Class 4: Lambda

SI 413 - Programming Languages and Implementation

Dr. Daniel S. Roche

United States Naval Academy

Fall 2011
Procedures are First-Class

Functional languages generally give procedures *first-class status*:

- They can be given names.
- They can be arguments to procedures.
- They can be returned by procedures.
- They can be stored in data structures (e.g. lists).
Procedures are First-Class

Functional languages generally give procedures *first-class status*:

- They can be given names.
  
  \[
  \text{(define (f x) (+ x 10))}
  \]

- They can be arguments to procedures.

- They can be returned by procedures.

- They can be stored in data structures (e.g. lists).
Procedures are First-Class

Functional languages generally give procedures *first-class status*:

- They can be given names.
- They can be arguments to procedures.
  
  (filter symbol? '(1 a b 5))
  (map + '(1 2 3) '(4 5 6))

- They can be returned by procedures.
- They can be stored in data structures (e.g. lists).
Procedures returning procedures

Example: Get the Java division procedure for a sample input

(define (java-divider sample)
  (if (inexact? sample) / quotient))
Procedures returning procedures

Example: Get the Java division procedure for a sample input

\[
(\text{define (java-divider sample)}
  \begin{align*}
   & (\text{if (inexact? sample)} / \text{quotient})
\end{align*}
\]

Useful when combined with higher-order procedures:

\[
(\text{define (java-divide-all tops bottoms)}
  \text{(map (java-divider (car tops)) tops bottoms)})
\]
Storing procedures in a list

Maybe we want to apply different functions to the same data:

```
(define (apply-all alof alon)
  (if (null? alof)
    ()
    (cons ((car alof) alon)
      (apply-all (cdr alof) alon)))))
```

Then we can get statistics on a list of numbers:

```
(apply-all (list length mean stdev) (list 2.4 5 3.2 3 8))
```
The lambda calculus is a way of expressing computation

Developed by Alonzo Church (left) in the 1930s

Believed to cover everything that is computable (Church-Turing thesis)

*Everything* is a function: numbers, points, booleans, . . .

Functions are just a kind of data!
Anonymous functions in Scheme

`lambda` is a special form in Scheme that creates a nameless function:

```
(lambda (arg1 arg2 ...) 
  expr-using-args)
```
Anonymous functions in Scheme

**lambda** is a special form in Scheme that creates a nameless function:

```
(lambda (arg1 arg2 ...) 
    expr-using-args)
```

```
(define (make-adder n) (lambda (x) (+ n x)))
```
Anonymous functions in Scheme

\texttt{lambda} is a special form in Scheme that creates a nameless function:

\begin{verbatim}
(lambda (arg1 arg2 ...)  
    expr-using-args)
\end{verbatim}

\begin{verbatim}
(define (make-adder n) (lambda (x) (+ n x)))
\end{verbatim}

\begin{verbatim}
(define (double f) (lambda (x) (f (f x))))
\end{verbatim}
Lambda with higher-order functions

Remember the range function:

```
(define (range a b)
  (if (> a b) null (cons a (range (+ a 1) b))))
```

Write the following functions **without** using recursion.

1. **(half L)** divides each element in L by 2.
2. **(facsum n)** gives the sum of all integers less than n that divide n.
3. **(my-factorial n)** computes n!
4. **(my-length L)** returns the length of the list L.
Behind the curtain

You have already been using `lambda`!

- `(define (f x1 x2 ... xn) exp-using-xs)`
  is the same as

  `(lambda (x1 x2 ... xn) exp-using-xs)`

  `(let ((x1 e1) (x2 e2) ... (xn en)) exp-using-xs)`
Behind the curtain

You have already been using `lambda`!

- `(define (f x1 x2 ... xn) exp-using-xs)`
  is the same as
  `(define f (lambda (x1 ... xn) exp-using-xs))`
Behind the curtain

You have already been using \texttt{lambda}!

- $(\text{define } (f \ x_1 \ x_2 \ \ldots \ x_n) \ \text{exp}\text{-}using\text{-}xs)$
  is the same as
  $(\text{define } f \ (\text{\texttt{lambda} } (x_1 \ \ldots \ x_n) \ \text{exp}\text{-}using\text{-}xs))$

- $(\text{let } ((x_1 \ e_1) \ (x_2 \ e_2) \ \ldots \ (x_n \ e_n)) \ \text{exp}\text{-}using\text{-}xs)$
  is the same as

Behind the curtain

You have already been using lambda!

- \((\text{define } (f \ x1 \ x2 \ldots \ xn) \ \text{exp-using-xs})\)
  is the same as
  \((\text{define } f \ (\lambda (x1 \ldots \ xn) \ \text{exp-using-xs}))\)

- \((\text{let } ((x1 \ e1) \ (x2 \ e2) \ldots \ (xn \ en)) \ \text{exp-using-xs})\)
  is the same as
  \(((\lambda (x1 \ x2 \ldots \ xn) \ \text{exp-using-xs}) \ e1 \ e2 \ldots \ en)\)