# Scheme in Scheme 2

## What's next

- Adding set!
- Dynamic vs. lexical variable scope
- Extending mcscheme v1 with libraries
- Can mcscheme execute mcscheme?

## Adding set!

- mcs.ss v1 has *define* but not *set!*
- *define* gives a variable in the current local environment a value, adding it if necessary
- *set!* Finds a variable in the environment (local or inherited) and changes its value
- Adding set! To mcs.ss requires us to add a new special form to mceval and an associated function to do the work

## Adding set! (2)

**The new special form code:**

```
(define (mcdefine exp env)
  (cond
    …
  ;; (set! x (+ x 1))
  ((eq? (first exp) 'set!)
    (mcset (second exp)
           (mceval (third exp) env)
           env))
  …)
```

## Adding set! (3)

**The mcset function:**

```
(define (mcset var val env)
  (cond ((null? env)
         (mcerror "Unbound: " var))
        ((massoc var (first env))
         (set-mcdr! (massoc var (first env)) val)
         (void))
        (else (mcset var (rest env) env))))
```

## Dynamic vs. lexical scope

- We start with the global environment
- Calling a function creates a local environment
- The code in a function can reference local variables (e.g., function parameters) and non-local variables
- Where do we look for these non-loval variables?
- Two common schemes: lexical and dynamic

## Dynamic vs. lexical scope

- Lexical variable scope:

  Non-local variables in a function referenced in environment in which function was defined
- Dynamic variable scope:

  Non-local variables in a function referenced in environment that called function
- Consider

```
> (define x 1)
> (define (foo x)  (x-plus-1))
> (define (x-plus-1) (+ x 1))
> (foo 100)
```

## Dynamic vs. lexical scope (2)

- Consider

```
> (define x 1)
> (define (foo x)  (x-plus-1))
> (define (x-plus-1) (+ x 1))
> (foo 100)
```

- Under lexical scoping
  - Returns 2
- Under dynamic scoping
  - Returns 102

## Dynamic vs. lexical scope (3)

- Its easy to change mcs.ss to use dynamic scope
- We have to change the apply routine that calls a user-defined function
  - Apply creates a new environment for local variables by extending an environment
  - Need to extend caller's environment instead of the one stored with user-defined function
- Change mcapply and how it's called
  - Add global variable dynamic_scoping? to switch back and forth

## Dynamic vs. lexical scope (4)

- Was (for lexical):

```
;; (foo x)
(else (mcapply (mceval (first exp) env)
               (map (lambda (x) (mceval x env))
                    (rest exp))))))
```

- Change (for dynamic):

```
;; (foo x)
(else (mcapply (mceval (first exp) env)
               (map (lambda (x) (mceval x env))
                    (rest exp))
               env)
```

Pass the caller's environment

## Dynamic vs. lexical scope (5)

- Was (for lexical):

```
;; (foo x)
(else (mcapply (mceval (first exp) env)
               (map (lambda (x) (mceval x env))
                    (rest exp))))))
```

- Change (for dynamic):

```
;; (foo x)
(else (mcapply (mceval (first exp) env)
               (map (lambda (x) (mceval x env))
                    (rest exp))
               env)
```

Pass the caller's environment

## Dynamic vs. lexical scope (6)

Change mcapply to take caller's environment and use it if dynamic_scope? is true

```
(define (mcapply proc args caller_env)
 (cond ((procedure? proc) (apply proc args))
       ((and (pair? proc) (eq? (first proc) 'LAMBDA))
        ;; e.g. add1: (LAMBDA (x) (+ x 1) (..def_env..))
        (mceval (third proc)
           (cons (make-frame (second proc) args)
                 (if dynamic_scope?
                     caller_env
                     (fourth proc))))
       (else (mcerror "Undefined:" proc))))
```

## Extending via libraries

- A common way to extend a languages is via libraries
- Can include *standard* libraries that are always loaded as well as optional ones
- It's easy to do once the language core is implemented
- If we have an efficient interpreter or (better yet) a compiler, it may be reasonably efficient

## Can McScheme Interpret McScheme?

- In theory yes, in practice no
- McScheme v1 implements a subset of Scheme, call it $S_0$
- The McScheme v1 code uses a larger Scheme subset, call it $S_1$
- To allow this we'll have to enlarge $S_0$ and/or decrease $S_1$ so that $S_1 \leq S_0$

## mceval

```
(define (mceval exp env)
 (cond
  ((or (number? exp) (string? exp)
       (boolean? exp) (eof-object? exp)) exp)
  ((symbol? exp) (lookup exp env))
  ((eq? (first exp) 'quote) (second exp))
  ((eq? (first exp) 'begin) (last (map (lambda (x)(mceval x env)) (rest exp))))
  ((eq? (first exp) 'if)  (if (mceval (second exp) env)
                              (mceval (third exp) env)
                              (mceval (fourth exp) env)))
  ((eq? (first exp) 'define)
   (mcdefine (second exp) (mceval (third exp) env) env))
  ((eq? (first exp) 'load) (call-with-input-file (second exp) mcload))
  ((eq? (first exp) 'lambda)
   (list 'LAMBDA (second exp) (third exp) env))
  (else (mcapply (mceval (first exp) env)
                 (map (lambda (x)(mceval x env)) (rest exp))))))
```

## Getting this to work

- Forgo using some features in $S_1$, e.g., the more complex define form
  - replace (define (mceval exp env) …)
  - with (define mceval (lambda (exp env) …))
- Enlarge $S_0$ by adding functions to stdlib.ss and require this in mcs.ss
  - e.g.: (define map (lambda (f l) …))
- Some functions we need are special forms, e.g., cond and or
  - defining these in stdlib.ss will require macros!

## Conclusion

- We studied an interpreter for a very limited subset of Scheme
- It relies on the host language (Scheme) for many details (e.g., representing lists, primitive functions, read and print)
- We can easily expand this to a much larger subset of Scheme