Chapter 5

Feature structures

The feature structure module outlines which feature names, feature values and feature structures occur in the Minimalist Program. Furthermore, operations on feature structures such as checking are defined. Many of the ideas described in this module derive from feature-based grammar [Shi86, Joh88]. Furthermore, ideas from typed feature logic [Car92] are adopted.

In the minimalist framework ideas about features are not worked out in much detail. The distinction between feature and value is not made by Chomsky [Cho93, Cho95]. Examples of what Chomsky calls features are: [nominative], [3 person] and [-human]. Apparently, both combinations of a feature and a value, and isolated values are referred to as features. Zwart does distinguish between features and values.¹

The definition of checking is not made explicit in minimalist work. This chapter outlines the definition of checking that was deduced from the minimalist literature and from experiments with formalizations and implementations.

In Section 5.1 I will give an introduction to the treatment of features within the minimalist framework. Note that I sometimes refer to the formalization in the introduction. For instance, when I sum up the feature names that occur in the Minimalist Program, I add, for the sake of convenience, all the feature names that occur in the formalization which do not correspond to a feature name from the Minimalist Program. Linguistic motivation for certain choices in the formalization is also given in this section. In Section 5.2 I will discuss the formalization of feature structures.

¹See for instance [Zwa97, Page 186-187].
5.1 Introduction

Feature structures  The notion ‘feature structure’ is not applied in the Minimalist Program. It is assumed that the nodes of a derivation tree may contain features, also called ‘feature bundles’. As I noted above in this section, what is referred to as a feature is actually a feature-value pair. In the next two sections we will see that operations on features, such as feature checking, require that the features belonging to a node can be treated as a unit. Therefore the features belonging to a node will be referred to as ‘feature structures’.

Feature names and feature values  Within the minimalist theory the most well-known feature names are [person], [number], [gender], [case], [tense], [category] and [wh]. All the feature names, except (maybe) for [wh] speak for themselves, especially when their possible values will be presented later on in this section. With the feature name [wh] we can indicate whether a word is an interrogative word or not. The [wh] feature forces, for instance, interrogative words like *wie* (who) to move to the sentence-initial position (i.e. the specifier of CP) in sentences such as the one in Example 5.1.

Example 5.1

Wie ziet hij?
(Who sees he?)
Who does he see?

Furthermore, the formalization contains a number of feature names that are less common or not common at all within the minimalist framework: [word], [sememe], [inversion], [comp cat], [speccat] and [determiner].

The feature [word] represents the phonological information of a lexical item. The feature takes an inflected word as its value, for instance walks, which is an inflected form of the verb to walk. Hence, the feature [word] is a simplification of the phonological features of a lexical item. The exact character of the phonological features needed within the minimalist framework is not worked out yet. Therefore I applied this simplified version of phonological features.

The feature [sememe] represents the semantic information of a lexical item. The feature takes the stem of a word as its value, for instance walk which is the stem (i.e. uninflected form) of the verb to walk. The Minimalist Program is not explicit about semantic features and therefore I use the feature name [sememe] to replace semantic features. A more precise approach to semantic features is not required for the formalization. Of course LF
is the interface where semantic features are needed, but as the role of this interface has not yet been studied very intensively within the minimalist framework, this aspect of semantic features falls outside the realm of the formalization project described here.²

The feature [inversion] enables movement of the verb to the head of the CP, to obtain the right order for yes/no-questions. In Example 5.2 we see that in yes/no-questions in Dutch, the finite verb moves to a sentence-initial position. The inflection of the inverted form (koop) of the verb that is required in yes/no-questions is different from the inflection of the non-inverted form (koopt). Hence, in the lexicon, koop and koOPT are two different items, with different values for the feature [inversion]. The rest of the features, such as [tense] are equal for both lexical items. The analysis of inversion described here is a simplification of Zwart’s analysis of inversion (cf. [Zwa97, Page 245ff]).

Example 5.2

(a) Jij koopt het boek
   You buy the book
(b) Koop jij het boek?
   (Buy you the book?)
   Do you buy the book?

The features [comp cat] and [spec cat] indicate respectively the category of the complement and specifier that a given head may select. It is also possible that no complement or specifier is selected. Note that these features are not applied in the Minimalist Program. In Chapter 7 I will go deeper into the necessity of the features [comp cat] and [spec cat].

The feature [determiner] is also a feature which does not occur in the minimalist framework. I apply it to indicate whether a noun needs a determiner or not. For instance, singular count nouns like meisje (girl) need a determiner, while personal pronouns like wie (who) may not occur with a determiner.

In the following table, all the feature names mentioned in this paragraph so far are summed up together with the possible values they received in the formalization:

<table>
<thead>
<tr>
<th>feature name</th>
<th>feature value</th>
</tr>
</thead>
<tbody>
<tr>
<td>persons</td>
<td>first, second, third</td>
</tr>
<tr>
<td>number</td>
<td>singular, plural</td>
</tr>
<tr>
<td>gender</td>
<td>masculine, feminine, neuter</td>
</tr>
<tr>
<td>case</td>
<td>nominative, genitive, accusative, dative</td>
</tr>
</tbody>
</table>

²Cf. [Cor96, Page 3], [Cor97, Page 7] and [Sta96, Page 105] for comparable approaches to semantic and phonological features.
Besides the atomic feature names (i.e., feature names that take atoms as their values), the formalization contains three complex feature names. Complex feature names take a feature structure as their value. The three complex feature names in our version of the Minimalist Program are: [agreement], [object] and [subject].

The feature name [agreement] takes a feature structure as its value that may contain [person], [number] and [gender] features.

The complex feature names [subject] and [object] represent the features of the subject and the object of the verb respectively. The complex feature name [object] may contain [case] and [agreement] features. The complex feature name [subject] may only contain an [agreement] feature.\(^3\)

The use of the [object] feature may not be self-evident. In the Minimalist Program it is assumed that there is both subject and object agreement, although in most languages object agreement is commonly satisfied because the agreement features of the object, as opposed to the features of the subject, do not influence the morphology of the verb. An example of a language with object agreement is Mohawk (cf. Baker [Bak96]).

**Operations on feature structures** The most important operation on feature structures is feature checking. Feature checking is applied when a constituent moves in the Minimalist Program. Moreover, a constituent may only move when it can check one or more formal features (contained in a feature structure) in the position it moves to. In our non-derivational version of the Minimalist Program this is put in the following way: constituents that have a connection with a lower constituent in the same tree must check their features against the head of the functional projection they are part of. The first argument of the function \(\text{Check, fstruct1}\), corresponds to the head of the functional projection; the second argument, \(\text{fstruct2}\), corresponds to the constituent which is connected to a lower constituent. \(\text{fstruct2}\)

\(^3\)The feature name [subject] may not contain a [case] feature since it occurs within feature structures describing verbs, and verbs do not assign case to the subject. According to the Minimalist Program [Cho95, Page 174], the verbs assigns case to the object while the functional head T assigns case to the subject. See also Chapter 8.
contains at least as many feature-value pairs as \textit{fstruct1}.

For instance, \textit{fstruct2} is a feature structure that contains all formal features of the verb \textit{loopt} (walks) that adjoins to AgrS in Example 5.3. The feature structure \textit{fstruct2} is represented in Example 5.4. \textit{fstruct1} is the feature structure containing the formal features of one of the functional projections, say AgrSP, where the verb \textit{loopt} adjoins to, to check its features. The feature structure \textit{fstruct1} is represented in Example 5.5. As not all features of \textit{fstruct2} are always checked in the same functional projection, it is clear that \textit{fstruct2} must contain as many features as \textit{fstruct1} or more. In the case of Example 5.3 \textit{fstruct2}, associated with V, contains more features than \textit{fstruct1}, associated with AgrS.

Example 5.3


graph
\{    edge[style=dashed]     AgrSP Spec AgrS     AgrS TP     AgrS T AgrS     AgrO T     V AgrO \}
\}

\textit{loopt}

Example 5.4

\[
\begin{array}{c}
\text{AGREEMENT} \\
\text{PERSON} \quad \text{third} \\
\text{NUMBER} \quad \text{singular} \\
\text{GENDER} \quad \text{masculine} \\
\text{CASE} \quad \text{nominative} \\
\text{TENSE} \quad \text{present} \\
\text{INVERSION} \quad \text{no} \\
\end{array}
\]

Example 5.5

\[
\begin{array}{c}
\text{AGREEMENT} \\
\text{PERSON} \quad \text{third} \\
\text{NUMBER} \quad \text{singular} \\
\text{GENDER} \quad \text{masculine} \\
\end{array}
\]
Chomsky’s treatment of Wh-movement in the Minimalist Program (cf. [Cho93]) played an important role in determining how to formalize checking. According to Chomsky, Wh-words have Wh-features while other nouns do not. Wh-words have to move higher than other nouns because they need to check this extra feature (in the specifier of CP).\footnote{Cf. [Cho93].} In Example 5.6 we see that the subject DP moves to the specifier of CP (to check its Wh-feature).

In the formalization we allow that all feature structures belonging to words of the category noun may have a Wh-feature. Hence, the lexical item referring to Wh-words contains the feature-value pair \textit{WhWord Yes}. Non-Wh-nouns, on the other hand, should not contain a Wh-feature. If we assume that checking is a unification operation (see Appendix) then the feature could check with any possible value (either \textit{Yes} or \textit{No}), since the unification of two feature structures \texttt{fstruct1} and \texttt{fstruct2} yields (if it exists) the smallest feature structure \texttt{fstruct3} that is subsumed by both \texttt{fstruct1} and \texttt{fstruct2}. A feature structure A subsumes another feature structure B if B contains all the feature-value pairs that A contains and possibly more (cf. [Shi86]). \texttt{fstruct1} and \texttt{fstruct2} unify if feature names that occur in both feature structures have compatible feature values. If a feature name only occurs in one of the two feature structures that are unified, this feature name (together with its feature value) is part of \texttt{fstruct3}.

\textit{Yes} is definitely the wrong value for a non-Wh-noun. Therefore we could choose an approach in which all non-Wh-words get the value \textit{No} for the feature \textit{WhWord}. But this approach also causes problems. The idea behind checking in Chomsky’s framework is namely that if a certain formal feature is present, it \textbf{must} be checked. Hence, a constituent with a feature-value pair \textit{WhWord No} should check its features. But movement to the specifier of CP is only desirable for DPs containing a Wh-word, not for DPs containing a non-Wh-word. It is of course possible to define that all infe\-\textit{ctional features except for \textit{WhWord No} have to be checked, but making checking an asymmetric variant of unification is more elegant. In that case \textit{Yes} can be considered the only possible value for the feature \textit{WhWord}. If the feature then is not present in a feature structure belonging to a non-Wh-word this really means that the word does not have a \textit{WhWord} feature. Hence, no feature needs to be checked in the specifier of CP.

The formalization of Zwart’s framework is different. As lexical heads no longer need to check all their features, it is possible to introduce a second value for the Wh-feature. Besides the value \textit{Yes} we can introduce the value \textit{No} since the feature-value pair \textit{WhWord No} may remain unchecked without causing the derivation to crash. Nevertheless, we maintain the definition of checking as it was developed for Chomsky’s framework. Namely, in Zwart’s
framework we need to verify that all features of the functional heads are checked before LF. This can be done by requiring the moved element to contain at least the features contained by the functional head serving as a landing site.

Example 5.6

Furthermore the feature structure module contains two operations that can modify a given feature structure: the operations \texttt{Remove} and \texttt{Keep}. These two operations take a feature structure \texttt{fstruct1} and a set of feature names \texttt{nset} as their argument and yield a new feature structure \texttt{fstruct2} by removing or keeping \texttt{nset} from \texttt{fstruct1}. We will come across these operations several times in the coming chapters. For instance, the operation \texttt{Keep} is used to get rid of all the non-formal (i.e. phonological and semantic) features in a feature structures if we want to check the formal features of a lexical constituent against those of a functional constituent.

5.2 The formalization

\textbf{Feature structures} In the Minimalist Program it is assumed that the nodes of derivation trees may contain feature-value pairs, often simply called features. I assume that with each node of the final LF-tree a feature structure is associated. A feature structure is a set of zero or more feature-value
pairs. In the Minimalist Program the notion ‘feature structure’ is not as
common as it is in other linguistic frameworks such as feature-based gram-
mar [Shi86, Joh88]. However, feature structures are applied in the formal-
ization, because, if the feature-value pairs of a node are treated as a unit,
it is easier to refer to the features of a node. Furthermore, this approach
facilitates defining operations on the features of a node, such as feature
checking.

In the formalization, feature structures are defined as ‘functions from
feature names to feature values’, as we see in Definition 5.1. There
are three sorts defined in this example: FeatureNames (feature names),
FeatureValues (feature values) and FeatureStruct (feature structures).
Feature structures are defined as a sub-sort of feature values, which means
that complex feature values are allowed (see also Section 5.1). The function
Value is a function from feature structures and feature names to feature
values. This means that given a feature structure and a feature name, the
function yields the feature value belonging to the feature name in the given
feature structure. The function is partial (PARTIAL) because not every fea-
ture structure contains every possible feature name. It is even possible that
a feature structure does not contain any feature name at all. In this case
the feature structure is an empty feature structure (see also Section 4.1).

The fact that Value is a function implies that it is impossible that a
feature name can occur more than once per feature structure with di-

ferent feature values, since functions may only yield one result.

The function Value is crucial to the formalization of equality in feature
structures: if for all feature names name in the feature structures fstruct1
and fstruct2 applies that the feature values val are equal (i.e. of the
same value), then fstruct1 and fstruct2 are equal. The feature structures
fstruct1 and fstruct2 do not have to be in the same order in order for the
structures to be equal. The only requirement is that they contain exactly
the same feature-value pairs. If fstruct1 does not contain a given feature
name name, then fstruct2 may not contain the feature name name either
(i.e. partiality).

Definition 5.1

\[
\begin{align*}
\text{SORT} & \quad \text{Name} & \quad \text{FeatureName} \\
\text{SORT} & \quad \text{Val} & \quad \text{FeatureValue} \\
\text{SORT} & \quad \text{FStruct} & \quad \text{FeatureStruct} \\
\text{DECL} & \quad \text{Name} & \quad \text{Name} \\
\text{DECL} & \quad \text{Val} & \quad \text{Val} \\
\text{DECL} & \quad \text{FStruct} & \quad \text{FStruct} \\
\text{FUNC} & \quad \text{Value} & \quad \text{FStruct, Name} \rightarrow \text{Val} \\
\end{align*}
\]
AXIOM FORALL name 
  
  ( Value (fstruct1, name) = Value (fstruct2, name) ) 
  
  ==> fstruct1 = fstruct2

**Feature names and feature values** The feature module of the formalization contains feature names, among other things. Some of those take atomic values, namely: **Category, Person, Number, Case, Gender, WhWord, Determiner, CompCat, SpecCat, Tense, Inversion, Sememe, and Word.**

Atomic feature names are referred to as \textit{AtomNameS} in the formalization. Definition 5.2 shows that \textit{AtomNameS} are a sub-sort (\texttt{<<<}) of \textit{FeatureNameS}.

**Definition 5.2**

\begin{verbatim}
SORT FeatureNameS
SORT AtomNameS <<< FeatureNameS
SORT StructNameS <<< FeatureNameS
\end{verbatim}

The rest of the feature names (\textit{Agreement, Subject and Object}) take complex values (that is, feature values that are feature structures). Complex feature names are referred to as \textit{StructNameS} in the formalization. Definition 5.2 shows that \textit{StructNameS} is a sub-sort (\texttt{<<<}) of \textit{FeatureNameS}.

Together with feature values, feature names build feature-value pairs. In the formalization a type is assigned to each feature value.\footnote{Cf. \cite{Car92}.} For example, all person values are of the type \textit{Values[Person]}. In this way it is impossible to assign, for instance, a number value such as \textit{Plural} to the feature name \textit{Person}, as number values are of the type \textit{Values[Number]} and \textit{Person} needs a value of the type \textit{Values[Person]}. As we saw in Section 3.2, individual objects such as the feature value \textit{Plural} take a sort name as a type: \textit{Values[Number]} in the case of \textit{Plural (OBJ Plural : Values[Number])}.

In the formalization, feature values are of the sort \textit{FeatureValueS} (see Definition 5.3). This sort has two sub sorts: \textit{AtomS} and \textit{FeatureStructS}. The former sub-sort represents atomic feature values (such as \textit{Plural}); the latter represents complex feature values (i.e. feature values that are feature structures, such as the value of the feature name \textit{Agreement}, which is a feature structure that may contain person, number and gender features).

**Definition 5.3**

\begin{verbatim}
SORT FeatureValueS
SORT AtomS <<< FeatureValueS
SORT FeatureStructS <<< FeatureValueS
\end{verbatim}
Complex values are not only assigned a type, for instance `Values[Agreement]`. For complex values it is also specified which types of feature-value pairs can form part of the feature structure.\(^6\) A feature structure that serves as a value for the feature `Agreement` contains zero or more of the features `Person`, `Number` or `Gender` as its value. A feature structure that serves as a value for `Subject` or `Object` contains zero or more of the features `Agreement` or `Case`. Note that not all of the possible feature names for a certain type of feature structure must be present in each occurrence of that type. Hence, a feature structure does not need to be well-typed in Carpenter's sense [Car92, Page 88]. For example, the functional projection T has a `Subject` feature which has a value only containing a `Case` feature as only the case feature of the subject must be checked here. Hence, the subject agreement features of T are underspecified.

For each type of lexical item (i.e. noun, verb etc.) it is also specified which type of feature-value pairs it may contain. For example, a feature structure that represents a lexical item describing a noun, contains different features from a feature structure belonging to a complementizer. The above is summarized in the following table:

<table>
<thead>
<tr>
<th>complex feature name</th>
<th>feature names</th>
</tr>
</thead>
<tbody>
<tr>
<td>Agreement</td>
<td>Person, Number, Gender</td>
</tr>
<tr>
<td>Subject</td>
<td>Agreement, Case</td>
</tr>
<tr>
<td>Object</td>
<td>Agreement, Case</td>
</tr>
</tbody>
</table>

The following table shows which features-value pairs a given lexical item may contain:

<table>
<thead>
<tr>
<th>Lexical item</th>
<th>Features</th>
</tr>
</thead>
<tbody>
<tr>
<td>N</td>
<td>Word, Sememe, Agreement, Case, Determiner, WhWord, CompCat and SpecCat</td>
</tr>
<tr>
<td>D</td>
<td>Word, Sememe, Agreement, Case, Determiner, WhWord, CompCat and SpecCat</td>
</tr>
<tr>
<td>V</td>
<td>Word, Sememe, Subject, Object, CompCat, SpecCat, Inversion and Tense</td>
</tr>
<tr>
<td>AgrO</td>
<td>Object, CompCat and SpecCat</td>
</tr>
<tr>
<td>AgrS</td>
<td>Subject, CompCat and SpecCat</td>
</tr>
<tr>
<td>T</td>
<td>Subject, Tense, CompCat and SpecCat</td>
</tr>
<tr>
<td>C</td>
<td>Word, Sememe, Inversion, Tense, WhWord, Agreement, CompCat and SpecCat</td>
</tr>
</tbody>
</table>

In Chapter 8 I will argue why certain lexical items contain certain features.

\(^6\)Cf. type signature [Car92].
Operations on feature structures

The feature module defines an important operation on feature structures, namely Check. Check defines the minimalist operation feature checking. Furthermore, two less central operations Remove and Keep are defined here.

The operation Check is a boolean function that checks whether two feature structures, fstruct1 and fstruct2, match. The feature structures need to match in the sense that any feature name that occurs in fstruct1 must occur in fstruct2. For feature-value pairs that occur in both fstruct1 and fstruct2 that have a feature structure as a value, those feature structures will also have to be checked against each other. If a feature name occurs in fstruct1 and not in fstruct2 checking fails, or, to put it differently, checking succeeds only if fstruct1 subsumes fstruct2.

Definition 5.4 shows the formal definition of checking. The first conjunct of the definition says that each atomic feature name aname that occurs in fstruct1 with the atomic feature value atom must also occur in fstruct2 with the same feature value. The second conjunct says that for all the complex feature names sname in fstruct1 that have a complex feature value fstruct3, fstruct3 must be checked against the value of the feature sname in fstruct2.

Definition 5.4

\begin{verbatim}
PUBLIC Check : FeatureStruct, FeatureStruct -> BoolS
AXIOM fstruct1 Check fstruct2
    \leftrightarrow FORALL aname, atom
    ( Value (fstruct1, aname) = atom
    => Value (fstruct2, aname) = atom
    )

And FORALL sname, fstruct3
    ( Value (fstruct1, sname) = fstruct3
    => fstruct3 Check Value(fstruct2, sname)
    )
\end{verbatim}

Furthermore the module about feature structures contains two operations that can change a given feature structure.

Remove is a function that given a feature structure fstruct1 and a set of features nset returns the value fstruct2 where for all feature names name the following holds: if name is not a member of nset then the value of name for fstruct2 equals the value of name for fstruct1, and else the value of name for fstruct2 is undefined (see Definition 5.5).

The function Keep has the opposite effect of the function Remove: if name is not a member of nset then the value of name for fstruct2 is unde-
fined, and else the value of name for fstruct2 equals the value of name for fstruct1 (see Definition 5.5).

**Definition 5.5**

FUNC Remove : FeatureStructS, Sets[FeatureNameS] -> FeatureStructS

FUNC Keep : FeatureStructS, Sets[FeatureNameS] -> FeatureStructS

DECL nset : Sets[FeatureNameS]

AXIOM fstruct1 Remove nset = fstruct2

=> ( Not (name In nset)
    => Value (fstruct2, name) = Value (fstruct1, name)
    )

And ( name In nset
    => Value (fstruct2, name) = Undef
    )

AXIOM fstruct1 Keep nset = fstruct2

=> ( Not (name In nset)
    => Value (fstruct2, name) = Undef
    )

And ( name In nset
    => Value (fstruct2, name) = Value (fstruct1, name)
    )

5.3 **Summary**

In this chapter I outlined why I introduced the notion ‘feature structure’ in the formalization, although this notion does not occur in the Minimalist Program. The main reason is that it is convenient to be able to treat the features of a node in a tree as a unit. I also described how feature structures are formalized in the formalization. Furthermore, I described the feature names that occur in the formalization, plus their possible values. Not all the feature names used in the formalization are present in the minimalist framework. Finally, I introduced some operations on feature structures of which the most important is the operation ‘feature checking’. I argued that the features of a lexical constituent can only be checked against the features of a functional head if the feature structure belonging to the lexical constituent contains at least as many feature-value pairs as the feature structure belonging to the functional head.