

S1.2: OCaml Basics

CSci 2041:

Advanced Programming Principles

University of Minnesota,
Prof. Van Wyk,
Spring 2022

1

Primary characteristics of the OCaml programming language

- ▶ evaluation is by expression evaluation - no assignment statements (in the pure functional core of the language)
- ▶ strong, static type system - no run-time type errors ever! (types are inferred, if not written explicitly)
- ▶ support for structured data - lists, tuples, records, inductive types (a.k.a. algebraic data types)
- ▶ pattern matching of data
- ▶ higher-order function, curried functions
- ▶ automatic memory management, *i.e.* garbage collection
- ▶ sophisticated module system

2

OCaml: expression evaluation

- ▶ Expression evaluation is the primary means of computation in OCaml, and other functional languages.
- ▶ Roughly, expressions are what appear on the right hand side of assignment statements in imperative languages
- ▶ If we give up assignments, while-loops, for-loops, etc. what can we do?
- ▶ What do functional programming languages provide that makes such a thing at all reasonable?

Recursion, higher-order functions, for starters.

3

Working with the OCaml interpreter in lecture

- ▶ We will use `utop` as a “toplevel” REPL for OCaml.
(REPL = read-evaluate-print-loop)
- ▶ The history files contain all commands and expressions typed in.
- ▶ Note that OCaml also reports the type of values it computes.
- ▶ Go back and review the recordings if you have questions about the functions we write in class.
- ▶ OCaml also has 2 compilers, one that generates byte-code for a virtual machine, another that generates native machine code.

4

Simple expression evaluation

- ▶ OCaml's treatment of integers, Booleans, strings, etc. is not very different from other languages.
- ▶ Let's consider some examples in `utop`.
- ▶ One unexpected item: integer operators

`+ - * /`

are different from floating point operators

`+. -. *. /.`

5

OCaml types

- ▶ In our demos we saw the following basic types:
`int`, `float`, `bool`, `string`, `char`.
- ▶ These have operators and functions that are not unexpected.
- ▶ Type errors are reported for expressions such as `1 + "Hello"`.
- ▶ Exceptions arise from some errors in values (of the correct type), such as division by 0.

6

OCaml modules and utop

- ▶ OCaml modules organize code into useful components.
- ▶ The `String` and `Char` modules contains functions you might expect.
- ▶ `utop` shows what names are valid as you type in expressions and this can be useful in looking for helpful functions in the libraries.
- ▶ Try typing `Char.u` and you'll see some possible function names at the bottom of your screen.
- ▶ Continue typing `pp`, hit 'tab' then `;;` and press return. `utop` shows you the type of that function. This should give you some idea what it does.
- ▶ Use this for the `String`, `List`, and other modules.

7

Name bindings: let-declarations and let-expressions

- ▶ In all languages, we give names to values. We'll call this "binding" a value to a name.
- ▶ In OCaml, we can `bind` values to names in two ways.
 - ▶ `let var = expr`
e.g. `let x = 7 ;;`
This is a `let-declaration`, it only appears at the top level of a file or in `utop`.
 - ▶ `let var = expr in expr`
e.g. `let y = 8 in y * y ;;`
This is a `let-expression`, it can appear anywhere an expression can appear in OCaml.
- ▶ Let's look at some examples.

8

Name bindings: let-declarations and let-expressions

- ▶ OCaml `infers` the types for these identifiers, but we can provide them explicitly.
- ▶ The term "variables" is a bit of a misnomer in that we cannot change (or vary) their associated value.
- ▶ We can nest let-expressions.
- ▶ A variable declaration's *scope*: program text over which the variable can be used. This is a static property.
- ▶ A variable declaration's *extent*: program execution time in which reference may occur (and the value must be accessible in memory).
- ▶ These will get more interesting with `closures`.

9

OCaml programs

- ▶ OCaml programs are sequences of [declarations](#). (see notes)
- ▶ A declaration is a `let` without the `in` part.
- ▶ We've also seen type declarations in the whirlwind tour.
- ▶ e.g. `let x = 7`
- ▶ e.g. `let hw = "Hello " ^ "World"`
- ▶ e.g. `type tree = Empty | ...`
- ▶ In the interpreter, we can enter
 - ▶ expressions, declarations, or commands/directives (e.g. `#use ...`)We terminate each with `;;`
- ▶ In files, we do not need the `;;`.
The next `let` or end-of-file indicates the end of each declaration.
- ▶ Code can be found in [getting_started.ml](#) in the public repo in [Course-Resources/Sample-Programs](#).

10

Using files in utop

In `utop`, commands to `utop` are preceded by `#`, for example

- ▶ `#use "getting_started.ml";;`
- ▶ `#quit ;;`

11

Functions, over simple data

- ▶ Defined by a `let` declaration.
- ▶ e.g. `let square (x: int) : int = x * x`
- ▶ See others in [getting_started.ml](#).

12

Functions are just values

- ▶ `let inc = fun x -> x + 1`
- ▶ We can write function literals, that is, lambda-expressions. Historically, these are written as $\lambda x \rightarrow x + 1$
- ▶ OCaml provides more intuitive declarations:
`let name parameters = expr`
e.g. `let inc x = x + 1`

13

Exercise S1.2: #1: Circle area

Write an OCaml function named `circle_area` with type `float -> float` that computes (surprise) the area of a circle given its radius.

How to turn these in:

1. write them on paper, turn them in at the end of class
2. if you're doing this remotely
 - ▶ go to your individual repository
 - ▶ `% mkdir In-class`
 - ▶ create a file named `01_24.ml`
 - ▶ add all OCaml solutions for today here
 - ▶ commit and push after lecture

14

Function types

- ▶ Consider the type of our increment function

```
# let inc x = x + 1 ;;  
val inc : int -> int = <fun>
```

- ▶ Now, what about add?

```
# let add x y = x + y ;;  
val add : int -> int -> int = <fun>
```

- ▶ We might expect (incorrectly) something like
`int, int -> int`

15

Function types

- In fact, `let add x y = x + y` is just short-hand for

```
# let add = fun x -> (fun y -> x + y) ;;  
val add : int -> int -> int = <fun>
```

- The function type operator `->` is right associative.

- `int -> (int -> int)`
and
`int -> int -> int`
are equivalent.

16

Curried functions

- Curried functions “take their arguments one at a time.”

- `# let add x y = x + y ;;`
`val add : int -> (int -> int) = <fun>`

- We can define increment using add.

```
# let inc = add 1 ;;  
val inc : int -> int = <fun>
```

17

Curried functions

- This means we may need to use parenthesis to group each individual argument.

```
# add (1 + 3) 7 ;;  
- : int = 11
```

- The “function application operator” is implicit.
We just write the arguments after the function.

- This “operator” is left associative.
Can we add parenthesis to make this explicit?

- Let’s consider some examples and possible errors ...

18

Recursive functions

- ▶ “let-rec” expressions
- ▶ e.g.

```
let rec fact n =  
    if n = 0 then 1 else n * fact (n-1)
```
- ▶ The scope of the variable in a let-rec includes the defining expression.
- ▶ This is not the case if a non-recursive let.

19

Exercise S1.2: #2:

Write an OCaml function named `power` with the type `int -> float -> float`.

`power 2 3.0` should return `9.`

`power 3 3.0` should return `27.`

`power 3 3.2` should return `32.768`.

Remember, put this in a function named `mm_dd.ml` or turn in paper after class.
(Always use 2 digits for the month and the day.)
(If you're present in lecture, turn in paper.)

20

Exercise S1.2: #3:

We previously wrote `cube` as follows:

```
let cube x = x *. x *. x
```

Write another version that uses `power`, preferably taking advantage of the curried nature of functions in OCaml.

What is the minimal number of characters needed to do this?

21

Different varieties of phrases

There are *different kinds* of phrases in OCaml and other languages.

1. “expressions” - that evaluate to a value
2. “statements” - in C, Java, etc. These perform some action, perhaps changing a value in memory.
3. “types” - a sub-language for “type expressions”
4. “declarations” - declaring new names/variables
5. “patterns” - (coming soon)...

22

The language of types

- ▶ Types are an important “sub-language” in OCaml and other languages.
- ▶ There are constants: `int`, `float`
- ▶ Variables: `'a`, `'b`
- ▶ Operators: `->`
- ▶ Constructors: We wrote `tree` and `list` on Friday, we'll, see others later.
- ▶ These form a proper “language of types.”

23

Types as an organizing principle

- ▶ Types (or type expressions) provide the first approximation of understanding what a function does.
- ▶ Without a proper language of types, it is difficult to even think in these terms.
- ▶ This is missing in dynamically typed languages like Scheme, Clojure, Python, etc.
- ▶ In this regard, *static* checking is not the point. Being able to think properly about types - if they are checked at compile time, at runtime, or never(!) - is what matters now.
- ▶ Thinking in terms of types shapes our thinking and helps us design programs.

24

Greatest common divisor

- ▶ How can we compute the greatest common divisor of two positive integers?
- ▶ A strategy we will follow here:
 1. Pick a number that must be greater than or equal to the GCD.
We can use the minimum of the two inputs for this.
 2. Decrement it by one until it is a common divisor.
We know we'll find it, even if we go all the way down to 1.
- ▶ How can we design such a function?

25

Designing a `gcd` function

1. What is the type of `gcd`?
2. Any useful helper functions it should call?
3. “decrement” sounds like an assignment statement, but we don't have those.
4. Instead, use an argument to a recursive function that changes for each recursive call.

26

Technique: auxiliary parameters

- ▶ In the example, above we sort of wanted a variable to be updated in a loop by an assignment statement.
- ▶ The technique of [auxiliary parameters](#) can be used in these situations.
- ▶ A recursive function will have these “extra” parameters that are changed for each recursive call so that in the “next” call the values are “updated.”
- ▶ In the greatest common divisor function `gcd`, `d` is this kind of parameter in [decrement](#).
- ▶ Remember this! You'll want to use this later.

27

Exercise S1.2: #4: Least common multiple

Below is our implementation of `gcd`:

```
let gcd (x: int) (y: int) : int =
  let min = if x < y then x else y
  in
  let rec decrement (d: int) : int =
    if x mod d = 0 && y mod d = 0
    then d      (* found it! *)
    else decrement (d-1)
  in
  decrement min
```

Now write a function called `lcm` that finds the list common multiple of two positive integers.

It should have the type `int -> int -> int`.

For example, `lcm 4 6` is `12`.

28

Technique: accumulating parameters

- ▶ A specific kind of auxiliary parameter is an `accumulating parameter`.
- ▶ The term “accumulating” suggests the value of these collect, or accumulates, information that will eventually be the result we seek.
- ▶ Let’s apply this technique in a new implementation of `sum_to`.
- ▶ Here we’ll count up from 0, accumulating the sum of the numbers in an accumulating parameter.
- ▶ We’ll call this `sum_to_accum_up`, again in `getting_started.ml`.

29

Types as contracts

- ▶ We can think of the types (sometimes called “type signatures”) of functions as defining a rudimentary contract. This contract can be read as the function saying
 - ▶ if you give me input value of the correct type
 - ▶ then I’ll give you an output value of the correct type.
- ▶ e.g. `square` will return an `int` if it is given an `int`.
A similar statement can be made for `circle_area`.
- ▶ But `sum_to: int -> int` fails to hold up this contract for inputs that are `ints` but that are negative.
- ▶ So what do we do when types are not accurate enough to specify the desired contract?

30

When types are not accurate enough to specify the desired contract

Sometimes the types of a function are not accurate enough.

- For the input: `sum_to` will not terminate normally on some `int` inputs.

Here, we can check early in the function that the input is not valid for the function.

- For the output: some functions may do some work only to discover that they cannot return an output of the specified type.

For example, a function `square_root : int -> int` might return an `int` only when the input is a square and fail otherwise.

`square_root 9` evaluates to `3` but `square_root 8` fails.

We may call these function “incomplete” or “non-total” since they do not return results for all the values in their domain (as indicated by the types of their inputs).

How do we handle these situations?

31

Technique: exceptions for non-total functions

- One way we can “give up” is to raise an exception.
- For now, we’ll use the `Failure` exception, as follows:

```
let rec sum_to (n: int) : int =  
  if n < 0  
  then raise (Failure "sum_to does not allow negative input")  
  else if n = 0  
  then 0  
  else n + sum_to (n-1)
```

- In other settings, a different string would be given to `Failure`.
- We’ll look at `option` types later as another solution to this problem.

32

Lists and tuples

We now turn to some more traditional data structures.

- lists — unbounded number of elements of the same type
- tuples — bounded number of elements of different types

33

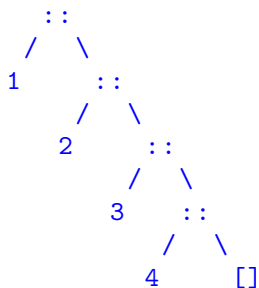
Lists

- ▶ Lists are ubiquitous in functional languages and sometimes serve the same purpose as arrays in imperative languages.
- ▶ Literals
`[]`, `[1; 2; 3; 4]`
- ▶ Value constructors
 1. `[]` (the empty list)
 2. `::` (read “cons”)
- ▶ `1 :: [] = [1]`
- ▶ `1 :: 2 :: 3 :: [] = [1; 2; 3]`
- ▶ Append operator: `@`
`[1; 2; 3] @ [4; 5; 6] = [1; 2; 3; 4; 5; 6]`

34

Lists are just funny looking trees

- ▶ Lists are structured, inductive data, as we saw on Friday.
- ▶ Lists are constructed by either `[]` or `::`.
 - ▶ The `[]` nodes have no children.
 - ▶ The `::` nodes have 2 children: a value and another list.
- ▶ The syntax `[1; 2; 3; 4]` is just “short-hand” for `1 :: 2 :: 3 :: 4 :: []`.
- ▶ This list has structure, like a tree that slants to the right.



35

List type expressions

- ▶ The type of a list of integers is written `int list`.
- ▶ `list` is not a type, it is a type constructor, like `->`
- ▶ It is written as a postfix operator.

36

Pattern matching and lists

- ▶ We've seen how to make lists using `[]` and `::`, but how can we deconstruct them to inspect the head and tail?
- ▶ Answer: pattern matching - "a much better switch statement"
- ▶

```
let rec sum xs =  
  match xs with  
  | [] -> 0  
  | x::rest -> x + sum rest
```
- ▶ `match ... with clauses`
- ▶ A `clause` has the form:
| *pattern* -> *expr*
- ▶ Let's write some list functions; these will be in `lists_tuples.ml` in the public repo₃₇

Exercise S1.2: #5:

Write an OCaml function named `all` that returns `true` if all elements of the list are `true`.

What is the type of this function?

That is, if there are no elements that are `false`.

Recall `sum`:

```
let rec sum xs =  
  match xs with  
  | [] -> 0  
  | x::rest -> x + sum rest
```

38

Exercise S1.2: #6: Even 2 ways

Write a function `even2ways` that checks if an integer list only contains even values and has an even number of elements.

```
let rec even2ways (xs : int list) : bool = ...
```

More interesting patterns

- ▶ Consider a function for concatenating string with a separator.
- ▶ `string_concat : string -> string list -> string`
- ▶ `string_concat "," ["a"; "b"; "c"]` evaluates to `"a,b,c"`
- ▶ `string_concat "," ["a"]` evaluates to `"a"`
- ▶ `string_concat "," []` evaluates to `"`

40

Matching deeper into the list structure

- ▶ Recall our function for checking that a list has even length.
- ▶ Can we use pattern matching to step through the list 2 elements at a time until we get to the empty list or the list with one element?
- ▶ Yes, let's write that in `getting_started.ml` and call the function `even_length_v3`.

41

Pattern matching and name declarations

Recall:

```
let rec sum nums =  
  match nums with  
  | [ ] -> 0  
  | x::xs -> x + sum xs
```

- ▶ Note that `x` and `xs` are **declarations** just like `sum` and `nums` in the first line.
- ▶ If `nums` is `[]` then

`[]` ← the data



`[]` ← the pattern

- ▶ pattern `[]` matches the data `[]`
- ▶ thus the value of the `match` expression is the value of the expression in this clause - that is `0`.

42

Pattern matching and name declarations

Recall:

```
let rec sum nums =  
  match nums with  
  | [ ] -> 0  
  | x::xs -> x + sum xs
```

► Note that `x` and `xs` are **declarations** just like `sum` and `nums` in the first line.

► If `nums` is `1 :: 2 :: 3 :: []` then
`1 :: 2 :: 3 :: []` ← the data

 ← the pattern

- the pattern `[]` does not match `::`
- thus, we move on to try the next pattern in the sequence

43

Pattern matching and name declarations

Recall:

```
let rec sum nums =  
  match nums with  
  | [ ] -> 0  
  | x::xs -> x + sum xs
```

► Note that `x` and `xs` are **declarations** just like `sum` and `nums` in the first line.

► If `nums` is `1 :: 2 :: 3 :: []` then
`1 :: 2 :: 3 :: []` ← the data

 ← the pattern

1. `::` (in the pattern) matches `::` (in the data), then
2. `x` matches `1`, and has this value in the expression, then
3. `xs` matches `2 :: 3 :: []`, and has this value in the expression

44

More pattern matching and name declarations

Recall:

```
let rec string_concat sep strs =  
  match strs with  
  | [ ] -> ""  
  | s::[] -> s  
  | s1::s2::[] -> s1 ^ sep ^ s2  
  | s::rest -> s ^ sep ^ string_concat sep rest
```

► Again, `s`, `s1`, `s2`, `s` (again), and `rest` are **declarations** just like `string_concat`, `sep`, and `strs` in the first line.

► Let's consider different data and how it matches different patterns.

► (Yes, the 3rd pattern is no necessary but it is useful in understanding pattern matching.)

45

More pattern matching and name declarations

Recall:

```
let rec string_concat sep strs =  
  match strs with  
  | [] -> ""  
  | s::[] -> s  
  | s1::s2::[] -> s1 ^ sep ^ s2  
  | s::rest -> s ^ sep ^ string_concat sep rest
```

- When `strs` is `"a" :: "b" :: "c" :: []` does it match the **first** pattern?

`"a" :: "b" :: "c" :: []` ← the data



← the pattern

- No. the pattern does not match the data.

46

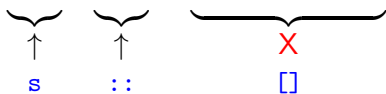
More pattern matching and name declarations

Recall:

```
let rec string_concat sep strs =  
  match strs with  
  | [] -> ""  
  | s::[] -> s  
  | s1::s2::[] -> s1 ^ sep ^ s2  
  | s::rest -> s ^ sep ^ string_concat sep rest
```

- When `strs` is `"a" :: "b" :: "c" :: []` does it match the **second** pattern?

`"a" :: "b" :: "c" :: []` ← the data



← the pattern

1. `::` (in the pattern) matches `::` (in the data), then
2. `s` matches `"a"`, and has this value in the expression, then
3. `[]` **does not** match the `[]` in `2 :: 3 :: []`, so no match.

47

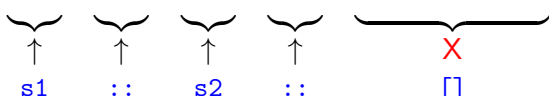
More pattern matching and name declarations

Recall:

```
let rec string_concat sep strs =  
  match strs with  
  | [] -> ""  
  | s::[] -> s  
  | s1::s2::[] -> s1 ^ sep ^ s2  
  | s::rest -> s ^ sep ^ string_concat sep rest
```

- When `strs` is `"a" :: "b" :: "c" :: []` does it match the **third** pattern?

`"a" :: "b" :: "c" :: []` ← the data



← the pattern

- No. As above, matching two `::` and `s1` and `s2` succeed, but `[]` **does not** match the `[]` in `"c" :: []`, so no match.

48


More pattern matching and name declarations

Recall:

```
let rec string_concat sep strs =  
  match strs with  
  | [] -> ""  
  | s::[] -> s  
  | s1::s2::[] -> s1 ^ sep ^ s2  
  | s::rest -> s ^ sep ^ string_concat sep rest
```

- When `strs` is `"a" :: "b" :: "c" :: []` how does it match the **fourth** pattern?

`"a" :: "b" :: "c" :: []` ← the data



`s :: rest` ← the pattern

- Yes!

1. `::` (in the pattern) matches `::` (in the data)
2. `s` matches `"a"`, and has this value in the expression
3. `rest` matches `"b" :: "c" :: []` and has this value in the expression

49

More pattern matching and name declarations

Recall:

```
let rec string_concat sep strs =  
  match strs with  
  | [] -> ""  
  | s::[] -> s  
  | s1::s2::[] -> s1 ^ sep ^ s2  
  | s::rest -> s ^ sep ^ string_concat sep rest
```

- Patterns are matched in order, from top to bottom.
- The **first** one that matches is taken.
- Moving the fourth pattern to be the second would prevent the other two from ever matching since the fourth one is more general.
- That is, it matches any data that the other two do, and then some more.

50

Exercise S1.2: #7: Checking empty lists

Write an OCaml function named `is_empty` that returns `true` if the list is empty, and `false` otherwise.

List functions

- ▶ OK, what is your solution?
- ▶ Do your patterns match those in my solution?
- ▶ Good style: use `_` in patterns when the value matched is not used in the corresponding expression.

52

sum and length

Compare

- ▶

```
let rec sum xs =  
  match xs with  
  | [] -> 0  
  | x::rest -> x + sum rest
```
- ▶

```
let rec length xs =  
  match xs with  
  | [] -> 0  
  | _::rest -> 1 + length rest
```

53

Parametric polymorphism

- ▶ `sum: int list -> int`
- ▶ `is_empty: 'a list -> bool`
- ▶ `length: 'a list -> int`
- ▶ What is different (and interesting) about these types?
- ▶ `is_empty` and `length` take lists of any type, while `sum` takes only lists integers
- ▶ This idea is called *generics* in Java, implemented rather poorly as *templates* in C++.

54

Exercise S1.2: #8:

Write an OCaml function named `head` that returns the front element of the list.

This specification is intentionally incomplete - what decisions must you make to complete this function?

What is its type?

55

Incomplete patterns

- ▶ OK, please share your solution.
- ▶ Let's consider different variations.
- ▶ OCaml warns you if your patterns are “*non-exhaustive*”
It even tells you what you're missing.
- ▶ What *should* we do for our function?
- ▶ What does OCaml do?

56

Exercise S1.2: #9:

Will this function work? Will `drop_value 2 [1;2;3]` evaluate to `[1; 3]`?

```
let rec drop_value to_drop l =  
  match l with  
  | [] -> []  
  | to_drop :: tl -> drop_value to_drop tl  
  | hd :: tl -> hd :: drop_value to_drop tl
```

How to turn questions that do not ask for OCaml code [online](#):

- ▶ put these in a file named `mm_dd.md`
- ▶ These are Markdown files and are what your syllabus and assignments are written in.
Take a look at these to see how this document creation language works.
- ▶ Create a *h2* header, by writing `##` followed by the exercise number - in this case `8`.
That is add a line
`## 8`
- ▶ put your answer after this line
- ▶ commit and push after lecture

57

Exercise S1.2: #10:

Fix this function so that `drop_value 2 [1; 2; 3]` evaluates to `[1; 3]`.

```
let rec drop_value to_drop l =  
  match l with  
  | [] -> []  
  | to_drop :: tl -> drop_value to_drop tl  
  | hd :: tl -> hd :: drop_value to_drop tl
```

58

Recursive list functions

- ▶ Many functions over lists use a `match` with a clause for each of the 2 constructors:
 - ▶ `[]` and
 - ▶ `::`
- ▶ Lists are `inductive data`, processed by `recursive functions`.
- ▶ The structure of the data matches the structure of the function.
- ▶ That previous point is important!
- ▶ The OCaml `function` syntax is a useful shortcut.

```
let rec sum_v2 = function  
  | [] -> 0  
  | x::rest -> x + sum_v2 rest
```

59

Tuples

Recall the following from an earlier slide:

- ▶ lists — unbounded number of elements of the same type
- ▶ tuples — bounded number of elements of different types

For example:

- ▶ a pair with an `int` and a `string` such as `(1, "one") : int * string`
- ▶ a pair of two `int` values, such as `(1,2) : int * int`
- ▶ a triple with an `int` and two `trees`, such as
`(3, Empty, Fork (4, Empty, Empty)) : int * tree * tree`
- ▶ an `int` and `char list`, such as `(3, ['a'; 'b'; 'c']) : int * char list`

60

Tuples

- ▶ At first glance, tuples are like records but elements are identified by position, not by field name.

- ▶ These are called **product types**.

The values of a tuple type of 2 elements are those in the Cartesian product of the element types.

- ▶ Pattern matching is used here as well.

But these patterns are “**irrefutable**”.

This is because there is only one constructor for a tuple.

In contrast, lists have two: `[]` and `::`, and we have to check which one it is.

61

Example: adding values in a `int * int` pair

We can use a `match` to pattern match on a pair to get its values out:

```
let add_pair (p: int * int) : int =  
  match p with  
  | (n1, n2) -> n1 + n2
```

Because they are irrefutable, we can use patterns in let bindings:

```
let add_pair_v2 (p: int * int) : int =  
  let (n1, n2) = p in n1 + n2
```

We can even move this match into the function arguments:

```
let add_pair_v3 ((n1, n2) : int * int) : int =  
  n1 + n2
```

62

Exercise S1.2: #11:

Write a function named `first_of_3` to return the first element of a triple.

What is its type?

Write the type of `first_of_3` in a comment above the function definition.

(Put this in today's `.ml` file if you're watching online.)

63

Technique: Tuples for returning multiple values

- ▶ Functions return only a single value, but what if we want to return two?
e.g. a function that returns both the square and cube of an integer.
- ▶ The “single value” the function returns can be a tuple value.
- ▶ So wrap up the multiple values in a tuple to make a “single value.”
- ▶ Let’s write `square_cube`.

64

Technique: Getting values out of tuples

We just did this a few slides ago with `add_pair`.

Another example:

```
let square_cube (n: int) : (int * int) =  
  let sq = n * n in (sq, sq * n)  
  
let add_square_cube (n: int) : int =  
  let sc = square_cube n in  
  match sc with  
  | (s, c) -> s + c  
  
let add_square_cube_shorter (n: int) : int =  
  match square_cube n with  
  | (s, c) -> s + c  
  
let add_square_cube_shortest (n: int) : int =  
  let (s, c) = square_cube n in s + c
```

65

Revisiting curried functions

- ▶ Languages you’ve seen before just pass all their arguments to a function in a tuple.
- ▶ They pass all their arguments at once.
- ▶ Curried functions let you pass multiple values to a function as well, but they do it one at a time.
- ▶ This leads to extra flexibility and thus reusability - one of the goals we discussed at the beginning of the course.

66

Exercise S1.2: #12:

What type would you use to represent fractions?

We can give names to interesting types, by writing

```
type fraction = ... your answer to this question
```

67

Exercise S1.2: #13:

Consider a function to add two fractions.

What is its type?

What is its value?

68

Pulling the pieces together

- ▶ A *partial mapping* from, say, strings to integers could be represented by a value of the type `(string * int) list`.
- ▶

```
let m = [ ("dog", 1); ("chicken", 2);  
          ("dog", 3); ("cat", 5) ]
```
- ▶ `lookup_all "cat" m` evaluates to `[5]`
- ▶ `lookup_all "moose" m` evaluates to `[]`
- ▶ `lookup_all "dog" m` evaluates to `[1; 3]`
- ▶ Can we write `lookup_all`?

69

Exercise S1.2: #14:

- ▶ A *partial mapping* from, say, strings to integers could be represented by a value of the type `(string * int) list`.
- ▶ `let m = [("dog", 1); ("chicken", 2);
 ("dog", 3); ("cat", 5)`
- ▶ `lookup_all "cat" m` evaluates to `[5]`
- ▶ `lookup_all "moose" m` evaluates to `[]`
- ▶ `lookup_all "dog" m` evaluates to `[1; 3]`
- ▶ Write `lookup_all`. What is its type?

70

Exercise S1.2: #15:

Rewrite `fib` to use pattern matching.

`fib` was:

```
let rec fib x =  
  if x = 0 then 0 else  
    if x = 1 then 1 else fib (x-1) + fib (x-2)
```

71

What goes wrong with programs?

- ▶ Programs, of course, may have errors.
- ▶ We may detect these statically - before the program runs. Typically this is when the compiler runs or the program is loaded into the interpreter.
- ▶ Or dynamically - when the program runs.
- ▶ For example, syntax errors are statically detected by a compiler. Division by 0 is detected at run-time.

72

Exercise S1.2: #16:

What other types of errors are can you name?

List as many as can.

Also note when they can be detected? By a compiler, or only at run-time.

(This goes in a `.md` file in you're watching online.)

73

Errors in programs - static errors

Some errors will be detected automatically, others will not be.

Static errors - the best kind!

We are told of a problem without needing to run the program.

- ▶ syntax errors
- ▶ static type errors

74

Errors in programs - dynamic errors

Dynamic errors - these are detected during the execution of the program.

The program terminates abnormally.

Well, at least we know something was wrong, even if it is a bit late.

- ▶ division by 0 - the processor sends an interrupt
- ▶ memory accesses outside of process' memory space
- ▶ type errors in dynamically-typed languages (Python, Clojure)
- ▶ exception that detect programmer specified errors.

75

Errors in programs - not detected

The program just keeps going, with bogus data - Oh no...

- ▶ Invalid operations that do not fail.
- ▶ Invalid memory accesses *inside* of process' memory spaces
- ▶ Adding a string to an integer in an untyped language (machine code).

76

```
3                false        "World"

fun x -> x * x     [false; false; false ]

"Hello"          (4, 'z')

fun x -> x + 1     -10

fun n -> int_of_string n

[ 1; 2; 3 ]       true

[ true; false ]   [ 4; 5; 6 ]

(1, 'c')          45

[ fun x -> x + 1 ; fun x -> x * x ]
```

77

```
int
  3                -10          45
bool
  true            false
string
  "Hello"         "World"
int -> int
  fun x -> x + 1    fun x -> x * x
int -> string
  fun n -> int_of_string n
int list
  [ 1; 2; 3 ]      [ 4; 5; 6 ]
bool list
  [ true; false ]  [false; false; false ]
int * char
  (1, 'c')         (4, 'z')
(int -> int) list
  [ fun x -> x + 1 ; fun x -> x * x ]
```

78

Organizing those values above.

- ▶ There are many types of values.
- ▶ We will group them by types.
- ▶ A **type** is a name for a set of **set of values**.
- ▶ **int** is a set of values
- ▶ **int list** is a set of values
- ▶ **'a list** is a set of values
It contains **int list** and **string list** and
int list list ...

79

Strong static type systems

- ▶ OCaml has a *strong, static* type system
- ▶ It is a *safe* language.
- ▶ What does "safe" mean?
- ▶ *strong* = program never execute type-incorrect operation or invalid memory access
- ▶ *static* = this is checked before the program runs.
- ▶ *safe* = strong, static type system

80

Expressiveness of types

- ▶ OCaml doesn't detect division by 0, but the hardware will.
- ▶ Since the hardware doesn't detect type-incorrect operations or invalid memory accesses (within the users allotted memory space), OCaml must prevent these.
- ▶ So OCaml can detect all invalid operations
 - ▶ some through the static type system
 - ▶ some through dynamic (run-time) checks

81

Static vs Dynamic Typing

- ▶ A *static* type system works at compile time, before the program runs, to detect type errors.
 - ▶ Java, C, OCaml, Haskell have static type systems.
- ▶ A *dynamic* type system works at program run time, as the program executes.
 - ▶ Python, Scheme, Ruby, Clojure have dynamic type systems.
- ▶ Static type systems are preferred
 - ▶ Error are detected when the programmer can fix them.
 - ▶ Statically-typed languages are more efficient since run-time checking of types is avoided.

82

Type systems

The challenge - design strong static type systems that are

1. expressive, and

- ▶ It is difficult to have a static type for non-zero integers.
- ▶ So the question becomes

“What properties can types express?”

2. easy to use.

- ▶ Type inference can help with this as we don't need to write down all the types. But it is recommended to write types for some parts to provide machine-checked documentation.

83

Different varieties of phrases

There are *different kinds* of phrases in OCaml and other languages.

1. “expressions” - that evaluate to a value

2. “statements” - in C, Java, etc.

These perform some action, perhaps changing a value in memory.

3. “types” - a sub-language for “type expressions”

4. “declarations” - declaring new names/variables

5. “patterns” - that match values

84

The language of types

- ▶ Types are an important “sub-language” in OCaml and other languages.
- ▶ There are constants: `int`, `float`
- ▶ Variables: `'a`, `'b`
- ▶ Operators: `list`, `->`, `*`

These form a proper language of types.

85

Exercise S1.2: #17: Operator precedence and associativity

- ▶ What is the precedence of `*` compared to `->` ?
How could we find out?
- ▶ What is the associativity of `*` ?
Does it matter?

86

Wait, is `*` an operator?

Try:

- ▶ `let x : int * int * int = (1,2,3) ;;`
- ▶ `let x : (int * int) * int = (1,2,3) ;;`
- ▶ `let x : int * (int * int) = (1,2,3) ;;`

We really have several “mix-fix” operators.

`*` and `*...*` and `*...*...*`

87

Types as an organizing principle

- ▶ Types (or type expressions) provide the first approximation of understanding what a function does.
- ▶ Without a proper language of types, it is difficult to even think in these terms.
- ▶ This is missing in dynamically typed languages like Scheme, Clojure, Python, etc.
- ▶ In this regard, *static* checking is not the point. Being able to think properly about types - if they are checked at compile time, at runtime, or never(!) - is what matters now.
- ▶ Thinking in terms of types shapes our thinking and helps us design programs.

88

Recap

- ▶ Parametric polymorphism.
- ▶ Lists as inductive data structures.
- ▶ Language of types.
- ▶ Types as an organizing principle.

89