Functional Programming

Lecture 2: Lambda abstraction

Viliam Lisý

Artificial Intelligence Center
Department of Computer Science
FEE, Czech Technical University in Prague

viliam.lisy@fel.cvut.cz

Last lecture

- What is (pure) functional programming
- Why do we care about it?
- Recursion is the main tool
- Scheme

  s-expression, quote, identifiers, define, if, cond

Avoiding infinite recursion

1. First expression of the function is a `cond`
2. The first test is a termination condition
3. The "then" of the first test is not recursive
4. The `cond` pairs are in increasing order of the amount of work required
5. The last `cond` pair is a recursive call
6. Each recursive call brings computations closer to the termination condition

Recursion

Tail recursion

- Last thing a function does is the recursive call

Analytic / synthetic

- Return value from termination condition / composed

Tree / linear recursion

- Function is called recursively multiple times (qsort)

Indirect recursion

- Function A calls function B which calls A

Lists

The key data structure of Lisp

S-expressions are just lists

```
'(+ 1 2 3 4 5)
```

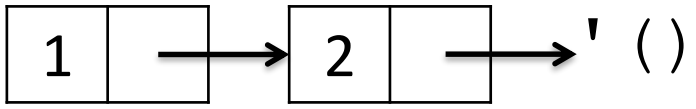
Lists can be created by a function

```
(list item1 item2 … itemN)
```

List are linked lists of pairs with `'()` at the end

sometimes abbreviated by .

```
(cons 1 2), (cons 1 (cons 2 '()))
```



Lists

Pairs forming the lists can be decomposed by

- `car` *[car]* — first element of the pair
- `cdr` *[could-er]* — second element of the pair
- `(caddr x)` shortcut for `(car (cdr (cdr x )))`

Empty list is a null pointer

- `null?` tests whether the argument is the empty list

Append

Append one list to another

```
(define (append2 a b)
  (cond
    ((null? a) b)
    (#t (cons (car a)
              (append2 (cdr a) b)))
  )
)
```

Evaluation strategy

Defines the order of evaluating the expressions

- influences program termination, not the result

Evaluation of scheme is eager (or strict)

- left to right
- evaluate all arguments before executing a function

Evaluation of some special forms is lazy

- if, and, or, **lambda**

Equality

Function = is only for numbers

Equivalence of the objects `eqv?`

```
(eqv? 1 1), (eqv? 'a 'a) ===> #t
(eqv? (list 'a) (list 'a))  ===> #f
```

More restrictive version is `eq?`

- Typically the same pointer

Recursive version of `eqv?` on lists is

```
(equal? (list 'a) (list 'a)) ===>#t
```

Debugging Basics

Tracing function calls and returns

```
#lang scheme
(require racket/trace)
(trace append2)
(untrace append2)
```

Helper print-outs

```
(begin (display x) <do-work>)
```

Lambda abstraction

A construction for creating nameless procedures

```
(lambda (arg1 … argN) <expr>)
```

Define for functions is an abbreviation

```
(define (<var> <formals>) <body>)
```

Is the same as

```
(define <var>
   (lambda (<formals>) <body>))
```

Filter

Returns the elements form a list that satisfy a given predicate

```
(define (my-filter pred list)
  (cond ((null? list) list)
        ((pred (car list))
           (cons
             (car list)
             (my-filter pred (cdr list))
         ) )
        (#t (my-filter pred (cdr list)))
   )
)
```

Scheme syntax

E in Expressions

I in Identifiers (variables)

K in Constants

E := K | I | (E_0 E*) | (lambda (I*) E2) | (define I E')

Let

Motivation

reuse of computation/result is often required

e.g., `minimum, roots` from the labs

```
(let ((<var1> <exp1>)
      (<var2> <exp2>))
            <body-using-var1-var2>)
```

Implementing let

```
(let ((x <exp1>)
      (y <exp2>)) <body>)
```

Can be implemented as

```
((lambda (x y) <body>) <exp1> <exp2>)
```

Let as lambda

```
(define (my-min3 ls)
  (if (null? (cdr ls))
      (car ls)
      ((lambda (m)
         (if (< (car ls) m)
             (car ls)
             m
         )
       ) (my-min3 (cdr ls)))
) )   )
```

Merge sort - split

```
(define (split ls)
  (cond
    ((null? ls) (cons '() '()))
    ((null? (cdr ls)) (cons ls '()))
    (#t (let ((p (split (cddr ls))))
          (cons (cons (car ls) (car p))
                (cons (cadr ls) (cdr p))))))
  )
)
```

Merge sort - merge

```
(define (merge as bs)
  (cond
    ((null? as) bs)
    ((null? bs) as)
    ((<= (car as) (car bs))
     (cons (car as) (merge (cdr as) bs)))
    (#t (merge bs as))
  )
)
```

Merge sort

```
(define (merge-sort ls)
  (cond
    ((null? ls) ls)
    ((null? (cdr ls)) ls)
    (#t (let  ((p (split ls))
               (sas (merge-sort (car p)))
               (sbs (merge-sort (cdr p)))
              )
              (merge sas sbs)
))))
```

Implementing let

```
(let ((x <exp1>)
      (y <exp2>)) <body>)
```

Can be implemented as

```
((lambda (x y) <body>) <exp1> <exp2>)
```

Let*

We might want to use the earlier definitions in the following.

```
(let ((x <exp>))
  (let ((y <exp-with-x>)) <body-x-y>)
```

Equivalent to

```
(let* ((x <exp>)
       (y <exp-with-x>)) <body-x-y>)
```

Merge sort

```
(define (merge-sort ls)
  (cond
    ((null? ls) ls)
    ((null? (cdr ls)) ls)
    (#t (let* ((p (split ls))
               (sas (merge-sort (car p)))
               (sbs (merge-sort (cdr p)))
              )
              (merge sas sbs)
))))
```

Scheme home assignments

Three connected assignments

- Robot simulation
- Population evaluation
- Code synthesis

Why this assignment?

Work on your own

Submit by midnight of the day of your lab

- https://cw.felk.cvut.cz/brute/ (in 2 weeks)