# CSCI 2041: Advanced Programming (in this iteration: OCaml)

Sebastiaan Joosten

Text

September 6 2023

UNIVERSITY OF MINNESOTA
Driven to Discover®

## Advanced programming goals

- Make you a better programmer

## Beginner programming goals

- As a programmer, you:
  - Know some programming patterns:
    loops, if-then-else, reuse of methods
  - Think about a program as a series of steps
  - Can compare two programs and decide which is best

## Advanced programming goals

- To be better programmer, you'll learn:
  - New programming patterns
  - To think *what* to compute first,
    rather than the steps to take
  - To reason about what programs compute
  - To manipulate programs

## What is OCaml and why do we learn it?

- We’re learning ‘functional programming’: programming with function calls, avoiding a global state.
- OCaml makes it very easy to program with function calls.
- Some languages you know also allow that, but:
  - Most are *imperative:* think ‘sequence of commands’.
  - They make it easy to (accidentally) use global state.
- At the same time, it makes it hard to accidentally use (global) state.

## Example program: sum all numbers 1 through 100

```
let rec sumFrom s e (* sum from s(tart) to e(nd) *)
  = s + (if s = e then 0 else sumFrom (s+1) e)
let sumAll = sumFrom 1 100
```

- In this program we have:
  - expressions
  - definitions
  - recursion
  - integers (types)
  - comments
- What we don't have:
  - (re)assignment
  - state

## Quiz time!

## Quick look at the results
(hope this works)

## My quizzes

- Part of your grade instead of ‘lecture participation’.
- Don’t have to be hard or easy:
  - Points are awarded *only* for clicking ‘submit’ *during lecture*
- Talking through the questions and answers is allowed
- If you didn’t bring a device:
  - Come to me after class
    (I will give you a link to submit late)

# Using ocamlc and utop

- utop is a fancy calculator. We can:
  - Enter expressions and see their value
  - Add definitions (overriding old ones)
  - Add definitions from a file
- ocamlc is used to compile programs.
  - Pass filenames as arguments
  - Run "./a.out" as your compiled program
  - This requires a program that 'does' stuff, so we'll look at that later

## utop

- type ‘utop’ on the command line. If you get:
  - “Utop: command not found”. Congratulations: you’ve found the command line, but still need to either:
    - install ocaml and utop locally
    - log in to a CSE machine that has it installed
  - “Welcome to utop version 2.9.2 (using OCaml version 4.07.0)!” (or something similar). Congratulations, you can now enter expressions.
- Yesterday’s lab was supposed to get you to set this up…

## utop session

```
utop #
```

## utop session

```
utop # let rec sumFrom s e (* sum from
s(tart) to e(nd) *)
  = s + (if s = e then 0 else sumFrom
(s+1) e)
let sumAll = sumFrom 1 100
```

## utop session

```
utop # let rec sumFrom s e (* sum from
s(tart) to e(nd) *)
  = s + (if s = e then 0 else sumFrom
(s+1) e)
let sumAll = sumFrom 1 100
;;
```

## utop session

```
utop # let rec sumFrom s e (* sum from
s(tart) to e(nd) *)
  = s + (if s = e then 0 else sumFrom
(s+1) e)
let sumAll = sumFrom 1 100
;;
val sumFrom : int -> int -> int = <fun>
val sumAll : int = 5050
```

## dealing with errors:

```
utop # let rec sumFrom s e (* sum from
s(tart) to e(nd)
;;
;;
...?
```

## dealing with errors:

```
utop # let rec sumFrom s e (* sum from
s(tart) to e(nd)
;;
;;
...?*);;
Error: Syntax error
```

Syntax error at the ;;
means: whatever you started typing required something more

## dealing with errors:

```
utop # let rec sumFrom s e (* sum from
s(tart) to e(nd)
;;
;;
...?*);;
Error: Syntax error
```

Syntax error at the ;;
means: whatever you started typing required something more

## dealing with errors:

```
utop # *);;
Characters 0-2:
Warning 2: this is not the end of a
comment.
Error: Syntax error
```

other syntax errors typically are easier to understand

## dealing with errors:

```
utop # let rec sumFrom s e = if s then 0
else (sumFrom (s+1) e) + 1;;
Error: This expression has type bool but
an expression was expected of type int
```

type errors:

… check where the underline is for a hint

… check for other occurrences of the underlined word

## dealing with errors:

error is really here!

```
utop # let rec sumFrom s e = if s then 0
else (sumFrom (s+1) e) + 1;;
Error: This expression has type bool but
an expression was expected of type int
```

## Getting the error in the right place

add explicit type information

```
utop # let rec sumFrom (s : int) (e: int)
= if s then 0 else (sumFrom (s+1) e) + 1;;
Error: This expression has type int
       but an expression was expected of
       type bool because it is in the
       condition of an if-statement
```

## Forgetting the ‘rec’ keyword

```
utop # let sumFrom s e
   = s + (if s = e then 0
             else sumFrom (s+1) e)
let sumAll = sumFrom 1 100
;;
Error: Unbound value sumFrom
```

## Forgetting the 'rec' keyword (2)

```
utop # let sumFrom s e
  = s + (if s = e then 0 else sumFrom
(s+1) e)
let sumAll = sumFrom 1 100
;;
val sumFrom : int -> int -> int = <fun>
val sumAll : int = 5050
```

If you were typing along, this is what you'll see instead …

reason there's no error: 'sumFrom' refers to the old version of sumFrom

## Functions and function application

```
let rec sumFrom s e = s + (if s = e then 0 else
sumFrom (s+1) e)
```

- We read how 'sumFrom 1 100' works:
  - Replace s by 1 and e by 100 in the definition body:

```
1 + (if 1 = 100 then 0 else sumFrom (1+1) 100)
```

  - … then calculate this.
- The "1 + …" is actually another function application:
- We can try this in utop:
  (+) 1 5049

```
utop # (+) 1 5049;;
- : int = 5050
```

## Outline this week (i.e. Friday)

- More on function application
- we’ll look at programs that ‘do’ something:
  - Printing
  - Debugging
  - Using ocamlc