A RANDOM WORD GENERATOR
FOR PRONOUNCEABLE PASSWORDS

NOVEMBER 1975

Prepared for

DEPUTY FOR COMMAND AND MANAGEMENT SYSTEMS
ELECTRONIC SYSTEMS DIVISION
AIR FORCE SYSTEMS COMMAND
UNITED STATES AIR FORCE
Hanscom Air Force Base, Bedford, Massachusetts

Approved for public release; distribution unlimited.

Project No. 522N
Prepared by
THE MITRE CORPORATION
Bedford, Massachusetts
Contract No. F19628-75-C-0001
When U.S. Government drawings, specifications, or other data are used for any purpose other than a definitely related government procurement operation, the government thereby incurs no responsibility nor any obligation whatsoever, and the fact that the government may have formulated, furnished, or in any way supplied the said drawings, specifications, or other data is not to be regarded by implication or otherwise, as in any manner licensing the holder or any other person or corporation, or conveying any rights or permission to manufacture, use, or sell any patented invention that may in any way be related thereto.

Do not return this copy. Retain or destroy.

REVIEW AND APPROVAL

This technical report has been reviewed and is approved for publication.

F. WAH LEONG, Captain, USAF
Project Officer
Air Force Data Services Center

ROGER R. SCHELL, Major, USAF
Project Engineer

FOR THE COMMANDER

FRANK J. EMMA, Colonel, USAF
Director, Information Systems
Technology Applications Office
Deputy for Command and Management Systems
The random word generator is a PL/I program designed to run on Honeywell's Multics system that generates random pronounceable words suitable for use as passwords for Multics users.
ACKNOWLEDGEMENT

Special acknowledgement is extended to Lt. Col. Robert Park and 1Lt. Brian Woodruff for their expert guidance in the preparation of Section IV of this report on the statistical analysis of the random word generator.
TABLE OF CONTENTS

LIST OF ILLUSTRATIONS 5

SECTION I  BACKGROUND 7

SECTION II METHODOLOGY 9
   REQUIREMENTS AND GOALS 9
   PRONOUNCEABLE? 10
   THE POOR APPROACH 11
   THE IDEAL APPROACH 11
   THE COMPROMISE APPROACH 12
   SYLLABLES 13
   JUXTAPOSITION 14
   MISCELLANEOUS ASSUMPTIONS 15
      Consecutive Vowels 15
      The Vowel "y" 15
      The Silent "e" 16
      The Initial "y" 16
      Three Identical Units 16
   SUMMARY 16

SECTION III IMPLEMENTATION DETAILS 18
   SPECIFICATION OF RULES 18
      The Digram Table 18
      The Unit Table 20
      Random Units 21
      The Algorithm 21
   RESULTS 24

SECTION IV  ANALYSIS 25
   NUMBER OF WORDS 26
   PROBABILITY OF A WORD 28
   MOST PROBABLE WORD 29
   DISTRIBUTION OF PROBABILITIES 31
      Determination of Distribution 32
      Application of the Distribution 33
      AN ALTERNATIVE METHOD 35

SECTION V  CONCLUSION 40
   EVALUATION 40
   OTHER APPLICATIONS 41

APPENDIX I  TABLES 43
<table>
<thead>
<tr>
<th>APPENDIX II</th>
<th>RANDOM WORD ALGORITHM</th>
<th>50</th>
</tr>
</thead>
<tbody>
<tr>
<td>APPENDIX III</td>
<td>SOURCE CODE</td>
<td>58</td>
</tr>
<tr>
<td>APPENDIX IV</td>
<td>2000 RANDOM WORDS</td>
<td>113</td>
</tr>
<tr>
<td>APPENDIX V</td>
<td>STATISTICS</td>
<td>129</td>
</tr>
<tr>
<td>APPENDIX VI</td>
<td>DOCUMENTATION</td>
<td>132</td>
</tr>
<tr>
<td>APPENDIX VII</td>
<td>MODIFIED SOFTWARE FOR UNIFORM DISTRIBUTION</td>
<td>173</td>
</tr>
<tr>
<td>REFERENCES</td>
<td></td>
<td>193</td>
</tr>
</tbody>
</table>
## LIST OF ILLUSTRATIONS

<table>
<thead>
<tr>
<th>Figure Number</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Random_word_ Flowchart</td>
<td>23</td>
</tr>
<tr>
<td>2</td>
<td>Number of Words of 6, 8, and 10 Letters</td>
<td>28</td>
</tr>
<tr>
<td>3</td>
<td>Distribution of Probabilities of Random Words</td>
<td>32</td>
</tr>
<tr>
<td>4</td>
<td>Distribution of Probabilities of 2653 Eight Letter Words</td>
<td>34</td>
</tr>
<tr>
<td>5</td>
<td>Enlarged Left Edge of Distribution</td>
<td>36</td>
</tr>
<tr>
<td>6</td>
<td>Distribution of Probabilities of 5893 Six Letter Words</td>
<td>130</td>
</tr>
<tr>
<td>7</td>
<td>Distribution of Probabilities of 1039 Ten Letter Words</td>
<td>131</td>
</tr>
</tbody>
</table>
SECTION I

BACKGROUND

The random word generator is a PL/I program designed to run on Honeywell's Multiplexed Information and Computer System (Multics), a large timesharing system. The purpose of the program is to generate passwords that serve to authenticate the identities of users of Multics.

Users of the standard Multics system authenticate themselves at each login (start of a terminal session) by typing in a password known only to the user and the system. Usually the user selects a password for himself at initial login and can change this password at any subsequent login. The ability to change a password is useful when a user suspects that someone else may have guessed his password.

A password is the key to user identification and protection. From previous experience, however, it has become apparent that user-selected passwords are frequently fairly easy to guess. For example, passwords are often the user's own first name, a name of a family member, or his telephone number. [1] Some Multics installations, such as the Air Force Data Services Center (AFDSC), are used to process classified information, and installations like the AFDSC cannot take a chance that one of their user's passwords will be guessed. The solution to the problem at the AFDSC was a decision to assign passwords to users, rather than to allow users to pick their own. [2]

The administrative overhead of assigning a random password manually whenever a user changes his password is generally too much of a burden -- especially when one considers that only a select few individuals may have access to other users' passwords. Instead, the possibility of providing computer generated passwords that are printed out at the user's console on each password change was investigated. A user's request to change his password would then become a call to a system password generator program. This password generator has been given the more general and descriptive name of "random word generator" in this report.

Section II of this paper discusses the goals and methods used by the random word generator. Section III contains implementation details and discusses the word generating algorithm. Section IV contains an analysis of the algorithm and presents certain statistics. A sample of random words generated can be found in Appendix IV.
Most of this report describes the random word generator that is being made available for users on Multics. In order to satisfy a more stringent criterion of "randomness," a modified version of the random word generator has also been prepared that generates random words that are all equally probable. This modified version is discussed at the end of Section IV.
SECTION II
METHODOLOGY

REQUIREMENTS AND GOALS

The need for the Multics random word generator program can be satisfied by fulfilling the following two requirements:

- generate easily remembered words, and
- make the words difficult to guess.

The first requirement is very important because users might be inclined to write down passwords that are difficult to remember, thereby increasing the chances of a password compromise. Also, if too many users forget their passwords, the administrative overhead of getting the users logged in again could be greater than that of distributing easier-to-remember passwords manually. The need for the second requirement is apparent.

Both of these requirements are of course far too subjective for direct implementation as a computer program. It is necessary to restate these requirements in a much more concrete manner so that meaningful algorithms can be designed.

The requirement of "making the words difficult to guess" is most easily satisfied by giving the program the ability to generate a very large set of possible words, and the ability to generate these words in a random manner. Both these capabilities can easily be achieved, and thus we will be more concerned at this point with the first requirement.

Consider that the random word generator either needs a large database of words to choose from -- an impractical approach that has been discarded -- or has to form words out of sequences of letters it creates through some algorithm. The requirement of rememberability can then be fulfilled if these sequences of letters are of one or more of the following types:

1. Sequences of letters that can be easily visualized, such as "aabbab" or "xyxyxy".
2. Sequences of letters that form real English words.
3. Sequences of letters that form pronounceable "words", but are not necessarily real words.

Of these three choices, methods 1 and 2 suffer from the difficulty of specifying a practical algorithm for such sequences. Alternatively, method 1 could be implemented using rules that yield an arbitrarily defined subset of all possible easy-to-visualize sequences, but this subset is likely to be small for a reasonable number of rules. There is no alternative for method 2 other than storing a vast database of real words.

The third method — that of using pronounceable sequences — is the selected approach. The database required for this method is relatively small, the rules can be fairly well-defined, and the set of words that can be generated is quite large. Realizing that the more "English" a word looks the easier it usually is to remember, an attempt was made to restrict the set of words generated to those which obeyed some kinds of rules of English pronunciation. This attempt was restructured as an attempt to make it theoretically possible for the word generator to form most English words.

Because of this goal of making the generated words look like English, the word "pronounceable" in this paper refers not just to structures that can be phonetically vocalized, but to a set of more restrictive and English-looking structures. For example, "tsip" is easily pronounceable, but is "un-English" because of the "ts" at the beginning of the word. A different type of example is the "gh" combination. The word "cough" is pronounceable because "gh" in this context can be pronounced like "f", but "ghrom" is not pronounceable as "from" because "gh" never sounds like "f" at the beginning of a word.

PRONOUNCEABLE?

One may wonder how a goal of "pronounceability" can be attained with well-defined rules, considering how undefined and exception-laden the rules of English pronunciation are. The answer is simple: the program does not care how a given word is to be pronounced -- it only needs to make sure that the word can be pronounced. For example, "tophat" could be pronounced "top-hat" or "to-fat", depending on the reader's preference. On the other hand, "tophsat" is only pronounceable as "tof-sat", not "top-hsat". Also, the vowel "o" in this word might be pronounced in one of several ways.

Everyone knows that a given letter or sequence of letters could be pronounced differently in different contexts, but the program is usually not required to distinguish between the different contexts or pronunciations. Unlike the "rules of pronunciation", the "rules of
pronounceability" can be made fairly precise.¹

However precise, the rules and method used to generate pronounceable words have been arrived at in a "refined" ad hoc manner and based on the author's intuition. The method is not described in any published source. Hence, the words generated may be considered pronounceable only by the author. Others may find some of these words very difficult to pronounce, as when trying to pronounce a foreign word with a strange combination of letters. Because of this possible bias, the program was designed to incorporate as few global rules as possible within its text. An external data base, a table, is used to contain most of the subjectively determined rules. These external rules consist of "yes" or "no" answers to various questions asked by the program. The answers to the questions can easily be modified to suit the user's preference. "New" rules -- those asking new kinds of questions -- cannot be added without modifying the program.

THE POOR APPROACH

Letters are poor sources on which to base rules of pronounceability. Not only do individual letters sound different in different words, but pairs or triplets of letters often form single sounds that may be unlike any of the component letters. Determining whether a letter is pronounceable or "legal" in a given word often involves knowing how the letter is to be pronounced, which is in turn dependent on such things as its position in the word or syllable and adjacent letters. The large number of details that have to be checked for each letter makes determination of pronounceability very complex.

THE IDEAL APPROACH

The ideal approach that will always generate good pronounceable words would be to relieve the program of any notion of letters and use

¹Phonological theory is a well developed science that in part attempts to describe the phonetic structure of English (and other languages) in a complete and consistent manner. The totality of rules and theorems used in such a description form far too complex a system for the scope of the application discussed in this report. Creation of a smaller subset of this system -- one that might be small enough to implement and would still give reasonable results -- appeared to be too vast an undertaking. Thus, standard phonological theory was not considered in this work.
"phonemes"\(^2\) instead. A phoneme is an "element" of pronunciation -- a unit of sound that cannot be usefully broken down into smaller sounds. For example, the pair "sh" as pronounced in English can always be represented as a single phoneme; the vowel "a" can be represented as one of several phonemes depending on its context. If the rules could be defined, it should be possible to put together random phonemes to form a pronounceable phoneme-word.

Unfortunately, though this method yields good pronounceable sequences, the translation from phonemes to letters is very difficult and very un-algorithmic. An example of this difficulty is the phoneme representing the sound of "k". This phoneme can be translated into "c", "k", or "ck". Which one should be used? At the end of a word, usually any one of these will work, and another randomization factor has to be included to make the choice. At the beginning of a word, "k" is always legal, "ck" is never legal, and "c" is legal only if the following letter is not "e", "i", or "y" (in which case "c" would have been pronounced like "s"). Then, to determine whether "c" is a candidate the following phoneme must first be translated into letters, which may in turn depend on other adjacent phonemes. One can fix the translation so that the "k" sound is always translated to the letter "k", but then the goal of being able to generate most English words would be far from satisfied (not to mention that the letter "c" would never be used).

THE COMPROMISE APPROACH

One may notice that with the phoneme method the program "knows" how the generated word is pronounced -- specifically not a requirement as mentioned earlier. A compromise approach was chosen that uses simple "units", instead of phonemes, that consist of a single letter or a two-letter pair. A given unit is considered by the program as having only one "sound" in all its usages, although in reality that unit may be pronounced in many different ways.

Rules can be determined for each unit, without regard to how that unit is pronounced in context, by merely stating a rule that includes all usages of that unit. This composite rule is usually simpler than all of the individual rules for the different pronunciations of that unit. For example, the letter "g" can be treated as a unit, and the rule for this unit at the beginning of a word says "this unit may only be followed by a, e, i, l, o, r, u or y." In some of these cases "g" is pronounced soft and in others hard -- in fact there is no simple rule for how "g" is pronounced (e.g., "gigantic" and "giggle"),

\(^2\)also called "phonetic segments" in phonology.
but the program doesn’t need to make any distinctions.

The 34 units presently used are listed below. These units are stored in a table and are input to the random word generator. A larger or smaller set can be defined if experience indicates these to be unsatisfactory.

```
a f k p v ch th
b g l r w gh wh
c h m s x ph qu
d i n t y rh ck
e j o u z sh
```

Note that the letter "q" is the only letter not appearing as a one-letter unit because English usage makes it more convenient to treat "qu" as a unit. Many two-letter vowel combinations, such as "ea", "ie", "ai", etc., that should be considered separate units are not included because little loss of generality occurs (i.e., the set of words that can be generated is nearly the same whether these vowels are separate units or not). Also, double letter pairs like "ll", "rr" and "tt" need not be included for similar reasons. On the other hand, the pair "sh" is needed because words such as "shrink" and "wash" are not pronounceable when "s" and "h" are treated as separate units.

SYLLABLES

Besides the rules used by the program, there is a primary assumption that governs the formation of words: if pronounceable syllables are concatenated (subject to some minor restrictions), they will form a pronounceable word. Thus, the task of the random word generator is to form pronounceable syllables.

This task requires precise definition of "syllable"; thus the following definition is made at this point: a syllable is an arbitrary series of units that contains exactly one or two consecutive vowel units. Vowel units are "a", "e", "i", "o", "u", and "y". For example, the following are legal syllables (where "v" represents a vowel unit and "c" represents a non-vowel unit, or consonant unit):

```
ccv cvvccc cv vv vc
```

and the following are illegal syllables:

```
cc (no vowels)
vvv (more than 2 consecutive vowels)
vccvc (all vowels not consecutive)
vvc (all vowels not consecutive)
```
Note that each of the last two examples can possibly be split into two syllables, such as "vo-cvo" and "v-cv".

The above definition of "syllable" seems to work in English except for one common case: the silent "e" at the end of words or sometimes syllables often forms a syllable containing two non-contiguous vowels. Of course, that is because English usually only requires a vowel sound in a syllable, and in the case of silent "e" the "e" should not be considered a vowel. The program, however, has no way of telling whether the "e" is silent. To make matters worse, there are words, such as "subtle", "bugle", "little" that do have a final syllable whose only vowel is the final (silent) "e". These are common enough cases in English to warrant special consideration in the word generator.

JUXTAPOSITION

The random word generator forms syllables from left to right, by combining random units one at a time. For each new unit the program determines whether that unit can legally be appended to the units already in the syllable. If it cannot, the unit is discarded and another random unit is tried.

In English the legality of a unit is usually determined by checking immediately adjacent units. Units separated from each other frequently affect each other's pronunciation but only occasionally determine whether the construction is legal or not. The random word generator uses rules of juxtaposition as the bases for creating pronounceable syllables.

Each time the program gets a new random unit, it forms a pair consisting of that unit and the previous unit. This unit-pair is looked up in a table and bits of information are extracted that specify what can be done with that pair. For example, the unit-pair "rt" will have bits specifying that the pair may not begin a syllable and that a vowel must precede this pair if it is entirely contained within a syllable. The table may sometimes specify that a unit-pair must always be split between two syllables (for example, the pair "kp"), which is one way in which a new syllable can be started. Some pairs, such as "hh", can never appear together, even if split between syllables. The different types of rules that can be specified in the table are discussed in the next section.

\[3\] The vowel pair "ue" in "baroque" and "catalogue" is another exception, though much less common.
MISCELLANEOUS ASSUMPTIONS

Several more assumptions have to be made before syllables can be generated properly. Again, these assumptions were arrived at intuitively and no claim is made for their completeness. The assumptions discussed below are those that have been incorporated into the program structure, as opposed to those that are specified in external tables. They are presented in order of importance.

Consecutive Vowels

A rule, in part already stated, involving consecutive vowels, says that a maximum of two consecutive vowel units is permitted. This rule pertains to all consecutive vowel units even across syllables. The reason for this extension across syllables is that sequences such as "aiea" look "funny" and are sometimes difficult to pronounce, even though there can be a syllable split in the middle. The English language itself "admits" of this difficulty between consecutive words by trying to correct it in two common cases: the use of "an" instead of "a", and the alternate pronunciation of "the", when the following word begins with a vowel sound. There are few English examples of more than two consecutive vowels (the "eau" combination is one of them). Note that the word "queen" is legal according to random word generator rules because "qu" is considered to be a consonant unit.

A difficulty with this assumption involves the unit "y". For purposes of syllabification, "y" must be treated as a vowel (i.e., a syllable can contain the single vowel "y"), but for the above assumption "y" should not be treated as a vowel. Three-vowel sequences involving "y" are very common: "eye" and "you" being two examples. Thus the requirement of at most two consecutive vowel units must be waived if one of the vowels is "y".

The Vowel "y"

In order to solve the consecutive vowel problem above it sufficed to treat "y" always as a consonant. However, it should also be legal for "y" to be the only vowel in a syllable. Therefore, for the purposes of syllabification only, the random word generator treats "y" as a vowel only if the "y" is not immediately preceded by a vowel within that syllable. The sequence "vowel-y-vowel" would thus have to be split between two syllables, but "y-vowel-vowel" would not. The additional rule about silent "e" below allows a "vowel-y-e" sequence to end a word.
The Silent "e"

The special case of final "e" has previously been mentioned. The final "e" in a word in English is almost never pronounced and therefore cannot be used as the only vowel in the last syllable. There is no problem taking care of such exceptions in a uniform way. However, there is a very large set of exceptions to this final "e" rule: words such as "meddle", "nestle", "double" -- all ending in "le" -- are legal words in English, yet no vowel is pronounced in the last syllable. The rules used by the word generator do not allow final syllables of "ble" and "tle" and therefore such words cannot be generated. This class of words appears to be the largest that cannot be handled by the word generator. In order to solve this deficiency it would be necessary to first include "le" as a unit in the table, and then make special kinds of tests to determine whether this unit is legal in a given context. It is not possible, without creating new rules specific to this "le" unit, to specify the necessary restrictions. Creating new rules for this case was considered feasible, but appeared to be too awkward and so was left out.

The Initial "y"

The unit "y" may not be the only vowel in the first syllable of a word if the word begins with "y". Only strange words like "yclept" violate this rule. This is a minor point but must be taken care of explicitly. Otherwise, many strange words are generated.

Three Identical Units

There is nothing in the rules so far stated that prohibits three or more identical consecutive consonants. This condition may possibly be legal, provided that no more than two consecutive consonants occur in the same syllable. Instead of trying to force a syllable split between such groups, the decision was made to merely limit the number of consecutive identical units to two. Note that this restriction is not a pronounceability problem, but a case of an un-English-looking construction.

SUMMARY

The goal of the random word generator is to generate easily remembered words that are difficult enough to guess to be suitable for passwords. This goal has been translated into requirements of pronounceability and randomness. An attempt was made to include almost all English words in the set of words that can be generated, and to exclude constructions that are never found in English words.
The random word generator works by forming pronounceable syllables and concatenating them to form a word. Rules of pronounceability are stored in a table for every unit and every pair of units. The rules are used to determine whether a given unit is illegal or legal, based on its position within the syllable and adjacent units. Most rules and checks are syllable oriented and do not depend on anything outside the current syllable. In a few cases checks do extend outside the current syllable. These cases are:

1. Three identical consecutive units
2. Three consecutive vowel units
3. Silent "e" at the end of a word
4. "y" beginning a word
5. Certain illegal pairs of units
SECTION III
IMPLEMENTATION DETAILS

The random word generator is organized as a main procedure that references two tables and an external procedure. The user supplies the two tables: a "unit" table that defines the units (such as those listed on page 7) and specifies rules about each unit, and a "digram" table that specifies rules about all possible pairs of such units. The random_unit subroutine, which returns a random unit when called by random_word_, must also be provided by the user. The method used by this subroutine to generate the random units may be any method desired and based on any distribution. Such a distribution might, for example, be based on the frequency of use of the individual units in English.

SPECIFICATION OF RULES

As mentioned in Section II, the random word generator uses two types of rules: those that are fixed and embodied in the program structure and those that are variable and embodied in external tables. The fixed rules are general in that they are not specific to any one letter or unit. The tables specify rules pertaining to individual units or the juxtaposition of units. The tables will be discussed first, followed by specification of the internal rules.

The Digram Table

This table contains one entry for every possible pair of units (digram), whether that pair is allowed or not. Thus, with 34 different units, there would be 1156 entries. The entry for each pair consists of eight bits of information that together form the "rules" for that particular digram. Each bit is a yes or no answer to a specific question asked by the program. The name of each of these bits and the questions answered are as follows:

1. must_begin  Must this pair begin a syllable?
2. not_begin    Is this pair prohibited from beginning a syllable?
3. break        Is this pair illegal within a syllable (i.e. must it be split between two syllables)?
4. prefix       Must this pair be preceded by a vowel unit if it does not begin a syllable?
5. suffix      Must this pair be followed by a vowel unit if it does not end a syllable?
6. end         Must this pair end a syllable?
7. not_end     Is this pair prohibited from ending a syllable?
8. illegal_pair Is this pair illegal (even if split between syllables)?

Obviously all eight bits are inherently non-independent. There are actually far fewer combinations of these eight bits that can be specified. Out of these, less than sixteen combinations are ever used in practice due to the structure of the English language. Thus, four bits yielding 16 combinations would be enough. The actual internal representation of these bits only affects speed and storage space, however, and is not of importance in this discussion. In addition, some other application of the random word generator (perhaps with a different language) may use more combinations. Appendix I contains the digram table currently in use for the 34 units defined on page 7.

An example will best illustrate the usage of these bits. Consider the digram table entry for the pair of units "f" and "l" as shown in Appendix I. The bits that are set for "fl" are:

must_begin  suffix  not_end

The must_begin bit says that if an "fl" is encountered in a syllable, it must begin that syllable. The suffix bit says that the unit following "fl" must be a vowel if "fl" is not the last pair in a syllable. The not_end bit says that "fl" may not be the last pair in a syllable. The specification of the digram "fl", thus, restricts its use within a syllable as the first pair in one of the following six contexts:

fla..., fle..., fli..., flo..., flu..., fly...

where "..." signifies additional units within the syllable. Of course, if there are any further restrictions on the use of the pairs "la", "le", etc. that prevent them from appearing after the "f", these restrictions must be taken into account. Note that none of the eight digram bits except illegal_pair apply when the pair is split between two syllables. If "fl" is split, the "l" becomes the first unit of
the next syllable, and rules for pairs beginning with "1" must be examined. A quick glance at the digram table shows that all pairs beginning with "1" have the not_begin bit set, except the six pairs:

la, le, li, lo, lu, and ly,

and processing can continue with this information.

The random word generator makes sure that at all times the rules specified in the digram table are satisfied for every two consecutive units in the word being formed.

The Unit Table

In addition to rules for unit pairs, there is a table containing four bits of information pertaining to the individual units. For each unit, the four bits are as follows.

1. not_begin_syllable

   This bit indicates that this unit may not begin a syllable. This bit is redundant in that the digram table can specify that all possible pairs beginning with this unit may not begin a syllable. The purpose for using this bit is for efficiency -- when generating the first unit of a new syllable, the program would otherwise have to search through all possible digrams beginning with this unit in order to determine whether this unit is legal. This bit is currently set for the units "x" and "ck". A small number of words in English do begin with "x", but they are mostly technical or scientific terms.

2. no_final_split

   This bit indicates that this unit, when appearing at the end of a word, must not be the only vowel in the final syllable. This bit is only set when the "vowel" bit is set, and is currently set only for the unit "e".

3. vowel

   This bit is set for vowel units. It is currently set for the units: a, e, i, o, u, but may also be set for any units consisting of vowel pairs or that are to be treated as vowels that one might add to the table at some future time.
4. alternate_vowel

This bit indicates that this unit is to be treated as either a vowel or a consonant, depending on context as discussed in Section II of this report on page 9. This bit is set only for the unit "y".

Admittedly these four bits are highly specialized and at least bits 2 and 4 could just as easily be incorporated into the program logic as tests for specific units. However, the program actually works with numbers representing units, rather than the units themselves, and the assignment of a particular number to a particular unit is arbitrary. By using a bit in the unit table for all special cases, all references to specific letters or units are removed from the program. Refer to Appendix I for the unit table currently in use.

Random Units

As stated earlier, the random word generator requires the user to supply the subroutine random_unit. This routine is called by the word generator each time a random unit is needed. The random units are generated based on some predetermined distribution. Of course, not all units thus generated will be acceptable to the word generator in every position of the word: random_unit will be repeatedly called until an acceptable unit is returned. The actual distribution of legal units is different for every position in a particular word, which, for any unit, depends on the units that precede it and the digram and unit tables. The random_unit subroutine itself makes no tests for legal units, but merely uses its fixed distribution each time it is called.

The distribution of units that is currently in use along with the digram and unit tables discussed earlier is shown in Appendix VI in the description of the random_unit subroutine. There is another entry point in random_unit called random_vowel, which is called by the word generator for efficiency in cases when it is known that only a vowel unit will be acceptable. The distribution of vowels returned by this second entry is also shown.

The Algorithm

The digram table, the unit table, and the random_unit subroutine are considered user-supplied in that they may be modified without affecting the word generator program logic. The external rules were specified in the two tables. The algorithm used to generate random words based on these external rules defines the fixed internal rules. The internal rules cannot be modified without changing the logic of the algorithm. The complete algorithm is shown in Appendix II, writ-
ten in a PL/I-like language, and a high level flowchart is shown in figure 1. Appendix III contains the source program listing of random_word which implements this algorithm.

The function of the main body of the algorithm is to determine whether a given unit, generated by random_unit, can be appended to the end of the partial word formed so far. If illegal, the unit is discarded and random_unit is called again. Once a unit is accepted, various state variables are updated and a unit for the next position in the word is tried. A unit previously generated and accepted can never be discarded.

The flowchart in figure 1 shows generally how a word is built up. The names in all capitals (INDEX, SYLLABLE_LENGTH, etc.) are references to variables initialized within the flowchart. Names in quotes (e.g., "syllable_length") refer to the bits in the digram table or unit table for the last pair of units or the current unit. The array UNIT holds the units of the word as they are generated, where UNIT(INDEX) is the current unit.

Beginning at the top of the flowchart, the first unit of the word is selected at random by random_unit and inserted into UNIT(1). If this unit is legal, according to rules in the unit table, the second unit of the word is selected and loaded into UNIT(2). This time the rules must be satisfied for both the unit table entry for UNIT(2) and the digram table entry for the pair [UNIT(1),UNIT(2)]. If a given unit is not acceptable, another is tried in its place. When the end of the word is reached (as determined by the number of letters desired by the caller of random_word_), additional checks are made before the algorithm can terminate.

If the digram table is consistent, there should always be some unit that will be legal for any legal state of the algorithm. However, self-consistency checks on the digram table are extremely difficult to make. Therefore, an arbitrary limit of 100 tries is placed on generating any particular unit. If 100 calls to random_unit fail to yield a legal unit, the whole word is discarded and the program starts over. This 100 tries limit is not explicitly shown in the flowchart but is contained in the program text (see Appendix III).

Another observation concerning the 100 limit is that, because the program is dealing with random events, it is theoretically possible for 100 tries to fail to yield a legal unit even though there is a

---

4 A "state" here is defined by the values of the state variables used in the algorithm as given in Appendix II, and includes the units already accepted as part of the word being formed.
Figure 1. Random_Word_Flowchart
unit that is legal. Thus, in order to prevent excessively long loops, it is useful to place a limit on the number of tries, even though the digram table may be consistent.

RESULTS

Appendix IV contains a printout of 2000 random words of five to eight letters. The length of five to eight letters was chosen for this run because such words are more pronounceable than longer words, and fewer than five letters causes too many duplicates to be generated, thus making the words unsuitable for use as passwords. Actually the random word generator has a capability of generating words of any length. The words in the printout have been sorted alphabetically merely as an aid to checking certain constructs. The possible relationships between successively generated words depend on the random number generator in use, which is outside the scope of this discussion.

Notice in the printout that alongside each word is the same word divided into syllables (hyphenated). An interesting by-product of the algorithm is the ability to determine syllable divisions in the word generated. In certain cases the syllable split can not be precisely defined. For example, the word "without" can be split as "with-out" or "wi-thout" according to the rules. In such cases the random word generator makes an arbitrary (but predetermined) choice of where to split the syllables.

It may also be that certain hyphenations are not the most logical as an aid to pronunciation. An example can again be found in "without", which would be hyphenated as "wit-hout" if the "t" and "h" were generated as individual units instead of as a "th" unit. The decisions about hyphenation made by the program are built into the algorithm and are based on what the author considers the most likely to be acceptable in the general case.
SECTION IV

ANALYSIS

Of the two requirements of the random word generator stated on page 3, the requirement of making the words "difficult to guess" was stated as being easy to achieve by giving the word generator the capability of generating a very large set of words. The more difficult requirement of pronounceability guided the design, and it was intuitively assumed that a large enough set of words to satisfy some criterion of randomness would automatically result.

In this section an attempt is made to present some quantitative measurements and statistics that may allow one to determine whether the word generator is actually "random enough" in some sense. With a tool as crucial to the security of the system as a password generator, it must be assured that the passwords really are "difficult to guess".

There is no one quantity to be calculated that will provide a meaningful description of the random word generator's effectiveness in all its possible applications. For example, in an application where the random words are used to create identifiers of individuals, the total number of possible words and the probability of duplication are of interest. In the application as a Multics password generator, duplicates are not as important as the probability that a given user's password will be guessed by another user. For other applications the probability distribution of the words might be required.

It is hoped that enough statistical data is provided in this section so that, with sufficient further analysis, most quantities of interest can be calculated. A complete statistical analysis is beyond the scope of this discussion. However, attention will be focused on areas of interest to users of the random word generator as a password generator for Multics.

---

5In Multics, it is of little value to know a password without knowing the name of the user to whom it belongs; i.e., one cannot login to the system merely by typing a password and thereby impersonating whoever that password happened to belong to. Other systems may actually use the password to identify rather than verify.
The following four topics have been chosen for consideration:

- total number of different pronounceable words,
- probability of a given word being generated,
- most probable word, and
- distribution of word probabilities.

Some quantitative measure of each of the above has been obtained, but through empirical analysis of the random word generator's output rather than through an analysis of the algorithm. Analysis of the algorithm would of course yield the most precise statistics, but the complexity of the algorithm and its states, and the large amount of data in the supporting tables (which might be subject to change by anyone), make such an analysis extremely difficult and somewhat limited in applicability. Instead, minor modifications to the random word generator and some additional programs were incorporated to supply the data necessary for this analysis. If a change is made in one of the supporting tables, new data can be obtained merely by re-running these additional programs.

It should be noted that all of the statistics and numerical figures presented in this section apply only when the tables are set up as in Appendix I. The methods used to obtain the results, however, apply to any tables the user may supply.

NUMBER OF WORDS

The number of possible random words, though extremely difficult to determine by analyzing the algorithm, can be established to any degree of accuracy in a fairly simple manner.

Consider all possible words of a given length L that can be formed from the 26 letters of the alphabet, without regard to pronounceability. If N is the number of such words, then

\[ N = 26^L. \]  

(1)

Out of these, a certain fraction f are "pronounceable" according to random word generator rules. The value of f may, of course, depend on L. If we can determine f, we can calculate the number of pronounceable words n of a given length simply by

\[ n = fN. \]  

(2)

An estimate for the value of f can be obtained by picking a random subset of size m out of the N words, and finding out what fraction of this subset is pronounceable. The larger the value of m, the
smaller the probable error we will have in our estimated value of f. Actually the accuracy of our estimate can be expressed in terms of a probability that its absolute error is less than a certain amount.

Generating a random subset of \( N \) words of length \( L \) is easy with a uniform random number generator. With a small modification\(^6\) the random word generator can be given a particular word, and will "run through its rules" to determine whether the word is legal (i.e., pronounceable). A sample run of 100,000 words of eight letters was made. The length of eight letters was chosen for this run because that is the maximum length of a user's password acceptable to Multics. There were 2653 acceptable words out of this run of 100,000, yielding an estimate for \( f \) of \( .02653 \). For eight letters,

\[
N = 26^8 = 208,827,064,576
\]

and the estimated value for \( n \) is

\[
.02653N = 5.540 \times 10^9.
\]

The accuracy of the estimate for \( f \) as determined above can be calculated as a confidence interval for \( f \). This confidence interval is written approximately, for large \( m \), in the form

\[
\left[ \frac{k}{m} \pm \frac{z}{\sqrt{\frac{k(1-k)}{m}}} \right]
\]

where \( k \) is the number of acceptable words out of the sample of \( m \), and \( z \) is an appropriate percentage point of the standard normal distribution. For example, we might be interested in a 95% confidence interval, which corresponds to a value of \( z = 1.96 \). In the sample of 100,000 above, this yields

\[
[.02653 \pm .00099]
\]

For other confidence regions, and for sample runs of words of different lengths, see figure 2.

\(^6\)The only change is to use a special version of the random_unit subroutine (which is user-supplied) that supplies units of a known word rather than random units.
The words produced by the random word generator are not all equally probable for two reasons. First, different units have different probabilities of being generated by random_unit. Second, not all units thus generated are always acceptable. The probability of a given word must be calculated by examining the conditional probabilities of the individual units in that word.

Since random words are created left to right, at a given point during the creation of a word the units accepted so far determine which units may follow. Thus the probability of a particular unit appearing in a particular position of a word is the ratio of that unit's probability (of being returned by random_unit) to the total probability of all the units that are legal in that position. This calculation can be made for each unit based only on the units that precede it. In order to calculate the probability of a particular word, the probabilities of the individual units in that word are determined in this manner and then multiplied together.7

---

7The 100-try limit discussed on page 15 may cause entire words to be rejected even though some units were accepted. However, test runs have shown that the 100-try limit is almost never reached.

Figure 2. Number of Words of 6, 8 and 10 Letters

<table>
<thead>
<tr>
<th>word length</th>
<th>confidence</th>
<th>number of words</th>
<th>range</th>
<th>minimum</th>
<th>maximum</th>
</tr>
</thead>
<tbody>
<tr>
<td>6 letters</td>
<td>99.9%</td>
<td></td>
<td>1.745x10^7</td>
<td>1.896x10^7</td>
<td></td>
</tr>
<tr>
<td></td>
<td>99%</td>
<td></td>
<td>1.761x10^7</td>
<td>1.880x10^7</td>
<td></td>
</tr>
<tr>
<td></td>
<td>95%</td>
<td></td>
<td>1.770x10^7</td>
<td>1.867x10^7</td>
<td></td>
</tr>
<tr>
<td></td>
<td>90%</td>
<td></td>
<td>1.782x10^7</td>
<td>1.858x10^7</td>
<td></td>
</tr>
<tr>
<td>8 letters</td>
<td>99.9%</td>
<td></td>
<td>5.191x10^9</td>
<td>5.889x10^9</td>
<td></td>
</tr>
<tr>
<td></td>
<td>99%</td>
<td></td>
<td>5.269x10^9</td>
<td>5.812x10^9</td>
<td></td>
</tr>
<tr>
<td></td>
<td>95%</td>
<td></td>
<td>5.331x10^9</td>
<td>5.787x10^9</td>
<td></td>
</tr>
<tr>
<td></td>
<td>90%</td>
<td></td>
<td>5.363x10^9</td>
<td>5.714x10^9</td>
<td></td>
</tr>
<tr>
<td>10 letters</td>
<td>99.9%</td>
<td></td>
<td>1.464x10^12</td>
<td>1.794x10^12</td>
<td></td>
</tr>
<tr>
<td></td>
<td>99%</td>
<td></td>
<td>1.499x10^12</td>
<td>1.759x10^12</td>
<td></td>
</tr>
<tr>
<td></td>
<td>95%</td>
<td></td>
<td>1.535x10^12</td>
<td>1.735x10^12</td>
<td></td>
</tr>
<tr>
<td></td>
<td>90%</td>
<td></td>
<td>1.546x10^12</td>
<td>1.712x10^12</td>
<td></td>
</tr>
</tbody>
</table>
The method described above works because, for each position of the word, the random word generator keeps trying random units until a legal unit is found. The unacceptable units play no part in the probability that a particular legal unit will appear. For example, suppose in a given position of a particular word the only legal units are "e" and "a". If it is known that the probability of "e" appearing at random is 0.057, and the probability of "a" is 0.047, then the probability that the unit will be an "e" is 0.057/(0.057+0.047).

Since the random word generator does not throw out a unit once it has been accepted, it is merely necessary to multiply the individual conditional unit probabilities together to arrive at the probability of the word. Note that this probability only applies to words of a given length (i.e., the length of the word whose probability is calculated). The random word generator does not pick a length, but is asked to generate a word of a specified length. If random lengths are supplied to the random word generator, the distribution of these random lengths must be figured into the probability of the word calculated.

The special program described in the previous subsection that "gives" the random word generator known words was modified to calculate the probability of the known word in the manner described. The answer is exact in most cases, and the method will work regardless of the definition of the units, the tables, or the nature of the algorithm. The only restriction is that the word generator not discard units that have already been accepted in a given position of a word, and that the distribution of the units returned by random_unit remain constant during the formation of the word.

MOST PROBABLE WORD

The "most probable" word (or words) and its probability as determined in the above manner is meaningful to those interested in the difficulty of guessing random words. In the password application, for

Some words can be divided into units in one of two ways. For example, "w-i-t-h-o-u-t" and "w-i-th-o-u-t" are two ways of specifying the units of "without", both of which are legal. An exact calculation of the probability of this word would require adding the probabilities of both forms. In general, however, the probability of the version that contains more units (i.e., "w-i-t-h-o-u-t") is much lower because of the extra unit, and thus makes little difference in the total probability of the word. In calculating probabilities of words containing two-letter units that may possibly be split into two one-letter units the calculation is based on the word with the two-letter units.
example, it does not matter if there are one billion random words if the most probable word has a probability as high as 50% (even though the probability of all other words may be small). A systematic method for guessing a particular user's password would be to first try the most probable word and work down from there. If the first word tried has a high probability, a large set of legal words is of little value.

As important as this statistic is, it appears that only an extremely complex analysis will yield the most probable word. The obvious method of selecting only the most probable units to form a probable word does not work. For example, two of the most probable units are "e" and "t". One might expect that the most probable six letter word is something like "teetee". Actually, a word like "heehee" is almost twice as probable. A simple calculation can show that the first two units of a word are much more likely to be "he" than "te". Even though "t" has twice the probability of being first, the set of legal units following "t" is greater than the set of units following "h". It turns out that with the tables in use there are only six units that may follow "h", whereas there are eight that may follow "t". The probability that one of those six will be "e" is fairly high. The low probability of "h" multiplied by the high probability of "e" yields a value greater than the probability of the pair "te".9

An empirical approach to arriving at the most probable word might be to generate a large number of random words and to calculate their probabilities. Unfortunately, even the most probable word may have a probability sufficiently low so that millions of words might be generated before the most probable word appears. Moreover, one would have no assurance that any particular word really is the most probable.

Once more, intuition and a "feeling" of the rules and restrictions of the algorithm were relied upon. The utility programs previously described made it easy to try many expected high-probability words manually. In this way, a guess of the highest probability words of 6, 7, 8 and 10 letters has been made. The words are listed below, along with their probabilities. There may actually be several words of each length with the same probability. The results below only apply if the specific digram and unit tables listed in Appendix I are used, and if the distribution of the units is as listed in that appendix.

---

9 One should also consider that the use of two-letter units increases the probability of certain words. The six-letter word "quequo" is an order of magnitude more probable that "teetee", because it actually only contains four units (qu-e-qu-o), even though the probability of "qu" coming from the random unit is very low.
The probabilities above only apply to words of the specific length shown. For example, if the word generator is asked to generate words of a random length of 6, 7 or 8 letters, and each length is equally likely, then the probabilities above are multiplied by 1/3. Of course, the probability of the six letter word is so high that the other two words are of little interest if six letter words are allowed.

**DISTRIBUTION OF PROBABILITIES**

The ability to calculate the probability of a given word, and the total number of words allows us to arrive at an approximate distribution of the probabilities of the pronounceable words, from most probable to least probable. This distribution yields a kind of profile of the word generator that may be the best overall measure of the word generator’s effectiveness. One method of arriving at such a distribution is outlined below. As with the number of words, the accuracy of the distribution curve depends on the size of the sample of random words used.

Assume that all n pronounceable words of length L are listed in order of probability, and that a "word number" x, \( x \) running from 1 to n, is assigned to each word, where \( x = 1 \) for the most probable word. Let \( p(x) \) be the probability of word \( x \). If we had all \( n \) words, we could plot \( x \) against \( p(x) \) as in figure 3 to obtain a series of points. The distribution \( p(x) \) will be loosely referred to as a "curve" although strictly it is not a continuous function. Of course, \( p(x) \) is monotonically non-increasing by definition. The area under...

---

\(^{10}\)The letter "x" has been chosen for the word number instead of the more obvious choice of "i" to represent an integer in order to be more consistent with the notation generally used for some of the calculations in the following pages that treat \( x \) as a continuous variable.
the curve is unity, or more precisely

$$\sum_{x=1}^{n} p(x) = 1. \quad (7)$$

Once the curve is obtained, quantities like the total probability of the m most probable words, the probability of duplicates within a certain number of tries, etc., can be calculated or measured from a graph of the curve.

Figure 3. Distribution of Probabilities of Random Words

**Determination of Distribution**

If all n pronounceable words (and probabilities) were available, producing p(x) exactly would be no problem. In reality, we can only obtain a certain fraction of the n words. If we could in some way select every millionth word in the ordered list of n words, we could
still estimate a curve of \( p(x) \) by merely plotting every millionth point in figure 3 and interpolating to get the values in between. The accuracy of such a plot will depend on the "smoothness" of \( p(x) \) in some sense (and of course on the method used to make the interpolation).

There is no direct way to arrive at every millionth word in the list. We can generate \( k \) random words but we have no way of knowing what their positions are in the list (i.e., their values of \( x \)). In fact, if we generate \( k \) random words, their values of \( x \) will not be evenly distributed in the interval \([1, n]\), but will be weighted towards the lower end since the words of higher probability are more likely to appear at random. It is possible, however, to pick a random subset of the \( n \) words that is evenly distributed in the interval.

The uniform random word generator discussed near the top of page 21 can be used to provide a large enough set of equally likely random words so that the desired number \( k \) of these will be pronounceable. The \( k \) words thus obtained can be assumed to be equally spaced in the interval \([1, n]\) because they were arrived at in a manner totally independent of their probabilities. That is to say, the least probable of the \( k \) words has just as high a probability of appearing (using the uniform generator) as does the most probable word.

An approximate graph of \( p(x) \) was obtained by taking the 2653 pronounceable words used to estimate the value of \( n \) in (4) and ordering them according to probability. Each word was assigned an index \( i \),

\[
i = 1, 2, \ldots, k, \quad (8)
\]

where \( i = 1 \) for the most probable of these words. For each word, the position on the \( x \)-axis was determined by

\[
x(i) = \frac{i}{k + 1} \quad (9)
\]

A plot of the probabilities of the 2653 words is shown in figure 4.

**Application of the Distribution**

Figure 4 is a complete profile of the word generator and it can be used to measure various quantities. For example, the total probability of the \( m \) most probable words is simply the area under the curve from \( x = 0 \) to \( x = m \). The number of words that make up any given fraction of the population can also easily be measured.

Remember that in figure 4 the value of \( x \) is actually the "word
Figure 4. Distribution of Probabilities of 2653 Eight Letter Words
number" where \( x = 1 \) for the most probable word and \( x = n \) for the least probable. The value of \( x(1) \) for \( i = 1 \) in our sample of 2653 pronounceable words has a value of approximately 2,000,000 as calculated by (9). The most probable eight letter word out of the entire population is of course at \( x = 1 \). If we can believe for a moment that figure 4 is an exact representation, we can enlarge the extreme leftmost end of the curve where \( x \) is small as in figure 5, and extrapolate to the left of the point at \( x = 2,000,000 \) to double check the determination of the most probable word on page 25. Of course this extrapolation is not mathematically valid since there is no sound basis for assuming that the curve continues in any specific pattern. However, it does appear that extrapolation yields a value of the most probable word very close to that obtained by trial and error.

Another check on the distribution curve can be made by measuring the area under the curve. In order to approximately calculate this area, Simpson's rule was used where the first point (at \( x = 1 \)) was assumed to be the most probable word as previously determined, and successive points are at intervals of \( n/2654 \). The area thus calculated came out to 1.006, only 0.6% off the expected value of 1.000.

Figures 4 and 5 apply only to a specific sample run for eight letter words. Appendix V presents similar data for six and ten letter words as a comparison. Of course, a different digram table or unit table would greatly change these distributions.

AN ALTERNATIVE METHOD

The main difficulty in the analysis of the random word generator lies in the complexity of the algorithm. The nature of the algorithm is such that a highly asymmetric distribution of probabilities of words results, with some words being many orders of magnitude more probable than others. The goal of the preceding analysis was to provide information as to the shape of the probability distribution curve so that the word generator's suitability for any particular application could be examined.

In its application as a Multics password generator, the results of figure 4 may indicate that the word generator is not suitable for passwords due to the high probability of the words at the leftmost end of the curve. Some installations may need passwords that have a probability less than \( 2.19 \times 10^{-8} \). It is possible to improve this probability by changing the digram and unit tables and the distribution of the units returned by the \texttt{random_unit_} subroutine, but it is very difficult to anticipate the effect of any particular change on the probability distribution. Once the change is made in the tables, there is no easy way to determine what the most probable word actually is.
Figure 5. Enlarged Left Edge of Distribution
There is an alternative method, however, that can be employed without changing any of the tables, that yields a distribution that is much easier to determine.

In order to illustrate this alternative method, assume that we wish to improve the word generator's distribution so that no word is more probable than a word composed of six random letters. This example was chosen because the six random letter criterion for passwords is applied to several systems in use today. This criterion translates to a maximum probability of

\[
\frac{1}{26^6} = 3.24 \times 10^{-9}
\]  

(10)

for any word. The most probable eight letter word shown on page 25 has a probability of \(2.19 \times 10^{-8}\) -- too high by a factor of seven, and Multics does not allow (nor would it be desirable considering the rememberability requirement) passwords longer than eight characters. Note, however, that the total number of pronounceable eight letter words from equation (4) is much greater than the total number of random six letter words. Thus, if there were some way to force the word generator to generate all \(n\) pronounceable words with equal probability, then the probability of any particular word would be \(1/n\) and the analysis would be trivial.

By utilizing the random word generator in a slightly different way, at an additional cost in overhead, it is possible to force the probabilities of all words to be equal without changing the total number of words. Consider the method used to obtain the estimated value of \(n\) in equation (4). This value was obtained by generating words at random such that all eight letter words are equally likely, and computing the fraction of those that were pronounceable. A method for generating equally probable pronounceable words, then, is to generate equally likely random words and test them for pronounceability until an acceptable word turns up. All acceptable words thus generated are equally probable, and the randomness criterion is satisfied. Appendix VII contains the source code and the documentation for the two program modules that have been altered in order to optionally produce uniformly distributed random words. In addition, for those interested, a listing of the Multics encipher subroutine is included. This is the subroutine used to generate random numbers.

The question that arises is whether this "sampling" method is feasible considering the possible additional computer time required for testing and rejecting words. The answer depends on the fraction of words that are accepted, and whether it is more or less expensive.
to test a word for pronounceability rather than to generate it.

The fraction of pronounceable words was determined on page 21 for eight letter words using specific digram and unit tables. The value .02653 is an acceptance ratio of approximately 1 in 37. Thus we would expect, on the average, to make 37 tries before getting an acceptable word.

The time required to generate pronounceable eight letter words with the word generator is somewhat greater than the time required to generate a word at random and to test its pronounceability. This is because the word generator does not accept every unit supplied to it by random_unit in the word being formed, whereas all units of a pronounceable word to be tested are immediately accepted. The time required to test an unpronounceable word is usually less than the time required to test a pronounceable word because the whole word is rejected as soon as an illegal unit is encountered. If we expect, on the average, to make 37 tries to find a pronounceable word, we would expect the time required to do this to be no more than 37 times that required to generate one pronounceable word. This is born out by empirical evidence indicating that the average time required to find a pronounceable eight letter word by trial and error is about 10 times that required to generate one pronounceable word. In Multics computer time, the figures are about .10 second and .01 second per word respectively.

Since the number of pronounceable words is much greater (by a factor of 18) than that required to fulfill the six random letter criterion, it may be possible to significantly increase the pronounceability of words generated by modifying the tables to yield a smaller set of possible words. Consider the possible results if, for example, one could delete 17 out of every 18 words in the list in Appendix IV and save only the most pronounceable ones. Of course, by restricting the rules further, the "acceptance ratio" of 1 in 37 is decreased, thereby resulting in additional time spent finding a pronounceable word. A decrease by a factor of 18 will increase the average time of .10 seconds for finding a pronounceable eight letter word to several seconds. This may not be tolerable for some installations.

Thus, particularly when employing the alternative sampling method for generating pronounceable words, one must weigh the advantages of

---

11 The time of .10 second for the sampling method was obtained by using a modified version of the random word generator that discards a partial word and starts over any time a unit is rejected. In this way extra (possibly unused) units are not generated as in the case when a whole random word is first created before testing.
pronounceability against overhead in computer time in determining what modifications to make to the tables. The advantage of using the sampling method is that the only statistical quantity to be determined, the total number of words, is fairly easily estimated. The effect of a change in the digram table on this estimate can be quickly examined.
SECTION V

CONCLUSION

EVALUATION

The random word generator described in this report has been successfully implemented and demonstrated on the Multics system. The list of 2000 random words contained in Appendix IV was shown to various people, and it became apparent that the degree to which words are considered pronounceable varies a great deal among individuals. Some people had many more complaints than others about particular words. In most cases, complaints were about words containing certain combinations of units that the individual did not consider to be "legal".

For the most part, combinations considered illegal could be directly eliminated using the rules of the digram table. In other much less frequent cases, eliminating the offending construct was either impossible or would result in eliminating many more constructs that should be legal. As mentioned earlier, there were many fewer complaints about the shorter words of four to six letters than about the longer ones.

The statistics discussed in Section IV and Appendix V, if presenting an unsatisfactory performance for a particular application, can be improved by modifying the digram table and unit table or by altering the manner in which the random word generator is utilized as discussed near the end of Section IV. For example, by eliminating all double-letter units, the most probable word will have a much lower probability than that indicated on page 25. This is done, however, at the cost of a reduction in the total number of different words. Or, at an increase in overhead, the alternative method for generating words discussed on page 29 can be employed. By modifying the rules in the digram table, particular statistical properties can be adjusted, but one must be aware of the interrelationships between the various properties and performance features before changing the tables to achieve a given result.

In conclusion it appears that the tables that are input to the word generator and the manner in which it is used could be tailored for almost any application, whether the main interest is in pronounceability or performance.

40
OTHER APPLICATIONS

Pronounceability and randomness were the primary goals of the word generator in its use for generating random words. However, the support program discussed in Section IV that gives the word generator a word to be tried, combined with the word generator's ability to divide a word into syllables, partially supports the facility of a general purpose word hyphenator for text processing. The random word generator can be given any word for hyphenation. If the word is accepted, the word will be returned hyphenated into syllables just as if it had been randomly generated. If rejected, the word is illegal according to random word generator rules and the tables. Of course, the hyphenation will not always be the "right" one according to the dictionary, but exceptions are apt to be few.\(^1\) Perhaps some pre- or post-processing, combined with a list or dictionary of exceptional constructs, can yield a word hyphenator of general utility.

\(^{1}\)An example of a common exception is found in words containing "tion", which is hyphenated "ti-on" as two syllables. This and many other problems could be avoided by defining new kinds of double letter units and changing the tables. Such modifications may make the word generator unsuitable for generating random words because many unpronounceable words may be considered legal. However, when used as a hyphenator the legality of a word is of no concern.
APPENDIX I

TABLES

The following pages list the unit table and digram table as described in Section III. The unit table appears first. Each entry in the unit table has the following format:

\[ n \, cc \, wxyz \]

where:

- **n** is a unit number (1 to 34).
- **cc** is the unit (1 or 2 letters).
- **w** is 0 or 1, representing the value of "not_begin_syllable".
- **x** is the value of "no_final_split".
- **y** is "vowel".
- **z** is "alternate_vowel".

The digram table follows, with each entry of the following format:

\[ abd-cc+ef \]

where:

- **a** is 0 or 1, which is the value of "begin".
- **b** is "not_begin".
- **d** is "break".
- **-** appears if "prefix" is set, otherwise blank.
- **cc** is a pair of units (2, 3, or 4 letters).
- **+** appears if "illegal_pair" is set. If it is "-", that means "suffix" is set. Otherwise it is blank.
- **e** is "end".
- **f** is "not_end".

43
<p>| | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>a</td>
<td>0010</td>
<td>10</td>
<td>j</td>
</tr>
<tr>
<td>2</td>
<td>b</td>
<td>0000</td>
<td>11</td>
<td>k</td>
</tr>
<tr>
<td>3</td>
<td>c</td>
<td>0000</td>
<td>12</td>
<td>l</td>
</tr>
<tr>
<td>4</td>
<td>d</td>
<td>0000</td>
<td>13</td>
<td>m</td>
</tr>
<tr>
<td>5</td>
<td>e</td>
<td>0110</td>
<td>14</td>
<td>n</td>
</tr>
<tr>
<td>6</td>
<td>f</td>
<td>0000</td>
<td>15</td>
<td>o</td>
</tr>
<tr>
<td>7</td>
<td>g</td>
<td>0000</td>
<td>16</td>
<td>p</td>
</tr>
<tr>
<td>8</td>
<td>h</td>
<td>0000</td>
<td>17</td>
<td>r</td>
</tr>
<tr>
<td>9</td>
<td>i</td>
<td>0010</td>
<td>18</td>
<td>s</td>
</tr>
<tr>
<td>10</td>
<td>t</td>
<td>0000</td>
<td>19</td>
<td>u</td>
</tr>
<tr>
<td>20</td>
<td>v</td>
<td>0000</td>
<td>21</td>
<td>w</td>
</tr>
<tr>
<td>22</td>
<td>x</td>
<td>0000</td>
<td>23</td>
<td>y</td>
</tr>
<tr>
<td>24</td>
<td>z</td>
<td>0000</td>
<td>25</td>
<td>gh</td>
</tr>
<tr>
<td>26</td>
<td>ch</td>
<td>0000</td>
<td>27</td>
<td>wh</td>
</tr>
<tr>
<td>28</td>
<td>ph</td>
<td>0000</td>
<td>29</td>
<td>rh</td>
</tr>
<tr>
<td>30</td>
<td>sh</td>
<td>0000</td>
<td>31</td>
<td>th</td>
</tr>
<tr>
<td>32</td>
<td>wh</td>
<td>0000</td>
<td>33</td>
<td>qu</td>
</tr>
<tr>
<td>34</td>
<td>ck</td>
<td>1000</td>
<td>35</td>
<td></td>
</tr>
</tbody>
</table>

The Unit Table
| 011 phqu 01 | 100 shm -01 | 011 thgh 01 | 000 qug +00 | 011 ckv 01 |
| 000 phck+00 | 100 shn -01 | 011 thph 01 | 000 qun +00 | 011 ckw 01 |
| 100 rha 01 | 000 sho 00 | 000 thrh+00 | 000 qui 00 | 000 ckx +00 |
| 000 rhb +00 | 010 shp 00 | 011 thsh 01 | 000 quj +00 | 010 cky 00 |
| 000 rhc +00 | 100 shr -01 | 000 thth+00 | 000 quk +00 | 011 ckz 01 |
| 000 rhd +00 | 011 shs 01 | 000 thwh+00 | 000 quil +00 | 011 ckch 01 |
| 100 rhe 01 | 000 sht -00 | 011 thqu 01 | 000 qum +00 | 011 ckgh 01 |
| 000 rhf +00 | 000 shu 00 | 000 thck+00 | 000 quv +00 | 011 ckw+00 |
| 000 rhg +00 | 011 shv 01 | 100 wha 01 | 000 quv +00 | 000 ckrh+00 |
| 000 rhh +00 | 000 shw -01 | 000 whb +00 | 000 quv +00 | 011 cksh 01 |
| 100 rhi 01 | 000 shx +00 | 000 whc +00 | 000 quv +00 | 011 ckth 01 |
| 000 rhj +00 | 000 shy 00 | 000 whd +00 | 000 quv +00 | 000 ckwh+00 |
| 000 rhk +00 | 011 shz 01 | 100 whe 01 | 000 quv +00 | 011 ckq 01 |
| 000 rhl +00 | 011 shch 01 | 000 whf +00 | 000 quv +00 | 000 ckck+00 |
| 000 rhm +00 | 011 shgh 01 | 000 whg +00 | 000 quv +00 | 000 ckrh+00 |
| 000 rhn +00 | 011 shph 01 | 000 whh +00 | 000 quw +00 | 011 ckp 01 |
| 100 rho 01 | 000 shrh+00 | 100 whi 01 | 000 qux +00 | 011 ckf 01 |
| 000 rhp +00 | 000 shsh+00 | 000 whj +00 | 000 qux +00 | 011 ckx 01 |
| 000 rrh +00 | 011 shth 01 | 000 whk +00 | 000 qux +00 | 011 ckn 01 |
| 000 rhs +00 | 000 shwh+00 | 000 whl +00 | 000 qux +00 | 011 cke 01 |
| 000 rht +00 | 011 shqu 01 | 000 whm +00 | 000 qug +00 | 000 ckrk+00 |
| 100 rhf 01 | 000 shc 00 | 000 whn +00 | 000 qug +00 | 011 ck 01 |
| 000 rhv +00 | 000 shd 00 | 100 who 01 | 000 quh +00 | 011 cks 01 |
| 000 rhw +00 | 011 shb 01 | 000 whp +00 | 000 quh +00 | 000 cks 01 |
| 000 rhx +00 | 011 shc 01 | 000 whr +00 | 000 quh +00 | 011 ckt 01 |
| 100 rhy 00 | 011 sho 00 | 000 whs +00 | 000 quh +00 | 000 cky 01 |
| 000 rhz +00 | 011 she 00 | 000 whz +00 | 000 quh +00 | 011 ckh 01 |
| 000 rch +00 | 011 shf 01 | 000 whu +00 | 000 quh +00 | 011 ckw 01 |
| 000 rhch+00 | 011 shg 01 | 000 whv +00 | 011 cka 01 | 011 ckh 01 |
| 000 rhgh+00 | 011 shh 01 | 000 whw +00 | 011 ckb 01 | 011 cks 01 |
| 000 rhph+00 | 011 shi 00 | 000 whx +00 | 011 ckc 01 | 011 ckw 01 |
| 000 rhhr+00 | 011 shj 01 | 100 why 00 | 011 cke 01 | 011 cky 01 |
| 000 rhsh+00 | 011 shk 01 | 000 whz +00 | 011 cke 01 | 011 cky 01 |
| 000 rhth+00 | 011 shl 01 | 000 whch+00 | 011 ckk 01 | 011 cky 01 |
| 000 rhwh+00 | 011 shm 01 | 000 whgh+00 | 011 ckk 01 | 011 cky 01 |
| 000 rhqu+00 | 011 shn 01 | 000 whhp+00 | 011 ckk 01 | 011 cky 01 |
| 000 rhck+00 | 011 sho 00 | 000 whrh+00 | 011 ckl 01 | 011 cky 01 |
| 000 rsha 00 | 011 shp 01 | 000 whsh+00 | 011 ckl 01 | 011 cky 01 |
| 000 sha 00 | 000 shq 00 | 000 whth+00 | 011 ckl 01 | 011 cky 01 |
| 011 shb 01 | 011 shc 01 | 000 whu+00 | 011 ckl 01 | 011 cky 01 |
| 011 she 00 | 011 shf 01 | 000 whv+00 | 011 ckl 01 | 011 cky 01 |
| 011 shg 01 | 011 shh 01 | 000 whq+00 | 011 ckl 01 | 011 cky 01 |
| 011 shj 01 | 011 shi 00 | 000 whr+00 | 011 ckl 01 | 011 cky 01 |
| 011 shk 00 | 011 shj 01 | 000 whs+00 | 011 ckl 01 | 011 cky 01 |
| 010 shl -01 | 011 shk 01 | 000 wht+00 | 011 ckl 01 | 011 cky 01 |
| 100 shl -01 | 011 shh 01 | 000 whv+00 | 011 ckl 01 | 011 cky 01 |
APPENDIX II
RANDOM WORD ALGORITHM

This appendix lists the algorithm described on page 15. Below is a description of the notation and the variables used to describe a state.

State Variables

Each loop through the algorithm produces new values of several state variables and possibly adds a unit to the random word being formed. The variables used to describe the state are defined as follows. "Binary" variables may have the value "true" or "false"; "decimal" variables have a number as their value.

vowel_found       Set when a vowel is found in a syllable (binary).
last_vowel_found  Value of vowel_found for previous unit in the random word (binary).
syllable_length   Number of units in syllable so far (decimal, initially 1).
index             Number of units in word so far (decimal, initially 1).
cons_count        Number of consecutive consonants (decimal).
nchars            Number of letters in word to be generated. Initially this is set to the length of the word (in letters) desired. This value is decremented each time a two-letter unit is generated so that the number of units (index) can be compared to nchars to determine if the end of the word has been reached.

unit(1), unit(2), ... unit(index)  Unit(i) represents the i'th unit in the word. Unit(index) is the current unit.

In addition to the state variables, two variables are defined for use only internal to the algorithm. They are used to simplify the notation.
A binary variable which is set when the unit just generated is a vowel (or an alternate_vowel to be treated as a vowel).

A binary variable which gets set when a "break" pair (as defined on page 12) is encountered, or when the previous pair was a "suffix" pair and the current unit is not a vowel.

Notation

The following names are used in the algorithm for the eight flags in the digram table:

begin,
not_begin,
end,
not_end,
break,
prefix,
suffix,
and illegal_pair.

If one of these names appears with a value in parentheses immediately following it, as "break(i)", the reference is to the "break" flag for the pair of units [unit(i-1), unit(i)]. If no value appears, the reference is to the pair [unit(index-1), unit(index)] -- that is, the reference is to the last two units.

The following names are used for the flags in the unit table:

no_final_split,
not_begin_syllable,
vowel,
alternate_vowel,
and double_letter.

The "double_letter" flag was not explicitly mentioned in the discussion earlier. It is set for units consisting of more than one letter. A value in parentheses following the name, as in vowel(i), refers to the vowel flag for unit(i). If no value appears the reference is to the flag for unit(index), or the current unit.

Three procedures are referred to: "random_unit" is a user-supplied procedure to generate a random unit; "random_vowel" is a user supplied procedure to generate a random vowel unit; and "done" is an internal procedure appearing near the end of the algorithm.
Algorithm

The algorithm is shown in the following pages. The text of the algorithm is essentially the same as the main body of the random_word subroutine appearing in Appendix III. The algorithm is shown here only for completeness -- it can stand alone and does not depend on any support subroutines. Since the random_word subroutine as shown in Appendix III is well documented and commented, no comments are supplied below. A correspondence between this algorithm and the random_word subroutine can easily be made.

The RANDOM_WORD procedure has an internal procedure DONE listed near the end. The extents of the if-then-else clauses are indicated by indentation.
BEGIN procedure RANDOM_WORD

retry:
  if syllable_length = 1
  then
    if index = nchars
    then call random_vowel
    else call random_unit
    if (index ≠ 1 & illegal_pair)
    then go to retry
    syllable_length = 2
  if vowel ! alternate_vowel
  then cons_count = 0
  else cons_count = 1
  last_vowel_found = 0
  if double_letter
  then
    if index = nchars | (index = nchars-1 & ~vowel)
    then go to retry
    else nchars = nchars - 1
  else
    if (syllable_length = 2 & ~vowel_found & index = nchars) | (~vowel_found | not_end(index-1)) & suffix(index-1)
    then call random_vowel
    else call random_unit
    if illegal_pair |
      (unit(index)=unit(index-1)=unit(index-2) & index>2)
    then go to retry
    if double_letter & index = nchars
    then goto retry
    else nchars = nchars - 1
  if vowel ! (alternate_vowel & ~vowel(index-1))
  then v = 1
  else v = 0
  if syllable_length > 2 & suffix(index-1) & ~v
  then b = 1
  else b = break
  if syllable_length = 2 & not_begin
  then go to no_good
  if vowel_found
  then
    if cons_count ≠ 0
    then
      if begin
      then

53
RANDOM WORD ALGORITHM
------------------------ (continued)

if syllable_length ≠ 3 & not_end(index-2)
    then
        if not_end(index-1)
            then go to no_good
            else call done(v,2)
        else call done(v,3)
    else
        if not_begin
            then
                if b
                    then
                        if not_end(index-1)
                            then go to no_good
                            else call done(v,2)
                        else
                            if v
                                then
                                    if not_end(index-1) | not_begin_syllable
                                        then go to no_good
                                        else call done(1,2)
                                    else call done("end,end")
                                else call done(1,0)
                            else call done(1,3)
                    else call done("end,end")
                else
call done(1,0)
        else
            if v & vowel(index-2) & index > 2
                then go to no_good
            else
                if end
                    then call done(0,1)
                else
if begin
then
if last_vowel_found
then
if v
then
if syllable_length = 3
then
if alternate_vowel(index-2)
then go to no_good
else call done(1,3)
else
if not_end(index-2)
then go to no_good
else call done(1,3)
else
if syllable_length = 3
then
if alternate_vowel(index-2)
then call done(1,3)
else go to no_good
else
if not_end(index-2)
then
if not_end(index-1)
then go to no_good
else call done(0,2)
else call done(1,3)
else
if not_end(index-1) & syllable_length > 2
then go to no_good
else call done(v,2)
else
if b
then
if not_end(index-1) & syllable_length > 2
then go to no_good
else call done(v,2)
else call done(1,0)
else
if b
then go to no_good
else
if end
then

55
if \( v \) then call done(0,1) else go to no_good
else
  if \( v \) then
    if begin & syllable_length > 2 then go to no_good
    else call done(1,0)
  else
    if begin then
      if syllable_length > 2 then go to no_good
      else call done(0,3)
    else call done(0,0)
  go to retry

no_good:
  if double_letter then nchars = nchars + 1
  go to retry

BEGIN procedure DONE:
called with 2 arguments: call done(vf,sl)

  if sl ≠ 2 & syllable_length ≠ 2 & prefix & ^vowel(index-2) then
    if vowel_found then
      if not_end(index-1) then go to no_good
      else call done(0,2)
      return
    else go to no_good
  else
    if sl ≠ 1 & index = nchars & (not_end | vf = 0) then go to no_good
    if index = nchars & no_final_split & sl ≠ 1 & ^vowel(index-1) then
      if ^vowel_found | not_end(index-1) | syllable_length < 3 then go to no_good
      else sl = 0
    if \( v \) \& sl = 1 then cons_count = 0
else
  if si = 0
    then cons_count = cons_count + 1
    else cons_count = min(syllable_length-1, cons_count+1)
  if si = 0
    then syllable_length = syllable_length + 1
    else syllable_length = si
  if syllable_length > 3
    then last_vowel_found = vowel_found
    else last_vowel_found = 0
  vowel_found = vf
end

END procedure DONE
END procedure RANDOM_WORD
APPENDIX III
SOURCE CODE

The following pages contain source program listings of every command and subroutine applicable to the random word generator. The listings are in alphabetical order by program name. Below is a list of the programs with a brief description of the function of each. Complete documentation of the usage of each of these may be found in Appendix VI.

In Multics there is a distinction between commands and subroutines. Commands are callable by the user from his terminal, while subroutines can only be called by other subroutines or commands. The naming convention for programs specifies that subroutine names end with a trailing underscore, while command names should not. Generally the commands described below are user interfaces to specific subroutines.

columns
Prints lists of random words in columns.

convert_word_
Subroutine that converts an array of unit numbers to characters. Used by generate_word_.

convert_word_char_
Subroutine that inserts hyphens into a given word and formats the word for printing. Used by hyphen_test.

digram_table Compiler for digram table.

generate_word_
Standard interface for user-written programs to generate a random word. Used by generate_words.

generate_words
Command to generate a list of random words.

get_line_length
Command and subroutine to return the line length of the output medium for the purposes of formatting output. Used by columns and digram_table Compiler. Also called get_line_length_.

hyphen_test
Command to hyphenate a supplied word using the rules of the random word generator. Also used to calculate the probability of a given word.
<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>hyphenate_</td>
<td>Subroutine interface to perform same function as hyphen_test. Used by hyphen_test.</td>
</tr>
<tr>
<td>random_unit_</td>
<td>Standard subroutine for generating a random unit. Referenced by generate_word_ and hyphenate_ and called by random_word_.</td>
</tr>
<tr>
<td>random_unit_stat_</td>
<td>Assembly language program containing definitions of external static variables used by random_unit_.</td>
</tr>
<tr>
<td>random_word_</td>
<td>Subroutine implementing the word generator. Called by generate_word_ and hyphenate_.</td>
</tr>
<tr>
<td>read_table_</td>
<td>Subroutine that compiles the digram table. This is an internal interface only called by digram_table_compiler.</td>
</tr>
</tbody>
</table>
COMPILATION LISTING OF SEGMENT columns
Compiled by: Experimental PL/I Compiler of Tuesday, March 25, 1975 at 14:19
Compiled on: 04/01/75 1720.7 est Tue
Options: check source

1 /* This program takes a segment consisting of ASCII lines
2 and rearranges them so that they may be printed out in
3 columns on a page, reading top to bottom in each column.
4 To call, enter command:
5 columns seg -option-
6 1 seg is the pathname of the segment to reformat.
7 2 option is one or more of the following
8 -sm, -segment specifies that output will be put in the segment
9 named seg, columns and can later be printed.
10 Default sends output to terminal.
11 -line_length nn is a number from 1 to 132 that specifies the length of
12 the output line. If missing, the defaults are:
13 output to terminal -- length of terminal line
14 output to segment -- 132
15 output via file_output -- 132
16 -no_pagination, -npq specifies that the page length is assumed
17 to be infinite, and no page skips occur.
18 In this case each columns reads all the way
down to the bottom of the printout.
19 -page_length nn page length to use, rather than the default of 60.
20 -pl nn
21 -adjust, -ad fill in extra blank space in between columns.
22 */
23
do: columns: procedure;
36 dcl ev_dec_check_entry(char(*), fixed bin) returns(fixed bin(35));
37 dcl return_code(fixed bin);
38 dcl total_length(fixed bin); /* length of each line in output segment, which may include padding at end of last column */
39 dcl length_wanted(fixed bin); /* output segment line length minus padding after newlines */
40 dcl last_line(fixed bin); /* line number in output of last line on current page */
41 dcl get_line_length_entry returns(fixed bin);
42 dcl status_bit(12) aligned;
43 dcl gmode(128) init('128'" = ");
44 dcl max_length(fixed bin init(0)); /* max real input line length plus padding if adjust is on */
45 dcl real_max_length(fixed bin); /* max input line length not counting newlines */
46 dcl last_fixed bin; /* number of output lines to put on current output page */
47 dcl charcnt(fixed bin(35)) init(1);
48 dcl colcnt(fixed bin);
49 dcl linecnt(fixed bin);
50 dcl newline char(1) static init(" ");
51 *);
52 dcl newline_tab char(2) static init(" ");
53 *);dcl string char(132) varying;
54 dcl real_length(fixed bin);
56 do1 set_dir_entry returns(char(168) aligned);  
57 do1 li fixed bin init(0);  
58 do1 (s, input_seq_ptr, scratch_seq_ptr init(null, output_seq_ptr) ptr;  
59 do1 total_output_lines fixed bin;  
60 do1 total_lines fixed bin; /* number of lines in input segment */  
61 do1 stop switch bit(4) static init(0);  
62 do1 (com_err, lons) fixed bin; /* number of output lines per page */  
63 do1 (io_write_ptr entry (ptr, fixed bin, fixed bin);  
64 do1 (code_ptr entry (fixed ptr, fixed bin(35));  
65 do1 segment switch bit(4) static init(0);  
66 do1 lines fixed;  
67 do1 page_length fixed init (60);  
68 do1 interval fixed bin; /* number of output lines per page of output */  
69 do1 arg_count fixed bin;  
70 do1 (char, fixed);  
71 do1 (bitcount, (i) fixed bin(24));  
72 do1 (nolumns fixed bin; /* number of columns on output */  
73 do1 null built-in;  
74 do1 arg_length fixed bin;  
75 do1 (n) fixed bin;  
76 do1 i fixed bin;  
77 do1 j fixed bin;  
78 do1 (l) fixed bin;  
79 do1 (char, ) based(p) unaligned;  
80 do1 code fixed bin(35) init(0);  
81 do1 (error_table, fixed bin(35));  
82 do1 (error_table, fixed bin(35)) ext fixed bin(35);  
83 do1 (char, ) entry (char, char, fixed bin(5), ptr,  
84 do1 fixed bin(35));  
85 do1 (ptr, fixed bin, fixed bin(35));  
86 do1 (ptr, fixed bin(24), fixed bin(35));  
87 do1 (ptr, fixed bin(35));  
88 do1 expand_char entry (ptr, fixed bin, ptr, ptr, fixed bin(35));  
89 do1 (dirname, char(168) aligned);  
90 do1 (name, char(32) aligned);  
91 do1 (char, ) entry (char, char, char, fixed bin(24),  
92 do1 fixed bin(2), ptr, fixed bin(35));  
93 do1 _directory_entry entry (ptr, fixed bin(35));  
94 do1 adjust bit(1) aligned init(0);  
95 do1 (lons) based(input_seq_ptr);  
96 do1 (total_output_lines, nolumns, char(168), ) based(scratch_seq_ptr);  
97 do1 (total_output_lines, char(168), ) based(scratch_seq_ptr);  
98 do1 (total_output_lines, char(168), ) based(scratch_seq_ptr);  
99 do1 (newlines, char(3));  
100 do1 static init((63)*101);  
102 do1 /* get pointer to segment to read */  
103 do1 (code_ptr entry (s, arg_length, code); /* 1st argument is segment name */  
104 do1 if code "0; then do;  
105 do1 call com_err; (code, "columns",  
106 do1 "Usage is: columns [options] -arg("  
107 do1 call expand_char (p, arg_length, addr(dirname), addr(ename), code);  
108 do1 if code "0 then call unix (code, args);  
109 do1 return;  
110 do1 end;  
111 do1 call expand_char (p, arg_length, addr(dirname), addr(ename), code);  

if input_seg_ptr = null then call ugly (code, before(dirname, "") || ": " || name);
if bitcount = 0 then call ugly (error_table_zzero_length, arg);
chars = call(bitcount/9); /* total number of characters in segment */

/* get rest of arguments */

then segment_switch = "mb";
else if arg = "--segment" ; arg = "--ma"
then segment_switch = "mb"
else if arg = "--line" ; arg = "--line_length" then do;
arg_count = arg_count + 1;
call cu_base_ptr (arg_count, p, arg_length, code);
if code = 0 then call ugly (code, "");
l = (cx_dec_check (arg, return_code));
if return_code = 0 : l <= 0 | l > 132
then call ugly (0, "bad line length. " || arg);
if ll = 0
then call ugly (error_table_silo_nonsense, arg);
ll = 1;
end;
else
if arg = "--pagenum" ; arg = "--no_pagination"
then page_length = 0;
else if arg = "--page_length" ; arg = "--pl" then do;
arg_count = arg_count + 1;
call cu_base_ptr (arg_count, p, arg_length, code);
if code = 0 then call ugly (code, "Length of page.");
page_length = (cx_dec_check (arg, return_code));
if return_code = 0 : page_length <= 0
then call ugly (0, "Bad page length. " || arg);
end;
else
if arg = "--adjust" ; arg = "--ad" then do;
adjust = "mb"
end;
else call ugly (error_table_sbadopt, arg);
end;

if ll = 0 /* line length not specified, get from user_output */
then
if segment_switch
then ll = 150;
else ll = get_line_length();
call hcs_make_seg (**, "columns_temp", ": 0", 0, scratch_seg_ptr, code); /* output will go here */
if scratch_seg_ptr = null then call ugly (code, "columns_temp in process directory");
/* Now that we have pointer to temporary segment, this stuff can be set */
if substr(lines, nchar, 1) "s newline /* This may cause problems if not checked */
then call ugly (0, "Last line in segment does not end in newline character");
/* find number of lines in segment and longest line */
do lines = 1 by 1 while(charcont=notchars);
/* we expect that most lines won't contain tabs in them, therefore
the code below avoids calling tab_expander if no tabs are in the line.
It takes advantage of the fact that searching for one of two characters
is no more expensive than searching for one. */
1 = search (substr (lines, charcont), newline_tab); /* look for newline or tab */
if substr (lines, charcont+1, 1) = newline then do; /* was it a newline? */
   real_length = 1; /* it was a newline, we have real length of line */
end;
else do;
   /* the line must have contained a tab */
   i = index (substr (lines, charcont), newline); /* find newline */
   real_length = length (tab_expander (substr (lines, charcont, i)));
end;
if real_length > 11 /* new line not found within 11 characters */
then
do;
   if max_length < 11 /* is this the first time we found a line too big? */
   then call loa_f1oa_stream ("error_output", "Line in segment is longer than 10 characters", 11);
   call loa_f1oa_stream ("error_output", "line ": " ", tab_expander (substr (lines, charcont, i+1)), nlines);
end;
charcont = charcont + 1;
max_length = max (max_length, real_length);
end;
if max_length > 11 then call utry (0, "error in input segment");
/* set some initial variables */
ncolumns = (11+1)/max_length; /* maximum number of columns that will fit */
real_max_length = max_length;
if adjust then do; /* try to put in extra padding */
   max_length now becomes length of line in each column plus padding
but the last line in each column has no padding, so its length is real_max_length */
max_length = max_length + (11 + 1 - max_length*ncolumns)/(ncolumns-1);
end;
linecont = 1;
total_lines = nlines - 1;
total_output_lines = (total_lines + ncolumns - 1)/ncolumns; /* number of lines in output segment */
if page_length = 0 then interval = total_output_lines;
else interval = page_length;
/* we are working on the output segment one page at a time. The variable i gets incremented
by the number of input lines (lineval) to be put on each output page. The input lines
are processed in order -- we start at the top of the leftmost column, inserting input lines,
and proceeding down that column. When reaching the bottom, we go to the top of the next
column. The very last column on the last page may have blanks ending it. */
do i = 1 to total_output_lines by interval;
   if page_length = 0
      then last = total_output_lines; /* number of lines on page */
   else last = min (page_length, total_output_lines - i + 1);
   last_line = i + last - 1; /* This forces columns on last page to be shorter so that minimum output lines are used */
   do color = 1 to ncolumns;
      /* work on one column at a time */
      do linecont = i to last_line;
      /* go down each column, inserting lines from input segment */
columns.list

```c
230  if nlines > total_lines
231      then output_file(linecnt, colcnt) = ";" /* all blanks if no more input lines */
232  else do;
233      j = search (substr (lines, charcnt), newline_tab);
234      if subst (lines, charcnt + j - 1, 1) = newline
235         then output_file(linecnt, colcnt) = substr (lines, charcnt + j - 1);
236      else output_file(linecnt, colcnt) = tab_expander (substr (lines, charcnt, index (substr (lines, charcnt), newline) - 1));
237      charcnt = charcnt + index (substr (lines, charcnt), newline);
238      nlines = nlines + 1;
239  end;
240  if colcnt == ncolumns
241      then subst(output_file(linecnt, colcnt), real_max_length) = newline;
242  end;
243  end;
244  end;
245  /* we can get rid of input segment */
246  call term_seen_ptr (input_seen_ptr, code);
247  /* if output is to segment, move to segment and set bit count */
248  total_length = max_length - ncolumns;
249  length_wanted = total_length - (max_length - real_max_length);
250  if segment_switch then do;
251      call hcs_make_seg (get_wr_ptr (), before (ename, "") [: "columns", ",", 01010b, output_seen_ptr, code);
252      call hcs_truncate_seg (output_seen_ptr, 0, code);
253      if code == 0 then exit output_seen_ptr_error;
254      en = 1; /* move segment, pass_length lines at a time, with formsfeed in between */
255      do i = 1 by interval to total_output_lines;
256         do linecnt = j to min (i + interval - 1, total_output_lines);
257            /* if adjust is on, this line may truncate extra spaces at end of each line (but the newline is in the right place) */
258            subst (whole_line, nn, length_wanted) = output_lines (linecnt);
259            nn = nn = length_wanted;
260      end;
261  if linecnt < total_output_lines then do;
262      subst (whole_line, nn, 1) = "\n";
```
"; /* insert formfeed */
206  en = en + 1;
207  end;
208  end;
209  call hca_set_bc_seg(output_seg_ptr, (nn-1)*9, code);
210  if code ^= 0 then goto output_seg_ptr_error;
211  call term_set_seg_ptr(output_seg_ptr, code);
212  if code ^= 0 then
213  output_seg_ptr_error:
214  call ugly (code, before(name, ") || ".columns");
215  end;
216  else do;
217  if page_length ^= 0
218  then call los_write_ptr (addr(newlines), 0, 3);
219  else do 1 : 1 to total_output_lines:
220  call los_write_ptr (scratch_seg_ptr, total_length*(i-1), length_wanted); /* write out the line */
221  if page_length ^= 0 then if mod1, page_length) = 0
222  then call los_write_ptr (addr(newlines), 0, 66 - page_length);
223  end;
224  if page_length ^= 0
225  then call los_write_ptr (addr(newlines), 0, 1);
226  else call los_write_ptr (addr(newlines), 0, 66 - mod1, page_length);
227  end;
228  /* enter here to clean up when finished */
229  terminate:
230  if scratch_seg_ptr ^= null then call hca_setentry_seg(scratch_seg_ptr, code);
231  /* routine to turn tabs into spaces */
232  tab_expander: proc(string) returns (char(*)) reducible;
233  dcl tab_char(1) init(" ") static;
234  dcl string char(*);
235  dcl tab_position fixed bin;
236  tab_position = index (string, tab);
237  if tab_position ^= 0
238  then
239   return (tab_expander(C
240   substr(string, 1, tab_position - 1) ||
241   substr((10)*", mod(tab_position - 1, 10) * 1) ||
242   substr(string, tab_position + 1));
243  else do:
244   return(string);
245  end;
246  end;
247  /* print an error message and terminate */
248  ugly: proc (code, message);
249  dcl code fixed bin(35);
250  dcl message char(*);
251  dcl error (code, ".columns", message);
252  goto terminate; /* nonlocal goto to finish up */
COMPILATION LISTING OF SEGMENT convert_word
Compiled by: Experimental PL/I Compiler of Tuesday, March 25, 1975 at 14:19
Compiled on: 04/01/75 1721.0 ed Tue
Options: check source
1 convert_word: proc (word, hyphens, word_length, expanded_word, hyphenated_word);
2 dcl word(0:*), fixed bin;
3 dcl hyphens(0:*), bit(1), aligned;
4 dcl word_length fixed bin;
5 dcl expanded_word char(*);
6 dcl hyphenated_word char(*);
7 dcl l fixed bin;
8 dcl no_hyphens bit(1), aligned;
9 dcl word_index fixed bin init(1);
10 dcl hyphenated_index fixed bin init(1);
11
12 /********** include file diagram_structure.incl.pl */
13
14 dcl digrams$digrams external;
15 dcl digrams$units fixed bin external;
16 dcl digrams$letters external;
17 dcl digrams$rules external;
18
19 /* This array contains information about all possible pairs of units */
20
dcl 1 digrams(n_units, n_units) based (addr(digrams$digrams)),
21
dcl 1 digrams$units(n_units) based (addr(digrams$units)),
22
dcl letters(0:n_units) char(2) aligned based (addr(digrams$letters)),
23
dcl letters_split(0:n_units) based (addr(digrams$letters)),
24
dcl 1 rules$units aligned based (addr(digrams$rules)),
25
" This array has rules for each unit */
26
dcl n_units defined digrams$units fixed bin;
27
8 /********** end include file diagram_structure.incl.pl */
12 13 no_hyphens = "b; 
14 15 convert_word: 
16 do i = 1 to word_length: 
17 if substr(letters(word(i)),2,1) = " " 
18 then 
19 do; 
20 substr(expanded_word, word_index, 1) = substr(letters(word(i)),1,1); 
21 if "no_hyphens then 
22 do; 
23 substr(hyphenated_word, hyphenated_index, 1) = substr(letters(word(i)),1,1); 
24 hyphenated_index = hyphenated_index + 1; 
25 end; 
26 word_index = word_index + 1; 
27 end; 
28 else
29 do; 
30 substr(expanded_word, word_index, 2) = letters(word(i)); 
31 if "no_hyphens then 
32 do; 
33 substr(hyphenated_word, hyphenated_index, 2) = letters(word(i)); 
34 hyphenated_index = hyphenated_index + 2; 
35 end; 
36 word_index = word_index + 2; 
37 end; 
38 if "no_hyphens 
39 then 
40 if hyphens(i) 
41 then 
42 do; 
43 substr(hyphenated_word, hyphenated_index, 1) = "-"; 
44 hyphenated_index = hyphenated_index + 1; 
45 end; 
46 end; 
47 if "no_hyphens then if hyphenated_index <= length(hyphenated_word) then substr(hyphenated_word, hyphenated_index) = "/"; 
48 if word_index <= length(expanded_word) then substr(expanded_word, word_index) = "/"; /* fill out with spaces */ 
49 return; 
50 convert_word$no_hyphens: entry (word, word_length, expanded_word); 
51 no_hyphens = "/"; 
52 goto convert_word; 
53 goto convert_word; 
54 end;
COMPILATION LISTING OF SEGMENT convert_word_char_
Compiled by: Experimental FL/I Compiler of Tuesday, March 29, 1975 at 14:19
Compiled on: 04/01/75  T21:1 edit Tue
Options: check source
1 convert_word_char_: proc (word, hyphens, last, result);
  2 dol i fixed bin;
  3 dcl result char[*] varying;
  4 dcl word char[*];
  5 dcl hyphens[*] bit(1) aligned;
  6 dcl last fixed bin;
  7 if last < 0
  8 then
  9 do;
 10   result = word || "***";
 11   return;
 12 end;
 13 result = "";
 14 do i = 0 to length(word);
 15 if i ≠ 0
 16 then
 17 do;
 18   result = result || substr(word,i,1);
 19 if hyphens[i] then result = result || "-";
 20 end;
 21 if last > 0 & last ≠ i
 22 then result = result || "**";
 23 end;
 24 end;
COMPILED LISTING OF SEGMENT digram_table_compiler
Compiled by: Experimental PL/I Compiler of Tuesday, March 25, 1975 at 14:19
Compiled on: 04/01/75 1721.1 edt Tue
Options: check source

1 /* This command compiles a source segment containing digrams for
2 the word generator and puts the compiled output in a segment
3 named "digrams".
4
5 Usage: digram_table_compiler pathname -option-
6
7 Where: option may be one of the following:
8
9 -ls, -list Lists the output on the terminal after compilation.
10 -ls, n, -list n Lists as above, but in n columns.
11
12 Usage: print_digram_table -n-
13
14 n Lists the output in n columns. Allow 14 positions for each column.
15 This call assumes that the digrams segment already exists
16 and has been compiled correctly.
17 */
18
19 digram_table_compiler: procedure;
20 del (start, size) fixed bin;
21 del rows fixed bin;
22 del c_unarg_ptr entry(fixed bin,ptr, fixed bin, fixed bin[35]);
23 del code fixed bin(35);
24 del code fixed bin;
25 del cv_dec_check_entry (char[*], fixed bin) returns (fixed bin[35]);
26 del hsz_truncate_dec entry (ptr, fixed bin, fixed bin[35]);
27 del hsz_terminate_name entry (char[*], fixed bin[35]);
28 del coe_err_entry options(variable);
29 del (error_table_encore, error_table_adopt) external fixed bin(35);
30 del get_line_length_entry returns (fixed bin);
31 del read_table_entry (ptr, fixed bin(24)) returns(bit(1));
32 del compile bit(1);
33 del who(char[25]) varying;
34 del list bit(1);
35 del segptr ptr static int(null);
36 del dname char(168) aligned;
37 del ename char(32) aligned;
38 del ename_length fixed bin;
39 del null builtin;
40 del arg char(length) based (pp);
41 del hsz_terminate_count entry (char[*], char[*], char[*],
42 fixed bin(24), fixed bin(2), ptr, fixed bin[35]);
43 del hsz_terminate_noname entry (ptr, fixed bin[35]);
44 del expand_path_entry (ptr, fixed bin, ptr, ptr, fixed bin[35]);
45 del be fixed bin(24);
46 del i fixed;
1 1
1 2 /********** include file digram_structure.incl.pli *********/
1 3
del digrams$digrams external;
1 4 del digrams$units fixed bin external;
1 5 del digrams$units external;
1 6 del digrams$letters external;
1 7 del digrams$rules external;
1 8
1 9 /* This array contains information about all possible pairs of units */
digram_table_compiler.list 04/01/75 1725.0 edit Tue

1
2 1 dcl 1 digrams(n_units, n_units) based (addr(digrams&diggrams)).
3 2 begin bit(1), /* on if this pair must begin syllable */
4 3 2 not_begin bit(1), /* on if this pair must not begin */
5 4 2 end bit(1), /* on if this pair must end syllable */
6 5 2 not_end bit(1), /* on if this pair must not end */
7 6 2 break bit(1), /* on if this pair is a break pair */
8 7 2 prefix bit(1), /* on if vowel must precede this pair in same syllable */
9 8 2 suffix bit(1), /* on if vowel must follow this pair in same syllable */
10 9 2 illegal_pair bit(1), /* on if this pair may not appear */
11 10 2 pad bit(1); /* this makes 9 bits/entry */
12 11 /* This array contains left justified 1 or 2-letter pairs representing each unit */
13 12 24 dcl letters(0:n_units) char(2) aligned based (addr(digrams&letters));
14 13 /* This is the same as letters, but allows reference to individual characters */
15 14 28 dcl 1 letters_split(0:n_units) based (addr(digrams&letters)),
16 15 29 2 first char(),
17 16 30 2 second char(),
18 17 31 2 pad char(2);
19 18 32 /* This array has rules for each unit */
20 19 34 35 dcl 1 rules(n_units) aligned based (addr(digrams&rules)),
21 20 36 2 no_final_split bit(1), /* can’t be the only vowel in last syllable */
22 21 37 2 not_begin_syllable bit(1), /* can’t begin a syllable */
23 22 38 2 vowel bit(1), /* this is a vowel */
24 23 39 2 alternate_vowel bit(1), /* this is an alternate vowel, (i.e., "y") */
25 24 40 41 dcl n_units defined digrams$n_units fixed bin;
26 25 42 43 //*********** end include file digram_structure.incl.pl1 ***********
27 44 45 dcl pp_ptr;
28 46 47 49 dcl [i, k] fixed;
29 50 51 dcl max fixed;
30 52 53 dcl length fixed bin;
31 54 55 dcl loc_options(entry options(variable));
32 56 57 dcl argno fixed bin;
33 58 59 dcl (diff, last, columns init(0), remainder, middle, first) fixed;
34 60 dcl loc_entry options(variable);
35 61 62 63 who " = "digram_table_compiler";
36 64 65 goto start1;
37 66 67 68 69 69 70 who " = "dtc";
38 71 72 73 74 75 76 77 78 79 goto start1;
80 81 82 83 84 compile = "Yb; /* set switch to compile */
85 86 call cu&action_ptr (1, pp, length, code);
87 88 argno = 1;
89 90 if code = 0 then goto argerr;
90 91 call expand_ath_(pp, length, addr(dimname), addr(ename), code);
91 92 if code = 0 then goto argerr;
digram_table_compiler.list

71 ename_length = index(ename, '*');
72 if ename_length = 0
73 then ename_length = 32;
74 else ename_length = ename_length - 1;
75 if ename_length >= 4
76 then
77 if substr(ename, ename_length - 4) = "dic"
78 then ename_length = ename_length - 4;
79
80 argno = 2;
81 call cu_arg_ptr (2, pp, length, code); /* get option */
82 if code = 0
83 then list = "0b"; /* no listing desired */
84 else
85 if arg = "-ls" | arg = "-list"
86 then do;
87 list = "r"b;
88 argno = 3;
89 end;
90 else do;
91 code = error_table$badopt;
92 goto argerr;
93 end;
94 goto get_ncolumns;
95
96 put: entry;
97 who = "put";
98 goto start2;
99
100 print_digram_table: entry;
101 who = "print_digram_table";
102
103 start2:
104
105 list = "f"b;
106 argno = 1;
107 compile = "0"b;
108
109 get_ncolumns:
110 call cu_arg_ptr (argno, pp, length, code);
111 if code = 0
112 then ncolumns = get_line_length(1)/14;
113 else do;
114 ncolumns = cr_dec_check_ (arg, codes);
115 if codes = 0
116 then do;
117 code = error_table$fbadopt;
118 goto argerr;
119 end;
120 end;
121
122 if "compile then goto dont_compile;
123 /* now initiate the source segment */
124
call hcs$initiate_count ((dirname), substr(ename, 1, ename_length) ||
126 "dir", "", bc, 0, segptr, code);
127 if segptr = null
then do;
  call diag_err_ (code, who, \"a\"\"a.dtc\", dirname, substr(ename, 1, ename_length));
  return;
end;

/* compile the segment */
call hcs terminated_name ("digrams", code); /* terminate previous copies */
if read_table (segptr, bc) /* any error */
  then
do;
call diag_err (0, who, \"Error in source segment.\")
  return;
end;
/* terminate the source now */
call hcs terminated_name (segptr, code);
if \"list\" then return; /* if no listing wanted, leave now */
dont_compile:
if compile then call loa_ (\"/\"\"/\"\"/\"");
nrows = (n_units-1)/n_columns + 1; /* This is the first reference to the digrams segment */
if n_columns\# 0
  then
do;
do i = 1 to nrows;
do j = 1 by nrows while (j < n_units);
call loa_cmd (\"2d\"\"a\"\"b\"\"b\"\"b\"\"b\"\", letters(j), rules(j).not_begin, rules(j).no_final_split, rules(j).vowel, rules(j).alternate_vowel);
end;
call loa_ (\"");
end;
call loa_ (**);
end;
call loa_ (**);
do start = 1, n_columns*59-nrows) + 1 by n_columns*60
while (start<n_units)*2;
  if start = 1
    then size = min(n_units, n_columns*59-nrows));
  else size = min(n_units, n_columns)+60);
diff = size/n_columns;
remainder = size - diff*n_columns;
lst = (size + n_columns - 1)/n_columns + start - 1;
do first + start to last;
middle = first + remainder+diff + 1);
if first + last + middle = first
  then max = middle - (diff+1));
else max = middle + (n_columns - remainder - 1)*diff;
do i = 1 to middle by diff while(i<max), middle+diff to max by diff;
j = (i-1)/n_units;
k = i - (i-1)/n_units;
call loa_cmd (\"b\"\"b\"\"b\"\"b\"\", charac(i) \"charac(j) \"letters(k) \"charac() \"b\"\"b\"\", diagrams[j,k].begin, diagrams[j,k].not_begin, diagrams[j,k].break, diagrams[j,k].end, diagrams(j,k).not_end);
end;
call loa_ (\"");
end;
if start = 1
  then call loa_cmd(copy("/",n6-firrst-nrows));
diagram_table_compiler.list

187    else call ioa annum(copy("#", start+66-first));
188    end;
189    end;
190    return;
191    char: proc returns(char(1));
192    if digram[j,k].prefix then return("-"); else return(" * ");
193    end;
194    char: proc returns (char(1));
195    if digram[j,k].illegal_pair
196    then return("*");
197    else
198    if digram[j,k].suffix
199    then return("~");
200    else return(" *");
201    end;
202    end;
203    end;
204    char: proc(c) returns(char(2));
205    dcl c fixed;
206    if letters_split(c).second = " 
207    then return(" * "); letters_split(c).first;
208    else return(letters(c));
209    end;
210    end;
211    argerr:
212    if code = erro_table$noarg
213    then call com_err_ (code, who);
214    else
215    do;
216    call ou$arg ptr (argno, pp, length, 0); 
217    call com_err_ (code, who, arg);
218    end;
219    end;
220    end;
/* This procedure is the subroutine interface to generate random words.
    It is called when the standard distribution of random units (as returned by
    random_unit) is desired. The clock value is used as the starting seed unless
    generate_word_init_seed is called.

    generate_word: procedure (word, hyphenated_word, min, max);
    dot word char(*);
    dot hyphenated_word char(*);
    dot min fixed bin;
    dot max fixed bin;
    dot (random_unit_.random_unit_.random_vowel) entry (fixed bin);
    dot convert_word_entry (fixed bin, (0:* bit(1) aligned,}
    fixed bin, char(*), char(*));
    dot random_word_entry (fixed bin, (0:* bit(1) aligned,
    fixed bin, fixed bin, entry, entry);
    dot hyphens (0:20) bit(1) aligned;
    dot random_word (0:20) fixed bin;
    dot length_in_units fixed bin;
    dot random_length fixed bin;
    dot unique_bits entry returns (bit(70));
    dot encipher_entry (fixed bin(7), (0:* bit(1), (0:* bit(1), (0:* bit(1), fixed bin(7)) external);
    dot first_call bit(1) static aligned init("b1"b);

    /* On the very first call to this procedure in a process (if the
       init_seed entry was not called), use unique_bits to get a
       random number to initialize the random seed. */
    if first_call then do:
        random_unit_stat_seed(1) = fixed (unique_bits ());
        first_call = "b1"b;
    end;

    /* Get the length of the word desired. We use the old value
       of the seed to determine this length so that the length of the word
       will not in some way be correlated with the word itself.
       We calculate this to be a uniformly distributed random number between
       min and max. */
    random_length = mod (abs (fixed (random_unit_stat_seed(1), 17)), (max - min + 1)) + min;
    /* encipher the seed to get a random number and the next value of the seed */
    call encipher (random_unit_stat_seed(1), random_unit_stat_seed, random_unit_stat_seed, 1);
    /* Get the random word and convert it to characters */
    call random_word (random_word, hyphens, random_length, length_in_units, random_unit, random_unit.random_vowel);
    call convert_word (random_word, hyphens, length_in_units, word, hyphenated_word);
    return;
    /* This entry allows the user to set the seed. If the seed argument is zero, we
    go back to using the clock value.
    */
    generate_word_init_seed: entry (seed);
generate_word_list

56 do1 seed fixed bin(35);
57
58 if seed = 0 then first_call = "1b";
59 else do;
60 random_unit_stat_seed(1) = seed;
61 first_call = "0b";
62 end;
63 return;
64 end;
generate_words.list

seeds = value("seed");
call generate_word_findseed (seeds);
end;
else do;
new_words = cv_dsec_check (arg, result); /* look for number of words */
if result = 0 & new_words > 0
then max_words = new_words;
else call ugly (error_table $badopt, arg);
end;
end;
/* Below we decide whether minimum, maximum, both, or none have been specified, */
and set their default values accordingly. */
if new_words = 0 then max_words = 1;
if minimum_length = -1
then if maximum_length = -1
then do;
minimum_length = 6;
maximum_length = 8;
end;
else minimum_length = 4;
else if maximum_length = -1
then maximum_length = 20;
if minimum_length < 4 : minimum_length > maximum_length :
maximum_length = 20 then
call ugly (0, "bad value of lengths: \{minimum(21 required)");
maximum_hyphenated = maximum_length + 2*maximum_length/3; /* maximum length of hyphenated word */
hp_ptr = addr (substr (area, maximum_length + 2)); /* where hyphenated word is put */
if hyphenate /* for efficiency, put newline character in expected place in output string */
then do;
substr (unhyphenated_word, maximum_length + 1, 1) = " ";
substr (hyphenated_word, maximum_hyphenated + 1, 1) = new_line;
output_line_length = maximum_length + maximum_hyphenated + 2;
end;
else do;
substr (unhyphenated_word, maximum_length + 1, 1) = new_line;
output_line_length = maximum_length + 1;
end;
/* generate max_words and write them all out */
do i = 1 to max_words;
call generate_word_unhyphenated_word (hyphenated_word, minimum_length, maximum_length);
call locs_write_ptr (addr(area), 0, output_line_length);
end;
ugly (procedure (codex, message));
cdl (code, codex) fixed bin(35);
cdl message char(*);
call com_err (codex, "generate_words", message);
goto return;
end;

generate_words.list

114 value: procedure (name) returns (fixed bin(35));
115 dol number fixed bin(35);
116 dol name char (*);
117 argno = argno + 1;
118 call cy_dec_check_ (arg, result);
119 if code = 0 then call ugly (code, "Value of " || name);
120 number = cy_dec_check_ (arg, result);
121 if result = 0 || number < 0
122 then call ugly (0, "Bad " || name || " value. " || arg);
123 return(number);
124 end;
125 return;
126 end;
COMPILATION LISTING OF SEGMENT get_line_length
Compiled by: Experimental PL/I Compiler of Tuesday, March 25, 1975 at 14:19
Compiled on: 04/01/75 1721.4 edit Tun
Options: check source

1 get_line_length: gll: pro;
2 dol com_err_entry options(variable);
3 dol loa_entry options(variable);
4 dol active_fnc_err_entry options(variable);
5 dol cu_ifact return_arg entry (fixed bin, ptr, fixed bin, fixed bin(35));
6 dol cu_arg_count entry (fixed bin);
7 dol return_string char(max_length) varying based (return_string_ptr);
8 dol code fixed bin(35);
9 dol error_table $not_act_fnc, error_table $wrong_no_of_args fixed bin(35) external;
10 dol error_table $nondesc fixed bin(35) external;
11 dol max_length fixed bin;
12 dol return_string_ptr ptr;
13 dol goode char(168);
14 dol narga fixed bin;
15 dol active_fnc fixed bin;
16 dol error_message(2) entry options(variable) variable init (com_err_, active_fnc_err_);
17 /* entry on active function or command call */
18 call cu_ifact return_arg (narga, return_string_ptr, max_length, code);
19 if code $ error_table $not_act_fnc
20 then /* not called as active function */
21 do;
22 active_fnc = 0; /* set flag */
23 call cu_arg_count (narga); /* if called as a command, see if any arguments are present */
24 end;
25 else active_fnc = 1; /* this was an active function call */
26 if code $ error_table $nondesc /* if no descriptors, assume we were called as a command */
27 then do: narga = 0; active_fnc = 0; end;
28 if narga $ 0
29 then do; /* arguments not allowed */
30 call error_message(active_fnc = 1) (error_table $wrong_no_of_args, "set_line_length");
31 return;
32 end;
33 if active_fnc = 0
34 then do;
35 call temp char(3); /* bug in compiler requires this statement */
36 temp = line_length();
37 call loa_ (temp);
38 else return_string = line_length();
39 return;
40 /* entry on subroutine call */
41 get_line_length_: entry return(fixed bin);
42 return (fixed(line_length(), 17));
43 /* internal procedure to get length of line */
44 line_length: procedure return (char(*));
45 dol loa $change mode entry (char(*), char(*), char(*), bit(72) aligned);
46 dol status bit(72) aligned;
47 dol $ expanded status based (addr(status)),

2 bits bit(36);
2 status_code fixed bin(35);
dcl numbers_index fixed bin;
dcl ll_index fixed bin;
call log_8rangle "user_output", ".", xmode, status);
if status_code "x 0
then return (*132*);
ll_index = index(xmode, ".11"); /* look for "11" in returned modes */
if ll_index = 0
then return (*132*);
else
if ll_index = 1
/* if "11" is not at beginning of string */
then /* then it should be preceded by ",." */
do;
ll_index = index(xmode, ",11");
if ll_index = 0
then return (*132*);
ll_index = ll_index + 1;
end;
nnumbers_index = verify (substr(xmode, ll_index * 2), "0123456789"); /* find end of number after "11" */
if numbers_index <= 1 l numbers_index > 4
then return (*132*);
/* this number must be 1-3 digits */
else return (substr(xmode, ll_index * 2, numbers_index = 1));
end;
end;
# Compilation Listing of Segment hyphen_test

Compiled by: Experimental FL/I Compiler of Tuesday, March 25, 1975 at 14:19

Compiled on: 04/01/75  IT21.5 ed  Tue

Options: check source

1 hyphen_test: hi proc;
2 dol cu_farg_ptr entry(fixed,ptr,fixed, fixed bin(35));
3 dol length fixed bin;
4 dol j fixed bin;
5 dol status fixed bin;
6 dol hyphenate_entry (char(*), (* bit(1) aligned, fixed bin);
7 dol hyphenate_probability_entry(char(*), (* bit(1) aligned, fixed bin, float bin);;
8 dol probability float bin;
9 dol hyphens(20) bit(1) aligned;
10 dol loc_entry options(variable);=
11 dol arg char(length) based(argptr);=
12 dol argptr ptr;
13 dol code fixed bin(35);
14 dol l fixed bin;
15 dol convert_word_char_entry(char(*), (* bit(1) aligned, fixed bin, char(*) varying);=
16 dol result char(30) varying;
17 dol calculate bit(1) aligned init("0b");
18
dol l = 1 by 1;
19 call cu_farg_ptr (1, argptr, length, code);
20 if code = 0 then return;
21 if arg = "-probability" | arg = "-pb" then calculate = "1b";
22 else do;
23 if calculate
24 then call hyphenate_probability(arg, hyphens, status, probability);
25 else call hyphenate_(arg, hyphens, status);
26 call convert_word_char_(arg, hyphens, status, result);
27 if calculate
28 then call loc_("a" "f", result, probability);
29 else call loc_(result);
30 end;
31 end;
32 end;
33 end;
This procedure tries to hyphenate a word supplied by the caller.

call hyphenate_(word, returned_hyphens, code);

1) word A word consisting of ASCII letters to be hyphenated.
2) The first character may be uppercase or lowercase; the other
3) characters may be lowercase only.

3) returned_hyphens
4) A one bit in this array means that the corresponding
5) character in word is to have a hyphen after it.

3) code
4) Status code: 0 word has been successfully hyphenated.
5) -1 word contains illegal characters.
6) -2 word was more than 20 or less than 3 characters.
7) Any positive value of code means the word couldn't
8) be hyphenated. In this case code is the position of the
9) first character that was not accepted.

The word is hyphenated by using random_word and whatever existing digram
1) table is in use by random_word to determine the syllabification and pronounceability
2) of the word supplied. The standard random_unit routine is not used,
3) except that random_unit_probabilities is called by hyphenate_probability.

hyphenate_: procedure(word, returned_hyphens, code);
dol word char[4];
dol code fixed bin;
dol debug dual bit(1) init("D00");
dol loc_fnl entry optional(1) fixed;
dol word_array[20] fixed bin static; /* word spread out into units */
dol hyphenated_word[20] bit(1) aligned; /* returned from random_word */
dol returned_hyphens[20] bit(1) aligned; /* hyphens to be returned to caller */
dol split_point fixed bin; /* set on internal call to 2-letter unit to be split */
dol word_length_in_chars fixed bin static; /* length of word in characters */
dol word_length_in_units fixed bin static; /* length of word_array in units */
dol l fixed bin;
dol j fixed bin;
dol char char(1);
dol word_index fixed bin static; /* index into word_array */
dol returned_word[20] fixed bin; /* word returned by random_word */
dol vowel_flag bit(1); /* 1 when random_vowel is called */
dol last_good_unit fixed bin; /* word_index of last good unit */
dol new_unit fixed bin;
dol random_word_entry (0:1) fixed bin, (0:1) bit(1) aligned,
fixed bin, fixed bin, entry, entry);
dol random_unit_probabilities entry (0:1) fixed bin, (0:1) bit(1) aligned,
fixed bin, fixed bin, entry, entry);
dol calculate bit(1) static; /* says we're calculating the probability of a word */
dol p float bin static; /* accumulated product of probability for the word */
dol total_p_this_unit float bin; /* total sum of probabilities of units that could be accepted in this position */
dol returned_length fixed bin;
hyphenate_list

56 /* probabilities of generating each unit at random */
57 /* obtained from a call to random_unit_probabilities */
59
dcl {unit_probabilities based (u_p_ptr), vowel_probabilities based (v_p_ptr)} dim (n_units) float bin;

dcl (u_p_ptr, v_p_ptr) static ptr init(null());

dcl first_call static bit(1) init(true);

1 2 /********* include file digram_structure.inc.pl **********/
3
4 dcl digrams$digrams external;
5 dcl digrams$n_units fixed bin external;
6 dcl digrams$letters external;
7 dcl digrams$rules external;
8
9 /* This array contains information about all possible pairs of units */
10
11 dcl 1 digrams(n_units, n_units) based (addr(digrams$digrams)),
12 2 begin bit(1), /* on if this pair must begin syllable */
13 2 not_begin bit(1), /* on if this pair must not begin */
14 2 end bit(1), /* on if this pair must end syllable */
15 2 not_end bit(1), /* on if this pair must not end */
16 2 break bit(1), /* on if this pair is a break pair */
17 2 prefix bit(1), /* on if vowel must precede this pair in same syllable */
18 2 suffix bit(1), /* on if vowel must follow this pair in same syllable */
19 2 illegal_pair bit(1), /* on if this pair may not appear */
20 2 pad bit(1); /* this makes 9 bits/entry */
21
22 /* This array contains left justified 1 or 2-letter pairs representing each unit */
23
24 dcl letters(0:n_units) char(2) aligned based (addr(digrams$letters));
25
26 /* This is the same as letters, but allows reference to individual characters */
27
28 dcl 1 letters_split(0:n_units) based (addr(digrams$letters)),
29 2 first char(1),
30 2 second char(1),
31 2 pad char(1);
32
33 /* This array has rules for each unit */
34
35 dcl 1 rules(0:n_units) aligned based (addr(digrams$rules)),
36 2 no_final_split bit(1), /* can't be the only vowel in last syllable */
37 2 not_begin_syllable bit(1), /* can't begin a syllable */
38 2 vowel bit(1), /* this is a vowel */
39 2 alternate_vowel bit(1); /* this is an alternate vowel, (i.e., "y") */
40
41 dcl n_units defined digrams$n_units fixed bin;
42
43 /********* end include file digram_structure.inc.pl ***********/
44
45 split_point = 0;
46 calculate = "0b"; /* we aren't calculating probabilities, just hyphenating */
47 goto continue;
48
49
/*
68 /* This entry is the same as hyphenate_, except that an additional value returned
69 is the probability that the word would have been generated by random_word
70 using the current digram table and random_unit_subroutine_. On the first call
71 to this entry, random_unit_probabilities is called to obtain the probabilities
72 of all units. If these change within a process, hyphenate_freset must be called
73 before hyphenate_#probability is called again.
74 */
75
76 hyphenate_#probability: entry (word, returned_hyphens, code, probability);
77 split_point = 0;
78 p = 1;
79 calculate = "t"b;
80 if first_call then do;
81 allocate unit_probabilities, vowel_probabilities;
82 call random_unit_probabilities (unit_probabilities, vowel_probabilities);
83 first_call = "0"b;
84 end;
85 goto continue;
86
87 /* This entry is used to reset the probability arrays in case a new
88 version of random_unit_ (with different probabilities) is used.
89 Note that if a new version of digrams is also supplied, the old
90 digrams must be terminated. */
91
92 hyphenate_freset: entry; first_call = "1"b;
93 if v_p_ptr "x null" then free unit_probabilities, vowel_probabilities;
94 return;
95
96 /* This entry point is called internally as a recursive call to hyphenate_.
97 It is referenced when random_word_ did not accept the word because a 2-letter unit
98 was illegal. In this case we call this entry and tell hyphenate_ to split the 2-letter
99 unit into 2 separate units. The splitpoint argument specifies which one to do this with. */
100
101 hyphenate_#split: entry (word, returned_hyphens, code, splitpoint);
102 doi splitpoint fixed bin;
103 split_point = splitpoint;
104
105 continue:
106 word_length_in_chars = length(word);
107 if word_length_in_chars > 20 || word_length_in_chars < 3
108 then
109 do;
110 code = -2;
111 if calculate then probability = 0;
112 return;
113 end;
114
115 /* Now that we have the word we want to hyphenate, we try to divide it up into units as defined
116 in the digram table. We start with the first two letters in the word, and see if they are equal to any
117 of the 2-letter units. If they are, we store the index of that unit in the word_array, and increment
118 our word_index by 2. If they are not, we see if the first letter is equal to any of the 1-letter units.
119 If it is, we store that unit and increment the word_index by 1. If still not found, the character is
120 not defined as a unit in the digram table and the word is illegal. Of course, the word may still not be
121 "legal" according to random_word_ rules of pronunciation and the digram table, but we'll find that out
122 later. */
123 */
hyphenate_list

125 word_index = 1;
126 do i = 1 to word_length_in_chars;
127  chars = substr(word, 1, min(2, word_length_in_chars - i + 1));
128  if i = 1 then substr(chars,1,1) = translate(substr(chars,1,1),"abcdefgijklmnopqrstuvwxyz","ABCDEFGHIJKLMNOPQRSTUVWXYZ");
129  j = 1;
130  do j = 1 to n_units while(chars"letters(j)); /* look for 2-letter unit match */
131 end;
132  if j < n_units & word_index "split_point
133 then /* match found */
134 do
135    word_array(word_index) = j; /* store 2-letter unit index */
136    word_index = word_index + 1;
137    j = j + 1; /* skip over next unit */
138 end;
139 else
140   /* two-letter unit not found, search for 1-letter unit */
141   char = substr(chars,1,1);
142   if i = 1 then char = translate(char,"abcdefgijklmnopqrstuvwxyz","ABCDEFGHIJKLMNOPQRSTUVWXYZ");
143   char = substr(char,1,1);
144   do j = 1 to n_units while(char"letters(j));
145 end;
146 if j < n_units
147 then /* match found */
148 do
149    word_array(word_index) = j; /* store 1-letter unit index */
150    word_index = word_index + 1;
151 end;
152 else /* not found, unit is illegal */
153   code = -1;
154   if calculate then probability = 0;
155 return;
156 end;
157 end;
158 end;
159 end;
160 word_length = word_index - 1;
161 word_index = 0;
162 /* Now call random_word..., trying to get the word hyphenated. Special versions of random_unit and
163 random_vowel are supplied that return units of the word we are trying to hyphenate rather than
164 random units.
165 */
166 call random_word( returned_word, hyphenated_word, word_length_in_chars, returned_length, random_unit, random_vowel);
167 goto accepted;
168 /*
169 If random_unit ever finds that random_word... did not accept a unit from the word to be hyphenated,
170 a nonlocal goto directly to this label (which pops random_word... off the stack) is made, and we
171 abort the whole operation. If the last unit tried (i.e., the one not accepted) was a 2-letter unit,
172 we might be able to make the word legal by splitting that unit up into two 1-letter units and
173 starting all over. Unfortunately, this is a lot of code and complication for a relatively rare case.
174 */
175 not_accepted: word_index = word_index - 1; /* index of last unit accepted */
176 p = 0; /* zero probability if word was not accepted */
177}
hyphenate_list

183 accepted; if debug then if calculate then call laa_fnn1 (*/*);
184 j = 1;
185 returned_hyphens = "0";
186 do l = 1 to word_length;
187 if l > word_index & word_index < word_length /* we never got done with the word */
188 then
189 do;
190 code = j; /* word was not accepted */
191 if letters_splitword_array(i).second = "a" /* was it not accepted because of an illegal */
192 if split_point = 0 /* 2-letter unit? */
194 p = 1;
195 call hyphenate_dsplit (word, returned_hyphens, code, 1); /* try again with split pair */
196 end; /* Note: In even rarer cases, the unit that might be split to make this word legal is not the
197 unit that was rejected, but a previous unit. It's too hard to deal with this case, so we'll refuse the word,
198 even though it might be legal. As an example, using the standard dgram table, "preag-hu-o" is a legal word.
199 However, our first attempt was to supply p-r-e-x-h-u-o units. Random_word_rejects the
200 "u" because it may not follow a "sh" unit in this context. Since "u" is not a 2-letter
201 unit, we can't try to split it up, so the word is thrown out. However, p-r-e-x-h-u-o
202 would have been acceptable to random_word_. This is the only case where a
203 word that could have been produced by random_word__ will be rejected by hyphenate__ */
204 if calculate then probability = p;
205 return; /* otherwise, return */
206 end;
207 /* set returned_hyphens bits corresponding to character in word. Note that
208 hyphens returned from random_word_ (hyphenated_word array) point to units,
209 not characters. */
211
212 if letters_split(word_array(i).second = "a"
213 then j = j + 2;
214 else j = j + 1;
215 returned_hyphens(j-1) = hyphenated_word(i);
216 end;
217 code = 0;
218 if calculate then probability = p;
219 return;
220 /*
hypenate_list
04/01/75 1729.2 edr Tue

/*
 * The internal procedures random_unit and random_vowel keep track of the acceptance or rejection of
 * units they are supplying to random_word_. Most of the code in the first part is to calculate probabilities
 * when hypenate_probability is called.
 */

random_vowel: proc (returned_unit);
  dl: returned_unit fixed in;
  vowel_flag = "";
  goto generate;
random_unit: entry (returned_unit);
  vowel_flag = "O";
generate:
  if at this point, we either calculate probabilities or just go for another unit */
  if probabilities are being calculated, we proceed as follows:
  In every position of the word, we send off to random_word_. all possible units except the one
  that is actually in the word. We send these as negative numbers so that random_word_ will not actually use
  them, but will tell us whether they are legal. Since we know the probabilities of all units, the
  total of the probabilities of the acceptable units can be calculated and normalized to 1 in order
  to determine the probability of the unit we are actually trying. For example, if "s" is the only legal
  unit in a given position of the word, then its probability of appearing in that position is 1, since
  random_word_ will not accept anything else.
  When all units but the actual unit have been tried, we send off the actual unit with a positive sign.
  It should be accepted by random_word_ if the word is legal, and the ratio of its probability
  to the total probability of the legal units is the probability of the unit being in this word position.
  This multiplied by the product of these probabilities of the previous units gives us a "running product"
  that will eventually yield the probability of the whole word.
/

if calculate then do; /* we are calculating */
if debug then
  if returned_unit < 0 then
    call log_6uni ("a", letters(-returned_unit));
  if returned_unit > 0 then do; /* this is the first unit of the word */
    total_p_this_unit = 0; /* initialize probabilities */
    word_index = 1;
  end;
else if returned_unit > 0 then do; /* it tried to start a word all over on us */
  new_unit = word_array(word_index); /* get the current unit from the word */
  if returned_unit > 0 then do; /* was the last unit accepted? */
    if returned_unit > new_unit then do; /* yes, was it the one from this word position? */
      total_p_this_unit = 0; /* initialize for next word position */
      word_index = word_index + 1;
      new_unit = word_array(word_index); /* get next unit from word, which now becomes current unit */
    end;
    returned_unit = 0;
  end;
else do; /* unit just accepted was not the one at this word position */
  if vowel_flag /* add its probability to total for this position and keep trying more units */
    then total_p_this_unit + total_p_this_unit + vowel_probabilities(-returned_unit);
  else total_p_this_unit + total_p_this_unit + unit_probabilities(-returned_unit);
  end;
end;
if -returned_unit + new_unit then goto not_accepted; /* current unit was not accepted */
COMPILATION LISTING OF SEGMENT random_unit
Compiled by: Experimental P/L Compiler of Tuesday, March 25, 1975 at 14:19
Compiled on: 04/01/75 1721.7 edf Tue
Options: check source

/* This is the standard random unit generating routine for random_word_.
 It is specified in the call to random_word_ by generate_word_.
 It does not reference the digram table, but assumes that it contains
 34 units in a certain order. This routine attempts to return
 unit indexes with a distribution approaching that of the distribution
 of the 34 units in English. In order to do this, a random number
 (supposedly uniformly distributed as returned from encipher_)
 is used to do a table lookup into an array containing unit indexes.
 There are 211 entries in the array for the random_unit_entry point.
 The probability of a particular unit being generated is equal to the
 fraction of those 211 entries that contain that unit index. For example,
 the letter "a" is unit number 1. Since unit index 1 appears 10 times
 in the array, the probability of selecting an "a" is 10/211.
 Changes may be made to the digram table without affect to this procedure
 providing the letter-to-number correspondence of the units does
 not change. Likewise, the distribution of the 34 units may be altered
 (and the array size may be changed) in this procedure without affecting
 the digram table or any other programs using the random_word_subroutine.
*/

random_unit_entry: procedure (number);
    do numbers (0-210) fixed static init ((101),(82),(123),(124),(125),(126),
        (87),(68),(109),(810),(811),(612),(613),(1114),(1015),(616),
        (1017),(818),(1019),(620),(621),(622),(823),(824),(825),
        (626),(627),(628),(629),(630),(334);
    do vowel_numbers (0-11) fixed static init(1,1,5,5,5,9,9,15,15,20,20,24);
    do encipher_entry (fixed bin(71), (#) fixed bin(71), fixed bin);
    do random_unit_stat &seed(1) external fixed bin(71);
    do number fixed bin;
    call encipher_ (random_unit_stat &seed(1), random_unit_stat &seed, random_unit_stat &seed, 1);
    number = numbers (mod (abs (fixed (random_unit_stat &seed(1), 171)), 211));
    return;
random_vowel: entry (number);
    call encipher_ (random_unit_stat &seed(1), random_unit_stat &seed, random_unit_stat &seed, 1);
    number = vowel_numbers (mod (abs (fixed (random_unit_stat &seed(1), 171)), 123);
    return;

/* This entry returns the probabilities of the 34 units in two arrays.
The first array contains the probabilities of all units assuming
the random_unit_entry was called. The second array contains the
probabilities of all units assuming random_vowel was called.
Of course, there will be a lot of zeros in this second array, since
most units aren't vowels.
This entry is used by hyphenate_setuprobability to find out what the
probabilities of the different units are. Hyphenate_does not know
how many units are what their probabilities are. It also
makes no assumption about the unit index - to - letter correspondence
of the units. Thus this program can be replaced without changing
anything in hyphenate_.
*/

probabilities: entry (unit_probs, vowel_probs);
56     float bin;
57     float bin;
58     float bin;
59     unit_probs, vowel_probs = 0;
60     unit_probs, vowel_probs = 0;
61     /* These probabilities are calculated merely by adding up the number of
62        occurrences of each of the unit indexes in the numbers array and the
63        vowel_numbers array. */
64     vowel_numbers array. */
65     do i = 0 to 210;
66     unit_probs = unit_probs(numbers(i)) * 1;
67     if i < 12
68     then vowel_probs = vowel_probs (vowel_numbers(i)) * 1;
69     end;
70     end;
71     unit_probs = unit_probs/211; /* Normalize these values so they add up to 1.0 */
72     vowel_probs = vowel_probs/12;
73     return;
74     end;
ASSEMBLY LISTING OF SEGMENT wudidcpg\word_gen\random_unit_stat.alm
ASSEMBLED ON: 04/01/75 1725.6 ed Tux
OPTIONS USED: la symbols new_call new_object
ASSEMBLED BT: ALM Version 4.5, September 1974
ASSEMBLER CREATED: 02/24/75 1625.7 ed Mon

000000
000010 000010
000010 aa 012345 676543
000011 aa 123456 765432

name random_unit_stat_
use linkc
join /link/linkc
segdef seed
even
seed: oct 012345676543,123456765432
end

NO LITERALS
**COMPILATION LISTING OF SEGMENT random_word**

Compiled by: Experimental PL/I Compiler of Tuesday, March 25, 1975 at 14:19

Compiled on: 04/01/75  1721.7  ctc Tue

Options: check source

1  /* This procedure generates a pronounceable random word of
2   caller specified length and returns the
3   word and the hyphenated (divided into syllables) form of the word.
4
5   dl random_word_entry (0:* fixed, (0:* bit(1) aligned, fixed, fixed, entry, entry);
6   call random_word_ (word, hyphens, length, n, random_unit, random_vowel);
7   word
8   random word, 1 unit per array element. (Output)
9
10  hyphens
11  position of hyphens, bit 0 indicates hyphen appears after
12  corresponding unit in "word". (Input)
13
14  length
15  length of word to be generated in letters. (Input)
16
17  n
18  actual length of word in units. (Output)
19
20  random_unit
21  routine to be called to generate a random unit. (Input)
22
23  random_vowel
24  routine to be called to generate a random vowel. (Input)
25
26  */

27  random_word_: procedure(password,hyphenated_word,length,word_length,random_unit,random_vowel);

28  /******** include file digram_structure.incl.pl; ******* /

29  1 4 dl digrams$digrams external;
30  5 dl digrams$n_units fixed bin external;
31  6 dl digrams$letters external;
32  7 dl digrams$rules external;
33
34  /* This array contains information about all possible pairs of units */
35
36  11 dl 1 digrams$n_units, n_units) based (addr(digrams$digrams)),
37
38  2 begin bit(1),  /* on if this pair must begin syllable */
39  3 not_begin bit(1),  /* on if this pair must not begin */
40  4 end bit(1),  /* on if this pair must end syllable */
41  5 not_end bit(1),  /* on if this pair must not end */
42  6 break bit(1),  /* on if this pair is a break pair */
43  7 split bit(1),  /* on if vowel must precede this pair in same syllable */
44  8 suffix bit(1),  /* on if vowel must follow this pair in same syllable */
45  9 illegal_pair bit(1),  /* on if this pair may not appear */
46  10 pad bit(1);  /* this makes 9 bits/entry */
47
48  /* This array contains left justified 1 or 2-letter pairs representing each unit */
49
50  24 dl letters(0:n_units) char(2) aligned based (addr(digrams$letters));
51
52  /* This is the same as letters, but allows reference to individual characters */
53
54  28 dl 1 letters_split(0:n_units) based (addr(digrams$letters)),
55
56  2 first char(1),
57  2 second char(1),
random_word_list

04/01/75  1729.4  edit Tue

page 2

1  31
2  32
3  33 /* This array has rules for each unit */
4  34
dcl rules(n_units) aligned based (addr(digrams$rules)),
5  35 2 no_final_split bit(1), /* can't be the only vowel in last syllable */
6  36 2 not_begin_syllable bit(1), /* can't begin a syllable */
7  37 2 vowel bit(1), /* this is a vowel */
8  38 2 alternate_vowel bit(1); /* this is an alternate vowel, (i.e., "y") */
9  39
dcl n_units defined digrams$units fixed bin;
10  40
11  41 /*** and include file digram_structure.incl.pl: Different */
12  42
dcl debug bit(1) aligned static init("0"b); /* set for printout of words that can't be generated */
13  43 dcl password(0:* fixed bin;
14  44 dcl hypenanted_word(0:* fixed bit(1) aligned;
15  45 dcl length fixed bin;
16  46 dcl word_length fixed bin;
17  47 dcl number fixed bin(27);
18  48 dcl nchars fixed; /* number of characters in password */
19  49 dcl index fixed init(1); /* index of current unit in password */
20  50 dcl i fixed;
21  51 dcl syllable_length fixed init(1); /* 1 when next unit is 1st in syllable, 2 if 2nd, etc. */
22  52 dcl conson_count fixed init(0); /* count of consecutive consonants in syllable preceding current unit */
23  53 dcl vowel_found aligned init(0); /* 1 if vowel was found in syllable before this unit */
24  54 dcl last_vowel_found aligned init(1); /* same for previous unit in this syllable */
25  55 dcl (first, second) fixed init(1); /* index into digram table for current pair */
26  56 dcl (random_unit, random_vowel) entry (fixed);
27  57 dcl loa$gen entry options(variable);
28  58
doi = 0 to length;
29  59 password(i):= 0;
30  60 hypenanted_word(1):= "0"b;
31  61 end;
32  62 nchars := length;
33  63
34  64 /* get rest of units in password */
35  65 unit := 0;
36  66 do index := 1 by 1 while(index <= nchars);
37  67 if syllable_length := 1 then
38  68 do; /* on first unit of a syllable, use any unit */
39  70 keep_trying: unit := abs(unit); /* last unit was accepted (or first in word), make positive */
40  71 goto first_time;
41  72 retry: unit := -abs(unit); /* last unit was not accepted, make negative */
42  73
43  74 first_time: if index <= nchars /* if last unit of word must be a syllable, it must be a vowel */
44  75 then call random_vowel(unit);
45  76 else call random_unit(unit);
46  77 password(index):= abs(unit); /* put actual unit in word */
47  78 if index := 2 then if digrams(password(index)-1).password(index).illegal_pair
48  79 then goto retry; /* this pair is illegal */
49  80 if rules(password(index)).not_begin_syllable then goto retry;
50  81 if letters_split.second(password(index)) = "0" then
random_word_list

if index = nchars
    then goto retry;
else
    if index = nchars-1 & 'rules(password(index)).vowel & 'rules(password(index)).alternate_vowel
        then goto retry; /* last unit was a double-letter unit and not a vowel */
    else if unit < 0
        then goto keep_trying;
    else nchars = nchars - 1;
else if unit < 0 then goto keep_trying;
syllable_length = 2;
if rules(password(index)).vowel | rules(password(index)).alternate_vowel
    then
        do;
        cons_count = 0;
        vowel_found = "1"b;
        end;
    else
        do;
        cons_count = 1;
        vowel_found = "0"b;
        end;
    last_vowel_found = "0"b;
    end;
else
    do;
    call generate_unit;
    if second = 0 then goto all_done; /* we have word already */
    end;
end;

/* enter here at end of word */
all_done:
word_length = index - 1;
return;

/* various other entries */
debug_on: entry;
debug = "1"b;
return;
debug_off: entry;
debug = "0"b;
return;
118 /* Procedure GENERATE_UNIT */
119
120 /* generate next unit to password, making sure */
121 /* that it follows these rules: */
122 /* 1. Each syllable must contain exactly 1 or 2 consecutive vowels, */
123 /* where y is considered a vowel. */
124 /* 2. Syllable end is determined as follows: */
125 /* a. Vowel is generated and previous unit is a consonant and */
126 /* syllable already has a vowel. In this case new syllable is */
127 /* started and already contains a vowel. */
128 /* b. A pair determined to be a "break" pair is encountered. */
129 /* In this case new syllable is started with second unit of this pair. */
130 /* c. End of password is encountered. */
131 /* d. "begin" pair is encountered legally. New syllable is started */
132 /* with this pair. */
133 /* e. "end" pair is legally encountered. New syllable has nothing yet. */
134 /* f. Try generating another unit if: */
135 /* a. third consecutive vowel and not y, */
136 /* b. "break" pair generated but no vowel yet in current syllable */
137 /* or previous 2 units are "not_end". */
138 /* c. "begin" pair generated but no vowel in syllable preceding */
139 /* begin pair, or both previous 2 pairs are designated "not_end". */
140 /* d. "end" pair generated but no vowel in current syllable or in "end" pair. */
141 /* e. "not_begin" pair generated but new syllable must begin */
142 /* (because previous syllable ended as defined in 2 above). */
143 /* f. vowel is generated and 2a is satisfied, but no syllable break is possible in previous 3 pairs. */
144 /* g. Second & third units of syllable must begin, and first unit is "alternate_vowel". */
145 /* The done routine checks for required prefix vowels & end of word conditions. */
146
147 generate_unit: procedure;
148
dcl t x aligned like digrams;
149
dcl try_for_vowel bit(1) aligned;
150
dcl unit_count fixed init(1) */ count of tries to generate this unit */
151
dcl v bit(3) aligned;
152
dcl i fixed;
153
155 first = password(index-1);
156
157 /* on last unit of word and no vowel yet in syllable, or if previous pair */
158 /* requires a vowel and no vowel in syllable, then try only for a vowel */
159
160 if syllable_length = 2 /* this is the second unit of syllable */
161 then try_for_vowel = "vowel_found & index<chars;" /* last unit of word and no vowel yet, try for vowel */
162 else /* this is at least the third unit of syllable */
163 if "vowel_found & digrams(password(index-2),first).not_end */
164 then try_for_vowel = digrams(password(index-2),first).suffix;
165 else try_for_vowel = "0b;"
166 goto keep_trying; /* on first try of a unit, don't make the tests below */
167
168 /* come here to try another unit when previous one was not accepted */
169
170 try_more:
171
test = abs(unit); /* last unit was not accepted, set sign negative */
172 if unit_count = 100
173 then
174 do;
if debug
then
do:
call loc$mnl("100 tries failed to generate unit."");
do i = 1 to index;
call loc$mnl("a", letters(password(1)));
end;
call loc$mnl("/");
end;
call random_word(password, hyphenated_word, length, index, random_unit, random_vowel);
second = 0;
return;
end;
*/ come here to try another unit whether last one was accepted or not */
keep_trying:
if try_for_vowel
then call random_vowel(unit);
else call random_unit(unit);
second = abs(unit); /* save real value of unit number */
if unit > 0 then unit_count = unit_count + 1; /* count number of tries */
*/ check if this pair is legal */
if digrams(first,second).illegal_pair
then goto try_more;
else
if first = second /* if legal, throw out 3 in a row */
then
if index > 2
then
if password(index-2) = first
then goto try_more;
if letters_split(second).second " = " /* check if this is 2 letters */
then
if index = nchars /* then if this is the last unit of word */
then goto try_more; /* then a two-letter unit is illegal */
else nchars = nchars - 1; /* otherwise decrement number of characters */
password(index) = second;
if rules(second).alternate_vowel
then v = rules(first).vowel;
else v = rules(second).vowel;
x.begin = digrams(first,second).begin;
x.not_begin = digrams(first,second).not_begin;
x.end = digrams(first,second).end;
x.not_end = digrams(first,second).not_end;
x.break = digrams(first,second).break;
x.prefix = digrams(first,second).prefix;
x.suffix = digrams(first,second).suffix;
x.illegal_pair = digrams(first,second).illegal_pair;
if syllable_length > 2 /* force break if last pair must be followed by a */
then /* vowel and this unit is not a vowel */
if digrams(password(index-2),first).suffix
then
if v then break = "i" /*if last pair was not end, new_unit gave us a vowel*/
*/ In the notation to the right, the series of letters and dots stands
random_word_list  08/01/75  1729.4  est  Tue  page 6

for the last n units in this syllable, to be interpreted as follows:

v stands for a vowel (including alternate_vowel)

c stands for a consonant

x stands for any unit

the dots are interpreted as follows (o is used as example)

...c one or more consecutive consonants

c...c zero or more consecutive consonants

...c one or more consecutive consonants from beginning of syllable

c ...c zero or more consecutive consonants from beginning of syllable

the vertical line | marks a syllable break.
The group of symbols indicates what units there are in current

syllable. The last symbol is always the current unit.
The first symbol is not necessarily the first unit in the

syllable, unless preceded by dots. Thus, *voc.cv* should be

interpreted as "...xvoc..cx" (i.e., add "...cx" to the beginning of all

syllables unless dots begin the syllable.). */

if syllable_length = 2 & not_begin /* pair may not begin syllable */

then goto loop; /* rule 3e. */

if vowel_found

then

if cons_count = 0

then

begin

/* vs...cx */

then

if syllable_length = 3 & not_end_[3] /* vs...cx begin */

then

/* can we break at vs..clx */

if not_end_[2] /* no, try a break at vs..clx */

then goto loop; /* rule 3c, */

else call done(v,2);

/* vs..clx begin, treat as break */

else call done(v,3);

/* vs..clx begin */

else

if not_begin

/* vs...cx "begin */

then

if break

/* vs...cx not_begin */

then

if not_end_[2]

/* vs..clx break */

then goto loop; /* rule 3b, can't break */

else call done(v,2); /* vs..clx break */

else

if v

then

/* vs...xv 'break not_begin */

if not_end_[2] /* try break at vs..clv */

then goto loop; /* rule 3f, break no good */

else call done("tv"v,2);

/* vs..clv treat as break */

else

if end

/* vs...cc 'break not_begin */

then call done("tv"v,1);

/* vs...clv end */

else call done("tv"v,0);

/* vs...cc 'break 'end not_begin */

else

if v

then

/* vs...cx "begin 'not_begin */

if not_end_[3] & syllable_length = 3 /* vs...cv rule 2a says we must break somewhere */

then

if not_end_[2]

then

if cons_count > 1

/* vs..clx doesn't work */

then

/* vs...cv */
else call done(v,2); /* ...v*/x break */
else call done("r",b,0); /* ...v"end "begin "break */
else if break /* ...ex */
then goto loop; /* rule 3b, ...clx break no good */
else if end /* ...ex "break */
then if v /* ...ex end */
then call done("0",b,1); /* ...clv end (new syllable) */
else goto loop; /* rule 3b, ...clv end no good */
else if v /* ...ex "end "break */
then if begin & syllable_length > 2 /* ...cv "end "break */
then goto loop; /* c...clcv "end "break begin, rule 3c */
else call done("v",b,0); /* ...cv "end "break "begin */
else if begin /* ...cv "break "end */
then if syllable_length > 2 /* ...ccv begin */
then goto loop; /* rule 3c, ...ccv begin */
else call done("v",b,3); /* ...ccv begin */
else call done("x",b,0); /* ...ccv "end "break "begin */
********* return here when unit generated has been accepted *********/
return;

********* enter here when unit generated was good, but we don’t want to use it because
it was supplied as a negative number by random_unit or random_vowel; *********
if letters_split(second).second "* "
then nchars = nchars + 1; /* pretend unit was no good */
unit = unit; /* make positive to say that it would have been accepted */
goto keep_trying;

********* enter here when unit generated is no good *********/
loop: if letters_split(second).second "* " then nchars = nchars + 1;
*/ goto try_more;
random_word_list

MAX = 1000

/*
  391   * PROCEDURE DONE */
  392
  393   * this routine is internal to generate_unit because it can return to loop */
  394   * call done when new unit is generated and determined to be
  395   * legal. Arguments are new values of:
  396   * vt vowel_found
  397   * vb syllable_length (number of units in syllable. 0 means increment for this unit)
  398   */
  399
do: procedure (vt, vb);
  400  dcl vt bit(1) aligned;
  401  dcl vb fixed;
  402
  403  * if we are not within first 2 units of syllable, check if
  404  * vowel must precede this pair */
  405  if al ^ a then if syllable_length ^ a then if prefix then if 'rules.vowel(password(index-2))
  406  then /* vowel must precede pair but no vowel precedes pair */
  407  if vowel_found /* if there is a vowel in this syllable, */
  408  then /* we may be able to break this pair */
  409  if not_end(2) /* check if this pair may be treated as break */
  410  then goto loop; /* no, previous 2 units can't end */
  411  else /* yes, break can be forced */
  412
do;
  413  call done(O+b,2); /* ...ex ex or ...ex */
  414  return;
  415  end;
  416  else goto loop; /* no vowel in syllable */
  417
  418  * Check end of word conditions. If end of word is reached, then
  419  1. we must have a vowel in current syllable, and
  420  2. This pair must be allowed to end syllable
  421 */
  422
  423  if al ^ a then
  424  if index ^ nochars
  425  then
  426  if index ^ nochars
  427  then
  428  if rules(second).no_final_split /* this bit is on for "e" */
  429  then
  430  if al ^ a then
  431  if rules.vowel(first) /* e preceded by vowel is ok, however */
  432  then;
  433  else
  434  if vowel_found(syllable_length); /* otherwise previous 2 letters must be */
  435  then goto loop; /* able to end the syllable */
random_word_list

448     if unit < 0
449       then goto accepted_but_keep_trying;
450     else sl = 0;
451   if unit < 0 then goto accepted_but_keep_trying;
452     if v: sl = 1;
453     then cons_count = 0;
454       /* this unit is a vowel or new syllable is to begin */
455     else
456       if sl = 0
457         then cons_count = cons_count + 1;
458       else /* a new syllable was started some letters back, cons_count gets */
459         cons_count = min(sl-1,cons_count+1);  /* incremented, but no more than number of units in syllable */
460     if sl = 0
461       then syllable_length = syllable_length + 1;
462     else syllable_length = sl;
463     if syllable_length > 3
464       then last_vowel_found = vowel_found;
465     else last_vowel_found = "C";
466     vowel_found = vf;
467     if index - syllable_length + 1 < nchars
468       then hyphenated_word(index - syllable_length + 1) = "-";
469     end done;
470     end generate_unit;
471 /*
472 */
/*
475     * PROCEDURE NOT_END_ */
476     * not_end_(i) returns "1"t when ( password(index-i), password(index-i+1) )
477     * may not end a syllable, or when password(index-i+2) may not begin a syllable */
478
479     * not_end_: procedure(i) returns(bit(1));
480     
481     def i: fixed;
482     then return( "rules.vowel(password(i))" );
483     if i = "1" then
484     then
485     if rules.not_begin_syllable(password(index-i+2)) then return("1") b;
486     return(digram(password(index-i),password(index-i+1)),not_end);
487     end:
488
489     end;
*/
/* This subroutine compiles the digram table, given a pointer to the
  segment containing the source. It returns a flag if compiling was
  unsuccessful */

5 read_table; procedure (source_table_ptr, bit) returns(bit1);

7 dcl source_table char(1048575) based (source_table_ptr);
8 dcl source_table_ptr ptr;
1 2 ********** include file digram_structure.incpl; **********
3
4 dcl digrams$digrams external;
5 dcl digrams$n_units fixed bin external;
6 dcl digrams$letters external;
7 dcl digrams$rules external;
8
9 /* This array contains information about all possible pairs of units */
10
11 dcl 1 digrams(n_units, n_units) based (addr(digrams$digrams)).
12 2 begin bit(1), /* on if this pair must begin syllable */
13 2 not_begin bit(1), /* on if this pair must not begin */
14 2 end bit(1), /* on if this pair must end syllable */
15 2 not_end bit(1), /* on if this pair must not end */
16 2 break bit(1), /* on if this pair is a break */
17 2 prefix bit(1), /* on if vowel must precede this pair in same syllable */
18 2 suffix bit(1), /* on if vowel must follow this pair in same syllable */
19 2 illegal_pair bit(1), /* on if this pair may not appear */
20 2 pad bit(1); /* this makes 9 bits/entry */
21
22 /* This array contains left justified 1 or 2-letter pairs representing each unit */
23
24 dcl letters(0$n_units) char(2) aligned based (addr(digrams$letters));
25
26 /* This is the same as letters, but allows reference to individual characters */
27
28 dcl 1 letters_split(0$n_units) based (addr(digrams$letters)),
29 2 first char(1),
30 2 second char(1),
31 2 pad char(2);
32
33 /* This array has rules for each unit */
34
35 dcl 1 rules(n_units) aligned based (addr(digrams$rules)),
36 2 no_final split bit(1), /* can't be the only vowel in last syllable */
37 2 not_begin syllable bit(1), /* can't begin a syllable */
38 2 vowel bit(1), /* this is a vowel */
39 2 alternate_vowel bit(1); /* this is an alternate vowel, (i.e., "y") */
40
41 dcl n_units defined digrams$n_units fixed bin;
42
43 ********** end include file digram_structure.incpl; **********

9
10 dcl (1, j, k, l) fixed bin;
11 dcl errflag bit(1) init(0b);
12 dcl fatal_flag bit(1) init(0b);
read_table_list

13 dol neither_is_vowel bit(1);  
14 dol p ptr;  
15 dol 1 x based (p) like digrams;  
16 dol letters(0:99) aligned char(2); /* storage for letters until we know how many there are */  
17 dol 1 rules(99) aligned like rules; /* ditto for rules */  
18 dol code fixed bin (35);  
19 dol flag bit(1);  
20 dol char char(1) init(* =);  
21 dol be fixed bin(24);  
22 dol bitcount fixed bin(24);  
23 dol cleanup condition;  
24 dol term_seg_ptr entry (ptr, fixed bin(35));  
25 dol get_group_id_entry returns(char(32) aligned);  
26 dol hcs_8deletentry_seg entry (ptr, fixed bin(35));  
27 dol hcs_8lake_seg entry (char(*), char(*), char(*), fixed bin(5), ptr, fixed bin(35));  
28 dol (hcs_8add_acl_entries, hcs_8delete_acl_entries) entry  
29 /* char(*), char(*), ptr, fixed bin, fixed bin(35));  
30 dol 1 acl aligned,  
31 2 user_name char(32),  
32 2 modes bit(36),  
33 2 pad bit(36),  
34 2 code fixed bin(35);  
35 dol null builtin;  
36 dol loc fixed init(1);  
37 dol end bit(1);  
38 dol new_line char(1) init (* =);  
39 */  
40 dol com_err_assume_name entry options(variable);  
41 dol hcs_last_kb_use entry(ptr, fixed bin(24), fixed bin(35));  
42 dol get_pdir_entry returns(char(160) aligned);  
43 dol get_wdir_entry returns(char(160) aligned);  
44 dol acl entry options(variable);  
45 dol alm_prog based(prog_ptr) char(262144);  
46 dol prog_ptr ptr static init(null);  
47 dol seg_index init(1) fixed bin;  
48 49 /* This procedure creates an ALM program containing empty blocks of storage.  
50 After finding out how many units there are, the size of each of these  
51 blocks can be determined. The ALM program is then assembled, and  
52 segdef's are thus created which point to the beginning of each of  
53 these blocks.  
54 The first statement of the ALM program will be:  
55 56 equ n,xxxx  
57 58 where xxxx will be the number of units determined. The rest of  
59 the statements are below: */  
60 61 dol almStatements(g) char(30) varying init (  
62 *segdef digrams*,  
63 *segdef n_units*,  
64 *segdef letters*,  
65 *segdef rules*,  
66 /*bas n_units,*/  
67 /*bas digrams,(n_units*3)/4, */  
68 /*bas letters,(n_units*3)/4, */  
69 /*bas rules,(n_units*3)/4, */  
70 /*bas rules,n_units, */  
71 /* n_units fixed bin */  
72 /* digram_units,n_units bits */  
73 /* letters(n_units) char(2) aligned */  
74 /* rules(n_units) aligned, 2 (01,02,03,04) bit(1) */
read_table_list

*end*;
72
73
74
75
dcl loa_entry_options (variable);
77
dcl loa_fnml entry_options (variable);
78
/* check if a dollar sign ends segment */
79
80
81
82
83
84
85
86
do i = 1 to 90 while(char = ":"); /* read until semicolon */
87    char = substr(source_table, loc, 1);
88
89
90
91
do;
92
call loa_fnml (*alpha character expected*);
93    fatal_error;
94    fatal_flag = "*b*;
95    goto err;
96
97    substr(letters (_,1),1,1) = char;
98    char = substr(source_table, loc, 1);
99
100
101
102
103
104
105
106
107
108
do;
109    rules(_,not_begin Syllable = char) = "*s*;
110    char = substr(source_table, loc, 1);
111
112
113
114
115
116
117
118
119
120
121
122
123
124
125
126
127
128
substr(letters(_,2),1) = char;
read_table_list

129   char = substr(source_table, loc, 1);
130   loc = loc + 1;
131   goto try_bits;
132   end;
133   
134   /* check character following for comma, new_line, or semicolon */
135   if char = "" & char = ";" & char = " new_line"
136   then
137   do;
138   call ioa$_$nil ("comma, blank, zero, one, or letter expected");
139   goto fatal_error;
140   end;
141   
142   /* check if this unit is already defined */
143   if 1 = 1
144   then
145   do j = 1 to l - 1;
146   if letters (j) = letters (i)
147   then do;
148   call ioa$_$nil ("duplicate unit specification **a**, letters (j));
149   goto fatal_error;
150   end;
151   end;
152   end;
153   
154   /* set vowel flags */
155   rules (i).vowel = letters (i) = "a" ; letters (i) = "e" ;
156   rules (i).vowel = letters (i) = "i" ; letters (i) = "o" ; letters (i) = "u";
157   rules (i).alternate_vowel = letters (i) = "y";
158
159   end;
160   if i > 90
161   then do;
162
163   call ioa ("Too many units defined"); /* more than 90 units */
164   return (T');
165   return (T');
166   end;
167   end;
168   /* this is the one unit for aborted compilation */
169   It deletes the temporary segment containing the adm program, and
170   deletes the acl entry of digrams that references this process's id. */
171   on condition(cleanup)
172   begin;
173   if prog_ptr = null
174   then call hcs$_$delete_entry (prog_ptr, code);
175   call hcs$_$delete_acl_entries (get_word ()), "digrams", addr(acl), 1, code);
176   end;
177   end;
178   /* now that we know how many units, we can create the adm program */
179   /* first create the source segment in the process directory */
180   call hcs$_$make_seg ("", "digrams.adm", 0, 0, prog_ptr, code);
181   if prog_ptr = null
182   then do;
183   error_in_adm_prog:
187 call cme.err.$suppress_name (code, "digram_table_compiler", "digrams.sh in process directory");
188 return("\n");
189 end;
190
191 call addline ("equ 0," [] subst(character(i=1), verify(character(i=1), ")"); /* first line of ALM program */
192 do j = 1 to hbound(alg_statements,1); /* all the rest of the lines */
193 call addline (alg_statements(j));
194 end;
195
196 /* set the bit count of the source segment */
197 call hos.$set_bc_seg (proc_ptr, (seg_index - 1))
199 if code = 0 then goto error_in_alg_prog;
200 /* assemble the ALM program */
201 /* assemble the ALM program */
203 call alg (before(get_pdir(), ")"); /* addgrams */
204
205 /* Hopefully we got no errors. If we didn't, we can't tell */
206 /* delete the alg program, and set the acl of the object program */
207 to rw for this process */
208 /* hos.$setentry_seg (proc_ptr, code); */ ignore code */
210 proc_ptr x null(); /* just to be clean */
211 acl.user_name = get_group_id();
212 acl.mode = "101b"
213 acl.pad = "0"
214 call hos.$add_acl_entries (get_wdir(), "digrams", acl(user), i, code);
215 if code = 0
216 then do;
217 call cme.err.$suppress_name (code, "digram_table_compiler", "digrams");
218 return("\n");
219 end;
220
221 /* Store stuff into the object segment */
222 n_units = i + 1; /* This is the first reference to the object segment */
224 letters(i) = "";
225 do i = 1 to n_units;
226 letters(i) = letters(i);
227 rules(i) = rules(i);
228 end;
229
230 /* digram table is compiled now */
231
do i = 1 to n_units;
233 do j = 1 to n_units;
234 p = subst(digram(i, j));
235 x.begin, x.not_begin, x.end, x.not_end, x.break, x.prefix, x.suffix = "0";
236 char = subst(source_table, loc, i);
237 do while (char = new_line);
238 loc = loc + 1;
239 char = subst(source_table, loc, i);
240 end;
241 if char = "$" then do; call lca ("Illegal $ -- premature end"); return("\n"); end;
242 if char = " "; char = "";
243 then do;
read_table_list

245 x.begin = char."11;  
246 loc + loc + 1; 
247 call next_char_bit;  
248 x.not BEGIN = char."11;  
249 call next_char_bit;  
250 x.break = char."11;  
251 call next_char;  
252 x.prefix = char."-";  
253 end;  
254 call next_char;  
255 if char = "*" : char = "-" then goto errb;  
256 if char = letters_split(i).first then goto errb;  
257 call next_letter(i);  
258 call next_char;  
259 if char = "*" : char = "-" then goto errb;  
260 if char = letters_split(j).first then  
261 do;  
262  
263 /* in case the second unit of a digram pair specification is illegal,  
264 this sequence attempts to get in sync again so that messages will not  
265 be printed indefinitely. If the first unit is illegal,  
266 no attempt is made to get in sync. */  
267 k = 1;  
268 errb: do k = max(k,1) to n_units while(char = letters_split(k).first);  
269 end;  
270 /* this takes care of skipping some units or duplicating the last unit */ 
271 if k < n_units  
272 then  
273 do;  
274  
275 if letters_split(k).second = "+"  
276 then  
277 do;  
278 char = subst(source_table,loc,1);  
279 if char = letters_split(k).second  
280 then  
281 do;  
282 k = k + 1;  
283 goto errb;  
284 end;  
285 end;  
286 end;  
287  
288 /* if the unit can't be found, assume it's there but spelled wrong */ 
289 errb: j = j - 1;  
290 /* if there is an extra digram that can't be found, we'll get another message */ 
291 call lopa_jnll("out of order or illegal letter"); goto errb;  
292 end;  
293 call next_letter(j);  
294 char = subst(source_table,loc,1);  
295 loc = loc + 1;  
296 if char = "+" & char = "new_line" & char = "#"  
297 then  
298 do;  
299 if char = "*" & char = "-" & char = "#"  
300 then  
301 erra:  
302 do;  
303 call lopa_jnll("alpha character expected");  
304 goto errb;  
305 end;  
306
if char == "\n"
    then x.suffix = "\n"b;
else
    if char == "\n"
        then x.illegal_pair = "\n"b;
call next_bit;
if end then goto loop;
x.end = char"";
call next_bit;
if end then goto loop;
if not_end = char="";
char = subst(source_table,loc,1);
if char "* new_line & char "*"
then do; call ioa_snfl("end of line expected"); goto err; end;
loc = loc + 1;
end loop;
neither is vowel = "rules.vowel() & "rules.vowel() & "rules.alternate_vowel() & "rules.alternate_vowel();
if (x.begin & (x.not_begin(x.end & neither is vowel)||(x.not_end & neither is vowel):(x.break & "rules.vowel();))
then (rules.not_begin_syllable() & x.break)
(x.end & (x.not_end !(x.not_begin & neither is vowel):(x.break & "rules.vowel();)))
(x.break & "x.not_begin & rules.vowel() & x.not_end & rules.vowel();)
(x.beginix & (x.not_begin & x.end & (x.beginix & x.suffix) & x.illegal_pair)
en then
do;
call ioa_snfl("consistency error");
do k = 1 to loc-1 while (subst(source_table, loc-k, 1) "* new_line);
end;
do l = 0 to brg-loc while (subst(source_table, loc+1, 1) "* new_line);
end;
if "errflag then
    call ioa_snfl(" before * on following line");
call ioa_snfl(" before * on following line");
substitute(source_table, loc-k, 1) "* new_line);;
end;
if "errflag then return("*b; /* fatal error, can't continue */
char = subst(source_table,loc-1,1);
do loc = loc by 1 while (char "*" &
char = subst(source_table,loc,1);
end;
errflag = "*b;
end;
read_table_list

/* get next letter, space, or "-" */

next_char: procedure;
char = subst(source_table,loc,1);
loc = loc + 1;
if (char["a" | char]="z") & char["a" | char]="z" then do: call ioa$_nml("alpha character expected"); goto err; end;
end;

/* get next space or ":" */
	next_char_bit: procedure;

char = subst(source_table,loc,1);
if char=" " & char="" then do: call ioa$_nml("space or ":" expected"); goto err; end;

/* get next space, ":", ",", or new_line */

next_bit: procedure;
char = subst(source_table,loc,1);
end = "0";
loc = loc + 1;
if char=" " then do: if char="." | char=new_line then end="1"; else do: call ioa$_nml("space, ",", or new line expected"); goto err; end;
end;

/* get next letter if this unit is a 2-letter unit */
	next_letter: proc(i);

dcl i fixed bin;
if letters_split(i).second="" then do:
call next_char;
if char="letters_split(i).second then do:
call ioa$_nml("" letters_split(i).second ":" expected"); goto err; end;
end;
end;
end;
end;
read_table_list

419 /* Add a line to ALM program */
420
421 addline: proc (string);
422 do string char(30) varying;
423 substr(aln_prog, seg_index, length(string) + 1) = string ++
424 ";
425 seg_index = seg_index + length(string) + 1;
426 end;
427
428 end;
APPENDIX IV

2000 RANDOM WORDS

The 2000 random words listed on the following pages were generated in one particular sample run using the tables described in Appendix I. See page 18 for a description of this listing.
2000 RANDOM WORDS

acbra anpedavi anpe-da-vi
acbrasju anpedavi anpe-da-vi
acmacico anpedavi anpe-da-vi
acnaw anpedavi anpe-da-vi
adakgem anpedavi anpe-da-vi
addazov anpedavi anpe-da-vi
addeocro anpedavi anpe-da-vi
adfarvra anpedavi anpe-da-vi
adfrobgaa anpedavi anpe-da-vi
adicoc anpedavi anpe-da-vi
adkrev anpedavi anpe-da-vi
adlisa anpedavi anpe-da-vi
adoif anpedavi anpe-da-vi
adtemruf anpedavi anpe-da-vi
aduvj anpedavi anpe-da-vi
afttwir asejyaha asej-ja-ha
agniji asfha ash-fa
agromjax ashuwirp ashu-wirp
agrovca asuwoj as-su-woj
aiaboc atennga ateu-nga
aioboaj atkopye atko-yej
aijsav atoldt atol-t
aitjje atojshyn ato-shyn
ajdama atshub atsh-ub
ajherj ajhe-ry
ajkealv ahuva au-va
ajlytya aupkahy aup-ka-hy
ajnek auptcu aupt-cu
akdil avay a-va-y
akhec avarmev av-car-mev
akhibwos avriss av-riss
akjaruj atthwy at-thwy
akkloko akkloko
akprujo avutman av-ut-man
aktadssu awwecca aw-wec-ba
alcho aycale ay-cale
algofwee ayjedsi ay-jeds-i
alltomp alyow a-lyow
alkkeye aymsfop ayms-fop
alwafi ayootta ayoot-ta
ambbrigno aypvihy ayp-vi-hy
amshy ay sig
amvacs babfeby bab-fe-by
amvuti bacgebvo bac-geb-vo
anafniv bafdacga bac-dac-ga
<table>
<thead>
<tr>
<th>boshtpel</th>
<th>bosht-pel</th>
<th>ceasjota</th>
<th>ceas-jo-ta</th>
<th>ciscreny</th>
<th>cis-cren-y</th>
</tr>
</thead>
<tbody>
<tr>
<td>boudcof</td>
<td>boud-cof</td>
<td>ceays</td>
<td>ceays</td>
<td>civbybab</td>
<td>civ-by-bab</td>
</tr>
<tr>
<td>bowyt</td>
<td>bo-wyt</td>
<td>ceehat</td>
<td>cee-hat</td>
<td>civjece</td>
<td>civ-jece</td>
</tr>
<tr>
<td>bradcur</td>
<td>brad-cur</td>
<td>ceekasom</td>
<td>cee-ka-som</td>
<td>cixdo</td>
<td>cix-do</td>
</tr>
<tr>
<td>brefep</td>
<td>bref-fep</td>
<td>ceemm</td>
<td>ceemm</td>
<td>clefno</td>
<td>clef-no</td>
</tr>
<tr>
<td>brerthy</td>
<td>brer-thy</td>
<td>ceethpun</td>
<td>ceeth-pun</td>
<td>cleyhy</td>
<td>cley-hy</td>
</tr>
<tr>
<td>brighe</td>
<td>brighe</td>
<td>ceetvif</td>
<td>ceet-vif</td>
<td>cislo</td>
<td>cli-slo</td>
</tr>
<tr>
<td>brikan</td>
<td>bri-kaw</td>
<td>cegmowec</td>
<td>ceg-mo-wec</td>
<td>cliwa</td>
<td>cli-wa</td>
</tr>
<tr>
<td>broctbar</td>
<td>broct-bar</td>
<td>cegpu</td>
<td>ceg-pu</td>
<td>clorgcy</td>
<td>clorg-cy</td>
</tr>
<tr>
<td>brolwoi</td>
<td>brol-woi</td>
<td>cegvu</td>
<td>ceg-vu</td>
<td>coasbebi</td>
<td>coas-be-bi</td>
</tr>
<tr>
<td>bronjept</td>
<td>bron-jept</td>
<td>ceigl</td>
<td>ceil-gi</td>
<td>cocuevo</td>
<td>co-cue-vo</td>
</tr>
<tr>
<td>bruosta</td>
<td>bru-os-ta</td>
<td>ceindhy</td>
<td>ceind-hy</td>
<td>codfri</td>
<td>cod-fr1</td>
</tr>
<tr>
<td>bruvluke</td>
<td>bruv-lufe</td>
<td>cejiogmo</td>
<td>ce-ji-og-mo</td>
<td>coftti</td>
<td>coft-ti</td>
</tr>
<tr>
<td>bryarne</td>
<td>bryarne</td>
<td>cemjatbu</td>
<td>cem-jat-bu</td>
<td>comtdoa</td>
<td>com-t-doa</td>
</tr>
<tr>
<td>bryffoj</td>
<td>bryf-foj</td>
<td>cenjo</td>
<td>cen-jo</td>
<td>conclay</td>
<td>con-clay</td>
</tr>
<tr>
<td>byryse</td>
<td>byryse</td>
<td>cenved</td>
<td>cenv-ed</td>
<td>conmeco</td>
<td>con-m-e-co</td>
</tr>
<tr>
<td>bucushu</td>
<td>bu-cu-shu</td>
<td>ceoliho</td>
<td>ce-o-li-ho</td>
<td>copsgha</td>
<td>cops-gha</td>
</tr>
<tr>
<td>buebwu</td>
<td>bueb-wu</td>
<td>ceowf</td>
<td>ce-ow-f</td>
<td>copuac</td>
<td>co-pu-ac</td>
</tr>
<tr>
<td>busdene</td>
<td>bus-id-ene</td>
<td>cerba</td>
<td>cer-ba</td>
<td>corlibbu</td>
<td>cor-lib-bu</td>
</tr>
<tr>
<td>butoc</td>
<td>bu-tooc</td>
<td>cesta</td>
<td>ces-tha</td>
<td>coshrvg</td>
<td>co-shrvg</td>
</tr>
<tr>
<td>buyen</td>
<td>bu-yen</td>
<td>ceugcho</td>
<td>ceug-cho</td>
<td>cotkn1</td>
<td>cot-kni</td>
</tr>
<tr>
<td>buyonvna</td>
<td>bu-yov-na</td>
<td>ceuhau</td>
<td>ceu-hau</td>
<td>cotrufa</td>
<td>cott-ru-fa</td>
</tr>
<tr>
<td>bybguf</td>
<td>byb-guf</td>
<td>cewfu</td>
<td>cew-fu</td>
<td>couhile</td>
<td>cou-hile</td>
</tr>
<tr>
<td>bybhihyp</td>
<td>byb-hi-hyp</td>
<td>cewphjo</td>
<td>cewph-jo</td>
<td>couwkuf</td>
<td>cou-wk-fu</td>
</tr>
<tr>
<td>bycij</td>
<td>by-ci-j</td>
<td>cezcy</td>
<td>cez-cy</td>
<td>cysfo</td>
<td>coys-fo</td>
</tr>
<tr>
<td>byhojoc</td>
<td>by-ho-joc</td>
<td>chafja</td>
<td>chaf-ja</td>
<td>cozrer</td>
<td>coz-rer</td>
</tr>
<tr>
<td>bypio</td>
<td>by-i-pi-o</td>
<td>chaheog</td>
<td>cha-he-og</td>
<td>crafnced</td>
<td>craf-nec</td>
</tr>
<tr>
<td>byjoary</td>
<td>by-joar-ry</td>
<td>chatmum</td>
<td>chat-mum</td>
<td>craktau</td>
<td>crak-ta-mu</td>
</tr>
<tr>
<td>bykmoei</td>
<td>byk-me-ol</td>
<td>chetki</td>
<td>chet-ki</td>
<td>crasco</td>
<td>cras-co</td>
</tr>
<tr>
<td>bylij</td>
<td>by-lij</td>
<td>chewchut</td>
<td>chew-chut</td>
<td>cravpo</td>
<td>crav-po</td>
</tr>
<tr>
<td>bynri</td>
<td>byn-ri</td>
<td>chimwed</td>
<td>chim-wed</td>
<td>crejorry</td>
<td>cre-jor-ry</td>
</tr>
<tr>
<td>bypvee</td>
<td>byp-vpee</td>
<td>chishrai</td>
<td>chi-shrai</td>
<td>crelerhi</td>
<td>cre-ler-hi</td>
</tr>
<tr>
<td>buryc</td>
<td>by-ruc</td>
<td>chivi</td>
<td>chi-vi</td>
<td>crerju</td>
<td>crer-ju</td>
</tr>
<tr>
<td>bysstoct</td>
<td>bys-stoct</td>
<td>choasdy</td>
<td>choas-dy</td>
<td>cribmac</td>
<td>crib-mac</td>
</tr>
<tr>
<td>bytatha</td>
<td>by-ta-tha</td>
<td>chrubfu</td>
<td>chrub-fu</td>
<td>crova</td>
<td>cro-va</td>
</tr>
<tr>
<td>cacky</td>
<td>cack-y</td>
<td>chuspryn</td>
<td>chus-pryn</td>
<td>cujapki</td>
<td>cru-jap-ki</td>
</tr>
<tr>
<td>caged</td>
<td>ca-ged</td>
<td>chysnior</td>
<td>chys-ni-or</td>
<td>crybtol</td>
<td>cryb-toi</td>
</tr>
<tr>
<td>caibo</td>
<td>caib-bo</td>
<td>ciaki</td>
<td>ci-a-ki</td>
<td>cryce</td>
<td>cryce</td>
</tr>
<tr>
<td>cajhet</td>
<td>caj-thett</td>
<td>cibhihi</td>
<td>cib-bi-hi</td>
<td>cryjawl</td>
<td>cry-jawl</td>
</tr>
<tr>
<td>calkny</td>
<td>cal-kny</td>
<td>cickka</td>
<td>cick-ka</td>
<td>cubyuct</td>
<td>cu-byuct</td>
</tr>
<tr>
<td>carurla</td>
<td>ca-rur-la</td>
<td>cienbaf1</td>
<td>cien-ba-fi</td>
<td>cucksja</td>
<td>cucks-ja</td>
</tr>
<tr>
<td>cataseo</td>
<td>cats-e-co</td>
<td>cifdabgi</td>
<td>cif-dab-gi</td>
<td>cucesmu</td>
<td>cu-ce-s-mu</td>
</tr>
<tr>
<td>caugewd</td>
<td>cau-gewd</td>
<td>cizgawm</td>
<td>ciz-gawm</td>
<td>cuitelre</td>
<td>cu-it-reb</td>
</tr>
<tr>
<td>cawete</td>
<td>ca-wete</td>
<td>cijpuyon</td>
<td>cij-pu-yon</td>
<td>cujmemy</td>
<td>cuj-me-my</td>
</tr>
<tr>
<td>cawilgu</td>
<td>cawl-gu</td>
<td>cilch</td>
<td>cilch</td>
<td>cuowyri</td>
<td>cu-oy-wri</td>
</tr>
<tr>
<td>ceakhen</td>
<td>ceak-hen</td>
<td>cingloo</td>
<td>cin-gloo</td>
<td>cupco</td>
<td>cup-co</td>
</tr>
</tbody>
</table>
2000 RANDOM WORDS

ledtyo fed-tyo  
feent feent  
fehukemo fe-hu-ke-mo  
feike feike  
fejwi fej-wi  
fejwo fej-wo  
fekblyba fek-bly-ba  
feansi fem-li-urp  
fenbawjo fen-baw-jo  
ferhyts fer-hyts  
feroda fe-ro-da  
feushno feu-shno  
fewjoli few-jo-li  
fevel fe-vel  
feysa fey-so  
ficfu fic-fu  
fierbju fierb-ju  
fiewods fie-wods  
filhi fil-hi  
filhii fil-hi  
fillwuth fi-li-wuth  
fasabthu fia-sab-thu  
fisdu fis-du  
fitbat fit-bat  
fitvajo fit-vajo  
fitwif fit-wif  
fiufrase fi-u-frase  
fleac fleac  
flecky fleck-y  
fleufdu fleu-fu  
flywond fly-wond  
foodry fob-dry  
foiogu fo-o-gu  
foovuso fu-vo-so  
fofirdry fo-fi-dry  
fofyind fo-fyind  
fofrli fog-fli  
foinlir foin-lir  
foivni foiv-ni  
fokcruris fok-crus  
fokwuv fok-wuv  
fonwwrri fon-wu-rrri  
fooyd fooyd  
fotha fot-ha  
fraca fra-ca  
fraja fra-ja  

ralav fra-lav  
frepond fre-pond  
frertci frert-ci  
frerwu frer-wu  
freyt freyt  
fridetyp fri-de-typ  
frivik fri-vik  
frondwix frond-wix  
froki froki  
fruand fru-and  
fuscami fuc-fami  
fucharaw fuchar-aw  
fuehega fue-hega  
fugheyne fug-heyn  
fugji frug-ji  
fuhol fru-hol  
fuifew fru-i-few  
fujyevi fruj-ye-vi  
funn-ga funn-ga  
fruto fru-tho  
fuyet fru-yet  
gadma frad-maj-bu  
gaihci gaih-ci  
gairgy gairg-y  
gaisuvi gai-su-vi  
galndo gald-no  
ganrhm gan-rym  
garive ga-rive  
gatirdo ga-tir-d  
gavfia gav-fa  
gavneg gav-neg  
geajd gaoj-du  
gecpoby gec-po-by  
gedad gead  
geddni gedd-ni  
gefocfef gaf-fef  
gefup gef-pup  
geicand geic-nand  
geilikak ge-li-kak  
gejljel gel-je-lo  
gemda gema  
genaho ge-na-ho  

gokays ge-o-kays  
gesfi ges-fi  
egitio geti-to  
geunjeaka geu-nej-ka  
gevjarik gev-jarik  
ghlasy ghas-sy  
ghawndik gahn-dik  
ghbeb gheb-geb  
ghecarn ghec-marn  
ghheo ghee-bo  
ghsfrap ghe-frap  
ghelim ghe-lim  
ghibgeks ghib-geks  
ghinwa ghin-wa  
ghits ghits  
ghorfun ghor-fun  
giadya gi-a-dya  
giwbog giw-bog  
giglozyme gig-blo-dy  
gilyueg gilf-yueg  
gilmak gilm-lak  
ginzuzi gin-zu  
gipdace gip-dac-ae  
gipoqui gi-po-qui  
gipromso gip-rom-so  
girtwu gir-twu  
igidawn gidi-warn  
ghlatt ghlatt  
glecius glecius  
glafaf gila-a-fuj  
glicy gli-cy  
glodim glo-dim  
gloejyt gloj-pyt  
glycelim gly-ceilm  
glymka glym-kon  
gogatgo gogat-go  
gobpo gob-po  
gocmi goc-mi  
gofya gof-ya  
gogruco gogru-co  
golducu goli-du-co  
goliw goli-w  
gojloser gjo-cso-dy  
gomwu gom-wu  
gongu gong-pa-cyu  
goonhuung ghu-hung-y
knect
knenouke
knesa
knifru
knige
knoacy
knoji ti
knoke
knuedcu
knuwa
knyd glak
knyings
knywo
kobvil wo
kofeto
koinbi
kojrup
kojya
kokbrye
kokfuo
kolesesc e
kol skja
ko oche
kop hija
kosup ja
kotvads
kovshya
koybji
koywo
kra inoy
krapiep
krarracu
krete
krethuto
kri vo
krochis
kromjo
krowe v
kruofe
kryov
kuvuko
kugwog
kulsh
kum vic nu
kuvhy
laglu
lagmijy
lag tu su
lalfe
lamlu
larrre
lar ro
larysk
lasawnt
latal vsi
lathheow
lecopotdu
leilve
lengure
lenhohay
leoneth
lerface
lesobkuo
lev scy va
lezda
lezycrec
lianda
lidnogya
lifgak
litloc
liis jo
liulm
liunawy d
lófo vu
logsjo
lokosmin
lo huf
lophyo
lopnute
lorpoier
lorphohu
lude nu
lutatkie
lyavdyg
lycaff
lycel
lygefgi
lyjeehok
lykzybi
lyruvipi
lythowa
lytis
lyuwag
lyvas
mabka
macjay
madudi
maklv
malnu
malzata
marvici
mawso
mavdodu
mawfbeny
mayrut
meapaku
medcahu
meeljilb
mehuvo
meise
memjel
memyn
mencu
mepeyo
mepoohej
metupjob
mevme wif
mexcu
mi ac haft
mijas
migshyje
mihuty
mikug
minri
misosvia
misvago
miukba
mizpi
modheca
moi thi
mojmo
momil
mo othalp
mowopy
mufiho
mujcry
muthoby
mutiplyk
2000 RANDOM WORDS

myhati my-ha-ti nemaserr ne-ma-serr nubhyev nub-hyev
myhow my-ho-w nenair ne-nair muchso much-so
myifeeg my-i-feeg nenomdu ne-nom-du nucwror nuc-wror
mykna my-ka neokipi ne-o-ki-pi nuishfi nu-ish-fi
myrofed my-ro-fed neppri nepp-ri nulnot nul-not
mysagere mys-gere nerga nerga nuvnilt nuv-nil-t
mytetvif my-tet-vi-f neshghee nesh-ghee nuvo-cida nu-voc-ci-da
mythov my-tho-vo nesob ne-sob nufvoi nu-voi-zi
naegryo nae-gryo nethsknu neths-knu nuwynip nu-wy-nip
nadasthu na-das-thu neudu neu-du oabfu oab-fu
nadvof nad-vof newexam ne-wex-am oadedglu oad-ed-glu
nafba naf-ba newmlo newm-lo oakdydu oak-dy-du
nafcybcy naf-cyb-cy neyfedwy ney-fed-wy oatkohot oat-tok-hot
nafohu na-fo-hu neywypep ney-wyep oawojion ao-wo-ji-on
nagejtu na-gej-tu nicja nic-ja obsso obs-so
naibfon naib-fon niediv ni-din obvipy ob-vi-py
nakro nak-ro niezokyuy nie-zo-kyu ocecppdif o-cepp-dif
nalbru nal-blu nijwyd nij-wyd ockprad ock-prad
nalurdy nal-dydy nikinkyo ni-kink-yo ocnoy oc-noy
nammene nammene nilkmku nilm-ku octrye oct-rye
napom na-pom niness ni-ness otttin oct-tin
nashiwra na-shi-wra nissma nis-nma odcrye od-crye
nastu nas-tu nimy nim-my odckua od-cu-ka
nattva natt-va nipcu nip-cu odhece od-hece
natynhim na-tyn-him nipha nip-ha odkos od-kos
naufaw nau-waf nisgrgyo nish-grgyo odvewju od-vew-ju
navfebdja nav-feb-da nisna nis-na odwreec od-wreec
navflent na-wflent niwair ni-wair ofcai of-cai
nayityha na-yi-ty-ha noocar noo-car ofnacyc of-nay-cy
nakaroci nay-kar-ci nocci noc-ci ofnic of-nic
neegpyrea ni-c-pry nocaete no-cae-te ofway of-way
negra ne-g-ra noclot noc-lo-t ofypad of-yad
necnccucu ne-cu-cu nocoss noc-coss ofyewn of-yewn
neereedu nee-re-deu nodyo noo-dyo ogbra og-bra
neerteoc neer-g-tec nothah nof-tha ogfata og-fa-ta
nefri ne-fri noisspa nois-s-pa ogiwru o-i-g-wru
negagut ne-ga-gut nokroi nok-roi ogjipca og-jip-ca
negip ne-gip nolsht nolsht oiboa oy-boa
negot ne-got noogo noo-go olvebs ol-vebs
nehidpro ne-hid-pro nopykfa no-pyk-fa oighcabo oigh-ca-bo
nehiru ne-hi-ru noudcen noud-cen oighir oig-hir
neice neice nurycel nou-ry-cell oimnib oim-nib
neipvan neip-van novcias nov-ci-as oimut oi-mut
nelrie nel-rie noxna nox-na oisckuma oisck-u-ma
nelshjers nelsh-je-er noycz noycz oiskasa ois-ka-sa
neltoag nel-to-ag noydu noy-du oitha oit-ha
oitques oit-ques
oitredi oit-re-di
ojenty o-jen-ty
ojheibu oj-hei-bu
ojikgi o-jik-gi
ojkojdy oj-koj-dy
ojojoga o-jojo-ga
ojphov oj-phov
ojstro oj-stro
ojugcry o-jug-cry
otkria ot-kri-a
otlif ot-lif
otost o-tost
otvayd ot-vayd
otynro o-tyn-ro
oudti oud-ti
ourhogs our-hogs
ourri our-ri
ouryjau ou-ry-jau
ovoyfen ov-cy-fen
ovglyt ov-glyt
ovgroo ov-groo
ovitte o-vitte
ovrock o-vo-je-tig
ovri o-vri-so
ovsmiri ov-smi-ri
ovteyo ov-te-yo
ovvvuc ov-vuc
owjope ow-jope
owciowo ow-ci-o-vo
oydjal oyd-ja-li
oyouvers o-you-vers
oytjejf oyt-je-fi
ozovgu o-zu-ov-gu
padtra pad-tra
paipl paipl
pajles paj-les
pakyith pak-yith
pauweu pauv-weu
pawdedo pawd-e-do
pawubi pa-wu-bi
paywu pay-wu
pedoghof pe-do-ghof
pedti ped-ti
pee-tieg pee-tieg
pefcarr pef-carr
pejvonga pej-von-ga
pekacag pek-cag
pesdray pes-dray
peshegha pes-he-gha
pessmu pes-shmu
pexgeuxa pex-geux-a
phache phache
phacwef phac-wef
phagfi phag-fi
phapvez phap-vez
phea-glo phea-glo
pheapomp pheapomp
pheli phel-ti
phijca phij-ca
phlegra phleg-va
phless phless
phlef phlef
phubilst phu-bilist
phnogso phu-nogs-o
phuwo phu-wo
pictten pict-ten
pijted pij-ted
pingsvu pings-vu
piyiffe pi-yiffe
plauht plauht
plecpu plec-pu
pleett pleett
plivu pli-vu
ployhat ploy-hat
plydahoi ply-da-hoi
pokij pok-ji
potheffo po-theff-fi
povboula pov-bou-la
pozygiti poz-cyg-ti
pracfror prac-frobr
pratardo pra-tar-do
projr projr
proucha prou-cha
pryee pryee
pryin pry-in
pryitkek pry-it-kekk
puits pu-its
pujbiagy puj-bi-ag-y
punipciv pu-nip-civ
purvmyoj purv-myoj
pusadeta pu-sa-de-ta
pyadvu pyad-vu
pydaff py-daff
pydetjtu pyd-tej-tu
pyjukra pyj-uk-ra
pyppocyo py-pyo-co
pyramid py-rab-mal
pyris py-ris
pyuijhi pyu-ij-hi
qua-gro qua-gro
2000 RANDOM WORDS  (concluded)

<table>
<thead>
<tr>
<th>Word 1</th>
<th>Word 2</th>
<th>Word 3</th>
</tr>
</thead>
<tbody>
<tr>
<td>yepros</td>
<td>yep-ros</td>
<td>yipweksa yip-weks-a</td>
</tr>
<tr>
<td>yerelifs</td>
<td>ye-re-lifs</td>
<td>yitjeyef yit-je-yef</td>
</tr>
<tr>
<td>yesfryac</td>
<td>yes-fryac</td>
<td>yitnesju yit-nes-ju</td>
</tr>
<tr>
<td>yesps</td>
<td>yesps</td>
<td>yivgadso yiv-gads-o</td>
</tr>
<tr>
<td>yetmawoi</td>
<td>yet-ma-woi</td>
<td>yivos yi-vos</td>
</tr>
<tr>
<td>yevon</td>
<td>ye-von</td>
<td>yivvi yiv-vi</td>
</tr>
<tr>
<td>yewlibol</td>
<td>yew-li-bol</td>
<td>yoarmoi yoar-moi</td>
</tr>
<tr>
<td>yewnshlo</td>
<td>yewn-shlo</td>
<td>yobcruro yob-cru-ro</td>
</tr>
<tr>
<td>yexpli</td>
<td>yex-pli</td>
<td>yocefkan yo-cef-kan</td>
</tr>
<tr>
<td>yibfeem</td>
<td>yib-feem</td>
<td>yoogi yoc-gi</td>
</tr>
<tr>
<td>yibluv</td>
<td>yib-luv</td>
<td>yocuct yo-cuct</td>
</tr>
<tr>
<td>yibnen</td>
<td>yib-nen</td>
<td>yogieth yo-geieth</td>
</tr>
<tr>
<td>yictryti</td>
<td>yic-tryti</td>
<td>yogway yog-way</td>
</tr>
<tr>
<td>yidbogdo</td>
<td>yid-bog-do</td>
<td>yombo yom-bo</td>
</tr>
<tr>
<td>yidjetli</td>
<td>yid-jet-li</td>
<td>yomciub yom-ci-ub</td>
</tr>
<tr>
<td>yiemtak</td>
<td>yiem-tak</td>
<td>yopojvob yoo-poj-vob</td>
</tr>
<tr>
<td>yigfrale</td>
<td>yig-frale</td>
<td>yororodi yor-roc-di</td>
</tr>
<tr>
<td>yijwis</td>
<td>yij-wis</td>
<td>yotylica yo-ty-li-ca</td>
</tr>
<tr>
<td>yikeej</td>
<td>yi-keej</td>
<td>yovshmu yov-shmu</td>
</tr>
<tr>
<td>yiklo</td>
<td>yik-lo</td>
<td>yoywufa yoy-weu-fa</td>
</tr>
<tr>
<td>yilak</td>
<td>yil-lak</td>
<td>yuebkovu yueb-ko-ru</td>
</tr>
<tr>
<td>yilvdu</td>
<td>yilv-du</td>
<td>yuetyo yue-tyo</td>
</tr>
<tr>
<td>yufugha</td>
<td>yu-fu-gha</td>
<td>yugmopyz yug-mo-pyz</td>
</tr>
<tr>
<td>yuhienfa</td>
<td>yu-hien-fa</td>
<td>yuitdib yu-it-dib</td>
</tr>
<tr>
<td>yunny</td>
<td>yun-n-y</td>
<td>yuondyd yu-on-dyd</td>
</tr>
<tr>
<td>yupashty</td>
<td>yup-shty</td>
<td>yupwijat yup-ji-jat</td>
</tr>
<tr>
<td>yureppri</td>
<td>yu-repp-ri</td>
<td>yurytha yu-ry-tha</td>
</tr>
<tr>
<td>yuyitboc</td>
<td>yu-yit-booc</td>
<td>zaney za-ney</td>
</tr>
<tr>
<td>zatshdu</td>
<td>zatsh-du</td>
<td>zatvel zat-vel</td>
</tr>
<tr>
<td>zefphlie</td>
<td>zef-phlie</td>
<td>zeyel ze-yel</td>
</tr>
<tr>
<td>zodagbo</td>
<td>zo-dag-bo</td>
<td>zujuce zu-juce</td>
</tr>
<tr>
<td>zwitufe</td>
<td>zwi-ute</td>
<td>zwudrul zu-drul</td>
</tr>
<tr>
<td>zyipumpa</td>
<td>zy-i-pun-pa</td>
<td>zylocwyt zy-loc-wyt</td>
</tr>
</tbody>
</table>
APPENDIX V

STATISTICS

In Section IV complete data was presented for an analysis of random words of eight letters. The two figures in this appendix are the probability distribution curves for words of six and ten letters, similar to that shown in figure 4 for eight letter words.
Figure 6. Distribution of Probabilities of 5893 Six Letter Words
Figure 7. Distribution of Probabilities of 1039 Ten Letter Words

MOST PROBABLE
1.81 x 10^{-10}

p[ x(i) ]

10^{-16}
10^{-15}
10^{-14}
10^{-13}
10^{-12}
10^{-11}
10^{-10}

1 100 200 300 400 500 600 700 800 900 1000
APPENDIX VI

DOCUMENTATION

The documentation on the programs contained in this appendix is in the form of Multics Programmers' Manual command and subroutine descriptions. [3] The individual write-ups are in alphabetical order.
The columns command will reformat a segment consisting of short lines into columns that read vertically down the page. The number of columns that will appear depends on the length of the longest line in the segment and the length of the output line. The reformatted segment is printed on the terminal with blank lines between pages, or can be stored for dprinting.

**Usage**

```
columns pathname -control_args-
```

1) **pathname**
   Name of the segment to be reformatted. Lines in the segment may be of any length up to 132 characters, and all lines need not be the same length.

2) **control_args** are one or more of the following:

   - **-segment, -sm** If present, this argument specifies that the reformatted segment will be stored in a segment called `pathname.columns`, in a format suitable for dprint. If this argument does not appear, output will be printed on the terminal.

   - **-line_length nn, -ll nn**
     If present, `nn` is the length of line to be used for output. This must be a number in the range 1 to 132. If this argument is missing, the length of line used will be 132 for the `-segment` option, or the length of the terminal output line if `-segment` is not specified. If the user is absentee or `file_output` is being used, a length of 132 will be used.

   - **-page_length nn, -pl nn**
     This argument sets the length of the page produced by columns. If output is to the terminal, `nn` lines will be printed on each page, and blank lines will be used to pad the end of each page up to a total of 66 lines per page. If output is to a segment, a page will be

133
columns

MULTICS PROGRAMMERS' MANUAL

Page 2

ejected every nn lines. The default page length is 60 lines. This control argument is incompatible with the -no_pagination control argument.

-adjust, -ad specifies that the blank space between the columns is to be adjusted so that the maximum amount of white space appears between the columns. If this control argument does not appear, there will always be one blank space between the columns. Note that the number of columns to be printed is not affected by the use of this control argument. The only effect is to possibly add some blanks between the columns that would otherwise appear at the end of each line of output.

-no_pagination, -npgn

This argument specifies that the output is not to be paged, i.e., the page length is assumed to be infinite. This argument is useful for terminal output when page breaks are not desired, and avoids extra blank lines at the end of the last page.

Notes

The command first determines how many columns can fit on a line by scanning the segment for the longest line, and using that length as the width of each column. If -adjust is not specified, there will be one blank space between columns, otherwise any extra space will be inserted between the columns. Lines from the input segment that are shorter than the longest line will be left justified in the columns.

When the -segment option is not specified (and -no_pagination is not specified), columns will put 60 lines per page, with 3 blank lines at the top and bottom of each page. When -segment is specified without -no_pagination, there will be 60 lines per page with no blank lines between pages (NEWPAGE characters are inserted into the output segment to eject a page when dprinting).

When dprinting the output segment, the -no_endpage option should be specified for the dprint command. This is necessary to avoid extra blank pages because columns formats its own pages.
Warning

This command expands tabs into spaces properly, but treats backspaces and all other control characters as single characters. Generally, if the segment contains any control characters (other than tabs and newlines) the columns on the output will not line up properly.
Subroutine

**Name:** convert_word_

This subroutine is used to convert the random word array returned by random_word_ to ASCII.

**Usage**

```plaintext
dcl convert_word_ entry ((0:* ) fixed bin, (0:* ) bit(1) aligned,
                  fixed bin, char(*), char(*));

call convert_word_ (word, hyphenated_word, word_length,
                  ascii_word, ascii_hyphenated_word);
```

1) **word**

   Array of random units returned from a previous call to random_word_. (Input)

2) **hyphenated_word**

   Array of bits indicating where hyphens are to be placed, returned from random_word_. (Input)

3) **word_length**

   Number of units in word, returned from random_word_. (Input)

4) **ascii_word**

   This string will contain the word, left justified, with trailing blanks. This string should be long enough to hold the longest word that may be returned. This is normally the value of "maximum" supplied to random_word_. (Output)

5) **ascii_hyphenated_word**

   This string will contain the word, with hyphens between the syllables, left justified within the string. The length of this string should be at least 3*maximum/2+1 to guarantee that the hyphenated word will fit. (Output)

**Entry:** convert_word_$no_hyphens

This entry can be used to obtain the ASCII form of a random word without the hyphenated form.
Usage

dcl convert_word_$no_hyphens ((0:* ) fixed bin, fixed bin,
char(*));

call convert_word_$no_hyphens (word, word_length, ascii_word);

Arguments are the same as above.
**Name:** convert_word_  

This subroutine facilitates printing of the hyphenated word returned from a call to hyphenate_.

**Usage**

dcl convert_word_char_ entry (char(*), (*) bit(1) aligned, fixed bin, char(*) varying);

call convert_word_char_ (word, hyphens, last, result);

1) **word**  
This string is the word to be hyphenated. (Input)

2) **hyphens**  
This is the array returned from a call to hyphenate_ that marks characters in word after which hyphens are to be inserted. (Input)

3) **last**  
This is the status code returned from hyphenate_. If negative, the result will be the original word, unhyphenated, with ** following it. If positive, the word will be returned hyphenated, but with an asterisk preceding the last’th character. If zero, the word will be returned hyphenated without any asterisks. (Input)

4) **result**  
This string contains the resultant hyphenated word. (Output)
Command

**Name:** digram_table_compiler, dtc

This command compiles a source segment containing the digrams for the random word generator and produces an object segment with the name "digrams".

**Usage**

`digram_table_compiler pathname -option-`

1) `pathname` is the pathname of the source segment. If the suffix ".dtc" does not appear, it will be assumed. Regardless of the name of the source segment, the output segment will always be given the name "digrams" and will be placed in the working directory.

2) `option-` may be the following:

   - `-list, -ls` lists the compiled table on the terminal. The table will be printed in columns to fit the terminal line length. If `file_output` is being used, lines will be 132 characters long.

   - `-list n, -ls n` lists the table as above, but uses `n` as the number of columns to print. Each column occupies 14 positions, thus a value of 5 will cause 5 columns to be printed, each line being 70 characters long. This option is useful when `file_output` is being used, so that the lines produced are not too long to fit on the terminal to be used to print the output file.

**Notes**

The compiler makes an attempt to detect inconsistent combinations of attributes, as well as syntax errors. If an error is encountered during compilation, processing of the source segment will continue if possible. The digrams segment in case of an error will be left in an undefined state.

141
During compilation, the ALM assembler is used. At that point the
letters "ALM" will be printed on the terminal. If compilation was
successful, no other messages should appear.

The listing produced by digram_table_compiler is in a format
suitable for printing on the terminal -- not for dprinting. This is
because blank lines are used for page breaks, instead of the "new
page" character as recognized by dprint.

Syntax

The syntax of the source segment is specified below. Spaces are
meaningful to this compiler and a space is only allowed where
specified as <space>. The new line character is indicated as
<new line>.

```
<digram table>::= <unit specs>;<[<new line>]<digram specs> $
<unit specs>::= <unit spec>[<delim><unit spec>]|...
<delim>::= ,[<new line>]<new line>
<unit spec>::= <unit name>[<not begin syllable>[<no final split>]]
<digram spec>::= [begin]<begin><not begin><break><prefix>[<suffix>[<end>[<not end>]]]
<unit name>::= <letter>[<letter>]
<letter>::= a|b|c|d|e|f|g|h|i|j|k|l|m|n|o|p|q|r|s|t|u|v|w|x|y|z
<not begin syllable>::= <bit>
<no final split>::= <bit>
<begin>::= <bit>
<not begin>::= <bit>
<brack>::= <bit>
<prefix>::= <space>|-
<suffix>::= <space>|+<end>::= <bit>
<not end>::= <bit>
<bit>::= <space>|1
```

The first part of the <digram table> consists of definitions of
the various units that are to be used and their attributes. The units
are defined as one or two-letter pairs, and the order in which they are defined is unimportant. For each unit, the attributes \(<\text{not begin syllable}>\) and \(<\text{no final split}>\) may be specified. In addition, if \(<\text{unit name}>\) is \(a\), \(e\), \(i\), \(o\), or \(u\), the "vowel" attribute is set. If the unit is \(y\), the "alternate vowel" attribute is set. A \(<\text{bit}>\) is assumed to be zero if specified as \(<\text{space}>\), or one if specified as \(1\).

The second part of \(<\text{digram table}>\) specifies all possible pairs of units and the attributes for each pair. The order in which these pairs must be specified depends on the order of the \(<\text{unit specs}>\) as follows:

Number the \(<\text{unit specs}>\)s from 1 to \(n\) in the order in which they appeared in \(<\text{unit specs}>\). The first \(<\text{digram spec}>\) must consist of the pair of units numbered \((1,1)\), the second \(<\text{digram spec}>\) is the pair \((1,2)\), etc., and the last \(<\text{digram spec}>\) is the pair \((n,n)\). All pairs must be specified, i.e., there must be \(n^2\) \(<\text{digram spec}>\)s. The \(<\text{bit}>\)s preceding or following each pair set the attributes for that pair as shown. The \(<\text{prefix}>\) and \(<\text{suffix}>\) indicators are set to 1 if specified as \("-"\). If \(<\text{suffix}>\) is specified as \("+"\), the "illegal pair" indicator will be set, and no other attributes may be specified for that \(<\text{digram spec}>\).

Example

The following is a very short example of a \(<\text{digram table}>\). Only four units are defined, "a", "b", "sh" and "e". The letter "e" is given the "no final split" attribute, the pair "aa" is given "illegal pair", the pair "ae" is given the "not begin", "break", and "not end" attributes, etc.

\[
\begin{align*}
\text{a, b, sh, e} & \ 1; \\
\text{aa, ab, ash, 11 ae} & \ 1 \\
\text{ba, 1 bb, 11 bsh} & \ 1, \text{be} \\
\text{sha, 11 shb} & \ 1, \text{shsh+, she, ea, eb, esh, ee} \\
\end{align*}
\]

Assume the above segment was named "dt.dtc". Below is an example of the command used to compile and list the table produced for dt.
```plaintext
digram_table_compiler

MULTICS PROGRAMMERS' MANUAL

Page 4

digram_table_compiler dt -ls
ALM

<table>
<thead>
<tr>
<th>1 a 0010</th>
<th>2 b 0000</th>
<th>3 sh 0000</th>
<th>4 e 0110</th>
</tr>
</thead>
<tbody>
<tr>
<td>000  aa 00</td>
<td>000  ba 00</td>
<td>000  sha 00</td>
<td>000  ea 00</td>
</tr>
<tr>
<td>000  ab 00</td>
<td>010  bb 00</td>
<td>011  shb 01</td>
<td>000  eb 00</td>
</tr>
<tr>
<td>000  ash 00</td>
<td>011  bsh 01</td>
<td>000  shsh 00</td>
<td>000  esh 00</td>
</tr>
<tr>
<td>011  ae 01</td>
<td>000  be 00</td>
<td>000  she 00</td>
<td>000  ee 00</td>
</tr>
</tbody>
</table>

The first line of output lists the individual units. The number preceding the unit is the unit index. The four bits following the unit are respectively:

not begin syllable
no final split
vowel
alternate vowel

Following the unit specifications are the digram specifications. Preceding each digram are three bits and a space (or possibly a "-"") with meanings corresponding to those specified in the source segment as follows:

begin
not begin
break
prefix (if "-" appears)

Immediately following each digram is a field which may be blank, "-", or "+". If "+", the "illegal pair" flag is set. Otherwise, the meaning of the "-" and following two bits are as follows:

suffix (if "-" appears)
end
not end
```
Name: print_digram_table, pdt

This entry merely prints the digram table on the terminal, assuming that it has already been compiled successfully. The segment "digrams" is assumed to be located in the working directory.

Usage

print_digram_table -n-

1) n is the number of columns in which to print the table. If not specified, the maximum number of columns that will fit in the terminal line will be used. Each column occupies 14 positions. If file_output is being used, the terminal line width is assumed to be 132.

Notes

This entry performs the same function as the -list option of digram_table_compiler.
Subroutine

Name: generate_word_

This subroutine returns a random pronounceable word as an ASCII character string. It also returns the same word split by hyphens into syllables as an aid to pronunciation.

Usage

```plaintext
declare generate_word_ entry (char(*), char(*), fixed bin, fixed bin);

call generate_word_ (word, hyphenated_word, min, max);
```

1) word is the random word, padded on the right with blanks. This string must be long enough to hold the word (at least as long as max). (Output)

2) hyphenated_word is the same word split into syllables. The length of this string must be greater than max to allow for the hyphens. A length of 3*max/2 + 1 will always be sufficient. (Output)

3) min is the minimum length of the word to be generated. This value must be greater than 3 and less than 21. (Input)

4) max is the maximum length of the word to be generated. The actual length of the word will be uniformly random between min and max. The value of max must be greater than or equal to min, and less than 21. (Input)

Note

Each call to generate_word_ should produce a different random word, regardless of when the call is made. However, as with any random generator, there is no guarantee that there will be no duplicates. The probability of duplication is greater with shorter words.
Entry: generate_word_$init_seed

This entry allows the user to specify a starting seed for generating random words. If a seed is specified, the exact same sequence of random words will always be generated on subsequent calls to generate_word_ providing the same values of min and max are specified. If this entry is not called in a process, the value of the clock is used as the initial seed on the first call to generate_word_, thereby "guaranteeing" different sequences of words in different processes.

Usage

```
declare generate_word_$init_seed entry (fixed bin(35));

call generate_word_$init_seed (seed);
```

1) seed is the initial seed value. If zero, the system clock will be used as the seed. (Input)
**Name:** generate_words, gw

This command will print random pronounceable "words" on the user's terminal.

**Usage**

    generate_words -control_args-

1) control_args may be selected from the following:

    nwords          is the number of words to print. If not specified, one word is printed.
    -min n           specifies the minimum length, in characters, of the words to be generated.
    -max n           specifies the maximum length of the words to be generated.
    -length n, -ln n specifies the length of the words to be generated. If this argument is specified, all words will be this length, and -min or -max may not be specified.
    -hyphenate, -hph causes the hyphenated form (divided into syllables) of each word to be printed alongside the original word.
    -seed SEED       On the first call to generate_words in a process, the system clock is used to obtain a starting "seed" for generating random words. This seed is updated for every word generated, and subsequent values of the seed depend on previous values (in a rather complex way). If the -seed argument is specified, SEED must be a positive decimal integer. For a given value of SEED, the sequence of random words will always be the same providing the same length values are specified. When no -seed argument is specified, the last value of the updated seed from the previous call to
generate_words will be used. To revert back to using the system clock as the seed, specify a zero value for SEED, i.e., -seed 0.

Notes

If neither -min, -max, nor -length are specified, the defaults are -min 6 and -max 8. In all other cases, the defaults are -min 4 and -max 20.

If -length is not specified, the lengths of the random words will be uniformly distributed between min and max. Words generated are printed one per line, with the hyphenated forms, if specified, lined up in a column alongside the original words.
Name: get_line_length, gll

This command or active function returns the current length of a line on the stream user_output. When used as a command, the line length is printed on the terminal. When used as an active function, the line length is returned as a character string.

Usage

get_line_length
[get_line_length]

Notes

The length of the line is obtained from the modes supplied to the tw_IOSIM. If user_output is attached through some other interface module (such as file_ when using file_output) the line length may be undefined. If the line length can not be obtained, a default length of 132 will be returned.
get_line_length_

Subroutine

Name: get_line_length_

This subroutine returns the current length of a line on the stream user_output.

Usage

dcl get_line_length_ entry returns (fixed bin);

ll = get_line_length_();

1) ll is the length of the line.

Notes

The length of the line is obtained from the modes supplied to the tw_IOSIM. If user_output is attached through some other interface module (such as file when using file_output) the line length may be undefined. If the line length can not be obtained, a default length of 132 will be returned.
Command

Names: hyphen_test, ht

This command uses the random word generator (the same one used by generate_words) to divide words into syllables. Words are printed on the terminal with hyphens between the syllables.

Usage

    hyphen_test -control_arg- -word1- ... -wordn-

1) control_arg      may be -probability (-pb), specifying that the probability of each of the words that follows be printed alongside the hyphenated word.

2) wordi.           are one or more words to be hyphenated. A word may consist of three to twenty alphabetic characters, only the first of which may be uppercase.

Notes

The control argument may appear anywhere in the command line. However, it only applies to words that follow. Words preceding the option will be hyphenated but no probabilities will be calculated.

If a word contains any illegal characters, or is not of three to twenty characters in length, the word will be printed unhyphenated, followed by **.

If the word could not be completely hyphenated because it was considered unpronounceable, an asterisk (*) will be printed out in front of the first character that was not accepted. The part of the word before the asterisk will be properly hyphenated.

The calculated probability is the probability that the word would have been generated by generate_words, assuming generate_words was requested to generate a word of that length only. If a range of lengths is requested of generate_words, each length has equal probability. For example, if generate_words is called to generate words of 6, 7, or 8 characters, there is a 33% probability that a
given word will have 8 characters. If hyphen_test is then asked to calculate the probability of a given 8 letter word, that probability should be divided by 3 to obtain the correct probability for the case of three possible lengths.
Name: hyphenate_

This subroutine attempts to hyphenate a word into syllables.

Usage

dcl hyphenate_ entry (char(*), (*) bit(1) aligned, fixed bin);
call hyphenate_ (word, hyphens, code);

1) word This is a left justified ASCII string, 3 to 20 characters in length. This string must contain all lowercase alphabetic characters, except the first character may be uppercase. Trailing blanks are not permitted in this string. (Input)

2) hyphens This array will contain a "1"b for every character in the word that is to have a hyphen following it. (Output)

3) code This is a status code, as follows:

   0 word has been successfully hyphenated.
  -1 word contains illegal (non alphabetic or uppercase) characters.
  -2 word was not from three to twenty characters in length.

Any positive value of code means that the word couldn't be completely hyphenated. In this case, code is the position of the first character in word that was not acceptable. The part of the word before code will be properly hyphenated. (Output)

Notes

This subroutine uses random_word_ to provide the hyphenation. It does this by calling random_word__$give_up and supplying its own version of random_unit and random_vowel that return specified units (of the particular word to be hyphenated) instead of random units.
The word supplied to hyphenate_ is first transformed into units by translating pairs of letters into single units if a 2-letter unit is defined for the pair, and then by translating the remaining single letters into units. See the write-ups of random_word_ and random_unit_ for a description of units. If any units of the word are refused by random_word_, hyphenate tries to determine if the refused unit was a 2-letter unit. If this is the case, then the 2-letter unit is broken into two 1-letter units and random_word_ is called again. In rare cases, hyphenate_ is not able to determine which 2-letter unit is at fault, and will return a status code indicating that the word is unpronounceable, when, in fact, it could have been properly divided by breaking up a 2-letter unit.

**Entry: hyphenate_$probability**

This entry returns information as above, but also supplies the probability of the word having been generated at random by generate_word_ or random_word_generator_. The assumption is made that generate_word_ or random_word_generator_ was asked to supply a word of exactly the same length as the word given to hyphenate_, rather than a range of lengths. If a range of lengths was asked of generate_word_, the probability must be divided by the number of different lengths (all lengths are equally probable).

**Usage**

dcl hyphenate_entry (char(*), (* bit(1) aligned, fixed bin, float bin);

    call hyphenate_ (word, hyphens, code, probability);

1) to 3) are as above.

4) probability is the probability as defined above. (Output)
Notes

If the supplied word is illegal (i.e. code is not zero), the probability will be returned as zero.

Entry: hyphenate_$debug_on, hyphenate_$debug_off

These entries set and reset a switch that causes hyphenate_probability to print, on user_output, all units (see random_word_ and random_unit_ for a description of units) that are illegal in a given position of the word. This entry is useful for debugging a digram table for random_word_.

Usage

dcl hyphenate_$debug_on entry;
dcl hyphenate_$debug_off entry;

call hyphenate_$debug_on;
call hyphenate_$debug_off;

Notes

An example of the output produced is as follows. The assumption is that hyphenate_probability is invoked by the hyphen_test command using the -probability option.

hyphenate_$debug_on
hyphen_test -probability fish
x,ck,f; b,c,d,f,g,h,j,k,m,n,p,s,t,v,w,x,y,z,gh,ph,
rh,sh,th,wh,qu,ck,i_; i,rh,wh,qu,sh;
fish 6.04127576e-5

In the above example, the units x and ck are shown to have been illegal as the first unit of the word, and the unit f, (underlined) is the first unit of the word that was accepted. All other units that were not printed are legal as the first unit of the word. Following the semicolon after f are the units that are illegal in the second position of the word (assuming that f is the first unit). Then i is shown as the legal unit that is taken from the word "fish". This repeats for each position of the word, ending in the legal unit sh (note only one underline).
If the supplied word is illegal, the last underlined letter in the output is (usually) the letter that was not accepted. In cases where hyphenate_ has to split up a 2-letter unit, the word will be shown to start over from the beginning.
Subroutine

**Name:** random_unit_

This subroutine provides a random unit number for random_word_ based on a standard distribution of a given set of units. It is referenced by the generate_word_ subroutine as an entry value that is passed in the call to random_word_. This subroutine assumes that the digram table being used by random_word_ is a standard table. The digram table itself is not referenced by this subroutine.

**Usage**

```plaintext
declare random_unit_ entry (fixed bin);

call random_unit_ (unit);
```

1) unit is a number from 1 to 34 that corresponds to a particular unit as listed in Notes below. (Output)

**Notes**

The table below contains the units that are assumed specified in the digrams supplied to random_word_. Shown in the table are the unit number, the letter or letters that unit represents, and the probability of that unit number being generated.

<table>
<thead>
<tr>
<th>Unit</th>
<th>Probability</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 a</td>
<td>.04739</td>
</tr>
<tr>
<td>2 b</td>
<td>.03792</td>
</tr>
<tr>
<td>3 c</td>
<td>.05687</td>
</tr>
<tr>
<td>4 d</td>
<td>.05687</td>
</tr>
<tr>
<td>5 e</td>
<td>.05687</td>
</tr>
<tr>
<td>6 f</td>
<td>.03792</td>
</tr>
<tr>
<td>7 g</td>
<td>.03792</td>
</tr>
<tr>
<td>8 h</td>
<td>.02844</td>
</tr>
<tr>
<td>9 i</td>
<td>.04739</td>
</tr>
<tr>
<td>10 j</td>
<td>.03792</td>
</tr>
<tr>
<td>11 k</td>
<td>.03792</td>
</tr>
<tr>
<td>12 l</td>
<td>.02844</td>
</tr>
<tr>
<td>13 m</td>
<td>.02844</td>
</tr>
<tr>
<td>14 n</td>
<td>.04739</td>
</tr>
<tr>
<td>15 o</td>
<td>.04739</td>
</tr>
<tr>
<td>16 p</td>
<td>.03792</td>
</tr>
<tr>
<td>17 q</td>
<td>.04739</td>
</tr>
<tr>
<td>18 r</td>
<td>.03792</td>
</tr>
<tr>
<td>19 s</td>
<td>.03792</td>
</tr>
<tr>
<td>20 t</td>
<td>.04739</td>
</tr>
<tr>
<td>21 u</td>
<td>.02844</td>
</tr>
<tr>
<td>22 v</td>
<td>.03792</td>
</tr>
<tr>
<td>23 w</td>
<td>.03792</td>
</tr>
<tr>
<td>24 x</td>
<td>.00474</td>
</tr>
<tr>
<td>25 y</td>
<td>.03792</td>
</tr>
<tr>
<td>26 z</td>
<td>.00474</td>
</tr>
<tr>
<td>27 a</td>
<td>.00474</td>
</tr>
<tr>
<td>28 b</td>
<td>.00474</td>
</tr>
<tr>
<td>29 c</td>
<td>.00474</td>
</tr>
<tr>
<td>30 d</td>
<td>.00948</td>
</tr>
<tr>
<td>31 e</td>
<td>.00948</td>
</tr>
<tr>
<td>32 f</td>
<td>.00948</td>
</tr>
<tr>
<td>33 g</td>
<td>.00948</td>
</tr>
<tr>
<td>34 h</td>
<td>.00948</td>
</tr>
</tbody>
</table>

**Entry: random_unit_$random_vowel**

This entry returns a vowel unit number only.

**Usage**

```plaintext
declare random_unit_$random_vowel (fixed bin);
```
call random_unit_$random_vowel (unit);

1) unit As above. (Output)

Notes
Below are listed the vowel units and their distributions.

1  a .167
5  e .250
9  i .167
15 o .167
20 u .167
24 y .083

Entry: random_unit_$probabilities

This entry returns arrays containing the probabilities of the units as listed in the table on the previous page. This entry is provided for hyphenate_$probability and any other program that might require this information. The probabilities must be computed when this entry is called, so it is suggested that the call be made only once per process and the values saved in internal static storage.

Usage

declare random_unit_$probabilities entry ((*) float bin,
(* float bin);

call random_unit_$probabilities (unit_probs, vowel_probs);

1) unit_probs This array contains the probabilities of the individual units assuming the random_unit_ entry is called to generate the random units. The value of unit_probs(i) is the probability of unit(i). (Output)

2) vowel_probs This array contains the probabilities of the units when random_vowel is called. Since there are only 6 vowels, most of these values will be zero. (Output)
Notes

A future version of random_unit_ may use different units with different probabilities. The size of the two arrays must be large enough to hold the maximum number of values that may be returned by random_unit_ (which is currently 34). Programs should not depend on the unit_index-to-letter correspondence as shown in the table. This information can be obtained by using the include file digram_structure.incl.pl1.
Subroutine

random_word_

Name: random_word_

This routine returns a single random pronounceable word of specified length. It is called by generate_word_, and allows the caller to specify the particular subroutines to be used to generate random units. For users desiring random words with an English-like distribution of letters, generate_word_ should be used.

Usage

dcl random_word_ entry ((0:*)) fixed, (0:*)) bit(1) aligned, fixed, fixed, entry, entry);

call random_word_(word, hyphens, char_length, unit_length, random_unit, random__vowel);

1) word The random word will be stored in this array starting at word(1) (word(0) will always be 0). The numbers stored will correspond to a "unit index" as described in Notes below. This array must have a length at least equal to the value of "char_length". Unused positions in this array, up to word(char_length), will be set to zero. (Output)

2) hyphens This array must be of length at least "char_length". A bit on in a position of this array indicates that the corresponding unit in "word" (including the very last unit) is the last unit of a syllable. (Output)

3) char_length Length of the word to be generated, in characters. (Input)

4) unit_length This is the length of the generated random word in units, i.e., the index of the last non-zero entry in the "word" array. The actual length of the word in equivalent characters will be the value of char_length. (Output)

5) random_unit This is the routine that will be called by random_word_ each time a random_unit is needed. The random_unit
random_word_

Page 2

routine is declared as follows:

```
dcl random_unit entry (fixed bin);
```

where the value returned is a unit index between 1 and n_units. If an English-like distribution of letters is desired, the "random_unit_" subroutine may be specified here. See Notes below. (Input)

6) random_vowel

This is the routine called by random_word_ when a vowel unit is required. This routine must return the index of a unit whose "vowel" or "alternate_vowel" bits are on. See Notes below. This routine is declared as follows:

```
dcl random_vowel entry (fixed bin);
```

If desired, the subroutine "random_unit_$random_vowel" may be specified in this place. (Input)

Notes

The word array can be converted into characters by calling convert_word_

In order to use random_word_, a digram table, contained in a segment named "digrams", must be available in the search path. This table can be created by the digram_table_compiler.

If the user supplies his own versions of random_unit and random_vowel, these subroutines will have to supply legal units that are recognized by the random_word_ subroutine. The include file "digram_structure.incl.pl1" can be used to reference the digram table to determine which units are available. If included in the source program, appropriate references to the following variables of interest in "digrams" will be generated:

```
dcl n_units fixed bin defined digrams$n_units;
```
dcl letters(0:n_units) char(2) aligned based(addr(digrams$letters));
dcl 1 rules(n_units) aligned based(addr(digrams$rules)),
   2 vowel bit(1),
   2 alternate_vowel bit(1),
   ....

where:

n_units is the number of different units.

letters(i) contains 1 or 2 characters (left justified) for the i'th unit.

rules.vowel(i), rules.alternate_vowel(i)

One of these two bits are set for the units that may be returned by a call to random_vowel.

When random_unit is called, a number from 1 to n_units must be returned. When random_vowel is called, a number from 1 to n_units, where one of the two bits in rules(i) is marked, must be returned.

Entry: random_word_$debug_on

This entry sets a switch in random_word_ that causes printing (on user_output) of partial words that could not be completed. This entry is of interest during debugging of random_word_ or for checking the consistency of the digram table prepared by the user.

Usage

dcl random_word_$debug_on entry;

call random_word_$debug_on;

Entry: random_word_$debug_off

This entry resets the switch set by debug_on.
The random_word subroutine can be used for certain special applications (such as the application used by hyphenate_), and there are certain features that help support some of these applications. The features described below are of little interest to most users.

The first feature allows the caller-supplied random_unit (and random_vowel) subroutine to find out whether random_word_ "accepted" or "rejected" the previous unit supplied by random_unit. Each time random_unit is invoked by random_word_, the value of the argument passed is the index of the previous unit that random_unit returned (or zero on the first call to random_unit in a given invocation of random_word_). The sign of the argument will be positive if this last unit was accepted. "Accepted" means that the last unit was inserted into the random word and the word index maintained by random_word_ was incremented. Once a unit is accepted, it is never removed. Thus a positive value of the unit index passed to random_unit means that a unit for the next position of the word is requested.

If the unit index passed to random_unit has a negative sign, the last unit was rejected according to the rules used by random_word_ and information supplied in the digram table. If the unit is rejected, random_word_ does not advance its word index and calls random_unit again for another unit for that same word position. With this information random_unit can keep track of the "progress" of the word being generated.

The feature described above is used by the special random_unit routine provided by hyphenate_. Since the random_unit routine for hyphenate_ is not really supplying random units (but is supplying units of the word to be hyphenated), it must know whether any particular unit is rejected by random_word_. Rejection then implies that the word is illegal according to random_word_ rules.

The second feature allows random_unit to "try" a certain unit without committing that unit to actually be used in the random word. The sign of each unit supplied to random_word_ by random_unit is checked. If the sign of the word is positive, random_word_ will
accept or reject the unit according to its rules, and will indicate this on the subsequent call to random_unit.

If the sign of the unit passed to random_word_ is negative, random_word_ will merely indicate (on the subsequent call to random_unit) whether that unit would have been accepted, but it never actually updates the word index. In other words, random_word_ always rejects the unit, but lets random_unit know whether the unit was acceptable.

This latter feature is used by hyphenate_$probability in order to determine which of all possible units are acceptable in a given position of the word. The random_unit routine used by hyphenate_$probability tries all possible units in each word position, and only allows random_word_ to accept the unit that actually appears in that position.
Name: read_table_

This subroutine is the compiler for the digram table for random_word_. It is called by digram_table_compiler.

Usage

declare read_table_ entry (ptr, fixed bin(24), returns (bit(1));

flag = read_table_ (source_ptr, bitcount);

1) source_ptr is a pointer to the source segment to be compiled. (Input)

2) bitcount is the bit count of the source segment. (Input)

3) flag is "0"b if compilation was successful. It is "1"b if an error was encountered.

Notes

If compilation was successful, the compiled table will be placed in the working directory with the name "digrams". If unsuccessful, the digrams segment may or may not have been created, and may be left in an inconsistent state (i.e., unusable by random_word_). Error messages are printed out on user_output as the errors are encountered, except that file system errors are printed on error_output.

This subroutine uses the ALM assembler for part of its work. As a result, the letters "ALM" will be printed on user_output sometime during the compilation.
APPENDIX VII

MODIFIED SOFTWARE FOR UNIFORM DISTRIBUTION

The following pages contain documentation and source listings for the two modules that have been altered to produce uniformly distributed random words as discussed near the end of Section IV. The two modules are generate_word_ and generate_words. In addition, a listing of the random number generator encipher_ is included for those interested in the algorithm used to obtain random numbers.

Except for generate_words and generate_word_, all other modules are unchanged from those shown in Appendix III and Appendix VI. These two modules have been modified in an upward compatible manner. Thus, when called as described in Appendix VI, they will perform exactly as described. In order to get uniformly distributed words, an additional entry point in generate_word_ and an additional control argument to the generate_words command have been provided.
Subroutine

**Name:** generate_word_

This subroutine returns a random pronounceable word as an ASCII character string. It also returns the same word split by hyphens into syllables as an aid to pronunciation.

**Usage**

```plaintext
declare generate_word_ entry (char(*), char(*), fixed bin, fixed bin);
call generate_word_ (word, hyphenated_word, min, max);
```

1) `word` is the random word, padded on the right with blanks. This string must be long enough to hold the word (at least as long as `max`). (Output)

2) `hyphenated_word` is the same word split into syllables. The length of this string must be greater than `max` to allow for the hyphens. A length of $3\times\text{max}/2 + 1$ will always be sufficient. (Output)

3) `min` is the minimum length of the word to be generated. This value must be greater than 3 and less than 21. (Input)

4) `max` is the maximum length of the word to be generated. The actual length of the word will be uniformly random between `min` and `max`. The value of `max` must be greater than or equal to `min`, and less than 21. (Input)

**Note**

Each call to `generate_word_` should produce a different random word, regardless of when the call is made. However, as with any random generator, there is no guarantee that there will be no duplicates. The probability of duplication is greater with shorter words.
Entry: generate_word_$init_seed

This entry allows the user to specify a starting seed for generating random words. If a seed is specified, the exact same sequence of random words will always be generated on subsequent calls to generate_word_, providing the same values of min and max are specified. If this entry is not called in a process, the value of the clock is used as the initial seed on the first call to generate_word_, thereby "guaranteeing" different sequences of words in different processes.

Usage

declare generate_word_$init_seed entry (fixed bin(35));
call generate_word_$init_seed (seed);

1) seed is the initial seed value. If zero, the system clock will be used as the seed. (Input)

Note

If the seed is a small integer, the first few words generated may not be quite as random as one might like, i.e., if 5 is specified for the value of seed the first word generated will be almost the same as when some other small integer is specified.

Entry: generate_word_$uniform

This entry is the same as generate_word_, except that the words produced are uniformly distributed. The probabilities of the words produced by generate_word_ are not all the same. This entry provides words with equal probability. The method used to generate uniformly distributed words results in a speed degradation that is worse with longer words. For eight letter words, factor of at least 10 should be expected. In addition, the set of words that may be produced is not quite as large (although certainly within 90% of the set produced by generate_word_).

Usage

declare generate_word_$uniform entry (char(*), char(*), fixed bin, fixed bin);
call generate_word_ (word, hyphenated_word, min, max);

Arguments are the same as above.
generate_words

Command

Name: generate_words, gw

This command will print random pronounceable "words" on the user's terminal.

Usage

    generate_words -control_args-

1) control_args may be selected from the following:

    nwords              is the number of words to print. If not specified, one word is printed.
    -min n             specifies the minimum length, in characters, of the words to be generated.
    -max n             specifies the maximum length of the words to be generated.
    -length n, -ln n   specifies the length of the words to be generated. If this argument is specified, all words will be this length, and -min or -max may not be specified.
    -hyphenate, -hph   causes the hyphenated form (divided into syllables) of each word to be printed alongside the original word.
    -seed SEED        On the first call to generate_words in a process, the system clock is used to obtain a starting "seed" for generating random words. This seed is updated for every word generated, and subsequent values of the seed depend on previous values (in a rather complex way). If the -seed argument is specified, SEED must be a positive decimal integer between 1 and 9999. For a given value of SEED, the sequence of random words will always be the same providing the same length values are specified. When no -seed argument is specified, the last value of the updated seed from the previous call to generate_words will be used. To revert back to
using the system clock as the seed, specify a zero value for SEED, i.e., -seed 0.

-**uniform**, -**uf**

The probability of the words produced by this command is not the same for all words, i.e., some words are more probable than others. This option changes the way in which the words are generated so that all words are equally probable. The number of different words that can result when this option is specified is a little smaller than the number of words that may be produced without this option. (One result of using this option is that the letter "q" will never appear.) The use of this argument results in a speed degradation by a factor of about 10 for eight letter words, and greater for longer words. This argument is useful when it is desirable to generate words whose probabilities are equal.

**Notes**

If neither -**min**, -**max**, nor -**length** are specified, the defaults are -**min** 6 and -**max** 8. In all other cases, the defaults are -**min** 4 and -**max** 20.

If -**length** is not specified, the lengths of the random words will be uniformly distributed between min and max. Words generated are printed one per line, with the hyphenated forms, if specified, lined up in a column alongside the original words.
This procedure enciphers an array of double words, i.e., fixed bin(71).
using the key that is provided. It has entries to both encipher and decipher:

- call encipher(key,input_array,output_array,array_length)
- call decipher(key,input_array,output_array,array_length)

where: key is fixed bin(71) key for coding
input_array[array_length] is fixed bin(71) array
output_array[array_length] is fixed bin(71) array
array_length is fixed bin(71) length (double words) of array

Coded 1 April 1973 by Roger R. Schell, Major, USAF
000010 aa 0 00002 2371 20 53 ldax apikey,* "Start with input key
000011 54
000011 mask_loop
55
000011 56
000011 aa 6 00050 7571 16 57 staq variables,6 "Create masks
000012 58 qr1 shift *Save copy of generator seed
000013 59 ar1 shift *Now generate pseudo-random number
000014 60 eraq variables,6
000015 61 staq variables,6
000016 62 qla size-shift
000017 63 als size-shift
000018 64 eraq variables,6
000019 65 staq variables,6 "Save result
000020 66
000022 67 eax6, 2,6 "Generate 9 double words
000023 68 cmpx6 18, du
000024 69 tnz mask_loop
000025 70
000025 71 *Next create 7-bit shift variables
000026 72
000026 73 eax6 0 "First 7 bits to upper A-reg
000027 74 lr1 11 "Zero for clearing half word
000027 75 eax0 0
000030 76 shift_loop:
000030 77 staq variables*1,6 "Upper A-reg is shift variable
000031 78 staq variables*1,6 "Zero lower half word
000032 79 sx10
000033 80 lls 7 "Save 7 bits in upper A-reg
000034 81 ans so000017777777
000035 82 eax6 1,6 "Generate 7 shift variables
000036 83 cmpx6 7, du
000037 84 tnz shift_loop
000038 85
000038 86 * Now that we have needed variables, apl the cipher
000039 87
000039 88
000039 89
000039 90 declaration of offsets of keying variables
000039 91 equ C0.0 "Initial cipher text from key
000040 92 equ M1.2 "Mask variables
000041 93 equ M2.4
000042 94 equ M3.6
000043 95 equ M4.8
000044 96 equ M5.10
000045 97 equ M6.12
000046 98 equ M7.14
000047 99 equ A1.16 "Amount of shift -- as address
000048 100 equ A2.17
000049 101 equ A3.18
000049 102 equ A4.19
000049 103 equ A5.20
000049 104 equ A6.21
000049 105 equ A7.22
000049 106
000049 107 lxl5 array_length,* "Get length (double words)
000049 107
000049 108 eax5 -1,5 "Check for zero or negative
000049 108
000049 109 tmi return
000049 110 eax6 0 "X6 is index into arrays
encipher_list
000043    aa 6 00050 3521 00 111  epnhp  variables=0
000044    cipher_loop:  variables=0
000044    aa 2 00000 2371 00 111  ldao bp10
000044    114
000045    "First compute select function
000046    116
000046    aa 6 00075 7771 20 117  lir  variables=45,*
000047    aa 6 00084 0371 00 118  adlao  variables=M6
000047    119  lir  variables=47,*
000050    120  eraq  variables=M7
000051    121  eax1  0,01  "Save select function
000052    122  "Compute value
000052    124  "
000052    125  ldao bp10
000053    126  lir  variables=43,*
000054    127  adlao  variables=M1
000055    128  canx1  well,du
000058    129  tns  2,10
000057    130  lir  variables=42,*
000050    131  eraq  variables=M2
000051    132  canx1  x0,du
000052    133  tns  2,10
000062    134  lir  variables=43,*
000064    135  adlao  variables=M3
000065    136  canx1  x0,du
000066    137  tns  2,10
000067    138  lir  variables=44,*
000070    139  eraq  variables=M4
000071    140  canx1  x0,du
000072    141  tns  2,10
000073    142  lir  variables=45,*
000074    143  adlao  variables=M5  "AQ contains computed key
000075    144
000075    145  epnhp  1p10,6  "Set BP -> next cipher text autokey
000076    146  eraq  aplinput_array,*
000077    147  staq  aploutput_array,*  "Return ciphered value
0000100   148  eax6  2,6  "Increment array offset
0000101   149  eax5  -1,5  "Check for end of array
0000102   150  tpl cipher_loop
0000103   151  return:  152  "Clean up the 'dirty blackboard' before returning
0000103   154
0000103   155  bool  rpt,5202  "RPT instruction
0000103   156
0000103   157  ldao  *  "Load AQ with garbage
0000104   158  eax6  0
0000105   159  vfd  8/11,2/0,1/1,7/0,1/2/rpt,6/2  "RPT instruction
0000106   160  staq  variables,6  "Overwrite keying variables
0000107   161
0000107   162  return
0000107   163
0000107   164  end
```c
/* This procedure is a modification of the standard subroutine interface to generate random words.
   A change has been made to the entry point generate_word_uniform.
   Except for that, the original functioning of generate_word_uniform is the same.
*/
generate_word : procedure (word, hyphenated_word, min, max);
  dcl word char(*)
  dcl hyphenated_word char(*)
  dcl min fixed bin
  dcl max fixed bin
  dcl (random_unit, random_unit $random_vowel) entry (fixed bin)
  dcl convert_word_entry ((0):* fixed bin, (0):* bit(1) aligned)
  dcl random_word_entry ((0):* fixed bin, (0):* bit(1) aligned)
  dcl random_word (0:20) bit(1) aligned
  dcl random_length fixed bin
  dcl unique_bits entry returns (bit(70))
  dcl encipher_entry (fixed bin(71), (* fixed bin(71), (* fixed bin(71), fixed bin)
  dcl random_unit_stat $seed fixed bin(71) external
  dcl saved_seed(1) fixed bin(71) static
  dcl first_call bit(1) static aligned init(*:1)

/* In the very first call to this procedure in a process (if the
   init_seed entry was not called), use unique_bits to set a
   random number to initialize the random seed. */
if first_call then do:
  saved_seed(1), random_unit_stat $seed = fixed (unique_bits ());
  first_call = "0";
end;

/* encipher the seed to get a random number and the next value of the seed */
call encipher_ (saved_seed(1), saved_seed, saved_seed, 1);

/* Get the length of the word desired.
   We calculate this to be a uniformly distributed random number between
   min and max. */
random_length = mod (abs (fixed (saved_seed(1), 71)), (max - min + 1)) + min;

/* Get the random word and convert it to characters */
call random_word_ (random_word, hyphens, random_length, length_in_units, random_unit, random_unit $random_vowel);
convert:
call convert_word_ (random_word, hyphens, length_in_units, word, hyphenated_word);
return;

/* This entry allows the user to set the seed. If the seed argument is zero, we
   go back to using the clock value. */
generate_word $init_seed: entry (seed);
call seed fixed bin(35);
```
generate_word_list

50 dcl whole_seed fixed bin(71);
57 dcl 1 half_seeds based (addr (whole_seed)),
58 2 (first, second) fixed bin(35);
60 if seed = 0 then first_call = /*
61 else do:
62 half_seeds = seed;
63 random_bits, saved_seed(1), random_unit_stat_stat = whole_seed;
64 first_call = /*
65 index = array_size;
66 end;
67 return;
68 */
generate_word_list

08/19/75 1017.; edit Tue

page 3

*/

69 /* This entry point generates uniformly distributed words.
70 It does this by "giving" random_word, uniformly distributed random
71 words and keeps trying until an acceptable word turns up.
72 */
73
74 The random numbers are obtained from an array of 72-bit numbers generated by
75 encipher_. This array is overlayed by 5-bit numbers to give random
76 numbers in the range -16 to +15. The random unit and random_vowel routines
77 below that supply the next random unit number only supply the numbers
78 of single-letter units. This is because it's difficult to obtain
79 a uniform distribution of words with double letter units.
80 Note that only unit numbers 32 or less are returned. Thus, if there is
81 a single letter unit with a number greater than 32, it will never
82 appear.
83 */
84
generate_word_uniform: entry (word, hyphenated_word, min, max);
85
dcl bits_size fixed bin static init (10);
86
dcl array_size fixed bin static init (144);
87
dcl index fixed bin static init(9999);
88
dcl array (array_size) based (addr(random_bits)) fixed bin(4) unsigned;
89
dcl random_bits (10) fixed bin (77) static;
90
dcl number fixed bin;
91
dcl max_number fixed bin;
92
dcl digrams*digrams external;
93
dcl digrams*units fixed bin external;
94
dcl digrams*letters external;
95
dcl digrams*rules external;
96
97 /* This array contains information about all possible pairs of units */
98
dcl 1 digrams (n_units, n_units) based (addr (digrams*digrams)),
99
dcl 2 begin bit (1),
100 dcl 2 not_begin bit (1),
101 dcl 2 end bit (1),
102 dcl 2 not_end bit (1),
103 dcl 2 break bit (1),
104 dcl 2 prefix bit (1),
105 dcl 2 suffix bit (1),
106 dcl 2 illegal_pair bit (1),
107 dcl 2 pad bit (1);
108
109 /* This array contains left justified 1 or 2-letter pairs representing each unit */
110
dcl letters (2*n_units) char (2) aligned based (addr (digrams*letters));
111
112 /* This is the same as letters, but allows reference to individual characters */
113
dcl 1 letters_split (2*n_units) based (addr (digrams*letters)),
114
dcl 2 first char (1),
115 dcl 2 second char (1),
116 dcl 2 pad char (2);
generate_word_list

1 33 /* This array has rules for each unit */
2 34
3 35 dcl 1 rules (n_units) aligned based (addr (digits$rules)),
4 36 2 no_final_split bit (1),
5 37 2 not_begin_sylable bit (1),
6 38 2 vowel_bit (1),
7 39 2 alternate_vowel_bit (1),
8 40
9 41 dcl n_units defined digrams$units fixed bin;
10 42
11 43 /* ****** end include file digram_structure.incl.pl ****** */
12
13 94
14 95 if first_call then do;
15 96  saver_seed(1) = fixed (unique_bits());
16 97 first_call = "0";
18 98 end;
19 100
20 101 /* get length of word, if a range was specified */
21 102
22 103 if max "= min then do;
23 104 random_length = max - min + 1;
24 105 max_number = divide (32, random_length, 17, 0) * random_length;
25 106 number = 33;
26 107 do while (number > max_number);
28 108 number = random_number();
29 109 end;
30 110 random_length = number - random_length * divide (number, random_length, 17, 0) * min;
31 111 end;
32 112 else random_length = max;
33 113
34 114 /* now get the random word */
35 115
36 116 try_again: call random_word_ (random_word, hyphens, random_length, length_in_units, random_unit,random_vowel);
37 117 goto convert;
38 118
39 119 /* specialized random_unit and random_vowel routines */
40 120
41 121 /* These routines return a random unit number. If the previous
42 122 unit was not accepted, they do a nonlocal goto to
43 123 try the word all over. */
44 124
45 125 random_unit: proc (n);
46 126 dcl n fixed bin;
47 127 if n < 0 then goto try_again;
48 128 loop:
49 129 n = 999;
50 130 do while (n > n_units);
51 131 n = random_number();
52 132 end;
53 133 if substr (letters(n), 2, 1) "= " then goto loop; /* keep trying until a one-letter unit is found */
54 134 end;
55 135
56 136 random_vowel: proc (n);
57 137 dcl n fixed bin;
58 138 if n < 0 then goto try_again;
59 140 loop:
60 141 n = 999;
generate_word_list

do while (n > n_units);
  n = random_number();
end;

if substr (letters(n), 2, 1) "= " then goto loop; /* keep trying until a one-letter unit is found */
if "rules.vowel(n)" then goto try_again;
end;

/* routine to generate a random number 1 to 32 */
random_number: proc returns [fixed bin(5)];
if index >= array_size then do; /* no more numbers left in array */
call encrypter_1 [saved_seed(1), random_bits, random_bits, bits_size]; /* get an array of numbers */
saved_seed(1) = random_bits (1);
index = 16;
end;
else index = index + 1;
return (array (index) + 17); /* return the next random number from array */
end;
/* This procedure is a modified version of the standard generate_words command.
   The change is to accept the -uniform control argument and to call generate_word_uniform
   when that argument is specified, instead of generate_word. Otherwise, it is identical
   to the standard version.
*/
generate_words: gw: procedure;
col cu_arg_ptr entry (fixed,ptr_freed,fixd bin35);
col cu_arg_ptr_rel entry (fixed bin, ptr, fixd bin, fixd bin35, ptr);
col cu_arg_list_ptr entry (ptr);
col argno fixed;
col new_line char(1) init(*
*);
col error_table_tabadopt external fixd bin35;
col arglen fixd bin;
col [generate_word, generate_word_uniform] entry (char(*), char(*), fixd bin, fixd bin);
col generate_word entry (char(*), char(*), fixd bin, fixd bin) variable init (generate_word);
col generate_word_uniform seed entry (fixd bin35);
col ia_s_write_ptr entry (ptr, fixd bin, fixd bin);
col argcptr ptr;
col hyphenate bit(1) init("0b");
col cv_dec_check_entry (char(*), fixd bin) returns (fixd bin35);
col max_length fixd bin init(-1); /* set to max length of words */
col min_length fixd bin init(-1); /* min length of words */
col seed_value fixd bin35 init(-1); /* value of seed typed by user */
col com_err_entry options(variable);
col i fixd, code fixd bin35 init(0);
col unique_bits entry returns (fixd bin70);
col result fixd bin;
col nwords fixd init(0);
col max_words fixd init(0);
col [arg char(arglen) based (argptr) unaligned;
col [max_hyphenated fixd bin;
col area char35; /* where output line goes */
col output_line_length fixd bin; /* length of the output line in area */
col unhyphenated_word char (maximum) based (addr(area));
col hyphenated_word char (maximum) based (hph_ptr);
col hph_ptr ptr; /* pointer to position in area where hyphenated word goes */
col arglistptr ptr;
col cu_arg_list_ptr (arglistptr);
do argno = 1 by 1 while(code = 0);
col cu_arg_ptr (argno, argstr, arglen, code);
if code = 0
then
then hyphenate = "b"
else
if arg = "-max"
then maximum_length = value("maximum");
else
if arg = "-min"
then minimum_length = value("minimum");
else
if arg = "-uniform" : arg = "uf"
generate_words.list

then generate_word = generate_word_uniform;
else
if arg = "-length" : arg = "-ln"
then do;
maximum_length = value("length");
minimum_length = maximum_length;
end;
else
if arg = "-seed" then do;
seed_value = value("seed");
if seed_value = 0 /* if seed not zero, use its characters instead of value */
then encipher generates non-random numbers if there aren't a lot of leading
zeros in the key. */
then seed_value = fixed (unspec (char(arg,4))): /* take up to first 4 characters */
call generate_word_limit_seed (seed_value);
end;
else do;
nwords = cv_dec_check (arg, result); /* look for number of words */
if result = 0 & nwords > 0
then max_words = nwords;
else call ugly (error_table & adapt, arg);
end;
end;

/* Below we decide whether minimum, maximum, both, or none have been specified,
and set their default values accordingly. */

if nwords = 0 then max_words = 1;
if minimum_length = -1
then if maximum_length = -1
then do;
minimum_length = 0;
maximum_length = 0;
end;
else minimum_length = 4;
else if maximum_length = -1
then maximum_length = 20;
if minimum_length < 4 | minimum_length > maximum_length
then maximum_length = 20 then
call ugly (0, "Bad value of lengths: \ minimum<\ maximum required.");

maximum_hyphenated = maximum_length + 2*maximum_length; /* maximum length of hyphenated word */

hph_ptr = addr (substr (area, maximum_length/2)); /* where hyphenated word will be put */
if hyphenate
then do;
substr (unhyphenated_word, maximum_length + 1, 1) = " ";
substr (hyphenated_word, maximum_hyphenated + 1, 1) = new_line;
output_line_length = maximum_length + maximum_hyphenated + 2;
end;
else do;
substr (unhyphenated_word, maximum_length + 1, 1) = new_line;
output_line_length = maximum_length + 1;
end;
/* generate max_words and write them all out */
DO i = 1 to max_words;
   call generate_word (unhyphenated_word, hyphenated_word, minimum_length, maximum_length);
   call ios_write_ptr (addr(area), 0, output_line_length);
END;

UGLY: PROCEDURE (CODEX, MESSAGE);
   DCL (CODEX, CODEX) FIXED BIN(35);
   DCL MESSAGE CHAR(*);
   CALL CON_ERR_ (CODEX, "GENERATE_WORDS", MESSAGE);
   EOTO RETURN;
END;

VALUE: PROCEDURE (NAME) RETURNS (FIXED BIN(35));
   DCL NUMBER FIXED BIN(35);
   DCL NAME CHAR(*);
   ARGNO = ARGNO + 1;
   CALL CON_ARGPTR_REL (ARGNO, ARGPTR, ARGLEN, CODE, ARLELISTPTR);
   IF CODE = 0 THEN CALL UGLY (CODE, "Value of " || NAME);
   NUMBER = CY_DEC_CHECK (ARGX, RESULT);
   IF RESULT = 0 OR NUMBER < 0 THEN CALL UGLY (0, "Bad " || NAME || " value. " || ARGX);
   RETURN(NUMBER);
END;

RETURN;
END;
REFERENCES

