Functional Programming

In this set of notes you will learn about:

- Why functional programming?
- Historical origins of functional programming
- Functional programming in Scheme
- Higher-order functions

Note: this set of notes covers Chapter 11 Sections 11.1 to 11.2, except Sections 11.2.2, 11.2.4, and 11.2.5 which you are not required to study

Why Functional Programming?

- Imperative programming languages are more widely used
- However, many commercial applications exist for functional programming:
  - Symbolic data manipulation

Relationship of Functional Programming to Other Material of This Course

- Functional programming languages depend heavily on polymorphism (Section 3.5)
- Several are dynamically scoped (Section 3.3.2)
- All employ recursion (Section 6.6) for repetitive execution
- All functional programming language implementations use garbage collection (Section 3.2.3) to reclaim memory
- Functional programs have no side effects and all expressions are referentially transparent (Sections 6.1.2 and 6.3)
- Subroutine closures (Section 3.4.1)
- First-class function values (Section 3.4.2)

Historical Origins of Functional Programming

- Church’s thesis:
  - All models of computation are equally powerful and can compute any function
- Turing’s model of computation: Turing machine
  - Reading/writing of values on an infinite tape by a finite state machine
- Church’s model of computation: lambda calculus
  - Inspired functional programming as a concrete implementation of lambda calculus
- Computability theory
  - A program can be viewed as a constructive proof that some mathematical object with a desired property exists
  - A function is a mapping from inputs to output objects and computes output objects from appropriate inputs
  - For example, the proposition that every pair of nonnegative integers (the inputs) has a greatest common divisor (the output object) has a constructive proof implemented by Euclid’s algorithm written as a "function"

\[
gcd(a,b) = \begin{cases} 
  a & \text{if } a = b \\
  gcd(a-b,b) & \text{if } a > b \\
  gcd(a,b-a) & \text{if } b > a
\end{cases}
\]
**Functional Programming**
- *Functional programming* defines the outputs of a program as mathematical function of the inputs with no notion of internal state (no side effects)
  - Example: pure functional programming languages: Miranda, Haskell, and Sisal
- Non-pure functional programming languages include imperative features that affect global state (e.g. through destructive assignments to global variables)
  - Example: Lisp, Scheme, and ML
- Useful features often missing in imperative programming languages:
  - *First-class function values*: the ability of functions to return newly constructed functions
  - *Higher-order functions*: functions that take other functions as input parameters or return functions
  - *Polymorphism*: the ability to write functions that operate on more than one type of data
  - *Aggregate constructs for constructing structured objects*: the ability to specify a structured object in-line, e.g. a complete list or record value
  - *Garbage collection*

**Lisp**
- Lisp (LISt Processing language) was the original functional language
- Lisp and its dialects are most widely used
- Simple and elegant design of Lisp:
  - *Homogeneity of programs and data*: a program in Lisp is a list and can be manipulated in Lisp
  - *Self-definition*: a Lisp interpreter can be written in Lisp
  - *Interactive*: interaction with user through "read-eval-print" loop

**Scheme**
- Scheme is a popular Lisp dialect
- Like Lisp, scheme adopts Cambridge Polish notation for expressions
  - A simple expression is an atom, e.g. a number, string, or identifier name
  - An expression is written as a list starting with the function name or operator followed by its arguments which are expressions:
    `(function arg1 arg2 arg3 ...)
- *"Read-eval-print" loop* provides user interaction: an expression is read, evaluated by evaluating the arguments first and then the function/operator is called after which the result is printed
  - Input: 9
    - Output: 9
  - Input: (+ 3 4)
    - Output: 7
  - Input: (* 2 3) 1
    - Output: 7
- User can load a program from a file with the load function
  - (load "my_scheme_program")
  - The file name should use the .scm extension

**Data Structures**
- The only data structures in Lisp and Scheme are *atoms* and *lists*
- Atoms:
  - Number, e.g. 7
  - String, e.g. "abc"
  - Identifier name (variable), e.g. x
  - Boolean values true #t and false #f
  - Symbol: a symbol is a quoted identifier that is not evaluated, e.g. 'y
    - Input: a
    - Output: Error: unbound variable a
  - Input: 'a
    - Output: a
- Lists:
  - To distinguish list data structures from expressions a quote (') is used to quote the lists for input to the Scheme interpreter:
    - (elt1 elt2 elt3 ...)
    - Input: (3 4 5)
      - Output: (3 4 5)
    - Input: ('a 6 (x y) "s")
      - Output: ('a 6 (x y) "s")
    - Input: ('a (+ 3 4))
      - Output: ('a (+ 3 4))
    - Input: ()
      - Output: ()
- Empty list () is also identical to false #f in Scheme

Note: You can run the Scheme interpreter and try the examples in these notes by executing the scheme command on xi. To exit Scheme, type (exit)
List Operations

- **car** returns the head of a list
  - Input: `(car '(2 3 4))`
  - Output: 2
- **cdr** (pronounced "coulter") returns the rest of a list (list without the head)
  - Input: `(cdr '(2 3 4))`
  - Output: `(3 4)`
- **cons** joins a head to the rest of a list
  - Input: `(cons 2 '(3 4))`
  - Output: `(2 3 4)`
- **More examples:**
  - Input: `(car '(2))`
  - Output: 2
  - Input: `(car '())`
  - Output: Error
  - Input: `(cdr '(2 3))`
  - Output: `(3)`
  - Input: `(cdr (cdr '(2 3 4)))`
  - Output: `(4)`
  - Input: `(cdr '(2))`
  - Output: `()`
  - Input: `(cons 2 '())`
  - Output: `(2)`

Type Checking

- The type of an expression is determined at run time
- Most functions check types dynamically to make sure that the arguments are of the proper type
- **Type predicate functions:**
  - `(boolean? x)` ; is `x` a Boolean?
  - `(char? x)` ; is `x` a character?
  - `(string? x)` ; is `x` a string?
  - `(symbol? x)` ; is `x` a symbol?
  - `(number? x)` ; is `x` a number?
  - `(list? x)` ; is `x` a list?
  - `(pair? x)` ; is `x` a non-empty list?
  - `(null? x)` ; is `x` an empty list?

If-Then-Else

- **Special forms** resemble functions but have special evaluation rules
- A **conditional expression** in Scheme is written using the `if` special form:
  - `(if condition thenexpr elseexpr)`
  - Input: `(if #t 1 2)`
  - Output: 1
  - Input: `(if #f 1 "a")`
  - Output: "a"
  - Input: `(if (string? "s") (+ 1 2) 4)`
  - Output: 3
  - Input: `(if (> 1 2) "yes" "no")`
  - Output: "no"
- A more general if-then-else can be written using the `cond` special form:
  - `(cond (condition1 value1) (condition2 value2) ...)`
  - where the `condition value pairs` is a list of `(cond value)` and the condition of the last pair can be `else` to return a default value
  - Input: `(cond ((< 1 2) 1) ((>= 1 2) 2))`
  - Output: 1
  - Input: `(cond ((< 2 1) 1) ((= 2 1) 2) (else 3))`
  - Output: 3

Testing

- **eq?** tests whether its arguments refer to the same object
  - Input: `(eq? 'a 'a)`
  - Output: `#t`
  - Input: `(eq? '(a b) '(a b))`
  - Output: `()` (false: the lists are not stored at the same location in memory!)
- **equal?** tests whether its arguments have the same recursive structure
  - Input: `(equal? 'a 'a)`
  - Output: `#t`
  - Input: `(equal? '(a b) '(a b))`
  - Output: `#t`
  - To test numerical values, use `=`, `<`, `>`, `<=`, `>=`, `even?`, `odd?`, `zero?`
- **member** tests membership of an element in a list and returns the rest of the list that starts with the first occurrence of the element, or returns false
  - Input: `(member 'y '(x (3 y) z))`
  - Output: `(y z)`
  - Input: `(member 'y '(x (3 y) z))`
  - Output: `()`
Lambda Expressions

- A Scheme lambda expression is a nameless function specified with the lambda special form:
  \( \text{lambda formalparameters functionbody} \)
  where the formal parameters are the function inputs and the function body is an expression that is the resulting value of the function.
- Examples:
  - \( \text{(lambda (x) (* x x))} \); is a squaring function: \( x \to x^2 \)
  - \( \text{(lambda (a b) (sqrt (+ (* a a) (* b b)))} \); is a function:
    \( a \ b \to \sqrt{a^2+b^2} \)

  A function is applied in an expression by assigning the evaluated actual parameter(s) to the formal parameters and returning the evaluated function body.
  - The form of a function call in an expression is:
    \( \text{function arg1 arg2 arg3...} \)
    where function can be a lambda expression.
  - Input: \( \text{(lambda (x) (* x x)) 3} \)
  - Output: 9
  - That is, \( x=3 \) in \( (* x x) \) which is evaluated and returned.

Functions

- A function is globally defined using the define special form:
  \( \text{define name function} \)
- For example:
  \( \text{(define sqr (lambda (x) (* x x))} \)
  defines function sqr
  - Input: \( \text{(sqr 3)} \)
  - Output: 9
  - Input: \( \text{(sqr (sqr 3))} \)
  - Output: 81
  - \( \text{(define hypot (lambda (a b) (sqrt (+ (* a a) (* b b))))} \)
  defines function hypot
  - Input: \( \text{(hypot 3 4)} \)
  - Output: 5

Example Recursive Functions on Lists

- Sum the elements of a list:
  \( \text{(define sum (lambda (lst)) (if (null? lst) 0 (+ (car lst) (sum (cdr lst)))} \)

  Input: \( \text{(sum '(1 2 3))} \)
  Output: 6

- Check if element is in list:
  \( \text{(define in? (lambda (elt lst) (cond ((null? lst) #f) ; if list is empty, return false}
  ((= elt (car lst)) #t) ; if element is the head, return true
  (else (in? elt (cdr lst))))} \)

  Input: \( \text{(in? 2 '(1 2 3))} \)
  Output: #t

Bindings

- An expression can have local name-value bindings defined with the let special form:
  \( \text{(let listofnameandvaluepairs expression) \)
  where name and value pairs is a list of pairs (name value) and expression is returned in which each name is replaced with its value in the list.
  - Input:
  \( \text{(let ((a 3) (b 4) (hypot a b))} \)
  - Output: 5

  A name can be bound to a function in let
  - Input:
  \( \text{(let ((sqr (lambda (x) (* x x))) (y 3)) (sqr y))} \)
  - Output: 9
Recursive Bindings

- An expression can have local recursive function bindings defined with the letrec special form

(letrec listofnameandvaluepairs expression)

where name and value pairs is a list of pairs (name value) and expression is returned where each name is replaced with its value

- Input:
  (letrec ((fact (lambda (n)
      (if (= n 1)
        1
        (* n (fact (- n 1)))))
      )
    )
  )

- Output: 120

- This allows the local factorial function fact to refer to itself

I/O and Sequencing

- display prints a value
  - Input: (display "Hello World!"
  - Output: "Hello World!"
  - Input: (display (+ 2 3))
  - Output: 5

- newline advances to a new line
  - Input: (newline)

- read returns a value from standard input

- begin sequences a series of expressions
  - Example:
    (begin
      (display "Hello World!"
    )

- Example:
  (let ((x 1)
    (y (read))
    (plus +))
  )

- (newline)

Loops

- do takes a list of name-init-update triples, a termination test with final value, and a loop body

(do listoftriples condition body)

- Example:
  (do ((i 0 (+ i 1)))
    ((>= i 10) "done")
    (display i)
    (newline)
  )

Since everything is an expression in Scheme, a loop must return a value which in this case is the string "done"

Higher-Order Functions

- A function is called a higher-order function (also called a functional form) if it takes a function as an argument or returns a function as a result

- Scheme has several higher-order functions, for example:

  - apply takes a function and a list and applies the function with the elements of the list as arguments
    - Input: (apply + '(3 4))
    - Output: 7
    - Input: (apply (lambda (x) (* x x)) '(3))
    - Output: 9

  - map takes a function and a list and returns a list after applying the function to each element of the list
    - Input: (map odd? '(1 2 3 4))
    - Output: (#t #t #f)
    - Input: (map (lambda (x) (+ x x)) '(1 2 3 4))
    - Output: (1 4 9 16)

- Here is a function that applies a function to an argument twice:
  - (define twice
    (lambda (f n) (f (f n)))
  )
  - Input: (twice sqrt 81)
  - Output: 3
Non-Pure Constructs: Assignments

- Assignments are considered bad in functional programming because they can change the global state of the program and possibly influence function outcomes
- \texttt{set!} assigns a value to a variable, for example:
  - (define a 0)
  - ...
  - (set! a 1) ;; overwrite a with 1
  - ...
  - (let ((a 0))
      (begin
        ...
        (set! a (+ a 1)) ;; increment a by 1
      )
    )

- \texttt{set-car!} overwrites the head of a list
- \texttt{set-cdr!} overwrites the tail (rest) of a list

Examples

- Recursive factorial function:
  - (define fact
    (lambda (n)
      (if (zero? n) 1 (* n (fact (- n 1))))
    )
  )

- Iterative factorial function:
  - (define iterfact
    (lambda (n)
      (do ((i 1 (+ i 1))
           (f 1 (* f i))
           )
           (> i n) f)
              ;; note: loop body is omitted
      )
    )

Examples of List Functions

- \texttt{fill}:
  - (define fill
    (lambda (num elt)
      (cond
        ((= 0 num) '())
        (else (cons elt (fill (- num 1) elt))))
    )
  )

- \texttt{between}:
  - (define between
    (lambda (start end)
      (if (> start end)
        '()
        (cons start (between (+ start 1) end))
      )
    )
  )

- \texttt{zip}:
  - (define zip
    (lambda (lst1 lst2)
      (cond
        ((null? lst1) '())
        ((null? lst2) '())
        (else (cons (list (car lst1) (car lst2)) (zip (cdr lst1) (cdr lst2)))))
    )
  )

- \texttt{take}:
  - (define take
    (lambda (num lis)
      (cond
        ((= num 0) '())
        (else (cons (car lis) (take (- num 1) (cdr lis))))
      )
    )
  )

Examples of Higher-Order Functions

- Reduce a list by applying a binary operator to all elements (i.e. \(elt1 + elt2 + elt3 + \ldots\)):
  - (define reduce
    (lambda (op lst)
      (if (null? (cdr lst))
        (op (car lst) (reduce op (cdr lst)))
        )
      )
    )
  )

- Filter elements of a list for which a condition (a predicate function) returns true:
  - (define filter
    (lambda (op lst)
      (cond
        ((null? lst) '())
        ((op (car lst)) (cons (car lst) (filter op (cdr lst))))
        (else (filter op (cdr lst))))
      )
    )
  )

Input:
- \texttt{f(3 *a*)}:
  - \texttt{Output: ("a" "a" "a")}

Input:
- \texttt{between 1 10}:
  - \texttt{Output: (1 2 3 4 5 6 7 8 9 10)}

Input:
- \texttt{zip '(1 2 3) '(a b c)}:
  - \texttt{Output: ((1 a) (2 b) (3 c))}

Input:
- \texttt{take 3 '(a b c d e f)}:
  - \texttt{Output: (a b c)}