Last time

- NFA -> DFA
- DFA minimization
- DFA complement

This time

- 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 (contd)

- 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

```
let add1 x = x + 1;;
let rec add (x,y) = if x=0 then y
                    else add(x-1, add1(y));;
add(2,3) = add1(add1(3)) =
          add(0,add1(add1(3))) =
          add1(add1(3)) =
          add1(3+1) = 3+1+1 = 5
```
Functional Languages (cont.)

- 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

A Small OCaml Program – Things to Notice

Use let to bind variables
No type declarations
Use (*) for comments (may nest)

Usage of basic commands:
- (* Small OCaml program *)
- let x = 37;
- let y = x + 5;
- print_int y;;
- print_string "\n";;

Line breaks, spacing ignored (like C, C++, Java, not like Ruby)
;; ends a top-level expression

Running Ocaml

- OCaml programs can be compiled using ocamlc
  - 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

Running Ocaml (cont.)

• Compiling & running the previous small program
  % ocamlc ocaml1.ml
  % ./a.out
  42

  (* Small OCaml program *)
  let x = 37;;
  let y = x + 5;;
  print_int y;;
  print_string "\n";;

• Files can be loaded at top level
  % ocaml
  Objective Caml version 3.08.3
  # use "ocaml1.ml";;
  val x : int = 37
  val y : int = 42
  42- : unit = ()

  # use loads in a file one line at a time
  # use "ocaml1.ml";;
  val x : int = 37
  val y : int = 42
  42- : unit = ()
  # x;
  - : int = 37

  This expression has type string but is here used with type int

Running Ocaml (contd)

Expressions can also be typed and evaluated at the top-level

# 3 + 4;;
- : int = 7
# let x = 37;;
val x : int = 37
# x;;
- : int = 37
# let y = 5;;
val y : int = 5
# let z = 5 + x;;
val z : int = 42
# print_int z;;
42- : unit = ()
# print_string "Colorless green ideas sleep furiously";;
Colorless green ideas sleep furiously- : unit = ()
# print_int "Colorless green ideas sleep furiously";;
This expression has type string but is here used with type int

Running Ocaml (cont)
Basic Types in Ocaml

- Read \( e : \! \! t \) as "expression \( e \) has type \( t \)"
  - 42 : int
  - true : bool
  - "hello" : string
  - 'c' : char
  - 3.14 : float
  - () : unit
- 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

More on the \texttt{let} construct

- \texttt{let} is more often used for local variables
  - \texttt{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 \( e2 \)

- \texttt{let pi = 3.14 in pi *. 3.0 *. 3.0;;}
  - \texttt{pi;;}

More on the \texttt{let} Construct (cont.)

- Compare to similar usage in Java/C
  - \texttt{let pi = 3.14 in}
  - \texttt{pi *. 3.0 *. 3.0;;}
  - \texttt{pi;;}
  - In the top-level, omitting in means “from now on”
  - \# let pi = 3.14;;
  - (* \( pi \) is now bound in the rest of the top-level scope *)

Nested \texttt{let}

- Uses of \texttt{let} can be nested

- \texttt{let pi = 3.14 in}
  - \texttt{let r = 3.0 in}
  - \texttt{pi *. r *. r;;}
  - (* \( pi \), \( r \) no longer in scope *)
  - \texttt{\{ float pi = 3.14; \}
  - \texttt{\{ float r = 3.0; \}
  - \texttt{pi * r * r; \}
  - \texttt{\} \}
  - (* \( pi \), \( r \) not in scope *)

Defining functions

- Use \texttt{let} to define functions
- List parameters after function name
- \texttt{let next x = x + 1;;}
- \texttt{next 3;;}
- \texttt{let plus (x, y) = x + y;;}
- \texttt{plus (3, 4);;}

Local Variables

- You can use \texttt{let} inside of functions for locals

- \texttt{let area r =}
  - \texttt{let pi = 3.14 in}
  - \texttt{pi *. r *. r}"

- And you can use as many \texttt{let} as you want

- \texttt{let area d =}
  - \texttt{let pi = 3.14 in}
  - \texttt{let r = d /. 2.0 in}
  - \texttt{pi *. r *. r}"

13

14

15

16

17

18
Function Types

• In OCaml, \rightarrow \text{ 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 -> int *)
  – let fn x = (float_of_int x) *. 3.14 (* type int -> float *)
  – print_string (* type string -> unit *)

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

Type annotations

• The syntax \text{(e : t)} asserts that \text{e} has type \text{t}
  – This can be added anywhere you like
  – Not used in the body of a function
  – Though for now it won’t hurt if used there

• \text{e1;e2} evaluates \text{e1} and then \text{e2}, and returns \text{e2}
  – Notice no ; at end
  – ; is a separator, not a terminator
  – Invoking \text{p (1,2)}
    – Prints “1 2”
    – Returns “Done!”

Lists in OCaml

• The basic data structure in OCaml is the list
  – Lists are written as \text{[e1; e2; ...; en]}
  – Lists must be homogeneous

Lists in OCaml (cont.)

• More on OCaml lists
  – The empty list is \text{[]} \# [ ]
  – The \text{'}a\text{ list} means “a list containing anything”
  – We’ll find out more about this later
  – Warning: Don’t use a comma instead of a semicolon
    – Means something different (we’ll see in a bit)
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
- 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
  - \(h::t\) 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 \([1; 2; 3]\)*

More examples

```ocaml
# let y = [1;2;3] ;;
val y : int list = [1; 2; 3]
# let x = 4::y ;;
val x : int list = [4; 1; 2; 3]
# let z = 5::y ;;
val z : int list = [5; 1; 2; 3]
```

- not modifying existing lists, just creating new lists
- The left argument of :: is an element
- Can you construct a list \(y\) such that \([1;2]::y\) makes sense?

Lists of lists

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

```ocaml
# let w = [1;2]::y ;;
This expression has type int list but is here used with type int list list
# let w = [1;2]::y ;;
This expression has type int list but is here used with type int list list
```

Practice

- What is the type of
  - \([1;2;3]\)
  - \([ [ ]; [ ]; [1.3;2.4] ]\)
  - \(\text{let } func x = x::(0::[ ]))\)

Pattern Matching

- To pull lists apart, use the match construct
  - \(\text{match } e \text{ with } p_1 \rightarrow e_1 \mid \ldots \mid p_n \rightarrow e_n\)
  - \(p_1\ldots p_n\) are patterns made up of
    - [], ::, and pattern variables
  - \(\text{match}\) 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**
  - `match e with p1 -> e1 | ... | pn -> en`

- **Code 1**
  - `let is_empty l = match l with
    | [] -> true
    | (h::t) -> 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**
  - `let hd l = match l with (h::t) -> 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 Example (cont.)

- **Code 3**
  - `let tl l = match l with (h::t) -> t`

- **Outputs**
  - `tl [1;2;3]` (* evaluates to [2;3] *)
  - `tl [1;2]` (* evaluates to [2] *)
  - `tl [1]` (* evaluates to [ ] *)
  - `tl []` (* 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 `t` ignored
  - Can replace with wildcard `_`
  - Code behavior is identical

Pattern matching - wildcards (cont.)

- **Code using `_`**
  - `let is_empty l = match l with
    | [] -> true
    | (_,_:_) -> false`
  - `let hd l = match l with (h:_) -> h`
  - `let tl l = match l with (_,t) -> 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 []` (* Exception: Match failure *)
  - `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**
  - `# 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**
  - `let addFirsts ((x::_) :: (y::_) :: _) = x + y`
    - `addFirsts [[[1;2];[4;5]];[7;8;9]]` =
  - `let addFirstSecond (x::_)::(_::y::_)::_ = x + y`
    - `addFirstSecond [[[1;2];[4;5]];[7;8;9]]` =
- **Note** – you probably won’t do this much or at all
  - You’ll mostly write recursive functions over lists instead