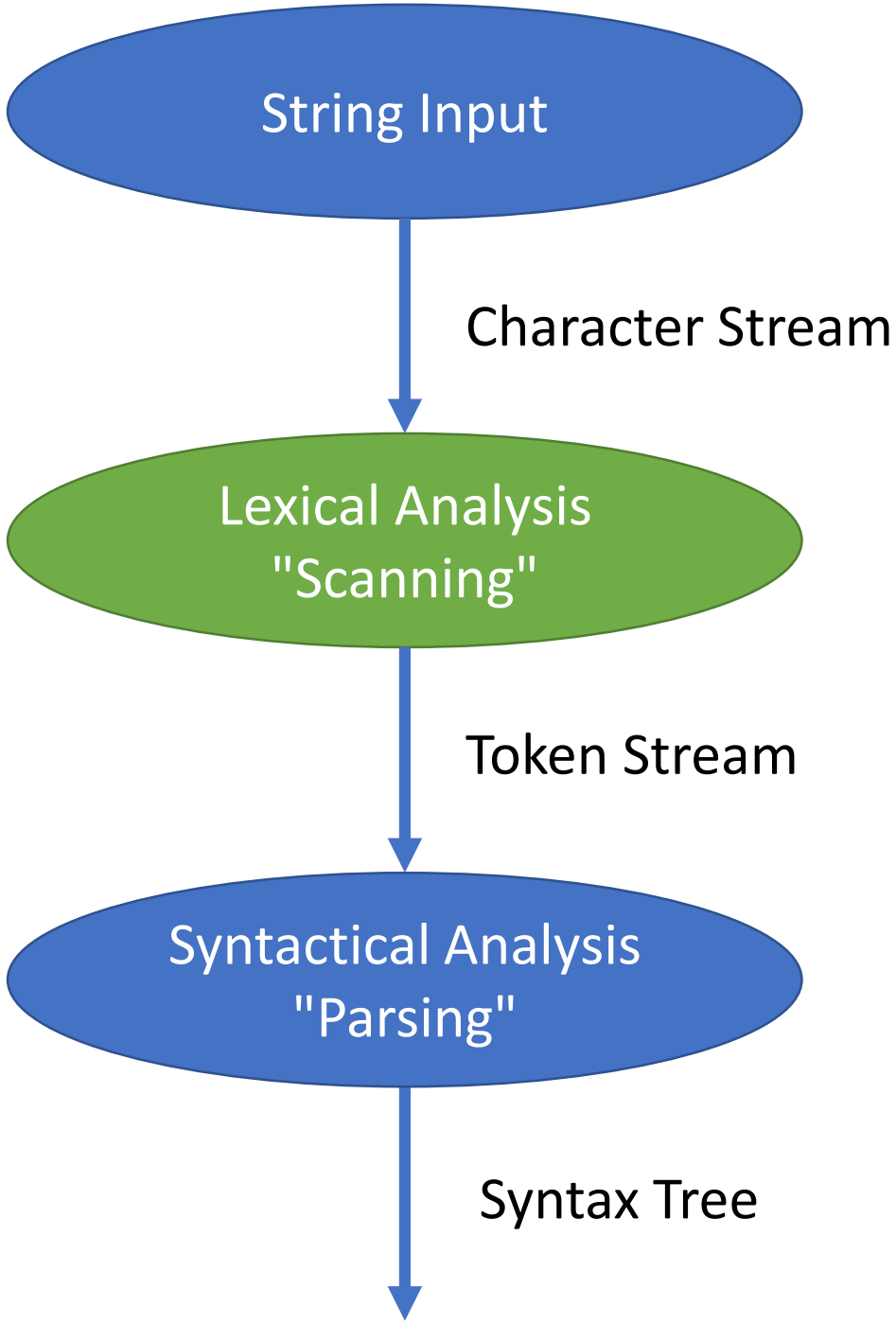
# little languages

## lecture 02:

# Options and Iterators

# The Road Ahead

- Last lecture we introduced a simple **grammar** for **vim**'s command language.
- When we parsed inputs from the grammar, each parsing resulted in a **syntactical tree structure** that can be represented digitally as a data structure.
- The algorithmic process any little language processor must take to transform an input string into a syntax tree is a multi-stage process shown to the right.
- The **first step toward lexical analysis is processing a character stream**. Today's focus on character iteration is the first step along the way. Soon you'll address scanning.



# Warm-up question: **pollev.com/compunc**

```
class Main {
    public static void main(String[] args) {
        // How would you call jumpOutOfPlane from here?
    }

    private static void jumpOutOfPlane(Parachute p) {
        System.out.println(p.getColor());
    }
}

class Parachute {
    private String color;

    Parachute(String color) {
        this.color = color;
    }

    String getColor() {
        return this.color;
    }
}
```

# On `null`, a source of many errors.

"I call it my billion-dollar mistake. It was the invention of the null reference in 1965.

At that time, I was designing the first comprehensive type system for references in an object oriented language (ALGOL W).

My goal was to ensure that all use of references should be absolutely safe, with checking performed automatically by the compiler.

But I couldn't resist the temptation to put in a null reference, simply because it was so easy to implement.

This has led to innumerable errors, vulnerabilities, and system crashes, which have probably caused a billion dollars of pain and damage in the last forty years."

-Tony Hoare

(Also invented Quicksort in 1959.)

# Consider `null` in Java...

- Conceptually, isn't it a *little* bit *insane* **`null`** is a valid value for *any* reference type?

  You: "Ok, so I've got a method named `jumpOutOfPlane` and it requires a single object parameter of type `Parachute` to be called. How do you call it?"

  Friend: "Easy. `jumpOutOfPlane(null)`"

  You: "Uhh.. no.. what.."
  Java: "YEP, that checks out! Here we goooo..."

- `NullPointerExceptions` only occur at runtime!
  - How do you prevent `NullPointerExceptions` in a method like `jumpOutOfPlane`?

# Checking for `null` in Java

- To protect a codebase from `nullPointerExceptions` you wind up having three options:

1. Traditionally you have to guard accessed references with null checks.

```
// This won't save you from a race condition in multithreaded code, though...
if (p == null) {
    System.out.println("Why did it come to this...");
} else {
    System.out.println(p.getColor());
}
```

2. Modern code bases will use Java 8's or Guava's `Optional<T>` type everywhere a null reference *may* be returned.
   - Instead of ever assigning or returning `null`, the value `Optional.empty()` is returned which is an object.
   - When you call a method that returns an `Optional<T>` you must still check a value is present. There's no escaping code to handle the two scenarios, but it is more explicit when an API is designed to *optionally* return a value.

3. Some code bases will use Java 8's `@NonNull` attribute and compiler option that does more to enforce strict `null` checking

# Rust has no `null` values*!

- In order to make *null* impossible, it means Rust code looks a lot like what very well engineered Java or C++ winds up looking like...

  *... and is strongly enforced by the compiler!*

- Rust's **`Option<T>`** enumeration type is used anywhere you may have:
  - **`Some`** actual value of type **`T`** or
  - **`None`** at all

*Aside: One of the motivating reasons for choosing Rust for this course is if you apply the lessons you learn in Rust to the work you do in other languages like Java, then you'll write much better, safer code elsewhere, too.*

* In the default, safe mode, which is where we'll be all semester. In unsafe Rust, there is null.

# Here's a definition of `Option<T>`

```
enum Option<T> {
    None,
    Some(T)
}
```

- This defines an enumerated type (enum) `Option` that is generic for any type T
- `Option<T>` has two variants: `None` or `Some(T)`
  - This means every value of type `Option<T>` must be *either* `Some(T)` or `None`
- In Rust, each variant of an enum can store data associated with it!
  - Some(T) is expressing the Some variant can hold a single value of type T
  - You will learn more about enums in Chapter 10

# Examples of **`Option<T>`** values

```
let a: Option<char> = Option::Some('a');
let b: Option<char> = Option::None;
```

- Consider the two variables above.
  - Both are of type `Option<char>` which you can read as:

  "This variable a's type tells us its value is either Some character or None at all."

- To construct `Option<T>` with an *actual* value you wrap it in a `Some(T)`

- The code above is explicitly typed. Notice `Some` and `None` are namespaced within the `Option` enum.

# Implicitly typed examples of **`Option<T>`**

```
let a = Some('a');
let b = None;
```

- The previous code is exactly equivalent and just as type safe as above.

- Using the `Option<T>`, `Some`, and `None` types are *so* common in Rust they're imported automagically as part of the standard prelude.
  - The same is true in Java. There's an implicit **`import java.lang.*`** in every file. (i.e. `java.lang.Math allows you to write Math.pow(2) anywhere in Java`)

- In *most* cases Rust can deduce a variable's type to a single type by looking at its usage
  - In cases where it cannot due to ambiguities, you will have to explicitly type your variables
  - The type inference capabilities of modern languages like Rust, Swift, C#, TypeScript, and Kotlin are descendent of Hindley and Milner's 1973 method featured in the programming language ML

# PollEv.com/compunc - What is the output?

```
let a: Option<char> = Some('a');
let b: Option<char> = None;

if let Some(c) = a {
    println!("{}", c);
}

if let Some(c) = b {
    println!("{}", c);
} else {
    println!("None");
}

if let None = b {
    println!("None");
}
```

# Aside: Destructuring Assignment in Python and (Java|Type)Script

Swap in Python:

```
a = 1
b = 2
a, b = b, a
print(a) // 2
```

Swap in modern JavaScript:

```
let a = 1;
let b = 2;
[a, b] = [b, a];
console.log(a); // 2
```

- If you can structure variable names on the LEFT hand side of the assignment operator in a structure that is aligned with the RIGHT hand side's structure...
- Then, values *from* the right hand side will be assigned to the corresponding variables on the left hand side.
- Destructuring assignment is even more capable than these examples
  - You can destructure assign object properties!

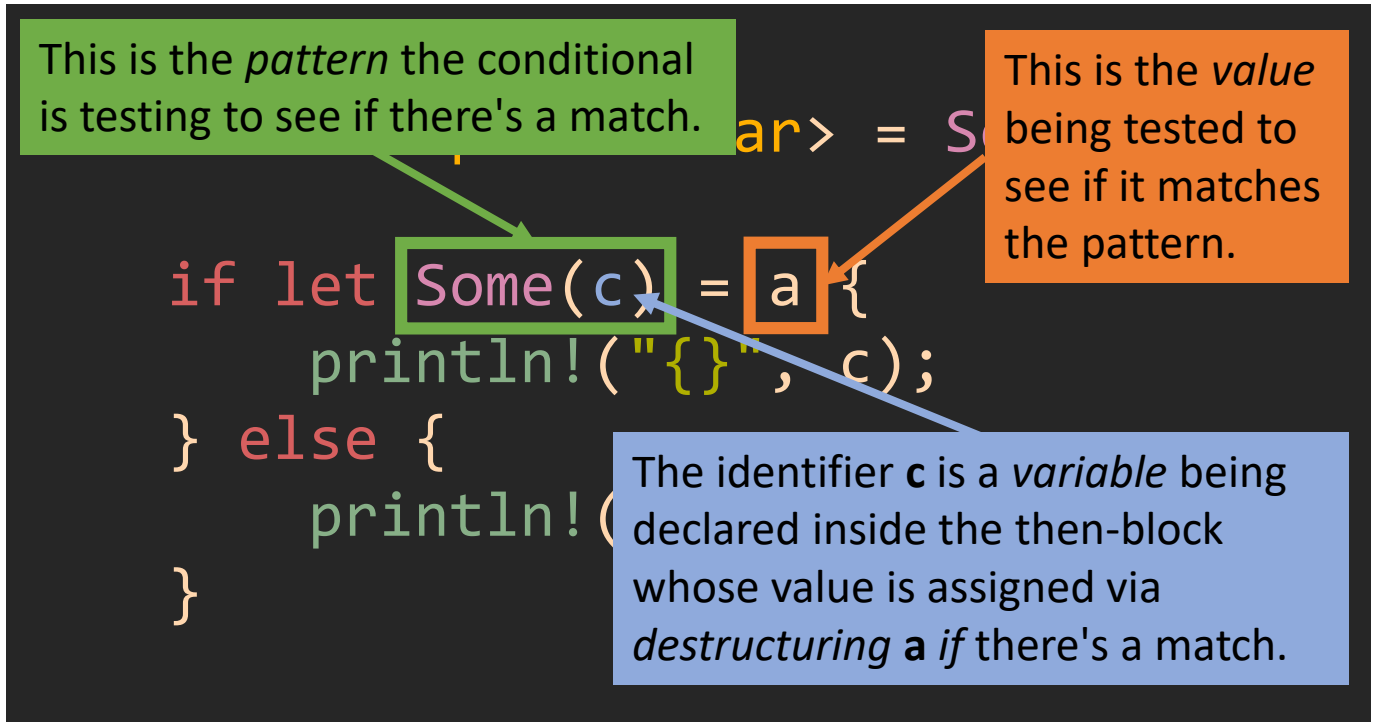
# Destructuring Assignment with `if let` (1/4)

- How do you access the data stored in a `Some<T>` value?
- You must do some kind of conditional check (`if-let`, `while-let`, or `match`) to ensure you actually have Some value.
- Since any time you're doing the work of checking for a particular kind of enum variant, you probably *also* want the value stored, Rust combines both steps via conditional destructured assignment.

```
let a: Option<char> = Some('a');

if let Some(c) = a {
    println!("{}", c);
} else {
    println!("None");
}
```

# Destructuring Assignment with `if let` (2/4)




"**If** the following **let** statement is a valid *destructuring assignment*:

```
let Some(c) = a
```

Then, carry out that destructuring assignment and continue into the then block.

Otherwise, jump to the else block."

# Destructuring Assignment with **`if let`** (3/4)

```
let a: Option<char> = Some('a');

if let Some(c) = Some('a') {
    println!("{}", c);
} else {
    println!("None");
}
```

Imagine substituting **a** in the `if-let` with its actual value as if we were interpreting the code...

**`let Some(c) = Some('a')`**

Is this a valid destructuring assignment?

Yes! So, *then*, **`'a'`** is assigned to the variable **c** in the then block via destructuring.

# Destructuring Assignment with **`if let`** (4/4)

```
let a: Option<char> = None;

if let Some(c) = a {
    println!("{}", c);
} else {
    println!("None");
}
```

Alternatively, **`a`**'s value could be **`None`**.

In which case, if you substitute **`a`** in the **`if-let`**, your let statement looks like:

```
let Some(c) = None
```

Is this a valid destructuring assignment? No! The structure on both sides of the assignment statement must correspond with one another.

So execution jumps to the **`else`** block.

# No Googling! PollEv.com/compunc

## Explain an iterator in ONE sentence!

- Speed round, you have exactly 2 minutes to respond.
- No Googling! Chat with your neighbors.
- GO!

# Iterating through a **`str`**'s Characters

- Iterators are Rust's preferred idiom for collection traversals
- Rust Iterators have a method named `next`
  - It returns an `Option<T>` where `T` is the type of `Item` the collection contains
- The str type's **`chars()`** method produces an iterator of **`chars`**
  - Thus, calling **`next()`** on it produces values of type **`Option<char>`**
- Since an iterator's state is mutated every time you call **`next()`**, iterators must be declared as **`mut`**able variables.

# Manually iterating through a **`str`**'s **`char`**s

- Convince yourself the output of the code listing to the right is:

```
a: a
b: b
no c!
```

- Do you notice the pattern here? You want to keep taking Some character until there's None left.

```
// Declare a string variable
let a_str = "ab";

// Establish an iterator
let mut itr = a_str.chars();

if let Some(c) = itr.next() {
    println!("a: {}", c);
}

if let Some(c) = itr.next() {
    println!("b: {}", c);
}

if let Some(c) = itr.next() {
    println!("c: {}", c);
} else {
    println!("no c!");
}
```

# Introducing `while let` iteration

```
// Declare a string variable
let a_str = "abcdefg";

// Establish an iterator
let mut itr = a_str.chars();

// Iterate through its values
while let Some(c) = itr.next() {
    println!("{}", c);
}
```

- Using a `while let` instead of an `if let` performs the same logical steps as before

- Like any other `while` statement, though, when the end of the block is encountered the conditional is checked again

# Preview:
# The **`match`** Statement

- In C, Java, JavaScript, etc. you know the **`switch`** statement.
- In Rust you'll use **`match`**
- The example right is the most basic form of **`match`**
  - It has 2 pattern matching "arms" trying to match c against the character 'a' and the character 'e
  - It has a default arm denoted with the _ for any case where no patterns matched.
  - Unless your pattern matching arms are *exhaustive* of every possible pattern, you must have a default arm.

```
let a_str = "abcde";
let mut itr = a_str.chars();

while let Some(c) = itr.next() {
    match c {
        'a' => {
            println!("a");
        }
        'e' => {
            println!("e");
        }
        _ => {
            println!("not a nor e");
        }
    }
}
```