c. An element cannot participate in more than one chain.
d. A chain is *complete* if and only if it is headed by a non-trace element. All chains must be complete.

The first two conditions allows both possibilities for movement discussed above in example (13). Condition (31c) ensures that two movement chains cannot share common heads or traces. Note that condition (31d) prevents the movement of traces. However, the same condition does not forbid a chain from being headed by an empty element as long as it is not a trace. For example, the chain conditions are compatible with the following analysis for the relative clause construction *the man I saw* (along the lines of Chomsky [12]):

(32) [*NP* [*NP* the man] [*CP* Op_i [$\bar{C}$ [*C*] [*IP* I saw [*NP*-t]$_i$]]]]

Here, the empty operator Op_i is assumed to head the chain $\langle Op_i,$ [*NP*-t]$\rangle$.

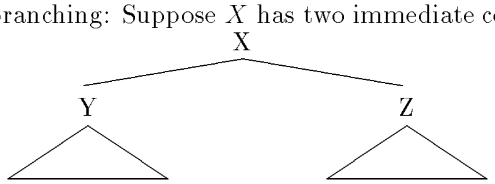
As with free indexation, the present system does not have the ability to automatically synthesize a constructive chain formation algorithm from the definitions given in (30) and (31). Hence, we must suply it with one. We now proceed to a description of an algorithm defined recursively over tree structures that builds chains in accordance with the chain conditions. For each constituent X in a structure, we will associate X with two sets: (1) the set of incomplete chains in X, to be denoted by C_X, and (2) the set of 'free' NPs in X, to be denoted by F_X, i.e. those NPs that do not participate in a non-trivial movement chain in X. There are two cases to consider. (For simplicity, we will omit for now the encoding of chain condition (31d), which only applies to the root node.

1. Base Case: X is a leaf node.
   (a) X is an empty noun phrase, i.e. [*NP*-e]. In accordance with chain condition (31a), there are two possibilities:
      i. [*NP*-e] may be free in X. Hence $C_X = \emptyset$ and $F_X = \{$ [*NP*-e]$\}$, or
      ii. [*NP*-e] may participate in a chain in X. Hence, $C_X = \{\langle$ [*NP*-e]$\rangle\}$ and $F_X = \emptyset$.
   (b) X is not an empty noun phrase. Then $C_X = \emptyset$ and $F_X = \emptyset$.

   In either case, the chain conditions will be satisfied.

2. Compositional Case: X is an interior node.
   (a) Unary branching: Suppose X has a single immediate constituent Y. Then X inherits the chains and free elements of Y: that is, $C_X = C_Y$ and $F_X = F_Y$. Since no additional elements are introduced by X, it is clear that if Y satisfies the chain conditions, then so will X.

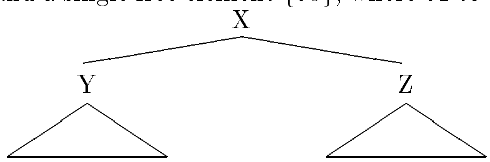
(b) Binary branching: Suppose X has two immediate constituents Y and Z:



$C_Y = \{c_1, \ldots, c_m\}$, $m \geq 0$ $\quad$ $C_Z = \{c_{m+1}, \ldots, c_p\}$, $p \geq m$

$F_Y = \{f_1, \ldots, f_n\}$, $n \geq 0$ $\quad$ $F_Z = \{f_{n+1}, \ldots, f_q\}$, $q \geq n$

The possible (partial) chains for X will simply be the union of the corresponding chains for Y and Z, possibly extended by the free elements of Y and Z. For example, consider the case when Y has a singleton chain $\langle e1 \rangle$ together with free elements $\{John, e4, e5\}$, and Z has a single chain $\langle e2, e3 \rangle$ and a single free element $\{e6\}$, where $e1$ to $e6$ are all empty NPs:



$C_Y = \{\langle e1 \rangle\}$ $\quad$ $C_Z = \{\langle e2, e3 \rangle\}$

$F_Y = \{John, e4, e5\}$ $\quad$ $F_Z = \{e6\}$

For example, one possibility would be to use $e6$ from Z to prefix the chain $\langle e1 \rangle$ from Y, thus obtaining the extended chain $\langle e6, e1 \rangle$. In general, the extensions, or prefixes, for either chain must be drawn from one of the 65 possible r-permutations of the available free elements shown in figure 3.18.[19] Here, each node corresponds to a distinct r-permutation of the free elements $John$, $e4$, $e5$ and $e6$. Of course, not every permutation need be a valid prefix. For example, $\langle e4, John \rangle$ is not a valid prefix because intermediate elements in a chain must be traces, as defined by (30). (Those r-permutations that are also valid prefixes have been marked in figure 3.18.)

We now proceed to define a non-deterministic operation `merge1` that extends chains using the possible r-permutations of the available free elements. Let `merge1` take as input a set of chains C and two sets of free elements F_1 and F_2. `merge1` will return C', the set of chains after extension, and F_1' and F_2', representing the 'unused' elements of F_1 and F_2, respectively:

(33)

[19] An r-permutation of n things is an ordered selection (without replacement) of r of them. Let P_n denote the sum of r-permutations for $r = 0, 1, \ldots, n$. In general, P_n will be the nearest integer to $n!e$ (Riordan [47]).

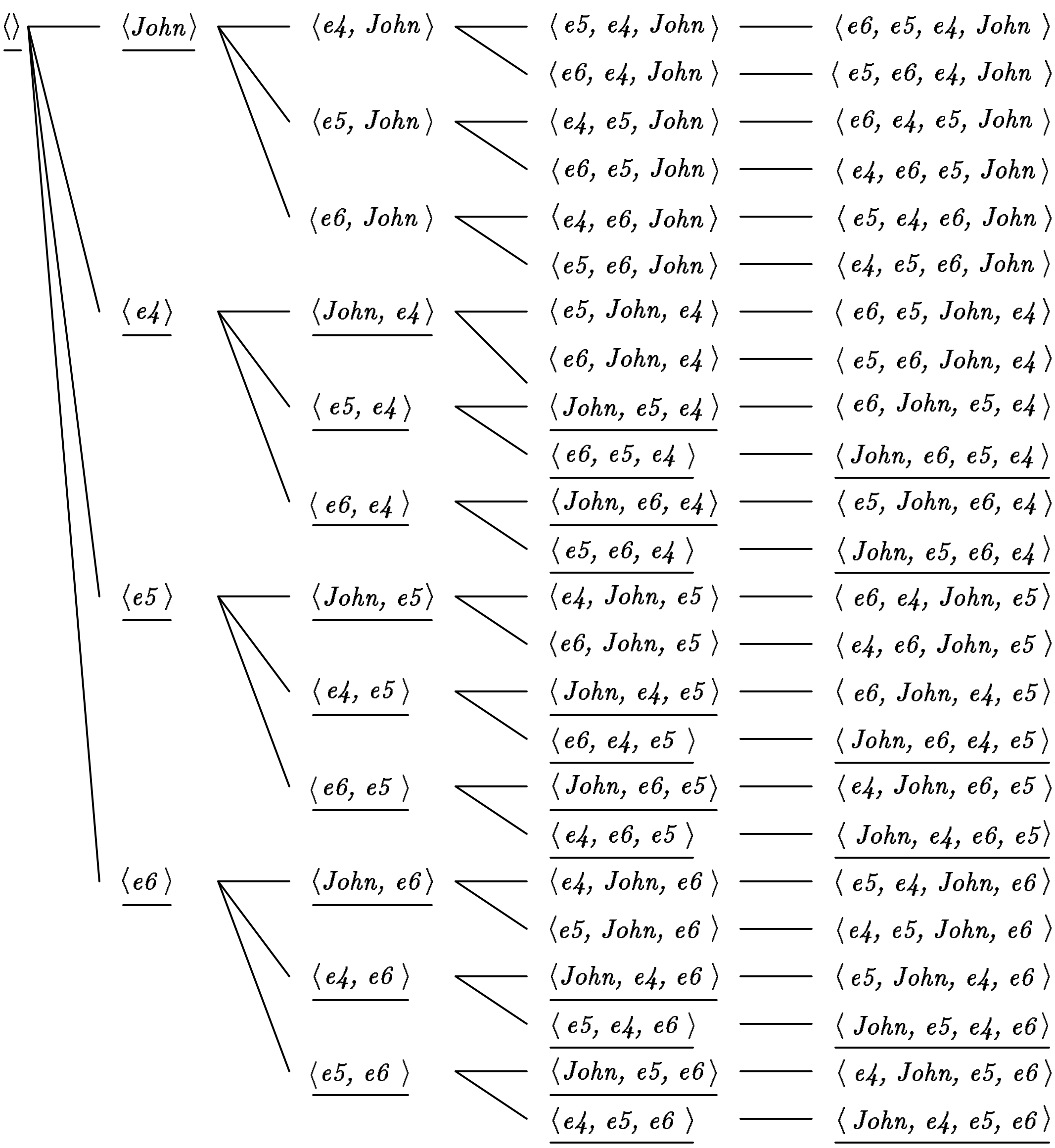


Figure 3.18 r-Permutations of $\{John, e4, e5, e6\}$.

Operation: `merge1` $: C \times F_1 \times F_2 \mapsto C' \times F_1' \times F_2'$.
<u>Either</u>:
1. Do nothing, i.e. let $C' = C$, $F_1' = F_1$ and $F_2' = F_2$,
<u>Or</u>:
2a. <u>Select</u> an element $f \in F_1 \cup F_2$, and
2b. <u>Select</u> a chain $c = \langle E_1, \ldots, E_n \rangle \in C$, and
2c. Let c' be the chain $\langle f, E_1, \ldots, E_n \rangle$, and
2d. Let C', F_1' and F_2' be the result of `merge1`(C'', F_1'', F_2'') where:
  i. $C'' = (C - \{c\}) \cup \{c'\}$ if c' is incomplete,
   otherwise let $C'' = C - \{c\}$,
  ii. Let $F_1'' = F_1 - \{f\}$ if $f \in F_1$, otherwise $F_1'' = F_1$,
  iii. $F_2'' = F_2 - \{f\}$ if $f \in F_2$, otherwise $F_2'' = F_2$,

`merge1` is non-deterministic because there are three choice points in its definition. First, we can either choose to extend a chain (step 2) or do nothing (step 1). Secondly, if we choose to extend a chain, we can form distinct chains by using different selections of free elements and chains at steps (2a) and (2b). Note also that `merge1` is guaranteed to terminate since F_1 and F_2 are finite sets. To show that `merge1` is correct, we need to show (1) that the chain conditions are preserved by `merge1`, and (2) that all possible extensions are permitted by the choice points in `merge1`:

i. Preservation of Chain Conditions:
Since X is an interior node, the requirement that $[$*NP*-e $]$ be allowed to participate in a chain, (31a), is trivially satisfied. Ignoring (31b) for the present, obviously, the assignment of an element to at most one chain, condition (31c), is trivially preserved when step (1) is taken. Let us consider the case when step (2) is taken instead. Here, parts (ii–iii) of step (2d) will ensure that all free elements will be removed from further consideration once they have been selected to extend a chain. Hence, condition (31c) is satisfied in either case.

ii. All Possible Extensions:
In general, the possible r-permutations of a set can be generated recursively by a number of different methods (see Page & Wilson [41] or Reingold *et al.* [45]). The permutation tree shown previously in figure 3.18 illustrates one such method. Here, each r-permutation (of length r), $r \geq 0$, is extended to derive the possible $(r+1)$-permutations by prefixing it with one of the unused free elements. The recursive procedure for `merge1` is basically an implementation

of this method.[20] For simplicity, suppose there is only one chain c in C. If step (1) is chosen, then `merge1` will return c unmodified. This is equivalent to prefixing c with the 0-permutation, or the empty sequence, at the root of the permutation tree. If step (2) is chosen instead, then c will be prefixed with one of the free elements. This is equivalent to moving from the root down to one of the singleton prefixes in the permutation tree. Next, `merge1` will be applied recursively. In the second application, we can choose to either remain at the current node in the permutation tree or move to an descendant one level further down, and so on. Moreover, the removal of completed chains in part (i) of step (2d) corresponds to the pruning of invalid prefixes as illustrated in figure 3.18. By induction on the recursive step, it is clear that `merge1` can visit all the valid prefixes in the permutation tree. Finally, the selection step in (2b) generalizes the method for the case when C happens to contain more than one chain. Hence, each time a particular chain gets selected, we move to a node one level further down in the permutation tree for that chain.[21]

For binary branching, we must apply `merge1` twice: once to extend the chains in Y and once more to extend the chains in Z. Let `merge2` represent the following composite operation:

(34) Operation: `merge2` : $(C_Y, F_Y) \times (C_Z, F_Z) \mapsto (C_X, F_X)$
1. Let `merge1`(C_Y, F_Z, F_Y) produce C'_Y, F'_Z and F'_Y.
2. Let `merge1`(C_Z, F'_Y, F'_Z) produce C'_Z, F''_Y and F''_Z.
3. Then let $C_X = C'_Y \cup C'_Z$ and $F_X = F''_Y \cup F''_Z$.

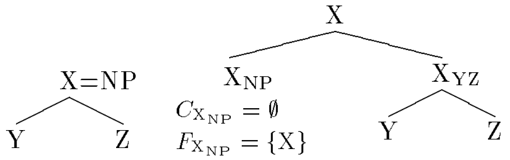
Notation: For simplicity, we will sometimes represent a pair of chains and free elements, e.g. $(C_{\mathrm{W}}, F_{\mathrm{W}})$, by the node to which the pair belongs, e.g. W in this case.

The last step indicates that the set of chains for X is just the union of the sets of chains for Y and Z extended by `merge1`. Similarly, the free elements of X are simply those free elements of Y and Z not incorporated into chain extensions by `merge1`.[22]

[20] All possible selections may be explored by a backtrack algorithm that systematically explores each alternative at all three choice points.

[21] Of course, since a free element will be removed from the appropriate free set once it has been selected to extend a chain, the prefixes represented by the current nodes in the permutation trees will be disjoint at any given point during the application of `merge1`.

[22] Note that `merge1` must be applied serially to preserve the property that a free element cannot be used to extend more than one chain.



(a) (b)

Figure 3.19 Expansion of binary branching when X=NP

We now return to consider chain condition (31c), which states that an overt NP has the option to head a (non-trivial) chain. If the interior node X itself happens to be a noun phrase, e.g. as in [*NP* [*NP who*] *that John knows*], then we must allow for the possibility that the noun phrase has been moved, as in [*NP who that John knows*]$_i$ *does he like* t_i (example [1:15a] from Lasnik & Uriagereka). In this case we can proceed as follows:

(35) 1. Let $C_{\mathrm{X_{NP}}} = \emptyset$ and $F_{\mathrm{X_{NP}}} = \{X\}$.
2. Let $C_{\mathrm{X_{YZ}}}$ and $F_{\mathrm{X_{YZ}}}$ be $\texttt{merge2}(Y, Z)$.
3. Then let C_X and F_X be $\texttt{merge2}((C_{\mathrm{X_{NP}}}, F_{\mathrm{X_{NP}}}), (C_{\mathrm{X_{YZ}}}, F_{\mathrm{X_{NP}}}))$.

Here, X will be allowed the option of participating in chain formation at step 3, as required by condition (31c). This procedure is equivalent to treating binary branching of the form shown in figure 3.19a as being equivalent (with respect to chain formation only) with the repeated binary branching structure shown in figure 3.19b.

(c) n-ary branching: Along the same lines, general n-ary branching, for $n > 2$, may be treated as instances of repeated binary branching. For example, suppose X has three constituents U, V and W. Then the chains and free elements of X will be given by $\texttt{merge2}(U, \texttt{merge2}(V, W))$.

We now return to consider the final chain condition, (31d), which states that all chains in a structure must be complete. Let the root node be denoted by S. Then, (31d) is equivalent to the following constraint:

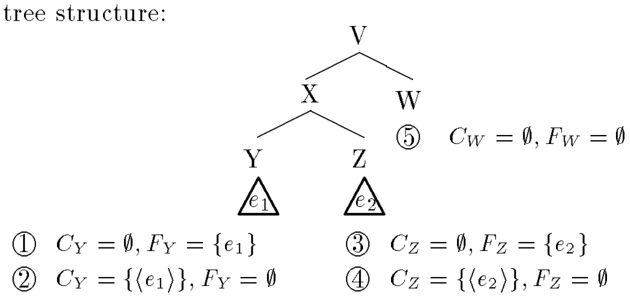
(36) $C_S = \emptyset$

That is, there can be no partial chains (left) at the root node.

To summarize, the chain formation procedure described so far admits all possible chain permutations that satisfy the chain conditions in (31). But, before going on to illustrate how the procedure is actually represented, we briefly consider two additional constraints to be imposed on chain formation.

Eliminating Redundancies in Chain Formation

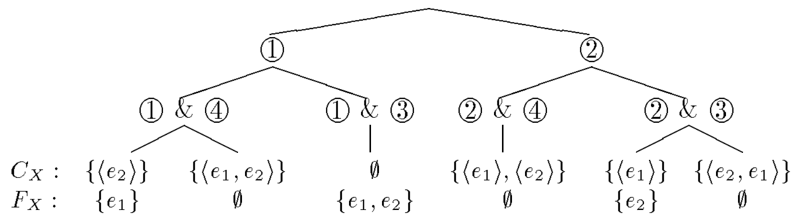
The chain formation procedure overgenerates in the sense that it is capable of producing some chain permutations in more than one way. For example, consider the followng tree structure:



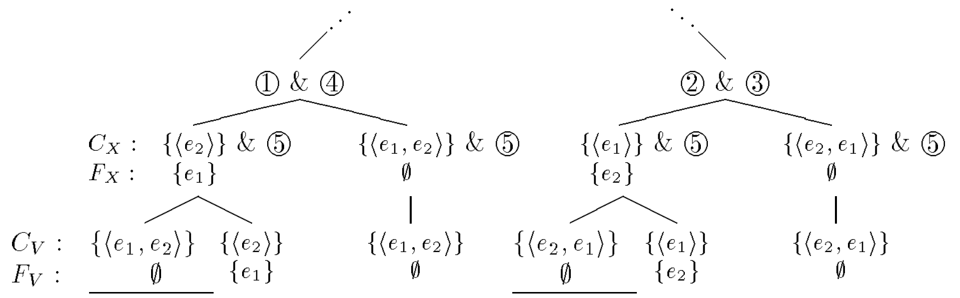
In particular, consider the node X that dominates two leaf nodes, Y and Z, both containing a single empty NP. Note that each child of X has two possible chain permutations, labelled as ① and ② for Y, and ③ and ④ for Z. Applying `merge2` to each chain permutation for Y and Z in turn, we can deduce that X has six possible chain permutations, as the decision tree in figure 3.20a illustrates. Let us now consider the possible chain permutations for node V, i.e. X's parent. Here, V also dominates a leaf node W such that $C_W = F_W = \emptyset$. Since W does not contribute to chain formation, it should be the case that V directly inherits the six permutations for X. However, the algorithm described so far will generate eight chain permutations for V, two of which will be repeated, namely $C_V = \{\langle e_1, e_2 \rangle\}, F_V = \emptyset$ and $C_V = \{\langle e_2, e_1 \rangle\}, F_V = \emptyset$. Figure 3.20b.(i–ii) illustrates the expansion of the decision tree in figure 3.20a for these two cases. The redundancy arises because the algorithm allows e_1 (*resp.* e_2) to extend the chain $\langle e_2 \rangle$ (*resp.* $\langle e_1 \rangle$) either when the chains for Y and Z are merged, or at the next stage when the chains for X and V are merged. To avoid generating these duplicate permutations, a free element must be permitted *exactly* one opportunity to extend any given chain. In terms of the definition of `merge2` given in (34), this can be assured by adding the following requirement:

(37) The first element chosen to extend any chain from C_Y in step 1 (*resp.* C_Z in step 2) must not come from F_Y (*resp.* F_Z).

Returning to the example in figure 3.20b.(i), e_1 may be used to extend $\langle e_2 \rangle$ when the chains for Y and Z are merged because e_1 and e_2 belong to different branches. However, e_1 cannot be used to extend $\langle e_2 \rangle$ once Y and Z have been merged, e.g. when constructing the chains for V or for any ancestor of V, because they will then be part of the same sub-tree. Hence, e_1 has exactly one opportunity to extend $\langle e_2 \rangle$; that is, when the chains for Y and Z are merged. Hence, the duplicate chain



(a) Decision tree for the chain permutations of X.



(i) Generating $C_V = \{\langle e_1, e_2\rangle\}, F_V = \emptyset$. (ii) Generating $C_V = \{\langle e_2, e_1\rangle\}, F_V = \emptyset$.

(b) Extensions of the decision tree in (a).

Figure 3.20 An example of redundant chain formation.

permutation in figure 3.20b.(i) (shown underlined) will be eliminated. A similar argument holds for the other case, i.e. the extension of $\langle e_1\rangle$ by e_2 in figure 3.20b.(ii).

In general, requirement (37) will not compromise the ability of `merge2` to produce all possible chain permutations.[23] In other words, the extra requirement only

[23] More formally, this can be proved by induction on tree structures:

1. *Base case:* We have seen that the algorithm correctly generates all possible chain permutations for leaf nodes. Since requirement (37) only applies to the compositional case, correctness for the base case is trivially preserved.
2. *Induction step:* Suppose that the algorithm correctly generates all possible chain permutations for two distinct sub-trees T_1 and T_2. Let node N be the parent of both T_1 and T_2. Now, consider the merging of the chains from T_1 and T_2 for node N. We know from earlier discussion that `merge2` produces all possible chain permutations for N. But requirement (37) will eliminate all instances generated by `merge2` that (initially) extends a chain with a free element from the same sub-tree. Let e and $c = \langle c_1, \ldots, c_n\rangle$ be any such free element and chain, respectively. Then (37) eliminates $c' = \langle e, c_1, \ldots, c_n\rangle$ generated for N in this manner. But, by the induction hypothesis, c' is also a chain permutation of T_1 (*resp.* T_2), assuming e is from T_1 (*resp.* T_2), since e and the elements of c all belong to T_1 (em resp. T_2). Hence, (37) eliminates duplicates.

$$[NP \cdots [NP\text{-t}\ ]_i\ \cdots\ ]_i$$

Figure 3.21 An example of an antecedent that dominates its own trace.

eliminates instances of chain permutations that are also generated by the algorithm in other ways. In fact, the algorithm will now only generate distinct chain permutations.[24] Combining these two properties, we have that the augmented algorithm generates all and only the possible chain permutations.

Eliminating Antecedents with Embedded Traces

Finally, the chain formation procedure also overgenerates in the sense that it is capable of constructing certain chains that cannot be produced by Move-α. For example, Move-α cannot generate an instance of an antecedent that dominates its own trace, as shown in figure 3.21. Therefore, the actual procedure must also contain an *"i-within-i"* condition that must be satisfied by all chains. In terms of the definition of `merge1` given in (33), the extended chain c' (constructed in step (2c)) must satisfy the following extra condition:

(38) $\forall i, 1 \leq i \leq n, \quad f \not\preceq E_i$

where the symbol $\preceq$ denotes the dominates relation.

The Representation of Chain Formation

The chain formation algorithm described above may be encoded in a fairly direct manner, as shown in figure 3.22. Here, the predicate `chainFormation/3` takes as

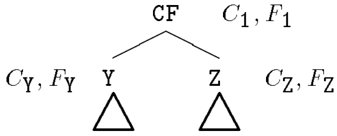
[24] As in the previous note, this can be proved by induction on tree structures. We augment the proof in the previous note as follows:

1. *Base case:* From the description of the algorithm, a leaf node can produce either one or two distinct chain permutations depending on whether it is an empty NP or not. Hence, the algorithm correctly generates all possible chain permutations *without duplicates* for leaf nodes.

2. *Induction step:* Suppose all possible chain permutations without duplicates are generated for T_1 and T_2. We now have to show that no duplicates can be introduced by extending the chains of T_1 and T_2. Since the first element, say e, added to any chain, say $c = \langle c_1, \ldots, c_n \rangle$, in T_1 (*resp.* T_2) must come from T_2 (*resp.* T_1), it is clear that the extended chain $\langle e, c_1, \ldots, c_n \rangle$ could not have been generated previously within either T_1 or T_2. Of course, subsequent extensions of this 'new' chain using free elements from either sub-tree does not alter the fact that it could not have been generated at a lower level. Also, since a free element cannot be selected twice, all prefixes, i.e. r-permutations of free elements, generated by `merge2` must be distinct. Hence all chain permutations for N must be distinct.

```
% (31d) All chains must be complete.
:- chainFormation(CF) top chainFormation(CF,[],_).% (36)  $C_S = \emptyset$

:- chainFormation produces [Chains,FreeSet]
        compositional_cases_on CF where    % Compositional case:
          CF with_constituents Y and Z st
                Y produces [CY,FY],
                Z produces [CZ,FZ],
                CF produces [C1,F1]
          then  merge2(CY,FY,CZ,FZ,C1,F1)
          finally extendChainIfNP(CF,C1,F1,Chains,FreeSet)
        else
          emptyNP(CF) then optStartChain(CF,Chains,FreeSet). % Base case.
```




```
emptyNP(NP) :- cat(NP,np), ec(NP).

% (optionally) start a chain
optStartChain(NP,[[NP]],[]).                % $C_{NP} = \{\langle NP \rangle\}, F_{NP} = \emptyset$
optStartChain(NP,[],[NP]).                  % $C_{NP} = \emptyset, F_{NP} = \{NP\}$

% (33) merge1 : $C \times F_1 \times F_2 \mapsto C' \times F_1' \times F_2'$
merge1(C,F1,F2,C,F1,F2).                    % Do nothing, or ...
merge1(C,F1,F2,Cp,F1p,F2p) :-
        selectElement(F,F1,F2,F1pp,F2pp),   % $F \in (F_1 \cup F_2), F_1'' = F_1 - \{F\}, F_2'' = F_2 - \{F\}$
        extendAChain(F,C,C1),               % C1 is C with some c $\in$ C extended by F
        merge1(C1,F1pp,F2pp,Cp,F1p,F2p).

% (34) merge2 : $(C_Y, F_Y) \times (C_Z, F_Z) \mapsto (C_X, F_X)$
merge2(CY,FY,CZ,FZ,CX,FX) :-
        merge1(CY,FZ,FY,CYp,FZp,FYp),       % Extend chains in Y.  $C_Y \mapsto C_Y'$
        merge1(CZ,FYp,FZp,CZp,FYpp,FZpp),   % Extend chains in Z.  $C_Z \mapsto C_Z'$
        union(CYp,CZp,CX),                  % $C_X = C_Y' \cup C_Z'$
        union(FYpp,FZpp,FX).                % $F_X = F_Y' \cup F_Z'$

% (35)
extendChainIfNP(NP,CYZ,FYZ,CX,FX) :-
        cat(NP,np)
        -> merge2([],[NP],CYZ,FYZ,CX,FX)    % merge2$((\emptyset, \{NP\}), (C_{X_{YZ}}, F_{X_{YZ}}))$ if NP
        ;  CYZ = CX, FYZ = FX.              % Do nothing if not an NP
```

Figure 3.22 Representation of the chain formation algorithm.

input a constituent `CF` in which chains are to be assigned, and produces as output an assignment of partial chains `Chains` and free elements `FreeSet` for that constituent. The top-level interface to chain formation is formed by a predicate with the same name, but of arity one instead. This top-level predicate (given a complete structure as input) will hold only if `chainFormation/3` holds with no partial chains (left) at the root. Hence, this 'top' condition encodes chain condition (31d), which states that all chains must be complete.[25] The remaining predicates shown directly encode each of the various chain formation operations in turn, namely chain condition (31a), `merge1`, `merge2` and chain condition (31c) (as realized by (35)). As with free indexation, the non-determinism present in these definitions will be automatically handled by the underlying PROLOG search mechanism. For example, chain condition (31a) is encoded (in part) by the two clauses that constitute predicate `optStartChain`. The choice between the two clauses represents the non-determinism present in the condition; that is, the option to participate in a (non-trivial) chain or not. Similarly, `merge1` is defined by two clauses, representing the choice between step 1 and step 2 in the procedure given in (33). The remaining two choice points in that procedure are contained in `selectElement` and `extendAChain` (neither definition shown here). Condition (37), which prevents the generation of duplicate chain permutations, is also enforced within these two predicates.[26] Finally, condition (38), which eliminates cases of impossible movement, is also enforced within `extendAChain`.[27]

This completes our discussion of how compositional definitions are used to represent combinatorial operations. In the next section, we will go on to illustrate how principles that involve *minimality* such as the Binding conditions may be con-

---

[25] Note that no such constraint is placed on the corresponding set of free elements, as indicated by the *don't care* '_' variable. In other words, any number of elements may remain free.

[26] There are numerous ways to encode condition (37). The actual implementation makes use of a simple marking feature associated with both chains and free elements. All chains and elements are initially unmarked when `merge1` is called in `merge2`. Also, marking will be local to `merge1`, i.e. the chains and free elements returned to `merge2` will not be marked. All elements selected from F_2 are marked. A marked element can only extend a marked chain. An unmarked element can extend any chain. If a chain is extended by an unmarked element, then the extended chain is marked. Hence, once a chain has been initially extended by an element from F_1, it can be further extended by any element from either F_1 or F_2.

[27] The actual implementation makes use of the property that all elements in a sequence will be coindexed. Indices are implemented as syntactic features of the form `index(I)`, where `I` is (always) a logical variable. Condition (38) implies that all NPs, e.g. $\mathrm{NP}_{j_1}, \mathrm{NP}_{j_2}, \ldots, \mathrm{NP}_{j_n}$, dominated by a larger NP cannot be coindexed with the larger NP, call it NP_i. Hence, for each large NP of this sort, we maintain the following list of constraints: $\{i != j_1, i != j_2, \ldots, i != j_n\}$. Here, $j != k$ means that variable j can never be unified with variable k. Hence, every time a chain is extended by some free element `F` within `extendAChain`, these constraints will be checked when the index of `F` is unified with the index of the head of the (partial) chain. (If any constraint is violated, then backtracking will ensue.) The implementation of PROLOG used in the current system does not directly support the use of 'anti-unifiers'. Hence, the aforementioned checks have to be manually implemented. To keep the definition of chain formation simple, we have omitted the code that implements anti-unification.

veniently encoded using the `in_all_configurations` form, but are automatically transformed into a compositional fashion.

3.5 Compositional Definitions Part II: Binding Conditions

The fundamental reason why compositional definitions are necessary for free indexation and chain formation to work with principle interleaving is that in these cases the relevant syntactic relation being expressed is essentially a *non-local* (or unbounded) one. For example, any noun phrase may be potentially coindexed with another noun phrase, regardless of the 'distance' between them. The same situation holds for chain composition. (Of course, there are many principles such as the Binding conditions, Control of *PRO* and Subjacency that may or may not impose locality restrictions on the subsequent assignments.) By contrast, the definitions of structural and inherent Case assignment are local ones because Case must be assigned under government, a *local* relation. In this section, we turn to the formulation of Binding conditions that express locality restrictions on the possible referents of anaphors, e.g. *himself*, pronominals, e.g. *him*, and referential expressions, e.g. names like *John*. In chapter 1, we introduced the following two Binding conditions:

(A) An anaphor must be coreferent with an element in its clause.
(B) A pronoun must ***not*** be coreferent with any element in its clause.

We also saw that this (purely local) definition proved sufficient to explain some simple facts, repeated here as (39):

(39) a. *[John$_1$ likes him$_1$] — *cf.* [John$_1$ likes him$_2$]
b. John$_1$ believes that [Mary likes him$_1$]
c. John$_1$ believes that [Mary likes him$_2$]
d. [John$_1$ likes himself$_1$] — *cf.* *[John$_1$ likes himself$_2$]
e. John believes that [Mary$_1$ likes herself$_1$]
f. *John$_1$ believes that [Mary likes himself$_1$]

Of course, the simple notion of the relevant Binding domain being the local clause cannot account for more complex examples such as the following (taken from Lasnik & Uriagereka):

(2:26a) John likes [Mary's pictures of him]
(2:26b) *John likes [Mary's pictures of himself]
(2:29) [John likes pictures of himself]
(2:91) [The men think that pictures of each other will be on sale]
(2:103) *The men think that [Mary's pictures of each other will be on sale]

Here, as in (39), we have used brackets to delimit the extent of the relevant Binding domain in each case. The problem now is that the domain can have an arbitrarily large extent. As with free indexation and chain formation, we will need to use compositional definitions if the Binding conditions are to be compatible with the principle interleaving control model. The difference being that it will be possible to retain the transparency of a universally quantified condition over tree structures for the Binding conditions, whilst maintaining compatibility with principle interleaving by using an automatic transformation between the two representations.

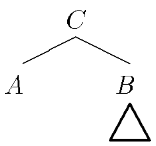
3.5.1 Using the `smallest_configuration` form

To handle the examples shown in the previous section, Lasnik & Uriagereka introduces the notion of a *governing category* (GC) as follows:

(40) (A) An anaphor must be A-bound in its GC.
(B) A pronominal must be A-free in its GC. *(adapted from (2:27))*

To be *bound* (equivalently, not *free*), an anaphor or pronominal must be coindexed with another noun phrase that *c-commands* the former noun phrase. (As with the previous definition, the same interpretation rule still hold; that is, if two elements are coindexed, then they are assumed to be coreferential.[28]) The canonical case in which the structural relation c-command obtains is the following:

(41)



Here, the constituent A c-commands B and any constituent dominated by B, where C is the first 'branching node' that dominates A. Note that the linear order of A and B is not significant. Finally, the term *A-bound* refers to being bound by an element in an A-position, i.e. a subject or object position. (Similarly, being A-free is the same as not being A-bound.)

We are now in a position to define the notion of a governing category:

(42) The GC of α is the minimal NP or S containing α,a governor of α, and a SUBJECT accessible to α. *(from ch.2, no.105)*

[28] There is one exception. The subject of a clause and inflection will be coindexed for agreement, not binding purposes. See the definition of `binds/3` in figure 3.28.

(43) β is accessible to α if *(from ch.2, no.106)*
 a. β c-commands α, and
 b. β is not coindexed with any category containing α

Here the notion SUBJECT (introduced by Chomsky [9]) extends the standard notion of subject, as in the subject of a clause, e.g. *pictures* in *pictures are on sale*, or the subject of a noun phrase, e.g. *Mary* in *Mary's pictures of him*, to include (non-overt) inflection (actually, the agreement component of inflection) present in tensed clauses.

The representation of the preceding definitions is shown in figure 3.23. The only new feature is the `smallest_configuration` construct introduced to define the notion of a governing category. It is meant to be read as follows: "`gc(X)`, the governing category of a constituent `X`, is defined to be the smallest configuration `CF` such that `CF` is a `gcDomain` (defined for `i2` and `np`) with the following components: (1) `X` itself; (2) an element `G` such that `G` governs `X` in `CF`; and finally (3) a SUBJECT `S` accessible to `X` in `CF`." (The `smallest_configuration` form is used to define many other cases of (minimal) domains. We shall see one such example shortly in the definition of Condition C.) We will omit discussion of the other predicates shown in the figure 3.23 since they are fairly straightforward translations of the appropriate definitions.[29]

The `smallest_configuration` form define an operation `gc(X,GC)` that can be called within the context of a universally quantified condition over tree structures, as will be explained below.

3.5.2 Reintroducing the `in_all_configurations` Form

Condition A of the Binding theory (repeated below) expresses a universal condition on anaphors:

(44) An anaphor must be A-bound in its GC.

Compare (44) with its representation shown in figure 3.24. The universal condition has the following reading: "in every configuration `CF` such that `CF` is an anaphor, then *if* `GC` is identified to be the governing category for `CF`, then `CF` must be A-bound in `GC`." The definition of an anaphor as a [+anaphoric] noun phrase and an A-bound constituent as an element that is bound by another element `A` in an A-position are both straightforward translations. However, the feature that distinguishes such a definition from the parallel (but strictly local) definition of, say, the Case filter, as shown earlier in figure 3.3, is the presence of the goal `gc(CF,GC)` that identifies a

[29] As with the *i-within-i* condition for chain formation (described in note 27), we have omitted the explicit constraint checks necessary to fully implement the notion of an accessible SUBJECT from the description in figure 3.23.

```
gcDomain(i2).  gcDomain(np).             % theory parameter.

:- gc(X) smallest_configuration CF st cat(CF,C), gcDomain(C)
        with_components
                X,
                G given_by governs(G,X,CF),
                S given_by accSubj(S,X,CF).

accSubj(Subj,A,CF) :-                    % Subj is accessible to A.
        bigSubject(Subj,A,CF),
        c_commandsBT(Subj,A,CF).

bigSubject(SUBJ,_,XP) :- subject(SUBJ,XP). % ordinary subject.
bigSubject(AGR,A,IP) :-                  % I(AGR)
        cat(IP,i2),
        IP has_feature agr(_),
        ISpec specifier_of IP,
        \+ properlyDominates(ISpec,A),
        AGR head_of IP.

% An ordinary subject.
subject(SUBJ,IP) :- cat(IP,i2), SUBJ specifier_of IP.
subject(SUBJ,NP) :- cat(NP,np), SUBJ specifier_of NP, cat(SUBJ,np).

properlyDominates(X,Y) :- \+ X=Y, X dominates Y. % defined in figure 3.28.

% Normal c-command + I(AGR) c-commands subject of IP.
% Compare this definition to the one shown in figure 3.28.
c_commandsBT(AGR,NP,IP) :-
        cat(IP,i2),
        IP has_feature agr(_),           % ..using inheritance of features.
        NP specifier_of IP,
        AGR head_of IP.
c_commandsBT(A,B,S) :-
        S has_constituents Cs,
        in(A,C1,Cs),                     % A,C1∈Cs, A≠C1.
        C1 dominates B.
c_commandsBT(A,B,S) :-
        S has_constituent C,
        c_commandsBT(A,B,C).
```

Figure 3.23 Governing Category.

```
:- conditionA in_all_configurations CF
        where anaphor(CF) then gc(CF,GC), aBound(CF,GC).

anaphor(NP) :- cat(NP,np), NP has_feature a(+).

aBound(B,S) :- binds(A,B,S), A has_feature apos.
```

Figure 3.24 The definition of condition A.

```
conditionA produces [CFs1]
        compositional_cases_on CF where
        CF with_constituents A and B st % Binary branching case.
                A produces [As],
                B produces [Bs],
                CF produces [CFs]
        then app1(As,Bs,CFs)          % append As to Bs ↦ CFs.
        finally conditionAml(CFs,CF,CFs2),
                (anaphor(CF)          % Is CF itself an anaphor?
                 -> add(CF,CFs2,CFs1) % If so, add it to the list.
                 ;  CFs1 = CFs2)       % If not, no change.
        else anaphor(CF) then initgc(CF,GC), CFs1 = [GC]. % Base case.
% GCs × CF ↦ GCs'.
% GCs' represents those elements of GC that remain incomplete in CF.
conditionAml([],_,[]).
conditionAml([GC|GCs],CF,GCs1) :-
        gc(X,GC,CF),                  % Try to complete the GC in CF.
        (completegc(GC)               % Is it now complete?
        -> abound(X,GC),              % If so, see if X is A-bound in it.
           GCs1 = GCs2                % If so, cond. A satisfied. Discard X.
        ;  GCs1 = [GC|GCs2]),         % If not, keep GC around.
        conditionAml(GCs,CF,GCs2).    % Go on to the next element...
```

Figure 3.25 The compositional version of condition A.

```
:- conditionB in_all_configurations CF
        where pronominal(CF) then gc(CF,GC), \+ aBound(CF,GC).

pronominal(NP) :- cat(NP,np), NP has_feature p(+).
```

Figure 3.26 Condition B.

configuration strictly larger than `CF`, i.e. `CF` is only an anaphor, but `GC` must include `CF`. In the current system, the definition of condition A is automatically transformed into the equivalent compositional definition shown in figure 3.25. We will not go into the details of the transformed definition except to note that the definition basically computes the list of anaphors with incomplete governing categories associated with every node in a phrase structure.[30]

We should also point out here that the grammar writer never needs to know (nor explicitly access) the reformulated version.

Finally, note that Condition B (repeated below) can also be encoded in a parallel fashion to Condition A. Compare the statement in (45) to the form in figure 3.26.

(45) A pronominal must not be A-bound in its GC.

In the next section, we will discuss one other example involving the `smallest_configuration` form, and outline how we can take advantage of the transformation process to produce not only a more perspicuous definition, but also a more efficient one.

Condition C of the Binding Theory

Transforming a definition into a compositional one has other benefits besides compatibility with the principle interleaving model. The chief benefit is the automatic elimination of some unnecesary work. To illustrate how this works, let us consider one last principle of the Binding theory, namely condition C which applies to referential expressions. The following definition is taken from Chomsky [12]:

(46) An r-expression must be A-free (in the domain of its operator)
(= (3:56))

[30] The compositional algorithm is actually much simpler than it appears. The `compositional_cases_on` form just wraps the standard inductive tree definition around the computation of the governing category in `conditionAml/3`. That is, `conditionAml/3` will be called for every interior node in a given structure (in a bottom up fashion). `conditionAml/3` will try to complete the GC for each unbound anaphor in each interior node. If a GC has been found, it calls `aBound/2` as required by the original definition in figure 3.24. Otherwise, to avoid unnecessary recomputation, it saves the partially computed GC for the next time around. Because of the bottom-up strategy used, the compositional definition is guaranteed to find the (smallest) GC.

```
:- conditionC in_all_configurations CF
        where rExpression(CF) then aBinderDom(X,D), aFreeCond(X,D).

rExpression(NP) :- cat(NP,np), NP has_feature_set [a(+),p(+)].

:- aBinderDom(X) smallest_configuration CF
        with_components
                X,
                A given_by aBinds(A,X,CF).

aBinds(A,B,S) :- binds(A,B,S), A has_feature apos.

aFreeCond(X,D) :-                % only way R-expr can be A-bound is
        variable(X),             % outside the domain of its operator
        X has_feature operator(Op),
        D has_constituent D1,
        properlyDominates(D1,Op).% defined in figure 3.23.
```

Figure 3.27 Condition C.

(46) is understood to be a disjunctive condition as shown in (47) (= (3:57) in Chomsky [12]) below with (47b) only applying when (47a) is inapplicable:

(47) a. An r-expression must be A-free in the domain of its operator.
b. An r-expression must be A-free.

The first case applies to variables such as e_1 in (48):

(48) [*NP* [*NP* The man_1] [*CP* Op_1 I saw e_1]] decided to shoot John

Here, for purposes of strong Binding, Chomksy [12] (which we will not discuss here), e_1 must also be bound by *the man*. Hence, the domain of the operator (Op) must be CP. The second case applies to non-variables such as *John* in (49).

(49) *John_1 can't stand John_1's teacher *(= (2:36a) in Lasnik & Uriagereka)*

Compare the definition in (46) with its realization in figure 3.27. Here, we have made use of the `smallest_configuration` form in the definition of `aBinderDom(X)` which finds the smallest constituent that dominates an A-binder of `X`. If such a domain can be found, then `aFreeCond(X,D)` must hold. Now, the only way that `aFreeCond(X,D)` can hold is if `X` is a variable such that its operator `Op` is properly dominated by some element `D1` within the domain `D` in which the variable happens to be A-bound.

```
binds(A,B,S) :-
        c_commands(A,B,S),
        coindexed(A,B),
        \+ A has_feature indexed(agr).% make sure not coindexed
                                      % for agreement.
% A c-commands B in S
c_commands(A,B,S) :-
        S has_constituents Cs,
        in(A,C,Cs),                   % A,C ∈ Cs, A ≠ C.
        C dominates B.
c_commands(A,B,S) :-
        S has_constituent C,
        c_commands(A,B,C).

coindexed(A,B) :-
        index(A,I),                   % defined previously in figure 3.17.
        index(B,J),
        sameIndex(I,J).               % primitive.

A dominates A.
A dominates B :-
        A has_constituent C,
        C dominates B.
```

Figure 3.28 Binding.

Now let us consider the computation of the (smallest) A-binder domain as defined by `aBinderDom`. Currently, because condition C is transformed into a compositional definition, each time a larger constituent is built that dominates an R-expression `X`, an attempt will be made to satisfy the conditions of `aBinderDom(X)`. That is, for each such constituent `CF` the parser will attempt to find (in a *bottom-up* fashion) an `A` such that `aBinds(A,X,CF)` holds. As soon as an A-binder is found, then the (smallest) domain is complete and `aBinderDom(X,D)` will succeed. Then, Condition C will succeed only if `aFreeCond(X,D)` holds. Now, consider the definition of `aBinds/3`. `aBinds/3` relies on the definition of `binds/3` shown in figure 3.28. In turn, `binds/3` is defined using `c_commands/3` which holds for all triples A, B and S such that A c-commands B for any constituents A and B within S. There are two clauses for `c_commands/3`:

1. The first clause implements the canonical case which was shown previously in (41)). Here, A (the c-commander) is an immediate sub-constituent of S.

```
:- aBinderDom(X) smallest_configuration CF
        with_components
                X,
                A given_by locallyABinds(A,X,CF).

locallyABinds(A,B,S) :- locallyBinds(A,B,S), A has_feature apos.

locallyBinds(A,B,S) :-
        locallyC_commands(A,B,S),
        coindexed(A,B),
        \+ A has_feature indexed(agr).
locallyC_commands(A,B,S) :-
        S has_constituents Cs,
        in(A,C,Cs),% A,C ∈ Cs, A ≠ C.
        C dominates B.
```

Figure 3.29 Localized version of `aBinderDom(X)`.

2. The second clause accounts for the case when A is not an immediate but a 'deeper' sub-constituent of S.

Hence, when `c_commands` is called in the course of computing an A-binder domain, first the immediate constituents of a candidate domain are tested, then if none of those immediate constituents turn out to be A-binders, the second clause is called to recursively pick out other possible candidates. However, this results in much redundant computation.[31] There is no reason to look any further (down) than the *local* sub-constituents for an A-binder for any particular domain because the bottom-up processing of the compositional definition ensures that previous (unsuccessful) attempts to locate an A-binder would have tried to use lower sub-constituents as candidate A-binders. In other words, the ability of the procedure to locate the smallest domain would not be affected if a minimal version of `c_commands` consisting of just the first clause was used instead. Hence, we can re-define `aBinderDom(X)` as shown in figure 3.29.

The current system automatically detects and produces these transformations. For example, an equivalent situation occurs with the definition of `c_commandsBT/3`. Returning to the governing category domain `gc(X)` (shown in figure 3.23), the same transformation is performed in the computation of an accessible SUBJECT. There, `c_commandsBT/3` has the same basic format as the definition of `c_commands/3` shown

[31] For binary branching, this will be proportional to the depth of the tree.

above.

Chapter 4

The Recovery of Phrase Structure

4.1 Motivation

The mechanism used to recover the phrase structure component of structural descriptions (SDs) is a crucial component of the parser. It must be efficient for the following reasons:

- First of all, linguistic principles that deal with syntactic constituents make broad use of structural relations such as 'subject-of', 'object-of', 'governs', etc. For simplicity, in the current implementation, parser operations require phrase structure descriptions to be recovered before structural relations can be evaluated. For example, the object of a verb *believe*, as in *believe Mary...*, will depend on the structure of the verb phrase (VP) built by phrase structure recovery, e.g. see (1a) and (1b):

  (1) a. [$_{VP}$ [$_{V}$ believe] [Mary]]
  (object is Mary)
  b. [$_{VP}$ [$_{V}$ believe] [Mary saw him]]
  (object is Mary saw him)

- Secondly, there may be many different phrase structure representations for any given input sentence. As is well known, there are two major sources of overgeneration: (1) *lexical ambiguity,* as in the surface form *saw* that may be syntactically realized as a noun or a verb; and (2) *structural ambiguity* in the sense that syntactic constituents may occupy many different positions in PS. For example, in English, a noun phrase that follows a verb may occupy various positions:

  (2) a. [I [$_{VP}$ believe [$_{NP}$ Mary]]]
  (object position)

b. [I [*VP* believe [[*NP* Mary] [*VP* saw him]]]]
(subject position of a complement clause)
c. [I [*VP* realized [*CP* [*NP* Mary] [*IP* [*NP*-e] [*VP* saw him]]]]]
(Comp — a position normally occupied by operators)
cf. [I [*VP* realized [*CP* [*NP* what time] [*IP* it was]]]]

The possibility of much overgeneration suggests that a substantial amount of the work spent in parsing will be spent on phrase structure recovery.[1]

- Thirdly, the cost effectiveness of some control strategies such as principle interleaving, and, to a lesser extent, principle ordering, is dependent on the *optimality* of the phrase structure recovery process in terms of error detection, in a sense to be made clear later.

For these reasons alone, it would seem worthwhile to expend significant effort to optimize phrase structure recovery. Moreover, as this chapter will describe, at least for the examples in Lasnik & Uriagereka, we can take advantage of well-understood and efficient parsing techniques developed for (subsets of) context-free languages (CFLs).

4.2 Roadmap

The remainder of the chapter is divided into four major sections. In the first section, we will discuss how phrase structure recovery is handled in a model of grammar that contains multiple levels of representation. Next, we will describe a phrase structure grammar for S-structure, the level of syntax that is recovered first; and outline an algorithm based on shift-reduce parsing techniques that will be sufficiently powerful to cope with the adopted grammar. Then, we will describe the implementation of the algorithm and how the rule schema specifying the grammar may be compiled into the tabular form required by the underlying machine. Finally, we will conclude by evaluating how well the shift-reduce machine matches the problem of recovering structure at the level of S-structure.

[1] As a rough guide, informal 'profiling' of an early version of the parser suggested that roughly 80% of the parsing effort went into phrase structure recovery for typical sentences from Lasnik & Uriagereka. This proportion dropped to the 20% level when grammar rules were compiled using the canonical LR(1)-based techniques described in this chapter; as opposed to on-line interpretation of the rules with no pre-compilation. Note that the 80% to 20% decrease translates into a twenty-five-fold improvement in phrase structure recovery times. Of course, the actual improvement will vary depending on the degree of ambiguity and the length of the input sentence.

$$\begin{tikzcd}
& \text{D-structure} \arrow[d] & \\
& \text{S-structure} \arrow[dl] \arrow[dr] & \\
\text{Phonetic Form} & & \text{Logical Form}
\end{tikzcd}$$

Figure 4.1 The standard model of phrase structure.

4.3 Multiple Levels of Representation

In this section, we will begin by reviewing a standard model of phrase structure that consists of multiple levels of representation. Next, we will outline, and discuss the computational advantages of, a two stage method to recover these levels of representation.

4.3.1 A Standard Model of Phrase Structure

In general, the phrase structure component of grammar may contain several *levels* of representation related by transformation. For instance, figure 4.1 contains a typical generative model found in the principles-and-parameters framework. In this standard model, the level of D-structure is generated by simple syntactic rules that obey the properties of $\overline{\text{X}}$-theory (to be defined later). It is also the level at which grammatical relations such as 'subject-of' and 'object-of' are structurally defined. For example, consider the (partially specified) D-structure [[*NP the police*] [*VP arrested* [*NP John*]]]. Here, the NPs *the police* and *John* are understood to be the subject and object of the verb *arrest*, respectively. Semantic, or *thematic*, relations are also directly expressed at this level through these grammatical relations. For instance, we say that the subject *the police* bears the thematic role (hereafter abbreviated to θ-role) 'agent' of the predicate 'arrest'. Similarly, we say that the object *John* bears the 'patient', or 'theme', θ-role. Note that in some cases not all such positions need be filled. For example, [[*NP-e*] [*VP was arrested* [*NP John*]]] where the subject position is occupied by an empty NP is also a valid D-structure. Here, *John* still bears the patient θ-role of 'arrest'.

The level of S-structure is generated from D-structure by the operation of Move-α that allows the iterated, possibly null, *movement* of syntactic units. For example, NP-movement is an instance of Move-α where NPs, usually non-empty, are allowed to move to positions occupied by empty NPs. Ignoring morphological considerations, this is how the passive form *John was arrested*, where *John* is understood to bear the patient θ-role of 'arrest', may be generated from the underlying D-structure [[*NP-e*] [*VP was arrested* [*NP John*]]]. More precisely, NP-movement will substitute *John* for [*NP-e*] leaving behind an empty constituent [*NP-t*] in the position previously occupied by *John*, thus obtaining the S-structure [[*NP John*] [*VP was arrested* [*NP-t*]]]. In standard terminology, [*NP-t*] is known as a *trace* of movement.

From S-structure, phonological rules derive the level of phonetic form that forms the interface between language and the external systems that produce speech. Also, from S-structure, the movement of elements relevant to logical interpretation such as quantifiers and *wh*-phrases produces the level of Logical Form (LF) that forms the interface to systems that interpret language. For example, consider the sentence *Mary loves everyone.* Roughly speaking, the LF associated with this example should have the interpretation (3a), where x is a logical variable. Now, the S-structure will have the form (3b). The corresponding LF, (3c), is formed by 'raising' the quantifier *everyone* into a position where it may be interpreted as having sentential scope. In this case, Quantifier Raising (QR) leaves a trace [*NP-t*] that will be interpreted as the logical variable that *everyone* quantifies over.

(3) a. for all persons x, loves(Mary,x) (Interpretation)
b. [[*NP* Mary] [*VP* loves [*NP* everyone]]] (S-structure)
c. [*IP* [*NP* everyone] [*IP* [*NP* Mary] [*VP* loves [*NP*-t]]]] (LF)

4.3.2 Recovering Multiple Levels of Representation

The parser basically adopts the standard model of phrase structure as outlined above. The only significant deviation is that the level of phonetic form has been replaced by a level of Surface Form (SF). That is, instead of an abstract phonetic representation, the input for parsing is just a sequence of words.[2]

Now, let us consider the phrase structure model as a 'black box'. Of the four levels of representation, only the two interface levels, SF and LF, are *externally visible* — the other two, namely D- and S-structure, being purely theory-internal levels of representation. Hence, given some arbitrary SF, the parsing problem is to explicitly compute the corresponding LF structures that satisfy the constraints of LF, and indirectly, all other constraints that apply at the other levels. Note that this does not imply that all levels of phrase structure must be explicitly recovered. In the current implementation, we adopt the two stage approach shown in figure 4.2. In the first stage, we recover the phrase structure component of S-structure from an input SF. The other two levels, LF and D-structure, are not recovered simultaneously.

[2] Currently, morphological forms are mostly simply 'spelled-out' in the lexicon. One exception is for contractions. For example, a contraction like *okotteru* ('*be angry about*') in Japanese will be analyzed (on-line) as having the form *okotte* + *iru* (progressive '*be*').

$$
\begin{tikzcd}
 & & \text{D-structure} & & \\
 & & \boxed{\text{Move-}\alpha} \arrow[u] & & \\
\text{SF} \arrow[r] & \boxed{\text{Recover Phrase Structure}} \arrow[r] & \text{S-structure} \arrow[u] \arrow[r] & \boxed{\text{LF-movement}} \arrow[r] & \text{LF}
\end{tikzcd}
$$

Figure 4.2 The implemented model of phrase structure recovery.

Instead, they are computed from initial S-structures at a later stage (as indicated by the dataflow in figure 4.2.) Partitioning the recovery of phrase structure in this manner has several computational advantages:

- *Recovering all levels simultaneously may involve doing unnecessary work.*

  For example, if an ill-formed structure may be eliminated by some constraint that applies at one particular level; then obviously, there is no need to recover any other level. Consider the (ill-formed) sentence **someone sleep*. The standard explanation for this is that the subject *someone* and the form of the verb *sleep* do not agree — cf. *someone sleeps*. In the theory adopted by the parser, the mechanism by which subject-verb agreement is enforced involves assigning an index feature to both the subject of a clause and an inflection element (hereafter abbreviated to Infl or I) that contains an agreement component (AGR) consisting of person, number and gender features. AGR is, in turn, associated with the verb through head movement. (The details of head movement (to be described in section 4.4.1 are not important for the purposes of this discussion.) Hence, as soon as a candidate S-structure has been assigned, e.g. [*IP* [*NP someone*] [$\bar{I}$ [*I-t*] [*VP* [*V* [*I*] [*V sleep*]]]]] — more precisely, once *someone* has been identified as the S-structure subject — no operation other than subject–Infl coindexation need be applied to rule out the ill-formed structure. In particular, there is no need to construct any part of a LF or D-structure representation.

- *Limiting overgeneration.*

  The decision to recover phrase structure at the level of S-structure first rather than at the level of D-structure or LF was largely motivated by the goal to limit overgeneration. Experience with an early version of the system that initially recovered phrase structure at D-structure suggested that overgeneration was a major bottleneck.[3] The basic problem in this case is that there is no simple

[3] Incidentally, one motivating factor for the D-structure generator was that the grammar would

correspondence between the order in which terminal elements of D-structure appeared and their relative positions in the SF. This is primarily caused by the power of Move-α to freely 'scramble' syntactic elements between D- and S-structure. Moreover, given the lack of a simple correspondence, modifying the recovery routine to allow for the effect of Move-α would require the addition of much detailed knowledge about how the Move-α operation works. Of course, this would severely compromise the goal of maintaining a strict separation between parser operations. Hence, the early version of the phrase structure recovery routine adopted the null hypothesis: that is, the free permutation, or scrambling, of elements between D- and S-structure.[4] Consequently, the routine hypothesized many structures that could not have generated the SF in question. For example, *John saw Mary* cannot be generated from a D-structure [[*NP Mary*] [*VP saw* [*NP John*]]]. Moreover, in order to exclude such impossible forms, Move-α must be applied to generate corresponding candidate S-structure forms. In turn, these candidates must be 'linearized' to check that the order of elements matches that of the original SF. By contrast, in the current theory, there is no powerful scrambling rule between SF and S-structure. In fact, apart from examples of genitive Case realization such as *John's book* and *pictures of themselves* where markers *'s* and *of* are not present at S-structure but inserted at SF, the 'yield' of the SF and S-structure consist of the same sequence of terminal elements. Of course, empty syntactic units such as [*NP-e*] in *John was arrested* [*NP-e*] may be inserted at the level of S-structure, but the linear precedence relations encoded in the SF will remain unaltered at S-structure. Hence, we can make of the invariance in linear precedence to avoid generating many of the aforementioned 'impossible' structures.[5] Oversimplifying somewhat, given a SF of length n, the old D-structure recovery routine would have to attempt to recover structures for $n!$ permutations of the SF, whereas the corresponding S-structure routine only has to deal with the original order.

---

be simpler, i.e. contain fewer terminal and non-terminal symbols, than the corresponding grammar at the level of S-structure. (Section 4.4.1 enumerates the extra cases that a grammar for S-structure has to handle.) Another potential advantage is that a smaller grammar may have a smaller characteristic finite state automata. That is, table size may be less of a problem when it comes to constructing shift-reduce machines.

[4] Although the other alternative of directly recovering LFs has not been explored, the presence of LF-movement parallels that of Move-α for D-structure.

[5] Note that there is no fundamental linguistic basis for this similarity between the level of SF and S-structure. In some theories linear precedence is irrelevant at S-structure. See Marantz (in Baltin *et al.* [8]) or Kashet [32] for an example of a purely 'hierarchical' version of S-structure.

Function: D-structure Constituent Recovery.
Input: A S-structure constituent X_S.
Output: The D-structure constituent at the position occupied by X_S.

1. If X_S occurs in a chain $(X_1, X_2, \ldots, X_n)$[6] then:
   (a) return X_1 if X_n is X_S, otherwise
   (b) return an empty category.[7]
2. Otherwise return X_S. (X_S has not been moved.)

Figure 4.3 D-structure constituent recovery algorithm.

- *Selective recovery of theory-internal levels of representation.*

  Complete LFs must be recovered since that is the designated output of the parser. However, there is no obligation to do the same for phrase structure at the level of D-structure since, as discussed earlier, such structures will not be externally visible. In fact, we can avoid redundant computation by only recovering fragments of structure at D-structure on a demand basis. Since D- and S-structure are related by Move-α, any portion of the phrase structure at D-structure may be deduced from S-structure once the derivation sequence, or *history of movement*, has been computed. In general, given any S-structure constituent we can deduce the element that occupies the S-structure position at D-structure using the algorithm shown in figure 4.3. In the current implementation, this algorithm is explicitly invoked to compute individual D-structure constituents as required. Complete D-structures are never recovered. For example, given a partially instantiated S-structure, such as (4a), together with the facts that: (1) *who* originated from the position occupied by [*NP-e*] (in standard terminology, *who* 'heads' the (movement) *chain* (*who*, [*NP-e*])), and (2) there are no other chains in the S-structure (ignoring the movement of heads such as verbs for the moment); then we can infer that the corresponding phrase structure at D-structure must be of the form (4b).

  (4) a. [*CP* [*NP* who] [[*C* did] [*IP* [*NP* John] [*VP* [*V* like] [*NP*-e]]]]]
  b. [*CP* [*NP*-e] [[*C* did] [*IP* [*NP* John] [*VP* [*V* like] [*NP* who]]]]]

[6] Notation: Let $(X_1, X_2, \ldots, X_n)$ denote a non-trivial movement chain of length $n, n > 1$ with head X_1. The head X_1 originates at the position occupied by the last element of the chain X_n known as a *base trace*. Elements $X_2, \ldots, X_{n-1}$ are *intermediate traces*. Each adjacent pair (X_i, X_{i+1}), or *link*, in the chain denotes a single instance of movement originating at the position occupied by X_{i+1} and terminating at the position occupied by X_i.

[7] Note that this algorithm will also return an empty category if X_S is an adjunct in an adjunction structure formed by movement.

PS Rules at S-structure =

PS Rules at D-structure $\left\{ \begin{array}{l} \overline{\mathrm{X}}\text{-rules: e.g. } \overline{\overline{\mathbf{X}}} \rightarrow Spec, \overline{\mathbf{X}} \\ \mathbf{X} = N, Spec = Det \\ [\mathit{NP}\ [\mathit{Det\ the}\ ]\ [\bar{N}\ \mathit{destruction\ of\ the\ city}\ ]] \\ \\ \text{Empty category rules: e.g. } NP \rightarrow \lambda \\ [\mathit{IP}\ [\mathit{NP}\text{-}e\ ]\ [\mathit{VP\ was\ followed\ John}\ ]] \\ \\ \text{Adjunction rules: e.g. } VP \rightarrow VP, Adv \\ [\mathit{VP}\ [\mathit{VP\ was\ followed\ John}\ ]\ [\mathit{Adv\ why}\ ]] \end{array} \right.$

+

Rules to account for movement $\left\{ \begin{array}{l} \text{e.g. } CP \rightarrow Adv, \overline{C}; \qquad Adv \rightarrow \lambda \\ [\mathit{CP}\ [\mathit{Adv\ why}\ ]\ [\bar{C}\ \mathit{was\ John\ followed}\ ]] \end{array} \right.$

Figure 4.4 Components of the phrase structure grammar for S-structure.

4.4 Recovery of Phrase Structure at S-structure

In the previous section, we have described a scheme for recovering phrase structure at the various levels of representation. Here, we focus on the first stage: the recovery of phrase structure at S-structure from input SFs. In this section, we will deal with two main issues: (1) obtaining a grammar for PS at S-structure, and (2) selecting a practical and efficient algorithm for building structures that conform to this grammar.

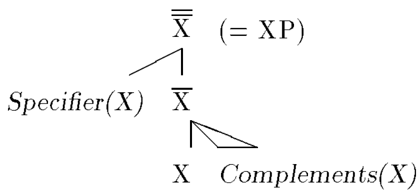
4.4.1 A Grammar of Phrase Structure at S-structure

In the generative model shown in figure 4.4, phrase structure at the level of S-structure is derived from corresponding phrase structure at D-structure through Move-α. How can we obtain a grammar for S-structure? Phrase structure at the level of D-structure is generated by rules that obey the principles of $\overline{\mathrm{X}}$-theory together with additional rules to cover cases not handled by $\overline{\mathrm{X}}$-theory, such as the introduction of empty categories and adjunction structures. We will first describe a grammar that covers the range of phrase structure at D-structure. Then we will consider the effect of Move-α on this grammar and revise the grammar accordingly to cover the range of phrase structure at S-structure.[8]

[8] 'Covering grammars' have also been employed for traditional transformational grammars. See Petrick [43], for example.

Phrase Structure Rules at D-structure

Phrase structure rules, with only a few exceptions, are defined according to the rules of $\overline{\mathrm{X}}$-theory. In $\overline{\mathrm{X}}$-theory, all phrases have the same basic internal structure. A typical *two-bar-level* prototype for a phrase might look something like the following:



Here, $\overline{\overline{\mathrm{X}}}$ represents the *maximal projection* that corresponds with the traditional notion of a complete phrase. For example, $\overline{\overline{\mathrm{N}}}$ and $\overline{\overline{\mathrm{V}}}$ denote a noun phrase (NP) and verb phrase (VP), respectively. The two other levels $\overline{\mathrm{X}}$ and X are known as the *intermediate projection* and *minimal projection*, or *head*, of the phrase. For example, the minimum projection of a noun or verb phrase will contain the noun or verb itself. The intermediate projection dominates both the head and the *complements* of the phrase. Typically, objects of a noun or verb will occupy the complement positions. Finally, the specifier position of a phrase may be filled by various elements such as subjects or determiners in the case of a noun, e.g. [*NP John's* [$\bar{N}$ [*N mother*]]] or [*NP the* [$\bar{N}$ [*N man*]]].

The relative order of the specifier and complements with respect to the head is parameterized according to different languages. For example, as we have described earlier, in Japanese heads will generally follow complements whereas the opposite is true for English. The $\overline{\mathrm{X}}$-prototype can also be extended to apply to sentences and full clauses:

1. In such theories, the maximal projection $\overline{\overline{\mathrm{I}}}$ (I for inflection), corresponds to the traditional category S (for sentence). The specifier position for I will be filled by the subject of the sentence, and its complement position by the main verb phrase. For example, *John saw Mary* will have the structure [*IP John* [$\bar{I}$ [*I*] [*VP saw Mary*]]]. Finally, it is assumed that the head of the sentence is occupied by inflection (I).

2. The prototype can be extended one level further to full clauses which are assumed to be headed by complementizers (sometimes abbreviated to Comp or C), e.g. *that* as in *that John saw Mary*. In this case, the complement position will be occupied by a sentence (IP), and the specifier position may be occupied by the usual clause-initial elements such as question operators. For example, *who did John see* might have the structure [*CP who* [$\bar{C}$ [*C did*] [*IP John see*]]].

```
head(n). head(v). head(a). head(p). head(i). head(c).
bar(n1). bar(v1). bar(a1). bar(p1). bar(i1). bar(c1).
max(np). max(vp). max(ap). max(pp). max(i2). max(c2).

proj(X,Y) :- head(X), bar(Y).
proj(X,Y) :- bar(X), max(Y).

spec(n1,np). spec(n1,det). spec(n1,[]).
spec(i1,np).
spec(c1,np). spec(c1,adv). spec(c1,[]).

noSpec(X1) :- \+ spec(X1,_).
noSpec(X1) :- spec(X1,[]).

compl(i,vp). compl(c,i2).
```

Figure 4.5 Predicates on category labels.[9]

Note that there are many variations of $\overline{\mathrm{X}}$-theory; for example, the specifier position of C may not be available in some versions, more bar levels might be used for certain categories such as verbs in others, and so on.

We will now turn to the problem of encoding the $\overline{\mathrm{X}}$-prototype. We will tackle this in two stages. First, we will define the categories that the prototype ranges over, together with fixed (as opposed to lexically determined) choices of specifiers and complements for appropriate categories. Then, we will define the grammar rules that represent the internal structure of the prototype.

We begin by defining a few simple predicates on category labels. Consider the definitions shown in figure 4.5. Here, the three one-place predicates `head/1`, `bar/1`, and `max/1` define a two bar-level scheme for nouns, verbs, adjectives, prepositions, inflection, and complementizers. The possible specifiers in each case are defined by the clauses of `spec/2`. Here, for example, we have allowed nouns to take a noun

[9] For convenience, we will repeat the description of PROLOG notation introduced in chapter 1:

1. Logical variables are distinguished from constants by the use of an uppercase letter as the initial character. The scope of a variable is a clause. Also, '_' is used to indicate an anonymous or 'don't care' variable. The scope of '_' is itself.

2. Elements enclosed in square brackets denote lists. In particular, `[]` denotes the empty list. `[Y]` denotes a list with one element `Y`, `[Y1,Y2]` a list of two elements, and so on. `[X|L]` denotes a list with first element `X` and with `L` representing the remainder of the list.

3. Outside of lists, (`,`) is used to denote conjunction. (`:-`) — to be read as 'if' — is used to denote implication and (`\+`) is used to denote negation as failure to prove.

```
rule XP -> [X1|specifiers(X1)] ordered specFinal st max(XP), proj(X1,XP).
rule specifiers(X1) -> [Y] st spec(X1,Y), \+ Y=[].
rule specifiers(X1) -> [] st noSpec(X1).

rule X1 -> [X|complements(X)] ordered headInitial st bar(X1), proj(X,X1).
rule complements(X) -> [] st lexicalProperty(X,grid(_,[])).
rule complements(X) -> [Y] st compl(X,Y).
rule complements(X) -> [Y] st lexicalProperty(X,subcat(Y$_,_),Y).
rule complements(X) -> [Y] st lexicalProperty(X,grid(_,[R]),csr(R,Y),Y).
rule complements(X) -> [Y1,Y2]
        st lexicalProperty(X,grid(_,[R1,R2]),csr(R1,R2,Y1,Y2),[Y1,Y2]).
```

Figure 4.6 $\overline{\text{X}}$-prototype rules.[10]

phrase (`np`), or a determiner (`det`) as a specifier, or have no specifier at all (denoted by `[]`). We have also specified that no specifier should be assumed as the default if a specifier is not mentioned in `spec/2`. This is accomplished by the first clause of `noSpec/2` which is intended to be read as: "`X1` may take no specifier if there is no specifier clause for `X1`." Finally, `compl/2` states that I and C take a verb phrase and sentence as a complement. (The possible complements of lexical categories such as verb and nouns will be specified through lexical entries as we shall see below.)

We are now in a position to define the $\overline{\text{X}}$-prototype. Consider the grammar rules shown in figure 4.6. The first group of rules defines the top half of the prototype. The first rule is intended to be read as: "`XP` derives `X1` followed by the specifiers of `X1` if `specFinal` holds," such that `XP` is a maximal projection and `X1` (immediately) projects to `XP`. (The `proj/2` relation was defined previously in figure 4.5.) Note that `specFinal` is a language parameter that will be defined to hold only for those languages for which specifiers come after heads and complements. If `specFinal` does not hold, then the order of the constituents will be reversed.[11] The next two rules state what the possible specifiers are:

[10]Note that some non-terminals such `complements(X)` and `specifiers(X)` are *pseudo*-non-terminals in the sense that they do not represent linguistic categories. For example, no constituents with a category label such as `Complements(c)` will be constructed. Although such non-terminals appear in the rule schema, they are eliminated when the rule schema is converted into the underlying LR-style tables used for parsing. (See section 4.5.3 for the description of this process.) More precisely, when the schema is grounded, all occurrences of pseudo-non-terminal symbols in the RHS of any rule will be replaced by the RHS of a rule that contains the relevant pseudo-non-terminal as its LHS.

[11]Actually, this can be replaced with just about any predicate. For example, we can write `specFinal(XP)` instead of just `specFinal` (and make the corresponding adjustments in the language parameters) to make the order sensitive to particular categories.

1. The first rule states that the specifier of `X1` will be `Y` provided `spec(X1,Y)` holds and Y is not `[]` (used to represent the null specifier).

2. The second rule states that `X1` may take no specifier (`[]`) if `noSpec(X1)` holds, i.e. when there is no specifier listed for `X1` or the null specifier itself is explicitly listed.

The second group in figure 4.6 defines the lower half of the prototype. As with `specFinal`, `headInitial` is a language parameter which will be defined to hold for languages like English, but not for Japanese (where heads generally follows complements). There are five choices for complements:

1. `[]`, i.e. no complement, provided `lexicalProperty(X,grid(_,[]))` holds. In general, `lexicalProperty(X,F)` is defined to hold only if there exists some element of category `X` with lexical property `F`. Now, we assume that nouns, verbs, adjectives and prepositions generally have θ-grids, that is a specification of the thematic roles associated with each particular element. For example, as we have described before, the verb *file* is associated with some 'filer' (represented by an 'agent' θ-role) and an object to be filed (represented by a 'theme' or 'patient' θ-role). We distinguish between *internal* and *external* roles which are realized as complements and specifiers, respectively. In the present system, this is indicated by having a verb like *file* have the lexical feature `grid([agent],[theme])`, the internal roles occupying the second argument of `grid`. Returning to the goal `lexicalProperty(X,grid(_,[]))`; this holds if there exists a lexical element of category `X` that takes no complements.

2. The second possibility is `[Y] st compl(X,Y)`, i.e. a single complement `Y` for category `X`. `compl/2` was defined previously in figure 4.5 for categories I and C.

3. The next possibility is `[Y]` if there is some element of category `X` that has lexical feature `subcat(Y$_,_)`. We will go into the details here except to mention that not all lexical subcategorization for complements will be done through θ-grids. Sometimes, it is more convenient to allow some elements, e.g. a verb, to have an object that it does not assign a θ-role to.

4. The fourth possibility is `[Y]` if there is some element of category `X` that selects one internal role, to be realized as an element of category `Y`. Here, we have adopted Chomsky's [12] Canonical Structural Realization (CSR) relation, represented by `csr/2`, that maps θ-roles to syntactic categories. For example, the role 'agent' will be realized as a `NP`.

5. The last case states that there may be two complements `[Y1,Y2]` that are also realized through the θ-grid.[12] For example, *persuade* selects a 'goal' and a 'proposition' as in *persuaded* [*NP John*] [*CP to leave*] *right away.* Note that the CSR relation may map a role into different syntactic categories. For example, a 'proposition' can be realized as an clause (`CP`) or a noun phrase (`NP`), as in *persuaded John of* [*NP the importance of leaving*].

(Note that the above rules will only accommodate lexical items with two or fewer objects (as defined through the θ-grid). If necessary, more rules can be added to handle three or more internal θ-roles.)

We will now turn to examples of other rules not included in the $\overline{\text{X}}$-prototype that are needed at D-structure:

- *Empty categories.*

  ```
  rule np with Features -> [] st ecNPFeatures(Features).% empty NP
  ```
  [I want [[*NP*-e] to win]] [*NP*-e]= *PRO (at S-structure)*
  [[*NP*-e]was arrested [*NP* John]] *Passive.*
  [you file which report without [[*NP*-e] reading [*NP*-e]]] *Parasitic gap.*
  ```
  rule c with Features -> [] st nullFeatures(Features). % empty Comp
  rule i with Features -> [] st nullFeatures(Features). % empty Infl
  ```

  The first rule allow the introduction of empty NPs. Such NPs may occur as empty subjects (embedded or otherwise) by the Extended Projection Principle (EPP) and as objects as required by thematic properties of individual predicates as shown above. Empty categories are also required at D-structure for the two non-lexical categories C and I. CP may be headed by an overt complementizer, as in ... *happy* [*CP* [*C that*] [*John left*]] or *happy* [*CP* [*C for*] [*John to leave*]], or be headed by an empty complementizer [*C*], as in [*CP* [*C*] [*IP John left*]]. In the current theory, at D-structure inflection is always empty, consisting of two components, AGR and TNS, implemented as syntactic features. For empty categories at D-structure, unlike their overt counterparts which have lexical entries (and thus 'pick up' their syntactic features from those entries), they must have features instantiated separately from the lexicon. In the empty category rules shown above, this is accomplished by predicates `ecNPFeatures/1` and `nullFeatures/1`.

- *Adjunction.*

[12] We include this last case to show that the representation can also handle theories that use non-binary branching. Actually, the present implementation does not make use of ternary branching. Instead, double objects are represented using binary branching with an extra bar level. The definitions shown here come from an earlier version of the theory used by the system.

```
adjunction rule vp -> [vp,adv].
adjunction rule vp -> [vp,pp].
adjunction rule np -> [np,c2].
```

(Non-movement) adjunction is required for relative clauses. We assume that the modifying clause will be right-adjoined (in English) to the 'head' noun phrase, following van Riemsdijk & Williams [46]. For example, *the man who I saw* will be assigned the following D-structure: [*NP* [*NP the man*] [*CP who* [*IP I saw*]]]. Similarly, adjunction of PPs to VPs is required to handle examples such as *John* [*VP* [*VP was arrested*] [*PP after leading the demonstration*]] and *John* [*VP* [*VP was arrested* [*PP by the police*]]]. Moreover, the object of *reading* in *which report did you file without reading* must occur in a VP adjunct for the empty NP to be properly licensed as a parasitic gap. (See pg.72ff in Lasnik & Uriagereka for the details.) Finally, adverbs such *often* and *completely*, are assumed to be adjoined to VPs at D-structure; for example, *John* [*VP often* [*VP sees ghosts*]]. Following Lasnik & Uriagereka, *wh*-adverbs such as *why* are also initially adjoined to a VP at D-structure. However, they will be subsequently fronted to a clause-initial position either at LF or S-structure. For example, the D-structure representation for *why did John leave* is assumed to have the form [*John* [*VP* [*Adv why*] [*VP leave*]]].

Phrase Structure Rules at S-structure

In the previous section, we have described a grammar for phrase structure at D-structure. By considering the effect of Move-α on this grammar, we will obtain the corresponding grammar for S-structure. Unfortunately, because this process has not been automated in the current system, the burden of specifying the additional phrase structures rules will fall on the grammar writer.[13]

The general process of Move-α allows any constituent to move anywhere in a given structure; the idea being that all instances of improper movement will be filtered out by some general principle. However, as Lasnik & Uriagereka note (pg.113), this is largely a promissory note that has not been made to work in full generality. To avoid generating structures that will not be relevant to any of the principles that have been implemented, the scope of Move-α has been restricted to cases of head movement, adjunct fronting and general movement of NPs; these being the 'core' cases handled by the principles of Case and θ- theory, and the

[13] However, the consequences of Move-α that are described in this section could be automatically derived from a description of the categories that move, the possible landing sites and whether substitution or adjunction (or both) are allowed in each case — although, given the somewhat non-regular nature of the non-$\overline{\mathrm{X}}$ phrase structure rules described in this section, it is not clear that such a system would offer much in the way of abstraction over the current approach.

Empty Category Principle (ECP). One consequence of this restriction is that the S-structure grammar must be updated if other instances of Move-α such as movement of intermediate projections or VPs are to be covered. With this caveat, we now enumerate those core cases:

- *Movement of noun phrases.*

  Movement of NPs generally fall into four categories distinguished by characteristics of the *landing* and *launching* sites. Such sites may be classified either as being A-positions, i.e. positions to which thematic roles may be assigned, such as object and subject positions; or as $\overline{\mathrm{A}}$-positions such as the specifier position of a complementizer. For example, passive or raising examples such as *John seems to be happy* are one instance of A-to-A-position movement. Simple *wh*-movement examples such as *what did you see* fall under the category of A-to-$\overline{\mathrm{A}}$-movement. Movement of *wh*-adverbs as in the example *why did John leave*, as described in the previous section, is an instance of $\overline{\mathrm{A}}$-to-$\overline{\mathrm{A}}$-movement. It seems difficult to construct a grammatical example of $\overline{\mathrm{A}}$-to-A-movement, the only remaining case, in the current theory.[14] In the current implementation, traces of movement are just instances of the generic empty NP ' [*NP*-e]'. If we assume that can be no adjunction to Comp at S-structure, e.g. as in Lasnik & Saito [34], then the cases we have described can be covered by the existing D-structure rules because all relevant A-positions (subject and object positions) and specifier of Comp already permit empty NPs.

- *Scrambling.*

  ```
  VP -> NP VP % adjunction to VP
  IP -> NP IP % adjunction to IP
  ```

  Although there is seems to be no need for adjunction in Comp at S-structure (as Lasnik & Saito indicate), adjunction of NPs at VP and IP is commonly assumed (see Hoji [29] for example) for cases of scrambling in languages such as Japanese. We repeat here the four examples augmented with traces for the Japanese equivalent of *John gave Mary a book* (shown previously as (8) in chapter 1):

[14] Of course, it would be wrong to rule out instances of $\overline{\mathrm{A}}$-to-A-movement on this basis. The phrase structure recovery routine should allow examples of structures with $\overline{\mathrm{A}}$-to-A-movement and not preempt the constraints imposed by independent principles. For example, Lasnik & Uriagereka (pgs. 98–100) discuss examples of 'Super-raising' such as **John seems that it is likely to leave* where *John* originates from the subject position of predicate *leave*. In a model of phrase structure that allows Comp to have two landing sites available, namely, specifier of Comp and Comp itself; then such a model can generate an S-structure with *John* moving from the specifier position of Comp (an $\overline{\mathrm{A}}$-position) to the matrix subject position (an A-position).

(5) a. [*IP* John-ga [*VP* Mary-ni hon-o ageta]]
b. [*IP* hon$_i$-o [*IP* John-ga [*VP* Mary-ni t_i ageta]]]
c. [*IP* Mary$_i$-ni [*IP* John-ga [*VP* t_i hon-o ageta]]]
d. [*IP* John-ga [*VP* hon$_i$-o [*VP* Mary-ni t_i ageta]]]

- *Head movement.*

```
adjunction rule i(IV) -> [v(V)] st raiseVtoI(V,_,IV).
adjunction rule v(VI) -> [v(V)] st lowerItoV(V,_,VI).

rule v -> trace.
rule i -> trace.

adjunction rule c(CIV) -> [v(V)] st raiseVtoC(V,_,CIV).
```

In the current implementation, a simple model of verbal inflection has been adopted. Depending on certain properties of the agreement component of inflection (Infl) and the verb in question, either the verb will raise to adjoin to Infl, or Infl will lower to the verb. (The details of the mechanism — encoded by predicates `raiseVto/3` and `lowerItoV/3` — are unimportant in the current context.) The distinction between the two cases may be seen in English from the order of VP-adverbs (assumed in the previous section to be left adjoined to VPs at D-structure) with auxiliary and 'proper' verbs, as in *John completely lost his mind* (Infl lowers to V) and *John has completely lost his mind* (V raises to Infl). These two cases are covered by the first four rules shown above. The last rule is provided to cover cases of 'subject-verb-inversion', as in (6a), and *Do*-support, as in (6b):

(6) a. [*CP* [*C* [*V* is]$_i$ [*C*]] [*IP* John [*VP* [*V*-t]$_i$ here]]]
b. [*CP* what [[*C* [*C*] [*I* [*I*]$_j$ [*V* do]$_i$]] [*IP* you [*I*-t]$_j$ [*VP* [*V*-t]$_i$ [*VP* see]]]]]

- *Movement of Adverbs.*

```
rule adv -> trace
```

Finally, rules must also be added to handle cases in English of adjunct movement at S-structure such as the fronting of wh-adverbs such as *why* as in *why did John move.* The above rule allows the introduction of empty adverbs only as traces of movement. In the current model, these cases are accomodated by allowing adverbs to occupy the specifier position of Comp at S-structure (the head position being reserved for zero-level projections).

Note that the actual phrase structure grammar to be constructed for S-structure will be language-dependent. For example, the scrambling rules are not used in English, and the rules allowing movement of *wh*-noun phrases and *wh*-adverbs will not be present in Japanese (since all such movement can only take place at LF in such languages). To provide some idea of the number of rules involved, the following table summarizes the situation for the phrase structure grammars used for the two languages:

Language	$\overline{\text{X}}$-prototype rules	Empty category rules	Adjunction rules	Others
English	12	5	8	6
Japanese	(same)	4	6	2

By any measure, the number of rules (around thirty or so) can be considered to be small and, therefore, quite manageable.

In the next section, we will turn our attention to the efficient recovery of phrase structure representations that conform to the rules defined above.

4.4.2 Canonical LR(1)-Based Phrase Structure Recovery

The rule schema given in the previous section expands into a context-free grammar (CFG).[15] Therefore, to recover structures at the level of S-structure, we can take advantage of any one of the rich variety of different parsing mechanisms that have been developed to parse languages generated by CFGs or specific subsets of the context-free languages. Among these are the general tabular methods such as the Cocke-Kasami-Younger (CKY), Earley's, and the Graham-Harrison-Ruzzo algorithms, or the restricted table-driven shift-reduce methods such as the LR(k) algorithms (to be described below). There are also many non-tabular methods, for example, another class of algorithms are based on the formal equivalence between CFGs and push-down automata. Yet another possibility is to simply interpret CF rules in a top-down (or recursive descent) fashion. Tomita [56] has claimed that the LR-style shift-reduce machines are more efficient than any other existing parsing algorithm *in practice*.[16] This is the method that we have adopted in the current

[15] Obviously, since each schema rule is of the form $A \rightarrow B_1, \ldots, B_n$ (n bounded), and this form is preserved under simple substitution of $\overline{\text{X}}$-categories for variables.

[16] Tomita principally compares the efficiency of Earley's algorithm to his extended-LR algorithm. Conceptually, Earley's algorithm and the shift-reduce algorithms, e.g. the LR algorithm extended to handle ambiguity, are very similar. Essentially, both algorithms manipulate configurations (sets of dotted productions) in similar ways. Nevertheless, Tomita shows that, in practice, LR-style parsing is generally more efficient than Earley's algorithm despite inferior worst-case complexity results. The reason for this is that LR-style parsing can be viewed as a *compiled* counterpart to Earley's algorithm. The essential difference is that all possible configurations are pre-computed when the LR machine is built off-line, whereas Earley's algorithm has the additional burden of computing and storing configurations during parse time.

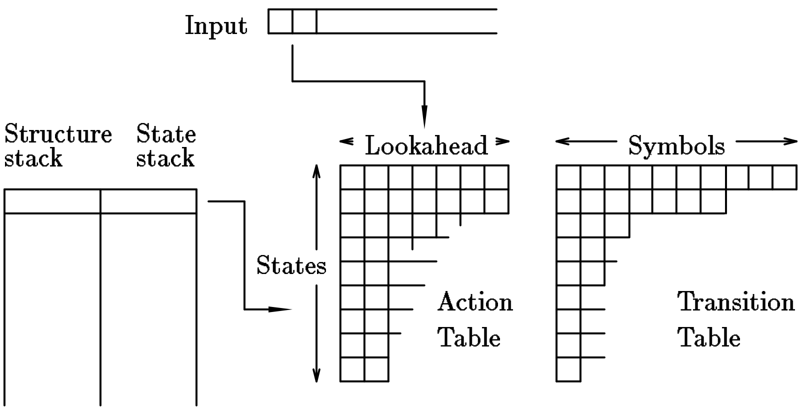


Figure 4.7 The components of a shift-reduce parser.

implementation.

Roadmap

This section is divided into two parts. In the first part, we will review the components of shift-reduce parsers in general, and the properties of LR-style parsing in particular. Then, we will discuss the problems of adapting LR-style parsing to the phrase structure grammar of S-structure described in the previous section.

Shift-Reduce Parsing

As is standard in the literature, a shift-reduce parser is a bottom-up parser that consists of a control and structure stack (sometimes combined); an input sequence, a prefix of which may form a limited lookahead buffer; and a table with an action and a transition component, as shown in figure 4.7. The structure stack holds terminal and non-terminal symbols that have already been recognized. The lookahead buffer contains a fixed number of input symbols (possibly zero depending on the parser) that have not yet been parsed. Based on the contents of the lookhead and the current state held on the top of the control stack, the action component of the table tells the parser what action to take next. There are four types of actions: (1) *Shift*: to recognize the first terminal symbol of the input by popping that symbol off the input onto the structure stack; or (2) *Reduce*: to recognize a non-terminal symbol A from a rule $A \rightarrow A_1 \ldots A_n$ by replacing the top n constituents of the structure stack (corresponding to $A_1 \ldots A_n$) with a (new) single non-terminal of category A and the n elements popped off the structure stack as immediate constituents. Additionally, the parser can: (3) *Accept*: signal that the sentential symbol has been recognized;

that is, a successful parse has been found; or (4) signal that an impossible, or 'error', state has been reached; that is, the sentential symbol cannot yield the given input. After performing the appropriate action, the transition component of the table tells the parser what state to go to next on the basis of the current state and the non-terminal symbol (after a reduce action) or terminal symbol (after a shift action) recognized. (For examples of shift-reduce parsing, the reader is referred to an introductory compiler text such as Aho & Ullman [7].)

The concept of LR(k) parsing, introduced by Knuth [33], is a class of *deterministic* shift-reduce parsers parameterized by k, the number of symbols of terminal lookahead. LR(k) parsers have many interesting properties; we briefly summarize a few of the most relevant ones below:

1. LR(k) parsers are of interest because they are the most powerful class of deterministic bottom-up parsers using up to k symbols of lookahead. This means that if a grammar G cam be parsed (deterministically) by *any* bottom-up parser using k symbols of lookahead; then an LR(k) parser can be constructed for G. However, not all CF grammar are LR(k) grammars.[17] The determinism requirement means that a LR(k) parser must be able to uniquely determine which action to take in any state given any lookahead. Equivalently, there may be no *conflicting* actions; instead, each action entry must contain a unique action.

2. Whatever the state of the parser, the contents of the structure stack will constitute a prefix of a *right sentential form*. (A *sentential form* for a grammar G with distinguished symbol S is any string β such that $S \Rightarrow^{\star} \beta$; that is, β is derived from S in zero or more steps, each step being the expansion of a non-terminal by some production in the grammar. If the rightmost non-terminal is always expanded, then β is a *right sentential form*.) Now, all LR-style parsers will admit only valid prefixes. An erroneous prefix, i.e. a prefix that cannot be part of a sentential form, will be detected as soon an attempt is made to shift an erroneous input symbol.

3. However, the size of LR(k) tables has proved to be a practical limitation on the size of grammars that can be handled and the number of symbols of lookahead that can be used. In fact, k is usually limited to one or zero. Even with this restriction, canonical LR(1) parsers are rarely used in the domain of (non-natural) programming languages. Instead, for practical LR-style parsing, parsers based on (sub-) classes such as the SLR(1) (*simple*-LR) and LALR(1) (*lookahead*-LR) parsers that trade power for reduced table size

[17] For example, no ambiguous CF grammar can be an LR(k) grammar for any k due to the deterministic action restriction.

are most commonly used.[18] However, one disadvantage with such a tradeoff is that these less-powerful parsers may do more work than strictly necessary. More precisely, in the case of parsing invalid prefixes, they may perform more reduce actions, i.e. build more constituents, than strictly necessary. The canonical LR(1) parser is *optimal* in the sense that it provides the earliest possible error detection, thereby reducing the amount of unnecessary work, given one symbol of lookahead.[19] (See the discussion of LR-parsing in Fischer & LeBlanc [21] for an explanation.)

In the interest of bringing the most powerful machinery to bear on the phrase structure recovery problem, the canonical LR(1) parser was selected as the target machine for the recovery of structure at the level of S-structure. Table size was a secondary consideration. However, as section 4.5.3 will describe, table size turned out not be a limiting factor for the grammar introduced in the previous section. In the next section, we will discuss what changes have to be made to the LR(1) algorithm to accommodate the grammar for phrase structure at S-structure.

Modifying the LR(1) Parser

The current implementation adopts the basic model of LR(1) shift-reduce parsing described in the previous section. However, some modifications to the basic model are necessary in order to handle the problems posed by the presence of empty categories in the S-structure grammar:

1. Although the S-structure rule schema expands into a CFG, the CFG is not an LR(1) grammar. The main reason for this is the presence of ambiguity (both lexical and structural), as discussed at the beginning of this chapter. At S-structure, the presence of movement traces will further exacerbate this problem:

   - *Shift-reduce conflicts.*

     For example, consider the choice that a LR(1) parser faces in a state where a verb may be expected and a verb occupies the first element of the input sequence.[20] In operational terms, the possibility of head movement (described in the previous section) creates a situation where the parser can either shift the verb at the head of the input, or reduce

[18] For example, YACC, perhaps the best-known LR-style parser-generator, is an LALR(1)-based parser-generator.

[19] Note that the only difference between these parsers is in the number of extraneous reductions before an error is detected. In particular, as stated above, all LR-style parsers never 'over-shift'.

[20] That is, a state where the sentential prefix may be augmented with a verb without creating an invalid prefix. More precisely, there will exist a dotted rule with a verb in its lookahead set in the configuration set corresponding to the current state.

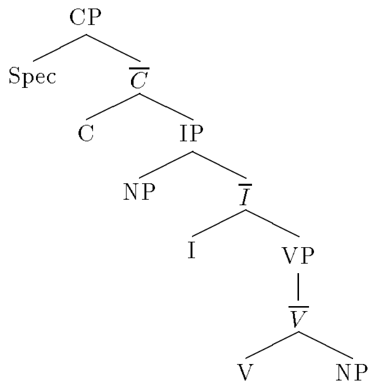


Figure 4.8 Basic clausal structure.

to create a verb trace (the expected verb may have moved elsewhere, as in *John* [*I* [*V has*] [*I*]] [*VP*• [*V-t*] [*V lost*] *his mind*], in which (•) marks the point reached by the parser and *lost* is the verb that heads the input). In standard terminology, this situation is known as a *shift-reduce conflict*.

- *Reduce-reduce conflicts.*

  Movement can also produce *reduce-reduce conflicts*. For example, suppose the parser has just built a verb phrase, say [*VP* [*V* [*I*] *lost*] [*NP his mind*]], as in *John lost his mind*. Moreover, for simplicity, assume that all input has been shifted. In this case, the parser can go on to complete a $\bar{\mathrm{I}}$, i.e. [$\bar{I}$ [*I-t*] [*VP* [*V* [*I*] *lost*] [*NP his mind*]]], by performing a reduce action using a production of the form $\bar{I} \rightarrow I, VP$. But despite the lack of further input, this is not always the correct decision. For instance, it might be necessary to add further material to the VP, e.g. an adverbial trace, resulting in [*VP* [*VP* [*V* [*I*] *lost*] [*NP his mind*]] [*Adv-t*]] as in *why has John lost his mind*. In this case, the correct decision will be to reduce using a production of the form $Adv \rightarrow \lambda$.

Hence, strict determinism must be relaxed if the LR(1) algorithm is to handle such conflicts. This is handled in the current implementation, by permitting each action table entry to hold an arbitrary number of actions. As a result, the parser can handle arbitrary CFGs.

2. Parsing termination is a classic problem posed by the presence of empty categories at S-structure. Consider the typical skeletal form of S-structure as

shown in figure 4.8. Now, each 'leaf' of this tree may derive the empty string. As discussed previously, the rule schema contains a production of the form $NP \rightarrow \lambda$ to account for NP-traces and *PRO*. From the head movement rules, both V and I may be realized as traces. Furthermore, C may be occupied by an empty complementizer. Finally, in the current theory, the specifier position of C need not be filled at all. Therefore, we can deduce that CP $\Rightarrow^\star \lambda$, i.e. full clauses may be empty. Hence, the parser will be able to build full clauses without shifting a single symbol. Morever, clauses may contain other clauses because some heads (such as the verb *believe*) may select a proposition that may be syntactically realized as a CP. The recursion may be also be slightly less direct. For example, a noun phrase may contain a relative clause which will also be realized as a CP. Taken together, the ability to recursively predict a CP without needing to consume any input will cause the parser not to terminate.

There are many ways to block infinite recursion through CP. For example, one can impose a depth bound on the structure stack. The solution that was adopted allows an arbitrary number of CPs to be predicted, but blocks infinite recursion by requiring each CP to consume some input. Basically, the parser uses a simple condition on traces to ensure that each trace is 'licensed' in the sense that a trace cannot exist without an original overt constituent of the same category being present somewhere in phrase structure. (Note that these conditions are not imposed for empty noun phrases. As described earlier, different conditions involving the θ-grid and the requirement that subjects be present are used for noun phrases.) To ensure termination, it will suffice to apply the restriction to cases of head movement only. From the previous section, we have that verbs can adjoin to Infl, as in (7a), or vice-versa, as in (7b):

(7) a. John [[*I* [*I*] [*V* has]$_i$] [*VP* completely [*V*-t]$_i$ lost his mind]]
    b. John [[*I*-t]$_i$ [*VP* completely [*V* [*V* lost] [*I*]$_i$] his mind]]

Raising the resulting Infl-verb complex to Comp is a further optional step to handle the cases of subject-verb-inversion and *do*-support discussed previously. Now consider the head movement rules from pg. 137 (repeated below):

```
adjunction rule i(IV) -> [v(V)] st raiseVtoI(V,_,IV).
adjunction rule v(VI) -> [v(V)] st lowerItoV(V,_,VI).

rule v -> trace.
rule i -> trace.

adjunction rule c(CIV) -> [v(V)] st raiseVtoC(V,_,CIV).
```

If a verb is adjoined to Infl; then, obviously, there must be a corresponding verb trace present, assuming that no traces may be deleted at S-structure. Similarly, there must be an Infl trace if Infl is adjoined to the verb. Finally, if the verb is adjoined to Comp; there must be both a Infl and a verb trace present elsewhere in phrase structure. That is, each of the three adjunction rules will license Infl or verb traces (or both). Then, let a rule of the form `X -> trace` have the special interpretation: "Generate an empty category `X` only if $[X\text{-}t\,]$ is licensed." (Of course, this licensing restriction can only work if this is the only way an empty verb can be generated; that is, there are no other productions of the form `V ->[]`.) Consequently, all CPs must now contain an overt verbal element. Infinite recursion through CP is now impossible since CP $\not\Rightarrow^{\star} \lambda$.

To enforce this restriction, the LR(1) machine model is slightly modified as follows:

(a) Production rules may communicate by placing constituents on a new environment stack that functions as a 'scratchpad'. For example, a rule such as `i(IV) -> [v(V)] st raiseVtoI(V,_IV)` that licenses a verb trace will, in addition to building an Infl constituent (with a verb adjoined), have the side-effect of placing a verb (complete with lexical features) on the environment stack.

(b) A production that introduces traces such as `v -> trace` will build a verb trace (and also pick up its lexical features) only if a verb has been previously placed on the environment stack.

(c) So far, we have assumed that all traces will have been predicted ahead of time (in a left-to-right manner). Of course, when an occurrence of a trace precedes the relevant head-adjunction structure, as in the case of Infl lowering to V in English, then the production that introduces the traces, `i -> trace` in this case, must look ahead (possibly many terminal symbols ahead) to the input buffer for licensing. (For head-final languages like Japanese, this must happen for the case when a verb raises to Infl.) The current system contains an automatic mechanism that computes the relative order of the elements Infl, V and Comp (for a given language) and tells the trace licensing mechanism to look either to the environment stack, or to the input buffer, or even (non-deterministically) to either side for an appropriate licensing head-adjunction structure.

This mechanism is also used to eliminate another source of infinite recursion. For example, consider the sentence *how did John lose his mind*. Here, *why* is adjoined to VP at D-structure, using a production of the form $VP \rightarrow VP, Adv$, resulting in the structure $[VP\ [VP\ lose\ his\ mind\,]\ [Adv\ how\,]]$. At

S-structure, *how* will move to Comp leaving behind a trace. As described in section 4.4.1, the trace is generated by a rule of the form `adv -> trace`. Hence, by using the same licensing restriction as for head movement, we can prevent the generation of an infinite class of structures of the form [*VP* [*VP*... [*VP* [*VP lost his mind*] [*Adv-t*]]... [*Adv-t*]] [*Adv-t*]] containing an arbitrary number of adjunctions.[21]

Having described the basic components of the modified LR(1) parser, in the next section, we will turn our attention to the efficient implementation of the parser, including further modifications of the LR(1) model to improve its performance, and the automatic compilation of the S-structure rule schema down to the level of the underlying machine.

4.5 Implementation and Compilation

So far we have discussed the rationale for adopting a multi-stage approach to recovering the levels of phrase structure from the point of view of computational efficiency; and, in particular, the adoption of a modified LR(1) algorithm for recovering the first level at S-structure. In this section, we will briefy describe the implementation, noting, where relevant, those features of the underlying model that affect how the parser builds phrase structure. We will also note further modifications of the scheme described so far that have been incorporated to further improve its efficiency. Finally, we will briefly discuss the method by which the S-structure rule schema may be turned into the tabular form required by the underlying LR(1)-based model.

4.5.1 Basic Implementation

The modified LR(1) machine introduced in the previous section contains the usual LR(1) action and transition tables, albeit slightly modified to allow entries to hold multiple actions; plus the three internal stacks that are used to manipulate machine states, to hold partially constructed constituents, and to facilitate the communication of contextual information. As with the regular linguistic principles discussed in chapter 3, the machine is represented as PROLOG predicates to be interpreted by the standard top-down, left-to-right, depth-first search mechanism used by PROLOG. In the following discussion, we will briefly mention how each of these components are

[21] Currently, the system is set up to allow only one iteration of `adv -> trace` to apply in any one 'context', or single state of the CFSM (Characteristic Finite State Machine) associated with the LR(1) machine. In other words, the infinite recursion will be terminated after the first iteration. This parameter can be varied for individual categories by the grammar writer on a state-by-state or global basis.

realized; we will also point out those salient features that will prove to be useful when the interleaved application of principles is considered in chapter 6.

- *The representation of the transition table.*

  In the current implementation, the transition and action tables are represented separately. Only the action table need depart from the standard LR(1) model to accommodate non-determinism. The transition table δ that tells the parser what state S' to go to next following a reduction or shift operation, based on the current state S and the non-terminal or terminal symbol V, i.e. $\delta : S \times V \mapsto S'$, remains deterministic. δ is represented by a predicate `transition` of arity three such that each clause `transition(`S`,`V`,`S'`)` holds iff $\delta(S, V) = S'$.

- *The representation of the action table.*

  All actions will alter the machine configuration, i.e. the control stack (CS), the structure stack (SS) and the environment stack (ES) plus the input sequence (I). Hence, actions may be regarded as instances of a prototype $f : \mathrm{CS} \times \mathrm{SS} \times \mathrm{ES} \times \mathrm{I} \mapsto \mathrm{CS}' \times \mathrm{SS}' \times \mathrm{ES}' \times \mathrm{I}'$. (The control and structure stacks have been separated so they need not operate in 'lockstep' for reasons that will become clear later.) Hence, we can represent f as a predicate `action` of arity ten such that each clause `action(`S`,`V`,CS,SS,ES,I,CS',SS',ES',I')` represents a single LR action in state S given lookahead symbol V.[22] As an example, suppose both shift and reduce (using the production $\bar{A} \rightarrow A, NP$) are possible actions in a state S_1 with lookahead N. Then, the action table will contain the following two clauses:

  1. Shift:

     `action(`S_1`,n,CS,SS,ES,[I1|I],[`S_2`|CS],[I1|SS],ES,I).`[23]

     where `I1` represents the noun to be shifted from the input to the structure stack (SS), and $\delta(S_1, N) = S_2$ holds.

  2. Reduce:

     `action(`S_1`,n,[`S_1`,`S_2`,`S_3`|CS],[NP,A|SS],ES,I,`
     `[`S_4`,`S_3`|CS], [A1|SS], ES,I).`

     where `A1` represents a constituent of category $\overline{\mathrm{A}}$ with immediate constituents `A` and `NP`, and $\delta(S_3, \bar{A}) = S_4$ holds.

[22] `action` is of arity ten rather than eight for efficiency reasons. Strictly speaking, the first two arguments are redundant since the current state is the same as the top of the control stack, and the single lookahead symbol is the category of the head of the input. The extra two arguments were added to take advantage of clause indexing under multiple arguments, which facilitates almost constant time access to individual clauses in the version of PROLOG used.

[23] We have represented all stacks and the input sequence as PROLOG lists.

The representation of the LR tables has the following interesting features:

- *No explicit error entries.*

  Note that there are no restrictions on the number of actions per table entry. Each possible action is represented as a separate clause. Furthermore, only shift, reduce and accept actions are explicitly represented by clauses. The action table is sparse in the sense that error entries are deliberately omitted. If there are no matching clauses (actions) for a given state and lookahead; then the parser will assume an error entry by default. Both conflicts and implicit error entries are automatically handled by the backtracking search mechanism used by PROLOG. If there is more than one matching clause for a given state and lookahead, as is the case when conflicts exist, then the interpreter will pick the first clause it encounters and leave a choice point so that it can try the other possibilities when it backtracks later. If there are no matching clauses, as is the case with an implicit error entry, then the interpreter will immediately backtrack to a previous choice point (if one exists) to search for other solutions. By forcing the interpreter to explore all choice points, the S-structure parser can find all possible parses.

- *Pre-computing the transition function.*

  For both reduce and shift actions, updating the control stack involves computing the transition function δ. In some cases, it is possible to eliminate the access to the transition table by computing the next state off-line. For example, consider the clause shown above for the shift operation. The current state and the category of the symbol to be shifted is already known when the action clause is constructed. Since these are the input arguments to δ, and δ is a (deterministic) function, this information suffices to determine the next state. Of course, the next state cannot always be computed ahead of time. For example, consider the clause shown above for the reduction of $\bar{A} \rightarrow A, NP$. Here, the top of the control stack, i.e. the current state S_1, and the lookahead symbol `n` are both known (off-line). Note that δ operates on the third element from the top of the control stack, S_3.[24] However, the value of the S_3 is not known at compile time. Hence the transition table must be consulted at parse time. In the current implementation, a partial evaluation mechanism is used to pre-compute the next stage whenever possible.[25]

[24] There are two non-terminals on the RHS of the rule for $\bar{A}$. The completion of each non-terminal will result in the addition of a single state to the control stack. Hence the reduce action must first pop off these two elements to restore the state that corresponds to the completion of $\bar{A}$.

[25] Whilst this cannot result in a large improvement in the overall efficiency of the LR(1) machine — after all, it is an instance of 'simple unfolding' (in the sense of section 2.6) — it is a fairly safe procedure in the sense that it is unlikely that there will be many situations in which the parser

- *Adding extra pre-conditions.*

  Since actions are represented as PROLOG clauses, table entries need not be limited to unit clauses only. In general, additional constraints can be placed on actions by introducing goals on the RHS of unit clauses. For example, this is how the trace licensing mechanism described in the previous section is implemented. Consider the following reduce action generated for a production `v -> trace`:

  ```
  action(116,n,CS,SS,ES,I,[55|CS],[[V-t]|SS],ES',I) :-
  predicted(v,ES,ES').
  ```

  This clause states that in state 116 with a noun as the lookahead symbol, the parser should build a new constituent, the verb trace [*V-t*], on the structure stack and proceed to state 55, *provided* a verb has been predicted on the environment stack.

  The option of adding extra conditions to action clauses is also exploited for the principle interleaving mechanism to be described in chapter 6. Basically, principle interleaving works by applying the constraints specified by linguistic principles to portions of phrase structure as they are built up in a bottom-up fashion using the S-structure parser described here. The essential idea will be to insert PROLOG goals (that call various constraints) in those reduce action clauses that build constituents on which those constraints apply. For example, if some constraint realized as a predicate P operates only on VPs and NPs; then for all clauses that reduce VPs or NPs, i.e. every time a VP or NP constituent is built, we should insert a goal $P(tos(\texttt{SS}'))$, i.e. apply P to the new constituent on the top of the structure stack.

4.5.2 Further Modifications

So far the major modifications to the basic LR(1) algorithm have been motivated by the need to handle specific properties of the S-structure grammar. We now turn to other (non-essential) modifications that have been tried purely for efficiency reasons, as opposed for coverage. One of these, namely, the elimination of the need to consult the transition table in some cases, has already been discussed. We now briefly mention modifications that have been designed to reduce the effect of allowing conflicting actions; that is, to reduce the degree of non-determinism present in the action table:

---

will do more work than before. To provide a rough idea of the applicability of the procedure: calls to the transition table were resolved off-line in 61% and 40% of the 895 and 765 LR(1) action table entries for English and Japanese, respectively.

- *Unbounded lookahead.*

  The basic motivation for the trace licensing mechanism was to prevent the parser from hypothesizing empty clauses which would lead to non-termination problems. By controlling the generation of certain empty categories, in particular, verb traces; all attempts to generate empty clauses were bound to fail sooner or later. But, from examining the behaviour of the algorithm, it was found that a considerable amount of unnecessary work, i.e. in building partial clauses, was performed by the parser. For example, relative clauses are analyzed as clausal adjuncts introduced by a production of the form `np -> [np,c2]`. Therefore, each time a noun phrase was constructed, the parser would attempt to build a relative clause even if there was no following input. Sometimes, it would (incorrectly) consume overt input elements intended for the matrix clause. For example, consider the sentence *John visited Mary*. Once the NP corresponding to *John* had been built, the parser would attempt to build a (relative) clause in the following ways:[26]

  1. Using no further input, as in [*NP* [*NP John*] [*CP*]] *visited Mary*.
  2. Using *visited* as part of the relative clause, as in [*NP* [*NP John*] [*CP (who) visited* [*NP-e*]]] *Mary* ....
  3. Using *visited Mary* as part of the relative clause, as in [*NP* [*NP John*] [*CP (who) visited Mary*]]....

  All three attempts will eventually fail (the first one because the relative clause is empty, and the other two because the matrix clause will have no overt verbal element). However, the parser will have performed many shifts and reductions, all of which are wasted because none of the three possibilities will lead to a complete parse. Given an arbitrarily long lookahead, a parser can, of course, detect before a single shift or reduce is performed (unnecessarily) that generating a relative clause for *John* is futile. Unfortunately, the parser described so far has only one symbol of lookahead to go on. Furthermore, uniformly extending the LR table to handle two or more symbols of lookahead would be prohibitively expensive. (Roughly speaking, each additional symbol would increase the number of table entries by a factor of $|V_T|$, the number of terminal symbols.) Hence, a solution was adopted that kept the LR table size almost constant, but allowed actions to inspect the input.

  Basically, we would like to be able to say something of the form:

[26]Note that we cannot get around this problem simply by adding a requirement of the form: "All relative clauses must have a relative pronoun". This would be an incorrect generalization. Ellipsis of the relative pronoun is permitted in many circumstances. For example, both *the man who I saw* and *the man I saw* are well-formed relative clauses.

In schema $\overline{\overline{\mathbf{X}}} \rightarrow \{$ *Specifiers*($\mathbf{X}$), $\overline{\mathbf{X}}$ $\}$,
if $\mathbf{X}$=C, then first check the input for an available verbal element.

The essential idea will be to impose this condition by adding a PROLOG goal to the relevant action clauses. Note that it will be possible for the extra goal to access the input sequence since it is one of the ten arguments to `action`. In the expanded grammar, the above stipulation is equivalent to generating a production of the form:

$CP \rightarrow$ `checkInput(I)`, $Spec$, $\overline{C}$

where `checkInput` is not a non-terminal, but a pre-condition of the production. `checkInput(I)` will be designed to succeed only if a verbal element in `I` (the input) is available. If no such element is available; then the production may not apply. Unfortunately, as it stands, the above production cannot be directly used by the LR machine. Attaching the condition to either a shift or reduce clause does not provide the desired behaviour because we want to check the input *before* predicting a CP, i.e. before a single terminal for the CP is shifted. Attaching the condition to the reduce CP clause would result in the condition being applied only *after* the entire RHS has been recognized; that is, only after $\overline{\mathrm{C}}$ has been reduced. The solution is to simply introduce a (new) dummy non-terminal at the point where the `checkInput` is to be invoked. Next, we augment the grammar by adding a dummy production that causes the LR machine to neither build structure nor read any input; but simply to have the side-effect of applying the desired condition. Hence, we generate the following two replacment productions:

$CP \rightarrow Dummy$, $Spec$, $\overline{C}$
$Dummy \rightarrow \lambda$ if `checkInput(I)`

The latter production will result in a reduce action of the form:

```
action(S1,V,CS,SS,ES,I,[S2|CS],SS,ES,I) :-
        checkInput(I).
```