David R. Cheriton School of Computer Science

CS 115: Introduction to Computer Science

Presented By Ahmed Ibrahim

Fall 2015

Module 10 - Agenda

- We will cover material on **functional abstraction** in a somewhat different order than the text.
- CS 115 will cover built-in functions that consume functions as **inputs**.
- We move to the **Intermediate teaching language** with the introduction of **local** definitions and **abstract list functions**.

What is abstraction?

- Abstraction consists of
  - finding similarities or common aspects, and
  - forgetting unimportant differences.

- For a **single function**, differences in parameter values are forgotten, and the similarity is captured in the function body.
- For **multiple functions**, similarity is captured in templates.
- For **multiple functions**, further abstraction is possible (using abstract functions).

Examples


```
(define (eat-apples alist)
 (cond
  [(empty? alist) empty]
  [(cons? alist)
   (cond
    [(not (symbol=? (first alist)
'apple))
     (cons (first alist) (eat-apples
(rest alist)))]
    [else (eat-apples (rest
alist))])]))
```

```
(define (select-even alist)
 (cond
  [(empty? alist) empty]
  [(cons? alist)
   (cond
    [(even? (first alist))

     (cons (first alist) (select-even
(rest alist)))]
    [else (select-even (rest
alist))])]))
```

Abstracting from these examples

- **Functional abstraction** is the process of creating abstract functions.
- Similarity: general structure (removing certain items)
- Difference: **predicate** used to decide what to remove

Similarity: general structure

```
(define ( ..... alist)
 (cond
  [(empty? alist) empty]
  [(cons? alist)
   (cond
    [(..... ? (first alist))
     (cons (first alist) (.... (rest alist)))]
    [else (..... (rest alist))])]))
```

Difference: predicate?

Abstract List Function

- Goal: form an abstract list **function** that consumes the predicate (function).
- Functions (such as predicates) are *first-class* values in the **Intermediate Student Language**.

A first class function is the one which could be passed as an argument

- Example: (filter even? alist)
- Filter is a built-in function in Racket.

The abstract list function filter

Abstract function

```
(define (filter pred alist)
  (cond
    [(empty? alist) empty]
    [(cons? alist)
     (cond
       [(pred (first alist))
            (cons (first alist) (filter pred (rest alist)))]
       [else (filter pred (rest alist))])]))
```

first-class

```
(define (eat-apples alist)
  (cond
    [(empty? alist) empty]
    [(cons? alist)
     (cond
       [(not (symbol=? (first alist)
'apple))
        (cons (first alist) (eat-apples
(rest alist)))]
       [else (eat-apples (rest
alist))])]))
```


Built-in function filter

(filter even? (list 6 7 8))
⇒ (cons 6 (filter even? (list 7 8)))
⇒ (cons 6 (filter even? (list 8)))
⇒ (cons 6 (cons 8 (filter even? empty)))
⇒ (cons 6 (cons 8 empty))

- The **abstract list function** filter performs the general operation of selecting items from lists.
- Racket provides such functions to apply common patterns.

Using filter


```
(define (select-even alist) (filter even? alist))

(define (symbol-not-apple? item) (not (symbol=? Item 'apple)) )
(define (eat-apples alist) (filter symbol-not-apple? alist))
```

- The built-in function filter consumes a predicate specifying which elements of the list are to be kept.
- The predicate must be a **one-parameter** function producing a boolean, where the type of the parameter is same as the type of the elements of the list.

REMEMBER

Advantages of functional abstraction

1. It reduces code size.
2. It avoids cut-and-paste.
3. Bugs can be fixed in one place instead of many.
4. Improving one functional abstraction improves many applications.

Abstracting from examples

```
(define (negate-list numlist)
(cond
[(empty? numlist) empty]
[else (cons
(– (first numlist))
(negate-list (rest numlist)))]
```

```
(define (compute-grades rlist)
(cond
[(empty? rlist) empty]
[else (cons
(final-grade (first rlist))
(compute-grades (rest rlist)))]))
```

```
(define ( f alist)
    (cond
        [(empty? alist) empty]
        [else (cons ( ??? (first alist)) ( f  (rest alist)))]))
```


The abstract list function map

```
(define ( ....  alist)
    (cond
        [(empty? alist) empty]
        [else (cons ( ??? (first alist))
                    ( ...  (rest alist)))]))
```

We are going to take what is applied to the first item of a list and make that a parameter

- Goal: form an **abstract list function** that applies a *function* to the elements of the list from the first element to the last.

Example: (map sqr alist)

- map is a built-in function in Racket.

The abstract list function map (cont.)

The abstract list function map performs the operation of **transforming** a **list** element-by-element into another list of the **same length**.

(map f (list x_1 x_2 ... x_n) equivalent to (list (f x_1) (f x_2) ... (f x_n))

Short definitions using map:

```
(define (negate-list numlist) (map – numlist))
(define (compute-grades rlist) (map final-grade rlist))
```

The function consumed by map must be a **one-parameter** function where the type of the parameter is the same as the type of the elements of the list.

REMEMBER

The abstract list function map

```
(define (map f alist)
    (cond
        [(empty? alist) empty]
        [else (cons (f (first alist)) (map f (rest alist)))]))
```

For this and other built-in abstract list functions, see the table on page 313 of the text (Figure 57 in Section 21.2).

Tracing map

(map sqr (list 3 6 5))
⇒ (cons (sqr 3) (map sqr (list 6 5))
⇒ (cons 9 (map sqr (list 6 5)))
⇒ (cons 9 (cons (sqr 6) (map sqr (list 5))))
⇒ (cons 9 (cons 36 (map sqr (list 5))))
⇒ (cons 9 (cons 36 (cons (sqr 5) (map sqr empty))))
⇒ (cons 9 (cons 36 (cons 25 (map sqr empty))))
⇒ (cons 9 (cons 36 (cons 25 empty)))

Additional Exercise

Write a function ***double*** that produces a list which is a copy of a given list alist except that all elements of alist have been doubled.