First-class functions

CS135 Lecture 18

Functions as first-class values

In Racket (at the “Intermediate student with `lambda`” level) functions themselves are “first-class” values.

They can be produced, consumed, and appear in lists like other values, such as numbers, symbols and strings. While functions aren’t usually considered “atoms”, you can think of them that way.

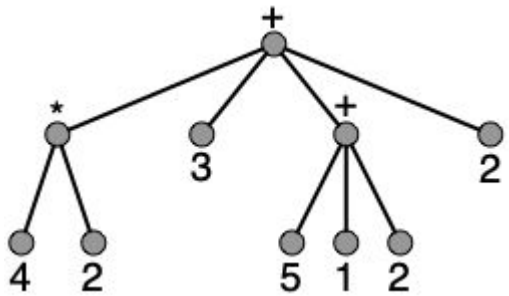
In lecture L17, we considered functions that produce and consume other functions.

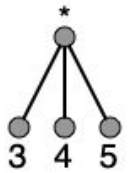
In this lecture, we’ll consider more complex examples, including lists of functions and composition.

L18.0 Lists of functions

Data definition for expression trees (lecture L15)

```
;; An arithmetic expression (AExp) is one of:
;; * Num
;; * (cons (anyof '+ '*) (listof AExp))

(define ae-eg1
  (list '+
        (list '* 4 2)
        3
        (list '+ 5 1 2)
        2))

(define ae-eg2 (list '* 3 4 5))
```





Applying an operator to a list of expression trees (L15)

```
;; apply an operator to a list of arithmetic expressions
;; apply-exp: (anyof '+ '*) (listof AExp) -> Num
(define (apply-exp op el)
  (cond [(empty? el) (cond [(symbol=? '+ op) 0]
                           [(symbol=? '* op) 1])]
        [(symbol=? '+ op) (+ (eval (first el))
                             (apply-exp op (rest el)))]
        [(symbol=? '* op) (* (eval (first el))
                             (apply-exp op (rest el)))]))
```

We changed the name to **`apply-exp`** since intermediate student reserves **`apply`**.

Evaluating an expression tree (from Lecture L15)

```
;; evaluate an arithmetic expression
;; eval: AExp -> Num
(define (eval exp)
  (cond [(number? exp) exp]
        [else (apply-exp (first exp) (rest exp))]))

(check-expect (eval ae-eg1) 21)
(check-expect (eval ae-eg2) 60)
```

A key observation is that **`eval`** can't be recursively applied to an operator followed by a list of **`AExp`**. We need a separate function (**`apply-exp`**) to do that.

Functions in lists

The mapping between symbols (`'+ '*`) and operations (`+ *`) is defined explicitly in the function. If we want to add another operation we would have to re-write it.

Since functions can be stored in list, we can represent the mapping between symbol and operation as an association list.

```
(define operations
  (list (list '+ (list + 0))
        (list '* (list * 1))))
```

The list associates a symbol with a list containing an operation and a base value.

Translating symbols to operations and base values

```
;; translate: Sym -> (list (Num Num -> Num) Num)
(define (translate op)
  (local
    [(define (nothing x y) 0)
     (define operations
       (list (list '+ (list + 0))
             (list '* (list * 1))))
     (define (lookup op lst)
       (cond [(empty? lst) (list nothing 0)]
             [(symbol=? op (first (first lst)))
              (second (first lst))]
             [else (lookup op (rest lst))]))]
    (lookup op operations)))
```

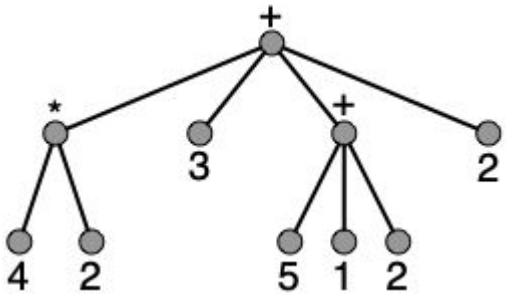
apply-exp with **translate**

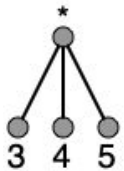
```
;; apply an operator to a list of arithmetic expressions
;; apply-exp: Sym (listof AExp) -> Num
(define (apply-exp op el)
  (local
    [(define operation (translate op))]
    (cond [(empty? el) (second operation)]
          [else ((first operation) ;; function application
                 (eval (first el))
                 (apply-exp op (rest el)))])))

(check-expect (apply-exp '+ (list 1 4 (list '* 3 3) 16)) 30)
```

Alternative approach: Modifying expression trees

We can also modify expression trees to directly store the operations as functions.

```
;; An arithmetic expression (AExp) is one of:
;; * Num
;; * (cons (anyof + *) (listof AExp))
(define ae-eg1
  (list +
        (list * 4 2)
        3
        (list + 5 1 2)
        2))
(define ae-eg2
  (list * 3 4 5))
```





Alternative approach: Modifying expression trees

The `eval` function doesn't change.

```
;; evaluate an arithmetic expression
;; eval: AExp -> Num
(define (eval exp)
  (cond [(number? exp) exp]
        [else (apply-exp (first exp) (rest exp))]))

(check-expect (eval ae-eg1) 21)
(check-expect (eval ae-eg2) 60)
```

Alternative approach: Modifying expression trees

The `apply-exp` function is shorter and simpler.

```
;; apply an operator to a list of arithmetic expressions
;; apply-exp: (Num Num -> Num) (listof AExp) -> Num
(define (apply-exp op args)
  (cond [(empty? args) (op)]
        [else  (op (eval (first args))
                   (apply-exp op (rest args)))]))
```

Depends on the (perhaps unexpected) behaviour that `(+)` is 0 and `(*)` is 1.

Is the contract really correct?

L18.1 Stable sorting

Reminder: Merging two sorted lists (from lecture L10).

```
;; merge: (listof Num) (listof Num) -> (listof Num)
;; Requires: lists must be sorted in ascending order
(define (merge lst1 lst2)
  (cond [(empty? lst1) lst2]
        [(empty? lst2) lst1]
        [(< (first lst1) (first lst2))
         (cons (first lst1) (merge (rest lst1) lst2))]
        [(> (first lst1) (first lst2))
         (cons (first lst2) (merge lst1 (rest lst2)))]
        [else (cons (first lst1)
                    (cons (first lst2)
                          (merge (rest lst1) (rest lst2))))]))
```

Generalizing `merge`

What about if we want to sort strings or numbers in decreasing order?

Without first-class functions we need to rewrite **`merge`** to replace the **<** and **>**, but now we can write merge once and pass pass predicates for the comparisons.

We can also simplify **`merge`** so that we only need to pass one predicate (<):

```
(check-expect
 (merge string<? (list "Alice" "Carol") (list "Bob"))
 (list "Alice" "Bob" "Carol"))

(check-expect (merge > '(6 4 2) '(7 5 3 1))
              '(7 6 5 4 3 2 1))
```

Simplifying **`merge`** to use only <

```
;; merge: (listof Num) (listof Num) -> (listof Num)
;; Requires: lists must be sorted in ascending order
(define (merge lst1 lst2)
  (cond [(empty? lst1) lst2]
        [(empty? lst2) lst1]
        [(< (first lst1) (first lst2))
         (cons (first lst1) (merge (rest lst1) lst2))]
        [else
         (cons (first lst2) (merge lst1 (rest lst2)))]))
```

Notice that the function still works for equal elements, preferring the second list.

Generalizing **merge**

```
(define (merge <? lst1 lst2)
  (cond [(empty? lst1) lst2]
        [(empty? lst2) lst1]
        [(<? (first lst1) (first lst2))
         (cons (first lst1) (merge <? (rest lst1) lst2))]
        [else
         (cons (first lst2) (merge <? lst1 (rest lst2)))]))
(check-expect
 (merge string<? (list "Alice" "Carol") (list "Bob"))
 (list "Alice" "Bob" "Carol"))
(check-expect (merge > '(6 4 2) '(7 5 3 1))
              '(7 6 5 4 3 2 1))
```

Sorting on multiple fields

Here's a list of CS135 students from a different term (not this term).

```
(define cs135
  '(("Banana" "Bob") ("Avocado" "Alice")
    ("Banana" "Carol") ("Aardvark" "Bob")
    ("Avocado" "Carol") ("Aardvark" "Alice")
    ("Cucumber" "Alice") ("Avocado" "Bob")
    ("Cucumber" "Carol") ("Bear" "Bob")))
```

Each list element is a `("`*family name*`" "`*given name*`")` pair. As you can see, while no two people have the same name, nobody has a unique family or given name,

Sorting on multiple fields

We want to sort this list according to English-language conventions: alphabetically, i.e., lexicographically, by family name and then by given name.

```
(define sorted-cs135
  '(("Aardvark" "Alice") ("Aardvark" "Bob")
    ("Avocado" "Alice") ("Avocado" "Bob")
    ("Avocado" "Carol") ("Banana" "Bob")
    ("Banana" "Carol") ("Bear" "Bob")
    ("Cucumber" "Alice") ("Cucumber" "Carol")))
```

For simplicity, we compare names with `string<?`, but be warned that sorting names with `string<?`, i.e., by codepoints, is not universally correct.

Stable sorting

A “stable” sorting algorithm is one that preserves the relative order of elements that are considered equal, i.e. `(<? x y)` and `(<? y x)` are both `false`.

For example, if two people in our list of names have the same family name, a stable sort on the given name guarantees that they appear in the same order in the sorted list.

```
(define given-sorted-cs135
  '(("Avocado" "Alice") ("Aardvark" "Alice")
    ("Cucumber" "Alice") ("Banana" "Bob")
    ("Aardvark" "Bob") ("Avocado" "Bob")
    ("Bear" "Bob") ("Banana" "Carol")
    ("Avocado" "Carol") ("Cucumber" "Carol")))
```

Stable sorting

```
'(("Banana" "Bob") ("Avocado" "Alice")
   ("Banana" "Carol") ("Aardvark" "Bob")
   ("Avocado" "Carol") ("Aardvark" "Alice")
   ("Cucumber" "Alice") ("Avocado" "Bob")
   ("Cucumber" "Carol") ("Bear" "Bob")))

 '(("Avocado" "Alice") ("Aardvark" "Alice")
   ("Cucumber" "Alice") ("Banana" "Bob")
   ("Aardvark" "Bob") ("Avocado" "Bob")
   ("Bear" "Bob") ("Banana" "Carol")
   ("Avocado" "Carol") ("Cucumber" "Carol"))
```

Stable sort on given name.

Stable `mergesort`

For `mergesort` to be stable, it must preserve the relative order of equal elements when we split the lists:

1. Split the list into two lists of nearly equal length: first half and second half.

   `(list 8 5 3 9 1 6 2 5 0 7)` ⇒ `(list 8 5 3 9 1) (list 6 2 5 0 7)`

2. Stably sort each of the two shorter lists.

   `(list 1 3 5 8 9) (list 0 2 5 6 7)`

3. Merge the sorted lists. When elements are equal choose from the first half.

   `(list 0 1 2 3 5 5 6 7 8 9)`

Splitting a list into a first half and second half (from L10)

```
(define (first-n n lst)
  (cond [(or (empty? lst) (zero? n)) empty]
        [else (cons (first lst)
                    (first-n (sub1 n) (rest lst)))]))
(check-expect (first-n 3 '(1 2 3 4 5 6 7))'(1 2 3))

(define (rest-n n lst)
  (cond [(or (empty? lst) (zero? n)) lst]
        [else (rest-n (sub1 n) (rest lst))]))
(check-expect (rest-n 3 '(1 2 3 4 5 6 7)) '(4 5 6 7))
```

Splitting a list into a first half and second half

```
;; split a list into nearly equal halves
;; split: (listof Any) -> (list (listof Any) (listof Any))
(define (split lst)
  (local
    [(define n (quotient (length lst) 2))]
    (list (first-n n lst) (rest-n n lst))))

(check-expect
 (split '(1 2 3 4 5 6 7)) '((1 2 3) (4 5 6 7)))
(check-expect
 (split '(a b c d e f)) '((a b c) (d e f)))
```

A function that makes stable sorting functions

```
;; produce a sort function from a predicate
;; make-sort: (X X -> Bool) -> ((listof X) -> (listof X))
(define (make-sort <?)
  (local
    [(define (mergesort lst)
       (cond [(or (empty? lst) (empty? (rest lst))) lst]
             [else
              (local
                [(define s (split lst))]
                (merge <?
                       (mergesort (second s))
                       (mergesort (first s))))]))]
    mergesort))
```

Stable sorting by given names

```
(define stable-sort-given-names
  (make-sort
   (lambda (x y) (string<? (second x) (second y)))))
(define given-sorted-cs135
  '(("Avocado" "Alice") ("Aardvark" "Alice")
    ("Cucumber" "Alice") ("Banana" "Bob")
    ("Aardvark" "Bob") ("Avocado" "Bob")
    ("Bear" "Bob") ("Banana" "Carol")
    ("Avocado" "Carol") ("Cucumber" "Carol")))

(check-expect
 (stable-sort-given-names cs135) given-sorted-cs135)
```

Composing sorting functions

Because **`make-sort`** makes stable sorting functions we can compose them to sort on multiple fields.

```
(define (sort-names lst)
 ((make-sort
   (lambda (x y) (string<? (first x) (first y))))
  ((make-sort
    (lambda (x y) (string<? (second x) (second y))))
   lst)))
(check-expect (sort-names cs135) sorted-cs135)
```

The sorter for given names is applied before the sorter for family names. We say that we are sorting with family names as the “primary key” and given names as the “secondary key”.

Sorting should be stable

Stable sorting ensures that the relative order of equal elements remains unchanged after a sort. This property is essential in multi-step sorting processes.

Spreadsheets, (e.g., Excel, Google Sheets) sort stably.

Imagine you have a spreadsheet with employee information. One column contains employee names and another column contains their departments. Initially, the data is sorted by name. If you then sort by department using a stable sorting algorithm, employees within the same department will remain in alphabetical order by name. This means you maintain employee name as a secondary sort key.

When writing a sorting function, care must be taken to ensure it is stable.

Racket's built-in `sort` is a stable mergesort

Racket has a built-in `sort` that you can now use.

```
sort: (listof X) (X X -> Bool) -> (listof X)
```

The list is the first argument and the predicate is the second argument.

```
(check-expect
 (sort '(8 4 3 9 1 6 2 5 0 7) >)
 '(9 8 7 6 5 4 3 2 1 0))
(check-expect
 (sort '("Bob" "Carol" "Alice") string<?)
 '("Alice" "Bob" "Carol"))
```

Sorting in Python is also stable, but the default sort in C or C++ may not be.

L18 Summary

L18: You should know

- How to work with lists of functions.
- Alternative forms for expression trees.
- The importance of stable sorting.
- How to implement stable sorting in mergesort.
- How to use Racket’s built-in sort function.

L18: Allowed constructs

Newly allowed constructs:

```
sort
```

Previously allowed constructs:

```
( ) [ ] + - * / = < > <= >= ; '
abs acos add1 and append asin atan boolean? char? char=?
char<? check-expect check-within cond cons cons? cos define
e else empty empty? even? exp expt false filter first
inexact? integer? lambda length list list? local log max min
not number? odd? or pi quotient rational? remainder rest
reverse second sin sqr sqrt string? string=? string<?
string->list list->string sub1 symbol? symbol=? tan third
true zero?
listof Any anyof Atom Bool Char Int Nat Num Rat Str Sym
```