write a 'parser' to run through every character

How do we do this?

Modules in rust.

→ Calc1.rs

new file in <u>src</u>

Calc1

calls function run from another rs file in the directory

Calc1::run() // in main

or <u>use calc1::run()</u> // before main

'module' | we write using this

mod calc1
Mod calc2 ⎬ tells compiler we have other file and we can use it.

std:: thing :: whatever()

calc1 :: run() $\geq$
Calc2 :: run()

<u>Calc 2 file</u>

calculate with any number of args

(ignoring prescedence)

fn run()

enum: data structure // multiple variance

enum Token{ // we Have token
// could be many things

Num(f64),

Add,
Sub

Functions

```
fn lex(input: Vec<&str>)
            ↑           ↖ type
       Identifyer
```

Not Like C

```
int main {
    return 1;
}
```

```
Fn main() -> i32 {
    1
}
```

Ok() works on Enums

Lex is not expected to return an error

4

lex → returns ~~Tokens~~ // multiple Tokens

To return multiple Tokens

Vec<Token>

↑

array of Tokens

```
fn lex(input: Vec<&str>) -> Vec<Token> {
    input.into_iter().fold(Vec::new(),
        |mut acc, e| {
```

↙ unnamed function

```
/* fold takes a closure
    let x = |a| a + 10;
    println!("{}", x(10));

fn x(a: i32) -> i32 {
    a + 10
}
```

equivalent

.fold() // what does it do?

0..100 creates iterator 0-99

↖ specifics # of elements

```
(0..100).for_each(|c| println!("{}", c));
```

for () variables, conditions of breaks

```
for number in 0..100 {
    println!("{}", number);
}
```

Same!

"clojure"

```
let result =
(0..100).fold(0, | mut acc, n| {

    ~~acc+n;~~ acc += n;

    acc
    });
println!("{}", result);
```

0+0+1+2+3

integer is interpreted by default
let result == let result : i32
all #'s from 0 to 99 added

from C,

```
let mut sum=0;
for n in 0..100 {
    sum+=n;
}
println!("{}", sum);
```

cntrl /   to comment out

~~Highted~~ highlighted code
(~~O~~nly VS code, not Vim)

```
let mut sum = 0;
  for n in 0..100{
      sim +=n;
  }
   println!("{}", sum )
```

.push(Token :: Add) , adds new ~~vector~~ element to Token

```
fn calculate() {   // To parse
        tokens: Vec<Token
```

```
Struct Parser {
    total: f64,
    last: Option<Token>,

}
```

```
#[derive(Default)]
struct Parser                              // These are fields
    total: f64
    last: Option<Token>,
```

This value is optional

Parsar::default
← creates a defined Struct
w/ default values.

Vec is a standard Struct

PascalCase ← structs, enums
traits

snake_case

fn calculate

tokens.into_iter().fold(Parser::default(),

|mut acc, t| {

match t {

Token::~~Token~~
Num(f) if
acc.last.is_none()
=> acc.total = f,

.unwrap() // unwraps to give token (the inner value)

.as_ref().unwrap()

takes ownership of the value unwrapped → compiler will not like it.

so we take the value as reference

$$\begin{tikzcd}
 & 0+2+2 \\
\text{Bugs} \quad \underline{+2\,2} \arrow[ur] \arrow[dr] & \text{this is valid} \\
\text{returns } 4 & \text{The + is kept in memory}
\end{tikzcd}$$

2 2

2 + 2 2 2 2