Testing, Contracts, Functions, Constants, and Conditionals
Last Lecture

- What is a syntactically valid program
  - Experience w/ DrScheme’s error messages
- What is a runtime error
  - More experience w/ DrScheme’s messages
- Design recipe (in that order)
  - Contract, Purpose, Examples,
  - Definition,
  - Tests
Today’s Goals

- Why does testing matter?
- Should we make our programs readable for humans?
  - Naming functions
  - Naming constants
- Conditional expressions
  - Special syntax makes life easier
Testing

- Should at least include the examples we wrote ahead of time
- Extremely useful, but NOT definitive
- For large-scale development, is crucial to software reliability

Aside: What other ways are there for checking correctness?

- Program equivalence
- Peer review, clean-room sw dev.
- Random testing, tracing/debugging tools,
- Static analysis, type checking (not so obvious!)
Contractual Obligations

- Should our program check the contract?
  - Advantage: more user friendly, safer for user.
  - Disadvantage: no “freebies”, slower program, more programming work, much bigger code, cannot always be done!

- What should a program do on a failure?
  - Nothing. In particular, NOT crash.
  - raise an (understandable) error
Organization of Programs

Compare the following two programs:

- (define (area-of-ring outer inner)
  (- (* 3.14 (* outer outer))
     (* 3.14 (* inner inner))))

- (define (area-of-ring outer inner)
  (- (area-of-disk outer)
     (area-of-disk inner)))

Tru’s point: Which one is more concise? 😊

Naming functions can make programs shorter, clearer, and easier to change.
“Auxiliary Functions” Can Help

Consider the program to the right

What does it do?

(define (profit price)
  (- (* (+ 120
            (* (/ 15 .10)
                (- 5.00 price)))
      price)
   (+ 180
      (* .04
       (+ 120
          (* (/ 15 .10)
              (- 5.00 price)))))))
“Auxiliary Functions” Can Help

Now consider this program

- What does it do?

```
(define (profit ticket-price)
  (- (revenue ticket-price)
      (cost ticket-price)))

(define (revenue ticket-price)
  (* (attendees ticket-price) ticket-price))

(define (cost ticket-price)
  (+ 180
     (* .04 (attendees ticket-price)))))

(define (attendees ticket-price)
  (+ 120 (* (/ 15 .10)
             (- 5.00 ticket-price))))
```
“Auxiliary Functions” Can Help

;; Good program design
(define (profit ticket-price)
  (- (revenue ticket-price)
      (cost ticket-price)))
(define (revenue ticket-price)
  (* (attendees ticket-price) ticket-price))
(define (cost ticket-price)
  (+ 180 (* .04 (attendees ticket-price))))
(define (attendees ticket-price)
  (+ 120 (* (/ 15 .10)
         (- 5.00 ticket-price))))

;; Bad program design
(define (profit price)
  (- (* (+ 120 (* (/ 15 .10)
                 (- 5.00 price)))
       (* .04
        (+ 180
          (* .04 (attendees ticket-price))))
    (+ 120 (* (/ 15 .10)
             (- 5.00 price)))))
Guidelines

- Naming **functions** helps
  - You write a program for the first time
  - Others read your program
    - Handy when you ask others for help with your code!
    - “Others” include you, too, after a few weeks

- Naming **constants** works similarly

- But you should always use your judgment
  - (define PI 3.13) vs. (define twenty-one 21)
Booleans and Conditionals

- With arithmetic, we can already do computation
  - But it’s pretty awkward (see HWK 2)
- Booleans values $\in \{\text{true, false}\}$ make conditionals easier
- In the lab, we were already used to the $=$ operator
  - Evaluating: $(= (+ 1 1) 2)$
  - Yields: true
Booleans and Conditionals

- $(> 5 \ 4)$ is “true”, $(> 1 \ 2)$ is “false”
- $(\text{and} \ (\text{not} \ false) \ (\text{or} \ false \ true))$ is $true$
- The “cond” statement lets us do more:
  
  ```scheme
  (define (day n)
      (cond
        [(= n 1) "Monday"]
        [(= n 2) "Tuesday"]
        ...)
  ```