Are they still around when we call \text{car} / \text{cdr}?
definition and execution

\[ \text{(define (cons x y)} \] \[ \text{(lambda (m) (m x y))}) \]
• x and y are referenced in the environment of the lambda expression’s definition
  » its lexical environment, which is in the definition of cons
• not the environment of its execution
  » its dynamic environment, which is in car

Pairs are the glue

• Using cons to build pairs, we can build data structures of unlimited complexity
• We can roll our own

• We can adopt a standard and use it for the basic elements of more complex structures

Lists

• By convention, a list is a sequence of linked pairs
  » car of each pair is the data element
  » cdr of each pair points to list tail or the empty list

\[ \text{(define e (cons 1 (cons 2 (cons 3 '()))))} \]
\[ \text{(define e (list 1 2 3))} \]

List construction

\[ \text{(define a (cons 1 (cons 2 (cons 3 '()))))} \]
\[ \text{(define a (list 1 2 3))} \]

procedure list

\[ \text{(list a b c ...)} \]
• list returns a newly allocated list of its arguments
  » the arguments can be atomic items like numbers or quoted symbols
  » the arguments can be other lists
• The backbone structure of a list is always the same
  » a sequence of linked pairs, ending with a pointer to null (the empty list)
  » the car element of each pair is the list item
  » the list items can be other lists

List structure

\[ \text{(define a (list 4 5 6))} \]
\[ \text{(define b (list 7 a 8))} \]
Examples of list building

- \(\text{cons} \ 1 \ (\text{cons} \ 2 \ '())\)
- \(\text{cons} \ 1 \ (\text{list} \ 2)\)
- \(\text{list} \ 1 \ 2\)

How to process lists?

- A list is zero or more connected pairs
- Each node is a pair
- Thus the parts of a list (this pair, following pairs) are lists
- A natural way to express list operations?

\[
\begin{align*}
\text{cdr} \ down \\
(\text{define} \ (\text{length} \ m)) \\
(\text{if} \ (\text{null?} \ m) \ 0 \\
\ (+ \ 1 \ (\text{length} \ (\text{cdr} \ m))))
\end{align*}
\]

\[
\begin{align*}
\text{sum the items in a list} \\
(\text{define} \ (\text{add-items} \ (\text{list} \ 2 \ 5 \ 4)) \\
(\text{define} \ (\text{add-items} \ m)) \\
(\text{if} \ (\text{null?} \ m) \ 0 \\
\ (+ \ (\text{car} \ m) \ (\text{add-items} \ (\text{cdr} \ m))))
\end{align*}
\]

\[
\begin{align*}
\text{Yet another list function} \\
(\text{define} \ e \ (\text{cons} \ 1 \ (\text{cons} \ 2 \ (\text{cons} \ 3 \ '()))))
\end{align*}
\]

\[
\begin{align*}
\text{cons up} \\
(\text{define} \ (\text{reverse} \ m)) \\
(\text{define} \ (\text{iter} \ \text{shrnk} \ \text{grow})) \\
(\text{if} \ (\text{null?} \ \text{shrnk}) \ \text{grow} \\
(\text{iter} \ (\text{cdr} \ \text{shrnk}) \ (\text{cons} \ (\text{car} \ \text{shrnk}) \ \text{grow}))) \ (\text{iter} \ m \ '())
\end{align*}
\]
multiply each list element by 2

```
(define (double-all m)
  (if (null? m)
      '()
      (cons (* 2 (car m)) (double-all (cdr m)))))

(double-all (list 4 0 -3))
```

Variable number of arguments

- We can define a procedure that has zero or more required parameters, plus provision for a variable number of parameters to follow
  - The required parameters are named in the `define` statement as usual
  - They are followed by a "." and a single parameter name
- At runtime, the single parameter name will be given a list of all the remaining actual parameter values

```
(same-parity x . y)
(define (same-parity x . y)
  ...)

> (same-parity 1 2 3 4 5 6 7)
(1 3 5 7)
> (same-parity 2 3 4 5 6 7)
(2 4 6)
>
The first argument value is assigned to x, all the rest are assigned as a list to y

map

- We can use the general purpose function `map` to map over the elements of a list and apply some function to them

```
(define (map p m)
  (if (null? m)
      '()
      (cons (p (car m))
            (map p (cdr m)))))

(define (double-all m)
  (map (lambda (x) (* 2 x)) m))
```