

# Deconstructing OCaml

What makes up a language

# Key components of a lang

---

- Units of computation
- Types
- Memory model

# Units of computation

# In OCaml

---

# In OCaml

---

- Expressions that evaluate to values
- Everything is an expression
  - int, bool, real
  - if-then-else
  - let-in
  - match
  - `fn x -> x+1`
  - `e1 e2`

# In Java/Python

---

# In Java/Python

---

- Store and update commands
- Message sends

# In Prolog

---



# In Prolog

---

- Logical facts
- Inference rules

**Mexican(CARNITAS) “Fact”**

**Food(CARNITAS) “Fact”**

**Mexican(X)  $\wedge$  Food(X)  $\Rightarrow$  Delicious(X) “Rule”**

---

**Delicious(CARNITAS) “Fact”**

# Types

# Types

---

- Used to classify things created by the programmer
- Classification used to check what can be done with/to those things

# In OCaml: Static typing

---

- Types are assigned statically at compile time
- Without computing values
- Rules state when expressions are type-correct

$$\frac{e1 : T1 \rightarrow T2 \quad e2 : T1}{e1 \ e2 : T2}$$

# In OCaml: Static typing

---

- How can one reuse code for different types?
  - parametric types: `'a * 'b -> 'b * 'a`
  - implicit forall
- Type “discovered” (inferred) automatically from code
  - less burden on the programmer

# In Python: Dynamic typing

---

- Types assigned to values/objects as they are computed, ie: dynamically
- Before an operation is performed, check that operands are compatible with operation

# In Python: Dynamic typing

---

- More programs are accepted by compiler
- More flexible, but find errors late

```
[1, "abc", 1.8, [ "efg", 20]]
```

```
let x = if b then 1 else "abc"
```

```
let y = if b then x + 1 else x ^ "efg"
```

# Dynamic vs. Static, OO vs. Func

---

	Statically typed	Dynamically typed
OO		
Functional		



# Dynamic vs. Static, OO vs. Func

---

	Statically typed	Dynamically typed
OO	Java	Python, Smalltalk
Functional	Ocaml, Haskell	Lisp/Scheme

# Polymorphism

---

- Can a language be dynamically typed, but not polymorphic?
- Every dynamically typed language is polymorphic
  - functions just simply work on any datatype that can be operated on at runtime
- Only need explicit polymorphism in statically typed languages to assign at compile time a suitably general polymorphic type

# Memory/Data model

aka: what do variables refer to?

# Data model in functional langs

---

- Environment of bindings (phonebook)

x	3
y	"abc"
z	[1;2;3]

- Never change a binding
  - add new bindings at the end of the phonebook

# Data model in functional langs

---

- Variables are names that refer into the phonebook
- Most recent entry looked up during evaluation
- Environment “frozen” inside function value so that the behavior of the function cannot be changed later on (easier reasoning)

# Data model in OO langs

---

- Variables are cells in memory
- Can change them by assigning into them
- Variables point to objects on the heap
- $x = x + 10$

# Data model in Prolog

---

- Variables in Prolog are unknowns to solve for

**Mexican(CARNITAS)**

**Food(CARNITAS)**

**$\forall X \text{ Mexican}(X) \wedge \text{Food}(X) \Rightarrow \text{Delicious}(X)$**

**Delicious(Y)?**

**Q: What is delicious?**

**A: CARNITAS!**

# Final words on functional programming



What's the point of all this?

---

# Advantages of functional progs

---

- Functional programming more concise  
“one line of lisp can replace 20 lines of C”  
(quote from <http://www.ddj.com/dept/architect/184414500?pgno=3>)

- Recall reverse function in OCaml:

```
let reverse = fold (::) [];;
```

- How many lines in C, C++?

# Don't be fooled

---

- Some of the programming assignments made you do certain things using fold in order to force you to think about it, even though using fold was not the easiest way to do it.
- But there are many cases where map and fold make life A LOT EASIER.

# Can better reason about progs

---

- No side effects. Call a function twice with same params, produces same value
- As a result, computations can be reordered more easily
- They can also be parallelized more easily

# So what?

---

- From the authors of map reduce:  
“Inspired by similar primitives in LISP and other languages”

<http://research.google.com/archive/mapreduce-osdi04-slides/index-auto-0003.html>

- The point is this: programmers who only know Java/C/C++ would probably not have come up with this idea
- Many other similar examples in industry

# This stuff is for real: F#

---

F# = Microsoft's Ocaml-on-steroids

<http://channel9.msdn.com/pdc2008/TL11/>

- Why FP is way cool
- How FP works with Objects (C#)
- How FP allows you to write parallel code  
... all with an extremely engaging speaker

# Remember

---

- The next time you use google, think of how functional programming has inspired some of the technical ideas behind their engine
- And of course:  
*“Free your mind”*  
-Morpheus

# Recap of the course so far

---

- 4 weeks of functional with Ocaml
- **Next: 4 weeks of OO with Python**
- After that: 1 week of constraint logic programming with Prolog



# OO at the highest level

---

- What is OO programming?

# OO at the highest level

---

- What is OO programming?
- Answer:
  - objects
  - message sends
  - dynamic dispatch

# Just to whet your appetite

---

- Say we have objects, like **cars**, **ducks**, **pig**, **cell\_phones**
- Say we have a message name: **make\_some\_noise**

# Just to whet your appetite

---

- Each object has its own implementation for **make\_some\_noise**: these are traditionally called methods.
- **car**: vroom vroom, **pig** : oink oink, **duck**: quack quack
- We can send **make\_some\_noise** to any object. Depending on the actually run-time object, we'll get a different noise!

# OO programming

---

- **Message:** the name of an operation
- **Method:** the implementation of an operation
- **Dynamic dispatch:** the act of determining at based on the dynamic type which method should be run for a given message send.
- These are the core ideas of OO

# This brings us to Python...

---

- We'll use Python as our vehicle for OO programming
- Fun and useful language
- Let's compare with OCaml along some of the dimensions we saw last time

# OCaml/Python comparison

---

	ML	Python
PL paradigm		
Basic unit		
Types		
DataModel		

# OCaml/Python comparison

---

	ML	Python
PL paradigm	functional	OO/imperative
Basic unit	Expr/value	Objects/ messages
Types	statically	dynamicacclly
DataModel	env lookup	“pointers” to mutable objs



# Python

---

- Python has a very relaxed philosophy
  - if something "can be done" then it is allowed.
- Combination of dynamic types + everything is an object makes for very flexible, very intuitive code.

# No static types

---

- **No static type system** to "prohibit" operations.
- No more of that OCaml compiler giving you hard-to-decypher error messages!
- And... No need to formally define the type system (although still need to define the dynamic semantics somehow)

# No static types: but what instead?

---

- **Dynamic typing**
- At runtime, every "operation" is translated to a method call on the appropriate object.
- If the object supports the method, then the computation proceeds.
- Duck-typing: if it looks like a duck, quacks like a duck, then it is a duck!

# Dynamic typing

---

- This loose, comfortable, free-style, philosophy is at the heart of python.
- But... beware, can get burned...
- One way to think about it:
  - Dynamic types good for quick prototyping
  - Static types good for large systems
  - Although...
  - Gmail in Javascript?

# Similarities to Ocaml

---

- Uniform model: everything is an object, including functions
- Can pass functions around just as with objects
- Supports functional programming style with map and fold

# Other cool things about Python

---

- A lot of stuff that you may first think is a "language feature" is actually just translated under the hood to a method call...
- Very widely used, supported.
- Has libraries for all sorts of things.

# Ok, let's start playing with Python!

---

- Like Perl, python is a "managed" or "interpreted" language that runs under the python environment, i.e. not compiled to machine code.
- Makes it convenient to rapidly write and test code!

# Ways to run Python code

---

- At an interactive Python prompt: like "read-eval-print" loop of ML,
- As shell scripts,
- As stand-alone programs run from the shell.



# Let's fire it up!

---

- Ok, let's give it a try...
- See py file for the rest...