CS450

Structure of Higher Level Languages

Lecture 10: Map, zip, enumerate, filter, expression evaluation

Tiago Cogumbreiro
The map stream
Map for streams

Given a stream $s$ defined as

$$e_0 \ e_1 \ e_2 \ e_3 \ e_4 \ \ldots$$

and a function $f$ the stream $(\text{stream-map } f \ s)$ should yield

$$(f \ e_0) \ (f \ e_1) \ (f \ e_2) \ (f \ e_3) \ (f \ e_4) \ \ldots$$
Map for streams

Spec

```racket
#lang racket
(require rackunit)

(define s0
  (stream-map (curry + 2) (naturals)))
(check-equal? (stream-get s0) 2)

(define s1 (stream-next s0))
(check-equal? (stream-get s1) 3)

(define s2 (stream-next s1))
(check-equal? (stream-get s2) 4)
```
Map for streams

Spec

```racket
#lang racket
(require rackunit)

(define s0
  (stream-map (curry + 2) (naturals))))
(check-equal? (stream-get s0) 2)

(define s1 (stream-next s0))
(check-equal? (stream-get s1) 3)

(define s2 (stream-next s1))
(check-equal? (stream-get s2) 4)
```

Solution

```racket
(define (stream-map f s)
  (define (stream-map-iter s)
    (cons (f (stream-get s))
          (thunk (stream-map-iter (stream-next s))))))

(define (stream-map f s)
  (define (stream-map-iter s)
    (cons (f (stream-get s))
          (thunk (stream-map-iter (stream-next s))))))
```
The stream of even numbers
Even naturals

Build a stream of even numbers. Tip: use stream-map and naturals.

0 2 4 6 8 10 12 ...

Spec

```racket
#lang racket
(requiere rackunit)
(define s0 (even-naturals))
(check-equal? (stream-get s0) 0)

(define s1 (stream-next s0))
(check-equal? (stream-get s1) 2)

(define s2 (stream-next s1))
(check-equal? (stream-get s2) 4)
```
Even naturals

Build a stream of even numbers. Tip: use stream-map and naturals.

0 2 4 6 8 10 12 ...

Spec

```racket
#lang racket
(require rackunit)
(define s0 (even-naturals))
(check-equal? (stream-get s0) 0)

(define s1 (stream-next s0))
(check-equal? (stream-get s1) 2)

(define s2 (stream-next s1))
(check-equal? (stream-get s2) 4)
```

Solution

```racket
(define (even-naturals)
  (stream-map
   (curry * 2)
   (naturals)))
```
Merge two streams
Zip two streams

- Given a stream \( s_1 \) defined as
  
  \[ e_1, e_2, e_3, e_4, \ldots \]

- and a stream \( s_2 \) defined as
  
  \[ f_1, f_2, f_3, f_4, \ldots \]

- the stream \((\text{stream-zip } s_1 \ s_2)\) returns
  
  \[ (\text{cons } e_1 \ f_1), (\text{cons } e_2 \ f_2), (\text{cons } e_3 \ f_3), (\text{cons } e_4 \ f_4), \ldots \]
Zip for streams

Spec

```racket
#lang racket
(require rackunit)
(define s0
  (stream-zip (naturals) (even-naturals)))
(check-equal? (stream-get s0) (cons 0 0))
(define s1 (stream-next s0))
(check-equal? (stream-get s1) (cons 1 2))
(define s2 (stream-next s1))
(check-equal? (stream-get s2) (cons 2 4))
```
Zip for streams

Spec

```racket
#lang racket
(require rackunit)
(define s0
  (stream-zip (naturals) (even-naturals)))
(check-equal? (stream-get s0) (cons 0 0))
(define s1 (stream-next s0))
(check-equal? (stream-get s1) (cons 1 2))
(define s2 (stream-next s1))
(check-equal? (stream-get s2) (cons 2 4))
```

Solution

```racket
(define (stream-zip s1 s2)
  (define (stream-zip-iter s1 s2)
    (cons
      (cons (stream-get s1)
            (stream-get s2))
    (thunk
      (stream-zip-iter
       (stream-next s1)
       (stream-next s2)))))
  (stream-zip-iter s1 s2))
```

Zip for streams
Exercises on streams
Zip two streams

- Given a stream $s_1$ defined as
  
  $e_1 \ e_2 \ e_3 \ e_4 \ldots$

- and a stream $s_2$ defined as
  
  $f_1 \ f_2 \ f_3 \ f_4 \ldots$

- The stream $(\text{stream-zip } s_1 \ s_2)$ returns
  
  $(\text{cons } e_1 \ f_1) \ (\text{cons } e_2 \ f_2) \ (\text{cons } e_3 \ f_3) \ (\text{cons } e_4 \ f_4) \ldots$
Enumerate a stream

- Build a stream from a given stream $s$ defined as
  
  $e_0 \ e_1 \ e_2 \ e_3 \ e_4 \ e_5 \ ...$

- The stream $(\text{stream-enum } s)$ returns
  
  $(\text{cons } 0 \ e_0) \ (\text{cons } 1 \ e_1) \ (\text{cons } 2 \ e_2) \ (\text{cons } 3 \ e_3) \ (\text{cons } 4 \ e_4) \ (\text{cons } 5 \ e_5) \ ...$
Enumerate a stream

Spec

```racket
#lang racket
(require rackunit)

(define s0 (stream-enum (even-naturals)))
(check-equal? (stream-get s0) (cons 0 0))

(define s1 (stream-next s0))
(check-equal? (stream-get s1) (cons 1 2))

(define s2 (stream-next s1))
(check-equal? (stream-get s2) (cons 2 4))
```
Enumerate a stream

Spec

```racket
#lang racket
(require rackunit)

(define s0 (stream-enum (even-naturals)))
(check-equal? (stream-get s0) (cons 0 0))

(define s1 (stream-next s0))
(check-equal? (stream-get s1) (cons 1 2))

(define s2 (stream-next s1))
(check-equal? (stream-get s2) (cons 2 4))
```

Solution

```racket
(define (stream-enum s)
  (stream-zip (naturals) s))
```
Filter

How would a filter work with streams?
Filter

Spec

```racket
#lang racket
(define s0
  (stream-filter (curry \(\leq\) 10)
    (naturals)))
(check-equal? (stream-get s0) 10)

(define s1 (stream-next s0))
(check-equal? (stream-get s1) 11)

(define s2 (stream-next s1))
(check-equal? (stream-get s2) 12)
```
Converting filter to stream-filter

; List version
------------------------------------------------------------------
1 (define (filter to-keep? l)
2   (cond
3     [(empty? l) l]
4     [(to-keep? (first l))
5       (cons (first l)
6         (filter to-keep? (rest l)))]
8     [else (filter to-keep? (rest l))]))

; Stream-version
------------------------------------------------------------------
1 (define (stream-filter to-keep? s)
2   (cond
3     ; ← no base case; streams are infinite
4     [(to-keep? (stream-get s)) ; ← first becomes stream-get
5       (cons (stream-get s)
6         ; Second element is always a thunk
7         (thunk (stream-filter to-keep? (stream-next s)))]
8     [else (stream-filter to-keep? (stream-next s))]); rest becomes stream-next
Drop every other element

Given a stream defined below, drop every other element from the stream. That is, given a stream $s$ defined as...

$$e_0 \ e_1 \ e_2 \ e_3 \ e_4 \ ...$$

stream (stream-drop-1 s) returns

$$e_0 \ e_2 \ e_4 \ ...$$
Drop every other element...

Spec

```racket
#lang racket
(require rackunit)

(define s0 (stream-drop-1 (naturals)))
(check-equal? (stream-get s0) 0)

(define s1 (stream-next s0))
(check-equal? (stream-get s1) 2)

(define s2 (stream-next s1))
(check-equal? (stream-get s2) 4)
```
Drop every other element…

Spec

```racket
#lang racket
(require rackunit)

(define s0 (stream-drop-1 (naturals)))
(check-equal? (stream-get s0) 0)

(define s1 (stream-next s0))
(check-equal? (stream-get s1) 2)

(define s2 (stream-next s1))
(check-equal? (stream-get s2) 4)
```

Solution

```racket
(define (stream-drop-1 s)
  ; for each e yield (i, e)
  (define enum-s (stream-enum s))
  ; given (i, e) only keep (even? i)
  (define even-s
    (stream-filter
      (lambda (x) (even? (car x)))
      (compose even? car)
      enum-s))
  ; convert (i, e) back to e
  (stream-map cdr even-s))
```
More exercises

- \((\text{stream-ref } s \ n)\) returns the element in the \(n\)-th position of stream \(s\)
- \((\text{stream-interleave } s_1 \ s_2)\) interleave each element of stream \(s_1\) with each element of \(s_2\)
- \((\text{stream-merge } f \ s_1 \ s_2)\) for each \(i\)-th element of stream \(s_1\) (say \(e_1\)) and \(i\)-th element of stream \(s_2\) (say \(e_2\)) return \((f \ e_1 \ e_2)\)
- \((\text{stream-drop } n \ s)\) ignore the first \(n\) elements from stream \(s\)
- \((\text{stream-take } n \ s)\) returns the first \(n\) elements of stream \(s\) in a list in appearance order
Evaluating expressions
Evaluating expressions

Our goal is to implement an evaluation function that takes an expression and yields a value.

\[
\text{expression} = \text{value} | \text{variable} | \text{function-call} \\
\text{value} = \text{number} \\
\text{function-call} = ( \text{expression}+ )
\]
How do we evaluate an expression

What is an expression?

expression = value | variable | function-call

How do we evaluate a value?
How do we evaluate an expression

What is an expression?

\[
\text{expression} = \text{value} \mid \text{variable} \mid \text{function-call}
\]

- How do we evaluate a value? **The evaluation of a value \( v \) is \( v \) itself.**

  \[
  \text{(check-equal? 10 (eval-exp (r:number 10)))}
  \]

- How do we evaluate a function call?
How do we evaluate an expression

What is an expression?

\[
\text{expression} = \text{value} \mid \text{variable} \mid \text{function-call}
\]

How do we evaluate a value? The evaluation of a value \( v \) is \( v \) itself.

\[
(\text{check-equal? } 10 \ (\text{eval-exp (r:number 10)}))
\]

How do we evaluate a function call? The evaluation of a function call evaluates each expression from left to right and then it applies the function to the arguments.
Example

How do we evaluate a function call? The evaluation of a function call evaluates each expression from left to right and then it applies the function to the arguments.

```
(eval-exp
 '(-
   (+ 3 2)
   (* 5 2))))
```

1. evaluate `-`
2. evaluate `(+ 3 2)`
3. evaluate `(* 5 2)`
Example

How do we evaluate a function call? The evaluation of a function call evaluates each expression from left to right and then it applies the function to the arguments.

\[
\text{eval-exp}\left(\text{\textbackslash -}\left(\text{eval-exp}\left(\text{\textbackslash +}\ 3\ 2\right)\left(\text{\textbackslash \ast}\ 5\ 2\right)\right)\right)\right)
\]

\[
= \left(\text{eval-exp}\ \text{\textbackslash -}\right)\left(\text{eval-exp}\ \text{\textbackslash +}\ 3\ 2\right)\left(\text{eval-exp}\ \text{\textbackslash \ast}\ 5\ 2\right)\right)
\]

\[
1. \ \text{evaluate }\text{\textbackslash -}\\
2. \ \text{evaluate }\text{\textbackslash +}, \ \text{evaluate }3, \ \text{evaluate }2
\]

\[
2. \ \text{evaluate }\text{\textbackslash \ast}, \ \text{evaluate }5, \ \text{evaluate }2
\]
Example

How do we evaluate a function call? **The evaluation of a function call evaluates each expression from left to right and then it applies the function to the arguments.**

```
(eval-exp
  '(-
    (+ 3 2)
    (* 5 2)))
```

= ((eval-exp '->
    (eval-exp '(+ 3 2))
    (eval-exp '(* 5 2))))

= ((eval-exp '->
    (eval-exp '+ 3 2)
    (eval-exp '* 5 2)))

①
← evaluate '->
← evaluate '(+ 3 2)
← evaluate '(* 5 2)

②
← evaluate '+, evaluate 3, evaluate 2
← evaluate '*', evaluate 5, evaluate 2

③
← numbers are values, so just return those
← numbers are values, so just return those
How do we evaluate arithmetic operators?

```
= ((eval-exp ' - )
  ((eval-exp ' + ) 3 2)
  ((eval-exp ' * ) 5 2))
```
How do we evaluate arithmetic operators?

\[
= ((\text{eval-exp } \cdot\cdot\cdot) \\
  \quad ((\text{eval-exp } \cdot\cdot\cdot) 3 2) \\
  \quad ((\text{eval-exp } \cdot\cdot\cdot) 5 2))
\]

\[
= (- \\
  \quad (+ 3 2) \\
  \quad (* 5 2))
\]

← Evaluate \(-\) as function -
← Evaluate \(\cdot\cdot\cdot\) as function +
← Evaluate \(*\) as function *
Evaluation of arithmetic expressions

1. When evaluating a number, just return that number
2. When evaluating an arithmetic symbol, return the respective arithmetic function
3. When evaluating a function call evaluate each expression and apply the first expression to remaining ones

Essentially evaluating an expression translates our AST nodes as a Racket expression.
Implementing eval-exp...
Specifying `eval-exp`

- We are use the AST we defined in Lesson 5, not datums.
- Assume function calls are binary.

```scheme
(check-equal? (r:eval-exp (r:number 5)) 5)
(check-equal? (r:eval-exp (r:number 10)) 10)
(check-equal? (r:eval-exp (r:variable? '+)) +)
(check-equal? (r:eval-exp (r:apply
  (r:variable '+)
  (list (r:number 10) (r:number 5)))) 15)
```
We are using the AST we defined in Lesson 5, not datums. Assume function calls are binary.

```
(define (r:eval-exp exp)
  (cond
   ; 1. When evaluating a number, just return that number
   [(r:number? exp) (r:number-value exp)]
   ; 2. When evaluating an arithmetic symbol, return the respective arithmetic function
   [(r:variable? exp) (r:eval-builtin (r:variable-name exp))]
   ; 3. When evaluating a function call evaluate each expression and apply the first expression to remaining ones
   [(r:apply? exp)
    ((r:eval-exp (r:apply-func exp))
     (r:eval-exp (first (r:apply-args exp)))
     (r:eval-exp (second (r:apply-args exp))))]
   [else (error "Unknown expression:" exp)])
```

Implementing `r:eval-builtin`

Spec

```scheme
(check-equal? (r:eval-builtin '+) +)
(check-equal? (r:eval-builtin '-' -)
(check-equal? (r:eval-builtin '/' )/)
(check-equal? (r:eval-builtin '*') *)
(check-equal? (r:eval-builtin 'foo') #f)
```
Implementing **r:eval-builtin**

**Spec**

```
(check-equal? (r:eval-builtin '+) +)
(check-equal? (r:eval-builtin '-') -)
(check-equal? (r:eval-builtin '/) /)
(check-equal? (r:eval-builtin '*') *)
(check-equal? (r:eval-builtin 'foo') #f)
```

**Solution**

```
(define (r:eval-builtin sym)
  (cond [(equal? sym '+) +]
        [(equal? sym '*') *]
        [(equal? sym '-') -]
        [(equal? sym '/') /]
        [else #f]))
```
Handling functions with an arbitrary number of parameters

(required for Homework 3)
Function **apply**

Function `(apply f args)` applies function `f` to the list of arguments `args`.

Examples

```
(check-equal? (apply + (list 1 2 3 4)) 10)
```

Example: implement `(sum l)` that takes returns the summation of all members in `l` using `apply`.

Spec

```
(check-equal? (sum (list)) 0)
(check-equal? (sum (list 1 2 3 4)) 10)
```
Function **apply**

Function `(apply f args)` applies function `f` to the list of arguments `args`.

**Examples**

```lisp
(check-equal? (apply + (list 1 2 3 4)) 10)
```

**Example:** implement `(sum l)` that takes returns the summation of all members in `l` using `apply`.

**Spec**

```lisp
(check-equal? (sum (list)) 0)
(check-equal? (sum (list 1 2 3 4)) 10)
```

**Solution**

```lisp
(define (sum l) (apply + l))
```
Handling multiple-args without apply

Some multi-arg operations can be implemented without the need of `apply`.

Implement \((\text{sum } 1)\) without using apply.

Spec

```
(check-equal? (sum (list)) 0)
(check-equal? (sum (list 1 2 3 4)) 10)
```
Handling multiple-args without apply

Some multi-arg operations can be implemented without the need of \texttt{apply}.

Implement \( (\text{sum } l) \) without using \texttt{apply}.

Spec

\[
\begin{align*}
(\text{check-equal? } (\text{sum } (\text{list})) 0) \\
(\text{check-equal? } (\text{sum } (\text{list } 1 \ 2 \ 3 \ 4)) 10)
\end{align*}
\]

Solution 1

\[
\begin{align*}
(\text{define } (\text{sum } l) \\
(\text{cond} \\
\quad [(\text{empty? } l) 0] \\
\quad [\text{else } (+ (\text{first } l) (\text{sum } (\text{rest } l)))]])
\end{align*}
\]

Solution 2 (foldl is tail-recursive)
Handling multiple-args without apply

Some multi-arg operations can be implemented without the need of `apply`.

Implement `(sum l)` without using `apply`.

Spec

```
(check-equal? (sum (list)) 0)
(check-equal? (sum (list 1 2 3 4)) 10)
```

Solution 1

```
(define (sum l)
  (cond
   [(empty? l) 0]
   [else (+ (first l) (sum (rest l)))]))
```

Solution 2 (foldl is tail-recursive)

```
(define (sum l) (foldl + 0 l))
```
Implementing functions with multi-args

How could we implement a function with multiple parameters, similar to `+`? **Use the . notation.**

The dot . notation declares that the next variable represents a list of zero or more parameters.

**Examples**

```scheme
(define (map-ex f . args)
  (map f args))

(check-equal? (list 2 3 4) (map-ex (curry + 1) 1 2 3))

(define (sum . l) (foldl + 0 l))
(check-equal? 6 (sum 1 2 3))
```