Chapter 8

Parsing schemata for unification grammars

The last decade has witnessed an overwhelming amount of different, but related unification grammar formalisms. Our informal introduction in Chapter 7 was based on PATR [Shieber, 1986], which is the smallest and simplest of these formalisms. Unlike formalisms as LFG [Kaplan and Bresnan, 1982], GPSG [Gazdar et al., 1985] or HPSG [Pollard and Sag, 1987], PATR was not primarily designed to capture some universal linguistic structure, but merely as a small, clean formalism that covers the essential properties found in most other unification grammars.

The logical foundations of constraint-based formalisms have been discussed by Kaspar and Rounds [1986], Smolka [1989, 1992] and Johnson [1991], who give various axiomatizations of feature structures in predicate logic. In such a logical approach, one describes a *constraint language* in which constraints can be expressed. Such constraints are formulae in first-order logic with equality. Constraints state that certain features must have certain values or be equal to certain other features. The semantic interpretation of such a formula (following Smolka) is a *feature graph*. The most interesting property is *satisfiability*. For a given formula it has to be decided whether a feature graph exists that is a model of the constraint.

A more fundamental treatment is given by Shieber [1992], who starts with the logical requirements for unification-based grammars and then sets out to investigate which models would be appropriate.

Our purpose, in this chapter and the next, is a rather different one. We will investigate how, for a given class of unification grammars, efficient parsers can be developed, by means of parsing schemata. Just like in the context-free case, we will be concerned with the question which items one likes to derive and which rules should be used for that. In addition, we extend the formalism with a notation that allows explicit specification of transfer of features between items.

Parsing of unification grammars is a combination of two problem areas, both of which are complex in itself. Parsing is our primary interest, and the linguistic and logical properties of unification grammars secondary. Hence we do not worry about how to specify suitable unification grammars for natural languages, nor are we particularly concerned with the logical properties of various unification grammar formalisms, but we assume a simple kind of unification grammar and address the question how efficient parsers can be defined.

In order to be precise we will give a detailed, formal account of our simple formalism, that establishes thoroughly what we have presented informally in Chapter 7. The results are virtually equal to those of Smolka and others, but we employ a rather more computational view and do not pretend to give a general treatise on unification grammars.

We do not make a distinction between syntax (constraints) and semantics (feature graphs); we see both domains as syntactic domains. The notion of satisfiability is replaced by *consistency*. There is a simple isomorphism between consistent constraints[1] and well-formed feature graphs. Thus we obtain an abstract notion of a feature structure that may materialize in two different avatars: either as a graph or as a constraint. We switch representation opportunistically to the domain that is most convenient at any given moment. For the purpose of (statically) describing a grammar, the constraint representation is the most useful. But the dynamics of a grammar, describing how a parse is to be obtained by *unification* of feature structures, are easiest understood in the feature graph domain.

Feature structures, both as graphs and constraint sets, are introduced in 8.1. For both representations we define a lattice and prove these to be isomorphic in 8.2. For a proper formalization of how features of different objects may relate to one another, we introduce *composite* feature structures in 8.3 and define lattices in 8.4. This formalism is used to define unification grammars in 8.5. Tree composition in Primordial Soup fashion is discussed in 8.6 and parsing schemata, finally, are defined in 8.7.

In 8.8, at last, we give another example. The canonical example sentence is parsed with grammar UG_1 (cf. Section 7.2) using an Earley-type parsing schema (cf. Section 7.1). An overview of other grammar formalisms is presented in 8.9, related approaches are briefly discussed in 8.10, and conclusions are summarized in 8.11.

[1] From Section 8.1 onwards, we will call these *constraint sets*. A constraint as a formula in first order logic with equality can be seen as a conjunction of a series of atomic constraints. For our purposes it will be more convenient to describe this as a set of atomic constraints, rather than a conjunction.

8.1 Feature structures

We will give two different formalizations of feature structures, as *constraint sets* and *feature graphs*, and prove these to be isomorphic. The attribute-value matrix (AVM) notation will be used as a convenient, informal notation to denote feature structures. The correspondence between AVMs, feature graphs and constraint sets is straightforward. In Figure 8.1 an AVM is shown with corresponding constraint set and feature graph.

In Figure 8.1(a)–(c) it is exemplified how the information contained in an AVM can be encoded in a graph. The features are represented by edges; the atomic values are represented by labels of terminal vertices. Internal vertices carry no label; their value is the feature structure represented by the outgoing edges. The root vertex can be labelled with an identifier for the object whose features are represented here.

In order to give a formal definition of the domain of feature graphs, we first introduce some auxiliary domains from which features and values can be drawn.

Definition 8.1 (*features, constants*)
$\mathcal{F}ea$ denotes a finite set of features. We write $f, g, h, \ldots$ for elements of $\mathcal{F}ea$.
$\mathcal{C}onst$ denotes a finite set of constants. We write $c, d, e, \ldots$ for elements of $\mathcal{C}onst$.
It is assumed that $\mathcal{F}ea$ and $\mathcal{C}onst$ are disjunct sets. Furthermore, we assume that a linear order has been defined on both sets $\mathcal{F}ea$ and $\mathcal{C}onst$.
In the sequel we will also need *sequences of features*. We write π, ϱ for elements of $\mathcal{F}ea^*$. A linear order on $\mathcal{F}ea^*$ is defined by the "lexicographic order" based on the linear order of $\mathcal{F}ea$:

(i) $\quad \pi < \pi\varrho$ for non-empty feature sequences ϱ;

(ii) $\quad \pi f \varrho < \pi g \varrho'$ if $f < g$.

This linear order on feature sequences will be used to define a suitable normal form for constraint sets. □

We recall some useful notions from graph theory and introduce appropriate notations.

Definition 8.2 (DAGS)
A directed graph is a pair $\Gamma = (U, E)$, with U a set of vertices[2] and E a set of edges. An edge is a directed pair (u, v) with $u, v \in U$. Usually we write $u \rightarrow v$ for $(u, v) \in E$.
A (possibly empty) sequence of edges $u_0 \rightarrow u_1, u_1 \rightarrow u_2, \ldots, u_{k-1} \rightarrow u_k$ is called a *path*. We write $u \longrightarrow v$ for a path from u to v.

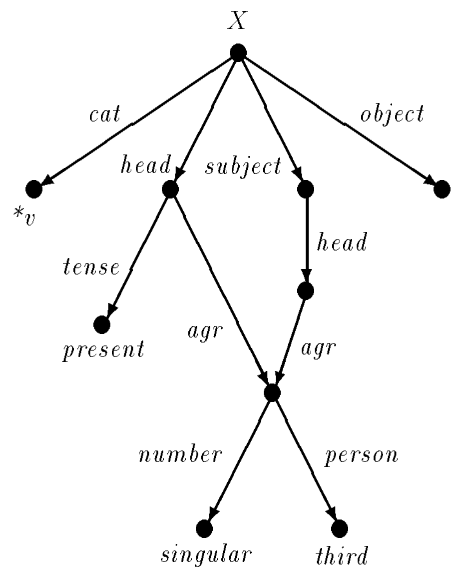
[2] We write U rather than V for the set of vertices, because V denotes the grammar variables $N \cup \Sigma$.

$$X \longmapsto \begin{bmatrix} cat & : & *v \\ head & : & \begin{bmatrix} tense: & present \\ agr & : \boxed{1} \begin{bmatrix} number: & singular \\ person & : third \end{bmatrix} \end{bmatrix} \\ subject: & & \begin{bmatrix} head: \begin{bmatrix} agr: \boxed{1} \end{bmatrix} \end{bmatrix} \\ object & : & [\ ] \end{bmatrix}$$

(a) an attribute value matrix

$$\begin{array}{ll} \{ & \langle X\ cat\rangle \doteq *v, \\ & \langle X\ head\ tense\rangle \doteq present, \\ & \langle X\ head\ agr\ number\rangle \doteq singular, \\ & \langle X\ head\ agr\ person\rangle \doteq third, \\ & \langle X\ subject\ head\ agr\ \rangle \doteq \langle X\ head\ agr\rangle, \\ & \langle X\ object\rangle \doteq [\ ] \qquad \} \end{array}$$

(b) a constraint set



(c) a feature graph

Figure 8.1: Three different representations of the same feature structure

A directed graph is called *cyclic* if there is a non-empty path $u \longrightarrow u$ for some vertex $u \in U$. A graph *acyclic* if it is not cyclic. We write DAG as abbreviation for a directed acyclic graph.
A *root* of a graph is a vertex u such that for all $v \in U$ there is a path from u to v.
A DAG is called *rooted* if it has exactly one root.
An edge $u \rightarrow v$ is an *outgoing edge* of u and and *incoming edge* of v.
A *leaf* is a vertex with no outgoing edges. □

Definition 8.3 (*feature graphs*)
$\mathcal{FG}$ is the class of finite, rooted DAGs with the following properties:

(i) every edge is labelled with a feature;

(ii) if f and g are labels of edges originating from the same vertex, then $f \neq g$;

(iii) leaves may be (but need not be) labelled with a constant;
non-leaf vertices do not carry a label.

We write $u \xrightarrow{f} v$ if $u \rightarrow v$ is labelled f; we write $u \xrightarrow{\pi} v$ if the sequence of steps from u to v is labelled with a sequence of features π. We write $label(u) = c$ if u is labelled with constant c and $label(u) = \varepsilon$ if u carries no label.
We write $\Gamma(X)$ for a feature graph that denotes the features of some (here unspecified) object X. □

An example of a constraint set was shown in Figure 8.1(b). In the definition of a constraint set, we have included a parameter X that can be used to identify an object for which constraints are to be specified. We will not use this parameter for a while, but include it here in anticipation of composite constraint sets that will be defined in Section 8.3.

Definition 8.4 (*constraint set*)
Let X be a (not further specified) object. Constraints on X can be drawn from different domains:

- The domain of *value constraints VC* is defined by

$$VC \;=\; \{\langle X\pi\rangle \doteq c \;\mid\; \pi \in \mathcal{F}ea^* \;\wedge\; c \in \mathcal{C}onst\};$$

- The domain of *existential constraints EC* is defined by

$$EC \;=\; \{\langle X\pi\rangle \doteq [\,] \;\mid\; \pi \in \mathcal{F}ea^*\}$$

where [ ] is a symbol that does not occur in $\mathcal{F}ea$ and $\mathcal{C}onst$;

- The domain of *coreference constraints* CC is defined by

$$CC \;=\; \{\langle X\pi\rangle \doteq \langle X\varrho\rangle \;\mid\; \pi,\varrho \in \mathcal{F}ea^*\}.$$

A *constraint set* $\chi(X)$ is a finite subset of $VC \times EC \times CC$.
As an ad-hoc general notation we write $\langle X\pi\rangle \doteq \mu$ for a constraint, where μ can be of the form c, [ ], or $\langle X\varrho\rangle$. □

Definition 8.5 (*closure of a constraint set*)
Let $\chi(X) \subset VC \times EC \times CC$ be a constraint set. The *closure* of $\chi(X)$, denoted $closure(\chi(X))$, is the smallest set satisfying

(i) if $\langle X\pi\rangle \doteq \mu \in \chi(X)$ then $\langle X\pi\rangle \doteq \mu \in closure(\chi(X))$;

(ii) if $\langle X\pi\rangle \doteq \langle X\pi'\rangle \in closure(\chi(X))$ and $\langle X\pi\varrho\rangle \doteq \mu \in closure(\chi(X))$ then $\langle X\pi'\varrho\rangle \doteq \mu \in closure(\chi(X))$;

(iii) if $\langle X\pi\rangle \doteq \langle X\varrho\rangle \in closure(\chi(X))$ then $\langle X\varrho\rangle \doteq \langle X\pi\rangle \in closure(\chi(X))$;

(iv) if $\langle X\pi\varrho\rangle \doteq \mu \in closure(\chi(X))$ then $\langle X\pi\rangle \doteq [\,] \in closure(\chi(X))$.

A constraint set $\chi(X)$ is called *closed* if $closure(\chi(X)) = \chi(X)$. □

Note that $closure(\chi(X))$ need not be a constraint set according to Definition 8.4: it could be an infinite set. If, for example, $\langle X\pi\rangle \doteq \langle X\pi\varrho\rangle \in \chi(X)$ then, by (ii) we obtain $\langle X\pi\varrho\rangle \doteq \langle X\pi\varrho\varrho\rangle \in \chi(X)$, $\langle X\pi\varrho\varrho\rangle \doteq \langle X\pi\varrho\varrho\varrho\rangle \in \chi(X)$, and so forth.

The purpose of the existential constraints added in (iv) is to identify the existence of all substructures. We will use them for the transformation of a constraint set into a graph.

The closure of the constraint set in Figure 8.1(b) is shown in Figure 8.2. The concept of a closed constraint set is useful because it defines a notion of *equivalence* that corresponds to our intuitive notion of when two constraint sets specify "the same information". We call $\chi_1(X)$ and $\chi_2(X)$ equivalent if $closure(\chi_1(X)) = closure(\chi_2(X))$. Closed constraint sets thus constitute a *normal form* for constraint sets, albeit a not very practical one. In the sequel we will define a more practical normal form.

Definition 8.6 (*consistency*)
A closed constraint set $\chi(X)$ is called *consistent* if it satisfies the following properties:

(i) if $\langle X\pi\rangle \doteq c \in \chi(X)$ and $\langle X\pi\rangle \doteq d \in \chi(X)$ then $c = d$;

(ii) if $\langle X\pi\rangle \doteq c \in \chi(X)$ and $\langle X\pi\varrho\rangle \doteq \mu \in \chi(X)$ then $\varrho = \varepsilon$;

(iii) $\langle X\pi\varrho\rangle \doteq \langle X\pi\rangle$ and $\langle X\pi\rangle \doteq \langle X\pi\varrho\rangle$ are not in $\chi(X)$ for any π and non-empty ϱ.

$$
\begin{array}{ll}
\{ & \langle X \rangle \doteq [\,], \\
& \langle X\ cat \rangle \doteq [\,], \\
& \langle X\ head \rangle \doteq [\,], \\
& \langle X\ head\ tense \rangle \doteq [\,], \\
& \langle X\ head\ agr \rangle \doteq [\,], \\
& \langle X\ head\ agr\ number \rangle \doteq [\,], \\
& \langle X\ head\ agr\ person \rangle \doteq [\,], \\
& \langle X\ subject \rangle \doteq [\,], \\
& \langle X\ subject\ head \rangle \doteq [\,], \\
& \langle X\ subject\ head\ agr \rangle \doteq [\,], \\
& \langle X\ subject\ head\ agr\ number \rangle \doteq [\,], \\
& \langle X\ subject\ head\ agr\ person \rangle \doteq [\,], \\
& \langle X\ object \rangle \doteq [\,], \\
& \langle X\ cat \rangle \doteq {*v}, \\
& \langle X\ head\ tense \rangle \doteq present, \\
& \langle X\ head\ agr\ number \rangle \doteq singular, \\
& \langle X\ head\ agr\ person \rangle \doteq third, \\
& \langle X\ subject\ head\ agr\ number \rangle \doteq singular, \\
& \langle X\ subject\ head\ agr\ person \rangle \doteq third, \\
& \langle X\ head\ agr\ \rangle \doteq \langle X\ subject\ head\ agr \rangle, \\
& \langle X\ subject\ head\ agr\ \rangle \doteq \langle X\ head\ agr \rangle \qquad \}
\end{array}
$$

Figure 8.2: Closure of the constraint set in Figure 8.1(b)

An arbitrary constraint set $\chi(X)$ is called consistent if $closure(\chi(X))$ is consistent. We write $\mathcal{CCS}$ for the set of consistent constraint sets. □

Corollary 8.7
If $\chi(X) \in \mathcal{CCS}$ then $closure(\chi(X)) \in \mathcal{CCS}$. □

Definition 8.8 (*mapping constraint sets to graphs*)
For each consistent constraint set $\chi(X) \in \mathcal{CCS}$ we define a graph, as follows. Vertices correspond to sets of left-hand sides of constraints. These sets, denoted $[\langle X\pi \rangle]$, are defined by

$$[\langle X\pi \rangle] \;=\; \{\langle X\pi \rangle\} \;\cup\; \{\langle X\varrho \rangle \;\mid\; \langle X\pi \rangle \;\doteq\; \langle X\varrho \rangle \in closure(\chi(X))\}.$$

The graph $\Gamma(X) = graph(\chi(X))$ is defined by

$$U \;=\; \{[\langle X\pi \rangle] \;\mid\; \langle X\pi \rangle \doteq [\,] \;\in closure(\chi(X))\},$$

$$E \;=\; \{[\langle X\pi \rangle] \xrightarrow{f} [\langle X\pi f \rangle] \;\mid\; \langle X\pi f \rangle \doteq [\,] \;\in closure(\chi(X))\}.$$

The label of a vertex $[\langle X\pi \rangle]$ is defined by

$$label([\langle X\pi\rangle]) \;=\; \begin{cases} c & \text{if } \langle X\pi\rangle \doteq c \in closure(\chi(X)) \\ \varepsilon & \text{otherwise} \end{cases} .$$ □

Lemma 8.9
For each $\chi(X) \in \mathcal{CCS}$ it holds that $graph(\chi(X)) \in \mathcal{FG}$.

Proof. Direct from the following observations:

- if $[\langle X\pi f\rangle] \doteq [\,] \in closure(\chi(X))$ then also $[\langle X\pi\rangle] \doteq [\,] \in closure(\chi(X))$, hence E is properly defined with respect to U;
- if $[\langle X\pi\rangle] \xrightarrow{f} u$ and $[\langle X\pi\rangle] \xrightarrow{f} v$ then $u = v$;
- the graph has a root $[\langle X\rangle]$;
- there are no $\langle X\pi\rangle \doteq c$ and $\langle X\pi\rangle \doteq d$ with $c \neq d$, hence each label is uniquely defined;
- moreover, if $\langle X\pi\varrho\rangle \doteq \mu \in closure(\chi(X))$ for non-empty ϱ then the consistency of $\chi(X))$ guarantees that there is no $\langle X\pi\rangle \doteq c \in closure(\chi(X))$, hence $label([\langle X\pi\rangle]) = \varepsilon$. □

Definition 8.10 (*mapping graphs to a constraint sets*)
For each feature graph $\Gamma(X) \in \mathcal{FG}$ we define a constraint set. To that end, we label each vertex with an auxiliary *path label*. If there are several paths to a vertex, we take the lowest one in lexicographical order. Formally: let r be the root of $\Gamma(X)$, then

$$path\_label(u) \;=\; \min\{\varrho \mid r \xrightarrow{\varrho} u\}.$$

A constraint set $constraints(\Gamma(X))$ is (uniquely) defined by

$$\chi_V(X) \;=\; \{\langle X\, path\_label(u)\rangle \doteq c \;\mid\; label(u) = c\},$$

$$\chi_E(X) \;=\; \{\langle X\, path\_label(u)\rangle \doteq [\,] \;\mid\; u \text{ is a leaf} \;\wedge\; label(u) \doteq \varepsilon\},$$

$$\chi_C(X) \;=\; \{\langle X\, path\_label(u)\rangle \doteq \langle X\varrho\rangle \;\mid\; r \xrightarrow{\varrho} u \;\wedge\; \varrho \neq path\_label(u)\},$$

$$\chi(X) \;=\; \chi_V(X) \cup \chi_E(X) \cup \chi_C(X).$$ □

Lemma 8.11
For each graph $\Gamma(X) \in \mathcal{FG}$ it holds that $constraints(\Gamma(X)) \in \mathcal{CCS}$.

Proof. Let $\Gamma(X) \in \mathcal{FG}$. We verify the constraints for consistency of Definition 8.6. (i) follows from the definition of $\chi_V(X)$; (ii) because in $\Gamma(X)$ only leaves are labelled; (iii) because the graph is acyclic. □

Definition 8.12 (*normal form*)
The function $nf : \mathcal{CCS} \longrightarrow \mathcal{CCS}$ is defined by

$$nf(\chi(X)) = constraints(graph(\chi(X)).$$

$nf(\chi(X))$ can be thought of as the *normal form* of a constraint set. It is, roughly speaking, a constraint set with constraints that are minimal in lexicographical order. We write $nf\mathcal{CCS}$ for the set of constraint sets that satisfy $nf(\chi(X)) = \chi(X)$. □

In order to compute a normal form, it is not necessary to construct a graph and then afterward deconstruct it. An algorithm to obtain the normal form of a constraint set is shown in Figure 8.3. It is left to the reader to verify the correctness of this algorithm; our main concern right now is the existence of the normal form, rather than its computation.

procedure *normalize* $\chi(X)$
begin
    repeat each of the following steps

        replace $\langle X\pi\rangle \doteq \langle X\varrho\rangle$ by $\langle X\varrho\rangle \doteq \langle X\pi\rangle$
        if $\varrho < \pi$;

        replace $\langle X\pi\varrho\rangle \doteq \mu$ by $\langle X\pi'\varrho\rangle \doteq \mu$
        if $\pi' < \pi$ and $\langle X\pi\rangle \doteq \langle X\pi'\rangle \in \chi(X)$;

        delete $\langle X\pi\varrho\rangle \doteq \langle X\pi'\varrho\rangle$ from $\chi(X)$
        if $\langle X\pi\rangle \doteq \langle X\pi'\rangle \in \chi(X)$ and $\varrho \neq \varepsilon$;

        delete $\langle X\pi\rangle \doteq [\,]$ from $\chi(X)$
        if $\langle X\pi\varrho\rangle \doteq \mu \in \chi(X)$ for some $\varrho \neq \varepsilon$
        or if $\langle X\pi\rangle = c$

    until no more of these steps can be applied
end;

Figure 8.3: A simple normalization procedure for constraint sets

Lemma 8.13
When we restrict *graph* to constraints in normal form only, the functions

$$graph : nf\mathcal{CCS} \longrightarrow \mathcal{FG} \quad \text{and}$$

$$constraints : \mathcal{FG} \longrightarrow nf\mathcal{CCS}$$

are bijections. Moreover, they are each other's inverse.

Proof: straightforward. □

8.2 Feature lattices

We will now define a lattice structure for constraint sets and feature graphs. First, we recall the definition of a lattice.

Definition 8.14 (*lattice*)
Let $\mathcal{X}$ be an arbitrary set (with elements $X, Y, \ldots$) and $\sqsubseteq$ a partial order on $\mathcal{X}$. The pair $(\mathcal{X}, \sqsubseteq)$ is called a *lattice* if

(i) There is a top element $T \in \mathcal{X}$ and a bottom element $B \in \mathcal{X}$ such that $B \sqsubseteq X \sqsubseteq T$ for each $X \in \mathcal{X}$.

(ii) For each pair of elements $X, Y \in \mathcal{X}$ there is a *lowest upper bound* (*lub*), denoted $X \sqcup Y$, that satisfies

- (a) $X \sqsubseteq X \sqcup Y$ and $Y \sqsubseteq X \sqcup Y$;
- (b) for each Z such that $X \sqsubseteq Z$ and $Y \sqsubseteq Z$ it holds that $X \sqcup Y \sqsubseteq Z$.

(iii) For each pair of elements $X, Y \in \mathcal{X}$ there is a *greatest lower bound* (*glb*), denoted $X \sqcap Y$, that satisfies

- (a) $X \sqcap Y \sqsubseteq X$ and $X \sqcap Y \sqsubseteq Y$;
- (b) for each Z such that $Z \sqsubseteq X$ and $Z \sqsubseteq Y$ it holds that $Z \sqsubseteq X \sqcap Y$. □

Definition 8.15 ($nf\mathcal{CCS}^L$, $\mathcal{FG}^L$)
We define a set $\perp_{CCS}$ by

$$\perp_{CCS} = VC \cup EC \cup CC.$$

(This is not a constraint set according to Definition 8.4, as $\perp_{CCS}$ is not finite)
We define a graph $\perp_{FG} = (U_\perp, E_\perp)$ by

$$U_\perp = r,$$

$$E_\perp = \{r \xrightarrow{f} r \mid f \in \mathcal{F}ea\}.$$

(This is not a feature graph according to Definition 8.4, as $\perp_{FG}$ is not a DAG. The vertex r can be thought of as labelled with all constants at once.)
Furthermore, we extend *graph* and *constraints* by

$$graph(\perp_{CCS}) = \perp_{FG},$$

$$constraints(\bot_{FG}) = \bot_{CCS} \ .$$

We extend the domains of constraint sets and feature graphs by

$$\begin{aligned} nf\mathcal{CCS}^L &= nf\mathcal{CCS} \cup \{\bot_{CCS}\}, \\ \mathcal{FG}^L &= \mathcal{FG} \cup \{\bot_{FG}\}. \end{aligned}$$

When it is clear from the context which domain is meant, we drop the index and simply write $\bot$ for inconsistent. □

Definition 8.16 (*subsumption*)
A *subsumption relation* $\sqsubseteq$ is defined on $\mathcal{CCS}^L$ by

$$\chi_1(X) \sqsubseteq \chi_2(X) \quad \text{if} \quad closure(\chi_1(X)) \subseteq closure(\chi_2(X)).$$

A *subsumption relation* $\sqsubseteq$ is defined on $\mathcal{FG}^L$ by

$$\Gamma_1(X) \sqsubseteq \Gamma_2(X) \quad \text{if} \quad constraints(\Gamma_1(X)) \sqsubseteq constraints(\Gamma_2(X)).$$ □

Note that $\chi(X) \sqsubseteq \bot$ for any $\chi(X)$. It happens to be the case that $\bot$ is the top element of the lattice structure over constraint sets. This is somewhat unfortunate, because in lattice theory $\bot$ usually denotes the *bottom* element. On the other hand, it is not uncommon to interpret $\bot$ as "inconsistent". This notational problem can be solved, simply by reversing the lattice structure. If we write $\sqsupseteq$ and $\sqcap$, rather than $\sqsubseteq$ and $\sqcup$, we have $\bot$ as the bottom of the lattice. This is equally problematic, however, as it is not intuitively appealing to write $\sqcap$ for a symbol that is to be interpreted as a *union* of constraints. Hence we stick to the notation as introduced in Definition 8.16.

Theorem 8.17 (*lattice structure*)

(*a*) $(nf\mathcal{CCS}^L, \sqsubseteq)$ is a lattice with bottom $\{\langle X \rangle \doteq [\,]\}$ and top $\bot_{CCS}$.

(*b*) $(\mathcal{FG}^L, \sqsubseteq)$ is a lattice with bottom $graph(\{\langle X \rangle \doteq [\,]\})$ and top $\bot_{FG}$.

(*c*) $graph : nf\mathcal{CCS}^L \longrightarrow \mathcal{FG}^L$ is an isomorphism with respect to $\sqsubseteq$;
$constraints : \mathcal{FG}^L \longrightarrow nf\mathcal{CCS}^L$ is the inverse isomorphism.

Proof.

(*a*) The top and bottom properties are trivial.
The existence of a *lub* for any two constraint sets $\chi_1(X), \chi_2(X) \in nf\mathcal{CCS}^L$ is shown as follows. We write χ' for $closure(\chi_1(X) \cup \chi_2(X))$.
If χ' is inconsistent, then $\bot$ is obviously the *lub*.
Otherwise, assume $\chi'' \in \mathcal{CCS}$ with $\chi_1(X) \sqsubseteq \chi''$ and $\chi_2(X) \sqsubseteq \chi''$.
Then $closure(\chi_1(X)) \subseteq closure(\chi'')$, and $closure(\chi_2(X)) \subseteq closure(\chi'')$.
Hence $\chi' \subseteq closure(\chi'')$, and $nf(\chi')$ is the least upper bound in $nf\mathcal{CCS}^L$.
The existence of a *glb* follows in similar fashion.

(c) Straight from Lemma 8.13 and Definition 8.16.

(b) Direct from (a) and (c). □

We can extend the relation $\sqsubseteq$ to cover the entire set of consistent constraint sets $\mathcal{CCS}$. Note, however, that $(\mathcal{CCS} \cup \{\bot\}, \sqsubseteq)$ is *not* a lattice, because the *lub* is not uniquely defined.

Corollary 8.18
For any pair of consistent constraint sets in normal form $\chi_1(X), \chi_2(X) \in nf\mathcal{CCS}$ it holds that

$$\chi_1(X) \sqcup \chi_2(X) \;=\; nf(\chi_1(X) \cup \chi_2(X))$$ □

We have defined $\sqcup$ as a least upper bound, derived from the subsumption relation $\sqsubseteq$. In practical applications, we see $\sqcup$ as an *operator* that allows to construct new feature structures by merging the features of existing feature structures. How such a merge is carried out in an efficient manner is not a direct concern here. We will come back to that issue in Chapter 9.

Having proven that normal forms of consistent constraint sets and feature graphs are isomorphic, we can abstract from the particular representation and simply call it a *feature structure*. We write $\varphi(X)$ to denote a feature structure, or simply φ if it is not relevant which object X is characterized by the features in φ. A feature structure will be interpreted in an opportunistic manner either as feature graph or as constraint set, whatever is most convenient.

We write $\varphi(X).\pi$ to denote the *substructure* of $\varphi(X)$ that is (in the graph representation!) the largest subgraph of which $[\langle X\pi\rangle]$ is the root. We write $\varphi(X).\pi = c$ if (in constraint set representation!) $\langle X\pi\rangle \doteq c \in closure(\varphi(X))$.

As an informal notation for feature structures we write AVMs, feature graphs or constraint sets. It is not required that a constraint set be in normal form. Normal forms were important because the lattice structure is defined on normal forms, but for any practical application any equivalent specification of a constraint set will do as well. Hence, as we are not going to use normal forms, we do not need to explicitly specify a linear order on $\mathcal{F}ea$ and $\mathcal{C}onst$.

With the conceptual machinery introduced so far, we can now explain the difference between type identity and token identity. Consider the following feature structures:

$$\varphi_1 = \begin{bmatrix} f: \begin{bmatrix} f: c \\ g: d \end{bmatrix} \\ \\ g: \begin{bmatrix} f: c \\ g: d \end{bmatrix} \end{bmatrix}, \qquad \varphi_2 = \begin{bmatrix} f: \boxed{1} \begin{bmatrix} f: c \\ g: d \end{bmatrix} \\ \\ g: \boxed{1} \end{bmatrix}.$$

Then the substructures $\varphi_1.f$ and $\varphi_1.g$ are called *type identical*: they have the same value, but they are different structures. The substructures $\varphi_2.f$ and $\varphi_2.g$ are called *token identical*: they refer to a single structure (and have the same value a fortiori). Note that $\varphi_1 \sqsubseteq \varphi_2$, because the constraint set of φ_2 can be obtained from the constraint set of φ_1 by *adding a constraint* (i.c. $\langle Xf \rangle \doteq \langle Xg \rangle$). The difference between these structures comes to light when either structure is unified with $\varphi' = \begin{bmatrix} g\colon \begin{bmatrix} h\colon e \end{bmatrix} \end{bmatrix}$, yielding

$$\varphi_1 \sqcup \varphi' = \begin{bmatrix} f\colon \begin{bmatrix} f\colon c \\ g\colon d \end{bmatrix} \\ g\colon \begin{bmatrix} f\colon c \\ g\colon d \\ h\colon e \end{bmatrix} \end{bmatrix} \qquad \sqsubseteq \qquad \varphi_2 \sqcup \varphi' = \begin{bmatrix} f\colon \boxed{1} \begin{bmatrix} f\colon c \\ g\colon d \\ h\colon e \end{bmatrix} \\ g\colon \boxed{1} \end{bmatrix}.$$

In the sequel, we write the usual equality symbol ($=$) for type identity and a dotted equality symbol ($\doteq$) to denote token identity. So we have $\varphi_1.f = \varphi_1.g$, $\varphi_2.f = \varphi_2.g$, $\varphi_2.f \doteq \varphi_2.g$, but $\varphi_1.f \not\doteq \varphi_1.g$.

The difference between type identity and token identity is only relevant for substructures. For *constants* it doesn't make any difference whether a value is token identical to or a copy of some given other constant.

8.3 Composite feature structures

So far we have defined feature structures, that capture the characteristic properties of some object. It is essential, however, to add the conceptual machinery that allows us to relate the features of different objects to one another. To this end we introduce feature structures that describe the features of a (finite) *set* of objects. Features can be shared between objects by means of token identity.

Composite constraint sets for sets of objects are only a minimal extension of the constraint sets of Section 8.1: coreferencing is allowed between (features of) different objects. In the domain of feature graphs, we get a set of graphs that may share subgraphs. Or, to put it differently, we get a single graph with multiple roots.

Definition 8.19 (*multi-rooted feature graphs*)
A *multi-rooted feature graph* is a structure $\Gamma(X_1, \ldots, X_k) = (U, E, R)$ with (U, E) a finite DAG and $R = \{r_1, \ldots, r_k\} \subseteq U$, with the following properties:

(i) every edge is labelled with a feature;

(ii) if f and g are labels of edges originating from the same vertex, then $f \neq g$;

(*iii*) leaves may be (but need not be) labelled with a constant, non-leaf vertices do not carry a constant label;

(*iv*) For every $u \in U$ there is some $r \in R$ such that $r \longrightarrow u$.

We call R the *root set* of the graph. The size of the root set must correspond to the number of formal parameters $X_1, \ldots, X_k$; the roots can be labelled with identifiers referring to the objects whose features are represented. Note that is it *not* required that a root r_i has no incoming edges. It is conceivable that one root is the descendant of another root (and also that several roots coincide). In that case, the features of one object are token identical with a substructure of the features of another object.
We write $\mathcal{MFG}$ for the class of multi-rooted feature graphs. □

Definition 8.20 (*composite constraint sets, closure*)
Let $X_1, \ldots, X_k$ denote a finite set of objects. A (composite) constraint set $\chi(X_1, \ldots, X_k)$ is a finite set of constraints from the domains of value constraints, existential constraints and *composite* coreference constraints, defined as follows:

$$VC = \{\langle X_i \pi\rangle \doteq c \mid 1 \leq i \leq k \ \wedge\ \pi \in \mathcal{F}ea^* \ \wedge\ c \in \mathcal{C}onst\},$$

$$EC = \{\langle X_i \pi\rangle \doteq [\,] \mid 1 \leq i \leq k \ \wedge\ \pi \in \mathcal{F}ea^*\},$$

$$CCC = \{\langle X_i \pi\rangle \doteq \langle X_j \varrho\rangle \mid 1 \leq i \leq k \ \wedge\ 1 \leq j \leq k \ \wedge\ \pi, \varrho \in \mathcal{F}ea^*\}.$$

The *closure* of a constraint set is obtained as in Definition 8.5, with X replaced by X_i or X_j as appropriate. □

Definition 8.21 (*consistency*)
A *closed* composite constraint set $\chi(X_1, \ldots, X_k)$ is called *consistent* if it satisfies the following properties:

(*i*) if $\langle X_i \pi\rangle \doteq c \in \chi(X_1, \ldots, X_k)$ and $\langle X_i \pi\rangle \doteq d \in \chi(X_1, \ldots, X_k)$ then $c = d$;

(*ii*) if $\langle X_i \pi\rangle \doteq c \in \chi(X_1, \ldots, X_k)$ and $\langle X_i \pi \varrho\rangle \doteq \mu \in \chi(X_1, \ldots, X_k)$ then $\varrho = \varepsilon$;

(*iii*) $\langle X_i \pi \varrho\rangle \doteq \langle X_i \pi\rangle$ and $\langle X_i \pi\rangle \doteq \langle X_i \pi \varrho\rangle$ are not in $\chi(X_1, \ldots, X_k)$ for any i π, and non-empty ϱ.

An arbitrary composite constraint set $\chi(X_1, \ldots, X_k)$ is consistent if *closure*($\chi(X_1, \ldots, X_k)$) is consistent.
We write $\mathcal{CCCS}$ for the set of consistent composite constraint sets. □

Definition 8.22 (*mappings, normal form*)
The mappings *graph* and *constraints* can be extended to composite constraint sets and multi-rooted feature graphs in the obvious way (and it can be verified

straightforwardly that these functions are well-defined).
The function $nf : \mathcal{CCCS} \longrightarrow \mathcal{CCCS}$ is defined by

$$nf(\chi(X_1,\ldots,X_k)) = constraints(graph(\chi(X_1,\ldots,X_k));$$

We write $nf\mathcal{CCCS}$ for the set of constraint sets that satisfy

$$nf(\chi(X_1,\ldots,X_k)) = \chi(X_1,\ldots,X_k). \qquad \square$$

Definition 8.23 (*substructures*)
Let $\Gamma(X_1,\ldots,X_k) = (U, E, \{r_1,\ldots,r_k\}) \in \mathcal{MFG}$ describe the features of a set of k objects. The feature graphs of a *subset* of this set of objects are described by a subgraph, as follows.
Let $\{X_{i_1},\ldots,X_{i_m}\} \subset \{X_1,\ldots,X_k\}$.
Then $\Gamma(X_{i_1},\ldots,X_{i_m}) = (U', E', \{r_{i_1},\ldots,r_{i_m}\})$ is defined by

$$\begin{aligned} U' &= \{u \in U \mid r_{i_j} \longrightarrow u \text{ for some } j\ (1 \leq j \leq m)\}, \\ E' &= \{u \rightarrow v \in E \mid u, v \in U'\}. \end{aligned}$$

Similarly, a substructure is defined for *closed* constraint sets[3].
Let $\chi(X_1,\ldots,X_k)$ be a closed constraint set. A (closed) substructure $\chi(X_{i_1},\ldots,X_{i_m})$ for $\{X_{i_1},\ldots,X_{i_m}\} \subset \{X_1,\ldots,X_k\}$ is defined by

$$\begin{aligned} \chi(X_{i_1},\ldots,X_{i_m}) \;=\; & \{\langle X_{i_j}\pi\rangle \doteq c \in \chi(X_1,\ldots,X_k) \mid 1 \leq j \leq m\} \cup \\ & \{\langle X_{i_j}\pi\rangle \doteq [\,] \in \chi(X_1,\ldots,X_k) \mid 1 \leq j \leq m\} \cup \\ & \{\langle X_{i_j}\pi\rangle \doteq \langle X_{i_l}\varrho\rangle \in \chi(X_1,\ldots,X_k) \\ & \qquad \mid 1 \leq j \leq m \wedge 1 \leq l \leq m\}. \end{aligned}$$

For $\chi(X_1,\ldots,X_k) \in nf\mathcal{CCCS}$ and $\{X_{i_1},\ldots,X_{i_m}\} \subset \{X_1,\ldots,X_k\}$ we define a substructure $\chi(X_{i_1},\ldots,X_{i_m})$ as follows.

Let $\chi'(X_1,\ldots,X_k) = closure(\chi(X_1,\ldots,X_k))$;
then $\chi(X_{i_1},\ldots,X_{i_m}) = nf(\chi'(X_{i_1},\ldots,X_{i_m}))$. $\square$

Definition 8.24 (*composite feature lattices*)
We define a set $\perp_{CCCS}$ by

$$\perp_{CCCS} = VC \cup EC \cup CCC.$$

As inconsistent $\mathcal{MFG}$ we define a multi-rooted graph $\perp_{MFG} = (U_\perp, E_\perp, R_\perp)$ with an infinite root set:

$$U_\perp = R_\perp = \{r_1, \ldots\},$$

[3] We cannot simply apply the same definition to arbitrary constraint sets: if a feature of some X_{i_j} is token identical with an object that is no longer represented in the substructure, all constraints relating to that part of the deleted substructure must be taken into account as well. Only in closed constraint sets it is guaranteed that every feature of an object is completely described by constraints for that object.

$$E_\perp = \{r_i \xrightarrow{f} r_j \mid r_i, r_j \in R_\perp \ \wedge\ f \in \mathcal{F}ea\}.$$

Each vertex r_i can be thought of as being labelled with all constants at once.
The functions *graph* and *constraints* are extended to map $\perp_{CCCS}$ and $\perp_{MFG}$ onto each other.
We define the domains

$$\begin{aligned} nf\mathcal{CCCS}^L &= nf\mathcal{CCCS} \cup \{\perp_{CCCS}\}, \\ \mathcal{MFG}^L &= \mathcal{MFG} \cup \{\perp_{MFG}\}. \end{aligned}$$

□

8.4 Composite feature lattices

Before we define subsumption on composite feature structures, we must clarify the distinction between objects and formal parameters. It is our purpose to derive a binary operator $\sqcup$ that can be used to unify feature structures. A feature structure $\varphi(X_1,\ldots,X_k) \sqcup \varphi(Y_1,\ldots,Y_l)$ combines the features of both structures. It is important to know, however, which X's and which Y's refer to identical objects. Let, for example, $X_3 = Y_2$ and all other X_i and Y_j be different. Then in the unified feature structure $\varphi(X_1,\ldots,X_k) \sqcup \varphi(Y_1,\ldots,Y_l)$ there is (a parameter for) an object that will contain both the features of $\varphi(X_3)$ and $\varphi(Y_2)$. (Note, however, that $\varphi(X_3)$ and $\varphi(Y_2)$ are separate feature structures. Features can be shared across objects (or parameters) *within a single composite feature structure*, but features can *not* be shared across different composite feature structures.) Hence it is essential to know which parameters denote which objects, so that the right pairs of features are unified when we unify two composite feature structures. Therefore we assume the existence of a (possibly infinite but countable) domain of objects and postulate that each parameter refers to an object.

In a practical notation, we could annotate the unification with which parameters should be considered to refer to the same object. The above case can be denoted as

$$\varphi(X_1,\ldots,X_k) \sqcup_{X_3=Y_2} \varphi(Y_1,\ldots,Y_l).$$

As indices to the unification we write (sequences) of equalities that denote correspondence between formal parameters of either argument. In the unlikely case that all formal parameters are different we could write $\sqcup_\emptyset$ (but this operation will not be used in the sequel). Hence, when we write an unqualified *lub* symbol $\sqcup$ it should be clear from the context which parameters of both arguments refer to the same object. This will usually be the case.

In practical use, we see $\sqcup$ as an operator that can be used to construct new feature structures from existing feature structures. But before we start using it, we have to define $\sqcup$ formally as a least upper bound in a lattice.

Definition 8.25 (*subsumption*)
A *subsumption relation* $\sqsubseteq$ is defined on $nf\mathcal{CCCS}^L$ as follows:

$$\chi_1(X_1, \ldots, X_k) \sqsubseteq \chi_2(Y_1, \ldots, Y_l) \text{ holds if}$$

$$(i) \quad \{X_1, \ldots, X_k\} \subseteq \{Y_1, \ldots, Y_l\}, \text{ and}$$
$$(ii) \quad closure(\chi_1(X_1, \ldots, X_k)) \subseteq closure(\chi_2(X_1, \ldots, X_k)).$$

A subsumption relation $\sqsubseteq$ is defined on $\mathcal{MFG}^L$ by

$$\Gamma(X_1, \ldots, X_k) \sqsubseteq \Gamma_2(Y_1, \ldots, Y_l) \text{ holds if}$$

$$constraints(\Gamma_1(X_1, \ldots, X_k)) \sqsubseteq constraints(\Gamma_2(Y_1, \ldots, Y_l)).$$ □

Theorem 8.26 (*lattice structure*)
The following statement hold:

(*a*) $(nf\mathcal{CCCS}^L, \sqsubseteq)$ is a lattice with the empty constraint set as bottom and top $\bot_{CCCS}$.

(*b*) $(\mathcal{MFG}^L, \sqsubseteq)$ is a lattice with the empty graph as bottom and top $\bot_{MFG}$.

(*c*) $graph : nf\mathcal{CCCS}^L \longrightarrow \mathcal{MFG}^L$ is an isomorphism with respect to $\sqsubseteq$; $constraints : \mathcal{MFG}^L \longrightarrow nf\mathcal{CCCS}^L$ is the inverse isomorphism.

Proof: straightforward extension of the proof of Theorem 8.17 and preceding lemmata. □

Corollary 8.27
For consistent composite constraint sets in normal form
$\chi_1(X_1, \ldots, X_k), \chi_2(Y_1, \ldots, Y_l) \in nf\mathcal{CCCS}$ it holds that

$$\chi_1(X_1, \ldots, X_k) \sqcup_{X_{i_1}=Y_{j_1}, \ldots, X_{i_m}=Y_{j_m}} \chi_2(Y_1, \ldots, Y_l) =$$
$$nf(\chi_1(X_1, \ldots, X_k) \cup \chi_2(Y_1, \ldots, Y_l) \cup \{\langle X_{i_1}\rangle \doteq \langle Y_{j_1}\rangle, \ldots, \langle X_{i_m}\rangle \doteq \langle Y_{j_m}\rangle\}).$$ □

As with constraint sets and feature graphs, we will blur the distinction between composite constraint sets and multi-rooted feature graphs. We simply write $\varphi(X_1, \ldots, X_k)$ to denote a *composite feature structure* for k objects. As in 8.1 we write Φ to denote *both* lattices $(nf\mathcal{CCCS}^L, \sqsubseteq)$ and $(\mathcal{MFG}^L, \sqsubseteq)$. If we need one particular representation we will pick the one that is easiest to work with, depending on the circumstances.

From a composite feature structure $\varphi(X_1, \ldots, X_k)$ one can derive a feature structure $\varphi(X_i)$ for any object, by taking the appropriate substructure. As a convenient notation we write

$$\varphi(X_i) = \varphi(X_1, \ldots, X_k)|_{X_i}$$

to denote that a feature structure for an object X_i is obtained by retrieving it from some composite structure.

Up to now we have only attributed features to *sets* of objects. It is possible that the objects themselves are contained in a structure of some kind. We call these *object structures* so as avoid confusion with feature structures. Typical object structures that we will use in the remainder of this chapter are

- A production $A{\rightarrow}\alpha$ from a context-free grammar.

  We write $\varphi(A{\rightarrow}\alpha)$ as a convenient notation for a composite feature structure $\varphi(A, X_1, \ldots, X_k)$ that describes features of left-hand and right-hand side symbols, where $\alpha = X_1, \ldots, X_k$.

- A tree $\langle A \rightsquigarrow \alpha\rangle$.

  We write $\varphi(\langle A \rightsquigarrow \alpha\rangle)$ as a convenient notation for a composite feature structure $\varphi(A, \ \ldots \ , X_1, \ldots, X_k)$, where $\alpha = X_1, \ldots, X_k$.

- An item $[A \rightsquigarrow \alpha]$.

  Items were introduced in Chapter 4 as *sets of* trees. Here we should see them as *abstractions* of trees: We only know the root and the yield of the item; we do not know (or do not want to know) the internal nodes. Consequently, features can be retrieved only from the nodes that are explicitly mentioned in the denotation of the item. Hence, a composite feature structure of an item $[A \rightsquigarrow \alpha]$ can be seen as a substructure of a composite feature structure of a tree $\langle A \rightsquigarrow \alpha\rangle$, from which the features of internal nodes have been deleted. We write $\varphi([A \rightsquigarrow \alpha])$ as a convenient notation for a composite feature structure $\varphi(A, X_1, \ldots, X_k)$ where $\alpha = X_1, \ldots, X_k$.
  A similar interpretation will be given to various kinds of items that give various kinds of partial specifications of trees. As an example, consider the item $[S{\rightarrow}NP{\bullet}VP, 0, 2]$, specifying the fact than an NP has been found by scanning the first two words (but we don't care to remember what those words were). A feature structure $\varphi([S{\rightarrow}NP{\bullet}VP, 0, 2])$ will be a composite feature structure $\varphi(S, NP, VP)$ that denotes the appropriate substructure of $\varphi(\langle S{\rightarrow}\langle NP \rightsquigarrow \underline{a}_1\underline{a}_2\rangle \; VP\rangle)$.

8.5 Unification grammars

With the lattice of (composite) feature structures, developed in in 8.1 and 8.3, we can now formally define a unification grammar as it has been informally presented in Chapter 7.

The definition of unification grammars that we present here is not the most compact one that is possible. One could eliminate the context-free backbone and let syntactic category be a feature as any other. If one abstracts from the syntactic category as a special feature, the definitions and notations become more terse, but somewhat more obscure. For the sake of clarity and compatibility with the other chapters, we will not do so.

We take it for granted that syntactic category is such a fundamental notion that every feature structure for every constituent constraints at least a *cat* feature. Hence, in order to obtain a legible notation, we continue to call nodes in a tree by their syntactic category, like we did with context-free grammars.

Definition 8.28 (*unification grammar*)
A unification grammar is a structure

$$\mathcal{G} = (G, \Phi, \varphi_0, W, \mathcal{L}ex).$$

The different parts of this structure are defined as follows:

- $G = (N, \Sigma, P, S)$ is a context-free grammar. We write V for $N \cup \Sigma$; it is not required that $N \cap \Sigma = \emptyset$, a syntactic category is allowed to be both terminal and nonterminal.
  Furthermore, P is a *multiset* of productions, i.e., it is allowed that a single context-free production occurs more than one time.

- $\Phi = \Phi(\mathcal{F}ea, \mathcal{C}onst)$ is the lattice of feature structures based on a set of features $\mathcal{F}ea$ and a set of constants $\mathcal{C}onst$. It is assumed (but not necessary) that $\mathcal{F}ea \cap \mathcal{C}onst = \emptyset$. We assume $cat \in \mathcal{F}ea$ and $V \subseteq \mathcal{C}onst$, allowing for syntactic categories to be represented in a feature structure.

- $\varphi_0 : P \rightarrow \Phi$ is a function that a assigns a composite feature structure to each production in the context-free grammar. For each production $A \rightarrow X_1, \ldots, X_k$ it is required that

  $$\varphi_0(A).cat = A, \quad \varphi_0(X_1).cat = X_1, \quad \ldots, \quad \varphi_0(X_k).cat = X_k$$

  (where we write $\varphi_0(A)$ as a shorthand for $\varphi_0(A \rightarrow X_1 \ldots X_k)|_A$ and $\varphi_0(X_i)$ likewise).
  Different feature structures can be attributed to a single context-free production by including the production more than once in P.[4]

- W is a set of lexicon entries, i.e., "real" word forms, as opposed to lexical categories in Σ. It is assumed (but not necessary) that $V \cap W = \emptyset$. We write $\underline{a}, \ldots$ for words in W.

- $\mathcal{L}ex$ is a function that assigns a set of feature structures to each word in W (a word may have different readings). Each $\varphi(\underline{a}) \in \mathcal{L}ex(\underline{a})$ for each $\underline{a} \in W$ must have a feature cat. Moreover, it is required that $\varphi(\underline{a}).cat \in \Sigma$.

[4] Alternatively, one could have P as a proper set and attribute a *set of* composite feature structures to each production. There is no need to use multisets, then, but in the remainder of the chapter the expression "$\varphi_0(A \rightarrow \alpha)$" has to be replaced by "some φ in $\varphi_0(A \rightarrow \alpha)$".

We write $\mathcal{UG}$ for the class of unification grammars $\mathcal{G}$ that satisfy the above properties. □

One could argue whether the lexicon is part of the grammar or a separate structure. The size of the grammar is reduced tremendously when the lexicon is not contained in the grammar. It is somewhat artificial, however, to assume a grammar with production features φ_0 existing independently of a lexicon $(W, \mathcal{L}ex)$. The trend in unification grammars is that more and more information is stored in the lexicon, and the productions merely serve to prescribe concatenation and feature unification.

The reason for introducing an alphabet W, consisting of words with lexicon entries, is the following. In context-free parsing of natural languages it is standard use to consider the *word categories*, rather than the words from the lexicon, as terminal symbols. In Chapters 2 and 3 we have introduced the notational convention that leaves $a, b, \ldots$ in a parse tree indicate a terminal symbol, while leaves $\underline{a}, \underline{b}, \ldots$ indicate that these leaves correspond to words from the actual sentence that has to be parsed. In Chapter 2 the underlined terminal symbols were added to the grammar in the following way:

- for the i-th word of the sentence, extra productions $a{\rightarrow}\underline{a}_i$ are added for each possible lexical category of that word.

Verification that a word occurs in the sentence, therefore, could be expressed in terms of tree operations. For each auxiliary production we can supply a feature structure structure (in constraint set notation)

$$\varphi_0(a{\rightarrow}\underline{a}_i) \;=\; \{\langle a\rangle \doteq \langle \underline{a}_i\rangle\}.$$

These auxiliary productions are *not* part of the grammar, but an implementation technique that is used to construct the parse of a given sentence. We will stick to this notation, for the moment, because it allows us to express the difference between terminals that have been matched with the sentence and those that haven't been matched yet.

When we abstract from trees to items, in Section 8.7, we will simply have initial items of the form $[a, j-1, j]$ with a feature structure $\varphi(a) \in \mathcal{L}ex(\underline{a}_j)$. The careful distinction between matched leaves and non-matched leaves will no longer be relevant then.

Grammars may include ε-productions. In Section 3.1 we defined trees in such a way that an ε-production generates a leaf labelled ε. Throughout the remainder of this chapter we will simply assume that such leaves labelled ε are not decorated with any features. With this restriction, an arbitrary production $A{\rightarrow}\alpha$ in all the following definitions also applies to $A{\rightarrow}\varepsilon$.

Definition 8.29 (*decorated trees*)
A *decorated tree* is a pair $(\tau, \varphi(\tau))$ with $\tau \in \mathcal{T}rees(G)$ (cf. Definition 3.10.(*iii*))

and $\varphi(\tau)$ a composite feature structure for the nodes in τ, satisfying the following conditions

(*i*) for each node A with children α there is some $A \rightarrow \alpha \in P$ such that $\varphi_0(A \rightarrow \alpha) \sqsubseteq \varphi(A \rightarrow \alpha)$;

(*ii*) for each node a with child $\underline{a}_i$ it holds that $\varphi(a) \doteq \varphi(\underline{a}_i)$;

(*iii*) for each node $\underline{a}_i$ there is some $\varphi'(\underline{a}_i) \in \mathcal{L}ex(\underline{a}_i)$ such that $\varphi'(\underline{a}_i) \sqsubseteq \varphi(\underline{a}_i)$.

We write $\mathcal{DT}rees(\mathcal{G})$ for the set of decorated trees for some unification grammar $\mathcal{G}$. □

In 8.6, like in Chapter 2, we will construct parse trees by means of composition of smaller trees. Any tree can be composed from atomic trees. When a new tree is created that is a composition of two existing trees, its features will be merged. In this way, context-free parse trees can be obtained that are decorated with feature structures. We should make sure, however, that the feature structure of a parse tree contains only "adequate" features (in a sense to be made precise shortly) which are derived from the productions and lexicon. One can always extend the decoration of a tree by adding new features out of the blue. For a decorated parse tree, it should be required that no unnecessary features have sneaked in. The following definition rules out "over-decorated" trees.

Definition 8.30 (*adequately decorated trees*)
We define adequate decoration of trees by induction on the tree structure.[5] Let $G \in \mathcal{UG}$ be a unification grammar and $(\tau, \varphi(\tau))$ a decorated tree. The adequacy of the decoration $\varphi(\tau)$ is defined as follows, depending on the form of τ:[6]

[5] The reader might wonder why we do not give a direct definition of a *minimally* decorated tree. One could call $(\tau, \varphi(\tau))$ minimally decorated if there is no decoration $\varphi'(\tau) \neq \varphi(\tau)$ such that $\varphi'(\tau) \sqsubseteq \varphi(\tau)$. The problem is, however, that adequately decorated trees need not be minimal. As an example, consider a grammar with the following productions:

$$A \rightarrow B, \quad \varphi(B) = [f : a], \tag{8.1}$$

$$A \rightarrow B, \quad \varphi(B) = [g : b], \tag{8.2}$$

$$B \rightarrow C, \quad \varphi(B) = [g : b]. \tag{8.3}$$

A tree $\langle A \rightsquigarrow C \rangle$ composed from the elementary trees of productions (8.1) and (8.3) is decorated adequately, but not minimal.
In a practical grammar, it is likely that every adequately decorated tree is also minimally decorated. One could rule out grammars that allow non-minimal adequate decoration by additional constraints on the features of the productions and lexicon. This is not very relevant for the current discussion, therefore we bypass the issue with a definition of adequacy that is based on what ought to be proper composition of decorated trees.

[6] See Definition 3.8 for various forms of linear tree notation.

- $\tau = \langle a \rightarrow \underline{a}_i \rangle$ (i.e. τ matches a terminal with a word in the sentence).

  Then the decoration is adequate if $\varphi(a) \doteq \varphi(\underline{a}_i) \in \mathcal{L}ex(\underline{a}_i)$.

- $\tau = \langle A \rightarrow \alpha \rangle$ (i.e. τ covers a single production).

  Then the decoration is adequate if $\varphi(\tau) = \varphi_0(A \rightarrow \alpha)$.

- $\tau = \langle A \rightarrow \langle \alpha \rightsquigarrow \beta \rangle \rangle$ (i.e., a production $\langle A \rightarrow \alpha \rangle$ constitutes the top of the tree).

  Let $\alpha = X_1 \ldots X_k$, $\beta = \beta_1 \ldots \beta_k$, such that $\langle X_i \rightsquigarrow \beta_i \rangle$ is a subtree of τ for $1 \leq i \leq k$.
  We distinguish between *degenerate* subtrees, having a single node $X_i = \beta_i$ and no edges and nondegenerate subtrees having more than one node and at least one edge. The (only) adequate decoration for a degenerate subtree is the empty feature structure.
  Then $\varphi(\tau)$ is an adequate decoration if there are adequately decorated trees

  $$(\langle A \rightarrow \alpha \rangle, \varphi'(\langle A \rightarrow \alpha \rangle)),$$
  $$(\langle X_1 \rightsquigarrow \beta_1 \rangle, \varphi'(\langle X_1 \rightsquigarrow \beta_1 \rangle)), \ \ldots, \ (\langle X_k \rightsquigarrow \beta_k \rangle, \varphi'(\langle X_k \rightsquigarrow \beta_k \rangle))$$

  such that

  $$\varphi(\langle A \rightarrow \langle \alpha \rightsquigarrow \beta \rangle \rangle) \ = \ \varphi'(\langle A \rightarrow \alpha \rangle) \sqcup \varphi'(\langle X_1 \rightsquigarrow \beta_1 \rangle) \sqcup \ldots \sqcup \varphi'(\langle X_k \rightsquigarrow \beta_k \rangle). \qquad \Box$$

Definition 8.31 (*parse tree*)
Let $\mathcal{G}$ be a unification grammar, $\underline{a}_1 \ldots \underline{a}_n$ a string in W^*. A *parse tree* for $\underline{a}_1 \ldots \underline{a}_n$ is an adequately decorated tree of the form

$$(\langle S \rightsquigarrow \underline{a}_1 \ldots \underline{a}_n \rangle, \ \varphi(\langle S \rightsquigarrow \underline{a}_1 \ldots \underline{a}_n \rangle))$$

with $\varphi(\langle S \rightsquigarrow \underline{a}_1 \ldots \underline{a}_n \rangle) \neq \bot$. $\Box$

Definition 8.32 (*result*)
Let $(\langle S \rightsquigarrow \underline{a}_1 \ldots \underline{a}_n \rangle, \ \varphi(\langle S \rightsquigarrow \underline{a}_1 \ldots \underline{a}_n \rangle))$ be a parse for the sentence $\underline{a}_1 \ldots \underline{a}_n$. The feature structure

$$\varphi(S) \ = \ \varphi(\langle S \rightsquigarrow \underline{a}_1 \ldots \underline{a}_n \rangle)|_S$$

is called a *result* of the sentence. $\Box$

In context-free parsing, parse trees are delivered as results. For unification grammars, it is assumed that the feature structure of the sentence symbol S contains all relevant information. The parse tree is not an interesting object as such, it serves only to compute $\varphi(S)$. Hence we can rephrase the parsing problem as follows.

The *parsing problem*, given sentence $\underline{a}_1 \ldots \underline{a}_n \in W^*$ and a grammar $\mathcal{G}$, is to find all results $\varphi(S)$.

Unlike the context-free case, we can also define a reversed problem.[7]

The *generation problem*, given a grammar $\mathcal{G}$ and a feature structure $\varphi(S)$, is to find a sentence $\underline{a}_1 \ldots \underline{a}_n \in W^*$ for which $\varphi(S)$ is a result.

In principle it should be possible to use a single unification grammar both for parsing and generation. If a grammar is to be used in both directions, it must be guaranteed that both the parsing algorithm and the generation algorithm halt. A unification grammar that is designed for use in a parser typically will not halt when used for generation. *Reversible* unification grammars, that can be used in either direction, are studied in by Appelt [1987], Shieber [1988], Shieber et al. [1990], Gerdemann [1991], and van Noord [1993].

8.6 Composition of decorated trees

In 8.5 we have defined what a valid parse tree is, but not yet how such a tree can be computed. We will now define an operator for tree composition. Using this operator, one can create ever larger and larger trees from the initial trees based on grammar productions and lexicon. Thus, in the framework of Chapter 2, we have a primordial soup populated with adequately decorated trees.

The primordial soup is sound if all parse trees for the sentence that may appear are adequately decorated and complete if all adequately decorated parse trees can be constructed.

We define a decorated tree composition operator $\lhd_i$ and extend that to a nondeterministic operator by dropping the index i. For technical reasons, the context-free tree composition operator is defined slightly differently from the way it was done in Chapter 2. (The difference is merely notational, the trees that can be composed are the same).

Definition 8.33 (*context-free tree composition*)
For a context-free grammar G and any $i \in \mathbb{N}$ a partial function

$$\lhd_i\colon \mathcal{T}rees(G) \times \mathcal{T}rees(G) \longrightarrow \mathcal{T}rees(G)$$

is defined as follows. Let $\tau = \langle X_0 \leadsto X_1 \ldots X_k\rangle$ and $\sigma = \langle Y_0 \leadsto Y_1 \ldots Y_l\rangle$ be context-free trees in $\mathcal{T}rees(G)$. Then

$$\tau \lhd_i \sigma = \begin{cases} \langle X_0 \leadsto X_1 \ldots X_{i-1}\, \langle X_i \leadsto Y_1 \ldots Y_l\rangle\, X_{i+1} \ldots X_k\rangle \text{ if } X_i = Y_0, \\ \text{undefined otherwise.} \end{cases}$$

[7] Wedekind [1988] has given such a definition for the generation problem in Lexical-Functional Grammar

In a more practical interpretation, we interpret $\lhd_i$ as an operator to create new trees from existing trees, rather than as a function. We drop the index i and obtain a nondeterministic operator $\lhd$. $\square$

Definition 8.34 (*decorated tree composition*)
For a feature grammar $\mathcal{G}$ and any $i \in \mathbb{N}$ a partial function

$$\lhd_i : \mathcal{DT}rees(\mathcal{G}) \times \mathcal{DT}rees(\mathcal{G}) \longrightarrow \mathcal{DT}rees(\mathcal{G})$$

is defined as follows. Let $(\tau, \varphi(\tau))$ and $(\sigma, \varphi(\sigma))$ be decorated trees with $\tau = \langle X_0 \leadsto X_1 \ldots X_k \rangle$ and $\sigma = \langle Y_0 \leadsto Y_1 \ldots Y_l \rangle$. Then

$$(\tau, \varphi(\tau)) \lhd_i (\sigma, \varphi(\sigma)) = \begin{cases} \text{undefined} & \text{if } \tau \lhd_i \sigma \text{ is undefined} \\ & \text{or } \varphi(\tau) \sqcup_{X_i = Y_0} \varphi(\sigma) = \bot, \\ & \\ (\tau \lhd_i \sigma, \ \ \varphi(\tau) \sqcup_{X_i = Y_0} \varphi(\sigma)) & \text{otherwise.} \end{cases}$$

As in Definition 8.33 we may drop the index i and interpret $\lhd$ as a nondeterministic operator.
We write $(\tau, \varphi(\tau)) \lhd (\sigma, \varphi(\sigma)) = \bot$ if the composition is not defined for any i. $\square$

The next lemma states that composition of adequately decorated trees yields an adequately decorated tree. This result will not come as a surprise. But to be formally correct it is necessary to state it as a separate result. Adequate decoration was defined inductively by expanding a *production* tree with adequately decorated trees. It follows easily (but not by definition) that arbitrary tree composition of adequately decorated trees yields an adequately decorated tree.

Lemma 8.35
Let $(\tau, \varphi(\tau)) \in \mathcal{DT}rees(\mathcal{G})$ and $(\sigma, \varphi(\sigma)) \in \mathcal{DT}rees(\mathcal{G})$ be adequately decorated trees. If $(\tau, \varphi(\tau)) \lhd (\sigma, \varphi(\sigma)) \in \mathcal{DT}rees(\mathcal{G})$ then $(\tau, \varphi(\tau)) \lhd (\sigma, \varphi(\sigma))$ is also adequately decorated.

Proof: by induction on the size of $(\tau, \varphi(\tau)) \lhd (\sigma, \varphi(\sigma))$.
Let $\tau = \langle A \rightarrow \langle \alpha \leadsto \beta \rangle \rangle$, $\alpha = X_1 \ldots X_k$, $\beta = \beta_1 \ldots \beta_k$ as in Definition 8.30. In the composed tree $\tau \lhd \sigma$, some leaf in some β_i is unified with the root of σ. Let $\varphi'(\langle X_i \leadsto \beta_i \rangle)$ be the adequate decoration of $\langle X_i \leadsto \beta_i \rangle$ from which the adequacy of $\varphi(\tau)$ is derived. Then, using the induction hypothesis, we find that

$$\begin{aligned} & (\langle X_i \leadsto \beta_i \rangle \lhd \sigma, \ \varphi'(\langle X_i \leadsto \beta_i \rangle) \sqcup \varphi(\sigma)) \\ = \ & (\langle X_i \leadsto \beta_i \rangle, \varphi'(\langle X_i \leadsto \beta_i \rangle)) \ \lhd \ (\sigma, \varphi(\sigma)) \end{aligned}$$

is adequate. It is easily verified that $(\tau, \varphi(\tau)) \lhd (\sigma, \varphi(\sigma))$ is obtained by composition of $(\langle A \rightarrow \alpha \rangle, \varphi'(\langle A \rightarrow \alpha \rangle))$ with $(\langle X_1 \leadsto \beta_1 \rangle, \varphi'(\langle X_1 \leadsto \beta_1 \rangle))$, $\ldots$, $(\langle X_{i-1} \leadsto \beta_{i-1} \rangle, \varphi'(\langle X_{i-1} \leadsto \beta_{i-1} \rangle))$, $(\langle X_i \leadsto \beta_i \rangle \lhd \sigma, \ \varphi'(\langle X_i \leadsto \beta_i \rangle) \sqcup \varphi(\sigma))$, $(\langle X_{i+1} \leadsto \beta_{i+1} \rangle, \varphi'(\langle X_{i+1} \leadsto \beta_{i+1} \rangle))$, $\ldots$, $(\langle X_k \leadsto \beta_k \rangle, \varphi'(\langle X_k \leadsto \beta_k \rangle))$, as in Definition 8.30. $\square$

Theorem 8.36 (*correctness of primordial soup for decorated trees*)
A decorated tree $(\tau, \varphi(\tau))$ with $\tau = \langle S \rightsquigarrow \underline{a}_1 \ldots \underline{a}_n \rangle$ that is obtained by tree composition $\lhd$ from decorated trees of the forms

$$(\langle A \rightarrow \alpha \rangle, \varphi_0(A \rightarrow \alpha))) \quad \text{and}$$

$$(\langle a \rightarrow \underline{a}_i \rangle, \varphi(a \rightarrow \underline{a}_i))) \quad \text{with } \varphi(\underline{a}_i) \in \mathcal{L}ex(\underline{a}_i) \text{ and } \varphi(a) \doteq \varphi(\underline{a}_i)$$

is adequate. Moreover, each adequately decorated parse can be constructed from such trees.

Proof.
The soundness (context-free parse trees are adequately decorated) is a direct consequence of Lemma 8.35. It is trivial to prove (with induction on the size of the tree) that *all* adequately decorated trees can be composed, hence completeness follows a fortiori. □

8.7 Parsing schemata for unification grammars

In 8.5 we have introduced unification grammars and 8.6 we have proven that the Primordial Soup framework for decorated trees is sound and complete. Integrating all this into context-free parsing schemata is mainly a matter of notation.

There is, however, a single important difference between parsing schemata for context-free grammars and unification grammars, with far-reaching consequences. In the context-free case any item needs to be recognized only once. When an already recognized item is recognized again, it should be ignored. For unification grammars, in contrast, a single item context-free item can be recognized multiple times, each time with a different decoration. These are to be regarded as different objects. Hence we may face the situation that a parsing schema with only a finite set of valid context-free items may yield infinitely many decorations to these items.

At this very abstract level we will not worry about infinitely many decorations for a single context-free item. There are various ways to construct parsing algorithms that recognize only a relevant finite subset of valid decorated items. This will be discussed at more length in Chapter 9.

We will first formulate a parsing schema **UG** that formalized what we did in Section 7.2: Constituents are recognized purely bottom-up. This can be regarded as the canonical parsing schema for unification grammars.

A domain of items can be defined by adding feature structures to the usual CYK items. We could write

$$\mathcal{I}_{UG} \;=\; \{[(X, \varphi(X)), i, j] \;\mid\; X \in V \;\wedge\; 0 \leq i \leq j \;\wedge\; \varphi(X) \neq \bot\}$$

where $\varphi(X)$ is obtained by restricting the composite feature structure of the tree $\langle X \rightsquigarrow \underline{a}_{i+1} \ldots \underline{a}_j \rangle$ to the features of the top node. Throughout the remainder of this chapter items are decorated with feature structures, therefore we do not need to mention $\varphi(X)$ explicitly in the notation of an item. Hence we write $[X, i, j]$ as usual, rather than $[(X, \varphi(X)), i, j]$.

The hypotheses represent all feature structures offered by the lexicon for all words in the sentence:

$$H \; = \; \{[a, j-1, j] \;\mid\; \varphi(a) \in \mathcal{L}ex(\underline{a}_j)\}. \tag{8.4}$$

Schema 8.37 (UG)
It is obvious, however, that deduction steps for productions with larger right-hand sides can be added in similar fashion.
For an arbitrary unification grammar $\mathcal{G} \in \mathcal{UG}$ we define a parsing system $\mathbb{P}_{UG} = \langle \mathcal{I}_{UG}, H, D_{UG}\rangle$ by

$$\begin{aligned}
\mathcal{I}_{UG} &= \{[X, i, j] \mid X \in V \wedge 0 \leq i \leq j \wedge \varphi(X) \neq \bot\};\\
\mathcal{D}^{\geq 1} &= \{[X_1, i_0, i_1], \ldots, [X_k, i_{k-1}, i_k] \vdash [A, i_0, i_k]\\
&\qquad \mid A{\rightarrow}X_1 \ldots X_k \in P \wedge k \geq 1 \wedge\\
&\qquad \varphi(A) = (\varphi_0(A{\rightarrow}X_1 \ldots X_k) \sqcup \varphi(X_1) \sqcup \ldots \sqcup \varphi(X_k))|_A\},\\
D^{\varepsilon} &= \{\vdash [A, j, j] \mid A{\rightarrow}\varepsilon \in P \wedge \varphi(A) = \varphi_o(A{\rightarrow}\varepsilon)\},\\
D_{UG} &= D^{\geq 1} \cup D^{\varepsilon};
\end{aligned}$$

and H as in (8.4).

Many unification grammars that have been written to cover (parts of) natural languages have only productions that are unary or binary branching. In that case, the definition of D can be simplified to:

$$\begin{aligned}
D^{(1)} &= \{[X, i, j] \vdash [A, i, j]\\
&\qquad \mid A{\rightarrow}X \in P \wedge \varphi(A) = (\varphi_0(A{\rightarrow}X) \sqcup \varphi(X))|_A\},\\
D^{(2)} &= \{[X, i, j], [Yj, k] \vdash [A, i, k] \mid A{\rightarrow}XY \in P \wedge\\
&\qquad \varphi(A) = (\varphi_0(A{\rightarrow}XY) \sqcup \varphi(X) \sqcup \varphi(Y))|_A\},\\
D_{UG} &= D^{(1)} \cup D^{(2)}.
\end{aligned}$$

Sets of deduction steps $D^{(k)}$ for other values of k can be added likewise. □

It is not necessarily the case that the parsing schema **UG** yields a finite set of decorated items for an arbitrary grammar and sentence; even worse, the parsing problem for an arbitrary unification grammar is undecidable. Several sufficient conditions that guarantee finiteness of the **UG** schema are known from the literature,[8] but no general necessary and sufficient condition is known. Hence we

[8]The *off-line parsability constraint* [Bresnan and Kaplan, 1982] and the stronger notion of *depth-boundedness* [Haas, 1989] guarantee a finiteness.

simply *assume* that a grammar $\mathcal{G}$ has been defined in such a way that the parsing schema **UG** will halt. For unification grammars designed for parsing natural languages this does not seem to be problem. The underlying idea is that the *meaning* of a sentence, that will be captured somewhere in the result, is derived compositionally from the meaning words, via intermediate constituents; there is little reason to write a grammar such that ever more meaning is added to the same constituent.

In the sequel, we will assume that a unification grammar $\mathcal{G}$ has the property that for any string only a finite number of valid decorated items exists. How the grammar writer guarantees that this is the case (for example by making sure that one of the sufficient conditions is kept) is of no concern to us here. When we discuss other parsing schemata, the finiteness issue will come up again. Adding other fancy kinds of deduction steps — notably top-down prediction of features — may jeopardize the finiteness. In such a case we will show for a newly defined schema **P** that *if* a parsing system $\mathbf{UG}(\mathcal{G})$ halts, then $\mathbf{P}(\mathcal{G})$ will also halt. In other words, the finiteness in bottom-up direction is the responsibility of the grammar writer, whereas the finiteness in top-down direction is the responsibility of the parser constructor.

Earley-type parsers for unification grammars that incorporate top-down prediction are discussed, among others, by Shieber [1985a], Haas [1989], and Shieber [1992]. In Chapter 11 a *head-driven* parsing schema will be defined that starts parsing those words that can be expected to yield features that are most restrictive for top-down prediction.

We will now look at an Earley parser, formalizing what has been informally explained in Section 7.1. A domain of items for the Earley schema is properly described by

$$\begin{aligned}\mathcal{I}_{Earley(UG)} = \{[(A{\rightarrow}\alpha{\bullet}\beta, \varphi(A{\rightarrow}\alpha{\bullet}\beta)), i, j] \mid\ & A{\rightarrow}\alpha\beta \in P \ \wedge\ 0 \leq i \leq j \ \wedge \\ & \varphi_0(A{\rightarrow}\alpha\beta) \sqsubseteq \varphi(A{\rightarrow}\alpha{\bullet}\beta) \ \wedge \\ & \varphi(A{\rightarrow}\alpha{\bullet}\beta) \neq \bot \quad \};\end{aligned} \tag{8.5}$$

In order to simplify the notation, we attach identifiers to items. When an item is subscripted with a symbol $\xi, \eta, \zeta, \ldots$, this symbol can be used in the remainder of the expression to identify the item. Moreover, we write $\varphi(\xi)$ for the feature structure $\varphi(A{\rightarrow}\alpha{\bullet}\beta)$ of an item $[(A{\rightarrow}\alpha{\bullet}\beta, \varphi(A{\rightarrow}\alpha{\bullet}\beta)), i, j]_\xi$. Furthermore, as with the CYK items, we do not mention the feature structure explicitly in the item. Thus we simplify (8.5) to

$$\begin{aligned}\mathcal{I}_{Earley(UG)} = \{[A{\rightarrow}\alpha{\bullet}\beta, i, j]_\xi \mid\ & A{\rightarrow}\alpha\beta \in P \ \wedge\ 0 \leq i \leq j \ \wedge \\ & \varphi_0(A{\rightarrow}\alpha\beta) \sqsubseteq \varphi(\xi) \wedge\ \varphi(\xi) \neq \bot \quad \};\end{aligned} \tag{8.6}$$

Another useful notational convention is the following. Rather than writing $\varphi(\xi)|_X$ for the feature structure of X derived from some composite feature structure within an item ξ, we write $\varphi(X_\xi)$.

Schema 8.38 (Earley(UG))
For an arbitrary unification grammar $\mathcal{G} \in \mathcal{UG}$ a parsing system $\mathbb{P}_{Earley(UG)} = \langle \mathcal{I}_{Earley(UG)}, H, D_{Earley(UG)} \rangle$ is defined by $\mathcal{I}_{Earley(UG)}$ as in (8.6);

$$
\begin{aligned}
D^{Init} &= \{ \vdash [S \rightarrow \bullet\gamma, 0, 0]_\xi \mid \varphi(\xi) = \varphi_0(S \rightarrow \gamma) \}, \\
D^{Scan} &= \{ [A \rightarrow \alpha \bullet a\beta, i, j]_\eta, [a, j, j+1]_\zeta \vdash [A \rightarrow \alpha a \bullet \beta, i, j+1]_\xi \\
&\qquad \mid \varphi(\xi) = \varphi(\eta) \sqcup \varphi(a_\zeta) \}, \\
D^{Compl} &= \{ [A \rightarrow \alpha \bullet B\beta, i, j]_\eta, [B \rightarrow \gamma \bullet, j, k]_\zeta \vdash [A \rightarrow \alpha B \bullet \beta, i, k]_\xi \\
&\qquad \mid \varphi(\xi) = \varphi(\eta) \sqcup \varphi(B_\zeta) \}, \\
D^{Pred} &= \{ [A \rightarrow \alpha \bullet B\beta, i, j]_\eta \vdash [B \rightarrow \bullet\gamma, j, j]_\xi \\
&\qquad \mid \varphi(\xi) = \varphi(B_\eta) \sqcup \varphi_0(B \rightarrow \gamma) \}, \\
D_{Earley(UG)} &= D^{Init} \cup D^{Scan} \cup D^{Compl} \cup D^{Pred};
\end{aligned}
$$

and H as in (8.4). □

A unification grammar $\mathcal{G}$ for which $\mathbf{UG}(\mathcal{G})$ is finite, may cause an infinite number of top-down predictions. A simple way to solve this (and the standard way to parse a unification grammar with a conventional active chart parser) is to limit the top-down prediction to the context-free backbone and replace D^{Pred} by

$$D^{Pred'} = \{ \ [A \rightarrow \alpha \bullet B\beta, i, j]_\eta \vdash [B \rightarrow \bullet\gamma, j, j]_\xi \mid \varphi(\xi) = \varphi_0(B \rightarrow \gamma) \ \}.$$

It is not difficult to show that the modified Earley schema yields only finitely many different decorated items if the **UG** schema is known to do so. In Chapter 9 we will investigate more sophisticated techniques to prevent infinitely many decorations for a single context-free item.

We have given two examples of parsing schemata for unification grammars. It is clear that other context-free parsing schemata can be extended with feature structures in similar fashion.

8.8 The example revisited

We return to the example of Section 7.2 and show how the schema **Earley(UG)** can be used to parse our example sentence. The lexicon and productions for the cat catches a mouse were shown in figures 7.1 and 7.2 on pages 144 and 145. In a PATR-style grammar, the composite feature structures φ_0 are typically denoted by a constraint set. Here we will represent all feature structures, single and composite, by AVMs.

In an Earley item of the form $[A \rightarrow \alpha \bullet \beta, i, j]$, we are interested only in the features of A and β. Features of A will be used to transfer information upwards through a parse tree (when an item $[A{\rightarrow}\alpha\beta\bullet, i, k]$ is used at some later stage as the right operand of a *predict* step). Features of β that are known already are used as a filter to guarantee that β will be of "the right kind" in whatever sense imposed by those features. The features of α need not be remembered. Features of α that are of interest for the remainder of the parsing process will have been shared with A or β, other features are irrelevant. Our purpose, here, is to construct a resulting feature for S, rather than a context-free parse.

We start with an item $[S{\rightarrow}\bullet NP\ VP, 0, 0]$, supplied with the features from $\varphi_0(S{\rightarrow}NP\ VP)$. The decorated item is shown in Figure 8.4.

$$[S{\rightarrow}\bullet NP\ VP, 0, 0]$$

$$\begin{array}{lcl}
S & \longmapsto & \begin{bmatrix} cat & : S \\ head & : \boxed{1} \end{bmatrix} \\[2ex]
NP & \longmapsto & \boxed{2}\begin{bmatrix} cat : NP \end{bmatrix} \\[2ex]
VP & \longmapsto & \begin{bmatrix} cat & : VP \\ head & : \boxed{1}\,[\,] \\ subject & : \boxed{2} \end{bmatrix}
\end{array}$$

Figure 8.4: The initial item

No features are predicted for the subject (other than that its category should be NP). Hence, an item $[NP{\rightarrow}\bullet {*det}\ {*n}, 0, 0]$ is predicted that is decorated with $\varphi_0(NP{\rightarrow}{*det}\ {*n})$. For the sake of brevity we skip the deduction steps

$$[NP{\rightarrow}\bullet {*det}\ {*n}, 0, 0],\ [{*det}, 0, 1] \vdash [NP{\rightarrow}{*det}\bullet {*n}, 0, 1],$$

$$[NP{\rightarrow}{*det}\bullet {*n}, 0, 1],\ [{*n}, 1, 2] \vdash [NP{\rightarrow}{*det}\ {*n}\bullet, 0, 2];$$

the reader may verify that the decorated item $[NP{\rightarrow}{*det}\ {*n}\bullet, 0, 2]$ as displayed in Figure 8.5 is obtained. A *complete* step combines the items of Figures 8.4 and 8.5 into a decorated item $[S{\rightarrow}NP\bullet VP, 0, 2]$ as shown in Figure 8.6. The features of the NP have been included in the VP through coreferencing.

From Figure 8.6 we predict an item $[VP{\rightarrow}\bullet {*v}\ NP, 2, 2]$, as shown in Figure 8.7. The *subject* feature that is shared between VP and $*v$ causes the subject information to be passed down to the verb. Consequently, a verb can be accepted only if it allows a subject in third person singular. This is indeed the case for

$[NP{\rightarrow}{*}det\ {*}n{\bullet}, 0, 2]$

$$NP \longmapsto \begin{bmatrix} cat : NP \\ head : \begin{bmatrix} agr : \begin{bmatrix} number : singular \\ person : third \end{bmatrix} \\ trans : \begin{bmatrix} pred : \mathsf{cat} \\ det : + \end{bmatrix} \end{bmatrix} \end{bmatrix}$$

Figure 8.5: A completed *NP*

$[S{\rightarrow}NP{\bullet}VP, 0, 2]$

$$S \longmapsto \begin{bmatrix} cat : S \\ head : \boxed{1} \end{bmatrix}$$

$$VP \longmapsto \begin{bmatrix} cat : VP \\ head : \boxed{1}\ [\ ] \\ subject : \begin{bmatrix} cat : NP \\ head : \begin{bmatrix} agr : \begin{bmatrix} number : singular \\ person : third \end{bmatrix} \\ trans : \begin{bmatrix} pred : \mathsf{cat} \\ det : + \end{bmatrix} \end{bmatrix} \end{bmatrix} \end{bmatrix}$$

Figure 8.6: *Complete* applied to Figures 8.4 and 8.5

$$[VP \rightarrow \bullet {*v}\ NP, 2, 2]$$

$$
\begin{array}{lll}
VP & \longmapsto & \begin{bmatrix} cat & : VP \\ head & : \boxed{1} \\ subject & : \boxed{2} \end{bmatrix} \\
{*v} & \longmapsto & \begin{bmatrix} cat & : {*v} \\ head & : \boxed{1}\ [\ ] \\ subject & : \boxed{2} \begin{bmatrix} cat & : NP \\ head & : \begin{bmatrix} agr & : \begin{bmatrix} number : singular \\ person : third \end{bmatrix} \\ trans & : \begin{bmatrix} pred : \mathsf{cat} \\ det : + \end{bmatrix} \end{bmatrix} \end{bmatrix} \\ object & : \boxed{3} \end{bmatrix} \\
NP & \longmapsto & \boxed{3} \begin{bmatrix} cat : NP \end{bmatrix}
\end{array}
$$

Figure 8.7: *Predict* applied to Figure 8.6

$$[VP \rightarrow {*}v \bullet NP, 2, 3]$$

$$
\begin{array}{lll}
VP & \longmapsto & \begin{bmatrix}
cat & : & VP \\
head & : & \begin{bmatrix}
tense : & present \\
agr \;\; : & \boxed{1} \\
trans : & \begin{bmatrix} pred : & \mathsf{catch} \\ arg1 : & \boxed{2} \\ arg2 : & \boxed{3} \end{bmatrix}
\end{bmatrix} \\
subject & : & \begin{bmatrix}
cat \;\; : & NP \\
head : & \begin{bmatrix}
agr \;\; : & \boxed{1} \begin{bmatrix} number : & singular \\ person \;\; : & third \end{bmatrix} \\
trans : & \boxed{2} \begin{bmatrix} pred : & \mathsf{cat} \\ det \;\; : & + \end{bmatrix}
\end{bmatrix}
\end{bmatrix} \\
object & : & \boxed{3}
\end{bmatrix} \\
NP & \longmapsto & \begin{bmatrix}
cat \;\; : & NP \\
head : & \begin{bmatrix} trans : & \boxed{3} \end{bmatrix}
\end{bmatrix}
\end{array}
$$

Figure 8.8: *Scan* applied to Figure 8.7 and "catches" on page 144

the initial item $[*v, 2, 3]$, decorated with the lexicon entry for catches on page 144. Hence we obtain the item $[VP \rightarrow *v \bullet NP, 2, 3]$ with a decoration as shown in Figure 8.8. The $*v$ entry has been deleted, as its salient features are also contained in the VP feature structure. Note that $\langle NP\ head\ trans \rangle$ is now coreferenced with $\langle VP\ head\ trans\ arg2 \rangle$, through the coreference in the (no longer visible) feature structure of the verb.

We can continue to deduce decorated items in similar fashion. It is left to the reader to verify that application of the deduction steps

$$[VP \rightarrow *v \bullet NP, 2, 3] \ \vdash\ [NP \rightarrow \bullet *det\ *n, 3, 3],$$

$$[NP \rightarrow \bullet *det\ *n, 3, 3],\ [*det, 3, 4] \ \vdash\ [NP \rightarrow *det \bullet *n, 3, 4],$$

$$[NP \rightarrow *det \bullet *n, 3, 4],\ [*n, 4, 5] \ \vdash\ [NP \rightarrow *det\ *n \bullet, 3, 5],$$

$$[VP \rightarrow *v \bullet NP, 2, 3],\ [NP \rightarrow *det\ *n \bullet, 3, 5], \ \vdash\ [VP \rightarrow *v\ NP \bullet, 2, 5],$$

$$[S \rightarrow NP \bullet VP, 0, 2],\ [VP \rightarrow *v\ NP \bullet, 2, 5] \ \vdash\ [S \rightarrow NP\ VP \bullet, 0, 5]$$

results in a decorated final item as shown in Figure 8.9.

$$[S \rightarrow NP\ VP \bullet, 0, 5]$$

$$S \longmapsto \begin{bmatrix} \mathit{cat} & : S \\ \mathit{head}: & \begin{bmatrix} \mathit{tense}: & \mathit{present} \\ \mathit{agr}: & \begin{bmatrix} \mathit{number}: & \mathit{singular} \\ \mathit{person}: & \mathit{third} \end{bmatrix} \\ \mathit{trans}: & \begin{bmatrix} \mathit{pred}: & \mathsf{catch} \\ \mathit{arg1}: & \begin{bmatrix} \mathit{pred}: & \mathsf{cat} \\ \mathit{det}: & + \end{bmatrix} \\ \mathit{arg2}: & \begin{bmatrix} \mathit{pred}: & \mathsf{mouse} \\ \mathit{det}: & - \end{bmatrix} \end{bmatrix} \end{bmatrix} \end{bmatrix}$$

Figure 8.9: A final item

8.9 Other grammar formalisms

We will briefly mention some different kinds of unification grammars and then discuss the related formalisms of attribute grammars and affix grammars.

The earliest type of unification grammar is Definite Clause Grammar (DCG), defined by Pereira and Warren [1980]. DCG is based on *terms* rather than feature structures. It is inextricably linked with the programming language Prolog [Clocksin and Mellish, 1981]. DCG, basically, offers some additional syntactic sugar for encoding grammars directly into Prolog.

In the last decade, a variety of grammar formalisms based on feature structure unification has emerged. The Computational Linguistics community has been enriched with Lexical-Functional Grammar (LFG) [Kaplan and Bresnan, 1982], Functional Unification Grammar (FUG) [Kay, 1979, 1985], Generalized Phrase Structure Grammar (GPSG) [Gazdar et al., 1985], PATR[9] [Shieber 1986], Categorial Unification grammar (CUG) [Uszkoreit 1986], Unification Categorial grammar (UCG) [Zeevat et al., 1987], Head-Driven Phrase Structure Grammar (HPSG) [Pollard and Sag, 1987, 1993], Unification-based Tree Adjoining Grammars (UTAG) [Vijaj-Shanker et al., 1991]. This list is not exhaustive.

The word "grammar" that appears in all these formalisms, has subtly different meanings in different cases. On the one hand, one can see grammar as a formalism that has no meaning *per se*, but can be used to encode grammars for whatever purpose. Typical examples of this class are DCG, FUG and PATR. On the other hand, one can interpret grammar as a description of phenomena that occur in natural language. Such a grammar does not only offer a formalism but, more importantly, also a linguistic *theory* that is expressed by means of that formalism. Typical examples of this class are LFG, GPSG and HPSG. We will further discuss this in Chapter 15.

The feature structure formalism that we have used here is taken from the 1986 version of PATR (with exception of the extension to composite feature structures). It was designed by Shieber to be the most simple feature structure formalism, containing only the bare essentials. A lot of bells and whistles can be added, of course. The use of *lists*, which is admittedly cumbersome in PATR notation, can be simplified by introducing a special list notation. We have used untyped feature structures: any feature can have any value. In a *typed* feature structure formalism, the value of a feature is restricted to particular types specifically defined for that feature. A useful extension to increase the efficiency of unification grammar parsing is coverage of *disjunctive feature structures*. We will come back to this in Chapter 9.

We have stipulated — as in PATR — that feature graphs contain no cycles. The practical reason is that it simplifies the unification algorithms, and cyclic feature structures seem to have little linguistic relevance. In HPSG, the feature formalism does not explicitly ban cycles, but in the 1988 version [Pollard and Sag, 1987] they simply did not occur in any of the types prescribed for HPSG grammars. The

[9] The formalism is called PATR-II, to be precise, and quite different from a first version of PATR that has fallen into oblivion (and hence the letters "PATR" in PATR-II no longer form an acronym).

1993 version of HPSG [Pollard and Sag, 1993], however, has somewhat different types and found an application for cyclic structures. Some linguists argue that the head of a noun phrase is the determiner, rather than the noun (the so-called *DP hypothesis*). In the latest version of HPSG, this matter is solved by letting both the determiner and the noun regard themselves as head of the *NP* and each other as a subordinate constituent. Hence either constituent is subordinate to a subordinate structure of itself.

Unification grammars are related to *attribute grammars*, introduced by Knuth [1968, 1971], that have been used in compiler construction for 25 years. There are some basic differences between attribute grammars and unification grammars, but from a formal point of view there is little objection to call both constraint-based formalisms. The difference between both formalisms is to a large extent a difference in culture: attribute grammars are typically used by computer scientists to denote the semantics of programming languages, while unification grammars are typically used by computational linguists to capture syntactic and semantic properties of natural languages.

Attribute grammars stem from the age that higher programming languages all were *imperative* languages. The basic statement is the assignment: a value, obtained from evaluating an expression, is assigned to an identifier. Expressions can be functions (i.e. sub-programs computing a value) of arbitrary sophistication. Within the imperative programming paradigm, therefore, it is the most natural approach to define attributes of a constituent as functions of other attributes of other constituents. The constraints in an attribute grammar can be thought of assignments:[10]

$$\langle attribute \rangle := \langle expression \rangle$$

where $\langle expression \rangle$ is a function of attributes of other symbols in the same production.

Unification grammars, in comparison draw heavily upon the *declarative* programming style as incorporated in Prolog. A Prolog clause `foo(X,Y)` specifies the relation between `X` and `Y`. If `X` is instantiated then `foo` can be used to assign a value to a variable `Y`, and reversed, if `Y` is instantiated then a variable `X` can get a value by calling `foo`[11]. Similarly, in unification grammars we specify (commutative) equations that have to be true. In which order the features have to

[10] One could use attribute grammars also within the functional programming paradigm. Lazy evaluation can be used to solve some dependency problems easier and more elegantly than in the imperative paradigm, but the central notion of functional dependency remains.

[11] In the actual practice of Prolog programming, however, few clauses do really allow this. There is a difference between specification and computation: it is very well possible that the Prolog gets stuck in an infinite loop of the "wrong" argument is uninstantiated. This is similar to the fact that a unification grammar designed for parsing typically can't be used for generation, although the general formalism is bidirectional.

be computed is irrelevant, it is not even possible to express such considerations within the formalism.

Research on attribute grammars, therefore, tends to focus on other issues than research on unification grammars. A classical issue is that of *noncircularity*: if there is a circle of attributes in a parse tree that are all functionally dependent on each other, then it is impossible to compute a decoration for the tree. An often used sufficient (but not necessary) condition is that of *L-attributedness*. An attribute grammar is L-attributed, informally speaking, if all attributes can be computed in a single pass in a top-down left-to-right walk through a context-free parse tree. A subclass that is particularly useful in compiler construction is the class of *LR-attributed* grammars. These, roughly speaking, allow the attributed to be computed on the fly by an LR parser. The literature contains a host of different parsing algorithms for LR-attributed grammars. See, e.g., Jones and Madsen [1980], Pohlmann [1983], Nakata and Sassa [1986], Sassa et al. [1987], and Tarhio [1988]). Each one defines a particular class of grammars on which it is guaranteed to work correctly. All these classes are subtly different, however, because they depend on the guts of the proposed algorithm. A taxonomy is presented by op den Akker, Melichar and Tarhio [1980]. A fundamental treatment of attribute evaluation during *generalized* LR parsing (cf. Chapter 12) is given by Oude Luttighuis and Sikkel [1992, 1993].

"There are no fundamental differences between affix grammars [...] and attribute grammars [...]", Koster [1991a] remarks in an article on affix grammars for programming languages. "The two formalisms differ in origin and notation, but they are both formalizations of the same intuition: the extension of parsers with parameters".

Affix grammars are a particular kind of *two-level* or *van Wijngaarden* grammars [van Wijngaarden, 1965], and were formalized by Koster [1971]. One can see the context-free productions of an affix grammar as *production schemata*, defining sets of productions for different combinations of affix values that can be attributed to the symbols involved in the production. Hence, even though grammars written as an affix grammar can be automatically translated to attribute grammars, and reversed, the basic formalism of affix grammars is more general, because its lacks the predominant concern with functional dependency.

Unification grammars with a *finite* feature lattice can be formulated directly as affix grammars (so-called Affix Grammars over a Finite Lattice (AGFL), see Koster [1991b] for a simple introduction). Typically linguistic phenomena that can be modelled with finite feature lattices, or a finite domain of typed feature structures, are conjugation (i.e. the different forms of a verb) and declination (forms of nouns, adjectives, etc.,)

The main difference between affix grammars and both attribute grammars and unification grammars is again a cultural one. The school of affix grammars has its own followers and its own formalism, but the work done in that area can be formulated in terms of attribute grammars or unification grammars as well.

8.10 Related approaches

Some explicit parsing algorithms for unification grammars have been given in the literature. Haas [1989] gives a GHR algorithm (i.e. Graham, Harrison, and Ruzzo's optimization of Earley's algorithm, cf. Example 6.18) for grammars based on *terms*. Shieber [1992] gives an Earley parser for a general class of unification grammars, rather than just the PATR-formalism. The notation of Shieber [1992] — as opposed to the PATR variant of [Shieber, 1986], on which our treatment of unification grammars is based — allows for explicit control of feature percolation within *productions*; a production $A \rightarrow X_1 \ldots X_k$ is a structure with features $0, \ldots, k$ that address the separate constituents. Our concept of multi-rooted feature structures for describing feature sharing between different objects is more general, because it can deal with arbitrary object structures.

The subject discussed here has some clear links with Shieber [1992], but we have taken a rather different perspective. Whereas Shieber gives a most general account of unification grammars and discusses only a single parsing algorithm, we have used just a simple unification grammar but given a formalism that allows to specify arbitrary parsing algorithms in a precise but conceptually clear manner.

8.11 Conclusion

The main contribution of this chapter is the combination of parsing schemata and unification grammars in a single framework. Using the proper notation, parsing schemata for unification grammars are a straightforward extension of context-free parsing schemata. The hardest task was in fact to come up with a proper notation. Both parsing algorithms and unification grammars are complex problem domains on their own. In order to combine them into a single framework, a large conceptual machinery and a rich notation is needed. It is for good reason that most articles in the literature are specific in one domain, and informal in the other.

Context-free parsing is a *computational* problem area. A parse tree can be defined as an object that satisfies certain properties, but the only way to find these properties for a given sentence is to actually *construct* the parse tree. From this point of view, attribute grammars are the more natural way to extend context-free parsing with constraints and semantic functions. Decorating a tree with attributes (whether simultaneously or in a second pass) is indeed application of functions.

The literature on unification grammars, on the other hand, has a strong focus on the *declarative* character of such a grammar. One describes the constraints that are implied by the grammar, and the properties of individual words in the lexicon. The theory leans heavily on logic, hence the prime operational concern is that constraints can be expressed in a subset of first-order logic that allows automatic constraint resolution. This being proven, one can leave the act of *satisfying* the constraints to an appropriate machine. From this point of view it makes sense to concentrate on the static aspects of the grammar, rather than on the dynamic aspects of how to construct a parse.

The dynamics of unification and resolution *sec* have been studied extensively in the literature. It constitutes an auxiliary domain that is used as a tool in the construction of parsers for unification grammars, often in the form of the Prolog programming language. We have added a simple formalism that allows explicit specification of the dynamics of feature structure propagation in parsing algorithms.