OPERATOR PRECEDENCE PARSING

Tom Niemann

ePaper Press

PREFACE

While ad-hoc methods are sometimes used for parsing expressions, a more formal technique using operator precedence simplifies the task. For example an expression such as

```
(x*x + y*y)^.5
```

is easily parsed. Operator precedence parsing is based on bottom-up parsing techniques and uses a precedence table to determine the next action. The table is easy to construct and is typically hand-coded. This method is ideal for applications that require a parser for expressions and embedding compiler technology, such as yacc, would be overkill. The presentation is intuitive with examples coded in ANSI-C. The online version of this document also includes an example in Visual Basic. Source code may be downloaded at the web site listed below. An excellent and more theoretical treatment may be found in Compilers, Principles, Techniques and Tools, by Aho, Sethi, and Ullman.

> In 1976 I was teaching computer science at Diablo Valley College — a 2-year community college. An HP time-sharing computer was connected to 20 terminals that were all lodged in a terminal room that was adjacent to my office. During off hours it was a gathering place for students to experiment with their own projects and a fair amount of comraderie ensued. Evesdropping I picked up on the fact that they were all trying to write a program to parse expressions. In quick order I wrote my own program, based on the concepts presented here, and announced a challenge. In one week they were to choose their fastest parser and we would have a contest to see if it bested my efforts.
>
> On the day of the contest I was seated next to my challenger. Surrounded by the rest of the gang we entered the same rather lengthy expression on our terminals. At the count-down we both pressed the Enter key at the same time. Well, he pressed his but I delayed for a half second before pressing mine thus giving my opponent an advantage. A roar went up from the crowd when my program finished first by a clear margin. After listing the program on the printer they were struck by how short and simple it was. Better than dozens of if statements!

TOM NIEMANN
Portland, Oregon

web site: epaperpress.com

1. Theory, Part I

Operator precedence parsing is based on bottom-up shift-reduce parsing. As an expression is parsed tokens are shifted to a stack. At the appropriate time the stack is reduced by applying the operator to the top of the stack. This is best illustrated by example.

step	*opr*	*val*	*input*	*action*
1	**$**	**$**	**4 * 2 + 1 $**	shift
2	**$**	**$ 4**	*** 2 + 1 $**	shift
3	**$ ***	**$ 4**	**2 + 1 $**	shift
4	**$ ***	**$ 4 2**	**+ 1 $**	reduce
5	**$**	**$ 8**	**+ 1 $**	shift
6	**$ +**	**$ 8**	**1 $**	shift
7	**$ +**	**$ 8 1**	**$**	reduce
8	**$**	**$ 9**	**$**	accept

We define two stacks: **opr**, for operators, and **val**, for values. A "**$**" designates the end of input or end of stack. Initially the stacks are empty, and the input contains an expression to parse. Each value, as it is encountered, is *shifted* to the **val** stack. When the first operator is encountered (step 2), we shift it to the **opr** stack. The second value is shifted to the **val** stack in step 3. In step 4, we have a "*****" in **opr**, and a "**+**" input symbol. Since we want to multiply before we add (giving multiplication precedence), we'll *reduce* the contents of the **val**, applying the "*****" operator to the top of the **val**. After reducing, the "**+**" operator will be shifted to **opr** in step 5.

This process continues until the input and **opr** are empty. If all went well, we should be left with the answer in the **val**. The following table summarizes the action taken as a function of input and the top of **opr**:

	Input		
opr	**+**	*****	**$**
+	reduce	shift	reduce
*****	reduce	reduce	reduce
$	shift	shift	accept

When the input token is "**+**", and "**+**" is on **opr**, we reduce before shifting the new "**+**" to **opr**. This causes the left-most operator to execute first, and implies that addition is left-associative. If we were to reduce instead, then addition would be right-associative. When the input token is "*****", and "**+**" is in **opr**, we shift "*****" to **opr**. Later, when reducing, "*****" is popped before "**+**", giving precedence to multiplication. By appropriately specifying shift and reduce actions, we can control the associativity and precedence of operators in an expression. When we encounter an operator in the input stream, and an operator is already on the stack, take the following action:

- If the operators are different, shift to give higher precedence to the input operator.
- If the operators are the same, shift for right associativity.

2. Theory, Part II

Let's extend the previous example to include additional operators.

	input												
stack	**+**	**-**	*****	**/**	**^**	**M**	**f**	**p**	**c**	**,**	**(**	**)**	**$**
+	r	r	s	s	s	s	s	s	s	r	s	r	r
-	r	r	s	s	s	s	s	s	s	r	s	r	r
*****	r	r	r	r	s	s	s	s	s	r	s	r	r
/	r	r	r	r	s	s	s	s	s	r	s	r	r
^	r	r	r	r	s	s	s	s	s	r	s	r	r
M	r	r	r	r	r	s	s	s	s	r	s	r	r
f	r	r	r	r	r	r	r	r	r	r	s	r	r
p	r	r	r	r	r	r	r	r	r	r	s	r	r
c	r	r	r	r	r	r	r	r	r	r	s	r	r
,	r	r	r	r	r	r	r	r	r	r	r	r	e4
(	s	s	s	s	s	s	s	s	s	s	s	s	e1
)	r	r	r	r	r	r	e3	e3	e3	r	e2	r	r
$	s	s	s	s	s	s	s	s	s	e4	s	e3	a

The above table incorporates the following operators:

- "**M**", for unary minus.
- "**^**", for exponentiation. 5 ^ 2 yields 25.
- "**f**", for factorial. f(x) returns the factorial of x.
- "**p**", for permutations. p(n,r) returns the number of permutations for *n* objects taken *r* at a time.
- "**c**", for combinations. c(n,r) returns the number of combinations for *n* objects taken *r* at a time.

The following operator precedence is implied, starting at the highest precedence:

- unary minus
- exponentiation, right-associative
- multiplication/division, left-associative
- addition/subtraction, left-associative

The following errors are detected:

- *error e1*: missing right parenthesis
- *error e2*: missing operator
- *error e3*: unbalanced right parenthesis
- error e4: invalid function argument

Parentheses are often used to override precedence. This is easily accomodated in the operator precedence table using the following algorithm:

```
input   action
 '('    shift
 ')'    while opr[tos] != '('
           reduce
        shift ')'
        reduce '()'
```

On encountering a left parenthesis, we shift it to the opr stack. When we input a right parenthesis, the stack is popped until the matching left parenthesis is found. Then we shift the right parenthesis, and reduce by popping both parentheses off the stack.

For function calls, the function reference is shifted to the stack. Each comma-separated argument is also shifted to the stack. On encountering a comma, the operator stack is reduced until the opening parenthesis of the function call is visible, leaving the function parameter on the value stack. Then the comma is shifted to the stack, and popped on the next reduction. When the closing parenthesis of the function call is encountered, it will be shifted to the stack and reduced. This will expose the function call for subsequent reduction.

Operator classes may be used to minimize the size of the precedence table. For example, a single generic token representing a function call may be pushed on the operator stack. An ordinal, defining which function is referenced, is pushed on the value stack. For this purpose you may choose to implement each element on the value stack as a union to allow for different types.

Error-checking is not bullet-proof. For example, omitting commas between function arguments will be accepted, and work properly. Even more bizarre, reverse-polish notation, where operators follow operands, is also acceptable. This phenomenon occurs because operators are applied to the top elements of the value stack, and the order that operators and values are pushed is not significant. Major errors, such as omitting an operator, will be found by the final reduction, as the stacks will not be empty. More rigorous error-checking may be done by defining a boolean *follow* matrix. The current and previous tokens are used as indices into the matrix to determine if one can follow the other.

The table is implemented in the next section, using ANSI-C. Note that there are some differences between my solution and Aho's:

- Aho specifies three precedence relations, whereas I've only specified two (shift and reduce). You can use three if you wish; I'm the last one to stifle creativity!
- Aho uses one stack for both values and operators. I've used separate stacks, as the implementation is typically easier.

3. Practice

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <math.h>

typedef enum { false , true } bool;

/* actions */
typedef enum {
    S,                  /* shift */
    R,                  /* reduce */
    A,                  /* accept */
    E1,                 /* error: missing right parenthesis */
    E2,                 /* error: missing operator */
    E3,                 /* error: unbalanced right parenthesis */
    E4                  /* error: invalid function argument */
} actEnum;

/* tokens */
typedef enum {
    /* operators */
    tAdd,               /* + */
    tSub,               /* - */
    tMul,               /* * */
    tDiv,               /* / */
    tPow,               /* ^ (power) */
    tUmi,               /* - (unary minus) */
    tFact,              /* f(x): factorial */
    tPerm,              /* p(n,r): permutations */
    tComb,              /* c(n,r): combinations */
    tComa,              /* comma */
    tLpr,               /* ( */
    tRpr,               /* ) */
    tEof,               /* end of string */
    tMaxOp,             /* maximum number of operators */
    /* non-operators */
    tVal                /* value */
} tokEnum;

tokEnum tok;            /* token */
double tokval;          /* token value */

#define MAX_OPR         50
#define MAX_VAL         50
char opr[MAX_OPR];      /* operator stack */
double val[MAX_VAL];    /* value stack */
int oprTop, valTop;     /* top of operator, value stack */
bool firsttok;          /* true if first token */
```

```
char parseTbl[tMaxOp][tMaxOp] = {
/* stk      ------------------ input ----------------------- */
/*         +   -   *   /   ^   M   f   p   c   ,   (   )   $  */
/*         --  --  --  --  --  --  --  --  --  --  --  --  -- */
/* + */  { R,  R,  S,  S,  S,  S,  S,  S,  S,  R,  S,  R,  R },
/* - */  { R,  R,  S,  S,  S,  S,  S,  S,  S,  R,  S,  R,  R },
/* * */  { R,  R,  R,  R,  S,  S,  S,  S,  S,  R,  S,  R,  R },
/* / */  { R,  R,  R,  R,  S,  S,  S,  S,  S,  R,  S,  R,  R },
/* ^ */  { R,  R,  R,  R,  S,  S,  S,  S,  S,  R,  S,  R,  R },
/* M */  { R,  R,  R,  R,  R,  S,  S,  S,  S,  R,  S,  R,  R },
/* f */  { E4, E4, E4, E4, E4, E4, E4, E4, E4, E4, S,  R,  R },
/* p */  { E4, E4, E4, E4, E4, E4, E4, E4, E4, E4, S,  R,  R },
/* c */  { E4, E4, E4, E4, E4, E4, E4, E4, E4, E4, S,  R,  R },
/* , */  { R,  R,  R,  R,  R,  R,  R,  R,  R,  E4, R,  R,  E4},
/* ( */  { S,  S,  S,  S,  S,  S,  S,  S,  S,  S,  S,  S,  E1},
/* ) */  { R,  R,  R,  R,  R,  R,  E3, E3, E3, E4, E2, R,  R },
/* $ */  { S,  S,  S,  S,  S,  S,  S,  S,  S,  E4, S,  E3, A }
};

int error(char *msg) {
    printf("error: %s\n", msg);
    return 1;
}

int gettok(void) {
    static char str[82];
    static tokEnum prevtok;
    char *s, *ptr;

    /* scan for next symbol */
    if (firsttok) {
        firsttok = false;
        prevtok = tEof;
        gets(str);
        if (*str == 'q') exit(0);
        s = strtok(str, " ");
    } else {
        s = strtok(NULL, " ");
    }
```

```
    /* convert symbol to token */
    if (s) {
        switch(*s) {
        case '+': tok = tAdd; break;
        case '-': tok = tSub; break;
        case '*': tok = tMul; break;
        case '/': tok = tDiv; break;
        case '^': tok = tPow; break;
        case '(': tok = tLpr; break;
        case ')': tok = tRpr; break;
        case ',': tok = tComa; break;
        case 'f': tok = tFact; break;
        case 'p': tok = tPerm; break;
        case 'c': tok = tComb; break;
        default:
            tokval = strtod(s, &ptr);
            if (*ptr) {
               printf(“error: non-numeric data encountered\n”);
               return 1;
            }
            tok = tVal;
            break;
        }
    } else {
        tok = tEof;
    }

    /* check for unary minus */
    if (tok == tSub) {
        if (prevtok != tVal && prevtok != tRpr) {
            tok = tUmi;
        }
    }

    prevtok = tok;
    return 0;
}

int shift(void) {
    if (tok == tVal) {
        if (++valTop >= MAX_VAL)
            return error("val stack overflow");
        val[valTop] = tokval;
    } else {
        if (++oprTop >= MAX_OPR)
            return error("opr stack overflow");
        opr[oprTop] = (char)tok;
    }
    if (gettok()) return 1;
    return 0;
}
```

```
double fact(double n) {
    double i, t;
    for (t = 1, i = 1; i <= n; i++)
        t *= i;
    return t;
}

int reduce(void) {
    switch(opr[oprTop]) {
    case tAdd:
        /* apply E := E + E */
        if (valTop < 1) return error("syntax error");
        val[valTop-1] = val[valTop-1] + val[valTop];
        valTop--;
        break;
    case tSub:
        /* apply E := E - E */
        if (valTop < 1) return error("syntax error");
        val[valTop-1] = val[valTop-1] - val[valTop];
        valTop--;
        break;
    case tMul:
        /* apply E := E * E */
        if (valTop < 1) return error("syntax error");
        val[valTop-1] = val[valTop-1] * val[valTop];
        valTop--;
        break;
    case tDiv:
        /* apply E := E / E */
        if (valTop < 1) return error("syntax error");
        val[valTop-1] = val[valTop-1] / val[valTop];
        valTop--;
        break;
    case tUmi:
        /* apply E := -E */
        if (valTop < 0) return error("syntax error");
        val[valTop] = -val[valTop];
        break;
    case tPow:
        /* apply E := E ^ E */
        if (valTop < 1) return error("syntax error");
        val[valTop-1] = pow(val[valTop-1], val[valTop]);
        valTop--;
        break;
    case tFact:
        /* apply E := f(E) */
        if (valTop < 0) return error("syntax error");
        val[valTop] = fact(val[valTop]);
        break;
    case tPerm:
        /* apply E := p(N,R) */
        if (valTop < 1) return error("syntax error");
        val[valTop-1] = fact(val[valTop-1])/
            fact(val[valTop-1]-val[valTop]);
        valTop--;
        break;
```

```
    case tComb:
        /* apply E := c(N,R) */
        if (valTop < 1) return error("syntax error");
        val[valTop-1] = fact(val[valTop-1])/
            (fact(val[valTop]) *
            fact(val[valTop-1]-val[valTop]));
        valTop--;
        break;
    case tRpr:
        /* pop () off stack */
        oprTop--;
        break;
    }
    oprTop--;
    return 0;
}

int parse(void) {

    printf("\nenter expression (q to quit):\n");

    /* initialize for next expression */
    oprTop = 0; valTop = -1;
    opr[oprTop] = tEof;
    firsttok = true;
    if (gettok()) return 1;

    while(1) {

        /* input is value */
        if (tok == tVal) {
            /* shift token to value stack */
            if (shift()) return 1;
            continue;
        }

        /* input is operator */
        printf("stk=%d tok=%d\n", opr[oprTop], tok);
        switch(parseTbl[opr[oprTop]][tok]) {
        case R:
            if (reduce()) return 1;
            break;
        case S:
            if (shift()) return 1;
            break;
        case A:
            /* accept */
            if (valTop != 0) return error("syntax error");
            printf("value = %f\n", val[valTop]);
            return 0;
```

```
        case E1:
            return error("missing right parenthesis");
        case E2:
            return error("missing operator");
        case E3:
            return error("unbalanced right parenthesis");
        case E4:
            return error("invalid function argument");
        }
    }
}

int main(void) {
    while(1) parse();
    return 0;
}
```