From Precedence Climbing to Pratt Parsing

Theodore S. Norvell

An earlier article by me explored a number of approaches to recursive descent parsing of expressions, including a particularly simple, efficient, and flexible algorithm invented (as nearly as I can tell) by Martin Richards [7], which I called "precedence climbing".

Recently it was pointed out to me by Andy Chu that precedence climbing is really a special case of Pratt parsing, an algorithm (or approach) invented by Vaughn Pratt [6] in the early 70s. Andy wrote a blog post about it [1]. He goes as far as to say they are the same algorithm; I'd prefer to say that Pratt parsing is a generalization of precedence climbing.

The purpose of this post is to explore this connection by starting with precedence climbing and refactoring it to use the command pattern until we arrive at a Pratt parser. We can then expand the language that can be parsed; this expansion is extremely modular in the sense that we can add new syntactic constructs just by adding a few entries to tables.

For background, read my earlier article on expression parsing [5], especially the part on precedence climbing.

If your purpose is learn about Pratt parsing, this post might or might not be for you.

- If you already know about precedence climbing or are willing to learn about it first, then this article provides a gentle introduction to Pratt parsing, since it shows how Pratt parsing is a refactoring and generalization of the precedence climbing algorithm you already (will) know.
- But, if you don't know about precedence climbing and want to jump straight into learning about Pratt parsing, this article is probably not the best. There are many tutorials on the web; I haven't read many of them, since I had access to Pratt's original paper; a guide to some of these can be found at http://www.oilshell.org/blog/2016/11/02.html [2].

The precedence climbing algorithm

This section is a quick review. For a full explanation see my earlier article. We'll start with an ambiguous left-recursive grammar.

```
S --> E0 end
E0 --> E10
E10 --> E20 | E20 "=" E20
E20 --> E30 | E20 "+" E30 | E20 "-" E30
E30 --> E40 | E30 "*" E40 | E30 "/" E40
E40 --> E50 | E40 "!"
E50 --> E60 | E60 "^" E50
E60 --> P
P --> "-" E30 | "(" E0 ")" | v
```

We have (in order of increasing precedence)

- a nonassociative binary operator "=" (e.g., "a=b=c" is not in the language of S),
- left-associative operators "+", and "-"
- a prefix operator "-"
- left-associative operators "*" and "/"
- a postfix operator "!"
- a right-associative operator "^".

The set of tokens includes "=", "+", "-", "*", "/", "!", "^", "(", ")", and a set of variables represented by the symbol v in the grammar. A special token "end" marks the end of the input.

To see that the grammar is ambiguous, notice that the string `-a*b` can be derived in two ways. The ambiguity is resolved by saying that -, as a prefix operator, should have higher precedence than the binary operators *, /, !, and ^. For example `-a*b` should mean the same `-(a*b)`, not `(-a)*b`. Unlike the precedence between binary operators, this precedence is not fully captured by the grammar. The reason is that I don't know a nice way to do that.

Note that the postfix operator ! has lower precedence than the binary operator ^, so that "a^b!" means the same as "(a^b)!" and "a! ^ b" is not an expression. We'll look at how to add expressions like "a! ^b" to the language later.

The interface to the lexical analyser is via two procedures: `next` returns the next token as a string and `consume` advances the lexical analyser so that `next` returns the subsequent token. When there are no more tokens, `next` will return special token value `end`; further calls to `consume` have no effect on `next`.

After a few grammar transformations, converting the grammar to code, some code transformations, and adding some tree building code, we get the following algorithm, which I call preceedence climbing. The "climbing is because if you build a parse tree above the sequence of input tokens, the loop in procedure E climbs up (toward the root) the tree along the left edge of the tree and

each recursive call to E climbs up the mini-tree that forms the right operand of a binary or prefix operand.

```
S is const t := E(0) ; expect( end ) ; output(t)

E(p) is
    precondition 0 ≤ p
    var t := P
    var r := +inf
    loop
        const l := next
        exit unless p≤prec(next)≤r
        consume
        if isBinary(l) then
            const y := E( rightPrec(l) )
            t := mknode(binary(l), t, y)
        else t := mknode(postfix(l), t)
        r := nextPrec(l)
    return t

P is
    const n := next
    consume
    if n="-" then
        const t:= E(30)
        return mknode(prefix('-', t))
    else if n = "(" then
        const t := E(0)
        expect(")")
        return
    else if n is a variable then
        const t :=  mkleaf(n)
        consume
        return t
    else
        error( "Unexpected token '" +n+ "'" )

expect( tok ) is
   if next = tok
       consume
   else
       error( "Unexpected token '" +next+ "'; a '" +tok+ "' was expected."
```

and four tables

b	=	+	-	*	/	!	^
prec(*b*)	10	20	20	30	30	40	50
rightPrec(*b*)	20	21	21	31	31	NA	50
nextPrec(*b*)	9	20	20	30	30	40	49
isBinary(*b*)	true	true	true	true	true	false	true

Additionally, every other token should have a prec of -1. Since p is nonnegative, this will force an exit from the loop in E when a token is encountered that can't be a binary or postfix operator.

The algorithm handles prefix, postfix, and infix operators in a compact way. We can modify the algorithm to handle other operators such as, say, a ternary

y **if** x **else** z

operator, or an array indexing operation like

x[y]

I have done this, for example for my C++ expression parser. However, these extensions are ad hoc and complicate the E and P procedures. To accommodate operators like these and additional prefix operators in a modular way, we can use Pratt parsing.

Categories of tokens

Before going on, let's look at a way of classifying occurrences of tokens into three categories

- "N" token occurrences are occurrences of tokens that begin an expression. So any variable occurrence is an N occurrence, any occurrence of "(" is an N occurrence, and occurrences of "-" when it's used as a prefix operator are N occurrencse.
- "L" token occurrences have some operand on the left. This includes occurrences of binary and postfix operators.
- "O" token occurrences are all others. For our grammar, these are any occurrences of ")" and the end of input marker are "O" occurrences.

Let's look at an example

```
a + - b * ( c - b ! ) end
N L N N L N N L N L O O
```

In the precedence climbing algorithm: N occurrences are those that are used to make the initial choice in the `P` procedure and that are then consumed as opposed to leading to an error being reported; and L occurrences are those directly consumed by the `E` procedure. In the example, the first "-" is consumed by `P`, so it is an N, while the second "-"is consumed by `E`, so it is an L.

O token occurences are usually associated with some earlier L or N token occurrence. E.g., each ")" is associated with an earlier occurance"(" which is an N occurrance. When we look at expressions like

y **if** x **else** z ,

we will see that the "if" is an L occurrence and the "else" is an O occurence. The "end" token is a bit of an exception in that it's not connected to anything; but if we had a special "begin" token to mark the start of the token stream, then the "begin" would be an N occurrence and the "end" would be connected to the "begin".

So that's a three way classification of token occurrences We can also classify tokens

- N tokens are tokens that can occur as N occurrences. Examples are "-", "(", and any variable.
- L tokens are tokens that can occur as L occurrences. Examples are "-", "+","*", "/", "!", and any variable.
- O tokens are tokens that can't occur as N or L occurrences. Examples are ")" and "end".

So a token can be both N and L token (for example "-"), but O tokens can't be anything else and every token will be in (at least) one of these these three categories.

Dealing with L tokens

Refactor to use the command pattern

First we'll refactor the treatment of binary and postfix operations. The idea is to make procedure E more data-driven by treating binary and postfix operators uniformly. We'll map each token to a command object via a table, which I'll call `leftComm`.

Each command object has three methods

```
    abstract class LeftCommand

        abstract LeD( x, op ) returns Tree

        abstract LBP returns int

        NBP returns int is
            return LBP
```

Let's look at each method

- LBP stands for "left binding power". It replaces the `prec` table. L tokens will have a nonnegative LBP, while all others have an LBP of -1.
- NBP stands for the "next binding power". It replaces the `nextPrec` table. I.e. it gives the highest precedence of the operator that this operator can be a left operand of.
- LeD stands for "left denotation". It takes a left operand (x) and an operator (op) and returns the value denoted by the operator considered as an L token. In the case of a binary operator, it is responsible also for parsing the right operand. The LeD method won't be called if the LBP is negative.

We'll rewrite the E procedure to use commands. The old E is

```
E(p) is
    precondition 0 ≤ p
    var t := P
```

```
    var r := +inf
    loop
        const l := next
        exit unless p≤prec(next)≤r
        consume
        if isBinary(l) then
            const y := E( rightPrec(l) )
            t := mknode(binary(l), t, y)
        else t := mknode(postfix(l), t)
        r := nextPrec(l)
    return t
```

The new E is

```
E(p) is
    precondition 0 ≤ p
    var t := P
    var r := +inf
    loop
        const l := next
        const c := leftComm[l]
        exit unless p ≤ c.LBP ≤ r
        consume
        t := c.LeD( t, l )
        r := c.NBP
    return t
```

As happens often in OO programming, we've replaced a choice made by an if command with dynamic dispatch. Here we've replaced the lines

```
    if isBinary(l) then
        const y := E( rightPrec(l) )
        t := mknode(binary(l), t, y)
    else
        t := mknode(postfix(l), t)
```

With the line `t := c.LeD( t, l )`.

So the responsibility of the LeD method is to parse any operands that come after the operator and to build a representation (denotation) of the operation.

For every token that isn't an L token, the `leftComm` table uses a direct instance of `DefaultLeftCommand`, (below) ensuring that its `LBP` is -1. This will force an exit from the loop in `E` when a non-L token is encountered, as the `p` parameter is never less than 0.

```
    class DefaultLeftCommand extends LeftCommand

        override LBP is return -1

        override LeD( x, op ) is // Will not be called!
            assert false

leftComm[ "(" ] := new DefaultLeftCommand
leftComm[ ")" ] := new DefaultLeftCommand
```

```
leftComm[ "end"] := new DefaultLeftCommand
leftComm[ v ] := new DefaultLeftCommand
```

for all variables `v`.

Binary operators

Binary and postfix operators have an LBP that is nonnegative

```
abstract class BinaryOrPost( lbp ) extends LeftCommand
    precondition 0 ≤ lbp

    override LBP is
        return lbp
```

> I should explain that the lbp variable used in the precondition and the implementation of LBP refers to the value of the constructor parameter.

Binary operators have an LeD method that parses the right operand. There are 3 variations. The RBP method indicates the "right binding power" and determines the left binding power of the lowest precedence operator that can be in a right operand.

```
abstract class BinaryOperator( lbp ) extends BinaryOrPost( lpb )

    protected abstract RBP

    override LeD( x, op ) is
        const y := E( this.RBP )
        return mknode( binary(op), x, y )


class BopLeftAssoc( lbp ) extends BinaryOrPost( lpb )

    override RBP is return 1 + this.LBP

class BopRightAssoc( lbp ) extends BinaryOrPost( lpb )

    override RBP is return this.LBP

class BopNonAssoc( lbp ) extends BinaryOrPost( lpb

    override RBP is return 1 + this.LBP

    override NBP is return this.LBP - 1
```

We can declare some binary operators

```
leftComm[ "=" ] := new BopNonAssoc( 10 )
leftComm[ "+" ] := new BopLeftAssoc( 20 )
leftComm[ "-" ] :=  new BopLeftAssoc( 20 )
leftComm[ "*" ] :=  new BopLeftAssoc( 30 )
leftComm[ "/" ] :=  new BopLeftAssoc( 30 )
leftComm[ "^" ] :=  new BopRightAssoc( 50 )
```

Let's try some examples.

- For a left associative operator, the RBP is one more than the LBP meaning that an operator of the same precedence can not be in the right operand. Consider an expression "a + b + c end" and an invocation of E(0) the LBP of + is 20, so it's RBP is 21 and its NBP is 20.
  1. 'a' is consumed by a recursive call to P; the + is consumed in the first iteration of E's loop, as the loop guard p ≤ c.LBP ≤ r evaluates to 0 ≤ 20 ≤+inf, which is true
  2. Now there is a call to the LeD method for + with arguments x='a' and op='+'.
     1. The LeD method invokes E(21.)
        1. This invocation invokes P which consumes the 'b'.
        2. In the first iteration it sees the second '+' but refuses to consume it, as the loop guard evaluates to 21 ≤ 20 ≤+inf which is false. This is the turning point.
        3. The invocation of E(21) returns 'b' as a tree.
     2. The invocation of LeD returns a tree +[a,b].
  3. That makes t = +[a,b]
  4. At the end of the first iteration, r is set to 20, which is the NBP of +.
  5. In the second iteration we see the second '+'. The loop guard is p ≤ c.LBP ≤ r evaluates to 0 ≤ 20 ≤ 20 and invokes the LeD method for '+'.
     1. The LeD invokes E(21.)
        1. E(21) consumes the c and c as a tree. It encounters the end, which as we'll see later causes it to return
     2. The LeD method returns +[+,a,b],c]
  6. In the third iteration, the invocation of E(0) sees the "end" token which causes it to return with the result of +[+,a,b],c]
- For right associative operators we make the right precedence the same as the LBP. This means that operators of the same precedence will be consumed in the recursive call to E. Consider 'a^b^c' and again a call to E(0). The LPB of '^' is 40, so its RPB will be 40 also.
  1. 'a' is consumed by a recursive call to P; the first ^ is consumed in the first iteration of E's loop, as the loop guard p ≤ c.LBP ≤ r evaluates to 0 ≤ 40 ≤+inf, which is true
  2. Now there is a call to the LeD method for ^ with arguments x='a' and op='^'.
     1. The LeD method invokes E(40.)
        1. This invocation invokes P which consumes the 'b'.
        2. In the first iteration it sees the second '^'. the loop guard evaluates to 40 ≤ 40 ≤+inf which is true. This is where it differs from a left associative operator. So the second '^' is consumed.

3. The LeD method for '^' is recursively invoked with x='b' and op='^'
    1. It recursively invokes E(40) which returns 'c' after running into the 'end' token
    2. The LeD method returns the tree ^[b,c]
4. The invocation of E(40) sees the end and returns ^[b,c]

2. The LeD method constructs a tree ^[a,^[b,c]] and returns it.

3. The invocation of E(0), see the 'end' and returns ^[a,^[b,c]]

- Finally nonassociative operators are similar to left associative operators in that they set the right binding power to one more than the left binding power. But they also set the NBP to to one less than the LPB. Consider a a=b=c and a call to E(0). The LBP of '=' is 10 so its RBP is 11 and its NPB is 9. Things start out much like the + example above.
    1. 'a' is consumed by a recursive call to P; the first '=' is consumed in the first iteration of E's loop, as the loop guard p ≤ c.LBP ≤ r evaluates to 0 ≤ 10 ≤+inf, which is true
    2. Now there is a call to the LeD method for '=' with arguments x='a' and op = '='.
        1. The LeD method invokes E(11.)
            1. This invocation invokes P which consumes the 'b'.
            2. In the first iteration it sees the second '=' but (similar to a left associative operator) refuses to consume it, as the loop guard evaluates to 11 ≤ 10 ≤+inf which is false.
            3. The invocation of E(11) returns 'b' as a tree.
        2. The invocation of LeD returns a tree =[a,b].
    3. That makes t = =[a,b]
    4. At the end of the first iteration, r is set to 9, which is the NBP of =.
    5. So when the second '=' is encountered in the second iteration we have a loop guard of p ≤ c.LBP ≤ r which evaluates to 0 ≤ 10 ≤ 9, which is false. This is where it differs from a left-associative operator.
    6. E(0) returns with a value of =[a,b], and there are remaining tokens, [=, c, end] in the input stream.

Postfix operators

Postfix operators don't have a right operand and so they don't parse one in their LeD method. Otherwise, they are similar to binary operators.

```
class Postfix( lbp ) extends BinaryOrPost( lpb )

    override LeD( x, OK ) is
        return mknode( postfix(op), x )
```

We'll add our postfix operator to the table

```
leftComm[ "!" ] :=  new Postfix( 40 )
```

Note that "a! ^ b" is not allowed, just as it is not allowed in the original grammar. Recall that the NBP value for a binary or postfix operator gives the highest precedence of an operator that this operator can be a left operand of. Since the NBP of ! is 40 --the default is that it is the same as the LBP -- the "a!" can not be the left operand of a ^. If we want to allow expressions like "a! ^ b", we just need to arrange that `leftComm[ "!" ].NBP = +inf`. That allows expressions that end with a bang to be left operands of any binary operator, which seems reasonable.

```
class Postfix( lbp) extends BinaryOrPost( lpb )

    override LeD( x, OK ) is
        return mknode( postfix(op), x )

     override NBP is return +inf
```

Now "a! ^ b" will be parsed. So will "a ^ b ! ^ c", which will have the same meaning as "(a ^ b)! ^ c" since ! has lower precedence.

Not-quite-binary operators

At this point, we can see how to implement the indexing (x[y]) operation and the conditional (y if x else z) operation. I'll give these precedences of 60 and 5 respectively.

```
leftComm[ "[" ] := new IndexOp( 60 )
leftComm[ "if"] := new ConditionalOp( 5 )
```

The other new tokens are 0 tokens and so have the default left command.

```
leftComm[ "]" ] := new DefaultLeftCommand
leftComm[ "then"] := new DefaultLeftCommand
leftComm[ "else"] := new DefaultLeftCommand

class IndexOp( lbp ) extends BinaryOrPost( lbp )

    override LeD( x, op ) is
        const y := E( 0 )
        expect( "]" )
        return mknode( binary( "index" ), x, y )

class ConditionalOp( lbp ) extends BinaryOrPost( lpb )

    override LeD( y, op ) is
        const x := E( 0 )
        expect( "else" )
        const z := E( RPB )
        return mknode( ternary( "if" ), x, y, z )
```

```
    override NBP is
        return this.LBP - 1

    protected RBP is
        return this.LBP + 1
```

> I've made the conditional nonassociative to rule out "confusing" expressions like `x if a else y if b else z`. (OK, so you don't find that confusing? Then make it right associative by making NBP=LBP=RBP or left associative by making NBP=LBP=RBP-1.)

Chaining relational operators

Let's treat = as a "chaining operator" rather than nonassociative. That is `x=y=z` should mean `(x=y) and (y=z)` (supposing we had an "and" operator).

```
class Chaining( lbp ) extends BinaryOrPost( lbp )

    override LeD( x, op ) is
        const y := E( 1 + this.LBP )
        const t := mknode( binary( op ), x, y )
        if leftComm[ next ].LPB = this.LBP
            const nextOp := next
            consume
            const t1 := this.LeD( y, nextOp )
            return mkNode( binary( "and", t, t1 )
        else
            return t

    override NBP is
        return this.LBP - 1
```

and we change the entry for = in the table to

```
leftComm[ "=" ] := new Chaining( 10 )
```

Now let's give "=" some friends

```
leftComm[ "<" ] := new Chaining( 10 )
leftComm[ "≤" ] := new Chaining( 10 )
leftComm[ ">" ] := new Chaining( 10 )
leftComm[ "≥" ] := new Chaining( 10 )
```

Now "a ≤ b = c < d" means just what it should.

From a software engineering point of view, the important point is how modular these additions and alterations are. The basic parsing procedure `E` did not need to change; we just added some new classes and altered the table.

Dealing with N tokens

Can we make the treatment of prefix operators and other N tokens equally modular? Well, I wouldn't ask if we couldn't. Remember the N tokens are "-", "(", and the variables v.

Here is P again

```
P is
    const n := next
    consume
    if n="-" then
        const t:= E(30)
        return mknode(prefix('-', t))
    else if n = "(" then
        const t := E(0)
        expect(")")
        return
    else if n is a variable then
        const t :=  mkleaf(n)
        consume
        return t
    else
        error( "Unexpected token '" +n+ "'" )
```

Our next change is to make the P procedure data driven; i.e. to use dynamic dispatch to choose the course of action.

```
P is
    const n := next
    consume
    return nullComm[n].NuD( n )
```

`nullComm` is a table analogous to `leftComm`, but used for N tokens.

```
nullComm[ "-" ] := new UnaryOp( 30 )
nullComm[ "(" ] := new Parens
nullComm[ v ] := new Leaf
```

The last line needs to be repeated for each possible variable -- although the same command object can be shared for all entries. All other tokens (all non-N tokens) should map to an instance of `ErrorNullCommand`.

```
nullComm[ "end" ] := new ErrorNullCommand
nullComm[ ")" ] := new ErrorNullCommand
nullComm[ "]" ] := new ErrorNullCommand
nullComm[ "if" ] := new ErrorNullCommand
nullComm[ "then" ] := new ErrorNullCommand
nullComm[ "else" ] := new ErrorNullCommand
nullComm[ "=" ] := new ErrorNullCommand
...
nullComm[ "^" ] := new ErrorNullCommand
```

Now we need some classes for commands

```
abstract class NullCommand
    abstract NuD( op ) returns Tree
```

```
class ErrorNullCommand extends NullCommand

    override NuD( op ) is
        error( "Unexpected token '" +op+ "'" )

class UnaryOp( rbp ) extends NullCommand

    protected const rbp := rbp

    override NuD( op ) is
        const y := Exp( this.rbp )
        return mkNode( prefix( op ), y )

class Parens extends NullCommand

    override NuD( op ) is
        const t := E( 0 )
        expect( ")" )
        return t

class Leaf extends NullCommand

    override NuD( op ) is
        return mkLeaf( op )
```

Not-quite-unary operators

Now it's easy to add new kinds of prefix operators and such. For example, we can add expressions like `if x then y else z` to the language like this.

```
nullComm[ "if" ] := new IfThenElse( )

class IfThenElse extends NullCommand

    override NuD( op ) is
        const x := Exp( 0 )
        expect( "then" )
        const y := Exp( 0 )
        expect( "else" )
        const z := Exp( 0 )
        return mknode( ternary( "if" ), x, y, z )
```

It might not be the best language design choice, but it is interesting that we can use both syntaxes for conditional expressions in the same language; there is no difficulty for the parser to handle, for example

```
    if a if b else c then d if e else f else if g then h else i if j else k
```

This works because the parser can classify an occurrence as N or L based only on what the token is and what came before it. This works for "if" for the same reason that it works for "-". Similarly we can add a prefix ! operator. There is no problem

understanding "!a!!!" as long as you know the relative precedence of prefix and postfix !.

Tokens that are both L and N token

So far "-" and possibly "if" are can be either L and N tokens depending on context. Identifiers are N tokens only.

The The Scala language allows identifiers as binary operators. For example you can write "s contains b" instead of "s.contains(b)". Since an N occurrence can't follow an identifier, the parser can classify "contains" as an L occurrence. (I don't know if the Scala compiler uses Pratt parsing and I'm not advocating this as a language design choice.) Let's for fun allow identifiers to be binary operators with the highest precedence. Add to the table.

```
leftComm[ v ] := new BopLeftAssoc( 70 )
```

for all variables `v`.

Alternatively, Haskell takes a different approach from Scala. In Haskell an expression followed by another expression represents application, so, for example "f (g x) y" should parse to a tree apply[apply[f,apply[g, x]], y]. Again we need to treat variables as L tokens as well as N tokens; and this time also "(", and in fact any N token that isn't already also an L token. The problem is that the occurrences of "(", "x", and "y" in the example are both L occurrences and N occurrences They are L occurrences because they have a complete subexpression to the left and N occurrences because they start a complete subexpression. We can mange this in two ways.

The first is, provided we can put a token back into the token stream, we can treat it as an L token first, then put it back, and then treat it as an N token. Let's suppose the procedure "unconsume" puts a token back into the stream of incoming tokens. We can design a command

```
class Application( lbp ) extends BinaryOperator( lpb )

    override RBP is return 1 + this.LBP

    override LeD( x, op ) is
        unconsume( op )
        const y := E( this.RBP )
        return mknode( binary("apply"), x, y )
```

We add to the table

```
leftComm[ "(" ] := new Application( 70 )
leftComm[ v ] := new Application( 70 )
```

for every v and so on for every N token that isn't already also an L token. In particular we leave "-" alone, so that "x - y"

results in -[x, y] rather than apply[x, -[y]].

A second way to achieve same end is to remove the call to consume from the loop of the E procedure. Then we don't need the "unconsume" call in the LeD for application, but we need to add a consume to every other implementation of LeD.

These examples, I think, illustrate how flexible the parsing technique it.

The End and other O tokens.

What about the "end" token. Did I forget about it? Yes I did. Let's address that now. "end" is always an O token. Any token that can't be an L token or an N token is an O token and has the default LeftCommand and the default NullCommand. Any O token has the default LeftCommand and so it has a LBP of -1, meaning that if it is encountered where an L token is expected it ends any expression. It has the default NullCommand, meaning that if it's encountered where an N token is expected it is an error. So the 'end' token is just like any other O token. The only thing that makes "end" special is that our main routine uses it as follows and it is not used anywhere else.

```
S is
    var t := E(0)
    expect("end")
    return t
```

Summary

Here is the final version of the algorithm after inlining P.

```
S is
    var t := E(0)
    expect("end")
    return t

E(p) is
    precondition 0 ≤ p
    const n := next
    consume
    var t := nullComm[n].NuD( n )
    var r := +inf
    loop
        const l := next
        const c := leftComm[l]
        exit unless p ≤ c.LBP ≤ r
        consume
        t := c.LeD( t, l )
        r := c.NBP
    return t
```

When not to use Pratt parsing

Pratt parsing works great when there is essentially only one kind of thing that you are parsing. Consider a grammar for a simple imperative programming language

...TODO...

Comparing with Pratt's paper

Pratt's paper was not written in an object-oriented style. Instead he puts procedures directly in tables. For this reason, he has 3 tables (NuD, LeD, LBP) rather than 2. (The addition of NBP is my own; if Pratt had had it he'd have had 4 tables.) Using classes and objects makes it easy to share code, for example sharing the LeD method between the 3 subclasses of binary operators.

Pratt assumes dynamic scoping of variables, which would have been the norm in LISP implementations at the time. He also describes the main algorithm with a diagram that seems to miss some of the details.

I took the names LeD, NuD, and LBP from Pratt. I would have used other names myself, but I think it's worth the price of poor naming to be consistent with Pratt's and other presentations.

Implementation details and other design choices

One thing that might bother you is the need to have tables that map a large (perhaps infinite) set of tokens. For most applications, there is no need to change the tables at run time, so the tables can be implemented as a pair of procedures. (In OO terms we need a factory that has methods leftComm and nullComm.) And in most cases all we need to know is the category of token, not the actual token string; so we can use an array indexed by the token categories.

In some cases we do want to be able to change the tables during run time. For example, Haskell lets the programmer declare the precedence of binary operators; and most Prolog dialects allow one to declare new operators along with associativity and precedence. If we do want to change the table on the fly, a hash map can be used.

Having reduced the number of tables from 3 in Pratt's paper to 2, the next step might be to have one table and only one kind of command. I.e. ensure that every concrete command has all 4 methods (LeD, LBP, NBP, and NuD). I didn't do this, because we might want to vary the tables independently. Consider adding function

application syntax x(y) to the language. What does this have to do with parenthesization? Nothing really. Yet if I only had one command class for "(", it would have responsibilities for both parenthesization and function application. The treatment of a token when it's used as an N token and when it's used as an L token are two different responsibilities and shouldn't be in the same class.

Another approach is to do away with the tables and use objects as tokens that have the LeD, LBP, NBP, and NuD methods built into them. The comments of the previous paragraph apply.

As mentioned above, the NBP method is my own addition. It's used for making binary operators --and the like-- nonassociative. If you don't have need of it, don't implement it and change the E procedure to

```
E(p) is
    var token := next
    consume
    var t := nullComm[token].NuD( token )
    loop
        token := next
        const c := leftComm[token]
        exit unless p ≤ c.LBP
        consume
        t := c.LeD( t, token )
    return t
```

References

[0] Eli Bendersky, "Top-Down operator precedence parsing", January 2010, http://eli.thegreenplace.net/2010/01/02/top-down-operator-precedence-parsing

[1] Andy Chu, "Pratt Parsing and Precedence Climbing Are the Same Algorithm", Nov 2016. http://www.oilshell.org/blog/2016/11/01.html

[2] Andy Chu, "Review of Pratt/TDOP Parsing Tutorials", Nov 2016. http://www.oilshell.org/blog/2016/11/02.html

[3] Andy Chu, "Pratt Parsing Without Prototypal Inheritance, Global Variables, Virtual Dispatch, or Java". "http://www.oilshell.org/blog/2016/11/03.html"

[4] Keith Clarke, "The top-down parsing of expressions", Research Report 383, Dept of Computer Science, Queen Mary College. Archived at http://antlr.org/papers/Clarke-expr-parsing-1986.pdf.

[5] Theodore S. Norvell, "Parsing Expressions by Recursive Descent", 1999 (revised 2013), http://www.engr.mun.ca/~theo/Misc/exp_parsing.htm

[6] Vaughn R. Pratt, "Top down operator precedence", Proceedings of the 1st symposium on principles of programming languages (POPL), 1973, http://dl.acm.org/citation.cfm?id=512931

[7] Martin Richards and Collin Whitby-Stevens, *BCPL -- the language and its compiler*, Cambridge University Press, 1979.

Acknowledgement

Thanks to Andy Chu for pointing out the connection between precedence climbing and Pratt parsing.