A PHONOLOGICAL RULE COMPILER[*]

Jeffrey A. Barnett[†]

System Development Corporation
2500 Colorado Avenue, Santa Monica, California
University of California at Irvine
Department of Information and Computer Science

This paper reports on a language and system for describing and operating a set of phonological transformation rules. The rules are written in a fairly general rewrite language which is compiled into machine code for execution on an IBM 370 computer. The rule compiler and run-time package have been implemented within the SDC LISP system. A set of 37 rules have been compiled into a module that is integrated with the lexical matching procedure used in a continuous speech understanding system.[1,2].

The general goal of the rule system is to generate a set of alternative pronunciations for lexicon entries. Therefore, the transformations are from one phonemic string into another. A rule set is usually specific to a single user or a small group with similar speaking characteristics; the effect of this and other conditions on the number of necessary rules will be examined. Further, the system is not specifically designed as a rule tester but rather as a rule implementation tool. This makes considerations of operating efficiency extremely important.

The issues of ordering and backtracking are examined in the context of this system and its operational objectives. The conclusion reached is that rule ordering in this system is inappropriate and that automatic backtracking is unnecessary because the user's knowledge of phonology provides better methods of handling the problem than could be generated by the computer from abstract examination of a rule.

[*]Originally appeared in the *Proceedings of the IEEE symposium on Speech Recognition*, April 15–19, 1974, 188–192. IEEE Catalogue No. 74CH0878–9 AE.

[†]Email: jbb@notatt.com

The next section gives a general overview of the system. Other sections discuss the rule language, rule ordering and backtracking, and the implementation. The concluding section presents some performance statistics.

General Description

In the SDC speech understanding system, the lexical matching component is given a proposed lexical entry and boundary conditions. The boundary conditions may include beginning and ending times and immediate neighbor words on the left and right, if known. Alternative pronunciations are mapped against a parametric representation of the acoustic data, and the best match is scored and returned, assuming that the score is acceptable. The mapping is done syllable by syllable.

The phonological rules used by the system are roughly grouped into three sets. The first set of rules generate the set of legal pronunciations that may be realized by changes in the phonemic spelling. The second rule set deals with intra-syllable co-articulation effects that may depend upon phonetic features. The third rule set works with interactions over syllable and word boundaries and utterance initial and final effects. The phonological rule compiler produces the module implementing the first set of rules.

Obviously, the above characterizations of membership in the three rule sets is not sharp. That is, a rule could possibly belong to more than one set. The assignment of a rule to a particular set when there is an alternative is governed mostly by considerations of run-time efficiency. That is, the rule is placed in the system at the point where the least amount of additional computation will be introduced.

The entries in the lexicon are words and a few special items. Spellings are phonemic, with syllable and interior word boundaries and vowel stress also shown. Three levels of stress are used: 0 means reduced, 1 means unstressed, and 2 means stressed. The representation of the phonemes is in a coding agreed upon by a group of ARPA contractors for input and output by computers. Besides the normally recognized phonemes of English, the system includes symbols for flap t, glottal stop, er (as in bird), vowel stress, and syllable and word boundaries. In this paper, references to phonemes include all of the above.

When the mapping procedure is requested to verify an hypothesized lexicon entry against the acoustic data, the spelling is retrieved from the lexicon, and the rules in the first set are operated. This produces the list of alternative pronunciations. This list is given to the actual matching modules for verification. The lexicon spelling and all alternatives produced by the first rule set are assumed to be legitimate pronunciations. This has two major effects. First, it precludes using a morphological lexicon, because a concatenation of morphs is in general not pronounceable and may require the application of many rules to become so. Second, the lexicon and first rule set may need modification for different speakers. This is not a particular defect of this system. Any systematic modeling of a speaker's performance will need some custom tailoring for that particular speaker. Furthermore, it is believed that most of the changes necessary to accommodate new speakers will take place inside the first rule set and the lexicon. In our present system, these are the places most easily modified.

The rule set in the present system contains 37 rules. Several things keep the size of the rule set used in the system relatively small: 1. A major consideration is that a given lexicon and rule set need serve only a single user or a small community with similar speaking habits. 2. The output of a reconstruction must map back into another phonemic string. 3. The original input string does not carry any features denoting the surface structure so no rules dealing with these phenomena are included at this level of the system. 4. The limited feature set used in the system leads to fairly general schemata, thereby causing a reduction in the number of necessary rules.

The Rule Language

A phonological rule has four parts: name, pattern, reconstruction, and conditional. A rule's name maybe any identifier chosen by the user. A pattern consists of a nucleus and a left and right context. The pattern defines a schema of phonemic sequences. When such a sequence is found in a spelling, the part matching the nucleus is replaced by the sequence generated by the reconstruction part of the rule. The conditional is an optional part of a rule. When present, the conditional specifies certain necessary interrelationships between members of the input sequence matched by the patterns (This specification task has been accomplished by using bound variables in some rewrite schemes.[3]) Thus the necessary and sufficient criterion that a rule apply to an input sequence is that the pattern match part of the sequence and that the conditional, if present, be satisfied.

The pattern, reconstruction, and conditional portions of a rule are built up from phoneme and feature names. In addition, the reconstruction and conditional may reference phonemes and their features in the sequence matched by the pattern. Each phoneme is categorized by some combinations of the values of six features (or properties). The features and their possible values are:

KIND: `VOWEL`, `CONST` (consonant), `BOUNDARY` major categories

CLASS: `NASAL`, `GLIDE`, `LATERAL`, `CENTRAL`, `FRIC` (ative), `PLOS` (ive), `AFRIC` (affricate), `MISC` (ellaneous) consonantal categories

PLACE: `LABIAL`, `DENTAL`, `ALVEOLAR`, `ALVPAL` (alveolar-palatal), `PALATAL`, `VELAR` consonantal places of articulation

VOICE: `+VOICE`, `-VOICE` voiced-unvoiced indicator

STRESS: 1, 2, 3 vowel stress

NAME: ARPA contractor's set

In the examples that follow, the phonemic names used will be: `*` – syllable boundary, `#` – word boundary, `DX` – flap t, `R` – rant, `N` – net, `M` – met, `NX` – sing, `AX` – schwa, `IH` – bit, `IY` – beat, `TH` – thing, `T` – ten, `D` – debt, `P` – pet.

Language features will be introduced by example. A rule will be given in the phonological rule language. Then the name, reconstruction, pattern and its components, and conditional will be listed separately, followed by a prose description.

```
$ DEGEMINATION                                                    (1)
NIL = /CONST/ OPT BOUNDARY, CONST IF NAME@1 EQ NAME@3;
name              DEGENINATION
reconstruction    NIL
pattern           /CONST/ OPT BOUNDARY, CONST
   nucleus        CONST
   left context   vacuous
   right context  OPT BOUNDARY, CONST
conditional       IF NAME@1 EQ NAME@3
```

(1) states that the first member of a pair of doubled consonants may be deleted. The rule also applies if the consonant pair is separated by a syllable or word boundary. The dollar sign and semicolon delimit the definition. `DEGEMINATION` is the rule name, and `NIL` is the void reconstruction sequence

denoting deletion. Within the pattern, the portion surrounded by slashes is the nucleus. There is no left context, and the right context is the optional syllable or word boundary followed by a consonant. In a pattern, `OPT` introduces the specification of a phoneme whose presence is optional, that is, it is allowed but not necessary in the input string.

The conditional states that the rule may only apply if the first and third items specified by the pattern have the same name. In the enumeration of parts of a pattern, an optional phrase counts as one item whether or not the specified phoneme is present in the matched spelling.

A simple rule for generating flaps might be

```
$ FLAP DX = VOWEL, OPT R, * /T OR D/ VOWEL;                    (2)
```

name	`FLAP`
reconstruction	`DX`
pattern	`VOWEL, OPT R, * /T OR D/ VOWEL`
nucleus	`T OR D`
left context	`VOWEL, OPT R, *`
right context	`VOWEL`
conditional	not used

(2) states that a t or d may be replaced by a flap t when followed by a vowel and preceded by a syllable ending in a vowel or a vowel followed by an `R`. In a pattern, the `OR` connective may be used to specify alternative matching criteria. If the phoneme in the input string fulfills either criterion, it matches the `OR` phrase. This is an example of a simple replacement rule.

An example of an insertion rule is

```
$ HOMORGANIC.STOP.INSERTION                                              (3)
(PLOS PLACE@1 -VOICE) = NASAL, OPT * // (PLOS-VOICE) OR (FRIC-VOICE)
   IF CLASS@3 EQ FRIC OR PLACE@1 NQ PLACE@3;
```

name	`HOMORGANIC.STOP.INSERTION`
reconstruction	`(PLOS PLACE@1 -VOICE)`
pattern	`NASAL, OPT * // (PLOS-VOICE) OR (FRIC-VOICE)`
nucleus	vacuous
left context	`NASAL, OPT *`
right context	`(PLOS-VOICE) OR (FRIC-VOICE)`
conditional	`IF CLASS@3 EQ FRIC OR PLACE@1 NQ PLACE@3`

(3) illustrates several interesting features of the phonological rule language. The reconstruction specifies the unvoiced plosive, which is homor-

ganic with the nasal (i.e., the place of articulation is "borrowed" from the nasal), the first item in the pattern. The right context specifies either an unvoiced plosive or an unvoiced fricative. This is an example of a language capability to specify "feature bundles" for parts of the pattern. Thus, (3) states that an appropriate plosive may be inserted after a nasal (or a nasal and a syllable boundary) and before an unvoiced fricative or before an unvoiced plosive that is heterorganic with the preceding nasal. In the word "something" a `p` would be inserted between the `m` and the `th`.

```
$ ING.REDUCTION                                              (4)
IH:0, N OR AX, N = *, REP 0 CONST /IY:0 OR IH:0, NX/ #;
name             ING.REDUCTION
reconstruction   IH:0, N OR AX, N
pattern          *, REP 0 CONST /IY:0 OR IH:0, NX/ #
  nucleus        IY:0 OR IH:0, NX
  left context   *, REP 0 CONST
  right context  #
conditional      not used
```

For polysyllabic words ending in "ng," (4) abbreviates the pronunciation. For example, the word tracking, would be modified to trackin' or trackan'. The vowel name followed by a colon specifies the level of stress. `IY:0` means a reduced `IY`. To specify the occurrence of a sequences of phonemes satisfying the same description, a repetition phrase may be used. A repetition phrase begins with the word `REP` followed by the minimum length in phonemes of the sequence to be matched. Thus, "`REP 0 CONST`" matches any sequence of zero or more consonants. In a reconstruction, `OR` separates alternative substitution sequences. Therefore, (4) behaves as if it were two separate rules with the same pattern (and conditional if one had been used). One of the rules would have the reconstruction sequence `IH:0, N` and the other would have the reconstruction sequence `AX, N`.

Rule Ordering and Backtracking

Rule Ordering

In constructing a system to operate transformation rules, the implementer must decide upon an ordering strategy for applying the rules to an input string. In [3], Chomsky and Halle describe a particular scheme for use with a large set of mandatory rules. This scheme has been approximated in a rule testing system reported on by Bobrow and Fraser.[4] The input string to a Chomsky-Halle system is a sequence of segments, normally derived from a morphological base, which is "parenthesized" to show surface structure. The set of rules is applied in a fixed order to the most deeply nested sequence, then the inner parentheses are dropped. This process is repeated until all parenthetic structure has been removed.

When a rule is applied to the input sequence (at a particular parenthesis level), it is matched to all places in the sequence to which it applies before any reconstruction action is taken. All reconstructions for this rule are then made simultaneously at the present level. Following this, the next rule in order is applied to the input sequence at the present working level. After all rules have cycled in this manner, the inner parentheses are removed, etc.

Several differences between the intention of the Chomsky-Halle system and the system described in this paper should be noted. In the former, the original string contains surface structure bracketing, and the original spelling is made from a concatenation of morphemes. Also, the rules are expressed in terms of a rich feature set that can construct an element in the output string that may be represented only in terms of the features. Further, the output is a single string that is an idealized pronunciation. In the latter, the original input string does not contain bracketing and is itself assumed to be a pronounceable, syllabic spelling. The rules are expressed in terms of a simple feature set, and the derived results may be representable as a phonemic string needing no features, only phoneme name and vowel stress.

These considerations lead to the selection of different ordering strategies for applying rules. First, it is clear that the derivation of a pronounceable result from a non-pronounceable input requires the application of some mandatory rules. Just as clearly, the generation of the set of alternative pronunciations from a base pronunciation will involve optional rules. The necessity for ordering the kind of rules in the Chamsky-Halle system is well defined in [3]. However, the same arguments do not apply here. The size of the alphabet that codes the output string is much smaller; therefore the size of the input alphabet to the "next" rule is also smaller. This means that the complexity of the rule interaction should be sufficiently reduced as to not need rigid ordering to avoid exotic surprises. Also, if each rule transforms a legal pronunciation into another legal pronunciation then $\text{rule}_1(\text{rule}_2(s))$ and $\text{rule}_2(\text{rule}_1(s))$ may produce different output, and both must therefore be generated. The objection may be raised that if $s = \text{rule}_1(\text{rule}_2(s))$ then an unordered evaluation of the rule set could lead to an infinite application sequence; two principal things keep this form happening. First, the lexicon entries should follow some set of conventions. Second, direct inverse rules need not be coded. For instance, a lexicon convention might be that flap t is not used if a t or a d would also be legitimate in the pronunciation. Then, only the rules generating the flap from t and d would be included, not their inverses. This obtains both pronunciations when appropriate without infinite looping.

Because of the above considerations, a decision was made to apply the rules in an essentially unordered manner. Application of all the rules is essentially continuous in all orders and sequences until the full set of alternative pronunciations derivable by the rule set has been obtained.

Backtracking

In addition to considerations of rule ordering, the system implementer must adopt a strategy for handling backtracking. In the described rule language, the need for backtracking can occur only through the use of optional (`OPT`) and repetition (`REP`) phrases. The nucleus of the pattern

```
* /CONST, OPT CONST, R/ VOWEL                    (5)
```

would appear to match any syllable initial consonant cluster ending in an `r`. However, without backtracking, the thr in "three" would not be properly matched. `CONST` would match th and `OPT CONST` would match the r, leaving nothing to match the `R` part of the pattern. To implement (5) in a non-backtracking system, one could write

```
* /CONST, OPT (CONST-R), R/ VOWEL                (6)
```

The only case not handled by (6) is a three consonant cluster ending in rr. The problems become worse when there are multiple optional and repetition phrases in the same pattern. However, the desirability and payoff for implementing automatic backtracking varies most directly with the allowable complexity of specification in the pattern, and not so much with pattern length. This is because, as the complexity grows, the determination of items that fulfill one specification but not the others becomes more difficult. Also, the tendency to use partial rather than complete specifications hightens this effect in the face of added complexity. In a phonetic (principally feature-driven) transformational system, backtracking is virtually a necessity. In the described system, automatic backtracking has not been implemented. The pattern descriptions are fairly simple, and the user is usually a far better decision maker on the exact criterion to stop a repetition or accept an optional occurrence. In most cases, the criterion will be derived from the user's knowledge of phonology rather than from some abstract examination of the pattern. For example, (6) is a legitimate rewrite of (5) because the pair rr should not occur in an English syllable. Thus, the fact that the lexicon is coded in syllables imposes additional constraints that are not easily known by the system but that are useful to the rule implementer.

A problem with automatic backtracking is a possible great increase in execution time. For instance, if a pattern contains n optional and no repetitive phrases, then in the worst case, the system will have to attempt 2^n different matches of the pattern to the input string. A pattern that contains a repetitive phrase represents an infinite rule schema. But the fact that a given input string contains a fixed number of elements bounds the amount of computation. If a pattern containing n repetition and no optional phrases is matched to an input string containing p elements not matched by repetition phrases, then in the worst case, the system would have to try approximately $p^n/n!$ distributions of the input string to matches of the repetition phrases.

System Implementation

The Rule Compiler

The phonological rule compiler has been implemented within the SDC LISP system, which operates on an IBM 370 Model 145. The syntax analysis pass was integrated into the INFIX language translator using the $ as the escape token to the special rules language. The INFIX translator normally transforms an ALGOL-like language into LISP. In all, the rule compiler comprises six passes: syntax analyzer, expander, boolean optimizer, register allocator, code generator, and assembler.

The output of the compiler is one function for each reconstruction sequence specified by a rule. The output functions are independent in that they call no other functions. However, the functions do not use standard linkage conventions so that the run-time driver, `APPLY`, must make special invocations. Each of these functions serves as a partial predicate. If the input string satisfies the rule operated by a function, the value is the length of the reconstruction sequence and the location in the string at which the substitution is to be performed. If the rule is not satisfied, the function returns false. The actual string reconstruction by substitution is performed by a single function included in the run-time package. A brief description of the compiler passes appears in the paragraphs below.

Syntax Analyzer. The syntax pass is implemented as a language extension of the LISP INFIX translator. The input token string is turned into a Polish-prefix, list-structured representation. Syntax and local semantic errors are detected and reported to the user. As is normal with use of the INFIX system, errors cause immediate entrance to an editor for correction by the user. The parsing algorithm is top-down, and the rule language is so constructed that backup is never necessary over more than one token. The token parsing package (lexical analyzer) is borrowed from the INFIX system.

Expander. The expander pass completes the task (started by the syntax analyzer) of turning the input into a Polish-prefix command and test sequence. This is accomplished in two phases: explicit indexing and rippling. The `AX` in the pattern, "`*/R/AX`" would be turned into "`(EQ (NAME 3) AX)`" by the explicit indexing phase. Thus, after this phase, the internal representation of the conditional and pattern are structurally equivalent.

The rippling phase reorders and merges the various components of the reconstruction, pattern, and conditional. In the rule,

```
$ EXAMPLE AX = CONST /VOWEL/ CONST IF STRESS@2 NQ 2;
```
(7)

the nucleus would become

```
(AND (EQ (KIND 2) VOWEL) (NQ (STRESS 2) 2))
```
(8)

The basic algorithm for combining the conditional with parts of the pattern is

- If the conditional is an `AND` form, then apply this algorithm to each embedded form, otherwise go to the next step.
- Find the highest-numbered index reference to a pattern element from the form, and attach the form to that element with an `AND` operator.

The algorithm allows all tests to be performed as quickly as possible, thus allowing the rules to take a failure exit after a minimum of computation time.

The algorithm for moving elements of the reconstruction sequence into the pattern sequence is

- If the element has no indexed references, put it at the and of the pattern sequence.
- If the element has any indexed references, put it just after the pattern element referenced by the highest-numbered index reference from the reconstruction element.

The motivation for placing a reconstruction element at the end is to save time if the rule does not match the input string. The reason for placing reconstruction elements at other than the end is saving general-purpose registers that may become necessary if an optional or repetitive phrase occurred between a place referenced by an index and the place of element insertion. (See paragraph on register allocator below, for more details.)

Boolean Optimizer. The boolean optimization pass factors and simplifies `AND` and `OR` forms. For example, the expression

```
(AND (OR a b c) (OR x c y))                (9)
```

would be transformed into

```
(OR c (AND (OR a b) (OR x y)))             (10)
```

Several other transformations not illustrated by (9) and (10) are also performed by this pass.

Register Allocator. The operation of the register allocator is very straightforward except when indexed references are made across an optional or repetitive phrase. The difficulty is that the number of phonemes separating the referencer and referencee is not resolvable until execution time.

A very simple solution to this problem has been adopted. Code operating after a repeat or optional phrase assumes that the least permissible length has occurred. The indices output by the expander pass are computed under this assumption. A common base register is incremented every time a match has been made that exceeds the minimum permissible length. When cross-references are made, another register is loaded with a copy of the common base before the operation of the optional or repetition phrase. This second register will be used when the cross-references are made. Care is taken to not use more registers than necessary.

Code Generator. The code generation pass turns the output of the register allocator into symbolic machine language. The algorithms used try to optimize the output test and branch instructions. A property of the algorithms is that if two inputs to the code generator are equivalent under DeMorgan transformations, double negation, and/or transitivity, then the output code is bit-for-bit identical. The test and move instructions used for the matching and reconstruction are all memory-to-memory or immediate-to-memory. An example of code output by the rules compiler is given below.

Assembler. The assembler is the LAP assembler, which is part of the LISP system. It was not constructed or modified for use with the rule compiler.

Run-time Package. The run-time package consists of those functions necessary to operate the assembled rules as well as some additional user aids to facilitate testing.

`GENAPPLY` is a function that, when called, causes compilation of the rule loop driver function, `APPLY`. `APPLY` is implemented as a large loop containing explicit calls on each rule function and proper linkage to the string rewrite procedure.

`RECONSTRUCT` is the rewrite procedure that computes the phonemic name of an element built from features such as the reconstruction part of (3). It also rewrites the input string with the appropriate changes. The changed string is placed in a new array; the original is not damaged.

`DELRULE` is a function of one argument, a rule name. The named rule is deleted from the system. The functions `CHECKSPELL` and `MAKESPELL` are provided to error-check spellings and translate the spellings into the internal representation. The main program of the rule operating system is `RUNRULE`. Its argument is a phonemic input string. The output is the list of alternative pronunciations.

Internal Representation

For reasons of efficiency, each phoneme in a spelling sequence is internally coded as a 32-bit integer rather than as a LISP identifier. A phonemic spelling is stored as an array of these integers. The 32 bits are subdivided into four 8-bit bytes in which the features are represented. Byte 0 is used for general classification information: the phoneme's kind (boundary, consonant, or vowel), the stress level for vowels, and whether the phoneme is voiced. For consonants, bytes 1 and 2 specify the class and place of articulation, respectively. In each of these bytes only one bit may be turned on. For vowels and boundaries, bytes 1 and 2 are unused. Byte 3 is an 8-bit code for the phoneme's name. Much redundant information is present in the selected representation. However, the 370's order codes contain several instructions for bit and byte testing that can exploit this to advantage.

Example Compilation

Below is the output of the phonological rule compiler for (3). See the IBM Principles of Operation[5] for an explanation of the 370's order code.

```
FUNCTION HOMORGANIC.STOP.INSERTION
      TM   1(R1),8                   is 1st phone a nasal?
      BCR  10,R8                     no, take failure exit.
      MVC  CHANGE(4),=F'1074790400'  move (plos-voice) to change area.
      MVC  CHANGE+2(1),2(R1)         move place@1 to change area.
      LR   R2,R1                     save base before an optional
                                     phrase.
      CLI  7(R1),192                 *?
      BC   7,L101                    no.
      LA   R1,4(R1)                  yes, bump common base.
L101  LA   R3,4(R1)                  compute substitution address.
      ST   R3,BMRK                   save beginning address for
                                     substitution.
      ST   R3,EMRK                   save ending address for
                                     substitution.
      TM   4(R1),16                  is next item voiced?
      BCR  5,R8                      yes, take failure exit.
      TM   5(R1),32                  is item a fricative?
      BC   5,L102                    yes.
      CLC  6(1,R1),2(R2)             no, place@1 EQ place@3?
      BCR  8,R8                      no, take failure exit.
      TM   5(R1),16                  is it a plosive?
      BCR  10,R8                     no, take a failure exit.
L102  L    R7,=F'1'                  yes, rule matches.  specify
                                     reconstruction length of one
                                     and exit.
```

Performance Measurement

The phonological language and operating system described in this paper have been extensively used in a speech understanding system. The system's lexicon contains approximately 150 entries with an average length of 10 phonemes including syllable and word boundaries. The rule set used contains 37 rules of varying complexity. The measurements given herein were made on the 150-entry lexicon with the 37-rule set.

Compiler Timing

The average compilation time for a rule was .758 seconds. The time to operate `GENAPPLY` was 4.377 seconds. The rules were on disk and the compilation times include input.

Operation Measurements

The average number of alternative pronunciations per item in the lexicon was 2.289, including the original spelling in the lexicon. The probability that a given rule would apply at a given spot in an input string was .00321. The average rule operation time, including matching, reconstruction, and `APPLY` loop, is approximately 55 micro-seconds. The average total operating time of `RUNRULE` on a lexicon entry is .0437 seconds.

Conclusion

The phonological rule language system described in this paper has as its general goal the generation of alternative pronunciations for lexicon entries. The issues of ordering and backtracking are examined in this context, and the conclusions drawn that automatic backtracking is unnecessary and ordering the application of the rules is inappropriate. A general aim of the system has been run-time efficiency. That this has been achieved is supported by the data presented in the Performance Measurement section above. Some details of the optimizing compiler for the rules language have been presented. The lack of automatic backtracking in the system has made it possible to compile rather then interpret, as is done in all other known systems with similar objectives.

Acknowledgements

Several people have made valuable contributions to this project and I would like to take this opportunity to thank them for their help and guidance. Douglas Pintar aided in the implementation and debugging of the system. Peter Ladefoged and Rollin Weeks made contributions to the design decisions as to what constituted a necessary and sufficient system. Martin Kay helped stimulate my interest in the indeterminacy problems of rule ordering and backtracking.

This research was supported by the Advanced Research Project Agency of the Department of Defense under Contract No. DAHCI5–73–C–0080.

References

[1] “A Voice Activated Data Management System,” H. B. Ritea, this conference.

[2] “Predictive Syllable Mapping in a Continuous Speech Understanding System,” R. Weeks, this conference.

[3] “The Sound Patterns of English,” Chomsky and Halle, Chapter 8, Harper and Row, 1968.

[4] “A Phonological Rule Tester,” Daniel G. Bobrow and J. Bruce Fraser. Communications of the ACM, Vol 1l/Number ll/Nov. 1968, page 766–772.

[5] IBM Systern/370 Principles of Operation, IBM, GA22–7000–2, Third edition, 1972.