AN INTERACTIVE PHONOLOGICAL

RULE TESTING SYSTEM

Victoria A. Fromkin

and

D. Lloyd Rice

University of California, Los Angeles

September 1969

One of the many ways the high-speed computer is useful to linguistic researchers is for the evaluation of generative grammars. Several programming systems for this purpose have been described in the literature. [1,2,3,4] A transformational generative grammar consists of a syntactic component, a phonological component and a semantic component. This paper is concerned solely with the phonological component. While this component is a dependent part of the entire grammar, systems of phonological rules for specific languages, i.e, the phonological components of the grammars of these languages, have been separately presented by Chomsky and Halle [5], Kuroda [6], Schachter and Fromkin [7] and others. The Sound Pattern of English [5] (hereafter, SPE) includes the 'formalism used for presenting phonological rules and the schemata that represent them, and the interpretation of this formalism'. (p. 390) This formal description is taken as the basis for the rule structures discussed in this paper.

Chomsky and Halle state that 'The rules of the grammar operate in a mechanical fashion; one may think of them as instructions that might be given to a mindless robot, incapable of exercising any judgment or imagination in their application. Any ambiguity or inexplicitness in the statement of rules must in principle be eliminated, since the receiver of the instructions is assumed to be incapable of using intelligence to fill in gaps or to correct errors'. They find it 'a curious fact' however that 'this condition of preciseness of formulation...has led many linguists to conclude that the motivation for such grammars must be...some..use of computers'. We also believe that there are more basic theoretical motives in clarity and completeness; we further believe that this very explicitness makes possible the use of the computer for testing such rules.

Furthermore, the complexities of natural language are reflected in the components phonological rules. Anyone who has attempted to teach a group of graduate students the phonology of English, using the rules presented in SPE can attest to the fact that even a single rule schema presents endless problems for the brightest of students when he attempts to expand the schema and to apply this set of rules to convert an abstract surface structure of a sentence into its phonetic representation. While the linguist or the student may be possessed with greater intelligence than the mindless robot, he is also possessed with human fallibility, and limited time and energy. For these reasons, the mindless robot can perform far more effectively than a minded human. The computer program which is described in this paper was written to aid human phonologists in the writing of rules, the testing

of rules, and the teaching of phonology. The importance of such a computerized phonological rule tester becomes very apparent when one selects at random any twenty-five English words, attempts to provide what one assumes to be the underlying phonological representation, and then applies the rules of SPE as specified. One of the authors of this paper made such an attempt. After more hours than she wishes to remember, and using every possible underlying segment, she found that eleven of these randomly chosen words could not be correctly derived. Nor were these strange foreign loan words, unless one believes the word 'America' to be an exceptional item in the English vocabulary.

This example is not offered as a criticism of Sound Pattern of English, probably the most important published book on English phonology and phonological theory. Nor are we concerned here with any theoretical weaknesses which may or may not be present in this work. What is apparent is that had a phonological rule tester been used, prior to the publication of this set of rules, many of the problems in rule ordering, omitted contexts etc. could have easily been corrected, and those rules which present problems and which cannot work would have at least been revealed. Furthermore, because of the speed of the computer, one could have tested not only twenty-five words, but hundreds -- determining the correct underlying forms of formatives, and thereby providing a **lexicon on which the rules could operate.**

A major problem encountered in setting up a program for such a tester is the notational compromise often necessary between the computer input format and the rule description schemes used by linguists to express their phonological rules, The Phonological Rule Tester of Bobrow and Fraser [8] solves this problem by offering a variety of logical combinatorial devices which may be used to group either segments within a rule or complete rules for disjunctive or conjunctive application. Such a system has very general descriptive capability, but complex rules appear in the computer input form rather different from the linguist's format.

In consideration of the descriptive powers of such systems, it appears that the input format should be made as specific as possible to the proposed theoretical structure since a more general descriptive scheme requires the rule writer to learn a more powerful meta-language than is needed. This parallels the general direction of development of computer languages; from the general machine-oriented coding to the specific problem-oriented languages.

This paper describes the translational core or compiler of a system which accepts phonological rules in a format very close to that formalized in The Sound Pattern of English and produces as output the coding similar to phonetic segments necessary to evaluate the input rules in a phonological testing program. The input format of this system is especially applicable to keyboard entry on a CRT graphic terminal such as the IBM 2260 and is planned for possible use in an on-line classroom system for teaching the properties and operation of the phonological component, as well as for the writing and testing of phonological rules by the linguist.

The rule testing program consists of an input block, a sequence of phonological rules, and a printout block (see Figure 1). The input block will accept a string of characters from the operator's console representing the underlying form of a word or phrase or any form assumed to occur in a derivation

in the phonological component. This form is then tested against the environmental conditions specified by the stored rules and modified according to those rules whenever a match is found. The string of phonological units,i.e, segments and boundaries, and/or the binary matrix resulting from the application of any rule may be optionally displayed on the operator's console after the application of that rule.

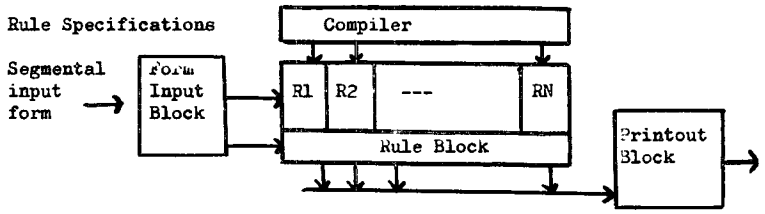

Figure 1: The Phonological Rule Tester

The structure of the program is such that any rule or sequence of rules can be tested using the same input and output blocks. The rules initially coded for testing and described in this paper are taken from Chomsky and Halle (1968). The program however, is not limited to these particular rules, but can be used to test any set of rules comprising the phonological component of a generative grammar.

The Input Format for Rule Description

The input to the phonological component consists of a structurally analyzed string consisting of syntactic brackets (e.g. Noun Phrase, Noun, Adjective, Verb etc), segments, and boundaries. The segments and boundaries are composite feature bundles.

The system used in our Phonological Rule Tester specifies these units in any of several ways:

a. As a combination of upper-case alphabetic characters representing the various phonological segments defined in the system;
b. As a cluster of distinctive feature specifications enclosed in angle brackets. These may be spaced horizontally or vertically, i.e.,
⟨+voc - cons - round⟩
⟨+voc⟩
⟨-cons⟩
⟨-round⟩,
but are to be considered simultaneously as a cluster rather than conjunctively as in the square bracketed series;
c. As a sequence of segments of specific predetermined types;

"V" indicates any vowel segment, i.e., defined as +voc
"C" indicates any non Vowel segment, i.e. a true consonant, a liquid or a glide, i.e., defined as either -voc or +cons ,
"X" indicates any sequence of units not containing the boundary unit #,
i"C" indicates a string of at least i consonants,
i,j"C" indicates a string of at least i and not more than j consonants,

d. As any of several boundary units #, +, or = , which signify themselves.
e. as a combination of brackets and upper-case alphabetic characters representing the syntactic brackets defined in the system.

Rules in the Phonological Component are of the form

$$A \rightarrow B \;/\; X \text{ -- } Y$$

where 'A and B are single units or the null element; the arrow stands for 'is actualized by'; the diagonal line means 'in the context'; and X and Y represent respectively, the left and right hand environments in which A appears. These may also be null, or may consist of units or strings of units and include labeled syntactic brackets.

Our system accepts rules written in this format, i.e.
LHS → RHS / context specification.

A rule is applicable if the LHS matches some unit in the test string and any context specified in the rule is found to exist at or adjacent to the matched unit. The context specification may consist of any sequence of one or more units, and must include a marker -- to indicate where the LHS fits into the specified context, or more exactly, how the environment must be configured around the matched unit in order for the rule to be applied.

In the Phonological Component of a Grammar, two partially identical rules may be coalesced into a single rule schema by enclosing the corresponding non-identical parts in braces, i.e. $A \rightarrow B \;/\; \text{--}\{\}$. Schema using such braces coalesce a conjunctive series of rules. The rules are applied in order. A conjunctive series of units is written in our program as a vertical list bounded left and right by columns of left and right square brackets. This corresponds to the braces. For example, given the phonological rule (1)

(1) $V \rightarrow \tilde{V}$ -- $\begin{bmatrix} +\text{Nasal} \\ [V] \end{bmatrix}$

has the interpretation that the rule will be tried first with the context specification consisting of the segment symbolized as N (i.e. nasal), and then with the context consisting of the segment V.

In our system a rule containing a conjunctive series is matched against the test string taking each of the conjunctive items in the order they appear in the series, applying the rule immediately any time the matching string including the current item matches the appropriate portion of the test string.

In the phonological theory underlying our system, rules may also be disjunctively ordered. Such rules are represented in schema by the use of parentheses and angled brackets.

A disjunction is written as a unit or sequence of units enclosed in parenthesis. It differs from the conjunctive series in the sequence of applicability to a particular test string. A disjunction is matched against the

test string by considering first the context including the disjunctive item. If this match is successful the rule is applied and no further matching is attempted. If the first match is unsuccessful, then the match is attempted omitting the disjunctive item, applying the rule if a match is found.

Clearly, the context must specify exactly one relative position of the LHS, marked by the double dash, --. Thus, the LHS position marker may be in a conjunctive series if it appears once in each item of the series, but it may not occur inside a disjunction.

The items of a conjunctive series or of a disjunction may in turn include either conjunctive series or disjunctions. Conjunctive series must be written with the bracket columns extending below and not above the line external to the conjunction. Extra spaces may be included either horizontally or vertically for clarity and in some cases may be needed for disambiguation. Rule (12) from SPE would be expressed as follows in this system:

```
                [æ --  [rC       ]              ]
ø → w /         [      [([r])    ]              ]
                [      [ [lm]   #]              ]
                [                               ]
                [ ⟨+round -voc +cons⟩ --        ]
```

A context specification may consist of stacked contexts according to the convention that

A → B /D -- E / C -- F

is interpreted as

A → B / C D -- E F.

System Structure

The rule testing program proper consists of 4 sections. They are: 1. the system storage definitions which include definition of the feature set used, 2. the mechanics necessary to accept an input form and set up the test matrix with the features of the input form, 3. a rule test loop which controls cyclic ordering, and 4. the routines to print out the results, either in segment string or in binary matrix form, following the application of any rule. The rules are then included as blocks of coding inserted as desired in the test loop.

Initially, four values are defined which determine the size of the various tables and matrices in the system.

```
DECLARE      L (60)  CHAR  (2);
DECLARE      F (60,20), M (50,20) BIT (1);
DECLARE      S (50)  FIXED;
```

The amount of memory reserved may be easily changed by altering only the lines defining the size limits.

An array L of CHARACTER STRING variables is declared to have length 60 and a logical matrix F is declared to have a length of 60 columns, each column having 20 elements or bits. Immediately after program execution has begun, input of the feature set is requested. The feature specification consists of the

character representation for each phonological unit followed by at least 1 space followed by an ordered string of + 's and - 's corresponding to the feature value assignment. The ordering is as in Table 2 below. The character representation of the nth unit entered is stored in the nth element of the string array L and the feature values are stored as 1's and 0's in the nth column of the feature reference matrix F (n 60). If less than 20 binary values are specified for any unit the remainder of the column is filled with 0's (i.e. -'s). Table 1 is a listing of the units and feature values used for testing the rules of English in the present study (from Chomsky and Halle, 1968).

Symbol		Phonemic Unit	Symbol		Phonemic Unit
UI	++-++----+	(ɨ)	TH	+-+---++---+--	(θ)
II	++-+-----+	(ɪ)	DH	+-+---++--++--	(ð)
UU	++-++---++	(ʉ)	N	+-+---++--+-+-	(n)
EE	++-------+	(ɵ)	S	+-+---++---+-+	(s)
OO	++--+---++	(ɒ)	Z	+-+---++--++-+	(z)
ə́ə́	++---+---+	(ɜ)	C	+-+---++-----+	(c)
AA	++--++---+	(ɑ)	CH	+-++---+-----+	(c͡ç)
OE	++---+--++	(ɶ)	J	+-++---+--+--+	(ɟ͡ʝ)
%%	++--++--++	(ɔ)	SH	+-++---+---+-+	(ʃ͡s)
I	++-+------	(i)	ZH	+-++---+--++-+	(ʒ͡ʑ)
U	++-++---+-	(u)	K	+-+++-------	(k)
E	++--------	(e)	G	+-+++-----+---	(g)
&	++--+-----	(ʌ)	X	+-+++------+--	(x)
&&	++--+----+	(ɐ)	NG	+-+++-----+-+-	(ŋ)
O	++--+---+-	(o)	H	+----+-----+--	(h)
@	++---+----	(æ)	KW	+-+++---+-----	(kʷ)
%	++--++--+-	(ɛ)	GW	+-+++---+-+---	(gʷ)
Y	+--+------	(ɣ)	XW	+-+++---+--+--	(xʷ)
W	+--++---+-	(w)	+	-+-	
$	+--------	(ɛ)	=	---	
?	++--++----	(ə)	#	--+	
R	+++----+--++--	(r)	[	--++	
L	+++---++--++--	(l)	]	--+++	
P	+-+---+-------	(p)	]N	--++++	
B	+-+---+---+---	(b)	]A	--+++-+	
F	+-+---+----+-+	(f)	]V	--+++--+	
V	+-+---+---++-+	(v)	]C	--+++---+	
M	+-+---+---+-+-	(m)	]P	--+++----+	
T	+-+---++------	(t)			
D	+-+---++--+---	(d)			

Table 1: Unit Feature Values

In order to generate computer instructions as necessary to manipulate the values in the binary matrix, the rules as specified above must be made compatible with the requirements of the internal logical structure of the computer. This is accomplished through a compilation process on the above rules.

A logical matrix M is delcared to have a length of 50 columns, each column having 20 elements. The Jth feature in the Ith column is referred to with the notation M(I,J). The value of each M(I,J) may be either 0 or 1, representing logical False or True (the feature value - or +) respectviely. The input string (the form to be tested) is then stored as a pattern of features in the test matrix M such that each unit occupies one column of the matrix, allowing the entry of any string of segments and boundaries up to length 50. The features for each unit are stored in the corresponding column of M by transferring the values from the appropriate column of the previously defined feature matrix F.

Symbols were chosen to have mnemonic value relating to the features used. These symbols are assigned values corresponding to the row in the matrix F having that feature value.

Value	Symbol	Feature Represented
1	SEG	segment
2	VOC	vocalic
3	CONS	consonantal
4	H1 HIGH	high
5	BACK	back
6	LOW	low
7	ANT	anterior
8	COR	coronal
9	ROUND	round
10	TENSE	tense
11	VOICE	voice
12	CONT	continuant
13	N1 NASAL	nasal
14	STRID	strident

Table 2: Feature Values in Matrix Rows

Several more rows of the matrix M are delcard so that they are available for specification of diacritic information about each unit. The number of such spaces is determined by the declared size of 20, the column height. Table 3 defines six additional matrix rows.

Value	Symbol	Diacritic Represented
15	FLAG 20	Rule 20
16	FLAG 30	Rule 30
17	FLAG 32	Rule 32
18	FLAG 34	Rule 34
19	DMSR	D (see Main Stress Rule)
20	FVSR	F (see Vowel Shift Rule)

Table 3: Diacritic Feature Value in Matrix Rows

An additional row associated with M is delcared to have length 50 and elements which may have any integer value (up to the computer word size). This row has the symbol S and is used to store the stress value assigned to each unit. The stress on the Ith unit is referenced by the notation S(I). The value 0 is initially stored in the array S for all units entered, which represents [-stress] for all units. This is very convenient from the point of view of the programming language as an integer value 0 is also logical 0 while any integer value greater than zero is read as logical 1. It is planned at a later date to be able to enter a non-zero stress value for any unit in the input string.

A way was needed to store the information in the matrix representing boundary units and the syntactic bracketing of the input string. Because the previously described distinctive feature set has the common feature [+segment] it is clear that the positions in a column of the matrix representing this feature set need only be so defined when the first position has the value [+segment]. When the first element in a column has the value [-segment] the next 13 spaces are in effect free to be defined so as to represent the boundary unit information. Thus a duplicate set of values are defined on the matrix as in Table 4.

Value	Symbol	Feature Represented
1	SEG	segment
2	FB	formative boundary (FB)
3	WB	word boundary (WB)
4	BRAC	bracket
5	RBRAC	right bracket
6	NBRAC	noun bracket
7	ABRAC	adjective bracket
8	VBRAC	verb bracket
9	SBRAC	stem bracket
10	PBRAC	prefix bracket

Table 4: Boundary Unit Feature Values in Matrix Rows

Positions 4 through 10 representing bracketing information are defined only in case the feature set $\begin{bmatrix} +seg \\ -FB \\ +WB \end{bmatrix}$ occurs in positions 1 through 3. Presence of bracketing (m(I,BRAC)=1) then implies occurrence of the word boundary #. At present, 4 spaces remain in the matrix column for addition of syntactic markers other than those defined here.

When entry of the segmental form to be tested is complete and before the test cycle is begun, the matrix positions corresponding to the diacritics Rule n (FLAG 20 through FLAG 34) are set to 1.

At a point within the test sequence when the adjustment rules have been applied, the test string is scanned and the bracketing is located. A pair of pointers, LEFT and RIGHT, are set to the left and right innermost brackets. If no brackets are specified in the input form, brackets are added

to the left and right ends of the form as referents for these pointers. Corresponding to the cyclic order of application of the phonological rules, all rules begin with environmental serarch at LEFT and continue right to RIGHT. This is accomplished in the programming language with a DO-END statement pair as follows:

```
        DO  I= LEFT TO RIGHT;
      . . .
      . This block of coding is executed repeatedly
      . with I= LEFT, LEFT +1, LEFT +2, ....RIGHT;
      .
      . . .
        END;
```

Any reference to I within the DO-loop range uses the current value of the variable I for that repetition of the loop.

Because several of the rules needed for the phonetic specifications in a language require insertion or deletion of phonological units in the string, it is desirable to be able to print out the results of the application of any rule after that rule has been applied to the string. This ability has been provided in the present program with the characteristic that the results may be optionally printed after the application of any rule in the test sequence.

Rule Coding

We may now consider the coding for one of the rules to be programmed. In rule 32, Glide Vocalization, we have the specification;

$$\begin{bmatrix} -\text{cons} \\ +\text{back} \end{bmatrix} \rightarrow [+\text{voc}] \;\; / \;\; \begin{bmatrix} \alpha\ \text{round} \\ \alpha\ \text{high} \\ \text{V} \end{bmatrix} \;\; \text{———}$$

First, the Ith unit must be checked to see if it has the feature [+segment]. If not, the scan is continued to the next unit. If it does, the occurrence of the features $\begin{bmatrix} -\text{cons} \\ +\text{back} \end{bmatrix}$ is then checked. If this fails, we also continue the scan. This is represented by the following coding.

```
        DO  I  = LEFT TO RIGHT
        IF  7 M (I,S9) then go to end 32
        IF M (I,C1) |7 M (I,B1) then go to end 32
        ...
end 32: end;
```

(PL/1 uses the symbol 7 for "NOT" and 1 for "OR")

$\begin{bmatrix} \alpha\ \text{round} \\ \alpha\ \text{high} \\ \text{V} \end{bmatrix}$ The next step is to determine the occurrence of the environment in the preceding segment. Chomsky and Halle (1968 discuss the con-

vention that any rule be interpreted as applicable in the presence of the formative boundary +, which has the features $\begin{bmatrix} - \text{ seg} \\ + \text{ FB} \\ - \text{ WB} \end{bmatrix}$ in any case in whih the rule is other wise appliable. That is, no rule should be blocked by the presence of + in any context where that + is unmarked in the environmental conditions. On the other hand, if the + is marked in the environmental specificatin it must be present in the string before that rule is applicable.

From the preceding discussion we see that the environment for this rule must be interpreted as

$$\begin{bmatrix} \alpha \text{ round} \\ \alpha \text{ high} \\ V \end{bmatrix} \qquad (+) \qquad \text{————}$$

To reference any unit a fixed distance to the left or right of the currently scanned unit I, it would be possible to add or subtract a constant to the column pointer I. That is, M(I-1,J) would reference the unit immediately to the left of I. In this case, however, the unit in question may be either 1 or 2 spaces to the left of I, depending on whether the unit at I-1 has the features $\begin{bmatrix} - \text{ seg} \\ + \text{ FB} \\ - \text{ WB} \end{bmatrix}$. Actually, it is necessary to check only the first 2 features $\begin{bmatrix} - \text{ seg} \\ + \text{ FB} \end{bmatrix}$, as the set $\begin{bmatrix} - \text{ seg} \\ + \text{ FB} \\ + \text{ WB} \end{bmatrix}$ is not defined in the unit vocabulary and may be assumed not to occur. A set of pointers is available to indicate the distance of the desired unit from the currently scanned unit, L1 through L9 for distance to the left and R1 through R9 for distance to the right. These pointers, when used in a rule, are initially set to 1 at the beginning of the environmental search in each matrix position. With this convention, I-L1 initially refers to the unit immediately to the left of the Ith unit. If the unit I-1 is found to have the features of the formative boundary + then L1 is set equal to 2. I-L now refers correctly to the segment to be checked for the environmental condition specified.

```
          DO I = LEFT TO RIGHT
          L1 = 1;
          IF ¬M(I,SEG) the go to end 32;
          IF  M(I,CONS ¬M(I,BACK) then end 32;
          IF  M(I -1,SEG) and M(I-1,FB) the L1=2;
           .
           .
           .
END32:     END;
```

If M(I-L1,ROUND)¬=M(I-L1,HIGH) the go to end32;

[V] is defined to be the coincidence of the features [+ vocalic, - consonantal] which may be checked simply in one statement, while application of this rule specifies the value assignment M(I,voc)=1. Following application of the rule the printout option flags are checked and if either is set the corresponding print subroutine is executed. The coding for the rule may now be completed.

	Description
DO I=LEFT TO RIGHT;	Units to be scanned start at left-most unit, I=LEFT and include successive units I=LEFT+1, I=LEFT+2, to right-most unit I=RIGHT
L1 = 1;	Set the pointer equal to 1, at unit to immediate left of I.
IF¬M(I,SEG) the go to end32;	If the currently scanned unit, I, is specified as [-segment] to to next I (i.e. Ln+1).
IF M(I,CONS) 1¬M(I,BACK) then go to end32;	If scanned unit, I, is either [+consonantal] or [-back] (i.e. does not match the rule condition), go to the next unit.
IF¬M(I-1,SEG) and M(I-1,FB) the L1=2;	If the unit immediately to the left of I is specified as [-segment] and [+FB], then set the pointer to 2 (i.e. I-L1 will refer to two units to the left of I).
IF¬M(I-L1,SEG) the end32;	If I-L1 is a [-segment] go to next unit.
IF M(I-L1,ROUND) ¬=M(I-L1,HIGH) then go to end32;	If the unit in the left environment does not have the same feature values for roundness and highness (i.e. does not meet the rule condition, round, high), go to the next unit.

```
IF ¬M(I-L1,VOC ⎮M(I-L1,CONS)
then go to end32;
```

If the unit in the left-most environment is either [-vocalic] or [+consonantal] (i.e. not a true vowel), go to the next unit.

```
M(I,VOC)=1;
```

All the conditions have been satisfied; change the value of the feature [vocalic] from - to + (i.e. apply Rule 32).

```
PUT LIST ('RULE 32, At',I);
PRINT 'R32,AT" :I
```

Instruction to print the rule number (R32) and state the matrix feature column to which it has been applied, i.e. I.

```
IF P(32,1) THEN CALL STROUT;
```

If a display of the string, resulting from application of Rule 32 is desired, go to subroutine STROUT.

```
IF P(32,2) THEN CALL MATOUT;
```

If a display of the matrix resulting from application of Rule 32 is desired, go to subroutine MATOUT.

```
END32:  END;
```

Scan unit Ln+1, where Ln = previously scanned I.

Compiler Code Generation

To illustrate, the output coding to evaluate a simple right-handed context of the form

A -⟩B / -- context

will be examined. It will be seen that this coding can be generalized to evaluate a left-handed context as well. If the context matching process is considered to be anchored at the point between the LHS position marker and the context body, then conjunctive and disjunctive items farther to the right in the context may be tried without rematching items to their left in the context string. This would be true even after the rule has been applied to the currently matched unit, provided that the matched unit is again tested against the LHS after application of the RHS to that unit and before the context match continues.

The run-time environment in the object machine requires a single push-down stack and a few simple variable storage locations. A test string is assumed to be stored in the object machine which may have been entered prior to execution of the rule match or may be the result of application of a prior rule in the syste.

The semantic for matching particular units in the test string will not be described, but will be abbreviated in the output coding as

IF MATCH UNIT__ ; ELSE GO TO __;

which is taken to mean that a jump to the ELSE GO TO label occurs if the specified unit was not successfully matched. Further abbreviations in the output coding are in the application of the RHS of a rule, indicated by

DO RULE;

and in the declaration of program block and procedure structures. Otherwise, the coding presented constitutes a valid PL/I program segment.

Examining the coding necessary to evaluate simple contextual expressions including an un-nested disjunction, it may be seen that no looping back to previously matched units is necessary. When the left parenthesis is encountered the current location of the match pointer, stored in the variable P, is saved. If any subsequent item match fails before the right parenthesis is encountered the pointer location is reset to the saved value and the matching process resumes with the next unit outside the parenthesis. The saved pointer location is erased when the right parenthesis is encountered whether or not the disjunctive item was successfully matched. This scheme achieves the desired disjunctivity quite simply in that only one match is attempted. If the match of units inside the disjunction succeeds, the matching process continues normally. If it fails, the enclosed string is effectively ignored and the matching process continues as before. This process may be made recursive to any level by saving the pointer location in a push-down stack, freeing the top stack item when a

right parenthesis is encountered. Such a stack may easily be implemented in PL/I by using the CONTROLLED form of dynamic storage allocation for a variable STK. A new level in the stack is secured with the statement ALLOCATE STK;, saving all previous values. The top level is erased with the statement FREE STK;, bringing the previous value into accessibility.

In the coding examples presented below, two variables, LEFT and RIGHT, are assumed to contain the currently applicable left and right limits for matching the test string. These will be set by scanning the test string to locate the innermost syntactic brackets or other such test string delimiters. The index variable N will be used to indicate the left-most end of the matching process; in this case, the anchoring point following the LHS position marker. The statement MATCH UNIT __; ELSE GO TO __; is assumed to increment the current match pointer P and fail at any time the value of P exceeds the right delimiter value, stored in RIGHT.

The coding to evaluate a context of the form
RULE N: A ->B / -- C D (E (F G) H J) K
would have the following appearance.

```
RULE:   DO I=LEFT TO RIGHT;
        P=I;
        IF MATCH UNIT A; ELSE GO TO NEXT;
        IF MATCH UNIT C; ELSE GO TO NEXT;
        IF MATCH UNIT D; ELSE GO TO NEXT;
        ALLOCATE STK;
        STK=P;
        IF MATCH UNIT E; ELSE GO TO PN1;
        ALLOCATE STK;
        STK=;
        IF MATCH UNIT F; ELSE GO TO PN2;
        IF MATCH UNIT G; ELSE GO TO PN2;
        GO TO SK2;

PN2:    P=STK;
SK2:    FREE STK;
        IF MATCH UNIT H; ELSE GO TO PN1;
        IF MATCH UNIT I; ELSE GO TO PN1;
        GO TO SK1;

PN1:    P=STK;
SK1:    FREE STK;
        IF MATCH UNIT J; ELSE GO TO NEXT;
        DO RULE;
NEXT:   END RULEN;
```

The attempt to formulate the coding to evaluate a context including a conjunctive series brings to light a different type of problem. It is not possible to match units from left to right in an orderly fashion as for simple or disjunctive contexts. Once a match for the entire string has been attempted using the first item of the conjunctive series, it is necessary,

whether the rule was applied or not, to reset the current match pointer to its value at the time the left bracket was encountered, and then continue the matching process using the units of the second conjunctive item as the matching patterns. In order to loop back in this manner, it is necessary to save three values during a matching pass over the string; 1) The bracket-pair number, 2) The pointer value at the time the left bracket is encountered, and 3) The item number within the bracket pair. These three values are saved in the push-down stack in the order listed when the conjunctive series match is begun. It is convenient in the PL/I language to accomplish the branching by using the stacked values as subscript values in an assigned-label GO TO statement. The labels ITEM(1,1):, ITEM(1,2):, ITEM(1,3):,.... are attached to the statements in the coding which perform the pointer reset following matching of the corresponding conjunctive items. Branching is accomplished with the statement GO TO ITEM(I,J); following the proper assignment of values to the variables I and J.

An initial value of zero is put in the stack prior to rule evaluation. The stack is then checked for a non-zero top item before it is unstacked for label assignment and an empty stack indicates that all conjunctive items in the rule have been used in the matching process. If the stack is not empty, the top two items are unstacked and stored as the variables J and P respectively. The remaining top stack item is accessed and the value stored in the variable I, but it is not freed from the stack. The value of P must then be restored to the stack so it will be handled properly by the end- of-item coding. The details of this scheme may be seen in the following example, coded to evaluate a context of the form

RULE J: A →B / -- C [DE / FGH / I] J

```
            DECLARE ITEM(1,3) LABEL;
            ALLOCATE STK;
            STK=0;
RULE J:     DO N=LEFT TO RIGHT;
            P=N;
            IF MATCH UNIT A; ELSE GO TO NEXT;
            IF MATCH UNIT C; ELSE GO TO NEXT;
            ALLOCATE STK;
            STK=1;
            ALLOCATE STK;
            STK=P;
            ALLOCATE STK;
            STK=1;
            IF MATCH UNIT D; ELSE TO TO B11;
            IF MATCH UNIT E; ELSE GO TO B11;
            GO TO BR1;
B11:        J=STK;
            FREE STK;
            P=STK;
ITEM(1,1):  ALLOCATE STK;
            STK=J + 1;
            IF MATCH UNIT F; ELSE GO TO B12;
            IF MATCH UNIT G; ELSE GO TO B12;
```

```
        IF MATCH UNIT H;ELSE GO TO B12;
        GO TO BR1;
B12:    J=STK;
        FREE STK;
        P=STK;
ITEM(1,2):  ALLOCATE STK;
        STK=J + 1;
        IF MATCH UNIT I; ELSE GO TO B13;
        GO TO BR1;
B13:    FREE STK;
ITEM(1,3):  FREE STK;
        FREE STK;
        GO TO NEXT;
BR1:    IF MATCH UNIT J; ELSE GO TO NEXT;
        DO RULE;
NEXT:   IF STK=OTHEN GO TO SCAN;
        J=STK;
        FREE STK;
        P=STK;
        FREE STK;
        I=STK;
        ALLOCATE STK;
        STK=P;
        GO TO ITEM(I,J);
SCAN:   END RULEJ;
```

A further complication arises when a conjunctive series is embedded inside of a disjunction. Specifically, the pointer location should not be reset to the value stored in the stack for any failure to match the internal sequence of units, but only if the match fails for all items in the embedded conjunctive series. Because the last conjunctive item may fail to match, while a previously tested item matched successfully, it is necessary to use a "rule applied" flag, which is cleared (reset) when entering the match of a disjunction and set by any application of the rule. The setting of this flag determines the action taken concerning the pointer setting on exit from the disjunction, when all conjunctive items have been tried.

The Compiling Process

It may be seen from the coding examples given that the output from the compiler occurs essentially in the same order as the symbols in the linear input form, suggesting that a preliminary stage of syntactic analysis is unnecessary. It is only necessary to save the RHS specification in the compiler from the time it is input until it is output in coded form at the end of the context coding. Observing the three different types of failure-to-match exit branches, it appears that the most direct solution is a three-state table driven translator used in conjunction with a number of indices defined during the compiling operation for the purpose of counting brackets and parenthesis, generating sequential labels, etc. Entries in the table indicate for each of the three states what output coding should be generated and what compiler index operations should be carried out as a result of each possible input symbol.

The table and listing of compiler actions shown below specifies a compiler system capable of producing PL/I coding such as shown in the examples. Notations used in the compiler table and action specifications

are explained briefly.

1. Upper-case letters in the output are output as shown.
2. Lower-case letters in the output represent compiler variables for which the currently assigned value is output.
3. Abbreviated output coding has the meaning discussed above, for example, DO RULE expressed the coding necessary to incorporate into the marked unit in the test string the characteristics or features given as the RHS of the rule.
4. The state transfer from state 2 on input of a right parenthesis is a conditional transfer, depending on the value of the compiler variable m. The test is shown as a fourth pseudo-state.
5. Compiler initialization, shown as state 0, must be accomplished at the beginning of compilation for each rule.
6. Three of the input actions are identical for all states, indicating that it is unnecessary to store those actions in the state table.
7. No action is specified for error inputs. It is assumed that the compiler would respond with some indication of the trouble, for example, a comma input when in state 2 could cause the reply "Comma illegal inside parenthesis".

The compiler uses seven variables, four of them, i,j,k and l, as push-down stacks with the CONTROLLED attribute, and three, m,n and o as simple variables.

Compiler State Table

State	Inputs						
	Unit	[	,	]	(	)	END OF LINE
0. Action:	Allocate l; l=1; n=0; o=0;						
Next State:	1						
1. Action:	1	4	5	6	7	9	10
Next State:	1	3	3	1	2	1	0
2. Action:	2	4	error	error	7	8	10
Next State:	2	3			2	4	0
3. Action:	3	4	5	6	7	error	10
Next State:	3	3	3	1	2		0
4. Action:	Conditional transfer state, no input;						
Next State:	If m=0 then go to state 1., else go to state 2.						

Compiler Actions

Action number	Compiler Operations	Output Code
1.	Output	" IF MATCH UNIT __; ELSE GO TO NEXT;"
2.	Output	" IF MATCH UNIT __; ELSE GO TO PNk;"
3.	Output	" IF MATCH UNIT __; ELSE GO TO Bij;"
4.	n=n+1	
	Allocate i;	
	i=n;	
	Allocate j;	
	j=i;	
	l=l+1;	
	m=0;	
	Output	" ALLOCATE STK; STK=i;"
	Output	ALLOCATE STK; STK=P;"
	Output	ALLOCATE STK; STK=1;"
5.	Output	" GO TO BRi;"
	Output	'Bij: J=STK; FREE STK; P=STK;"
	Output	'ITEM(i,j): ALLOCATE STK; STK=J+1;"
	j=j+1;	
6.	l=l-1;	
	Output	" GO TO BRi;"
	Output	'Bij: FREE STK;
	Output	'ITEM(i,j): FREE STK; FREE STK;"
	If l>0 then go to 6a.	
	Output	" IF FLAG=0 THEN GO TO PNk;"
	Output	" FREE STK;"
6a.	Output	" GO TO NEXT;"
	Output	'BRi:"
	FREE i;	
	FREE j;	
7.	o=o+1;	
	Allocate k;	
	k=o;	
	Allocate l;	
	l=0;	
	m=m+1;	
	Output	" ALLOCATE STK; STK=0;"
	Output	" FLAG=0;"
8.	Free l;	
	m=m-1;	
	Output	" GO TO SKk;"
	Output	'PNk: P=STK;"
	Output	'SKk: FREE STK;"
	Free k;	

```
9.      Free 1;
        Output          "           GO TO SKk;"
        Output          'PNk:       P=STK; FREE STK;"
        Output          'SKk:"
        Free k;
10.     Free 1;
        Output          "           DO RULE;FLAG=1;"
        Output          'NEXT:      IF STK=0 THEN GO TO SCAN;"
        Output          "           J=STK;FREE STK;"
        Output          "           P=STK;FREE STK;"
        Output          "           I=STK;ALLOCATE STK;STK=P;"
        Output          "           GO TO ITEM(I,J);"
        Output          'SCAN:      END RULE;"
```

The Generality of the Process

The only references to left-right directionality in the matching scheme described are in the left to right scan of the current LHS marker in attempting to fit the test string and in the assumption that the coding for matching particular units included an instruction to increment the matching location pointer,P. A left-handed context may be evaluated by similar coding by letting the pattern match move from the LHS outward, i.e., to the left. The same compiling system can be used by reversing the symbols of the left-hand context during the initial linearization, substituting left for right and right for left brackets and parenthesis. Thus, a rule of the form

A ->B / EF(G)H --I $\begin{bmatrix} J \\ K \end{bmatrix}$

would appear in the linear format as

A ->B / H(G)FE -- I[J,K]+

An additional dimension would be added to the compiler state table, providing for the production of unit match coding which would decrement instead of incrementing the current matching pointer. The LHS marker would still scan the test string from left to right. If the LHS marker occurred within the items of a conjunction, separate coding would have to be produced for the left and right parts of each item. The details of the matching process for this case have not been worked out, but do not appear to present any major difficulties for the system presented here.

Bibliography

1. Blair, F., Programming of the Grammar Tester in Specification and Utilization of a Transformational Grammar, Sci. Rep. 1, IBM Corp. Yorktown Hts. New York, 1966

2. Friedman, Joyce, A Computer System for Transformational Grammar. Computer Sci. Rep. CS-84 AF-21, Stanford, Ca., Jan. 1968

3. Gross, L.N., On-Line Programming System User's Manual MTP-59, The MITRE Corp., Bedford, Mass., March 1967

4. Londe, D. and Schoene, W., TGT, Transformational Grammar Tester, TM-3759/000/00, System Development Corpo. Santa Monica, Ca., 1967

5. Chomsky, Noam, The Sound Pattern of English, Harper and Row, N.Y., 1968

6. Kuroda, S.-Y., Yawelmani Phonology, MIT Press, Cambridge, 1967

7. Schachter, Paul and Fromkin, Victoria, A Phonology of Akan: Akuapem, Asante and Fante, Working Papers in Phonetics No. 9, UCLA, August 1968

8. Bobrow, D.G. and Fraser, Brice, The Phonological Rule Tester, Comm. ACM, vol 11, no 11, November 1968

9. Rice, D. Lloyd, and Hofshi, Reuben, An Interactive Phonological Rule Tester, Working Papers in Phonetics, No. 10, UCLA, Dec. 1968

10. Chomsky, Noam. Some General Properties of Phonological Rules, Language vol 43 no 1, March 1967

11. Kimball, J.P., Conjunctive Stacks and Disjunctive Sequences in Language Change, Quarterly Prog. Rep. Research Lab. of Electronics, No. 88, Mass. Inst. of Technology, Jan. 1968