Chapter 3

Formalization

This chapter contains a global description of a project on formalization of fragments of the Minimalist Program [Cho93], and of ideas of Zwart [Zwa97] in combination with a revised version of the Minimalist Program [Cho95]. The project was carried out in cooperation with the Department of Computing Science of the University of Groningen. There, Erik Saaman (in cooperation with Gerard Renardel de Lavalette and Rix Groenboom) developed a formal-specification language called AFSL (Almost Formal Specification Language) in which the formalization of the Minimalist Program is written (See [Saa] and [Gro97]). The project on the formalization of a fragment of the Minimalist Program served as a case study for the formal-specification language AFSL. Another case study for AFSL was on formalizing rules for anaesthesiologists. The use of AFSL within this case study and the formal model are presented in [GSRdL96], the full specification can be found in [GRdL96]. For the artificial intelligence aspects (the diagnostic reasoning principles) I refer to [RdLGR+97].

In Section 3.1 of this chapter I will advance several reasons for formalization. Section 3.2 contains a brief description of the formal-specification language AFSL. In Section 3.3 I will discuss the differences between Chomsky’s 1993 framework as discussed in Chapter 1 and Zwart’s 1997 framework. And finally, in Section 3.4 I will discuss which parts of the Minimalist Program are covered by the project described here and in what way.

3.1 Why formalization?

The objective of the formalization project described here is to obtain a formal and explicit version of a part of the Minimalist Program [Cho93],
After the two implementation projects described in Chapter 2 there appeared to be enough reason to start with the formalization of the relevant parts of the theory before implementing them.

I used the two implementation projects, both conducted in the programming language Prolog, to get an impression of the possibilities of formalizing parts of the Minimalist Program. The projects, which were both based on Chomsky’s 1993 version of the minimalist theory, revealed that the main problem was to find clear and explicit definitions of the relevant notions in the literature. This lack of explicitness was a reason for not continuing a project on the formalization of an earlier version of Chomskyan linguistic theory [Dor93]. I think, on the other hand, that formalization can contribute to the development of new theories such as the Minimalist Program. By providing formal definitions of important notions it becomes, for instance, easier to find inconsistencies and insufficiencies in the theory.

I concluded that it would be more interesting to come up with clear definitions of a range of important notions from the Minimalist Program than to build working implementations. Since specification languages can be considered to be a middle course between natural languages and programming languages, as we will see further on in this section, a formal specification language seemed to be more appropriate for this goal than a programming language. The remainder of this section deals with reasons for formalization, both from the point of view of computing science and from the point of view of linguistics.

First I will discuss a reason for formalization from the point of view of computing science. When developing software a possible approach is: making an informal description of the system and then encode a program using a programming language. In computing science it is generally accepted that more advanced software development techniques are essential. The utilization of formal specification languages is one possible technique. In this approach an additional step is introduced between the informal description and the final implementation: the informal description is translated into a formal description in a specification language and the formal description is then translated into a working implementation. In this way the step from informal to formal is made more gradually, which reduces the chance of mistakes.

Also, when using a formal specification language, the ideas that are formalized are possibly made more explicit than when using a programming language right away. A programming language often forces the user to ap-

---

1 See Fong [Fon91] and Stabler [Sta92] for formalizations and implementations of Government and Binding Theory [Cho81, Cho86a]. See Stabler [Sta96] and Cornell [Cor96, Cor97] for formalizations and implementations of the Minimalist Program [Cho93, Cho95].

2 Cf. [MNB94], [MCW96] and [Gro97].
ply certain data structures while the choice of a data structure for a given concept can be postponed when using a formal-specification language. For example, in the Prolog implementation of our formalization trees are implemented as lists. The data structure ‘list’ has a lot of properties which trees implemented as lists inherit. In the formalization no choice is made for a certain data structure with respect to trees. By postponing the choice for a certain data structure one is forced to be more explicit.

An additional advantage of the use of a formal-specification language is that formalizations are more accessible to people without a background in programming in general or a certain programming language in particular. In the case of the Minimalist Program this is of major interest. The project described here not only provides a whole range of explicit definitions which might be of interest to linguists working within the minimalist theory. It also provides a formal version of part of the theory which is easier to grasp for linguists working in other frameworks and which facilitates the detection of consequences of, and inconsistencies in the theory.

This brings us to the attractiveness of formalization from a linguistic point of view. In informally stated theories it is hard to discover errors. Furthermore it is nearly impossible to determine the consequences of informal theories. Hence, a formally stated theory is not only attractive for computational linguists wanting to implement the theory, it is also very important for linguists working within the theoretical framework.

When using a specification language one is forced to be precise. Every notion used in a definition needs to be defined itself. Of course, programming languages also offer this advantage, but I discussed above reason for preferring the more gradual step of formalization. The following definition (3.1) from Radford [Rad88, Page 110] shows that this is not necessarily the case when using natural language.

**Definition 3.1**
A node X dominates another node Y if X occurs higher up the tree than Y and is connected to Y by an unbroken set of solid lines (branches)

Definition 3.1 contains notions like ‘higher up’, ‘unbroken set’ and ‘solid lines’ that are never defined. These notions are assumed to be known by the reader. However, [Rad88, Page 109] claims that ‘any adequate description of a phenomenon in any field of enquiry (in our present case, Syntax) must be maximally explicit, and to be explicit, it must be formal – i.e. make use of theoretical constructs which have definable formal properties’.

If we have a closer look at Definition 3.1, we see that it is possible to interpret the definition in such a way that the sister (X) of the mother of a node Y is dominating Y, which should not be true (see Example 3.1). This example shows that it is advantageous to use a formal language instead of
natural language when definitions are needed. Naturally, formal definitions can be incorrect, but then at least the mistake is explicit and therefore easier to discover.

Example 3.1

\[
\begin{array}{c}
W \\
/ \ \\
X \\
/ \ \\
Z \\
/ \\
Y
\end{array}
\]

Definition 3.2 contains the natural language version and Definition 3.3 the formal version of the definition of ‘proper domination’ that is applied in the formalization. The notions ‘node’ and ‘mother’ are also defined in the formalization. A description of the language AFSL, which might be useful to understand Definition 3.3, is given in Section 3.2.

Proper domination is a type of domination where a node cannot dominate itself. Reflexive domination is a type of domination where a node can dominate itself. This distinction is made because it is relevant for the formalization. Note that the Definition 3.1 describes proper domination.

Definition 3.2
A node nd₁ properly dominates a node nd₂ if and only if nd₁ is the mother of nd₂ or if there exists a node nd₃ of which nd₁ is the mother and that properly dominates nd₂.

Definition 3.3

\[
\text{AXIOM} \quad \text{nd}_1 \text{ PropDominates } \text{nd}_2 \quad \Longleftrightarrow \quad \text{nd}_1 = \text{Mother } \text{nd}_2 \\
\text{Or} \quad \exists \text{nd}_3 \quad \left( \text{nd}_1 = \text{Mother } \text{nd}_3 \\
\text{And} \quad \text{nd}_3 \text{ PropDominates } \text{nd}_2 \right)
\]

In the literature there has been a lot of discussion about what the formalization of grammatical theory should imply [Cho57, Pul89, Cho90, Lud92, Sta92a, Pol93]. However, broadly speaking most participants in the discussion will agree on the three conditions for the adequate formalization of
grammatical theory which Pullum [Pul89] formulates. The conditions are paraphrased from ideas of Stoll [Sto61] and come down to the following:

1. it must be clear whether a certain mathematical object (e.g. a tree) can represent a structural description according to the theory\(^3\)

2. it must be clear whether a certain formal object represents a rule (or constraint or principle or condition etc.) of the grammar

3. it must be clear whether a certain structural description is generated by (or admitted by) a given set of rules

Only Ludlow [Lud92] claims that Pullum’s conditions are ‘far too strong’. Ludlow’s claim is based on the fact that he concludes that Pullum’s conditions imply that there must be ‘an algorithm to determine the predictions of the theory’. If this conclusion was right, Pullum’s conditions would indeed have been far too strong, since there is no science in which all the logical consequences (theorems) of a theory could be determined. However, Pollard [Pol93] argues that Pullum’s conditions are much more reasonable than what Ludlow claims.

Pollard shows that in the first condition, Pullum expresses that the models of a theory have to be defined. One has to be able to decide whether a certain object is a possible structural description or not. The constraints on structural descriptions do not have to be stated in first-order logic. A clear description in natural language is enough. In the formalization a special kind of trees, phrase markers, are the models of the theory. A tree is a phrase marker if it satisfies \(\chi\)-Theory and if all the movements it contains are made according to the rules of the theory as we will see in Chapter 8. A structural description carries semantic and phonetic information. Hence, in the formalization it is possible to test whether a tree is a structural description by checking if it is a correct object at the interface levels LF and PF.

Pollard argues that Pullum’s second condition demands that there is a clearly described finite set of rules. It has to be possible to check whether a given constraint is actually one of the constraints of the theory. This is only possible if all the constraints of a theory are unambiguously stated, possibly in natural language.

According to Pollard the demands of the third criterion would not have been reasonable if structural descriptions were infinite objects. However, structural descriptions are always finite (i.e. they have a finite amount of nodes or points), and therefore it is always possible to decide whether a

---

\(^3\) A structural description is a representation of the certain linguistic properties of an expression.
given structural description is licensed by the finite set of constraints of a given theory.

Chomsky [Cho57, Page 5] emphasizes the importance of precisely constructed models for linguistic structure. He argues that formalized theories may lead to the discovery of inconsistencies, and more positively to solutions for problems other than those for which the theory was designed.

Pullum [Pul89] shares this point of view with Chomsky. However, Pullum claims that Chomsky denied the importance of formalization on several occasions since 1979. On the other hand, Chomsky [Cho90] argues that this criticism is based on a misunderstanding.

To understand the discussion between Pullum and Chomsky, two notions are essential: E-language and I-language. Chomsky defines E-language as the set of well-formed (grammatical) expressions [Cho86b, Cho90]. I-language is defined as a (generative) grammar, which is a component of the language faculty of human beings [Cho86b, Cho90].

The notions E-language, I-language and structural description are interrelated in the following way. An I-language strongly generates a structural description for each expression. Hence, an I-language licenses certain structures and rejects others. Furthermore, an I-language weakly generates an E-language. Since an E-language is the set of all valid phonetic expressions, each of these expressions is assigned a structural description by the I-language. This makes the notion ‘E-language’ the less informative one.

According to Chomsky [Cho90], the misunderstanding between Pullum and him is based on confusion about the importance of the notion ‘E-language’. Pullum [Pul89, Page 138] defines formal linguistics as ‘the study of languages and grammars’, which is interpreted by Chomsky [Cho90, Page 144] as ‘the study of E-languages and the I-languages that weakly generate them’. Pullum assumes that Chomsky denies the importance of formal linguistics in general, while Chomsky claims that he only denies the importance of the notion ‘E-language’ for formal linguistics in the claim that grammar might ‘characterize languages that are not recursive or even not recursively enumerable, or even (…) not generate languages at all without supplementation of other faculties of mind’. Chomsky argues that ‘it is meaningless to ask whether (…) such an expression as misery loves company is, or is not, a member of the E-language weakly generated by [the I-language] L; and nothing would follow from a discovery (or stipulation) one way or another. These expressions have their status, determined by L; they are parsable, appropriate in certain situations, have a definite meaning, etc.’

Hence, in principle Chomsky recognizes the importance of formalization of the I-language, although he does not think that ‘Pullum’s injunction’ to make ‘a concerted effort’ to meet ‘the criteria for formal theories set
out in logic books’ would be taken seriously by the natural sciences. The implicit question is then why linguistics, which is much less advanced than the natural sciences, should take this seriously. As we saw above, Pollard [Pol93] shows that Pullum’s conditions are not much more reasonable than Chomsky claims here.

We can conclude that the formalization of linguistic theories is a reasonable and useful enterprise. But the availability of precise definitions in natural language would already be an enormous improvement. Pollard and Sag [PS94, Page 9] for instance, who emphasize the importance of formalization, claim in their book about HPSG that their rules and principles are expressed clearly and unambiguously, although put in natural language.

When I was collecting definitions for the formalization I noted that there are a lot of notions that do not have definitions at all. For example, the notion ‘feature checking’, which is a central notion within the Minimalist Program as we will see in the coming chapters, is not defined properly. From the literature we can deduce that feature checking implies that the features of a moving element (i.e. constituent) and a node that serves as a landing site for the moving element have to match. But we do not know, for instance, whether the moving element and the landing site must have exactly the same number of features or not. In Chapter 5 we will see that it is essential that for the definition of ‘feature checking’ that the moving element contains the same feature value pairs as the landing site, and possibly more. The whole idea of feature checking does not work without this addition. However, this detail could not be found in the literature.

The formalization described in the following chapters provides precise definitions for parts of the Minimalist Program. Which parts of the theory are covered is described in Section 3.4, but first I will give a brief description of the formal-specification language AFSL.

### 3.2 The FSA method and the language AFSL

In this section I will give a brief introduction to the specification language AFSL. The language is developed as a part of the Formal System Analysis method. Therefore this method will also be briefly discussed below.

**FSA** The Formal System Analysis (FSA) method, with its formal-specification language AFSL, is especially developed for the formalization of *knowledge domains*. Knowledge domains have certain typical features that impose requirements upon a formal-specification language:

- the language must be ‘lean and clean’, as domain experts must be able to understand the formalization. Therefore AFSL is based on
first-order logic.

- knowledge domains often lack clear and stable definitions, especially when the described theory is still evolving. Therefore the principle of stepwise formalization is introduced in AFSL: informal axioms can be applied to bridge the gap between the informal definitions in a dictionary and the formal definitions in the axiomatization.\(^4\)

- the variety of the domain requires that the language has a wide spectrum of possibilities. For instance, the language must be modular and provide the possibility of typing.

The formal-specification language AFSL is the result of the FSA project (cf. [Gro97], [Saal]) that has been carried out at the Department of Computing Science of the University of Groningen. The method for the construction of formal specifications that is developed within the scope of the FSA research project comprises three separate parts, of which the language AFSL is one. The other two parts are: a set of guidelines for formalization, and tool support to enable validation of a specification.

Roughly we can distinguish two types of specification methods:

- model-oriented specification: an approach to specification where a system is specified by defining an explicit model of it.

- property-oriented specification: an approach to specification where a system is specified in terms of its desired properties. Of course, the specification implicitly defines a model.

The FSA method applies the property-oriented method. This can be exemplified with the formalization project described here. The formalization includes a declarative description of trees within the Minimalist Program. Thus, the formalization describes the properties of trees that are correct according to the Minimalist Program. The type of tree (phrase marker) that is described is the model of the minimalist theory. We define which trees are legal phrase markers within the minimalist theory without referring to the process of constructing phrase markers.

The FSA method supports the whole process of making an implementation using formalization (see Figure 3.1). Firstly, the requirements of the relevant system are stated informally in natural language. Secondly, a formalization of the requirements is made. Thirdly, a formal implementation is based on the formalization. In principle this implementation is executable because all definitions from the formalization have been made

\(^4\)See further on in this section for more detailed descriptions of the notions 'dictionary' and 'axiomatization'.

explicit. However, a last (fourth) phase may be needed for the construction of an actually executable program. This program is informal since it has certain properties that differ from their formal counterparts. For instance, a program cannot have access to all possible integer values, but only to a finite amount of integers since programming languages must be finite.

![Diagram of formalization process]

Figure 3.1: Implementing while using a formal-specification language

The set of guidelines for formalization that was mentioned above consists among other things of a division of the formalization in three parts:

- a dictionary
- a signature
- an axiomatization

The dictionary is a list of informal descriptions of basic concepts of the domain. The axioms in the axiomatization stage are based on the descriptions from the dictionary. For instance, in Definition 3.2 we find the dictionary item for ‘proper domination’. In the discussion of the formalization in the chapters about the modules of the formalization we will not give any dictionary items. The formalization of the dictionary items and the informal descriptions that are given to explain the formalization have completely replaced the dictionary. Hence, it would be redundant to also give the dictionary items.

The signature is the first step towards formalization. Names (identifiers) and types are assigned to all the concepts from the dictionary. Identifiers refer to individual objects, sorts or functions. Individual objects take a sort name as a type. For instance, the object Category is of the sort AtomNameS (atomic feature names, i.e. feature names that take an atomic
value) which is a sub-sort of FeatureNameS (feature names). Another sub-sort of FeatureNameS is StructNameS, that is, feature names that take a feature structure as their values. For instance, the feature name Agreement refers to an object of the sort StructNameS. Its value is a feature structure containing one or more of the features Person, Number and Gender. The signature given here is represented in the formalization in the following way:

\[
\text{SORT FeatureNameS} \\
\text{SORT AtomNameS} \\
\text{SORT StructNameS} \\
\text{OBJ Category : AtomNameS} \\
\text{OBJ Person : AtomNameS} \\
\text{OBJ Number : AtomNameS} \\
\text{OBJ Gender : AtomNameS} \\
\text{OBJ Agreement : StructNameS}
\]

Functions are of the type:

\[A_1, \ldots, A_n \rightarrow B\]

where \(A_1, \ldots, A_n\) and \(B\) are sort names. For instance, LogicalForm is a function (or rather, a predicate) that is of the type TreeS \(\rightarrow\) BoolS. A tree is correct at LF if all the features that need to be checked are checked. In the formalization this is represented as follows:

\[
\text{FUNC LogicalForm : TreeS \rightarrow BoolS}
\]

The axiomatization is the formal counterpart of the dictionary. All the concepts described in the dictionary are translated into formal axioms.

The formalization process for the specification of the Minimalist Program started with collecting a list of relevant notions. Then definitions for the important notions of the theory in natural language were formulated. It was not possible to find explicit definitions for all the notions we collected. The definitions in natural language were translated into the formal-specification language AFSL. The relevant notions were collected in a so-called dictionary. The dictionary contains items consisting of a description of a notion in natural language (Definition 3.2). When the dictionary was sufficiently worked out, we started on the actual formalization. The formalization process resulted in the need to reformulate some of the definitions in the dictionary. It also happened that notions were removed from
the dictionary because they turned out to be irrelevant for the formalization. An example of a notion that has been removed is `branch’. While developing the dictionary, the notion ‘branch’ (connection between nodes with a mother-daughter relation) seemed relevant with respect to trees. In the formalization the notion was not applied in any of the functions and therefore it was removed from the dictionary.

**AFSL** I conclude this section with a brief description of the language AFSL. Note that the parts of the formalization that are given in the following chapters are always described extensively in the accompanying text so that the description of the language given here will probably be superfluous.

Modules are the basic building blocks of the specification-language AFSL. In a module the names of objects (OBJ), sorts (SORT), sub sorts (SUBSORT) and functions (FUNC) can be introduced. The introduction of the names is also known as the signature of the formalization. Below I will give some examples of name introductions from the formalization.

The object *Agreement* is defined as follows in the formalization:

\[
\text{OBJ Agreement : StructNameS}
\]

Agreement is a feature name that takes a feature structure as its value (StructNameS).

Definition 3.4 shows the name introduction of the sort FeatureValueS and its sub sorts.

**Definition 3.4**

\[
\begin{align*}
\text{SORT FeatureValueS} \\
\text{SORT AtomS} & \lll \text{FeatureValueS} \\
\text{SORT FeatureStructS} & \lll \text{FeatureValueS}
\end{align*}
\]

\lll means ‘is a sub-sort of’. Hence, atoms (AtomS) and feature structures (FeatureStructS) are both sorts that are sub sorts of feature values (FeatureValueS).

The SUBSORT construction is used to define an already introduced sort as a sub-sort. For instance, we could define FeatureValueS from Definition 3.4 as a sub-sort of the sort X by adding the following line to the formalization:

\[
\text{SUBSORT FeatureValueS} \lll X
\]

An example of the introduction of a function name is:
FUNC PossibleValue : FeatureNameS, FeatureValueS -> BoolS

It can either be true or false that a given feature value is a possible value for a given feature name. For instance, ‘first’ is not a possible feature value for the feature name ‘number’, but it is for the feature name ‘person’.

Sort names always end with a capital ‘S’ (for sort) in AFSL. The language contains seven predefined sort names:

- **EmptyS**: the empty sort
- **ObjectS**: the sort of all objects (every sort is a sub-sort of ObjectS)
- **BoolS**: the sort of boolean values
- **RealS**: the sort of real numbers
- **NatS**: the sort of natural numbers (including 0). NatS is a sub-sort of RealS
- **StringS**: the sort of strings
- **CharS**: the sort of strings of length 1. CharS is a sub-sort of StringS

AFSL contains two predefined function symbols: one for equality (=) and one for inequality (=/=).

Since AFSL is a first-order language, we can only quantify over objects (i.e. sort elements). Variables for quantification can be introduced using the DECL construction. In Definition 3.3, repeated below as 3.5, we see an example of the use of dummies (variables) within an AFSL axiom. The dummy for the sort **NodeS** is nd as is indicated in the following line from the formalization:

DECL nd : NodeS

Dummies can be extended with numbers (e.g. nd1, nd2) when dummies are needed for different objects of the same sort.

**Definition 3.5**

AXIOM nd1 PropDominates nd2

<=> nd1 = Mother nd2

Or EXISTS nd3

   ( nd1 = Mother nd3
   And nd3 PropDominates nd2
   )
In Definition 3.6 we find the first part of the parameterized module `FeatureNameM`. If this module is imported in another module in the following way:

```plaintext
IMPORT FeatureNameM [Name]
```

we dispose of all the properties that are assigned to the parameter `Name` in the module `FeatureNameM`. But which properties are assigned to `Name`?

- `Name` is an object of the sort `FeatureNameS` (feature names)
- the sort `ValueS[Name]` is a sub-sort of `FeatureValueS` (feature values). `ValueS[Name]` is the set of possible feature values for the feature name `Name`. For instance, the possible values for the feature name ‘number’ are ‘singular’ and ‘plural’
- the sort `ValueS[Name]` inherits the properties of sorts as described in the module `SortM`

**Definition 3.6**

```plaintext
MODULE FeatureNameM [Name]

OBJ Name : FeatureNameS

SORT ValueS[Name] << FeatureValueS

IMPORT SortM[ValueS[Name]]
```

As we see in Definition 3.6, not only modules but also names (of objects, sorts as well as functions) can be parameterized. This is very useful in cases as the following:

```plaintext
FUNC Value[Name] : FeatureStructS -> PARTIAL~ValueS[Name]
```

`Value[Name]` is a function from feature structures to values. In other words, if we want to know the value of a given feature name (`Name`) in a given feature structure, the value that is yielded has to be a possible value for the given feature name. The `PARTIAL` construction indicates that not every feature structure will always yield a feature value for a given feature name. The reason for this is the fact that not every feature name is present in every feature structure. In fact, a feature structure can even be empty.
3.3 Verb movement

In this section I will describe a version of the Minimalist Program that was developed by Zwart [Zwa97]. This version is a combination of Chomsky’s 1993 version of the Minimalist Program (cf., [Cho93]), Chomsky’s 1995 version of the Minimalist Program (cf., [Cho95]), and some of Zwart’s new ideas that enable the proper description of verb movement in Germanic languages such as Dutch and German, that show an asymmetry with respect to the position of the finite verb between main and subordinate clauses.

Example 3.2 shows that in subordinate clauses (a), the final verb appears to the right of the subject and the direct object. In main clauses (b) on the other hand, the final verb appears in between the subject and the direct object.

**Example 3.2**

(a) dat de hond een bal vindt
    (that the dog a ball finds)
    that the dog finds a ball

(b) De hond vindt een bal
    The dog finds a ball

This asymmetry is not straightforward for the Minimalist Program. Word order differences between languages are dealt with by the strong/weak parameter, but construction-specific word order differences must be explained in another way. Zwart [Zwa97] provides such an explanation.

The most striking changes that Zwart introduces in relation to Chomsky’s 1993 version (of the Minimalist Program as described in Chapter 1) mainly concern feature checking and movement.

In Chomsky’s 1993 version of the Minimalist Program, LF was reached when all lexical constituents of a given sentence had checked all their formal features (such as case and tense). In Chomsky’s 1995 version and in Zwart’s framework functional constituents have to attract lexical constituents. Hence, the focus at LF switches from lexical to functional heads. A tree is an LF-representation when all functional constituents it contains have checked all their features.5

Chomsky [Cho95, Page 230] considers lexical items to be bundles of three

---

5This switch of focus is mainly made for reasons having to do with the principle of Greed. This principle says that elements may only move in order to satisfy their own feature checking requirements. If we change the focus from lexical elements to functional elements the principle changes accordingly: Elements may only attract other elements to satisfy their own feature checking requirements. Hence, not the moving node but instead the attracting node determines when movement is needed. See [Cho95, Page 294f] and [Zwa97, Page 184] for details.
different types of features: formal (or syntactic or inflectional) features, semantic features and phonological features. In Chomsky’s 1993 version as described in Chapter 1 this distinction was not (explicitly) made. The formal features are the only features that have to be checked. Therefore, it would be most natural from a minimalist point of view that only formal features move. However, Chomsky [Cho95, Page 262] assumes that isolated formal features are uninterpretable at PF. For instance, PF cannot interpret an isolated Wh-feature of an interrogative word. It needs the whole constituent (e.g. whose car), and this constituent contains more than just a Wh-feature. Therefore semantic and phonological features must move along with the formal features.

Zwart applies an approach where phonological features are added to lexical items only after the syntactic derivation has been completed. This approach is called *postlexicalism* (introduced as *Distributed Morphology* by Halle and Marantz [HM93]). Before the derivation, a universal lexicon is consulted with only semantic and syntactic information about lexical items. After the derivation, more specifically at PF, a language-specific lexicon is consulted to find the phonological features of a given lexical item on the basis of its semantic and formal features. A consequence of the postlexicalist approach is that features cannot be deleted after they are checked since the syntactic information they contain, for instance that a given lexical item is first person singular, is needed at PF to retrieve the corresponding lexical item in the language-specific lexicon (see also Chapter 6). Note that the strong/weak parameter still determines the moment of Spell-Out and so the PF positions of lexical constituents.

The idea that feature checking always implies deletion of features was introduced by Chomsky [Cho93] (see Chapter 1). Chomsky assumed that formal features cannot be interpreted at LF and therefore these features had to be deleted in the course of the derivation. However, in the description of the 1995 version of the Minimalist Program [Cho95, Page 277ff], Chomsky concludes that some formal features, namely the agreement and Wh-features of the noun and categorial features in general, are interpreted at LF. Hence, not all formal features are considered to be uninterpretable at LF and therefore not all formal features need to be deleted when checked.

Since the postlexicalist approach Zwart proposes is irreconcilable with the deletion of any formal features in the course of the derivation, Zwart [Zwa97, Page 186] is forced to modify the notion of interpretability. He proposes that a functional constituent XP is interpretable if its label contains only formal features that are checked by the lexical constituents that

---

6This fact implies that the strong/weak parameter is not represented in the lexicon. The lexicon used during the derivation is universal and therefore it cannot be used to express word order differences between languages. In Chapter 9 we will see that I formulated this parameter separately.
moved to XP for feature checking. Also overt movement (i.e. movement before Spell-Out) is driven by this 'proper pairing condition' according to Zwart [Zwa97, Page188], but, of course, the moment of Spell-Out is still determined by the weak/strong parameter [Zwa97, Page 187].

Zwart introduces the notion 'LC-features' (lexical-categorial features). The notion 'LC-features' covers semantic and categorial features (such as noun and verb). Zwart claims that LC-features only have to move along with the formal features in their movements if this is required by PF. As we saw above, isolated formal features are uninterpretable at PF.

LC-features of a given lexical constituent do not have to follow the moving formal features, provided that the PF-position of this lexical constituent already contains LC-features of another lexical constituent. If movement of LC-features is required this is called Last Resort movement. The Examples 3.3 and 3.4 respectively show a sentence where Last Resort movement is not required and a sentence where Last Resort movement is required.

Example 3.3 shows a tree where the LC-features of the final verb stay in its position of lexical insertion. The highest position of its chain with strong features (the head of CP), i.e. the PF-position, contains the LC-features of the lexical item dat (that). Therefore PF does not require the LC-features of the verb to move along with its formal features.

**Example 3.3**
Zwart assumes that subject-initial main clauses do not necessarily contain a CP (see Example 3.4). The head of AgrSP becomes the PF-position of the verb, because this is the highest position in its chain with strong features. As the head of AgrS does not contain any LC-features of another lexical item the LC-features have to move along with the formal features. Therefore the final verb is spelled out in the head of AgrSP in main clauses.

Example 3.4

Summarizing, Zwart introduces an economical kind of movement where phonological features are never involved in movement, while LC-features are only involved in movement as a Last Resort. Zwart’s approach enables us to derive a different word order for subject-initial main clauses and subordinate clauses by introducing Last Resort movement of LC-features for sentences where lexical constituents at PF occur without LC-features. Last Resort movement is necessary because isolated formal features are uninterpretable at PF.

3.4 The formalization

The formalization described in the Chapters 4 through 9 consists of two separate parts, which have a great deal in common. The first part formalizes the ideas of Chomsky’s 1993 framework [Cho93] as described in Chapter 1. As was mentioned before, this framework does not properly describe verb movement in Germanic languages such as Dutch and German. Therefore I decided to make a formalization of Zwart’s framework [Zwa97], as described
in the previous section, which does give a proper description of verb movement in Dutch. In the following chapters I only give separate descriptions of the two frameworks if there are significant differences. In principle only the formalization of Zwart's framework is discussed.

The formalization described here is not a procedural representation of what Chomsky calls 'the computational system'. I consider the computational system to be a declarative system which represents linguistic competence, that is, a grammar. On this declarative or representational system one could base procedural systems such as parsers and generators.

The formalization described here has a lot in common with Brody's Lexico-Logical Form [Bro95]. Brody's framework is non-derivational. He argues that the Minimalist Program is redundant in having both movement derivations and chains representing movements and provides evidence against movement derivations. A radically minimalist framework without derivations is sketched, where LF is the only level of representation. Hence, LF is the only syntactic level where conditions on representations, such as the principle of Full Interpretation, can hold. According to Brody, the LF-to-PF mapping is not syntactic but purely phonological. Hence, Brody maintains PF as an interface level but LF is the only interface level which is considered to be a level of representation. In Brody's framework lexical input is not related with the interface levels through a derivation: the lexicon and the semantic interpretation rules can access the same interface which is called Lexico-Logical Form (LLF).

Brody's ideas relate to the formalization in the following way. The formalization presented here defines conditions on LF-trees. The system can judge whether a tree is a correct LF-tree or not. LF-trees can be considered to be the trees produced in the final stage of the derivation, where all features that need checking are checked. In the LF-tree all movements of lexical constituents (from their position of lexical insertion to their LF-positions) are visualized by chains. The PF-representation, which is a list of words representing a sentence, can be derived from a correct LF-tree by finding the highest position with strong features in a chain.

Cornell [Cor97, Page 26] defines PF in the same way and notes that the definition of PF only can be put representationally, not derivationally since it is based on completed chains. To determine the PF-position of a chain

---

7For representational versions of Government and Binding Theory see [Kos87] and [Riz90].
8Note that Chomsky [Cho93] considers LF and PF to be the only levels of representation.
9Cornell also notes that to define the PF-position as the upper strong position of the chain is not completely correct. Chomsky forbids the occurrence of strong features in the covert component. Cornell suggests to add the requirement that a chain should consist of a 'continuously strong prefix' followed by a continuously weak 'suffix', although he does not apply this idea.
3.4. THE FORMALIZATION

the whole chain needs to be finished. A strong position is not automatically
the PF-position of a chain since a chain can contain more than one strong
position, and there is only one PF-position per chain. Hence, the PF-
position of a chain cannot be determined in the course of a derivation. Also
within a derivational approach to the Minimalist Program the PF-position
of chains must be determined representationally.

Cornell [Cor97] argues that both derivational and representational ap-
proaches to the Minimalist Program have their value for the development of
the field: derivational minimalism can provide the proof theory whereas rep-
resentational minimalism can provide the model theory. Hence, according
to Cornell, the merit of a representational approach can be the elucidation
of syntactic structures. Derivational approaches do not do this: they
only produce a final LF-PF pair. Note that Cornell’s system is based on a
derivational system developed by Stabler [Sta96].

The objective of the formalization project described here was to spec-
ify the basic ideas of the Minimalist Program. Only a few basic types of
sentences in Dutch are dealt with:

- declarative main clauses: both transitive and intransitive (e.g. Wim
  believes Tony respectively Wim speaks.)
- subordinate clauses (e.g. that Wim believes Tony)
- Wh-questions: questions starting with an interrogative word (e.g.
  Who believes Tony?)
- yes/no-questions: questions which can be answered with either ‘yes’
or ‘no’ (e.g. Does Wim believe Tony?)

The choice to represent linguistic competence as a declarative system
resulted in the fact that the operations Merge and Move have not been
formalized. The system judges completed derivation trees and does not
have any knowledge about how to construct a derivation tree.

During the formalization process the domain has been divided into six
modules: trees, feature structures, the lexicon, X-Theory, movement and
interfaces. All the modules are discussed in detail in separate chapters.\(^\text{10}\)
The content of the modules is briefly summarized in this section.

- Chapter 4: the module about trees contains basic knowledge about
derivation trees: what is the number of mothers and daughters a node
can have, which functions can apply to nodes etc.

- Chapter 5: the feature structure module outlines which feature names,
  feature values and feature structures occur in the Minimalist Program.

\(^{10}\)See also [VS96] for a description of an earlier version of the formalization.
Furthermore, operations on feature structures such as feature checking are defined.

- Chapter 6: what is the character and the format of the lexicon and how is it integrated in the rest of the theory?
- Chapter 7: the module about \( \mathcal{X} \)-Theory not only specifies \( \mathcal{X} \)-rules but, for instance, also the position of specifiers, heads and adjuncts within a projection (XP).
- Chapter 8: the chain module describes the chains that are allowed by the Minimalist Program. As we saw before movement is connected with feature checking in the Minimalist Program. Therefore features play an important role in this module.
- Chapter 9: the interface module specifies a function to determine whether a tree is a correct representation at the interface level LF. Furthermore, a function is provided to deduce the PF-representation of a sentence from an LF-tree. A PF-representation contains not only phonological information about all the lexical items that occur in a given sentence; it also represents all the lexical items in the order in which we read or hear them.

The formalization described here is validated by an implementation in Prolog. The implementation will not be discussed in the following chapters since the translation from AFSL to Prolog was mostly very straightforward. Furthermore I assume that the formalization in AFSL will be easier to grasp for readers without a background in programming.