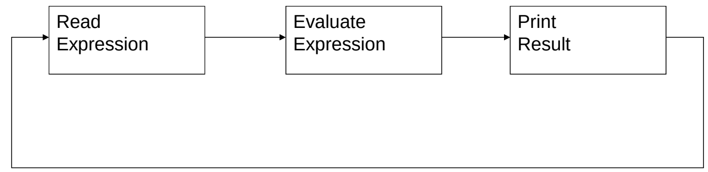
Functional Languages (Objectives)

- The student will be able to write simple expressions in Scheme syntax (s-expressions).
- Given an assignment involving lists, the student will be able to manipulate the basic list structure for Scheme data.
- Given a programming assignment, the student will be able to write Scheme functions to solve the problem.
- Given a programming assignment, the student will be able to write recursive functions to solve the problem.
- Given a two functionally similar programs, the student will be able to create a functional abstraction of the two programs.
- Given a programming assignment, the student will be able to use 1st-class functions to solve the assignment.
- Given a programming assignment involving laziness, the student will be able to construct a solution in lazy Scheme.

Getting Started

- Interpretation is done in a read-eval-print loop



- expression-oriented syntax rather than statement-oriented.
  - In pure Scheme, there is no assignment.
- dynamic type checking

Basic Data

- The basic unit of syntax in Scheme is the atom.
  - numbers, variables, symbols, strings
- If there is no definition of an atom the result will be an error
- Examples (maroon characters from interpreter)

```
> 2
  2
> X
   x: undefined;
    cannot reference undefined identifier
> +
   #<procedure:+>
```


Basic Data

- Quoted atoms – atoms with a leading single quote
- Evaluates to the sequence of characters following the quote (stops at first blank)
- Examples

```
> '2
  2
> 'x
  x
> 'bbb
  bbb
```

Basic Data

- Quoted flat lists – a sequence of atoms enclosed by parentheses with a leading single quote
- Evaluate to a Scheme flat list of atoms
- Examples

> '(a)
(a) {a flat list containing the atom a)
> '(a b c d)
(a b c d) {a flat list containing the atoms a, b, c, d}
> '()
() {an empty or null list)
> '(list of atoms)
(list of atoms) {a flat list containing the atoms list, of, atoms}

Basic Data

- Quoted s-lists (or just a list)– a list of atoms and/or lists
  - Evaluates to the list itself
  - A flat list is a list
- Example

> '(a (a b) a)
(a (a b) a) {a list of a, a list of a and b, and a}
> '(a (a (b a)))
(a (a (b a))) {a list of a and a list of a and a list of b and a}
> '((a))
((a)) {a list of a list of a}
> '(())
(()) {a list of the null list}

Practice Problems

- Which of the following are atoms (possibly quoted)?
  - 2
  - '2
  - 'x
  - '()
  - 'turkey
  - '(x)

Practice Problems

- Which of the following are flat lists of atoms?
  - (a b)
  - '(a b)
  - '(a b c d)
  - '()
  - '(a b (a) b)
  - '((a))
  - a
  - (b c)
- Which of the above are lists?

S-Expressions

- S-expressions – an atom or list with or without the quote
- Example:
  > (+ 1 2)
  3
- General form:
  (operator operand_1 operand_2 ... operand_n)
  - n can be 0

Evaluation

(operator operand_1 operand_2 ... operand_n)

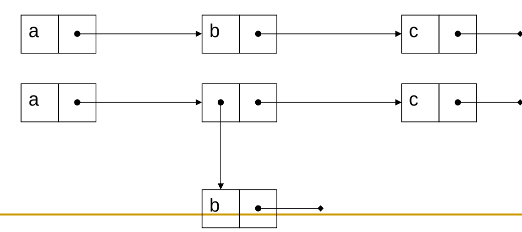
- evaluate operator – must be a function
- evaluate operand_1 through operand_n in no particular order
- call operator with arguments
  (+ 1 2)
  1. evaluate + to the addition function
  2. evaluate 1 and 2 to their numeric values
  3. apply + to 1 and 2
  4. return result -- 3

Operations on Numeric Data

- Some Scheme functions
  - (- 3 4) → -1
  - (* 3 4) → 12
  - (/ 3 4) → $\frac{3}{4}$
  - (sqrt 4) → 2
  - (expt 2 4) → 16
  - (> 3 4) → #f
  - (<= 3 4) → #t

Scheme Lists

- Scheme list have two parts
  1. first – the head of list
  2. rest – the list without the first element
- The two parts form a cons cell
  - Example '(a b c) and '(a (b) c)



List Functions

- (first L) - returns the head of the list L
- (rest L) – returns a list containing everything in L except the first element
- (cons S-exp L) – constructs a new list with S-exp as the head of the list and L as the rest of the list.

Examples

1. (first '(a b c)) →
2. (first 'hotdog) →
3. (first '((a b c) x y z)) →
4. (first '()) →

5. (rest '(a b c)) →
6. (rest 'hotdog) →
7. (rest '((a b c) x y z)) →
8. (rest '()) →

Examples

1. (cons 'peanut '(butter and jelly))
   →
2. (cons '(mayo and) '(peanut butter and jelly))
   →
3. (cons '((help) this) '(is very ((hard) to learn)))
   →
4. (cons '(a b c) '())
   →
5. (cons 'a 'b)
   →
6. (cons '(a b (c)) '(d))
   →

Examples

1. (cons 'a (first '((b) c d))
   →
2. (cons 'a (rest '((b) c d))
   →

Practice Problems

1. Construct a series of cons expressions to construct the following lists:
   a) '(1 2 3 4)
   b) '(1 (2 3 4))
   c) '(1 (2 3) 4)
   d) '((1 2) (3 4))
   e) '(((one)))
2. Evaluate the following:
   a) (first '(a b c d))
   b) (rest '(a b c d))
   c) (rest '((a (b c) d)))
   d) (first '((a (b c) d)))
   e) (first (rest '(a b c d)))
   f) (first (rest '((1 2) (3 4)))
   g) (rest (first '((1 2) (3 4)))

Other Functions

- determine if a list is null
  (null? '(a)) →
  (null? '()) →
- determine if two atoms are equal
  (eq? 'a 'a) →
  (eq? 'a 'b) →
  (eq? '(a) '(a)) →
- determine if two lists are equal
  (equal? 'a 'a) →
  (equal? '(a) '(a)) →
  (equal? '(a (b)) '(a (b))) →

Conditionals

- if
```
> (if (eq? 2 2) 2 3)

> (if (< 3 2) 'yes 'no)

> (if (null? '(a))
      'null
     (first '(a)))
```
- if-then-else
```
> (cond
     [(> 3 3) 'error]
     [(= 3 3) 'good]
     [else 'phbbt])
```


Variable and Function Definitions

- Variables are defined using the keyword define

  (define pi 3.14159)

  - Note that there are no type specifications (dynamic typing)
- Functions are the programming feature upon which Scheme is grounded – like mathematics
  - Parameters are pass-by-value
  - Use a modified form of define
  - Example: square a number

```
(define (square x)
   (* x x))

> (square 2)
  4
```

Practice Problems

- Write Scheme functions to
  1. compute the cube of a number
  2. return the second element of a three-element list
  3. return the sum of the elements of a three-element list

Recursion

- A recursive function is one that is defined in terms of itself.
- loops and simple recursion are equivalent
- Recursion has three major components (like inductive proofs)
  1. Base case or stopping condition
  2. Assumption that the function will work on a recursive call even though the function is not completely written.
  3. Recursive step (calling the function recursively)

Ten Commandments of Recursive Programming

1. Always ask null? or zero? as the first question in expressing any function.
2. Use cons to build lists.
3. When building a list, describe the first typical element and then cons it onto the natural recursion.
4. When recurring on a flat list of atoms ask two questions on them and use rest for the natural recursion. When recurring on a list of S-expressions ask three questions on the first and recur on both first and rest.
5. When building a value with + use 0 as the base case, with * use 1 and with cons use '().
6. Always change at least one argument when recurring. The changes should move closer to termination
7. Simplify only after correct
8. Recur on all subparts
9. Use other functions to abstract
10. Abstract function with common structures into a single function.

Example Recursion

```
int SumList(NumberList *L) {

    int sum = 0;

    while (!Null(L)) {
      sum += First(L);
      L = Rest(L);
    }

    return sum;
}

sum = SumList(L);
```

```
(define (SumList L sum)
    (if (null? L)
        sum
        (SumList (rest L)
            (+ sum (first L)))))

(SumList L 0)

(define (SumList L)
    (if (null? L)
        0
        (+ (first L)
           (SumList (rest L)))))

(SumList L)
```

Example

- Define a function that takes a list and determines if it is a list of numbers.
  1. What is the stopping condition?
  2. How do we operate on the first element?
  3. On what do we recur?

  (define (lon? L)

Example

- Define a function that takes a flat list and an atom and determines if that atom is in the list.
  1. What is the stopping condition?
  2. How do we operate on the first element?
  3. On what do we recur?

  (define (member? L a)

Example

- Define a function that takes two numbers and adds the first to itself the second number times.
  1. What is the stopping condition?
  2. How do we operate on the first element?
  3. On what do we recur?

  (define (addxn x n)

Modifying Variables

- Pure functional languages do not allow a programmer to modify variables via assignment.
- How do you change the value of a variable?
  - You don't. Instead, you create a new value and bind that value to a new variable via a function call.
- Examples
  1. Write a function that removes the third element of a list.

     Solution: write a function that constructs (cons) a new list consisting of all elements but the third.

Example

- Write a function that takes an atom and a flat list and removes the first occurrence of the atom from the flat list.
  1. What is the stopping condition?
  2. How do we operate on the first element?
  3. On what do we recur?

  (define (rember a L)

Practice Problems

1. Write a function that takes a number, n, and a flat list of atoms, L, and returns the n^{th} atom in the flat list.
2. Write a function that takes two flat lists of numbers, L1 and L2, and creates a new list containing the sums of the corresponding elements.
3. Write a function that takes an atom, a, and a flat list of atoms, L, and replaces the first 'b in the list with a.

Recursively Defined Data

- One can take a recursive grammar definition of data and use the grammar to design functions that operate on data of that form
- Consider the following definition of a flat list of atoms

  <f-list> → '() | (<atom> . <f-list>) // empty or cons cell

  This structure defines the structure of a function operating on a flat list.

Examples

1. Write a function that determines if a flat list of atoms, L, contains no numbers.
2. Write a function that substitutes a 1 for the first occurrence of a 3 in a flat list of atoms, L.

Tree Recursion on lists

- Consider the following definition of a list

  <list> → ‘() | (<atom> . <list>) | (<list> . <list>)

- Use this definition to write a function that takes an atom, a, and a list, M, and determines if a is in M.
- Write a function that takes a list of numbers, M, and returns the sum of the numbers.

Practice Problems

- Write a function that takes an atom, a, and a list of atoms, M, and removes a from the M.
- Write a function that takes an atom, a, and a list, M and substitutes all the occurrences of ‘b with a.

Local Definitions

- To create a variable with local scope inside of an expression, use the local keyword

  (local (<def>*)
  <exp>)

  - Where each <def> is a variable or function defintion.
- This defines the variables and functions to have scope within <def>* and <exp> only.

Examples

```
(define (h L)
    (local ( (define a (first L))
             (define d (rest L)))
      (if (eq? a 1)
          (cons 2 d)
          (cons a (cons 2 d)))))
```

```
(define (g L)
    (local
      ( (define (third L)
          (first (rest (rest L)))) )
      (if (eq? (first L) 2)
            (+ (third L) (third L))
            (* (third L) (third L))
      )))
```

Example

```
(define (count a L)
   (local ( (define (count-help a L n)
               (cond
                  [(null? L)     n]
                  [(eq? a (first L))
                        (count-help a (rest L) (add1 n))]
                  [else (count-help a (rest L)  n)])))
     (count-help a L 0))
)
```


Practice Problem

- Finish the following function that is to compute the length of a flat list. Use local to define a helper function that takes an extra parameter in which the current length of the flat list is stored.

```
(define (length L)
     (local … ))
```


1st-class Functions

- In Scheme functions can
  - have no name
  - be passed as arguments to functions
  - can have a return value that is a function
- In a sense functions are data just as integers are.
- The following defines a function with no name.

  (lambda (<var>*) <exp>)

- No names

  (define (add2 x) (+ x 2))
  (add2 5)

  can be written as

  ((lambda (x) (+ x 2))
  5)

Functions as Parameters

- The Scheme function map has two parameters: a function having one parameter, f, and a list, L. It then applies f to each member of L and returns a list of the results.

  (map (lambda (x) (+ x 1)) '(3 4 5 6)) → '(4 5 6 7)

Functions as Return Values

- A function can be returned by another function

```
(define (compose f g)
  (lambda (x)
    (f (g x))))
```

To what is k bound below?

```
(define k (compose add1 sub1))
```


Functional Abstraction

- Two or more functions can have their common structures abstracted and parameterized to allow them to be made specific
- Example
  1. (* x y) may be defined as adding x to x, y times.
  2. (expt x y) may be defined as multiplying x by x, y times

  These two functions will be quite similar

```
(define (times x y)
  (if (zero? y)
    0
    (+ x (times x (sub1 y)))))

(define (expt x y)
  (if (zero? y)
    1
    (* x (expt x (sub1 y)))))
```


Example

```
(define (abstract-fn f base)
  (lambda (x y)
      (if (zero? y)
         base
         (f x ((abstract-fn f base)
                 x (sub1 y))))))

(define times (abstract-fn + 0))
(define expt (abstract-fn * 1))
```


Example

- Create a functional abstraction of the following 2 functions.

```
(define (rem-first a L)
  (cond
    [(null?L) '()]
    [(eq? a (first L)) (rest L)]
    [ else (cons (first L)
            (rem-first a
              (rest L)))]))

(define (add1-first n L)
  (cond
    [(null? L) '()]
    [(eq? n (first L))
      (cons (add1 (first L))
            (rest L))]
    [else (cons (first L)
            (add1-first n
              (rest L)))]))
```

Practice Problems

1. Define the function compose3 that creates a function that composes three functions of one variable. Use compose to define compose3.
2. Functionally abstract the functions to the right.

```
(define (add-pairs P)
   (cond
    [(null? P) '()]
    [else (cons
              (+ (first P)
                 (first (rest P)))
           (add-pairs (cddr P)))]
   ))
(define (mul-pairs P)
  (cond
    [(null? P) '()]
    [else (cons
              (* (first P)
                 (first (rest P)))
           (mul-pairs (cddr P))))]
   ))
```


Practice Problem

- Create a functional abstraction for the following

```
(define (filter a L)
 (cond
   [(null? L) '()]
   [(eq? a (first L)) (cons (first L) (filter a (rest L)))]
   [else (filter a (rest L))]))

(define (remove a L)
 (cond
   [(null? L) '()]
   [(eq? a (first L)) (remove a (rest L))]
   [else (cons (first L) (remove a (rest L)))]))
```


Data with Functions

- In Scheme, basic data types can be represented as functions. First we look at Booleans

```
(define (true x y) x)
(define (false x y) y)

(define (if test then els)
   (test then els))

(if true 1 2) →
(if false 1 2) →
```


Lists as Functions

```
(define (cons a d)
   (lambda (g)
    (g a d)))
(define (car L)
   (L (lambda (a d) a)))
(define (cdr L)
   (L (lambda (a d) d)))

(car (cons 1 '())) →
(cdr (cons 1 '())) →
```

Laziness

- In a lazy language, arguments to functions are not evaluated until they are used.
  - Why is this useful?
    - unnecessary code
    - short-circuited code
    - infinite streams
- What does the following definition produce in lazy Scheme?

```
(define s-list (cons 1 s-list))
(define ones (cons 1 ones))
```


Lazy Evaluation

- How can we tell if a language is lazy?
  - What will a Scheme interpreter return when evaluating the following expressions?

```
((lambda (x) 3) (first empty))

((lambda (y) 1) ((lambda (x) (x x)) (lambda (x) (x x))))
```

  - If Scheme were lazy, the answers are

    3

    1

Builtin Lazy Functions

- How do we examine an infinite list?

```
(define (take n L)
  (if (or (zero? n) (empty? L))
      empty
      (cons (first L) (take (sub1 n) (rest L)))))
```

- (take 5 ones) returns

  (cons 1 (delay ...))

  Why?

Forcing Evaluation

- Lazy Scheme has routines that force evaluation: ! and !!
  - ! applies to single values
  - !! applies to lists (or recursive structures)

  (!! (take 5 ones)) returns

  (1 1 1 1 1)

Builtin Lazy Functions

- take and infinite lists are so common in lazy languages that lazy Scheme implements them
  - (cycle 1) is an infinite list of 1s
  - (!! (take 5 (cycle 1)) returns

    (1 1 1 1 1)

Building Non-repeating Lists

- Consider the following function

```
(define (zipOp f L1 L2)
  (if (or (empty? L1) (empty? L2))
      empty
      (cons (f (first L1) (first L2))
            (zipOp f (rest L1) (rest L2)))))
```

  This function allows us to combine lists.

  (!! (zipOp + ‘(1 1 1 1) ‘(1 2 3 4)) ➔ ‘(2 3 4 5)

  In Scheme zipOp is called map and takes a function and an arbitrary number of lists as input. The function must have the same arity as the number of lists.
- Map can be used to construct infinite lists that are non-repeating.

Examples

- Construct the infinite list of positive integers.
- Construct the infinite list of fibonacci numbers.

```
(define integers (cons 1 (map add1 integers)))

(define fib (cons 0 (cons 1 (map + fib (rest fib)))))
```


Practice Problem

- Construct the infinite list of factorials (i.e., (0! 1! 2! …).

```
(define factorial (cons 1 (map *  factorial integers)))
```

Lazy Evaluation

- Where is lazy evaluation used?
  - To avoid computation that may only rarely need to be done.
  - yes |rm –r /classes/cs4121