383

The top-down parsing of expressions

Keith Clarke
Dept. of Computer Science and Statistics, Queen Mary College, London, E1 4NS.

1

Summary

Using grammar and program transformation a compact, efficient and easily used method of parsing infix expressions is derived. The resulting algorithm does not require the construction of precedence matrices, neither does it normally require transformation of the reference grammar for expressions. The method uses only two parsing procedures with functions which give the numerical precedence of each operator, and indicate which operators are left-associative and which right-associative.

KEYWORDS Recursive-descent parsing Expression parsing Operator-precedence parsing

Introduction

Infix expressions can easily be parsed by conventional recursive descent methods, but the resulting parser is very inefficient - even for a trivial expression a number of procedure calls equal to the number of precedence levels is needed. It has been stated[4] that the transformations needed to make the grammar suitable as the basis of such a parser are tedious, and the final parser too opaque. For these reasons compiler writers often recourse to operator-precedence parsers for expressions, embedded in a top-down parser for the rest of the language.

Richards[1] gives a procedure for parsing expressions which uses recursion rather than an explicit stack despite being derived using the operator-precedence technique. The procedure is essentially identical to the one given here (performing the same number of tests of each operator symbol), but requires the construction of the operator-precedence matrix. We show how the procedure can be derived by grammar- and program-transformation from the recursive descent method, using only numerical precedences and an indication of left- or right-associativity. The resulting program can parse an expression using a number of procedure calls approximately equal to the number of nodes in its abstract syntax tree, independent of the number of precedence levels in the grammar. The operator symbols of any particular expression grammar appear only in two small functions, which frequently removes the need for grammar transformation before the method can be used. Hanson[2] shows how the number of procedure definitions used in a recursive descent parser can be reduced, but does not show how the number of procedure activations can be reduced. For this reason, Hanson's method is not in fact equivalent to Richards'.

The algorithm

Two simple functions (Priority and RightAssoc) are used to map character values to numbers (representing precedences) and booleans, respectively. Examples are given below. These encode the usual relative precedences and associativities of addition, multiplication and exponentiation. Non-operator characters are assigned a precedence of zero, so that it is not necessary to specify what characters can follow an expression.

```
proc Priority(c) =
     case c of
            "+" => 1  □  "*" => 2  □  "↑" => 3
     otherwise 0
     endcase
endproc

proc RightAssoc(c) =
     case c of "↑" => true
     otherwise false
     endcase
endproc
```

The next two procedures complete the parser for formulae composed of these infix operators and single-character variable names, constructing the correct abstract syntax tree.

The notation used is similar to that of Bornat[3]. The keyword **initialise** is used for a declaration of an updatable location, initialised to the value of the expression given; **let** similarly declares a non-updatable variable. The two functions mkleaf and mknode construct new syntax tree nodes containing operands and operators respectively. Function CurrC returns the character at the current reading position, without side-effects. Similarly, function GetC returns the current character, but also advances the reading position by one. A version of Dijkstra's guarded commands (eg Gries[5]) has been used in the imperative parts of the program to express conditional execution and iteration.

An expression is parsed using an initial call E(1). It is clear by inspection that the number of calls of E is one more than the number of operator nodes of the syntax tree constructed, while the number of calls of P is equal to the number of leaves of the tree. Each use of parentheses in the expression parsed leads to one additional call of each procedure.

```
proc E(prec) =
  initialise p:=P();
  do Priority(CurrC()) ≥ prec →
       let oper = GetC();
       let oprec = Priority(oper);
       p := mknode(p, oper, E(if RightAssoc(oper) then oprec else oprec+1))
  od;
  return p
endproc

proc P() =
  case CurrC() of
  "(" =>
          begin
               GetC();
               let p=E(1);
               if GetC() ≠ ")" → fail fi;
               return p
          end
  otherwise
          return mkleaf(GetC())
  endcase
endproc
```

The structure of the parsing routine is directly related to the ambiguous operator grammar:

$$E = P \{ "+" E \mid "*" E \mid "\uparrow" E \}$$
$$P = "(" E ")" \mid letter$$

where curly brackets indicate zero or more repetitions of their contents, and vertical bar indicates choice. The analyser uses 'pragmatic' information about the operators to guide a deterministic one-track parse in such a way that simple formulae are recognised more efficiently than with conventional one-track analysers based on unambiguous grammars.

Conventional top-down parsing

There is a standard technique (eg. Aho[4]) for constructing a top-down parser for infix expressions, given an ambiguous grammar and precedence/associativity information. A grammar rule is introduced for each level of precedence. For each level, either all the operators are left associative or else they are right associative. In the former case the rule takes the form:

$$E_n = E_{n+1} \{ O_n E_{n+1} \}$$

where n is the precedence of the operators O_n introduced in this rule. For right associative operators a right-recursive rule is used:

$$E_n = E_{n+1} [O_n E_n]$$

Square brackets indicate an optional element. The rule for the highest precedence level uses the rule for expression primaries, which we have already seen, in place of the reference to a greater precedence level.

eg. for the operators as used in the program above:

```
E1 = E2 { "+" E2 }
E2 = E3 { "*" E3 }
E3 = P [ "↑" E3 ]
P = "(" E1 ")" | id
```

Derivation of the parser

This is shown using the example grammar already introduced. The reader is invited to confirm that at no step is a particular property of the grammar used in any way that would reduce the generality of the result. We proceed by substituting, in each of the rules for expressions, for the leading non-terminal symbol, giving:

```
E1 = P [ "↑" E3 ] { "*" E3 } { "+" E2 }
E2 = P [ "↑" E3 ] { "*" E3 }
E3 = P [ "↑" E3 ]
P = "(" E ")" | id
```

In the programs below the construction of the parse tree is not shown and as the definitions of P, Priority and RightAssoc are the same in each example they are not repeated.

The analyser produced from these rules is even larger than that produced directly, but is amenable to optimisation. Consider for example the procedure for rule E1:

```
proc E1() =
   P();
   if CurrC()="↑" → GetC(); E3() fi;
   do CurrC()="*" → GetC(); E3() od;
   do CurrC()="+" → GetC(); E2() od
endproc
```

This would be the procedure called, say, to parse the right hand side of an expression. For example input such as "x:=0" the symbol following the zero would be tested three times before the procedure returned. However, it is possible to collapse the succession of tests into one.

Clearly the final loop can only be executed in an initial state such that the current input character is not "*". This is an invariant of the loop, since the procedure E2 finishes with another loop that also establishes this state. There is a general result for loop-transformations that is useful here: provided it is not possible for B and C to be true at the same time, and not B is an invariant of instruction T, the following two programs are equivalent.

```
    I: do B→S od; do C→T od
and
    J: do B→S  □ C→T od
```

Assuming termination, program I executes statement S a finite number of times, then executes statement T a finite number of times. In general, program J produces an interleaving of executions of S and T. But the invariance of not B means that no further executions of S are possible after the first execution of T. The requirement that B and C cannot be true at the same time ensures that program J is deterministic. Both programs produce the state not B solely by repeated execution of S, so the number of executions must be the same. Similarly given the same initial state the number of executions of T must be the same.

Another useful, standard result[3] is that:

```
    do B→S □ C→T od
```

is equivalent to

```
    do B or C → if B→S □ C→T fi od
```

Using these results the two loops of procedure E1 can be transformed, eventually giving the following:

```
    proc E1() =
       P();
       if CurrC()="↑" → GetC(); E3() fi;
       do CurrC() ∈ {"*", "+"} →
             case GetC() of "*" => E3()  □  "+" => E2() endcase
       od
    endproc
```

The if statement is replaced by a loop since the call E3() ensures that such a loop can never iterate more than once. Then a similar argument to the above allows us to merge the resulting loop with the final one, giving:

```
proc E1() =
   P();
   do CurrC() ∈ {"↑", "*", "+"} →
      case GetC() of "↑" => E3()  □ "*" => E3()  □  "+" => E2() endcase
   od
endproc
```

The two other procedures E2 and E3 are readily transformed to:

```
proc E2() =
   P();
   do CurrC() ∈ {"↑", "*"} →
      case GetC() of "↑" => E3() □ "*" => E3() endcase
   od
endproc

proc E3() =
   P();
   do CurrC() ∈ {"↑"} →
      case GetC() of "↑" => E3() endcase
   od
endproc
```

Obviously the most general version of the case statement could be used in all three procedures. The condition-parts of the loops are encoded using tests of numerical precedences. In E1, E2 and E3 the substitution is Priority(CurrC()) ≥ 1, Priority(CurrC()) ≥ 2 and Priority(CurrC()) ≥ 3, respectively.

The three procedures are now identical, except in the use of the constants 1, 2 and 3 respectively in procedures E1, E2 and E3. We therefore parameterize on this number, collapsing the procedures into one:

```
proc E(n) =
   P();
   do Priority(CurrC()) ≥ n →
      case GetC() of "↑" => E(3) □ "*" => E(3) □ "+" =>  E(2) endcase
   od
endproc
```

This version is further simplified and generalised on the basis of two observations. First, the call following each left associative operator is to the procedure for the next higher precedence. Second, the call after a right associative operator is found is to the procedure for the same precedence. This is a direct consequence of the way the unambiguous grammar was constructed. Thus,

```
proc E(n) =
   P();
   do Priority(CurrC()) ≥ n →
      let oper = GetC();
      let oprec = Priority(oper);
      E(if RightAssoc(oper) then oprec else oprec+1)
   od
endproc
```

This procedure no longer depends directly on the operator symbols or the number of precedence levels.

User-declared infix operators

In many applications the precedence and associativity functions could, for efficiency, be encoded as tables indexed on characters or lexical items. Alternatively, the syntactic properties of each operator could be stored in the symbol table. This facilitates the provision of user-defined infix operators. Precedence and infix status 'declarations' are properly regarded as compiler directives, and need to be recognised by the parser. If it is required to make such directives follow the usual scope rules, the parser will also have to undo the effects of the directives on block exit.

Unary prefix operators

These are conveniently recognised by the procedure P - this is simplest if such operators bind more tightly than any infix operator (as in Algol 68, but not in Pascal). The extra grammar productions required appear as new alternatives in the rule for primaries (ie. P), and hence each prefix operator is recognised by its own guard in the **case**-statement in procedure P. If it is required that "- - 3" be interpreted as "-(-3)", the following production may be used:

$$P = \text{"-"}\ P \mid id \mid \text{"("}\ E_1\ \text{")"}$$

It is possible to arrange that "-3↑4" be interpreted as "-(3↑4)", but "-3+4" as "(-3)+4", using a procedure for primaries derived from:

$$P = \text{"-"}\ E_2 \mid id \mid \text{"("}\ E_1\ \text{")"}$$

Thus one may define a language in which "-maxint+maxint" evaluates to zero without overflow, and also

"-3↑2" evaluates to minus nine. The Pascal productions[6] that exclude "4*-3" from the language are difficult to enforce efficiently using the present method. It is not unknown for compilers to accept such 'superlanguage' features, perhaps because it is hard to imagine a useful error report.

Operator precedence

The method described here requires that all the operators for a particular precedence level have the same associativity. The operator precedence method does not have this restriction; see for example the rules for the shift operators in BCPL[1]. However, although production of the operator precedence matrix is usually considered trivial, there is no obvious relationship between the grammar and the final parser. Previously[4] there has often been some doubt as whether or not an operator precedence parser 'accepts exactly the desired language'. Bornat[7] gives a useful discussion of the use of recursion in operator precedence parsing, together with a readily accessible, simplified version of Richards' algorithm[1].

Conclusion

The recursive descent technique makes possible the construction of a compact and easily comprehended parser directly from the usual informal description of the expression part of a programming language. No construction of precedence matrices is necessary. It is gratifying that the resulting parser is more efficient than one directly constructed from an unambiguous grammar, although the gain is in practice small. On favourable input (expressions involving only two levels of precedence out of a possible eight), using Pascal for the implementation, an improvement of about 25% in the execution time of a simple calculator was obtained. Simple expressions, on which the algorithm does well, are of course more commonly encountered than complex ones.

Since the operator symbols are recognised as such solely by their precedence, user-declaration of infix operators is trivial, and carries no executional overhead.

In contrast with other top-down techniques, additional levels of precedence can be added without changing the cost of parsing simple or complex expressions.

Many language definitions use ambiguous expression grammars, disambiguated by 'informal' remarks about the precedences and associativities of the various operators. The algorithm shown can be used in these cases without modification, by encoding the informal remarks into two simple functions.

References

1. M. Richards, and C. Whitby-Strevens, *BCPL - the language and its compiler*, Cambridge University Press, Cambridge, 1979.

2. D.R. Hanson, 'Compact Recursive-descent Parsing of Expressions', *Software Practice and Experience*, vol 15(12), 1205-1212 (1985).

3. R. Bornat, *Programming from First Principles*, Prentice-Hall, London, 1986. (to appear)

4. A.V. Aho, R. Sethi and J.D.Ullman, *Compilers: Principles, Techniques and Tools* Addison-Wesley, 1985.

5. D. Gries, *The Science of Programming*, Springer-Verlag, New York, 1981.

6. K. Jensen and N. Wirth, *Pascal: User Manual and Report*, Springer-Verlag, New York, 1978.

7. R. Bornat, *Understanding and Writing Compilers* Macmillan, London, 1979.