Overview

- Finish up function calls
  - Tail recursion, Short circuiting
- Comparison of object oriented and functional programming
- Other Language Types
  - Markup languages
    - Set of annotations to text
  - Query languages
    - Make queries to databases & information systems
    - Used together in
      - Web interface to databases

Tail Calls

- A tail call is a function call that is the last thing a function does before it returns
  - Not just function call in last line of code in function

```
let add x y = x + y
let f x = add x x (* tail call *)
```

```
let rec len = function
  | [] -> 0
  | (x::t) -> 1 + (len t) (* not tail call, performs +1 *)
```

```
let rec len a = function
  | [] -> a
  | (x::t) -> len (a + 1) t (* tail call *)
```

Tail Recursion

- Recall that in OCaml, all looping is via recursion
  - Seems very inefficient
  - Needs one stack frame for each recursive call
- A function is tail recursive
  - If it is recursive
    - And recursive call is a tail call
- If function is tail recursive
  - Can reuse stack frame for each recursive call
Tail Recursion (cont.)

Function is not tail recursive

• Use stack frame store return value
• Add 1 to return value, use as new return value

```
let rec len l = match l with
  | [] -> 0
  | (_::t) -> 1 + (len t)

len [1; 2]
```

eax: 2

Tail Recursion (cont.)

Function is tail recursive

• Same stack frame can be reused for the next call
• Since we’d just pop it off and return anyway

```
let rec len a l = match l with
  | [] -> a
  | (_::t) -> (len (a + 1) t)

len 0 [1; 2]
```

eax: 2

Short Circuiting

Will OCaml raise a `Division_by_zero` exception?

```
let x = 0

if x <> 0 && (y / x) > 100 then
  print_string "OCaml sure is fun"

if x == 0 || (y / x) > 100 then
  print_string "OCaml sure is fun"
```

• No: `&&` and `||` are short circuiting in OCaml
  > `e1 && e2` evaluates `e1`. If false, it returns false. Otherwise, it returns the result of evaluating `e2`
  > `e1 || e2` evaluates `e1`. If true, it returns true. Otherwise, it returns the result of evaluating `e2`

Short Circuiting (cont.)

C, C++, Java, and Ruby all short-circuit `&&`, `||`

• But some languages don’t, like Pascal (although Turbo Pascal has an option for this):

```
x := 0;
...
if (x <> 0) and (y / x > 100) then
  writeln('Sure OCaml is fun');
```

• So this would need to be written as

```
x := 0;
...
if x <> 0 then
  if y / x > 100 then
    writeln('Sure OCaml is fun');
```
OOP vs. FP

- Object-oriented programming (OOP)
  - Computation as interactions between objects
  - Objects encapsulate mutable data (state)
    - Accessed / modified via object’s public methods

- Functional programming (FP)
  - Computation as evaluation of functions
  - Mutable data used to improve efficiency
  - Higher-order functions implemented as closures
    - Closure = function + environment

An Integer “Stack” Abstraction in Java

```java
import java.util.Stack;

public class Stack {
    private Node theStack;

    public void push(Integer v) {
        theStack = new Node(v, theStack);
    }

    public Integer pop() {
        if (theStack == null)
            throw new NoSuchElementException();
        Integer temp = theStack.val;
        theStack = theStack.next;
        return temp;
    }
}
```

A “Stack” Abstraction in OCaml

```ocaml
module type STACK = sig
  type 'a stack
  val new_stack : unit -> 'a stack
  val push : 'a stack -> 'a -> unit
  val pop : 'a stack -> 'a
end

module Stack : STACK = struct
  type 'a stack = 'a list ref

  let new_stack () = ref []
  let push s x = s := (x::!s)
  let pop s = match !s with
    | [] -> failwith "Empty stack"
    | (h::t) -> s := t; h
  in
  (push, pop)
```

Another “Stack” Abstraction in OCaml

```ocaml
let new_stack () =
  let this = ref [] in
  let push x = this := (x::!this) in
  let pop () = match !this with
    | [] -> failwith "Empty stack"
    | (h::t) -> this := t; h
  in
  (push, pop)
```

```ocaml
let s = new_stack ();;
val s : ('_a -> unit) * (unit -> '_a) = (<fun>, <fun>)

Pervasives.fst s 3;;
(* applies 1st part of s to 3 *)
- : unit = ()

Pervasives.snd s ();;
(* applies 2nd part of s to () *)
- : int = 3
```
Two OCaml Stack Implementations

1st implementation (OOP style)
- Based on modules
- Specifies methods for
  - Creating stack
  - Pushing value onto stack parameter
  - Popping value from stack parameter

2nd implementation (FP style)
- Based on closures
- Creating stack returns tuple containing
  - Closure for pushing value onto created stack
  - Closure for popping value from created stack

Relating Objects and Closures

- An object...
  - Is a collection of fields (data)
  - ...and methods (code)
  - When a method is invoked
    - Method has implicit this parameter that can be used to access fields of object

- A closure...
  - Is a pointer to an environment (data)
  - ...and a function body (code)
  - When a closure is invoked
    - Function has implicit environment that can be used to access variables

Relating Objects and Closures (cont.)

```ocaml
class C {
  int x = 0;
  void set_x(int y) { x = y; }
  int get_x() { return x; }
}

let make () =
  let x = ref 0 in
  ( (fun y -> x := y),
    (fun () -> !x) )

x = 0
```

Encoding Objects with Functions

- We can apply this transformation in general
  ```ocaml
class C { f1 ... fn; ml ... mn; }
```
  becomes
  ```ocaml
  let make () =
  let fi = ...
  ... and fn = ... in
  { fun ..., (* body of ml *)
    ...
    fun ..., (* body of mn *)
  }
```
- make () is like the constructor
- The closure environment contains the fields

```ocaml
C c = new C();
c.set_x(3);
int y = c.get_x();
```
Recall a Useful Higher-Order Function

Map applies an arbitrary function \( f \)
- To each element of a list
- And returns the resulting modified list
Can we encode this in Java?
- Using object oriented programming

let rec map\( f \) = function
| [] -> [] |
| (h::t) -> (f h)::(map \( f \) t) |

A Map Method for Stack

Problem – Write a map method in Java
- Must pass a function into another function
Solution
- Can be done using an object with a known method
- Use interface to specify what method must be present

A Map Method for Stack (cont.)

Examples
- Two classes which both implement Function interface

```java
class AddOne implements Function {
    Integer eval(Integer arg) {
        return new Integer(arg + 1);
    }
}
```

```java
class MultTwo implements Function {
    Integer eval(Integer arg) {
        return new Integer(arg * 2);
    }
}
```

The New Stack Class

```java
class Stack {
    class Node {
        Integer val; Node next;
        Node (Integer v, Node n) { val = v; next = n; }
        Node map(Function f) {
            if (next == null)
                return new Node(f.eval(val), null);
            else return new Node(f.eval(val), next.map(f));
        }
    }
    Node theStack;
    ...
    Stack map(Function f) {
        Stack s = new Stack();
        s.theStack = theStack.map(f);
        return s;
    }
}
```
Applying Map To A Stack

Then to apply the function, we just do

```java
Stack s = ...;
Stack t = s.map(new AddOne());
Stack u = s.map(new MultTwo());
```

• We make a new object
  > That has a method that performs the function we want
• This is sometimes called a callback
  > Because map “calls back” to the object passed into it
• But it’s really just a higher-order function
  > Written more awkwardly

Relating Closures and Objects

```java
interface F {
    Integer eval(Integer y);
}
class C {
    static Integer app(F f, Integer x) {
        return f.eval(x);
    }
}
```

```java
let app f x = f x

a = 3
F adder = new G(3);
C.app(adder, 4);
```

```java
class G implements F {
    Integer a;
    G(Integer a) { this.a = a; }
    Integer eval(Integer y) {
        return new Integer(a + y);
    }
}
```

```java
class C {
    Typ h(F f, ...) {
        ...f.eval(y)...
    }
}
```

Code as Data

Closures and objects are related
• Both of them allow
  > Data to be associated with higher-order code
  > Pass code around the program

The key insight in all of these examples
• Treat code as if it were data
  > Allowing code to be passed around the program
  > And invoked where it is needed (as callback)

Approach depends on programming language
• Higher-order functions (OCaml, Ruby, Lisp)
• Function pointers (C, C++)
• Objects with known methods (Java)
Code as Data (cont.)

- This is a powerful programming technique
  - Solves a number of problems quite elegantly
    - Create new control structures (e.g., Ruby iterators)
    - Add operations to data structures (e.g., visitor pattern)
    - Event-driven programming (e.g., observer pattern)
  - Keeps code separate
    - Clean division between higher & lower-level code
  - Promotes code reuse
    - Lower-level code supports different callbacks

Markup Languages

- Set of annotations (tags) added to text
  - Example – `<tag> text </tag>`
- Describe how text is
  - Structured, laid out, formatted...
- First used in publishing industry
  - Typesetting, proofreading
    - nroff, troff, TeX, LaTeX
  - Mostly replaced by WYSIWYG editors like MS Word
    - What you see is what you get
- Regained importance with advent of web
  - Used to describe format & presentation of web pages

History of Markup Languages

- Generalized Markup Language (GML)—1960s
  - Describe both structure & presentation of content

- Hypertext Markup Language (HTML)—1991
  - Web pages
    - Hypertext links parts of document to other documents

- Extensible Markup Language (XML)—1998
  - Language for describing tags (meta-language)
  - User can create tags and describe their uses
  - Used to describe documents w/ structured information

Markup Language – HTML

- Example
  `<html>
  <head><title>Bread Recipe</title></head>
  <body>
  <h1>Bread</h1>
  <ol>
  <li>Flour
  <li>Yeast
  <li>Water
  </ol>
  </body>
  </html>`
Markup Language – XML

- Example
  
  ```xml
  <recipe name="Bread">
    <title>Bread</title>
    <ingredient>Flour</ingredient>
    <ingredient>Yeast</ingredient>
    <ingredient>Water</ingredient>
  </recipe>
  ```

HTML / XML Elements

- **Element**
  - A start tag, an end tag, and data in between
  - Examples
    - `<director> Tyler Perry </director>`
    - `<actor> Tyler Perry </actor>`

- **Attribute**
  - A name-value pair separated by an equal sign (=)
  - Used to attach additional information to an element
  - Example
    - `<city ZIP="20742"> College Park </city>`

HTML Elements

- **Structural**
  - Describes purpose of text
  - Examples
    - `<h1>` Level 1 heading `<h1>`
    - `<ol>` Ordered list `<ol>`
    - `<ul>` Unordered list `<ul>`
    - `<li>` List item `<li>`

HTML Elements (cont.)

- **Presentation (cont.)**
  - Describes appearance of text
  - Examples
    - `<b>` boldface `<b>`
    - `<i>` italics `<i>`
    - `<p>` line spacing `<p>`

- **Hypertext**
  - Links part of document to other documents
  - Examples
    - `<a>` Anchor `<a>`
    - `<a href="http://www.cs.umd.edu"> URL link `<a>`
XML Document

- An XML element with nested XML elements
- Example
  ```xml
  <movies>
  <movie year="2005">
    <title> Diary of a Mad Black Woman </title>
    <director> Tyler Perry </director>
  </movie>
  <movie year="2006">
    <title> Madea's Family Reunion </title>
    <director> Tyler Perry </director>
  </movie>
  </movies>
  ```

Comparing HTML With XML

- HTML
  - Fixed set of tags
  - Presentation oriented
  - No data validation capabilities
  - Single presentation
- XML
  - Extensible set of tags
  - Content oriented
  - Standard Data infrastructure
  - Multiple output forms

Markup Language Usage

- Started with documents
  - Separate presentation from content
    - Keep presentation elsewhere (CSS, XSL)
  - Descriptive markup
- Now also used to organize
  - Metadata
    - Data about data, used to help understand / manage data
    - Example: `<LastName optional="true"> Smith </LastName>`
  - Transactions
    - Single unit of work for application
  - Applications
    - Helping applications interact / work together

Query Languages

- Make queries to
  - Databases
  - Information systems
- Goals
  - Data retrieval
  - Data management
- Examples
  - SQL (1970s) – Query relational databases
  - LDAP (1993) – Query directory services for TCP/IP
Databases (DB)

- A structured collection of data (records)
  - Whose content can be quickly and easily accessed, managed, updated

- Database model
  - Hierarchical
    - Records are stored in a tree
  - Network
    - Records have links to other records
  - Relational
    - Records are stored in tables (relations)

Tables (Relations)

- Each column constitutes an attribute
- Each row constitutes a record or tuple

---

### SQL (Structured Query Language)

- Queries for relational database systems
- Allows for complete
  - Table creation, deletion, editing
  - Data extraction (queries)
  - Database management & administration

---

### SQL – Creating Database

- Types of attributes
  - char, varchar, int, decimal, date, etc.
  - varchar is a string with varying # of chars
- Not Null
  - Each record must have a value
- Primary key
  - Must be unique for each record

```sql
CREATE TABLE tableName (
    name VARCHAR(55),
    sex CHAR(1) NOT NULL,
    age INT(3),
    birthdate DATE,
    primary key (name)
);
```
SQL – Inserting Values

INSERT INTO tableName (name, sex, age) VALUES ('Bob', 'M', 42);

INSERT INTO tableName (age, name, sex,) VALUES (42, 'Bob', 'M');

- Identical result
- Order of fields do not matter

SQL – Retrieving Values

Operations in the form
- Select ...
- From ...
- Where ...

Means
- Select a column
- From a database
- Where x meets y condition

Database Server

- Accepts requests to access database
  - Requests in query language (e.g., SQL)

MySQL
- Multithreaded
- Multiuser
- SQL database management system (DBMS)
- Open source
  - Free download of Community Edition