Example Scheme Function: member

- member takes an atom and a simple list; returns #T if the atom is in the list; #F otherwise

```scheme
(define (member atm lis)
  (cond
   ((null? lis) #f)
   ((eq? atm (car lis)) #t)
   (else (member atm (cdr lis))))
)
```

Example Scheme Function: equalsimp

- equalsimp takes two simple lists as parameters; returns #T if the two simple lists are equal; #F otherwise

```scheme
(define (equalsimp lis1 lis2)
  (cond
   ((null? lis1) (null? lis2))
   ((null? lis2) #f)
   ((eq? (car lis1) (car lis2))
    (equalsimp (cdr lis1) (cdr lis2)))
   (else #f))
)
```

Example Scheme Function: equal

- equal takes two general lists as parameters; returns #T if the two lists are equal; #F otherwise

```scheme
(define (equal lis1 lis2)
  (cond
   ((not (list? lis1)) (eq? lis1 lis2))
   ((not (list? lis2)) #f)
   ((null? lis1) (null? lis2))
   ((null? lis2) #f)
   ((equal (car lis1) (car lis2))
    (equal (cdr lis1) (cdr lis2)))
   (else #f))
)
```
Example Scheme Function: LET

- General form:

  \[
  (\text{LET} \ (\text{name}_1 \ \text{expression}_1) \ (\text{name}_2 \ \text{expression}_2) \ldots \ (\text{name}_n \ \text{expression}_n) \)
  \]
  \[
  \text{body}
  \]

- Evaluate all expressions, then bind the values to the names; evaluate the body

Example

\[
(\text{DEFINE} \ (\text{quadratic_roots} \ a \ b \ c)
\]
\[
(\text{LET} \ (\text{root_part_over_2a} \ (/ \ (\text{SQRT} (- (* b b) (* 4 a c)))(* 2 a)))
(\text{minus_b_over_2a} \ (/ (- 0 b) (* 2 a)))
)
(\text{DISPLAY} (+ minus_b_over_2a root_part_over_2a))
(\text{NEWLINE})
(\text{DISPLAY} (- minus_b_over_2a root_part_over_2a))
)\]

Tail Recursion in Scheme

- Definition: A function is tail recursive if its recursive call is the last operation in the function
- A tail recursive function can be automatically converted by a compiler to use iteration, making it faster
- Scheme language definition requires that Scheme language systems convert all tail recursive functions to use iteration

Tail Recursion in Scheme (cont'd)

- Example of rewriting a function to make it tail recursive, using a helper function

  Original:
  \[
  (\text{DEFINE} \ (\text{factorial} \ n)
  (\text{IF} (= n 0)
  1
  (* n \ (\text{factorial} \ (- n 1))))
  )
  \]

  Tail recursive:
  \[
  (\text{DEFINE} \ (\text{facthelper} \ n \ \text{factpartial})
  (\text{IF} (= n 0)
  \text{factpartial}
  \text{facthelper}((- n 1) (* n \text{factpartial})))
  )
  (\text{DEFINE} \ (\text{factorial} \ n)
  (\text{facthelper} n 1))
  )\]
Scheme Functional Forms

• Composition
  – The previous examples have used it
    – (CDR (CDR '(A B C))) yields (C)

• Apply to All: one form in Scheme is mapcar
  – Applies the given function to all elements of the given list;
    (DEFINE (mapcar fun lis)
      (COND
        ((NULL? lis) '())
        (ELSE (CONS (fun (CAR lis))
                   (mapcar fun (CDR lis))))))

Functions That Build Code

• It is possible in Scheme to define a function that builds Scheme code and requests its interpretation
• This is possible because the interpreter is a user-available function, EVAL

Adding a List of Numbers

(DEFINE (adder lis)
  (COND
    ((NULL? lis) 0)
    (ELSE (EVAL (CONS '+' lis)))))

• The parameter is a list of numbers to be added; adder inserts a + operator and evaluates the resulting list
  – Use CONS to insert the atom + into the list of numbers.
  – Be sure that + is quoted to prevent evaluation
  – Submit the new list to EVAL for evaluation

COMMON LISP

• A combination of many of the features of the popular dialects of LISP around in the early 1980s
• A large and complex language: the opposite of Scheme
• Features include:
  – records
  – arrays
  – complex numbers
  – character strings
  – powerful I/O capabilities
  – packages with access control
  – iterative control statements
ML

- A static-scoped functional PL with syntax that is closer to Pascal than to LISP
- Uses type declarations, but also does type inferencing to determine the types of undeclared variables
- It is strongly typed (whereas Scheme is essentially typeless) and has no type coercions
- Includes exception handling and a module facility for implementing abstract data types
- Includes lists and list operations

ML Specifics

- Function declaration form:
  \[
  \text{fun } \text{name} \left( \text{parameters} \right) = \text{body};
  \]
  e.g., \[
  \text{fun cube} \left( x : \text{int} \right) = x * x * x;
  \]
  - The type could be attached to return value, as in \[
  \text{fun cube} \left( x : \text{int} \right) : \text{int} = x * x * x;
  \]
  - With no type specified, it would default to \text{int} (the default for numeric values)
  - User-defined overloaded functions are not allowed, so if we wanted a \text{cube} function for real parameters, it would need to have a different name
  - There are no type coercions in ML

ML Specifics (cont’d)

- ML selection
  \[
  \text{if expression then then_expression else else_expression}
  \]
  where the first expression must evaluate to a Boolean value

- Pattern matching is used to allow a function to operate on different parameter forms
  \[
  \text{fun fact(0) = 1}
  | \quad \text{fact(n : int) : int = n * fact(n - 1)}
  \]

ML Specifics (cont’d)

- Lists
  - Literal lists are specified in brackets: \textbf{e.g.}, \[3, 5, 7\]
  - [] is the empty list
  - CONS is the binary infix operator, ::
    \[4 :: [3, 5, 7], \text{which evaluates to} [4, 3, 5, 7]\]
  - CAR is the unary operator \text{hd}
  - CDR is the unary operator \text{tl}

  \[
  \text{fun length([]) = 0}
  | \quad \text{length(h :: t) = 1 + length(t)};
  \]

  \[
  \text{fun append([], lis2) = lis2}
  | \quad \text{append(h :: t, lis2) = h :: append(t, lis2)};
  \]
ML Specifics (cont’d)

- The `val` statement binds a name to a value (similar to `DEFINE` in Scheme)
- e.g., `val distance = time * speed;`
- As is the case with `DEFINE`, `val` is nothing like an assignment statement in an imperative language

Haskell

- Similar to ML (syntax, static scoped, strongly typed, type inferencing, pattern matching)
- Different from ML (and most other FPL’s) in that it is purely functional (e.g., no variables, no assignment statements, and no side effects of any kind)
- Syntax differences from ML
  
  ```haskell
  fact 0 = 1
  fact n = n * fact (n - 1)
  
  fib 0 = 1
  fib 1 = 1
  fib (n + 2) = fib (n + 1) + fib n
  ```

Function Definitions with Different Parameter Ranges

```haskell
fact n
| n == 0 = 1
| n > 0 = n * fact(n - 1)
```

```haskell
sub n
| n < 10    = 0
| n > 100   = 2
| otherwise = 1
```

```haskell
square x = x * x
(Works for any numeric type of x)
```

Lists

- List notation: Put elements in brackets
  
  e.g., directions = "north", "south", "east", "west"
  
- Length: #
  
  e.g., #directions is 4
  
- Arithmetic series with the .. operator
  
  e.g., [2..10] is [2, 4, 6, 8, 10]
  
- Catenation is with ++
  
  e.g., [1, 3] ++ [5, 7] results in [1, 3, 5, 7]
  
- CONS, CAR, CDR via the colon operator (as in Prolog)
  
  e.g., 1:[3, 5, 7] results in [1, 3, 5, 7]
Factorial Revisited

\[
\text{product} \; [] = 1 \\
\text{product} \; (a:x) = a \times \text{product} \; x \\
\text{fact} \; n = \text{product} \; [1..n]
\]

List Comprehension

- Set notation
- List of the squares of the first 20 positive integers:
  \[
  [n \times n \mid n \leftarrow [1..20]]
  \]
- All of the factors of its given parameter:
  \[
  \text{factors} \; n = [i \mid i \leftarrow [1..n \div 2], \quad n \mod i == 0]
  \]

Quicksort

\[
\text{sort} \; [] = [] \\
\text{sort} \; (a:x) = \\
\quad \text{sort} \; [b \mid b \leftarrow x; b \leq a] \; ++ \\
\quad [a] \; ++ \\
\quad \text{sort} \; [b \mid b \leftarrow x; b > a]
\]

Lazy Evaluation

- A language is \textit{strict} if it requires all actual parameters to be fully evaluated
- A language is \textit{nonstrict} if it does not have the strict requirement
- Nonstrict languages are \textbf{more efficient} and allow some interesting capabilities: \textit{for example: infinite lists}
- \textbf{Lazy evaluation:} only compute those values that are necessary
- \textbf{Examples:}
  - Positive numbers: \textit{positives} = \{0..\}
  - Determining if 16 is a square number
    \[
    \text{member} \; [] \; b = \text{False} \\
    \text{member}(a:x) \; b = (a == b) \; || \; \text{member} \; x \; b \\
    \text{squares} = [n \times n \mid n \leftarrow [0..]] \\
    \text{member} \; \text{squares} \; 16
    \]
Member Revisited

- The member function could be written as:
  ```
  member [] b = False
  member(a:x) = (a == b) || member x b
  ```
- However, this would only work if the parameter to
  squares was a perfect square; if not, it will keep
  generating them forever. The following version will
  always work:
  ```
  member2 (m:x) n
  | m < n = member2 x n
  | m == n = True
  | otherwise = False
  ```

Comparing FPL’s and IPL’s

- **Imperative Languages:**
  - Efficient execution
  - Complex semantics
  - Complex syntax
  - Concurrency is programmer designed
- **Functional Languages:**
  - Inefficient execution
  - Simple semantics
  - Simple syntax
  - Programs can automatically be made concurrent

Applications of Functional Languages

- **APL** is used for throw-away programs
- **LISP** is used for artificial intelligence
  - Knowledge representation
  - Machine learning
  - Natural language processing
  - Modeling of speech and vision
- **Scheme** is used to teach introductory
  programming at some universities

Summary

- Functional programming languages use function application,
  conditional expressions, recursion, and functional forms to control
  program execution instead of imperative features such as variables
  and assignments
- LISP began as a purely functional language and later included
  imperative features
- Scheme is a relatively simple dialect of LISP that uses static
  scoping exclusively
- COMMON LISP is a large LISP-based language
- ML is a static-scoped and strongly typed functional language which
  includes type inference, exception handling, and a variety of data
  structures and abstract data types
- Haskell is a lazy functional language supporting infinite lists and set
  comprehension.
- Purely functional languages have advantages over imperative
  alternatives, but their lower efficiency on existing machine
  architectures has prevented them from enjoying widespread use