Lecture 10: OCaml

CPSC3520/ECE3520 Programming Systems

Dr. Xiaoyong (Brian) Yuan

Learning Objectives

By the end of this lecture, you should be able to:

- Understand the advantages of using OCaml (Why OCaml?).
- Gain familiarity with OCaml's syntax and core programming constructs.
- Learn how to define and work with OCaml functions.
- Develop skills in handling complex data types and object-oriented programming in OCaml.

Why OCaml?

- **It's Great for Compilers**: It's much easier in OCaml than in C++, Python, Java, etc.
- **It's Succinct**: Would you prefer to write 10,000 lines of code or 5,000?
- **Its Type System Catches Many Bugs**: It catches missing cases, data structure misuse, certain off-by-one errors, etc. Automatic garbage collection and lack of null pointers make it safer than Java.
- **Lots of Libraries and Support**: All sorts of data structures, I/O, OS interfaces, graphics, support for compilers, etc.
- **Industry Impact**:
  - Jane Street trades billions of dollars per day using OCaml programs.
  - Tezos, a blockchain platform, is built in OCaml to support smart contracts.
  - Microsoft has developed F# based on OCaml. See: `https://dotnet.microsoft.com/en-us/languages/fsharp`
  - Intel uses OCaml for verification.

Why OCaml?

A student's comment on OCaml

Never have I spent
so much time
writing so little
that does so much

OCAML Resources

- The OCaml home site, containing distributions and documentation, can be found at: `https://ocaml.org//`
- Installation guide: `https://ocaml.org/docs/installing-ocaml`
- A OCaml user's manual in various formats is available at: `https://ocaml.org/manual/5.2/index.html`
- OCaml Programming: Correct + Efficient + Beautiful: `https://cs3110.github.io/textbook/ocaml_programming.pdf`

OCAML Installation

- MacOS, Linux: Follow official installation guide: `https://ocaml.org/docs/installing-ocaml`
- Windows: This is a bit complicated.
- A straightforward and recommended solution is to install Windows Subsystem for Linux (WSL), Ubuntu `https://learn.microsoft.com/en-us/windows/wsl/install`, then install OCaml following the above Linux installation guide.
- An alternative solution:
  1. Install C compiler, e.g., via Visual Studio
  2. Follow this guide: `https://opam.ocaml.org/blog/opam-2-2-0-windows/`. Key points: 1) Set environment variable (permanently: setx OPAMROOT "
- Last option: OCAML Playground `https://ocaml.org/play` - You can complete your assignment using the online playground.

CAML Pragmatics

(O)CAML is similar to that of LISP and ML, in the sense that:

- Interactive use (a compiler is available) is typical and used for incremental development.
- A top-level loop that performs type checking, argument evaluation (call by value), code compilation, evaluation and printing of the result is used.
- The language is based upon a *core language*, supplemented by a module system.
- Like ML, (O)CAML is dependent upon a *basis library*.

CAML Specifics and Distinctions

1. The command-line version of CAML is invoked by typing `ocaml` at the command prompt.
2. Input is case-sensitive.
3. Comments are the same format as in SML.
4. All variable names must begin with a lowercase letter; names beginning with a capital letter are reserved for constructors for user-defined data structures.
5. Recursive function definitions must include the `rec` designator.
6. Integer and Floating point operations are distinguished by separate symbols.
7. CAML allows multiple-argument functions to be represented in either curried or 'tuple' form. However, the forms cannot be mixed.

CAML Specifics and Distinctions (Cont.)

8. CAML uses the semicolon (;) to delineate list elements.
9. The interactive system prompt is the # character, and a pair of semicolons (;;) is the OCaml expression terminator. ;; is only necessary for interpreted mode use of ocaml. After an expression is entered, the system compiles it, executes it and prints the outcome of the evaluation.
10. `ocaml` phrases are either simple expressions or `let` definitions of identifiers which may be either values or functions.
11. (Like ML), explicit type declaration of function parameters is not necessary. `ocaml` will try to infer the type from usage in the function body.
12. Recursive functions must be defined with the `let rec` binding.

Hello World in OCaml

Example: Hello World

```
print_string "Hello 3520!!\n!"
```

Running with Interpreter

```
$ ocaml hello.ml
Hello 3520!!
```

Compile a native executable and run:

```
$ ocamlopt -o hello hello.ml
$ ./hello
Hello 3520!!
```

Getting Out

There are several ways to exit the ocaml interpreter:

- `#quit;;`
- CTRL-D

Commands beginning with a hash character #, such as `#quit` or `#help`, are not evaluated by OCaml; they are interpreted as commands.

Simple CAML Examples

Example: Basic Arithmetic and Function Definition

```
# 1+2*3;;
- : int = 7

# let pi = 4.0 *. atan 1.0;;
val pi : float = 3.14159265359

# let square x = x *. x;;
val square : float -> float = <fun>

# square(sin pi) +. square(cos pi);;
- : float = 1
```

- The `*.` operator is used for floating-point multiplication in OCaml, distinguishing it from `*`, which is for integer multiplication.
- This distinction helps prevent type errors by ensuring that the appropriate operator is used for each data type.

Using `let` in OCaml

- The `let` keyword binds a value to a name.

Example

```
# let x = 50;;
val x : int = 50
# x * x;;
- : int = 2500
```

- **Local Bindings:** Names can be defined within an expression using `let ... = ... in ....`

Example

```
# let y = 50 in y * y;;
- : int = 2500
# let a = 1 in let b = 2 in a + b;;
- : int = 3
```

- The name `y` is only valid within the expression following the `in` keyword.

CAML Functions

Example: Function Definitions and Applications

```
# let calc input = List.hd(input);;
val calc : 'a list -> 'a = <fun>

# calc([1;2;3]);;
- : int = 1

# let calct (input, input2) = List.hd(input)*List.hd(input2);;
val calct : int list * int list -> int = <fun>
```

Structural vs. Physical Equality

- Physical Equality - compares pointers (addresses), not values.

Physical Equality (==, !=)

```
# 1 == 3;;                     (* false *)
# 1 == 1;;                     (* true *)
# 1.5 == 1.5;;                 (* false, unexpected *)
# let f = 1.5 in f == f;;      (* true *)
# "a" == "a";;                 (* false, unexpected *)
# let a = "hello" in a == a;; (* true *)
```

- Structural Equality - compares values, regardless of pointers.

Structural Equality (=, <>)

```
# 1 = 3;;                     (* false *)
# 1 = 1;;                     (* true *)
# 1.5 = 1.5;;                 (* true *)
# let f = 1.5 in f = f;;      (* true *)
# "a" = "a";;                 (* true *)
```

Note: Use structural equality to avoid unexpected behavior.

Recursion, Pattern Matching, List and trace

Example: Recursive Function with Pattern Matching

```
let rec member = function
(x, []) -> false
| (x, h::t) ->
if (h = x)
then true
else member (x, t) ;;
```

- This function, `member`, recursively checks if an element `x` exists in a list.
- `Pattern Matching` is used to handle two cases:
  - If the list is empty (`[]`), it returns `false`.
  - If the head of the list (`h`) matches `x`, it returns `true`; otherwise, it recurses on the tail `t`.
- The `rec` keyword allows the function to call itself recursively.

Using the 'use' Directive

Example: Using the 'use' Directive and Tracing a Function

```
# #use "member.caml";;
val member : 'a * 'a list -> bool = <fun>
# #trace member;;
member is now traced.
# member("a",["c";"b";"a"]);;
member <-- (<poly>, [<poly>; <poly>; <poly>])
member <-- (<poly>, [<poly>; <poly>])
member <-- (<poly>, [<poly>])
member --> true
member --> true
member --> true
- : bool = true
```

- The `#use` directive loads the code in the specified file into the OCaml session.
- The `#trace` directive enables tracing, which logs each call and return of the function `member`.
- `<poly>` represents polymorphic types, indicating that `member` can accept any type.

Built-In Functions

- Look at the Pervasives Module - without explicit import
- Some are in infix form:

Example: Infix Operators

```
# (+);;
- : int -> int -> int = <fun>

# 7+3;;
- : int = 10
```

- Can always ask for the signature:

Example: Function Signatures

```
# List.hd;;
- : 'a list -> 'a = <fun>

# sqrt;;
- : float -> float = <fun>
```

Defining (As Yet Unnamed) Functions

The ‘match’ form using the `function` keyword:

Syntax: Using ‘match’ with Function Patterns

```
function pattern_1 -> expr_1
| ...
| pattern_n -> expr_n
```

- This expression evaluates to a functional value with **one argument**.
- From the manual (abbreviated/enhanced): When this function is applied to value v, this value is matched against each pattern $pattern_1$ to $pattern_n$. If one of these matchings succeeds (tested in order), then expression expri associated with the matched pattern is evaluated, and this becomes the returned value of the function.

The λ-calculus and CAML

Example: Defining Anonymous Functions

```
# function p -> sqrt p;;
- : float -> float = <fun>

# ((function p -> sqrt p) 2.0);;
- : float = 1.4142135623730951
```

The Identity Function

Example: Using the Identity Function

```
# (function a -> a);;
- : 'a -> 'a = <fun>

# (function a -> a) [1;2];;
- : int list = [1; 2]

# (function a -> a) 3.14159;;
- : float = 3.14159

# (function a -> a) "hiMom";;
- : string = "hiMom"

# (function a -> a) (1,2,3);;
- : int * int * int = (1, 2, 3)
```

Be Careful of Type

Example: Type Error with Incorrect Input

```
# function p -> sqrt p;;
- : float -> float = <fun>

# ((function p -> sqrt p) 2);;
This expression has type int but is here used with type float
```

Tuples

Example: Defining a Tuple

```
(* here's a tuple *)

# (1,2,3);;
- : int * int * int = (1, 2, 3)
```

Tuples vs. Lists in OCaml

- **Size:** Tuples are fixed-size; lists are variable-size.
- **Element Types:** Tuples can have mixed types; lists are homogeneous.
- **Syntax:** Tuples use `( )` with commas, e.g., `(1, 2)`; lists use `[ ]` with semicolons, e.g., `[1; 2]`.

Tuples as Function Arguments

Example: Passing Tuples as Arguments

```
# function p1 p2 p3 -> p1*p2*p3;; (* function must have 1 arg *)
Syntax error

(* solution - use a tuple as the argument *)

# function (p1,p2,p3) -> p1*p2*p3;;
- : int * int * int -> int = <fun>

# ((function (p1,p2,p3) -> p1*p2*p3) (1,2,3));;
- : int = 6

(* you can mix tuples within tuples *)

# function (p1,p2,(p3,p4),p5) -> p1*p2+p3+p4/p5;;
- : int * int * (int * int) * int = <fun>

# (function (p1,p2,(p3,p4),p5) -> p1*p2+p3+p4/p5) (2,3,(1,4),2);;
- : int = 9
```

Definition Approach 2

This is Currying.

Curried Function Definition

```
fun parameter_1 parameter_2 ... parameter_n -> expr
```

- **Currying** is a technique where a function with multiple parameters is transformed into a series of nested functions, each taking a single parameter.
- In OCaml, curried functions allow partial application, meaning you can supply one argument at a time, creating intermediate functions.

Example

Example: Alternative Function Definition (Currying)

```
# fun p1 p2 p3 -> p1*p2*p3;;
- : int -> int -> int -> int = <fun>

# (fun p1 p2 p3 -> p1*p2*p3) 1 2 3;;
- : int = 6

(* compare with previous function definition --
notice signature difference! *)

# function (p1,p2,p3) -> p1*p2*p3;;
- : int * int * int -> int = <fun>
```

Library Function Argument Interface

All the standard library functions are curried.

Example: Using Curried Library Functions

```
# List.nth;;
- : 'a list -> int -> 'a = <fun>

# List.nth [1;2;3;4] 0;;
- : int = 1

# List.nth ([1;2;3;4],0);;
This expression has type int list * int but is here
used with type 'a list
```

More Examples of Defining and Using λ-Functions

Example: Defining and Applying λ-Functions

```
# ((function x -> x*x) 5);;
- : int = 25

# ((function (x,y) -> x*y) (3,4));;
- : int = 12

# ((function x -> List.hd x) ["hi";"mom"]);;
- : string = "hi"

# ((function y -> List.tl y) ["hi";"mom"]);;
- : string list = ["mom"]
```

Naming the Functions (`let`)

- General ocaml syntax:

  ```
  let <name-of-something> = <expr>
  ```

- ```
  # let a = 10;;
  val a : int = 10
  ```

  Bad: imperative programming. Do not use.

- `let <fn-name> = <function-defn>`
- Example:

Naming example

```
# let a = function x -> x*x;;
val a : int -> int = <fun>

# a 7;;
- : int = 49
```
```

More Complex Strategy

The strategy becomes more complex since there are:

- Alternative ways to define functions
- The ‘match’ variant
- The `rec` designator – required for recursive definitions
- Shortcuts

Recursive Function Definitions

For a function to be defined recursively, you need to

1. Give it a name (so it can be referenced in the function body); and
2. Use the `rec` designator, i.e., `let rec ...`

Example

Example: Recursive Function with `let rec`

```
let rec listMinrev2 = function x ->
if x==[] then
    failwith "listMinrev2 should not be used on an empty list"
        else
            if List.tl(x)==[] then List.hd(x)
            else min (List.hd x) (listMinrev2 (List.tl x));;
```

- This function, `listMinrev2`, recursively finds the minimum element in a non-empty list.
- `let rec` defines a recursive function.
- The function uses pattern matching to handle cases:
  - If the list is empty (`x==[]`), it raises an error using `failwith`.
  - If the list has only one element (`List.tl(x) == []`), it returns that element as the minimum.
  - For longer lists, it compares the head of the list (`List.hd x`) with the minimum of the tail (`listMinrev2 (List.tl x)`), returning the smaller value.

Imperative vs. Functional Example

Example: Imperative vs. Functional Programming Style

```
(* imperative *)
# let w = 1;;
val w : int = 1
# let w = w + 1;;
val w : int = 2

(* functional programming example *)
# let wfun = function w -> w + 1;;
val wfun : int -> int = <fun>
# wfun w;;
- : int = 3
# w;;
- : int = 2
```

- **Functional Programming** avoids modifying variables or states directly; instead, it creates new values or functions. Here, `wfun w` calculates `w + 1` without changing the original `w`.
- Functional programming promotes immutability, making code easier to reason about and reducing side effects.

Examples: Defining Named Functions

Example: Recursive Function with Type Inference Issue

```
let rec recursiveFn2 = function (n) ->
if n==0 then []
else
sqrt n :: recursiveFn2 (n-1) ;;
```

- `recursiveFn2`, is intended to create a list of square roots for each number from `n` down to 1.
- The above code will cause a type inference error because `sqrt` expects a `float`, but `n` is an `int`.

Solution to the Type Inference Issue

Example: Corrected Recursive Function

```
let rec recursiveFn3 = function (n) ->
if n==0 then []
else
sqrt (float_of_int n) :: recursiveFn3 (n-1) ;;
```

- `float_of_int` is used to convert `n` to a `float` before applying `sqrt`.

Tracing Revisit

Tracing and Untracing Commands

```
# #trace function-name;;
(* After executing this directive, all calls to the
   function named function-name will be traced *)

# #untrace function-name;;
(* Stop tracing the given function. *)

# #untrace_all;;
(* Stop tracing all functions traced so far. *)
```

Tracing the Recursion (#trace)

Example: Tracing a Recursive Function

```
# #trace recursiveFn3;;
recursiveFn3 is now traced.

# recursiveFn3 6;;
recursiveFn3 <-- 6
recursiveFn3 <-- 5
recursiveFn3 <-- 4
recursiveFn3 <-- 3
recursiveFn3 <-- 2
recursiveFn3 <-- 1
recursiveFn3 <-- 0
recursiveFn3 --> []
recursiveFn3 --> [1.]
recursiveFn3 --> [1.41421356237309515; 1.]
recursiveFn3 --> [1.73205080756887719; 1.41421356237309515; 1.]
- : float list = [2.44948974278317788; 2.23606797749979; 2.;
1.73205080756887719; 1.41421356237309515; 1.]
```

Simplified Function Definition

Recursive Function Definition

```
# let rec recursiveFn3 = function (n) ->
if n==0 then []
    else
    sqrt (float_of_int n) :: recursiveFn3 (n-1) ;;

val recursiveFn3 : int -> float list = <fun>
# recursiveFn3 4;;
- : float list = [2.; 1.73205080756887719; 1.41421356237309515;
1.]
```

Don't need the parens on the argument:

Simplified Recursive Function Definition

```
# let rec recursiveFn3 = function n ->
if n==0 then []
    else
    sqrt (float_of_int n) :: recursiveFn3 (n-1) ;;
val recursiveFn3 : int -> float list = <fun>
```

Simplified Function Definition

Further Simplified Recursive Function Definition

```
# let rec recursiveFn3 n =
if n==0 then []
    else
    sqrt (float_of_int n) :: recursiveFn3 (n-1) ;;
val recursiveFn3 : int -> float list = <fun>

# recursiveFn3 4;;
- : float list = [2.; 1.73205080756887719; 1.41421356237309515;
1.]
```

More of the Simplified Syntax

Examples: Simplifying Function Syntax

```
(* original *)
let rec boardform1D = function (n) ->
if n==0 then []
    else
    "empty" :: boardform1D (n-1) ;;

(* simpler *)
let rec boardform1Dr1 n =
if n==0 then []
    else
    "empty" :: boardform1Dr1 (n-1) ;;

(* original *)
let rec boardform2D = function (n,m) ->
if n==0 then []
    else
    boardform1D(m) :: boardform2D (n-1,m) ;;

(* simpler *)
let rec boardform2Dr1 (n,m) =
if n==0 then []
    else
    boardform1D(m) :: boardform2Dr1 (n-1,m) ;;
```

- The simplified versions remove the use of `function` and directly take parameters `n` and `(n, m)`, making the code more concise.

A Long List Recursion Example

Example: Recursive Function to Build a List

```
# let rec recursiveFn = function (n) ->
if n==0 then []
    else
    "Clemson" :: recursiveFn (n-1) ;;

val recursiveFn : int -> string list = <fun>
```

- This function, `recursiveFn`, generates a list containing the string `"Clemson"` repeated `n` times.
- It uses pattern matching to check if `n` is zero. If so, it returns an empty list; otherwise, it prepends `"Clemson"` and recurses with `n-1`.

Using the Recursive Function

Example: Calling the Recursive Function

```
# recursiveFn 10;;
- : string list = ["Clemson"; "Clemson"; "Clemson"; "Clemson"; "Clemson";
                   "Clemson"; "Clemson"; "Clemson"; "Clemson"; "Clemson"]

# recursiveFn (10);;
- : string list = ["Clemson"; "Clemson"; "Clemson"; "Clemson"; "Clemson";
                   "Clemson"; "Clemson"; "Clemson"; "Clemson"; "Clemson"]

# (recursiveFn 10);;
- : string list = ["Clemson"; "Clemson"; "Clemson"; "Clemson"; "Clemson";
                   "Clemson"; "Clemson"; "Clemson"; "Clemson"; "Clemson"]

# (recursiveFn (10));;
- : string list = ["Clemson"; "Clemson"; "Clemson"; "Clemson"; "Clemson";
                   "Clemson"; "Clemson"; "Clemson"; "Clemson"; "Clemson"]
```

Tracing the Recursive Function

Example: Tracing a Recursive Function

```
# #trace recursiveFn;;
recursiveFn is now traced.

# recursiveFn 8;;
recursiveFn <-- 8
recursiveFn <-- 7
recursiveFn <-- 6
recursiveFn <-- 5
recursiveFn <-- 4
recursiveFn <-- 3
recursiveFn <-- 2
recursiveFn <-- 1
recursiveFn <-- 0
recursiveFn --> []
recursiveFn --> ["Clemson"]
recursiveFn --> ["Clemson"; "Clemson"]
recursiveFn --> ["Clemson"; "Clemson"; "Clemson"]
recursiveFn --> ["Clemson"; "Clemson"; "Clemson"; "Clemson"]
recursiveFn --> ["Clemson"; "Clemson"; "Clemson"; "Clemson"; "Clemson"]
recursiveFn --> ["Clemson"; "Clemson"; "Clemson"; "Clemson"; "Clemson"; "Clemson"]
recursiveFn --> ["Clemson"; "Clemson"; "Clemson"; "Clemson"; "Clemson"; "Clemson"; "Clemson"]
recursiveFn --> ["Clemson"; "Clemson"; "Clemson"; "Clemson"; "Clemson"; "Clemson"; "Clemson"; "Clemson"]
```

Enhanced “building a list” Example

Example: Building a List with Specified Size and Element

```
let rec listBuild = function (size, element) ->
(* check for proper use *)
if size <= 0 then
    failwith "listBuild should only be used by trained personnel"
else
    if size == 1 then [element]
    else element :: listBuild (size-1, element);;
```

- This function, `listBuild`, creates a list of the specified size, where each element is the specified value.
- It checks for valid input and raises an error if the size is non-positive.

Avoid Mixing Curried and Tuple Function Designs

Example: Curried vs. Tuple Function Definitions

```
# let calc input1 input2 = List.hd input1 * List.hd input2;;
val calc : int list -> int list -> int = <fun>

# calc [1;2;3] [3;2;1];;
- : int = 3

# let calc2 (input1, input2) = List.hd input1 * List.hd input2;;
val calc2 : int list * int list -> int = <fun>

# calc2 ([1;2;3], [3;2;1]);;
- : int = 3
```

- The first example uses curried arguments, while the second example uses a tuple.
- Mixing curried and tuple-based arguments can lead to errors, as shown below.

Errors from Mixing Curried and Tuple Designs

Example: Errors from Incorrect Argument Types

```
# calc ([1;2;3], [3;2;1]);;
This expression has type int list * int list but is here used
with type int list

# calc2 [1;2;3] [3;2;1];;
This function is applied to too many arguments, maybe you
forgot a `;'
```

- Attempting to use a tuple with a curried function or vice versa results in type errors.
- Ensure consistency in function argument structure to avoid these issues.

Pattern Matching with `match`

Syntax for `match`

```
match expr with
  | pattern_1 -> expr_1
  | ...
  | pattern_n -> expr_n
```

- The `match` expression allows pattern matching to evaluate expressions based on specific patterns.
- Each pattern `pattern_i` is matched in sequence, and the corresponding `expr_i` is executed when a match is found.

Example of Pattern Matching

Example: Zero Checker

```
# let isZero n = match n with
  | 0 -> true
  | _ -> false;;

val isZero : int -> bool = <fun>

# isZero 3;;
- : bool = false

# isZero 0;;
- : bool = true
```

- This function checks if a number is zero.
- The _ pattern acts as a catch-all, matching any non-zero values.

Adding Two Lists Element-wise

Example: Element-wise List Addition

```
# let rec add2lists = fun x y ->
  if (x = [] && y = []) then []
  else ((List.hd x) + (List.hd y)) ::
  add2lists (List.tl x) (List.tl y);;
val add2lists : int list -> int list -> int list = <fun>

# add2lists [1;2;3;4] [4;3;2;1];;
- : int list = [5; 5; 5; 5]
```

Order of Function Definitions

Example: Function Dependency Error

```
(* file1.caml *)
let f1 = function (n) -> f2(n);;
let f2 = function (m) -> m * m * m;;
```

- Attempting to use `f2` in `f1` before `f2` is defined results in an error.
- OCaml requires functions to be defined before they are used in another function.

Solution: Correct Order of Function Definitions

Example: Resolving Dependency by Reordering

```
(* file2.caml *)
let f2 = function (m) -> m * m * m;;
let f1 = function (n) -> f2(n);;



# #use"file2.caml";;
val f2 : int -> int = <fun>
val f1 : int -> int = <fun>

# f1 6;;
- : int = 216
```

- By defining `f2` first, we avoid the dependency issue, allowing `f1` to reference `f2` without error.

Mutually Recursive Functions

- There is a potential problem with functions that recur through each other.
- The problem is the definition of one without a forward reference to the other.

odd-even.caml

```
(********************************************************
NOTE: THIS IS A PAIR OF MUTUALLY RECURSIVE FUNCTIONS AND
REQUIRES SPECIAL HANDLING IN ocaml.
********************************************************)

let even n =
if (n==0) then
true
    else odd (n-1);;

let odd m =
if (m==0) then
false
    else even (n-1);;
```

Mutually Recursive Functions (Cont.)

(Attempted) use of odd-even.caml

```
# #use "odd-even.caml";;
File "odd-even.caml", line 8, characters 15-18:
Unbound value odd #
```

Why?

Mutually Recursive Functions (Cont.)

odd-even-mut-rec.caml

```
let rec even n =
  if (n == 0) then true
  else odd (n - 1)
and (* here's the mutual recursion *)
odd m =
  if (m == 0) then false
  else even (m - 1);;
```

- The `and` keyword allows defining mutually recursive functions in OCaml.
- Here, `even` and `odd` call each other to determine whether a number is even or odd.

Mutually Recursive Functions (Cont.)

Example: Testing the Recursive Functions

```
# #use "odd-even-mut-rec.caml";;
val even : int -> bool = <fun>
val odd : int -> bool = <fun>

# even 6;;
- : bool = true

# odd 6;;
- : bool = false

# even 7;;
- : bool = false

# odd 7;;
- : bool = true
```

- Testing the `even` and `odd` functions shows that they correctly identify even and odd numbers by calling each other recursively.

Tracing Mutually Recursive Functions

Example: Tracing `even` and `odd` Functions

```
# #trace even;;
even is now traced.
# #trace odd;;
odd is now traced.

# even 6;;
even <-- 6
odd <-- 5
even <-- 4
odd <-- 3
even <-- 2
odd <-- 1
even <-- 0
even --> true
odd --> true
even --> true
odd --> true
even --> true
odd --> true
even --> true
- : bool = true
```

OCAML Advanced Examples - Tail of a List

Writing a function `last : 'a list -> 'a option` to return the last element of a list

Statement

```
# last ["a" ; "b" ; "c" ; "d"];;
- : string option = Some "d"
# last [];;
- : 'a option = None
```

Solution

```
# let rec last = function
  | [] -> None
  | [ x ] -> Some x
  | _ :: t -> last t;;
val last : 'a list -> 'a option = <fun>
```

OCAML Advanced Examples - Duplicate the Elements of a List

Duplicate the Elements of a List

Statement

```
# duplicate ["a"; "b"; "c"; "c"; "d"];;
- : string list = ["a"; "a"; "b"; "b"; "c"; "c"; "c"; "c";
"d"; "d"]
```

Solution

```
# let rec duplicate = function
    | [] -> []
    | h :: t -> h :: h :: duplicate t;;
val duplicate : 'a list -> 'a list = <fun>
```

OCAML Advanced Examples - Length of a List

Writing a function `length: 'a list -> int` to find the number of elements in a list.

Statement

```
# length ["a"; "b"; "c"];;
- : int = 3
# length [];;
- : int = 0
```

Solution 1

```
# let rec length = function
  | [] -> 0
  | _ :: tail -> 1 + length tail;;
val length : 'a list -> int = <fun>
```

Solution 2

```
# let length list =
    let rec aux n = function
      | [] -> n
      | _ :: t -> aux (n + 1) t
    in aux 0 list;;
val length : 'a list -> int = <fun>
```

Which one is better?

Tail Recursion

- Let's start with an uninteresting function, which counts from 1 to n:

count function

```
# let rec count n =
if n = 0 then 0 else 1 + count (n - 1)
val count : int -> int = <fun>
```

- Any potential issue?
- Counting to 10 is no problem;
- Counting to 100,000 is no problem;
- But try counting to 1,000,000 and you'll get the following error:

  `Stack overflow during evaluation (looping recursion?).`

- Reason: call stack has a limited size - This stack contains one element for each function call that has been started but has not yet been completed.

Tail Recursion (Cont.)

- Let's start with an uninteresting function, which counts from 1 to n:

count function

```
# let rec count n =
    if n = 0 then 0 else 1 + count (n - 1)
```

- Solution? Tail Recursion!

count function with tail recursion

```
# let rec count_aux n acc =
  if n = 0 then acc else count_aux (n - 1) (acc + 1)
# let count_tr n = count_aux n 0
```

- A good compiler (like OCaml) can notice when a recursive call is in tail position, meaning that "there's no more computation to be done after it returns".
- A recursive call in tail position does not need a new stack frame. The compiler can recycle the space.
- Let's revisit the `length` function. Think about how to implement a `Fibonacci` function.

Tail Recursion (Cont.)

- Let's start with an uninteresting function, which counts from 1 to n:

count function

```
# let rec count n =
    if n = 0 then 0 else 1 + count (n - 1)
```

- Solution? Tail Recursion!

count function with tail recursion

```
# let rec count_aux n acc =
  if n = 0 then acc else count_aux (n - 1) (acc + 1)
# let count_tr n = count_aux n 0
```

- Recipe for Tail Recursion
  - introduce a helper function: add extra arguments, e.g., accumulator
  - write a main function to call the helper function
  - the helper function returns the accumulator to the main function
- Introducing an accumulator is still considered pure functional programming. Why?

OCAML Advanced Examples - Eliminate Consecutive Duplicates

Statement

```
# compress ["a"; "a"; "a"; "a"; "b"; "c"; "c"; "a"; "a"; "d"; "e"; "e"; "e"; "e"];;
- : string list = ["a"; "b"; "c"; "a"; "d"; "e"]
```

Solution

```
# let rec compress = function
    | a :: (b :: _ as t) ->
    if a = b then compress t else a :: compress t
    | smaller -> smaller;;
val compress : 'a list -> 'a list = <fun>
```

`smaller -> smaller` handles cases where the list has fewer than two elements.

OCAML Advanced Examples - Determine Prime Factors

Determine the Prime Factors of a Given Positive Integer.

Statement

```
# factors 315;;
- : int list = [3; 3; 5; 7]
```

Solution

```
# let factors n =
    let rec aux d n =
      if n = 1 then [] else
        if n mod d = 0 then d :: aux d (n / d) else aux (d + 1) n
    in
      aux 2 n;;
val factors : int -> int list = <fun>
```

Caml in One Slide

Apply a function to each list element; save results in a list

"Is recursive"

Passing a function

Case splitting

Pattern Matching

Local name declaration

Polymorphic

Types inferred

List support

Recursion

Anonymous functions

```
# let rec map f = function
    [] -> []
  | head :: tail ->
      let r = f head in
      r ::  map f tail;;
val map :  ('a -> 'b) -> 'a list -> 'b list

# map (function x -> x + 3) [1;5;9];;

- :  int list = [4; 8; 12]
```

Compiling OCaml Programs - Bytecode Compilation

- Create a new file named `hello.ml`

hello.ml

```
let _ = print_endline "Hello, OCaml!"
```

- **Bytecode Compilation**: Using `ocamlc`

Bytecode Example

```
# ocamlc -o hello hello.ml
# ./hello
Hello, OCaml!
```

`ocamlc` produces portable bytecode executables.

Compiling OCaml Programs - Dune for large projects

Directory Setup

```
my_project/
|-- dune   (Describes the build configuration.)
|-- hello.ml   (Contains the OCaml source code.)
|-- dune-project   (Declares the project and its version.)
```

Dune File Content (dune)

```
(executable
 (name hello))
```

Dune Project File (dune-project)

```
(lang dune 3.0)
```

Compiling and Running

```
# dune build hello.exe
# ./_build/default/hello.exe
Hello, OCaml!
```

Executing a Script

- The `ocaml` command starts the toplevel system for Objective Caml.
- This command initiates the interactive read-eval-print loop but can also be used to execute a script file non-interactively.
- Usage:

Script Execution

```
ocaml [script-file]
```

CAML I/O

- Caml provides numerous functions for input and output operations.
- Input and output are specified by input and output channels: `in_channel` and `out_channel`, respectively. The defaults are `stdin` and `stdout`.
- Output functions are included in the `Pervasives` module:
  - `print_string` with signature `string -> unit`, which prints a string on the standard output and returns `unit`.
  - `print_int`, `print_float`, and `print_newline` are also provided for integer, float, and newline output.

I/O Channel Signatures

Examples of I/O Channels

```
# stdin;;
- : in_channel = <abstr>

# stdout;;
- : out_channel = <abstr>

# stderr;;
- : out_channel = <abstr>

# open_out;;
- : string -> out_channel = <fun>

# open_in;;
- : string -> in_channel = <fun>

# close_out;;
- : out_channel -> unit = <fun>

# close_in;;
- : in_channel -> unit = <fun>

# Printf.fprintf;;
- : out_channel -> ('a, out_channel, unit) format -> 'a = <fun>

# Scanf.fscanf;;
- : in_channel -> ('a, Scanf.Scanning.scanbuf, 'b) format -> 'a -> 'b = <fun>
```

Using `Printf.printf`

- The `Printf` module includes powerful functions for formatted printing, particularly useful for C programmers familiar with `printf`.
- A formatted string (format) is used to specify the structure of the output, similar to `printf` in C.

Basic Example

```
# Printf.printf "Hi Mom";;
Hi Mom- : unit = ()
```

More Versatile Example with Format String

Formatted Output Example

```
# open Printf;;
# printf "\n\n\n%s\n\n%d" "And the answer is:" 10;;



And the answer is:


10- : unit = ()
#
```

- The `open Printf` command brings all the functions from the `Printf` module into the current scope.
- This allows you to call `printf` directly without prefixing it with `Printf.`, simplifying the code.
- The example demonstrates using a format string with placeholders (`%s` for a string and `%d` for an integer) to print formatted text.

File Writing Example with `Printf`

Example: Writing to a File

```
# let my_chan = open_out "camlTest.out";;
val my_chan : out_channel = <abstr>

# Printf.fprintf my_chan "Hi Bob\n";;
- : unit = ()

(* still nothing in file-- until-- *)

# flush my_chan;;
- : unit = ()

(* now contents are written to the file
further extension---- *)

# Printf.fprintf my_chan "\n %d %d %s\n" 10 20 "done";;
- : unit = ()

# flush my_chan;;
- : unit = ()

(* here are file contents so far: *)

Hi Bob

10 20 done

#
```

Additional Examples with `Printf.printf`

Integer and Float Formatting

```
# Printf.printf "\n\n %i \n\n" 5;;


5


- : unit = ()

# Printf.printf "\n\n %f \n\n" 5.9;;


5.900000


- : unit = ()
```

Complex Format Example

Example: Mixed Formatting

```
# open Printf;;
# printf "%f %f %f" 1.2 3.4 5.6;;
1.200000 3.400000 5.600000- : unit = ()

# printf "%s %d %f" "hi" 123 3.45;;
hi 123 3.450000- : unit = ()
```

Objects in CAML

- OCaml supports object-oriented programming, enabling the creation of classes and objects.
- Consider the declaration of two CAML objects shown below.
- `object ... end`: The object construct is used to define the body of the class

Example: Defining Objects in OCaml (`vehicle.caml`)

```
class vehicle =
  object
    val mutable name = "batmobile"
    method print_name = name
  end;;

class boat =
  object
    val mutable name = "leaky"
    val mutable capacity = 4
    method how_big = capacity
  end;;
```

Results of Object Creation

Example: Using Defined Objects

```
# #use "vehicle.caml";;

class vehicle :
  object val mutable name : string method print_name : string end

class boat :
  object
    val mutable capacity : int
    val mutable name : string
    method how_big : int
  end

# let my_vehicle = new vehicle;;
val my_vehicle : vehicle = <obj>

# my_vehicle#print_name;;
- : string = "batmobile"

# let my_boat = new boat;;
val my_boat : boat = <obj>

# my_boat#how_big;;
- : int = 4
```

- Objects can be instantiated using the `new` keyword.
- Each object can call its respective methods (e.g., `print_name` for `vehicle` and

Inheritance in OCaml

- Classes in OCaml can inherit properties and methods from other classes, enabling a hierarchy.
- Consider the modified class definitions below to allow inheritance.

Example: Class Inheritance (vehicle2.caml)

```
class vehicle =
  object
    val mutable name = "batmobile"
    method print_name = name
  end;;

class boat =
  object
    inherit vehicle
    val mutable capacity = 4
    method how_big = capacity
  end;;
```

Behavior of the Inherited Class Structure

Example: Using Inherited Methods

```
# #use "vehicle2.caml";;

class vehicle :
  object val mutable name : string method print_name : string end

class boat :
  object
    val mutable capacity : int
    val mutable name : string
    method how_big : int
    method print_name : string
  end

# let my_vehicle = new vehicle;;
val my_vehicle : vehicle = <obj>

# let my_boat = new boat;;
val my_boat : boat = <obj>

# my_vehicle#print_name;;
- : string = "batmobile"

# my_boat#print_name;;
- : string = "batmobile"
```

- The `boat` class inherits the `print_name` method from `vehicle`, allowing both `vehicle` and `boat` instances to use it.

Conclusion

- OCaml is a functional programming language that emphasizes immutability, reducing the complexity of state management.
- OCaml provides a strong type inference system, which makes code easier to read and maintain.
- OCaml provides advanced pattern-matching capabilities to support complex data handling.
- OCaml supports object-oriented programming.
- OCaml provides libraries and debugging tools, such as tracing and formatted printing for efficient development and testing.

References

- Dr. Madhusudan Parthasarathy, CS 421
- Dr. Stephen A. Edwards, COMS W4115