Dialects of ML

- **ML** (Meta Language)
  - Univ. of Edinburgh, 1973
  - Part of a theorem proving system LCF
    - The Logic of Computable Functions
- **SML/NJ** (Standard ML of New Jersey)
  - Bell Labs and Princeton, 1990
  - Now Yale, AT&T Research, Univ. of Chicago, etc...
- **OCaml** (Objective CAML)
  - INRIA, 1996
  - French Nat'l Institute for Research in Computer Science

Other dialects

- MoscowML, ML Kit, Concurrent ML, etc...
- SML/NJ and OCaml are most popular

Languages all have the same core ideas

- But small and annoying syntactic differences
- So you should not buy a book with ML in the title
  - Because it probably won’t cover OCaml

Features of ML

- “Mostly functional”
  - Some assignments
- Higher-order functions
  - Functions can be parameters and return values
- Type inference
  - No need to write types in the source language
    - But the language is statically typed
  - Supports parametric polymorphism
    - Generics in Java, templates in C++

Features of ML (cont.)

- Data types and pattern matching
  - Convenient for certain kinds of data structures
- Exceptions
- Garbage collection

Functional Languages

- In a pure functional language
  - Every program is just an expression evaluation
  - Example code:
    ```
    let add1 x = x + 1;;
    let rec add (x,y) = if x=0 then y else add(x-1, add1(y));;
    add(2,3) = add(1,add1(3)) = add(0,add1(add1(3)))
    = add1(add1(3)) = add1(3+1) = 3+1+1 = 5
    ```
OCaml has similar basic behavior
- Program = expression evaluation

But has additional features
- To ease the programming process
- Features support
  - Less emphasis on data storage
  - More emphasis on function execution

OCaml Interpreter
Expressions can be typed and evaluated at the top-level

- `# 3 + 4 ;`
- `val x : int = 7 ;`
- `val y : int = 37 ;`
- `val z : int = 42 ;`

File types and values of each expr

- `# use "ocaml" ;`
- `val x : int = 37 ;`
- `val y : int = 5 ;`
- `val z : int = 42 ;`

No type declarations

- `val x : int = 37 ;`

Run, OCaml, Run
- OCaml programs can be compiled using `ocaml`
  - Produces `.cmo` ("compiled object") and `.cmi` ("compiled interface") files
  - We'll talk about interface files later
  - By default, also links to produce executable `a.out`
    - Use `-o` to set output file name
    - Use `-c` to compile only to `.cmo/.cmi` and not to link
    - You can use a Makefile if you need to compile your files

- Not needed for this course

OCaml Interpreter (cont.)
Files can be loaded at top level

- `# use "ocaml" ;`
- `val x : int = 37 ;`
- `val y : int = 5 ;`
- `val z : int = 42 ;`

- `# use "ocaml" ;`
- `val x : int = 37 ;`
- `val y : int = 5 ;`

- `# use "ocaml" ;`
- `val x : int = 37 ;`

OCaml Interpreter (cont.)
Files can be loaded at top level

- `# use "ocaml" ;`
- `val x : int = 37 ;`
- `val y : int = 5 ;`
- `val z : int = 42 ;`

- `# use "ocaml" ;`
- `val x : int = 37 ;`
- `val y : int = 5 ;`

- `# use "ocaml" ;`
- `val x : int = 37 ;`
This Lecture

- Basic OCaml types
- Defining variables & functions with `let`
- Lists
- Pattern matching with `match`

Basic Types in OCaml

- Read `e : t` as “expression e has type t”
  42 : int
  3.14 : float
  "hello" : string
  'c' : char
  () : unit (* don't care value *)

- OCaml has static types to help you avoid errors
  - Note: Sometimes the messages are a bit confusing
  - This expression has type bool but is here used with type int
  - Watch for the underline as a hint to what went wrong
  - But not always reliable

The Let Construct

- `let` is often used for defining local variables
  - `let x = e1 in e2` means
    - Evaluate e1
    - Then evaluate e2, with x bound to result of evaluating e1
  - x is not visible outside of e2

```
let pi = 3.14 in pi *. 3.0 *. 3.0;
pi;
```

- `; ;` versus `;`
  - `; ;` ends an expression in the top-level of OCaml
    - Use it to say: “Give me the value of this expression”
    - Not used in the body of a function
    - Not needed after each function definition
      - Though for now it won't hurt if used there
  - `e1; e2` evaluates e1 and then e2, and returns e2
    - Notice no ; at end
    - ; is a separator, not a terminator
    - Invoking `p (1,2)`
      - Prints "1 2"
    - Returns "Done!"

Examples – Semicolon

- Definition
  - `e1 : e2` (* evaluate e1, evaluate e2, return e2*)
  - `1 ; 2 ;`
    - (* 2 – value of 2nd expression is returned *)
  - `(1 + 2) ; 4 ;`
    - (* 4 – value of 2nd expression is returned *)
  - `1 + (2 ; 4) ;`
    - (* 5 – value of 2nd expression is returned to 1 + *)
  - `1 + 2 ; 4 ;`
    - (* 4 – because + has higher precedence than ; *)
Nested Let

- Uses of let can be nested

```
let pi = 3.14 in
let r = 3.0 in
  pi *. r *. r;
(* pi, r no longer in scope *)

{ float pi = 3.14;
  float r = 3.0;
  pi *. r *. r;
}
/* pi, r not in scope */
```

Local Variables

- You can use let inside of functions for locals

```
let area r =
  let pi = 3.14 in
    pi *. r *. r
```

- And you can use as many lets as you want

```
let area d =
  let pi = 3.14 in
    let r = d /. 2.0 in
      pi *. r *. r
```

Defining Functions

- Use let to define functions

```
let next x = x + 1;;
next 3;;
let plus (x, y) = x + y;;
plus (3, 4);;
```

- List parameters after function name

```
let pi = 3.14 in
let r = d /. 2.0 in
pi *. r *. r
```

Function Types

- In OCaml, \( ightarrow \) is the function type constructor
  - The type \( \text{t1} \rightarrow \text{t2} \) is a function with argument or domain type \( \text{t1} \) and return or range type \( \text{t2} \)

- Examples
  - let next x = x + 1 (* type int \( \rightarrow \) int *)
  - let fn x = (float_of_int x) *. 3.14 (* type int \( \rightarrow \) float *)
  - print_string (* type string \( \rightarrow \) unit *)

- Type a function name at top level to get its type

Lists in OCaml

- The basic data structure in OCaml is the list
  - Lists are written as [e1; e2; ...; en]
    # [1;2;3]
    : int list = [1;2;3]
  - Notice type of list is int list
    ➢ Lists must be homogeneous

Lists in OCaml (cont.)

- More on OCaml lists
  - The empty list is []
    # []
    _ : `a list
  - The `_` means "a list containing anything"
    ➢ We’ll find out more about this later
  - List elements are separated using semicolons
  - Warning: Don’t use a comma instead of a semicolon
    ➢ Means something different (we’ll see in a bit)
Consider a Linked List in C

```c
struct list {
    int elt;
    struct list *next;
};
```

i = 0;
...
while (l != NULL) {
    l = l->next;
    i++;
}

Lists in OCaml are Linked

- `[1;2;3]` is represented above
  - A nonempty list is a pair (element, rest of list)
  - The element is the head of the list
  - The pointer is the tail or rest of the list
  - ...which is itself a list!

Thus in math a list is either
- The empty list `[]`
- Or a pair consisting of an element and a list
  - This recursive structure will come in handy shortly

Lists are Linked (cont.)

- :: prepends an element to a list
  - :: is the list with h as the element at the beginning and t as the "rest"
  - :: is called a constructor, because it builds a list
  - Although not emphasized, :: does allocate memory

Examples

- 3::[] (* The list [3]*)
- 2::(3::[]) (* The list [2; 3]*)
- 1::(2::(3::[])) (* The list [1; 2; 3]*)

Lists of Lists

- Lists can be nested arbitrarily
  - Example: `[ [9; 10; 11]; [5; 4; 3; 2] ]`
    - Type = int list

Practice

- What is the type of
  - `[1;2;3]` `int list`
  - `[[1;2;3];[1;2;4]]` `float list list`
  - `let func x = x::(0::[])` `int -> int list`
Pattern Matching

- To pull lists apart, use the `match` construct
  \[
  \text{match } e \text{ with } p_1 \rightarrow e_1 | \ldots | p_n \rightarrow e_n
  \]
- `p_1\ldots p_n` are patterns made up of
  - `[]`, `\_`, and pattern variables
- `match` finds the first `p_k` that matches shape of `e`
  - Then `e_k` is evaluated and returned
  - During evaluation of `p_k`, pattern variables in `p_k` are bound to the corresponding parts of `e`

Pattern Matching Example

- Match syntax
  \[
  \text{match } e \text{ with } p_1 \rightarrow e_1 | \ldots | p_n \rightarrow e_n
  \]
- Code 1
  \[
  \text{let } is\_empty \ 1 \ = \ \text{match } l \text{ with}
  \[
  [] \rightarrow \text{true}
  | (h::t) \rightarrow \text{false}
  \]
- Outputs
  - `is_empty []` (*evaluates to true*)
  - `is_empty [1]` (*evaluates to false*)
  - `is_empty [1;2]` (*evaluates to false*)

Pattern Matching Example (cont.)

- Code 2
  \[
  \text{let } \text{hd} \ 1 \ = \ \text{match } l \text{ with } (h::t) \rightarrow h
  \]
- Outputs
  - `hd [1;2;3]` (*evaluates to 1*)
  - `hd [1;2]` (*evaluates to 1*)
  - `hd [1]` (*evaluates to 1*)
  - `hd []` (*Exception: Match failure*)

Pattern Matching – Wildcards

- An underscore `_` is a wildcard pattern
  - Matches anything
  - Doesn’t add any bindings
  - Useful when you want to know something matches
    - But don’t care what its value is
- In previous examples
  - Many values of `h` or `_` ignored
  - Can replace with wildcard `_`
  - Code behavior is identical

Pattern Matching – Wildcards (cont.)

- Code using `_`
  \[
  \text{let } is\_empty \ 1 \ = \ \text{match } l \text{ with}
  \[
  [] \rightarrow \text{true}
  | (\_::\_) \rightarrow \text{false}
  \]
  \[
  \text{let } \text{hd} \ 1 \ = \ \text{match } l \text{ with } (h::t) \rightarrow h
  \]
  \[
  \text{let } t l \ 1 \ = \ \text{match } l \text{ with } (_::t) \rightarrow t
  \]
- Outputs
  - `is_empty[1]` (*evaluates to false*)
  - `is_empty[ ]` (*evaluates to true *)
  - `hd [1;2;3]` (*evaluates to 1 *)
  - `tl [1;2;3]` (*evaluates to [2;3]*)
  - `hd [1]` (*evaluates to 1 *)
  - `tl [1]` (*evaluates to [ ]*)
  - `tl []` (*Exception: Match failure*)
Pattern Matching – Missing Cases

- When pattern is defined
  - OCaml will warn you about non-exhaustive matches
- When pattern is used
  - Exceptions for inputs that don’t match any pattern
- Example
  ```ocaml
  let hd l = match l with (h::_) -> h;
  Warning: this pattern-matching is not exhaustive.
  Here is an example of a value that is not matched:
  []
  # hd [];;
  Exception: Match_failure ("", 1, 11).
  ```

Pattern Matching – An Abbreviation

- let f p = e, where p is a pattern
  - is shorthand for let f x = match x with p -> e
- Examples
  - let hd (h::_) = h
  - let tl (_::t) = t
  - let f (x::y::_::_) = x + y
  - let g [x; y] = x + y
- Useful if there’s only one acceptable input

Pattern Matching – Lists of Lists

- Can pattern match on lists of lists as well
- Examples
  ```ocaml
  let addFirsts
  ((x::_) :: (y::_) :: _) = x + y
  addFirsts [[1;2];[4;5];[7;8;9]] = 5
  let addFirstSecond
  (x::_;::(y::_;::_)) = x + y
  addFirstSecond [[1;2];[4;5];[7;8;9]] = 6
  ```
- Note – you probably won’t do this much or at all
  - You’ll mostly write recursive functions over lists instead