FILOMAT (Niš) **13** (1999), 157–172

ON SOME PROPERTIES OF REVERSE POLISH NOTATION

Predrag V. Krtolica and Predrag S. Stanimirović

Abstract. An extension of the reverse Polish notation, as well as the extension of the corresponding algorithms for transforming infix expressions to the postfix ones and vice versa, are suggested. Further, some properties of reverse Polish notation are investigated. These properties are important in the simplification of the corresponding infix expression. A software implementing improved algorithms for the infix to postfix transformation and vice versa is developed.

1. Introduction

Years ago, reverse Polish notation became the complementary part of many textbooks in computer science (e.g. [1], [5], [6], [7]). By this notation, the arithmetic expression was noted in a postfix manner, instead of using the usual infix notation.

For example, the infix expression

$$a + b$$

in reverse Polish was noted like

$$ab + .$$

Also, the algorithms for transforming a postfix expression to the infix one, and vice versa, are well known (see e.g. [1], [5], [6], [7]). Usually, these algorithms include stack usage and they could be restated as the following:

Received January 27, 1999

2000 *Mathematics Subject Classification.* 68Q40.

Key words and phrases. Reverse Polish notation, grasp of the operator.

The algorithm for transforming an infix expression to the postfix one:

(1I) Examine the current element of the infix expression.
(2I) If the current element is an operand, send it to the output and go to Step (6I).
(3I) If the current element is the left parenthesis, push the element and go to Step (6I).
(4I) If the current element is an operator then do the following:
if it has the higher priority than top of stack push that operator, else pop the operator from stack, send it to the output, and repeat Step (4I).
Note that we assume the parenthesis as the lowest priority element, and if the stack is empty that it should be treated as top of stack with the lowest priority.
(5I) If the current element is the right parenthesis, then pop operators from stack and send them to the output until the left parenthesis is poped (which is not supposed to be sent to the output).
(6I) If there are more elements of the input expression, take the next and go to the Step (2I). Otherwise, pop the rest of the stack, send it to the output, and stop.

The algorithm for transforming a postfix expression to the infix one:

(1P) Examine the current element of the expression.
(2P) If the element is an operand, push it.
(3P) If the element is the binary operator, then pop two operands, execute the operation and push the result; but, if the element is the unary operator, then pop the only one operand from the stack, execute operation and push the result [1].
(4P) If there are more elements of the input expression, take the next and go to Step (2P).
(5P) If the input expression is exhausted, then the infix expression is in top of the stack; pop it and stop.

Note that in the second algorithm, at the end we have the value of the input postfix expression if we deal with the operand values, but the symbolic infix expression if we use stack of strings instead. Actually, instead of operands or the parts of the expressions we can push the pointers to the corresponding strings. In this way, we are able to get the symbolical expression back in the infix form.

The algorithms restated above could be extended to process n-ary operators. Of course, the list of these operators should be known, as well as

the number of required arguments. For illustration purposes we introduce operator *fun* which has three arguments - $fun(x, y, z)$. Now, the algorithms should be restated as the following:

In the algorithm for transforming the infix to the postfix expression we include the additional step, let us say, (3Ia):

(3Ia) If the element of the expression is comma go to the Step (6I).

In the algorithm for transforming the postfix to the infix expression the Step (3P) is changed:

(3Pa) If the element is the binary operator, then pop two operands, execute the operation and push the result; but, if the element is the unary operator, then pop the only one operand from the stack, execute the operation and push the result. If the element is n-ary operator (it means threenary in our case), pop n operands, execute the operation (i.e. make the string representing function call - $fun(x, y, z)$ if the x, y and z are the three top operands) and push the result.

The program which transforms any infix expression to the postfix one and vice versa, in the beginning accepts the string representing the input expression and then separates different elements of the expression. The elements of the expression could be operators $+$, $-$, $*$, $/$, ^ (which means exponenting by constant integer exponent), left and right parenthesis, unary operators, i.e. standard function names (*neg*, *plus*, *sin*, *cos*, *tan*, *ctg*, *log*, *exp*, *sqrt*; this list could be easily extended), and operands both variables and constants (integer and fixed point real values). We get an array of strings which represents expression elements. After that, the operator of exponenting (e.g. x^3) is replaced by repeated multiplication ($x * x * x$).

Now, the function which actually makes the transformation to the postfix notation is called.

The other function transforms the begotten postfix expression back to the infix form. Recall that we use pointers to strings, instead of values of the operands, to get the symbolical infix expression.

Example 1.1. In this example we show the application of an expression containing threenary operator *fun*.

```
You entered the following expression
x^2+x*cos(fun(x1,x2,x3))/x
This expression in postfix is
x x * x x1 x2 x3 fun cos * x / +
Now, we transform postfix expression back to infix
x*x+x*cos(fun(x1,x2,x3))/x
```

The paper is organized as the following. In the second section, we observe and prove some properties of reverse Polish notation concerning the transformation of such an expression back into the infix form. In the third section, a few useful rules which can be used in the simplification of the infix form are formulated. A few implementation details are also presented.

2. Properties of the Extended Reverse Polish Notation

We suppose that the input expression is transformed into the reverse Polish notation, where all of its *elements* (variables, constants and operators) are separated. Hence, we actually deal with an array of strings representing the elements of the input expression. We denote this array of expression elements as $postfix$, where $postfix[i]$, for each $i \geq 0$, is a string which denotes an expression element, i.e. a variable, a constant, or an operator.

Definition 2.1. The *grasp* of the element $postfix[i]$ is the number of its preceding elements which form operand(s) of the element $postfix[i]$. We denote the grasp of the element $postfix[i]$ by $GR(postfix[i])$. Integer i is called the index of the element $postfix[i]$. Index i of the element $postfix[i]$ will be alternatively denoted by $IND(postfix[i])$.

Remark 2.1. The element $postfix[i]$ in the array $postfix$, representing the reverse Polish notation of the corresponding expression, can be the operator or the simple operand (variable or constant). We concern every simple operand as *0-ary operator*, and assume that its grasp is *zero*.

Example 2.1. For example, the grasp of the operator $+$ in $ab+$ is two, because two preceding elements should be the operands of the operation $+$. In some more complex postfix expression

$$1\ 2\ x\ sqrt\ *\ /\ neg\ x\ sqrt\ x\ sqrt\ *\ / \tag{2.1}$$

which is the reverse Polish of the expression

$$neg(1/(2 * sqrt(x)))/(sqrt(x) * sqrt(x))$$

the grasps of the operators contained in this expression are given in the following table:

$postfix[3] = sqrt$	1
$postfix[4] = *$	3
$postfix[5] = /$	5
$postfix[6] = neg$	6
$postfix[8] = sqrt$	1
$postfix[10] = sqrt$	1
$postfix[11] = *$	4
$postfix[12] = /$	12

Definition 2.2. The *grasped elements* of the operator $postfix[i]$ are the *grasping* left preceding elements in the array $postfix$ which form operand(s) of the operator $postfix[i]$. The index of the most left element among them is called *the left grasp bound.* The left grasp bound of the operator $postfix[i]$ is denoted by $LGB(postfix[i])$.

Definition 2.3. The element $postfix[i]$ is called *the main element* or *head* for the expression formed by $postfix[i]$ and its grasped elements.

Remark 2.2. An arbitrary element $postfix[i]$ can be considered as the operator acting on operands $arg_1, \dots, arg_n$. Heads of these operands will be denoted by $op_1, \dots, op_n$.

Example 2.2. Consider the expression (2.1). The element $postfix[12] = /$ has two following arguments:

$$arg_1 = 1\ \ 2\ \ x\ \ sqrt\ \ *\ \ /\ \ neg, \qquad arg_2 = x\ \ sqrt\ \ x\ \ sqrt\ \ *$$

Heads of these arguments are

$$op_1 = neg = postfix[6], \qquad op_2 = * = postfix[11].$$

Similarly, the operator $neg = postfix[5]$ takes the operand

$$arg_1 = 1\quad 2\quad x\quad sqrt\quad *$$

whose head is

$$op_1 = * = postfix[4].$$

Lemma 2.1. *Assume that $postfix[i]$ is n-ary operator which takes operands whose heads are $op_1, \dots op_n$, respectively. Then, the following statements are valid:*

(a) $GR(postfix[i]) = i - LGB(postfix[i])$.

(b) $GR(postfix[i]) = n + \sum_{k=1}^{n} GR(op_k) = n + \sum_{k=1}^{n} (IND(op_k) - LGB(op_k)).$

(c) $LGB(postfix[i]) = i - n - \sum_{k=1}^{n} (IND(op_k) - LGB(op_k)).$

(d) $i = IND(postfix[i]) = n + \sum_{k=1}^{n} IND(op_k) + p,$

$LGB(postfix[i]) = \sum_{k=1}^{n} LGB(op_k) + p,$

for some integer p.

(e) $op_{n-j} = postfix\left[i - \sum_{k=1}^{j-1} GR(op_{n-k}) - j - 1\right], \quad j = 0, \dots, n-1.$

Proof. **(a)** It follows immediately from Definitions 2.1 and 2.2.

(b) From Definitions 2.1 – 2.3 it is quite clear that the heads $op_1, \dots, op_n$ and its grasped elements are the grasped elements of the element $postfix[i]$. Hence,

$$GR(postfix[i]) = n + \sum_{k=1}^{n} GR(op_k).$$

From **(a)**, for every $k = 1, \dots, n$ the grasp of op_k is equal to

$$IND(op_k) - LGB(op_k).$$

Then, we have

$$GR(postfix[i]) = n + \sum_{k=1}^{n} (IND(op_k) - LGB(op_k)).$$

(c) This part easily follows from **(a)** and **(b)**.

(d) From part **(c)** we obtain the identity

$$i - LGB(postfix[i]) = n + \sum_{k=1}^{n} IND(op_k) - \sum_{k=1}^{n} LGB(op_k).$$

Since i and $LGB(postfix[i])$ as well as $IND(op_k)$ and $LGB(op_k)$, $k = 1, \dots, n$ are non-negative integers, we have

$$i = n + \sum_{k=1}^{n} IND(op_k) + p, \quad LGB(postfix[i]) = \sum_{k=1}^{n} LGB(op_k) + p,$$

where the integer p is equal to

$$p = i - n - \sum_{k=1}^{n} IND(op_k) = LGB(postfix[i]) - \sum_{k=1}^{n} LGB(op_k).$$

(e) For each $j = 0, \dots, n-1$, position of the head op_{n-j} can be obtained subtracting the number

$$\sum_{k=1}^{j-1} GR(op_{n-k}) + j + 1$$

from $i = IND(postfix[i])$. The number

$$\sum_{k=1}^{j-1} GR(op_{n-k})$$

contains the grasps of the previous heads op_{n-k}, $k = 0, \dots, j-1$, and the number $j+1$ represents locations of op_{n-k}, $k = 0, \dots, j-1$ as well as locations of op_{n-j} and $postfix[i]$. □

Lemma 2.2. *If the grasp of an arbitrary n-ary operator $postfix[i]$ is greater than n, then at least one of its arguments heads is also an operator.*

Proof. Assume that the grasp of n-ary operator $postfix[i] = f(arg_1, \ldots, arg_n)$ is greater than n. If $op_1, \ldots, op_n$ are heads of $arg_1, \ldots, arg_n$, respectively, then according to Lemma 2.1, we get

$$GR(postfix[i]) = n + \sum_{k=1}^{n} GR(op_k) > n.$$

This implies the existence of an integer $k \in \{1, \ldots, n\}$ which satisfies $GR(op_k) > 0$. Consequently, op_k is also the operator. □

In the case $n = 2$ we obtain the following.

Corollary 2.1. *If the grasp of any binary operator $postfix[i]$ is greater than 2, then at least one of the two preceding elements in reverse Polish notation of the expression ($postfix[i-1]$ and $postfix[i-2]$) is also the operator (unary or binary).*

Theorem 2.1. *Assume that the grasp of an arbitrary binary operator $postfix[i]$ is greater than 2.*

(a) *If the difference between the grasp of the operator $postfix[i]$ and the grasp of its first preceding operator is equal to 2, then it is not necessary to insert parenthesis around at least one of the two operands of the operator $postfix[i]$. Specifically,*

(i) *if the difference of index i and the index of the first preceding operator with respect to $postfix[i]$ is equal to 1, then it is not necessary to insert parenthesis around the first expression-operand of the operator $postfix[i]$;*

(ii) *if the difference of index i and the index of the first preceding operator with respect to $postfix[i]$ is equal to 2, then it is not necessary to insert parenthesis around the second expression-operand of the operator $postfix[i]$.*

(b) *In the opposite case, when the difference between the above mentioned grasps is greater than 2, the parenthesis should be inserted around both expression-operands. The exception is in the case when one of the expression-operands is unary operator call. In this case, the parenthesis could be omitted.*

Proof. Let the operator $postfix[i]$ take the arguments with heads op_1 and op_2.

(a) In accordance with Lemma 2.2, for the case $n = 2$, at least one of the argument heads op_1 and op_2 is also the operator. From part (e) of Lemma 2.1 we get $op_2 = postfix[i-1]$.

(i) According to suppositions, $op_2 = postfix[i-1]$ is an operator. In view of Lemma 2.1, we get

$$2 = GR(postfix[i]) - GR(postfix[i-1]) = 2 + GR(op_1).$$

Hence, $GR(op_1) = 0$, which implies that op_1 is a simple operand.

(ii) In this case, $op_1 = postfix[i-2]$ is an operator. According to Lemma 2.1, we get

$$2 = GR(postfix[i]) - GR(postfix[i-2]) = 2 + GR(op_2).$$

Hence, $GR(op_2) = 0$, which implies that $op_2 = postfix[i-1]$ is a simple operand.

(b) Assume that $op_2 = postfix[i-1]$ is an operator. From Lemma 2.1 one can verify the following:

$$GR(postfix[i]) - GR(postfix[i-1]) = 2 + GR(op_1) > 2.$$

Hence, $GR(op_1) > 0$, which implies that op_1 is not a simple operand. The case when the operand $op_1 = postfix[i-2]$ is an operator can be proved in a similar way. □

3. Simplification Rules and Implementation Details

In this section we propose a few rules for the simplification of the infix expression which is derived from a postfix expression. Also, we describe a few routines for the application of these rules.

The value of left grasp bound for $postfix[i]$, equal to

$$LGB(postfix[i]) = i - n - \sum_{k=1}^{n}(IND(op_k) - LGB(op_k))$$

can be calculated by applying the effective procedure listed below in the C++ like pseudocode:

```
void left_bound(int z, int *ind)
{
do {
   *ind = *ind-1;
   if ( is_unary_operator(postfix[*ind]) )
        left_bound(1,ind);
   else
        if ( is_binary_operator(postfix[*ind]) )
             left_bound(2, ind);
        else
           if ( is_nary_operator(postfix[*ind]) )
               left_bound(n,ind);
   z = z-1;
   }
while (z);
}
```

The parameter z is the arity of the operator, while the *ind* is the pointer to the index of the operator whose grasp is calculated. In the function call time, $*ind$ has the value of the index i of the operator $postfix[i]$ for which we calculate the left grasp bound. When the function is done, $*ind$ is the index of the corresponding left grasp bound. The functions **is_unary_operator**, **is_binary_operator**, and **is_nary_operator** recognize the unary, binary, and n-ary operator, respectively.

The grasp $GR(postfix[i])$ of an arbitrary element $postfix[i]$ of the reverse Polish notation can be calculated as it follows:

```
int grasp(int i)
   {
   int* ptr_lgb;
   int start = i;
   ptr_lgb = &start;
   if ( is_unary_operator(postfix[i]) )
        left_bound(1,ptr_lgb);
   else
      if ( is_binary_operator(postfix[i]) )
            left_bound(2, ptr_lgb);
      else
          if ( is_nary_operator(postfix[i]) )
                left_bound(n, ptr_lgb);
   return (i - *ptr_lgb);
   }
```

Main simplification rules, which are used for the elimination of the unnecessary parenthesis in the infix expression, can be formulated as the following.

Rule 1. (a) If the current operator $postfix[i]$ in the reverse Polish notation of the expression, during postfix to infix transformation, is the binary $+$, then it is not necessary to insert the parenthesis around its operands.

(b) If the current operator $postfix[i]$ in the reverse Polish notation of the expression is the binary $-$, then the following is valid:

(i) The parenthesis are not necessary around the first argument;
(ii) The parenthesis around the second argument are necessary only if the element $postfix[i-1]$ is one of the binary operators $+$ or $-$.

The reason for (a) and the first part of (b) originates in the fact that $postfix[i]$ is the lowest priority operator and all of the possible operators in expression-operands are the operators of the same or higher priority, and they will be applied before the current operator. If the expression-operands contain only the operands of the same priority, for the well-known associativity, the parenthesis are redundant.

The second part of (b) is justified as the following. If $postfix[i-1]$ is not one of the binary operators $+$ or $-$, then the second argument of the $postfix[i]$ is a product, a quotient, or an unary operator call. All of them have the higher priority than $postfix[i]$ and this case is reduced to the case (a).

From Theorem 2.1 we get the following rules. In these rules we assume that $postfix[i]$ is an binary operator which takes two operands arg_1 and arg_2 whose heads are op_1 and op_2, respectively.

Rule 2. Let the grasp of the operator $postfix[i]$ be greater than 2.

(i) If $GR(postfix[i]) - GR(postfix[i-1]) = 2$ and $postfix[i-1]$ is an unary or binary operator, then it is not necessary to insert parenthesis around the first expression-operand arg_1, which is determined by the head $op_1 = postfix[i - GR(postfix[i-1]) - 2]$.

(ii) If $GR(postfix[i]) - GR(postfix[i-2]) = 2$ and $postfix[i-2]$ is an unary or binary operator, then it is not necessary to insert parenthesis around the second expression-operand arg_2, which is determined by the head $op_2 = postfix[i-1]$.

(iii) The exception of the case (i) is raised when $postfix[i] = *$ and $postfix[i-1] = *$ or $postfix[i-1] = /$. Also, the exception of the case (ii) is raised when $postfix[i] = *$ and $postfix[i-2] = *$ or $postfix[i-2] = /$. Then, the parenthesis are not necessary around both operands arg_1 and arg_2. There is another exception of the case (ii), when $postfix[i] = /$ and $postfix[i-2] = *$ or $postfix[i-2] = /$.

Then, there is no need for the parenthesis around both of the arguments.

Rule 3. Let the grasp of an arbitrary binary operator $postfix[i]$ be greater than 2 and the difference between its grasp and the grasp of the first preceding operator be greater than 2. Then the parenthesis should be inserted around both expression-operands arg_1 and arg_2. The exceptions are aroused in the following cases:

- (i) One (or both) of the expression-operands arg_1 and arg_2 is unary operator call, i.e. when at least one of the heads op_1, op_2 is unary operator. Then, the parenthesis should be omitted around this (or both) argument(s).
- (ii) The operator $postfix[i] = *$ and one (or both) of the heads of its arguments are $*$ or $/$. Then, the parenthesis should be omitted around this (or both) argument(s).
- (iii) The operator $postfix[i] = /$ and $op_1 = *$ or $op_1 = /$. Then, the parenthesis should be omitted around the first argument arg_1.

From Definition 2.1 immediately follows Rule 4.

Rule 4. If $postfix[i]$ is a binary operator and $GR(postfix[i]) = 2$, then both of its operands, arg_1 and arg_2, are simple and parenthesis around them could be omitted.

It seems that Rules 1–4 cover all the possible cases concerning the omitting or the inserting parenthesis while the postfix expression is converted into the infix one. So, we make the following conjecture.

Conjecture 3.1. *The Rules 1–4 remove all unnecessary parenthesis.*

The Rules 1 to 4 could be employed in the following pseudocode routines to avoid the insertion of unnecessary parenthesis.

Suppose that we want to make a string `res` representing the application of the operator $postfix[i]$ on its operands, denoted by `arg_1` and `arg_2`, while `op_1` and `op_2` represent the heads of these operands. Function `is_low_prior` recognizes when $postfix[i]$ is binary $+$ or $-$.

The part (a) of Rule 1 and Rule 4 can be implemented in the following pseudocode.

```
if (postfix[i]=='+' || GR(postfix[i]) == 2)
  {
  strcat(res, arg_1);
  strcat(res,postfix[i]);
  strcat(res, arg_2);
  }
```

The part (b) of Rule 1 and Rule 4 can be implemented in the following pseudocode.

```
if (postfix[i]=='-' || GR(postfix[i]) > 2)
  {
  strcat(res, arg_1);
  strcat(str3,content(postfix[i]));
  if (postfix[i-1]. == '+' || postfix[i-1] == '-' )
          {
          strcat(res,"(");
          strcat(res, arg_2);
          strcat(res,")");
          }
  else
          strcat(res, arg_2);
  }
```

In the pseudocode listed below this case, using Rules 2 and 3, we make a string `res` with the parenthesis insertion only when it is necessary. Note that we illustrate usage of Rules 2 and 3 only for the case $postfix[i] = *$ (for division we have the similar code).

We also note that the value $GR(postfix[i])$ is equal to $grasp(i)$.

```
if (postfix[i] == '*')
  if (GR(postfix[i])-GR(postfix[i-1]) == 2)
   { /* Part (i) of Rule 2 */
   op_2=postfix[i-1]; op_1=postfix[i-GR(op_2)-2];
   strcat(res,arg_1);
   strcat(res,postfix[i]);
   if (GR(op_2)>2 && is_not_unop(op_2) &&
       !(postfix[i-1] == '*' || postfix[i-1] == '/'))
     /*Part (iii) of Rule 2*/
     { /* postfix[i-1] is an operator */
     strcat(res,"(");
     strcat(res,arg_2);
     strcat(res,")");
     }
   else   /* postfix[i-1] is not an operator */
     strcat(res,arg_2);
   }  /* End of part (i) of Rule 2 */
  else
   if (GR(postfix[i])-GR(postfix[i-2]) == 2)
      { /* Part (ii) of Rule 2 */
```

```
    op_2=postfix[i-1]; op_1=postfix[i-2];
    if (is_not_unop(postfix[i-2]) &&
       !(postfix[i-2]=='*' || postfix[i-2]=='/'))
      /*Part (iii) of Rule 2*/
          {
          strcat(res,"(");
          strcat(res,arg_1);
          strcat(res,")");
          }
    else
          strcat(res,arg_1);
    strcat(res,postfix[i]);
    strcat(res,arg_2);
    }  /* End of part (ii) of Rule 2 */
 else
    { /* Rule 3 */
    op_2=postfix[i-1]; op_1=postfix[i-GR(op_2)-2];
    if (is_not_unop(op_1) && !(op_1=='*' || op_1=='/'))
        {
        strcat(res,"(");
        strcat(res,arg_1);
        strcat(res,")");
        }
    else
        strcat(res,arg_1);
    strcat(res,postfix[i]);
    if (is_not_unop(op_2) && !(op_2=='*' || op_2=='/'))
        {
        strcat(res,"(");
        strcat(res,arg_2);
        strcat(res,")");
        }
    else
        strcat(res,arg_2);
    }
```

Example 3.1. This example illustrates how this software can eliminate redundant parenthesis when transforming an expression back into the infix form, using rules 1, 3, and 4.

```
You entered the following expression
(((((x)*z)-y)/(y+z)))
```

```
This expression in postfix is
x z * y - y z + /
Now, we transform postfix expression back to infix
(x*z-y)/(y+z)
```

The outmost parenthesis are omitted by virtue of the reverse Polish notation. Applying Rule 4, from $(x) * z$ we get $x * z$. Further, applying Rule 1, from $(x * z) - y$ we get $x * z - y$. Finally, an application of Rule 3 gives us necessity of parenthesis in $(x * z - y)$ and $(y + z)$.

Example 3.2. In this example we show the elimination of the redundant parenthesis using Rule 2.

```
You entered the following expression
(x)/(y+z)
This expression in postfix is
x y z + /
Now, we transform postfix expression back to infix
x/(y+z)
```

Example 3.3. In the following table we give some more examples for the elimination of the unnecessary parenthesis.

Table 3.1.

Input expression	Output expression	Applied rules
$(((x)+(3))*((2*y)/((z)+1.23)))-4.37/(x)$	$(x+3)*2*y/(z+1.23)-4.37/x$	3(ii)-(iii),4
$((x*y)+(y-(2*(z))))/((x*3.14)+(y-4))$	$(x*y+y-2*z)/(x*3.14+y-4)$	1(a)-(b),3,4
$(cos(2*(x-y)+(x+(2*z))))*(x*(y)*z)$	$cos(2*(x-y)+x+2*z)*x*y*z$	1,3(ii),4
$(x)/(exp(x\char`^2)/(y))*(x*0.5/(z))$	$x/(exp(x*x)/y)*x*0.5/z$	2(i)-(ii),3(i)-(ii),4
$sin((exp((x)+3)/((1.2)/y)))/(cos(y/z/w))$	$sin(exp(x+3)/(1.2/y))/cos(y/z/w)$	2(iii),3(i),4

4. The Application and Conclusion

In many applications in artificial intelligence and expert systems there is a need for the symbolic manipulations with mathematical expressions. Specifically, the symbolic derivation is included in many problems of the analysis and the optimization. Symbolic derivation could be done by making the expression tree, and then by making the corresponding tree derivative. The building of the expression tree begins with the transforming of the entered expression to the reverse Polish notation.

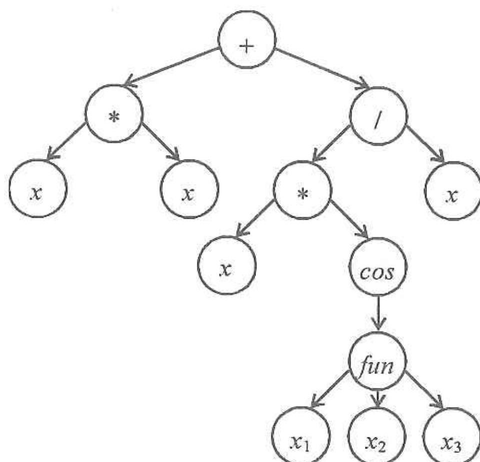

Fig. 4.1. The expression tree for the expression from Example 1.1.

Using previously described extended reverse Polish notation, which includes any n-ary operator, we get the "hybrid" expression trees as it is shown in Fig 4.1.

The properties of the reverse Polish, discussed in Section 2, are important in the transformation of the postfix expression to the infix one, for the elimination of unnecessary parenthesis, so the infix expression is as simple as possible. They give us the criteria when the parenthesis could be omitted and, yet, preserving the rightful precedence of the operators in the target infix expression. If there are no such criteria, we should insert parenthesis around every operand from the stack (simple or compound) when we are about to apply the operator.

References

[1] Gries, D., *Compiler Construction for Digital Computers*, John Wiley & Sons, Inc., New York, London, Sydney, Toronto, 1971.

[2] Krtolica, P. V. and Stanković, M. S., *QADE – Program for Qualitative Analysis of Differential Equations*, Proc. of II Math. Conf. in Priština 1996 (Lj. D. Kočinac, eds.), Priština, 1997, pp. 229–243.

[3] Parker, T. S. and Chua, L. O., *INSITE – A Software Toolkit for the Analysis of Nonlinear Dynamical Systems*, Proc. IEEE **75** (August 1987), 1081-1089.

[4] Parker, T. S. and Chua, L. O., *Practical Numerical Algorithms for Chaotic Systems*, Springer-Verlag, New York, 1989.

[5] Sedgewick, R., *Algorithms in C*, Addison-Wesley Publishing Company, Reading, MA, 1990.

[6] Tanenbaum, A. S., *Structured Computer Organization*, Prentice Hall, Englewood Cliffs, NJ, 1990.

[7] Tremblay, J.-P. and Sorenson, P.G., *The Theory and Practice of Compiler Writing*, McGraw-Hill Book Company, New York, 1985.

Predrag V. Krtolica
Department of Mathematics, Faculty of Sciences and Mathematics, University of Niš
Ćirila i Metodija 2, 18000 Niš, Yugoslavia
E-mail: `krca@archimed.filfak.ni.ac.yu`

Predrag S. Stanimirović
Department of Mathematics, Faculty of Sciences and Mathematics, University of Niš
Ćirila i Metodija 2, 18000 Niš, Yugoslavia
E-mail: `pecko@archimed.filfak.ni.ac.yu`