An Introduction to Periphrase, a Linguistic Programming Language

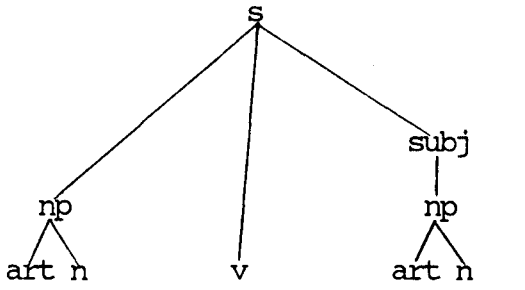
Larry G. Childs

An Introduction to PeriPhrase, a Linguistic Programming Language

Larry G. Childs, A.L.P. Systems

PeriPhrase is a high-level computer language developed in recent years by A.L.P. Systems in Provo for use in various natural language processing applications. PeriPhrase is used for several different projects within A.L.P. Systems, and is also commercially available. This paper is only intended to give the reader a flavor for PeriPhrase. It is not a complete nor even a particularly systematic description of the language. However, on the basis of a few representative examples, I hope to provide a general idea of what it is like to work with this string processing language. These examples will be based largely on my work in machine translation from German to English at A.L.P. Systems.

In my work, I use PeriPhrase to build up a syntactic phrase structure tree for each sentence in a German text. The function of this tree is to serve as the basis of a transfer grammar (also written in PeriPhrase) which turns the sentence into English structure. For example, the phrase structure tree for the following sentence would look something like this.



Note that this phrase structure tree may seem a little unusual in that it does not map the verb and the direct object into a VP like most traditional transformational grammars do. I bring this up to make the point that the PeriPhrase language is designed as a generalized tool. It does not dictate that any particular model or theory of grammar be used. The shape of the tree and even the names of the nodes are left entirely up to the person writing the PeriPhrase programs.

Statements in PeriPhrase are in the form of linguistic rules. These rules consist of a pattern matching section, which searches for elements in a sentence, and a rewrite section, which builds a tree structure on the elements that were matched, thus building up parse trees like the one shown above.

The rules may be as simple as:

```
ART N => NP[...].
```

which says that when an article (ART) and a noun (N) are found in a sentence, a noun phrase node (NP) is created, and all of the pattern elements on the left-hand side of the rule (namely, the ART and the N) become constituents of the new NP node ([...]). In practice, however, the rules tend to be somewhat more complicated than this, and employ a wide variety of features besides simple pattern matching which make them very powerful but also sometimes very complicated, indeed.

One commonly used feature is that of modifying the pattern elements with "attributes". For example, in German it is not sufficient merely to say that an article and a noun form a noun phrase; different types of noun phrases (such as dative plural or nominative singular) are formed from different types of articles and nouns. The first rule below indicates that a nominative singular NP is formed from an ART whose inflectional ending is "er", and a masculine singular N. The second rule forms an accusative singular NP from an ART with an "en" ending and a masculine singular N.

```
ART(ending=er)  *ADJ   N(number=sing, gender=masc)
     => NP[...](number:=sing, case:=nom).

ART(ending=en)   *ADJ   N(number=sing, gender=masc)
    => NP[...](number:=sing, case:=acc).
```

Another feature introduced in these rules is that of the optional pattern element. The '*' ("Kleene star") in front of the adjective (ADJ) means that any number of adjectives (including zero) may occur between the article and the noun in the sentence.

PeriPhrase rules do not have to be context-free, as can be seen in the following example:

```
NP(case#nom)  V  NP(case=nom)  => 1 2 3:=SUBJ.
```

The numbers on the right-hand side of the rule are called pronouns and refer to the pattern elements on the left-hand side. For example, in this rule, the pronouns one through three refer to the first, second and third pattern elements on the left-hand side respectively. The only effect of this rule is to rename the nominative NP node to subject (SUBJ), if that NP is found in the context of NP, verb (V), nominative NP. The "#" used with the attribute of the first NP is a "not" operator, and in this

context means "match on any NP whose case is not nominative."

A variation of this would be to create a SUBJ node above the nominative NP node, as is the case in the above tree, rather than merely renaming the NP node. This would be done by the following rule:

```
NP(case#nom)  V  NP(case=nom)  => 1 2 SUBJ[3].
```

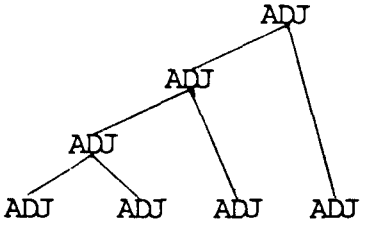
The square brackets indicate the immediate constituents of the nodes on the right-hand side, and can be used to create rather complex tree structures in just one rule. For example, the rule below not only creates a new SUBJ node above the nominative NP, it also maps all of the elements of the pattern into a sentence (S) node. This one rule along with the two previous rules which created NP's is sufficient to create the phrase structure tree that was shown above.

```
NP(case#nom)  V  NP(case=nom)  => S[1, 2, SUBJ[3]].
```

In order to illustrate still more PeriPhrase rule features, let us now look at the following rule whose function is to map up strings of adjectives (ADJ) into a single ADJ node.

```
ADJ  ADJ => ADJ[...].
```

For example, if a sentence contained four adjectives in a row, represented here by "ADJ ADJ ADJ ADJ", the above rule would create the following tree structure:



This shows the recursive nature of PeriPhrase rules. As long as a rule matches a pattern in a sentence, it continues to be applied until the sentence elements have been modified sufficiently that the rule no longer matches any string of elements in the sentence.

In actual practice, this rule is likely to be a bit more complicated because strings of adjectives are generally punctuated by commas and/or conjunctions in German as well as in English, as can be seen in "tired, hungry(,) and poor." The rule to show that either a comma (COM) or a conjunction (CONJ) must occur between the adjectives is shown below:

```
ADJ  {COM | CONJ} ADJ => ADJ[...].
```

To make this rule even more general, we could put a '*' in front of the braces like this '*{COM | CONJ}', which indicates that commas and conjunctions are optional intervening elements.

This particular type of rule can also be used to show another useful PeriPhrase feature, namely that of attribute variables. In German, adjectives can have several different inflectional endings (such as "e" or "en"), and this rule should only apply in German if both adjectives in the rule have the same ending. It doesn't matter which ending it is, as long as it is the same for both adjectives. This restriction can be put in the rule by means of a variable, as seen below:

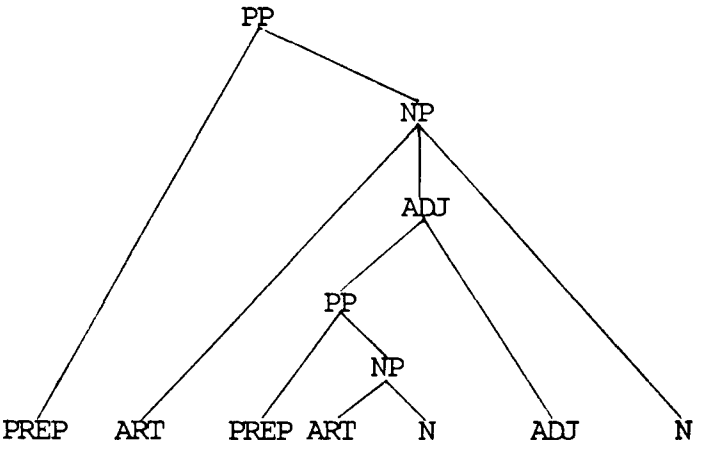
```
ADJ(endvar:=ending)  *{COM | CONJ} ADJ(ending=endvar)
      => ADJ[...].
```

Whatever ending the first ADJ has is loaded into a variable which we have called "endvar", and the attribute restriction on the second adjective is that its ending must be the same as the ending stored in the variable endvar.

Actually, the recursiveness of the individual PeriPhrase rules comes because they are combined into various groups of rules called "packets", which are recursive. As long as at least one rule in a packet fires, i.e. succeeds in matching, the rules in that packet are applied again until none of them fire any more. At that point, the next packet is tried. And on a larger scale, the group of packets which constitutes a PeriPhrase program is also recursive, i.e. the group of packets is tried repeatedly until none of the packets in the group fire any more.

There are many more details involved with the control mechanism of when to apply rules to a sentence, as well as many ways of modifying this control mechanism. For example, the number of rules that the PeriPhrase program writer chooses to put into a packet can greatly affect the order in which the rules of the program are applied, and there are ways to change the order in which packets are tried depending on which rules have fired. However it would go far beyond the scope of this paper to discuss the control mechanism in detail here. Let the following simple example suffice to show some of the power inherent in the recursive nature of PeriPhrase.

In German it is not uncommon to have noun phrases and prepositional phrases nested inside other noun phrases and prepositional phrases, as in the following phrase:



which can be translated as "under the bridge built by the Romans." This whole tree structure can be built with just the following three rules because of the way that rules recur in PeriPhrase.

```
DET  *ADJ  N  => NP.
PREP  NP  => PP.
PP  ADJ   =>   ADJ.
```

In the simplest case, these three rules could be put into one packet, but because the packets are recursive as well, the same tree structure could be created by putting each of these rules inside different packets.

Two other extremely important features of PeriPhrase are "complex rules" and "actions." An action is a computer program or function which is external to PeriPhrase itself, and which can be called from PeriPhrase rules. Actions are written in standard programming languages (I use the C language in my work), and are used to check information, query the user, and even duplicate the functions of PeriPhrase rules themselves.

A good illustration of one use of actions is in conjunction with complex rules. A complex rule is one in which there are two or more rewrite sections. To explain this, let us look at following German sentence which has two valid syntactic parses:

```
Gestern   wurde der  Onkel  von    meinem Freund weggeschickt.
Yesterday was   the  uncle  of/by  my     friend  sent away.
```

In English, the two possible readings can be expressed as:

Yesterdy, the uncle of my friend was sent away.
and
Yesterday, the uncle was sent away by my friend.

In the analysis of the German source sentence, the question of which parse to use hinges on whether the prepositional phrase (PP) is a post-modifier of the NP, or whether it is a separate sentence unit. These two possible parses for the same pattern can be indicated by a complex rule that would look something like this:

```
NP(case=nom)  PP(npmod#no);  check_conjoined(x)
=> choose (x)
{
     NP[...]
|
     1 2(npmod:=no)
}.
```

The "npmod" attribute here, which indicates whether the PP can modify the previous NP, is merely a device to keep the rule from going into an infinite loop if the second rewrite section is chosen. If it were not there, the second rewrite section would not change the sentence elements in any way, and therefore the pattern section of the rule would continue to match infinitely.

In order to decide which rewrite section to use, this rule makes a call to an action routine which we have called "check_conjoined". The purpose of this action is to return either a "1" or a "2" in the variable "x", which the PeriPhrase rule then uses to decide whether to choose the first or second rewrite section respectively.

This "check_conjoined" routine could be written in several ways. One possibility is to have it automatically return a default value, based perhaps on a statistical analysis of which parse is the most common. In the interactive system that I use, a routine of this sort would typically query the user. In such an interactive mode, the action could be written to present the German sentence on the screen with the noun phrase and the prepositional phrase highlighted, and then ask the user whether the prepositional phrase modifies the noun phrase. The user's response would then be passed back to the PeriPhrase rule, the appropriate rewrite section would be chosen, and the PeriPhrase analysis would continue.

Another possibility which is planned but not yet actually implemented is to make complex rules junctures for "backtracking". This is a mechanism by which alternate parse trees are built up so that every possible parse of a sentence can be represented by its own tree. It is planned that after PeriPhrase built a parse tree based on one of the rewrite sections for a complex rule, it could "backtrack" until it found such a juncture, and then build trees based on the other rewrite possibilities for that rule. Building all possible parse trees and then rejecting the ill-formed ones is a standard parsing technique for resolving apparent ambiguities and recognizing true ones.

As I mentioned earlier, I also use PeriPhrase to perform the transfer from German to English. There are three basic transformation operations which can be performed on a PeriPhrase parse tree, namely reordering, deletion, and insertion of nodes.

In the sentence "Den Mann biß der Hund," whose parse tree was shown above, it is necessary to reorder the subject and the direct object when translating into English. After the parse tree has been built up, then a rule such as the following would match and reorder the sentence to the correct English word order:

```
S[NP, V, SUBJ]  =>  1[4, 3, 2].
```

Note that the square brackets on the left-hand side indicate the tree structure to be matched. In other words, this rule will match if it finds an S node which has NP, V, and SUBJ as its immediate constituents. Note also that in reordering a non-terminal node such as NP, PeriPhrase automatically reorders the entire tree structure which is under that node.

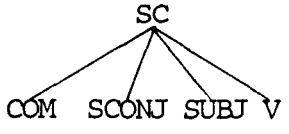
A node can be deleted simply by putting a minus sign in front of its pronoun on the right-hand side of a rule. When translating from German to English, it is often necessary to delete the commas which delimit subordinate clauses as in:

Der Mann weiß, daß der Hund beißt.

which translates as:

The man knows that the dog bites.

If the parse tree structure for the subordinate clause looked like:



then a very simple rule for deleting the comma would look like this:

```
SC[COM, SCONJ, SUBJ, V]  => 1[-2, 3, 4, 5].
```

The final transfer function is that of insertion of terminal nodes into the tree. In translating from German to English, for example, the possessive relationship which is expressed by the genitive case in German is often expressed by the preposition "of" in English, as in:

die Frau des Bürgermeisters
the wife the mayor

which could be translated as:

the wife of the mayor.

The following rule both inserts the word "of" on the terminal level and creates a prepositional phrase structure above it:

```
NP  NP(case=gen)  =>  1 PP["of":=PREP, 2].
```

There are many other aspects of PeriPhrase which could not be covered in a paper of this scope. For example, not all of the operators used in the pattern matching section were mentioned. Only a superficial description of the control mechanism for applying rules was given. And the PeriPhrase development environment with its excellent and comprehensive debugging facilities was not even discussed. However, more complete descriptions are available for those who are interested in actually using the language[1]. This paper will have served its purpose if the reader has gained a feel for what I, as a user, have found to be a very useful and also very exiting language with which to work.

Notes:

[1] Beesley, Kenneth R. and David Hefner. "PeriPhrase: A Linguistic Programming Language," in Proceedings of The Second Annual Artificial Intelligence & Advanced Computer Technology Conference (San Diego, April 29-May 1, 1986), pp. 377-395.

Beesley, Kenneth R., and David Hefner. "PeriPhrase: Lingware for Parsing and Structural Transfer," in Proceedings of Coling '86, 11th International Conference on Computational Linguistics (Bonn, August 25-29, 1986), pp 390-392.