Simple but Powerful Pratt Parsing

Apr 13, 2020

Welcome to my article about Pratt parsing — the monad tutorial of syntactic analysis. The number of Pratt parsing articles is so large that there exists a survey post :)

The goals of this particular article are:

- Raising an issue that the so-called left-recursion problem is overstated.
- Complaining about inadequacy of BNF for representing infix expressions.
- Providing a description and implementation of Pratt parsing algorithm which sticks to the core and doesn't introduce a DSL-y abstraction.
- Understanding the algorithm myself for hopefully the last time. I've implemented a production-grade Pratt parser once, but I no longer immediately understand that code :-)

This post assumes a fair bit of familiarity with parsing techniques, and, for example, does not explain what a context free grammar is.

Introduction

Parsing is the process by which a compiler turns a *sequence* of tokens into a *tree* representation:

```
                            Add
                 Parser     / \
"1 + 2 * 3"    ------->   1  Mul
                            / \
                           2   3
```

There are many approaches to this task, which roughly fall into one of the broad two categories:

- Using a DSL to specify an abstract grammar of the language
- Hand-writing the parser

Pratt parsing is one of the most frequently used techniques for hand-written parsing.

BNF

The pinnacle of syntactic analysis theory is discovering the context free grammar notation (often using BNF concrete syntax) for decoding linear structures into trees:

```
Item =
    StructItem
  | EnumItem
  | ...

StructItem =
    'struct' Name '{' FieldList '}'

...
```

I remember being fascinated by this idea, especially by parallels with natural language sentence structure. However, my optimism quickly waned once we got to describing expressions. The natural expression grammar indeed allows one to see what is an expression.

```
Expr =
    Expr '+' Expr
  | Expr '*' Expr
  | '(' Expr ')'
  | 'number'
```

Although this grammar looks great, it is in fact ambiguous and imprecise, and needs to be rewritten to be amendable to automated parser generation. Specifically, we need to specify precedence and associativity of operators. The fixed grammar looks like this:

```
Expr =
    Factor
  | Expr '+' Factor

Factor =
    Atom
  | Factor '*' Atom

Atom =
    'number'
  | '(' Expr ')'
```

To me, the "shape" of expressions feels completely lost in this new formulation. Moreover, it took me three or four *courses* in formal languages before I was able to reliably create this grammar myself.

And that's why I love Pratt parsing — it is an enhancement of recursive descent parsing algorithm, which uses the natural terminology of precedence and associativity for parsing expressions, instead of grammar obfuscation techniques.

Recursive descent and left-recursion

The simplest technique for hand-writing a parser is recursive descent, which models the grammar as a set of mutually recursive functions. For example, the above item grammar fragment can look like this:

```
fn item(p: &mut Parser) {
    match p.peek() {
        STRUCT_KEYWORD => struct_item(p),
        ENUM_KEYWORD   => enum_item(p),
        ...
    }
}

fn struct_item(p: &mut Parser) {
    p.expect(STRUCT_KEYWORD);
    name(p);
    p.expect(L_CURLY);
    field_list(p);
    p.expect(R_CURLY);
}

...
```

Traditionally, text-books point out left-recursive grammars as the Achilles heel of this approach, and use this drawback to motivate more advanced LR parsing techniques. An example of problematic grammar can look like this:

```
Sum =
    Sum '+' Int
  | Int
```

Indeed, if we naively code the `sum` function, it wouldn't be too useful:

```
fn sum(p: &mut Parser) {
    // Try first alternative
    sum(p); ❶
    p.expect(PLUS);
    int(p);

    // If that fails, try the second one
    ...
}
```

❶ At this point we immediately loop and overflow the stack

A theoretical fix to the problem involves rewriting the grammar to eliminate the left recursion. However in practice, for a hand-written parser, a solution is much simpler — breaking away with a pure *recursive* paradigm and using a loop:

```
fn sum(p: &mut Parser) {
    int(p);
    while p.eat(PLUS) {
        int(p);
    }
}
```

Pratt parsing, the general shape

Using just loops won't be enough for parsing infix expressions. Instead, Pratt parsing uses *both* loops *and* recursion:

```
fn parse_expr() {
    ...
    loop {
        ...
        parse_expr()
        ...
    }
}
```

Not only does it send your mind into Möbeus-shaped hamster wheel, it also handles associativity and precedence!

From Precedence to Binding Power

I have a confession to make: I am always confused by "high precedence" and "low precedence". In `a + b * c`, addition has a lower precedence, but it is at the top of the parse tree...

So instead, I find thinking in terms of binding power more intuitive.

```
expr:   A       +       B       *       C
power:      3       3       5       5
```

The `*` is stronger, it has more power to hold together `B` and `C`, and so the expression is parsed as `A + (B * C)`.

What about associativity though? In `A + B + C` all operators seem to have the same power, and it is unclear which `+` to fold first. But this can also be modeled with power, if we make it slightly asymmetric:

```
expr:      A       +       B       +       C
power:  0      3      3.1      3      3.1     0
```

Here, we pumped the right power of `+` just a little bit, so that it holds the right operand tighter. We also added zeros at both ends, as there are no operators to bind from the sides. Here, the first (and only the first) `+` holds both of its arguments tighter than the neighbors, so we can reduce it:

```
expr:     (A + B)     +     C
power:  0          3    3.1    0
```

Now we can fold the second plus and get `(A + B) + C`. Or, in terms of the syntax tree, the second `+` really likes its right operand more than the left one, so it rushes to get hold of `C`. While he does that, the first `+` captures both `A` and `B`, as they are uncontested.

What Pratt parsing does is that it finds these badass, stronger than neighbors operators, by processing the string left to right. We are almost at a point where we finally start writing some code, but let's first look at the other running example. We will use function composition operator, `.` (dot) as a *right* associative operator with a high binding power. That is, `f . g . h` is parsed as `f . (g . h)`, or, in terms of power

```
  f     .    g     .    h
0   8.5    8   8.5    8   0
```

Minimal Pratt Parser

We will be parsing expressions where basic atoms are *single character* numbers and variables, and which uses punctuation for operators. Let's define a simple tokenizer:

```rust
#[derive(Debug, Clone, Copy, PartialEq, Eq)]
enum Token {
    Atom(char),
    Op(char),
    Eof,
}

struct Lexer {
    tokens: Vec<Token>,
}

impl Lexer {
    fn new(input: &str) -> Lexer {
        let mut tokens = input
            .chars()
            .filter(|it| !it.is_ascii_whitespace())
            .map(|c| match c {
                '0'..='9' |
                'a'..='z' | 'A'..='Z' => Token::Atom(c),
                _ => Token::Op(c),
            })
            .collect::<Vec<_>>();
        tokens.reverse();
        Lexer { tokens }
    }

    fn next(&mut self) -> Token {
        self.tokens.pop().unwrap_or(Token::Eof)
    }
    fn peek(&mut self) -> Token {
        self.tokens.last().copied().unwrap_or(Token::Eof)
    }
}
```

To make sure that we got the ~~precedence~~ binding power correctly, we will be transforming infix expressions into a gold-standard (not so popular in Poland, for whatever reason) unambiguous notation — S-expressions:

`1 + 2 * 3 == (+ 1 (* 2 3))`.

```rust
use std::fmt;

enum S {
    Atom(char),
    Cons(char, Vec<S>),
}

impl fmt::Display for S {
    fn fmt(&self, f: &mut fmt::Formatter<'_>) -> fmt::Result {
        match self {
            S::Atom(i) => write!(f, "{}", i),
            S::Cons(head, rest) => {
                write!(f, "({}", head)?;
                for s in rest {
                    write!(f, " {}", s)?
                }
                write!(f, ")")
            }
        }
    }
}
```

And let's start with just this: expressions with atoms and two infix binary operators, `+` and `*`:

```rust
fn expr(input: &str) -> S {
    let mut lexer = Lexer::new(input);
    expr_bp(&mut lexer)
}

fn expr_bp(lexer: &mut Lexer) -> S {
    todo!()
}

#[test]
fn tests() {
    let s = expr("1 + 2 * 3");
    assert_eq!(s.to_string(), "(+ 1 (* 2 3))")
}
```

So, the general approach is roughly the one we used to deal with left recursion — start with parsing a first number, and then loop, consuming operators and doing ... something?

```rust
fn expr_bp(lexer: &mut Lexer) -> S {
    let lhs = match lexer.next() {
        Token::Atom(it) => S::Atom(it),
        t => panic!("bad token: {:?}", t),
    };

    loop {
        let op = match lexer.next() {
            Token::Eof => break,
            Token::Op(op) => op,
            t => panic!("bad token: {:?}", t),
        };

        todo!()
    }

    lhs
}

#[test]
fn tests() {
    let s = expr("1"); ❶
    assert_eq!(s.to_string(), "1");
}
```

❶ Note that we already can parse this simple test!

We want to use this power idea, so let's compute both left and right powers of the operator. We'll use `u8` to represent power, so, for associativity, we'll add `1`. And we'll reserve the `0` power for the end of input, so the lowest power operator can have is `1`.

```rust
fn expr_bp(lexer: &mut Lexer) -> S {
    let lhs = match lexer.next() {
        Token::Atom(it) => S::Atom(it),
        t => panic!("bad token: {:?}", t),
    };

    loop {
        let op = match lexer.peek() {
            Token::Eof => break,
            Token::Op(op) => op,
            t => panic!("bad token: {:?}", t),
        };
        let (l_bp, r_bp) = infix_binding_power(op);

        todo!()
    }

    lhs
}

fn infix_binding_power(op: char) -> (u8, u8) {
    match op {
        '+' | '-' => (1, 2),
        '*' | '/' => (3, 4),
        _ => panic!("bad op: {:?}", op)
    }
}
```

And now comes the tricky bit, where we introduce recursion into the picture. Let's think about this example (with powers below):

```
a   +   b   *   c   *   d   +   e
  1   2   3   4   3   4   1   2
```

The cursor is at the first `+`, we know that the left `bp` is `1` and the right one is `2`. The `lhs` stores `a`. The next operator after `+` is `*`, so we shouldn't add `b` to `a`. The problem is that we haven't yet seen the next operator, we are just past `+`. Can we add a lookahead? Looks like no — we'd have to look past all of `b`, `c` and `d` to find the next operator with lower binding power, which sounds pretty unbounded. But we are onto something! Our current right priority is `2`, and, to be able to fold the expression, we need to find the next operator with lower priority. So let's recursively call `expr_bp` starting at `b`, but also tell it to stop as soon as `bp` drops below `2`. This necessitates the addition of `min_bp` argument to the main function.

And lo, we have a fully functioning minimal Pratt parser:

```rust
fn expr(input: &str) -> S {
    let mut lexer = Lexer::new(input);
    expr_bp(&mut lexer, 0) ❺
}

fn expr_bp(lexer: &mut Lexer, min_bp: u8) -> S { ❶
    let mut lhs = match lexer.next() {
        Token::Atom(it) => S::Atom(it),
        t => panic!("bad token: {:?}", t),
    };

    loop {
        let op = match lexer.peek() {
            Token::Eof => break,
            Token::Op(op) => op,
            t => panic!("bad token: {:?}", t),
        };

        let (l_bp, r_bp) = infix_binding_power(op);
        if l_bp < min_bp { ❷
            break;
        }

        lexer.next(); ❸
        let rhs = expr_bp(lexer, r_bp);

        lhs = S::Cons(op, vec![lhs, rhs]); ❹
    }

    lhs
}

fn infix_binding_power(op: char) -> (u8, u8) {
    match op {
        '+' | '-' => (1, 2),
        '*' | '/' => (3, 4),
        _ => panic!("bad op: {:?}", op),
    }
}

#[test]
fn tests() {
    let s = expr("1");
    assert_eq!(s.to_string(), "1");

    let s = expr("1 + 2 * 3");
    assert_eq!(s.to_string(), "(+ 1 (* 2 3))");

    let s = expr("a + b * c * d + e");
    assert_eq!(s.to_string(), "(+ (+ a (* (* b c) d)) e)");
}
```

❶ `min_bp` argument is the crucial addition. `expr_bp` now parses expressions with relatively high binding power. As soon as it sees something weaker than `min_bp`, it stops.

❷ This is the "it stops" point.

❸ And here we bump past the operator itself and make the recursive call. Note how we use `l_bp` to check against `min_bp`, and `r_bp` as the new `min_bp` of the recursive call. So, you can think about `min_bp` as the binding power of the operator to the left of the current expressions.

❹ Finally, after parsing the correct right hand side, we assemble the new current expression.

❺ To start the recursion, we use binding power of zero. Remember, at the beginning the binding power of the operator to the left is the lowest possible, zero, as there's no actual operator there.

So, yup, these 40 lines *are* the Pratt parsing algorithm. They are tricky, but, if you understand them, everything else is straightforward additions.

Bells and Whistles

Now let's add all kinds of weird expressions to show the power and flexibility of the algorithm. First, let's add a high-priority, right associative function composition operator: `.`:

```
fn infix_binding_power(op: char) -> (u8, u8) {
    match op {
        '+' | '-' => (1, 2),
        '*' | '/' => (3, 4),
        '.' => (6, 5),
        _ => panic!("bad op: {:?}", op),
    }
}
```

Yup, it's a single line! Note how the left side of the operator binds tighter, which gives us desired right associativity:

```
let s = expr("f . g . h");
assert_eq!(s.to_string(), "(. f (. g h))");

let s = expr(" 1 + 2 + f . g . h * 3 * 4");
assert_eq!(s.to_string(), "(+ (+ 1 2) (* (* (. f (. g h)) 3) 4))");
```

Now, let's add unary `-`, which binds tighter than binary arithmetic operators, but less tight than composition. This requires changes to how we start our loop, as we no longer can assume that the first token is an atom, and need to handle minus as well. But let the types drive us. First, we start with binding powers. As this is an unary operator, it really only have right binding power, so, ahem, let's just code this:

```
fn prefix_binding_power(op: char) -> ((), u8) { ❶
    match op {
        '+' | '-' => ((), 5),
        _ => panic!("bad op: {:?}", op),
    }
}

fn infix_binding_power(op: char) -> (u8, u8) {
    match op {
        '+' | '-' => (1, 2),
        '*' | '/' => (3, 4),
        '.' => (8, 7), ❷
        _ => panic!("bad op: {:?}"),
    }
}
```

❶ Here, we return a dummy `()` to make it clear that this is a prefix, and not a postfix operator, and thus can only bind things to the right.

❷ Note, as we want to add unary - between `.` and `*`, we need to shift priorities of `.` by two. The general rule is that we use an odd priority as base, and bump it by one for associativity, if the operator is binary. For unary minus it doesn't matter and we could have used either `5` or `6`, but sticking to odd is more consistent.

Plugging this into `expr_bp`, we get:

```
fn expr_bp(lexer: &mut Lexer, min_bp: u8) -> S {
    let mut lhs = match lexer.next() {
        Token::Atom(it) => S::Atom(it),
        Token::Op(op) => {
            let ((), r_bp) = prefix_binding_power(op);
            todo!()
        }
        t => panic!("bad token: {:?}", t),
    };
    ...
}
```

Now, we only have `r_bp` and not `l_bp`, so let's just copy-paste half of the code from the main loop? Remember, we use `r_bp` for recursive calls.

```rust
fn expr_bp(lexer: &mut Lexer, min_bp: u8) -> S {
    let mut lhs = match lexer.next() {
        Token::Atom(it) => S::Atom(it),
        Token::Op(op) => {
            let ((), r_bp) = prefix_binding_power(op);
            let rhs = expr_bp(lexer, r_bp);
            S::Cons(op, vec![rhs])
        }
        t => panic!("bad token: {:?}", t),
    };

    loop {
        let op = match lexer.peek() {
            Token::Eof => break,
            Token::Op(op) => op,
            t => panic!("bad token: {:?}", t),
        };

        let (l_bp, r_bp) = infix_binding_power(op);
        if l_bp < min_bp {
            break;
        }

        lexer.next();
        let rhs = expr_bp(lexer, r_bp);

        lhs = S::Cons(op, vec![lhs, rhs]);
    }

    lhs
}

#[test]
fn tests() {
    ...

    let s = expr("--1 * 2");
    assert_eq!(s.to_string(), "(* (- (- 1)) 2)");

    let s = expr("--f . g");
    assert_eq!(s.to_string(), "(- (- (. f g)))");
}
```

Amusingly, this purely mechanical, type-driven transformation works. You can also reason why it works, of course. The same argument applies; after we've consumed a prefix operator, the operand consists of operators that bind tighter, and we just so conveniently happen to have a function which can parse expressions tighter than the specified power.

Ok, this is getting stupid. If using `((), u8)` "just worked" for prefix operators, can `(u8, ())` deal with postfix ones? Well, let's add `!` for factorials. It should bind tighter than `-`, because `-(92!)` is obviously more useful than `(-92)!`. So, the familiar drill — new priority function, shifting priority of `.` (this bit *is* annoying in Pratt parsers), copy-pasting the code...

```
let (l_bp, ()) = postfix_binding_power(op);
if l_bp < min_bp {
    break;
}

let (l_bp, r_bp) = infix_binding_power(op);
if l_bp < min_bp {
    break;
}
```

Wait, something's wrong here. After we've parsed the prefix expression, we can see either a postfix or an infix operator. But we bail on unrecognized operators, which is not going to work... So, let's make `postfix_binding_power` to return an option, for the case where the operator is not postfix:

```rust
fn expr_bp(lexer: &mut Lexer, min_bp: u8) -> S {
    let mut lhs = match lexer.next() {
        Token::Atom(it) => S::Atom(it),
        Token::Op(op) => {
            let ((), r_bp) = prefix_binding_power(op);
            let rhs = expr_bp(lexer, r_bp);
            S::Cons(op, vec![rhs])
        }
        t => panic!("bad token: {:?}", t),
    };

    loop {
        let op = match lexer.peek() {
            Token::Eof => break,
            Token::Op(op) => op,
            t => panic!("bad token: {:?}", t),
        };

        if let Some((l_bp, ())) = postfix_binding_power(op) {
            if l_bp < min_bp {
                break;
            }
            lexer.next();

            lhs = S::Cons(op, vec![lhs]);
            continue;
        }

        let (l_bp, r_bp) = infix_binding_power(op);
        if l_bp < min_bp {
            break;
        }

        lexer.next();
        let rhs = expr_bp(lexer, r_bp);

        lhs = S::Cons(op, vec![lhs, rhs]);
    }

    lhs
}

fn prefix_binding_power(op: char) -> ((), u8) {
    match op {
        '+' | '-' => ((), 5),
        _ => panic!("bad op: {:?}", op),
    }
}

fn postfix_binding_power(op: char) -> Option<(u8, ())> {
    let res = match op {
        '!' => (7, ()),
        _ => return None,
    };
    Some(res)
}

fn infix_binding_power(op: char) -> (u8, u8) {
    match op {
        '+' | '-' => (1, 2),
        '*' | '/' => (3, 4),
        '.' => (10, 9),
        _ => panic!("bad op: {:?}"),
    }
}
```

```
#[test]
fn tests() {
    let s = expr("-9!");
    assert_eq!(s.to_string(), "(- (! 9))");

    let s = expr("f . g !");
    assert_eq!(s.to_string(), "(! (. f g))");
}
```

Amusingly, both the old and the new tests pass.

Now, we are ready to add a new kind of expression: parenthesised expression. It is actually not that hard, and we could have done it from the start, but it makes sense to handle this here, you'll see in a moment why. Parens are just a primary expressions, and are handled similar to atoms:

```
let mut lhs = match lexer.next() {
    Token::Atom(it) => S::Atom(it),
    Token::Op('(') => {
        let lhs = expr_bp(lexer, 0);
        assert_eq!(lexer.next(), Token::Op(')'));
        lhs
    }
    Token::Op(op) => {
        let ((), r_bp) = prefix_binding_power(op);
        let rhs = expr_bp(lexer, r_bp);
        S::Cons(op, vec![rhs])
    }
    t => panic!("bad token: {:?}", t),
};
```

Unfortunately, the following test fails:

```
let s = expr("(((0)))");
assert_eq!(s.to_string(), "0");
```

The panic comes from the loop below — the only termination condition we have is reaching eof, and `)` is definitely not eof. The easiest way to fix that is to change `infix_binding_power` to return `None` on unrecognized operands. That way, it'll become similar to `postfix_binding_power` again!

```rust
fn expr_bp(lexer: &mut Lexer, min_bp: u8) -> S {
    let mut lhs = match lexer.next() {
        Token::Atom(it) => S::Atom(it),
        Token::Op('(') => {
            let lhs = expr_bp(lexer, 0);
            assert_eq!(lexer.next(), Token::Op(')'));
            lhs
        }
        Token::Op(op) => {
            let ((), r_bp) = prefix_binding_power(op);
            let rhs = expr_bp(lexer, r_bp);
            S::Cons(op, vec![rhs])
        }
        t => panic!("bad token: {:?}", t),
    };

    loop {
        let op = match lexer.peek() {
            Token::Eof => break,
            Token::Op(op) => op,
            t => panic!("bad token: {:?}", t),
        };

        if let Some((l_bp, ())) = postfix_binding_power(op) {
            if l_bp < min_bp {
                break;
            }
            lexer.next();

            lhs = S::Cons(op, vec![lhs]);
            continue;
        }

        if let Some((l_bp, r_bp)) = infix_binding_power(op) {
            if l_bp < min_bp {
                break;
            }

            lexer.next();
            let rhs = expr_bp(lexer, r_bp);

            lhs = S::Cons(op, vec![lhs, rhs]);
            continue;
        }

        break;
    }

    lhs
}

fn prefix_binding_power(op: char) -> ((), u8) {
    match op {
        '+' | '-' => ((), 5),
        _ => panic!("bad op: {:?}", op),
    }
}

fn postfix_binding_power(op: char) -> Option<(u8, ())> {
    let res = match op {
        '!' => (7, ()),
        _ => return None,
    };
    Some(res)
}
```

```
fn infix_binding_power(op: char) -> Option<(u8, u8)> {
    let res = match op {
        '+' | '-' => (1, 2),
        '*' | '/' => (3, 4),
        '.' => (10, 9),
        _ => return None,
    };
    Some(res)
}
```

And now let's add array indexing operator: `a[i]`. What kind of -fix is it? Around-fix? If it were just `a[]`, it would clearly be postfix. if it were just `[i]`, it would work exactly like parens. And it is the key: the `i` part doesn't really participate in the whole power game, as it is unambiguously delimited. So, let's do this:

```rust
fn expr_bp(lexer: &mut Lexer, min_bp: u8) -> S {
    let mut lhs = match lexer.next() {
        Token::Atom(it) => S::Atom(it),
        Token::Op('(') => {
            let lhs = expr_bp(lexer, 0);
            assert_eq!(lexer.next(), Token::Op(')'));
            lhs
        }
        Token::Op(op) => {
            let ((), r_bp) = prefix_binding_power(op);
            let rhs = expr_bp(lexer, r_bp);
            S::Cons(op, vec![rhs])
        }
        t => panic!("bad token: {:?}", t),
    };

    loop {
        let op = match lexer.peek() {
            Token::Eof => break,
            Token::Op(op) => op,
            t => panic!("bad token: {:?}", t),
        };

        if let Some((l_bp, ())) = postfix_binding_power(op) {
            if l_bp < min_bp {
                break;
            }
            lexer.next();

            lhs = if op == '[' {
                let rhs = expr_bp(lexer, 0);
                assert_eq!(lexer.next(), Token::Op(']'));
                S::Cons(op, vec![lhs, rhs])
            } else {
                S::Cons(op, vec![lhs])
            };
            continue;
        }

        if let Some((l_bp, r_bp)) = infix_binding_power(op) {
            if l_bp < min_bp {
                break;
            }

            lexer.next();
            let rhs = expr_bp(lexer, r_bp);

            lhs = S::Cons(op, vec![lhs, rhs]);
            continue;
        }

        break;
    }

    lhs
}

fn prefix_binding_power(op: char) -> ((), u8) {
    match op {
        '+' | '-' => ((), 5),
        _ => panic!("bad op: {:?}", op),
    }
}

fn postfix_binding_power(op: char) -> Option<(u8, ())> {
    let res = match op {
```

```
        '!' | '[' => (7, ()), ❶
        _ => return None,
    };
    Some(res)
}

fn infix_binding_power(op: char) -> Option<(u8, u8)> {
    let res = match op {
        '+' | '-' => (1, 2),
        '*' | '/' => (3, 4),
        '.' => (10, 9),
        _ => return None,
    };
    Some(res)
}

#[test]
fn tests() {
    ...

    let s = expr("x[0][1]");
    assert_eq!(s.to_string(), "([ ([ x 0) 1)");
}
```

❶ Note that we use the same priority for `!` as for `[`. In general, for the correctness of our algorithm it's pretty important that, when we make decisions, priorities are never equal. Otherwise, we might end up in a situation like the one before tiny adjustment for associativity, where there were two equally-good candidates for reduction. However, we only compare right `bp` with left `bp`! So for two postfix operators it's OK to have priorities the same, as they are both right.

Finally, the ultimate boss of all operators, the dreaded ternary:

```
c ? e1 : e2
```

Is this ... all-other-the-place-fix operator? Well, let's change the syntax of ternary slightly:

```
c [ e1 ] e2
```

And let's recall that `a[i]` turned out to be a postfix operator + parenthesis... So, yeah, `?` and `:` are actually a weird pair of parens! And let's handle it as such! Now, what about priority and associativity? What associativity even is in this case?

```
a ? b : c ? d : e
```

To figure it out, we just squash the parens part:

```
a ?: c ?: e
```

This can be parsed as

```
(a ?: c) ?: e
```

or as

```
a ?: (c ?: e)
```

What is more useful? For `?`-chains like this:

```
a ? b :
c ? d :
e
```

the right-associative reading is more useful. Priority-wise, the ternary is low priority. In C, only `=` and `,` have lower priority. While we are at it, let's add C-style right associative `=` as well.

Here's our the most complete and perfect version of a simple Pratt parser:

```rust
use std::{fmt, io::BufRead};

enum S {
    Atom(char),
    Cons(char, Vec<S>),
}

impl fmt::Display for S {
    fn fmt(&self, f: &mut fmt::Formatter<'_>) -> fmt::Result {
        match self {
            S::Atom(i) => write!(f, "{}", i),
            S::Cons(head, rest) => {
                write!(f, "({}", head)?;
                for s in rest {
                    write!(f, " {}", s)?
                }
                write!(f, ")")
            }
        }
    }
}

#[derive(Debug, Clone, Copy, PartialEq, Eq)]
enum Token {
    Atom(char),
    Op(char),
    Eof,
}

struct Lexer {
    tokens: Vec<Token>,
}

impl Lexer {
    fn new(input: &str) -> Lexer {
        let mut tokens = input
            .chars()
            .filter(|it| !it.is_ascii_whitespace())
            .map(|c| match c {
                '0'..='9'
                | 'a'..='z' | 'A'..='Z' => Token::Atom(c),
                _ => Token::Op(c),
            })
            .collect::<Vec<_>>();
        tokens.reverse();
        Lexer { tokens }
    }

    fn next(&mut self) -> Token {
        self.tokens.pop().unwrap_or(Token::Eof)
    }
    fn peek(&mut self) -> Token {
        self.tokens.last().copied().unwrap_or(Token::Eof)
    }
}

fn expr(input: &str) -> S {
    let mut lexer = Lexer::new(input);
    expr_bp(&mut lexer, 0)
}

fn expr_bp(lexer: &mut Lexer, min_bp: u8) -> S {
    let mut lhs = match lexer.next() {
        Token::Atom(it) => S::Atom(it),
        Token::Op('(') => {
            let lhs = expr_bp(lexer, 0);
```

```
            assert_eq!(lexer.next(), Token::Op(')'));
            lhs
        }
        Token::Op(op) => {
            let ((), r_bp) = prefix_binding_power(op);
            let rhs = expr_bp(lexer, r_bp);
            S::Cons(op, vec![rhs])
        }
        t => panic!("bad token: {:?}", t),
    };

    loop {
        let op = match lexer.peek() {
            Token::Eof => break,
            Token::Op(op) => op,
            t => panic!("bad token: {:?}", t),
        };

        if let Some((l_bp, ())) = postfix_binding_power(op) {
            if l_bp < min_bp {
                break;
            }
            lexer.next();

            lhs = if op == '[' {
                let rhs = expr_bp(lexer, 0);
                assert_eq!(lexer.next(), Token::Op(']'));
                S::Cons(op, vec![lhs, rhs])
            } else {
                S::Cons(op, vec![lhs])
            };
            continue;
        }

        if let Some((l_bp, r_bp)) = infix_binding_power(op) {
            if l_bp < min_bp {
                break;
            }
            lexer.next();

            lhs = if op == '?' {
                let mhs = expr_bp(lexer, 0);
                assert_eq!(lexer.next(), Token::Op(':'));
                let rhs = expr_bp(lexer, r_bp);
                S::Cons(op, vec![lhs, mhs, rhs])
            } else {
                let rhs = expr_bp(lexer, r_bp);
                S::Cons(op, vec![lhs, rhs])
            };
            continue;
        }

        break;
    }

    lhs
}

fn prefix_binding_power(op: char) -> ((), u8) {
    match op {
        '+' | '-' => ((), 9),
        _ => panic!("bad op: {:?}", op),
    }
}

fn postfix_binding_power(op: char) -> Option<(u8, ())> {
```

```rust
fn postfix_binding_power(op: char) -> Option<(u8, ())> {
    let res = match op {
        '!' => (11, ()),
        '[' => (11, ()),
        _ => return None,
    };
    Some(res)
}

fn infix_binding_power(op: char) -> Option<(u8, u8)> {
    let res = match op {
        '=' => (2, 1),
        '?' => (4, 3),
        '+' | '-' => (5, 6),
        '*' | '/' => (7, 8),
        '.' => (14, 13),
        _ => return None,
    };
    Some(res)
}

#[test]
fn tests() {
    let s = expr("1");
    assert_eq!(s.to_string(), "1");

    let s = expr("1 + 2 * 3");
    assert_eq!(s.to_string(), "(+ 1 (* 2 3))");

    let s = expr("a + b * c * d + e");
    assert_eq!(s.to_string(), "(+ (+ a (* (* b c) d)) e)");

    let s = expr("f . g . h");
    assert_eq!(s.to_string(), "(. f (. g h))");

    let s = expr(" 1 + 2 + f . g . h * 3 * 4");
    assert_eq!(
        s.to_string(),
        "(+ (+ 1 2) (* (* (. f (. g h)) 3) 4))",
    );

    let s = expr("--1 * 2");
    assert_eq!(s.to_string(), "(* (- (- 1)) 2)");

    let s = expr("--f . g");
    assert_eq!(s.to_string(), "(- (- (. f g)))");

    let s = expr("-9!");
    assert_eq!(s.to_string(), "(- (! 9))");

    let s = expr("f . g !");
    assert_eq!(s.to_string(), "(! (. f g))");

    let s = expr("(((0)))");
    assert_eq!(s.to_string(), "0");

    let s = expr("x[0][1]");
    assert_eq!(s.to_string(), "([ ([ x 0) 1)");

    let s = expr(
        "a ? b :
         c ? d
         : e",
    );
    assert_eq!(s.to_string(), "(? a b (? c d e))");

```

```
    let s = expr("a = 0 ? b : c = d");
    assert_eq!(s.to_string(), "(= a (= (? 0 b c) d))")
}

fn main() {
    for line in std::io::stdin().lock().lines() {
        let line = line.unwrap();
        let s = expr(&line);
        println!("{}", s)
    }
}
```

The code is also available in this repository, Eof :-)