Functional Programming

- *As software becomes more and more complex, it is more and more important to structure it well.*
- *Well-structured software is easy to write and to debug, and provides a collection of modules that can be reused to reduce future programming costs.*
- Map-Reduce has its roots in functional programming, which is exampled in languages such as Lisp and ML.

Functional Programming

- Examples

- // Regular Style
```
Integer timesTwo(Integer i) {
  return i * 2;
}
```

- // Functional Style
```
F<Integer, Integer> timesTwo = new F<Integer, Integer>() {
  public Integer f(Integer i) { return i * 2; }
}
```

Functional Programming

- A key feature of functional languages is the concept of higher order functions, or functions that can accept other functions as arguments.
- // Regular Style

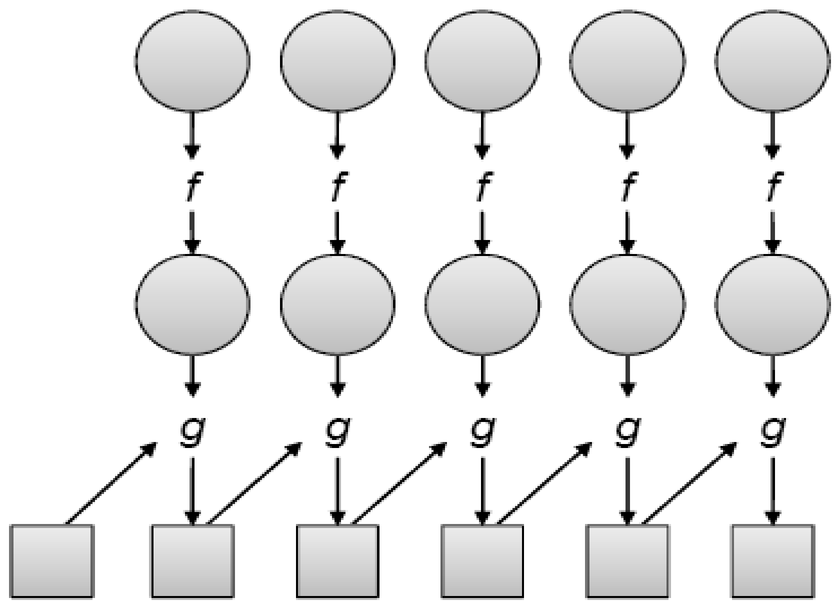
```
List<Integer> oneUp = new ArrayList<Integer>();
for (Integer i: ints)
  oneUp.add(plusOne(i));
```

// Functional Style

```
List<Integer> oneUp = ints.map(plusOne);
```

Functional Programming

Two common built-in higher order functions are **map** and **fold**.



map and *fold*, two higher-order functions commonly used together in functional programming: *map* takes a function f and applies it to every element in a list, while *fold* iteratively applies a function g to aggregate results.

First class Function

- *First-class function values: the ability of functions to return newly constructed functions*

-
```
F<Integer, Integer> timesTwo = new F<Integer, Integer>() {
  public Integer f(Integer i) { return i * 2; }
}
```

Functional Programming

- Functional programming is a declarative programming paradigm
- Computation is more implicit (**suggested but not communicated directly**) and functional call is the only form of explicit control

- 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 (e.g **Map-reduce**)

- 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***

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

Concepts of 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)*
- *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 in functional languages

- 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.*
  - ***constructs*** *for constructing structured objects: the ability to specify a structured object in-line, e.g. a complete list or record value.*
  - ***Garbage collection*** *for memory management.*

LISP

- **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

Data Structures

- **LISP & Scheme**
- 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

Data Structures

- **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**: () ------------------------Empty list

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)

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:
- 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) ((= 2 1) 2) (else 3)) Output: 3*

Lambda Abstraction

- A **Scheme** *lambda abstraction is a nameless function specified* with the lambda special form:
- **(lambda *formal-parameters function-body)***
- 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: x2*
  - (lambda (a b) (sqrt (+ (* a a) (* b b)))) *; is a function:*

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)))**
    - Input: (sqr 3) Output: 9
    - Input: (sqr (sqr 3)) Output: 81
- **(define hypot (lambda (a b) (sqrt (+ (* a a) (* b b)))))**
    - Input: (hypot 3 4) Output: 5

I/O and Sequencing

- **“read-evaluate-print”**
- **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**)
- Example:
  - *(begin*
  - *(display "Hello World!")*
  - *(newline)*
  - *(end)*
  - *)*

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 (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
  - Input (fill 3 “a”) output (“a” “a” “a”)

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

Assignment

What are the Beauties of functional programming?