Scheme: Closures

CSE 413, Autumn 2007  
Programming Languages

Review: Higher Order Functions

• Take other functions as arguments (or)
• Return a function as a result

Review: map

• Example of a built in higher order function
• (map function alist)  
  » applies function to the elements of alist

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

Can we implement cons/car/cdr?

• If we focus on the behaviors that are defined what do we actually need to do?
  • (cons a b)
  • (car something)
  • (cdr something)

something

• We tend to think of the something returned by cons as a structured data variable of some sort
• However, the only actual requirement on something is that we can recover a and b from it  
  using procedures named car and cdr
• How about we use a procedure definition for something ...

Procedural representation of pairs

(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)))

usage

(define a (cons 1 2))
(car a)
(cdr a)
Procedural cons and car

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

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

Lexical closure

- Take another look at the definition of cons
- \begin{verbatim}
(define (cons x y)
  (lambda (m) (m x y)))
(define (car z)
  (z (lambda (p q) p)))
\end{verbatim}
- Where did the values of x and y come from?
- Are they still around when we call car / cdr?

current symbol definitions

- Lambda expressions evaluate to what is called a lexical closure
  - a coupling of code and a lexical environment (a scope)
  - The lexical environment is necessary because the code needs a place to look up the definitions of symbols it references

definition and execution

- 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

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

\begin{verbatim}
(define (same-parity x . y)
  ...)
\end{verbatim}

\begin{verbatim}
> (same-parity 1 2 3 4 5 6 7)
(1 3 5 7)
> (same-parity 2 3 4 5 6 7)
(2 4 6)
\end{verbatim}

The first argument value is assigned to x, all the rest are assigned as a list to y