CSE 3302
Lecture 12: Functional Languages
12 Oct 2010

Nate Nystrom
University of Texas at Arlington
Functional languages

Key idea 1:
A variable names a value not a location

• let x = 3   // x is 3. x will always be 3.

• x = 4      // error: re-assignment not allowed
What does this give you?

Functions have no side-effects

- Only inputs to a function are its arguments
- Only output of a function is its return value

If a function is called multiple times with the same arguments, it will always return the same value (can memoize)

Can run functions concurrently without interference

No accidental coupling between components (i.e., thru shared variables)
Key idea 2:
Functions are values, just like integers, strings, etc.

• Can create new functions at run time
• Can pass functions to other functions
• Can return functions from functions
• Can put functions in variables
What does this give you?
What does this give you?

Power!
What does this give you?

Power!

(mwa ha ha ha ha)
What does this give you?

Higher-level, more declarative programming style

Can build programs by composing functions
Passing functions to functions

In Scala:

Sum all the elements of a list xs:
• \( \text{xs.foldLeft}(0)(\_ + \_ ) \)

Multiply all the elements of a list xs:
• \( \text{xs.foldRight}(1)(\_ * \_ ) \)

Sum of squares:
• \( \text{xs.map}(x \mapsto \text{x}^2).\text{foldRight}(0)(\_ + \_ ) \)

_ + _ is a function that adds two integers
_ * _ is a function that multiples two integers
Passing functions to functions

In OCaml:

Sum all the elements of a list \( xs \):
- \( \text{fold\_left} \ (+) \ 0 \ xs \)

Multiply all the elements of a list \( xs \):
- \( \text{fold\_left} \ (*) \ 1 \ xs \)

Sum of squares:
- \( \text{fold\_left} \ (+) \ 0 \ (\text{map} \ (\text{fun} \ x \ -> \ x\times x) \ xs) \)

\(+\) is a function that adds two integers
\(*\) is a function that multiples two integers
Note that function calls are so ubiquitous in functional languages that in most functional languages the syntax for a function call is juxtaposition:

\[ f \ x \]

vs.

\[ f(x) \]
Passing functions to functions

Scala:

```scala
val xs = List("a", "B", "c", "D", "e")
val less = (x: String, y: String) => x < y
val iless = (x: String, y: String) => x.toLowerCase < y.toLowerCase

scala> xs.sort(less)
res4: List[java.lang.String] = List(B, D, a, c, e)
scala> xs.sort(iless)
res5: List[java.lang.String] = List(a, B, c, D, e)
```
Operators as functions

Most of these languages allow built-in operators to be used as functions

OCaml:
(+) is a function that takes two ints and returns their sum
(*) is a function that takes two ints and returns their product

Scala:
_  +  _  is shorthand for  \((x,y) \Rightarrow x+y\)
• But this only works in some contexts
  (e.g., cannot do val plus = _ + _)
• Why?
Some useful functions on lists:

These are found in most functional languages
Examples here are OCaml

foldLeft, fold_left, foldl
fold_left (+) 0 [1;2;3;4;5] == (((((0 + 1) + 2) + 3) + 4) + 5)

foldRight, fold_right, foldr
fold_right (+) [1;2;3;4;5] 0 == (1 + (2 + (3 + (4 + (5 + 0)))))

map
map (fun x -> x*x) [1;2;3;4;5] == [1;4;9;16;25]
Some more useful functions on Lists

Scala (other languages have similar functions):

```scala
scala> List(-2, -1, 0, 1, 2).filter(_ > 0)
res6: List[Int] = List(1, 2)

scala> List(-2, -1, 0, 1, 2).count(_ > 0)
res7: Int = 2

scala> List(-2, -1, 0, 1, 2).forall(_ > 0)
res9: Boolean = false

scala> List(-2, -1, 0, 1, 2).exists(_ > 0)
res10: Boolean = true

scala> List(-2, -1, 0, 1, 2).partition(_ > 0)
res11: (List[Int], List[Int]) = (List(1, 2), List(-2, -1, 0))
```
Creating functions at run-time

Scala
- (x:Int) => x+1
- (x:Int, y:Int) => x+y
- (x:Int) => (y:Int) => x+y

Python
- lambda x: x+1
- lambda x,y: x+y
- lambda x: lambda y: x+y

Scheme
- (lambda (x) (+ x 1))
- (lambda (x y) (+ x y))
- (lambda (x) (lambda (y) (+ x y)))

OCaml
- fun x -> x+1
- fun (x, y) -> x+y
- fun (x) -> fn(y) -> x+y
Functions that take functions

Scala:

```scala
def map[S,T](f: S => T, list: List[S]): List[T] = {
  list match {
    case Nil => Nil
    case head :: tail => f(head) :: map(f, tail)
  }
}
```

```scala
scala> map[Int,Int](+_1, List(1, 2, 3))
res6: List[Int] = List(2, 3, 4)
```
Currying

Any function that takes multiple args can be written to take the first arg and return a function that takes the rest

This is called *currying* (after Haskell Curry)

- nb. should really be *schönfinkeling* (after Moses Schönfinkel)

```
    def sum(x: Int, y: Int) = x + y
    vs.
    def curriedSum(x: Int) = (y: Int) => x + y
```

Apply the arguments one at a time:

```
    sum(1,2) vs. curriedSum(1)(2)
```
Currying

def curriedSum(x: Int) = (y: Int) => x + y

curriedSum(1)(2) adds 1 and 2
curriedSum(1) is a function that adds 1 to its argument

scala> map[Int,Int](curriedSum(1), List(1, 2, 3))
res7: List[Int] = List(2, 3, 4)
Currying the map function

Scala:

```scala
def curriedMap[S,T](f: S => T) = (list: List[S]) => {
    list match {
        case Nil => Nil
        case head :: tail => f(head) :: map(f, tail)
    }
}
```

```scala
curriedMap[Int,Int](_+1)(List(1, 2, 3))
```

```
res8: List[Int] = List(2, 3, 4)
```
Currying the map function

Scala:

```scala
def curriedMap2[S,T](f: S => T)(list: List[S]) = {
  list match {
    case Nil => Nil
    case head :: tail => f(head) :: map(f, tail)
  }
}
```

```scala
scala> curriedMap2[Int,Int](_+1)(List(1, 2, 3))
res9: List[Int] = List(2, 3, 4)
```
Currying the map function

```scala
def curriedMap2[S,T](f: S => T)(list: List[S]) = {
  list match {
    case Nil => Nil
    case head :: tail => f(head) :: map(f, tail)
  }
}
```

Thus:

```scala
curriedMap2[Int,Int](_+1)
```

is a function that takes a `List[Int]` and adds 1 to each element of the list.
Currying is so common, and so useful, that its the default in most functional languages. Example: OCaml:

```ocaml
fun add x y = x + y
```

is syntactic shorthand for:

```ocaml
let add = fun x -> fun y -> x + y
```

not:

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

To call:

```ocaml
add 1 2
```

Not:

```ocaml
add(1,2)
```
Some functional languages

Lambda calculus
• Alonzo Church 1933
• a mathematical notation for describing all computation

LISP family
• LISP - John McCarthy 1958
• Scheme - Guy Steele 1980

ML family
• ML - Robin Milner late 1970s
• Standard ML (SML) - 1990
• Objective Caml (OCaml) - Xavier Leroy 1996
• F# - OCaml on .NET - Don Syme @MSR

Haskell
• Simon Peyton-Jones (SPJ) and others 1990s
LISP

First functional language
• John McCarthy 1958
“LISt Processing”
• key data structure is linked list (cons cell)
Core of the language is the lambda calculus

Introduced:
• first-class functions
  • `(lambda (x) (+ 1 x))`
• garbage collection
• eval – code is data

Many dialects: Common LISP, Emacs LISP, Autolisp, Scheme
Robin Milner @ Edinburgh late 1970s
• “Meta language”
  • “meta” = “about”
  • Designed for writing programs that manipulate other programs

Features:
• pattern matching
• statically typed
• type inference
• module system

• will discuss these later in the course

Dialects: SML, OCaml, F#
Haskell

Key features:

- Same core type system as ML
- Type classes (like interfaces, but better—or at least different)
- lazy evaluation
  - expressions not evaluated until used
  - more on this next week!
- pure functional
  - output of a function depends only on its input arguments
  - essential in a lazy language
  - no side-effects, no mutable state (i.e., no assignment)
  - but, *monads* are a clever way to do side-effects but stay pure
- ML (and other languages) are *impure* (functions can do I/O, do some limited assignment)
Scala

Scala is **not** a functional language!

- name refer to locations, not values

\[
\begin{align*}
\text{var } x &= 3 \\
 x &= x + 1
\end{align*}
\]

- the meaning of \(x\) changes as the program executes

But Scala supports *functional programming*

- first-class functions

Other imperative languages that support functional programming: Python, Ruby, Perl, C#
First-class functions in C#: delegates

// declare a delegate
delegate void Callback(string message);

// declare a method that takes a delegate
void callMeBack(Callback cb) {
    cb("hello");
}

// declare a method with the same type as the delegate
Callback handler = delegate (string m) {
    { System.Console.WriteLine(m); }
};

// call the method
callMeBack(handler);
First-class functions in Java

Kinda.

// fun square x = x*x

interface Apply1<S,T> {
    T apply(S arg);
}

new Apply1<Integer,Integer>() {
    public Integer apply(Integer x) { return x * x; }
}
First-class functions in C
First-class functions in C

Nope.
Second-class functions in C

Function pointers are **not** first class
- cannot create new functions at run-time
- (no lambda expressions)

• can only refer to existing functions
Variables are either *bound* or *free* in a given expression

<table>
<thead>
<tr>
<th>Expression</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>x</code></td>
<td><code>x</code> is free</td>
</tr>
<tr>
<td><code>fun x -&gt; x</code></td>
<td><code>x</code> is bound</td>
</tr>
<tr>
<td><code>fun x -&gt; x + y</code></td>
<td><code>x</code> is bound, <code>y</code> is free</td>
</tr>
</tbody>
</table>

A function can *capture* a free variable in its scope

<table>
<thead>
<tr>
<th>Expression</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>fun x -&gt; x + y</code></td>
<td>captures <code>y</code>, but not <code>x</code></td>
</tr>
<tr>
<td><code>fun y -&gt; fun x -&gt; x + y</code></td>
<td>does not capture <code>y</code> or <code>x</code></td>
</tr>
</tbody>
</table>
Closures

Aka “lambda”, “anonymous function”
Object that represents a function at run time
- first-class
- does not necessarily have a name
  - fun x -> x + 1
- but can be bound to a name
  - let add1 = fun x -> x + 1

Closures:
- capture free variables in their context
- closure is represented as a pair: (environment, code)
let x = 1
let y = 2
fun a -> a*x + y

Environment:

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>x</td>
<td>1</td>
</tr>
<tr>
<td>y</td>
<td>2</td>
</tr>
</tbody>
</table>

Closure:

- env
- code

Note: since names refer to values, not locations, the environment is constant

compiled("fun a -> a*env.x + env.y")
Lambda calculus

Simple functional language
• core of all functional programming languages
• Church 1933

Only 3 constructs:
• variables \( x \)
• functions (abstractions) \( \lambda x. e \)
  • a function that has a single formal parameter \( x \) and a body \( e \)
• function application \( (e_1 e_2) \)
  • invokes function \( e_1 \) on argument \( e_2 \)

The only values are functions
• Can encode all other languages in the lambda calculus
Some words on syntax

The body $e$ of $\lambda x. e$ extends as far right as possible

- $\lambda x. \lambda y. y x \quad == \quad \lambda x. (\lambda y. (y x))$
- $\lambda x. ((\lambda y. y) x)$

Function application is left associative:

- $x y z \quad == \quad (x y) z$
- $x (y z)$
Function application

Beta reduction
• \((\lambda x. e_1) \ e_2 \rightarrow e_1[e_2/x]\)

Read \(e_1[e_2/x]\) as “\(e_1\) with \(e_2\) for \(x\)”
• replace free occurrences of \(x\) in \(e_1\) with \(e_2\)
• (will define this more precisely next week)

Identity function
• \(\lambda x. x\)
• \((\lambda x. x) \ e \rightarrow x[e/x] = e\)

Omega
• \((\lambda x. x \ x) (\lambda x. x \ x) \rightarrow (\lambda x. x \ x) (\lambda x. x \ x)\)
Lambda calculus encodings

Booleans

• true = \lambda x. \lambda y. x
• false = \lambda x. \lambda y. y
• if = \lambda x. \lambda y. \lambda z. x y z

How does this work?

if true M N
\((\lambda x. \lambda y. \lambda z. x y z)\) true M N
\((\lambda y. \lambda z. true y z)\) M N
\((\lambda z. true M z)\) N
true M N
\((\lambda x. \lambda y. x)\) M N
\((\lambda y. M)\) N
M

if false M N
\((\lambda x. \lambda y. \lambda z. x y z)\) false M N
\((\lambda y. \lambda z. false y z)\) M N
\((\lambda z. false M z)\) N
false M N
\((\lambda x. \lambda y. y)\) M N
\((\lambda y. y)\) N
N
Lambda calculus encodings

Numbers (Church numerals)

- \(0 = \lambda f. \lambda x. x\)  
  (note: same as false!)
- \(1 = \lambda f. \lambda x. f x\)
- \(2 = \lambda f. \lambda x. f (f x)\)
- \(3 = \lambda f. \lambda x. f (f (f x))\)
- ... 
- \(n = \lambda f. \lambda x. \text{"apply f to x n times"}\)

\[
\begin{align*}
\text{add1} & = \lambda n. \lambda f. \lambda x. f (n f x) \quad \text{adds 1 to n} \\
\text{add1 1} & = (\lambda n. \lambda f. \lambda x. f (n f x)) 1 \rightarrow \lambda f. \lambda x. f (1 f x) \\
& \Rightarrow \lambda f. \lambda x. f ((\lambda f'. \lambda x'. f' x') f x) \\
& \Rightarrow \lambda f. \lambda x. f (f x) = 2
\end{align*}
\]
Lambda calculus encodings

- Can encode add, subtract, multiply, exponentiation, etc.
- Can encode data structures: pairs, lists, etc.

- So: when necessary, can assume the language contains these features
- (remember the lambda calculus is a formalism, not a practical programming language)
Church-Turing Thesis

Alonzo Church 1936:
• any decidable problem can be computed with the lambda calculus

Alan Turing 1936:
• any decidable problem can be computed with a Turing machine

J.B. Rosser 1939, Stephen Kleene 1943:
• Turing machines and lambda calculus are equivalent
Questions?