Where Scheme Fits In

- Derived from Lisp around 1975 by Sussman and Steele.
- One of the major changes was replacement of Lisp's dynamic scoping by static scoping.
- Small, elegant and simple language.
- Both functional and imperative features. To start with, we ignore imperative features.

Where Scheme Fits In Continued

- Prefix notation.
- Used in practical projects. For example, it is used in implementing GnuCash, the Guile extension language, and as the basis for DSSSL stylesheet language.

Functional Programming Essentials

- All computation is done by evaluating functions.
- Functions are first-class ... i.e., they may be passed to and from other functions, stored in data-structures, etc.
- No destructive assignment, i.e., something like `a = a + 1` impossible. Similar to Mathematics, a variable can have only a single value. Consequently, no loops!

- The lack of destructive assignment results in *referential transparency*. This means that each expression denotes a single value which cannot be changed by evaluating the expression or by allowing different parts of a program to share the expression.
- Referential transparency makes it possible to reason about programs using simple equality reasoning.

Non-Referentially Transparent C Code

```
int a = 1;

int f(x) {
   a = !a;
   return (a) ? x + 1 : x + 4;
}

f(3) => 7
f(3) => 4
f(3) => 7
f(3) => 4
...
```

Scheme: Lexical Issues

- Whitespace used to separate tokens, otherwise ignored.
- Comments extend from `;` to end-of-line.
- All whitespace chars are delimiters, as are:

  `(   )   ;   "   '   `` ` `` | [   ]   {   }`

- An identifier is a maximal sequence of non-delimiter chars that does not start with `#` or `,`. Examples: `x`, `symbol?`, `set!`, `<=>`.
- Case-insensitive: `if` and `IF` are equivalent.

Scheme: Syntactic Issues

- Scheme uses prefix expressions of the form: `(`*function arg1 arg2* . . . *argn*`)`.
- Each *argi* can be a primitive or a prefix expression.
- Basically a linearization of the AST.

Example: in programs/fact.scm:

```
(define (fact n)
  (if (<= n 0)
      1
      (* n (fact (- n 1)))))
```

Scheme Data: Overview

- Primitive data referred to as atom's. Primitives include boolean literals, number literals, character literals, string literals and symbols. The rest of this presentation largely ignores non-basic numbers, characters, strings.
- Constructed data using type constructors. The most basic constructor is the pair constructor using `cons`; can also have *vector*'s. This presentation concentrates on pair's.

Naming Conventions

A predicate is a procedure that always returns a boolean value. By convention, predicates usually have names that end in `?`. Example: `(number? x)` which returns `#t` (representing *true*) if its argument `x` is a number.

A mutation procedure is a procedure that alters a data structure. By convention, mutation procedures usually have names that end in `!`. Example: `(set! a 22)` destructively changes the value of `a` to `22`. This presentation ignores mutation except to illustrate the power of closures to encapsulate state.

Simple Scheme Data — Booleans

Constants `#t` for *true* and `#f` for *false*. Boolean contexts treat any value not equal to `#f` as *true*. That is, Scheme has a single falsey value, namely `#f`.

```
> (boolean? #f)   ;type-testing predicate.
#t
> (boolean? #t)
#t
> (boolean? 123)
#f
> (not #f)
#t
> (not 123)
#f
>
```

Simple Scheme Data — Numbers

Unlimited precision integers, rationals, reals, complex. Predicates `number?`, `complex?`, `real?`, `integer?`, `rational?`. Usual arith. operations; = for testing number equality, `eqv?` for general equality.

```
> (number? 2+3i)
#t
> (integer? 22/7)
#f
> (rational? 22/7)
#t
> (+ 1 2 3)
6
> (* 1 2 3)
6
```

```
> (/ 1 2 3)
1/6
> (/ 3)
1/3
> (max 22/7 3 0.6)
3.142857142857143
> (abs -1)
1
> (= 1 0)
#f
> (<= 3 3)
#t
>
```

Simple Scheme Data — Symbols

Normally, identifiers are used as *variable* names. However, if identifiers are quoted, then it is a literal representing a *symbol*.

```
> a
reference to undefined identifier: a
> (quote a)
a
> 'a
a
> '<=>
<=>
```

Simple Scheme Data — Symbols Continued

```
> (symbol? '!@#$)
#t
> (symbol? 12)
#f
> (symbol? '12)
#f
> (symbol? 'e+2)
#t
>
```

Compound Scheme Data – Dotted Pairs

- A dotted pair is a record structure with two fields called the *car* and *cdr* (for historical reasons), aka head and tail respectively.
- The pair with field `car` equal to x and field `cdr` equal to y is denoted as `(`x `.` y`)`.
- Pairs are constructed using the constructor `cons`.
- The two fields are accessed using the accessor functions `car` and `cdr` respectively.

Compound Scheme Data – Dotted Pairs Continued

```
> (cons 'a 'b)
(a . b)
> (pair? '(a . b))
#t
> (pair? 'a)
#f
> (car '(a . b))
a
> (cdr '(a . b))
b
>
```

S-Expressions

Dotted pairs along with primitives constitute S-expressions. Specifically, a S-expression (symbolic expression) is the smallest set of expressions such that:

- All Scheme primitives like booleans, numbers, characters, symbols, strings and `()` are S-expressions.
- If x and y are S-expressions, then so is the pair `(cons` x y`)` denoted also as `(` x `.` y `)`.

Basically, denotes binary trees with internal nodes `.` or `cons`. Can be used to denote any tree-like data-structure.

Compound Scheme Data – Lists

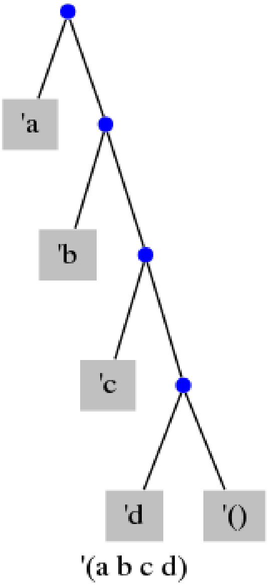
- The set of Scheme lists is defined as the smallest set L such that:
  - The empty list (denoted as `()`) is in L.
  - Any pair whose `cdr` field contains an element of L is also in L.
- Examples of lists:

```
()
(a . ())
(a . (b . ()))
```

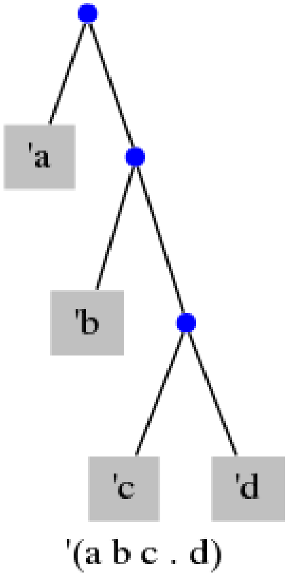
Compound Scheme Data – Lists

- A simpler notation uses `(a b c d)` to denote the list `(a . (b . (c . (d . ()))))`.
- A chain of pairs not ending in the empty list is called an *improper list* (it really is not a list). For example, the improper list `(a . (b . (c . d)))` can be simplified to `(a b (c . d))` but no further.
- Arbitrary Scheme programs can be represented as lists; i.e., a Scheme program is a Scheme datum! This means Scheme is a homoiconic language: the primary representation of a program is a data-structure within the language.

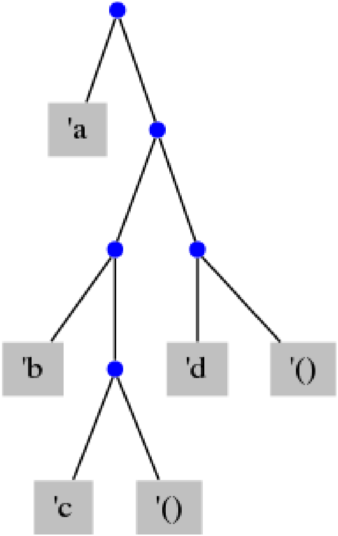
List Structure



Improper List Structure



Nested List Structure



'(a (b c) d)

Lists Functions

- Lists can be constructed using the constructor `list`. Example: `(list 1 2 3)` results in `(1 2 3)`.
- The predicate `list?` returns `#t` iff applied to a argument which is a proper list. Examples: `(list? '(1 2 3))` results in `#t`, whereas `(list? '(1 2 . 3))` results in `#f`.
- List elements can be accessed by index (starting at 0) using `list-ref`. Example: `(list-ref '(a b c) 1)` results in `b`.
- `list-tail` returns the tail of a list starting at a specified index. Example: `(list-tail '(a b c) 1)` returns `(b c)`.
- The predicate `null?` recognizes the empty list. Examples: `(null? '())` returns `#t` whereas `(null? '(1))` returns `#f`.

Lists Functions Continued

- `length` returns the length of a list. Example: `(length '(a b))` returns `2`.
- `append` can be used to append multiple lists. Example: `(append '(a b) '(1 2) '() '(c))` returns `(a b 1 2 c)`
- `member` can be used to check whether an element is a member of a list. Examples: `(member 'b '(a b c))` returns `(b c)`, which is interpreted as *true* within a boolean context; `(member 1 '(a b c))` returns `#f`.
- `assoc` is used for searching association lists which are lists of pairs, where the `car` of each pair is regarded as a key. If found, then the return value is the first pair with matching key, else `#f`. Example: `(assoc 'b '((a 1) (b 2) (c 3)))` returns `(b 2)`.

List Examples

```
> '(a . (b . (c . d)))
(a b c . d)
> (list? '(a b))
#t
> (list? '(a (b . c)))
#t
> (list? '(a . (b . c)))
#f
> (length '(a (b . c)))
2
> (length '(a . (b . c)))
length: expects argument of type <proper list>;
```

List Examples Continued

```
> (append '(a b c) '(1 2 3) '(x y z))
(a b c 1 2 3 x y z)
> (member 'b '(a b c))
(b c)
> (member 'x '(a b c))
#f
>
```

Contractions for Repeated `car`/`cdr`'s

- Can use a fixed-size list to implement a record by using a particular element as a particular field.
- An example employee record would be `(` *NAME SSN GENDER POSITION* `)`.
- Could have accessor functions: *NAME*: `(car` *EMPLOYEE*`)` and *POSITION*: `(car (cdr (cdr (cdr` *EMPLOYEE*`))))`.
- Can abbreviate *POSITION*: `(cadddr` *EMPLOYEE*`)`.
- Most Lisp's allow reasonable number of combinations of `a`'s and `d`'s like `cadr`, `cddr`, `cadar`, etc.

Functions

- A function is specified by a lambda expression of the form
  `(lambda` *Params Body*`)`
  where *Params* is usually a list of identifiers specifying the formal parameters of the function and *Body* is a S-expression (which will typically contain free occurrences of the formal parameters) giving the body of the function.
- A function is given a name by assigning it to a global variable using `define`.
  ```
  (define add1 (lambda (x) (+ x 1)))
  ```
- Function definition can also be abbreviated to not use an explicit `lambda`:
  ```
  (define (add1 x) (+ x 1))
  ```

Function Calls

- When a function is called, all its actual parameters are first evaluated, then its body is evaluated with each free occurrence of a formal in the body replaced by the actual.

```
(add1 (* 3 2)) =>
((lambda (x) (+ x 1)) (* 3 2)) =>
((lambda (x) (+ x 1)) 6) =>
(+ 6 1) =>
7
```

Special Forms

- A `lambda` expression looks like a function application of the *function* `lambda`, as does `define`.
- However, when a `lambda` or `define` expression is evaluated, the parameters are not evaluated.
- Such forms with different evaluation rules are referred to as *special forms*.
- Other built-in special forms include `if`, `when`, `unless`, `cond`, `and`, `or`, etc.

Condition Checking

- `(if` *Cond TrueExp FalseExp*`)` : A special form which first evaluates only *Cond*. If *Cond* evaluates to `#f`, the result is the evaluation of *FalseExp*; otherwise it is the evaluation of *TrueExp*.
- `(when` *Cond Exp*`)`, `(unless` *Cond Exp*`)` : Use if one of the branches of the `if` is empty.
- A multi-way `if` written as `(cond [`*Cond1 Exp1*`] [`*Cond2 Exp2*`] ... [`else *ElseExp*`])`. Each condition-expression pair is referred to as a `cond`-clause.
- Syntactically, multiple expressions are allowed within `when` and `unless` expressions and within a `cond`-clause; this is useful when the expressions have side-effects.

Condition Checking Examples

```
> (if '() 'a 'b)
a
> (if #f 'a 'b)
b
> (if (> 2 1) 'a 'b)
a
> (if (> 2 1) 'a (/ 1 0))
a
> (if (> 1 2) 'a (/ 1 0))
/: division by zero
> (unless (> 1 2) 'a)
a
> (when (> 1 2) 'a)
> (cond ((> 1 2) 'a) ((< 2 1) 'b) (else 'c))
c
>
```

Example — Factorial

In programs/fact.scm:

```scheme
(define (fact n)
  (if (<= n 0)
      1
      (* n (fact (- n 1)))))
```

Factorial Log

```
> (load "fact.scm")
> (fact 4)
24
> (fact -4)
1
> (fact 20)
2432902008176640000
> (fact 100)
933262154439441526816992388562667
004907159682643816214685929638952
175999932299156089414639761565182
862536979208272237582511852109168
64000000000000000000000000
>
```

In programs/my-length.scm:

```
;;Return length of a list.
(define (my-length list)
  (if (pair? list)
      (+ 1 (my-length (cdr list)))
      0))
```

List Length Log

```
> (load "my-length.scm")
> (my-length '())
0
> (my-length '(a b (1 2) (a b c) d))
5
> (my-length 'a)
0
> (my-length '(a . b))
1
> (my-length '(a b (1 2) (a b c) d . e))
5
>
```

Structural Recursion

Structural recursion is related to the proof-method of structural induction. When a data-structure is defined recursively, related functions can be written using cases corresponding to the cases in the recursive definition.

- A list is either *empty* or is a *pair* consisting of some *head* and some *tail* which is a list. Hence define function with two cases for *empty* and *pair*. In the former case, return function value for *empty* list; in the latter case, make recursive call for *tail* and combine returned result with *head* as return value of function.
- Recall that a non-negative integer is either 0 or the successor of a non-negative integer.

Structural Recursion Continued

- A non-empty array is either a 1-element array or it is a 1-element array followed by a non-empty array; basing a search function on this recursive definition leads to linear search.
- A non-empty array is either a 1-element array or it is a sequence of two arrays of length that differ at most by 1; basing a search function on this recursive definition leads to binary search.

In programs/err-length.scm:

```
;;List length with error checking.
(define (err-length list)
  (cond ((null? list) 0)
        ((pair? list)
         (if (integer? (err-length (cdr list)))
             (+ 1 (err-length (cdr list)))
             (err-length (cdr list))))
        (else "error")))
```

List Length with Error Checking Log

```
> (load "err-length.scm")
> (err-length '())
0
> (err-length '(a (b c) d))
3
> (err-length '(a (b c)  . d))
"error"
> (err-length 'a)
"error"
>
```

Using let

- Allows naming of intermediate results and avoid repeated evaluation.
- Not destructive assignment: just names some value for scope consisting of body of `let`.
- Example:

```
> (let ((fact4 (fact 4))
        (fact5 (fact 5)))
       (+ (* fact4 fact4)
          (* fact5 fact5)))
14976
>
```

Sequential let

`let*` allows values of variables previously defined within the same `let*` to be used in subsequent definitions.

```
> (let ((x 1) (y x)) (+ x y))
reference to undefined identifier: x
> (let* ((x 1) (y x)) (+ x y))
2
>
```

Recursive let

`letrec` allows values of variables previously defined within the same `letrec` to be used in earlier definitions.

```
> (let* ([even
          (lambda (x)
            (or (equal? x 0) (odd (- x 1))))]
         [odd
          (lambda (x)
            (and (not (equal? x 0))
                 (even (- x 1))))])
    (even 2))
odd: undefined;
 cannot reference undefined identifier
  ...
```

Recursive let Continued

```
> (letrec ([even
            (lambda (x) (or (equal? x 0)
                            (odd (- x 1))))]
          [odd
            (lambda (x)
              (and (not (equal? x 0))
                   (even (- x 1))))])
    (even 2))
#t
```

Example — List Length with Error Checking Revisited

In programs/let-length.scm:

```
(define (let-length list)
  (cond ((null? list) 0)
        ((pair? list)
         (let ((cdr-length
                (let-length (cdr list))))
           (if (integer? cdr-length)
               (+ 1 cdr-length)
               cdr-length)))
        (else "error")))
```

List Length with Error Checking Revisited Log

```
> (load "let-length.scm")
> (let-length '())
0
> (let-length '(a (b . c) d))
3
> (let-length '(a (b  c) . d))
"error"
> (let-length 0)
"error"
>
```

Example — Appending Two Lists

In programs/my-append.scm:

```
(define (my-append list1 list2)
  (if (null? list1)
      list2
      (cons (car list1)
            (my-append (cdr list1) list2))))
```

Appending Two Lists Log

```
> (load "my-append.scm")
> (my-append '(a b c) '(1 2 3))
(a b c 1 2 3)
> (my-append () ())
()
> (my-append () '(1 2))
(1 2)
> (my-append () '(1 2 . 3))
(1 2 . 3)
> (my-append '(a . b) '(1 2 . 3))
car: expects argument of type <pair>; given b
>
```

Example — Reversing a List

In programs/my-reverse.scm:

```
(define (my-reverse ls)
  (if (null? ls)
      '()
      (append (my-reverse (cdr ls))
              (list (car ls)))))
```

Reversing a List Log

```
> (load "my-reverse.scm")
> (my-reverse '(1 2 3))
(3 2 1)
> (my-reverse '(a))
(a)
> (my-reverse '())
()
> (my-reverse 'a)
cdr: expects argument of type <pair>; given a
> (my-reverse '(a . b))
cdr: expects argument of type <pair>; given b
>
```

Example — Reversing a List Revisited

Above my-reverse is $O(n^2)$ where n is the number of elements in the list. Can avoid $O(n^2)$ behavior by using an accumulator: In programs/lin-reverse.scm

```
(define (lin-reverse list)
  (aux-reverse '() list))

(define (aux-reverse acc ls)
  (if (null? ls)
      acc
      (aux-reverse (cons (car ls) acc)
                   (cdr ls))))
```

Reversing a List Revisited Log

```
> (load "lin-reverse.scm")
> (lin-reverse '(1 2 3))
(3 2 1)
> (lin-reverse '())
()
> (lin-reverse '(1))
(1)
> (lin-reverse '(1 (2 3 4) 5))
(5 (2 3 4) 1)
>
```

Accumulating Parameters

- A primary function is often implemented as a wrapper which simply calls an auxiliary function with additional accumulating parameters. For example, `reverse` is a wrapper around the auxiliary function `aux-reverse`.
- The accumulating parameter is given some initial value when the wrapper calls the auxiliary function. For example, when `reverse` calls `aux-reverse`, it is called with 2 parameters: an accumulating parameter initialized to `()` and the original list.

Accumulating Parameters Continued

- As the auxiliary function recurses, the accumulating parameter for the recursive call is updated (non-destructively, since the parameter for the recursive call is different from the incoming parameter). For example, the recursive call to `aux-reverse` is made with the accumulating parameter set to the `cons` of the `car` of the incoming list being reversed and the incoming accumulating parameter.

- When the auxiliary function recurses, it's return value is simply the return value of the recursive call. For example, the recursive case for `aux-reverse` simply returns the return value of the recursive call.
- When the auxiliary function terminates its recursion, the value of the accumulating parameter is returned as the result. For example, the base case for `aux-reverse` simply returns the value of the accumulating parameter `acc`.

Fibonacci Function

Recursive function from programs/rec-fib.scm:

```
(define (rec-fib n)
  (if (< n 2)
      n
      (+ (rec-fib (- n 1))
         (rec-fib (- n 2)))))
```

Fibonacci Function Log

```
> (load "rec-fib.scm")
> (rec-fib 5)
5
> (rec-fib 10)
55
> (rec-fib 20)
6765
>
```

Fibonacci Function in C

Recursive and iterative functions from fib.c:

```
static int rec_fib(int n)
{
  return
    (n < 2) ? n
            : rec_fib(n - 1) + rec_fib(n - 2);
}
```

Fibonacci Function in C Continued

```c
static int iter_fib(int n)
{
  if (n < 2) {
    return n;
  }
  else {
    int acc0 = 0;
    int acc1 = 1;
    int i;
    for (i = 2; i <= n; i++) {
      int t = acc0; acc0 = acc1; acc1 += t;
    }
    return acc1;
  }
}
```

Accumulator Fibonacci in Scheme

Use additional arguments `acc0`, `acc1` and `i` as in C function.
In programs/iter-fib.scm:

```
(define (iter-fib n)
  (letrec
    ([iter-fib-aux
      (lambda (acc0 acc1 i n)
        (if (> i n)
            acc1
            (iter-fib-aux acc1 (+ acc0 acc1)
                               (+ i 1) n)))])
    (if (< n 2)
        n
        (iter-fib-aux 0 1 2 n))))
```

Accumulator Fibonacci in Scheme: Log

```
> (load "iter-fib.scm")
> (iter-fib 5)
5
> (iter-fib 10)
55
> (iter-fib 20)
6765
>
```

Counting All Occurrences of Non-Pairs in a List

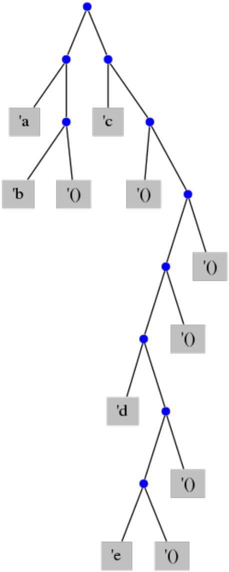
programs/count-non-pairs.scm:

```
(define (count-non-pairs ls)
  (if (not (pair? ls))
      1
      (+ (count-non-pairs (car ls))
         (count-non-pairs (cdr ls)))))
```

Counting All Occurrences of Non-Pairs in a List Log

```
> (load "count-non-pairs.scm")
> (count-non-pairs 'a)
1
> (count-non-pairs '())
1
> (count-non-pairs '(() () () ()))
5
> (count-non-pairs '(1 (2 . 3) 4))
5
> (count-non-pairs '((a b) c () ((d (e)))))
11
```

List Structure for Last Example



'((a b) c () ((d (e)))))

Flattening a List

programs/my-flatten.scm:

```
(define (my-flatten ls)
  (cond
   ((null? ls) '())
   ((pair? (car ls))
    (append (my-flatten (car ls))
            (my-flatten (cdr ls))))
   (else (cons (car ls)
               (my-flatten (cdr ls))))))
```

Flattening a List Log

```
> (load "my-flatten.scm")
> (my-flatten '(a (b) (c d ()) e ((f) g)))
(a b c d () e f g)
> (my-flatten '( () ((())) ((()) ())))
(() () () ())
> (my-flatten 'a)
car: expects argument of type <pair>; given a
> (my-flatten '(a . b))
car: expects argument of type <pair>; given b
>
```

Tail Recursion

A function is tail-recursive if the **absolutely** last thing it does before returning is calling itself.

- In conventional programming languages, recursion leads to heavy use of stack space.
- Scheme guarantees that if a function is tail-recursive, then the recursive call does not use additional stack space. This is referred to as tail-recursion optimization.

Note: `fact` is not tail-recursive, because recursive-call return value must be multiplied by `n` before return. `iter-fib-aux` is tail-recursive.

Tail Recursion in C

A tail-recursive function

```
int f(params) {
  if (baseCase(params)) {
    return g(params); /* non-recursive */
  }
  else {
    return f(newParams);
  }
}
```

Tail Recursion in C Continued

Tail recursive function is replaced by

```
int f(params) {
loop:
  if (baseCase(params)) {
    return g(params); /* non-recursive */
  }
  else {
    params = newParams;
    goto loop;
  }
}
```

Tail Recursion in C Continued

Previous function is equivalent to:

```c
int f(params) {
  while (!baseCase(params)) {
    params = newParams;
  }
  return g(params);  /* non-recursive */
}
```

Tail-Recursive Factorial in C

Consider iterative C factorial from programs/fact.c:

```
int fact(int n) { //
  int acc = 1;
  while (n > 1) {
    acc = acc * n;
    n = n - 1;
  }
  return acc;
}
```

Equivalent Scheme function in programs/iter-fact.scm. Note that `aux-fact` is tail-recursive, hence it is guaranteed to run in constant stack-space.

```
(define (iter-fact n)
  (letrec ([aux-fact
           (lambda (acc n)
             (if (> n 1)
                 (aux-fact (* acc n) (- n 1))
                acc))])
    (aux-fact 1 n)))
```

DFA Simulation

- A Deterministic Finite Automaton (DFA) consists of a set of states, a set of inputs and transitions from state to state based on the input.
- A distinguished state is the initial state.
- Some set of states are final states.
- A sequence of input symbols is accepted if starting in the initial state, the machine makes transitions on the input symbols and lands up in a final state.

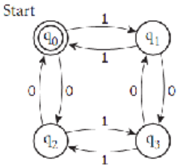
Scheme: DFA Representation

Represent DFA as a 3-element list `(` *Initial Transitions Finals*`)`.

Initial
: The initial state of the DFA.

Transitions
: A list of transitions, where each *transition* is represented as a 2-element list `((`*FromState Input*`)` *ToState*`)` representing the transition from state *FromState* on input symbol *Input* to state *ToState*.

Finals
: A list of final states.




```
(define zero-one-even-dfa
 '(q0                                                  ; start state
   (((q0 0) q2) ((q0 1) q1) ((q1 0) q3) ((q1 1) q0)   ; transition fn
    ((q2 0) q0) ((q2 1) q3) ((q3 0) q1) ((q3 1) q2))
   (q0)))                                              ; final states
```

Figure 10.2 DFA to accept all strings of zeros and ones containing an even number of each. At the bottom of the figure is a representation of the machine as a Scheme data structure, using the conventions of Figure 10.1.

DFA Example

DFA Simulation: Code

```
(define (simulate dfa input)                    ;
  (cons (car dfa)                               ;start state
        (if (null? input)
            (if (infinal? dfa) '(accept)
                '(reject))
            (simulate (move dfa (car input))
                      (cdr input)))))

(define (infinal? dfa)
    (memq (car dfa) (caddr dfa)))
```

DFA Simulation Code: Continued

```scheme
(define (move dfa symbol)                ;
  (let ((curstate (car dfa))
        (trans (cadr dfa))
        (finals (caddr dfa)))
    (list
     (if (eq? curstate 'error)
         'error
         (let
             ((pair (assoc (list curstate symbol)
                           trans)))
           (if pair (cadr pair) 'error)))
     trans
     finals)))
```

DFA Simulation: Log

```
(simulate
 '(q0                                          ;start state
   (((q0 0) q2) ((q0 1) q1)                    ;transitions
    ((q1 0) q3) ((q1 1) q0)
    ((q2 0) q0) ((q2 1) q3)
    ((q3 0) q1) ((q3 1) q2))
   (q0))                                       ;final state
 '(0 1 1 0 1))
```

```
=> (q0 q2 q3 q2 q0 q1 reject)
```

First Class Functions

- Scheme allows functions as arguments to other functions.
- Scheme allows functions to return functions.
- We often want to `map` each element of a list via some function.
- We also often want to reduce all the elements of a list to a single element via some function.

Map function in programs/my-map.scm:

```
(define (my-map f ls)
  (if (null? ls)
      ls
      (cons (f (car ls)) (my-map f (cdr ls)))))
```

Mapping a List Log

```
> (load "my-map.scm")
> (my-map add1 '(1 2 3))
(2 3 4)
> (my-map add1 ())
()
> (my-map (lambda (x) (* 3 x)) '(1 2 3))
(3 6 9)
> (my-map length '(() (a) (a b c (d e) f)))
(0 1 5)
>
```

Anonymous Mapping Functions

- Previous transparency illustrated the use of a anonymous function.

```
> (my-map (lambda (x) (* 3 x)) '(1 2 3))
(3 6 9)
```

- Anonymous functions avoid cluttering up global namespace with functions used only within some other function.

Local Mapping Functions

Alternatively, could use `let` to give a temporary name to the function.

```
> (let ((times3 (lambda (x) (* 3 x))))
       (my-map times3 '(1 2 3)))
(3 6 9)
> times3
reference to undefined identifier: times3
```

What if function is recursive? Consider mapping factorial as in programs/map-fact-let.scm:

```
(define (map-fact-let ls)
  (let ([fact1
          (lambda (n)
            (if (< n 1)
                1
                (* n (fact1 (- n 1)))))])
    (map fact1 ls)))
```

map-fact-let Log

```
> (load "map-fact-let.scm")
> (map-fact-let '(3 4 5))
fact1: undefined;
 cannot reference undefined identifier
>
```

Recursive let

Use letrec as in map-fact-letrec.scm:

```
(define (map-fact-letrec ls)
  (letrec
      ((fact
         (lambda (n)
           (if (< n 1)
               1
               (* n (fact (- n 1)))))))
    (map fact ls)))
```

Recursive let Log

```
> (load "map-fact-letrec.scm")
> (map-fact-letrec '(-1 0 1 2 3 5 6))
(1 1 1 2 6 120 720)
>
```

Reducing a List

- Consider summing the elements of a list. We can do that by accumulating a sum by applying + to each successive element and a accumulator.
- Consider multiplying the elements of a list. We can do that by accumulating a product by applying $*$ to each successive element and a accumulator.
- Define `(reduce '(a1 a2 ... aN) z f)` to be `(f a1 (f a2 ... (f aN z)))`

Scheme Reduce

Scheme definition in programs/reduce.scm:

```scheme
(define (reduce ls z f)
  (if (null? ls)
      z
      (f (car ls) (reduce (cdr ls) z f))))
```

```
> (load "reduce.scm")
> (reduce '(1 2 3 4 5) 0 +)
15
> (reduce '(1 2 3 4 5) 1 *)
120
```

Functions Which Return Functions

```
> (define add-n (lambda (n) (lambda (x) (+ n x))))
> ((add-n 4) 5)
9
> (define add-5 (add-n 5))
> add-5
#<procedure:STDIN::27>
> (add-5 6)
11
> ((add-n -4) 3)
-1
>
```

Destructive Assignment

`(set! v e)` sets the value of variable `v` to expression `e`. For example:

```
> (let ((f (lambda (x) (+ x 5))))
       (display (f 10)) (newline)
       (set! f (lambda (x) (* x 10)))
       (f 10))
15
100
>
```

A bank account:

```scheme
(define (account init-balance)
  (let ((balance init-balance))
    (let ((deposit
           (lambda (amount)
             (set! balance (+ amount balance))
            balance))
          (withdraw
           (lambda (amount)
             (when (>= balance amount)
                   (set! balance
                         (- balance amount)))
            balance)))
      (list deposit withdraw))))
```

Closures: Bank Account Continued

```scheme
(define (deposit account)   ;
  (car account))

(define (withdraw account)
  (cadr account))
```

Bank Account Log

```
> (load "balance.scm")
> (define a1 (account 100))
> (define a2 (account 50))
> ((deposit a1) 20)
120
> ((withdraw a1) 40)
80
> ((deposit a2) 80)
130
> ((withdraw a2) 20)
110
> ((deposit a1) 100)
180
>
```

References

- Text, Chapter 11, through 11.3.
- Harold Abelson, Gerald Jay Sussman with Julie Sussman, *Structure and Interpretation of Computer Programs*, MIT Press, 1984.
- R. Kent Dybvig, *The Scheme Programming Language*, Second Edition, Prentice-Hall, 1996.
- *Revised Report on the Algorithmic Language Scheme*, 1998.
- Dorai Sitaram, *Teach Yourself Scheme in FixNum Days*.
- George Spring and Daniel P. Friedman, *Scheme and the Art of Programming*, McGraw-Hill, 1989.
- Learn racket in Y Minutes