2. Functional Programming

Overview

- Why functional programming?
- Historical origins of functional programming
- Functional programming today
- Concepts of functional programming
- A crash course on programming in Scheme

---

### Why Functional Programming?

- Functional programming is a different programming paradigm.
- Imperative programming languages are more widely used.
  - Integrated software development environments for procedural and object-oriented programming languages are "industrial strength."
- However, many (commercial) applications exist for functional programming:
  - Symbolic data manipulation
  - Natural language processing
  - Artificial intelligence
  - Automatic theorem proving and computer algebra
  - Algorithmic optimization of programs written in pure functional languages

Note: this set of notes covers Chapter 11 Sections 11.1 to 11.2. You are not required to study Sections 11.2.2, 11.2.4, and 11.2.5.
Why Functional Programming in This Course?

- A functional language will be used to illustrate a diversity of programming language concepts
- Functional programming languages are
  - Compiled and/or interpreted (Section 1.4)
  - Have simple syntax (Chapter 2)
  - Use garbage collection (Section 3.2.3) for memory management
  - Are statically scoped or dynamically scoped (Section 3.3)
  - Use higher-order functions and subroutine closures (Section 3.4.1)
  - Use first-class function values (Section 3.4.2)
  - Depend heavily on polymorphism (Section 3.5)
  - Employ recursion (Section 6.6) for repetitive execution
  - Programs have no side effects and all expressions are referentially transparent (Sections 6.1.2 and 6.3)

Origin 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
  - This 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"

\[
\hat{i} \quad a \quad \text{if } a = b \\
\hat{i} \quad \hat{i} \quad \text{gcd}(a,b) = \hat{i} \quad \text{gcd}(a-b,b) \quad \text{if } a > b \\
\hat{i} \quad \hat{i} \quad \text{gcd}(a,b-a) \quad \text{if } b > a
\]
Functional Programming Today

- Attractive model of computation
  - Absence of side effects makes expressions referentially transparent: the value of an expression depends solely on the function return values in it and not on evaluation order and/or values of global variables
  - A function can always be counted on to return the same results with the same input parameters
  - Dangling and/or uninitialized pointer references do not occur
- Significant improvements in theory and practice of functional programming have been made in recent years
  - Easier to debug and maintain programs
- Remaining obstacles to functional programming:
  - Social: most programmers are trained in imperative programming
  - Commercial: not many libraries, not very portable, and no integrated development environments for functional languages

Concepts of Functional Programming

- *Functional* programming defines the outputs of a program as a 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 with side effects that affect global state (e.g. through destructive assignments to global variables)
  - Example: Lisp, Scheme, and ML
- Useful features are found in functional languages that are often missing in imperative 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: 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 dialects are still the most widely used
- Simple and elegant design of Lisp:
  - Homogeneity of programs and data: a Lisp program is a list and can be manipulated in Lisp as a list
  - Self-definition: a Lisp interpreter can be written in Lisp
  - Interactive: interaction with user through "read-eval-print" loop

A Crash Course on Scheme

- Scheme is a popular Lisp dialect
- Lisp and Scheme adopt Cambridge Polish notation for expressions:
  - An expression is an atom, e.g. a number, string, or identifier name
  - An expression is a list whose first element is the function name (or operator) followed by the arguments which are expressions:
    `(function arg1 arg2 arg3 ...)
- The "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

Note: You can run the Scheme interpreter and try the examples in these notes by executing the scheme command. To exit Scheme, type `(exit). You can download an example Scheme program "Eliza".
### Scheme Data Structures

- The only data structures in Lisp and Scheme are *atoms* and *lists*.
- **Atoms** are:
  - Numbers, e.g. 7
  - Strings, e.g. "abc"
  - Identifier names (variables), e.g. x
  - Boolean values true #t and false #f
  - Symbols which are quoted identifiers which will not be evaluated, e.g. 'y
    - Input: a
    - Output: Error: unbound variable a
- **Lists**:
  - To distinguish list data structures from expressions that are written as lists, a quote (') is used to quote the list:
    - (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: ()
- Note: the empty list () is also identical to false #f in Scheme

### Primitive List Operations

- **car** returns the *head* (first element) of a list
  - Input: (car '(2 3 4))
  - Output: 2
- **cdr** (pronounced "coulder") returns the *tail* of a list (list without the head)
  - Input: (cdr '(2 3 4))
  - Output: (3 4)
- **cons** joins an element and a list to construct a new list
  - Input: (cons 2 '(3 4))
  - Output: (2 3 4)
- **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 only at run-time
- Functions need to check the types of their arguments explicitly
- 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:
  \[
  \text{if} \quad \text{condition} \quad \text{thenexpr} \quad \text{elseexpr}
  \]
  - Input: (if #t 1 2)
  - Output: 1
  - Input: (if #f 1 "a")
  - Output: "a"
  - Input: (if (>= 1 2) (+ 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:
  \[
  \text{cond} \quad \text{listofconditionvaluepairs}
  \]
  where the *condition value pairs* is a list of \((\text{cond} \ \text{value})\) pairs 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 two arguments refer to the same object in memory
  - **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 structure
  - **Input:** (equal? 'a 'a)
  - **Output:** #t
  - **Input:** (equal? '(a b) '(a b))
  - **Output:** #t
- **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 '("s" x 3 y z))
  - **Output:** (y z)
  - **Input:** (member 'y '(x (3 y) z))
  - **Output:** ()

## Lambda Abstraction

- A Scheme **lambda abstraction** 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:**
  - (lambda (x) (* x x)) ; is a squaring function: \( x \mapsto x^2 \)
  - (lambda (a b) (sqrt (+ (* a a) (* b b)))) ; is a function:
    \[
    (a b) \mapsto \sqrt{a^2 + b^2}
    \]
Lambda Application

- A lambda abstraction is applied 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 \text{function} can be a lambda abstraction.
- Example:
  - Input: \((\lambda (x) \times x)) 3\)
  - Output: 9
  - That is, \(x=3\) in \((\times x)\) which evaluates to 9

Defining Global Functions in Scheme

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

\[
\text{letrec \ listofnameandvaluepairs \ expression}
\]

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

- Input:
  \[
  \text{letrec ((fact (lambda (n) (if (= n 1) 1 (* n (fact (- n 1))))) ) (fact 5))}
  \]

- 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 (its value is the value of the last expression)
  - **Example:**
    ```scheme
    (begin
      (display "Hello World!"
      (newline)
    )
    
    (let ((x 1)
      (y (read))
      (plus +)
    )
      (begin
        (display (plus x y))
        (newline)
      )
    )
    ```

Loops

- **do** takes a list of name-init-update triples, a termination test with final value, and a loop body
  ```scheme
  (do listoftriples condition body)
  ```
  - **Example:**
    ```scheme
    (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 newly constructed function as a result.
- Scheme has several built-in 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
  - `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 ())
- 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.
- `set!` assigns to a variable a new value, for example:
  - (define a 0)
    ... (set! a 1); overwrite a with 1
    ...
  - (let ((a 0))
    (begin
      ...
      (set! a (+ a 1)); increment a by 1
      ...
    ))
- `set-car!` overwrites the head of a list.
- `set-cdr!` overwrites the tail (rest) of a list.
Scheme 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
        )
    )
  )

Example Recursive Functions on Lists

- Sum the elements of a list:
  (define sum
    (lambda (lst)
      (if (null? lst)
          0
          (+ (car lst) (sum (cdr lst))) ; add value of head and sum of rest of list
    )
  )

- Check if element is in list:
  (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))) ; keep searching rest of list
      )
    )
  )

Examples of List Functions

- (define fill
    (lambda (num elt)
      (cond
        ((= 0 num) '())
        (else (cons elt (fill (- num 1) elt)))
      )
    )
  )
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))
        (car lst)
        (op (car lst) (reduce op (cdr lst)))))
  )
  )
  )
  o Input: (reduce + '(1 2 3))
  o Output: 6

- 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 (cdrlst))))
        (else (filter op (cdr lst)))
      )
    )
    )
  )
  )
  o Input: (filter odd? '(1 2 3 4 5))
  o Output: (1 3 5)

- Input: (fill 3 "a")
  o Output: ("a" "a" "a")

- (define between
  (lambda (start end)
    (if (> start end)
      '
    (cons start (between (+ start 1) end)))
  )
  )
  )
  )
  o Input: (between 1 10)
  o Output: (1 2 3 4 5 6 7 8 9 10)

- (define zip
  (lambda (lst1 lst2)
    (cond
      ((null? lst1) '())
      ((null? lst2) '())
      (else (cons (list (car lst1) (car lst2)) (zip (cdr lst1) (cdr lst2))))
    )
  )
  )
  )
  )
  o Input: (zip '(1 2 3) '(a b c))
  o Output: ((1 a) (2 b) (3 c))

- (define take
  (lambda (num lis)
    (cond
      ((= num 0) '())
      (else (cons (car lis) (take (- num 1) (cdrlis)))))
  )
  )
  )
  )
  o Input: (take 3 '(a b c d e f))
  o Output: (a b c)