CS450

Structure of Higher Level Languages

Lecture 16: Evaluating expressions; variable arguments

Tiago Cogumbreiro

Today we will...

1. Exercises on streams
2. Learn the first steps of implementing a language
3. Design an interpreter of arithmetic operations
4. Handling operations with multiple arguments

Exercises on streams

Zip two streams

- Given a stream `s1` defined as

```
e1 e2 e3 e4 ...
```

- and a stream `s2` defined as

```
f1 f2 f3 f4 ...
```

- the stream `(stream-zip s1 s2)` returns

```
(cons e1 f1) (cons e2 f2) (cons e3 f3) (cons e4 f4) ...
```

Enumerate a stream

UMASS BOSTON

Build a stream from a given stream `s` defined as

```
e0  e1  e2  e3  e4  e5 ...
```

the stream `(stream-enum s)` returns

```
(cons 0 e0)  (cons 1 e1)  (cons 2 e2)  (cons 3 e3)  (cons 4 e4)  (cons 5 e5) ...
```

Enumerate a stream

Spec

```
#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

```
#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

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

Filter

How would a filter work with streams?

Filter

Spec

```
#lang racket
(define s0
  (stream-filter (curry ≤ 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 ------------------------------------------------------------
(define (filter to-keep? l)
  (cond
   [(empty? l) l]
   [(to-keep? (first l))
    (cons (first l)
          (filter to-keep? (rest l)))]
   [else (filter to-keep? (rest l))]))
; Stream-version ----------------------------------------------------------
(define (stream-filter to-keep? s)
  (cond
    ; <- no base case; streams are infinite
    [(to-keep? (stream-get s)) ; <- first becomes stream-get
     (cons (stream-get s)
           ; Second element is always a thunk
           (thunk (stream-filter to-keep? (stream-next s))))]
    [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...

```
e0 e1 e2 e3 e4 ...
```

> stream (`stream-drop-1 s`) returns

```
e0 e2 e4 ...
```

Drop every other element...

Spec

```
#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

```
#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

```
(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

- (`stream-ref s n`) returns the element in the `n`-th position of stream `s`
- (`stream-interleave s1 s2`) interleave each element of stream `s1` with each element of `s2`
- (`stream-merge f s1 s2`) for each `i`-th element of stream `s1` (say `e1`) and `i`-th element of stream `s2` (say `e2`) return (`f e1 e2`)
- (`stream-drop n s`) ignore the first `n` elements from stream `s`
- (`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.

```
expression = value | variable | function-call
value = number
function-call = ( expression+ )
```

How do we evaluate an expression

UMASS BOSTON

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?

```
expression = value | variable | function-call
```

How do we evaluate a value? **The evaluation of a value v is v itself.**

```
(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?

```
expression = value | variable | function-call
```

> How do we evaluate a value? **The evaluation of a value v is v itself.**

```
(check-equal? 10 (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 '-
← evaluate '(+ 3 2)
← 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.**

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

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?

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

= (-
    (+ 3 2)
    (* 5 2))
```

```
← Evaluate '- as function -
← Evaluate '+ 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.

```
(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)
```

Implementing eval-exp

> 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`

UMASS BOSTON

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)
```

Implementing `r:eval-builtin`

UMASS BOSTON

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

```
(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)
```

Solution

```
(define (sum l) (apply + l))
```

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)
```

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)

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

```
(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))
```