A LEXICAL SYNTHESIS APPROACH TO USER-ORIENTED INPUT SPECIFICATION

Karla Hoffman and Christoph Witzgall

National Bureau of Standards
Washington, D.C. 20234

Abstract

Modern large-scale application programs often call for flexible natural-language type input capabilities. Although the language structure tends to be relatively simple in such cases, it is not unusual to encounter difficulties with lexical decoding, e.g., in recognizing reserved words embedded in text and in developing suitable delimiting rules for unique meaningful interpretation. Further difficulties arise if the input language has to be changed frequently and on short notice: it may then become increasingly hard to modify the delimiting rules so that previously formulated input as well as new capabilities are properly covered.

This paper presents a general and highly flexible "lexical synthesis" approach to the lexical decoding problem based on systematic string recognition rather than delimiting rules. It has successfully been implemented in an operating general-purpose lexical synthesis package ULEX.

1. INTRODUCTION

Flexibility and ease of use suggest that key specifications to computer program use, as much as possible, natural-language constructs and conventional mathematical nomenclature. Such user-prepared input constructs must form a "language" based on the character set available to the computer, e.g., EBCDIC or BCD, together with a well defined dictionary of allowable words, and a "grammar" which characterizes acceptable syntactic structures. The language may be simple compared to high-level programming languages such as ALGOL and FORTRAN; nevertheless, sound programming procedure requires that a proper grammar be defined such that each acceptable construct is meaningful and moreover has a unique interpretation.

The process of interpreting constructs and testing their acceptability against the specified grammar is generally referred to as "parsing". It is frequently simplified if one defines the underlying grammar in terms of "words" rather than "letters", that is, if certain strings of characters rather than single characters are chosen as the basic symbols. In this situation, parsing must be preceded by a "lexical decoding" procedure, whose purpose is to decompose given strings of characters into strings of words each of which symbolizes a basic grammar unit or "token".

In the remainder of the introduction, we will define the terminology used throughout this paper and outline some possible approaches to the problem of lexical decoding. The rest of the text will describe a method called "lexical synthesis" for solving this problem. A general-purpose lexical synthesis package, ULEX, has been prepared by the authors for the Urban Transportation Planning System (UTPS) program library developed and maintained by the Urban Mass Transit Administration (UMTA) in the U.S. Department of Transportation. It is presently in use, and will be briefly discussed at the end of the paper.

We refer to the character strings which denote tokens as words, and the total character strings to be decomposed as phrases. Words and tokens must be distinguished because of the possibility of homonyms, i.e. identical words denoting different tokens, and of synonyms, i.e. different words denoting the same token. The task of lexical decoding can now be described more precisely: it is to pass from a given phrase to a string of tokens such that words for these tokens constitute the phrase.

Consider for example the following conditional expression

IF AEND GE BEND THEN SQRT ( ER + 5 ) ELSE FS** 2

The blank characters indicate the grouping into words, which represent suitable tokens — in the above example there are 15 of them. Once the words have been replaced by their tokens, subsequent parsing will be easier and most ambiguities will have been eliminated. If the language permits both notations "GE" and "\ge", then this would be an example of a synonym.

The most common approach to lexical decoding uses "delimiting rules" — mostly the occurrence of special characters ("reserved words") — to signal the terminations of the character strings into which the construct is to be decomposed. The grammar of delimiting rules must be such that it is either violated by a given construct or else it guarantees a unique decomposition of that construct into words denoting tokens. This approach to lexical decoding is known as lexical analysis.

Lexical analysis has some serious drawbacks. It relies on delimiting rules which are guaranteed to produce unique decompositions. Such delimiting
rules tend to be "brittle" in that even small changes in the language structure may invalidate them, causing a serious obstacle to "upward compatibility". They are often complicated, hard to document and learn, as well as "unforgiving". The occasional user may not be able to afford the time necessary to master such a language. Also, the danger of mininterpretation due to syntax error increases with the intricacy of the delimiting rules. Finally, the task of modifying the delimiting rules so that previously formulated input as well as new capabilities are properly covered becomes prohibitively complex.

An alternative approach to lexical decoding which avoids many of these drawbacks is lexical synthesis. Lexical synthesis is much more flexible because it prevents ambiguity rather than preventing it, and therefore does not have to rely on intricate delimiting rules or "reserved" words.

Pure lexical synthesis does not infer the beginning and the end of words from the characters encountered. Instead, a dictionary of admissible words is presumed given at the outset in the form of a "dictionary", and the problem is to segment a given phrase into words present on the dictionary. More precisely, all possible decompositions of each given phrase must be found. For example, if the dictionary consists of the strings:

$$\text{A} \
\text{AB} \
\text{AAB}$$

then the phrase $\text{ABABAAB}$ admits the decompositions:

$$\text{AB/A/AB} \
\text{AB/AAAB}$$

Pure lexical synthesis will seldom be sufficient, however. The first and foremost reason for this is that all admissible words have to be in the dictionary. However, units like numbers or variable-names which are introduced by the construct itself cannot be listed in the dictionary beforehand, unless all possible numbers and all possible variable-names are put on the dictionary.

The second reason is that too much ambiguity will result, particularly if homonyms are present. Usually most decompositions of a given phrase are ruled out by their obvious inability to be parsed. But parsing each decomposition in turn so as to ascertain uniqueness is too high a price to pay in most applications since these will, in general, admit numerous decompositions.

This paper will show how the pure lexical synthesis approach can be modified so as to overcome the difficulties mentioned above, while still allowing much more flexibility than would be possible with lexical analysis. It is quite obvious, that this modified lexical synthesis approach is not intended for very long phrases. In this case, we expect that "false starts" may be so numerous as to be unacceptably costly in terms of computer time. We have found that for the phrases arising in our applications, running time was not a problem.

The efforts that led to this paper were initiated and actively supported by Dr. Robert B. Dial of UMTA, who contributed to many discussions and who made available his unpublished manuscript "Lexique". Don J. Orser of NBS brought the work of F. H. Sussenguth to our attention.

2. EXTENDING LEXICAL SYNTHESIS

In this section, we discuss steps to alleviate the two main shortcomings of the lexical synthesis approach identified earlier: The limitations imposed by the size of the dictionary and the problem of ambiguity.

We address the problem of dictionary size first. Suppose a language contains words denoting nonnegative integer numbers, like 15735, 12, and 0, and that its other words are composed of non-numeric characters. It is clearly impossible to include all number words in a dictionary. However, we can include the figures 0,1,2,3,4,5,6,7,8,9, associating ten preliminary tokens or "pretokens" with these symbols. Strings of consecutive numerical pretokens can then be recognized in a subsequent scanning step, and identified with definitive tokens representing integers. Only minor modifications are required to deal with floating-point numbers and other number representations.

This scanning step is, of course, a lexical analysis, albeit a very rudimentary one. If the language in question contains numerical characters in other words, e.g. in variable-names, then one might suspect that the simple scheme above might not work, or at least would require a more complicated subsequent lexical analysis. These complications lie however, in the direction of increased ambiguity of decomposition (a problem which will be addressed shortly) rather than in that of difficulty of scanning: once the numerical pretokens have been assigned together with other tokens in a particular decomposition, then the process of combining these pretokens into definitive tokens is the same as before. We submit that for many languages, the use of pretokens together with a subsequent rudimentary lexical analysis permit an efficient application of a lexical synthesis approach.

The remaining problem is that of unnecessary ambiguities. By this we mean the occurrence of decompositions which are in accordance with the dictionary but clearly make no "sense" at all (e.g. $X = Y * +$, is an expression which consists of the words "$X\text{"}, "$=\text{"}, "$Y\text{"}, "$*\text{"}, "$+\text{"}, but has no mathematical interpretation). Indeed, in most applications parsable token sequences satisfy a set of obvious necessary conditions, and nonparsable decompositions can be detected early if they violate one of these conditions.

The problem of ambiguity due to homonyms occurs in both lexical analysis and lexical synthesis. Consider what happens when a FORTRAN programmer uses the word "IF" as a variable-name. If lexical analysis is used, then it will not be sufficient just to recognize the character string "IF" as constituting a word of the language. The lexical analyzer must also realize from the context which meaning (i.e. which token) is the "correct" one.
The process of lexical synthesis needs a similar mechanism which checks in some sense the "meaning" of a token before selecting it from among several possible ones.

We propose here the following general weeding-out scheme: with each token we associate one or more attributes. For each such attribute, we list those attributes which may immediately follow it. The understanding is that two tokens may follow each other in legal succession only if their attributes are in legal succession. If there are several attributes associated with either of the two tokens, then it suffices for legal succession of the tokens that some attribute of the first token is legally succeeded by some attribute of the second.

The succession rules for attributes are conveniently displayed in a matrix of 0,1-entries called the legality matrix. An entry of 1 in the position corresponding to a pair of attributes indicates that their succession is legal.

It is convenient to introduce two intrinsic attributes which do not belong to words. Called "beginning of string" and "end of string", they are used for expressing that certain attributes may not start or terminate a string, respectively. The first row of a legality matrix will be reserved for "beginning of string", the first column for "end of string".

The main reason for formulating the rules for legal succession in terms of attributes rather than tokens is that the attributes depend only on the language --- and are therefore a fixed part of a program --- while the tokens themselves are application dependent. We illustrate this point, and the use of attributes in general, with some examples.

In many input procedures, the user supplies information using mathematical nomenclature and natural-language constructs to specify how his data files are to be manipulated and transformed. In these cases, operator symbols such as $\sqrt{}$, $\ast$, $\ast$, are given, and the user applies these symbols together with his variable-names and constants to explain the transformations desired, e.g. $let\ Z = \sqrt{(S)}\ast X + Y$. In such cases, the number of variable-names is input dependent, but all variables will share a single attribute called "variable". To assure that a variable does not succeed another variable, which would lead to a plainly meaningless construct, the corresponding entry in the legality matrix is zero.

Similarly, one may elect to have 3 attributes associated with ",", whose token will be "decimal point". These attributes will be "starting point", "middle point", and "end point". Associating the attribute "cipher" with each of the figures 0, 1, ..., 9 and designing legality rules which prevent the attribute "starting point" from following the attribute "cipher" and also "cipher" from following "end point", while accepting only "cipher" as preceding "end point" and "middle point" or succeeding "starting point" and "middle point", assures that floating point numbers like \( .31, 5, .23/7.38 \) admit legal attribute assignments. For instance, \( .31 \) resolves into the tokens "decimal point", ",", and "1" with attributes "starting point", "cipher", and "cipher". On the other hand, strings like 5.3 and single decimal points are ruled out. The string 

\[
5.1.3
\]

is still legal, however. In order to exclude strings of this kind, one may associate an additional attribute "mantissa cipher" with figures. The attribute would have to be preceded by "mantissa cipher", "starting point", or "middle point". Furthermore, "starting point" and "middle point" could be succeeded only by "mantissa cipher", just as "end point" and "middle point" could be preceded only by "cipher".

When using the legality matrix to partition a phrase, three possibilities can occur: 1) A single legal sequence of tokens is found. 2) No legal decomposition is possible. 3) More than one legal sequence of tokens is found. There are several rules for determining legality. We illustrate this point, and the use of attributes in general, with some examples.

Consider a free form of input which ignores blanks, and suppose a user names three variables "PARK", "RIDE", and "PARK AND RIDE". If the logical operator "AND" is on the dictionary, then whenever the user inputs the string "PARK AND RIDE" it will not be clear whether he is referring to the variable "PARK" or "RIDE" or the three word logical expression "PARK AND RIDE". One way of resolving ambiguities is to put optional quotes around variable-names. Thus "PARK AND RIDE" indicates the variable of this name, whereas writing 'PARK' AND 'RIDE' makes it clear that the three-word expression is desired. If the language provides this optional use of quotes then an additional attribute for tokens designating variables is needed, namely "quoted variable" in addition to "variable". If two attributes "quote open" and "quote closed" are associated with the word consisting of a single quote ("), then legality rules which require "quoted variable" to follow "quote open" and to precede "quote closed" will ensure that such pairs of quotes will always enclose a single quoted variable.

We close this section with a more complete example to illustrate how a legality matrix determines which decompositions are allowable. Consider all ways in which to score 0 points in American football. Writing the phrase to be decomposed as

\[
\text{*** field goal (Attribute 3)}
\]

we have the following dictionary of single scoring possibilities:

\[
* \text{ point after (Attribute 2)}
\]

\[
** \text{ safety (Attribute 3)}
\]

\[
*** \text{ field goal (Attribute 3)}
\]

\[
**** \text{ touchdown (Attribute 4)}
\]
and the corresponding legality matrix:

<table>
<thead>
<tr>
<th>Begins By</th>
<th>Followed by</th>
<th>EOS</th>
<th>Point</th>
<th>Safety/Touchdown</th>
</tr>
</thead>
<tbody>
<tr>
<td>BOS</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>Point After</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>Safety or Field Goal</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>Touchdown</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>

Note that "Beginning of String (=BOS)" and "End of String (=EOS)" are assigned attribute 1. Any decomposition of the phrase into segments representing one of the above scoring possibilities would be a decomposition in accordance with the above dictionary. Examples of such decompositions are:

/*****/
/*******/

The second example is clearly not "legal", because a single point can only be scored immediately after a touchdown (six points).

The expanded lexical synthesis problem consists of finding all legal decompositions, namely decompositions whose words are assigned tokens with compatible attributes. The legality matrix is a vehicle for checking the legality of decompositions. The next section will briefly describe how given character sequences can be decomposed into dictionary words.

3. THE BASIC DECOMPOSITION PROBLEM

For purposes of exposition—and because it is a fundamental algorithm of some independent interest—we describe first an algorithm for the pure lexical synthesis problem. This problem requires finding all possible decompositions of a given phrase in accordance with a given dictionary, disregarding homonyms.

To attack the problem of obtaining all segmentations, an ordering of the segmentations is necessary. The following ordering scheme will be used: Let \( n \) be the number of characters in the phrase, and to be analyzed. Then we associate with any given segmentation a vector

\[
L(s) = (k_1, k_2, \ldots, k_n)
\]

where \( k_1 \) is the length of the first segment, \( k_2 \) is the length of the second, and so forth. Typically there will be fewer segments than characters in \( s \), and the remaining components of \( L(s) \) are set to zero. For example, the following segmentation of a 10 character phrase

\[
1 1 1 1 1 1 1 1 1 1
\]

will be represented by the vector

\[
(2, 3, 1, 1, 0, 0, 0, 0, 0, 0).
\]

In general, vector

\[
L^1 = (k_1^1, k_2^1, k_3^1, \ldots, k_n^1)
\]

proceeds vector

\[
L^2 = (k_1^2, k_2^2, k_3^2, \ldots, k_n^2)
\]

if

\[
\begin{align*}
&k_1^1 < k_1^2, \text{ or} \\
&k_1^1 = k_1^2 \text{ and } k_2^1 < k_2^2, \text{ or} \\
&k_1^1 = k_1^2 \text{ and } k_2^1 = k_2^2 \text{ and } k_3^1 < k_3^2, \ldots
\end{align*}
\]

In other words, the \( n \)-vectors \( L \) are ordered lexicographically.

The order relationship of two \( n \)-vectors is defined by the first position in which the \( r \) components differ.

**EXAMPLE:**

**Dictionary:** AA, AAA

**Phrase:** AAAAAAAA

**Segmentation vectors in order:**

\[
(2, 2, 2, 0, 0, 0, 0, 0) \\
(2, 3, 0, 0, 0, 0, 0, 0) \\
(2, 3, 3, 0, 0, 0, 0, 0) \\
(3, 2, 3, 0, 0, 0, 0, 0) \\
(3, 3, 2, 0, 0, 0, 0, 0)
\]

The approach to segmentation described here is one which will find the possible segmentations in lexicographical order. One proceeds recursively, trying to find a smallest word which fits a front segment. The same thing is done for the remaining portion of the string to be analyzed, and so on. If no word is found matching the front portion of the same length, then there is no segmentation. If the search fails after some segments have been established, then one backtracks to the last such segment and tries to make it longer. If this works, then one tries to segment the remainder. If it does not work, one backtracks to the next-to-last segment and tries to make it longer, etc. If no success is achieved, then again there is no segmentation.

Starting from a successful segmentation, in order to find the next one, one tries again to enlarge the next-to-last segment (there is no use trying to enlarge the last segment, obviously) and, if successful, to segment the remainder string. If the latter fails, one tries to enlarge earlier sections. When it is established that there are no (or no more) segmentations, the process of finding all segmentations has been completed.

After having determined a segmentation, all possible combinations of homonyms may be generated "odometer"-style. In this fashion, all token sequences compatible with a given phrase—but disregarding attributes—can be generated sequentially. The next section modifies this procedure so as to combine the process of decomposition with the process of legality checking.

4. RESTRICTED LEXICAL SYNTHESIS

In this section, we describe a method for finding those segmentations of a given phrase which are legal in the sense of section 2: each token has
one or more eligible attributes, and there exists a selection of an eligible attribute for each token in the token sequence so that the resulting sequence of attributes has only permissible successions.

A possible approach is to use the algorithm described in the previous section to generate successively all different token sequences compatible with a given phrase, and then to test the legality of each such token sequence separately. A simple backtracking 'trial and error' scheme, which we will not elaborate on, can be used for deciding whether there exists a legal assignment of attributes to the token sequence in question. However, this straightforward approach, based on generating all segmentations possible in terms of the dictionary, would be quite inefficient for any but the simplest problems.

The approach we adopt instead is to modify the basic decomposition algorithm so as to combine the process of decomposition with the process of legality checking. This involves the legal assignment of attributes to tokens before decompositions are completed, and will keep us from wasting effort in trying to complete an illegal decomposition. (Note that an illegal partial token sequence will give rise to an illegal token sequence for any possible way of completion.)

The method can be used to find all legal token sequences. However, the same legal token sequence may be generated repeatedly, because there may be several ways of assigning attributes which legally succeed each other. In other words, the method is capable of successively finding all pairs of legal token sequences and related legal attribute assignments. We emphasize however, that the latter is not the main purpose of the method; rather its purpose is to detect ambiguity, that is, to find two distinct legal token sequences, or to establish that the one found is unique.

In some applications, the legal assignment of attributes is unique for each legal sequence of tokens. This is plainly the case if there is just one eligible attribute for each token. Then all legal token sequences will be found in sequence without duplications. The example of finding all combinations by which football scores can be achieved (see section 2) is of this kind.

A partial segmentation of a given phrase is a segment of that phrase which has been decomposed into tokens whose corresponding attributes have produced a legal and, therefore, grammatically acceptable portion of a phrase. For this algorithm, the first partial segmentation will be the shortest initial portion of the phrase corresponding to a word on the dictionary which has associated with it a token which can begin a phrase.

The method proceeds as follows. Suppose a partial segmentation of the given phrase has been achieved, tokens for the resulting words assigned and attributes for the tokens selected in a legal fashion. We then try to enlarge the partial segmentation by decomposing the remainder of the phrase. Consider the shortest initial portion of the remainder which can be found in the dictionary, examine the first available token for this word, and search among the attributes of the token for one that fits the preceding attribute. If such an attribute exists, the partial sequence of tokens has been extended in a legal fashion, and the reduced remainder, if there is still one, is considered next. If none of the attributes of the new token fits, then a different token is considered, and so on, until either an extension of the legal token sequence results, or the last word is found to be not suitable. In this case, we search for the next longer fitting word, as in the basic segmentation procedure, and if we are successful, we will proceed again to try finding suitable tokens and attributes. If the remainder is found to be intractable, the attribute of the last legal token is replaced, if possible, by an alternative one keeping the succession legal, and a new attempt is made to segment the remainder. If there is no such attribute, a different token is substituted for the last legal word, again in such a fashion that legality is preserved. All possible legality preserving attributes for the new token are tried successively until either an extension of the legal sequence results, or the last token is found not to fit. Then other tokens are tried, which may or not meet with success. If no further extension is found, then the last word is replaced by a longer one, and the search for legal tokens and attributes is repeated.

Note that the above algorithm fits the general backtracking scheme if we consider sequences which start with the attribute "beginning of string", followed by the first word in the phrase, followed by a token for that word, followed by an attribute of that token, followed by the next word in the phrase, and so on. Each sequence found defines a linearly ordered finite set of feasible extending elements. If the last element in the sequence is a word, then the extending elements are the tokens, i.e. the homonyms, of that word. If the last element is a token, then the extending elements are those attributes of the token which can legally succeed the previous attribute. If the last element is an attribute, then the extending elements are found as front ends of the remainder of the phrase.

The algorithm then consists either of extending the current sequence by adding a new element at the end, or of replacing the last element by the feasible extension next in line. Upon reaching a "dead end", one "back-tracks", that is, discards the last elements successively until an element which is still legal is reached.

The method of lexical synthesis detailed here requires that all allowable words exist in a dictionary. The next section will describe a method for organizing such a dictionary.

5. THE SUSSEXGUTH DICTIONARY

The algorithms for lexical synthesis require the
availability of a dictionary of admissible strings of tokens. This section describes a form of organizing a dictionary, due to Sussenguth [3], that is particularly well suited for these algorithms. This approach stores the words of the dictionary in the form of a "tree" (a tree is a graph which contains no circuits and has at most one branch entering each node). The characters of the strings are identified with nodes of the tree, permitting fast dictionary searches. For example:

**DICTIONARY**

<table>
<thead>
<tr>
<th>Word</th>
<th>Dictionary Entry</th>
</tr>
</thead>
<tbody>
<tr>
<td>AB</td>
<td></td>
</tr>
<tr>
<td>ABCD</td>
<td></td>
</tr>
<tr>
<td>CDA</td>
<td></td>
</tr>
<tr>
<td>ABD</td>
<td></td>
</tr>
</tbody>
</table>

**SUSSENGUTH TREE**

```
          A
         / \
        B   C
       / \ / \   /
      D   E   F  G
```

In the above tree, every path starting at "root" node 0 and proceeding to a node with "BREAK" describes a word in the dictionary and has associated with it a pointer to this word. In order to determine whether a given word belongs to the dictionary, one looks for the first character of the word among the immediate descendents (i.e. filial set) of node 0. If a match is found, then the immediate descendents of the corresponding node are searched for occurrences of the second character, and so on.

A similar procedure would be used whenever a new word was to be added to the dictionary. The word to be added is examined character by character to see if the string already exists on the dictionary. If only part of the word exists on the dictionary, then additional nodes are added to the branch describing the corresponding partial string, in a filial manner, until the entire word has been built into the dictionary. The "BREAK" flag signalizing the completion of the word and the word pointer are then associated with the node corresponding to its last character.

Since the characters associated with nodes describe the dictionary entirely, no other dictionary description is necessary. However, the same word represents different tokens whenever homonyms are present. Associated with each word in the dictionary is, therefore, a pointer to the set of all tokens represented by this word. Furthermore, each token will have similarly associated a set of associated attributes. (Semantic information associated with tokens for parsing purposes resides outside the Sussenguth dictionary.)

This organization allows the dictionary to be both searched and altered efficiently, and is well suited for lexical synthesis.

6. A GENERAL-PURPOSE LEXICAL SYNTHESIS PACKAGE: ULEX

The problem of lexical decoding was encountered when developing the input processor for a comprehensive regression package for UTPS (The Urban Transportation Planning System), a collection of compatible computer programs for transportation planners, which is developed and extensively disseminated by MFTA (The Urban Mass Transit Administration).

Based on the considerations presented earlier, our chosen solution to this lexical decoding problem was the development of a general-purpose lexical synthesis package called ULEX. This program has four entries besides the main entry ULEX, which reads the legality matrix and initializes the work vectors. Two entries build and make deletions from the Sussenguth dictionary. UWORD is the entry which adds a single token to the dictionary, USCRAP deletes from the dictionary the token last entered. The entry UREQUEST reads the phrase to be analyzed. Finally, UREQUEST is interrogatory: upon each call, it produces a token sequence for the phrase. This token sequence is guaranteed to be different from the one produced immediately before; if there is no such token sequence left, this fact is reported. Therefore, two calls to UREQUEST are sufficient to decide whether a unique legal token sequence for the given phrase exists. (While the above capabilities of ULEX were sufficient for our purposes, additional capabilities are clearly desirable such as a look-up capability for given words yielding all homonyms and the deletion of arbitrary specified dictionary entries.)

The UTPS application mentioned above requires that transportation planners be able to input phrases in a free format mode. These input phrases are often quite complicated, since conditional expressions, various binary and unary functions, lagged expressions, and many other operations are part of the language. Some examples of typical input statements follow:

```
LET (AUTO:=AUTOTIME**2 + 2.42*LOG(AUTO*1.345))

LET (X := IF not= B THEN E ELSE MOD(P,10))
```

The operator library allows synonyms for many of the most commonly used words (e.g. "NOTE","NP","NE"). This synonym flexibility helps a diversity of users to input statements using the language constructs with which they are most familiar. Although synonyms require more entries on the Sussenguth dictionary, they do not increase the number of tokens and, in general, do not complicate a lexical synthesis approach to decomposition since the legality matrix will not be altered.

During the lexical decoding phase of the processing, constants are expressed as single-character string tokens. A subsequent user-supplied procedure to collect these tokens is required. Similarly, a prescanner is employed to screen out those portions of the input which are to be subjected to lexical synthesis.

Twenty-six attributes were sufficient to analyze all input strings containing operator words, variable-names and constraints supplied by the user. Note again that the corresponding legality matrix is a fixed part of the program, since it is language rather than application dependent.
Our experiences with the lexical synthesis approach in the above context have been extremely encouraging. They have led us to believe that this approach could be useful for many other lexical decoding problems of similar size and scope.

REFERENCES


