CMSC 330: Organization of Programming Languages

Functional Programming with OCaml
Functional Programming

- Functions: inputs $\rightarrow$ outputs
- Higher-order functions
  - Functions as arguments
  - Functions as return values
- Recursion
- Inductive definitions
  - Base case(s)
  - Recursive case(s)
Background

• Early AI research
  – Linguists: natural language processing
  – Psychologists: information storage and retrieval
  – Mathematicians: automatic theorem proving

• List processing (LISP)
  – Invented by John McCarthy at MIT in 1958
  – First compiler in 1962

• Two main variations
  – Scheme (1970s)
  – Common LISP (1984)
Sample LISP Programs

(print "Hello world")

(defun factorial (n)
  (if (<= n 1)
      1
      (* n (factorial (- n 1))))

(defun -reverse (list)
  (let ((return-value '()))
    (dolist (e list) (push e return-value))
    return-value))
Background

• Late 1970's – ML (MetaLanguage) developed
  – Robin Milner at the University of Edinburgh
  – Part of a theorem proving system
  – Static typing and type inference
  – Fewer parentheses!

• Descendants
  – Caml (developed at INRIA in 1986)
  – Standard ML (SML) (1990) (now SML/NJ)
  – Haskell (1990)
  – F# (2005)
Dialects of ML

• Many other dialects of ML
  – But SML/NJ and OCaml are most popular
  – O = “Objective,” but probably won’t cover objects

• Functional languages share many core ideas
  – But small and annoying syntactic differences
  – If you buy a book, make sure it covers OCaml
Functional languages

• In a pure functional language, every program is just an expression evaluation
• Canonical “Hello, World” example is a factorial function:

```plaintext
let rec fact x = if x=0 then 1
  else x * fact(x-1)

fact(4) = 4 * fact(3)
  = 4 * 3 * fact(2)
  = 4 * 3 * 2 * fact(1)
  = 4 * 3 * 2 * 1 * fact(0)
  = 4 * 3 * 2 * 1 * 1
  = 24
```
Features of OCaml

• Higher-order functions
  – Functions can be parameters and return values
• “Mostly functional”
• Data types and pattern matching
  – Convenient for certain kinds of data structures
• Type inference
  – Statically typed
  – Hindley-Milner type inference
  – Parametric polymorphism (generics in Java, templates in C++)
• Exceptions
• Garbage collection
Obtaining OCaml

• Official website:  
  http://caml.inria.fr/

• Try it online:  
  http://try.ocamlpro.com/

• It is probably easiest to install through a package manager
  – Use cygwin on Windows  
    • Run installer, then find and install “ocaml” package
  – Use MacPorts on OS X  
    • “sudo port install ocaml”
  – Use apt-get on Ubuntu  
    • “sudo apt-get install ocaml”
Run, OCaml, Run

• 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

• There is also an interpreter ("ocaml") that is similar to "irb" for Ruby
  – Use CTRL-D (Unix) or CTRL-Z (Windows) to exit
A Small OCaml Program- Things to Notice

Use (* *) for comments (may nest)

Use let to bind variables

No type declarations

Need to use correct print function (OCaml also has printf)

Line breaks, spacing ignored

```
(* A small OCaml program *)
let x = 37;;
let y = x + 5;;
print_int y;;
print_string "\n";;
```
Run, OCaml, Run (cont’d)

• Compiling and running the previous small program:

```ocaml
(* A small OCaml program *)
let x = 37;;
let y = x + 5;;
print_int y;;
print_string "\n";;
```

```bash
% ocamlc ocamll.ml
% ./a.out
42
% ```
Run, OCaml, Run (cont’d)

Expressions can also be typed and evaluated in the interpreter:

```ocaml
# 3 + 4;;
- : int = 7

# let x = 37;;
val x : int = 37

# let y = 5;;
val y : int = 5

# let z = 5 + x;;
val z : int = 42

# print_int z;;
42- : unit = ()
```

gives type and value of each expr

“-” = “the expression you just typed”

unit = “no interesting value” (like void)
Expressions can also be typed and evaluated in the interpreter:

```ocaml
# print_string "Hello World";;
Hello World- : unit = ()

# print_int "Hello World";;
This expression has type string but is here used with type int
```
Run, OCaml, Run (cont’d)

- Files can be loaded at the top-level

% ocaml

Objective Caml version 3.08.3

# #use "ocaml1.ml";;
val x : int = 37
val y : int = 42
42- : unit = ()

- : unit = ()
# x;;
- : int = 37

ocaml1.ml:

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

#use loads in a file one line at a time
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 (* don’t care value *)

• OCaml is statically-typed
  – Note: Sometimes the messages are a bit confusing

  # 1 + true;;
  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 Let Construct

- **let** is often used for 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`

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

- bind `pi` in body of `let`
- floating point multiplication
- error
More on the Let Construct (cont’d)

• Compare to similar usage in Java/C

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

```c
{ float pi = 3.14;
  pi * 3.0 * 3.0;
}
pi;
```

• In the top-level, omitting in means “from now on”:

```ml
# let pi = 3.14;;
(* pi is now bound in the rest of the top-level scope *)
```
Nested Let

- Uses of `let` can be nested

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

```ocaml
{
    float pi = 3.14;
    float r = 3.0;

    pi * r * r;
}
/* pi, r not in scope */
```
Defining Functions

- Use `let` to define functions
- List parameters after the function name
- No return statement
- No parentheses on function calls

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

- You can use `let` inside of functions for locals

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

- And you can use as many `lets` as you want

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

• In OCaml, \( \rightarrow \) is the function type constructor
  – The type \( t_1 \rightarrow t_2 \) is a function with argument or domain type \( t_1 \) and return or range type \( t_2 \)

• 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 \((e : t)\) asserts that “\(e\) has type \(t\)”
  – This can be added anywhere you like
    
    ```
    let (x : int) = 3
    let z = (x : int) + 5
    ```

• Use to give function parameter and return types
  
  ```
  let fn (x:int):float =
      (float_of_int x) *. 3.14
  ```
  – Note special position for return type
  – Thus `let g x:int = ...` means `g` returns `int`

• Very useful for debugging, especially for more complicated types
;;; 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
  let print_both (s, t) = print_string s; print_string t;
   "Printed s and t."
  – notice no ; at end---it’s a separator, not a terminator

print_both ("Colorless green ", "ideas sleep")
Prints "Colorless green ideas sleep", and returns
"Printed s and t."
Lists in OCaml

• The basic data structure in OCaml is the list
  – Lists are written as \([e1; e2; \ldots; en]\)
    
    # [1;2;3]
  – Notice \texttt{int list} – lists must be \textit{homogeneous}
  – The empty list is \texttt{[]}  
    
    # []
    
    - : 'a list
  – The \texttt{'a} means “a list containing anything”
    • we’ll see more about this later
  – \textbf{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;
};
...
struct list *l;
...
i = 0;
while (l != NULL) {
    i++;
    l = l->next;
}
```
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 OCaml a list is either
  - The empty list \([]\)
  - Or a pair consisting of an element and a list
    - This recursive structure will come in very handy
Lists are Linked (cont’d)

- :: 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 it’s 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

# let w = [1;2]::y ;;

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

• 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`)
Practice

• What is the type of:
  – `[1;2;3]` \textbf{int list}
  – `[ [ []; []; [1.3;2.4] ] ]`
  – `[ [ []; []; [1.3;2.4] ] ]`
    \textbf{float list list list}
  – let \textbf{func x = x::(0::[])}
    \textbf{int -> int list}
Example

let is_empty l = match l with
  | [] -> true
  | (h::t) -> false

is_empty [] (* evaluates to true *)
is_empty [1] (* evaluates to false *)
is_empty [1;2;3] (* evaluates to false *)
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
- `match` finds the first \(p_k\) that matches the shape of \(e\)
  - Then \(e_k\) is evaluated and returned
  - During evaluation of \(e_k\), pattern variables in \(p_k\) are bound to the corresponding parts of \(e\)
- 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
Example

match e with p₁ -> e₁ | ... | pₙ -> eₙ

let is_empty l = match l with
   [] -> true
| (h::t) -> false

is_empty [] (* evaluates to true *)
is_empty [1] (* evaluates to false *)
is_empty [1;2;3] (* evaluates to false *)
Pattern Matching (cont’d)

• let hd l = match l with (h::t) -> h
  - hd [1;2;3] (* evaluates to 1 *)
• let hd l = match l with (h::_) -> h
  - hd [] (* error! no pattern matches *)
• let tl l = match l with (h::t) -> t
  - tl [1;2;3] (* evaluates to [2; 3] *)
Missing Cases

- Exceptions for inputs that don’t match any pattern
  - OCaml will warn you about non-exhaustive matches

- 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:
[]
```