Starting Point
Informal Scheme Semantics

- To evaluate \((E_1 \ E_2 \ldots \ E_n)\), recursively evaluate \(E_1, E_2, \ldots, E_n\) - \(E_1\) should evaluate to a function - and then apply the function value of \(E_1\) to the arguments given by the values of \(E_2, \ldots, E_n\).

- In the base case, there are self evaluating expressions (e.g. numbers and symbols). In addition, various special forms such as quote and if must be handled separately.
Theme

- Programs are functions and their semantics involve function application. Programs may also produce functions by returning functions as values. In pure functional programming, this is it, there are no variables, side effects, nor loops. This simplifies semantics but does not reduce computational power.

- We will investigate the style of programming this implies, and how to model the semantics of such programs. We conclude with a scheme interpreter [SICP – “Structure and Interpretation of Computer Programs by Abelson and Sussman”].
Outline

- Review scheme syntax and semantics
- Functional programming
  - Programs are functions – for every input there is a unique output
  - No variables ⇒ no assignment and no loops
  - Use recursion for control
  - Functions are first class objects
    - Pass as arguments and return as values
  - Simple semantics [value semantics] (no side effects, referential transparency)
Outline

- Substitution model of computation
- Modeling state with assignment
- Environments
- Streams and delayed evaluation
- Scheme interpreter (meta-circular)
Scheme Syntax and Semantics

- **S expressions** ($E_1 \ldots E_n$)
- **Special forms**
  - Self-evaluating: numbers, strings, …
  - `(quote E)`
  - `(if exp E1 E2)`
  - `(cond ((P1 E1) \ldots (Pt Et)))`
  - `(lambda (p1 \ldots pn) E1 \ldots Et)`
  - `(define name E)`
  - `(let ((b1 v1) \ldots (bt vt) E)`
Higher Order Functions

- **sort**: (sort '(4 3 2 1) <) => (1 2 3 4)
- **map**: (map sqr '(1 2 3 4)) => (1 4 9 16)
- **filter**: (keep-matching-items '(1 2 3 4 5) odd?) => (1 3 5)
- **reduce**: (reduce * 1 '(1 2 3 4)) => 24

- **Functions that return functions**
  - (define (make-adder x) (lambda (y) (+ x y)))
  - **Function composition**
    - (define (compose g f) (lambda (x) (g (f x))))
  - **Currying**
    - (define (curry f a) (lambda (b) (f a b)))
    - ((curry + x) y)
Substitution Model of Computation

- function application corresponds to substituting the argument expressions into the formal parameters of the function body

- Order of evaluation
  - Applicative vs. normal order
  - lambda calculus
  - Church-Rosser
Substitution Example

- (define (square x) (* x x))
- (define (sum-of-squares x y) (+ (square x) (square y)))
- (define (f a) (sum-of-squares (+ a 1) (* a 2)))

[applicative order]
- (f 5) ⇒ (sum-of-squares (+ 5 1) (* 5 2))
  ⇒ (+ (square 6) (square 10))
  ⇒ (+ (* 6 6) (* 10 10)) ⇒ (+ 36 100) ⇒ 136

[normal order]
- (f 5) ⇒ (sum-of-squares (+ 5 1) (* 5 2))
  ⇒ (+ (square (+ 5 1)) (square (* 5 2)))
  ⇒ (+ (* (+ 5 1) (+ 5 1)) (* (* 5 2) (* 5 2)))
  ⇒ (+ (* 6 6) (* 10 10)) ⇒ (+ 36 100)
Order Matters

(define (p) (p))

(define (test x y)
  (if (= x 0)
      0
      y))

(test 0 (p))
Environments

- When assignment is introduced the substitution model falls apart and a different, more complicated model, must be used.

- Environments used to store variables and bindings.
  - Values can change
  - Assignment supported
  - List of frames to support nested scope
Modeling State with Assignment

(define (make-withdraw balance)
  (lambda (amount)
    (if (>= balance amount)
        (begin (set! balance (- balance amount))
            balance)
        "Insufficient funds")))

Make-withdraw can be used as follows to create two objects W1 and W2:

(define W1 (make-withdraw 100))
(define W2 (make-withdraw 100))
(W1 50)
50
(W2 70)
30
(W2 40)
"Insufficient funds"
(W1 40)
10
Modeling State with Assignment

(define (make-account balance)
  (define (withdraw amount)
    (if (>= balance amount)
        (begin (set! balance (- balance amount))
               balance)
        "Insufficient funds")
  (define (deposit amount)
    (set! balance (+ balance amount))
    balance)
  (define (dispatch m)
    (cond ((eq? m 'withdraw) withdraw)
          ((eq? m 'deposit) deposit)
          (else (error "Unknown request -- MAKE-ACCOUNT" m))))
  dispatch)

(define acc (make-account 100))
  ((acc 'withdraw) 50) 50
  ((acc 'withdraw) 60) "Insufficient funds"
  ((acc 'deposit) 40) 90
  ((acc 'withdraw) 60) 30
Cost of Assignment

(define (make-simplified-withdraw balance)
  (lambda (amount)
    (set! balance (- balance amount))
    balance))

(define W (make-simplified-withdraw 25))
(W 10)
15
(W 10)
5

(define (make-decrementer balance)
  (lambda (amount)
    (- balance amount)))

(define D (make-decrementer 25))
(D 10)
15
(D 10)
15
Substitution Model Fails

(make-decremerter 25) 20)
 ⇒ ((lambda (amount) (- 25 amount)) 20)
 ⇒ (- 25 20) ⇒ 5

((make-simplified-withdraw 25) 20)
 ⇒ ((lambda (amount) (set! balance (- 25 amount)) 25) 20)
 ⇒ (set! balance (- 25 20)) 25
Environment Model Solves Problem

(define W1 (make-withdraw 100))
(W1 50)
(define W2 (make-withdraw 100))
Streams

- **Sequence of elements**
  - `(cons-stream x y)`
  - `(stream-car s)`
  - `(stream-cdr s)`
  - `(stream-null? s)`
  - `the-empty-stream`
Streams Motivation and E.G.

(define (sum-primes a b)
  (define (iter count accum)
    (cond ((> count b) accum)
      ((prime? count) (iter (+ count 1) (+ count accum)))
      (else (iter (+ count 1) accum))))
  (iter a 0))
(define (sum-primes a b)
   (accumulate +
               0
               (filter prime? (enumerate-interval a b))))
Streams are Not Lists

Consider the following example

(car (cdr (filter prime?  
  (enumerate-interval 10000 1000000)))))

This would be extremely inefficient if implemented with lists

- Do not build the entire stream of elements
- Get the next element from the stream when needed
- Necessary for potentially infinite streams
Delayed Binding

(cons-stream <a> <b>) ⇔ (cons <a> (delay <b>))

(define (stream-car stream) (car stream))
(define (stream-cdr stream) (force (cdr stream)))

(delay <exp>) ⇔ (lambda () <exp>)

(define (force delayed-object)
    (delayed-object))
Delayed Binding in Action

(stream-car
 (stream-cdr
  (stream-filter prime?
   (stream-enumerate-interval 10000 1000000))))

(define (stream-enumerate-interval low high)
  (if (> low high)
      the-empty-stream
      (cons-stream
       low
       (stream-enumerate-interval (+ low 1) high))))

(cons 10000
  (delay (stream-enumerate-interval 10001 1000000)))
Delayed Binding in Action

(define (stream-filter pred stream)
  (cond ((stream-null? stream) the-empty-stream)
    ((pred (stream-car stream))
      (cons-stream (stream-car stream)
        (stream-filter pred
          (stream-cdr stream))))
    (else (stream-filter pred (stream-cdr stream))))

stream-cdr forces

(cons 10001
  (delay (stream-enumerate-interval 10002 1000000))))
Delayed Binding in Action

(cons-stream (stream-car stream)
  (stream-filter pred (stream-cdr stream)))

which in this case is

(cons 10007
  (delay
    (stream-filter
      prime?
      (cons 10008
        (delay
          (stream-enumerate-interval 10009
            1000000))))))
Scheme Interpreter

1. To evaluate a combination (a compound expression other than a special form), evaluate the subexpressions and then apply the value of the operator subexpression to the values of the operand subexpressions.

2. To apply a compound procedure to a set of arguments, evaluate the body of the procedure in a new environment. To construct this environment, extend the environment part of the procedure object by a frame in which the formal parameters of the procedure are bound to the arguments to which the procedure is applied.
(define (eval exp env)
  (cond ((self-evaluating? exp) exp)
        ((variable? exp) (lookup-variable-value exp env))
        ((quoted? exp) (text-of-quotuation exp))
        ((assignment? exp) (eval-assignment exp env))
        ((definition? exp) (eval-definition exp env))
        ((if? exp) (eval-if exp env))
        ((lambda? exp)
         (make-procedure (lambda-parameters exp)
                         (lambda-body exp)
                         env))
        ((begin? exp)
         (eval-sequence (begin-actions exp) env))
        ((cond? exp) (eval (cond->if exp) env))
        ((application? exp)
         (apply (eval (operator exp) env)
                (list-of-values (operands exp) env)))
       (else
        (error "Unknown expression type -- EVAL" exp))))
Scheme Interpreter: apply

(define (apply procedure arguments)
  (cond ((primitive-procedure? procedure)
         (apply-primitive-procedure procedure arguments))
        ((compound-procedure? procedure)
         (eval-sequence
          (procedure-body procedure)
          (extend-environment
           (procedure-parameters procedure)
           arguments
           (procedure-environment procedure))))
        (else
         (error "Unknown procedure type -- APPLY" procedure)))))
Dynamic vs. Static Scope

- **Static scope**

  ```
  (define (make-adder x) (lambda (y) (+ x y)))
  (define add1 (make-adder 1))
  (define x 2)
  (add1 2)
  3
  ```

- **Dynamic scope**

  4
Scheme Interpreter: dynamic eval

(define (eval exp env)
  (cond ((self-evaluating? exp) exp)
        ((variable? exp) (lookup-variable-value exp env))
        ((quoted? exp) (text-of-quotation exp))
        ((assignment? exp) (eval-assignment exp env))
        ((definition? exp) (eval-definition exp env))
        ((if? exp) (eval-if exp env))
        ((lambda? exp) (make-procedure (lambda-parameters exp) (lambda-body exp) env)) ; don't need to store env - not used. JRJ
        ((begin? exp) (eval-sequence (begin-actions exp) env))
        ((cond? exp) (eval (cond->if exp) env))
        ((application? exp) (apply (eval (operator exp) env) (list-of-values (operands exp) env) env)) ; added env as an argument to apply. JRJ
        (else (error "Unknown expression type -- EVAL" exp))))
(define (apply procedure arguments env) ; added env parameter. JRJ
  (cond ((primitive-procedure? procedure)
    (apply-primitive-procedure procedure arguments))
  ((compound-procedure? procedure)
    (eval-sequence
      (procedure-body procedure)
      (extend-environment
        (procedure-parameters procedure)
        arguments
        env )))); extend runtime env instead of procedure-environment
  (else
    (error
      "Unknown procedure type -- APPLY" procedure))))
Practice Exercises

- Implement map and reduce

- Trace scheme interpreter from SICP using as input the following two expressions [you will have to add =, *, - as primitive procedures for this to work].

   (define (fact n)
      (if (= n 0) 1 (* n (fact (- n 1)))))
   (fact 3)
Practice Exercises

- Add the let expression to the interpreter
  - How can you convert \((\text{let } ((n_1 \ \text{v}_1) \ \ldots \ (n_t \ \text{v}_t)) \ E)\) to an equivalent scheme expression already handled by the scheme interpreter

- Modify the SICP interpreter to use dynamic instead of static scope. Provide an example where the same code provides two different answers depending on which scoping rule is used.
Practice Exercises

- Try out the streams example in MIT-scheme (streams are supported)
  - the-empty-stream
  - stream-enumerate-interval
  - stream-filter
  - prime?
  - (stream-car (stream-cdr (stream-filter prime? (stream-enumerate-interval 10000 1000000))))
Practice Exercises

- Modify the SICP interpreter to support streams
  - cons-stream
  - delay
  - Force
  - stream-car
  - stream-cdr
- Run the previous example using your modified interpreter
Assignment 6

- Re-implement assignment 2 or 3 (mini language with lists) to support procedures as first class objects
  - Proc values
    - proc(...) … end should be an expression which can be assigned to variables, passed as an argument, and returned by a procedure
  - Nested scope
    - You should support nested scope
    - Version 1 uses static scope
    - Version 2 uses dynamic scope
  - Test with make-adder [both static and dynamic scope]