



# Scheme Primitives and Function Definitions

---

Prof. Robert “Corky” Cartwright  
Department of Computer Science  
Rice University



# Course Overview

---

- Functional program design in Scheme
  - Data-directed (functional) program design 2-10
  - Algorithm design 11-14
  - Applied functional programming 15-17
- Object-oriented (OO) program design in Java 18-45
  - ...



# Today's Goals

---

- Common basic types
- Common primitive operations
- Rules for reducing programs
- Simple programs =
  - Variable definitions (Constants)
  - + Function definitions
- The design recipe
- Errors
- Data definitions



# Basic (primitive) types of data

---

numbers:

- naturals: 0, 1, 2, ... // number theory
- integers: ..., -1, 0, 1, ... // include negatives
- rational numbers: 3/4, 0, -1/3, ... // include fractions
- inexact numbers: #i0.123, #i0, ... // floating point numbers

Operations: +, -, \*, /, expt, remainder

Scheme computes exact answers on exact inputs if possible

booleans: false, true

Operations: not, and, or, ...

Symbols: 'A, 'a, 'Aa, 'Corky, ...

Operations: ... // none important for now

Other basic types: strings, lists, ... // none important for now



# Mixed-type Operations and Primitive Computation

---

- Basic relational operators
  - `equal?` // all data values
  - `=, <, >, <=, >=` // only on numbers
- Primitive computation  $\equiv$  application of a basic operation to constants
  - Basic operation  $\equiv$  basic function
  - Soon, we will see how to define our own (non-primitive) functions
- Function application in Scheme: parenthesized prefix notation
  - Scheme uses parenthesized prefix notation uniformly for **everything**
  - `(+ 2 2)`, `(sqrt 25)`, `(remainder 7 3)`
  - Bigger example: `(* (+ 1 2) (+ 3 4))`
  - How does this compare to writing `1+2*3+4` ?
- Scheme syntax is simple, uniform, and avoids possible ambiguity



# Computation is repeated reduction

---

- *Every Scheme program execution is the evaluation of a given expression constructed from primitive or defined functions and variables (names for constants).*
- *Evaluation proceeds by repeatedly performing the leftmost possible reduction (simplification) until the resulting expression is a **value**.*
- *A **value** is any constant. We will identify all of the expressions that are values as we explicate the language. Numbers, booleans, symbols are all values.*



# Reduction for primitive functions

- A *reduction* is an atomic computational step that replaces some expression by a simpler expression as specified by a Scheme evaluation rule (law). Every application of a basic operation to values yields a value (where run-time error is a special kind of value).

- Example

```
(* (+ 1 2) (+ 3 4))  
=> (reduces to) (* 3 (+ 3 4))  
=> (* 3 7) => 21
```

- Always perform leftmost reduction
- The following is **not** an atomic step, and so **not** a reduction

```
(- (+ 1 3) (+ 1 3)) □□ 0
```

# Programs =

# Variable Definitions + Function Definitions

---

- Variables are simply names for values
  - `pi`, `my-SSN`, `album-name`, `tax-rate`, `x`
- Variable definitions
  - `(define freezing 32)`
  - `(define boiling 212)`
- Function definitions
  - `(define (area-of-box x) (* x x))`
  - `(define (half x) (/ x 2))`
- Function applications (just as we saw before)
  - `(area-of-box 2)`
  - `(half (area-of-box 3))`
- Almost **any** function `f` used in a program can be written in the form
  - `(define (f v1 ... vn) <expression>)`

where `<expression>` is constructed from constants, variables, function applications, and a few other constructs TBN.





# Reductions for defined functions

---

- Assume we have declared the two functions
  - `(define (area-of-box x) (* x x))`
  - `(define (half x) (/ x 2))`
- Then Scheme can perform these reductions
  - `(half (area-of-box 3))`
  - `=> (half (* 3 3))`
  - `=> (half 9)`
  - `=> (/ 9 2)`
  - `=> 4.5`
- Reduction stops when we get to a value or an error



# Example: Solve quadratic equation

```
;; Contract solve-quadratic: number number number -> number  
;; Purpose: (solve-quadratic a b c) finds the larger root of  
a*x*x + b*x + c = 0 given it has real roots and a != 0
```

**Step 2**

```
;; Examples: (solve-quadratic 1 0 -25) = 5  
;;             (solve-quadratic 5 0 -20) = 2  
;;             (solve-quadratic 1 -10 25) = -4  
;;             . . . and other examples
```

**Step 3**

```
;; Template instantiation: (degenerate)  
;; (define (solve-quadratic a b c) ... )
```

**Step 4**

```
;; Code  
;; (define (solve-quadratic a b c)  
;;   (/ (+ (- b) (sqrt (- (* b b) (* 4 a c)))) (* 2 a)))
```

**Step 5**

```
;; Tests for solve-quadratic  
;; (check-expect exp ans) reports error if exp != ans  
  (check-expect (solve-quadratic 1 0 -25) 5)  
  (check-expect (solve-quadratic 5 0 -20) 2)  
  (check-expect (solve-quadratic 1 -10 25) -4)
```

**Step 6**



# Syntax Errors

---

- A syntactically correct **expression** can be
  - An *atomic* expression, like
    - a number `17`, `4.5`, `#i0.34`
    - a variable `radius`
  - A *compound* **expression**,
    - starting with `(`
    - followed by basic or program-defined operation such as `+` or `⊖`
    - one or more **expressions** separated by spaces
    - ending with `)`
- Syntax errors: `3 + 4`   `+(3, 4)`   `3)`   `(5`



# Runtime Errors

---

- Happen when basic operations are applied to illegal arguments
- Consider the following examples:
  - `(sqrt 1 2 3 4) => error: sqrt applied to more than one argument`
  - `(18 17) => error: 18 applied as function ;;`
  - `(/ 1 0) => error: division by zero`
  - `(+ 1 'a) => error: second argument in application of + is not a number`
- If a reduction produces an error, the computation is aborted and the error is returned as the result.
- Try things like that in DrScheme, and make a mental note of the error messages you get back.



# Conditional Expressions

---

- An expression that distinguishes different forms of data
- Form:

```
(cond [question-1 result-1
      [question-2 result-2]
      ...
      [question-n result-n]
      [else default-result])
```

- Square brackets are used above for clarity. In Scheme, they are synonymous with parentheses, but balancing brackets must match.
- `else` is optional. If omitted and none of the questions are true, the result is a run-time error (like division by zero).



# Reduction of Conditional Expressions

---

```
(cond [true  result
      ...  ]
=> result
```

```
(cond [false  result
      ...  ]
=> (cond ... )
```

```
(cond [else  result])
=> result
```

```
(cond [false result])
=> error: all question results were false
```



# Conditional Expression Examples

---

```
(cond [(> 12 0) 5] [else -5])  
=> (cond [true 5] [else -5])  
=> 5
```

Given

```
(define (abs x)  
  (cond [(>= x 0) x]  
        [else (- x)]))
```

```
(abs -10)  
=> (cond [(>= -10 0) -10] [else (- -10)])  
=> (cond [false -10] [else (- -10)])  
=> (cond [else (- -10)]) => (- -10) => 10
```



# The Design Recipe

---

How should I go about writing programs?

1. Analyze problem and define any requisite data types.
2. State contract (type) and purpose for *function* that solves the problem.
3. Give examples of function use and result.
4. Select and instantiate a template for the function body.
5. Write the function itself.
6. Test it, and confirm that tests succeeded.

The order of the steps of the recipe is important





# The Design Recipe (Big Picture)

---

- Encourages systematic problem solving
- Works best if keep our functions small
- We will learn how to repeatedly decompose problems into simpler problems until we reach problems that can be solved by simple expressions as in `solve-quadratic`
- Decomposition driven by structure of data being processed: *data-directed* design



# Reminders

---

- New homework (HW1) is posted online
  - Due next Friday, so you will get to check it over in lab; don't wait until your lab to get started.
  - Sign up for mailing list to get any updates, discussions
  - Make absolutely sure you follow the **recipe** in writing Scheme programs.
  - Partners: Talk to people after class, at lab, etc.
  - For Scheme programs, follow format of the sample solution in the Scheme HW Guide.
  - For hand evaluations, follow the format of the hand evaluation problems posted in the Scheme HW Guide.
  - Submit your assignment using Owlspace.



# Epilog

---

- Reminder: continue digesting chs. 1-10 in HTDP Section 8.3 is particularly important and it is not wordy.
- Next class
  - *Inductive Data* definitions
  - Amplified design recipe
- Challenge problem: What happens if we use rightmost reduction instead of leftmost? Can you devise a program using the Scheme subset given in this lecture such that some invocation of that program (expression composed from constants and and basic and program-defined operations defined in the program) behaves differently (either in terms the result produced by the computation or lack thereof) under rightmost evaluation than leftmost evaluation. Hint: focus on pathological behavior and note that two different errors are not equivalent.