Overview

- What is functional programming?
- Historical origins of functional programming
- Functional programming today
- Concepts of functional programming
- Functional programming with Scheme
- Learn (more) by example
What is Functional Programming?

- Functional programming is a declarative programming style (programming paradigm)

- **Pro:** flow of computation is declarative, i.e. more implicit

- **Pro:** promotes building more complex functions from other functions that serve as building blocks (component reuse)

- **Pro:** behavior of functions defined by the values of input arguments only (no side-effects via global/static variables)

- **Cons:** function composition is (considered to be) stateless

- **Cons:** programmers prefer imperative programming constructs such as statement composition, while functional languages emphasize function composition
Concepts of Functional Programming

- Functional programming defines the outputs of a program purely as a mathematical function of the inputs with no notion of internal state (no side effects)
  - A *pure function* can be counted on to return the same output each time we invoke it with the same input parameter values
  - No global (statically allocated) variables
  - No explicit (pointer) assignments
    - Dangling pointers and un-initialized variables cannot occur
  - Example pure functional programming languages: Miranda, Haskell, and Sisal

- Non-pure functional programming languages include “imperative features” that cause side effects (e.g. destructive assignments to global variables or assignments/changes to lists and data structures)
  - Example: Lisp, Scheme, and ML
**Functional Language Constructs**

- Building blocks are functions
- No statement composition
  - 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
- Functional languages can be typed (Haskell) or untyped (Lisp)

**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`
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"
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
  - Strongly typed (with type inference)
  - 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 think 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
  - Algorithmic optimization of functional programs
LISP and Scheme

- The original functional language and implementation of Lambda Calculus
- 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

- 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. '(a b c)
  - Function invocations written in list notation: the first list element is the *function* (or operator) followed by the arguments to which it is applied:

  \[(function \text{ arg}_1 \text{ arg}_2 \text{ arg}_3 \ldots \text{ arg}_n)\]

- For example, \(\sin(x^2 + 1)\) is written as \((\sin (+ (* x x) 1))\)
Read-Eval-Print

- The "Read-eval-print" loop provides user interaction in Scheme
- An expression is read, evaluated, and the result 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")

Note: a file should use the .scm extension
Working with Data Structures

- 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
  - 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 application.
  - 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)` (pronounced "coulder") returns the tail of list `xs`
- `(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 '()) ⇒ 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)`
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) ⇒ 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 \( x_i \) for which condition \( c_i \) is true:

\[
(\text{cond } (c_1 \ x_1) \ (c_2 \ x_2) \ldots \ (\text{else } x_n))
\]

- Examples
  - \( (\text{cond } (#f \ 1) \ (#t \ 2) \ (#t \ 3)) \Rightarrow 2 \)
  - \( (\text{cond } ((< \ 1 \ 2) \ "one") \ ((>= \ 1 \ 2) \ "two") ) \Rightarrow "one" \)
  - \( (\text{cond } ((< \ 2 \ 1) \ 1) \ ((= \ 2 \ 1) \ 2) \ (\text{else } 3)) ) \Rightarrow 3 \)

- Note: “else” is used to return a default value
Logical Expressions

- Relations
  - Numeric comparison operators <, <=, =, >, <=, and <>

- Boolean operators
  - (and \( x_1 \ x_2 \ldots \ x_n \)), (or \( x_1 \ x_2 \ldots \ x_n \))

- Other test operators
  - (zero? \( x \)), (odd? \( x \)), (even? \( x \))
  - (eq? \( x_1 \ x_2 \)) tests whether \( x_1 \) and \( x_2 \) refer to the same object
    - (eq? 'a 'a) \( \Rightarrow \) #t
    - (eq? '(a b) '(a b)) \( \Rightarrow \) #f
  - (equal? \( x_1 \ x_2 \)) tests whether \( x_1 \) and \( x_2 \) are structurally equivalent
    - (equal? 'a 'a) \( \Rightarrow \) #t
    - (equal? '(a b) '(a b)) \( \Rightarrow \) #t
  - (member \( x \) \( xs \)) returns the sublist of \( xs \) that starts with \( x \), or returns ()
    - (member 5 '(a b)) \( \Rightarrow \) ()
    - (member 5 '(1 2 3 4 5 6)) \( \Rightarrow \) (5 6)
Lambda Calculus: Functions = Lambda Abstractions

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

  \[(\text{lambda } \text{args } \text{body})\]

  where \text{args} is a list of formal arguments and \text{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:
  - \[(\text{lambda } (x) (+ x 1))\]
  - \[(\text{lambda } (x) (* x x))\]
  - \[(\text{lambda } (a b) (\text{sqrt} (+ (* a a) (* b b))))\]
Lambda Calculus: Invocation
= Beta Reduction

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

\[(function \; arg_1 \; arg_2 \; \ldots \; arg_n)\]

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
  - \[( (\text{lambda } (x) (* \; x \; x)) \; 3 ) \Rightarrow (* \; 3 \; 3) \Rightarrow 9 \]
  - \[( (\text{lambda } (f \; a) \; (f \; (f \; a))) \; (\text{lambda } (x) (* \; x \; x)) \; 3 ) \]
    \[\Rightarrow (f \; (f \; 3))\]
    \[\Rightarrow (f \; ( (\text{lambda } (x) (* \; x \; x)) \; 3 ))\]
    \[\Rightarrow (f \; 9)\]
    \[\Rightarrow ( (\text{lambda } (x) (* \; x \; x)) \; 9 )\]
    \[\Rightarrow (* \; 9 \; 9)\]
    \[\Rightarrow 81\]
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
Using Local Names

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

\[
\text{(let ( (name}_1 \ \text{x}_1) \ (name}_2 \ \text{x}_2) \ \ldots \ (name}_n \ \text{x}_n) \ ) \ \text{expression}
\]

where \(name_1, name_2, ..., name_n\) in expression are substituted by \(x_1, x_2, ..., x_n\)

- Examples
  - \(\text{(let ( (plus +) (two 2) ) (plus two two)) } \Rightarrow 4\)
  - \(\text{(let ( (a 3) (b 4) ) (sqrt (+ (* a a) (* b b))) } \Rightarrow 5\)
  - \(\text{(let ( (sqr (lambda (x) (* x x)) ) (sqrt (+ (sqr 3) (sqr 4))) } \Rightarrow 5\)
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

\[
\text{letrec } ( (\text{name}_1 \ x_1) (\text{name}_2 \ x_2) \ldots (\text{name}_n \ x_n) ) \ expr
\]

where \text{name}_i in expr refers to \(x_i\)

- Examples
  - (letrec ( (fac (lambda (n) (if (zero? n) 1 (* n (fac (- n 1)))))) )
            (fac 5)) \(\Rightarrow\) 120
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!
Blocks

- (begin $x_1 \ x_2 \ldots \ x_n$) sequences a series of expressions $x_i$, evaluates them, and returns the value of the last one $x_n$

Examples:
- (begin
  (display "Hello World!"
  (newline)
)
- (let ( (x 1)
  (y (read))
  (plus +)
)
  (begin
    (display (plus x y))
    (newline)
  )
)
Do-loops

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

\[
(\text{do} (\text{triples}) \ (\text{condition} \ \text{ret-expr}) \ x_1 \ x_2 \ \ldots \ x_n)
\]

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

- Example (displays values 0 to 9)

```lisp
(\text{do} \ ((i \ 0 \ (+ \ i \ 1)))
  \ ((\geq \ i \ 10) \ "\text{done}")
  (\text{display} \ i)
  (\text{display} \ i)
  (\text{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
  - (apply f xs) takes a function f and a list xs and applies f to the elements of the list as its arguments
    - (apply '+ '(3 4)) ⇒ 7
    - (apply (lambda (x) (* x x)) '(3))
  - (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)
Non-Pure Constructs

- Assignments are considered non-pure in functional programming because they can change the global state of the program and possibly influence function outcomes.
- The value of a *pure function* only depends on its arguments.
- `(set! name x)` re-assigns `x` to local or global `name`
  - `(define a 0)`
  - `(set! a 1)` ; overwrite with 1
  - `(let ((a 0))`
    - `(begin)`
    - `(set! a (+ a 1))` ; increment `a` by 1
    - `(display a)` ; shows 1
  - )
- `(set-car! x xs)` overwrites the head of a list `xs` with `x`
- `(set-cdr! xs ys)` overwrites the tail of a list `xs` with `ys`
Example 1

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

- (fact 2)  \Rightarrow (if (zero? 2) 1 (* 2 (fact (- 2 1))))
  \Rightarrow (* 2 (fact 1))
  \Rightarrow (* 2 (if (zero? 1) 1 (* 1 (fact (- 1 1)))))
  \Rightarrow (* 2 (* 1 (fact 0)))
  \Rightarrow (* 2 (* 1 (if (zero? 0) 1 (* 0 (fact (- 0 1)))))
  \Rightarrow (* 2 (* 1 1))
  \Rightarrow 2
Example 2

- Iterative factorial
  
  ```scheme
  (define iterfact
    (lambda (n)
      (do ((i 1 (+ i 1))) ; i runs from 1 updated by 1
          (f 1 (* f i)) ; f from 1, multiplied by i
        ( (> i n) f ) ; until i > n, return f
      )
    )
  )
  ```
Example 3

- Sum 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)))
Example 4

- 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)  \Rightarrow (cons a (fill 1 a))
  \Rightarrow (cons a (cons a (fill 0 a)))
  \Rightarrow (cons a (cons a (())))
  \Rightarrow (a a)
Example 5

- Replace \( x \) with \( y \) in list \( xs \)

```scheme
(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)))))
    ))
```

- \((\text{subst } 3 \ 0 \ \text{(8 2 3 4 3 5)}) \Rightarrow \text{(8 2 0 4 0 5)}\)
Example 6

- 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)) \(\Rightarrow\) (and #t (and #t #f)) \(\Rightarrow\) #f
- (reduce * '(1 2 3)) \(\Rightarrow\) (* 1 (* 2 3)) \(\Rightarrow\) 6
- (reduce + '(1 2 3)) \(\Rightarrow\) (+ 1 (+ 2 3)) \(\Rightarrow\) 6
Example 7

- Higher-order filter operation: keep elements of a list for which a condition is true

```scheme
(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)`
Example 8

- 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 () ()))