WELCOME BACK TO (VIRTUAL) 538!
LOGISTICS

1. Mute your microphone
2. Click raise-hand to ask question
3. Ask questions on sli.do: #CS538
HW3 WRAPUP

- You implemented a lot of things:
  0. Syntax: from grammar to Haskell datatype
  1. Evaluator: from spec to code
  2. Parser: applicative/monadic parsing
  3. REPL: IO monad
- Ruse language
  - Toy version of Clojure/Scheme/Lisp
  - Lambda calculus with bells and whistles
  - Already quite powerful
FEEDBACK ON HW3?
HW4 OUT

- Four optional exercises
- Main piece: writing a RPN calculator
- WR4: Started material before break
  - Take a look at the notes in WR4
RUST’S RESULT TYPE

- Similar to Either
- Parametrized by type T and error type E

```rust
enum Result<T, E> {
    Ok(T),
    Err(E),
}

let all_ok = Ok("Everything ok!");
let error  = Err("Something went wrong!");
```
A FAMILIAR PATTERN

- Sequence error-prone computations
- Bail out as soon as we hit the first error

```rust
let res_1 = foo(x);
match res_1 {
    Err(e_1) => return Err(e_1);
    Ok(val_1) => {
        let res_2 = bar(val_1);
        match res_2 {
            Err(e_2) => return Err(e_2);
            Ok(val_2) => {
                let res_3 = baz(val_2);
                match res_3 { ... }
            }
        }
    }
}
```
PROPAGATING ERRORS

- Fixing error type, Result is a monad!
- No monads/do-notation in Rust, but: special syntax
- `?` unwraps value if Ok, or returns from function if Err

```rust
let val_1 = foo(x)?; // When foo returns a Result
let val_2 = bar(y?); // When y has type Result
```
MEMORY MANAGEMENT
PROGRAMS USE MEMORY

- Common across all programming languages
- During execution, a program may:
  - Request some amount of memory to use (*allocate*)
  - Return memory that it no longer needs (*free*)
  - System only hands out memory that is free
STACK ALLOCATION

- System keeps track of one address, the top of stack
  - Everything below top is allocated
  - Everything above top is free
- Last-in, first-out
  - To allocate: increase the top pointer
  - To deallocate: decrease the top pointer
STACK: BENEFITS

- Very fast
  - Allocating/deallocating is addition/subtraction
  - Lookups calculate offset of stack pointer
- Natural fit to block languages
  - When entering a block, allocate memory
  - When exiting a block, deallocate memory
  - Function calls/returns are similar
STACK: DRAWBACKS

- Allocation sizes must be fixed
  - Can’t grow/shrink previously allocated memory
  - Size of each allocation must be known statically
- Memory can’t persist past end of block
  - Memory allocated in function is freed on return
HEAP ALLOCATION

- Memory divided up into a bunch of small blocks
- System provides an allocator (e.g., malloc)
  - Keeps track of allocated/free blocks
- Programs request amount of memory from allocator
- Programs free memory by calling allocator
HEAP: BENEFITS

- **Flexibility**
  - Allocation sizes don’t need to be statically known
  - Can resize by allocating more and/or copying
- **Persistence**
  - Memory remains live until programs free it
  - Don’t have to free memory at end of blocks
HEAP: DRAWBACKS

- De-allocation is very easy to mess up
  - *Double free*: memory freed twice
  - *Use-after-free*: memory used after it was freed
  - *Memory leak*: program forgot to free memory
- Bugs are notoriously difficult to find
- Security holes, out of memory, crashes, etc.
WHO FREES HEAP MEMORY?
MANUAL MANAGEMENT

• Common in low-level programming languages

• Benefits
  ▪ Fastest, gives the programmer full control

• Drawbacks
  ▪ Programmers often mess up
  ▪ Bugs can be very hard to find
REFERENCE COUNTING

- Memory tracks how many things are pointing at it
- When count goes from one to zero, de-allocate
  - “Last one out, please turn off the lights”
- Benefits
  - Programmer doesn’t think about management
- Drawbacks
  - May leak memory if there are cycles
  - Need to constantly track counts for all allocations
  - Need to be sure the count is right
GARBAGE COLLECTOR (GC)

- System periodically sweeps through heap
  - Marks unreachable memory as free
  - Common in high-level programming languages
- Benefits
  - Programmer doesn’t think about management
  - Eliminate memory-management bugs
- Drawbacks
  - Slower, GC performance unpredictable
  - Maybe need a separate GC thread, pauses
THE STACK AND HEAP IN RUST
WHAT GOES ON THE STACK?

- Rough rule: anything with size
  1. known at compile time, AND
  2. fixed throughout execution
- Examples
  - Integers, pairs of integers, etc.
WHAT GOES ON THE HEAP?

• Rough rule: anything with size
  1. not known at compile time, OR
  2. varying throughout execution
• Examples: mutable datastructures
  ▪ Vectors, maps, mutable strings
TYPICALLY: A BIT OF BOTH

- On stack: constant size data
- On heap: variable size data
EXAMPLE: STRINGS

```rust
let s = String::from("hello");
```

- On stack: length (int), capacity (int), pointer to heap
- On heap: actual contents of string
THE OWNERSHIP MODEL IN RUST
BEST OF BOTH WORLDS

- Programmer follows certain *ownership* rules
  - Compiler knows where to insert de-allocation calls
  - Perfect memory management without GC
- However: programmer has to think a bit!
  - If rules are broken, the compiler complains
  - May need to add information to convince compiler
BASED ON C++ IDEA: RAII

- Resource Acquisition Is Initialization
- One of the worst names in the history of PL
  - Not really about acquisition
  - Not really about initialization
  - It is about resources
- Idea: when object goes out of scope, do cleanup
A POWERFUL IDEA

- Applies to many kinds of resources
  - Memory is not the only kind of resource!
- File handles and network sockets
  - Auto close when handle goes out of scope
- Locks and concurrency primitives
  - Auto unlock when value goes out of scope
OWNERSHIP PRINCIPLES

1. Each piece of data has a variable that is its owner.
2. Data can only have one owner at any time.
3. When owner goes out of scope, data is dropped.
String allocated on the heap and owned by variable `s`
Variable `s` goes out of scope at end of block
String is automatically de-allocated at end of block
MOVING, COPYING, CLONING
MOVING OWNERSHIP

• What happens when we assign a variable to another?

```rust
let x = String::from("foo");
let y = x;
```
DEPENDS ON THE TYPE!

- Default: ownership is *moved* from x to y
  - Before: x owns the string
  - After: y owns the string and x *does not*
- Shallow copy
  - Portion of data on the stack is copied
  - Portion of data on heap is *not* copied
  - Result: two things on stack pointing to same heap
ACCESSING DATA

- Remember: only one owner at a time
- Only the owner can read/modify the data

```rust
let x = String::from("foo");  // owner: x

let y = x;  // owner: y
println!("String: {}", y);  // OK
println!("String: {}", x);  // Not OK

let z = y;  // owner: z
println!("String: {}", y);  // Not OK
println!("String: {}", z);  // OK
```
**COPY INSTEAD OF MOVING?**

- For stack data: often easier to copy rather than move
- Controlled via the `Copy` trait
  - Assigning makes copy implicitly
  - Doesn’t invalidate previous variables

```rust
let x = 5;
let y = x; // automatically copied
println!("x = {}, y = {}", x, y); // x is still valid!
```
Sometimes, want to copy heap data too (*deep copy*).

Clone trait provides `.clone()` to do deep copy.

Explicit: not automatic (might be expensive).

```rust
let s = String::from("foo");
let t = s.clone();

// can use both s and t
println!("s = {}, t = {}", s, t);
```

Before: one string owned by `s`.

After: two separate strings, owned by `s` and `t`. 
SUMMARY

- Default: assignment moves ownership
- Copy: assignment copies data, no heap data
- Clone: make explicit copy by calling `clone()`
DROPPING
FREEING MEMORY

• When memory is no longer needed, return to system
  ■ Forget to return: memory leak!
• Would be nice: compiler inserts calls to free
• But how to know when to free?
  ■ Might depend on runtime behavior
DROPPING

• Idea: compiler knows where variable leaves scope
  ▪ This is known at compile time
  ▪ Automatically insert call to free memory here
• Data has exactly one owner
  ▪ Every data is freed once (and only once)

Result: avoid memory leaks in Rust
IN MORE DETAIL

- Compiler inserts calls to `mem::drop`
  - Can also call manually, if you want
  - Also known as a *destructor*
- Default behavior: data is dropped recursively
DROPPING STRUCTS

```rust
struct MyStruct1 { foo: MyStruct2, bar: String }
struct MyStruct2 { baz: String }
```

- Dropping a `MyStruct1`
  - Drop `foo`, then `bar`, then “wrapper”
- Dropping a `MyStruct2`
  - Drop `baz`, then “wrapper”
DROPPING ENUMS

enum MyEnum1 { foo(MyEnum2), bar(String) }
enum MyEnum2 { baz(String) }

• Dropping a MyEnum1
  ■ Drop foo OR bar, then “wrapper”

• Dropping a MyEnum2
  ■ Drop baz, then “wrapper”
CUSTOMIZING

- Run custom code when dropping
  - Print out stuff
  - Call other functions
  - Close file/connection
  - Change order things are dropped
DROP TRAIT

- Can customize the following method:

  ```rust
  fn drop(&mut self) { ... } // Note the type!!
  ```

- Does *not* take ownership of data
- Instead: takes `mutable reference` to data
  - Can mutate, replace, `Option::take()`, ...
- Data always freed when owner goes out of scope
  - No way to override (screw up) that part
DEMO
FUNCTIONS AND OWNERSHIP
PASSING AN ARGUMENT

- Function call moves ownership of arguments
- Think: new owner is argument variable in function
- When function ends, usual drop rules apply

```rust
fn main() {
    let old = String::from("foo"); // owner: old

    move_owner(old); // ownership moved

    println!("old is {}", old); // Not OK: old is not owner
}

fn move_owner(new: String) {
    println!("new is {}", new); // OK: new is owner
    ...
} // new out of scope, drops
```
RETURNING FROM FUNCTION

- Return values are similar: move ownership
- Think: new owner is variable holding return value
- If caller doesn’t store return value, it is dropped
AN ANNOYING PATTERN

- If caller wants to keep ownership of arguments, function must return arguments to return ownership

```rust
fn main() {
    let my_str = String::from("foo");  // owner: my_str

    let my_other_str = take_and_return(my_str);  // get ownership
}

fn take_and_return(a_str: String) -> String {  // owner: a_str
    // ... do some stuff ...

    // return ownership of a_str
    a_str
}
```
BORROWING A REFERENCE

- Make argument a reference
  - No need to return ownership after function
  - Other languages: “passing by reference”

```rust
fn main() {
    let my_str = String::from("foo"); // owner: my_str
    let my_ref = &my_str; // owner: still my_str
    borrow(my_ref); // owner: still my_str
}

fn borrow(a_ref: &String) { // owner: my_str
    // ... use reference a_ref ...
    // don't need to return ownership
}
MOVING OUT OF REF?

• Can’t move data from a borrow
  ▪ “Can’t move out of borrowed context”

```rust
fn borrow(a_ref: &mut String) {
    *a_ref = String::from("foo");    // OK: update a_ref

    let my_string: String = *a_ref;  // bad: can’t move String

    take_own(a_ref);                 // also bad!
}

fn take_own(a_str: String) { ... }
```