LECTURE 16

Functional Programming

WHAT IS FUNCTIONAL PROGRAMMING?

Functional programming defines the outputs of a program as a mathematical function of the inputs.

Functional programming is a *declarative* programming style.

- Focus is on *what* the computer should do, not *how* it should be done.
- Declarative programming is “the act of programming in languages that conform to the mental model of the developer rather than the operational model of the machine”.
- Like giving an address to your house instead of outlining the directions.

WHAT IS FUNCTIONAL PROGRAMMING?

- A program in a functional programming language is basically a function call which likely makes use of other functions.
  - The basic building block of such programs is the function.
  - Functions produce results (based on arguments), but do not change any memory state.
  - In other words, pure functions do not have any *side effects*.
- Everything is an expression, not an assignment.
  - In an imperative language, a program is a sequence of assignments.
  - Assignments produce results by changing the memory state.

PROS AND CONS OF FUNCTIONAL PROGRAMMING

- Pro: flow of computation is declarative, i.e. more implicit.
  - The program does not specify the sequence of actions for the computation.
  - This gives the language implementer more choices for optimizations.
- Pro: promotes building more complex functions from other functions that serve as building blocks (component reuse and modularization).
- Pro: behavior of functions defined by the values of input arguments only (no side-effects via global/static variables).
  - This makes it easier to reason about the properties of a program.
- Con: Not everything can be easily or effectively expressed by stateless (pure) functions.
- Con: programmers tend to prefer imperative programming constructs such as statement sequencing, while functional languages emphasize function composition.

CONCEPTS OF FUNCTIONAL PROGRAMMING

- A pure function maps the inputs (arguments) to the outputs with no notion of internal state (no side effects).
- *Pure functional* programming languages only allow pure functions in a program.
  - A pure function can be counted on to return the same output each time we invoke it with the same input parameter values.
  - Can be emulated in traditional languages: expressions behave like pure functions; many routines behave like pure functions.
  - No global (statically allocated) variables, no explicit assignments.
  - Example pure functional programming languages: Miranda, Haskell, and Sisal.
- *Non-pure functional* programming languages include "imperative features" that cause side effects.
  - Example: Lisp, Scheme, and ML.

FUNCTIONAL LANGUAGE CONSTRUCTS

- Building blocks are functions.
- No statement sequencing, only function composition.
- No variable assignments.
  - But: can use local "variables" to hold a value assigned once.
- No loops.
  - Recursion.
  - List comprehensions in Miranda and Haskell.
  - But: "do-loops" in Scheme.
- Conditional flow with if-then-else or argument patterns.
  - E.g `(if exp then_exp else_exp)`
- Functional languages are statically (Haskell) or dynamically (Lisp) typed.

Haskell examples:

```
gcd a b
  | a == b = a
  | a >  b = gcd (a-b) b
  | a <  b = gcd a (b-a)

fac 0 = 1
fac n = n * fac (n-1)

member x []    = false
member x (y:xs)
      | x == y = true
      | x <> y = member x xs
```

IMPACT OF FUNCTIONAL LANGUAGES ON LANGUAGE DESIGN

Useful features are found in functional languages that are often missing in procedural languages or have been adopted by modern 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 such as a complete list or record value.
- Garbage collection.

FUNCTIONAL PROGRAMMING TODAY

Significant improvements in theory and practice of functional programming have been made in recent years.

- Modular.
- Sugaring: 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 and aren't used to thinking in terms of function composition.
- Commercial: not many libraries, not very portable, and no IDEs.

APPLICATIONS

Many (commercial) applications are built with functional programming languages based on the ability to manipulate symbolic data more easily.

Examples:

- Computer algebra (e.g. Reduce system).
- Natural language processing.
- Artificial intelligence.
- Automatic theorem proving.
- Emacs Lisp used to customize Emacs.
- Google's map-reduce for data processing.

LISP AND SCHEME

The original functional language and implementation of Lambda Calculus (Church's model of computing).

Lisp and dialects (Scheme, Common Lisp) are still the most widely used functional languages.

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: user interaction via "read-eval-print" loop.

SCHEME – FIRST INTRODUCTION

Scheme is a popular Lisp dialect.

Lisp and Scheme adopt a form of prefix notation called Cambridge Polish notation.

Scheme is case-insensitive.

A Scheme expression is composed of

- Atoms, e.g. a literal number, string, or identifier name.
- Lists, e.g. `'(1 2 3)`
- Function invocations written in list notation: the first list element is the function (or operator) followed by the arguments to which it is applied: `(function a1 a2 a3 ... an)`
- For example, `sin(x*x+1)` is written as `(sin (+ (* x x) 1))`

CAMBRIDGE POLISH NOTATION

How do we express 100+200*300-400/20?

What is the traditional notation for (sin (- (+ 10 20) (* 20 40)))?

READ-EVAL-PRINT

On linprog, type “scheme” to start the system.

The "Read-eval-print" loop provides user interaction in Scheme.

An expression is read, evaluated, and the result printed.

```
1 ]=> 9
;Value: 9
1 ]=> (+ 3 4)
;Value: 7
1 ]=> (+ (* 2 3) 1)
;Value: 7
1 ]=> (exit)

Kill Scheme (y or n)? Yes
Morituras te saluto.
```

DATA STRUCTURES IN SCHEME

An expression operates on values and compound data structures built from atoms and lists.

- A value is either an atom or a compound list.
- Atoms are
  - Numbers, e.g. `7` and `3.14`
  - Characters: `#\a`
  - Strings, e.g. `"abc"`
  - Boolean values `#t` (true) and `#f` (false)
  - Symbols, which are identifiers escaped with a single quote, e.g. `'y`
  - The empty list `()`
- When entering a list as a literal value, escape it with a single quote. Without the quote it is a function invocation!
- For example, `'(a b c)` is a list while `(a b c)` is a function invocation.
- Lists can be nested and may contain any value, e.g. `'(1 (a b) "s")`

CHECKING THE TYPE OF A VALUE

- The type of a value can be checked with
  - `(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?`
- Examples
  - `(list? '(2))` → `#t`
  - `(number? "abc")` → `#f`
  - Portability note: on some systems false (`#f`) is replaced with `()`

WORKING WITH LISTS

- `(car xs)` returns the head (first element) of list `xs`.
- `(cdr xs)` returns the tail of list `xs` (a sublist with every element except the first).
- `(cons x xs)` joins an element `x` and a list `xs` to construct a new list.
- `(list x1 x2 … xn)` generates a list from its arguments.
- Examples:
  - `(car '(2 3 4))` → `2`
  - `(car '(2))` → `2`
  - `(car '())` → produces error
  - `(cdr '(2 3))` → `(3)`
  - `(car (cdr '(2 3 4)))` → `3` `; also abbreviated as (cadr '(2 3 4))`
  - `(cdr (cdr '(2 3 4)))` → `(4)` `; also abbreviated as (cddr '(2 3 4))`
  - `(cdr '(2))` → `()`
  - `(cons 2 '(3))` → `(2 3)`
  - `(cons 2 '(3 4))` → `(2 3 4)`
  - `(list 1 2 3)` → `(1 2 3)`

HOW TO REPRESENT COMMON DATA STRUCTURES

Everything is a list:

- Array? int A[4] = {1, 2, 3, 4}
- 2-d array: int a[4][4] = {{0,0,0,0}, {1,1,1,1}, {2,2,2,2}, {3,3,3,3}}
- Structure? struct ex {int a; char b;}
- An array of a structures? struct ex {int a; char b;} aa[3] = {{10, 'a'}, {20, 'b'}, {30, 'c'}};
- A tree?
  - Can use pre-order traversal list form: `(root left_tree right_tree)`

The major difference between scheme data structures and C++ data structures? No random access mechanism, list items are mostly accessed through `car` and `cdr` functions.

THE "IF" SPECIAL FORM

- Special forms resemble functions but have special evaluation rules.
- Evaluation of arguments depends on the special construct.
- The "if" special form returns the value of thenexpr or elseexpr depending on a condition

```
(if condition thenexpr elseexpr)
```

- Examples

```
(if #t 1 2)
(if #f 1 "a")
(if (string? "s") (+ 1 2) 4)
(if (> 1 2) "yes" "no")
```

THE “IF” SPECIAL FORM

Examples:

```
(if #t 1 2) → 1
(if #f 1 "a") → "a"
(if (string? "s") (+ 1 2) 4) → 3
(if (> 1 2) "yes" "no") → "no"
```

THE "COND" SPECIAL FORM

A more general if-then-else can be written using the "cond" special form that takes a sequence of (condition value) pairs and returns the first value xi for which condition ci is true:

```
(cond (c1 x1) (c2 x2) … (else xn) )
```

Note: "else" is used to return a default value.

Examples:

```
(cond (#f 1) (#t 2) (#t 3) )
(cond ((< 1 2) ''one'') ((>= 1 2) ''two'') )
(cond ((< 2 1) 1) ((= 2 1) 2) (else 3) )
(cond (#f 1) (#f 2))
```

THE “COND” SPECIAL FORM

Examples:

```
(cond (#f 1) (#t 2) (#t 3) ) → 2
(cond ((< 1 2) “one”) ((>= 1 2) “two”) ) → “one”
(cond ((< 2 1) 1) ((= 2 1) 2) (else 3) ) → 3
(cond (#f 1) (#f 2)) → error (unspecified value)
```

LOGICAL EXPRESSIONS

- Numeric comparison operators <, <=, =, >, >=
- Boolean operators (and x1 x2 … xn), (or x1 x2 … xn)
- Other test operators
  - (zero? x), (odd? x), (even? x)
  - (eq? x1 x2) tests whether x1 and x2 refer to the same object
    - `(eq? 'a 'a)` → `#t`
    - `(eq? '(a b) '(a b))` → `#f`
  - (equal? x1 x2) tests whether x1 and x2 are structurally equivalent
    - `(equal? 'a 'a)` → `#t`
    - `(equal? '(a b) '(a b))` → `#t`
  - (member x xs) returns the sublist of xs that starts with x, or returns ()
    - `(member 5 '(a b))` → `()`
    - `(member 5 '(1 2 3 4 5 6))` → `(5 6)`

LAMBDA CALCULUS

A lambda abstraction is a nameless function (a mapping) specified with the lambda special form:

```
(lambda args body)
```

where args is a list of formal arguments and body is an expression that returns the result of the function evaluation when applied to actual arguments.

A lambda expression is an unevaluated function.

Examples:

```
(lambda (x) (+ x 1))
(lambda (x) (* x x))
(lambda (a b) (sqrt (+ (* a a) (* b b))))
```

LAMBDA CALCULUS

A lambda abstraction is applied to actual arguments using the familiar list notation

```
(function arg1 arg2 ... argn)
```

where function is the name of a function or a lambda abstraction. Beta reduction is the process of replacing formal arguments in the lambda abstraction's body with actuals.

Examples

- `( (lambda (x) (* x x)) 3 )` → `(* 3 3)` → `9`
-
```
( (lambda (f a) (f (f a))) (lambda (x) (* x x)) 3 )
→ (f (f 3))                          where f = (lambda (x) (* x x))
→ (f ( (lambda (x) (* x x)) 3 ))     where f = (lambda (x) (* x x))
→ (f 9)                              where f = (lambda (x) (* x x))
→ ( (lambda (x) (* x x)) 9 )
→ (* 9 9)
→ 81
```

MORE LAMBDA ABSTRACTIONS

Define a lambda abstraction that takes three parameters and returns the maximum value of the three.

Define a lambda abstraction that takes two parameters, one scheme arithmetic expression (e.g. (+ 1 2 4)), and the other one a number. The function return true (#t) if the expression evaluate to the number, false (#f) otherwise.

MORE LAMBDA ABSTRACTIONS

Define a lambda abstraction that takes three parameters and returns the maximum value of the three.

```
(lambda (x y z) (if (> x y) (if (> x z) x z) (if (> y z) y z)))
```

Define a lambda abstraction that takes two parameters, one scheme arithmetic expression (e.g. (+ 1 2 4)), and the other one a number. The function return true (#t) if the expression evaluate to the number, false (#f) otherwise.

```
(lambda (x y) (if (equal? x y) #t #f)
```

…or even just `(lambda (x y) (equal? x y))`

Functions are used much more liberally in scheme (compared to C++). The boundary between data and program is not as clear as in C++.

DEFINING GLOBAL NAMES

A global name is defined with the “define” special form

```
(define name value)
```

Usually the values are functions (lambda abstractions). Examples:

- `(define my-name “foo”)`
- `(define determiners '(“a” “an” “the”))`
- `(define sqr (lambda (x) (* x x)))`
- `(define twice (lambda (f a) (f (f a))))`
- `(twice sqr 3)` → `((lambda (f a) (f (f a))) (lambda (x) (* x x)) 3)` → … → `81`

DEFINING GLOBAL NAMES

A global name defined by a lambda expression in essence creates a named function.

This allows recursion – one of the most important/useful concepts in Scheme.

```
(define fib
 (lambda (x)
  (if (= x 0) 0 (if (= x 1) 1 (+ (fib (- x 1)) (fib (- x 2)))))))
```

- Define a function that computes the sum of all elements in a list.
- Define a function that puts an element at the end of a list.

LOAD PROGRAM FROM FILE

Load function

- `(load filename)` loads the program from a file.
- `(load "myscheme")` loads the program in "myscheme.scm".

After loading a file, one has access to all global names in the program.

I/O

- `(display x)` prints value of x and returns an unspecified value.
  - `(display "Hello World!")` Displays: "Hello World!“
  - `(display (+ 2 3))` Displays: 5
- `(newline)` advances to a new line.
- `(read)` returns a value from standard input.
  - `(if (member (read) '(6 3 5 9)) "You guessed it!" "No luck")`
  - Enter: 5
  - Displays: You guessed it!
- `(read-line)` returns the string in a line (without the “\n”).

I/O WITH FILES

- Open file for reading: `(open-input-file filename)` → `file`
- Open file for writing: `(open-output-file filename)` → `file`
- Read one character from a file: `(read-char file)`
- Read a scheme object from file: `(read file)`
- Check the current character in a file without consuming the character: `(peek-char file)`
- Check end-of-file: `(eof-object? (peek-char file))`
- Write one character to a file: `(write-char char file)`
- Write a scheme object to file: `(write object file)`
- Close file: `(close-port file)`

I/O WITH FILES

```
1 ]=> (define p (open-output-file "test.txt"))

;Value: p

1 ]=> (write 2 p)

;Unspecified return value

1 ]=> (close-output-port p)

;Unspecified return value
```

BLOCKS

- `(begin x1 x2 … xn)` sequences a series of expressions `xi`, evaluates them, and returns the value of the last one, `xn`.

Example:

```
(begin
  (display "Hello World!")
  (newline)
 )
```

DEFINING A FACTORIAL FUNCTION

```
(define fact
  (lambda (n)
    (if (zero? n) 1 (* n (fact (- n 1))))
  )
 )

(fact 2)    → (if (zero? 2) 1 (* 2 (fact (- 2 1))))
            → (* 2 (fact 1))
            → (* 2 (if (zero? 1) 1 (* 1 (fact (- 1 1)))))
            → (* 2 (* 1 (fact 0)))
            → (* 2 (* 1 (if (zero? 0) 1 (* 0 (fact (- 0 1))))))
            → (* 2 (* 1 1))
            → 2
```

SUMMING THE ELEMENTS OF A LIST

```
(define sum
  (lambda (lst)
    (if (null? lst)
      0
      (+ (car lst) (sum (cdr lst)))
    )
  )
)

(sum '(1 2 3))   → (+ 1 (sum (2 3))
                 → (+ 1 (+ 2 (sum (3))))
                 → (+ 1 (+ 2 (+ 3 (sum ()))))
                 → (+ 1 (+ 2 (+ 3 0)))
```

GENERATE A LIST OF N COPIES OF X

```
(define fill
 (lambda (n x)
    (if (= n 0)
      ()
      (cons x (fill (- n 1) x)))
  )
)

(fill 2 'a)      ⇒ (cons a (fill 1 a))
                 ⇒ (cons a (cons a (fill 0 a)))
                 ⇒ (cons a (cons a ()))
                 ⇒ (a a)
```

REPLACE X WITH Y IN LIST XS

```
(define subst
  (lambda (x y xs)
    (cond
      ((null? xs)           ())
      ((eq? (car xs) x)     (cons y (subst x y (cdr xs))))
      (else                 (cons (car xs) (subst x y (cdr xs))))
    )
  )
)
```

LOCAL NAMES

The “let” special form (let-expression) provides a scope construct for local name-to-value bindings.

```
(let ( (name1 x1) (name2 x2) … (namen xn) ) expression)
```

where name1, name2, …, namen in expression are substituted by x1, x2, …, xn.

Examples

- `(let ( (plus +) (two 2) ) (plus two two))` → `4`
- `(let ( (a 3) (b 4) ) (sqrt (+ (* a a) (* b b))))` → `5`
- `(let ( (sqr (lambda (x) (* x x)) ) (sqrt (+ (sqr 3) (sqr 4)))` → `5`

LOCAL BINDINGS WITH SELF REFERENCES

- A global name can simply refer to itself (for recursion).
  - `(define fac (lambda (n) (if (zero? n) 1 (* n (fac (- n 1))))))`
- A let-expression cannot refer to its own definitions.
  - Its definitions are not in scope, only outer definitions are visible.
- Use the letrec special form for recursive local definitions

```
(letrec ( (name1 x1) (name2 x2) … (namen xn) ) expr)
```

where namei in expr refers to xi.

Example:

```
(letrec
 ( (fac (lambda (n)
     (if (zero? n) 1 (* n (fac (- n 1))))))) )

(fac 5)) → 120
```

DO-LOOPS

The “do” special form takes a list of triples and a tuple with a terminating condition and return value, and multiple expressions xi to be evaluated in the loop.

```
(do (triples) (condition ret-expr) x1 x2 … xn)
```

Each triple contains the name of an iterator, its initial value, and the update value of the iterator.

Example (displays values 0 to 9):

```
(do ( (i 0 (+ i 1)) ) ( (>= i 10) "done" )
      (display i)
      (newline)
)
```

HIGHER-ORDER FUNCTIONS

- A function is a higher-order function (also called a functional form) if
  - It takes a function as an argument, or
  - It returns a newly constructed function as a result
  - For example, a function that applies a function to an argument twice is a higher-order function.
  - `(define twice (lambda (f a) (f (f a))))`
- Scheme has several built-in higher-order functions
- (map f xs) takes a function f and a list xs and returns a list with the function applied to each element of xs.
  - `(map odd? '(1 2 3 4))` → `(#t #f #t #f)`
  - `(map (lambda (x) (* x x)) '(1 2 3 4))` → `(1 4 9 16)`

FIND THE LAST ELEMENT OF A LIST

```
(define last
  (lambda (lst)
      ; base case: list with one element, return the element
      ; recursive case: list with more than one element, return
      ; the last element of the tail of the list.
      (cond
            ((null? L)  '())
            ((null? (cdr lst)) (car lst))
            (else (last (cdr lst)))
      )
  )
)
```

FIND THE ITH ELEMENT OF A LIST

```
(define ith
  (lambda (i lst)
          ; base case: list == ‘(), error or return ()
          ; base case i == 0, return the first item of the list
          ; recursive case: i!=0 and list is not empty, return the
          ; i-1th element of the tail of the list
          (cond      ((null? lst) ‘())
                     ((= i 0) (car lst))
                     (else  (ith (- i 1) (cdr lst)))
          )
      )
)
```

SUM THE LAST THREE ELEMENTS OF A LIST

```
(define sumlast3
  (lambda (lst)
          ; base cases: (1) empty list, return 0, (2) one item list,
          ; return the item (3) two items list, (4) three items list
          ; recursive case: list with more than 3 items, sumlast3
          ; of the tail of the list.
          (cond     ((null? lst) 0)                 ; 0 item
                    ((null? (cdr lst)) (car lst))     ; 1 item
                    ((null? (cdr (cdr lst))) (+ (car lst) (sumlast3 (cdr lst))))  ; 2 items
                    ((null? (cdr (cdr (cdr lst)))) (+ (car lst) (sumlast3 (cdr lst))));
                    (else (sumlast3 (cdr lst)))   ; more than 3 items
  )
 )
```

CONCATENATE TWO LISTS

```
(define concat
  (lambda (lst1  lst2)
        ; base case: lst1 is empty, return lst2
        ; recursive case: lst1 is not empty, concat the tail of
        ;    lst1 and lst2, and put head of lst1 in the front
        (if (null lst1) lst2
            (cons (car lst1) (concat (cdr lst1) lst2))
        )
  )
)
```

HIGHER-ORDER REDUCTIONS

```
(define reduce
  (lambda (op xs)
    (if (null? (cdr xs))
      (car xs)
      (op (car xs) (reduce op (cdr xs)))
    )
  )
)
```

(reduce and '(#t #t #f)) → (and #t (and #t #f)) → #f

(reduce * '(1 2 3)) → (* 1 (* 2 3)) → 6

(reduce + '(1 2 3 4)) → (+ 1 (+ 2 (+ 3 4))) → 10

HIGHER-ORDER FILTERING

Keep elements of a list for which a condition is true.

```
(define filter
  (lambda (op xs)
    (cond
      ((null? xs) ())
      ((op (car xs))    (cons (car xs) (filter op (cdr xs))))
      (else       (filter op (cdr xs)))
    )
  )
)
```

```
(filter odd? '(1 2 3 4 5)) → (1 3 5)
```

```
(filter (lambda (n) (<> n 0)) '(0 1 2 3 4)) → (1 2 3 4)
```

BINARY TREE SEARCH

() are leaves and (val left right) is a node.

```
(define search
  (lambda (n T)
    (cond
      ((null? T)          ())
      ((= (car T) n)      (list (car T))
      ((> (car T) n)      (search n (car (cdr T)))
      ((< (car T) n)      (search n (car (cdr (cdr T))))
    )
  )
)
```

(search 1 '(3 () (4 () ()))) → (search 1 `(4 () ())) →
(search 1 `()) → ()

BINARY INSERTION

Binary tree insertion, where () are leaves and (val left right) is a node.

```
(define insert
  (lambda (n T)
    (cond
      ((null? T)     (list n () ()))
      ((= (car T) n)   T)
      ((> (car T) n)  (list (car T) (insert n (cadr T)) (caddr T)))
      ((< (car T) n)  (list (car T) (cadr T) (insert n (caddr T))))
    )
  )
)
```

```
(insert 1 '(3 () (4 () ()))) → (3 (1 () ()) (4 () ()))
```

EXTRA: THEORY AND ORIGIN OF FUNCTIONAL LANGUAGES

- Church's thesis:
- All models of computation are equally powerful.
- 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.
- Functional programming languages implement 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“.