Computational Linguistics SC579

Ch7. Complex Features and Unification

Prof. Feng Zhiwei

7.1 Single feature and complex feature

The linguistic constituents in the CFG were described by single features.

For example, we can use following single features to describe a CFG grammar.

(1) S→NPa VPa
(2) S→NPb VPb
(3) S→NPc VPc
(4) S→NPd VPd
(5) S→NPe VPe

In these rules, there are correspondence between the NP and VP, this correspondence can deal with the agreement between the gender, number and person between NP and VP.

Lexicon:

je: <cat> = NPa
tu: <cat> = NPb
elle: <cat> = NPa
nous: <cat> = NPc
vous: <cat> = NPd
ils: <cat> = NPe
tombe : <cat> = VPa
tombes: <cat> = VPb
tombons: <cat> = VPc
tombez: <cat> = VPd
tombent: <cat> = VPe

This grammar generates exactly six French sentences, of which ‘elle tombe’ (‘she falls’) is typical.

<cat> of ‘elle’ is NPa. So the <cat> of verb ‘tomber’ (‘to fall’) must be VPa, it equals to ‘tombe’. So we can get:

elle tombe.

By this means, we can generate following sentences:

Je tombe (I fall)
Tu tombes (you fall)
Nouis tombons (we fall)
Vous tombez (you fall)
Ils tombent (they fall)

All six sentences intuitively have exactly the same structure, yet our grammar employs five distinct rules nearly one rule for each. This is wasteful and obscures an evident generalization about simple French sentences.

Furthermore, if we try and extend this grammar to encompass the imperfect

Elle tombait (she was falling)

Or three words sentences such as

Elle est tombee (She has fallen)

Then we need still more S expansion rules, even if we decide to treat the auxiliary ‘etre’ form as an element of the verb phrase in the latter case.

In this case, our CFG grammar will become very complex.

In order to avoid the complicity of rules, we propose the following solution – complex feature apparatus:

Rule:

S→NP VP

<NPper> = <VPper>

<NPnum> = <VPnum)

Here ‘per’ represents the person, ‘num’ represents the number. <NPper> = <VPper> means the agreement of person and number between NP and VP. The feature of NP and VP in this rule becomes complex, but the form of the rule is simple. We need only one rule to describe the phenomena described by five rules.

However, the lexicon must become complex.

Lexicon:

je:	<cat> = NP
	<per> = 1
	<num> = sing
tu:	<cat> = NP
	<per> = 2
	<num> = sing
elle:	<cat> = NP
	<per> = 3
	<num> = sing
nous	<cat> = NP
	<per> = 1
	<num> = plur
vous:	<cat> = NP
	<per> = 2
	<num> = plur
ils:	<cat> = NP
	<per> = 3
	<num> = plur
tombe:	<cat> = VP
	<per> = 1
	<num> = sing
tombe:	<cat> = VP
	<per> = 3
	<num> = sing
tombes;	<cat> = VP
	<per> = 2
	<num> = sing

tombons: <cat> = VP
<per> = 1
<num> = plur
tombez: <cat> = VP
<per> = 2
<num> = plur
tombent: <cat> = VP
<per> = 3
<num> = plur

This grammar generates exactly the same six French sentences as our original grammar, but it does so with only a single rule.

The apparent cost:

- Each entry in lexicon seems more verbose (containing too many features)
- 'tombe' has two entries while previously one sufficed.

The application of complex feature can simplifies the description of rule.

7.2 Feature structures as graphs

- Feature matrix: We can use the feature matrix to represent the complex features.

For example, the French pronoun 'nous' can be represented as follows:

cat	NP
per	1
num	plur

A feature contains two parts: attribute and its value. It is a "attribute-value" pair. The single feature consists one "attribute-value" pair, complex feature consists several "attribute-value" pairs, it is a "feature matrix".

- Directed acyclic graph: We can also use directed acyclic graph (DAG) to represent the complex features.

The graph is directed because the arcs have directions, indicated by arrows, and it is acyclic because there are no cycles in it – it is not possible to get from a node to itself just by following the arrow. In our example, values gave all been atomic, which is to say that they have had no internal structure.

"nous" can be represented by directed acyclic graph as follows:

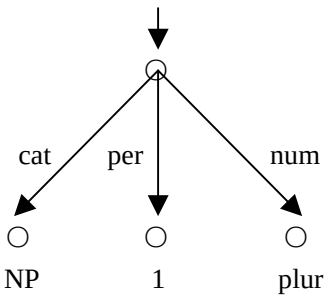

Fig. 1 directed acyclic graph

- Category-valued feature: Same time, we can also use “category-valued feature” in which features are allowed to take categories as their values (not only atomic value). Category-valued feature allow many significant grammatical generalizations to be captured rather straightforwardly. For example, in our French grammar, we can use category-valued feature “arg0” to handle the agreement. Our S expansion rule can simple require the subject (NP) to be identical to the values of the arg0 feature in the VP. We can thus put into the arg0 feature for a verb all the requirements it places on its subject (NP) – category, person, number, and so on.

This rule is shown, along with the revised lexical entries necessary for the verb forms. The lexical entries for the pronouns remain as in the previous example.

Rule:

S→X VP

<VP arg0> = X

Lexicon:

Pronoun:

je:	<cat> = NP
	<per> = 1
	<num> = sing
tu:	<cat> = NP
	<per> = 2
	<num> = sing
ell:	<cat> = NP
	<per> = 3
	<num> = sing
nous	<cat> = NP
	<per> = 1
	<num> = plur
vous:	<cat> = NP
	<per> = 2
	<num> = plur
ils:	<cat> = NP
	<per> = 3
	<num> = plur

verb:

tombe:	<cat> = VP
	<arg0 cat> = NP
	<arg0 per> = 1
	<arg0 num> = sing
tombe:	<cat> = VP
	<arg0 cat> = NP
	<arg0 per> = 3
	<arg0 num> = sing
tombes;	<cat> = VP
	<argo cat> = NP

<arg0 per> = 2
<arg0 num> = sing

tombons: <cat> = VP
<arg0 cat> = NP
<arg0 per> = 1
<arg0 num> = plur

tombez: <cat> = VP
<arg0 cat> = NP
<arg0 per> = 2
<arg0 num> = plur

tombent: <cat> = VP
<arg0 cat> = NP
<arg0 per> = 3
<arg0 num> = plur

We can use our graph notation to good effect in displaying the structure of the verbal categories assumed in our revised of the French grammar fragment.

For example, “tombons” can be represented by following DAG:

.

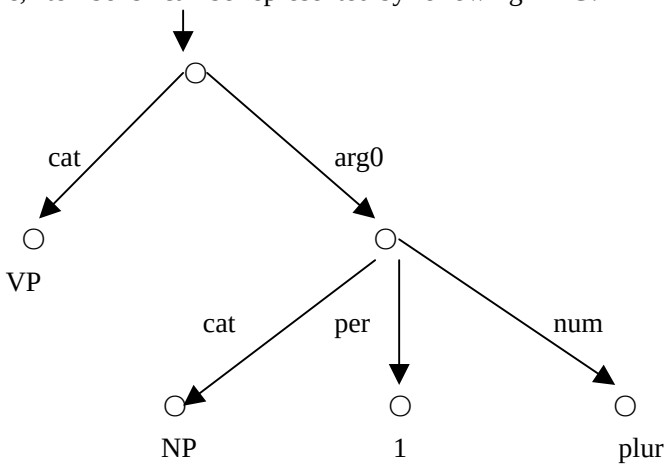

Fig 2. DAG for ‘tombons”

Fig2. shows the category to which the word “tombons” belongs. We will refer to a feature as being category-valued if and only if it is not atom valued. Thus, arg0 is category valued, whereas cat, per and num are all atom valued.

Notice that our use of notation such as “<arg0 num> = plur” can be seen to specify a path in the DAG(<arg0 num>) and tells us the label on the node at the end of the path (plur). All the feature on the end of the path is “atom-valued feature”. If a feature is not the atom-valued feature, it must be the category-valued feature.

- Inheritance of feature

In the DAG, there is the inheritance relationship between different categories. The VP can inherits the tense feature of verb. If VP is mother category, the verb is daughter category, due to the mother category inherits the features of daughter category, so the daughter category is known as the “head” of the mother category. The verb is the head of VP because VP inherits the tense feature of verb.

According to the concept of “head”, we can write a typical VP rule as follows:

Rule

VP→V NP PP

<V head> = <VP head>

This requires that the value of the "head" feature on the V and that on the mother VP be identical. If the head of the V contains an attribute value pair that is inconsistent with a pair in the head of the VP, then the rule is inapplicable. The values of head here will not be atomic but will themselves be DAGs.

So far, the feature graphs that we have presented have always been trees. However, if we have category-valued features it is convenient to take advantage of the flexibility offered by general DAGs.

Consider the following extended VP rule:

Rule

VP→V NP PP

<V head> = <VP head>

<VP verb> = <V>

We can represent this rule by following two DAGs:

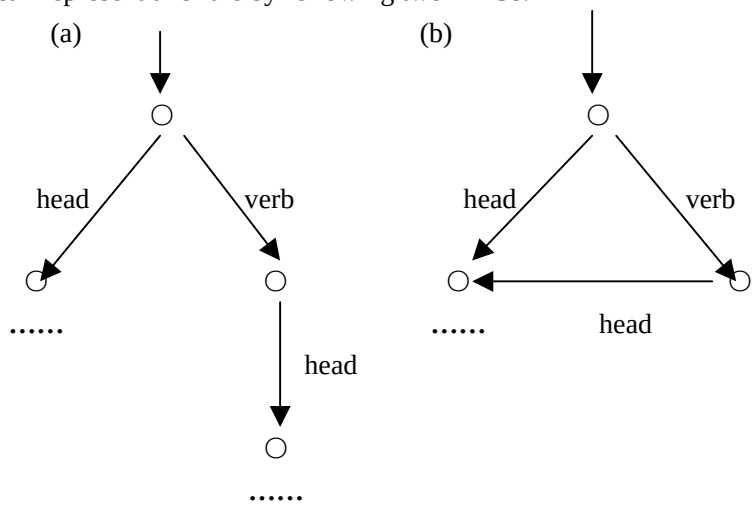

Fig. 3 alternative representations for VP category

In Fig. 3, the repeated values of the head features have been omitted. The first involves having two copies of the substructure that appears twice, whereas the second involves a sharing of the common substructure. Which of these we choose depends on what we want "=" to mean and what we expect to follow from what.

From:

<VP head> = <VP verb head>

and

<VP verb head num> = sing

we would like it to follow that

<VP head num> = sing

which is to say that the VP inherits all the head feature of its verb.

In a sharing interpretation, "=" means that two structures are the same structure, rather than that they just happen to have the same values for all features. We will interpret "=" as specifying that two categories or parts of categories must share, rather than simply have the same value.

The DAG in Fig. 3 (a) means: there are two head substructures, we can add some information to

one side of head in the DAG, and don't touch the head in the other side of DAG. However, the DAG in Fig. 3(b) means "sharing". We can not add some information to the heads in one side and don't touch the head on the other side. It is the interpretation of "sharing".

7.3 Subsumption, unification and generalization

Computational implementations of principles of feature matching crucially depend upon notion of subsumption, unification and generalization.

We can define subsumption as follows:

7.3.1 Subsumption:

A category A subsumes a category B if and only if:

- every atom-valued feature specification in A is in B
- for every two feature values that share anywhere in A, the corresponding values share in B, and
- for every category-valued feature specification in A, there is a value for that feature in B, and the value of the feature in A subsumes the value in B.

If A subsumes B, then we may refer to B as an extension of A or say that B extends A".

Category A properly subsumes category B if A contains less information than B; that is, it is a less informative description. Thus, every piece of information in A must be present in B, but not vice versa.

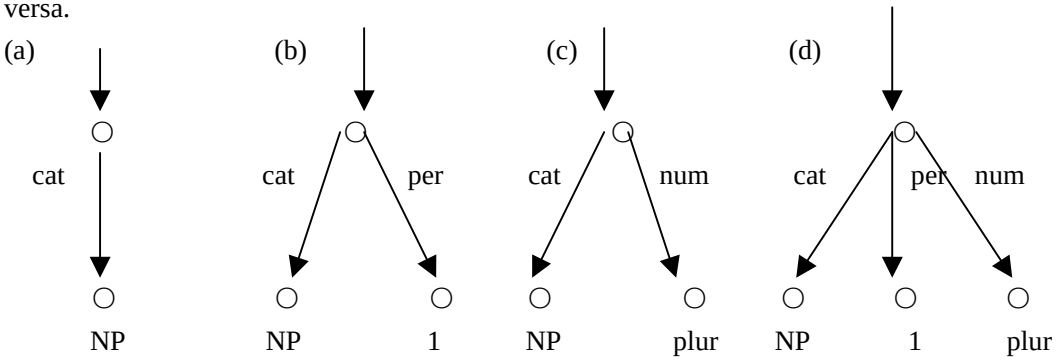

Fig 1. Example categories

In Fig 1., category (a) subsumes both category (b) and category (c), but neither (b) nor (c) subsumes the other. However, category (b) and (c) each subsume category (d).

We can represent the ordering categories on subsumption in Fig.2. In Fig.2, the set of categories can be displayed on a **lattice diagram** with less informative categories shown above their extension or the more informative categories.

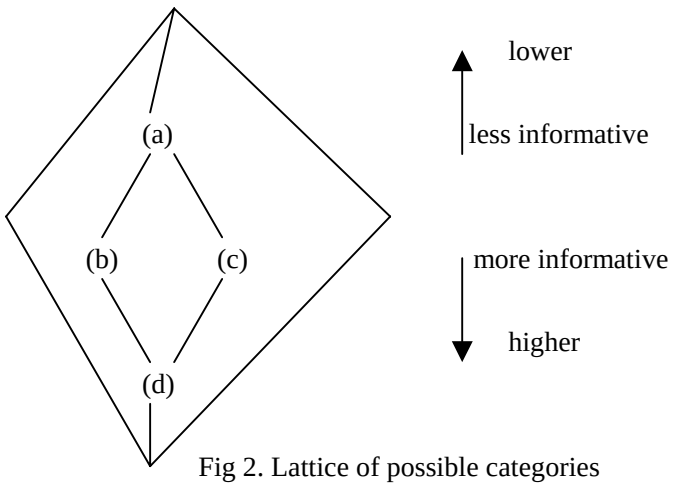

Fig 2. Lattice of possible categories

An important operation on categories is that of unification. This notion is closely analogous to the operation of union on sets. Unification is undefined for categories containing feature specifications that contradict each other.

The definition of Unification is as follows:

7.3.2 Unification:

The unification of two categories is the smallest category that extends both of them, if such a category exists. Otherwise the unification is undefined.

In terms of the lattice diagram, the unification of two categories is the highest category that is below both of them. It contains the pooled information from the two categories, but no more. In our example, category (d) can be seen to be the unification of the set containing categories (b) and (c). category (d) also is the unification of the set containing categories (a), (b) and (c).

However, if we look for the unification of (b) with the category in Fig. 3, we find it to be undefined.

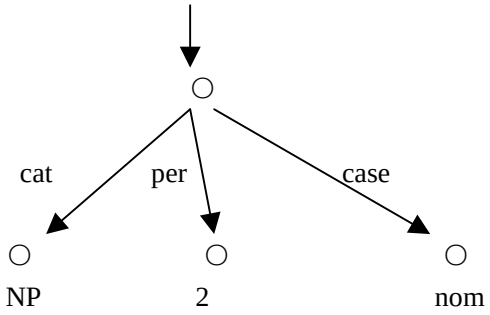

Fig. 3 further example category

7.3.3 Generalization

The generalization of two categories is the largest category that subsumes both of them.

In the lattice diagraph, the generalization of two categories is the lowest category that comes above both. In our example, category (a) is the generalization of the set containing categories (b) and (c). category (a) is likewise the generalization of the set containing categories (a), (b) and (c), and of the set containing (d) and the category of Fig. 3.

The generalization in this domain is similar as the intersection in set theory.

7.3.4 Feature path and reentrant structure

Most important notion is unification. Now we shall discuss the operation of unification.:

We use the DAG to represent the attribute-value pair.

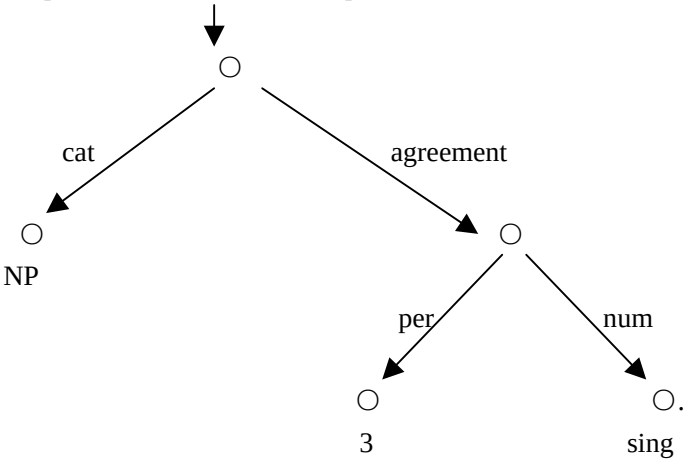

Fig. 4 DAG for feature structure

This correspondent matrix of the DAG is as follows:

$$\begin{bmatrix} \text{cat} & \text{NP} \\ \text{agreement} & \begin{bmatrix} \text{num} & \text{sing} \\ \text{Per} & 3 \end{bmatrix} \end{bmatrix}$$

In DAG, a **feature path** is a list of features through a feature structure leading to a particular value. For example, in Fig. 3, we can say that the <agreement num> path leads to the value sing, <agreement per> path leads to the value 3.

If there is the shared feature structure, such feature structure will be referred to as **reentrant structure**. In the case of a reentrant structure, two feature paths actually lead to the same node in the structure.

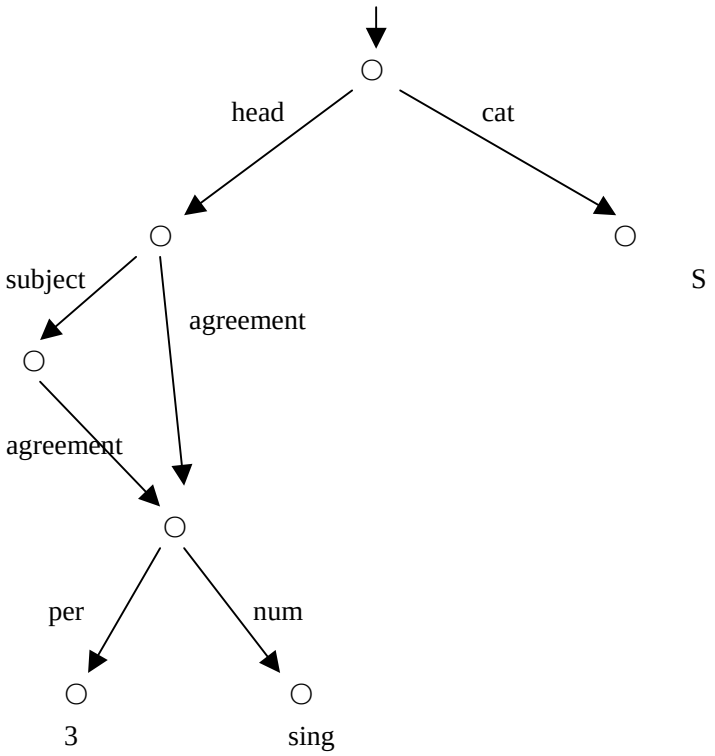

Fig. 4 a feature structure with shared values

In Fig. 4, the <head subject agreement> path and the <head agreement> path lead to the same location. They shared the feature structure

$$\begin{bmatrix} \text{per} & 3 \\ \text{num} & \text{sing} \end{bmatrix}$$

The shared structure will be denoted in the matrix diagram by adding numerical indexes that signal the values to be shared.

$$\begin{bmatrix} \text{cat} & \text{s} \\ \text{head} & \begin{bmatrix} \text{agreement} & ① & \begin{bmatrix} \text{num} & \text{sing} \\ \text{per} & 3 \end{bmatrix} \\ \text{subject} & & \begin{bmatrix} \text{agreement} & ① \end{bmatrix} \end{bmatrix} \end{bmatrix}$$

The reentrant structures give us the ability to express linguistic knowledge in the elegant ways..

7.4 Unification of feature structures

There are two principle operations:

- Merging the information content of two structure that are compatible;
- Rejecting the merging of structures that are incompatible.

Following are the examples:

(1)

$$\begin{bmatrix} \text{num} & \text{sing} \end{bmatrix} \cup \begin{bmatrix} \text{num} & \text{sing} \end{bmatrix} = \begin{bmatrix} \text{num} & \text{sing} \end{bmatrix}$$

(2)

$$\begin{bmatrix} \text{num} & \text{sing} \end{bmatrix} \cup \begin{bmatrix} \text{num} & \text{plur} \end{bmatrix} = \text{fials!}$$

(3)

$$\begin{bmatrix} \text{num} & \text{sing} \end{bmatrix} \cup \begin{bmatrix} \text{num} & [] \end{bmatrix} = \begin{bmatrix} \text{num} & \text{sing} \end{bmatrix}$$

(4)

$$\begin{bmatrix} \text{num} & \text{sing} \end{bmatrix} \cup \begin{bmatrix} \text{per} & 3 \end{bmatrix} = \begin{bmatrix} \text{num} & \text{sing} \\ \text{per} & 3 \end{bmatrix}$$

(5) The reentrant structure

$$\begin{bmatrix} \text{agreement} & ① \begin{bmatrix} \text{num} & \text{sing} \\ \text{per} & 3 \end{bmatrix} \\ \text{subject} & \begin{bmatrix} \text{agreement} & ① \end{bmatrix} \end{bmatrix}$$

$$\cup \begin{bmatrix} \text{subject} & \begin{bmatrix} \text{agreement} & \begin{bmatrix} \text{per} & 3 \\ \text{num} & \text{sing} \end{bmatrix} \end{bmatrix} \end{bmatrix}$$

$$= \begin{bmatrix} \text{agreement} & ① \begin{bmatrix} \text{num} & \text{sing} \\ \text{per} & 3 \end{bmatrix} \\ \text{subject} & \begin{bmatrix} \text{agreement} & ① \end{bmatrix} \end{bmatrix}$$

(5) The copying capability of unification

$$\begin{bmatrix} \text{agreement} & ① \\ \text{subject} & \begin{bmatrix} \text{agreement} & ① \end{bmatrix} \end{bmatrix}$$

$$\cup \begin{bmatrix} \text{subject} & \begin{bmatrix} \text{agreement} & \begin{bmatrix} \text{per} & 3 \\ \text{num} & \text{sing} \end{bmatrix} \end{bmatrix} \end{bmatrix}$$

$$= \begin{bmatrix} \text{agreement} & ① \\ \text{subject} & \begin{bmatrix} \text{agreement} & ① \begin{bmatrix} \text{per} & 3 \\ \text{num} & \text{sing} \end{bmatrix} \end{bmatrix} \end{bmatrix}$$

(6) The difference between features that actually share values and the features merely have similar values.

$$\begin{bmatrix} \text{agreement} & [\ \text{num} \quad \text{sing}\ ] \\ \text{subject} & [\text{agreement} \quad [\text{num} \quad \text{sing}]\ ] \end{bmatrix}$$

$$\cup \begin{bmatrix} \text{subject} & [\text{agreement} \quad \begin{bmatrix} \text{per} & 3 \\ \text{num} & \text{sing} \end{bmatrix}] \end{bmatrix}$$

$$= \begin{bmatrix} [\text{agreement} \quad [\text{num} \quad \text{sing}]\ ] \\ \text{subject} \quad [\text{agreement} \quad \begin{bmatrix} \text{num} & \text{sing} \\ \text{per} & 3 \end{bmatrix}] \end{bmatrix}$$

(7) The failure of unification

$$\begin{bmatrix} \text{agreement} & ① \begin{bmatrix} \text{num} & \text{sing} \\ \text{per} & 3 \end{bmatrix} \\ \text{subject} & [\ \text{agreement} \quad ①\ ] \end{bmatrix}$$

$$\cup \begin{bmatrix} \text{agreement} & \begin{bmatrix} \text{num} & \text{sing} \\ \text{per} & 3 \end{bmatrix} \\ \text{subject} & [\ \text{agreement} \quad \begin{bmatrix} \text{num} & \text{plur} \\ \text{per} & 3 \end{bmatrix}\ ] \end{bmatrix}$$

Fails !

Feature structures are a way of representing partial information about some linguistic object or placing informational constrains on what the object can be. Unification can be seen as a way of merging the information in each feature structure, or describing objects which satisfy both sets of constraints.

Intuitively, unifying two feature structures produces a new feature structure which is more specific (has more information) that or is identical to, either of the input feature structure. So unification is very useful in NLP.

7.5 Complex feature and chart parsing

7.5.1 The description of rule by DAG

The rule of CFG can be described by the means of DAG.

For example, following rule

Rule

S→NP VP

<NP head> = <VP head>

<S subj> = NP

which, expanded into its full form, is:

Rule

X0 → X1 X2

<X0 cat> = S

<X1 cat> = NP

<X2 cat> = VP

<X1 head> = <X2 head>

<X0 subj> = <X1>

This rule says that:

Every category X0 can be realized as a category X1 followed by a category X2 as long as the cat of X0 is S, the cat of X1 is NP, the cat of X2 is VP, the head of X1 is the same as the head of X2 and the subj of X0 is X1.

We can represent this rule as follow by DAG:

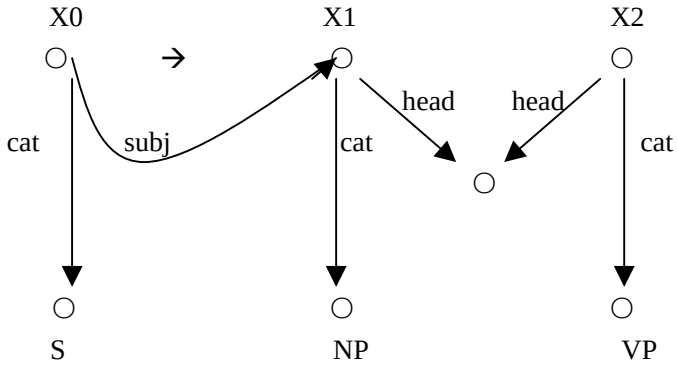

Fig. 1 DAG satisfying a rule

Given three categories X0, X1 and X2, we could proceed, through the conditions of the rule and verify whether they are satisfied by the categories.

7.5.2 Chart parsing with feature-based grammars:

What exactly are the issues to be faced when implementing a parser for a grammar that allows complex categories? The basic parsing problem is the same. Whether categories are atoms or arbitrary feature structures. All that is required is that the basic operations of parser (testing for the equality, looking for rules, and so on) be redefined to accept a different data structure for categories.

Let us start by surveying the main differences between parsing with monadic categories and with grammar involving complex categories.

- The raw material of the grammar, the rules and lexical entries, are different.
- We must start using DAGs in appropriate places in the chart.

An edge in a chart recognizer is a structure with following components:

<START> = .. some integer …

<FINISH> = .. some integer ..

<LABEL> = .. some category …

<FOUND> = .. some category sequence …

<TOFIND> = .. some category sequence …

If we would like to use complex categories in the chart parser, the dotted rule in any edge of the

chart will now have as well as DAGs for its FOUND and TOFIND categories.

Let us to see Fig. 2,

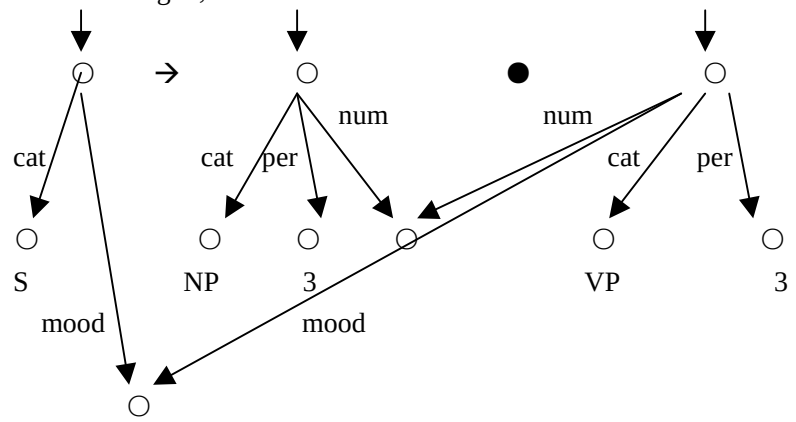

Fig. 2 dotted rule for active edge

The rule in Fig 2. indicates that we have found an NP with ‘per 3’ and that, if we can find a VP with ‘per 3’, and the ‘num’ of NP and VP is same, then we will have found an S with ‘mood’ the same with the ‘mood’ of the VP.

Let we see Fig. 3.

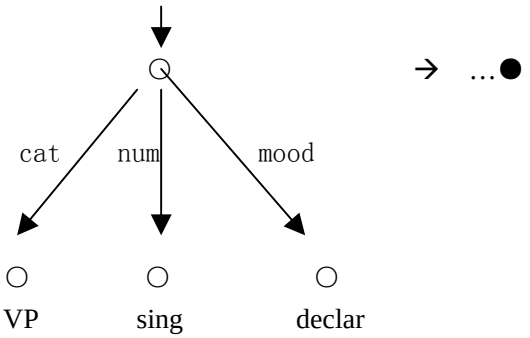

Fig. 3 Dotted rule for inactive edge

The rule in Fig. 3, indicates that we have found a VP with particular values ‘num sing’ and ‘mood declar’. It is an inactive, and the feature on the edag is compatible with the feature in the active edged in Fig. 2, so we can use the fundamental rule of chart parsing to combine them.

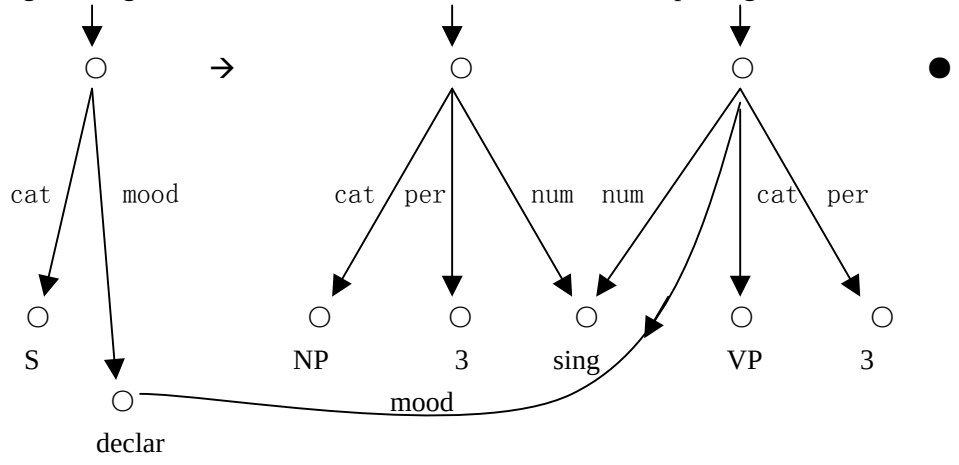

Fig. 4 result of applying the fundamental rule

Fig. 4 is the result of applying the fundamental rule. We can see that the mood of S is the

same as the mood of its VP, the value of mood is ‘declar’, the num of NP is the same as the num of VP, the value of ‘num’ is ‘sing’. What we have done here is here is to extend the whole sequence of DAGs just enough so that the DAG for the required VP and the DAG for the found VP are unified. In general, the result of an application of the fundamental rule is a new edge, and then the dot has been moved on one place to the right. In this process, the unification is the appropriate operation.

The backbone of this new rule is same with following monadic rule of chart parser:

<i, k, S→NP VP**.**>

The backbone of label in the active edge is:

<i, j, S→NP**.** VP>

The backbone of label in inactive edge is:

<j, k, VP→….>

We can use the fundamental rule with monadic features to represent the fundamental rule with complex features. The fundamental rule with complex features is more informative.

7.6 Representation of lexical knowledge

In a NLP system, the lexical knowledge is very important. It is the core of NLP system. Modern grammatical frameworks are increasingly emphasizing the centrality of the lexicon. So we will attempt to look a bit more carefully at what information needs to be represented and how to be represented.

We will begin by trying to consider just the syntactic information that the lexicon needs and how it may best be encoded

There are really only three types of purely syntactic information associated with words:

- The basic part of speech: For example, the word is a verb;
- What the word combines with: that is, its complements, and possibly its subject;
- Certain syntactically relevant inherent properties: for example, gender in the case of nouns.

All three types of information are generally encoded within the syntactic category of a word in a feature-based syntax.

Followings are the examples.

(1) Lexeme Maedchen (girl):

<cat> = N

<gender> = neut

This lexical entry tells us that the German word “Maedchen” is a neuter noun.

(2) Lexeme love

<cat> = V

<arg0 cat> = NP

<arg) case> = nom

<arg1 cat> = NP

<arg1 case> = acc

This lexical entry says that the English word ‘love’ is a verb that subcategorizes for an accusative object NP and combines with a nominative subject NP. It is clear that we are using the arg0 feature to code up information about subject of the verb and arg1 to code up information about the direct object.

(3) Lexeme give:

<cat> = V
<arg0 cat> = NP
<arg0 case> = nom
<arg1 cat> = NP
<arg1 case> = acc
<arg2 cat> = PP
<arg2 pform> = to

Here, '<arg2 pform> = to' means that the preposition of arg2 is 'to'.

The verb 'bet' (to risk money on the result of a future event) would force us to have an arg3 feature as well. In the sentence

He bets me ten dollars on the John's coming

Here, 'he' is arg0, 'ten dollars' is arg1, 'me' is arg2, "on John's coming" is arg3, it is the content of bet.

In the sentence

He bets me ten dollars

There is not the arg3.

The syntactical feature can be represented as following:

Rule1:

VP → V X1 X2 X3
<V arg1> = X1
<V arg2> = X2
<X arg3> = X3

This rule can represent first sentence.

Rule2

VP → V X1 X2
<V arg1> = X1
<V arg2> = X2
<V arg3> = 0

This rule can represent second sentence.

Generally speaking, four argument features (arg0, arg1, arg2, arg3) would be sufficient for an English grammar.

English contains thousands of simple transitive verbs like 'love'. If each one is given a separate fully specified entry like the one given here, then our English lexicon is going to be very large.

One simple way to avoid this is through the use of abbreviations expressed as "**macros**".

Macro syn_iV (intransitive verb – 'die')
<cat> = V
<arg0 cat> = NP
<arg0 case> = nom

Macro syn_tV (transitive verb – 'eat')
syn_iV
<arg1 cat> = NP
<arg1 case> = acc

Macro syn_dtV (ditransitive verb – 'give')
syn_tV

<arg2 cat> = PP

<arg2 pform> = to

Macro syn_datV (dative verb – ‘hand’)

syn_tV

<arg2 cat> = NP

<arg2 case> = acc

The idea of a macro is to have a single symbol, for instance syn_iV, which abbreviates a whole set of feature specifications – for instance <cat> = V, <arg0 cat> = NP, <arg0 case> = nom. In a lexical entry, whenever we include the name of a macro, it is as if we included the whole set of specifications that name abbreviates.

We can even have macro definitions that invoke other macros, as syn_tV does with syn_iV.

The advantage of macros is that we can have much shorter lexical entries that took like this:

-- Lexeme die

syn_iV

e.g. He died.

-- Lexeme elapse (to pass away)

syn_iV

e.g. 3 months have elapsed.

-- Lexeme eat

syn-iV

e.g. John eats.

-- Lexeme eat

syn_tV

e.g. We eat the fish.

-- Lexeme give

syn_tV

e.g You give me!

-- Lexeme give

syn_dtV

e.g You give the books to me..

-- Lexeme give

syn_datV

e.g. You give me the books.

-- Lexeme hand

syn_dtV

e.g. You hand that book to me.

-- Lexeme hand

syn_datV

e.g. You hand me that book.

-- Lexeme love

syn_tV

e.g John loves Mary.

Notice that many words will have several lexical entries in virtue of the various syntactic classes to which they belong.

Macro is a useful abbreviatory device for the writer of the lexicon.
Macro can be expanded as DAG.
For example,
Lexeme give:
syn_tV
can be expanded as following DAG

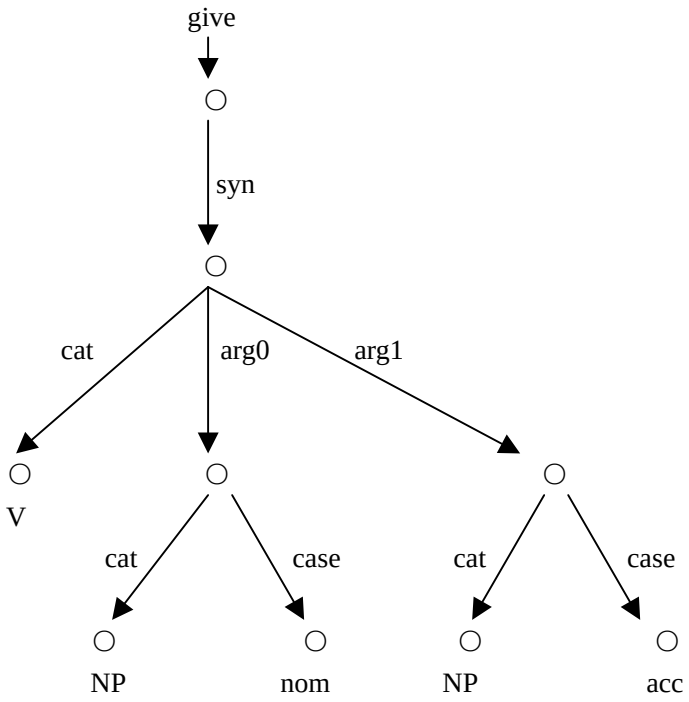

Fig. 5 expansion of macro

If the macro is expanded to lexical entries, then these lexical entries can be unified with other expanded macros, the parser can use the complex features to calculate.

7.7 Semantic features and macro
Multiple entries for what is morphologically the same verb are not required merely to indicate syntactic variants, as they also correspond to semantic differences.
This brings us to the semantic information needed in lexical entries.
The sense of ‘eat’ in ‘we eat’ is a function with one argument – that is, something that needs to combine with one object to produce a proposition.
The sense of ‘eat’ in ‘we eat fish’ is a function with two arguments – that is, something that needs to combine with two objects.
We will distinguish such senses as ‘eat1a’ and ‘eat2a’.
‘eat1a’ for the one-argument case, ‘eat2a’ for the two-argument case, and so on.
The final alphabetic character in these sense names (e.g. ‘2a’ in ‘eat2a’) is to provide for the case of words with semantic variants with the same number of arguments (e.g. ‘eat2a’ needs two arguments).
By this means, we can use ‘give3a’ for the semantic feature of ‘give’ in “we give fish to John”; and we use ‘give3b’ for the semantic feature of ‘give’ in “we give John fish”
We will assume that ‘eat1a’, ‘eat2a’, ‘give3a’, ‘give3b’, and so on, are simpley logical constants to

be found in our language of semantic representation, and we will completely ignore the logical relationship between these constants. Given such a simplistic view of semantics, all the lexicon has to do is to tell us what constants correspond to what words.

From now on, we assume that lexical entry DAGs split initially into 'syn' (syntax), 'sem' (semantics) and 'mor' (morphology) branches, rather that only consisting of the 'syn' branch. For example, the DAG of 'give2a' will become to as follows:

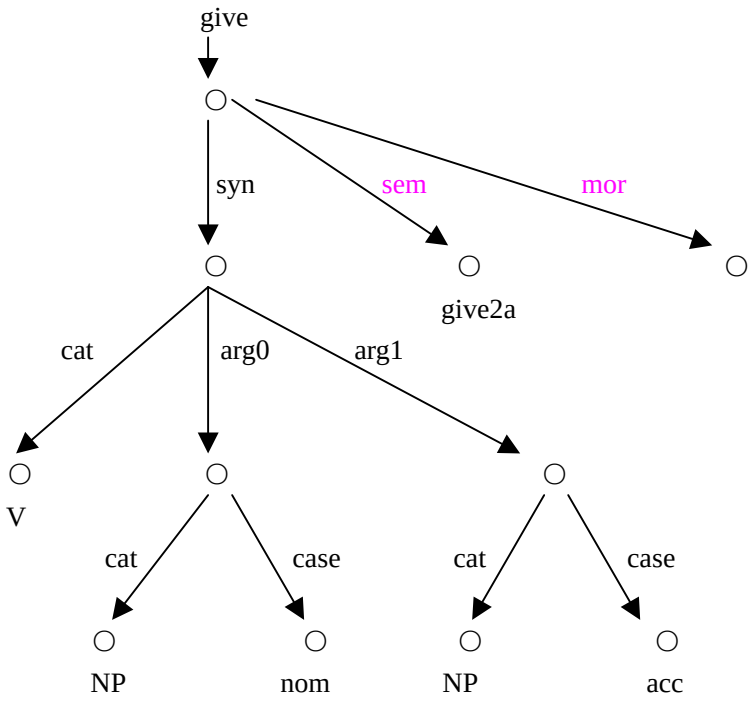

Fig 1. DAG for 'give2a'

This allow us to recast our lexical entries like this:

--Lexeme die:
    syn_iV
    <sem> = die1a

--Lexeme elapse:
    syn_iV
    <sem> = elapse1a

--Lexeme eat
    syn_iV
    <sem> = eat1a

--Lexeme eat:
    syn_tV
    <sem> = eat2a

--Lexeme give
    syn_tV
    <sem> = give2a

--Lexeme give
syn_dtV
<sem> = give3a

--Lexeme give
syn_datV
<sem> = give3b

--Lexeme hand
syn_dtV
<sem> = hand3a

--Lexeme hand
syn_datV
<sem> = hand3b

--Lexeme love
syn_tV
<sem> = love2a

These entries still lack morphological information.

7.8 Morphological features and macro

An English verb can appear in an absolute maximum of eight distinct forms, of which one is always particular (the root) and one is always predicable from the root (the present participle form in –ing) once you know the root.

For example, here are the various forms for the most irregular verb “be’ in English:

root – be

form1 – am
form2 – are
form3 – is
form4 – was
form5 – were
form6 – been
form7 – being

We use the feature “root” to encode the root, and “form1’ to “form7” as features for encoding the other seven potential verb forms. Here, “form1”. “form2”, and “form3” are the first, second and third-person present forms, respectively; “form4” is the first person singular past tense form; “form5’ is the second person singular past tense form; “form6’ is the past participle form; and “form7’ is the present participle form.

However, the regular English verb is only manifested in four distinct forms, all of them predicable from the root.

root – stamp

form1 – stamp
form2 – stamp
form3 – stamp**s**
form4 – stamped
form5 – stamped
form6 – stamped
form7 – stamping

We will adopt an analysis in which each of the forms has a stem and an ending, representing the basis word from which they are formed (always the root for regular verbs), together with the ending that is added for this form.

We can provide an abbreviation regular verbs as follows:

Macro mor_reg_V
<mor form1 stem> = <mor root>
<mor form1 ending> = ε
<mor form2 stem> = <mor root>
<mor form2 ending> = ε
<mor form3 stem> = <mor root>
<mor form3 ending> = s
<mor form4 stem> = <mor root>
<mor form4 ending> = ed
<mor form5 stem> = <mor root>
<mor form5 ending> = ed
<mor form6 stem> = <mor root>
<mor form6 ending> = ed
<mor form7 stem) = <mor root>
<mor form7 ending> = ing

where ε indicates the empty ending.

Now we assume that the stem and the ending will be combined in the word using the rule of English orthography, so that "love' with 'ing' gives rise to 'loving' and not 'loveing'.

Given this macro under its intended interpretation, we can exhibit the lexical entry for "love":

Lexeme love
<mor root> = love
mor_regV
sin_tV
<sem> = love2a

The first line here is self-evidently redundant, since the name of the lexical entry is identical with the root form. We can by convention always name a lexical entry in this way. Accordingly, we will henceforth treat:

Lexeme xxx

yyy

…:

zzz

as abbreviating:

Lexeme xxx

<mor root> = xxx

yyy

…

zzz

This means that our entry for “love’ can be simplified to:

Lexeme love

mor_regV

syn_tV

<sem> = love2a

Given our various macro and other conventions, this entry can be expanded to a set of feature specifications:

Lexeme love

<mor root> = love

<mor form1 stem> = love

<mor form1 ending> = ε

<mor form2 stem> = love

<mor form2 ending> = ε

<mor form3 stem> = love

<mor form3 ending> = s

<mor form4 stem> = love

<mor form4 ending> = ed

<mor form5 stem> = love

<mor form5 ending> = ed

<mor form6 stem> = love

<mor form6 ending> = ed

<mor form7 stem) = love

<mor form7 ending> = ing

<syn cat> = V

<syn arg0 cat> = NP

<syn arg0 case> = nom

<syn arg1 cat> = NP

<syn arg1 case> = acc

<sem> = love2a

Not all the verbs are regular. Both ‘eat’ and ‘give’ have wholly particular past tense forms together with ‘-en’ past participles. Their present tense and present participle forms are regular. In this case,

we will use another macro definition to capture a class of verbs including both of these.

```
Macro mor_presV
    <mor form1 stem> = <mor root>
    <mor form1 ending> = ε
    <mor form2 stem> = <mor root>
    <mor form2 ending> = ε
    <mor form3 stem> = <mor root>
    <mor form3 ending> = s
    <mor form4 stem> = <mor form5 stem>
    <mor form4 ending>= ε
    <mor form5 ending> = ε
    <mor form6 stem> = <mor root>
    <mor form6 ending> = en
    <mor form7 stem> = <mor root>
    <mor form7 ending> = ing
```

Here, for ‘eat’, the stem of form4 and form5 is ‘ate’; for ‘give’. The stem of form4 and form5 is ‘gave’.

Equipped with these abbreviations, our lexicon now looks as follows:

```
--Lexeme die:
    mor_regV
    syn_iV
    <sem> = die1a
 --Lexeme elaps:
    mor_regV
    syn_iV
    <sem> = elapse1a
--Lexeme eat:
    mor_presV
    <mor form4 stem> = ate
    syn_iV
    <sem> = eat1a
--Lexeme eat:
    mor_presV
    <mor form4 stem> = ate
    syn_tV
    <sem> = eat2a
--Lexeme give:
    mor_presV
    <mor form4 stem> = gave
    syn_tV
    <sem> = give2a
--Lexeme give:
```

```
        mor_presV
        <mor form4 stem> = gave
        syn_dtV
        <sem> = give3a
--Lexeme give
        mor_presV
        <mor form4 stem> = gave
        syn_datV
        <sem> = give3b
--Lexeme hand:
        mor_regV
        syn_dtV
        <sem> = hand3a
--Lexeme hand:
        mor_regV
        syn_datV
        <sem> = hand3b
--Lexeme love:
        mor_regV
        syn_tV
        <sem> = love2a
```

7.9 Word Form Clause (WFC)

Now we have a way to express certain kinds of lexical knowledge. We can think about how this knowledge is to be used in a NLP system.

For the moment, let us consider this from the point of view of a language analysis system. Such a system is represented with a sequence of unanalyzed words as its input and needs to determine the relevant features for each words – that is, syntactic features for parsing and semantic features for determining the meaning. But, as all the lexical knowledge is about word roots, to determine the relevant features, we need to determine how the word can be analyzed as being some form of a root which is in the lexicon.

For this, we need to be able to analyze a word in terms of the possible stems and endings that make it up. Basically, the approach is to look at the end of the word for a regular English ending and propose the part of the word before this ending as a possible root.

The process will have to be modified to take account of the rules of English orthography. For instance, ‘loving’ can be analyzed as stem ‘love’ with ending ‘ing’.

One way of looking at the relations between a word and a lexeme is in terms of a ‘Word Form Clause” (WFC). WFC state the conditions under which the relation can hold. Then when we have a word and a possible lexeme that it might be related to, we attempt to find a WFC that can be applied. As in application of grammar rule, we try to construct an extension of the DAG representing the word, which will only contain information about the stem and the ending, and the DAG representing the lexeme, which contain only general information that can be applied to all forms of the lexeme, in such a way that the WFC condition are satisfied. The computation of these extensions then results in the words inheriting extra syntactic and semantic features from the

lexeme, as well as specific properties described in the WFC.
Here is a WFC for the third-person singular, present Tense form class of English verbs:

WFC third_sing:
<word mor form> = <lexeme mor form3>
<word syn> = <lexeme syn>
<word syn cat> = V
<word syn arg0 per> = 3
<word syn arg0 num> = sing
<word syn tense> = pres
<word sem> = <lexeme sem>

This WFC says that a word is the third singular form of the lexeme if the following conditions can be satisfied. The ‘form’ of the word must be the same as the ‘form3’ form of the lexeme; that is, the stem values must both unify, as must the ending values. The ‘syn’ path of the word must unify with the ‘syn’ specifications shown; it must unify with the ‘syn’ of the lexeme and have the mentioned extra properties. Finally, the ‘sem’ of the word must unify with the ‘sem’ of the lexeme.

Here is how we would use WFCs in a language analysis program. From an output of the morphological analyzer, we can build a partial description of a word, consisting of its <mor form>, with a stem and an ending. For example, for the word ‘loves’, we could build the following:

Word loves:
<mor form stem> = love
<mor form ending> = s

We now need to determine a lexical entry and a way in which this word is related to that entry. Let us assume that we try the lexeme ‘love’ and the relation ‘third_sing’. So, we now attempt to find extensions for the ‘loves’ DAG, which plays the role of the ‘word’, and for the DAG for the lexeme ‘love’, which plays the role of the ‘lexeme’, in such a way that the conditions of the WFC are satisfied. That is, we attempt the unifications specified in the ‘third_sing’ WFC, with ‘word’ taken to indicate the word DAG and ‘lexeme’ the DAG provided by the lexical entry for ‘love’. In this case, all the unification succeed. If they did not, we would have to try another possible WFC or lexical entry. As a result of the unification, we find an extension to the word DAG, describable as follows:

Word loves:
<mor form stem> = love
<mor form ending> = s
<syn cat> = v
<syn tense> = pres
<syn arg0 cat> = NP
<syn arg0 case> = nom
<syn arg0 per> = 3
<syn arg0 num> = sing
<syn arg1 cat> = NP
<syn arg1 case> = acc

<sem> = love2a

This is the kind of category structure that we can use for parsing.
To summarize, here is how lexical works in the language analyzer. Incoming words are processed by a morphological analyzer to yield possible stem and ending values. For each such result, possible lexemes are retrieved from the lexicon. For each possible partial word description and possible lexeme that it may be related to, possible WFCs are invoked. Each such successful invocation results in a full word description (DAG) which can be used in later stages such as parsing. In a system where we have many lexical entries and WFCs, indexing of the lexical data structures will be vital for efficient processing.

Reading list for students for Course CS-579 (EECS, KAIST, Deajeon, Korea)

[1] R. Hausser, Foundations of Computational Linguistics (man-machine communication in natural language), Springer-Verlag, 1999.
[2] G. Gazdar, Ch. Melish, Natural Language Processing in LISP, Addison-Wesley Publishing Company, 1989.
[3] T. Winograd, Language as a Cognitive Process, Vol.1: Syntax, Addison-Wesley, 1983.
[4] R. Grishman, Computational Linguistics, Cambridge Univ. Press, 1986.
[5] Feng Zhiwei, Fundamentals of Computational Linguistics, Commercial Press, Beijing, 2001.
[6] Feng Zhiwei, Natural Language Processing by Computer, Foreign Language Education Press, Shanghai, 1996.
[7] Feng Zhiwei, Mathematical Linguistics, Knowledge Press, Shanghai, 1985.
[8] Feng Zhiwei, Mathematics and Language, Hunan Education Press, Changsha, 1991.