CSC 330 / Spring 2026

OCaml
Records, Variants

Ibrahim Numanagić /nu-ma-nagg-ich/

based on Dan Grossman's lecture materials

What is a programming language?

1. **Syntax**: how to write programs down (spelling and grammar)?
2. **Semantics**: what programs mean (type checking and evaluation)?
3. **Idioms**: what are typical ways to use language features for commn needs?
4. **Libraries**: what is standard and easily available?
5. **Tools/: Ecosystem**: what support do you get?
    - REPL: IDE, debugger, message board, governance, standards, etc.

- We will mostly focus on semantics and idioms
- Carefully thinking about these makes us better programmers in *any* language
- Other parts of a PL are important and interesting (especially when trying to do any "real work")
- I can rant a lot how OCaml sucks at 4. and 5.

A side note…

We went so far without even seeing a Hello World in OCaml!

```
print_string "Hello world!\n"
```

That’s all, folks!

🤨: What kind of construct is this? How does this typecheck?

Another important note…

- Thus far, you are playing with OCaml through a REPL (Read-Eval-Print Loop)
- Visual Studio Code uses OCaml LSP (Language Server Protocol) extension to provide highlighting, error checking etc.
- However, that is **not** how you develop large OCaml apps!
  - OCaml is a *compiler*, and you can run it as:

    ```
    ocamlc -o program.exe program.ml
    ```

  - You can use `ocamlopt` instead of `ocamlc` to produce native binaries
  - For more complex projects, use a build system such as dune
  - OCaml can be significantly “upgraded” by using Jane Street’s Core library (terms and conditions apply, of course)

Building new types

- Already know many ways to build types
  - Base types: `bool`, `int`, `float`, `string` …
  - Type constructors: `t1 * t2`, `t1 -> t2`, `t list`, `t option`
- Can we define our own new types in OCaml? **YES**!
  - Examples: enumerations, trees, etc.
  - Type structure is often more important to program design than algorithms
- But first: More general way to think about types in any language

Anatomy of types in a language

- Languages generally provide *three kinds of building blocks* to define types:
  - Two ways to combine existing types `t1, t2, ..., tN` in a new type `t`
    - **each of**: a value of type `t` contains values of *each of* `t1, ..., tN`
    - **one of**: a value of type `t` contains values of *one of* `t1, ..., tN`
  - A way for values of type `t` to contain other (smaller) values of type `t`
    - **self reference**: a value of type `t` can contain other `t` values

The ability to nest and mix-and-match these blocks (a.k.a. *composition*) makes this even more powerful 💪!

Example types from building blocks

- Tuples are *each of*: an `int * bool` contains an `int` *and* a `bool`
- Options are *one of*: an `int option` contains an `int` *or* no data
- Lists throw in *self references* to use all three building blocks: an `int list` contains either `int` *and another* `int list`, *or* no data
- **Nesting** lets us build all kinds of interesting types
  - `((int * bool) option * (string list list)) option list`

- Types provide a compact language for expressing the “shape” of data

Records and variants

- **Records**: a new way to build each-of types, a little different than tuples
  - Named types
  - Named fields
- **Variants**: a new way to build our own one-of types
  - Named types
  - Quite different than most Java/Python stuff you've ever seen
  - Example: define a type whose values hold an `int` *or* a `string`
  - To *build* variant values, type definitions include constructors like `Some`, `None`, `::`, `[]` etc.
    - These are **not related** to Java or C++ constructors
  - To use variant values, match expressions test and unpack them

Records

Records: definition

- **Syntax**:

  ```
  type tname = { f1: t1; ...; fN: tN }
  ```

  where:

  - `t1, ..., tN` are (already-defined) types
  - `f1, ..., fN` are *field names* with same “spelling” rules as variables
  - Adds new (record) type `tname` and fields `f1, ..., fN`
    - Types are in their own namespace: types don’t conflict with variables/functions, fields
    - Field names are in their own name space: fields don’t conflict with variables/functions, types
    - However, they can **shadow** earlier type and field name definitions ⚠️
  - You must define a record type before you can build or use
    - That’s just OCaml’s rule (not fundamental)

Example

```
type lava_lamp = { height : float
                 ; color_liquid : string
                 ; color_lava : string }
let lamp = { height = 10.0; color_liquid = "yellow"; color_lava = "red" }

print_float lamp.height
```

Records: building

- **Syntax**:

  `{ f1: e1; ...; fN: eN }`

  where `e1, ..., eN` are expressions, and `f1, ..., fN` are *field names*
- **Type checking**:
  - If `t` is defined as type `t = { f1: t1; ...; fN: tN }`, and `e1` has type `t1`, and …, and `eN` has type `tN`:
    - then: `{ f1 = e1; ...; fN = eN }` has type `t`
    - else: report error and fail
- **Evaluation**:
  - If `e1` evaluates to value `v1`, and …, and `eN` evaluates to value `vN`:
    - then: `{ f1: e1; ...; fN: eN }` evaluates to `{ f1 = v1; ...; fN = vN }`

- ⚠ Elements are separated by semicolons `;` (you will get very weird error messages otherwise)!
- Records of values are values

Records: using

- **Syntax**: `e.f` where `e` is an expression and `f` is a field name
- **Type checking**:
  - If `e` has type `t`, and `t` is defined as `type t = { f1: t1; ...; f: tF; ...; fN: tN }`:
    - then: `e.f` has type `tF`
    - else: report error and fail
- **Evaluation**:
  - If `e` evaluates to `{ f1 = v1; ...; f = vF; ...; fN = vN }`:
    - then: `e.f` evaluates to `vF`

Tuples vs. records

- Semantically, there's no much difference. Compare:
  - `(1, true, "three")` vs. `{ a = 1; b = true; c = "three" }`
  - Tuples a little shorter, records easier to remember *what is where* (self-documenting)
  - Avoid large tuples and consider records when multiple fields have the same type
- Common language design question: **by position** (tuples) or **by name** (records)?
  - Functions do some of each: *position* for caller but *name* for callee

Demonstration 1: Records

Demonstration 1: Records

```
# (* Records have the same "expressive power" as tuples, just with
 * user-defined field names and different syntax for building and using
 * but our first time making our own new type (!) *)

type lava_lamp = { height : float
                 ; color_liquid : string
                 ; color_lava : string }

let my_lamp1 = { height = 13.5 +. 1.0
               ; color_liquid = "bl" ^ "ue"
               ; color_lava = "" ^ "green" ^ "" }

let my_lamp2 =
  { height = 14.4; color_liquid = my_lamp1.color_liquid; color_lava = "x" }

let a = my_lamp1.height
let b = my_lamp1.color_liquid
let c = my_lamp1.color_lava ;;
```

Demonstration 1: Records

```
# (* Records have the same "expressive power" as tuples, just with
 * user-defined field names and different syntax for building and using
 * but our first time making our own new type (!) *)

type lava_lamp = { height : float
                 ; color_liquid : string
                 ; color_lava : string }

let my_lamp1 = { height = 13.5 +. 1.0
               ; color_liquid = "bl" ^ "ue"
               ; color_lava = "" ^ "green" ^ "" }

let my_lamp2 =
  { height = 14.4; color_liquid = my_lamp1.color_liquid; color_lava = "x" }

let a = my_lamp1.height
let b = my_lamp1.color_liquid
let c = my_lamp1.color_lava ;;
 type lava_lamp = {
   height : float;
   color_liquid : string;
   color_lava : string;
 }
 val my_lamp1 : lava_lamp =
   {height = 14.5; color_liquid = "blue"; color_lava = "green"}
 val my_lamp2 : lava_lamp =
   {height = 14.4; color_liquid = "blue"; color_lava = "x"}
 val a : float = 14.5
 val b : string = "blue"
 val c : string = "green"
```

Demonstration 1: Records

Demonstration 1: Records

```
 # let concat_liquid_colors ((lamp1 : lava_lamp), (lamp2 : lava_lamp)) =
   lamp1.color_liquid ^ " " ^ lamp2.color_liquid

let epsilon = 0.0001
let same_height (lamp1, lamp2) = Float.abs (lamp1.height -. lamp2.height) < epsilon
let is_same = same_height (my_lamp1, my_lamp2) ;;
```

Demonstration 1: Records

```
# let concat_liquid_colors ((lamp1 : lava_lamp), (lamp2 : lava_lamp)) =
  lamp1.color_liquid ^ " " ^ lamp2.color_liquid

let epsilon = 0.0001
let same_height (lamp1, lamp2) = Float.abs (lamp1.height -. lamp2.height) < epsilon
let is_same = same_height (my_lamp1, my_lamp2) ;;
val concat_liquid_colors : lava_lamp * lava_lamp -> string = <fun>
val epsilon : float = 0.0001
val same_height : lava_lamp * lava_lamp -> bool = <fun>
val is_same : bool = false
```

Variants

Variants

- Variants are not like records: **one-of types are not each-of types**!
- Let's use variant types to *enumerate* a fixed set of values for some new type
  - This is the simplest use of variant types
  - We will see more advanced use-cases soon!
- We name a new type and the values of that type are the constructors:

```
type si_unit =
  | Second | Meter | Kilogram | Ampere
  | Kelvin | Mole | Candela
```

Variants: building

```
type si_unit =
  | Second | Meter | Kilogram | Ampere
  | Kelvin | Mole | Candela
```

- Syntax: `First | optional`
  - Constructors must start with capital letter!
- Semantics: adds to environment the type name and the constructors
- These 7 constructors are now values (and therefore expressions)
  - 7 *different* ways to build a value of the new type `si_unit`
  - There are no *other* values of the type `si_unit`
    - e.g., `[Second; Meter; Second]` has type `si_unit list`

Booleans

- One of the simplest enumerations is our old friend `bool`
- Could almost be defined

```
type bool = true | false
```

- Only issue is capitalization rule (the language violated their own rule)
  - We can write

    ```
    type mybool = True | False
    ```

- Anyway, we can now define enumerations and *build* values of them, so need a way to *use* values of them

Using variants with match expressions

```
let string_of_si_unit u =
  match u with
   | Second -> "second"
   | Meter -> "meter"
   | Kilogram -> "kilogram"
   | Ampere -> "ampere"
   | Kelvin -> "kelvin"
   | Mole -> "mole"
   | Candela -> "candela"
let s =
  string_of_si_unit Ampere
```

- Match expression with one branch for each constructor
- Like a multi-way if (or `switch`-`case` on steroids), with one branch for each constructor
- Guides type checking: all branches must have same type and **every constructor must have a branch**
- Guides evaluation rules: evaluate exactly one branch

Back to `bool`

```
if e1 then e2 else e3
```

or

```
match e1 with
| true -> e2
| false -> e3
```

- In fact, you can use match expressions instead of conditional expressions
  - Yes, this works in OCaml!
- The right side is, however, **not** good style
  - Demonstrates that one feature (conditional expressions) can be explained entirely in terms of another more powerful feature (match expressions)
  - Another example of **syntactic sugar** (many more to come)

Beyond enumerations

- As promised, OCaml variant types are much more powerful
- Each constructor can carry data (or not)
  - Type definition indicates the type of data each constructor carries
- Self-reference allowed in type definition
  - Can carry “another of the same thing”
- See the last Demonstration for examples:
  - A silly example to learn the syntax and semantics
  - An example for different kinds of shapes
  - An example that uses self-reference to define arithmetic-expression trees

Demonstration 2: Variants

Demonstration 2: Variants

```
# (* Enumerations *)
type si_unit = Second | Meter | Kilogram | Ampere | Kelvin | Mole | Candela
let ss = [ Second; Meter; Second ]
let string_of_si_unit (u : si_unit) : string =
  match u with
  | Second -> "second"
  | Meter -> "meter"
  | Kilogram -> "kilogram"
  | Ampere -> "ampere"
  | Kelvin -> "kelvin"
  | Mole -> "mole"
  | Candela -> "candela"
let sa = string_of_si_unit Ampere

type si_prefix = Giga | Mega | Kilo | Milli | Micro | Nano
let scale p =
  match p with
  | Giga -> 1e9
  | Mega -> 1e6
  | Kilo -> 1e3
  | Milli -> 1e-3
  | Micro -> 1e-6
  | Nano -> 1e-9
let sg = scale Giga ;;
```

Demonstration 2: Variants

```
 # (* Enumerations *)
type si_unit = Second | Meter | Kilogram | Ampere | Kelvin | Mole | Candela
let ss = [ Second; Meter; Second ]
let string_of_si_unit (u : si_unit) : string =
  match u with
   | Second -> "second"
   | Meter -> "meter"
   | Kilogram -> "kilogram"
   | Ampere -> "ampere"
   | Kelvin -> "kelvin"
   | Mole -> "mole"
   | Candela -> "candela"
let sa = string_of_si_unit Ampere

type si_prefix = Giga | Mega | Kilo | Milli | Micro | Nano
let scale p =
  match p with
   | Giga -> 1e9
   | Mega -> 1e6
   | Kilo -> 1e3
   | Milli -> 1e-3
   | Micro -> 1e-6
   | Nano -> 1e-9
let sg = scale Giga ;;
 type si_unit = Second | Meter | Kilogram | Ampere | Kelvin | Mole | Candela
 val ss : si_unit list = [Second; Meter; Second]
 val string_of_si_unit : si_unit -> string = <fun>
 val sa : string = "ampere"
 type si_prefix = Giga | Mega | Kilo | Milli | Micro | Nano
 val scale : si_prefix -> float = <fun>
 val sg : float = 1000000000.
```

Variants: definition

```
type silly =
| A of int * bool * string list
| Foo of string
| Pizza
```

- **Syntax**:

  ```
  type tname = | C1 [of t1] | ... | CN [of tN]
  ```

  where `t1, ... tN` are types (and can include `tname` inside them), and `C1, ... CN` are **constructors**

  - Adds new (variant) type tname and constructors `C1, ... CN`
    - Types are in their own namespace
    - Constructors are in their own namespace
    - However, they can *shadow* earlier type and constructor definitions!
  - Must define a variant type before you can build or use it

Remember: `[...]` is *metasyntax* for optional

Variants: building

- **Syntax**:

  `C e`

- **Type checking**:
  - If `C` is in the environment due to type `t = ...| C of tC | ...`, and `e` has type `tC` (must provide correct arguments to `C`):
    - then: `C e` has type `t`
    - else: does not typecheck
- **Evaluation**: evaluate `e` to a value `v` and `C v` **is a value**
  - Basically, tag the data `v` with a `C`, producing a value of type `t`

Match expressions: the big picture

- A powerful and concise expression form that combines three things:
  - A multiway conditional that branches based on which variant of a one-of type you have
  - Extract the underlying data that was “tagged” with a constructor
  - Bind the extracted data to local variables that can then be used in the chosen conditional branch
- This is *very* elegant (once you get used to it) and avoids errors like forgetting cases or trying to extract the *wrong* data for the tag you have.

Match expressions: using variants

```
type silly =
| A of int * bool * string list
| Foo of string
| Pizza
```

- **Syntax** (will generalize this soon!):

  ```
  match e with p1 -> e1 | ...| pN -> eN
  ```

  where `e, e1, ..., eN` are expressions and `p1, ..., pN` are **patterns**

  - For now, each pattern should be one of these:
    - `C` if `C` is a constructor that carries no data
    - `C (x1, x2, ..., xn)` if `C` is a constructor that carries an N-tuple of data
    - `C x` otherwise
    - Here, `x1, ... xn` and `x` are *variables*

Patterns

- Patterns are not expressions, even though they kind of look like them
- They are used for *matching* and to *bind local variables when they match*

Match expressions: evaluation

- **Evaluation** (again, will generalize soon):

```
match e with p1 -> e1 | ...| pN -> eN
```

  - Evaluate `e` to some `C v` or just `C`
  - Find the pattern `pi` made with constructor `C`
  - Create local bindings:
    - `C` matching `C` creates no bindings
    - `C v` matching `C x` creates `x` bound to `v`
    - `C (v1, ..., vN)` matching `C (x1, ..., xN)` creates `x1` bound to `v1` , and …, and `xN` bound to `vN`
  - Evaluate the corresponding `ei` using these local bindings to produce an overall result

Match expressions: type checking

- Lots is going on, toward two goals:
  - All the types to work out so the evaluation rules won’t “run into problems”
  - Require one pattern for each constructor
    - Don’t forget any
    - Don’t repeat any

Match expressions: type checking

- **Type checking**:

```
match e with p1 -> e1 | ...| pN -> eN
```

  - Typecheck `e` to some type `t` where `type t = | C1 [of t1 ] | ... | CN [of t1]`
  - For each constructor `Ci` in `C1 ... CN`:
    - require exactly one pattern uses `Ci`
    - require the pattern has *the right number of variables*
    - require that the corresponding `ei` typechecks in an environment where the variable(s) in the pattern have the type(s) indicating in the variant type definition
  - All `ei` need to have the same type and that is the overall type

Demonstration 3: Variants

Demonstration 3: Variants

```
# (* Now variant types where one or more constructors carry [typed] data,
   which is much more interesting and powerful. *)
type silly = A of int * bool * string list | Foo of string | Pizza

let silly_over_silly s =
  match s with
  | A (x, y, z) -> List.hd z
  | Foo s2 -> s2 ^ s2
  | Pizza -> "ham and pineapple"

type shape =
  | Circle of float * float * float (* center-x, center-y, radius *)
  | Rectangle of float * float * float * float (* x1,y1,x2,y2 (opposite corners) *)
  | Triangle of float * float * float * float * float * float (* x1,y1,x2,y2,x3,y3 *)

let area s =
  match s with
  | Circle (x, y, radius) -> Float.pi *. radius *. radius
  | Rectangle (x1, y1, x2, y2) -> Float.abs ((x2 -. x1) *. (y2 -. y1))
  | Triangle (x1, y1, x2, y2, x3, y3) ->
      let a = x1 *. (y2 -. y3) in
      let b = x2 *. (y3 -. y1) in
      let c = x3 *. (y1 -. y2) in
      Float.abs ((a +. b +. c) /. 2.0)

let well_formed s = area s > epsilon ;;
```

Demonstration 3: Variants

```
# (* Now variant types where one or more constructors carry [typed] data,
   which is much more interesting and powerful. *)
type silly = A of int * bool * string list | Foo of string | Pizza

let silly_over_silly s =
  match s with
  | A (x, y, z) -> List.hd z
  | Foo s2 -> s2 ^ s2
  | Pizza -> "ham and pineapple"

type shape =
  | Circle of float * float * float (* center-x, center-y, radius *)
  | Rectangle of float * float * float * float (* x1,y1,x2,y2 (opposite corners) *)
  | Triangle of float * float * float * float * float * float (* x1,y1,x2,y2,x3,y3 *)

let area s =
  match s with
  | Circle (x, y, radius) -> Float.pi *. radius *. radius
  | Rectangle (x1, y1, x2, y2) -> Float.abs ((x2 -. x1) *. (y2 -. y1))
  | Triangle (x1, y1, x2, y2, x3, y3) ->
      let a = x1 *. (y2 -. y3) in
      let b = x2 *. (y3 -. y1) in
      let c = x3 *. (y1 -. y2) in
      Float.abs ((a +. b +. c) /. 2.0)

let well_formed s = area s > epsilon ;;
type silly = A of int * bool * string list | Foo of string | Pizza
val silly_over_silly : silly -> string = <fun>
type shape =
    Circle of float * float * float
  | Rectangle of float * float * float * float
  | Triangle of float * float * float * float * float * float
val area : shape -> float = <fun>
val well_formed : shape -> bool = <fun>
```

Demonstration 3: Variants

Demonstration 3: Variants

```
# let num_straight_sides s =
  (* will soon learn better style than these variable names *)
  match s with
  | Circle (x, y, r) -> 0
  | Rectangle (a, b, c, d) -> 4
  | Triangle (x1, x2, x3, x4, x5, x6) -> 3

let max_point s =
  let rec highest ps =
    (* local function assumes non-empty list *)
    if List.tl ps = [] then List.hd ps
    else
      let tl_ans = highest (List.tl ps) in
      if snd tl_ans > snd (List.hd ps) then tl_ans else List.hd ps
  in
  match s with
  | Circle (x, y, radius) -> (x, y +. radius)
  | Rectangle (x1, y1, x2, y2) ->
      highest [ (x1, y1); (x2, y2) ] (* any pt on top edge ok *)
  | Triangle (x1, y1, x2, y2, x3, y3) ->
      highest [ (x1, y1); (x2, y2); (x3, y3) ]

let a = area (Rectangle (1., 2., 3., 4.))
let nss = num_straight_sides (Triangle (5., 6., 3., 1., 2., 4.))
let mp = max_point (Circle (5., 6., 3.)) ;;
```

Demonstration 3: Variants

```
# let num_straight_sides s =
  (* will soon learn better style than these variable names *)
  match s with
  | Circle (x, y, r) -> 0
  | Rectangle (a, b, c, d) -> 4
  | Triangle (x1, x2, x3, x4, x5, x6) -> 3

let max_point s =
  let rec highest ps =
    (* local function assumes non-empty list *)
    if List.tl ps = [] then List.hd ps
    else
      let tl_ans = highest (List.tl ps) in
      if snd tl_ans > snd (List.hd ps) then tl_ans else List.hd ps
  in
  match s with
  | Circle (x, y, radius) -> (x, y +. radius)
  | Rectangle (x1, y1, x2, y2) ->
      highest [ (x1, y1); (x2, y2) ] (* any pt on top edge ok *)
  | Triangle (x1, y1, x2, y2, x3, y3) ->
      highest [ (x1, y1); (x2, y2); (x3, y3) ]

let a = area (Rectangle (1., 2., 3., 4.))
let nss = num_straight_sides (Triangle (5., 6., 3., 1., 2., 4.))
let mp = max_point (Circle (5., 6., 3.)) ;;
val num_straight_sides : shape -> int = <fun>
val max_point : shape -> float * float = <fun>
val a : float = 4.
val nss : int = 3
val mp : float * float = (5., 9.)
```