2. Functional Programming

Overview

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

Note: Study Chapter 11 Sections 11.1 to 11.2, except Sections 11.2.2, 11.2.4, and 11.2.5.

What is Functional Programming?

- Functional programming is a declarative programming paradigm
  - Computation is more implicit and functional call is the only form of explicit control
- But: imperative programming languages are more widely used
  - Integrated software development environments for procedural and object oriented programming languages are "industrial strength" and often include extensive libraries
- 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

Why Functional Programming in This Course?

- A functional language will be used to illustrate a diversity of the programming language concepts discussed in this course
- Functional programming languages are
  - Compiled and/or interpreted
  - Have simple syntax
  - Use garbage collection for memory management
  - Are statically scoped or dynamically scoped
  - Use higher-order functions and subroutine closures
  - Use first-class function values
  - Depend heavily on polymorphism
  - Employ recursion for repetitive execution
  - Programs have no side effects and all expressions are referentially transparent

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"

\[
\text{gcd}(a,b) = \begin{cases} 
  a & \text{if } a = b \\
  \text{gcd}(a-b,b) & \text{if } a > b \\
  \text{gcd}(a,b-a) & \text{if } b > a
\end{cases}
\]
### 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).
  - A pure function can always be counted on to return the same results for the same input parameters.
  - No assignments: dangling and/or uninitialized pointer references do not occur.
  - 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: the ability to specify a structured object in-line, e.g. a complete list or record value.
  - Garbage collection.

### 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.

### Lisp

- **Lisp** (LISP 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.

Note: You can run the Scheme interpreter and try the examples in these notes by executing the `scheme` command on the linprog stack (ssh linprog). To exit Scheme, type `(exit)`. You can download an example Scheme program "Eliza". More information on Scheme can be found at [http://www.swiss.ai.mit.edu/projects/scheme](http://www.swiss.ai.mit.edu/projects/scheme).
**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
    - Input: `'a`
    - Output: a

- **Lists**:
  - To distinguish list data structures from expressions that are written as lists, a quote (') is used to quote the list:
    - 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

**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:
  - (if condition thenexpr elseexpr)
    - Input: (if #t 1 2)
      - Output: 1
    - Input: (if #f 1 "a")
      - Output: "a"
    - Input: (if (> 1 2) "yes" "no")
      - Output: "no"
- A more general if-then-else can be written using the `cond` special form:
  - (cond condition value pairs)
    - where the condition value pairs is a list of (cond 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) (else 3))
        - Output: 3

**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: (cdr '(1))
    - Output: Error
  - Input: (cdr '(2 3))
    - Output: (3)
  - Input: (cdr (cdr '(2 3 4)))
    - Output: (4)
  - Input: (cdr '(2))
    - Output: ()
  - Input: (cons 2 '(1))
    - Output: (2)
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: `#t` (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`

- 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 '("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:
  - `(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\(^2\)
  - `(lambda (a b) (sqrt (+ (* a a) (* b b))))` ; is a function:
    - `(a b)\(\sqrt{a^2+b^2}\)`

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:
  - `(function arg1 arg2 arg3 ...)`
- where `function` can be a lambda abstraction.
- Example:
  - Input: `((lambda (x) (* x x)) 3)`
  - Output: `9` (That is, x=3 in \(* x x\) which evaluates to 9)

Defining Global Functions in Scheme

- A function is globally defined using the define special form:
  - `(define name function)`
- For example:
  - `(define sqr (lambda (x) (* x x)))`
  - defines function `sqr`
    - Input: `(sqr 3)`
    - Output: `9`
    - Input: `(sqr (sqr 3))`
    - Output: `81`
  - `(define hypot (lambda (a b) (sqrt (+ (* a a) (* b b))))`)
  - defines function `hypot`
    - Input: `(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 `(name namevalue)` and expression is returned in which each name is replaced with its value in the list.

  - Input:
    \[
    \text{(let ((a 3)}
    \text{(b 4)}
    \text{)}
    \text{(hypot a b)}
    \)
    \]

  - Output: `5`

- A name can be bound to a function in `let`

  - Input:
    \[
    \text{(let ((sqr \{lambda (x) (* x x)})}
    \text{(y 3)}
    \text{)}
    \text{(sqr y)}
    \)
    \]

  - Output: `9`
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
  - 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 #f #t ())
    - 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.
  - set! assigns to a variable a new value, for example:
    - (define a 0)
      ... (set! a 1) ; overwrite a with 1
    - (let ((a 1))
      ... (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))))))
    )
  - Input: (sum '(1 2 3))
  - Output: 6

Example Recursive Functions on Lists

- Sum the elements of a list:
  - (define sum
    (lambda (lst)
      (if (null? lst)
        0
        (+ (car lst) (sum (cdr lst)))))
      )
    )
  - Input: (sum '(1 2 3))
  - Output: 6
- Check if element is in list:
  - (define in?
    (lambda (elt lst)
      (cond
        ((null? lst) #f)
        ((= elt (car lst)) #t)
        (else (in? elt (cdr lst)))))
      )
    )
  - Input: (in? 2 '(1 2 3))
  - Output: #t

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 + ...):
  (define reduce
   (lambda (op lst)
     (if (null? (cdr lst))
         (car lst)
         (op (car lst) (reduce op (cdr lst)))))
   )
  )
  - Input: (reduce + '(1 2 3))
  - 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 (cdr lst))))
       (else (filter op (cdr lst))))
     )
   )
  - Input: (filter odd? '(1 2 3 4 5))
  - Output: (1 3 5)

Functional Programming Today

- Significant improvements in theory and practice of functional programming have been made in recent years
  - Strongly typed (with type inference)
  - Modular
  - Imperative language features that are automatically translated to functional constructs (e.g. loops by recursion)
  - Improved efficiency
- 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

Exercise 1: Explain the workings of a Turing Machine. Find information on the Web (for example in the Stanford Encyclopedia of Philosophy). Site your source of information.
Exercise 2: Rewrite the arithmetic expression 9 + 3 * 7 / (2 + 4) + 1 in Scheme.
Exercise 3: Which Scheme construct(s) will cause a Scheme program to depart from a purely functional programming model?
Exercise 4: Explain the difference between a functional and a special form.
Exercise 5: Using nested lists, show how you can implement a binary tree data structure (nodes carry values and have optional left and right subtrees).
Exercise 6: Why is Scheme homoiconic?