Martin Kay

MORPHOLOGICAL ANALYSIS

A computer program that is intended to carry out nontrivial operations on texts in an ordinary language must start by recognizing the words that the text is made up of. This is the procedure I call *morphological analysis.* It is necessary because the linguistically interesting properties of words cannot be discovered by examining the words themselves but are associated with them in an essentially arbitrary manner. Therefore, there must be a list – what we call a *dictionary* – to define the mapping of words into linguistically interesting properties and a process to look words up in this dictionary.

Many computer programs have been written in which morphological analysis consists of nothing more than accepting any unbroken string of letters encountered in a text as a word and referring it to a dictionary. This means that, in addition to what is usually found there, the dictionary must contain plural forms of nouns, all the forms of every verb, regular or irregular, all adverbs, and so forth. A machine dictionary of English constructed on these principles would contain four to six times as many entries as a standard dictionary but some of these entries could presumably consist of little more than a reference to the standard form of the word – the singular of the noun, the infinitive of the verb, or whatever. A modern computer could easily accommodate a dictionary of English enlarged in this way and it is an attractive thing to do if only because it reduces the problem of morphological analysis almost to triviality. The increase in the size of the dictionary is more alarming in the case of a highly inflected language. There are, however, many languages for which this solution is unthinkable and many for which it is clearly undesirable. In ancient Greek, Latin, and Sanskrit, for example, it was not customary to leave spaces between words so that

Galliaestomnesdi
visainpartestres

would have been a reasonable way for Caesar to write what would be printed as

Gallia est omnes divisa in partes tres

in a modern edition. Many languages, like German, admit compounding as a productive part of the grammar so that words like

Lebensversicherungsgeselschaftsangestelter

meaning " employee of a life insurance society " can be freely invented. Under these circumstances, the policy of referring unbroken strings of letters to the dictionary will clearly be inadequate.

In general, therefore, it is necessary to recognize lexical items in a text otherwise than by the simple fact that they are bounded by spaces or other non-alphabetic characters. The possibility of words being juxtaposed without any explicit boundary must be admitted and it may even be desirable to divest the space of its special status as a separator altogether and treat it like any other member of the alphabet. This opens the possibility of treating many kinds of idioms and fixed phrases as ordinary words that happen to contain spaces or other non-alphabetic characters.

But there is more to morphological analysis than recognizing lexical items in a connected text in the absence of explicit boundary markers. In general, when lexical items are conjoined, they undergo some change of form. In English, for example, the plural of nouns is regularly formed adding an *s*. But, if the noun ends in *j*, *s*, *x*, *z*, *sh*, or *ch*, an *e* is introduced before the *s*. If the singular form ends in *y*, then this is replaced by *ies* in the plural. These changes are specified in a chapter of the grammar called *morphographemics* which is much more copious in some languages than in English. In Sanskrit, for example, morphographemic rules are applied when one word is written after another and not only when grammatical affixes are appended. Thus, for example,

rajendra

is written instead of

raja indra

because of a grammatical rule requiring $a + i$ to be replaced by *e* wherever it occurs. Notice that rules of this kind make the use of spaces to delimit words almost impossible because, in a case like this, there is no non-arbitrary way of assigning the *e* to the first or the second word.

I have been treating grammatical items like inflexional affixes on a level with other lexical items. This seems reasonable, at least for the immediate purpose which is simply to decompose a text into items that are small enough to constitute a finite list in the description of the language and which are composed into larger items by productive processes. The trouble is that texts consist of more than a concatenation of lexical items occasionally modified by the action of morphographemic rules. Words undergo productive processes formally different from, though functionally identical to, the adjunction of other lexical items. The plural that is represented in English by adding an *s* appears in other languages by the repetition of a syllable or part of a syllable with or without some change in the vowel of that syllable. In addition to prefixing and suffixing, some languages admit infixing, a process by which the string of characters representing one lexical item is interrupted by a second item. In fact, the complete variety of the morphological processes used in the languages of the world has never been surveyed.

In the remainder of this paper, I shall outline a procedure for morphological analysis of which it is not too unreasonable to hope that it will accommodate most of the languages of the world while, at the same time, being efficient enough to be considered for inclusion in practical computer systems in competition with more specialized methods that have been proposed. The procedure I shall outline has the additional advantage that it can be made to blend in an interesting way with syntactic processes that can be expected to follow.

Morphographemic rules are made available to the procedure in the form of a set of string-rewriting rules whose job is to reduce the lexical items in a text to canonical forms which can then be referred to a dictionary.

For example, a rule approximately of the form

$$i\,e\,d \rightarrow y + e\,d$$

would transform the word *tried* into *try* + *ed*. The "+" represents a boundary between a pair of lexical items; operationally, it will be a lexical item in its own right and will occur as part of no other lexical item.

Consider the string

He tried the fuses

The morphographemic rewriting component of the system proposes three forms for the word *tried* and two for the word *fuses* so that a total of six strings are delivered to the next component of the system. Only one of these, namely

He try+ed the fuse+s

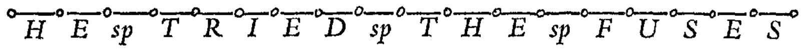
is correct. If more rewriting rules had been applied, a great many more strings would have resulted. In fact, if each of the words in a longer sentence were given two forms by the rewriting rules, then 1024 different strings would be generated.

Clearly, what is required is the ability to work with expressions with something like the following form:

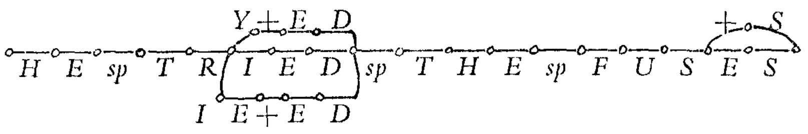
He tr(i/y+)ed the fus(es/+s)

in which the parentheses include alternative substrings separated by slashes. This should be more than just a notational convention but should reflect the inner workings of a system in which the amount of material generated and the amount of processing to be done on it is more nearly proportional to the sum than the product of the ambiguities.

Let the string to be analyzed be represented in a diagram of the following kind:



This kind of diagram is what I call a *chart.* Each letter labels an edge. There is an initial and a final vertex and the remaining vertices correspond to the points in the string at which a pair of letters meet. When the rewriting process is complete, the chart will look like this:



Each of the six strings that the morphological rewriting component must produce for this example is represented by a path from left to right through the chart. Instead of rewriting in the strict sense, the rules have caused new edges and vertices to be added to the chart. A rule like

i e d → *y + e d*

is interpreted as an instruction to look for instances of the string *ied* and to introduce a new path from the vertex before *i* to the vertex following *d* with the labels *y*, +, *e*, and *d*.

There are many simple ways of representing the same logical structure that a chart diagram represents inside a computer. One is to represent each edge by a quadruple ⟨label, character, alternate, successor⟩. Each edge has a unique label which, in the computer, can be the index of the edge in a set of three parallel arrays in which the other components are stored. The second component is the letter or other character represented by the edge. The alternate is the label of another edge incident from the same vertex. A vertex, in this representation, is simply the set of edges incident from it. The index of the first of these to be put in the chart serves also as the index of the vertex. The remaining edges are found by taking the alternate of the first edge as the second edge, the alternate of the second as the third, and so on until an edge is encountered that has no alternate. The chart displayed above, in which three rewriting rules have been applied to the string *He tried the fuses* is represented as follows:

Label	Character	Alternate	Successor
1	*H*	0	2
2	*E*	0	3
3	*sp*	0	4
4	*T*	0	5
5	*R*	0	6
6	*I*	19	7
7	*E*	0	8
8	*D*	0	9
9	*sp*	0	10
10	*T*	0	11
11	*H*	0	12
12	*E*	0	13
13	*sp*	0	14
14	*F*	0	15
15	*U*	0	16
16	*S*	0	17
17	*E*	28	18
18	*S*	0	0
19	*Y*	23	20
20	+	0	21
21	*E*	0	22

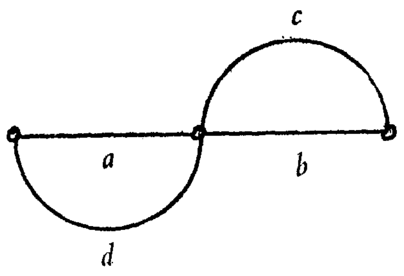
22	*D*	0	9
23	*I*	0	24
24	*E*	0	25
25	+	0	26
26	*E*	0	27
27	*D*	0	9
28	+	0	29
29	*S*	0	0

The first 18 entries represent the characters of the original string. The only changes that have been made to these are in the "alternate" column for entries 6 and 17. These correspond to the first characters of substrings to which rules have applied. Entry 6, for example, has 19 as its alternate and entries 19 through 22 represent the string *y+ed*. The successor of entry 22 is 9 indicating that *y+ed* is a replacement for the *ied* in entries 6, 7, and 8, the last of which also has 9 as its successor. The string *ied* was, in fact, rewritten by two different rules so that entry 19 also has an alternate and entries 23 through 27 represent the output of the second rule. Entry 6 is the head of an alternate chain that also contains 19 and 23, and these are indeed three edges that are all incident from the same vertex, a vertex that we can think of as represented by the number 6.

The chart comes close to achieving the economy desired of the rewriting component of the system. Notice, however, that there are still three separate edges labeled *d* preceding the second space. In order to see why this must be so, consider the following more abstract example:

1) Rewrite *a* as *d* when it precedes *b*
2) Rewrite *b* as *c* when it follows *a*

If the initial string is *ab*, then the rewriting process must deliver three strings, namely *ab*, *ac* and *db*. The set does not include *dc*. Now look at the diagrammatic representation. Interpreting the rules in the most straightforward way, one might expect to get

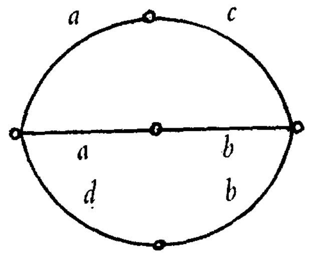


But this does include a path representing the string *dc*. If phrases like *when it precedes....* and *when it follows....* are excluded from the rules, so that we must say:

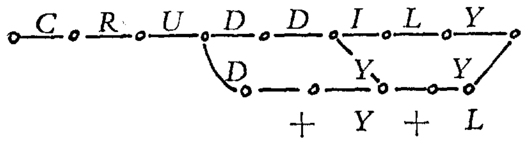
1a) Rewrite *ab* as *ac*

2a) Rewrite *ab* as *db*

we not only stress the mutual incompatibility of the two rules, but also produce a situation in which the most natural interpretation of the rules gives the correct result. We now get



It is possible for rules to operate on characters resulting from the application of previous rules. A compact statement of the rules of Sanskrit morphology would capitalize on this possibility to a large extent. In English, it might never be used in a realistic system, but it is not difficult to manufacture examples that are not hopelessly implausible even in English. Consider a word like *cruddily*. One rule might rewrite this as *cruddy+ly* and a second rule might then rewrite this as *crud+y+ly*. Part of the input to the second rule is the final *y* of *cruddy* which the first rule introduced. In the chart, this would appear as follows:

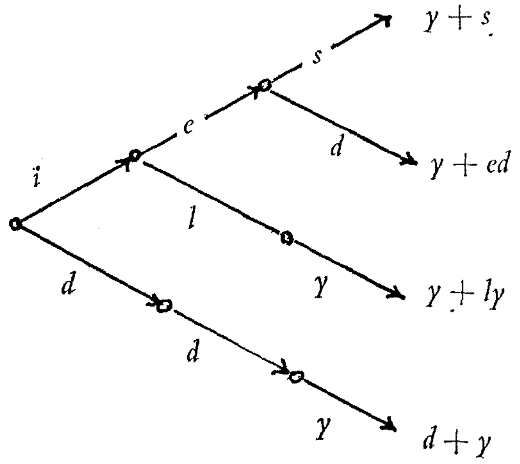


What this means is that, when substrings are being considered as candidates for rewriting, every path through the chart must be considered, including those that arise as a result of previous rewriting. This brings up the basic problem of syntactic analysis, namely, how can one guarantee that every possibility will be explored once, but no more than once. A full discussion of this problem is beyond the scope of this paper and I shall therefore limit the discussion to one simple but very general technique which can be modified in innumerable ways in the interests of efficiency.

Let us consider just four rewriting rules taken from the morphological section of a hypothetical English recognition grammar:

$$\begin{array}{llllllll}
i & e & s & \rightarrow & y & + & s & \\
i & e & d & \rightarrow & y & + & e & d \\
i & l & y & \rightarrow & y & + & l & y \\
d & d & y & \rightarrow & d & + & y &
\end{array}$$

A considerable amount of space and, as we shall see, work, can be saved by representing the rules also in diagrammatic form. Although it is, in many ways similar to a chart, I shall use the term *transition network*, or simply *network* to refer to this diagram so that it will be easy to refer to both in the same context. For the same reason, I shall refer to the lines in a transition network as *arcs* rather than edges. A network representing these four rules would be as follows:



In this representation, if several rules begin in the same way, the similar parts are only represented once and if the similar initial parts of a set of rules are matched against a set of edges, this will be a partial test of all the rules in the set. In the computer, this can be represented as follows:

Label	Character	Alternate	Successor	Replacement
1	I	7	2	
2	E	5	3	
3	S	4	0	$\rightarrow Y + S$
4	D	0	0	$\rightarrow Y + E\ D$
5	L	0	6	
6	Y	0	0	$\rightarrow Y + L\ Y$
7	D	0	8	
8	D	0	9	
9	Y	0	0	$\rightarrow D + Y$

All parts of the network can be reached from arc number 1. The set of alternates of arc number 1 correspond to sets of rules that differ in their very first character.

We are now ready to consider how a transition network can be applied to a chart in such a way as to allow all applicable rules to be identified and carried out exactly once. Like almost all non-deterministic procedures, this involves the use of a list of tasks that still remain to be done at any given moment during the execution of the procedure. This list is sometimes referred to as a *stack* because it is usual to maintain the policy of never removing from it any but the last entry made so that the last task remembered will always be the first to be carried out, and the word " stack " is reserved for lists that are used in this way. I shall use another term, namely *task list*, because I do not wish to suggest any such discipline. Indeed, it is a feature of this procedure that the tasks on the list can be carried out in any order whatsoever without altering the eventual result.

An entry on the task list is a triple ⟨arc, edge, vertex⟩. When the task is carried out, an attempt will be made to match the arc to the edge. If this completes the matching of the left-hand side of a rule with a substring in the chart, then some new material will be introduced into the chart beginning at the vertex named in the triple and ending at the vertex which is the successor of the named edge.

In greater detail, the task specified by the triple $\langle a, e, v\rangle$ is carried out as follows:

1. Let b be the alternate of a. If b is not zero, create a new task $\langle b, e, v \rangle$ and put it on the task list.
2. Consider the edge e and any other edges that there may be on its alternate chain in turn. Let d be the current edge. Carry out the process de-

scribed below for each new edge *d*. In other words, first let $d = e$, then set *d* to the alternate of *d* on each occasion unless *d* has an alternate of 0, in which case, stop.

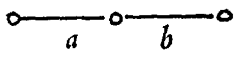
If the characters of the arc *a* and the edge *d* are not the same, do nothing. If they are the same and the arc has a replacement, then introduce a new sequence of edges into the chart to represent the replacement starting at vertex *v* and ending at the vertex which is the successor of *d*. Whether or not there is a replacement, if the arc has a successor *s* and the edge has a successor *t*, then create a new task $\langle s, t, v \rangle$ and put it on the task list.

It will be clear from this that a task can create any number of new tasks and this is why it is necessary to keep a list of them.

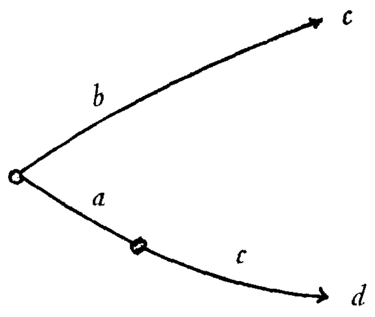
The entire process is set in motion by putting on the task list a task $\langle l, w, w \rangle$ for each vertex *w* in the chart. Recall that, for such purposes as this, a vertex is represented by the label of its first edge and, initially, there will be exactly one edge incident from each vertex. I might, therefore, just as well have said that a task is created for each edge in the chart. Tasks are now removed from the list and carried out until none remain.

The procedure, as described so far, will result in the rules being applied to the characters of the initial string and all possible results obtainable in this way being added to the chart. However, there is no provision for applying rules to substrings that arise, in whole or in part, from the application of other rules. Part of the problem can be solved by adding a new task $\langle l, n, n \rangle$ for each new vertex *n* that is added to the list as a result of applying a rule. In fact, the initial task list and these later additions would all be covered by a general requirement that a task $\langle l, n, n \rangle$ is created for every vertex *n* added to the chart, initially or later. But there remains the problem of ensuring that, when new edges are introduced incident from existing vertices, all processes that should apply to that edge do in fact apply. The difficulty is that at least some tasks will already have been applied to the edges incident from that vertex and will, by now, have disappeared without trace. A simple example will show how this can arise.

Suppose that the initial chart is as follows:



and that the rules are as in the following network:



The initial task list is as follows:

Arc	Edge	Vertex
1	1	1
1	2	2

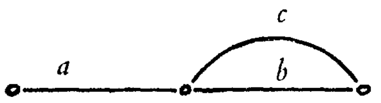
The first task is removed from the list and carried out. Since arc number 1 has an alternate, a new task is put on the list. Since we are assuming that the order of the task list is immaterial, let us add the new task at the head of the list so that we now have:

Arc	Edge	Vertex
2	1	1
1	2	2

The current edge has no alternates so that it is the only one to be considered in this task. The label on the edge is *a* and the label on the arc is *b*. These do not match and the task therefore comes to an end. The next task is ⟨2, 1, 1⟩. Arc 2 has no alternate so that no new task has to be created for it. There is only one edge to be considered and arc 2 does, in fact, have the same symbol as edge 1. There is no replacement but both arc and edge have successors. A new task is therefore created, and we place it, as before, at the head of the list, which is now as follows:

Arc	Edge	Vertex
3	2	1
1	2	2

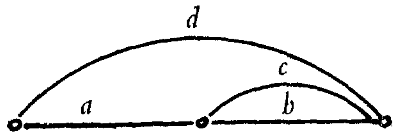
This new task is now immediately removed from the list to be carried out. Once again, there is only one edge to consider and its character is *b*. This does not match the *c* on the arc and the task therefore terminates. Notice that there are now no tasks on the list with vertex number 1 so that there is no longer any possibility of adding new edges at that vertex. The one task remaining will cause a new edge to be added at vertex 2 which could have been successfully used by the task just completed. But it is now too late. To complete the story, task ⟨1, 2, 2⟩ is now started, leaving the task list empty. The arc and edge match and a replacement is made giving the following as the final chart:



This problem can be solved in a variety of ways. Most of them consist in carefully controlling the order in which items are removed from the task list. For reasons that will emerge later, I shall here propose an alternative solution. With each vertex in the chart, there will be associated a *wait list* which will simply be a list of things to be done to any new edge added at that vertex. An entry on a wait list will be a couple ⟨arc, vertex⟩. Whenever a new edge is added at the given vertex, a new task will be created for each item on the wait list, the required triple being formed by adding the new edge to the couple on the wait list. It remains only to describe how new entries are made on a wait list. This is done in a third step which we now add to the description of how tasks are carried out.

3. Add the couple ⟨ *a*, *v* ⟩ to the wait list of the vertex to which *e* belongs *if it is not already there.*

In the example just considered, this will cause, among other things, a couple ⟨3, 1⟩ to be placed on the wait list for vertex 2. When the new edge representing the character *c* is appended to this vertex, a new task will automatically be created, namely ⟨3, 3, 1⟩, where 3 is taken to be the label of the new edge. When this task is carried out, it will lead immediately to the addition of a new edge at vertex 1 and the final chart will have the desired final form, namely:



The principal problems that arise in dictionary consultation concern (1) what to look up, and (2) how to store and gain access to the dictionary. The problem of what to look up is the problem of deciding which of the substrings that the chart contains at the end of the rewriting process should be referred to the dictionary. We shall see that the approach we take to the second of these problems is strongly conditioned by the solution we adopt to the first.

One of the earliest solutions to the problem of what to look up was the following: start at the beginning of the string and find the longest word in the dictionary that can be matched beginning at that point. Then repeat the process starting with the first unmatched character. This so-called *longest-match* procedure is easy to fault. In a word like *underivable*, it will surely recognize *under* as the longest initial component and go on to forage for something in the dictionary to match *ivable*. Furthermore, the scheme would doubtless have to be modified before it could be applied to data presented in the form of a chart because there could be more than one longest matching initial substring if they lay on different paths. Almost any process applied to a chart will necessarily have to be nondeterministic. But it is easy to see that the process of dictionary consultation must, in any case, be nondeterministic because there are many cases of morphological ambiguity. In English, there are a few words like *elipses* that are derivable from more than one stem; in this case *elipse* and *elipsis*. In other languages, such examples are much easier to find. The word *conti* in Italian is the plural of both *conto* and *conte* and there are many other words like it.

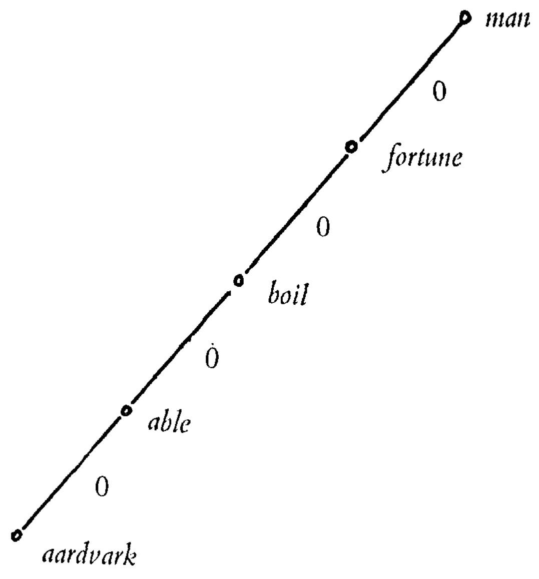
It seems, then, that nothing less than a procedure that exhausts the *coverings* of all the strings in the chart will have the generality we require. By a covering of a string I mean a segmentation of the string into non-overlapping substrings each of which is an entry in the dictionary. The English word *interminable* has only one correct segmentation that would be deemed correct in most texts, but it has at least two coverings by the English lexicon, for it is not only *in* + *terminate* + *able*, but also *inter* + *mine* + *able*. The second analysis might possibly be considered correct in a text on mining containing a sentence like *These strata are not interminable with presently available machinery*.

Logically, there is little to distinguish dictionary consultation from the application of rewriting rules. In one case strings of characters are rewritten by other strings of characters whereas, in the other, they are rewritten by strings of items of a fundamentally different kind, namely words or the lexical descriptions of words. The principal difference is that dictionary consultation is a process that does not apply to its own results; the input always consists of characters and the output of words. But there are practical considerations that are usually adduced to distinguish dictionary consultation as a special process.

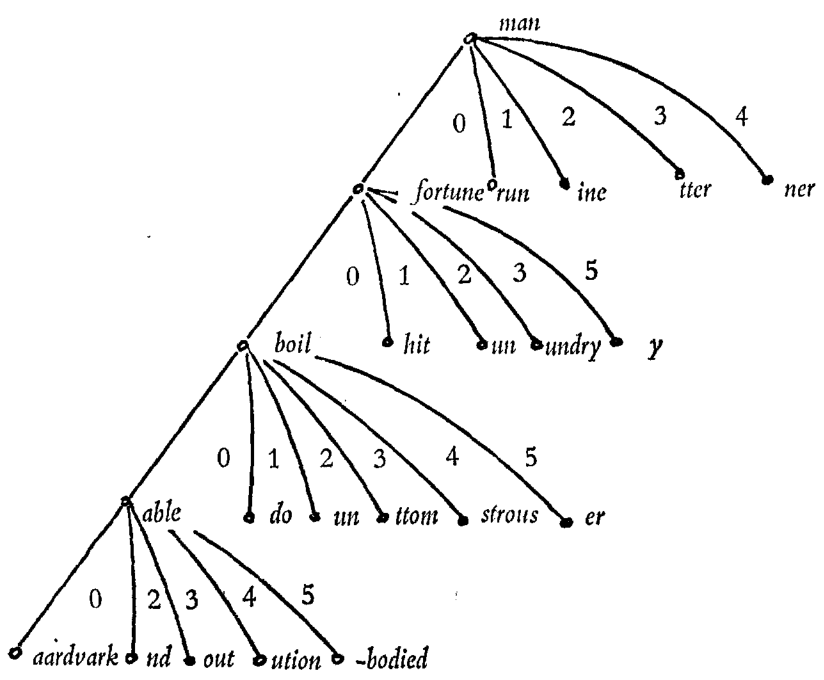
Dictionaries are very large relative to many of the other bodies of data that a linguistic computer program must treat and it has therefore usually been impossible to accommodate them in the rapid-access store of a computer. Reference to external files is notoriously slow and therefore expensive and special techniques are often necessary to make it practical. Now, machines are getting larger and they are frequently designed so that the rapid-access store appears to be much larger than it actually is. But, leaving these facts aside, we nevertheless find that the format I suggested for the rewriting rules can readily be generalized to accommodate a dictionary, and in ways that yield practical and efficient systems.

The heart of the idea is to conflate the initial parts of strings when they are the same so that the whole set of strings is stored in the form of a tree. The only disadvantage that this has as a storage scheme for dictionaries is that, especially near the root of the tree, an arc can have an inordinate number of alternatives so that an appreciable amount of time would have to be expended in searching through them. In English, for example, there are words beginning with each of the 26 letters of the alphabet so that it would take an average of 13 trials to find the initial letter of a word. There is a variety of ways to overcome this problem. The following is a somewhat simplified version of one that has proved efficient, both in time and storage. The method combines the representation in the form of a tree with the well known technique of binary search.

Sort the dictionary into alphabetical order and choose a word from somewhere near the center of the resulting list. This word – the whole word and not just its first letter – will occupy the root of the tree. Suppose the word is *man*. Divide the dictionary in two at this point and choose a second word from near the center of the list of words that preceded *man*. This will occupy a node connected to the root by a line labeled with a zero. Suppose the word is *fortune*. Now consider only the words that precede *fortune* in the list, pick a word from near the middle, and connect it to the node for *fortune* with a line also labeled with a zero. Proceed in this manner until the first word in the dictionary is appended to the end of the chain. What we have now is a diagram of the following kind:



The remainder of the dictionary now consists of five lists: the words following *man*, the words between *fortune* and *man* and so on. Consider first the words that follow *man* in alphabetical order. Divide this into four sublists according to the length of the initial substrings that the words share with *man*. Thus, there will be a sublist of words beginning with the letters *man*, a sublist beginning with *ma* but not having an *n* in the third place, a sublist beginning with *m* but without an *a* in position 2, and a sublist of words that follow *man* in the dictionary but which share no initial substring with it. Now remove the shared initial substrings from the beginnings of the members of these lists – three letters from the members of the first list, two from the second, and so forth. The fourth list remains unchanged. Observe that the alphabetical ordering of the lists is preserved because the same initial substring is removed from the beginning of *all* the words in a given sublist. Now choose a (truncated) word from near the center of each sublist and connect it to the node for *man* by a link labeled with a number one greater than the number of letters in the initial substring just removed from it. For example, if *matter* is chosen from the third list, it is first truncated to *atter* and is then used to label a node connected to the root of the tree by a link labeled with a 3. Repeat this process for each of the five lists, adding new nodes below each of the existing ones. The result will look something like this:



Each word in the tree (except *aardvark*) is connected by a link labeled " 0 " to a subtree of words all of which precede it in alphabetical order. All other links lead from a word to a subtree of words that follow it in the alphabet. A link labeled *n* connects a word to alphabetically later words that differ from it in the *n*th position, but not before.

Each of the words so far accommodated in the tree was chosen from the center of some sublist. These sublists are now installed below the corresponding words in exactly the same way.

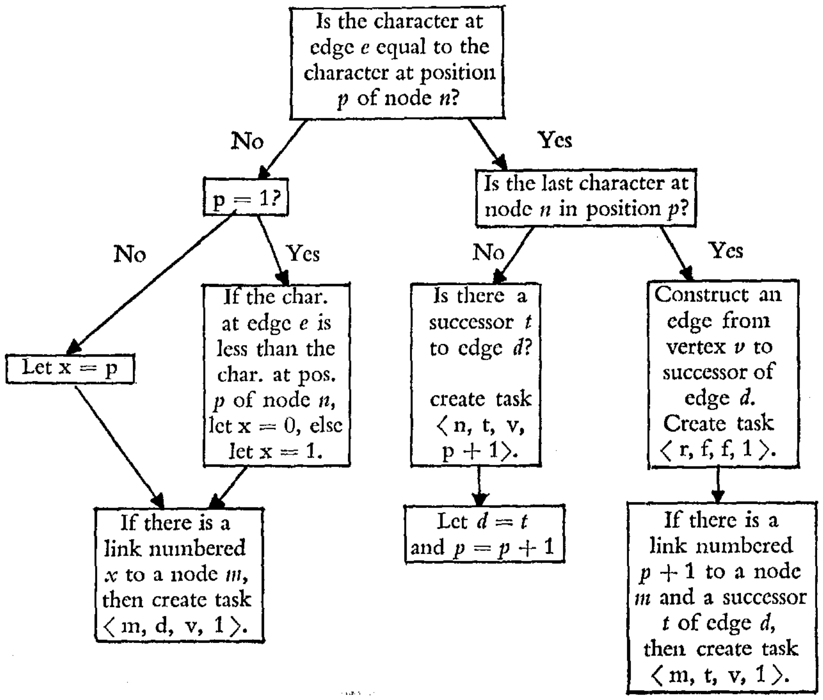
It is an entirely straightforward matter to look words up in a dictionary stored in this way. The algorithm is as follows:

1. Let *w* be the word to be looked up and *v* be, initially, the root of the tree.
2. If *w* occupies node *v*, then announce success; otherwise continue.
3. If *w* precedes the word at note *v* in alphabetical order, then let $n = 0$ and skip to step 6.
4. Let *n* equal the first position, counting from left to right, at which *w* differs from the word at node *v*.
5. Remove the first *n*-1 characters from *w*.
6. If there is a link labeled *n* below *v*, let *v* be the node at the end of

that link and return to step 2; otherwise announce that the word is not in the dictionary.

The method is readily recognizable as a variant of the familiar technique known as *binary search.* The principal difference lies in the treatment of words that share initial characters. This modification makes possible considerable economies in storage space because common initial substrings are only stored in one place. However, the variation also leads to interesting economies in the work that must be done to identify words in character strings stored in the form of a chart.

Suppose the rewriting procedure is changed to allow for two kinds of task, the original rewriting tasks and what I shall call *dictionary tasks.* A dictionary task will be represented on the task list by a quadruple ⟨node, edge, vertex, position⟩ in which the edge and vertex fill the same roles as in the rewriting task but, instead of an arc representing part of a rewriting rule, there is a dictionary node. Position is a number giving the number of characters of the substring that labels the dictionary node which have already been matched. A dictionary task ⟨*n*, *e*, *v*, *p*⟩ is carried out as follows:


Is the character at edge *e* equal to the character at position *p* of node *n*?
No
Yes
p = 1?
No
Yes
Is the last character at node *n* in position *p*?
No
Yes
Let x = p
If the char. at edge *e* is less than the char. at pos. *p* of node *n*, let x = 0, else let x = 1.
Is there a successor *t* to edge *d*? create task ⟨n, t, v, p + 1⟩.
Construct an edge from vertex *v* to successor of edge *d*. Create task ⟨r, f, f, 1⟩.
If there is a link numbered *x* to a node *m*, then create task ⟨m, d, v, 1⟩.
Let *d* = *t* and *p* = *p* + 1
If there is a link numbered *p* + 1 to a node *m* and a successor *t* of edge *d*, then create task ⟨m, t, v, 1⟩.

Whenever a dictionary task is carried out, a character in the dictionary is compared with the characters associated with each of the edges incident from a given vertex. Paths in the chart may diverge, but the process of looking up substrings in the dictionary is one process up to the point of divergence.

The principal advantage of this data structure for a dictionary is that it minimizes the cost of referring different possible analyses of the same word to the dictionary in cases where the differences occur mainly at the end. The algorithm for looking words up also combines with the string-rewriting strategy I have proposed in a happy manner. The organization also allows new words to be added to the dictionary in a straightforward manner without disturbing the existing structure – nothing more than a new link and a single new node at the bottom of the tree is ever required. A simple recursive strategy will restore the dictionary to the form of a simple alphabetical list. The method of constructing a tree of this kind from an alphabetical list that I have described was designed to make the structure clear and not as an algorithm for incorporation in a computer program. However, algorithms do exist, which are beyond the scope of this paper, for performing this operation simply and efficiently. But the principal advantages of doing morphological analysis in this way have still to be stated.

The use of wait lists makes it possible, as I have already remarked, to relax any restrictions there might otherwise have been on the order in which tasks awaiting execution are selected. If processing is allowed to continue, the same results are guaranteed to emerge. What this means is that there can be complete freedom in designing strategies that will increase the probability of reaching a satisfactory solution early in which case the option of abandoning the remaining tasks is open. For example, dictionary tasks might always be given priority over morphographemic rewriting tasks on the principle that a lexical item identified in the text in its standard dictionary form is likely to have been correctly identified, especially if it occurs between non-alphabetic characters. Tasks operating on parts of the chart further to the right might be given priority over those further to the left simply because they are nearer to reaching a conclusion. It would even be possible to augment the grammar in such a way as to allow individual rules to give an explicit priority to the tasks they create. I do not want to urge any one of these policies here, but merely to stress that the basic algorithm leaves the field entirely open.

Perhaps more important is the fact that other tasks, unconnected with morphological analysis can be interleaved freely between those that have been mentioned. The chart was originally designed for use in syntactic analysis[1] and recent work suggests that syntactic analysis can profitably be performed as a set of independent, self-synchronizing tasks as suggested here. If these tasks are intermixed with those required for morphological analysis, the range of possibilities for driving towards a likely solution while foreclosing none of the possibilities allowed by the grammar is greatly extended. What I am proposing is, in the large, obvious: do first what is likely to lead earliest to a successful conclusion and put off other things until later. The particular organization I have suggested is attractive because it seems likely to lead to this goal without any attendant increase in programming complexity. Indeed, these principles are likely to make for greater perspicuity of the resulting program.

[1] M. KAY, *Experiments with a powerful parser*, RM-5452-PR, Santa Monica (California), 1947.