Mariangiola Dezani-Ciancaglini

A parenthesis machine for string manipulation

Revue française d'automatique informatique recherche opérationnelle. Informatique théorique, tome 8, nº R3 (1974), p. 37-46

<http://www.numdam.org/item?id=ITA_1974__8_3_37_0>

R.A.I.R.O.
(8^{e} année, R-3, 1974, p. 37 à 46)

A PARENTHESIS MACHINE FOR STRING MANIPULATION

par Mariangiola DEZANI-CIANCAGLINI [1]

Communicated by G. AUSIELLO

Abstract. — *A parenthesis machine, provided with a stack and an auxiliary memory, is described and then used for :*

1) The evaluation of programs of a scheme P of parenthesis languages.

2) The implementation of a general bottom-up analyzer for context-free, total precedence languages.

INTRODUCTION

This paper describes a parenthesis machine, provided with a stack and an auxiliary memory, which can be used as a string manipulator. This parenthesis machine rewrites a sequence of symbols, belonging to a given alphabet, on the stack. A right parenthesis interrupts this rewriting and acts as a replacement command for the substring enclosed between this parenthesis and the corresponding left one. Such a replacement depends on the memory state and, in its turn, can modify it.

A pioneer work in this field is that of Dijkstra [5], in which the role of the right parenthesis is played by the character « *E* ». The present work is an improvement, essentially because :

i) the addition of left parentheses allows the string to be substituted to be of arbitrary length;

ii) the existence of an auxiliary memory gives much greater freedom in writing programs.

The present parenthesis machine has been used with some modifications for the reduction of λ-formulas to their principal β-η-normal forms, should they exist [2] [3].

In the present paper this parenthesis machine is described (section 1)

(1) Istituto di Scienza dell'Informazione, Torino.

and then directly applied. Section 2 shows a scheme P of parenthesis languages, solving some programming problems, such as :

1) the recursive calls of procedures with uniform conventions;
2) the transmission of parameter values.

Lastly, section 3 exhibits the embedding of a general bottom-up analyzer for context-free, total precedence languages in this parenthesis machine.

1. A PARENTHESIS MACHINE

We consider a parenthesis machine consisting of a stack and an auxiliary memory, that is a set of $\langle$ name, value $\rangle$ pairs (environment). The elementary actions of this machine are the copying of a string onto the stack and the interpretation of a given string as a command. The commands can modify the memory state, which in its turn determines the process of the computation because every « name » enclosed by a pair of parentheses at the top of the stack is replaced by the corresponding « value ». The machine evaluates a string built from an alphabet together with a legitimate use of parentheses which subdivide it arbitrarily. This string is copied from left to right on the stack until the first right parenthesis is reached. The point of interruption of copying is, as usual, denoted by a pointer. The right parenthesis constitutes the order to interpret the substring, enclosed between it and the corresponding left parenthesis, as a command. For this purpose it is sufficient (with regards to the parenthesis structure) that the string be « dynamically » legal, i.e., that the strings written onto the stack have legal parenthesis structures [1], which does not necessarily imply that the input string be « statically » legal, i.e., that it itself, have a legal parenthesis structure. This balance between the number of left and right parentheses has one exception; to make possible the elimination of the *go to* (see section 2.2) we assume, naturally enough, that a right parenthesis, to which no left parenthesis corresponds, acts as a STOP command.

EXAMPLE. The following is an example which, although not corresponding to any programming language, shows the evaluation technique of this machine. Let us assume that every string which does not occur as a « name » in the environment must be replaced on the stack with itself. We now give the successive stack configurations for the evaluation of the string :

$$(1(42)(3(114)5)(6(78)8))$$

(1) We say that a parenthesis structure is legal iff, scanning it from left to right, the number of left parentheses is always not less than the number of right parentheses, and they are equal at the end of the structure.

when the environment consists of the following pairs :

$$\langle 42,1 \rangle \langle 78,3 \rangle \langle 114,2 \rangle \langle 325,1 \rangle \langle 111638,5 \rangle$$

```
stack configurations        pointer positions
(1(42)                      (1(42)↑(3(114)5)(6(78)8))
(11(3(114)                  (1(42)(3(114)↑5)(6(78)8))
(11(325)                    (1(42)(3(114)5)↑(6(78)8))
(111(6(78)                  (1(42)(3(114)5)(6(78)↑8))
(111(638)                   (1(42)(3(114)5)(6(78)8)↑)
(111638)                    (1(42)(3(114)5)(6(78)8))↑
   5                        (1(42)(3(114)5)(6(78)8))↑
```

The string evaluated in this example has a statically legal parenthesis structure. Examples of programs with parenthesis structures only dynamically legal are given in section 2.

2. A SCHEME P OF PARENTHESIS LANGUAGES

Let us describe a scheme *P* of parenthesis languages. Every program of this scheme consists of a string of a given alphabet merged into a parenthesis structure dynamically legal (the only exception being a right parenthesis without a corresponding left one which acts as STOP command). The types of strings, which can occur enclosed in a parenthesis pair at the top of the stack, i.e., the types of commands, are six. Each is replaced uniquely at the top of the stack by a given string and can modify the environment. The following table lists the six types of strings, the strings replacing each one of these on the stack top, and the possible modifications of the environment :

TYPE NAME	TYPE OF STRING	TO BE REPLACED BY	MODIFICATION OF ENVIRONMENT
Functional string	$(f^{(n)}x_1 \ldots x_n)$	$f^{(n)}[x_1, \ldots, x_n]$	–
Predicative string	$(\alpha^{(n)}x_1 \ldots x_n)$	$\alpha^{(n)}[x_1, \ldots, x_n]$	–
Assigment string	(VALUE = : NAME)	ε	⟨NAME, VALUE⟩ (1)
Name string	(NAME)	VALUE (of the corresponding pair)	–
Jump string	($\vee$ LABEL)	String following ($\wedge$ LABEL) (2)	–
Label string	($\wedge$ LABEL)	ε	–

(1) If a pair whose first component is NAME already exists, it is deleted.
(2) This string must be uniquely determined, otherwise the program is incorrect.

where $f^{(n)}$ is an n-ary function, $\alpha^{(n)}$ is an n-ary predicate, ε denotes the empty string and $\vee$ and $\wedge$, respectively, true and false. It is not necessary to give formal definition of « LABEL », because any string following the two symbols $\vee$ and $\wedge$ is considered as a « LABEL ».

Example. Program for the computation of the G.C.D. of two integers a, b, with $a \geqslant b > 0$.

Let us define the following unary predicate and binary function :

$$\mathfrak{z}[x] = \begin{cases} \vee & \text{if} \quad x = 0 \\ \wedge & \text{if} \quad x \neq 0 \end{cases}$$

$m[x, y]$ = remainder upon division of x *by* y.

Then the desired program is :

$$(a(b(\wedge 1) = :x) = :y)((\mathfrak{z}(m(y)(x)))2)((x)((m(y)(x))(\vee 1)(\wedge 2)(x).$$

The successive stack configurations, environments and pointer positions for the case $a = 6$, $b = 4$ follow.

```
stack configurations          environments     pointer positions
(6(4(Λ1)                                         (6(4(Λ1)↑=:x...
(6(4=:x)                                         (6(4(Λ1)=:x)↑=:y...
(6=:y)                        <x,4>              (6(4(Λ1)=:x)=:y)↑((...
((ʒ(m(y)                      <x,4><y,6>         ...((ʒ(m(y)↑(x)))...
((ʒ(m6(x)                         "              ...((ʒ(m(y)(x)↑ ))...
((ʒ(m64)                          "             ...((ʒ(m(y)(x))↑ )...
((ʒ2)                             "             ...((ʒ(m(y)(x)))↑2)...
(Λ2)                              "              ...)))2)↑((x)((m(y)(x))...
((x)                              "              ...((x)↑((m(y)(x))...
(4((m(y)                          "              ...(x)((m(y)↑(x))...
(4((m6(x)                         "              ...(x)((m(y)(x)↑)...
(4((m64)                          "              ...(x)((m(y)(x))↑(V1)...
(4(2(V1)                          "              ...(V1)↑(Λ2)(x)...
(4(2=:x)                          "              ...(Λ1)=:x)↑=:y)...
(4=:y)                        <x,2><y,6>         ...(Λ1)=:x)=:y)↑((ʒ(m...
((ʒ(m(y)                      <x,2><y,4>         ...((ʒ(m(y)↑(x)))...
((ʒ(m4(x)                         "              ...((ʒ(m(y)(x)↑))...
((ʒ(m42)                          "              ...((ʒ(m(y)(x))↑)...
((ʒ0)                             "              ...((ʒ(m(y)(x)))↑2)...
(V2)                              "              ...)))2)↑((x)((m(y)(x))...
(x)                               "              ...(Λ2)(x)↑
2                                 "              ...(Λ2)(x)↑
```

2.1 Completness with respect to the partial recursive functions

We now give a constructive proof of the following fact : P is complete with respect to the partial recursive functions. Following [1] we use the fact that every partial recursive function can always be obtained from the basic functions :

$0^{(m)}$ zero function of m variables $0^{(m)}[x_1, \ldots, x_m] = 0$ $(m \geqslant 0)$

S^+ successor $S^+[x] = x + 1$
S^- predecessor $S^-[x] = x - 1$
$U_i^{(m)}$ selection function $U_i^{(m)}[x_1, ..., x_m] = x_i$ $(0 \leqslant i \leqslant m)$
by means of the successive applications of the following rules :

1) *Composition.* If $f^{(j)}$ is a partial recursive function and $g_i^{(m)}$ are partial recursive functions (where $m \geqslant 0, j > 0, 1 \leqslant i \leqslant j$), then also function $h^{(m)}$ defined by

$$h^{(m)}[x_1,..., x_m] = f^{(j)}[g_1^{(m)}[x_1, ..., x_m], ..., g_j^{(m)}[x_1, ..., x_m]]$$

is partial recursive.

2) *Recursion.* If f, g are partial recursive functions of one variable, p is a partial recursive function of two variables, α is a recursive predicate of one variable, then the function h defined by :

$$h[x] = \begin{cases} f[x] & \text{if} \quad \alpha[x] \text{ is true} \\ p[g[x], h[g[x]]] & \text{if} \quad \alpha[x] \text{ is false} \end{cases}$$

is also partial recursive (1).

Since the functions of the scheme P have been left indeterminate until now it is possible for scheme P to include the operators corresponding to the above-mentioned basic functions. The proof is therefore completed by giving the program schemes which carry out the rules of composition and recursion :

Composition : $(f^{(j)}(g_1^{(m)}x_1 \ldots x_m) \ldots (g_j^{(m)}x_1 \ldots x_m))$

Recursion : $(a(x= :a)(\wedge 1)((\alpha(a))2)((g(a))= :a)((p(a)(\vee 3(\vee 1)(\wedge 3)(a))= :a)$
$(\vee 4)(\wedge 2)((f(a))= :a)(\wedge 4))$.

Peculiar to the program scheme P is the fact that the composition is directly interpreted without any translation. It should be noted that, in the recursion program, the number of left parentheses is not statically equal to that of right ones, but that legal parenthesis structures are always dynamically obtained. This trick has been used for the very purpose of being able to memorize the number of times that function p has been applied, in order to calculate its arguments correctly. The recursion is implemented without having a stack of values for every variable appearing as parameter of the recursive function. Really, only the current value of each variable is retained in the

(1) The restriction to one variable is possible, because a pairing function j (e.g. the cantorian one $j(x, y) = x + \binom{x + y + 1}{2.}$) and its inverses K, L such that $j(K(z), L(z)) = z$ are definable inside the given scheme and a n-tuple $\langle x_1, ..., x_n \rangle$ can be expressed as $j(j(...(j(0, x_1 + 1), x_2), ..., x_n)$.

environment, while all preceding values which must still be used are, in fact, conserved on the stack.

EXAMPLE. We present the successive stack configurations, environments and pointer positions, according to the above program scheme for recursion in the case $\alpha[x] = \wedge$ and $\alpha[g[x]] = \vee$.

stack configurations	environments	pointer positions
(a(x=:a)	<a,x>	(a(x=:a)↑(∧1)...
(a(∧1)	"	(a(x=:a)(∧1)↑((...
(a((α(a)	"	...(∧1)((α(a)↑)2)...
(a((αx)	"	...(∧1)((α(a))↑2)...
(a(∧2)	"	...))2)↑((g(a))=:a)...
(a((g(a)	"	...((g(a)↑)=:a)...
(a((gx)	"	...((g(a))↑=:a)...
(a((g[x]=:a)	"	...((g(a))=:a)↑((p(a)...
(a((p(a)	<a,g[x]>	...((p(a)↑(∨3(∨1)...
(a((pg[x](∨3(∨1)	"	...(∨3(∨1)↑(∧3)...
(a((pg[x](∨3((α(a)	"	...(∧1)((α(a)↑)2)...
(a((pg[x](∨3((αg[x])	"	...((α(a))↑2)...
(a((pg[x](∨3(∨2)	"	...((α(a))2)↑((g(a))...
(a((pg[x](∨3((f(a)	"	...(∧2)((f(a)↑)=:a)...
(a((pg[x](∨3((fg[x])	"	...((f(a))↑=:a)...
(a((pg[x](∨3(f[g[x]]=:a)	"	...((f(a))=:a)↑(∧4))
(a((pg[x](∨3(∧4)	<a,f[g[x]]>	...((f(a))=:a)(∧4)↑)
(a((pg[x](∨3)	"	...((f(a))=:a)(∧4))↑
(a((pg[x](a)	"	...(∧3)(a)↑)=:a)...
(a((pg[x]f[g[x]])	"	...(∧3)(a))↑=:a)...
(a(p[g[x],f[g[x]]]=:a)	"	...(∧3)(a))=:a)↑(∨4)...
(a(∨4)	<a,p[g[x],f[g[x]]]>	...(a))=:a)(∨4)↑(∧2)...
(a)	"	...(∧4))↑
p[g[x],f[g[x]]]	"	...(∧4))↑

2.2 Inclusion of procedure feature

The introduction of procedures into this scheme P can be attained in a fairly natural way by allowing the value associated to a certain name to be a program. In this case, the creation of some pairs $\langle$ name, value $\rangle$ act as procedure definitions, and the replacement on the stack of the corresponding names (in parentheses) by their values act as procedure calls. Clearly [6], it is possible to eliminate the labels by means of procedures, thus representing more directly the links due to jumps. This modifies the evaluation of the jump string as follows :

symbols $\vee$ and $\wedge$ act as commands to copy onto the stack the string up to and including the right parenthesis corresponding to the left one which precedes them :

— symbol $\vee$ invokes the computation to proceed with the string which follows it, i.e. : $(\vee(\ldots))$ has as value $(\ldots)$

— symbol $\wedge$ invokes the deletion of the string which follows it, i.e. : $(\wedge(\ldots))$ has as value ε.

EXAMPLES. The following procedures :

$$d[x] = \begin{cases} f[x] & \text{if} \quad \alpha[x] \text{ is true} \\ d[g[x]] & \text{if} \quad \alpha[x] \text{ is false} \end{cases}$$

$$r[x] = \begin{cases} h[x] & \text{if} \quad \alpha[x] \text{ is true} \\ f[r[g[x]]] & \text{if} \quad \alpha[x] \text{ is false} \end{cases}$$

correspond, respectively, to the program schemes : $(x(d)$ $(x(r)$ with the environment :

$$\langle\, d, = :a)((\alpha(a))((a)(p)))((g(a))(d) \,\rangle \quad \langle\, p, = :b)(f(b))) \,\rangle$$
$$\langle\, r, = :a)((\alpha(a))((h(a))(s)))(f((g(a))(r) \,\rangle \quad \langle\, s,)(s) \,\rangle$$

The above programs exemplify cases where a right parenthesis without a corresponding left one interrupts the computation.

3. BOTTOM-UP ANALYSIS

We implement now a general bottom-up analyzer in the parenthesis machine described in section 1. The following hypotheses should be made about the grammar according to which we will parse the input string :

1) The grammar is a context-free, total precedence grammar [1].

2) Parenthesis symbols must not belong to the alphabet of terminals.

To carry out the bottom-up analysis of a string $x_1 \ldots x_n$, according to a grammar that satisfies the above conditions, it is sufficient to retain as the environment the matrix of the precedence among the terminal and non-terminal symbols and the production rules of the grammar itself, and to give as a program the string to be examined, parenthesized as follows :

$$(\# \, x_1) \ldots x_n) \, \#)$$

where the symbol « $\#$ » identifies the beginning and the end of the string.

The three possible relations $\doteq$, $<\!\cdot$, $\cdot\!>$ between two symbols y and z, terminal or not, give rise respectively to the following environment :

$$\begin{array}{ll} \langle\, yz, (y((z \,\rangle & \text{if} \quad y <\!\cdot\, z \\ \langle\, yz, y(z \,\rangle & \text{if} \quad y \doteq z \\ \langle\, yz, y)(z \,\rangle \,\rangle & \text{if} \quad y \cdot\!> z \\ \langle\, yz, \text{ERROR} \,\rangle & \text{otherwise} \end{array}$$

(1) For this and other definitions, we refer to the papers of Wirth and Weber [8] and Colmerauer [4].

The relations between every symbol y (terminal or not) and the symbol $\#$ are expressed by the environment :

$$\langle \# y, (\# ((y \rangle$$
$$\langle y \#, y) \#) \rangle$$

The p production rules of the grammar :

$$u_i \to \sigma_i \qquad (1 \leqslant i \leqslant p)$$

give rise to the following environment :

$$\langle \sigma_i, u_i) \rangle \qquad (1 \leqslant i \leqslant p).$$

To the previous environment should be added the following two pairs :

$$\langle S, S \rangle$$
$$\langle \# S \#, \text{YES} \rangle$$

so that at the end of the computation the stack should contain « YES » iff the examined string belongs to the language. If, on the other hand, the computation is interrupted because the top of the stack contains ERROR, then the string does not belong to the language.

Example. Let us consider the following grammar satisfying the desired conditions taken from [4] :

$$S \to a, \; S \to aSB, \; S \to bSB, \; B \to b$$

whose precedence matrix is :

$$\begin{array}{c|cccc}
 & \mathbf{S} & \mathbf{B} & \mathbf{a} & \mathbf{b} \\
\hline
\mathbf{S} & \cdot & \doteq & \cdot & \lessdot \\
\mathbf{B} & \cdot & \gtrdot & \cdot & \gtrdot \\
\mathbf{a} & \doteq & \gtrdot & \lessdot & \lessdot \\
\mathbf{b} & \doteq & \gtrdot & \lessdot & \lessdot
\end{array}$$

For our evaluation mechanism, the environment corresponding to the production rules are therefore :

$$\langle a, S) \rangle \quad \langle aSB, S) \rangle \quad \langle bSB, S) \rangle \quad \langle b, B) \rangle$$

and those corresponding to the precedence matrix :

$$\langle SS, \text{ERROR} \rangle \langle SB, S(B \rangle \langle Sa, \text{ERROR} \rangle \langle Sb, (S((b \rangle$$
$$\langle BS, \text{ERROR} \rangle \langle BB, B)(B \rangle \langle Ba, \text{ERROR} \rangle \langle Bb, B)(b \rangle$$
$$\langle aS, a(S \rangle \langle aB, a)(B \rangle \langle aa, (a((a \rangle \langle ab, (a((b \rangle$$
$$\langle bS, b(S \rangle \langle bB, b)(B \rangle \langle ba, (b((a \rangle \langle bb, (b((bb \rangle$$

to which should be added the following pairs which concern the symbol $\ddagger$:

$$\langle \ddagger S, (\ddagger ((S \rangle \langle \ddagger B, (\ddagger ((B \rangle \langle \ddagger a, (\ddagger ((a \rangle \langle \ddagger b, (\ddagger ((b \rangle$$
$$\langle S \ddagger, S) \ddagger)) \rangle \langle B \ddagger, B) \ddagger)) \rangle \langle a \ddagger, a) \ddagger)) \rangle \langle b \ddagger, b) \ddagger)) \rangle$$

and lastly :

$$\langle S, S \rangle \langle \ddagger S \ddagger, \text{YES} \rangle.$$

We now plot step by step the configurations of the stack during the parsing of the string *ababb*, to which corresponds the computation of the program : $(\ddagger a)b)a)b)b) \ddagger)$.

```
Stack configurations
1. (‡a)                          15. (‡((a(b(SB)
2. (‡((ab)                       16. (‡((a(bS(B(B‡)
3. (‡((a((ba)                    17. (‡((a(bS(BB)
4. (‡((a((b((ab)                 18. (‡((a(bSB)
5. (‡((a((b((a((bb)              19. (‡((aS)
6. (‡((a((b((a((b((b‡)           20. (‡(a(S(B‡)
7. (‡((a((b((a((b(b)             21. (‡(a(SB)
8. (‡((a((b((a((bB)              22. (‡(a S(B‡)
9. (‡((a((b((a(b)                23. (‡(aSB)
10. (‡((a((b((aB)                24. (‡S)
11. (‡((a((b(a)                  25. (‡((S‡)
12. (‡((a((bS)                   26. (‡(S)
13. (‡((a(b(S(B(B‡)              27. (‡S‡)
14. (‡((a(b(S(BB)                28. YES
```

CONCLUSION

This paper presents some applications of an evaluation mechanism, provided with a stack and an auxiliary memory, to string manipulation. The evaluation technique of this mechanism is naturally by value, the only exception being the rule added to allow the conditional procedure calls described in section 2.2 which imposes evaluation by name. Actually, this mechanism could be used as a « definition technique » of some programming languages in the sense of [7]. The main difference with the Vienna method is that the latter uses abstract syntaxes in which the generation trees are non-ordered, as every branch has a label. To the contrary, the present parenthesis machine treats trees whose branches are unlabeled and which must therefore be ordered. This implies, together with the parsing, a normalization of the language to be interpreted with respect, for example, to the relative positions of operators and operands.

REFERENCES

[1] C. Böhm, *On a Family of Turing Machines and the Related Programming Languages*, ICC Bulletin, *3*, 3 (1964).

[2] C. Böhm and M. Dezani, *A CUCH-Machine : the Automatic Treatment of Bound Variables*, Int. Journal of Computer and Information Sciences, *1*, 2, (1972), pp. 171-186.

[3] C. Böhm and M. Dezani, *Notes on a CUCH-Machine : the Automatic Treatment of Bound Variables*, Int. Journal of Computer and Information Sciences, *2*, 2 (1973), pp. 157-160.

[4] A. Colmerauer, *Total Precedence Relations*, Journal of the ACM, *17*, 1, (1970), pp. 14-30.

[5] E. W. Dijkstra, *An Attempt to Unify the Constituent Concepts of Serial Program Execution*, in Symbolic Languages in Data Processing. ed. Gordon and Breach, Rome (1962), pp. 237-252.

[6] J. McCarthy, *Recursive Functions of Symbolic Expression and their Computation by machine : Part I*, Comm. ACM, *3*, 4, pp. 184-195.

[7] P. Wegner, *The Vienna Definition Language*, ACM Computing Surveys, *4*, 1 (1972), pp. 5-62.

[8] N. Wirth and H. Weber, *Euler : a Generalization of ALGOL and its Formal Definitions : Part I and II*, Comm. ACM, *9*, 1 (1966), pp. 13-25, *9*, 2 (1966), pp. 89-99.

ACKNOWLEDGMENT

The author is grateful to Prof. C. Böhm who encouraged this research and to Prof. G. Ausiello for his criticism.