Functional Programming

- Different paradigm from imperative or OO
  - Everything is a value
  - Focus on expressions – no statements
  - Functions treated like data
  - No side effects
  - No iteration – recursion used instead
  - Mimics mathematical functions
  - Application of functions is primary activity

Functional Programming

- Advantages
  - Easier to analyze
  - Increased flexibility since functions can "build" functions
  - Simple and consistent semantics
  - Solve complex programming tasks easily
  - Orthogonal design – fewer surprises

- Disadvantages
  - Efficiency
  - Some type errors can't be detected until runtime
  - Often interpreted, not compiled
Pure Functional Programming

- Mathematical definition of function:
  - Mapping from values in **domain** to unique values in **range**
  - E.g., $y = f(x)$
  - Value of function application depends only on argument values (**referential transparency**)
  - Note: I/O functions cannot be referentially transparent
- No assignment in pure functional languages
  - No "variables", just values
  - No loops possible since no counters
- Value semantics rather than storage semantics
- Functions themselves are first class values

Scheme

- Simple dialect of LISP
  - Fixed "bug" of dynamic scope in LISP to static scope
  - LISP – 1959, Scheme – 1975
- Everything (data or program) is an **expression**
  - Expressions can be **atoms**, or
  - Expressions can be **lists**
- Memory automatically managed – no direct access
- Scheme is an untyped language
  - But has strong type checking (at runtime)
Scheme Basics

- Case insensitive
- Constant atom values built in
  - Numbers – integer or floating point
  - Strings – double quotes like C/C++
  - Literal characters, e.g., \a
  - Booleans - #t and #f
- Identifiers can be letters, digits, underscores, other special characters
  - But definitely not parentheses!
- Interpreter
  - Interactive – type in expression, get response
  - Interpreter is a read-evaluate-write loop

Scheme Syntax

- Simple enough to write as BNF:

  \[
  \text{expression} \rightarrow \text{atom} \mid \text{list} \\
  \text{atom} \rightarrow \text{number} \mid \text{string} \mid \text{identifier} \mid \text{char} \mid \text{bool} \\
  \text{list} \rightarrow ( \ \text{expr-list} \ ) \\
  \text{expr-list} \rightarrow \text{expression} \ \text{expr-list} \mid \text{expression}
  \]

- Note that the only terminals are ( and )
  - Operations are identifiers with value pre-defined as function
  - Comments begin with semi-colon
Scheme Expression Examples

42 — a number
"hello" — a string
#t — the Boolean value "true"
#\a — the character 'a'
(2.1 2.2 3.1) — a list of numbers
a — an identifier
hello — another identifier
(+ 2 3) — a list consisting of the identifier "+" and two numbers
(* (+ 2 3) (/ 6 2)) — a list consisting of the identifier "*" followed by two lists

Evaluation Rules

- Constant atoms evaluate to themselves
- Identifiers are looked up (in current environment) and replaced by the value found
- Lists are evaluated recursively by evaluating each element in the list
  - First element must evaluate to a function
  - Function is then applied to rest of the list
Example Expression Evaluation

> 42
42
> "hello"
"hello"
> #t
true
> #\a
#\a
> (2.1 2.2 2.3)
function call: expected a defined name or a primitive operation
name after an open parenthesis, but found a number
> a
reference to undefined identifier: a
> (+ 2 3)
5
> (* (+ 2 3) (/ 6 2))
15
> 

Scheme "Variables"

- No declarations
  - Not really a variable since no assignment (in pure functional)
  - Just a value associated with a name – a symbol
  - Type inferred from value (only lists, atoms, and functions)
- Predefined function to bind a symbol to a value
  - (define id expression)
- Example:
  > (define a 7)
  > (define b 13)
  > (define c (+ a b))
  > (* c a)
  140
Predefined Functions

- Arithmetic: +  -  *  /  sqrt
- Boolean: and  or  not
- Comparison: <  <=  >  >=  =
- Predicates: null?  list?
- Output: display  newline  write
- Choice: if
  - Form is (if expr1 expr2 expr3)
  - Example: (if (< a b) a b)

Defining Functions

- Special form of DEFINE for defining values that are functions
  - Name of function and names of formal parameters
  - Body of function
- Example
  
  > (define (f x) (* x 2))
  > (f 3)
  6
  > (define (g x y) (+ x y))
  > (g 4 7)
  11
  > (g 5 (f 4))
  13
Recursive Functions

- Functions can be defined with self
- Example
  > (define (csum n)
        (if (<= n 0) 0
           (+ n (csum (- n 1)))
        )
  )
  > (csum 5)
     15
  > (csum 100)
     5050

Suppressing Evaluation

- Sometimes we want to prevent or delay evaluation
- Predefined function quote (abbreviated with ')
- Example
  > (k 2 3)
     . reference to undefined identifier: k
  > (define x '(k 2 3))
  > (x)
     . procedure application: expected procedure, given: (k 2 3) (no arguments)
  > (define k g)
  > (eval x)
     5
Handling Lists

- Special functions to operate on lists
  - **CAR** - gives first element
  - **CDR** - gives list with first element removed
- Example
  ```scheme
  > (define l '("abc" "def" "ghi" "jkl") )
  > (car l)
  "abc"
  > (cdr l)
  ("def" "ghi" "jkl")
  > (car (cdr l))
  "def"
  > (list? (car l))
  #f
  > (list? (cdr l))
  #t
  ```

CAR and CDR

- Names come from original LISP implementation on IBM 704
  - **CAR** - Contents of Address Register
  - **CDR** - Contents of Decrement Register
- Functions can be combined for shorthand
  - **CADR** – CAR of the CDR
  - **CADDR** – CAR of the CDR of the CDR
  - etc.
- Example
  ```scheme
  > (define l2 '(((1 2) 3 (4 (5 6 7)))))
  > (caar l2)
  1
  > (cdar l2)
  (2)
  > (caaddr l2)
  4
  ```
Recursive List Functions

- Function to give n'th element of a list
- Example

```scheme
> (define (elementAt i l)
   (if (= i 1) (car l)
    (elementAt (- i 1) (cdr l))))

> (elementAt 1 l)
"abc"
> (elementAt 3 l)
"ghi"
> (elementAt 5 l)
. car: expects argument of type <pair>; given ()
```

Constructing Lists

- Predefined function for making a list from atoms:
  ```scheme
  > (list 1 2 3 4)
  (1 2 3 4)
  ```

- BUT the actual structure of a list is a pair (the car and cdr)

- **CONS** is used to create such a pair
Constructing Lists

> (cons 1 '())
(1)

> (cons 1 (cons 2 '()) )
(1 2)

> (cons 1 2)
(1 . 2)

> (cons (cons 1 (cons 2 '())) (cons 3 (cons (cons 4 (cons 5 (cons 6 '()))) '())))
((1 2) 3 (4 (5 6)))

Fall 2012  
CIS 425 – Scheme
**Multiway Selection**

- Generalization of if-then-else selection
  - List of conditions and corresponding expressions
  - Evaluated in sequence
  - Default condition (else)

- Example

```
> (define (lookup v L) (cond ((null? L) L)
  ((equal? v (caar L)) (cadar L))
  (else (lookup v (cdr L))))
> (define slist '((("a" 1) ("b" 2) ("c" 3) ("d" 4)))
> (lookup "a" slist)
1
> (lookup 3 slist)

()  
> (lookup "c" slist)
3
```

**More List Functions**

- Append one list to another

```
> (define (append L M)
  (if (null? L) M
  (cons (car L) (append (cdr L) M))))
> (append '(1 2 3) '(4 5))
(1 2 3 4 5)
```

- Reverse a list

```
> (define (rev L)
  (if (null? L) L
  (cons (rev (cdr L)) (car L))))
> (rev '(1 2 3 4))
(((()) . 4) . 3) . 2) . 1)  
Oops!
```
Recursion on Lists

- Solution: use append and list
  ```scheme
  > (define (rev1 L)
      (if (null? L) L
        (append (rev1 (cdr L)) (list (car L))))
  > (rev1 '(1 2 3 4))
  (4 3 2 1)
  ```

- But recursion could be very deep
  - Cannot return until everything reversed

Tail Recursion

- Arrange for **tail recursion**
  - Recursive call is last action in function
  - Interpreter can optimize nested function calls into loop
  - Often use accumulator

- Example
  ```scheme
  > (define (rev2 L A)
      (if (null? L) A
        (rev2 (cdr L) (cons (car L) A))))
  > (rev2 '(1 2 3 4) '())
  (4 3 2 1)
  ```
Anonymous Functions

- Functions are first class objects
  - A function is just another value
  - Can be computed and returned by a function
- An anonymous function is a value formed by LAMBDA
  - `(lambda (x y) (+ x y))` is a function
  - `((lambda (x y) (+ x y)) 2 3)` applies the function
- DEFINE has a special form that is just shorthand for defining an anonymous function
  - `(define add (lambda (x y) (+ x y)))`
  - `(add 2 3)`

Higher Order Functions

- Write a function to compose two functions
  - i.e., two functions as arguments, return value is a new function

  > `(define (compose f1 f2) (lambda (x) (f1 (f2 x))))`
  > `(define (f x) (+ x 7))`
  > `(define (g x) (* x x))`
  > `((compose f g) 4)`
  > `23`
  > `((compose g f) 4)`
  > `121`
Another Special Form

- Definition of "local variables"
  - Like DEFINE
  - List of pairs – binding name and expression
  - Final expression is value of LET, may use bindings

```
(let
  ( (x 42)
    (y 17)
  )
  (+ x y)
)
```

- Bind x to 42
- Bind y to 17
- This is the value of the let expression

Sequenced Bindings

- Special form of LET
  - Same syntax, but bindings may be used in subsequent bindings

```
> (define (cylindervol r h)
  (let*
    ((pi 3.1415926535) (area (* r r pi)))
    (* h area)))
> (cylindervol 3 10)
282.743338815
```

- Use pi from first binding in second
- If LET was used instead of LET*, then pi would be undefined in the area binding
Recursive Bindings

- Form of LET to permit recursive definitions

```scheme
(define (month i)
  (letrec
    ( (ML '("Jan" "Feb" "Mar" "Apr"))
      (elt (lambda (i L) (if (= i 1) (car L)
                        (elt (- i 1) (cdr L))))
        )
      (elt i ML)
    )
  )
)
```

Need definition of elt

If LET was used instead of LETREC, then elt would be undefined in the lambda

Functions and objects

- Functions can be used to model objects and classes in Scheme.

Consider the simple C++ class:

```cpp
class BankAccount {
  double balance;

public:
  BankAccount(double b) : balance(b) { }
  void deposit(double amt) { balance += amt; }
  void withdraw(double amt) { balance -= amt; }
  double getBalance() const { return balance; }
};
```
Functions and objects

- This can be modeled in Scheme as:

```scheme
(define (BankAccount balance)
  (define (getBalance) balance)
  (define (deposit amt)
    (BankAccount (+ balance amt)))
  (define (withdraw amt)
    (BankAccount (- balance amt)))
  (lambda (message)
    (cond
     ((eq? message 'getbalance) getBalance)
     ((eq? message 'deposit) deposit)
     ((eq? message 'withdraw) withdraw)
     (else (error "unknown message:" message)))))
```

Functions and objects

- This code can be used as follows:

```scheme
> (define acct1 (BankAccount 50))
> (define acct2 (BankAccount 100))
> ((acct1 'getbalance)) 50
> ((acct2 'getbalance)) 100
> (define acct1 (acct1 'withdraw) 40)
> (define acct2 (acct2 'deposit) 50)
> ((acct1 'getbalance)) 10
> ((acct2 'getbalance)) 150
> ((acct1 'setbalance) 100)
. unknown message setbalance
```