FILOMAT **14** (2000), 87–93

DEDUCING ABOUT THE NECESSITY OF THE PARENTHESIS

Predrag V. Krtolica and Predrag S. Stanimirović

Abstract

In this paper[1] we prove the conjecture made in [2]. This conjecture establishes the rules for deducing when the insertion of the parenthesis is needed while converting an postfix expression to the infix one.

1. Introduction

The simplification of the symbolic expressions is the most important issue in symbolic computation. The reverse Polish notation is oftenly used in symbolic manipulations with the various expressions. Reverse Polish notation is the complementary part of many textbooks in computer science (e.g. [1], [3], [4], [5]).

When we use reverse Polish notation in symbolic calculations, sooner or later, we should transform the postfix expression to the infix one. During this transformation we have imposed the following question: When the parenthesis around the arguments of an binary operator are needed, and when we could omit them and still get a rightful precedence of the operations?

In [2] the properties of the reverse Polish notation are investigated. These properties are further used in the formulation of the rules about inserting parenthesis around arguments of the binary operation while corresponding postfix expression is converted into the infix form. In [2] the algorithms for transformation of the infix expression to the postfix one, and vice versa, are developed. These algorithms can process the expressions containing the

[1]Received May 12, 2000

2000 Mathematics Subject Classification: 68Q40

Key words and phrases: Reverse Polish notation, grasp of the operator, symbolic computation

Typeset by $\mathcal{A}_{\mathcal{M}}\mathcal{S}$-TEX

usual binary arithmetic operators and standard functions. The formulated rules are incorporated in the algorithm for the transformation of the expressions in postfix to the infix form.

For the sake of completeness, we restate the results from [2]. Note that the expression in the reverse Polish notation is denoted by $postfix$, 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 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 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*.

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 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 1.3. The element $postfix[i]$ is called *the main element* or *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 $arg_1, \dots, arg_n$. Heads of these operands will be denoted by $op_1, \dots, op_n$.

Lemma 1.1. *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) $op_{n-j} = postfix\left[i - \sum_{k=1}^{j-1} GR(op_{n-k}) - j - 1\right], \quad j = 0, \ldots, n-1.$

Lemma 1.2. *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.*

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 reverse Polish notation 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 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.*

The above results are the base for the following rules concerning the necessity of inserting parenthesis around arguments of a binary operator.

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 $-$.

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, in the general case, 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.

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.

2. Are rules 1-4 enough for deducing about parenthesis?

In [2] we made the conjecture that Rules 1-4 remove all unnecessary parenthesis because there was no counterexample for this. Now, we are ready to give a formal proof for this claim.

Theorem 2.1. *The Rules 1–4 remove all unnecessary parenthesis.*

Proof. When we talk about the necessity of the parenthesis while the binary operations are applied, we should observe only the head of the expression, denoted by $head = postfix[i]$, and the heads of its arguments $op_1 = postfix[i - GR(postfix[i-1]) - 2]$ and $op_2 = postfix[i-1]$. We observe the non-trivial cases when *head* is one of the four arithmetic operators only. Any of the heads op_1 and op_2 could be one of the four arithmetic operators $+$, $-$, $*$ and $/$, or one of the functional operators, or a simple operand.

Therefore, there are

$$6 \times 6 \times 4 = 144$$

various possibilities.

If both of the arguments arg_1 and arg_2 are simple, we have $1 \times 1 \times 4 = 4$ different cases (one for each of the arithmetic operators), covered by Rule 4.

In

$$1 \times 5 \times 4 + 5 \times 1 \times 4 = 40$$

cases, when exactly one of the arguments arg_1 and arg_2 is simple, we apply Rule 2.

Hence, in the rest of the proof, we can assume, without loss of generality, that both of the arguments are not simple. Then, the possible cases for op_1 and op_2 are four arithmetic operators and, as the fifth kind of the operators, unary functional operators. Henceforth, we have

$$5 \times 5 \times 4 = 100$$

remaining possibilities for op_1, op_2 and *head*.

The $5 \times 5 \times 1 = 25$ cases, when $head = +$ are covered by the part (a) of Rule 1. The next $5 \times 5 \times 1 = 25$ cases, when $head = -$, are covered by the part (b) of Rule 1.

The remaining $5 \times 5 \times 2 = 50$ cases have $head = *$ or $head = /$. Two cases, when op_1 and op_2 are both functional operators, are covered by the Rule 3(i). Also,

$$1 \times 4 \times 2 + 4 \times 1 \times 2 = 16$$

cases, when exactly one of the op_1 and op_2 is the functional operator, are covered by the Rule 3(i).

What remains is the $4 \times 4 \times 2 = 32$ cases, $4 \times 4 \times 1 = 16$ when $head = *$ and $4 \times 4 \times 1 = 16$ cases when $head = /$.

Then, the following events should be anticipated.

- In 8 cases we have $op_1 = +|-$, $op_2 = +|-$ and $head = *|/$. Then, the parenthesis are necessary around both arguments. These events are covered by the general case of Rule 3.

- In 4 cases we have $op_1 = +|-$, $op_2 = *|/$ and $head = *$. Then, the parenthesis around the argument arg_2 should be omitted. These cases are covered by part (ii) of Rule 3.

- Similarly, in 4 cases we have $op_1 = *|/$, $op_2 = +|-$ and $head = *$. Then, the parenthesis around the argument arg_1 should be omitted. These cases are covered by part (ii) of Rule 3.

- In 4 cases we have $op_1 = *|/$, $op_2 = *|/$ and $head = *$. Then, the parenthesis around both arguments arg_1 and arg_2 should be omitted. These cases are covered by part (ii) of Rule 3.

- In remaining 12 cases we have $head = /$. These cases arise when

$$op_1 = +|-,\ \ op_2 = *|/ \text{ or } op_1 = *|/,\ \ op_2 = +|- \ \text{ or } op_1 = *|/,\ \ op_2 = *|/.$$

Then, the parenthesis are necessary around the both arguments, except in 8 cases, when $op_1 = *$ or $op_1 = /$, and the parenthesis around arg_1 are excessive. These cases are covered by part (iii) of Rule 3.

Hence, we can conclude that all possible cases are covered by the Rules 1–4, so these Rules can correctly deduce whether the parenthesis are needed or not.

All possibilities for op_1, op_2 and $head$ as well as the necessity of the parenthesis are arranged in the Table 2.1. By the sign f we denote that op_1 or op_2 are some of the unary functional operators, and s denote a simple operand.

Table 2.1.

head	op_1	op_2	arg_1 or (arg_1)	arg_2 or (arg_2)	# of cases
$+$	$+\|-\|*\|/\|f\|s$	$+\|-\|*\|/\|f\|s$	arg_1	arg_2	36
$-$	$+\|-\|*\|/\|f\|s$	$+\|-$	arg_1	(arg_2)	12
$-$	$+\|-\|*\|/\|f\|s$	$*\|/\|f\|s$	arg_1	arg_2	24
$*$	$+\|-$	$+\|-$	(arg_1)	(arg_2)	4
$*$	$+\|-$	$*\|/\|f\|s$	(arg_1)	arg_2	8
$*$	$*\|/\|f\|s$	$+\|-$	arg_1	(arg_2)	8
$*$	$*\|/\|f\|s$	$*\|/\|f\|s$	arg_1	arg_2	16
$/$	$+\|-$	$+\|-\|*\|/$	(arg_1)	(arg_2)	8
$/$	$*\|/\|f\|s$	$+\|-\|*\|/$	arg_1	(arg_2)	16
$/$	$+\|-$	$f\|s$	(arg_1)	arg_2	4
$/$	$*\|/\|f\|s$	$f\|s$	arg_1	arg_2	8

References

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

[2] P.V. Krtolica and P.S. Stanimirović, *On Some Properties of Reverse Polish Notation*, FILOMAT **13** (1999), 157–172.

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

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

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

Faculty of Philosophy
University of Niš
Ćirila i Metodija 2
18000 Niš, Yugoslavia
E-mail: krca@pmf.pmf.ni.ac.yu
E-mail: pecko@pmf.pmf.ni.ac.yu