CMSC 330: Organization of Programming Languages

OCaml 3
Nested Functions, Closures
Currying Is Standard In OCaml

- Pretty much all functions are curried
  - Like the standard library `map`, `fold`, etc.
  - See `/usr/local/ocaml/lib/ocaml` on Grace
    - In particular, look at the file `list.ml` for standard list functions
    - Access these functions using `List.<fn name>`
    - E.g., `List.hd`, `List.length`, `List.map`

- OCaml works hard to make currying efficient
  - Otherwise it would do a lot of useless allocation of closures (which we see later) when all arguments are provided
A Convention

- Since functions are curried, **function** can often be used instead of **match**
  - **function** declares an anonymous function of one argument
  - Instead of
    ```plaintext
    let rec sum l = match l with
        [] -> 0
        | (h::t) -> h + (sum t)
    ```
  - It could be written
    ```plaintext
    let rec sum = function
        [] -> 0
        | (h::t) -> h + (sum t)
    ```
A Convention (cont.)

Instead of

```ml
let rec map f l = match l with
  | [] -> []
  | (h::t) -> (f h)::(map f t)
```

It could be written

```ml
let rec map f = function
  | [] -> []
  | (h::t) -> (f h)::(map f t)
```
Nested Functions

- In OCaml, you can define functions anywhere
  - Even inside of other functions

```ocaml
let sum l = 
  fold (fun a x -> a + x) 0 l

let pick_one n = 
  if n > 0 then (fun x -> x + 1) 
  else (fun x -> x - 1) 
(pick_one -5) 6   (* returns 5 *)
```
You can also use `let` to define functions inside of other functions

```ocaml
let sum l =  
    let add a x = a + x in  
    fold add 0 l

let pick_one n =  
    let add_one x = x + 1 in  
    let sub_one x = x - 1 in  
    if n > 0 then add_one else sub_one
```
How About This?

- (Equivalent to...)

```ocaml
let addN n l =  
    let add x = n + x in  
    map add l
```

Accessing variable from outer scope

```ocaml
let addN n l =  
    map (fun x -> n + x) l
```
Returned Functions

- In OCaml a function can return another function as a result; this is what currying is doing
  - Consider the following example

```ocaml
let addN n = (fun x -> x + n)
(addN 3) 4 (* returns 7 *)
```

- When the anonymous function is called, `n` isn’t even on the stack any more!
  - We need some way to keep `n` around after `addN` returns
The Call Stack in C/Java/etc.

```c
void f(void) {
    int x;
    x = g(3);
}

int g(int x) {
    int y;
    y = h(x);
    return y;
}

int h(int z) {
    return z + 1;
}

int main(){
    f();
    return 0;
}
```
Now Consider Returning Functions

Uh oh...how does `add` know the value of `n`?

- OCaml does not read it off the stack
  - The language could do this, but can be confusing (see above)
- OCaml uses static scoping like C, C++, Java, and Ruby
Static Scoping (*aka* Lexical Scoping)

- **In static or lexical scoping**, (nonlocal) names refer to their nearest binding in the program text:
  - Going from inner to outer scope
  - In our example, `add` refers to `addN`'s `n`
  - C example:

```c
int x;
void f() { x = 3; }
void g() { char *x = "hello"; f(); }
```

Refers to the `x` at file scope – that’s the nearest `x` going from inner scope to outer scope in the source code.
Static vs. Dynamic Scoping

```
let x = 10 ;;
let f y = x + y ;;
let x = 5 ;;
let y = 7 ;;
let z = f (x + y) ;;
```

What is the value of z=?
Static vs. Dynamic Scoping

```
let x = 10 in
let f = fun y -> x in
let x = 20 in
f 5
```

What is the value of z=?
10 or 20?

static: 10
dynamic: 20
Closures Implement Static Scoping

- An environment is a mapping from variable names to values
  - Just like a stack frame

- A closure is a pair \((f, e)\) consisting of function code \(f\) and an environment \(e\)

- When you invoke a closure, \(f\) is evaluated using \(e\) to look up variable bindings
Example – Closure 1

\[
\text{let add } x = \text{(fun } y \rightarrow x + y) \\
\]

\[
(\text{add } 3) \rightarrow \langle \text{cl} \rangle 4 \rightarrow 3 + 4 \rightarrow 7
\]
Example – Closure 2

```ocaml
let mult_sum (x, y) =
  let z = x + y in
  fun w -> w * z
```

`(mult_sum (3, 4)) 5
→ <cl> 5
→ 5 * 7
→ 35`
Example – Closure 3

let twice (n, y) =
    let f x = x + n in
    f (f y)

twice (3, 4) → <cl> (<cl> 4) → <cl> 7 → 10
Example – Closure 4

```ocaml
let add x = (fun y -> (fun z -> x + y + z))
```

```
let add x = (fun y -> (fun z -> x + y + z))
```

add( ) took 3 arguments? The compiler figures this out and avoids making closures

(((add 1) 2) 3) → (((<cl> 2) 3) → (<cl> 3) → 1+2+3

```
fun y ->
  (fun z ->
    x+y+z)
```

fun z ->
  x+y+z

x = 1

x = 1

y = 2
Higher-Order Functions in C

- C supports function pointers

```c
typedef int (*int_func)(int);
void app(int_func f, int *a, int n) {
    for (int i = 0; i < n; i++)
        a[i] = f(a[i]);
}
int add_one(int x) { return x + 1; }
int main() {
    int a[] = {5, 6, 7};
    app(add_one, a, 3);
}
```
C does not support closures
  • Since no nested functions allowed
  • Unbound symbols always in global scope

```c
int y = 1;
void app(int(*f)(int), n) {
    return f(n);
}
int add_y(int x) {
    return x + y;
}
int main() {
    app(add_y, 2);
}
```
Higher-Order Functions in C (cont.)

- Cannot access non-local variables in C
- OCaml code
  
  ```ocaml
  let add x y = x + y
  ```

- Equivalent code in C is illegal
  
  ```c
  int (* add(int x))(int) {
    return add_y;
  }
  int add_y(int y) {
    return x + y; // x undefined
  }
  ```
Higher-Order Functions in C (cont.)

- OCaml code

```ocaml
let add x y = x + y
```

- Works if C supports nested functions
  - Not in ISO C, but in gcc; but not allowed to return them

```c
int (* add(int x))(int) {
    int add_y(int y) {
        return x + y;
    }
    return add_y;
}
```

- Does not allocate closure, so x popped from stack and add_y will get garbage (potentially) when called
Ruby supports higher-order functions
  - Use yield within method to call code block argument

```ruby
def my_collect(a)
  b = Array.new(a.length)
  0.upto(a.length-1) { |i| 
    b[i] = yield(a[i])
  }
  return b
end
b = my_collect([5, 6, 7]) { |x| x+1 }
```
Higher-Order Functions in Ruby (cont.)

- Ruby supports closures
  - Code blocks can access non-local variables
  - Binding determined by lexical scoping

```ruby
def twice
  yield
  yield
end
x = 1
twice {x += 1}
puts x  # 3
```

```ruby
def twice
  x = 0  #dynamic
  yield
  yield
end
x = 1  #lexical
twice {x += 1}
puts x  # 3 not 1
```
Ruby code blocks are actual variables

```ruby
def twice  # implicit block
  yield   # invoked with yield
  yield
end

# same as x += 2
\n
twice { x += 1 }

↓

def quad (&block) # explicit block
  twice (&block)  # used as argument
  twice (&block)
end

# same as x += 4
quad { x += 1 }
```
Higher-Order Functions in Ruby (cont.)

- Code blocks may be saved

```ruby
def quad (&block)  # explicit block
  c = block  # no ampersand!
  twice (c)  # used as argument
  twice (c)
end

def twice c  # arg = explicit closure
  c.call  # invoke with .call
  c.call
end
quad { x += 1 }  # same as x += 4
```
Ruby supports creating closures directly

- Proc.new
- proc
- lambda
- method

```ruby
\[
c1 = \text{Proc.new} \{ \text{x+=1} \}
c2 = \text{proc} \{ \text{x+=1} \}
c3 = \text{lambda} \{ \text{x+=1} \}
def foo
  \text{x+=1}
end
c4 = \text{method} \{ :foo \}
\]
```

```ruby
downarrow
\[
c.\text{call} \# \text{x+=1}
\]
```
Higher-Order Functions in Java/C++

- An object in Java or C++ is kind of like a closure
  - It has some data (like an environment)
  - Along with some methods (i.e., function code)
  - So objects can be used to simulate closures

- So is an anonymous Java inner class
  - Inner class methods can access fields of outer class

- Back in CMSC 132 (OOP II)
  - We studied how to implement some functional patterns in OO languages
Java 8 Supports Lambda Expressions

- Ocaml’s
  
  \[
  \text{function (a, b) -> a + b}
  \]

- Is like the following in Java 8

  \[
  (a, b) -> a + b
  \]

- Java 8 supports closures, and variations on this syntax
public class Calculator {
    interface IntegerMath { int operation(int a, int b); }  
    public int operateBinary(int a, int b, IntegerMath op) {
        return op.operation(a, b);
    }

    public static void main(String... args) {
        Calculator myApp = new Calculator();
        IntegerMath addition = (a, b) -> a + b;
        IntegerMath subtraction = (a, b) -> a - b;
        System.out.println("40 + 2 = " +
                          myApp.operateBinary(40, 2, addition));
        System.out.println("20 - 10 = " +
                          myApp.operateBinary(20, 10, subtraction));
    }
}