

# Parsing With Precedence

# Rolling your own recursive descent parser correctly handling operator precedence for fun and profit™

## PRECEDENCE

**Operators** apply operations to one or more **operands**.

When an expression contains several operators, they get applied in order of their **precedence**, highest to lowest. When equal, **associativity** decides. When parsing, compound expressions are seen as rooted trees, where internal nodes are operators, and the leaf nodes are **atomic expressions**<sup>[1]</sup> When building this tree, we informally say that higher precedence "binds tighter".

## EXAMPLE OPERATORS

EXAMPLE OPERATORS				
a() a[] a.member	Function call, indexing <sup>[2]</sup> Member access <sup>[3]</sup>	P O S T	9	Left-to-right
-a +a !a await a	Unary minus, plus, not Await expression	P R E	8	Right-to-left
a*b a/b	Multiplication, division	I N F I X	7	Left-to-right
a+b a-b	Addition, subtraction		6	
a<b a>b	Relational comparisons		5	
a==b a!=b	Equality comparisons		4	
a&&b	Logical AND		3	
a  b	Logical OR		2	
a=b a+=b	All assignments		1	

[1] Atomic expressions include: numbers, variables, parenthesized expressions.

[2] These operators consist of multiple tokens, and can have sub-expressions.

A single call to `lexer.advance()` is thus not enough, for most postfix operators.

[3] Member access is often listed as infix, with RHS required to be an identifier.

I chose postfix, since having infix operators above prefix often gets confusing.

# RUST

```

/// Consumes expressions and operators to build an expression tree.
/// Call with min_precedence = 0 to parse the entire expression.
/// Recurses with higher min_precedence to parse sub-expressions.
fn parse_expression(lexer: &mut Lexer, min_precedence: u32) -> ParseResult<Expression> {
    // If we have a prefix operator, always consume it, and recurse to parse its operand.
    // Otherwise, parse an atomic expression, e.g. 5 or (2+2) in parentheses
    let mut expression = if let Ok(op) = PrefixOp::try_from(lexer.current()) {
        lexer.advance(); // Consume the prefix operator token
        let rhs = parse_expression(lexer, op.precedence());
        Expression::PrefixOp(op, Box::new(rhs))
    } else {
        parse_atomic_expression(lexer)?
    };

    // Now we expand the expression as far as possible to the right by looping
    // to consume all infix and postfix operators where precedence >= min_precedence.
    // If an operator has too low precedence, we return to a parent parse_expression call.
    loop {
        if let Ok(op) = InfixOp::try_from(lexer.current()) {
            if op.precedence() < min_precedence { break; }
            lexer.advance(); // Consume the infix operator token
            // We recurse to parse its RHS, taking min_precedence from the operator.
            // If we are left associative, make min_precedence one higher.
            let rhs = parse_expression(lexer, op.precedence() + op.is_left_associative() as u32)?;
            expression = Expression::InfixOp(Box::new(expression), op, Box::new(rhs));
        }
        else if let Ok(op) = PostfixOp::try_from(lexer.current()) {
            if op.precedence() < min_precedence { break; }
            lexer.advance(); // Consume the postfix operator token
            expression = Expression::PostfixOp(Box::new(expression), op);
        }
        else { break; }
    }

    Ok(expression)
}

```

Full code at [github.com/hayed/operator-prec-poster](https://github.com/hayed/operator-prec-poster)

Full code at [github.com/haved/operator-prec-poster](https://github.com/haved/operator-prec-poster)

### EXAMPLE

## Lexer tokens

