Pratt Parsers: Expression Parsing Made Easy

Every now and then, I stumble onto some algorithm or idea that's so clever and such a perfect solution to a problem that I feel like I got smarter or gained a new superpower just by learning it. Heaps were one, just about the only thing I got out of my truncated CS education. I recently stumbled onto another: Pratt or "top-down operator precedence" parsers.

When you're writing a parser, recursive descent is as easy as spreading peanut butter. It excels when you can figure out what to parse based on the next bit of code you're looking at. That's usually true at the declaration and statement levels of a language's grammar since most syntax there starts with keywords—`class`, `if`, `for`, `while`, etc.

Parsing gets trickier when you get to expressions. When it comes to infix operators like `+`, postfix ones like `++`, and even mixfix expressions like `?:`, it can be hard to tell what kind of expression you're parsing until you're halfway through it. You *can* do this with recursive descent, but it's a chore. You have to write separate functions for each level of precedence (JavaScript has 17 of them, for example), manually handle associativity, and smear your grammar across a bunch of parsing code until it's hard to see.

Peanut butter and jelly, the secret weapon

Pratt parsing solves that. If recursive descent is peanut butter, Pratt parsing is the jelly. When you mix the two together, you get a simple, terse, readable parser that can handle any grammar you throw at it.

Pratt's technique for handling operator precedence and infix expressions is so simple and effective it's a mystery why almost no one knows about it. After the seventies, top down operator precedence parsers seem to have fallen off the Earth. Douglas Crockford's JSLint uses one to parse JavaScript, but his treatment is one of the very few remotely modern articles about it.

Part of the problem, I think, is that Pratt's terminology is opaque, and Crockford's article is itself rather murky. Pratt uses terms like "null denominator" and Crockford mixes in extra stuff like tracking lexical scope that obscures the core idea.

This is where I come in. I won't do anything revolutionary. I'll just try to get the core concepts behind top down operator precedence parsers and present them as clearly as I can. I'll switch out some terms to (I hope) clarify things. Hopefully I won't offend anyone's purist sensibilities. I'll be coding in Java, the vulgar Latin of programming languages. I figure if you can write it in Java, you can write it in anything.

What we'll be making

I'm a learn-by-doing person, which means I'm also a teach-by-doing one. So to show how Pratt parsers work, we'll build a parser for a tiny little toy language called *Bantam*. The language only has expressions since that's where Pratt parsing is really helpful, but that should be enough to convince you of its usefulness.

Even though Bantam is simple, it has a full gamut of operators: prefix (`+`, `-`, `~`, `!`), postfix (`!`), infix (`+`, `-`, `*`, `/`, `^`), and even a mixfix conditional operator (`?:`). It has multiple precedence levels and both right and left associative operators. It also has assignment, function calls and parentheses for grouping. If we can parse this, we can parse anything.

What we'll start with

All we care about is parsing, so we'll ignore the tokenizing phase. I slapped together a crude lexer that works and we'll just pretend that tokens are raining down from heaven or something.

A token is the smallest chunk of meaningful code. It has a type and a string associated with it. Given `from + offset(time)`, the tokens would be:

```
NAME "from"
PLUS "+"
NAME "offset"
LEFT_PAREN "("
NAME "time"
RIGHT_PAREN ")"
```

For this exercise, we won't be *interpreting* or *compiling* this code. We just want to parse it to a nice data structure. For our purposes, that means our parser should chew up a bunch of Token objects and spit out an instance of some class that implements Expression. To give you an idea, here's a simplified version of the class for a conditional expression:

```
class ConditionalExpression implements Expression {
  public ConditionalExpression(
      Expression condition,
      Expression thenArm,
      Expression elseArm) {
    this.condition = condition;
    this.thenArm   = thenArm;
    this.elseArm   = elseArm;
  }

  public final Expression condition;
  public final Expression thenArm;
  public final Expression elseArm;
}
```

(You gotta love Java's "please sign it in quadruplicate" level of bureaucracy here. Like I said, if you can tolerate this in Java, it can work in *any* language.)

We'll start from a simple Parser class. The parser owns the token stream, handles lookahead and provides the basic methods you need to write a top-down recursive descent parser with a single token of lookahead (it's LL(1)). This is enough to get us going. If we need more later, it's easy to extend it.

OK, let's build ourselves a parser!

First things first

Even though a "full" Pratt parser is pretty tiny, I found it to be hard to decipher. Sort of like quicksort, the implementation is a deceptively simple handful of deeply intertwined code. To untangle it, we'll build it up one tiny step at a time.

The simplest expressions to parse are prefix operators and single-token expressions. For those, the current token tells us everything we need to do. Bantam has one single-token expression: named variables. It has four prefix operators: `+`, `-`, `~`, and `!`. The simplest possible code to parse those is:

```
Expression parseExpression() {
  if (match(TokenType.NAME))       // Return NameExpression...
  else if (match(TokenType.PLUS))  // Return prefix + operator...
  else if (match(TokenType.MINUS)) // Return prefix - operator...
  else if (match(TokenType.TILDE)) // Return prefix ~ operator...
  else if (match(TokenType.BANG))  // Return prefix ! operator...
  else throw new ParseException();
}
```

But that's a bit monolithic. As you can see, we're switching off of a TokenType to branch to different parsing behavior. Let's encode that directly by making a Map from TokenTypes to chunks of parsing code. We'll call these chunks "parselets", and they will implement this:

```
interface PrefixParselet {
  Expression parse(Parser parser, Token token);
}
```

An parselet implementation to parse variable names is simply:

```
class NameParselet implements PrefixParselet {
  public Expression parse(Parser parser, Token token) {
    return new NameExpression(token.getText());
  }
}
```

We can use a single class for all of the prefix operators since they only differ in the actual operator token itself:

```
class PrefixOperatorParselet implements PrefixParselet {
  public Expression parse(Parser parser, Token token) {
    Expression operand = parser.parseExpression();
    return new PrefixExpression(token.getType(), operand);
  }
}
```

You'll note that it calls back into `parseExpression()` to parse the operand that appears after the operator (for example, to parse the `a` in `-a`). This recursion takes care of nested operators like `-+~!a`.

Back in Parser, the chained `if` statements are replaced with a map:

```
class Parser {
  public Expression parseExpression() {
    Token token = consume();
    PrefixParselet prefix = mPrefixParselets.get(token.getType());

    if (prefix == null) throw new ParseException(
        "Could not parse \"" + token.getText() + "\".");

    return prefix.parse(this, token);
  }

  // Other stuff...

  private final Map<TokenType, PrefixParselet> mPrefixParselets =
      new HashMap<TokenType, PrefixParselet>();
}
```

To define the grammar we have so far—variables and the four prefix operators—we'll add these helper methods:

```
public void register(TokenType token, PrefixParselet parselet) {
  mPrefixParselets.put(token, parselet);
}

public void prefix(TokenType token) {
  register(token, new PrefixOperatorParselet());
}
```

And now we can define the grammar like:

```
register(TokenType.NAME, new NameParselet());
prefix(TokenType.PLUS);
prefix(TokenType.MINUS);
prefix(TokenType.TILDE);
prefix(TokenType.BANG);
```

This is already an improvement over a recursive descent parser because our grammar is now more declarative instead of being spread out over a few imperative functions, and we can see the actual grammar all in one place. Even better, we can extend the grammar just by registering new parselets. We don't have to change the Parser class itself.

If we *only* had prefix expressions, we'd be done now. Alas, we don't.

Stuck in the middle

What we have so far only works if the *first* token tells us what kind of expression we're parsing, but that isn't always the case. With an expression like `a + b`, we don't know we have an add expression until after we parse the `a` and get to `+`. We have to extend the parser to support that.

Fortunately, we're in a good place to do so. Our current `parseExpression()` method parses a complete prefix expression including any nested prefix expressions and then stops. So, if we throw this at it:

```
-a + b
```

It will parse `-a` and leave us sitting on `+`. That's exactly the token we need to tell what infix expression we need to parse. Compared to prefix parsing, the only change for infix parsing is that there's another expression *before* the infix operator that the infix parser receives as an argument. Let's define a parselet that supports that:

```
interface InfixParselet {
  Expression parse(Parser parser, Expression left, Token token);
}
```

The only difference is that `left` argument, which is the expression we parsed before we got to the infix token. We wire this up to our parser by having another table of infix parselets.

Having separate tables for prefix and infix expressions is important because we sometimes have both a prefix and infix parselet for the same TokenType. For example, the prefix parselet for `(` handles grouping in an expression like `a * (b + c)`. Meanwhile, the *infix* parselet for `(` handles function calls like `a(b)`.

Now, after we parse the leading prefix expression, we look for an infix parser that matches the next token and wraps the prefix expression as an operand:

```
class Parser {
  public void register(TokenType token, InfixParselet parselet) {
    mInfixParselets.put(token, parselet);
  }

  public Expression parseExpression() {
    Token token = consume();
    PrefixParselet prefix = mPrefixParselets.get(token.getType());

    if (prefix == null) throw new ParseException(
        "Could not parse \"" + token.getText() + "\".");
    Expression left = prefix.parse(this, token);

    token = lookAhead(0);
    InfixParselet infix = mInfixParselets.get(token.getType());

    // No infix expression at this point, so we're done.
    if (infix == null) return left;

    consume();
    return infix.parse(this, left, token);
  }

  // Other stuff...

  private final Map<TokenType, InfixParselet> mInfixParselets =
      new HashMap<TokenType, InfixParselet>();
}
```

Pretty straightforward. We can implement an infix parselet for binary arithmetic operators like `+` like so:

```
class BinaryOperatorParselet implements InfixParselet {
  public Expression parse(Parser parser,
      Expression left, Token token) {
    Expression right = parser.parseExpression();
    return new OperatorExpression(left, token.getType(), right);
  }
}
```

Infix parselets also works for postfix operators. I'm calling them "infix", but they're really "anything but prefix". If there's some leading subexpression that comes before the token, the token will be handled by an infix parselet. That includes postfix expressions and mixfix ones like `?:`.

Postfix expressions are as simple as single-token prefix parselets: they just take the `left` operand and wraps it in another expression:

```
class PostfixOperatorParselet implements InfixParselet {
  public Expression parse(Parser parser, Expression left,
      Token token) {
    return new PostfixExpression(left, token.getType());
  }
}
```

Mixfix is easy too. It's similar to recursive descent:

```
class ConditionalParselet implements InfixParselet {
  public Expression parse(Parser parser, Expression left,
      Token token) {
    Expression thenArm = parser.parseExpression();
    parser.consume(TokenType.COLON);
    Expression elseArm = parser.parseExpression();

    return new ConditionalExpression(left, thenArm, elseArm);
  }
}
```

Now we can parse prefix, postfix, infix, and even mixfix expressions. With a pretty small amount of code, we can parse complex nested expressions like `a + (b ? c! : -d)`. We're done, right? Well… almost.

Excuse you, aunt Sally

Our parser *can* parse all of this stuff, but it doesn't parse it with the right precedence or associativity. If you throw `a - b - c` at the parser, it will parse the nested expressions like `a - (b - c)`, which isn't right. (Well, actually it is *right*—associative that is. We need it to be *left*.)

And this *last* step where we fix that is where Pratt parsers go from pretty nice to totally radical. We'll make two simple changes. We extend `parseExpression()` to take a *precedence*—a number that tells which expressions can be parsed by that call. If `parseExpression()` encounters an expression whose precedence is lower than we allow, it stops parsing and returns what it has so far.

To make that check we need to know the precedence of any given infix expression. We'll let the parselet specify it:

```
public interface InfixParselet {
  Expression parse(Parser parser, Expression left, Token token);
  int getPrecedence();
}
```

Using that, our core expression parser looks like this:

```
public Expression parseExpression(int precedence) {
  Token token = consume();
  PrefixParselet prefix = mPrefixParselets.get(token.getType());

  if (prefix == null) throw new ParseException(
      "Could not parse \"" + token.getText() + "\".");

  Expression left = prefix.parse(this, token);

  while (precedence < getPrecedence()) {
    token = consume();

    InfixParselet infix = mInfixParselets.get(token.getType());
    left = infix.parse(this, left, token);
  }

  return left;
}
```

That relies on a tiny helper function to get the precedence of the current token or a default value if there's no infix parselet for the token:

```
private int getPrecedence() {
  InfixParselet parser = mInfixParselets.get(
      lookAhead(0).getType());
  if (parser != null) return parser.getPrecedence();

  return 0;
}
```

And that's it. To wire precedence into Bantam's grammar, we set up a little precedence table:

```
public class Precedence {
  public static final int ASSIGNMENT  = 1;
  public static final int CONDITIONAL = 2;
  public static final int SUM         = 3;
  public static final int PRODUCT     = 4;
  public static final int EXPONENT    = 5;
  public static final int PREFIX      = 6;
  public static final int POSTFIX     = 7;
  public static final int CALL        = 8;
}
```

To make our operators parse their operands with the correct precedence, they pass an appropriate value back into `parseExpression()` when they call it recursively. For example, the BinaryOperatorParselet instance that handles the `+` operator passes in `Precedence.SUM` when it parses its right-hand operand.

Associativity is easy too. If an infix parselet calls `parseExpression()` with the *same* precedence that it returns for its own `getPrecedence()` call, you get left associativity. To be right-associative, it just needs to pass in *one less* than that instead.

Go forth and multiply

I've rewritten the parser for Magpie using this and it worked like a charm. I'm also working on a JavaScript parser using this technique and again it's been a great fit.

I Pratt parsers to be simple, terse, extensible (Magpie, for example, uses this to let you extend its own syntax at runtime), and easy to read. I'm at the point where I can't imagine writing a parser any other way. I never thought I'd say this, but parsers feel easy now.

To see for yourself, just take a look at the complete program.

Update 2022/05/14: John Cardinal has ported the code to C#.