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)
  – Concept introduced by Newell et al. in 1956
  – John McCarthy and Marvin Minsky at IBM & MIT
  – First specification in 1958

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

• Late 1970's – ML (MetaLanguage) developed
  – Robin Milner at the University of Edinburgh
  – Part of a theorem proving system
  – Statically typed
  – Fewer parentheses!

• Descendants
  – Miranda (1985)
  – Caml (developed at INRIA in 1986)
  – Standard ML (SML) (1990)
  – Haskell (1990)
Dialects of ML

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

• 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

- 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
Functional languages

- In a pure functional language, every program is just an expression evaluation

```ml
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
```
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

• OCaml programs can be compiled using `ocamllc`
  – 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
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";;
```

```
% ocamlc ocamll.ml
% ./a.out
42
```

Run, OCaml, Run (cont’d)

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

```ocaml
# 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
```

gives type and value of each expr

“-” = “the expression you just typed”

unit = “no interesting value” (like void)
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 has static types to help you avoid errors
  
  – 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

```ocaml
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

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

{
  float pi = 3.14;
  pi * 3.0 * 3.0;
}
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 Let

• Uses of let can be nested

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

```plaintext
{
  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 function name.
- No parentheses on function calls.
- No return statement.

```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 \(\rightarrow\) 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
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 functions 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; ...; en]
    # [1;2;3]
    - : int list = [1;2;3]
  – Notice int list – lists must be homogeneous
  – The empty list is []
    # []
    - : 'a list
  – The 'a means “a list containing anything”
    • we’ll see more about this later
  – 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 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’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`)

![Diagram of nested lists]

```plaintext
9 ——> 10 ——> 11 ——> [] ——> 5 ——> 4 ——> 3 ——> 2 ——> []
```
Practice

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