Technical Report CS78004-R

SYMBOL MANIPULATION IN AN INTERACTIVE ENVIRONMENT

Donald C. S. Allison

Department of Computer Science
College of Arts and Sciences
Virginia Polytechnic Institute and State University
Blacksburg, Virginia 24061

July 1978

ABSTRACT

In an interactive environment the opportunity exists for the on-line execution of an algorithm represented as a sequence of ordered commands. In particular, when the sequence of commands defines a normal Markov algorithm, the on-line environment provides a practical testing ground for one of the basic theories of computation. In order to develop and run a Markov algorithm a specification is required which will be capable of execution by a computer. One possibility is to represent the Markov algorithm and its data as a sequence of LISP S-expressions and to process the S-expressions using an extended LISP interpreter. In this paper the translation of Markov algorithms to LISP S-expressions is discussed along with a minimum set of commands for a Markov processor. Some of the difficulties in constructing more complicated algorithms are also discussed and several enhancements are suggested which would make the resultant Markov processor more practical and easier to use.

Key words: Markov algorithm, LISP, interactive

C. R. categories: 5.26, 5.7

1. INTRODUCTION

Symbol manipulation can be considered as the transformation of sequences of symbols. Examples of such tasks are the reversal of the order of the symbols in a sequence or the combination of two sequences of characters into a third sequence. It is believed that these tasks form a basis for all algorithmic activity. There is no proof for such an assertion; however, evidence continues to accumulate in confirmation.

In 1954 A. A. Markov introduced the concept of a normal algorithm in which the basic discrete step or command is to replace characters from a linear sequence or string by other characters [1]. He also conjectured that any algorithmic activity involving the manipulation of characters could be specified by a normal algorithm. This thesis can never be tested in all generality since no independent finite means exist for characterizing all algorithmic activities. However, the evidence of all attempted experiments substantiates his claim and Markov algorithms have been shown to be equivalent to other independent means of specifying algorithmic activities such as Turing machines [2], general recursive functions [3] and POST systems [4].

The main purpose for the specification of algorithmic activity is that some device will be able to execute the algorithm. In the case of Markov algorithms a suitable vehicle or device for the execution of the algorithm is a processor which is an extension of a LISP interpreter. In the LISP programming language all algorithms and their data are represented as S-expressions. Therefore it is possible to translate a

sequence of Markov commands into a sequence of S-expressions for execution by a processor which is familiar with this form of specification.

In section 2 normal Markov algorithms are defined and applied to several examples (see also [5], [6]) while in section 3 the translation of Markov algorithms into LISP S-expressions is described. This section also describes a minimum set of LISP-like commands which would be necessary for the execution of a Markov algorithm on a LISP-based processor. In section 4 some of the difficulties involved in constructing more complicated algorithms is discussed and in section 5 several enhancements are put forward which, if implemented, would make the Markov algorithm processor more practical and easier to use.

2. MARKOV ALGORITHMS

2.1 Introduction

In a Markov algorithm the basic discrete step is a replacement of characters from a linear sequence or string by other characters. These steps are called production rules. For example, the rule ab⟶a (read as "ab is replaced by a") applied to some string bababc represents the replacement of the first (or left most) occurrence of ab by a. In this case the string becomes baabc. A further application would produce baac.

In a Markov algorithm each rule specifies a replacement to be made on a string to be found in a register, R. R is assumed to have a recognizable left end. The first character of (R) can always be picked out --- but its right end can be extended indefinitely [the notation '(R)' is shorthand for 'the contents of R' or the string in R']. At any given moment, however, (R) has a finite and measurable length.

In transforming (R) it is likely that the operation will not be on the entire string but rather on some small contiguous portion of it. It is important that the Markov algorithm processor is capable of recognizing the occurrences of a given substring within a given string. These occurrences may be several and may overlap.

A string of no symbols is called the null or empty string. If a given string has n symbols, the null string is said to have n+1 occurence in S, before the first symbol (the first occurrence) and between every two adjacent symbols.

The symbols appearing in Markov algorithms are members of sets of symbols which are called alphabets. The collection of symbols which make up the domain of the data for a particular algorithm is said to be its data alphabet.

2.2 Definitions

(i) A simple production is written as A⟶B where A and B are strings in some alphabet.

(ii) A conclusive production is written as A⟶.B. In (i) and (ii) 'A' is called the antecedent and 'B' the consequent.

(iii) The production A⟶B (or A⟶.B) is said to be applicable to a string S if there is at least one occurrence of A in S. Otherwise the production is not applicable to S. The production is

interpreted as "replace the first occurrence of A in S by B."

(iv) A Markov algorithm is a finite sequence $P_1, P_2, \ldots . P_n$ of Markov productions to be applied to strings in a given alphabet according to the following rules.

Let S be a given string. The string is searched to find the first production P_i whose antecedent occurs in S. If no such production exists, the operation of the algorithm terminates without change in S (in this case a stalemate has been reached). If such a production does exist the first such production is applied to S. If this is a conclusive production the algorithm terminates after its application (in this case the algorithm is said to be closed). If, however, the production is simple a new search is conducted, starting with P_1 and using the string S' into which S has been transformed. If the operation ultimately ceases with a string S* in the register then S* is the result of applying the algorithm to S. If the algorithm has a name, say M, then it is possible to write:

$$M(S) = S^*$$

[Note the similarity between this definition and that of a function defined in terms of an independent variable].

2.3 Examples and Notations

Consider some examples of Markov algorithms.

(i) Alphabet {a,b,c,d};

exl [qvar[a;b;c;d];

1: ad $\longrightarrow$ .dc;

2: ba $\longrightarrow$;

3: a $\longrightarrow$ bc;

4: bc $\longrightarrow$bba;

5: $\longrightarrow$a]

where (a) exl is the name of the algorithm.

(b) qvar represents "quoted variables" and elements of the qvar list match only themselves.

(c) each production has the form label: antecedent$\longrightarrow$consequent.

(d) the entire sequence of productions is enclosed in square brackets.

If this algorithm is applied to the initial strings dcb, dbc and bdc

exl (dcb): R = dcb $\xrightarrow{5}$adcb $\xrightarrow{1}$dccb.

exl (dbc): R = dbc $\xrightarrow{4}$dbba $\xrightarrow{2}$db $\xrightarrow{5}$adb $\xrightarrow{1}$dcb.

exl (bdc): R = bdc $\xrightarrow{5}$abdc $\xrightarrow{3}$bcbdc $\xrightarrow{4}$bbabdc $\xrightarrow{2}$bbdc
$\xrightarrow{5}$abbdc $\xrightarrow{3}$bcbbdc $\xrightarrow{4}$bbabbdc $\xrightarrow{2}$bbbdc
$\xrightarrow{5}$.....

Clearly the application of exl to bdc does not terminate.

(ii) Construct an algorithm app, which is to append a string S to any given string of some alphabet A. Let * be a marker that is not an element of A

app [qvar[*];

1: *x$\longrightarrow$x*, x$\in A$;

2: $*\longrightarrow .S$;

3: $\longrightarrow *]$

where "$x \epsilon A$" is read as "x is a member of A."

Consider the application of this algorithm to an arbitrary string P of alphabet A. Since P does not initially contain *, production 3 is used to obtain *P. Production 1 is then used to move the * past each symbol in P to obtain P*. At this point production 1 no longer applies and the second produces PS and halts.

Note the new concept introduced in production 1 —— the variable x which ranges over the symbols of A. (To avoid ambiguity the symbol x should not be a member of A.) Thus the first line is not really a production but rather a production schema which can be obtained by substituting the symbols of A for x.

Note also the importance of the order of the productions. For example if 1 and 2 were interchanged, the result would be to transform P to SP and not PS and the productions represented by $*x\longrightarrow x*$ would never be used.

(iii) Given a string P, construct an algorithm copy which produces PP. Let the alphabet be A and introduce markers *,#,↑, which are not in A.

copy [qvar[*;#;↑];

1: xy# ⟶ y#x, $x \in A \wedge y \in A$;

2: *x ⟶ x#x*, $x \in A$;

3: # ⟶ ↑;

4: ↑ ⟶ ;

5: * ⟶ .;

6: ⟶ *]

If A is {a,b,c} and P = abc

copy (abc): R = abc $\xrightarrow{6}$ *abc $\xrightarrow{2}$ a#a*bc $\xrightarrow{2}$ a#ab#b*c

$\xrightarrow{1}$ a#b#ab*c $\xrightarrow{2}$ a#b#abc#c* $\xrightarrow{1}$ a#b#ac#bc*

$\xrightarrow{1}$ a#b#c#abc* $\xrightarrow{3,3,3}$ a↑b↑c↑abc*

$\xrightarrow{4,4,4}$ abcabc* $\xrightarrow{5}$ abcabc

3. LISP PROCESSING OF MARKOV ALGORITHMS

Since the Markov algorithm processor is based on a LISP interpreter all algorithms and their data must be presented as LISP S-expressions [7].

3.1 S-Expression Representation of Markov Algorithms

The general form of a simple production is

label: antecedent ⟶ consequent;

This can be considered as an M-expression which translates into the LISP S-expression

label* ((antecedent*) (consequent*))

For example, production 3 in ex1, namely

3: a⟶bc

becomes 3 ((A)(B C))

A conclusive production has the format

label: antecedent ⟶ . consequent

and this translates into

label* ((antecedent*)(STOP(consequent*)))

For example production 1 in exl, namely

1: ad⟶.dc

becomes 1 ((A D)(STOP(D C)))

In production 1 of app the concept of a variable was introduced

1: *x⟶x* x∈A;

where the variable x may match any symbol of the alphabet *A*. Alphabets are declared as LISP constants through the function cset. Thus if *A* is {a,b,c,d} the declaration command is CSET (ALPHA(A B C D)) where ALPHA is the translation of *A*.

The symbols x∈A are considered a predicate——a condition to be satisfied before a match for x is accepted. Therefore the form of a production is extended to include a new field as follows:

label: antecedent⟶consequent, predicate

which translates into

label*((antecedent*)(consequent*)(predicate*))

The predicate x∈A becomes the S-expression (MEMBER X ALPHA) and the predicate field takes the form of an association list where each predicate is associated with the antecedent symbol it qualifies. Thus the predicate field for this example is ((X.(MEMBER X ALPHA))) which is more normally written in list format as ((X MEMBER X ALPHA)). The entire production becomes

1((* X)(X *)((X MEMBER X ALPHA)))

More complicated productions such as production 1 of copy become

1((X Y #)(Y # X)((X MEMBER X ALPHA)(Y MEMBER Y ALPHA)))

Up to this point labels have been represented by the digits 1,2,... This is merely to ease reference to individual productions. In fact a label may be any allowable LISP atom, and the labels chosen do not in any way affect the order of application of the productions.

Finally the qvar list qvar[e_1; e_2;.....e_n] transcribes into QVAR(e_1* e_2*.....e_n*)

As an illustration, consider the translation of an entire definition, say that for copy given earlier, namely.

[qvar[*;#;↑];

1: xy# ⟶ y#x, $x \in A \wedge y \in A$;

2: *x ⟶ x#x*, $x \in A$;

3: # ⟶ ↑;

4: ↑ ⟶ ;

5: * ⟶ .;

6: ⟶ *]

This becomes the S-expression

(QVAR(* # ↑)

1 ((X Y #)(Y # X)((X MEMBER X ALPHA)(Y MEMBER Y ALPHA)))

2 ((* X)(X # X *)((X MEMBER X ALPHA)))

3 ((#)(↑))

4 ((↑)( ))

5 ((*)(STOP))

6 (( )(*)))

3.2 Matching of Symbols

In the definition of a Markov Algorithm in section 2.2 it was stated that a search is made for the first antecedent which occurs in (R). If one is found then the antecedent "matches" (R). In LISP S-expressions an antecedent is a list (possibly empty) of atomic symbols. There are two categories of symbols.

1. qvar symbols —— which appear in the qvar list and are thus constrained to match only themselves; for example, markers or individual members of an alphabet. Numbers are also treated in this way, 1 will match only 1, 2 will match only 2.

2. All other symbols may match any element in (R) constrained only by an associated predicate if one exists.

Note the uniqueness imposed upon matching——if a symbol appears more than once in an antecedent, each associated "match" must be the same.

examples (X X) matches (A A) assuming predicates are satisfied

but (X X) does not match (A B)

however (X Y) matches (A A) assuming predicates are satisfied

3.3 Markov Commands

In an interactive environment a number of editing commands are both necessary and desirable. A minimum list of commands is the following:

(i) MARKOV (name, definition)

The Markov algorithm definition given by the second argument is stored under the name given by the first argument. The Markov algorithm could be invoked by use of this name.

example MARKOV (COPY(QVAR(* # +)

1 (———)
:
:
6 (———)))

A value (name) will be returned——(COPY) in this example.

(ii) DISPLAY (name)

The productions of the Markov algorithm named by the argument will be printed out with each production starting on a new line.

(iii) DELRULE (name, label)

The production whose label is the second argument will be deleted from the Markov algorithm named in the first argument. No value will be returned.

example DELRULE (COPY 3)

(iv) ADDRULE (name, label, new production)

The new production, given by the third argument will be added to the Markov algorithm named by the first argument immediately before the production whose label is given as the second argument. If the second argument is NIL, the new production will be added after all existing productions. No value will be returned.

example ADDRULE (COPY 3 (21((# *)(*))))

will add the production

21 ((# *)(*))

to the Markov algorithm COPY immediately in front of production 3.

(v) QVAR (name, qvarlist)

The qvar list of the Markov algorithm named by the first argument will be replaced by the list given as the second argument. No

value will be returned.

example QVAR(COPY(* # ↑ @))

(vi) TRACE (list of names)

This will set the trace flag in each of the named Markov algorithms. Thereafter (R) will be printed out after the application of each successful production in the execution of a named Markov algorithm.

example TRACE((COPY APP EX1))

(vii) UNTRACE (list of names)

This will clear the trace flag in the named Markov algorithms.

4. DEVELOPING ALGORITHMS

4.1 Introduction

Consider some of the difficulties which arise when non-trivial problems are attempted. A typical problem is the following:

Given P_1 * P_2 in R, where P_1 and P_2 are strings from some alphabet A, construct a Markov Algorithm test which determines whether $P_1 = P_2$, that is whether P_1 and P_2, contain the same symbols in the same order. Leave R empty if $P_1 = P_2$ and non-empty otherwise.

There are many possible strategies for the solution of this problem, one of which is:

(i) Reverse P_2 that is, say (R) is abc*abc, then form abc*cba.

(ii) Delete equal pairs of characters from either side of the * symbol.

(iii) If symbols on either side of * are not equal replace * by #.

(iv) Finally, if R contains *, delete it and halt implying $P_1 = P_2$ with R empty; otherwise stalemate with non-empty R.

The first step (i) is the construction of an algorithm rev which

reverses (R). One possible algorithm which uses four productions is:

rev [qvar[*];

1: $*xy \longrightarrow y*x, x \in A \wedge y \in A$;

2: $**x* \longrightarrow x**, x \in A$;

3: $**x \longrightarrow .x, x \in A$;

4: $\longrightarrow *$]

If (R) = abc then

rev (abc): $R = abc \xrightarrow{4} *abc \xrightarrow{1} b*ac \xrightarrow{1} bc*a$

$\xrightarrow{1} *bc*a \xrightarrow{1} c*b*a \xrightarrow{4,4} **c*b*a$

$\xrightarrow{2} c**b*a \xrightarrow{2} cb**a \xrightarrow{3} cba$

The second (ii), third (iii) and fourth (iv) steps are easily implemented as the MARKOV algorithm comp

comp[qvar[*];

1: $x*x \longrightarrow *$, $x \in A$;

2: $x*y \longrightarrow x\#y$, $x \in A \wedge y \in A$;

3: $* \longrightarrow .$]

Therefore the required Markov Algorithm test may be formulated as

$$test\ (R) = comp(P_1 * rev(P_2))$$

However, this raises two problems:

(a) How can an algorithm which works on (R), be made to operate on some substring of (R)? In the above problem rev reverses (R) but it is only required to reverse that part of (R) following the marker *. This is called a focussing problem.

(b) How can two given algorithms be combined into one algorithm which is equivalent to the composition of the two? In the above problem test is constructed as the composition of comp and rev.

4.2 Focussing Problem

The general problem may be stated as follows: Given * and # not in A and (R) = $Q_1*Q_2\#Q_3$ where Q_1,Q_2 and Q_3 are strings over A construct a Markov algorithm m2 such that

$$m2(R) = Q_1*m1(Q_2)\#Q_3$$

Here m2 is algorithm m1 focussed upon the substring Q_2. The initial strings Q_1*Q_2 and $Q_2\#Q_3$are special cases of the more general string $Q_1*Q_2\#Q_3$.

Consider the following algorithm:

change [qvar[*];

1: $*x \longrightarrow *x'$ $x \in A$;

2: $x'y \longrightarrow x'y'$ $x' \in A' \wedge y \in A$]

Here the concept of an "alias" alphabet has been introduced where for any alphabet A there exists an infinite number of alias alphabets A', A'',...., such that for every $x \in A$ there is a corresponding $x' \in A'$, $x'' \in A''$.....For example if A were {a,b,c,d} then A' would be {a',b', c',d'} and A'' would be {a",b",c",d"}.

An examination of change shows that since $\# \notin A$ change (R) produces $Q_1*Q'_2\#Q_3$. For the special case Q_1*Q_2 change (Q_1*Q_2) yields Q_1*Q_2' but for $Q_2\#Q_3$ a modified algorithm change 1 is required which operates on the first half of the string to produce $Q'_2\#Q_3$.

change 1 [qvar[#];

1: $x\# \longrightarrow x'\#$ $x \in A$;

2: $xy' \longrightarrow x'y'$ $x \in A \wedge y' \in A'$]

In all cases after processing is complete the alias alphabet may be deleted by a production such as:

1: $x' \longrightarrow x,$ $\in A'$

For the test problem under consideration the first step is therefore to change P_2 to P_2' and then modify rev to operate on strings over the alias alphabet A'.

4.3 Composition Problem

The general composition problem may be stated as follows:
Given two algorithms m1 and m2, construct m3 such that

$$m3(R) = m2(m1(R))$$

Note that simple appending m2 to m1 (written as m1; m2) will not usually work because the execution of the m2 production rules will lead to the rules of m1 being rescanned and this may lead to further changes being invoked by m1. Clearly once m1 is completed some auxiliary rules must be applied which prevent these rules and the rules of m1 from being applicable again.

Let α be a marker not in the data alphabets of m1 or m2. Convert m1 into $m1^{(1)}$ by modifying all terminal productions in m1 so that they insert α into some point in R and so that they are no longer conclusive. For example $U \longrightarrow .V$ becomes $U \longrightarrow \alpha V$.

Consider the algorithm

n1[qvar[α];

1: $x\alpha \longrightarrow \alpha x$, $x \in A$;

2: $\alpha x \longrightarrow \alpha x'$, $x \in A$;

3: $x'y \longrightarrow x'y'$, $x' \in A' \wedge y \in A$]

which moves the marker α to the left end of the string over A and then converts that string to a corresponding string over A'.

Then n2 = n1; $x' \longrightarrow .x'$, $x' \in A$; $m1^{(1)}$
(stopping condition)

gives $n2(R) = \alpha m1(R)'$, that is m1(R) is computed, then converted to alias form and preceded by a marker α.

Suppose now that m2 is changed to m2' to operate an A' and further that $m2'^{(1)}$ is defined by altering those productions of m2' which reference the left end of (R) to depend on α which is now at the left end of (R). Then substituting $m2'^{(1)}$ for the "stopping condition" in n2 gives

$$n3 = n1;\ m2'^{(1)};\ m1^{(1)}$$

where $n3(R) = \alpha m2'(m1(R)')$

To finish the construction eliminate α and convert all aliases back to symbols in A. This can be accomplished by the algorithm n4. Let β be a marker not in the data alphabets of m1 or m2.

n4[qvar[α;β];

1: $x'\beta \longrightarrow \beta x'$, $x' \in A'$;

2: $\beta x' \longrightarrow \beta x$, $x' \in A$;

3: $xy' \longrightarrow xy$, $x \bar{\in} A \wedge y' \in A$;

4: $\alpha\beta \longrightarrow$.]

If $m2'^{(1)}$ is converted to $m2'^{(2)}$ by changing terminal rules so that they insert β into R and are no longer conclusive, then m3 may be written as

$$m3 = n4;\ n1;\ m2'^{(2)};m1^{(1)}$$

Here a general method of composition has been defined because m3 may now take part in further compositions in its own right, provided that additional markers and alias alphabets are introduced.

5. MORE EFFECTIVE STRING PROCESSING

There are several features which may be added to a basic Markov processor to make it more practical and easier to use. The choice of these features will obviously be influenced by the fact that the processor is LISP based. Consider facilities under three headings (1) sequencing, (2) evaluation of

a matched instance and (3) structured strings.

5.1 Sequencing

The rigid nature of the control rule is the basic reason behind many difficulties in the construction of Markov algorithms. Since processing must always continue at the first production after a successful application and at the next production after a failure ordering problems arise which can often only be resolved by the introduction of extra markers and/or alias alphabets. Some of these difficulties can be resolved by the introduction of sequencing — the association of a sequence field with each production which specifies a label to "go to" on failure. If no sequence field or a partial sequence field is specified the basic control rule operates. Every sequenced Markov algorithm can be shown to be equivalent to a normal Markov algorithm so the theoratical results obtained for Markov's general theory of algorithms are in no way compromised by using sequenced Markov algorithms.

The format of a production becomes

label: antecedent⟶consequent, predicate/sequence field;

where the sequence field takes the form S label 1 F label 2.

As an example of the use of sequencing consider the algorithm copy discussed in section 2. Using sequencing it may now be defined as:

```
copy [qvar[*;#];
        1:      ⟶*/S 2;
        2:   *x⟶x#x*, x∈A/S 3 F 4;
        3:  xy#⟶y#x, x∈A∧y∈A/S 3 F 2;
```

4: #⟶ /S 4;

5: *⟶ .]

Then copy (abc): R = abc $\xrightarrow{1}$ *abc $\xrightarrow{2}$ a#a*bc

$\xrightarrow{3,2}$ a#ab#b*c $\xrightarrow{3}$ a#b#ab*c

$\xrightarrow{3,2}$ a#b#abc#c* $\xrightarrow{3}$ a#b#ac#bc*

$\xrightarrow{3}$ a#b#c#abc* $\xrightarrow{3,2,4}$ ab#c#abc*

$\xrightarrow{4}$ abc#abc* $\xrightarrow{4}$ abcabc*

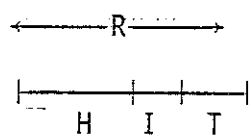
$\xrightarrow{4,5}$ abcabc

The S-expression definition for copy would be

```
(QVAR(* #)
1(( )(*)/S 2)
2((* X)(X # X *)((X MEMBER X AL))/S 3 F 4)
3((X Y #)(Y # X)((X MEMBER X AL)(Y MEMBER Y AL))/S 3 F 2)
4((#)( )/S 4)
5((*)(STOP)) )
```

5.2 Evaluation of a Matched Instance

In a Markov algorithm processor (with or without sequencing), when a production of the form A⟶B is applied to (R), the substring corresponding to the leftmost instance I of A in (R) is isolated together with two other substrings called the head H and the tail T.



Note that substring I is an instance of A with all the variables in A bound to particular values. I is then replaced by J which is the instance of B in which the variables are replaced by the bindings set

up in matching I with A. Hence (R) becomes the concatenation of H, J and T.

Consider now the situation where the creation of J may involve other "evaluation processes" in addition to simple substituting for bound variables. For example, consider an algorithm to "simplify" an arithmetic expression. This might contain a production of the form

label: $a*b \longrightarrow$ times (a,b), $a\epsilon N \wedge b\epsilon N$;

where N is the set of numbers and times is a function whose value is the product of its arguments. Note that N is not a finite alphabet. However, the tests $a\epsilon N$ and $b\epsilon N$ can be easily represented through one of the available LISP arithmetic predicates, for example numberp, fixp, floatp [7].

Thus an algorithm, simp, which is to simplify an unparenthesized arithmetic expression consisting only of integers and the operators + and * could be written as:

```
simp [qvar[*;+];
        1:  a*b⟶times(a,b),aϵI∧bϵI;
        2:  a+b⟶plus(a,b),aϵI∧bϵI/S 2;
    last    ⟶ . ]
```

where I represents the set of integers. Note that by default the sequence field in production 1 is S 1 F 2 and also that operator precedence influences the ordering of the productions.

As an example consider simp(6+4*3+2)

R: $6+4*3+2 \xrightarrow{1} 6+12+2 \xrightarrow{1,2} 18+2 \xrightarrow{2} 20 \xrightarrow{2,3} 20.$

Since times and plus could be described as normal Markov algorithms, the techniques of focussing and composition could be used to construct

simp as a normal Markov algorithm also. However, the construction of even this relatively simple function becomes a major task.

5.3 Structured Strings

Consider the situation where R contains a "parenthesized" expression which would normally be represented as a tree or a list structure. For example, consider the extension of the simplification algorithm discussed above to cater for expressions such as 1+(2+6*(2+3))*4. This might be translated to some parenthesis free notation such as postfix or prefix. However, this approach is restricted to a certain range of problems.

It is therefore necessary to take account of structure when matching and this implies the ability to pick out parenthesized substrings. Using the facilities described so far it is only possible to reference particular instances so the production schemes must cover all possible instances of parenthesized substrings. Consider the problem of simplifying 1+<2+6*<2+3>>*4 where '<' and '>' are used instead of '(' and ')' because of the particular meaning attached to parentheses in the LISP environment. The algorithm might be written as:

```
simp [qvar[*;+];
        1:  <a*b>⟶times(a,b), a∈n∧b∈n;
        2:  <a+b>⟶plus (a,b), a∈n∧b∈n;
        3:   a*b ⟶times(a,b), a∈n∧b∈n;
        4:   a+b ⟶plus (a,b), a∈n∧b∈n;
     last:      ⟶ .]
```

where all rules involving < and > precede those which do not. Within

the < > rules the ordering is not important whereas within the other rules the order is determined by the precedence ordering of the operations involved.

This situation can be improved by the introduction of "string variables." Where previously a variable was constrained to match only a single element of a particular alphabet, a new type of variable is introduced which may match a string of symbols from some alphabet. This facility, together with the "evaluation of an instance", would have been very useful in the construction of test in section 4. There a production of the form 1: *x⟶*rev(x), x⊆A/s 2; where 'x⊆A' is read as 'x is a string over A', would have reversed the string following the * marker and left (R) ready for the comparision phase.

This becomes a very powerful facility if predicates can be associated with string variables (e.g. x⊆A∧length(x)≤3) and if four new functions are introduced, namely:

(i) first(x)——gives first element of string x

(ii) last(x) ——gives last element of string x

(iii) x(s)A ——x is a string over A bounded by '(' on left and ')' on right

(iv) strip(x)——removes first and last symbols from x,e.g. strip applied to (a*b) would yield a*b

Then the algorithm simp could be written as:

```
simp[qvar[*;+];
        1:   x⟶simp(strip(x)), x(s)A;
        2:  a*b⟶times(a,b), a∈n∧b∈n;
        3:  a+b⟶plus(a,b), a∈n∧b∈n;
```

last: ⟶ .]

In this algorithm production 1 caters for all parenthesized expressions by stripping off the parenthesis and calling simp recursively.

REFERENCES

[1] A. A. Markov, The Theory of Algorithms, U.S. Dept. of Commerce, Office of Technical Services, No. OTS 60-51085, 1960.

[2] A. M. Turing, Proc. London Math. Soc., ser. 2, vol. 2, p.230-265, 1936.

[3] S. C. Kleene, Introduction to Metamathematics, Van Nostrand, New York, 1952.

[4] E. L. Post, J. Symb. Logic, vol. 1, p. 103-105, 1936.

[5] B. A. Galler and A. J. Perlis, A View of Programming Languages, Addison Wesley, 1970.

[6] R. R. Korfhage, Logic and Algorithms, Wiley, 1966.

[7] J. McCarthy et als, LISP 1.5 Programmer's Manual, M.I.T. Press, 1965.