

# Where are We?

## Part I: Basic software engineering

- How to represent things
- How to build programs around those representations

*Mid-term 1*

## Part II: Scaling Up

- Abstraction
- Algorithms and state

*Mid-term 2*

## Part III: Another notation, more libraries

- Java

# Advanced Scheme

A `<defn>` is one of

```
(define <var> <exp> )  
(define ( <var> <var> ... <var> ) <exp> )  
(define-struct <var> ( <var> ... <var> ) )
```

An `<exp>` is one of

```
<var>  
<con>  
<prim>  
( <exp> <exp> ... <exp> )  
( cond [ <exp> <exp> ] ... [ <exp> <exp> ] )  
( cond [ <exp> <exp> ] ... [ else <exp> ] )  
( and <exp> ... <exp> )  
( or <exp> ... <exp> )  
( local [ <defn> ... ] <exp> )  
( lambda ( <var> ... <var> ) <exp> )  
( set! <var> <exp> )  
( begin <exp> ... <exp> )
```

# Mini Scheme

A `<defn>` is one of

```
(define <var> <exp>)
```

```
(define <var> (lambda (<var>) <exp>))
```

An `<assign>` is

```
(set! <var> <exp>)
```

An `<exp>` is one of

```
<var>
```

```
<num>
```

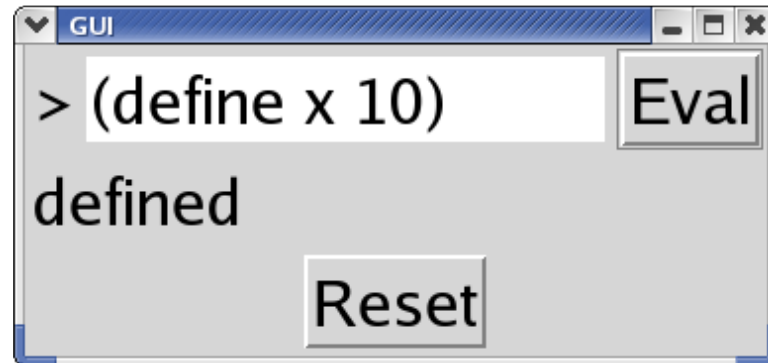
```
(+ <exp> <exp>)
```

```
(- <exp> <exp>)
```

```
(* <exp> <exp>)
```

```
(<var> <exp>)
```

# HW 10 and 11: Implementing DrMiniScheme

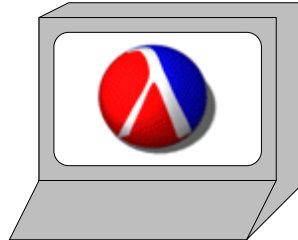


Key design problems for DrMiniScheme:

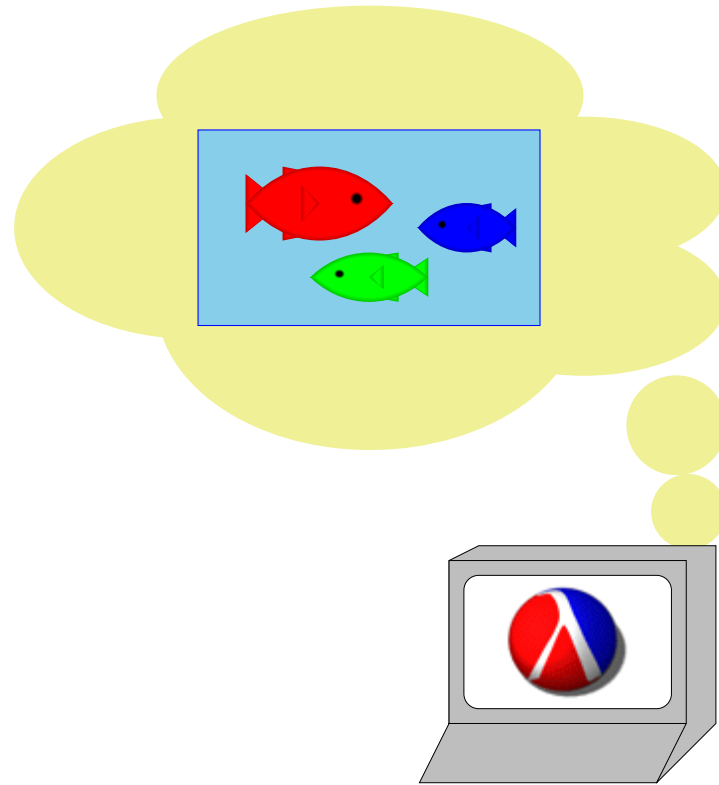
- Representing definitions and expressions
- Executing definitions and expressions
- Controlling the GUI

- **Representing definitions and expressions**
- **Converting strings to representations**
- **Evaluating Expressions**

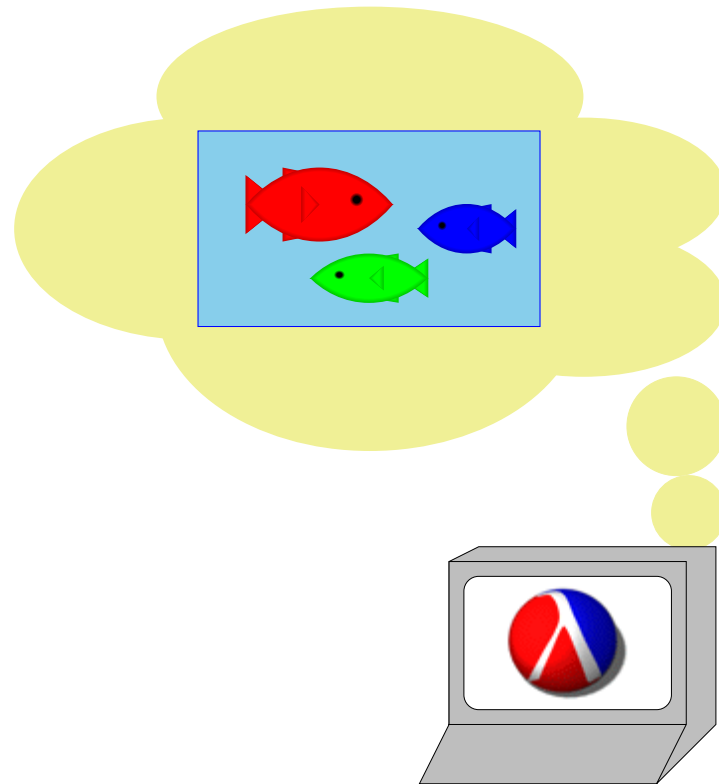
# Implementing Aquariums in Advanced Scheme



# Implementing Aquariums in Advanced Scheme



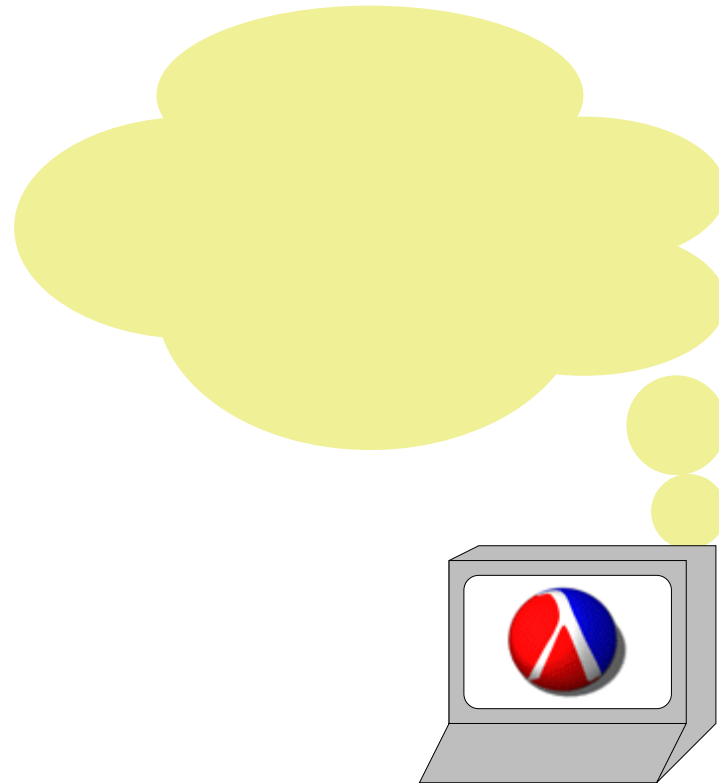
# Implementing Aquariums in Advanced Scheme



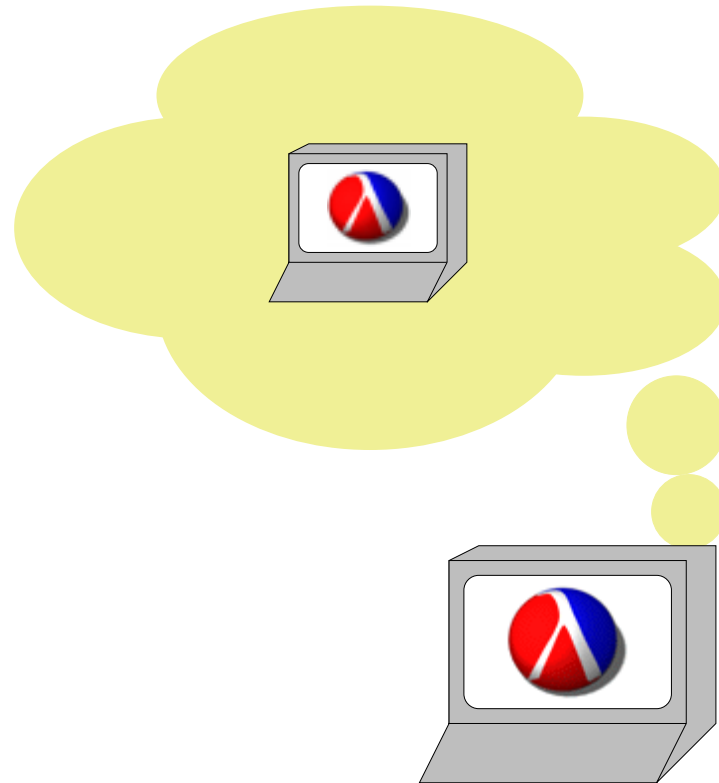
Represent fish, as opposed to stuffing *real* fish into DrScheme



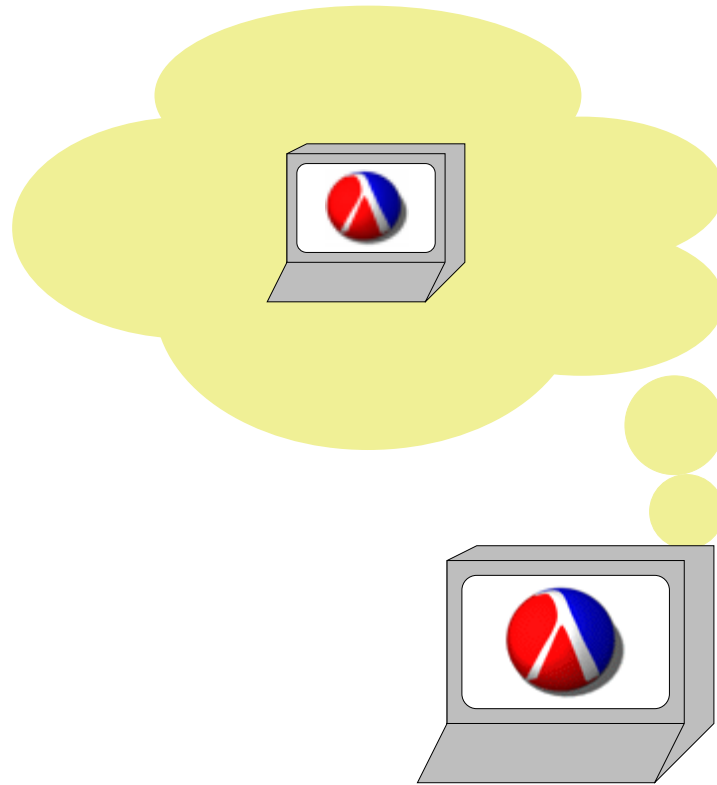
# Implementing Mini Scheme in Advanced Scheme



# Implementing Mini Scheme in Advanced Scheme



# Implementing Mini Scheme in Advanced Scheme



Represent Mini Scheme expressions, as opposed to typing *real* expressions into DrScheme

# Representing Mini Scheme Expressions

An `<exp>` is one of

`<var>`

`<num>`

`( + <exp> <exp> )`

`( - <exp> <exp> )`

`( * <exp> <exp> )`

`( <var> <exp> )`

---

We can't simply write

`( + 1 2 )`

to represent a Mini Scheme addition expression

# Representing Mini Scheme Expressions

An `<exp>` is one of

`<var>`

`<num>`

`( + <exp> <exp> )`

`( - <exp> <exp> )`

`( * <exp> <exp> )`

`( <var> <exp> )`

---

We can write

`' ( + 1 2 )`

which is almost as convenient!

# Representing Mini Scheme Expressions

An `<exp>` is one of

`<var>`

`<num>`

`( + <exp> <exp> )`

`( - <exp> <exp> )`

`( * <exp> <exp> )`

`( <var> <exp> )`

---

To represent the `<var>` `x`:

`'x`

# Representing Mini Scheme Expressions

An `<exp>` is one of

`<var>`

`<num>`

`( + <exp> <exp> )`

`( - <exp> <exp> )`

`( * <exp> <exp> )`

`( <var> <exp> )`

---

To represent the `<num>` 5:

`'5`

which is actually just 5

# Representing Mini Scheme Expressions

An `<exp>` is one of

`<var>`

`<num>`

`( + <exp> <exp> )`

`( - <exp> <exp> )`

`( * <exp> <exp> )`

`( <var> <exp> )`

---

To represent the application `( f ( + 1 2 ) )`

`' ( f ( + 1 2 ) )`

which is the same as

`( list ' f ( list ' + 1 2 ) )`



# Representing Mini Scheme Expressions

Data definition:

```
; A expr-snl is either  
; - sym  
; - num  
; - (list '+ expr-snl expr-snl)  
; - (list '- expr-snl expr-snl)  
; - (list '* expr-snl expr-snl)  
; - (list sym expr-snl)
```

# Representing Mini Scheme Expressions

A better data definition in the long run:

```
; A expr-snl is either  
;   - sym  
;   - num  
;   - add-expr-snl  
;   ...  
;  
; An add-expr-snl is  
;   (list '+ expr-snl expr-snl)  
;  
;   ...
```

# Representing Definitions and Assignments

```
; A defn-snl is either
;   - (list 'define sym expr-snl)
;   - (list 'define sym
;           (list 'lambda (list sym)
;                 expr-snl))
;
; An assign-snl is
; (list 'set! sym expr-snl)
```

## HW 10

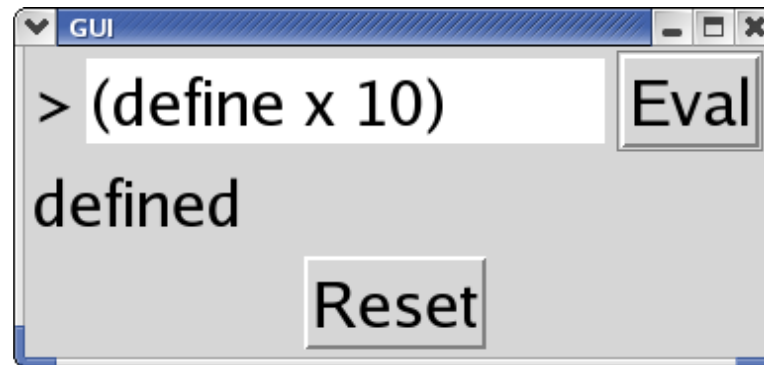
HW 10 simplification: only define/assign to numbers, and only evaluate variable names

```
; A defn-snl is  
; (list 'define sym num)  
;  
; An assign-snl is  
; (list 'set! sym num)  
;  
; An expr-snl is  
; sym
```

- **Representing definitions and expressions**
- **Converting strings to representations**
- **Evaluating Expressions**

# Converting a String to a Mini Scheme Expression

In the GUI, the definition/assignment/expression is available only as a string:



The `read-from-string` teachpack function converts a string by putting a quote in front of it

```
(read-from-string "1") → 1  
(read-from-string "(+ 1 2)") → '(+ 1 2)  
(read-from-string "(define x 7)")  
→ '(define x 7)
```

# The snl Datatype

```
; read-from-string : string -> snl

; An snl is either
;   - sym
;   - num
;   - list-of-snl
```

Example `snls`:

```
'x
1
'(1 1 1 x 1)
```

Not every `snl` is a `defn-snl`, `assign-snl`, or `expr-snl`

# Checking for Definitions

```
; is-defn? : snl -> bool
(define (is-defn? s)
  (and (list? s)
        (= 3 (length s))
        (symbol? (first s))
        (symbol=? 'define (first s))
        (symbol? (second s))
        (number? (third s))))
```



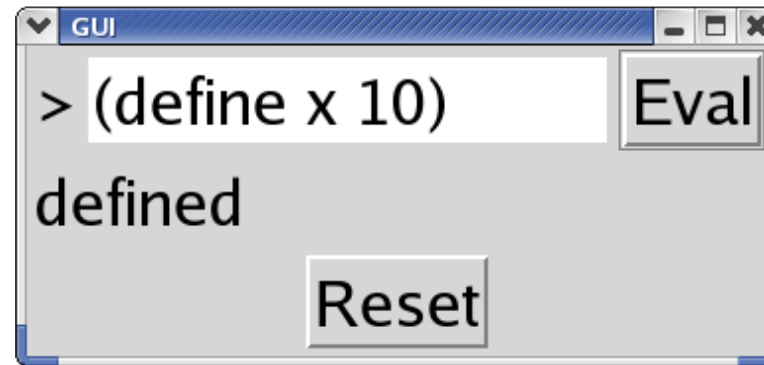
# Checking for Expressions (HW 11)

```
; is-expr? : snl -> bool
(define (is-expr? s)
  (or (number? s)
      (symbol? s)
      (is-plus? s)
      ...))
```

```
; is-plus? : snl -> bool
(define (is-plus? s)
  (and (list? s)
       (= 3 (length s))
       (symbol? (first s))
       (symbol=? '+ (first s))
       (is-expr? (second s))
       (is-expr? (third s))))
```

...

# Executing Code



When the **Eval** button is clicked:

- If it's a definition, record it
- If it's an assignment, do it
- If it's an expression, evaluate it

## Execution

```
; execute : snl -> string
(define (execute s)
  (cond
    [(is-defn? s) ...]
    [(is-assign? s) ...]
    [(is-expr? s) ...]
    [else "bad input"]))
...
; execute-string : snl -> string
; Used by the Execute button callback
(define (execute-string str)
  (local [(define snl (read-from-string str))])
  (cond
    [(boolean? snl) "bad input"]
    [else (execute s)])))
```

- **Representing definitions and expressions**
- **Converting strings to representations**
- **Evaluating Expressions**

## Evaluating Mini Scheme (HW 11)

```
(evaluate '3) "should be" 3
```

```
(evaluate '(+ 1 2)) "should be" 3
```

```
(evaluate '(+ 1 (* 2 5))) "should be" 11
```

Assuming (define f (lambda (x) (+ x 1))):

```
(evaluate '(f 7))
```

## Evaluating Mini Scheme (HW 11)

```
(evaluate '3) "should be" 3
```

```
(evaluate '(+ 1 2)) "should be" 3
```

```
(evaluate '(+ 1 (* 2 5))) "should be" 11
```

Assuming (define f (lambda (x) (+ x 1))):

```
(evaluate '(f 7))
```

```
"should be" 8
```

involves substituting 7 into (+ x 1)

# Evaluating Mini Scheme

```
; evaluate : expr-snl -> value
(define (evaluate s)
  (cond
    [(number? s) ...]
    [(symbol? s) ...]
    [(is-plus? s) ... (evaluate-plus s) ...]
    [(is-minus? s) ... (evaluate-minus s) ...]
    [(is-times? s) ... (evaluate-times s) ...]
    [(is-app? s) ... (evaluate-app s) ...]))
```

# Evaluating Mini Scheme

```
; ...  
; A plus-expr-snl is  
; (list '+ expr-snl expr-snl)  
; ...  
  
; evaluate-plus : plus-expr-snl -> value  
(define (evaluate-plus s)  
  ... (evaluate (second s))  
  ... (evaluate (third s))  
  ...)
```



## Evaluating Mini Scheme

```
; evaluate-app : plus-expr-snl -> value  
(define (evaluate-app s)  
  ... (first s)  
  ... (evaluate (second s))  
  ...)
```

Assuming the first is defined as a function, the next step is to substitute...

# Substitution

Assuming `(define f (lambda (x) (+ x 1)))`:

```
(evaluate-app '(f (- 20 5)))
```

```
→ → (substitute 'x 15 '(+ 1 x))
```

```
(substitute 'x 15 '(+ 1 x))
```

```
"should be" '(+ 1 15)
```

# Summary

## HW 10

- Just definitions, assignments, variable lookup

## HW 11

- Expression evaluation
- Optional exercises: errors, conditionals