Introduction

*Speed or simplicity? Why not **both**?*

`pest` is a library for writing plain-text parsers in Rust.

Parsers that use `pest` are **easy to design and maintain** due to the use of Parsing Expression Grammars, or *PEGs*. And, because of Rust's zero-cost abstractions, `pest` parsers can be **very fast**.

Sample

Here is the complete grammar for a simple calculator developed in a (currently unwritten) later chapter:

```
num = @{ int ~ ("." ~ ASCII_DIGIT*)? ~ (^"e" ~ int)? }
    int = { ("+" | "-")? ~ ASCII_DIGIT+ }

operation = _{ add | subtract | multiply | divide | power }
    add      = { "+" }
    subtract = { "-" }
    multiply = { "*" }
    divide   = { "/" }
    power    = { "^" }

expr = { term ~ (operation ~ term)* }
term = _{ num | "(" ~ expr ~ ")" }

calculation = _{ SOI ~ expr ~ EOI }

WHITESPACE = _{ " " | "\t" }
```

And here is the function that uses that parser to calculate answers:

```
lazy_static! {
    static ref PREC_CLIMBER: PrecClimber<Rule> = {
        use Rule::*;
        use Assoc::*;

        PrecClimber::new(vec![
            Operator::new(add, Left) | Operator::new(subtract, Left),
            Operator::new(multiply, Left) | Operator::new(divide, Left),
            Operator::new(power, Right)
        ])
    };
}

fn eval(expression: Pairs<Rule>) -> f64 {
    PREC_CLIMBER.climb(
        expression,
        |pair: Pair<Rule>| match pair.as_rule() {
            Rule::num => pair.as_str().parse::<f64>().unwrap(),
            Rule::expr => eval(pair.into_inner()),
            _ => unreachable!(),
        },
        |lhs: f64, op: Pair<Rule>, rhs: f64| match op.as_rule() {
            Rule::add      => lhs + rhs,
            Rule::subtract => lhs - rhs,
            Rule::multiply => lhs * rhs,
            Rule::divide   => lhs / rhs,
            Rule::power    => lhs.powf(rhs),
            _ => unreachable!(),
        },
    )
}
```

About this book

This book provides an overview of `pest` as well as several example parsers. For more details of `pest` 's API, check the documentation.

Note that `pest` uses some advanced features of the Rust language. For an introduction to Rust, consult the official Rust book.

Example: CSV

Comma-Separated Values is a very simple text format. CSV files consist of a list of

records, each on a separate line. Each record is a list of *fields* separated by commas.

For example, here is a CSV file with numeric fields:

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

Let's write a program that computes the **sum of these fields** and counts the **number of records**.

Setup

Start by initializing a new project using Cargo:

```
$ cargo init --bin csv-tool
     Created binary (application) project
$ cd csv-tool
```

Add the `pest` and `pest_derive` crates to the dependencies section in `Cargo.toml`:

```
[dependencies]
pest = "2.0"
pest_derive = "2.0"
```

And finally bring `pest` and `pest_derive` into scope in `src/main.rs`:

```
extern crate pest;
#[macro_use]
extern crate pest_derive;
```

The `#[macro_use]` attribute is necessary to use `pest` to generate parsing code! This is a very important attribute.

Writing the parser

`pest` works by compiling a description of a file format, called a *grammar*, into Rust code. Let's write a grammar for a CSV file that contains numbers. Create a new file named `src/csv.pest` with a single line:

```
field = { (ASCII_DIGIT | "." | "-")+ }
```

This is a description of every number field: each character is either an ASCII digit `0` through `9`, a full stop `.`, or a hyphen–minus `-`. The plus sign `+` indicates that the pattern can occur one or more times.

Rust needs to know to compile this file using `pest`:

```
use pest::Parser;

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

If you run `cargo doc`, you will see that `pest` has created the function `CSVParser::parse` and an enum called `Rule` with a single variant `Rule::field`.

Let's test it out! Rewrite `main`:

```
fn main() {
    let successful_parse = CSVParser::parse(Rule::field, "-273.15");
    println!("{:?}", successful_parse);

    let unsuccessful_parse = CSVParser::parse(Rule::field, "this is not
a number");
    println!("{:?}", unsuccessful_parse);
}
```

```
$ cargo run
  [ ... ]
Ok([Pair { rule: field, span: Span { str: "-273.15", start: 0, end: 7 },
inner: [] }])
Err(Error { variant: ParsingError { positives: [field], negatives: [] },
location: Pos(0), path: None, line: "this is not a number",
continued_line: None, start: (1, 1), end: None })
```

Yikes! That's a complicated type! But you can see that the successful parse was `Ok`, while the failed parse was `Err`. We'll get into the details later.

For now, let's complete the grammar:

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

The tilde `~` means "and then", so that `"abc" ~ "def"` matches `abc` followed by `def`. (For this grammar, `"abc" ~ "def"` is exactly the same as `"abcdef"`, although this is not true in general; see a later chapter about `WHITESPACE`.)

In addition to literal strings (`"\r\n"`) and built-ins (`ASCII_DIGIT`), rules can contain other rules. For instance, a `record` is a `field`, and optionally a comma `,` and then another `field` repeated as many times as necessary. The asterisk `*` is just like the plus sign `+`, except the pattern is optional: it can occur any number of times at all (zero or more).

There are two more rules that we haven't defined: `SOI` and `EOI` are two special rules that match, respectively, the *start of input* and the *end of input*. Without `EOI`, the `file` rule would gladly parse an invalid file! It would just stop as soon as it found the first invalid character and report a successful parse, possibly consisting of nothing at all!

The main program loop

Now we're ready to finish the program. We will use `File` to read the CSV file into memory. We'll also be messy and use `expect` everywhere.

```
use std::fs;

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

    // ...
}
```

Next we invoke the parser on the file. Don't worry about the specific types for now. Just know that we're producing a `pest::iterators::Pair` that represents the `file` rule in our grammar.

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

    let file = CSVParser::parse(Rule::file, &unparsed_file)
        .expect("unsuccessful parse") // unwrap the parse result
        .next().unwrap(); // get and unwrap the `file` rule; never fails

    // ...
}
```

Finally, we iterate over the `record`s and `field`s, while keeping track of the count and sum, then print those numbers out.

```
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!(),
        }
    }

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

If `p` is a parse result (a `Pair`) for a rule in the grammar, then `p.into_inner()` returns an iterator over the named sub-rules of that rule. For instance, since the `file` rule in our grammar has `record` as a sub-rule, `file.into_inner()` returns an iterator over each parsed `record`. Similarly, since a `record` contains `field` sub-rules, `record.into_inner()` returns an iterator over each parsed `field`.

Done

Try it out! Copy the sample CSV at the top of this chapter into a file called `numbers.csv`, then run the program! You should see something like this:

```
$ cargo run
  [ ... ]
Sum of fields: 2643429302.327908
Number of records: 5
```

Parser API

`pest` provides several ways of accessing the results of a successful parse. The examples below use the following grammar:

```
number = { ASCII_DIGIT+ }                // one or more decimal digits
enclosed = { "(.." ~ number ~ "..)" }    // for instance, "(..6472..)"
sum = { number ~ " + " ~ number }        // for instance, "1362 + 12"
```

Tokens

`pest` represents successful parses using *tokens*. Whenever a rule matches, two tokens are produced: one at the *start* of the text that the rule matched, and one at the *end*. For example, the rule `number` applied to the string `"3130 abc"` would match and produce this pair of tokens:

```
"3130 abc"
 |   ^ end(number)
 ^ start(number)
```

Note that the rule doesn't match the entire input text. It only matches as much text as possible, then stops if successful.

A token is like a cursor in the input string. It has a character position in the string, as well as a reference to the rule that created it.

Nested rules

If a named rule contains another named rule, tokens will be produced for *both* rules. For instance, the rule `enclosed` applied to the string `"(..6472..)"` would

match and produce these four tokens:

```
"(..6472..)"
 |  |   |  ^ end(enclosed)
 |  |   ^ end(number)
 |  ^ start(number)
 ^ start(enclosed)
```

Sometimes, tokens might not occur at distinct character positions. For example, when parsing the rule `sum`, the inner `number` rules share some start and end positions:

```
"1773 + 1362"
 |   |  |   ^ end(sum)
 |   |  |   ^ end(number)
 |   |  ^ start(number)
 |   ^ end(number)
 ^ start(number)
 ^ start(sum)
```

In fact, for a rule that matches empty input, the start and end tokens will be at the same position!

Interface

Tokens are exposed as the `Token` enum, which has `Start` and `End` variants. You can get an iterator of `Token`s by calling `tokens` on a parse result:

```
let parse_result = Parser::parse(Rule::sum, "1773 + 1362").unwrap();
let tokens = parse_result.tokens();

for token in tokens {
    println!("{:?}", token);
}
```

After a successful parse, tokens will occur as nested pairs of matching `Start` and `End`. Both kinds of tokens have two fields:

- `rule`, which explains which rule generated them; and
- `pos`, which indicates their positions.

A start token's position is the first character that the rule matched. An end token's position is the first character that the rule did not match — that is, an end token

refers to a position *after* the match. If a rule matched the entire input string, the end token points to an imaginary position *after* the string.

Pairs

Tokens are not the most convenient interface, however. Usually you will want to explore the parse tree by considering matching pairs of tokens. For this purpose, `pest` provides the `Pair` type.

A `Pair` represents a matching pair of tokens, or, equivalently, the spanned text that a named rule successfully matched. It is commonly used in several ways:

- Determining which rule produced the `Pair`
- Using the `Pair` as a raw `&str`
- Inspecting the inner named sub-rules that produced the `Pair`

```
let pair = Parser::parse(Rule::enclosed, "(..6472..) and more text")
    .unwrap().next().unwrap();

assert_eq!(pair.as_rule(), Rule::enclosed);
assert_eq!(pair.as_str(), "(..6472..)");

let inner_rules = pair.into_inner();
println!("{}", inner_rules); // --> [number(3, 7)]
```

In general, a `Pair` might have any number of inner rules: zero, one, or more. For maximum flexibility, `Pair::into_inner()` returns `Pairs`, which is an iterator over each pair.

This means that you can use `for` loops on parse results, as well as iterator methods such as `map`, `filter`, and `collect`.

```
let pairs = Parser::parse(Rule::sum, "1773 + 1362")
    .unwrap().next().unwrap()
    .into_inner();

let numbers = pairs
    .clone()
    .map(|pair| str::parse(pair.as_str()).unwrap())
    .collect::<Vec<i32>>();
assert_eq!(vec![1773, 1362], numbers);

for (found, expected) in pairs.zip(vec!["1773", "1362"]) {
    assert_eq!(Rule::number, found.as_rule());
    assert_eq!(expected, found.as_str());
}
```

`Pairs` iterators are also commonly used via the `next` method directly. If a rule consists of a known number of sub-rules (for instance, the rule `sum` has exactly two sub-rules), the sub-matches can be extracted with `next` and `unwrap`:

```
let parse_result = Parser::parse(Rule::sum, "1773 + 1362")
    .unwrap().next().unwrap();
let mut inner_rules = parse_result.into_inner();

let match1 = inner_rules.next().unwrap();
let match2 = inner_rules.next().unwrap();

assert_eq!(match1.as_str(), "1773");
assert_eq!(match2.as_str(), "1362");
```

Sometimes rules will not have a known number of sub-rules, such as when a sub-rule is repeated with an asterisk `*`:

```
list = { number* }
```

In cases like these it is not possible to call `.next().unwrap()`, because the number of sub-rules depends on the input string — it cannot be known at compile time.

The `parse` method

A `pest`-derived `Parser` has a single method `parse` which returns a `Result< Pairs, Error >`. To access the underlying parse tree, it is necessary to

`match` on or `unwrap` the result:

```
// check whether parse was successful
match Parser::parse(Rule::enclosed, "(..6472..)") {
    Ok(mut pairs) => {
        let enclosed = pairs.next().unwrap();
        // ...
    }
    Err(error) => {
        // ...
    }
}
```

Our examples so far have included the calls `Parser::parse(...).unwrap().next().unwrap()`. The first `unwrap` turns the result into a `Pairs`. If parsing had failed, the program would panic! We only use `unwrap` in these examples because we already know that they will parse successfully.

In the example above, in order to get to the `enclosed` rule inside of the `Pairs`, we use the iterator interface. The `next()` call returns an `Option<Pair>`, which we finally `unwrap` to get the `Pair` for the `enclosed` rule.

Using `Pair` and `Pairs` with a grammar

While the `Result` from `Parser::parse(...)` might very well be an error on invalid input, `Pair` and `Pairs` often have more subtle behavior. For instance, with this grammar:

```
number = { ASCII_DIGIT+ }
sum = { number ~ " + " ~ number }
```

this function will *never* panic:

```
fn process(pair: Pair<Rule>) -> f64 {
    match pair.as_rule() {
        Rule::number => str::parse(pair.as_str()).unwrap(),
        Rule::sum => {
            let mut pairs = pair.into_inner();

            let num1 = pairs.next().unwrap();
            let num2 = pairs.next().unwrap();

            process(num1) + process(num2)
        }
    }
}
```

`str::parse(...).unwrap()` is safe because the `number` rule only ever matches digits, which `str::parse(...)` can handle. And `pairs.next().unwrap()` is safe to call twice because a `sum` match *always* has two sub-matches, which is guaranteed by the grammar.

Since these sorts of guarantees are awkward to express with Rust types, `pest` only provides a few high-level types to represent parse trees. Nevertheless, you *should* rely on the meaning of your grammar for properties such as "contains *n* sub-rules", "is safe to `parse` to `f32`", and "never fails to match". Idiomatic `pest` code uses `unwrap` and `unreachable!`.

Spans and positions

Occasionally, you will want to refer to a matching rule in the context of the raw source text, rather than the interior text alone. For example, you might want to print the entire line that contained the match. For this you can use `Span` and `Position`.

A `Span` is returned from `Pair::as_span`. Spans have a start position and an end position (which correspond to the start and end tokens of the rule that made the pair).

Spans can be decomposed into their start and end `Position`s, which provide useful methods for examining the string around that position. For example, `Position::line_col()` finds out the line and column number of a position.

Essentially, a `Position` is a `Token` without a rule. In fact, you can use pattern

matching to turn a `Token` into its component `Rule` and `Position`.

Example: INI

INI (short for *initialization*) files are simple configuration files. Since there is no standard for the format, we'll write a program that is able to parse this example file:

```
username = noha
password = plain_text
salt = NaCl

[server_1]
interface=eth0
ip=127.0.0.1
document_root=/var/www/example.org

[empty_section]

[second_server]
document_root=/var/www/example.com
ip=
interface=eth1
```

Each line contains a **key and value** separated by an equals sign; or contains a **section name** surrounded by square brackets; or else is **blank** and has no meaning.

Whenever a section name appears, the following keys and values belong to that section, until the next section name. The key–value pairs at the beginning of the file belong to an implicit "empty" section.

Writing the grammar

Start by initializing a new project using Cargo, adding the dependencies `pest = "2.0"` and `pest_derive = "2.0"`. Make a new file, `src/ini.pest`, to hold the grammar.

The text of interest in our file — `username`, `/var/www/example.org`, *etc.* — consists of only a few characters. Let's make a rule to recognize a single character in that set. The built-in rule `ASCII_ALPHANUMERIC` is a shortcut to represent any uppercase

or lowercase ASCII letter, or any digit.

```
char = { ASCII_ALPHANUMERIC | "." | "_" | "/" }
```

Section names and property keys *must not* be empty, but property values *may* be empty (as in the line `ip=` above). That is, the former consist of one or more characters, `char+`; and the latter consist of zero or more characters, `char*`. We separate the meaning into two rules:

```
name = { char+ }
value = { char* }
```

Now it's easy to express the two kinds of input lines.

```
section = { "[" ~ name ~ "]" }
property = { name ~ "=" ~ value }
```

Finally, we need a rule to represent an entire input file. The expression `(section | property)?` matches `section`, `property`, or else nothing. Using the built-in rule `NEWLINE` to match line endings:

```
file = {
    SOI ~
    ((section | property)? ~ NEWLINE)* ~
    EOI
}
```

To compile the parser into Rust, we need to add the following to `src/main.rs`:

```
extern crate pest;
#[macro_use]
extern crate pest_derive;

use pest::Parser;

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

Program initialization

Now we can read the file and parse it with `pest`:

```
use std::collections::HashMap;
use std::fs;

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

    let file = INIParser::parse(Rule::file, &unparsed_file)
        .expect("unsuccessful parse") // unwrap the parse result
        .next().unwrap(); // get and unwrap the `file` rule; never fails

    // ...
}
```

We'll express the properties list using nested `HashMap`s. The outer hash map will have section names as keys and section contents (inner hash maps) as values. Each inner hash map will have property keys and property values. For example, to access the `document_root` of `server_1`, we could write `properties["server_1"]["document_root"]`. The implicit "empty" section will be represented by a regular section with an empty string `""` for the name, so that `properties[""]["salt"]` is valid.

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

    let mut properties: HashMap<&str, HashMap<&str, &str>> =
HashMap::new();

    // ...
}
```

Note that the hash map keys and values are all `&str`, borrowed strings. `pest` parsers do not copy the input they parse; they borrow it. All methods for inspecting a parse result return strings which are borrowed from the original parsed string.

The main loop

Now we interpret the parse result. We loop through each line of the file, which is either a section name or a key–value property pair. If we encounter a section name, we update a variable. If we encounter a property pair, we obtain a reference

to the hash map for the current section, then insert the pair into that hash map.

```
    // ...

    let mut current_section_name = "";

    for line in file.into_inner() {
        match line.as_rule() {
            Rule::section => {
                let mut inner_rules = line.into_inner(); // { name }
                current_section_name =
inner_rules.next().unwrap().as_str();
            }
            Rule::property => {
                let mut inner_rules = line.into_inner(); // { name ~ "="
~ value }

                let name: &str = inner_rules.next().unwrap().as_str();
                let value: &str = inner_rules.next().unwrap().as_str();

                // Insert an empty inner hash map if the outer hash map
hasn't
                // seen this section name before.
                let section =
properties.entry(current_section_name).or_default();
                section.insert(name, value);
            }
            Rule::EOI => (),
            _ => unreachable!(),
        }
    }

    // ...
```

For output, let's simply dump the hash map using the pretty-printed `Debug` format.

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

    println!("{:#?}", properties);
}
```

Whitespace

If you copy the example INI file at the top of this chapter into a file `config.ini` and run the program, it will not parse. We have forgotten about the optional spaces around equals signs!

Handling whitespace can be inconvenient for large grammars. Explicitly writing a `whitespace` rule and manually inserting it makes a grammar difficult to read and modify. `pest` provides a solution using the special rule `WHITESPACE`. If defined, it will be implicitly run, as many times as possible, at every tilde `~` and between every repetition (for example, `*` and `+`). For our INI parser, only spaces are legal whitespace.

```
WHITESPACE = _{ " " }
```

We mark the `WHITESPACE` rule *silent* with a leading low line (underscore) `_{ ... }`. This way, even if it matches, it won't show up inside other rules. If it weren't silent, parsing would be much more complicated, since every call to `Pairs::next(...)` could potentially return `Rule::WHITESPACE` instead of the desired next regular rule.

But wait! Spaces shouldn't be allowed in section names, keys, or values! Currently, whitespace is automatically inserted between characters in `name = { char+ }`. Rules that *are* whitespace-sensitive need to be marked *atomic* with a leading at sign `@{ ... }`. In atomic rules, automatic whitespace handling is disabled, and interior rules are silent.

```
name = @{ char+ }
value = @{ char* }
```

Done

Try it out! Make sure that the file `config.ini` exists, then run the program! You should see something like this:

```
$ cargo run
  [ ... ]
{
    "": {
        "password": "plain_text",
        "username": "noha",
        "salt": "NaCl"
    },
    "second_server": {
        "ip": "",
        "document_root": "/var/www/example.com",
        "interface": "eth1"
    },
    "server_1": {
        "interface": "eth0",
        "document_root": "/var/www/example.org",
        "ip": "127.0.0.1"
    }
}
```

Grammars

Like many parsing tools, `pest` operates using a *formal grammar* that is distinct from your Rust code. The format that `pest` uses is called a *parsing expression grammar*, or *PEG*. When building a project, `pest` automatically compiles the PEG, located in a separate file, into a plain Rust function that you can call.

How to activate `pest`

Most projects will have at least two files that use `pest`: the parser (say, `src/parser/mod.rs`) and the grammar (`src/parser/grammar.pest`). Assuming that they are in the same directory:

```
use pest::Parser;

#[derive(Parser)]
#[grammar = "parser/grammar.pest"] // relative to project `src`
struct MyParser;
```

Whenever you compile this file, `pest` will automatically use the grammar file to generate items like this:

```
pub enum Rules { /* ... */ }

impl Parser for MyParser {
    pub fn parse(Rules, &str) -> pest::Pairs { /* ... */ }
}
```

You will never see `enum Rules` or `impl Parser` as plain text! The code only exists during compilation. However, you can use `Rules` just like any other enum, and you can use `parse(...)` through the `Pairs` interface described in the Parser API chapter.

Warning about PEGs!

Parsing expression grammars look quite similar to other parsing tools you might be used to, like regular expressions, BNF grammars, and others (Yacc/Bison, LALR, CFG). However, PEGs behave subtly differently: PEGs are eager, non-backtracking, ordered, and unambiguous.

Don't be scared if you don't recognize any of the above names! You're already a step ahead of people who do — when you use `pest`'s PEGs, you won't be tripped up by comparisons to other tools.

If you have used other parsing tools before, be sure to read the next section carefully. We'll mention some common mistakes regarding PEGs.

Parsing expression grammar

Parsing expression grammars (PEGs) are simply a strict representation of the simple imperative code that you would write if you were writing a parser by hand.

```
number = {             // To recognize a number...
    ASCII_DIGIT+       //   take as many ASCII digits as possible (at
least one).
}
expression = {         // To recognize an expression...
    number             //   first try to take a number...
    | "true"           //   or, if that fails, the string "true".
}
```

In fact, `pest` produces code that is quite similar to the pseudo-code in the

comments above.

Eagerness

When a repetition PEG expression is run on an input string,

```
ASCII_DIGIT+      // one or more characters from '0' to '9'
```

it runs that expression as many times as it can (matching "eagerly", or "greedily"). It either succeeds, consuming whatever it matched and passing the remaining input on to the next step in the parser,

```
"42 boxes"
 ^ Running ASCII_DIGIT+

"42 boxes"
   ^ Successfully took one or more digits!

" boxes"
 ^ Remaining unparsed input.
```

or fails, consuming nothing.

```
"galumphing"
 ^ Running ASCII_DIGIT+
   Failed to take one or more digits!

"galumphing"
 ^ Remaining unparsed input (everything).
```

If an expression fails to match, the failure propagates upwards, eventually leading to a failed parse, unless the failure is "caught" somewhere in the grammar. The *choice operator* is one way to "catch" such failures.

Ordered choice

The choice operator, written as a vertical line `|`, is *ordered*. The PEG expression `first | second` means "try `first`; but if it fails, try `second` instead".

In many cases, the ordering does not matter. For instance, `"true" | "false"` will

match either the string `"true"` or the string `"false"` (and fail if neither occurs).

However, sometimes the ordering *does* matter. Consider the PEG expression `"a" | "ab"`. You might expect it to match either the string `"a"` or the string `"ab"`. But it will not — the expression means "try `"a"`; but if it fails, try `"ab"` instead". If you are matching on the string `"abc"`, "try `"a"`" will *not* fail; it will instead match `"a"` successfully, leaving `"bc"` unparsed!

In general, when writing a parser with choices, put the longest or most specific choice first, and the shortest or most general choice last.

Non-backtracking

During parsing, a PEG expression either succeeds or fails. If it succeeds, the next step is performed as usual. But if it fails, the whole expression fails. The engine will not back up and try again.

Consider this grammar, matching on the string `"frumious"`:

```
word = {     // to recognize a word...
    ANY*     //   take any character, zero or more times...
    ~ ANY    //   followed by any character
}
```

You might expect this rule to parse any input string that contains at least one character (equivalent to `ANY+`). But it will not. Instead, the first `ANY*` will eagerly eat the entire string — it will *succeed*. Then, the next `ANY` will have nothing left, so it will fail.

```
"frumious"
 ^ (word)

"frumious"
         ^ (ANY*) Success! Continue to `ANY` with remaining input "".

""
 ^ (ANY) Failure! Expected one character, but found end of string.
```

In a system with backtracking (like regular expressions), you would back up one step, "un-eating" a character, and then try again. But PEGs do not do this. In the rule `first ~ second`, once `first` parses successfully, it has consumed some characters that will never come back. `second` can only run on the input that

`first` did not consume.

Unambiguous

These rules form an elegant and simple system. Every PEG rule is run on the remainder of the input string, consuming as much input as necessary. Once a rule is done, the rest of the input is passed on to the rest of the parser.

For instance, the expression `ASCII_DIGIT+`, "one or more digits", will always match the largest sequence of consecutive digits possible. There is no danger of accidentally having a later rule back up and steal some digits in an unintuitive and nonlocal way.

This contrasts with other parsing tools, such as regular expressions and CFGs, where the results of a rule often depend on code some distance away. Indeed, the famous "shift/reduce conflict" in LR parsers is not a problem in PEGs.

Don't panic

This all might be a bit counterintuitive at first. But as you can see, the basic logic is very easy and straightforward. You can trivially step through the execution of any PEG expression.

- Try this.
- If it succeeds, try the next thing.
- Otherwise, try the other thing.

```
(this ~ next_thing) | (other_thing)
```

These rules together make PEGs very pleasant tools for writing a parser.

Syntax of pest parsers

`pest` grammars are lists of rules. Rules are defined like this:

```
my_rule = { ... }

another_rule = {        // comments are preceded by two slashes
    ...                 // whitespace goes anywhere
}
```

Since rule names are translated into Rust enum variants, they are not allowed to be Rust keywords.

The left curly bracket `{` defining a rule can be preceded by symbols that affect its operation:

```
silent_rule = _{ ... }
atomic_rule = @{ ... }
```

Expressions

Grammar rules are built from *expressions* (hence "parsing expression grammar"). These expressions are a terse, formal description of how to parse an input string.

Expressions are composable: they can be built out of other expressions and nested inside of each other to produce arbitrarily complex rules (although you should break very complicated expressions into multiple rules to make them easier to manage).

PEG expressions are suitable for both high-level meaning, like "a function signature, followed by a function body", and low-level meaning, like "a semicolon, followed by a line feed". The combining form "followed by", the sequence operator, is the same in either case.

Terminals

The most basic rule is a **literal string** in double quotes: `"text"`.

A string can be **case-insensitive** (for ASCII characters only) if preceded by a caret: `^"text"`.

A single **character in a range** is written as two single-quoted characters, separated by two dots: `'0'..'9'`.

You can match **any single character** at all with the special rule `ANY`. This is equivalent to `'\u{00}'..'\u{10FFFF}'`, any single Unicode character.

```
"a literal string"
^"ASCII case-insensitive string"
'a'..'z'
ANY
```

Finally, you can **refer to other rules** by writing their names directly, and even **use rules recursively**:

```
my_rule = { "slithy " ~ other_rule }
other_rule = { "toves" }
recursive_rule = { "mimsy " ~ recursive_rule }
```

Sequence

The sequence operator is written as a tilde `~`.

```
first ~ and_then

("abc") ~ (^"def") ~ ('g'..'z')        // matches "abcDEFr"
```

When matching a sequence expression, `first` is attempted. If `first` matches successfully, `and_then` is attempted next. However, if `first` fails, the entire expression fails.

A list of expressions can be chained together with sequences, which indicates that *all* of the components must occur, in the specified order.

Ordered choice

The choice operator is written as a vertical line `|`.

```
first | or_else

("abc") | (^"def") | ('g'..'z')        // matches "DEF"
```

When matching a choice expression, `first` is attempted. If `first` matches successfully, the entire expression *succeeds immediately*. However, if `first` fails, `or_else` is attempted next.

Note that `first` and `or_else` are always attempted at the same position, even if `first` matched some input before it failed. When encountering a parse failure, the engine will try the next ordered choice as though no input had been matched. Failed parses never consume any input.

```
start = { "Beware " ~ creature }
creature = {
    ("the " ~ "Jabberwock")
    | ("the " ~ "Jubjub bird")
}
```

```
"Beware the Jubjub bird"
 ^ (start) Parses via the second choice of `creature`,
           even though the first choice matched "the " successfully.
```

It is somewhat tempting to borrow terminology and think of this operation as "alternation" or simply "OR", but this is misleading. The word "choice" is used specifically because the operation is *not* merely logical "OR".

Repetition

There are two repetition operators: the asterisk `*` and plus sign `+`. They are placed after an expression. The asterisk `*` indicates that the preceding expression can occur **zero or more** times. The plus sign `+` indicates that the preceding expression can occur **one or more** times (it must occur at least once).

The question mark operator `?` is similar, except it indicates that the expression is **optional** — it can occur zero or one times.

```
("zero" ~ "or" ~ "more")*
 ("one" | "or" | "more")+
          (^"optional")?
```

Note that `expr*` and `expr?` will always succeed, because they are allowed to match zero times. For example, `"a"* ~ "b"?` will succeed even on an empty input string.

Other **numbers of repetitions** can be indicated using curly brackets:

```
expr{n}             // exactly n repetitions
expr{m, n}          // between m and n repetitions, inclusive

expr{, n}           // at most n repetitions
expr{m, }           // at least m repetitions
```

Thus `expr*` is equivalent to `expr{0, }`; `expr+` is equivalent to `expr{1, }`; and `expr?` is equivalent to `expr{0, 1}`.

Predicates

Preceding an expression with an ampersand `&` or exclamation mark `!` turns it into a *predicate* that never consumes any input. You might know these operators as "lookahead" or "non-progressing".

The **positive predicate**, written as an ampersand `&`, attempts to match its inner expression. If the inner expression succeeds, parsing continues, but at the *same position* as the predicate — `&foo ~ bar` is thus a kind of "AND" statement: "the input string must match `foo` AND `bar`". If the inner expression fails, the whole expression fails too.

The **negative predicate**, written as an exclamation mark `!`, attempts to match its inner expression. If the inner expression *fails*, the predicate *succeeds* and parsing continues at the same position as the predicate. If the inner expression *succeeds*, the predicate *fails* — `!foo ~ bar` is thus a kind of "NOT" statement: "the input string must match `bar` but NOT `foo`".

This leads to the common idiom meaning "any character but":

```
not_space_or_tab = {
    !(                  // if the following text is not
        " "             //     a space
        | "\t"          //     or a tab
    )
    ~ ANY               // then consume one character
}

triple_quoted_string = {
    """"
    ~ triple_quoted_character*
    ~ """"
}
triple_quoted_character = {
    !""""         // if the following text is not three apostrophes
    ~ ANY         // then consume one character
}
```

Operator precedence and grouping (WIP)

The repetition operators asterisk `*`, plus sign `+`, and question mark `?` apply to the immediately preceding expression.

```
"One " ~ "or " ~ "more. "+
"One " ~ "or " ~ ("more. "+)
    are equivalent and match
"One or more. more. more. more. "
```

Larger expressions can be repeated by surrounding them with parentheses.

```
("One " ~ "or " ~ "more. ")+
    matches
"One or more. One or more. "
```

Repetition operators have the highest precedence, followed by predicate operators, the sequence operator, and finally ordered choice.

```
my_rule = {
    "a"* ~ "b"?
    | &"b"+ ~ "a"
}

// equivalent to

my_rule = {
      ( ("a"*) ~ ("b"?) )
    | ( (&("b"+)) ~ "a" )
}
```

Start and end of input

The rules `SOI` and `EOI` match the *start* and *end* of the input string, respectively. Neither consumes any text. They only indicate whether the parser is currently at one edge of the input.

For example, to ensure that a rule matches the entire input, where any syntax error results in a failed parse (rather than a successful but incomplete parse):

```
main = {
    SOI
    ~ (...)
    ~ EOI
}
```

Implicit whitespace

Many languages and text formats allow arbitrary whitespace and comments between logical tokens. For instance, Rust considers `4+5` equivalent to `4 + 5` and `4 /* comment */ + 5`.

The **optional rules** `WHITESPACE` **and** `COMMENT` implement this behaviour. If either (or both) are defined, they will be implicitly inserted at every sequence and between every repetition (except in atomic rules).

```
expression = { "4" ~ "+" ~ "5" }
WHITESPACE = _{ " " }
COMMENT = _{ "/*" ~ (!"*/" ~ ANY)* ~ "*/" }
```

```
"4+5"
"4 + 5"
"4  +     5"
"4 /* comment */ + 5"
```

As you can see, `WHITESPACE` and `COMMENT` are run repeatedly, so they need only match a single whitespace character or a single comment. The grammar above is equivalent to:

```
expression = {
    "4"   ~ (ws | com)*
    ~ "+" ~ (ws | com)*
    ~ "5"
}
ws = _{ " " }
com = _{ "/*" ~ (!"*/" ~ ANY)* ~ "*/" }
```

Note that implicit whitespace is *not* inserted at the beginning or end of rules — for instance, `expression` does *not* match `" 4+5 "`. If you want to include implicit whitespace at the beginning and end of a rule, you will need to sandwich it between two empty rules (often `SOI` and `EOI` as above):

```
WHITESPACE = _{ " " }
expression = { "4" ~ "+" ~ "5" }
main = { SOI ~ expression ~ EOI }

"4+5"
"  4 + 5   "
```

(Be sure to mark the `WHITESPACE` and `COMMENT` rules as silent unless you want to see them included inside other rules!)

Silent and atomic rules

Silent rules are just like normal rules — when run, they function the same way — except they do not produce pairs or tokens. If a rule is silent, it will never appear in a parse result.

To make a silent rule, precede the left curly bracket `{` with a low line (underscore) `_`.

```
silent = _{ ... }
```

Atomic

`pest` has two kinds of atomic rules: **atomic** and **compound atomic**. To make one, write the sigil before the left curly bracket `{`.

```
atomic = @{ ... }
compound_atomic = ${ ... }
```

Both kinds of atomic rule prevent implicit whitespace: inside an atomic rule, the tilde `~` means "immediately followed by", and repetition operators (asterisk `*` and plus sign `+`) have no implicit separation. In addition, all other rules called from an atomic rule are also treated as atomic.

The difference between the two is how they produce tokens for inner rules. In an atomic rule, interior matching rules are silent. By contrast, compound atomic rules produce inner tokens as normal.

Atomic rules are useful when the text you are parsing ignores whitespace except in a few cases, such as literal strings. In this instance, you can write `WHITESPACE` or `COMMENT` rules, then make your string-matching rule be atomic.

Non-atomic

Sometimes, you'll want to cancel the effects of atomic parsing. For instance, you might want to have string interpolation with an expression inside, where the inside expression can still have whitespace like normal.

```
#!/bin/env python3
print(f"The answer is {2 + 4}.")
```

This is where you use a **non-atomic** rule. Write an exclamation mark `!` in front of the defining curly bracket. The rule will run as non-atomic, whether it is called from an atomic rule or not.

```
fstring = @{ "\"" ~ ... }
expr = !{ ... }
```

The stack (WIP)

`pest` maintains a stack that can be manipulated directly from the grammar. An expression can be matched and pushed onto the stack with the keyword `PUSH`, then later matched exactly with the keywords `PEEK` and `POP`.

Using the stack allows *the exact same text* to be matched multiple times, rather than *the same pattern*.

For example,

```
same_text = {
    PUSH( "a" | "b" | "c" )
    ~ POP
}
same_pattern = {
    ("a" | "b" | "c")
    ~ ("a" | "b" | "c")
}
```

In this case, `same_pattern` will match `"ab"`, while `same_text` will not.

One practical use is in parsing Rust "raw string literals", which look like this:

```
const raw_str: &str = r###"
    Some number of number signs # followed by a quotation mark ".

    Quotation marks can be used anywhere inside: """""""""",
    as long as one is not followed by a matching number of number signs,
    which ends the string: "###;
```

When parsing a raw string, we have to keep track of how many number signs `#` occurred before the quotation mark. We can do this using the stack:

```
raw_string = {
    "r" ~ PUSH("#"*) ~ "\""     // push the number signs onto the stack
    ~ raw_string_interior
    ~ "\"" ~ POP                // match a quotation mark and the number
signs
}
raw_string_interior = {
    (
        !("\"" ~ PEEK)     // unless the next character is a quotation
mark
                           // followed by the correct amount of number
signs,
        ~ ANY              // consume one character
    )*
}
```

Cheat sheet

Syntax	Meaning	Syntax	Meaning
`foo = { ... }`	regular rule	`baz = @{ ... }`	atomic
`bar = _{ ... }`	silent	`qux = ${ ... }`	compound-atomic
		`plugh = !{ ... }`	non-atomic
`"abc"`	exact string	`^"abc"`	case insensitive
`'a'..'z'`	character range	`ANY`	any character
`foo ~ bar`	sequence	`baz | qux`	ordered choice
`foo*`	zero or more	`bar+`	one or more
`baz?`	optional	`qux{n}`	exactly *n*
`qux{m, n}`	between *m* and *n* (inclusive)		
`&foo`	positive predicate	`!bar`	negative predicate
`PUSH(baz)`	match and push		
`POP`	match and pop	`PEEK`	match without pop

Built-in rules

Besides `ANY`, matching any single Unicode character, `pest` provides several rules to make parsing text more convenient.

ASCII rules

Among the printable ASCII characters, it is often useful to match alphabetic characters and numbers. For **numbers**, `pest` provides digits in common radixes (bases):

Built-in rule	Equivalent
`ASCII_DIGIT`	`'0'..'9'`
`ASCII_NONZERO_DIGIT`	`'1'..'9'`
`ASCII_BIN_DIGIT`	`'0'..'1'`
`ASCII_OCT_DIGIT`	`'0'..'7'`
`ASCII_HEX_DIGIT`	`'0'..'9' | 'a'..'f' | 'A'..'F'`

For **alphabetic** characters, distinguishing between uppercase and lowercase:

Built-in rule	Equivalent
`ASCII_ALPHA_LOWER`	`'a'..'z'`
`ASCII_ALPHA_UPPER`	`'A'..'Z'`
`ASCII_ALPHA`	`'a'..'z' | 'A'..'Z'`

And for **miscellaneous** use:

Built-in rule	Meaning	Equivalent
`ASCII_ALPHANUMERIC`	any digit or letter	`ASCII_DIGIT | ASCII_ALPHA`
`NEWLINE`	any line feed format	`"\n" | "\r\n" | "\r"`

Unicode rules

To make it easier to correctly parse arbitrary Unicode text, `pest` includes a large number of rules corresponding to Unicode character properties. These rules are divided into **general category** and **binary property** rules.

Unicode characters are partitioned into categories based on their general purpose. Every character belongs to a single category, in the same way that every ASCII character is a control character, a digit, a letter, a symbol, or a space.

In addition, every Unicode character has a list of binary properties (true or false) that it does or does not satisfy. Characters can belong to any number of these properties, depending on their meaning.

For example, the character "A", "Latin capital letter A", is in the general category "Uppercase Letter" because its general purpose is being a letter. It has the binary property "Uppercase" but not "Emoji". By contrast, the character "🅰", "negative squared Latin capital letter A", is in the general category "Other Symbol" because it does not generally occur as a letter in text. It has both the binary properties "Uppercase" and "Emoji".

For more details, consult Chapter 4 of The Unicode Standard.

General categories

Formally, categories are non-overlapping: each Unicode character belongs to exactly one category, and no category contains another. However, since certain groups of categories are often useful together, `pest` exposes the hierarchy of categories below. For example, the rule `CASED_LETTER` is not technically a Unicode general category; it instead matches characters that are `UPPERCASE_LETTER` or `LOWERCASE_LETTER`, which *are* general categories.

- `LETTER`
  - `CASED_LETTER`
    - `UPPERCASE_LETTER`
    - `LOWERCASE_LETTER`
  - `TITLECASE_LETTER`
  - `MODIFIER_LETTER`
  - `OTHER_LETTER`
- `MARK`
  - `NONSPACING_MARK`

    - SPACING_MARK
    - ENCLOSING_MARK
- NUMBER
    - DECIMAL_NUMBER
    - LETTER_NUMBER
    - OTHER_NUMBER
- PUNCTUATION
    - CONNECTOR_PUNCTUATION
    - DASH_PUNCTUATION
    - OPEN_PUNCTUATION
    - CLOSE_PUNCTUATION
    - INITIAL_PUNCTUATION
    - FINAL_PUNCTUATION
    - OTHER_PUNCTUATION
- SYMBOL
    - MATH_SYMBOL
    - CURRENCY_SYMBOL
    - MODIFIER_SYMBOL
    - OTHER_SYMBOL
- SEPARATOR
    - SPACE_SEPARATOR
    - LINE_SEPARATOR
    - PARAGRAPH_SEPARATOR
- OTHER
    - CONTROL
    - FORMAT
    - SURROGATE
    - PRIVATE_USE
    - UNASSIGNED

Binary properties

Many of these properties are used to define Unicode text algorithms, such as the bidirectional algorithm and the text segmentation algorithm. Such properties are not likely to be useful for most parsers.

However, the properties `XID_START` and `XID_CONTINUE` are particularly notable because they are defined "to assist in the standard treatment of identifiers", "such as programming language variables". See Technical Report 31 for more details.

- ALPHABETIC
- BIDI_CONTROL
- CASE_IGNORABLE
- CASED
- CHANGES_WHEN_CASEFOLDED
- CHANGES_WHEN_CASEMAPPED
- CHANGES_WHEN_LOWERCASED
- CHANGES_WHEN_TITLECASED
- CHANGES_WHEN_UPPERCASED
- DASH
- DEFAULT_IGNORABLE_CODE_POINT
- DEPRECATED
- DIACRITIC
- EXTENDER
- GRAPHEME_BASE
- GRAPHEME_EXTEND
- GRAPHEME_LINK
- HEX_DIGIT
- HYPHEN
- IDS_BINARY_OPERATOR
- IDS_TRINARY_OPERATOR
- ID_CONTINUE
- ID_START
- IDEOGRAPHIC
- JOIN_CONTROL
- LOGICAL_ORDER_EXCEPTION
- LOWERCASE
- MATH
- NONCHARACTER_CODE_POINT
- OTHER_ALPHABETIC
- OTHER_DEFAULT_IGNORABLE_CODE_POINT
- OTHER_GRAPHEME_EXTEND
- OTHER_ID_CONTINUE
- OTHER_ID_START
- OTHER_LOWERCASE
- OTHER_MATH
- OTHER_UPPERCASE
- PATTERN_SYNTAX
- PATTERN_WHITE_SPACE

- PREPENDED_CONCATENATION_MARK
- QUOTATION_MARK
- RADICAL
- REGIONAL_INDICATOR
- SENTENCE_TERMINAL
- SOFT_DOTTED
- TERMINAL_PUNCTUATION
- UNIFIED_IDEOGRAPH
- UPPERCASE
- VARIATION_SELECTOR
- WHITE_SPACE
- XID_CONTINUE
- XID_START

Example: JSON

JSON is a popular format for data serialization that is derived from the syntax of JavaScript. JSON documents are tree-like and potentially recursive — two data types, *objects* and *arrays*, can contain other values, including other objects and arrays.

Here is an example JSON document:

```
{
    "nesting": { "inner object": {} },
    "an array": [1.5, true, null, 1e-6],
    "string with escaped double quotes" : "\"quick brown foxes\""
}
```

Let's write a program that **parses** the JSON to an Rust object, known as an *abstract syntax tree*, then **serializes** the AST back to JSON.

Setup

We'll start by defining the AST in Rust. Each JSON data type is represented by an enum variant.

```
enum JSONValue<'a> {
    Object(Vec<(&'a str, JSONValue<'a>)>),
    Array(Vec<JSONValue<'a>>),
    String(&'a str),
    Number(f64),
    Boolean(bool),
    Null,
}
```

To avoid copying when deserializing strings, `JSONValue` borrows strings from the original unparsed JSON. In order for this to work, we cannot interpret string escape sequences: the input string `"\n"` will be represented by `JSONValue::String("\\n")`, a Rust string with two characters, even though it represents a JSON string with just one character.

Let's move on to the serializer. For the sake of clarity, it uses allocated `String`s instead of providing an implementation of `std::fmt::Display`, which would be more idiomatic.

```
fn serialize_jsonvalue(val: &JSONValue) -> String {
    use JSONValue::*;

    match val {
        Object(o) => {
            let contents: Vec<_> = o
                .iter()
                .map(|(name, value)|
                     format!("\"{}\":{}", name,
serialize_jsonvalue(value)))
                .collect();
            format!("{{{}}}", contents.join(","))
        }
        Array(a) => {
            let contents: Vec<_> =
a.iter().map(serialize_jsonvalue).collect();
            format!("[{}]", contents.join(","))
        }
        String(s) => format!("\"{}\"", s),
        Number(n) => format!("{}", n),
        Boolean(b) => format!("{}", b),
        Null => format!("null"),
    }
}
```

Note that the function invokes itself recursively in the `Object` and `Array` cases.

This pattern appears throughout the parser. The AST creation function iterates recursively through the parse result, and the grammar has rules which include themselves.

Writing the grammar

Let's begin with whitespace. JSON whitespace can appear anywhere, except inside strings (where it must be parsed separately) and between digits in numbers (where it is not allowed). This makes it a good fit for `pest`'s implicit whitespace. In `src/json.pest`:

```
WHITESPACE = _{ " " | "\t" | "\r" | "\n" }
```

The JSON specification includes diagrams for parsing JSON strings. We can write the grammar directly from that page. Let's write `object` as a sequence of `pair`s separated by commas `,`.

```
object = {
    "{" ~ "}" |
    "{" ~ pair ~ ("," ~ pair)* ~ "}"
}
pair = { string ~ ":" ~ value }

array = {
    "[" ~ "]" |
    "[" ~ value ~ ("," ~ value)* ~ "]"
}
```

The `object` and `array` rules show how to parse a potentially empty list with separators. There are two cases: one for an empty list, and one for a list with at least one element. This is necessary because a trailing comma in an array, such as in `[0, 1,]`, is illegal in JSON.

Now we can write `value`, which represents any single data type. We'll mimic our AST by writing `boolean` and `null` as separate rules.

```
value = _{ object | array | string | number | boolean | null }

boolean = { "true" | "false" }

null = { "null" }
```

Let's separate the logic for strings into three parts. `char` is a rule matching any logical character in the string, including any backslash escape sequence. `inner` represents the contents of the string, without the surrounding double quotes. `string` matches the inner contents of the string, including the surrounding double quotes.

The `char` rule uses the idiom `!(...) ~ ANY`, which matches any character except the ones given in parentheses. In this case, any character is legal inside a string, except for double quote `"` and backslash `\`, which require separate parsing logic.

```
string = ${ "\"" ~ inner ~ "\"" }
inner = @{ char* }
char = {
    !("\"" | "\\") ~ ANY
    | "\\" ~ ("\"" | "\\" | "/" | "b" | "f" | "n" | "r" | "t")
    | "\\" ~ ("u" ~ ASCII_HEX_DIGIT{4})
}
```

Because `string` is marked compound atomic, `string` token pairs will also contain a single `inner` pair. Because `inner` is marked atomic, no `char` pairs will appear inside `inner`. Since these rules are atomic, no whitespace is permitted between separate tokens.

Numbers have four logical parts: an optional sign, an integer part, an optional fractional part, and an optional exponent. We'll mark `number` atomic so that whitespace cannot appear between its parts.

```
number = @{
    "-"?
    ~ ("0" | ASCII_NONZERO_DIGIT ~ ASCII_DIGIT*)
    ~ ("." ~ ASCII_DIGIT*)?
    ~ (^"e" ~ ("+" | "-")? ~ ASCII_DIGIT+)?
}
```

We need a final rule to represent an entire JSON file. The only legal contents of a JSON file is a single object or array. We'll mark this rule silent, so that a parsed JSON file contains only two token pairs: the parsed value itself, and the `EOI` rule.

```
json = _{ SOI ~ (object | array) ~ EOI }
```

AST generation

Let's compile the grammar into Rust.

```
extern crate pest;
#[macro_use]
extern crate pest_derive;

use pest::Parser;

#[derive(Parser)]
#[grammar = "json.pest"]
struct JSONParser;
```

We'll write a function that handles both parsing and AST generation. Users of the function can call it on an input string, then use the result returned as either a `JSONValue` or a parse error.

```
use pest::error::Error;

fn parse_json_file(file: &str) -> Result<JSONValue, Error<Rule>> {
    let json = JSONParser::parse(Rule::json, file)?.next().unwrap();

    // ...
}
```

Now we need to handle `Pair`s recursively, depending on the rule. We know that `json` is either an `object` or an `array`, but these values might contain an `object` or an `array` themselves! The most logical way to handle this is to write an auxiliary recursive function that parses a `Pair` into a `JSONValue` directly.

```
fn parse_json_file(file: &str) -> Result<JSONValue, Error<Rule>> {
    // ...

    use pest::iterators::Pair;

    fn parse_value(pair: Pair<Rule>) -> JSONValue {
        match pair.as_rule() {
            Rule::object => JSONValue::Object(
                pair.into_inner()
                    .map(|pair| {
                        let mut inner_rules = pair.into_inner();
                        let name = inner_rules
                            .next()
                            .unwrap()
                            .into_inner()
                            .next()
                            .unwrap()
                            .as_str();
                        let value =
parse_value(inner_rules.next().unwrap());
                        (name, value)
                    })
                    .collect(),
            ),
            Rule::array =>
JSONValue::Array(pair.into_inner().map(parse_value).collect()),
            Rule::string =>
JSONValue::String(pair.into_inner().next().unwrap().as_str()),
            Rule::number =>
JSONValue::Number(pair.as_str().parse().unwrap()),
            Rule::boolean =>
JSONValue::Boolean(pair.as_str().parse().unwrap()),
            Rule::null => JSONValue::Null,
            Rule::json
            | Rule::EOI
            | Rule::pair
            | Rule::value
            | Rule::inner
            | Rule::char
            | Rule::WHITESPACE => unreachable!(),
        }
    }

    // ...
}
```

The `object` and `array` cases deserve special attention. The contents of an `array` token pair is just a sequence of `value` s. Since we're working with a Rust iterator,

we can simply map each value to its parsed AST node recursively, then collect them into a `Vec`. For `object`s, the process is similar, except the iterator is over `pair`s, from which we need to extract names and values separately.

The `number` and `boolean` cases use Rust's `str::parse` method to convert the parsed string to the appropriate Rust type. Every legal JSON number can be parsed directly into a Rust floating point number!

We run `parse_value` on the parse result to finish the conversion.

```
fn parse_json_file(file: &str) -> Result<JSONValue, Error<Rule>> {
    // ...

    Ok(parse_value(json))
}
```

Finishing

Our `main` function is now very simple. First, we read the JSON data from a file named `data.json`. Next, we parse the file contents into a JSON AST. Finally, we serialize the AST back into a string and print it.

```
use std::fs;

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

    let json: JSONValue =
parse_json_file(&unparsed_file).expect("unsuccessful parse");

    println!("{}", serialize_jsonvalue(&json));
}
```

Try it out! Copy the example document at the top of this chapter into `data.json`, then run the program! You should see something like this:

```
$ cargo run
  [ ... ]
{"nesting":{"inner object":{}},"an array":
[1.5,true,null,0.000001],"string with escaped double quotes":"\"quick
brown foxes\""}
```

Example: The J language

The J language is an array programming language influenced by APL. In J, operations on individual numbers (`2 * 3`) can just as easily be applied to entire lists of numbers (`2 * 3 4 5` , returning `6 8 10`).

Operators in J are referred to as *verbs*. Verbs are either *monadic* (taking a single argument, such as `*: 3` , "3 squared") or *dyadic* (taking two arguments, one on either side, such as `5 - 4` , "5 minus 4").

Here's an example of a J program:

```
'A string'

*: 1 2 3 4

matrix =: 2 3 $ 5 + 2 3 4 5 6 7
10 * matrix

1 + 10 20 30
1 2 3 + 10

residues =: 2 | 0 1 2 3 4 5 6 7
residues
```

Using J's interpreter to run the above program yields the following on standard out:

```
A string

1 4 9 16

 70  80  90
100 110 120

11 21 31
11 12 13

0 1 0 1 0 1 0 1
```

In this section we'll write a grammar for a subset of J. We'll then walk through a parser that builds an AST by iterating over the rules that `pest` gives us. You can find the full source code within this book's repository.

The grammar

We'll build up a grammar section by section, starting with the program rule:

```
program = _{ SOI ~ "\n"* ~ (stmt ~ "\n"+) * ~ stmt? ~ EOI }
```

Each J program contains statements delimited by one or more newlines. Notice the leading underscore, which tells `pest` to silence the `program` rule — we don't want `program` to appear as a token in the parse stream, we want the underlying statements instead.

A statement is simply an expression, and since there's only one such possibility, we also silence this `stmt` rule as well, and thus our parser will receive an iterator of underlying `expr` s:

```
stmt = _{ expr }
```

An expression can be an assignment to a variable identifier, a monadic expression, a dyadic expression, a single string, or an array of terms:

```
expr = {
      assgmtExpr
    | monadicExpr
    | dyadicExpr
    | string
    | terms
}
```

A monadic expression consists of a verb with its sole operand on the right; a dyadic expression has operands on either side of the verb. Assignment expressions associate identifiers with expressions.

In J, there is no operator precedence — evaluation is right-associative (proceeding from right to left), with parenthesized expressions evaluated first.

```
monadicExpr = { verb ~ expr }

dyadicExpr = { (monadicExpr | terms) ~ verb ~ expr }

assgmtExpr = { ident ~ "=:" ~ expr }
```

A list of terms should contain at least one decimal, integer, identifier, or parenthesized expression; we care only about those underlying values, so we make the `term` rule silent with a leading underscore:

```
terms = { term+ }

term = _{ decimal | integer | ident | "(" ~ expr ~ ")" }
```

A few of J's verbs are defined in this grammar; J's full vocabulary is much more extensive.

```
verb = {
    ">:" | "*:" | "-"  | "%" | "#" | ">."
  | "+"  | "*"  | "<"  | "=" | "^" | "|"
  | ">"  | "$"
}
```

Now we can get into lexing rules. Numbers in J are represented as usual, with the exception that negatives are represented using a leading `_` underscore (because `-` is a verb that performs negation as a monad and subtraction as a dyad). Identifiers in J must start with a letter, but can contain numbers thereafter. Strings are surrounded by single quotes; quotes themselves can be embedded by escaping them with an additional quote.

Notice how we use `pest`'s `@` modifier to make each of these rules atomic, meaning implicit whitespace is forbidden, and that interior rules (i.e., `ASCII_ALPHA` in `ident`) become silent — when our parser receives any of these tokens, they will be terminal:

```
integer = @{ "_"? ~ ASCII_DIGIT+ }

decimal = @{ "_"? ~ ASCII_DIGIT+ ~ "." ~ ASCII_DIGIT* }

ident = @{ ASCII_ALPHA ~ (ASCII_ALPHANUMERIC | "_")* }

string = @{ "'" ~ ( "''" | (!"'" ~ ANY) )* ~ "'" }
```

Whitespace in J consists solely of spaces and tabs. Newlines are significant because they delimit statements, so they are excluded from this rule:

```
WHITESPACE = _{ " " | "\t" }
```

Finally, we must handle comments. Comments in J start with `NB.` and continue to the end of the line on which they are found. Critically, we must not consume the newline at the end of the comment line; this is needed to separate any statement that might precede the comment from the statement on the succeeding line.

```
COMMENT = _{ "NB." ~ (!"\n" ~ ANY)* }
```

Parsing and AST generation

This section will walk through a parser that uses the grammar above. Library includes and self-explanatory code are omitted here; you can find the parser in its entirety within this book's repository.

First we'll enumerate the verbs defined in our grammar, distinguishing between monadic and dyadic verbs. These enumerations will be be used as labels in our AST:

```
pub enum MonadicVerb {
    Increment,
    Square,
    Negate,
    Reciprocal,
    Tally,
    Ceiling,
    ShapeOf,
}

pub enum DyadicVerb {
    Plus,
    Times,
    LessThan,
    LargerThan,
    Equal,
    Minus,
    Divide,
    Power,
    Residue,
    Copy,
    LargerOf,
    LargerOrEqual,
    Shape,
}
```

Then we'll enumerate the various kinds of AST nodes:

```
pub enum AstNode {
    Print(Box<AstNode>),
    Integer(i32),
    DoublePrecisionFloat(f64),
    MonadicOp {
        verb: MonadicVerb,
        expr: Box<AstNode>,
    },
    DyadicOp {
        verb: DyadicVerb,
        lhs: Box<AstNode>,
        rhs: Box<AstNode>,
    },
    Terms(Vec<AstNode>),
    IsGlobal {
        ident: String,
        expr: Box<AstNode>,
    },
    Ident(String),
    Str(CString),
}
```

To parse top-level statements in a J program, we have the following `parse` function that accepts a J program in string form and passes it to `pest` for parsing. We get back a sequence of `Pair`s. As specified in the grammar, a statement can only consist of an expression, so the `match` below parses each of those top-level expressions and wraps them in a `Print` AST node in keeping with the J interpreter's REPL behavior:

```
pub fn parse(source: &str) -> Result<Vec<AstNode>, Error<Rule>> {
    let mut ast = vec![];

    let pairs = JParser::parse(Rule::program, source)?;
    for pair in pairs {
        match pair.as_rule() {
            Rule::expr => {
                ast.push(Print(Box::new(build_ast_from_expr(pair))));
            }
            _ => {}
        }
    }

    Ok(ast)
}
```

AST nodes are built from expressions by walking the `Pair` iterator in lockstep with the expectations set out in our grammar file. Common behaviors are abstracted out into separate functions, such as `parse_monadic_verb` and `parse_dyadic_verb`, and `Pair`s representing expressions themselves are passed in recursive calls to `build_ast_from_expr`:

```
fn build_ast_from_expr(pair: pest::iterators::Pair<Rule>) -> AstNode {
    match pair.as_rule() {
        Rule::expr =>
build_ast_from_expr(pair.into_inner().next().unwrap()),
        Rule::monadicExpr => {
            let mut pair = pair.into_inner();
            let verb = pair.next().unwrap();
            let expr = pair.next().unwrap();
            let expr = build_ast_from_expr(expr);
            parse_monadic_verb(verb, expr)
        }
        // ... other cases elided here ...
    }
}
```

Dyadic verbs are mapped from their string representations to AST nodes in a straightforward way:

```
fn parse_dyadic_verb(pair: pest::iterators::Pair<Rule>, lhs: AstNode,
rhs: AstNode) -> AstNode {
    AstNode::DyadicOp {
        lhs: Box::new(lhs),
        rhs: Box::new(rhs),
        verb: match pair.as_str() {
            "+" => DyadicVerb::Plus,
            "*" => DyadicVerb::Times,
            "-" => DyadicVerb::Minus,
            "<" => DyadicVerb::LessThan,
            "=" => DyadicVerb::Equal,
            ">" => DyadicVerb::LargerThan,
            "%" => DyadicVerb::Divide,
            "^" => DyadicVerb::Power,
            "|" => DyadicVerb::Residue,
            "#" => DyadicVerb::Copy,
            ">." => DyadicVerb::LargerOf,
            ">:" => DyadicVerb::LargerOrEqual,
            "$" => DyadicVerb::Shape,
            _ => panic!("Unexpected dyadic verb: {}", pair.as_str()),
        },
    }
}
```

As are monadic verbs:

```
fn parse_monadic_verb(pair: pest::iterators::Pair<Rule>, expr: AstNode)
-> AstNode {
    AstNode::MonadicOp {
        verb: match pair.as_str() {
            ">:" => MonadicVerb::Increment,
            "*:" => MonadicVerb::Square,
            "-" => MonadicVerb::Negate,
            "%" => MonadicVerb::Reciprocal,
            "#" => MonadicVerb::Tally,
            ">." => MonadicVerb::Ceiling,
            "$" => MonadicVerb::ShapeOf,
            _ => panic!("Unsupported monadic verb: {}", pair.as_str()),
        },
        expr: Box::new(expr),
    }
}
```

Finally, we define a function to process terms such as numbers and strings. Numbers require some manuevering to handle J's leading underscores representing negation, but other than that the process is typical:

```
fn build_ast_from_term(pair: pest::iterators::Pair<Rule>) -> AstNode {
    match pair.as_rule() {
        Rule::integer => {
            let istr = pair.as_str();
            let (sign, istr) = match &istr[..1] {
                "_" => (-1, &istr[1..]),
                _ => (1, &istr[..]),
            };
            let integer: i32 = istr.parse().unwrap();
            AstNode::Integer(sign * integer)
        }
        Rule::decimal => {
            let dstr = pair.as_str();
            let (sign, dstr) = match &dstr[..1] {
                "_" => (-1.0, &dstr[1..]),
                _ => (1.0, &dstr[..]),
            };
            let mut flt: f64 = dstr.parse().unwrap();
            if flt != 0.0 {
                // Avoid negative zeroes; only multiply sign by
nonzeroes.
                flt *= sign;
            }
            AstNode::DoublePrecisionFloat(flt)
        }
        Rule::expr => build_ast_from_expr(pair),
        Rule::ident => AstNode::Ident(String::from(pair.as_str())),
        unknown_term => panic!("Unexpected term: {:?}", unknown_term),
    }
}
```

Running the Parser

We can now define a `main` function to pass J programs to our `pest`-enabled parser:

```
fn main() {
    let unparsed_file = std::fs::read_to_string("example.ijs")
      .expect("cannot read ijs file");
    let astnode = parse(&unparsed_file).expect("unsuccessful parse");
    println!("{:?}", &astnode);
}
```

Using this code in `example.ijs`:

```
_2.5 ^ 3
*: 4.8
title =: 'Spinning at the Boundary'
*: _1 2 _3 4
1 2 3 + 10 20 30
1 + 10 20 30
1 2 3 + 10
2 | 0 1 2 3 4 5 6 7
another =: 'It''s Escaped'
3 | 0 1 2 3 4 5 6 7
(2+1)*(2+2)
3 * 2 + 1
1 + 3 % 4
x =: 100
x - 1
y =: x - 1
y
```

We'll get the following abstract syntax tree on stdout when we run the parser:

```
$ cargo run
  [ ... ]
[Print(DyadicOp { verb: Power, lhs: DoublePrecisionFloat(-2.5),
    rhs: Integer(3) }),
Print(MonadicOp { verb: Square, expr: DoublePrecisionFloat(4.8) }),
Print(IsGlobal { ident: "title", expr: Str("Spinning at the Boundary")
}),
Print(MonadicOp { verb: Square, expr: Terms([Integer(-1), Integer(2),
    Integer(-3), Integer(4)]) }),
Print(DyadicOp { verb: Plus, lhs: Terms([Integer(1), Integer(2),
Integer(3)]),
    rhs: Terms([Integer(10), Integer(20), Integer(30)]) }),
Print(DyadicOp { verb: Plus, lhs: Integer(1), rhs: Terms([Integer(10),
    Integer(20), Integer(30)]) }),
Print(DyadicOp { verb: Plus, lhs: Terms([Integer(1), Integer(2),
Integer(3)]),
    rhs: Integer(10) }),
Print(DyadicOp { verb: Residue, lhs: Integer(2),
    rhs: Terms([Integer(0), Integer(1), Integer(2), Integer(3),
Integer(4),
    Integer(5), Integer(6), Integer(7)]) }),
Print(IsGlobal { ident: "another", expr: Str("It\'s Escaped") }),
Print(DyadicOp { verb: Residue, lhs: Integer(3), rhs: Terms([Integer(0),
    Integer(1), Integer(2), Integer(3), Integer(4), Integer(5),
    Integer(6), Integer(7)]) }),
Print(DyadicOp { verb: Times, lhs: DyadicOp { verb: Plus, lhs:
Integer(2),
    rhs: Integer(1) }, rhs: DyadicOp { verb: Plus, lhs: Integer(2),
        rhs: Integer(2) } }),
Print(DyadicOp { verb: Times, lhs: Integer(3), rhs: DyadicOp { verb:
Plus,
    lhs: Integer(2), rhs: Integer(1) } }),
Print(DyadicOp { verb: Plus, lhs: Integer(1), rhs: DyadicOp { verb:
Divide,
    lhs: Integer(3), rhs: Integer(4) } }),
Print(IsGlobal { ident: "x", expr: Integer(100) }),
Print(DyadicOp { verb: Minus, lhs: Ident("x"), rhs: Integer(1) }),
Print(IsGlobal { ident: "y", expr: DyadicOp { verb: Minus, lhs:
Ident("x"),
    rhs: Integer(1) } }),
Print(Ident("y"))]
```

Operator precedence (WIP)

This chapter will discuss two methods of dealing with operator precedence: directly in the PEG grammar, and using a `PrecClimber`. It will probably also include an explanation of how precedence climbing works.

Example: Calculator (WIP)

This section will walk through the creation of a simple calculator. It will provide an example of parsing expressions with operator precedence.

Final project: Awk clone (WIP)

This chapter will walk through the creation of a simple variant of Awk (only loosely following the POSIX specification). It will probably have several sections. It will provide an example of a full project based on `pest` with a manageable grammar, a straightforward AST, and a fairly simple interpreter.

This Awk clone will support regex patterns, string and numeric variables, most of the POSIX operators, and some functions. It will not support user-defined functions in the interest of avoiding variable scoping.