



Effects and Structures



Today's Goals

- Effects and effectful functions
- More about **and**
- Structures
 - When do we need them
 - Making new types with **define-struct**
 - Building and using structures
 - Designing programs that use structures



Effects

- Anything beyond normal math functions
- Examples:
 - Reading an input from a keyboard, mouse, audio mic
 - Drawing on the screen, playing music
 - Stopping with an error
 - Buying the groceries for the week, flying to denver
- Effectful function:
 - A convenient way to do effects
 - Convenient trick: If function completes, return **true**
 - Otherwise, program stops with an error



More about **and**

- It can take more than two arguments
`(and true true false true (= 1 (/ 2 0)) ...)`
`= false`
- This behavior is called “lazy” / “non-strict”
 - Starts evaluating left argument, moves right
 - If anything evaluates to false, it returns false
 - If all arguments evaluate to true, returns true
- Aside: What would **or** do?
- **and** is for sequencing effects
 - `(and (draw-house) (draw-road) (draw-yard) ...)`



Structures (records, tuples, ...)

- Structures are used to model compound or composite data
- Examples
 - “... A university course has an associated department and course numbers, as well as a class size”
 - “... A vehicle has a VIN number, model, year, engine size, and passenger capacity”
 - “... For each house, we will keep a record of room numbers, area, and price”
- How do we keep such data in one place?
 - How do we we construct or build such a value
 - How do we use it or take it apart



Program Design: Data definition

- Problem description
 - “... Each university **course** will have an associated **department** and **course numbers**, as well as a **class size**”
- Data definition

```
;; A course is a structure
;;      (make-course dept num size)
;; where dept is a symbol, and num
;; and size are numbers
(define-struct course (dept num size))
```



Creating and Using Structures

- Syntax for creating a structure:

```
(define this-class (make-course 'COMP 210 50))
```
- A structure is a value
 - It's big. But it's just like 1, true, or 'Rabbit
 - It's big. But it is NOT a reduction, like (+ 1 2)
- Syntax for extracting fields
 - (course-dept this-class),
 - (course-num this-class), ...
- Reduction for field extraction

```
(course-dept this-class)  
= (course-dept (make-course 'COMP 210 50))  
= 'COMP
```



The Design Recipe

How should I go about writing programs?

1. Analyze problem and define data types (and template)
2. State contract and purpose for function
3. Give examples of function use and result
4. Write the function itself following template from 1.
5. Test it, and record actual results of tests

The order of the steps of the recipe is important



Type --> Template --> Code

- We start from the data definition. Example:

```
(define-struct course (dept num size))
```
- Template for any function consuming course

```
;; (define (f c)
;;   ... (course-dept c) ...
;;   ... (course-num c) ...
;;   ... (course-size c) ...)
```
- Example of such a function

```
;; big-class? : course -> bool
;; empty-class? : course -> bool
;; change-dept : course dept -> course
```



Type --> Template --> Code

- Template for function consuming “course”

```
;; (define (f c)
;;   ... (course-dept c) ...
;;   ... (course-num c) ...
;;   ... (course-size c) ...)
```
- Templates will help us write the code

```
(define (big-class? c)
  (< 30 (course-size c)))
```
- Sophisticated types -> sophisticated templates ...
- This will help us write correct, sophisticated code



For Next Class

- Homework will be posted later today
 - We'll email to discussion list
- Reading:
 - Chapter 7 and companion notes
 - More cool type definition tools
- Quiz:
 - Chapter 7