# Rust: The Basics

## Rust, in Practice and in Theory
## Lecture 2

CAS CS 392 (M1)

# Outline

» Go over the basics of Rust, emphasizing syntax

» Remind ourselves how to build parse trees

» **Workshop:** Build a guessing game

» **If you finish:** Programming Practice (Homework)

# Recap

# Recap: What's Rust?

# Recap: What's Rust?

`Rust is a type-safe memory-safe PL`

# Recap: What's Rust?

Rust is a type-safe memory-safe PL

It's possible to write simple clean code that's *guaranteed* to be free of memory bugs

# Recap: What's Rust?

Rust is a type-safe memory-safe PL

It's possible to write simple clean code that's *guaranteed* to be free of memory bugs

It's an alternative to C or C++ which can be used in production settings for rapid development without fear of crashes or memory leaks

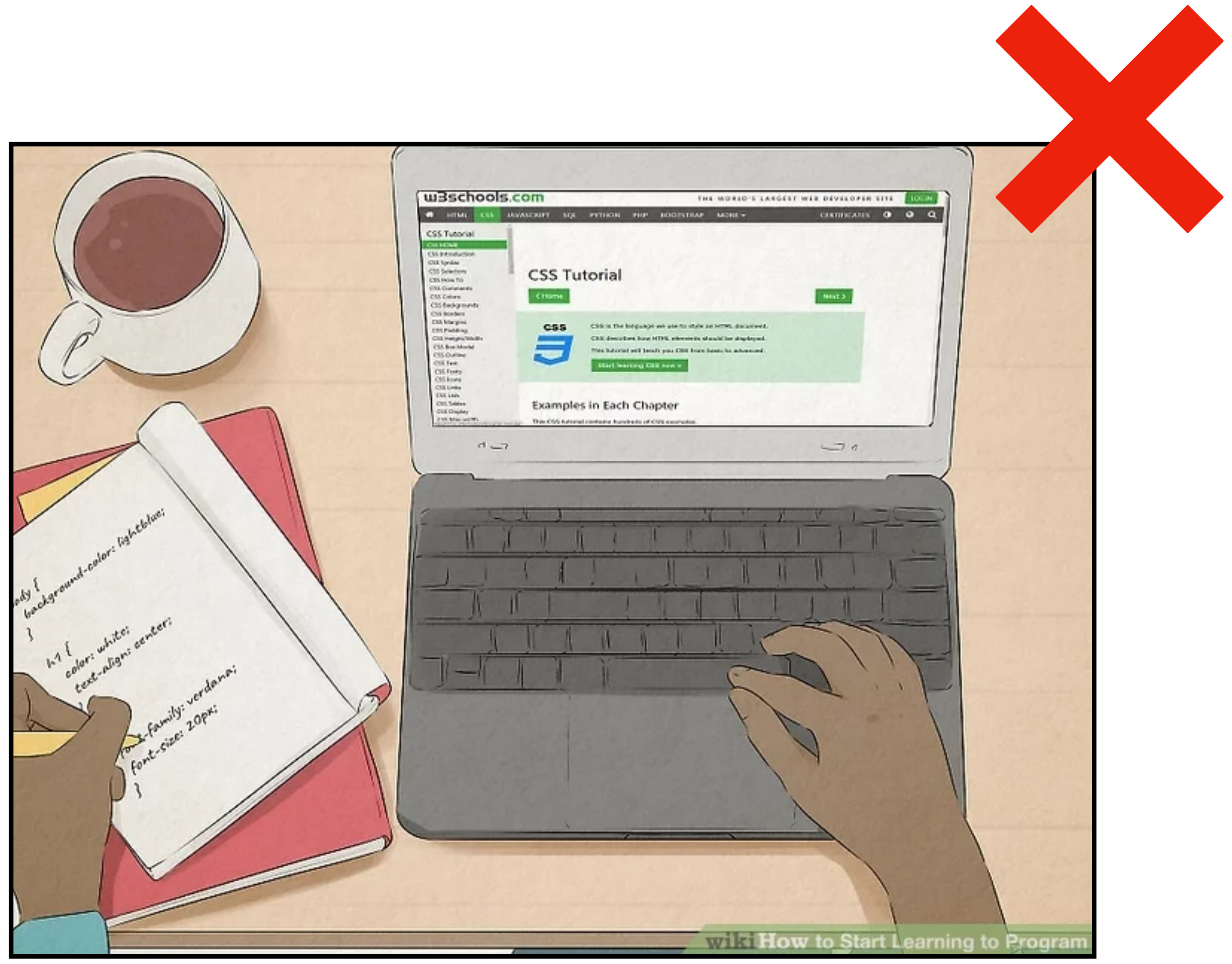
# Recap: How to learn a PL

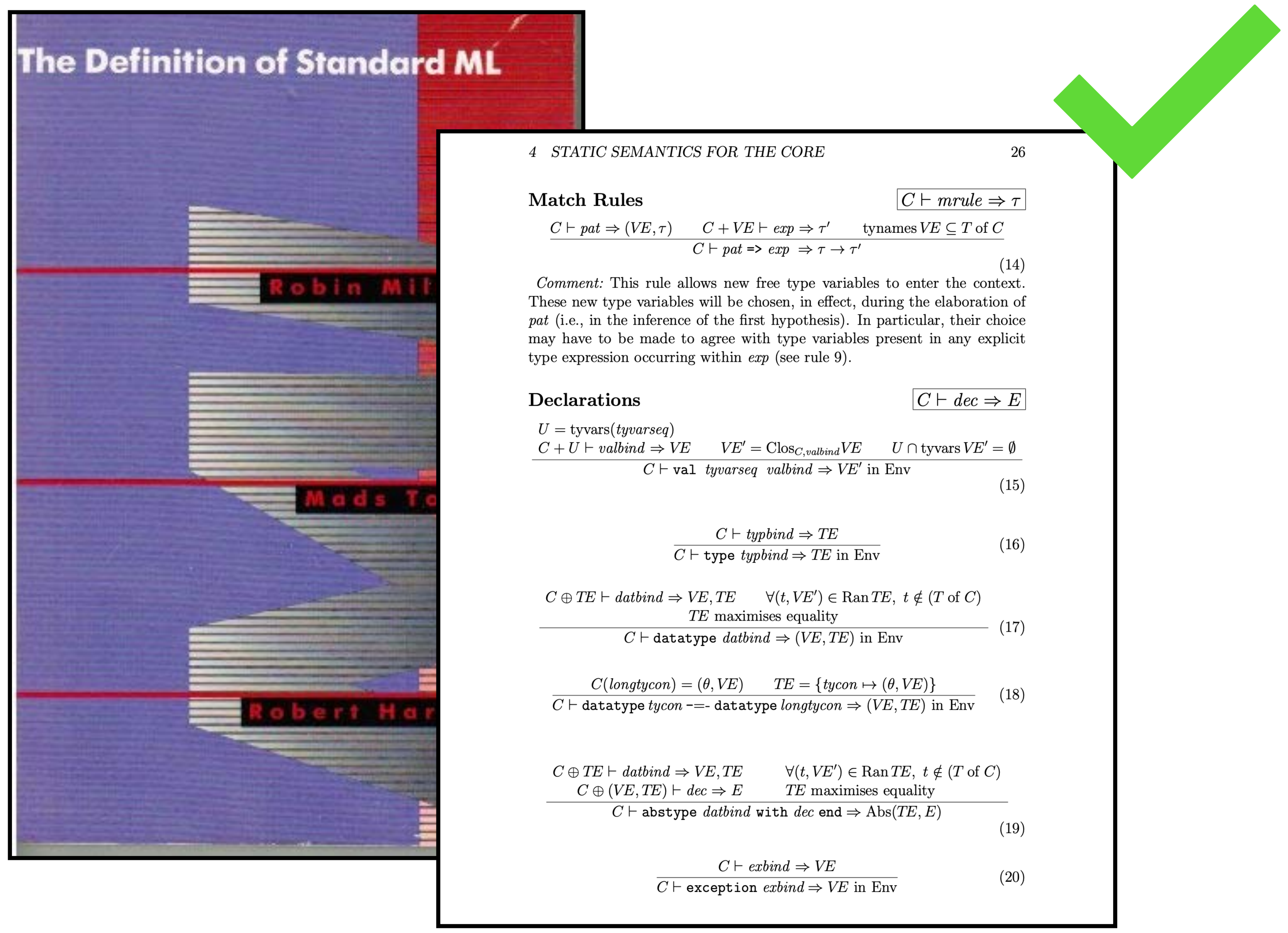
We tend to learn PLs the "wrong" (fast) way, i.e., reading tutorials and doing examples

In this course we want to learn Rust the "right" (slow) way, i.e., formally describe what Rust is doing

We *won't* learn many cool, advanced features of Rust that are useful in practice

We *will* learn why Rust makes us tackle with the type system in order to write safe code



# The Basics

# Rust as a basic PL

Rust has all the usual suspects for basic programs:

- variables and constants
- functions
- control-flow

# Variables and Constants

```
let x = 2;              // immutable variable
let x : i8 = 2;         // type annotated (immutable) variable
let rec x = 2;          // mutable variable
const X : i32 = 2;      // constant
```

Variable are immutable by default, and can be shadowed

Variables are written in **snake_case** by convention and constants in **SCREAMING_SNAKE_CASE**

# Variables (Grammar)

```
<var-decl>     ::= let <var-ident> = <expr>
                 | let mut <var-ident> = <expr>
<var-ident>    ::= ; snake_case ;

<const-decl>   ::= const <const-ident> : <ty> = <expr>
<const-ident>  ::= ; SCREAMING_SNAKE_CASE ;
```

We'll start, even now, thinking about how this syntax can be expressed as a BNF grammar

# A Quick Reminder: Parse Trees and Derivations

```
let var_name = 2 + 2
```

# Primitive Types

**Integers:** `i32` is the default

**Floats:** `f64` is the default

**Characters:** `char`, e.g., `'x'`

**Booleans:** `bool`, e.g., `true` and `false`

**Tuples:** e.g., `(i32, i32)`, where `p.i` is the accessor for the **i**th component

**Arrays:** e.g., `[bool; 5]`, arrays are **fixed-length** and `l[i]` is the accessor

**(with all the usual operators)**

# Data Types (Grammar)

```
<ty>          ::= <scalar-ty> | <compound-ty>
<scalar-ty>   ::= <int-ty> | <float-ty> | bool | char
<int-ty>      ::= <sint-ty> | <uint-ty>
<sint-ty>     ::= i16 | i32 | i64 | i128 | isize
<uint-ty>     ::= u16 | u32 | u64 | u128 | usize
<float-ty>    ::= f32 | f64
<compound-ty> ::= <tuple-ty> | <array-ty>
<tys>         ::= ϵ | <ty> | <ty> , <tys>
<tuple-ty>    ::= ( <tys> )
<array-ty>    ::= [<ty>; <expr>]
```

# A Quick Reminder: Parse Trees and Derivations

```
(i32, i32,)
```

# Literals (Grammar)

```
<int-lit>      ::= ; see docs ;
<float-lit>    ::= ; see docs ;
<char-lit>     ::= ; see docs ;
<bool-lit>     ::= true | false

<exprs>        ::= ϵ | <expr> | <expr> , <exprs>
<tuple-lit>    ::= ( <exprs> )
<field>        ::= ; see docs ;
<expr>         ::= <expr>.<field>

<list-lit>     ::= [ <exprs> ]
<expr>         ::= <expr> [ <expr> ]

<lit>          ::= <int-lit> | <float-lit>
                 | <char-lit> | <bool-lit>
                 | <tuple-lit> | <string-lit>
```

# Functions

```
fn sum_of_squares(x : u32, y : u32) -> u32 {
    let x_squared = x * x;
    let y_squared = y * y;
    x_squared + y_squared // NO SEMICOLON
}
```

Function definitions are standard. Parameters and output must be annotated

The body of a function is called a <u>block</u> which consists of a sequence of ;-separated statements

The last statement (if it is an expression) is the return value of function

# Functions (Grammar)

```
<fun-decl>   ::= fn <fun-ident>(<params>) <block>
               | fn <fun-ident>(<params>) -> <ty> <block>
<params>     ::= ϵ | <param> | <param> , <params>
<param>      ::= <var-ident> : <ty>
<block>      ::= { <stmts> }
<fun-ident>  ::= ; snake_case ;
```

# Statements (Grammar)

```
<stmts>       ::= ϵ
                | <expr>
                | <fun-decl> <stmts>
                | <stmt> ; <stmts>
                | <expr> ; <stmts>
                | <expr-no-sc> <stmts>
<stmt>        ::= <var-decl> | <const-decl>
<expr-no-sc>  ::= <if-expr> | <while-expr> | <for-expr>
```

# Control Flow

```
fn is_prime(n: i32) -> bool {
    for i in 2..n {
        if n % i == 0 {
            return false
        }
    }
    true
}
```

Control flow is standard

The most important thing to note is that control-flow is defined by ***expressions***

# Control Flow (Grammar)

```
<if-expr>         ::= if <expr> <block> <else-if-expr>
<else-if-expr> ::= ϵ | else <block> | else if <block> <else-if-expr>
<while-expr>      ::= while <expr> <block>
<for-expr>        ::= for <var-ident> in <expr> <block>
<ret-expr>        ::= return <expr>
<expr>            ::= <if-expr> | <while-expr> | <for-expr> | <ret-expr>
```

# A Quick Reminder: Parse Trees and Derivations

```
if true { 2 }
```

# Practice Problem (from Assignment 1)

*Write a function* ***is_perfect_cube*** *which determines if an* ***i32*** *is a perfect cube. Write it both in terms of simple control flow and in terms of type casting (this will require lookup in, say, Rust by Example)*

# Workshop:
# Programming a Guessing Game

# The Task

Work through the tutorial on building a **guessing game** in RPL. I'll walk around and answer questions. If you finish (or get bored) you can work on this week's **assignment** instead

**Note:** This is how I'll take attendance, so please make sure to talk to me before the end of lecture