CMSC 430:
Introduction to Compilers

Administrivia

Spring 2026

Course logistics

- Lectures every TuTh 9:30am - 10:45am (except midterms)
- ~8 assignments (assignment 1 released)
- 2 midterms (take-home, 24 hour) **March 12, April 16**
- 1 final project (counts as final exam)
- Several surveys and quizzes (on ELMS)

- Course is very cumulative; building essentially one program all semester

Key resources

- Class web page (syllabus, assignments, course notes)
  - https://www.cs.umd.edu/class/spring2026/cmsc430/index.html
- ELMS (announcements, recordings, grades)
- Piazza (communication, discussion)
- Gradescope (assignments, exams)
- Office Hours

Lecture Format

- Lectures will use slides, live-coding, lecturing, and discussing
- Recordings of lectures posted after class
- Quizzes help reinforce lecture concepts

How to Study?

- Course notes are comprehensive and flesh out lecture material
- Best learned by doing
- Material is alive, interact with it
- Requires significant time outside of class
- Talk to your peers
- Spend time with TAs in off-peak hours
- Participate on Piazza (teaching others is a great way to learn)

Assignments

- Study the material before starting
- Think first
- Start early
- Submit often
- Approach problems systematically
- Writing lots of code is the wrong path
- Going slower will get you there faster

Midterm Exams

- 24-hour take-home exams
- Same advice as assignments
- Designed to be easily done in a couple hours without rush
- Will take more than 24 hours if you haven't already mastered material
- See practice midterms for best preview

Final Project

- Like an assignment, slightly larger, but also more time
- Designed to relieve stress at end of semester
- Due at end of final exam period for this class
- Final project does **not** provide graded feedback before deadline

Quick Overview

- We’re going to build a programming language
  - with modern features: higher-order functions, data-types & pattern matching, automatic memory management, memory safety, etc.
  - implemented via compilation: targeting an old, widely used machine-level language, x86, with a run-time system written in C
  - paying close attention to correctness: using interpreters as our notion of specification

Quick overview

- Source language: Racket (like OCaml w/o types, different syntax)
- Target language: x86
- Host language: Racket
- Final result: self-hosting compiler (compiles its own source code)

```racket
(define (fib n a b)
  (cond
    [(zero? n) a]
    [else (fib (sub1 n) b (+ a b))]))
```

```asm
mov   rbx, dword string1
mov   rcx, [rbx]
mov   r8, 0x01010101010101
add   rcx, r8
mov   [rbx], rcx
```

Source: Racket

Compiler: Racket

Target: X86

OCaml to Racket

- **Racket = OCaml - Types - Syntax**
- Download and install Racket
- Read and follow course notes chapter on “From OCaml to Racket”

- Today:
  - Racket basics, symbols, functions, pairs, lists, structures

Racket Code

- Racket code can take a bit to get used to reading, but its uniform structure makes it easy to learn

```
(print "Hello, World!")
```

How to Use it

- Install instructions
- Use Dr. Racket, the IDE made and supported by the Racket team
- Or develop everything in a text editor

A REPL. (or repl)

```
% racket
Welcome to Racket v8.12 [cs].
> (+ 1 2)
3
```

Arithmetic in OCaml

- In OCaml, arithmetic was pretty straightforward:

```
1 + 2 * 2;;
- : int = 5

((1)) + (2 * 2);;
- : int = 5
```

Arithmetic in OCaml

- In Racket, an open bracket “**(**“ means function application. This mean redundant brackets don’t mean what you think!

```
>(+ 1 (* 2 2))
5
```

Functions

- Anonymous functions were straightforward in OCaml

```
 fun x y -> x + y;;
- : int -> int -> int = <fun>

(fun x y -> x + y) 3 4;;
- : int = 7

(fun x y -> x + y) 3;;   (* Partial application! *)
- : int -> int = <fun>
```

Functions in Racket

OCaml: `fun x y -> x + y;;`

Racket: `(λ (x) (λ (y) (+ x y)))`

Quiz

What does this expression evaluate to?

```
(λ (x) (λ (y) (+ x y))) 3 4
```

A. 7

B. Error

C. Something else

Quiz

What does this expression evaluate to?

```
(λ (x) (λ (y) (+ x y))) 3 4
```

A. 7

B. Error

C. Something else

Definitions in OCaml

- Definitions in OCaml used **let**. This is true for functions, too

```
let x = 3;;
val x : int = 3

let mul a b = a * b;;
val mul : int -> int -> int = <fun>

mul x 4;; - : int = 12
```

Definitions in Racket

- In Racket we define things with **define**

```
(define x 3)
(define y 4)
(+ x y)
```

- Also true for functions

```
(define mul
   (λ (a b)
      (* a b)))
(mul 3 4)
```

Definitions in Racket

```
(define mul
   (λ (a b)
      (* a b)))
(mul 3 4)
```

There’s a shorthand for function definitions that lets us avoid the lambda

```
(define (mul a b)
  (* a b))
```

Lists and pairs in Racket vs OCaml

- OCaml lists:
  - `[] : ‘a list`
  - `(::) : ‘a -> ‘a list -> ‘a list`
  - `[1;2;3]` **convient notation for lists**
- OCaml pairs (and tuples):
  - `(,) : ‘a -> ‘b -> ‘a * ‘b`
- Pairs and lists: fundamentally different things

Racket Lists

- Racket lists:
  - `'()` : `'a list`
  - `cons : 'a -> 'a list -> 'a list`
    - `(cons 1 (cons 2 (cons 3 '())))`
  - `list` convenient function for lists
    - `(list 1 2 3)`
- Racket pairs (and tuples):
  - `cons : 'a -> 'b -> 'a * 'b`
- Pairs and lists: made out of the same stuff

Lists and pairs in Racket vs OCaml

- Every *list* is either the empty list or the cons of an element onto a *list*.

- Every *pair* is the cons of two values.

- (All non-empty lists are pairs, too)

- (Chains of pairs that don't end in the empty list are called "improper lists" and print with a ".")
  - `'(1 2 3 . 4) is (cons 1 (cons 2 (cons 3 4)))`

Quiz

- Is this a valid OCaml definition?

```
let xs = ["jazz"; 1959]
```

A) Yes
B) No
C) I don’t understand the question and I won’t respond to it.

Quiz

- Is this a valid OCaml definition?

```
let xs = ["jazz"; 1959];;
```

A) Yes
B) No. All elements of a list must be of the same type
C) I don't understand the question and I won't respond to it.

Racket Lists

- Racket is Dynamically typed, so the following is perfectly valid

```
(list "jazz" 1959)
```

Lists and pairs: Destructors in OCaml

- Pattern matching using constructors for empty, cons, and tuples: `[]`, `::`, `(_,_)`.
-
- `fst`, `snd` functions for pairs (2-tuples).

Lists and pairs: Destructors in Racket

- Pattern matching using constructors for empty, cons: `'()`, `cons`.
- `car`, `cdr` functions for pairs.

```
(car (list 1 2 3)) ==> 1
(cdr '(1 2 3)) ==> '(2 3)
```

- Do yourself a favor

```
(define fst car)
(define snd cdr)
```

- `first`, `rest` functions for lists.

Literal pairs and lists

- Lists of literals can be written using the quote notation:
  - `'()`
  - `'(1 2 3)`
  - `'(x y z)`
  - `'("x" "y" "z")`
  - `'((1) (2 3) (4))`
- Pairs of literals can be written using the quote notation:
  - `'(#t . #f)`
  - `'(7 . 8)`
  - `'(1 2 3 . #f)      ;(cons 1 (cons 2 (cons 3 #f)))`

Pattern Matching

- Just like in OCaml, we can pattern match to help us define functions

```
(define (swap p)
  (match p
    [(cons x y) (cons y x)]))
```

```
(swap (cons 10 20))
'(20 . 10)
```

Pattern Matching

```
(define (is-two-or-four n)
  (match n
    [2 #t]
    [4 #t]
    [_ #f]))
```

Pattern Matching

```
(define (sum lst)
    (match lst
     ['() 0]
     [(cons h t) (+ h (sum t))]))

(sum (list 1 2 3))
 6
```

Data Types

- One of the more elegant features of typed-functional PLs is algebraic datatypes. Defining and then pattern matching on ADTs is a very powerful tool for reasoning about programs

```
type tree = Leaf
             | Node of int * tree * tree
```

Structures

- Racket does not have ADTs directly, but we can get close with struct. It lets us define a structured value. i.e. like a single constructor from a datatype in
- Define new record types

```
(struct coord (x y z))
```

defines:

- coord : constructor, pattern
- coord-{x,y,z} : accessor functions
- coord? : predicate

Binary Tree

```
(struct leaf ())
(struct node (i left right))
```

Pattern matching on structs

- Defining a function that checks whether a tree is empty

```
(define (empty? bt)
  (match bt
   [(leaf) #t]
   [(node _ _ _) #f]))

(define lst
  (node 10 (leaf)(leaf))
(empty? lst) ===> #f
```

Pattern matching on structs

```
(define (sum bt)
  (match bt
    [(leaf) 0]
    [(node v l r) (+ v (sum l) (sum r))]))

(define lst
  (node 10 (node 20 (node 30 (leaf) (leaf)) (leaf))
  (leaf)))

(sum lst) ===> 60
```

Symbols

- **An atomic string-like datatype**
  - Symbols are a useful datatype for representing enumerations
  - `red `yellow `green
  - `up `down `left `right

- Symbols are literals, written with the quote notation (more later). Two symbols are equal if they are spelled the same.

Symbols

- In compilers we often need symbols that can’t clash with any existing symbols. We use gensym to generate "fresh names”

```
> (gensym)
```

Racket ` **quote**

- A quoted thing can always be represented as an unquoted thing by pushing the **'** `inwards' . ' "stop" at symbols (i.e. **'CS**) or empty brackets **'()**

```
'(x y z) == (list 'x 'y `z)
```

- ' goes away at booleans, strings, and numbers. So:

```
'3 == 3
`"String" == "String"
'#t == #t
```

Pairs and Quote

- `'(1 2)` means `(list '1 ‘2)`
- `'(1 . 2)` means pair