SOFTWARE—PRACTICE AND EXPERIENCE, VOL. 15(12), 1205–1212 (1985)

Compact Recursive-descent Parsing of Expressions

DAVID R. HANSON
Department of Electrical Engineering and Computer Science, Princeton University, Princeton, NJ 08544 U.S.A.

SUMMARY

Compiler writing tools, such as parser generators, are commonly used to build new operators. Nevertheless, constructing compilers by hand remains common. Such compilers are often built using recursive-descent parsing for most of the language and operator-precedence parsing for expressions. This paper describes a simple technique for parsing expressions using recursive descent that avoids the usual proliferation of procedures that occurs when recursive descent is used to parse expressions. By taking advantage of the similarity of the productions describing expressions in most languages, the $n + 1$ procedures usually required to parse expressions with n precedence levels can be replaced with a table and two procedures.

KEY WORDS Recursive-descent parsing Expression parsing Precedence Compilers

INTRODUCTION

Although most new compilers are written using compiler writing tools, such as LEX[1] and YACC,[2] some continue to be written by hand. Also, manual techniques are likely to be used for other language processing applications, such as command languages, program generators and preprocessors. Typically, bottom-up parsers, such as LALR(1),[3] and top-down parsers, such as LL(1), are used in conjunction with parser generators. For parsers constructed by hand, recursive descent is the method of choice because it is easy to understand and implement, and it is reasonably efficient.[3,4]

Although recursive descent is the primary technique used in hand-constructed compilers, expressions are often parsed using operator precedence. Recursive descent requires additional non-terminals in order to encode the various levels of operator precedence. These non-terminals are unnecessary for operator-precedence parsing, but constructing a correct precedence table is often as difficult as introducing the correct non-terminals (see Section 5.3 of Reference 3).

The remainder of this paper describes a simple technique for parsing expressions using recursive descent without introducing additional non-terminals. The technique takes advantage of the similarity of the productions for the expression component of most languages, replacing repetitious code by a simple table ordered according to precedence.

This technique has been used by other compiler writers, but documentation of its use

* Permanent address: Department of Computer Science, The University of Arizona, Tucson, AZ 85721 USA.

0038–0644/85/121205–08$01.00

Received 25 February 1985
Revised 23 April 1985

is rare. A similar technique was used in BCPL and is described in Reference 5. Although the BCPL technique is technically equivalent to the scheme described here, there are two significant differences. First, in the BCPL scheme, the operators and their precedences and associativities are spread throughout the code instead of being encapsulated in tables as described below. Thus, to add operators, or to change any of the operators, the code must be understood and changed accordingly in contrast to simply modifying table entries. Secondly and perhaps more importantly, the technique described below is derived from an LL(1) grammar and straightforward recursive-descent procedures using program and grammar transformations. No equivalent transformation is given for the BCPL approach, which is presented by giving the resulting code.

PARSING EXPRESSIONS

Consider expressions defined by the following grammar where terminals are given in sanserif face.

⟨expr⟩ → ⟨expr⟩ + ⟨expr⟩ | ⟨expr⟩ * ⟨expr⟩ | (⟨expr⟩) | id

In most languages, * has higher precedence than +, so expressions such as id+id*id are interpreted as id+(id*id). To achieve this interpretation, additional non-terminals are introduced for each precedence level and the grammar above becomes

⟨expr⟩ → ⟨expr⟩ + ⟨term⟩ | ⟨term⟩
⟨term⟩ → ⟨term⟩ * ⟨factor⟩ | ⟨factor⟩
⟨factor⟩ → (⟨expr⟩) | id

In order to write the recursive procedures corresponding to these productions, the left recursion must be removed. Using standard techniques,[3] additional non-terminals are introduced and the grammar above becomes

⟨expr⟩ → ⟨term⟩ ⟨expr'⟩
⟨expr'⟩ → + ⟨term⟩ ⟨expr'⟩ | ϵ
⟨term⟩ → ⟨factor⟩ ⟨term'⟩
⟨term'⟩ → * ⟨factor⟩ ⟨term'⟩ | ϵ
⟨factor⟩ → (⟨expr⟩) | id

Using an extended BNF in which optional repetition is indicated by braces,[6] the primed productions can be eliminated and the final grammar is

⟨expr⟩ → ⟨term⟩ {+⟨term⟩}
⟨term⟩ → ⟨factor⟩ { * ⟨factor⟩ }
⟨factor⟩ → (⟨expr⟩) | id

Note, however, that the primed productions were necessary to arrive at this final grammar.

These productions can be used to guide the implementation of the corresponding procedures. In the following C[7] versions of the necessary procedures, lex() returns the next token, which is usually assigned to the global variable t. Details, such as inserting identifiers into a symbol table, type-checking, constructing abstract syntax trees and extensive error processing, have been omitted.

```
expr() {
    term();
    while (t= = '+') {
        t = lex();
        term();
        }
}

term() {
    factor();
    while (t = = '*') {
        t = lex();
        factor();
        }
}

factor() {
    if (t = = '(') {
        t = lex();
        expr();
        if (t ! = ')')
            error ("missing)");
        else
            t = lex();
        }
    else if (t = = ID)
        t = lex();
    else
        error("unrecognized expression");
}
```

In general, for n precedence levels there are $n + 1$ non-terminals; one for each level and one for the 'basis' case in which further division is impossible. Consequently, there are $n + 1$ procedures — one for each non-terminal. If the binary operators are all left associative, these procedures are very similar. As illustrated by expr and term above, the only essential differences are the operators expected and the procedure to be called; procedure k calls procedure $k + 1$.

This similarity can be exploited to replace procedures 1 to n by a single procedure and a table of operators ordered according to increasing precedence. Using a two-dimensional matrix in which each row represents one precedence level, the expr and term procedures given above can be replaced by the following code. Expression parsing is initiated by calling expr(0); the expr() in factor must be changed to expr(0).

```
int precedence[3][2] = {
    '+', 0,
    '*', 0,
    0, 0
};

expr(k)
int k;
{
    if (precedence[k][0] == 0)
        factor();
    else {
        expr(k + 1);
        while (isop(t, k)) {
            t = lex();
            expr(k + 1);
            }
        }
}

int isop (t, i)
int t, i;
{
    int j;

    for (j = 0; precedence[i][j] !=0; j = j + 1)
        if (t == precedence[i][j]
            return[1];
    return (0);
}
```

The two procedures, expr and factor and the associated table look-up procedure, isop, can be used for any expression grammar of the form

$$\langle \text{expr} \rangle \rightarrow \langle \text{expr} \rangle \text{ op } \langle \text{expr} \rangle \mid \langle \text{factor} \rangle$$

where ops are binary, left-associative operators. Adding additional operators is accomplished by expanding precedence; for example, most of the binary operators in C can be handled with these procedures using the table

```
int precedence[11][5] = {
    OR, 0, 0, 0, 0,
    AND, 0, 0, 0, 0,
    '|', 0, 0, 0, 0,
    '^', 0, 0, 0, 0,
    '&', 0, 0, 0, 0,
    EQ, NE, 0, 0, 0,
    '<', LE, GE, '>', 0,
    LSH, RSH, 0, 0, 0,
```

```
        '+', '-', 0, 0, 0,
        '*', '/', '%', 0, 0,
        0, 0, 0, 0, 0,
};
```

where OR, AND, EQ, NE, LE, GE, LSH and RSH are symbolic constants representing the tokens ||, &&, ==, !=, <=, >=, << and >> respectively. Other table representations can be used to avoid the wasted space and the linear search in the two-dimensional table, if necessary. For most applications, however, the additional space and time required are negligible.

DISCUSSION

Additional information can be stored in the precedence table or in companion tables to handle different kinds of operators and other tasks that must be performed during parsing. For example, error reporting and recovery can be incorporated into the technique by including additional information in the precedence table. A common approach is to pass sets of 'follow' and 'stopping' symbols to each parsing procedure for use in error recovery.[6] This approach can be implemented by adding these sets to each row of the precedence table. Similar additions can be made to accomodate other common error detection and recovery techniques.[4, 8]

Right-associative operators, such as exponentiation and assignment, can be handled by including an association table along with the precedence table. Left-associative operators give rise to left-recursive productions, such as those for ⟨expr⟩ and ⟨term⟩ given above. Right-associative operators, on the other hand, give rise to right-recursive productions. For example, exponentiation can be added to the example grammar by adding another non-terminal and the association productions:

⟨term⟩ → ⟨term⟩ * ⟨primary⟩ | ⟨primary⟩
⟨primary⟩ → ⟨factor⟩ ^ ⟨primary⟩ | ⟨factor⟩

The typical procedure for ⟨primary⟩ is

```
primary()
{
    factor();
    while (t == '^') {
        t = lex();
        primary();
        }
}
```

The call to primary in the 6th line corresponds to calling expr(k) instead of expr(k + 1) in the 10th line of expr. Assuming all operators at each precedence level have the same associativity, this difference can be encoded in a table in which each entry gives the associativity of the operators in the corresponding row in the precedence table. Only expr requires modification, and the additional non-terminals — ⟨primary⟩ in the example

above — are unnecessary. For example, the following implementation handles the common arithmetic operators. LEFT and RIGHT are symbolic definitions for the constants 1 and 0 respectively, and elements of associativity are associated with the corresponding rows in precedence.

```
int precedence [4][3] = {
    '+', '-', 0,
    '*', '/', 0,
    '^', 0, 0,
    0, 0, 0,
};
int associativity[4] = {LEFT, LEFT, RIGHT, 0};
expr(k)
int k;
{
    if (precedence [k][0] == 0)
        factor();
    else {
        expr(k+1);
        while (isop(t, k)) {
            t = lex();
            expr(k + associativity[k]);
            }
        }
}
```

Slightly more complicated table arrangements can be used to handle languages in which associativity varies on a per-operator basis.

Unary operators can also be handled using this parsing technique. Fortunately, unary operators have very high precedence in most programming languages, so they appear in the $(n + 1)$st procedure, such as factor in the example above. Otherwise, the operator tables and expr can be changed so that, upon entry, expr checks for the occurrence of unary operators at the kth level.

Likewise, semantic processing and code generation tasks can also be included in expr, and the associated operator-specific information can be added to the tables. For example, it is common to call operator-specific semantic routines when a subexpression is recognized and to propagate the results of these routines to be enclosing expressions by passing the results in a semantic record.[3] Such routines can be obtained from tables similar to associativity, for example, or by tables indexed by operator. The following version of expr illustrates this approach. The results of semantic routines are propagated by passing pointers to node structures, and the routines are obtained from the array semantics in which each entry corresponds to a row in precedence. For example, the following table contains routine names for the operators in the above version of precedence.

```
struct node *(*semantics[4])() = { &addop, &mulop, &expop, 0}
```

This declaration declares and initializes an array of pointers to functions that return pointers to nodes. nodes are also created, initialized and returned by factor.

```
struct node *expr(k)
int k;
{
    struct node *p, *factor();
    int op;

    if (precedence[k][0] == 0)
        p = factor();
    else {
        p = expr(k + 1);
        while isop (t, k)) {
            op = t;
            t = lex();
            p = (*semantics(k))(op, p, expr(k + associativity[k]));
            }
        }
    return (p);
}
```

*semantics[k] is the semantic processing function for operators at level k and is invoked to perform operator-specific semantic processing, such as type checking, and, perhaps, to emit code. For the simple case in which the semantics amounts to building an abstract syntax tree of an expression as it is parsed, *semantics[k] can be replaced by a function that constructs a node for the operator op with the given operands. Other table arrangements in which functions are associated with individual operators can be used in more complicated situations.†

In most cases, using parser generators and related compiler writing tools is preferable to hand construction. In the absence of such tools, however, the technique described here facilitates the construction of a recursive-descent parser for expressions that is compact and easy to understand, write and modify. This technique has been used successfully in compilers for the Y[9] and *EZ*[10] programming languages and for a subset of C. As these compilers demonstrate, this technique permits the effort to devise the appropriate grammar to be done once; the resulting code can be instantiated repeatedly.

ACKNOWLEDGEMENT

This work was supported in part by the U.S. National Science Foundation under Grant MCS–8302398.

† In such situations, experienced C programmers tend to encapsulate all of the information for each operator into a structure and combine precedence, associativity and semantics into a single table of structures and provide the necessary look-up functions.

REFERENCES

1. M. E. Lesk, 'LEX — a lexical analyzer generator', *Computing Science Tech. Rep. 39*, Bell Laboratories, Murray Hill, NJ, 1975.
2. S. C. Johnson, 'YACC — yet another compiler compiler', *Computing Science Tech. Rep. 32*, Bell Laboratories, Murray Hill, NJ, July 1975.
3. A. V. Aho and J. D. Ullman, *Principles of Compiler Design*, Addison Wesley, Reading, MA, 1977.
4. A. J. T. Davie and R. Morrison, *Recursive Descent Compiling*, Wiley, New York, NY, 1981.
5. M. Richards and C. Whitby-Strevens, *BCPL — The Language and Its Compiler*, Cambridge University Press, Cambridge, 1979.
6. N. Wirth, *Algorithms + Data Structure = Programs*, Prentice Hall, Englewood Cliffs, NJ, 1976.
7. B. W. Kernighan and D. M. Ritchie, *The C Programming Language*, Prentice Hall, Englewood Cliffs, NJ, 1978.
8. D. A. Turner, 'Error diagnosis and recovery in one-pass compilers', *Inf. Proc. Letters*, **6**, (4) 113–115 (1977).
9. D. R. Hanson, 'The Y programming language', *SIGPLAN Notices*, **16**, (2) 59–68 (1981).
10. C. W. Fraser and D. R. Hanson, 'A high-level programming and command language', *Proc. SIGPLAN '83 Symp. on Programming Language Issues in Software Systems*, San Francisco, CA, June 1983, pp. 212–219.

ERRATA: David R. Hanson, ‘Compact Recursive-descent Parsing of Expressions’, *Software—Practice & Experience*, **15**, 12 (Dec. 1985), 1205–1212.

There are several minor errors in this paper due to printer's errors.

The **`if`** statement in lines 6–7 of the function **`isop`** on p. 1208 is missing a right parenthesis and the expression in the **`return`** statement contains a syntax error. The complete statement should read

```
if (t == precedence[i][j])
      return (1);
```

The **`while`** loop in lines 10–14 of **`expr`** on p. 1211 is missing a left parenthesis and references the array **`semantics`** incorrectly; the complete loop should read

```
while (isop(t, k)) {
      op = t;
      t = lex();
      p = (*semantics[k])(op, p, expr(k + associativity[k]));
      }
```