PL CONCEPTS + OCAML BASICS

CONCEPTS

- Syntax: what does the program look like, i.e. how is it formatted, how is stuff typed?
- Semantics: What does the program mean? what does it do / compute.
  - → the same syntax can have different meanings in different languages
    - e.g. `x = 2` - in python we're making an assignment
      `x = 2` - boolean expression in ocaml (if x is defined)
  - → different syntax can have the same meaning
    - e.g. `x % 2 == 0 ? x + 2 : x + 1;`
      `if x mod 2 = 0 then x + 2 else x + 1;;`
- Paradigm: a way of looking at something
  - → In programming, there's different ways a language can be described
    - the way it executes
    - the organization of code
    - the syntax or grammar
    - are there loops or is it recursive?
    - Memory management

IMPERATIVE VS. FUNCTIONAL

- Imperative: procedural, where building blocks are procedures and statements
- Functional: states the desired result, but not exactly step-by-step how to compute it
  - → Immutable: states can't be modified after creation
    - any modification creates a NEW state
    - ex: changing the value of an element in a list
  - → Higher order: Functions are passed as arguments and/or returned as results

STATIC VS. DYNAMIC

- different times of type checking
- Static: type-checking happens before the program is run (Ocaml)
- Dynamic: type-checking happens at runtime, so variables can change their data type (Python)

IMPLICIT VS. EXPLICIT

- Implicit / latent typing: the programmer does not declare the type. i.e. in python: `x = 2`
- explicit / manifest typing: the programmer declares the type. i.e. in Java: `int x = 4`

OCAML

- Compiled and bootstrapped language
- Implicitly and statically typed
- built in types: int, float, char, string, bool and unit

- composite data types include: tuples, lists, options, and variants
- arithmetic operators are NOT overloaded
  - → +, -, *, / reserved ONLY for ints
  - → +., -., *., /. reserved for floats
- tuples: heterogeneous, can have different types and vary in length
  - → (1, "hello", [4])
- lists: homogeneous, only same type, can vary in length
- cons (::) vs. @
  - → elem :: rest_of_list
    - will add elem to the BEGINNING of the list
    - elem is type 'a, rest_of_list is type 'a list
  - → this can be nested
    - elem1 :: elem2 :: elem3 :: lst
    - As long as the last thing (lst) is a list the type of the elems, you're good
  - → [lst1] @ [lst2]
    - combines lst1 and lst2 into a single list. With the elements of lst2 appending the elements of lst1
    - you can use this to add an element (elem) to the END of a list
      - → [lst] @ [elem]
- no statements in ocaml, everything is an expression, and all expressions have values (they get evaluated), and all values have types
  - → functions are expressions, and those also have a typ
- Records and variants are both user-defined types
  - → records generally used for description
  - → variants generally used for polymorphism
- currying - transformation that takes in multiple arguments as a sequence of single argument functions
  - → let f a b = a+b in let g = f 5 in g 5 ;; evaluates to 10
- shadowing - if a variable within a scope has the same name as a variable in the outer scope, the outer variable gets shadowed by the inner variable
  - → let x = 5 in let x = 6 in x + 4 ;;
- side effects - any effect that happens besides the return result
  - → print statements, exceptions, etc.