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.
This lecture continues our exploration of the semantics of scheme programs. Previously we informally discussed the substitution model, environments, and provided an interpreter for scheme that operationally defined the semantics of scheme programs.

In this lecture we will more formally study the substitution model using the lambda calculus. We will also look at alternative (lazy) semantics for scheme and an implementation of streams.
The semantics of a pure functional programming language can be mathematically described by a substitution process that mimics our understanding of function application - i.e., substitute all occurrences of the formal parameters of a function in the function body by the arguments in the function call.

In order to formally study function definition and application and computability, logicians (Alonzo Church, Stephen Cole Kleene, and others) in the 1930s developed a formal notation called the lambda calculus and rules defining an equivalence relation on lambda expressions that encodes function definition and application.

They showed the universality of the lambda calculus, the uniqueness of normal forms, and the undecidablity of the equivalence of lambda expressions.
Outline

- Review substitution model of computation
- Review shortcomings of substitution model
- Review streams and delayed evaluation
  - Adding support for streams
  - Interpreter with lazy semantics
- Lambda calculus
  - Free and bound variables
  - alpha and beta reduction
  - Universality of lambda calculus
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) \(\Rightarrow\) (sum-of-squares (+ 5 1) (* 5 2))
\[\Rightarrow\] (+ (square 6) (square 10))
\[\Rightarrow\] (+ (* 6 6) (* 10 10)) \(\Rightarrow\) (+ 36 100) \(\Rightarrow\) 136

**[normal order]**

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

(define (p) (p))

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

(test 0 (p))
Environments

- When assignment is introduced the substitution model falls apart and a different model, 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
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-decrementer 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 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))
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-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))
        ((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))))
Stream Practice Exercises

- Complete the trace of

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

- Modify the scheme interpreter from SICP to support force, delay and streams. You should implement everything needed to execute the above computation.
Scheme Interpreter with Streams

; added to eval

((delay? exp) (eval-delay exp env))

((cons-stream? exp) (eval-cons-stream exp env))

; new functions

(define (delay? exp) (tagged-list? exp 'delay))

(define (eval-delay exp env)
  (make-procedure '() (cdr exp) env))

(define (eval-cons-stream exp env)
  (eval (list 'cons
            (cons-stream-op1 exp)
            (list 'delay (cons-stream-op2 exp)))
        env))
Scheme Interpreter with Streams

;;; The following need to be defined in the metacircular evaluator to
;;; complete the implementation of streams.

(define (force delayed-object)
  (delayed-object))

(define (stream-car stream)
  (car stream))

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

(define (stream-null? stream)
  (null? stream))
Lazy Evaluation

- Language with **normal order** semantics. Compound procedures are non-strict, while primitive procedures remain strict.
  - Procedure **strict** if arg evaluated before entering
  - Procedure **non-strict** if proc not evaluated before entering
- Delaying evaluation until last possible moment is called **lazy evaluation**.
- Delay evaluation of arguments by converting them to **thunks**.
- A thunk is forced only when its value is needed.
- **Memoize** thunks to avoid repeated evaluation.
Examples of Lazy Evaluation

(define (try a b)
  (if (= a 0) 1 b))

(try 0 (/ 1 0))

(define (unless condition usual-value exceptional-value)
  (if condition exceptional-value usual-value))

(unless (= b 0)
  (/ a b)
  (begin (display "exception: returning 0")
         0))
Lazy Scheme Interpreter: eval

((application? exp)
  (apply (actual-value (operator exp) env)
    (operands exp)
    env))

(define (actual-value exp env)
  (force-it (eval exp env)))

(define (eval-if exp env)
  (if (true? (actual-value (if-predicate exp) env))
      (eval (if-consequent exp) env)
      (eval (if-alternative exp) env))))
Lazy Scheme Interpreter: apply

(define (apply procedure arguments env)
  (cond ((primitive-procedure? procedure)
           (apply-primitive-procedure procedure
                                      (list-of-arg-values arguments env))) ; changed
       ((compound-procedure? procedure)
        (eval-sequence
         (procedure-body procedure)
         (extend-environment
          (procedure-parameters procedure)
          (list-of-delayed-args arguments env) ; changed
          (procedure-environment procedure))))
       (else
        (error "Unknown procedure type -- APPLY" procedure))))
Lazy Scheme Interpreter: apply

(define (list-of-arg-values exps env)
  (if (no-operands? exps)
      '()
      (cons (actual-value (first-operand exps) env)
            (list-of-arg-values (rest-operands exps)
                               env))))

(define (list-of-delayed-args exps env)
  (if (no-operands? exps)
      '()
      (cons (delay-it (first-operand exps) env)
            (list-of-delayed-args (rest-operands exps)
                                 env))))
Representing Thunks

(define (force-it obj)
  (if (thunk? obj)
      (actual-value (thunk-exp obj) (thunk-env obj))
      obj))

(define (delay-it exp env)
  (list 'thunk exp env))

(define (thunk? obj)
  (tagged-list? obj 'thunk))
Lazy Evaluation in Action

(define input-prompt ";;; L-Eval input:"

(define output-prompt ";;; L-Eval value:"

(define (driver-loop)
  (prompt-for-input input-prompt)
  (let ((input (read)))
    (let ((output
      (actual-value input the-global-environment)))
      (announce-output output-prompt)
      (user-print output)))
  (driver-loop))

(define the-global-environment (setup-
  environment))

(driver-loop)

;;; L-Eval input:
(define (try a b)
  (if (= a 0) 1 b))

;;; L-Eval value:
ok

;;; L-Eval input:
(try 0 (/ 1 0))

;;; L-Eval value:
1
Memoization

(define (fib n)
  (cond ((= n 0) 0)
        ((= n 1) 1)
        (else (+ (fib (- n 1))
                 (fib (- n 2)))))))

(fib 5)
Memoization

(define (memoize f)
  (let ((table (make-1d-table)))
    (lambda (x)
      (let ((previously-computed-result (1d-table/get table x #f)))
        (or previously-computed-result
            (let ((result (f x)))
              (1d-table/put! table x result)
              result))))))

(define memo-fib
  (memoize (lambda (n)
              (cond ((= n 0) 0)
                    ((= n 1) 1)
                    (else (+ (memo-fib (- n 1))
                              (memo-fib (- n 2))))))))
Memoized Thunks

(define (evaluated-thunk? obj)
  (tagged-list? obj 'evaluated-thunk))

(define (thunk-value evaluated-thunk) (cadr evaluated-thunk))

(define (force-it obj)
  (cond ((thunk? obj)
            (let ((result (actual-value
                           (thunk-exp obj)
                           (thunk-env obj))))
              (set-car! obj 'evaluated-thunk)
              (set-car! (cdr obj) result) ; replace exp with its value
              (set-cdr! (cdr obj)'())   ; forget unneeded env
              result))
  ((evaluated-thunk? obj)
   (thunk-value obj))
  (else obj))))
Streams for Free

- Previous implementation of streams required new special forms \texttt{delay} and \texttt{cons-stream} and the reimplementation of list functions for streams.

- In the lazy evaluator lists and streams are identical!

- Just need to make sure that \texttt{cons} is non-strict
  - Reimplement \texttt{cons} primitive
  - Or implement \texttt{cons}, \texttt{car}, and \texttt{cdr} as functions (see lambda calculus)
Lambda Calculus Expressions

1. \(< \text{exp} > \rightarrow \text{variable}\)
2. application: \(< \text{exp} > \rightarrow (\text{<exp} > \text{<exp} >)\)
3. abstraction: \(< \text{exp} > \rightarrow (\text{lambda} \text{variable} . \text{<exp} >)\)

- Use currying to allow more than one parameter
Free and Bound Variables

- The parameter of an abstraction is bound in the lambda body
  - (lambda (x) E)

- Variables not bound by a lambda are free

\[
\text{FV}(\text{exp}) = \begin{cases} 
\text{exp} & \text{exp is a symbol} \\
\text{FV(body(exp))} / \text{parameters(exp)} & \text{exp is an abstraction} \\
\text{FV(exp_1)} \cup \text{FV(exp_2)} & \text{exp = (exp_1 exp_2)}
\end{cases}
\]
Lambda Calculus Reductions

1. **alpha conversion**: 
   (lambda x.E) is equivalent to 
   (lambda y,E[y/x]) provided y does not appear 
   free in E and y is not bound by a lambda when 
   substituted for x in E.

2. **beta reduction**:
   ((lambda x.E) F) is equivalent to 
   E[F/x], where F is substituted for all free 
   occurrences of the variable x in E, provided all 
   free variables in F remain free when substituted 
   for x in E.
Universality of Lambda Calculus

1. Church numerals and arithmetic using lambda calculus
2. boolean logic, predicates, and conditional statements using lambda calculus
3. Data structures (lists, cons, car, cdr) using lambda calculus
4. Recursion using lambda calculus
Y Combinator

\[(\text{define fact } (\lambda (n) (\text{if } (= n 0) 1 (* n (\text{fact} (- n 1)))))\)]

1. \[(\text{define } Y (\lambda (g) ((\lambda (x) (g (x x))) (\lambda (x) (g (x x))))))\]

2. \[(g (Y g)) = (Y g) \text{ [fixed point]}\]

3. \[(\text{define } g (\lambda (n f) (\text{if } (= n 0) 1 (* n (f (- n 1)))))\]

4. \[(g (Y g) n)\]
Lambda Calculus Practice Exc. 1

Implement a scheme function (FV exp), which returns a list of the free variables in the scheme expression exp. The scheme expression will not have let, letrec, or let* statements - the only bindings come from lambda expressions.
Show that the and, or, and not functions using the lambda calculus in lecture 2b are correct by showing that the correct result is obtained for all combinations of the inputs.
Using beta-reduction show that (succ two) is three, where two and three are the Church encodings the numbers 2 and 3 (see the scheme code for Church numbers in lecture 2b.

Use induction to prove that ((cn add1) 0) = the number n, where cn is the Church encoding of n. You may assume that add1 correctly adds 1 to its input.
Lambda Calculus Practice Exc. 4

Trace through the expansion (use beta reduction) of \((g \ (Y \ g) \ 3)\) using the functions \(g\) and \(Y\) from the discussion on the \(Y\) combinator. Try implementing \(g\) and \(Y\) and computing \((g \ (Y \ g) \ 3)\) in scheme. What happens and why? Try this again using the normal order scheme interpreter in Section 4.2.2 of SICP.
Lambda Calculus Practice Exc. 5

Implement a scheme function that performs a beta reduction. You must also have a predicate which checks to see if the beta reduction is valid (i.e. the substitution does not cause problems with name collisions)