Yugoslav Journal of Operations Research
11 (2001), Number 1, 61-75

SYMBOLIC DERIVATION WITHOUT USING EXPRESSION TREES

Predrag V. KRTOLICA , Predrag S. STANIMIROVI]
Faculty of Science and Mathematics, University of Ni{,
ÆÐ] irila i Metodija 2, 18000 Ni{, Yugoslavia
krca, pecko@pmf.pmf.ni.ac.yu

Abstract: In this paper we describe a symbolic derivation method which uses no expression trees. The symbolic derivative is done directly from the reverse Polish notation of the input formula and the reverse Polish notation of its derivative. A transition from the reverse Polish notation of a given formula to the reverse Polish of its derivative is described.

Keywords: Symbolic derivation, reverse Polish notation, grasp of the operator.

1. INTRODUCTION AND PRELIMINARIES

Symbolic derivation is an important problem in many areas in computer science (e.g. artificial intelligence, optimization, etc.) where the input formulas are known only in the run-time. Usually, the symbolic derivation method includes making expression trees ([2], [7], [8], [10]). The traditional procedure for symbolic derivation consists of the following major steps:

Step S1. Convert the usual infix formula into reverse Polish notation.

Step S2. Make the expression tree of the input formula.

Step S3. Make the tree representing the derivative of the input formula.

Step S4. Read the tree derivative with modified inorder traversal and get the derivative of the input expression.

Although dynamic memory allocation is an efficient method, some time and memory cost should be paid when the corresponding expression tree is made. Additional problems, such as the garbage collection problem, could be generated.

In this paper, one method for symbolic derivation which "jumps" over the expression trees is suggested. We get the reverse Polish notation (RPN) of the symbolic derivative of the input formula directly from its RPN and the RPN of its derivative. More precisely, instead of steps S2 and S3 we construct the RPN of the derivative of the input expression. The transition from the RPN of the input formula to the RPN of its derivative is the subject of this paper. Step S4 is also modified. This formula representing the derivative is generated from its RPN.

Note that many of the early approaches to this problem also do not use dynamic structures (e.g. [3]).

We suppose that the input expression is transformed into the reverse Polish notation, where all of its elements (variables, constraints and operators) are separated. Hence, we are actually dealing 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.

The allowed operators are +, -, *, / (binary arithmetic operators), and unary operators *plus*(), *neg*() (representing unary + and unary -), *sin*(), *cos*(), ***tan***(), ***ctg***(), *log*(), *sqrt*(). Also, the operator $\wedge$ of exponentiating with integer constants is allowed. Constants could be integers or fixed point reals, while identifiers of variables could be specified as they are in programming languages.

Our derivation method is supported by the properties of the RNP investigated in [6]. These properties are used in the transition from the RPN of the given formula into the RPN of its derivative and in the corresponding simplifications.

For the sake of completeness, we shall restate briefly the main results from [6].

Main notations which are introduced in [6] are the following.

Definition 1.1. 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 1.1. The element *postfix*[i] in the array *postfix*, representing the RPN of the corresponding expression, can be the operator or the simple operand (variable or constant). We consider every simple operand as an *0*-ary operator, and assume that its grasp is *zero.*

Definition 1.2. 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 left-most 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 1.3. The element *postfix*[*i*] is called *the main element* or the *head* for the expression formed by *postfix*[*i*] and its grasped elements.

Remark 1.2. An arbitrary element *postfix*[*i*] can be considered as the operator acting on operands $\mathrm{arg}_1,\ldots,\mathrm{arg}_n$. The heads of these operands will be denoted by $\mathrm{op}_1,\ldots,\mathrm{op}_n$.

In the following lemma, the main properties of the introduced notions are investigated.

Lemma 1.1. Assume that *postfix*[*i*] is an *n*-ary operator which takes operands whose heads are $\mathrm{op}_1,\ldots,\mathrm{op}_n$, respectively. Then, the following statements are valid:

(a) $GR(postfix[i]) = i\text{-}LGB(postfix[i])$

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

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

for some integer p.

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

Finally, we shall restate a few simplification rules from [6]. These rules can be used in the simplification of the corresponding infix expression.

Lemma 1.2. If the grasp of an arbitrary *postfix*[*i*] is greater than *n*, then at least one of its argument heads is also an operator.

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

Corollary 1.1. If the grasp of any binary operator *postfix*[*i*] is greater than 2, then at least one of the two preceding elements in the RPN of the expression (*postfix*[*i*-1] and *postfix*[*i*-2]) is also the operator (unary or binary).

Theorem 1.1. 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 preceding operator is equal to 2, then it is not necessary to insert parentheses around at least one of the two operands of the operator *postfix*[*i*]. Specifically,

   i) if the difference between 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 parentheses around the first expression-operand of the operator *postfix*[*i*].

   ii) if the difference between 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 parentheses 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 parentheses 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.

2. MAKING THE DERIVATIVE AND ITS SIMPLIFICATION

Our method for the transition from the RPN of the given formula into the RPN of its derivative is a recursive algorithm which processes the expression in RPN backwards. The recursive function making such a transformulation has the prototype

```
void derive(int low, int upp, char* dx)
```

where low and upp are lower and upper index bounds, respectively, for the piece of the array postfix which is to be the subject of the processing. Also, dx is a pointer to the string representing the variable by which the derivation is to be made.

The last element of the array *postfix* is certainly an operator, binary or unary (in [6] it is shown how to process *n*-ary operators). The first call of the function derive is made for the parameters 0 and *i*, where *i* is the index of the last element in the RPN of the input formula. Further, the recursive calls of the function derive succeed according to the derivation rules. Our main goal is the construction of an array of strings, denoted by *derivat*, which contains the RPN of the derivative of the input formula. During the construction of the array *derivat* we use only the array *postfix*. We employ the derivation rules for the composed function, so this case is processed easily.

In the program, the integer array *grasp* is used, whose elements are defined by

$$grasp[k] = GR(postfix[k]),\ k = 1,\ldots,i.$$

The function which computes the grasp is described in [6].

Recursive call application naturally depends on the operator postfix[upp], which represents the head of the deriving formula.

At this moment, let us assume that postfix[upp], is a binary operator. Applying the result of Lemma 1.1, we conclude that the head of its second argument arg_2 is equal to:

op_2 = postfix[upp-1].

Grasped elements of the head op_2 are taken from

postfix[upp-1-grasp[upp-1]]

to postfix[upp-1]. Similarly, the head of the first argument arg_1 of postfix[upp] is

op_1 = postfix[upp-GR(op_2)-2] = postfix[upp-2-grasp[upp-1]].

Grasped elements of the head op_1 are from postfix[low] to

postfix[upp-2-grasp[upp-1]].

From the beginning, we describe the derivation of the sum or difference. We use a common sign $\pm$ for the sum or the difference operator. Naturally, this case is implemented as the following:

```
derive(low, upp-2-grasp[upp-1], dx);
derive(upp-1-grasp[upp-1], upp-1, dx);
derivat[index++]=postfix[upp];
```

Now, we shall investigate this code in detail. Variable index index is a global integer variable initially equal to zero, which represents the index of the array derivat.

Let the RPN of the expressions *x* and y be denoted by $\langle x \rangle$ and $\langle y \rangle$, respectively. Also, assume that the RPN of their derivatives are denoted by $\langle x' \rangle$ and $\langle y' \rangle$, respectively. Then, in the general case, the Polish notation of the expression $(x \pm y)'$ is denoted by $\langle x' \rangle \langle y' \rangle "\pm"$. The construction of the array *derivat*, based on the array *postfix*, is essentially the following transformation:

$$\langle x \rangle \langle y \rangle "\pm" \mapsto \langle x' \rangle \langle y' \rangle "\pm" .$$

By means of the expression:

```
derive(low, upp-2-grasp[upp-1], dx);
```

the following transformation is performed

$$\langle x \rangle \mapsto \langle x \rangle' ,$$

and the content of the array derivat is $\langle x' \rangle$.

After the function call

derive(upp-1-grasp[upp-1], upp-1, dx);

the content of the array derivat is $\langle x' \rangle \langle y' \rangle$.

Finally, the expression

derivat[index++] = postfix[upp];

completes the Polish notation in the form $\langle x' \rangle \langle y' \rangle "\pm"$.

Therefore, it turns out that it is not easy to simplify the obtained derivative by additional processing, while in the case of the tree methods, it is relatively simple. Simplification consisting of the elimination of redundant terms like $x+0$, $0+x$, $x*0$, $0*x$, $x*1$ and $1*x$ is quite important, not only for the clarity of the program output, but also for making higher order derivatives (which could be done by the repetitive application of the algorithm). For these reasons, we oriented ourselves to the simplification in the same pass with the derivation.

In the case

postfix[upp] = = "+" or postfix[upp] = = "-",

the implementation of the derivation and the simultaneous derivative simplification is described in the following major steps.

Step 1±. [Put the content $\langle x' \rangle \langle y' \rangle$ into the array derivat, and remember the starting positions of the expressions $\langle x' \rangle$ and $\langle y' \rangle$]

```
stind = index                                   /*Starting index for ⟨x'⟩ */

derive(low, upp-2-grasp[upp-1], dx);                    /*Putting ⟨x'⟩ */

stind1=index;                                   /*Starting index for ⟨y'⟩ */

derive(upp-1-grasp[upp-1], upp-1 dx);                   /*Putting ⟨y'⟩ */
```

Now, the content of the array derivat is equal to $\langle x' \rangle \langle y' \rangle$.

Step 2±. [Dynamic simplification]

Case 1**.** [y' = 0] If the derivative of the second argument is y' = 0 i.e. in the case

(2.1) derivat [index-1] = = "0"

the RNP for $(x \pm y)'$ is equal to $\langle x' \rangle$. We perform the simplification

$\langle x' \rangle "0" \mapsto \langle x' \rangle$

in the array derivat. It is sufficient to go back one element:

```
index--;
```

Case 2. [$x' = 0$] This case is detected by the expression

(2.2) `derivat [stind1-1] = = "0"`

Two subcases can be detected.

A. [$0 + y' = y'$] If the condition (2.2) is satisfied, and `postfix[upp] = = "+"`, then we must perform the following change on the array derivat:

$"0"\langle y'\rangle \mapsto \langle y'\rangle$.

We rewrite $\langle y'\rangle$, starting from the index stind = stind1-1, overwriting the first argument which is zero and omitting the operator "+" at end.

```
for (k = stind1-1; k < index-1; k++)
        derivat [k] = derivat [k+1];
index = k+1;
```

B. [$0 - y' = -y'$] Now, we assume that the condition (2.2) is satisfied, and `postfix[upp] = = "-"`. Then, the array derivat is changed by

$"0"\ \langle y'\rangle \mapsto \langle y'\rangle$ "neg".

In this case, it is sufficient to rewrite $\langle y'\rangle$, starting from the stind = stind1-1, and place the operator "neg" at the end.

```
for (k = stind1-1; k < index-1; k++)
        derivat [k] = derivat [k+1];
index = k+1;
derivat [index++]="neg";
```

Step 3. [$x' \neq 0$ and $y' \neq 0$] In the general case, when both of the conditions (2.1) and (2.2) are not satisfied, the RPN for $(x \pm y)'$ is equal to $\langle x'\rangle\langle y'\rangle"\pm"$, and simplifications are not needed. Since the content of the array derivat is actually equal to $\langle x'\rangle\langle y'\rangle$, we write

```
derivat [index++]= postfix [upp];
```

In the case postfix[upp] = = "$*$", derivation of the product x $*$ y could be implemented in a similar way. In this case, the content of the actual part of the array postfix is $\langle x\rangle\langle y\rangle$"$*$". Our intention is to put the following content in the array derivat:

$$\langle x\rangle\langle y'\rangle\text{"}*\text{"}\langle x'\rangle\langle y\rangle\text{"}*\text{""}+\text{"}$$

The algorithm, without simplification, is described in the following pseudocode, where the content of the constructed part of the array derivat is described in the comments.

```
for (k = low; k <= upp-2-grasp [upp-1]; k++)
        derivat [index++]=postfix [k];                    /* ⟨x⟩ */
derive(upp-1-grasp[upp-1], upp-1, dx);                    /* ⟨x⟩⟨y'⟩ */
derivat [index++] = "*";                                  /* ⟨x⟩⟨y'⟩"*" */
derive(low, upp-2-grasp[upp-1]; dx);                      /* ⟨x⟩⟨y'⟩"*"⟨x'⟩ */
for (k = upp-1-grasp [upp-1]; k <= upp-1; k++)
        derivat [index++]=postfix [k];                    /* ⟨x⟩⟨y'⟩"*"⟨x'⟩⟨y⟩ */
derivat [index++] = "*";                                  /* ⟨x⟩⟨y'⟩"*"⟨x'⟩⟨y⟩"*" */
derivat [index++] = "+";                                  /* ⟨x⟩⟨y'⟩"*"⟨x'⟩⟨y⟩"*""+" */
```

But, when we employ the simplification, the later pseudocode becomes as described below.

Step 1*. [Place the content $\langle x\rangle\langle y'\rangle$"$*$" and remember the starting positions of the expressions $\langle x\rangle$ and $\langle y'\rangle$]

The starting positions of the expression $\langle x\rangle$ and $\langle y'\rangle$ are denoted by strind1 and strind2, respectively.

```
strind1=index                    /*Remember the starting index for ⟨x⟩ */
for (k = low; k <= upp-2-grasp [upp-1]; k++)
        derivat [index++]=postfix [k];                    /* ⟨x⟩ */
strind2=index                    /*Remember the starting index for ⟨y'⟩ */
```

```
derive(upp-1-grasp[upp-1], upp-1, dx);                    /* ⟨x⟩⟨y'⟩ */

derivat [index++] = "*";                                  /* ⟨x⟩⟨y'⟩"*" */
```

Step 2*. [Simplification of the expression $\langle x\rangle\langle y'\rangle$"*"]

Case 1. [Simplification in terms "0"$\langle y'\rangle$"*"$\mapsto$"0" and $\langle x\rangle$"0""*"$\mapsto$"0"]

```
if (derivat [strind2-1]= = "0" ‖ derivat [index-2] = = '0')
    {
    index=strind1;
    derivat[index++]='0';
    }
```

Case 2. [Simplification of the form "1"$\langle y'\rangle$"*"$\mapsto\langle y'\rangle$]

In this case the condition derivat[strind2-1] = = "1" is satisfied. Then we rewrite $\langle y'\rangle$, starting from strind2-1, overwriting the first argument "1". Also, we omit the operator "*" at the end.

```
if (derivat [strind2-1] = = "1")
    {
    for (k = strind2-1; k <=index-2; k++)
            derivat [k] = derivat [k+1];
    index=k;
    }
```

Case 3. [Simplification of the form $\langle x\rangle$"1""*"$\rightarrow\langle x\rangle$]

In this case the condition derivat[index-2] = = "1" is satisfied.

We go back two elements:

```
if (derivat [index-2] = = "1")
    index -= 2;
```

Let us denote the most simple form of the expression $\langle x\rangle\langle y'\rangle$"*" by $\{\langle x\rangle\langle y'\rangle$"*"$\}$.

Step 3*. [Place the content $\{\langle x\rangle\langle y'\rangle"*"\}\langle x'\rangle\langle y\rangle"*"$ and remember the starting positions of the expressions $\langle x\rangle$ and $\langle y'\rangle$]

The starting positions of the expressions $\langle x\rangle$ and $\langle y'\rangle$ are denoted by strind1 and strind2, respectively.

```
strind1=index                          /*Remember the starting index for ⟨x'⟩ */
derive(low, upp-2-grasp[upp-1], dx);   /* {⟨x⟩⟨y'⟩"*"}⟨x'⟩ */
strind2=index                          /*Remember the starting index for ⟨y⟩ */
for (k = upp-1-grasp[upp-1]; k <= upp-1; k++)
        derivat [index++]=postfix [k];  /* {⟨x⟩⟨y'⟩"*"}⟨x'⟩⟨y⟩ */
derivat [index++] = "*";                /* {⟨x⟩⟨y'⟩"*"}⟨x'⟩⟨y⟩"*" */
```

Step 4*. [Simplification of the expression $\{\langle x\rangle\langle y'\rangle"*"\}\langle x'\rangle\langle y\rangle"*"$] It could be done similarly as in Steps 1*, 2* and 3*.

Step 5*. [Simplification of the expression $\{\langle x\rangle\langle y'\rangle"*"\}\{\langle x'\rangle\langle y\rangle"*"\}"+"$]

This simplification could be done similarly as in Steps $1\pm$ and $2\pm$, in the case of derivation of the sum.

In the case of division, the derivation rules impose the following transformation:

$$\langle x\rangle\langle y\rangle"/"\mapsto\langle x'\rangle\langle y\rangle"*"\langle x\rangle\langle y'\rangle"*""-"\langle y\rangle\langle y\rangle"*""/".$$

```
derive(low, upp-2-grasp[upp-1], dx);
for (k = upp-1-grasp[upp-1]; k <= upp-1; k++)
        derivat [index++]=postfix [k];
derivat [index++] = "*";
for (k = low; k <= upp-2-grasp [upp-1]; k++)
        derivat [index++]=postfix [k];
derive(upp-1-grasp[upp-1], upp-1, dx);
derivat [index++] = "*";
```

```
derivat [index++] = "-";

for (k = upp-1-grasp[upp-1]; k <= upp-1; k++)

        derivat [index++]=postfix[k];

for (k = upp-1-grasp[upp-1]; k <= upp-1; k++)

        derivat [index++]=postfix[k];

derivat [index++] = "*";

derivat [index++] = "/";
```

Simplification can be performed in a similar way as before.

Now, we assume that postfix[upp] is a unary operator. According to Lemma 1.1, the head of its argument arg1 is
op1=postfix[upp-1].

Also, the grasped elements of op1 are from postfix[low] to postfix[upp-1]. Consequently, we need only one recursive call

```
derive(low, upp-1, dx);
```

In the case of unary operators (i.e. function calls) we apply the derivations rules for composed functions. The trivial case is for the functions *plus* and neg, representing unary + and -.

We implement the derivation of the expressions involving the operator neg in the following routine:

```
derive(low, upp-1, dx);                                  /* ⟨x'⟩ */

derivat[index++] = postfix[upp];                         /* ⟨x'⟩"neg" */
```

The simplification "0""neg"$\mapsto$"0" is implemented in this way

```
if (postfix[index-1] = = "0")

        index --;
```

The derivation of expressions which contain the operator plus need only

```
derive(low, upp-1, dx);                                  /* ⟨x'⟩ */
```

Let us illustrate the derivation of unary operators on a couple more examples. For function sin we perform the transformation

$$\langle x\rangle\text{"sin"}\mapsto\langle x\rangle\text{"cos"}\langle x'\rangle\text{"*"}.$$

It is easy to see that the derivation of composed functions is done using the chain rule.

The implementation of the derivation with simplification is described below.

Step 1*. [Put the content $\langle x\rangle\text{"cos"}\langle x'\rangle$ into array derivat]

```
for (k = low; k <= upp-1; k++)
        derivat[index++]=postfix[k];                    /* ⟨x⟩ */
derivat[index++]="cos";                                 /* ⟨x⟩"cos" */
derive(low, upp-1, dx);                                 /* ⟨x⟩"cos"⟨x'⟩ */
```

Step 2*. [Possible simplification]

```
if (derivat [index-1] = = "1")
        index --;                                       /* ⟨x⟩"cos""1"↦⟨x⟩"cos" */
else
        if (derivat [index-1] = = "0")
          {                                             /*Argument of sin is constant*/
          index = low;
          derivat [index++]="0";                        /* ⟨x⟩"cos""0"→"0" */
          }
        else                                            /* General case */
          derivat [index++]="0";                        /* ⟨x⟩"cos"⟨x'⟩"*" */
```

The logarithm function log should be derived as follows:

$$\langle x\rangle\text{"log"}\rightarrow\langle x'\rangle\langle x\rangle\text{"/"}.$$

```
derive(low, upp-1, dx);                                 /* ⟨x'⟩ */
```

```
if (derivat [index-1] = = "0")
        {                                   /* Argument of log is constant */
        index = low;
        derivat[index++]="0";               /* ⟨log(x)'⟩="0"  */
        }
else
        {
        for (k = low; k <= upp-1; k++)
                derivat[index++]=postfix [k];          /* ⟨x'⟩⟨x⟩ */
        derivat[index++]="/";}                         /* ⟨x'⟩⟨x⟩"/" */
        }
```

For the square root, we have the following transformation:

$\langle x\rangle$"sqrt"$\mapsto\langle x'\rangle$"2"$\langle x\rangle$"sqrt""*""/" .

```
derive(low, upp-1, dx);
if (derivat [index-1]= = "0")
        {                                   /*Argument of sqrt is constant*/
        index = low;
        derivat [index++]="0";              /* ⟨sqrt(c)'⟩="0" */
        }
else
        {                                   /*General case*/
        derivat [index++]="2";
        for (k = low; k <= upp-1; k++)
                derivat [index++]=postfix [k];
        derivat [index++]="sqrt";
        derivat [index++]="*";
        derivat [index++]="/";
        }
```

Finally, let us assume that postfix[upp] is a simple operand. Then two cases can be considered. If postfix[upp] is equal to the variable dx, its derivative is 1, which is achieved by

```
derivat [index++]="1";
```

Otherwise, its derivate is zero:

```
derivat [index++]="0";
```

3. COMPARISONS

In comparison to methods based on expression trees, we can conclude that *grasped* elements correspond to the subtree nodes (not including the root of the subtree). Also, grasp[upp] is equal to the number of subtree nodes.

Moreover, symbolic differentiation based on the RPN can be compared with the known differentiation technique in LISP (see e.g. [1], [4], [5]). Let us denote the LISP's function for the symbolic differentiation by deriv. The expressions

```
derive(low, upp-2-grasp[upp-1], dx);
```

```
derive(upp-1-grasp[upp-1], upp-1, dx);
```

correspond to recursive calls of the LISP's function deriv for the first and the second argument, respectively:

```
(setq d1 (deriv (cadr x)) dx)
```

```
(setq d1 (deriv (caddr x)) dx)
```

In the case expression postfix[upp] = = "+" or postfix[upp] = = "-', the expression

```
derivat [index++] = postfix[upp];
```

is equivalent to one of the expressions

```
(list '+ d1 d2)     (list '- d1 d2)
```

Similar analogies can be established for all other binary operators, as well as for unary operators. We suggest the following reason for the application of the method based on the RPN: this algorithm requires only two global arrays, denoted by postfix and derivat, and its memory requirements are the minimum possible.

4. CONCLUSION

The suggested method allows the derivation of the input formula without using any expression tree. Only two arrays, postfix and derivat, are needed, so memory requirements are minimal. The time cost is also reduced, because of the elimination of handling the expression trees. The composed functions are processed easily, and the pass simplification gives simplified output the facilitating of finding higher other derivatives by repetition of the algorithm.

Reducing the memory and the time cost is important especially when we apply the suggested algorithm to symbolic derivation as a tool in other applications requiring symbolic derivatives (e.g. some applications in artificial intelligence, expert systems, optimization, etc.).

REFERENCES

[1] Abelson, H., and Sussman, G. J., Structure and Interpretation of Computer Programs, MIT Press, Cambridge, Massachusetts, 1985.

[2] Flanders, H., "Automatic differentiation of composite functions", in: A. Greiwank and G.F. Corliss, (eds.), Automatic Differentiation and Algorithms: Theory, Implementation, and Application, Proceedings of the First SIAM Workshop on Automatic Differentiation, Breckenridge, Colorado, January 6-8, 1991, 95-99.

[3] Hanson, J. W., Caviness, J. S., and Joseph, C., "Analytic differentiation by computer", Communications of the ACM, 5 (1962), 349-335.

[4] Henessey, L. W., Common LISP, McGrew-Hill Book Company, 1989.

[5] Hyv–nen, E., and Sepponen, J., Introduction to LISP and functional programming, Mir, Moskva, 1990 (in Russian).

[6] Krtolica, P.V., and Stanimirovi}, P.S., "On some properties of reverse polish notation", FILOMAT, 13 (1999) 157-172.

[7] Krtolica, P.V., and Stankovi}, M.S., "QADE - Program for the qualitative analysis of differential equations", in: LJ. D. Ko~inac, (ed.), Proc. of II Math. Conf. in Pri{tina, Pri{tina, 1997, 229-243.

[8] Parker, T.S., and Chua, L.O., "*INSITE* - A software toolkit for the analysis of nonlinear dynamical systems", Proc. IEEE 75, 1987, 1081-1098.

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

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