Chapter 2

Implementations

In this chapter I will present two small implementations which I used to explore the possibilities of the formalization of the Minimalist Program.

In Section 2.1 I will give a survey of structure-building operations within the minimalist framework [Cho93]. The survey is based on an implementation of the operations Move and Merge in Prolog. The implementation, which is described in Subsection 2.1.1, shows in detail which operations are needed to build minimalist trees. It reveals that some sub-cases of Merge and Move are not (sufficiently) described in the literature. In Subsection 2.1.2 I will give a schematic overview of what we learned from the implementation and in Subsection 2.1.3 I will show that the general ideas in the literature have to be reconsidered to be able to capture all structure-building operations. This does not mean that the ideas are wrong, but it does mean that they are incomplete.

In Section 2.2 I will describe a head-corner parser for a fragment of the Minimalist Program [Cho93]. I will argue that, because of the nature of the structure-building operations of the Minimalist Program, head-corner parsing is a suitable parsing technique for the Minimalist Program.

Furthermore, I will put forward a proposal to treat functional and lexical heads differently. Functional heads are not treated as head-corners of their mothers by the minimalist head-corner parser that is described here.

The parser that is described here only covers simple declarative sentences, possibly with a subordinate clause.

\[\text{\footnotesize 1This section is based on [Vee94].}\]
\[\text{\footnotesize 2This section is based on [Vee95a] and [Vee95b].}\]
2.1 Merge and Move

2.1.1 The implementation

In the implementation that is discussed here, I endeavoured to cover all structure-building operations that are needed to build a tree according to the Minimalist Program. The only structure-building operation that is not covered is the adjunction operation with two maximal projections of which the result is given in Example 2.1. The reason for the absence of this operation is the fact that it is not needed in the fragment that is covered by this chapter or later chapters of this work. Furthermore, this type of adjunction is considered to be impossible by many linguists (see for instance Kayne [Kay94]).

Example 2.1

The implementation covers three main operations: one concerning heads, one concerning complements and one concerning specifiers. In Example 2.2 I show the positions that heads, complements and specifiers occupy within a maximal projection XP. The three operations each consist of one or more sub-cases. The number of sub-cases corresponds to the number of possible types of fillers for the relevant position. In this way, the implementation gives a clear view of all structure-building operations that are needed to build minimalist trees. All the operations and their sub-cases will be discussed below.

Example 2.2

The operation concerning head positions consists of two sub-cases. One sub-case is a Merge operation that is used to create heads by extracting feature structures from the lexicon. I will call this \textit{lexical insertion}. The head is taken as the target. X is projected to \(X\) and the sister of X is
filled with a complement which itself is also built by the structure-building operations Merge and/or Move. The other sub-case is a Move operation that serves to create an adjunction structure. The structure that arises when the head or adjunction structure \( Y \) adjoins to the head \( X \) is given in Example 2.3. \( Y \) originates from a position within \( ZP \). This kind of movement is called head movement.

**Example 2.3**

```
  X
 / \   
X   ZP
 / \  
Y   X
```

**Example 2.4**

```
CP
|
|-- C
|  `- AgrSP
    |    `-- that
    |        `-- NP
    |            `-- Bill
    |                `-- Agr
    |                   `-- AgrS
    |                       `-- T
    |                           `-- AgrO
    |                               `-- V
    |                                   `-- AgrO
    |                                       `-- meets
    |                                           `-- wi
```

The operation for complement positions consists of only one sub-case. It is a Merge operation which fills complement positions with maximal projections (XPs). For instance, in Example 2.4, the complement NP \( \text{NP} \) has
been merged into its position as a sister to $V$. Movement to complement position does not exist.\textsuperscript{3} I suggest to call all types of insertion in complement position \textit{tree insertion}.

The operation concerning specifier positions consists of two sub-cases. The first sub-case is a \textit{Move} operation. I suggest calling this type of movement \textit{to-specifier movement}. The element moved originates from a specifier or complement position lower in the tree. For instance, in Example 2.4, the NP (with the index $i$) in the specifier position of the VP, is moved to the specifier position of TP for feature checking. An example of to-specifier movement originating from complement position is the movement of the object NP with the index $j$, which is located in the complement position of the VP. This NP has to move to the specifier position of AgrOP to check certain features.

The second sub-case of the operation concerning specifiers does not involve movement. It is a \textit{Merge} operation that combines a target (X) and a maximal projection (YP). The maximal projection is the specifier.Specifier positions can only be filled by NPs. In Example 2.4 this sub-case only occurs once: $V$ is merged with an NP, which is represented by an empty position with the index $i$ since it overtly moved to the specifier of TP. Complement positions are less sensitive concerning filler trees.\textsuperscript{4} For instance, AgrSP occurs as the complement of C, VP occurs as the complement of AgrO etc. Tree insertion in the sense of the insertion of an NP in specifier position can only occur in the lexical domain because that is the only place where we can insert new lexical material.\textsuperscript{5} A third sub-case for specifier positions would have been needed for the insertion of determiners in the specifier position of an NP (see Example 2.5), but we will see that with a slight modification we can eliminate this sub-case.

Example 2.5

\begin{center}
\begin{tikzpicture}
    \node {NP} child {node {Det} child {node[fill=white] {the} child {node {N} child {node {book}}}}};
\end{tikzpicture}
\end{center}

Until now, the only lexical insertion was the insertion of a head from the lexicon. A category that is spoiling the assumption that lexical insertion

\textsuperscript{3}Cf. [Cho93, Page 23].

\textsuperscript{4}See also [Hoe91, Page 27].

\textsuperscript{5}An exception to this rule may be the insertion of adverbs [cf. [Zwa93, Page 94]] and complementizers [see the head of the CP in Example 2.4].
is head insertion is the determiner. In my implementation I first chose to use NPs instead of DPs (see Example 2.6). This was done for reasons of simplicity. Only NPs of the type name, personal pronoun, noun and noun plus determiner are included. The NP analysis seemed sufficient for this goal. In the end this appeared to be an inefficient solution because the determiner in an NP analysis is not a head. In the DP analysis on the other hand, the determiner is a head (D) which takes an NP as its complement. In short we could say that it is more efficient to adopt the DP analysis because in that case we only have tree insertion (no lexical insertion) in specifier position and there is only one type of lexical insertion and that is the insertion of a head from the lexicon. The implementation was adjusted to the use of DPs instead of NPs. Therefore the third sub-case of the case concerning specifiers was no longer needed.

Example 2.6

\[
\text{DP} \quad \text{D} \\
\quad \text{NP} \\
\text{the book}
\]

2.1.2 Overview

All operations that are discussed in Subsection 2.1.1 except for the operation connected with the NP analysis are summarized in Table 2.1.

Table 2.1

<table>
<thead>
<tr>
<th>Structure-building operations of the Minimalist Program</th>
<th>tree insertion</th>
<th>lexical insertion</th>
<th>to-specifier movement</th>
<th>head movement</th>
</tr>
</thead>
<tbody>
<tr>
<td>Merge</td>
<td>X</td>
<td>X</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>Move</td>
<td>-</td>
<td>-</td>
<td>X</td>
<td>X</td>
</tr>
<tr>
<td>head</td>
<td>-</td>
<td>X</td>
<td>-</td>
<td>X</td>
</tr>
<tr>
<td>complement</td>
<td>X</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>specifier</td>
<td>X</td>
<td>-</td>
<td>X</td>
<td>-</td>
</tr>
<tr>
<td>from head</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>X</td>
</tr>
<tr>
<td>from complement</td>
<td>-</td>
<td>-</td>
<td>X</td>
<td>-</td>
</tr>
<tr>
<td>from specifier</td>
<td>-</td>
<td>-</td>
<td>X</td>
<td>-</td>
</tr>
</tbody>
</table>

\(^6\)See [Abn87] for a description of the determiner phrase.
In the first two rows of the table, we see that tree insertion and lexical insertion are associated with the operation Merge, while to-specifier movement and head movement, of course, are associated with the operation Move.

The number of crosses in the rows head, complement and specifier in the table corresponds to the total number of sub-cases. There are two operations to create heads (lexical insertion and head movement), two operations to create specifiers (tree insertion and to-specifier movement), and there is one operation to create complements (tree insertion).

The only operations that have crosses in the last three rows are the movement operations in the third and fourth column. This follows because movement operations are the only operations that take elements from other positions to specifier or head positions.

2.1.3 Problems

During the implementation of Move and Merge it turned out that the descriptions in Section 1.6 do not cover all the operations that we need to build the trees that the minimalist framework requires. In this section we saw that Chomsky [Cho93, Page 23] argues that the moved tree must be contained in the target tree. Here I will argue why this is not always the case.

Head movement, which is a Move operation, yields an adjunction structure. When a head is moved, it is adjoined to another head that is higher in the tree. It receives a position that is created by an adjunction operation. For example, when a verb from within the VP is moved to AgrO to check its features, we get the structure given in Example 2.7.

Example 2.7

If we have a closer look at head movement, it does not seem to fit into the definition of the operation Move. Chomsky only illustrates the Move operation with an example of XP-movement. In the case of XP-movement (= to-specifier movement) the target is a $\mathbf{Y}$ (see Example 2.8). $\mathbf{Y}$ has two daughters: Y and ZP. The target tree, $\mathbf{Y}$, contains the element XP that is moved to the specifier position of YP.
Example 2.8

In the case of head movement, the moved element is not contained by the target tree, but by the complement of the target phrase marker. For instance, in Example 2.7, the sister of V (the lower AgrO) is the target tree. The higher AgrO is the projection of the target. Example 2.7 shows that the target AgrO is a terminal element. Hence, it is impossible that the moved V comes from the target. Instead, the moved element comes from the VP-complement. Therefore, I need to redefine ‘Move operation’ as in Definition 2.1.

Definition 2.1

A Move operation is a structure-building operation in which a moved element is combined with a target tree to form a new tree. The element that is moved has to be contained in the complement domain of the head of the target tree.

If we adopt this definition both to-specifier movement and head movement are covered.

In Subsection 2.1.1 I refer to Chomsky [Cho93] for the reason why movement to complement positions is not possible. Now that we have adopted the new definition for the notion ‘Move operation’, I can come up with a very clear reason why movement to complement positions is impossible. The new definition says that moved elements must originate from the complement domain. From this definition we can derive that it is impossible to move an element to a position that is filled with the tree from which the element should originate. Therefore, movement to complement positions is excluded.

2.1.4 Summary

In this section I discussed an implementation of the ideas about structure-building operations in the Minimalist Program [Cho93]. The implementa-

---

5 See Section 1.7 for a detailed description of head movement.
6 For the definition of ‘complement domain’ see [Cho93, Page 11] and Chapter 8. At this point we simply will assume that the complement domain is the complement.
tion covers both Move and Merge operations. It is concluded that there are three kinds of Merge operations (that is: tree insertion in the complement and the specifier position and lexical insertion in the head position) and two kinds of Move operations (that is: head movement and to-specifier movement).

I demonstrated that it is possible to implement the structure-building operations as described by Chomsky when some slight modifications are made. A new definition of the Move operation is given, which says that the moved element in a Move operation has to be contained in the complement domain of the head of the target tree, not in the target tree as the original definition says. From this new definition we can derive that movement to the complement position is impossible. An element (tree) cannot be moved to a position that contains the tree from which the moved element must originate.

2.2 A minimalist head-corner parser

In this section I will discuss a head-corner parser which associates an input string with trees such as the one in Example 2.10, where each chain only contains one visible (i.e., nonempty) element. The visible element represents the Spell-Out position of the chain.

2.2.1 Head-corner parsing

The main idea behind head-driven parsing [Kay89] is that the lexical entries functioning as heads contain valuable information for the parsing process. For example, if a verb is intransitive it will not require a complement, while it will require a complement if it is a transitive verb. Therefore, the head is parsed before its sisters in a head-driven parser. A head-corner parser [Kay89] [BvN93] is a special type of head-driven parser. Its main characteristic is that it does not work from left to right but instead works bidirectionally. That is, first a potential head of a phrase is located and next the sisters of the head are parsed. The head can be in any position in the string and the non-head sisters can either be to the right or to the left.

Top-down and bottom-up information is combined in head-corner parsers. The top-down information is contained in the head-corner relation. The head-corner relation is the reflexive and transitive closure of the head relation. An example of two categories that stand in a head relation are $\overline{N}$ and NP, i.e. $\overline{N}$ is the head of NP. X-Theory contains the following general rule (Example 2.9):

---

9Note that each chain has its own index.
Example 2.9

XP $\rightarrow$ YP $\overline{X}$

The head of a rule is one of its right-hand side daughters (in this case $\overline{X}$). Given the fact that the Minimalist Program assumes $\overline{X}$-Theory, it is most natural to assume that $\overline{X}$ is the head of NP and $X$ is the head of $\overline{X}$. The head-corner relation is transitive and therefore both $N$ and $\overline{N}$ are head-corners of NP.

A head-corner parser starts the parsing process with a prediction step. This step is completed when a lexical head is found that is head-corner of the goal according to the head-corner table. The goal is the type of constituent that is parsed, i.e. the root of the tree that is built. Then an $\overline{X}$-rule is selected in which the lexical head is in the right-hand side. The sisters of the head are parsed recursively. In the minimalist head-corner parser that I am proposing here, a head always has only one sister because minimalist trees are at most binary branching. The left-hand side of the rule is consulted to find the mother of the head. Then the head-corner table is used to decide whether this mother is a head-corner of the goal. If this is the case the whole process is repeated by selecting a rule with the new head-corner (i.e. the mother of the first head-corner) in its right-hand side. The sisters are parsed recursively, etcetera.

In Chapter 1 it was claimed that movement is invariably leftward and that Move and Merge are bottom-up operations. The VP is built before other projections. Constituents of VP are moved to higher projections by the operation Move. Suppose that the parser should consider AgrS as the head-corner of AgrSP, which would accord with $\overline{X}$-Theory (see Example 2.10). Then the head (AgrS), which should be filled with an adjoined verb by movement from T, is created before T, AgrO and V. To avoid these purely top-down steps the head-corner table for the minimalist head-corner parser is not constructed completely according to $\overline{X}$-Theory. For instance, instead of AgrO, VP is the head-corner of AgrO. This processing solution is faithful to the ideas of the Minimalist Program in the sense that in this way the tree is built up in an absolutely bottom-up way (i.e. starting from V) so that a position that should be filled by movement is always created after the position from which the moved element comes. Hence, the parser is not looking for the grammatical head of functional projections but instead it looks for the lexical basis. When parsing a sentence, the parsing process always starts with the verb and not with one of the functional heads. The Minimalist Program postulates functional heads which have no phonological content. Lexical heads like verbs are more useful for the parsing process, as we already saw at the beginning of Subsection 2.2.1.

The list in Example 2.11 illustrates that functional heads like AgrO
and Agr$S$ are not considered as head-corners. Lexical projections like VP and NP are treated according to $X$-Theory. If we consider the head-corner table in Example 2.11 in combination with the tree in Example 2.10 we establish the fact that the parser searches its way down to the verb as soon as possible. The top-down prediction step moves from the goal Agr$SP$ to Agr$S$ to Agr$OP$ to Agr$O$ to VP to $V$ and finally to the lexical head-corner $V$ where the bottom-up process starts as the Minimalist Program requires.

**Example 2.10**

![Diagram](image)

**Example 2.11**

Elements of the head-corner relation:

\[
\begin{align*}
hc(\text{Agr}\bar{S}, \text{Agr}SP). & \quad hc(V, \text{Agr}O). \\
hc(\text{TP}, \text{Agr}\bar{S}). & \quad hc(V, VP). \\
hc(T, \text{TP}). & \quad hc(V, \bar{V}). \\
hc(\text{Agr}OP, T). & \quad hc(\bar{N}, NP). \\
hc(\text{Agr}O, \text{Agr}OP). & \quad hc(N, \bar{N}).
\end{align*}
\]

### 2.2.2 Structure-building operations and head-corner parsing

If we have a closer look at the definitions of Merge and Move we see that they resemble the strategy of head-corner parsing a lot. In both cases we start with a head. In the case of Merge and Move this is called the target tree; in the case of head-corner parsing it is called the (lexical) head-corner.
In both cases we use an X-rule to obtain more information about the mother and the sister of the head. In the definitions of Merge and Move the sister also has to be built up by Merge and/or Move or it is an XP that is moved from a position lower in the tree to this position. In the parser that is discussed here a sister is parsed or the sister is linked to a lower position (see Subsection 2.2.3).

The next step for Merge and Move is to consider the new phrase marker that is built as the new target tree and apply Merge or Move to this tree. The next step in the parsing process is to check if the mother of the head-corner is a head-corner of the goal. If this holds, the whole process starts again with the mother as a head-corner.

2.2.3 The algorithm

The parser that is discussed here is based on the head-corner parser in [BvN93]. In a head-corner parser, the positions the words of an input string occupy in relation to each other is essential. Each word is assigned two numbers which indicate its position in the sentence. For instance, if a sentence consists of three words the first word is assigned the numbers 0 and 1, the second word is assigned the numbers 1 and 2 and the third word is assigned the numbers 2 and 3. Hence, the third word of the sentence is the word that is located between the second and the third position of the sentence.

The position numbers play an essential part in the parsing process. If we are looking for a head-corner within a sentence from position 0 to position 8, the position numbers of the head-corner may never be lower than 0 or higher than 8. Suppose we find a head-corner from position 5 to position 6. Then we know that if the head-corner has a sister it must have 5 as its rightmost position if it is a left sister, or 6 as its leftmost position if it is a right sister, since sisters, of course, are adjacent.

The minimalist head-corner parser presented here mainly consists of the following three predicates: parse, head-corner and predict. The parsing process starts with calling parse. As we see below parse calls among other things the predict and the head-corner predicate.

In the predicate parse below, E0 indicates the first position of the sentence that is parsed and E indicates the last position of the sentence that is parsed. P0 and P indicate the first and last position, respectively, of the specific constituent that is parsed at a given moment. The first time the predicate parse is called, P0 equals E0 and P equals E, since the constituent (Cat) we are looking for is the category of a complete sentence (for instance, AgrOP). In later calls, E0 and E always indicate the first and last position of the complete sentence but P0 and P can for instance relate to an NP that is parsed at that moment.
As we saw earlier, the idea behind head-corner parsing is to find a head-corner and to parse its sisters recursively. Detecting a new head-corner is the main task of the parse predicate.

\[
\text{parse(Cat/CatTree,PO,P,EO,E) if there is a Cat from position PO to P}
\]

\[
\text{% in the sentence, within the range EO, E}
\]

\[
\text{parse(Goal,PO,P,EO,E) :-}
\]

\[
\text{predict(Goal,Lex,Q0,Q),}
\]

\[
\text{between(Q0,Q,EO,E), % Q0 and Q are between EO and E}
\]

\[
\text{head_corner(Lex,Goal,Q0,Q,PO,P,EO,E).}
\]

The predicate `predict` locates a lexical head-corner. The relation `hc` implements the head-corner table.

\[
\text{predict(Goal/GoalTree,Lex/LexTree,Q0,Q)}
\]

\[
\text{% if Lex from position Q0 to Q may be head-corner of Goal}
\]

\[
\text{predict(Goal/_,Lex/tree(Lex,w(Word)),Q0,Q) :-}
\]

\[
\text{hc(Lex,Goal),}
\]

\[
\text{chart(Word,Q0,Q),}
\]

\[
\text{lex(Word,Lex).}
\]

Because trees in the Minimalist Program are at most binary branching, there will never be both left and right daughters in the same rule (in addition to the head). Furthermore, there is always at most one right or left daughter in each rule. It is impossible to parse both the left and the right daughters within the same `head_corner` clause. We need separate clauses to parse left and right daughters (see respectively the third and the second `head_corner` clause given below). For example, the second `head_corner` clause parses a sister of the head-corner which is to the right of the head-corner. The position of the head-corner is from Q0 to Q1. The position of the sister is from Q1 to Q, with Q <= E and Q0 <= Q1 <= Q (see the line `parse(Dtr/DtrTree,Q1,Q,Q1,E)` in the second `head_corner` clause below and Example 2.12). If Q0 = Q1 the head is empty (e.g. because it moved to another position in the tree).

**Example 2.12**
The predicate rule implements X-rules. In these rules Small represents the head-corner, Dtr represents the sister of the head-corner and Mid represents their mother. The first argument of the predicate rule indicates at which side of the head-corner the other daughter of Mid can be found. The third argument is the left-hand side of the X-rule. The second and the fourth argument represent the right-hand side of the X-rule.

head_corner(Small, Small, P0, P, P, E).

head_corner(Small/SmallTree, Goal/GoalTree, Q0, Q1, P0, P, E, E) :-
  rule(right/Small/SmallTree, Mid/MidTree, Dtr/DtrTree),
  parse(Dtr/DtrTree, Q1, Q1, E),
  hc(Mid, Goal),
  head_corner(Mid/MidTree, Goal/GoalTree, Q0, Q, P0, P, E, E).

head_corner(Small/SmallTree, Goal/GoalTree, Q1, Q, P0, P, E, E) :-
  rule(left/Small/SmallTree, Mid/MidTree, Dtr/DtrTree),
  parse(Dtr/DtrTree, Q0, Q1, E, Q1),
  hc(Mid, Goal),
  head_corner(Mid/MidTree, Goal/GoalTree, Q0, Q, P0, P, E, E).

In addition, we need separate clauses for Merge and Move. As we concluded in Subsection 2.2.2, Merge has a lot in common with head-corner parsing. Therefore the plain head_corner clauses as given above represent Merge. To account for movement I added movement predicates after the call to parse in a head_corner clause. The example given below is the clause that describes movement to specifier positions.

head_corner(Small/SmallTree, Goal/GoalTree, Q1, Q, P0, P, E, E) :-
  rule(left/Small/SmallTree, Mid/MidTree, Dtr/DtrTree),
  parse(Dtr/DtrTree, Q0, Q1, E, Q1),
  toSpecifier_movement(MidTree, SubTree, DtrTree)
  hc(Mid, Goal),
  head_corner(Mid/MidTree, Goal/GoalTree, Q0, Q, P0, P, E, E).

DtrTree is the constituent that is moved to a specifier position. The root of MidTree gets the moved constituent as its left daughter. SubTree contains the position where the moved constituent comes from. To check if it is possible to move the constituent from SubTree to MidTree a simple solution is chosen. Within the predicate toSpecifier_movement, a table with possible movements is consulted: move(specifier:X, specifier:Y), or move(complement:X, specifier:Y).\footnote{See also Chapter 8.} X and Y represent respectively the category of the root of SubTree and the category of the root of MidTree. If a possible movement from SubTree to MidTree exists, the features and the chain indexes of the starting and final position of the moved constituent are
unified. Head movement is treated in a way similar to movement to specifier positions.

The fact that functional heads are not head-corners causes the necessity of an unusual head-corner clause. The clause below is needed to be able to consider the linguistic head as a daughter and the complement as a head-corner (compare the second through fourth argument of rule in the clause below with the same arguments within the head-corner clauses above). The following clause uses a rightward rule to find a daughter to the left of the head-corner. The type of rule that is used is comparable with the head-corner clause above that is used to parse right daughters, but the indexes for the sentence positions in this clause are the same as the indexes in the head-corner clause above that is used to parse left daughters. The predicate functional ensures that Dtr is a functional head. In the head-corner clause for movement to specifier positions, Mid should also be functional. Here it is not necessary to add the predicate functional because all possible movements are movements to functional projections. Therefore it would be redundant to prove that Mid is functional.

head-corner(Small/SmallTree, Goal/GoalTree, Q1,Q,P0,P,E0,E) :-
    rule(right,Dtr/DtrTree, Mid/MidTree, Small/SmallTree),
    functional(Dtr),
    parse(Dtr/DtrTree, Q0, Q, E0, Q1),
    hc(Mid, Goal),
    head-corner(Mid/MidTree, Goal/GoalTree, Q0, Q, P0, P, E0, E).

An example of the application of the head-corner clause that is given above is the case in which Small is a VP. A rule that could apply in this case is the rule where Dtr is AgrO and Mid is AgrO. AgrO is the regular head of the rule, but VP is the head-corner in our parser. Therefore the rightward rule can be applied to find a daughter that is to the left of the head-corner.

2.2.4 Parsing versus generation

At the end of Subsection 2.2.1 I chose not to consider functional heads as head-corners of their mothers. This choice was made because the structure-building operation Merge starts with constructing a VP before the projections to which constituents from VP are moved are constructed. Another motivation to start with VP is that V contains information that is useful for the rest of the structure-building process. For instance, if the verb is intransitive we know that V does not require a complement sister, and we know that we do not need an AgrOP on top of VP to check the features of the object. The fact that V contains lexical information and functional heads like AgrO and AgrS do not, could be used as a justification for the fact that
2.2. A MINIMALIST HEAD-CORNER PARSER

the latter are not head-corners. The main idea of head-driven parsing is, as was stated before, that heads contain relevant information for the parsing process, and that they consequently should be parsed before their sisters. In the Minimalist Program not all heads are lexical. Functional projections do not have phonological content. They obtain their phonological content via movement of elements from positions lower in the tree. This special status of functional heads makes them less useful in the parsing process.

The Minimalist Program is a generative framework. Grammatical descriptions are cast in terms of rules to generate structural descriptions. Because we are dealing with parsing (as opposed to generation) here, there are certain further mechanisms postulated by the parser, ones not foreseen in the the purely linguistic framework. In the minimalist framework, lexical information belonging to a chain is available from the moment that the first position of the chain is created, because that is the moment when the lexicon is consulted. Lexical elements enter the tree at the bottom of their chain and the lexical information that they bring can be used during the whole length of the tree-building process. For example, a verb enters the chain in the head position of VP. Therefore, the lexical information belonging to the verb can be utilized to determine whether the verb needs a complement or not. Later on in the process, the lexical information can guide the decision whether an AgrOP should be build. An AgrOP is namely only needed in a transitive sentence.

When parsing a sentence the lexicon is not by definition consulted at the beginning of the chain. Example 2.10, repeated here as 2.13, shows a tree that contains traces and visible constituents. The position containing a visible constituent is the Spell-Out position of that chain. The parser consults the lexicon at the moment in which the Spell-Out position of a chain is reached. Consequently, when a trace is created before Spell-Out, the features belonging to that trace are unknown. Because all positions in a chain are linked, the features of all traces of a chain are known as soon as the Spell-Out position is reached. For instance, in Example 2.13 I assume that the PF-position of the verb is where it is adjoined to AgrS. In that case there is a trace in V, and at the moment in which it should be determined whether the verb needs a complement or not, the features of V are still unknown. Therefore the parser will have to backtrack to try different possibilities.

It can be concluded that the absolute bottom-up approach for the building of trees is more useful for generation than for parsing. In generation, lexical information can be used as soon as a position that is the beginning of a chain is created. In parsing we will have to wait until the Spell-Out position is reached. In spite of this, I chose not to consider functional heads as heads in order to specify an absolutely bottom-up process. This bottom-
up approach is preferred because in this way a position to which a certain constituent is moved is created after the position from which the constituent is moved. If we do not choose this approach, sometimes positions will be created which need a moved element from a subtree that does not exist yet. This could be inefficient, and it is not a direct implementation of the ideas of the minimalist framework.

Example 2.13

![Diagram of sentence structure]

2.2.5 Summary

It appeared to be possible to implement the ideas that are described here in a head-corner parser. A distinction is made between lexical and functional heads. Functional heads are not possible head-corners, while lexical heads are. In this way the parser is forced to build up trees in a bottom-up way, which makes that the parser can simulate the bottom-up structure-building operations Merge and Move in a faithful way.