



Functional Abstraction and Polymorphism

Guest: Dr. John Greiner
Department of Computer Science
Rice University



Course Overview

- Data-driven program design 2-17
- Abstraction and good design 19-23
- Algorithms
25-28
- Accumulators 30-32
- Side effects 34-37
- Mutable structures, and objects
39-43
- Design Recipes
 - Introduced and used throughout the course



Abstracting Designs

- “The elimination of repetitions is the most important step in the (program) editing process” – Textbook
- Lots of “cut and paste” is generally a bad thing.
 - Anyone have this experience?
- **Abstractions** help us avoid this problem.



The Need for Abstractions

```
;; contains-doll? : los -> boolean
;; to determine whether alos contains
;; the symbol 'doll
(define (contains-doll? alos)
  (cond [(empty? alos) false]
    [else (or (symbol=? (first alos)
                        'doll)
                (contains-doll?
                 (rest alos))))]))
```



The Need for Abstractions

```
;; contains-car? : los -> boolean
;; to determine whether alos contains
;; the symbol 'car
(define (contains-car? alos)
  (cond [(empty? alos) false]
    [else (or (symbol=? (first alos)
                        'car)
                (contains-car?
                 (rest alos))))]))
```



Creating Abstractions

How can we write one function that replaces

- contains-doll?
- contains-car?
- contains-pizza?
- contains-comp210?
- ...



Creating Abstractions

;; contains? : **symbol**, los -> boolean

;; to determine whether alos contains

;; the symbol **s**

(define (contains? **s** alos)

(cond [(empty? alos) false]

[else (**or** (symbol=? (first alos)

s)

 (contains? **s**

 (rest alos))))))



Using Abstractions

- How do we use contains?

(contains? 'doll (list ...))

(contains? 'car (list ...))

- How can we better define contains-doll?, contains-car?

(**define** (contains-doll? alos)

(contains? 'doll alos))

(**define** (contains-car? alos)

(contains? 'car alos))

- This idea is called **reuse**. Let's run with it!



It will get real cool up here ...

```
;; below : lon number -> lon
;; to construct a list of those numbers
;; in alon that are below t
(define (below alon t)
  (cond [(empty? alon) empty]
        [else
         (cond [(< (first alon) t)
                  (cons (first alon)
                        (below (rest alon) t))]
               [else (below (rest alon) t))]))))
```



It will get real cool up here ...

```
;; above : lon number -> lon
;; to construct a list of those numbers
;; in alon that are above t
(define (above alon t)
  (cond [(empty? alon) empty]
        [else
         (cond [(> (first alon) t)
                  (cons (first alon)
                        (above (rest alon) t))]
               [else (above (rest alon) t))]))))
```



Creating Abstractions

How can we write one function that replaces

- below
- above
- equal
- same-sign-as
- ...



Creating Abstractions

```
;; filter1 : comparison lon number -> lon
;; to construct a list of those numbers n
;; in alon such that (test t n) is true
(define (filter1 test alon t)
  (cond [(empty? alon) empty]
        [else
         (cond [(test (first alon) t)
                (cons (first alon)
                      (filter1 test (rest alon) t))]
               [else (filter1 test (rest alon) t))]))))
```



Using Abstractions

- The magic moment: how do we use filter1 ?

```
(filter1 < (list ...) 17))
```

```
(filter1 > (list ...) 17))
```

- How can we better define above, below ?

```
(define (below alon t) (filter1 < alon t))
```

```
(define (above alon t) (filter1 > alon t))
```

- Both functions will work just as before!



Similarity in Types

Repetition also happens in type definitions.

A lon is one of:

- empty
- (cons n alon),
where n is a number and alon is a lon.

A los is one of:

- empty
- (cons s alos),
where s is a symbol and alos is a los.



Abstracting Types

Textbook uses (listOf X).

A [X] is one of:

- empty
- (cons x alox),

where x is an X and alox is a [X].

A variable at the type level.

Often called **polymorphism** or **generics**.



Abstracting Types

Type	Example(s)
[number]	(list 1 2 3)
[symbol]	(list 'a 'b 'pizza)
[any]	(list 1 2 3) (list 'a 'b 'pizza) empty (list 1 'a +)

Important! [X] is NOT [any].



Revisiting filter1

What is a more precise description of `test`'s type?

```
:: filter1 : comparison lon number -> lon
;; to construct a list of those numbers n
;; in alon such that (test t n) is true
```

```
:: filter1 : (number number -> boolean) [number] number
;;         -> [number]
```



Revisiting filter1

How would we rewrite this to use on symbols instead?

What would the type be?

```
:: filter1 : comparison lon number -> lon
```

```
:: to construct a list of those numbers n
```

```
:: in alon such that (test t n) is true
```

```
(define (filter1 test alon t)
  (cond [(empty? alon) empty]
        [else
         (cond [(test (first alon) t)
                (cons (first alon)
                      (filter1 test (rest alon) t))]
               [else (filter1 test (rest alon) t))]))))
```



Revisiting filter1

```
:: filter1 : (number number -> boolean) [number] number  
::         -> [number]
```

```
:: filter1 : (symbol symbol -> boolean) [symbol] symbol  
::         -> [symbol]
```

And similarly for types other number and symbol.
So, what is filter1's **most general type**?

```
:: filter1 : (X X -> boolean) [X] X -> [X]
```

```
:: filter1 : (X Y -> boolean) [X] Y -> [X]
```



Final thoughts

- Function abstraction adds **expressivity**.
- Type abstraction (polymorphism) does the same.
- Together, both work *really* well.

- Programming will continue to get better as we add abstraction mechanisms to our language(s).



For Next Class

- Homework will be put online today
- Reading:
 - Ch 20: Functions as values
- Quiz on reading