

CSC 330 / Spring 2026

Programming Languages Introduction

Luke Kuligowicz

Ibrahim Numanagić /nu-ma-nagg-ich/

based on Dan Grossman's Lecture materials

Welcome!

- ▶ This course is for you to learn some fundamental concepts of **programming languages**
- ▶ After completing it, you should:
 - become better programmers, even in languages that won't be covered
 - be able to distinguish surface differences from deeper principles
 - develop a systematic methodology for what programs mean
 - know enough technical jargon to support further study

What will I do?

- ▶ Rough summary: *experience a world of OCaml, Racket and APL clones*
- ▶ A more accurate summary:
 - Distinguish **functional programming** from **object-oriented programming**
 - Use both effectively
 - Think recursively
 - Understand **static** and **dynamic typing**
 - Reason about **naming** and **scoping** in different software settings
 - (If time permits) Explore the arcane land of array languages
 - And more weird stuff...

Programming Languages

- ▶ Fundamentals: design, implementation, theory, practice
- ▶ View fundamentals through **OCaml** , **Racket** , **Ruby** (and maybe APL clones such as **J**, **K**, or **Q**)
 - Learn a lot from seeing the same idea in different guise
 - Travel to see where you're from
- ▶ Focus on **functional programming** (FP)
 - Avoid mutation
 - Functions are values



Examples

► 🐪 OCaml:

```
let rec foldl f acc xs =  
  match xs with  
  | [] -> acc  
  | x :: xs' ->  
    foldl f (f acc x) xs'
```

► 🍏 APL:

```
life←{  
  ↑1 ω∇.^3 4=+/,~1 0 1  
  ∘.⊖1 0 1∘.ϕ<ω  
}
```

► 💎 Ruby:

```
def clean_string(str)  
  str  
    .chars  
    .each_with_object([]) {  
      |ch, obj| ch == "#" ?  
        obj.pop : obj << ch  
    }  
    .join  
end  
clean_string("aaa#b")
```

► 🌀 Racket/Scheme:

```
(require 2htdp/image)  
(let sierpinski ([n 8])  
  (if (zero? n)  
      (triangle 2 'solid 'red)  
      (let ([t (sierpinski (- n 1))])  
        (freeze (above t (beside t t))))))
```

A whole new world

- ▶ New languages
- ▶ New editors
 - I recommend using [Visual Studio Code](#)
 - Other editors ([Dr. Racket](#), Emacs, vim) are also OK (but YMMV)
- ▶ New mode of exploring (REPL)
- ▶ Set the environment as soon as possible
 - **Do not** delay fighting installation demons!

Mindset

- ▶ It's too easy to forget how mind-bending programming first was (and is!)
 - When you're thinking about something that you don't understand, you have a terrible, uncomfortable feeling called confusion
 - Learning new things often means working through confusion: that's OK and expected
- ▶ It's too easy to think everything is a nail and we only need hammers
 - We become what we behold. We shape our tools and then our tools shape us.
 - Learning a diverse set of tools makes us more robust when facing new problems!

Mindset

- ▶ Let go of everything you know
 - We'll come back around to the “mainstream” friends (Java 🤔) eventually
- ▶ Treat OCaml (and other PLs) as a new alien game we're learning to play 🙄
 - We'll describe them systematically
 - Lots of new concepts, but we'll be able to compare and contrast languages more rigorously
 - Don't try to “translate everything into Java” (or Python, or C/C++)
- ▶ I will motivate the entire course later once we establish a shared vocabulary

Syllabus

Calendar

- ▶ **Classes:**

- TWF @ 10.30 – 11.20 (all sections)

- ▶ **Office hours:**

- *Ibrahim*: T @ 11.30–13.00 in ECS 526

- ▶ (letting me know in advance is recommended)

- ▶ Office hours will commence on **Jan 20th**

- ▶ **TAs:** Mahmoud Chick Zaouali, Dennis Huang, Luke Kuligowicz

Prerequisites

► Solid programming background

- You should have mastered at least one programming language
- You should know concepts such as recursion, expectations, classes, objects, loops etc.
- If you cannot do FizzBuzz, please drop the course

► High level of computer literacy is expected

- e.g., installing new software, dealing with weird packages, using command-line tools, etc.

► Familiarity with the Unix/Linux command line **is highly recommended**

- No fancy IDEs this time: all compilers and interpreters will be invoked via command line
- Windows users might consider using WSL (I don't use Windows so I cannot provide support for it)
- If you never used command line, learn it!
 - Google “command line tutorial” (Windows) or “bash tutorial” (Linux or macOS)


First steps

1. Read and study **the syllabus** on Brightspace
2. Sign up for Microsoft Teams
 - ▶ We will use Brightspace for course materials; everything else is handled through Teams
3. Set up your [programming environment for OCaml](#)

Course materials

- ▶ No books needed!
- ▶ All required materials will be posted on Brightspace
- ▶ The course is heavily inspired by UW's CSE 341; [lots of materials can be found there](#)
- ▶ It's recommended to check out [Cornell's CS 3110](#) as well
- ▶ More materials will be posted over the course of the semester

Lectures

- ▶ Take notes! Preferably with pen and paper first 
- ▶ Interaction is encouraged: more fun and useful for everybody!
 - if you like fooling around with various PLs, feel free to chime in
- ▶ Lecture **won't** be recorded
 - Slides will be provided before each lecture
 - Code listings and annotations will be available after the lecture

Expectations (a.k.a. the most important slide)

- ▶ **Five (5) assignments:** each worth 3%, total **15%**
 - Expect one assignment every two weeks
 - Submit code and comments
 - Doesn't compile? Zero
 - Doesn't run? Zero
 - You need to get at least 50% of the combined assignment grade to pass the course.
- ▶ **Two (2) in-class midterms:** total **40%**
 - Each midterm weighs 20%
 - Expect one midterm at the end of each month
 - Pen and paper
 - ▶ In 2026? Believe it or not, even FAANG agrees with me...
- ▶ **Final exam:** total **45%**
 - You need to solve at last 30% of the final exam to pass the course.

Assignments

- ▶ Start early! Some problems may need sleep
- ▶ Questions tend to be very precise and concise
- ▶ Three late days per assignment; seven in total for the whole semester (strict)
 - No help on Teams during the late period
- ▶ There will be some bonus problems
 - Usually harder than the rest
 - Don't attempt these until the rest is done

Assignment: some hints...

► You should:

1. Understand the concepts being addressed
2. Write code demonstrating understanding of the concepts
3. Test your code to ensure you understand and have correct programs (if you don't, we will)
4. Play around with variations, incorrect answers, etc.

In general, I expect you to play and fool around with all PLs and try all sorts of combinations. If you don't like that, then this course is not for you. There's nothing more dreadful than theoretical PL exploration devoid of practice.

Also, if you prefer to prompt LLMs instead of playing with a compiler, get lost. It is way cheaper for me to pay Anthropic CAD 200 each month than to hire you for CAD 10,000+/month to do the same.

Communication

- ▶ We will use Teams for class discussion
 - Check the syllabus for details
 - You can ask private questions, post anonymously, help other students etc.
 - **Teams desktop app is a gigantic pile of <redacted>.** Use it in a browser for (slightly) better experience.
- ▶ VERY IMPORTANT RULE: **DO NOT EMAIL ME OR TAs! USE TEAMS!**
 - E-mail me only and only if it is a personal matter or a private feedback!

Cheating

- ▶ Don't cheat! Assignments and projects are supposed to be done on your own
- ▶ It is OK to use some help. But make sure to acknowledge all sources properly:
 - Say “Sami gave me an idea for the problem 3”
 - Say “I found the idea on Wikipedia: here is the link ...”
 - Say “I found this in a textbook XX by YY!”
- ▶ **DON'T ADAPT OR COPY/PASTE OTHER PEOPLE'S CODE**
- ▶ **DON'T USE LLMs!** You will fail your exams *miserably!*.
- ▶ If you get caught... heh, good luck
 - You will get zero on the assignment (really! crying and wailing won't help at all)
 - Repeated offences will be met with F, and you might get expelled from the university
 - **I take these things seriously!**

Conduct

- ▶ (Again) Read the damn syllabus!
- ▶ **Be professional and respectful, and use common sense!**
- ▶ Don't post offensive stuff on Teams!
- ▶ If you have problems, let me know!
- ▶ Use Teams if you struggle with course materials—we can all help!
- ▶ Do not share or distribute the course material without my permission

and now...

let's conquer OCaml!

Syntax & Semantics

- ▶ Syntax: how programs are written down
 - Roughly “spelling and grammar”
- ▶ Semantics: what programs actually *mean*
 - **Compile-time** (**static**) semantics: type checking
 - **Runtime** (**dynamic**) semantics: evaluation
- ▶ Both syntax and semantics matter
 - Semantics is the essence of a PL
 - Syntax is more about taste

Why two semantics?

- ▶ **Type checking** (**static** semantics) first makes a guarantee (proves invariant)
 - Happens before a program starts to run
- ▶ **Evaluation** (**dynamic** semantics) then runs the program
 - Because program type checks, many run-time errors are *impossible*!
 - Example:

```
1 + "hello"
```

- ▶ There is no dynamic semantics for this (non)-program in OCaml
- ▶ We don't care! Will never try to run a program that would evaluate such an expression
 - ▶ Unlike Python and its infamous `TypeError` exceptions...

Language constructs

- ▶ Today:

- Variable bindings
- Simple expressions

Variables

Variable bindings: syntax

► Syntax:

```
let x = e
```

where `let` is a **keyword**, `x` a **variable**, and `e` is an **expression**

- Many, many forms!
- Defined recursively!

```
(* static env  = ... *)  
(* dynamic env = ... *)
```

```
let x = 34
```

```
(* static env  = ...; x : int *)  
(* dynamic env = ...; x -> 34 *)
```

Note: metavariables (`x`, `e` etc.) are in blue with yellow-ish background.

Variable bindings: semantics

► Type checking:

```
let x = e
```

- Type check `e` with some type `t`
- Add binding `x: t` to the static environment

► Evaluation:

- Evaluate `e` to some value `v`
- Add binding `x ↦ v` to the dynamic environment

Structure

- ▶ A program is just a sequence of bindings
- ▶ Typecheck each binding in order
 - using **static environment** from previous bindings
- ▶ Evaluate each binding in order
 - using **dynamic environment** from previous bindings
- ▶ So far, only variable bindings
 - More kinds of bindings coming soon!

Expressions

Expressions

- Many kinds of **expressions** in OCaml:

```
34          true          xyz
e1 + e2     e1 < e2       if e1 then e2 else e3
```

- They are recursively defined: can be arbitrarily large
 - Expressions can have subexpressions that have sub-subexpressions that have ...

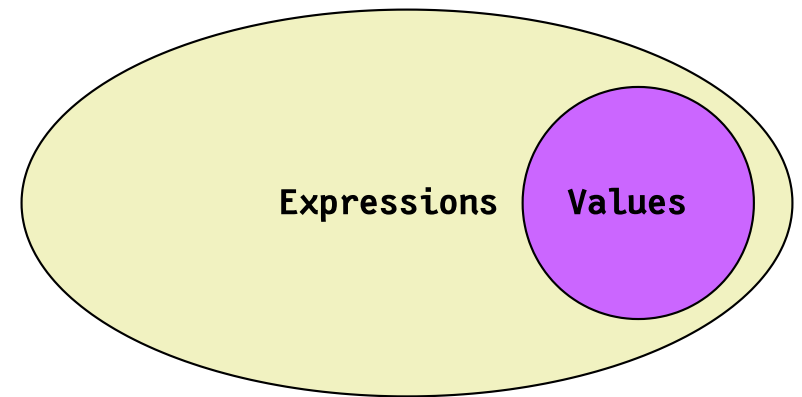
Expressions: syntax, typechecking, evaluation

- ▶ **Syntax**: how to write down that kind of expression (keywords, etc.)
- ▶ **Semantics**: what that kind of expression **means**
 - Type checking rules (**static** semantics)
 - ▶ Produces a type or reports an error
 - Evaluation rules (**dynamic** semantics)
 - ▶ Produces a value (or runs forever or raises an exception)
 - ▶ Only defined for well-typed expressions


This is a systematic approach to defining and understanding programming languages!

Values

- ▶ All **values** are *expressions*
 - But not all expressions are values!
- ▶ Values are *the answers* returned by evaluation
 - A value always evaluates to itself immediately
- ▶ Examples:
 - `12 27 13` are values of type `int`
 - `true false` are (the only) values of type `bool`



Variables: accessing

- ▶ **Syntax:** a sequence of letters, numbers, `_`, or `'` (e.g., `a`, `b0`, `_x`, `'t`)
 - : an identifier must start with a letter or `_`
- ▶ **Type checking:**
 - Look up in current static environment
 - ▶ if found: return corresponding type
 - ▶ else: fail
- ▶ **Evaluation:**
 - Look up in current dynamic environment
 - Return corresponding value

: Type checking ensures that the variable is always bound (cannot fail during evaluation)!

Expressions: addition

► **Syntax:** `e1 + e2` where `e1` and `e2` are expressions

► **Type checking:**

– If `e1` has type `int` and `e2` has type `int`

► then: `e1 + e2` has type `int`

► else: report error and fail

► **Evaluation:**

– If `e1` evaluates to value `v1` and `e2` evaluates to `v2`

► then: `e1 + e2` evaluates to the sum of `v1` and `v2`

Type checking ensures that `e1` and `e2` will always be ints!

Expressions: comparison (less than)

- ▶ **Syntax:** `e1 < e2` where `e1` and `e2` are expressions
- ▶ **Type checking:**
 - If `e1` has type `int` and `e2` has type `int`
 - ▶ then: `e1 < e2` has type `bool`
 - ▶ else: report error and fail
- ▶ **Evaluation:**
 - If `e1` evaluates to value `v1` and `e2` evaluates to value `v2`
 - ▶ then: `e1 < e2` evaluates to `true` if `v1` is less than `v2`; otherwise `false`

Expressions: conditionals (if-then-else)

► Syntax:

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

where `e1`, `e2`, and `e3` are expressions

► Type checking:

- If `e1` has type `bool` and `e2` and `e3` have the same type `t`
 - then: `if e1 then e2 else e3` has type `t`
 - else: report error and fail

⚠: Both branches must have the same type!

► Evaluation:

- If `e1` evaluates to `true`:
 - then: return result of evaluating `e2`
 - else: return result of evaluating `e3`

✍: Only one of the branches is evaluated!

The foundation we need

- ▶ Lots more types, expressions, and bindings
 - Interesting programs coming soon!
- ▶ Syntax, type checking, and evaluation will continue to guide us
 - For A1: functions, tuples, lists, options, local bindings
 - Earlier problems need fewer features
- ▶ **We will NOT need:**
 - Mutation (a.k.a. assignment): use **recursion** and/or **new variables**
 - Statements: use **expressions**
 - Loops: use **recursion**

Demonstration 1: Variable bindings and simple expressions

Demonstration 1: Variable bindings and simple expressions

```
$ utop
```

Demonstration 1: Variable bindings and simple expressions

```
$ utop
```

```
Welcome to utop version 2.16.0 (using OCaml version 5.4.0)!
```

```
Type #utop_help for help about using utop.
```

```
#
```

Demonstration 1: Variable bindings and simple expressions

```
$ utop
```

```
Welcome to utop version 2.16.0 (using OCaml version 5.4.0)!
```

```
Type #utop_help for help about using utop.
```

```
#
```

```
# let x = 34 ;;
```

Demonstration 1: Variable bindings and simple expressions

```
$ utop
```

```
Welcome to utop version 2.16.0 (using OCaml version 5.4.0)!
```

```
Type #utop_help for help about using utop.
```

```
#
```

```
# let x = 34 ;;
```

```
val x : int = 34
```

```
(* static env = ...; x : int *)
```

```
(* dynamic env = ...; x -> 34 *)
```

Demonstration 1: Variable bindings and simple expressions

```
$ utop
```

```
Welcome to utop version 2.16.0 (using OCaml version 5.4.0)!
```

```
Type #utop_help for help about using utop.
```

```
#
```

```
# let x = 34 ;;
```

```
val x : int = 34
```

```
(* static env = ...; x : int *)
```

```
(* dynamic env = ...; x -> 34 *)
```

```
# let y = 17 ;;
```

Demonstration 1: Variable bindings and simple expressions

```
$ utop
```

```
Welcome to utop version 2.16.0 (using OCaml version 5.4.0)!
```

```
Type #utop_help for help about using utop.
```

```
#
```

```
# let x = 34 ;;
```

```
val x : int = 34
```

```
(* static env = ...; x : int *)
```

```
(* dynamic env = ...; x -> 34 *)
```

```
# let y = 17 ;;
```

```
val y : int = 17
```

```
(* static env = ...; x : int; y : int *)
```

```
(* dynamic env = ...; x -> 34; y -> 17 *)
```

Demonstration 1: Variable bindings and simple expressions

```
$ utop
```

```
Welcome to utop version 2.16.0 (using OCaml version 5.4.0)!
```

```
Type #utop_help for help about using utop.
```

```
#
```

```
# let x = 34 ;;
```

```
val x : int = 34
```

```
(* static env = ...; x : int *)
```

```
(* dynamic env = ...; x -> 34 *)
```

```
# let y = 17 ;;
```

```
val y : int = 17
```

```
(* static env = ...; x : int; y : int *)
```

```
(* dynamic env = ...; x -> 34; y -> 17 *)
```

```
# let z = (x + y) + (y + 2) ;;
```

Demonstration 1: Variable bindings and simple expressions

```
$ utop
```

```
Welcome to utop version 2.16.0 (using OCaml version 5.4.0)!
```

```
Type #utop_help for help about using utop.
```

```
#
```

```
# let x = 34 ;;
```

```
val x : int = 34
```

```
(* static env = ...; x : int *)
```

```
(* dynamic env = ...; x -> 34 *)
```

```
# let y = 17 ;;
```

```
val y : int = 17
```

```
(* static env = ...; x : int; y : int *)
```

```
(* dynamic env = ...; x -> 34; y -> 17 *)
```

```
# let z = (x + y) + (y + 2) ;;
```

```
val z : int = 70
```

```
(* static env = ...; x : int; y : int; z : int *)
```

```
(* dynamic env = ...; x -> 34; y -> 17; z -> 70 *)
```

Demonstration 1: Variable bindings and simple expressions

```
$ utop
```

```
Welcome to utop version 2.16.0 (using OCaml version 5.4.0)!
```

```
Type #utop_help for help about using utop.
```

```
#
```

```
# let x = 34 ;;
```

```
val x : int = 34
```

```
(* static env = ...; x : int *)
```

```
(* dynamic env = ...; x -> 34 *)
```

```
# let y = 17 ;;
```

```
val y : int = 17
```

```
(* static env = ...; x : int; y : int *)
```

```
(* dynamic env = ...; x -> 34; y -> 17 *)
```

```
# let z = (x + y) + (y + 2) ;;
```

```
val z : int = 70
```

```
(* static env = ...; x : int; y : int; z : int *)
```

```
(* dynamic env = ...; x -> 34; y -> 17; z -> 70 *)
```

```
# let q = z + 1 ;;
```

Demonstration 1: Variable bindings and simple expressions

\$ utop

Welcome to utop version 2.16.0 (using OCaml version 5.4.0)!

Type #utop_help for help about using utop.

#

```
# let x = 34 ;;
```

```
val x : int = 34
```

```
(* static env = ...; x : int *)
```

```
(* dynamic env = ...; x -> 34 *)
```

```
# let y = 17 ;;
```

```
val y : int = 17
```

```
(* static env = ...; x : int; y : int *)
```

```
(* dynamic env = ...; x -> 34; y -> 17 *)
```

```
# let z = (x + y) + (y + 2) ;;
```

```
val z : int = 70
```

```
(* static env = ...; x : int; y : int; z : int *)
```

```
(* dynamic env = ...; x -> 34; y -> 17; z -> 70 *)
```

```
# let q = z + 1 ;;
```

```
val q : int = 71
```

```
(* static env = ...; x : int; y : int; z : int; q : int *)
```

```
(* dynamic env = ...; x -> 34; y -> 17; z -> 70; q -> 71 *)
```

Demonstration 1: Variable bindings and simple expressions

Demonstration 1: Variable bindings and simple expressions

```
# let abs_of_z =  
  if z < 0 then  
    -z  
  else  
    z ;;
```

Demonstration 1: Variable bindings and simple expressions

```
# let abs_of_z =  
  if z < 0 then  
    -z  
  else  
    z ;;  
val abs_of_z : int = 5  
(* static env = ...; x : int; y : int; z : int; q : int; abs_of_z : int *)  
(* dynamic env = ...; x -> 34; y -> 17; z -> 70; q -> 71; abs_of_z -> 70 *)
```

Demonstration 1: Variable bindings and simple expressions

```
# let abs_of_z =  
  if z < 0 then  
    -z  
  else  
    z ;;  
val abs_of_z : int = 5  
(* static env = ...; x : int; y : int; z : int; q : int; abs_of_z : int *)  
(* dynamic env = ...; x -> 34; y -> 17; z -> 70; q -> 71; abs_of_z -> 70 *)  
  
# (* Simpler way to do the same via Ocaml standard library: *)  
let abs_of_z_simpler = Int.abs z ;;
```

Demonstration 1: Variable bindings and simple expressions

```
# let abs_of_z =  
  if z < 0 then  
    -z  
  else  
    z ;;  
val abs_of_z : int = 5  
(* static env = ...; x : int; y : int; z : int; q : int; abs_of_z : int *)  
(* dynamic env = ...; x -> 34; y -> 17; z -> 70; q -> 71; abs_of_z -> 70 *)  
  
# (* Simpler way to do the same via Ocaml standard library: *)  
let abs_of_z_simpler = Int.abs z ;;  
val abs_of_z_simpler : int = 70
```

Coming up

- ▶ Semantics of expressions
- ▶ Make sure to get OCaml set and working!
 - Really! OCaml ecosystem is not as nice and user-friendly as Python, Java or C++ (I mean, I can complain endlessly about those as well if needed)

