Parsing Expressions by Recursive Descent

Theodore Norvell (C) 1999-2001

Parsing expressions by recursive descent poses two classic problems

1. how to get the abstract syntax tree to follow the precedence and associativity of operators and
2. how to do so efficiently when there are many levels of precedence

The classic solution to the first problem does not solve the second. I will present the classic solution, a well known alternative known as the "Shunting Yard Algorithm", and a less well known one that I have called "Precedence Climbing".

Contents

An example grammar for expressions

Consider the following example grammar, *G*,

```
E --> E "+" E
    | E "-" E
    | "-" E
    | E "*" E
    | E "/" E
    | E "^" E
    | "(" E ")"
    | v
```

in which `v` is a terminal representing identifiers and/or constants.

We want to build a parser that will

1. Produce an error message if its input is not in the language of this grammar.
2. Produce an "abstract syntax tree" (AST) reflecting the structure of the input, if the input is in the language of the grammar.

Each input in the language will have a single AST based on the following precedence and associativity rules:

- Parentheses have precedence over all operators.
- ^ (exponentiation) has precedence over /, *, -, and +.
- * and / have precedence over unary - and binary - and +.
- Unary - has precedence over binary - and +.
- ^ is right associative while all other operators are left associative.

For example the first three rules tell us that

```
a ^ b * c ^ d + e ^ f / g ^ (h + i)
```

parses to the tree

```
+( *( ^(a,b), ^(c,d) ), /( ^(e,f), ^(g,+(h,i)) ) )
```

while the last rule tells us that

```
a - b - c
```

parses to `-(-(a,b),c)` rather than `-(a,-(b,c))`, whereas

```
a ^ b ^ c
```

parses to `^(a, ^(b,c))` rather than `^(^(a,b), c)`.

The precedence of unary - over ^ tells us that

```
- a ^ - b
```

parses to -(^(a, -(b))). Some programming language designers choose to put unary operators at the highest level of precedence I took a different choice here because having some binary operators with higher precedence than some unary operators makes the parsing problem just a bit more challenging.

Aside: I am assuming that the desired output of the parser is an abstract syntax tree (AST). The same considerations arise if the output is to be some other form such as reverse-polish notation (RPN), calls to an analyzer and code generator (for one-pass compilers), or a numerical result (as in a calculator). All the algorithms I present are easily modified for these forms of output.

Recursive-descent recognition

The idea of recursive-descent parsing is to transform each nonterminal of a grammar into a subroutine that will recognize exactly that nonterminal in the input.

Left recursive grammars, such as *G*, are unsuitable because a left-recursive production leads to an infinite recursion in the recursive-descent parser. While the parser may be partially correct, it may not terminate.

We can transform *G* to a non-left-recursive grammar *G*1 as follows:

```
E --> P {B P}
P --> v | "(" E ")" | U P
B --> "+" | "-" | "*" | "/" | "^"
U --> "-"
```

The braces "{" and "}" represent zero or more repetitions of what is inside of them. Thus you can think of E as having an infinity of alternatives:

```
E --> P | P B P | P B P B P | ... ad infinitum
```

The language described by this grammar is the same as that of grammar *G*: *L(G*1*) = L(G)*.

Not only is left recursion eliminated, but the *G*1 is unambiguous and each choice can be made by looking at the next token in the input.

*Aside: Technically, G*1 *is an example of what is called an LL*(1) *grammar. I don't want to make this essay more technical than it needs to be, so I'm not going to stop and go into what that means. End of Aside.*

Let's look at a *recursive descent recognizer* based on this grammar. I call this algorithm a recognizer because all it does is to recognize whether the input is in the language of the grammar or not. That is it does not produce an abstract syntax tree, or any other form of output that represents the contents of the input.

I'll assume that the following subroutines exist:

- "next" returns the next token of input or special marker "end" to represent that there are no more input tokens. "next" does not alter the input stream.
- "consume" reads one token. When "next=end", consume is still allowed, but has no effect.
- "error" stops the parsing process and reports an error.

Using these, let's construct a subroutine "Expect", which I will use throughout this essay

```
expect( tok ) is
   if next = tok
       consume
   else
       error
```

We will now write a subroutine called "Erecognizer". If it does not call "error", then the input was an expression according to the above grammars. If it does call "error", then the input contained a syntax error, e.g. unmatched parentheses, a missing operator or operand, etc.

```
Erecognizer is
   E()
   expect( end )

E is
    P
    while next is a binary operator
```

```
        consume
        P

P is
    if next is a v
         consume
    else if next = "("
         consume
         E
         expect( ")" )
    else if next is a unary operator
         consume
         P
    else
         error
```

Notice how the structure of the recognition algorithm mirrors the structure of the grammar. This is the essence of recursive descent parsing.

The difference between a recognizer and a parser is that a parser produces some kind of output that reflects the structure of the input. Next we will look at a way to modify the above recognition algorithm to be a parsing algorithm. It will build an AST, according to the precedence and associativity rules, using a method known as the "shunting yard" algorithm.

The shunting yard algorithm

The idea of the shunting yard algorithm is to keep operators on a stack until we are sure we have parsed both their operands. The operands are kept on a second stack. The shunting yard algorithm can be used to directly evaluate expressions as they are parsed (it is commonly used in electronic calculators for this task), to create a reverse Polish notation translation of an infix expression, or to create an abstract syntax tree. I'll create an abstract syntax tree, so my operand stacks will contain trees.

The key to the algorithm is to keep the operators on the operator stack ordered by precedence (lowest at bottom and highest at top), at least in the absence of parentheses. Before pushing an operator onto the operator stack, all higher precedence operators are cleared from the stack. Clearing an operator consists of removing the operator from the operator stack and its operand(s) from the operand stack, making a new tree, and pushing that tree onto the operand stack. At the end of an expression the remaining operators are put into trees with their operands and that is that.

The following table illustrates the process for an input of x*y+z. Stacks are written with their tops to the left. The sentinel value acts as an operator of lowest precedence

Remaining input	Operand Stack	Operator Stack	Next Action
x * y + z end		sentinel	Push x on to the operand stack
* y + z end	x	sentinel	Compare the precedence of * with the precedence of the sentinel
* y + z end	x	sentinel	It's higher, so push * on to the operator stack
y + z end	x	binary(*) sentinel	Push y on to the operand stack
+ z end	y x	binary(*) sentinel	Compare the precedence of + with the precedence of *
+ z end	y x	binary(*) sentinel	It's lower, so make a tree from *, y, and x
+ z end	*(x,y)	sentinel	Compare the precedence of + with the precedence of the sentinel
+ z end	*(x,y)	sentinel	It's higher, so push + on to the operand stack
z end	*(x,y)	binary(+) sentinel	Push z on to the operator stack
end	z *(x,y)	binary(+) sentinel	Make a tree from +, z, and *(x,y)
end	+(*(x,y), z)	sentinel	

Compare this to parsing x + y * z.

Remaining input	Operand Stack	Operator Stack	Next Action
x + y * z end		sentinel	Push x on to the operand stack
+ y * z end	x	sentinel	Compare the precedence of + with the precedence of the sentinel
+ y * z end	x	sentinel	It's higher, so push + on to the operator stack
y * z end	x	binary(+) sentinel	Push y on to the operand stack
* z end	y x	binary(+) sentinel	Compare the precedence of * with the precedence of +.
* z end	y x	binary(+) sentinel	It's higher so, push * on to the operand stack
z end	y x	binary(*) binary(+) sentinel	Push z on to the operand stack
end	z y x	binary(*) binary(+) sentinel	Make a tree from *, y, and z
end	*(y, z) x	binary(+) sentinel	Make a tree from +, x, and *(y,z)
end	+(x, *(y, z))	sentinel	

In addition to "next", "consume". "end", "error", and "expect", which are explained in the previous section, I will assume that the following subroutines and constants exist:

- "binary" converts a token matched by B to an operator.
- "unary" converts a token matched by U to an operator. We require that functions "unary" and "binary" have disjoint ranges.
- "mkLeaf" converts a token matched by v to a tree.
- "mkNode" takes an operator and one or two trees and returns a tree.
- "push", "pop", "top": the usual stack operations.
- "empty": an empty stack
- "sentinel" is a value that is not in the range of either unary or binary.

In the algorithm that follows I compare operators and the sentinel with a > sign. This comparison is defined as follows:

- binary(x) > binary(y), if x has higher precedence than y, or x is left associative and x and y have equal precedence
- unary(x) > binary(y), if x has precedence higher or equal to y's
- op > unary(y), never (where op is any unary or binary operator)
- sentinel > op, never (where op is any unary or binary operator)
- op > sentinel (where op is any unary or binary operator): This case doesn't arise.

Now we define the following subroutines:

Aside: I hope the pseudo-code notation is fairly clear. I'll just comment that I'm assuming that parameters are passed by reference, so only 2 stacks are created throughout the execution of EParser.

```
Eparser is
   var operators : Stack of Operator := empty
   var operands : Stack of Tree := empty
   push( operators, sentinel )
   E( operators, operands )
   expect( end )
   return top( operands )

E( operators, operands ) is
    P( operators, operands )
    while next is a binary operator
       pushOperator( binary(next), operators, operands )
       consume
       P( operators, operands )
    while top(operators) not= sentinel
       popOperator( operators, operands )

P( operators, operands ) is
    if next is a v
         push( operands, mkLeaf( v ) )
         consume
    else if next = "("
         consume
         push( operators, sentinel )
         E( operators, operands )
         expect( ")" )
         pop( operators )
    else if next is a unary operator
         pushOperator( unary(next), operators, operands )
         consume
         P( operators, operands )
    else
         error

popOperator( operators, operands ) is
   if top(operators) is binary
        const t1 := pop( operands )
        const t0 := pop( operands )
        push( operands, mkNode( pop(operators), t0, t1 ) )
   else
        push( operands, mkNode( pop(operators), pop(operands) ) )

pushOperator( op, operators, operands ) is
    while top(operators) > op
```

```
      popOperator( operators, operands )
   push( op, operators )
```

The classic solution

The classic solution to recursive-descent parsing of expressions is to create a new nonterminal for each level of precedence as follows. *G*2:

```
E --> T {( "+" | "-" ) T}
T --> F {( "*" | "/" ) F}
F --> P ["^" F]
P --> v | "(" E ")" | "-" T
```

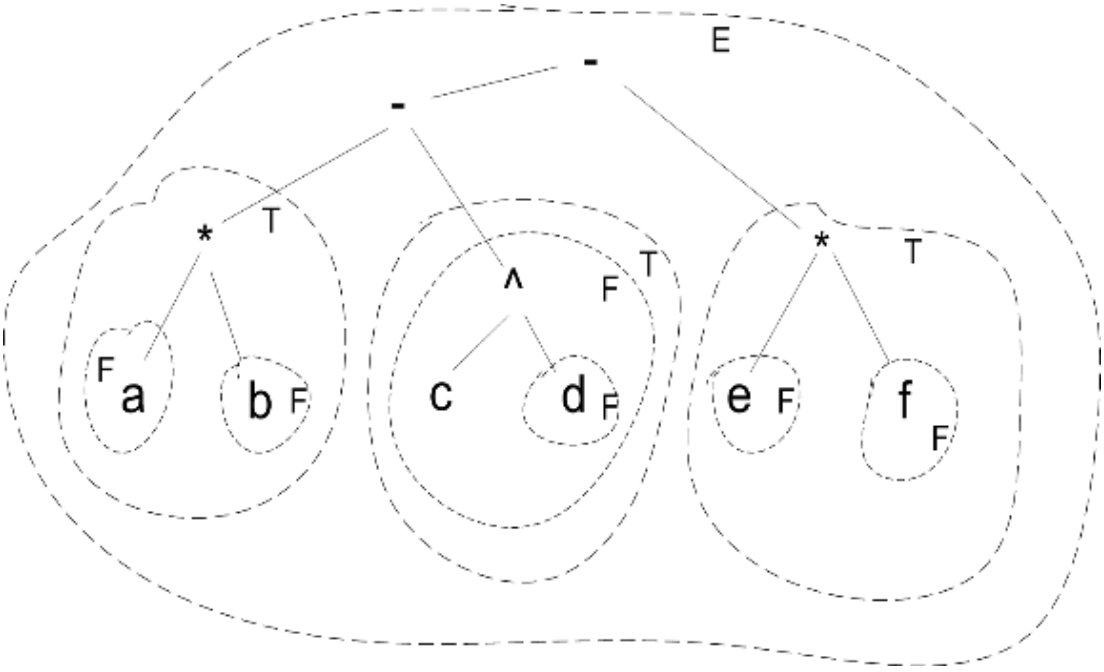
(The brackets [and] enclose an optional part of the production. As before, the braces { and } enclose parts of the productions that may be repeated 0 or more times, and | separates alternatives. The unquoted parentheses (and) serve only to group elements in a production.)

Grammar *G*2 describes the same language as the previous two grammars: *L(G2) = L(G1) = L(G)*

The grammar is ambiguous; for example, -x*y has two parse trees. The ambiguity is resolved by staying in each loop (in the productions for E and T) as long as possible and by taking the option if possible (in the production for F). With that policy in place, all choices can be made by looking only at the next token of input.

Note that the left-associative and the right-associative operators are treated differently; left-associative operators are consumed in a loop, while right-associative operators are handled with right-recursive productions. This is to make the tree building a bit easier.

Here is an example of parsing a*b - c^d - e*f



Each contour line shows what is recognized by each invocation of E, T, or F. For instance we can see that the top level call to E makes invokes T three times; these three invocations of T respectively recognize a*b, c^d, and e*f. Not shown are the calls to P, of which there is one for each variable.

We can transform this grammar to a parser written in pseudo code.

```
Eparser is
   var t : Tree
   t := E
   expect( end )
   return t

E is
   var t : Tree
   t := T
   while next = "+" or next = "-"
      const op := binary(next)
      consume
      const t1 := T
```

```
      t := mkNode( op, t, t1 )
   return t

T is
   var t : Tree
   t := F
   while next = "*" or next = "/"
      const op := binary(next)
      consume
      const t1 := F
      t := mkNode( op, t, t1 )
   return t

F is
   var t : Tree
   t := P
   if next = "^"
        consume
        const t1 := F
        return mkNode( binary("^"), t, t1)
   else
        return t

P is
   var t : Tree
   if next is a v
        t := mkLeaf( next )
        consume
        return t
   else if next = "("
        consume
        t := E
        expect( ")" )
        return t
   else if next = "-"
        consume
        t := F
        return mkNode( unary("-"), t)
   else
        error
```

It may be worthwhile to trace this algorithm on a few example inputs.

Although this is the classic solution, it has a few drawbacks:

- The size of the code is proportional to the number of precedence levels.
- The speed of the algorithm is proportional to the number of precedence levels.
- The number of precedence levels is built in.

When there are a large number of precedence levels, as in the C and C++ languages, the first two disadvantages become problematic. In Pascal the number of precedence levels was deliberately kept small because —I suspect— its designer, Niklaus Wirth, was aware of the shortcomings of this method when the number of precedence levels is large.

The size problem can be overcome by creating one subroutine that is parameterized by precedence level rather than writing a separate routine for each level. But the speed problem remains. Note that the number of calls to parse an expression consisting of a single identifier is proportional to the number of levels of precedence

Precedence climbing

A method that solves all the listed problems of the classic solution, while being simpler than the shunting-yard algorithm is what I call "precedence climbing". (Note however that we will climb *down* the precedence levels.)

Consider the input sequence

```
a ^ b * c + d + e
```

The E subroutine of the classic solution will deal with this by three calls to T, and by consuming the 2 "+"s, building a tree

+(+(result of first call, result of second call), result of third call)

We say that this loop directly consumes the two "+" operators.

The precedence climbing algorithm has a similar loop, but it always directly consumes the first binary operator, then it consumes the next binary operator that is of lower precedence, then the next operator that is of lower precedence than that. When it consumes a left-associative operator, the same loop will also consume the next operator of equal precedence. Let me rewrite the example with operators written at different heights according to their precedence:

```
          +   +
        *
    ^
  a   b   c   d   e
```

One loop can consume all 4 operators, creating the tree

+(+(*(^(result of first call, result of second call) result of 3rd call), result of 4th call), result of 5th call)

Each operator is assigned a precedence number. To make things more interesting lets add a few more binary operators and use the following precedence tables:

Unary operators

-	4

Binary operators

||	0	Left Associative
&&	1	Left Associative
=	2	Left Associative
+, -	3	Left Associative
*, /	5	Left Associative
^	6	Right Associative

We use the following grammar *G3* in which nonterminal Exp is parameterized by a precedence level. The idea is that Exp(p) recognizes expressions which contain no binary operators (other than in parentheses) with precedence less than p

```
E --> Exp(0)
Exp(p) --> P {B Exp(q)}
P --> U Exp(q) | "(" E ")" | v
B --> "+" | "-"  | "*" |"/" | "^" | "||" | "&&" | "="
U --> "-"
```

The loop implied by the braces, { and }, in the production for Exp(p) presents a problem: when should the loop be exited? This choice is resolved as follows:

- If the next token is a binary operator and the precedence of that operator is greater or equal to p,
  then the loop is (re)entered.
- Otherwise the loop is exited.

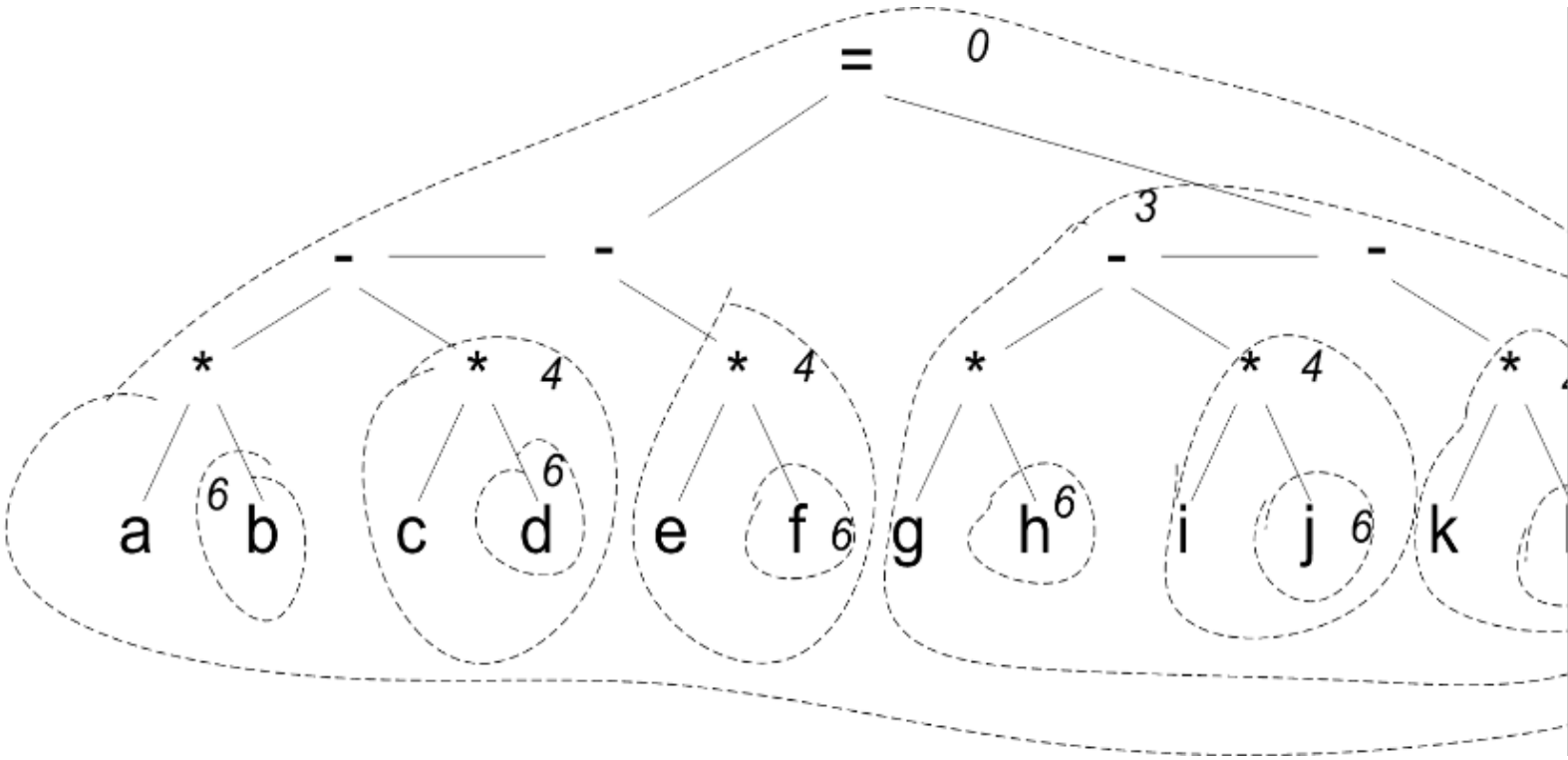
In the productions for Exp(p) and P, the recursive use of Exp is parameterized, by a value q. So there is a second choice to resolve: how is q chosen? The value of q is chosen according to the previous operator:

- In the binary operator case:
  - if the binary operator is left associative, q = the precedence of the operator + 1,
  - if the binary operator is right associative, q = the precedence of the operator.
- After unary operators,
  - q=the precedence of the operator.

Consider what will happen in parsing the expression, a * b - c * d - e * f = g * h - i * j - k * l. To make things clearer, I'll present this expression 2 dimensionally to show the precedences of the operators:

```
2                    =
3          -     -           -     -
5        *     *     *     *     *     *
        a b   c d   e f   g h   i j   k l
         0  0     0     0
```

The call to Exp(0) will consume exactly the operators indicated by a 0 underneath. The sub-expressions: a, b, c*d, e*f, and g*h-i*k-k*l will be parsed by calls to P and Exp(6), Exp(4), Exp(4) and Exp(3) respectively. The whole parse is illustrated by

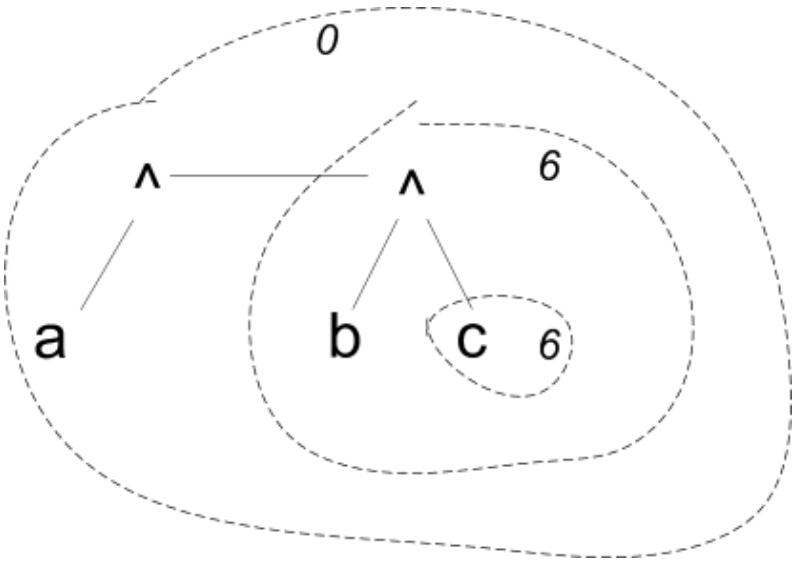


In this picture, each call to Exp is indicated by a dashed contour. The number immediately inside the contour indicates the value of the p parameter. Not shown are the calls to P, of which there is one for each variable, in this example.

What about right-associative operators? Consider an expression

```
a^b^c
```

Because of the different way right-associative operators are treated, Exp(0) will only consume the first ^, as the second will be gobbled up by a recursive call to Exp(6).



A recursive-descent parser based on this method looks like this:

```
Eparser is
   var t : Tree
   t := Exp( 0 )
```

```
   expect( end )
   return t

Exp( p ) is
    var t : Tree
    t := P
    while next is a binary operator and prec(binary(next)) >= p
       const op := binary(next)
       consume
       const q := case associativity(op)
                  of Right: prec( op )
                     Left:  1+prec( op )
       const t1 := Exp( q )
       t := mkNode( op, t, t1)
    return t

P is
    if next is a unary operator
         const op := unary(next)
         consume
         q := prec( op )
         const t := Exp( q )
         return mkNode( op, t )
    else if next = "("
         consume
         const t := Exp( 0 )
         expect ")"
         return t
    else if next is a v
         const t := mkLeaf( next )
         consume
         return t
    else
         error
```

Implementations

I've used precedence climbing in a JavaCC parser for a subset of C++. I've also used it in a parser based on monadic parsing written in Haskell. I'd be happy to mail either grammar to anyone who is interested.

Michael Bruce-Lockhart has implemented a table driven version of the precedence climbing algorithm. Download it here parser.js and parserTest.htm.

Alex de Kruijff has written an implementation of the precedence climbing algorithm as a Java library called Kilmop. You can find it here.

Bibliographic Notes

I'm not sure who invented what I am calling the classic algorithm. (Anyone know?) Certainly it was made popular by Niklaus Wirth who used it in various compilers, notably for Pascal. I learned it from one of Wirth's books.

The Shunting Yard Algorithm was invented by Edsger Dijkstra around 1960 in connection with one of the first Algol compilers. It is described in a Mathematisch Centrum report (starting around page 21). I think I first saw a version of it described in an ad for the TI-58/59 calculators, two of the earlier calculators to handle precedence

I first saw what I've called the precedence climbing method described by Keith Clarke in a posting to comp.compilers in 1992. It is closely related to the Top Down Operator Precedence method proposed by Vaughn Pratt in 1972.

Acknowledgement

Thanks to Colas Schretter for pointing out an error in the precedence climbing algorithm and suggesting a correction.