The material on these slides is based on notes provided by Dexter Kozen.
About OCaml

- A functional programming language
  - All computation is done with functions
  - No state (mutable variables), simple and clean
  - Functions are first class: you can pass them, return them, etc.
  - Support for object-oriented programming (the O in OCaml)
- Statically typed, type-safe language
  - You can’t apply the wrong operations to the wrong data (e.g., can’t try to divide two strings)
  - Avoids many “silly errors” and provides security (e.g., no buffer overflows)
Ocaml Features

- **Garbage collection**: automatic memory management
- **Type inference**: You don’t have to write the type information down. The compiler will figure it out.
- **Parametric polymorphism**: Write functions and data structures that can be used with any type. Note that Java provides *subtype polymorphism*.
- **Algebraic data types**: Can build data structures easily. *Pattern matching* makes working with them convenient.
- **Advanced modules**: Encapsulate implementations behind interfaces. *Functors* (functions that manipulate modules) add functionality beyond most languages’ module systems.
Weak/Strong, Static/Dynamic Typing

Weak typing (implicit conversions)

Strong typing (explicit conversions)

Static typing (compile-time checks)

Dynamic typing (runtime checks)

C

ML

Java

VBA

Scheme

Assembler

Perl

JavaScript

PHP
History

- Robin Milner and others at the University of Edinburgh were working on theorem provers.
- Problem: the provers would sometimes put incorrect “proofs” together, and claim they were valid.
- Designed ML (Meta Language) as a language to let you construct valid proofs.
- In the early ‘80s, there was a schism in the ML community.
  - French developed Caml and later Objective Caml (OCaml).
  - British and Americans developed Standard ML (SML).
OCaml

How to Write + Run Code

• ocaml
  • REPL

• ocaml file.ml
  • Run the interpreter

• ocamlc file.ml; ./a.out
  • Run the compiler
OCaml

Types

- primitive
  - int
  - float
  - bool
  - char
  - string
  - unit

- constructed
  - lists
  - tuples
  - functions
Type Inference

- The compiler deduces the type of an expression
  \( y = x + 1 \)
- Compiler knows that + takes two integers and returns an integer, so \( x \) and \( y \) should be integers.
  \( z = x + 1.0 \)
- This should produce an error, since \( x \) is being used as both an int and a float
- Compiler infers types by aggregating type information, and reducing expressions to implicitly typed values
Type Inference

- Statically determining whether a program will have a type-error is impossible
- All statically typed languages are conservative, and may reject some programs that are perfectly okay
- Milner formulated the type-inference system for ML, and proved its soundness
- Note that Milner also worked on concurrent programming languages (CCS, CSP, pi-Calculus), and won the Turing award – in large part to his work on ML
# OCaml

## Syntax (subset)

<table>
<thead>
<tr>
<th>Syntactic class</th>
<th>Grammar rule</th>
<th>Example</th>
</tr>
</thead>
<tbody>
<tr>
<td>identifiers</td>
<td>$x, f$</td>
<td>a, x, y, x_y, z100…..</td>
</tr>
<tr>
<td>constants</td>
<td>$c$</td>
<td>1, 01, 1.0, true, “hello” ‘A’</td>
</tr>
<tr>
<td>unary operator</td>
<td>$u$</td>
<td>-, not</td>
</tr>
<tr>
<td>binary operator</td>
<td>$b$</td>
<td>+, *, -, &lt;, &gt;, ^, !, =,...</td>
</tr>
<tr>
<td>terms</td>
<td>$e ::= x</td>
<td>c</td>
</tr>
<tr>
<td>declarations</td>
<td>$t ::= x = e</td>
<td>f (x, …, x) : t = e$</td>
</tr>
</tbody>
</table>
Program Errors

• An expression can have a legal syntax, but may be wrong. The expression must be well-typed.

• Ways that a program can be wrong:
  – Syntax errors: let 0 x =
  – Type errors: “x” + 3
  – Semantic errors: 1/0
  – More general errors: computes the wrong output
let abs (x : int) : int =  
   if x < 0 then -x else x

let abs : int -> int =  
   function x -> if x < 0 then -x else x

let abs = fun x -> if x < 0 then -x else x

val abs : int -> int = <fun>

• Every expression and declaration has both a type and a value.
• Here, we have bound the name abs to a function whose type is int -> int.
Type Checking

$ ocaml
# let abs = fun x -> if x < 0 then -x else x ;;
val abs : int -> int = <fun>

# abs 3;;
- : int = 3

# abs "foo";;
Error: This expression has type string but an expression was expected of type
     int
Scope

• Variable declarations **bind** variables within a **scope**

```
let x = e1 in e2
```

• The scope of x is the expression e2.

• Functions also bind variables

```
let f x = e1 in e2
```

The scope of the formal parameter x is the expression in e1. The scope of the variable f (which is bound to a function value) is the body of the let, e2.
Scope

- A let expression can introduce multiple variables

```ocaml
let x = 2 and y = 3 in x + y
```

- To declare a recursive function, the function must be in scope in its own body. In Ocaml, you use let rec instead of let for this.

```ocaml
let rec even x = x = 0 || odd (x-1)
    and odd x = not (x = 0 || not (even (x-1)))
in
odd 3110
```
Curried functions

- A function with multiple parameters is really just syntactic sugar for a function passed a tuple as an argument

```ocaml
let plus (x, y) = x + y;;
val plus : int * int -> int = <fun>
plus (2, 3)
```

- Ocaml has another way to declare functions with multiple arguments. In curried form:

```ocaml
let plus x y = x + y
```

- Notice there are no commas between the parameters
Curried functions

• There are also no commas when calling the function:

plus 2 3

• Functions *really* only have one argument. The plus function is being passed one argument, 2, and it returns a function that takes one argument. (plus 2) (3)

\[
\text{plus 2 3} \\
= ((\text{function } (x : \text{int}) \rightarrow \text{function } (y : \text{int}) \rightarrow x + y) 2) 3 \\
= (\text{function } (y : \text{int}) \rightarrow 2 + y) 3 \\
= 2 + 3 \\
= 5
\]
Lists

• Lists are immutable. You cannot change the elements of a list.
• Lists are homogenous. The can only contain one type.
• Examples: [1;2;3], 1::[2;3], [1;2]@[3;4]
Pattern Matching

- Use *match expressions* to extract elements from a list

```ocaml
match lst with
| []  -> 0
| [x] -> 1
| _   -> 2
```

- Returns 0 if the list is empty, 1 if the list has 1 element, and 2 if it has 2 or more elements
Pattern Matching

• Use recursive functions to do something to every element in a list

```
let rec length (lst : string list) : int =
  match lst with
  | [] -> 0
  | h :: t -> 1 + length t
```

• This function computes the length of the list lst.
Variant Types

- Allows you to have variables that contain more than one kind of value
- Similar to an enum in Java, or a tagged union in C

```ocaml
type answer = Yes | No | Maybe;;

- Type keyword declared name for the type.
- Declared with a set of `constructors`

```ocaml

type eitherPoint = TwoD of float * float
| ThreeD of float * float * float
```
Pattern Matching

- Ocaml patterns are very rich
- Can match on variant types, tuples, records, and can pull the values apart

```ocaml
let answer_to_string (a : answer) : string =
    match a with
    | Yes -> "yes"
    | No -> "no"
```
Polymorphism

• Suppose we want to write a function that swaps positions of a pair:

```ocaml
let swapInt ((x : int), (y : int)) : int * int = (y, x)
and swapReal ((x : float), (y : float)) : float * float = (y, x)
and swapString ((x : string), (y : string)) : string * string = (y, x)
```

• Better way:
  ```ocaml
  let swap ((x : 'a), (y : 'b)) : 'b * 'a = (y, x);;
  ```

• Or without type declarations:
  ```ocaml
  # let swap (x, y) = (y, x);;
  val swap : 'a * 'b -> 'b * 'a = <fun>
  ```
Mapping

• Apply a function to every element in a list, and return a new list:

List.map f [a; b; c] = [f a; f b; f c]

• Copy a list (using an anonymous function)

let copy = map (fun x -> x)
Folding

- Apply a function to every element in a list, accumulates a result:

\[
\text{fold}_\text{right} \ f \ [a; \ b; \ c] \ r = f \ a \ (f \ b \ (f \ c \ r)) \\
\text{fold}_\text{left} \ f \ r \ [a; \ b; \ c] = f \ (f \ (f \ r \ a) \ b) \ c
\]

- Sum all the elements in a list:

\[
\begin{align*}
\text{let sum}_\text{right}_\text{to}_\text{left} \ il &= \text{fold}_\text{right} \ (+) \ il \ 0 \\
\text{let sum}_\text{left}_\text{to}_\text{right} &= \text{fold}_\text{left} \ (+) \ 0
\end{align*}
\]
Folding

- Fold is very powerful!
- Can write many other list functions in terms of fold

```ocaml
type intlist = Nil | Cons of (int * intlist)
let mapp f lst = fold_right (fun x y -> Cons (f x, y)) lst Nil
```

- Note that fold_left would give the result in reverse order

```ocaml
let length = fold_left (fun x _ -> 1 + x) 0
```

- Select a subset of the list

```ocaml
let filter f lst =
  fold_right (fun x y -> if f x then Cons (x, y) else y) lst Nil
```
Last Slide

• Next lecture: Review!