A Gentle Introduction to Parsers

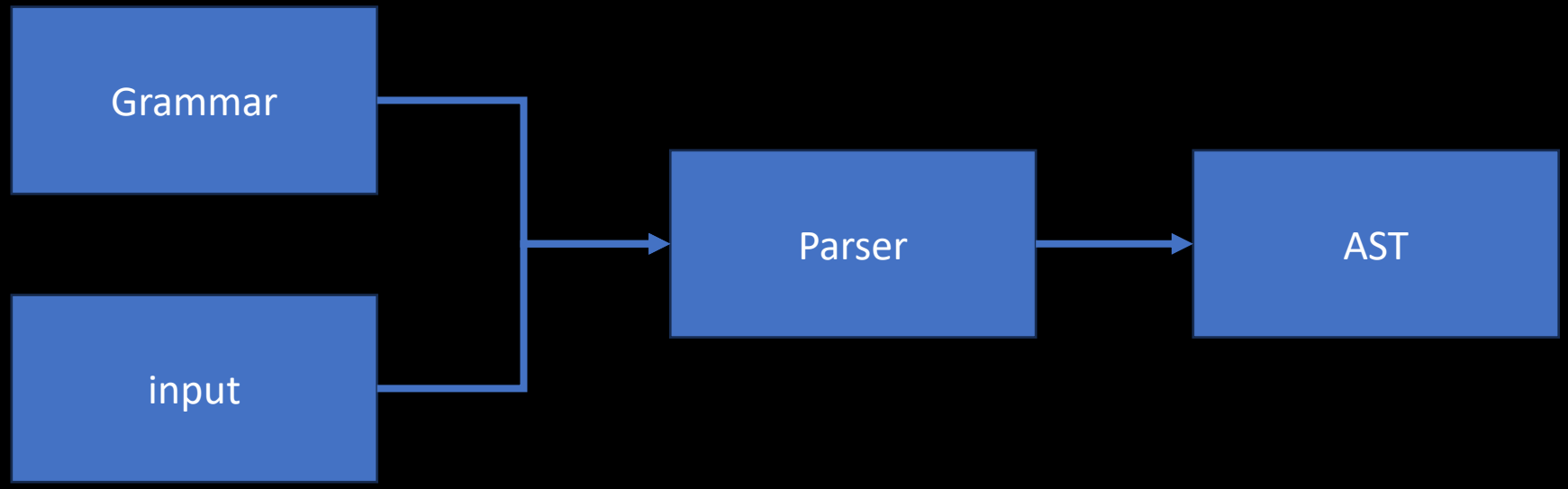
Parsing?

The process of analyzing a string of symbols, either in natural language, computer languages or data structures, conforming to the rules of a formal grammar.

https://en.wikipedia.org/wiki/Parsing

Parsing?

Given an input, and a set of grammar, produce an abstract syntax tree



Parsing?

Given an input, and a set of grammar, produce an abstract syntax tree

```
(a + b) * (c * d / e + f)
```

Parsing?

Given an input, and a set of grammar, produce an abstract syntax tree

```
(a + b) * (c * d / e + f)
```

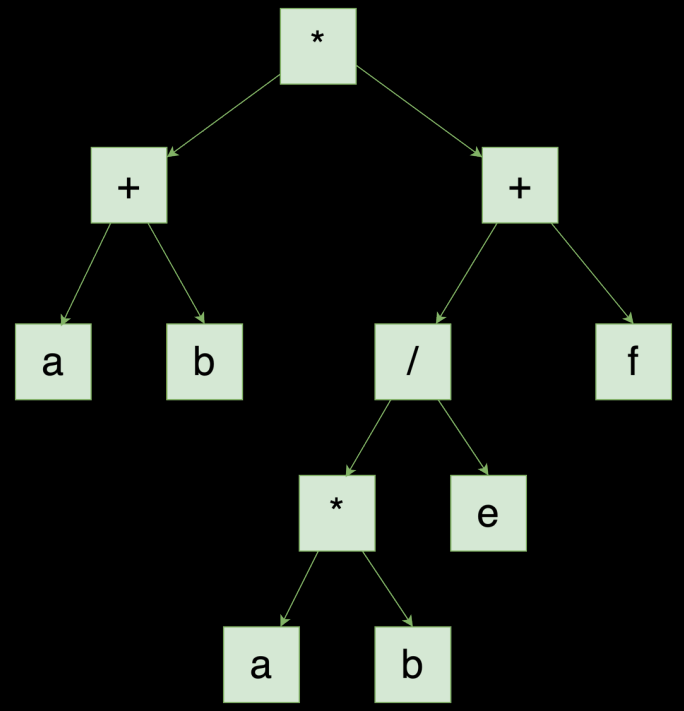
```
variable = {ASCII_ALPHA_LOWER }
atom = _{ ASCII_DIGIT | variable | "(" ~ expr ~ ")" }
bin_op = _{ add | subtract | multiply | divide | modulo }
        add = { "+" }
        subtract = { "-" }
        multiply = { "*" }
        divide = { "/" }
        modulo = { "%" }
expr = { atom ~ (bin_op ~ atom)* } equation = _{ SOI ~ expr ~ EOI }
```

Parsing?

Given an input, and a set of grammar, produce an abstract syntax tree

```
(a + b) * (c * d / e + f)
```

```
variable = {ASCII_ALPHA_LOWER }
atom = _{ ASCII_DIGIT | variable | "(" ~ expr ~ ")" }
bin_op = _{ add | subtract | multiply | divide | modulo }
        add = { "+" }
        subtract = { "-" }
        multiply = { "*" }
        divide = { "/" }
        modulo = { "%" }
expr = { atom ~ (bin_op ~ atom)* } equation = _{ SOI ~ expr ~ EOI }
```



Parsing?

Given an input, and a set of grammar, produce an abstract syntax tree

```
(a + b) * (c * d / e + f)
```

```
variable = {ASCII_ALPHA_LOWER }
atom = _{ ASCII_DIGIT | variable | "(" ~ expr ~ ")" }
bin_op = _{ add | subtract | multiply | divide | modulo }
        add = { "+" }
        subtract = { "-" }
        multiply = { "*" }
        divide = { "/" }
        modulo = { "%" }
expr = { atom ~ (bin_op ~ atom)* } equation = _{ SOI ~ expr ~ EOI }
```

```
- expr
  - expr
    - variable: "a"
    - add: "+"
    - variable: "b"
  - multiply: "*"
  - expr
    - variable: "c"
    - multiply: "*"
    - variable: "d"
    - divide: "/"
    - variable: "e"
    - add: "+"
    - variable: "f"
- EOI: ""
```

Motivation

Computers are stupid.

“`(a + b) * (c * d / e + f)`” is too difficult to process

Motivation

Computers are stupid.

“`(a + b) * (c * d / e + f)`” is too difficult to process

1. Is the expression correct?
2. What does “a”, ”b”mean? What does ”+” “/” mean?
3. What does “(“ and “)” mean?

Motivation

Is the expression correct?

```
(a + b) * (c * d / $ + f)
```

```
 --> 1:20
  |
1 | (a + b) * (c * d / $ + f)
  |                    ^---
  |
  = expected integer or variable
```

Motivation

What does “a”, ”b”mean? What does ”+” “/” mean?

```
(a + b) * (c * d / e + f)
```

```
- expr
  - expr
    - variable: "a"
    - add: "+"
    - variable: "b"
  - multiply: "*"
  - expr
    - variable: "c"
    - multiply: "*"
    - variable: "d"
    - divide: "/"
    - variable: "e"
    - add: "+"
    - variable: "f"
```

The AST is enriched with information about the different parts of the expression

Motivation

What does “(“ and “)” mean?

```
(a + b) * (c * d / e + f)
```

```
- expr
  - expr
    - variable: "a"
    - add: "+"
    - variable: "b"
  - multiply: "*"
  - expr
    - variable: "c"
    - multiply: "*"
    - variable: "d"
    - divide: "/"
    - variable: "e"
    - add: "+"
    - variable: "f"
```

Parser already processed and validated “(“ and “)”. We don’t care about them in the business logic

History

History

John Backus

History

The earliest compilers were written with the definition of the langauge **buried deeply within the code**. With these compilers it **was very difficult to verify** that the compiler accepted all of the langauge syntax and only the language syntax. This became especially difficult when the definition of the language was changed for later versions.

https://www.andrews.edu/~bidwell/456/history.html

History

Algol

Peter Naur

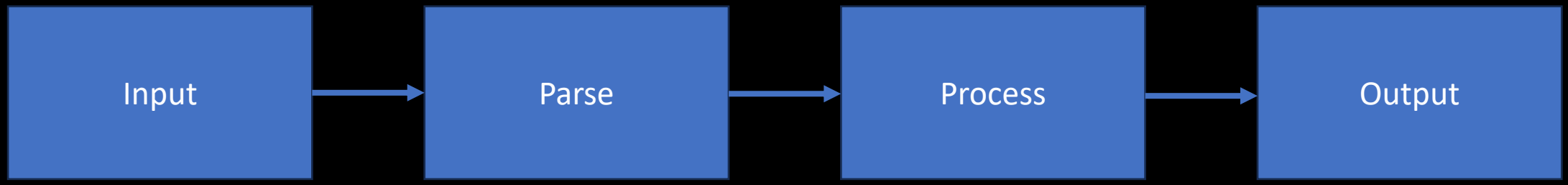
BNF (Backus-Naur Form)

```
(2.0 * PI) / n

<expression>  ::=  <expression> + <term>
              |    <expression> - <term>
              |    <term>
<term>        ::=  <term> * <factor>
              |    <term> / <factor>
              |    <factor>
<factor>      ::=  number
              |     name
              |     ( <expression> )
```

CSE 341 -- S. Tanimoto Syntax and Types 5

https://courses.cs.washington.edu/courses/cse341/03sp/slides/Syntax-and-Type/sld005.htm

Parsing



This is maintainable and easy to debug. When you mix parsing logic and processing logic in the same code, I wish you good luck debugging it

Technical intro: pest.rs

Technical intro: pest.rs

- Generating grammar code at compile time
- Easy to use
- Good errors
- Fantastic documentation
- Big community

Technical intro: pest.rs

```
65279,1179403647,1463895090
3.1415927,2.7182817,1.618034
-40,-273.15 13,42
65537
```

numbers.csv

Technical intro: pest.rs

```
65279,1179403647,1463895090
3.1415927,2.7182817,1.618034
-40,-273.15 13,42
65537
```

numbers.csv

```
field = { (ASCII_DIGIT | "." | "-")+ }
record = { field ~ ("," ~ field)* }
file = { SOI ~ (record ~ ("\r\n" | "\n"))* ~ EOI }
```

project/src/csv.pest

Rules are defined in a separate file –> even easier to maintain ☺

Technical intro: pest.rs

```
field = { (ASCII_DIGIT | "." | "-")+ }
record = { field ~ ("," ~ field)* }
file = { SOI ~ (record ~ ("\r\n" | "\n"))* ~ EOI }
```

project/src/csv.pest

```
use pest::Parser;
use pest_derive::Parser;

#[derive(Parser)]
#[grammar = "csv.pest"]
pub struct CSVParser;
```

project/src/main.rs

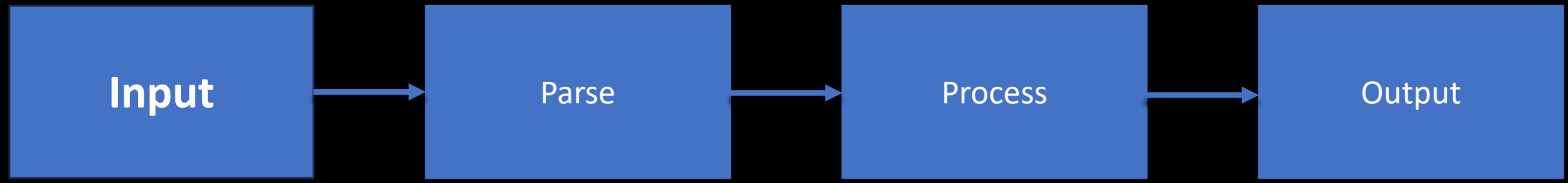
Technical intro: pest.rs

```
field = { (ASCII_DIGIT | "." | "-")+ }
record = { field ~ ("," ~ field)* }
file = { SOI ~ (record ~ ("\r\n" | "\n"))* ~ EOI }
```

project/src/csv.pest

```
use std::fs;

fn main() {
  let unparsed_file = fs::read_to_string("numbers.csv").expect("cannot read file");

// ...
}
```

project/src/main.rs



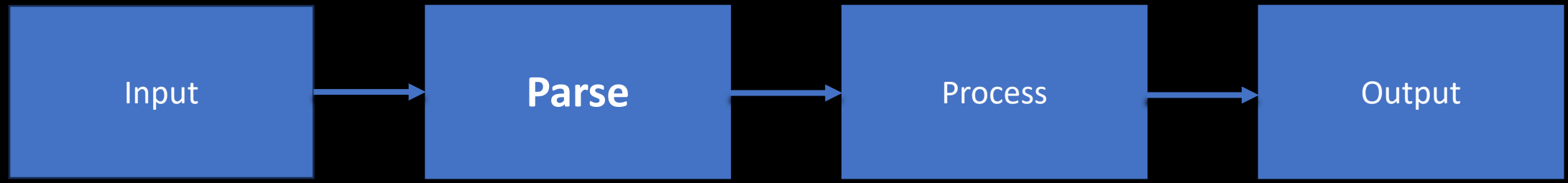
Technical intro: pest.rs

```
field = { (ASCII_DIGIT | "." | "-")+ }
record = { field ~ ("," ~ field)* }
file = { SOI ~ (record ~ ("\r\n" | "\n"))* ~ EOI }
```

project/src/csv.pest

```
fn main() {
  // ...

  let file = CSVParser::parse(Rule::file, &unparsed_file)
              .expect("unsuccessful parse")
              .next().unwrap();

   // ...
}
```

project/src/main.rs



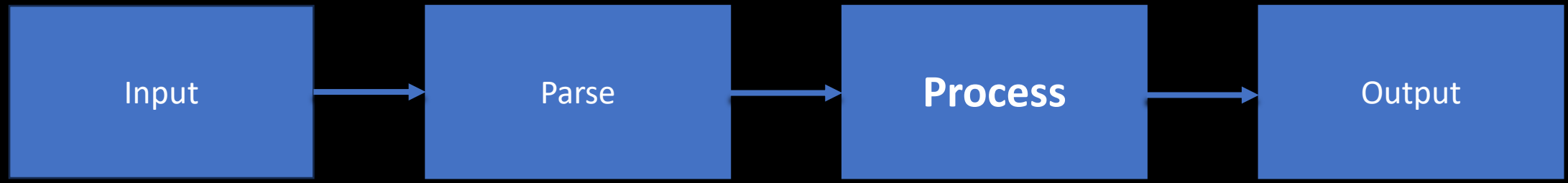
Technical intro: pest.rs

```
field = { (ASCII_DIGIT | "." | "-")+ }
record = { field ~ ("," ~ field)* }
file = { SOI ~ (record ~ ("\r\n" | "\n"))* ~ EOI }
```

project/src/csv.pest

```
fn main() {
  // ...

  let mut field_sum: f64 = 0.0;
  let mut record_count: u64 = 0;

  for record in file.into_inner() {
    match record.as_rule() {
       Rule::record => {
               record_count += 1;

            for field in record.into_inner() {
               field_sum += field.as_str().parse::<f64>().unwrap();
          }
       }
       Rule::EOI => (),
        _ => unreachable!(),
     }
   }
 // ...
}
```

project/src/main.rs



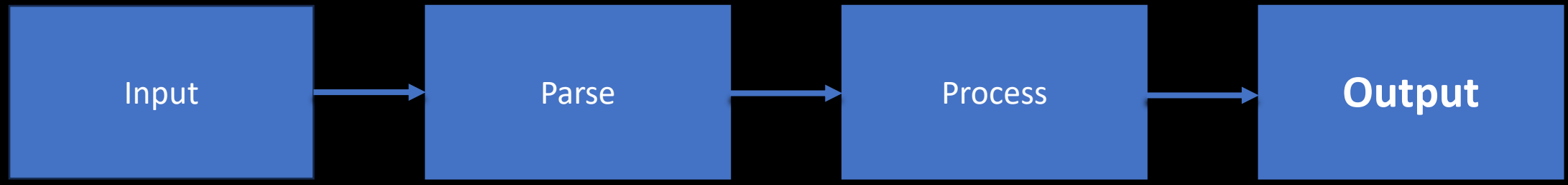
Technical intro: pest.rs

```
field = { (ASCII_DIGIT | "." | "-")+ }
record = { field ~ ("," ~ field)* }
file = { SOI ~ (record ~ ("\r\n" | "\n"))* ~ EOI }
```

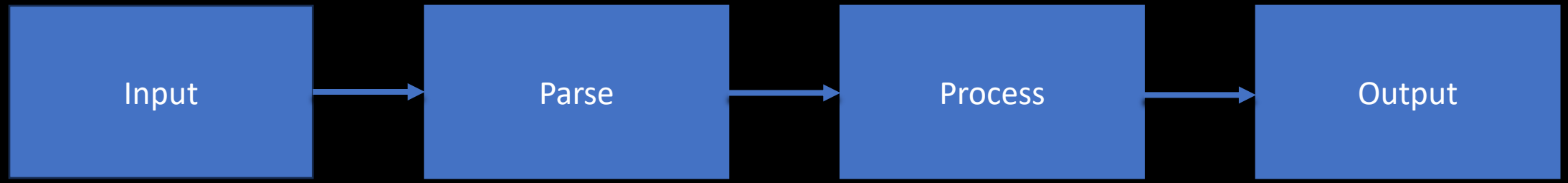
project/src/csv.pest

```
fn main() {
  // ...

println!("Sum of fields: {}", field_sum);
println!("Number of records: {}", record_count);

}
```

project/src/main.rs



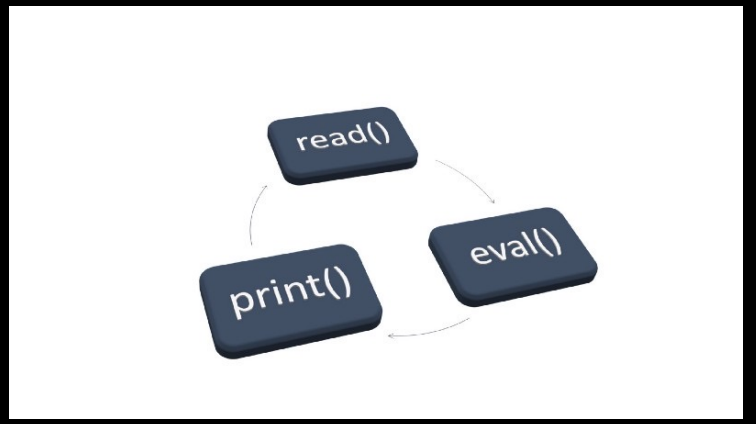
Technical intro: pest.rs



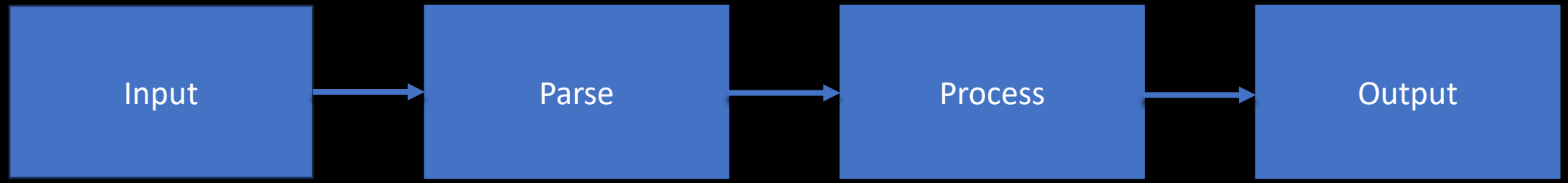
This approach is traditionally known as REPL

- Read
- Eval
- Print
- Loop

https://en.wikipedia.org/wiki/Read%E2%80%93eval%E2%80%93print_loop



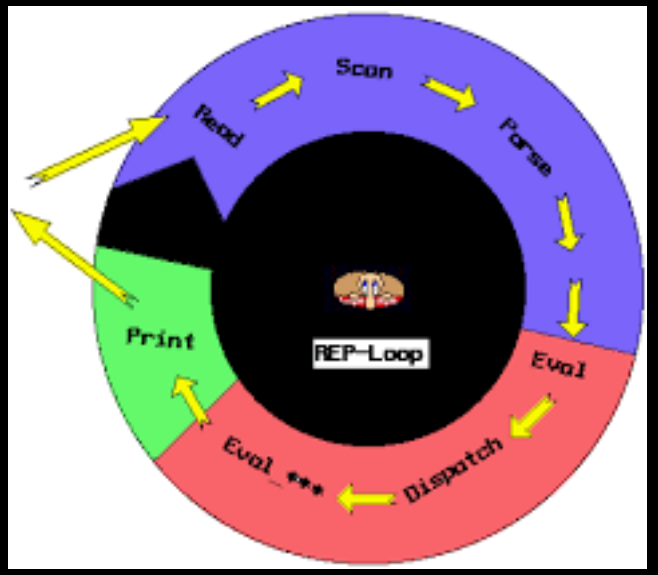
Technical intro: pest.rs



This approach is traditionally known as REPL

- Read
- Eval
- Print
- Loop

https://en.wikipedia.org/wiki/Read%E2%80%93eval%E2%80%93print_loop



Technical intro: pest.rs

```
field = { (ASCII_DIGIT | "." | "-")+ }
record = { field ~ ("," ~ field)* }
file = { SOI ~ (record ~ ("\r\n" | "\n"))* ~ EOI }
```

Technical intro: pest.rs

```
field = { (ASCII_DIGIT | "." | "-")+ }
record = { field ~ ("," ~ field)* }
file = { SOI ~ (record ~ ("\r\n" | "\n"))* ~ EOI }
```

+	One or more
|	Or
ASCII_DIGIT	A digit between 0 and 9

Field: At least one digit, containing a dot or a minus

Note: didactic example. Dot and minus are not enforced to be in their correct position

Technical intro: pest.rs

```
field = { (ASCII_DIGIT | "." | "-")+ }
record = { field ~ ("," ~ field)* }
file = { SOI ~ (record ~ ("\r\n" | "\n"))* ~ EOI }
```

~: Followed by

*: Zero or more

Record: a field followed by zero or more fields seperated by a comma

Technical intro: pest.rs

```
field = { (ASCII_DIGIT | "." | "-")+ }
record = { field ~ ("," ~ field)* }
file = { SOI ~ (record ~ ("\r\n" | "\n"))* ~ EOI }
```

SOI Start of input
EOI End of input

File: contains one or more records seperated by a new line

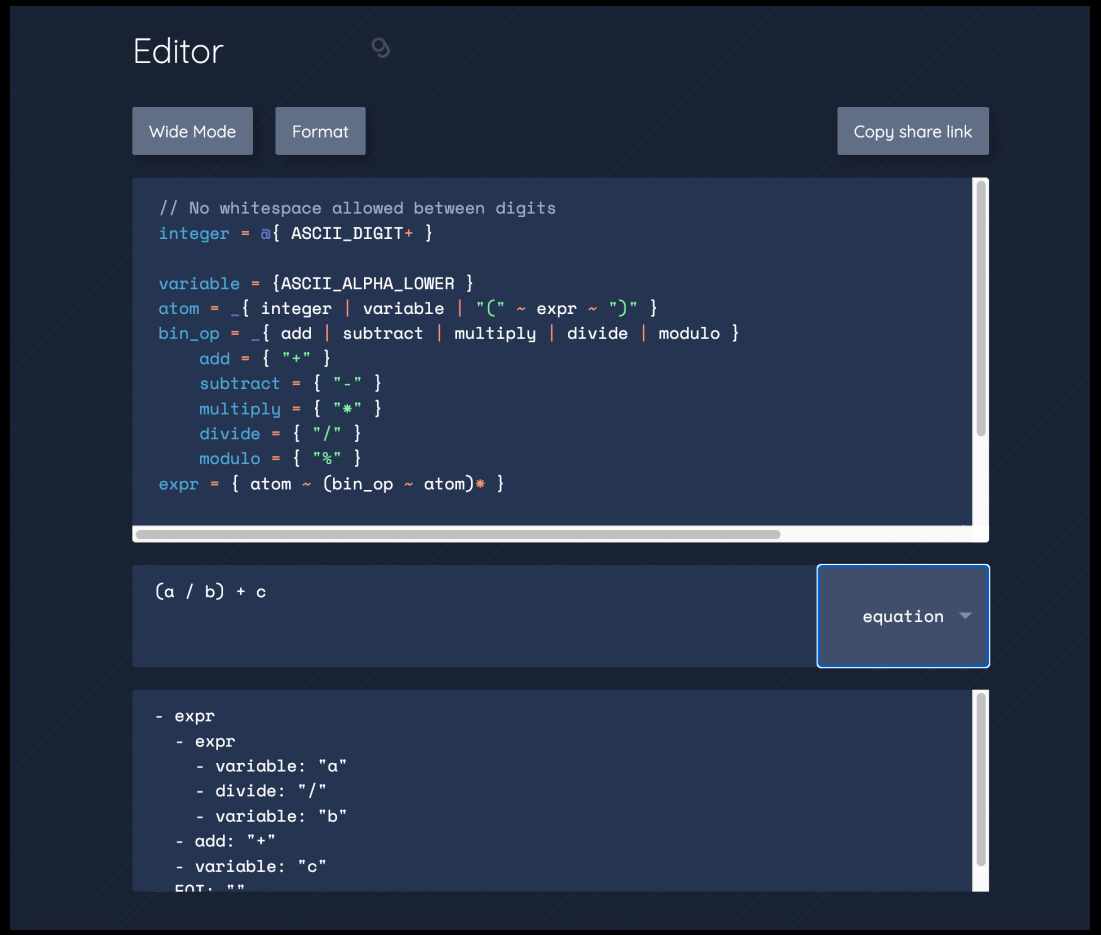
Technical intro: pest.rs

```
field = { (ASCII_DIGIT | "." | "-")+ }
record = { field ~ ("," ~ field)* }
file = { SOI ~ (record ~ ("\r\n" | "\n"))* ~ EOI }
```

The pest.rs book: https://pest.rs/book/intro.html (very easy to follow. Believe me)

pest.rs documentation: https://docs.rs/pest/latest/pest/

Technical intro: pest.rs



pest.rs online editor: Validate and test your grammar online: https://pest.rs/

Small remark

Traditionally, lexing and parsing are two seperated steps

Lexing: Which tokens do I need to accept and use in my parser

Parsing: Which rules should my program accept

Small remark

Traditionally, lexing and parsing are two seperated steps

Lexing: Which tokens do I need to accept and use in my parser

Parsing: Which rules should my program accept

```
(define imprec/cc2-parser
  (parser
   (tokens imprec/cc2-tokens imprec/cc2-empty-tokens)

   (precs (left AMB))
   (grammar
    (expr
     ((number)    $1)
     ((variable)  $1)
     ((let)       $1)
     ((letrec)    $1)
     ((if)        $1)
     ((procedure) $1)
     ((apply)     $1)
     ((assign)    $1)
     ((block)     $1)
     ((while)     $1)
     )

    (number
     ((NUM) (num-exp $1)))

    (let
        ((LET VAR DEF expr IN expr) (let-exp $2 $4 $6))
        ((LET PERSISTENT VAR DEF expr IN expr) (let-exp $3 $5 $7)))

    (letrec
        ((REC VAR DEF expr IN expr) (rec-exp $2 $4 $6)))

    (if
     ((IF expr THEN expr ELSE expr) (ite-exp $2 $4 $6)))
```

This is a parser. You define the grammar (syntax) of the program

Small remark

Traditionally, lexing and parsing are two seperated steps

Lexing: Which tokens do I need to accept and use in my parser

Parsing: Which rules should my program accept

```
(define imprec/cc2-lexer
  (lexer
   ;; Keywords
   ["let"
    (token-LET)]
   ["letrec"
    (token-REC)]
   ["in"
    (token-IN)]
   ["if"
    (token-IF)]
   ["then"
    (token-THEN)]
   ["else"
    (token-ELSE)]
   ["proc"
    (token-PROC)]
   ["begin"
    (token-BEGIN)]
   ["end"
    (token-END)]
   [#\=
    (token-DEF)]
   [(:: #\: #\=)
    (token-ASS)]
   [#\;
    (token-SEM)]
   [#\,
    (token-COM)]
   [#\newline
    (imprec/cc2-lexer input-port)
```

This is a lexer. You define which tokens to accept

Small remark

Traditionally, lexing and parsing are two seperated steps

Lexing: Which tokens do I need to accept and use in my parser

Parsing: Which rules should my program accept

pest.rs does not have a distinction between lexing/parsing

This makes it more practical (less boilerplate to write ☺)