CMSC 430 - 1 June 2026

Introduction to Introduction to Compilers

Logistics

Overview of Compilers

Intro to Racket

Logistics

Learning objectives

Logistics

You should be able to answer:

- Where is the course web page?
- How do I find the syllabus?
- What is the basic structure of this course?
- How do I answer my own questions about the course?
- What strategies will help me succeed?

About me

David Van Horn (he/him)

You can call me: DVH or Professor Van Horn

Research area: Programming Languages

Background: UVM (B.S.), Brandeis (Ph.D.), Northeastern (Research Prof.)

Professor at UMD since 2014

Designed CMSC 430 and its materials

Contact: dvanhorn@cs.umd.edu

Compilers comes at you fast

Course logistics

Lectures every Weekday (except holidays, exams)

Assignments: 10, due Mon/Thur

Exams: 3 - 6/12, 6/26, 7/10

Several surveys and quizzes (on ELMS)

Course is very cumulative; building essentially one program all semester

Summer Compilers comes at you *faster*

Course logistics

Unrelenting pace

Plan to spend several hours a day on 430

Every two weekdays is a WEEK of material

First exam is **TWO WEEKS**

Semester at a glance

Course logistics

Week 1-2	Assignment 1-3	Exam 1
Week 3-4	Assignment 4-7	Exam 2
Week 5-6	Assignment 8-10	Exam 3

Grades

Course logistics

Assignments	30%
Quizes & surveys	20%
Exams	50%

Key resources

Course logistics

- Class web page (syllabus, assignments, course notes, slides)
  `https://www.cs.umd.edu/class/summer2026/cmsc430/`
- ELMS (announcements, recordings, grades)
- Piazza (communication, discussion)
- Gradescope (submissions)

Day to day: lectures

How to CMSC 430

- Lectures will use slides, live-coding, lecturing, and discussing
- Be here now
- Recordings of lectures posted after class
- Quizzes help reinforce lecture concepts due **before** next class

Day to day: studying
How to CMSC 430

- 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)

Day to day: assignments

How to CMSC 430

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

Day to day: exams
How to CMSC 430

- 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

* If you have already mastered the material!

Day to day: office hours

How to CMSC 430

- Use online forums (Piazza)
- Prof available 30 minutes before class for public discussion
- Email to set up private session
- TA office hours scheduled will be announced soon

Day to day: feedback

How to CMSC 430

- Feedback is welcome!
- Send email or Piazza post if OK with non-anonymous
- Anonymous feedback: https://forms.gle/99yTz7HVfopCaDMz9

Advice from future you

Quotes from Fall 2025 End of the Road Survey

- Start assignments early and read the spec carefully. Understanding the concepts first saves a lot of time later.
- Start projects as early as you possibly can, and do the practice midterms before the real midterms. Spend more time rereading and understanding the notes outside of class.
- Small misunderstandings can compound quickly, so staying ahead is much more important than trying to catch up later.
- Spending time to design effective tests is very important, and helps you understand how your compiler works, what might need to be changed, and potential edge cases.
- Step through the assembly generated by the compiler if you're stuck.
- Spend less time trying to do everything in your head and more time making memory diagrams, writing out your thought process, and manually tracing your code.
- Just because a portion of the compiler/interpreter is written for you doesn't mean that you shouldn't be reading them to better understand what YOU need to write.
- It's going to be a grind, but just keep sticking with it.

Compilers

Learning objectives

Compilers

You should be able to answer:

- What is a programming language?
- What makes a programming language:
  - general-purpose vs domain-specific?
  - high-level vs low-level?
  - statically-typed vs dynamically typed vs untyped?
- Why does programming language design matter?
- What is a compiler?
- What does it mean for a compiler to be correct?
- Why does compiler correctness matter?

What is a programming language?

No, really, I’m asking…

A set of instructions that the computer can understand to execute

Abstracted language that is eventually translated down to binary, higher level is closer to natural language

Let's you implement algorithms

What is a programming language?

A *programming language* is a formal language for **expressing computations**, with

- rules for what programs look like (**syntax**), and
- rules for what those programs mean or do when run (**semantics**).

What is a programming language?

A *general purpose* programming language is intended to express computations across many domains.

Examples: C, Java, Rust, OCaml, Racket, JavaScript, WASM

A *domain-specific* programming language targets a particular domain.

Examples: PostScript, SQL, Makefiles, Shader languages

What is a programming language?

A *high-level* programming language is designed to express computations without mentioning machine details.

Examples: Java, Racket, Haskell, OCaml, Python

Often have:

- Automatic memory management
- Strong safety guarantees
- Abstractions based in mathematics: functions, relations, types

What is a programming language?

A *low-level* programming language is designed to express computations while mentioning machine details.

Examples: C, Rust, Assembly, WASM

Often have:

- Explicit control over machine state
- Security vulnerabilities
- An imperative view of computations

What is a programming language?

A *statically-typed* programming language has syntactic rules in addition to grammatical rules for forming programs. These rules ensure good properties of programs before they are run.

Examples: Java, OCaml, Rust

Often have:

- Strong safety guarantees
- Classes of errors eliminated by the type system
- Good programs ruled out by the type system

What is a programming language?

A *dynamically-typed* programming language forgoes syntactic rules and instead imposes checks at run-time. These checks ensure good properties of programs when they are run.

Examples: Racket, Python, JavaScript

Often have:

- Strong safety guarantees
- Run-time type errors
- Overhead associated with run-time checking

What is a programming language?

None of these have precise meanings, they are cultural:

- dynamically-typed, statically-typed
- high-level, low-level
- general-purpose, domain-specific

Real languages occupy design points along a spectrum

What isn't a programming language?

- Musical notation: syntax and semantics, not computational
- Pseudocode: describes computations, but lacks formal syntax and semantics
- Markdown: syntax and semantics for document structure, not computational
- JSON: syntax and semantics for data values, but not computational

What is a programming language?

No, really, I'm asking…

Human readable way to create computer instructions

Combination of syntax and semantics for creating behavior

Has to be Turing Complete (maybe)

Way of adding abstractions to make coding faster

Toolbox that prioritizes certain methods of solving problems

What is a compiler?

No, really, I’m asking…

Translate from one programming language to another

Traditionally compiling to machine code

What is a compiler?

A compiler is a program that translates programs written in one language into programs written in another. A compiler is one possible implementation strategy for a language. Often compilers translate from higher-level languages to lower-level languages. Often compilers improve the efficiency of a program but doing work at compile time in order to avoid work at run-time. A compiler must preserve the meaning of the original program.

Languages are not compiled. Programs are compiled.

Design & implementation of programming languages

Quick overview

We’re going to build a programming language

- with modern features
- specified via interpreter
- implemented via compiler
- paying close attention to correctness

Design & implementation of programming languages

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, validated by informal reasoning and testing

Design & implementation of programming languages

Quick overview

We’re going to build a programming language

- 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

Learning objectives

Racket

You should be able to answer:

- What is the syntax of Racket?
- What is the computational model of Racket?
- What are the values of Racket?
- How is Racket similar to OCaml?
- What is the subset of Racket we will need for this course?
- What will we be using Racket for?

Ocaml to Racket

Racket = OCaml - Types - Syntax

Download and install Racket

Read and follow course notes chapter on “From OCaml to Racket”

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 (`eq?`) if they are spelled the same.

Lists and pairs in Racket vs OCaml

Constructors in OCaml

OCaml lists:

- `[]  : α list`
- `(::) : α → α list → α list`
- `[.;.;.;...]` convient notation for lists

OCaml pairs (and tuples):

- `(,) : α → β → α * β`

Pairs and lists: **fundamentally different things**

Lists and pairs in Racket vs OCaml

Constructors in Racket

Racket lists:

- `'()  : α list`
- `cons : α → α list → α list`
- `list` convenient function for lists

Racket pairs (and tuples):

- `cons : α → β → α * β`

Pairs and lists: **made out of the same stuff**

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 ".")

Lists and pairs in Racket vs OCaml

Destructors in OCaml

Pattern matching using constructors for empty, cons, and tuples: `[]`, `::`, `(_,_)`

`fst`, `snd` functions for pairs (2-tuples)

`hd`, `tl` functions for lists

Lists and pairs in Racket vs OCaml

Destructors in Racket

Pattern matching using constructors for empty, cons: `'()`, `cons`

`car`, `cdr` functions for pairs

`first`, `rest` functions for lists

Literal pairs and lists

A notation for writing down compound literals

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)`

Structures

Defining new record types

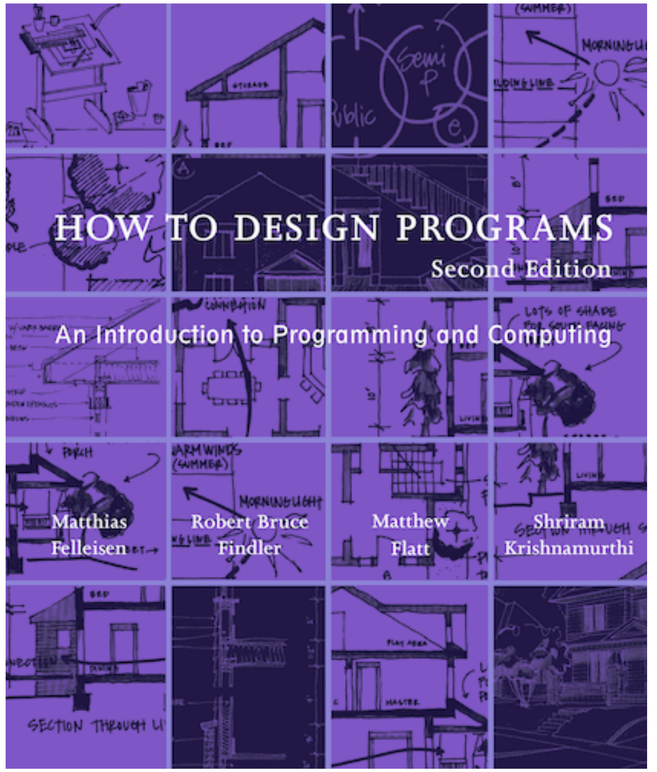
`(struct coord (x y z))` defines:

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

Systematic programming

Steps of systematic program design

1. From Problem Analysis to Data Definitions
2. Signature, Purpose Statement, Header
3. Examples
4. Template
5. Definition
6. Testing



Keys:

- write examples before code
- structure of functions follow structure of data
- live and die by the function signature

For next time

- Read syllabus
- Take ELMS survey & quiz
- Install Racket & langs package
- Read OCaml to Racket notes
- Read a86 notes