# Parsing Expressions by Recursive Descent

Theodore Norvell (C) 1999 with updates later on.

This article is about parsing expressions such as `a*b - a*d - e*f` using a technique known as recursive descent. I've assumed you know at least a little bit about context-free grammars and parsing.

Parsing expressions by recursive descent poses two classic problems

1. how to get the abstract syntax tree (or other output) 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
- Recursive-descent recognition
- The shunting yard algorithm
- The classic solution
- Precedence climbing
- Deriving precedence climbing
- Bibliographic Notes


## 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 unary - and the binary operators /, *, -, and +.
- * and / have precedence over unary - and binary - and +.
- Unary - has precedence over binary - and +.
- ^ is right associative while all other binary 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 binary ^ over unary - 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 chose to give unary - a lower precedence than *, /, and ^ because having some binary operators with higher precedence than some unary operators makes the parsing problem just a bit more challenging and raises some issues that otherwise wouldn't come up.

> 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 for recursive-descent parsing because a left-recursive production leads to an infinite recursion. While the parser may be partially correct, it may not terminate.

We can transform *G* to an equivalent 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*, that is, *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, G1 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* rather than a *parser* because all it does is to recognize whether the input is in the language of the grammar or not. 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 both their operands have been parsed. 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 operator 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 operator 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. (For example `unary("-")` and `binary("-")` are not equal.)
- "`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 )
```

Usually the shunting yard algorithm is presented without the use of recursion. This may be more efficient and might aid in generating better error messages, but I find the code a bit harder to understand.

## 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.)

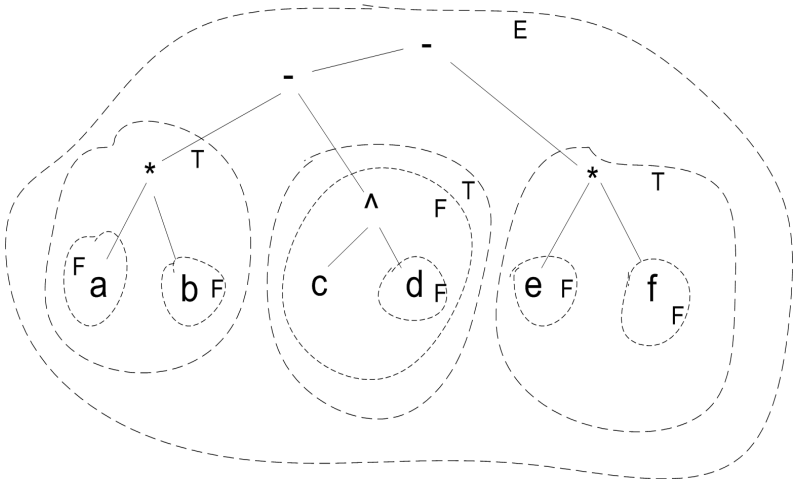
Grammar *G*2 describes the same language as the previous two grammars: $L(G2) = L(G1) = L(G)$

The grammar is ambiguous; for example, `-x*y` has two parse trees: `E(T(F(P("-", T(F(P("x")), "*")))))` and `E(T(F(P("-",T(F(P("x"))))),"*",F(P("y"))))`. 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 these policy in place, all choices can be made by looking only at the next token of input.

> Aside: If our precedence had been such that our unary operator had highest precedence, then the grammar would not have been ambiguous. For those who are interested in such things, I'll note that this grammar is not LL(1); LL(1) grammars are never ambiguous. Nevertheless everything works out just fine, if we adopt the policies mentioned in the previous paragraph. End of Aside.

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` by recursive descent.




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 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. Another way to look at it is that the contour lines show the parse tree (or would if I'd included the contour lines for P). The solid lines show the AST that we would like to be constructed.

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.

This classic solution 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 and the set of operators 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.

For languages in which the set of operators and their precedences and associativity are not hard-coded, we need a more flexible approach.

## Precedence climbing

A method that solves all the listed problems for 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 *G*3, 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.

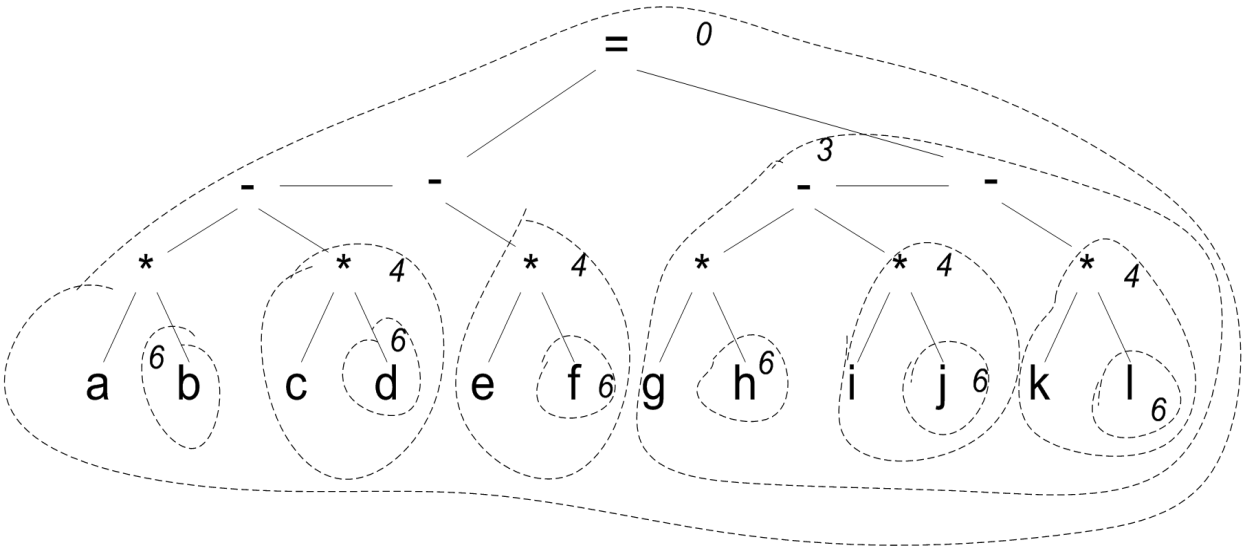
In the productions for `Exp(p)` and `P`, the recursive use of `Exp` is parameterized, by a value `q`. So there is a second choice to resolve: how is `q` chosen? The value of `q` is chosen according to the previous operator:

- 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 directly 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

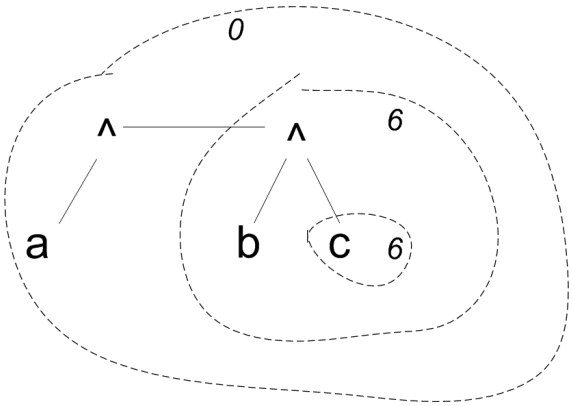



In this picture, each call to `Exp` is indicated by a dashed contour. The number immediately inside the contour indicates the value of the p parameter. Not shown are the calls to `P`, of which there is one for each variable, in this example.

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 directly 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.

Christian Kaps has created an implementation of the precedence climbing algorithm in PHP. You can find it at https://github.com/mohiva/pyramid.

Terrence Parr has implemented the idea of precedence climbing in version 4 of the ANTLR parser generator. You can read about it in his new book.

## Deriving precedence climbing

> As I write this section, it is 2013. I first posted the previous sections over 10 years ago. Lately I've thought a bit more about precedence climbing. I don't want to make the previous section any longer than it already is, so I'm adding this new section. For the interested, it presents an approach to deriving the algorithm by a combination

of grammar transformation and program transformation. Keith Clarke did a similar thing long ago [0], but I thought I'd take a crack at it myself. After completing this section, I read Clarke's paper and found that his approach and presentation is almost the same as mine. The main difference is that I generalize the algorithm to handle postfix operators and nonassociative operators. Also, as discussed later, Clarke uses a clever idea to avoid needing the variable I've called "r".

If you are not interested in this, then skip to the next section.

Let's start with a left-recursive grammar.

```
S --> E0 end
E0 --> E1 | E1 "=" E1
E1 --> E2 | E1 "+" E2
E2 --> E3 | E2 "*" E3
E3 --> E4 | E3 "!"
E4 --> E5 | E5 "^" E4
E5 --> P
P --> "-" E2 | "(" 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),
- a left-associative operator "+",
- a prefix operator "-"
- another left-associative operator "*"
- a postfix operator "!"
- a right-associative operator "^".

Nonassociative operators aren't found in many languages. That's because we can ban "a=b=c", but it is harder to ban "a=(b=c)". Nevertheless, they are easy to add to the grammar and make the problem just a bit more interesting.

This grammar is not unambiguous, but it almost is. The ambiguity is because "-" has lower precedence than "*" and "^". For example, `-a*b` can be parsed two ways: like `-(a*b)` and like `(-a)*b`. I won't worry about the ambiguity for now. I'll just note that the parse we want is the one that is like `-(a*b)`; later on, we'll check that our parser really does produce the right tree for `-a*b`. There are other ways of handling unary operators; one advantage of the present approach is that `a*-b` is in the language.

This grammar has both a prefix and postfix operators. But there is a subtle lack of symmetry between them. Note that `a^-b` is in the language of S, whereas `b!^a` is not. In essence, I'm treating postfix operators as binary operators that have no right operand. We'll come back to this issue at the very end of this section. Meanwhile, we'll leave the grammar as is.

Note that we can use the same symbol for a prefix or binary operators. (E.g., making '-' a binary operator as well as prefix is perfectly fine.) However, I'll assume that the sets of postfix and binary operators don't overlap. The reason is that it is easy to distinguish a unary from a binary operator based on the left context: unary operators follow the start of the expression, left parentheses, binary operators, and unary operators. It is not so easy to tell a binary operator from a postfix operator. Also if we allow operators to be unary, binary and postfix, there is more ambiguity. For example `a--b` is unambiguous if `-` is unary and binary, but ambiguous, if `-` is also postfix.

### First transformation: Eliminate left recursion and factor

There is a transformation rule that A --> a | A b can be written as A --> a {b} where {b} means 0 or more b. We can apply rule this to nonterminals E1, E2, and E3.

Another rule says we can rewrite A --> s | s b as A --> s [b], where [b] means 0 or 1 b. We apply this to nonterminals E0 and E4.

The result is

```
S --> E0 end
E0 --> E1 [ "=" E1 ]
E1 --> E2 { "+" E2 }
E2 --> E3 {"*" E3 }
E3 --> E4 { "!" }
E4 --> E5 [ "^" E4 ]
E5 --> P
P --> "-" E2 | "(" E0 ")" | v
```

At this point, if we convert to recursive descent code, we get the "classic" algorithm.

### Second transformation: Substitutions

Next we substitute right-hand sides for left-hand sides. We start by rewriting `E4`'s rule to `E4 --> P ["^" E4]`. Then `E3` can be rewritten as `E3 --> P ["^" E4] { "!" }` and we work backwards like this to `E0`. The result is

```
S --> E0 end
E0 --> P [ "^" E4 ] { "!" } {"*" E3 } { "+" E2 } [ "=" E1 ]
E1 --> P [ "^" E4 ] { "!" } {"*" E3 } { "+" E2 }
E2 --> P [ "^" E4 ] { "!" } {"*" E3 }
E3 --> P [ "^" E4 ] { "!" }
E4 --> P [ "^" E4 ]
P --> "-" E2 | "(" E0 ")" | v
```

(`E5` is no longer reachable, so I trashed it.)

It's a hard to believe we are making progress; bear with me.

### Third transformation: Compaction

Now we want to take advantage of the similarity of the rules for E0 through E4. To do this we need a bit more notation. First we'll use a parameterized nonterminal E: so `E(`*n*`)` below will match the same strings as E*n* above. Second, I'll use the notation `?(`*B*`)`, where *B* is a boolean expression, as a "test". The meaning of a test is that a particular alternative of the grammar can only be taken when *B* is true. For example `X [ ?(B) Y]` means the same as `X [Y]` when *B* is true, but means the same as `X` when *B* is false. Here is the compacted grammar.

```
S --> E(0) end
E(p) --> P [ ?(p≤4) "^" E(4) ] { ?(p≤3) "!" } { ?(p≤2) "*" E(3) } { ?(p≤1) "+" E(2) } [ ?(p≤0) "=" E(1
P --> "-" E(2) | "(" E(0) ")" | v
```

Now we are getting somewhere.

### Making a deterministic recognizer

The fourth transformation is easier to explain as a code transformation than as a grammar transformation, so, to get ready, we convert our grammar to a recognizer using the usual recursive descent magic. I won't add any tree building commands yet, as they clutter things up.

```
S is E(0) expect( end )

E(p) is
    P
    if p≤4 and next="^" then ( consume ; E(4) )
    while p≤3 and next="!" do  consume
    while p≤2 and next="*" do ( consume ; E(3) )
    while p≤1 and next="+" do ( consume ; E(2) )
    if p≤0 and next="=" then ( consume ; E(1) )

P is
    if next="-" then ( consume ; E(2) )
    else if next = "(" then ( consume ; E(0) ; expect( ")" ) )
    else if next is a v then consume
    else error
```

This is a deterministic algorithm for an ambiguous grammar: While converting from a grammar to a set of subroutines, I've made a decision about what happens in the ambiguous cases. So, even though there are no tree building commands, at this point we should stop and consider strings such as `-a*b` and `-a+b`, and see whether they are parsed "correctly". I.e., we should ask: if we did have tree building commands, would the right tree be built? Here is what happens for `-a*b` . `S` calls `E(0)`, `E(0)` calls `P`, and `P` consumes the "`-`" and then calls `E(2)`. After consuming "`a`", the call to `E(2)` sees the "`*`". The crucial decision is to consume the "`*`" and the "`b`" in the loop, rather than returning from `E(2)` as soon as "`a`" is consumed. The right tree is built -- or would be, if we were building trees. By contrast, in parsing `-a+b`, `E(2)` consumes the "`a`" and then must return; so it is the call to `E(0)` that consumes the "`+`". The tree we would get is the same as that for `(-a)+b`, just as desired.

### Fourth transformation: Combining the while loops and the if commands

The next transformation is the trickiest one. We want to combine the sequence of if commands and while loops in procedure E into one single while loop. If I have two identical loops sequentially, say

```
    while A do X
    while A do X    ,
```

we can rewrite them as one loop

```
    while A do X
```

What if the loops are different?. Suppose we have two successive **while** commands

```
    while A do X
    while B do Y  ,
```

and, furthermore, know that **not** `A` is a loop invariant of the second loop (i.e. that **not** `A` **and** `B` implies that executing `Y` leaves `A` false), then we can rewrite to

```
    while A or B do if A then X else Y
```

Similarly, if we have an **if** command followed by a **while** command

```
    if A then X
    while B do Y  ,
```

and, furthermore, know that **not** `A` is an invariant of the loop and that after executing `X`, `A` will be false, then we can rewrite to

```
    while A or B do if A then X else Y
```

What if there is no special relationship between the conditions and the commands? Then we can create one. We can use a counter to keep track of how many of the original loops and ifs our one loop is still emulating: For example

```
if A do X
while B do Y
while C do Z
```

can *always* be rewritten as

```
var r := 2
while 2≤r and A
   or 1≤r and B
   or 0≤r and C
do
    if 2≤r and A then      (X ; r := 1)
    else if 1≤r and B then (Y ; r := 1)
    else                   (Z ; r := 0)
```

where `r` is a fresh variable. The `r` variable acts as a ratchet; it prevents backsliding. Once `X` is executed, we set `r` to ensure `X` is never executed again. Once `Y` is executed, we set `r` to ensure that `X` is never executed in the future. Once `Z` is executed, we set r to ensure that neither `X` nor `Y` is executed in the future.

Applying this loop fusing idea to our latest `E` procedure we get

```
E(p) is
    P
    var r := 4
    while p≤4≤r and next="^"
       or p≤3≤r and next="!"
       or p≤2≤r and next="*"
       or p≤1≤r and next="+"
       or p≤0≤r and next="="
    do
        if p≤4≤r and next="^" then      (consume ; E(4) ; r := 3)
        else if p≤3≤r and next="!" then (consume ;        r := 3)
        else if p≤2≤r and next="*" then (consume ; E(3) ; r := 2)
        else if p≤1≤r and next="+" then (consume ; E(2) ; r := 1)
        else /* p≤0≤r and next="=" */   (consume ; E(1) ; r := -1)
```

### Fifth transformation: Making it table driven

Take a look at the latest `E` procedure. If we could eliminate the differences between the branches of the nested ifs, we wouldn't need any branching within the loop body. If we could eliminate the differences between the disjuncts of the loop guard, we could greatly simplify the loop guard. The differences come down to whether an operator is binary or postfix and three numbers:

- `prec(b)`: the precedence of the operator `b` (i.e., the highest value of `p` such that an invocation of `E(p)` can directly consume the operator),
- `rightPrec(b)`: the lowest precedence allowed for operators in `b`'s right operand, and
- `nextPrec(b)`: a value for `r` that prevents the loop from consuming operators whose precedence is too low.

For example, for + these numbers are respectively 1 (from `p≤1≤r and next="+"`), 2 (from `E(2)`), and 1 (from `r := 1`).

We make the following three tables.

| ***b*** | **=** | **+** | ***** | **!** | **^** |
|---|---|---|---|---|---|
| **prec(*b*)** | 0 | 1 | 2 | 3 | 4 |
| **rightPrec(*b*)** | 1 | 2 | 3 | NA | 4 |
| **nextPrec(b)** | -1 | 1 | 2 | 3 | 3 |

Now we rewrite the latest E procedure to use the tables and, after simplifying, we get

```
E(p) is
    P
    var r := 4
    while next is a binary or a postfix operator
      and p≤prec(next)≤r do
        const b := next
        consume
        if b is binary then E( rightPrec(b) )
        r := nextPrec(b)
```

This transformation is the one that really makes the algorithm both compact and efficient. The number of operations executed and the size of the code (excluding the table) are now both independent of the number of precedence levels.

We don't really need three tables. We can define `rightPrec(b)=1+prec(b)` when `b` is left-associative or nonassociative, and `rightPrec(b)=prec(b)` when `b` is right-associative. We can define `nextPrec(b)=prec(b)` when `b` is left-associative or postfix, and `nextPrec(b)=prec(b)-1` when `b` is right-associative or nonassociative. (If we have a postfix operator `$` and want to prevent, say, `a$$`", we make `nextPrec('$')=prec('$')-1`.

The really alert reader may notice that we didn't have the `r` variable or the `nextPrec` function in the previous section. We need them here because we have nonassociative operators and postfix operators. It also simplifies the argument that the transformation is correct. I'll leave it as an exercise to eliminate `r` when it's not needed. (Or you can read Clarke's paper [0].)

In the case of postfix operators, the use of `r` and `nextPrec` ensures that, for example, `a!^b` is an error, just as it is in the original grammar. If we want to allow such strings, we can set `nextPrec('!')` to 4. In the first version of procedure E, this amounts to a backwards jump like this:

```
E(p) is
    P
    L: if p≤4 and next="^" then ( consume ; E(4) )
    if p≤3 and next="!" then ( consume ; goto L)
    while p≤2 and next="*" do ( consume ; E(3) )
    while p≤1 and next="+" do ( consume ; E(2) )
    if p≤0 and next="=" then ( consume ; E(1) )
```

In general, the nextPrec for any postfix operator can be the precedence of the highest precedence operator. This allows a postfix operator to be followed by any binary or postfix operator, just as a prefix operator can follow any binary or prefix operator.

### Adding tree building operations

Finally, we add the tree building operations. Strictly speaking they should have been added as soon as we went to code, so that we could see that the right tree is built for ambiguous inputs and that subsequent transformations have no effect on the tree that is finally built.

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

E(p) is
    var t := P
    var r := 4
    while next is a binary or a postfix operator
      and p≤prec(next)≤r do
        const b := next
        consume
        if b is binary then
            const t1 := E( rightPrec(b) )
            t := mknode(binary(b), t, t1)
        else t := mknode(postfix(b), t)
        r := nextPrec(b)
    return t

P is
    if next="-" then ( consume ; const t:= E(2) ; return mknode(prefix('-', t)) )
    else if next = "(" then ( consume ; const t := E(0) ; expect(")") ; return t )
    else if next is a v then ( const t :=  mkleaf(next) ; consume ;  return t )
    else error
```

## Bibliographic Notes

Recursive descent seems to have been first described by Peter Lucas, who, with a team from IBM's Vienna laboratory, used it in their ALGOL 60 compiler. In his report [2], he writes

> The translator will be designed in such a way, that each meaning of a metalinguistic variable corresponds to a subroutine which carries out the transformation of the string of symbols.

Another early use of recursive descent parsing was in an ALGOL 60 Compiler for the Elliott Brothers' 803 and 503 computers; it was designed by a team of three programmers led by C. A. R. Hoare. From Hoare's report on that compiler:

> The main work is done by a set of procedures, each of which is capable of processing one of the syntactic units defined in the ALGOL 60 report. Where one syntactic unit is defined as consisting of other units, the procedure will be capable of activating other procedures, and where necessary, itself. For example, the procedure "compile arithmetic expression" must be capable of compiling the bracketed constituents of an arithmetic expression, which are themselves arithmetic expressions; this is achieved by a recursive entry to the very procedure "compile arithmetic expression" which is currently engaged on translating the whole expression. [3]

You can use this compiler yourself; see http://elliott803.sourceforge.net/docs/algol.html. You can read the (disassembled) compiler at http://www.billp.org/ccs/ElliottAlgol/.

I'm not sure who invented what I am calling the classic algorithm. (Anyone know?) It was made popular, I think, 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 apparently invented by Edsger Dijkstra around 1960 in connection with one of the first ALGOL 60 compilers. It is described in a Mathematisch Centrum report (starting around page 21) [1]. I say "aparently", as essentially the same algorithm is described by Friedrich. Bauer and Klaus Samelson in 1960 [5]. I think I first saw a version of it described in an ad for the TI SR-52 and SR-56 calculators, two of the earliest pocket calculators to handle precedence. (Prior to 1976, pocket calculators either used RPN or treated 2+3*4 as (2+3)*2.)

I first saw what I've called the precedence climbing method described by Keith Clarke in a posting to comp.compilers in 1992. Keith gives a proof of its correctness, relative to the classic algorithm, by means of program transformation in a 1985 report "The top-down

parsing of expressions" [0]. The algorithm appears to have been first invented by Martin Richards for use in the CPL and BCPL compilers. It can be found in the section 6.6 of *BCPL -- the language and its compiler* [4] and can still be found in recent distributions of BCPL's compiler. Neither Clarke nor Richards had a special name for the algorithm, so my one contribution (besides extending it to postfix and non-associative operators) is suggesting the name "precedence climbing".

It turns out that precedence climbing is a special case of a more flexible technique called Pratt parsing. Pratt parsing was first described in a paper by Vaughn Pratt [6]. This connection was brought to my attention by Andy Chu who wrote about it in a blog post [7]. I've explored this connection in my own web article From Precedence Climbing to Pratt Parsing [8].

### References

[0] 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.

[1] Edsger W. Dijkstra, "Algol 60 translation : An Algol 60 translator for the x1 and Making a translator for Algol 60", Research Report 35, Mathematisch Centrum, Amsterdam, 1961. Reprint archived at http://www.cs.utexas.edu/users/EWD/MCReps/MR35.PDF.

[2] Peter Lucas, "The Structure of Formula-Translators", ALGOL Bulletin, Issue Sup 16, Sep. 1961. Archived at http://archive.computerhistory.org/resources/text/algol/algol_bulletin/AS16/AS16.HTM.

[3] C. A. R. Hoare, "Report on the Elliott ALGOL translator", The Computer Journal, Vol. 5, No. 2, pp. 127-129, 1962. Archived at http://comjnl.oxfordjournals.org/content/5/2/127.short.

[4] Martin Richards and Collin Whitby-Stevens, *BCPL -- the language and its compiler*, 1st ed., Cambridge University Press, 1979.

[5] F. L. Bauer and K. Samelson, "Sequential formula translation", Communications of the ACM, vol 3, #2, February, 1960.

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

[8] Theodore S. Norvell, "From Precedence Climbing to Pratt Parsing", 2016, www.engr.mun.ca/~theo/Misc/pratt_parsing.htm

## Acknowledgement

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

I am grateful to Keith Clarke and Martin Richards for helping me trace the origins of what I've called precedence climbing and to Andy Chu for pointing out the connection to Pratt parsing.

Thanks to everyone who took the trouble to e-mail me to tell me that they found this article useful, to point me to an implementation, or to tell me that I've made an error.