The Grammar of German in the Grammatical Framework (Draft, January 9, 2025)

Hans Leiß
leiss@cis.uni-muenchen.de
Bereiter Anger 10
81541 München

Contents

Introduction

Why write this? As documentation of the existing `Lang` and `LangGer` for outsiders of `GF`, such as linguists, computer scientists and programmers. As explanation of principal limits of `Lang` and current limits of `LangGer` for `GF`-users. To provide a list of necessary improvements for `GF`-developers.

1. Syntax of Natural Languages

For non-linguists, in particular for computer scientist or logicians interested in natural language, we sketch some basic differences between formal languages and natural languages.

We do not discuss the usage of language, like utterances of expressions by a speaker or writer and the effects of such utterances on hearers or readers, but just the form of expressions. For different possible intentions behind an utterance, like asking, informing, commanding, there are typical forms of expressions, but there is no clear-cut correspondence. We only discuss the form of linguistic expressions, as they are written in standard writing conventions, without the often important aspects added by intonation. The interpretation of expressions is situation-dependent and also ignored, although occasionally some comparison with the semantics of formal languages may be made.

1.1. Grammatical Notions

- kinds (types) *categories* of expressions

  In formal language theory, a category A of expressions is interpreted as a set of strings $A \subseteq \{ w \mid w \in L^* \}$ over an alphabet (or lexicon) L.

  Todo 1: Relate concatenation $A \cdot B$ of string sets with function application and division A/B and $B\backslash A$ with incomplete expressions. In formal language theory, concatenation is the only construction of type $A \to B \to (A \cdot B)$. In categorial grammar, concatenation also has types $A \to A\backslash B \to B$ and $A/B \to B \to A$, for example.

- grammatical *constructions* of expressions (from immediate constituents)

  Syntactic constructions are typed functions $f : A_1 \times \ldots \times A_n \to B$ or $f : A_1 \to (A_2 \to (\ldots (A_n \to B) \ldots))$. The more precise a grammar should be, the more categories and constructions are needed. With few categories, the constructions will typically *overgenerate*, i.e. produce strings $w = f(w_1, \ldots, w_n) : B$ from strings $w_1 : A_1, \ldots, w_n : A_n$ which are actually not in use. (For example, most binary verbs combine with noun phrases in singular or plural as object, but some, e.g. *to collect*, impose a number restriction on their argument; thus, a construction $f : V2 \to NP \to VP$ overgenerates by giving e.g. *collect a (single) stamp*.) If a grammar's construction $f : A \to B$ overgenerates, the intended linguistic construction is a *partial* function of type $A \to B$, and A is wider than the intended set of possible arguments of f. There is a trade-off between preciseness of language description and grammar size in terms of number of categories and constructions.

- grammatical *functions* or *relations* between expressions.

  The n different immediate constituents $e_1, \ldots, e_n$ of an expression e stand in n different (functional) *syntactic relations* $f_1, \ldots, f_n$ to the expression e. We say that e_i *is the* f_i *of* e, or "e_i *realizes the function* f_i *in* e". For example, a noun phrase may be the subject of a clause, or realize the subject function in the clause. Conversely, if the subject of a

clause is a noun phrase, we speak of a ***nominal*** subject, if it is an infinitive, we speak of an ***infinitival*** subject, etc.

(Expressions can function as ***complement*** of a verb, noun or adjective in a compound expression. The complements of a verb (or: of a basic clause) are ***subject*** and ***object***. The verb, noun or adjective functions as ***head*** in these *head + complements*-expressions. Complemented words can be extended by ***modifiers***, such as: nouns by attributes, relative clauses, and adverbs; verbs (and basic clauses) by adverbs and adverbial clauses (conditional clause? coordination?); adjectives by adadjectives. Other syntactic functions: *predicate* (and *subject*) in a basic clause, *attributive function* of adjective phrases in noun phrases, *predicative function* of adjective or noun phrases in basic clauses, *adverbial function* of adverbs and adjective phrases in basic clauses and basic noun phrases; *coordinator* or *subordinator* in a conjoined expression?)

Remark 1: A subexpression typically realizes a unique syntactic function or role in its enclosing expression, but sometimes realizes two functions. For example, *Fermat's* in *Fermat's last theorem*, simultaneouly plays the role of a determiner and of a possessive. Similarly, in ACI-constructions, e.g. *wir sahen den Hund die Katze jagen*, the accusative noun phrase *den Hund* is the object of the main verb *sehen* as well as the implicit subject of the embedded verb *jagen*.

basic phrase = *head* combined with all *complements*,
phrase = *basic phrase* combined with *modifiers/attributes*.[1]

Ok for noun phrase and adjective phrase, but difference between verb phrase and clause: the verb phrase misses the subject of its head verb, the clause does not.

- forms of expressions, agreement and rection: forms depending on inflection parameters (declination for nouns, conjugation for verbs); congruence of form parameters, and dependence of parameters from (main verb agrees with subject, or subject governs main verb in finiteness features? *die Leute mögen das nicht. Urlaub zu machen, ist schön.*
- grammatical transformations: mappings between expressions of different categories

Define ***syntactic arity*** or ***complement frame*** to specify the type of possible complement expressions of verbs, nouns and adjectives.

Language in use: utterances, imperatives, indexicals, reflexives and speaker, reciprocals, plural subjects

1.2. Parallellism within a language

- predication/attribution/relativization
- declarative/relative/interrogative clause, NP, Adv
- negation of subsentential phrases
- personal/relative/reflexive pronoun
- word order and intonation

[1] Todo 2: relate these to the head-complement-structures and head-adjunct-structures of HPSG

2. Formal Languages and their Grammars

In Formal Language Theory, a ***formal language*** is a set of ***string***s, i.e. sequences $(a_1, \ldots, a_n)$ of finite lengths n of elements $a_1, \ldots, a_n$ from a finite set Σ of ***letter***s, the ***alphabet*** or ***vocabulary***. The set of all strings over Σ is denoted by Σ^*. The string () of length 0 is called the ***empty string*** and denoted by ϵ. Two strings $u = (a_1, \ldots, a_n)$ and $v = (b_1, \ldots, b_m)$ can be ***concatenated*** to a string $u \cdot v = (a_1, \ldots, a_n, b_1, \ldots, b_m)$ of length $n+m$. The concatenation operation $\cdot$ is associative, i.e. $(u \cdot v) \cdot w = u \cdot (v \cdot w)$ for all strings u, v, w, and has ϵ as neutral element, i.e. $\epsilon \cdot w = w \cdot \epsilon = w$ for all strings w, so that $(\Sigma^*, \cdot, \epsilon)$ is a ***monoid***. It is actually the ***free monoid generated by*** Σ, i.e. no other identities $u = v$ between strings u, v hold except those that follow from associativity of $\cdot$ and neutrality of ϵ. We usually write $a_1 \cdots a_n$ for $(a_1, \ldots, a_n)$ and uv for $u \cdot v$.

The concatenation operation can be lifted from strings to formal languages $A, B \subseteq \Sigma^*$ by

$$A \cdot B = \{\, u \cdot v \mid u \in A \text{ and } v \in B \,\},$$

and with $\{\epsilon\}$ as neutral element gives rise to the ***power set monoid*** $(\mathcal{P}(\Sigma^*), \cdot, \{\epsilon\})$ of all formal languages over Σ. On the set level, besides this multiplicative monoid $(\mathcal{P}(\Sigma^*), \cdot, \{\epsilon\})$ there is the additive, commutative monoid $(\mathcal{P}(\Sigma^*), \cup, \emptyset)$, with set union

$$A \cup B = \{\, u \mid u \in A \text{ or } u \in B \,\}$$

as associative operation and the empty set $\emptyset$ as neutral element. Their combination gives the ***semiring of all formal languages*** over Σ,

$$(S, +, \cdot, 0, 1) = (\mathcal{P}(\Sigma^*), \cup, \cdot, \emptyset, \{\epsilon\}),$$

where in addition to the monoid properties of $(S, +, 0)$ and $(S, \cdot, 1)$ the distributivity and annihilation properties

$$A \cdot (B + C) \cdot D = (A \cdot B \cdot D) + (A \cdot C \cdot D) \quad \text{and} \quad A \cdot 0 \cdot B = 0$$

hold, for all $A, B, C, D \in S$. On the semiring, there is a partial order $\leq$ defined by $A \leq B$ iff $A + B = B$, the subset relation.

The terminology of Formal Language Theory can be misleading when applied to natural languages. First, though the alphabet of a written natural language is finite, its vocabulary often is not: it may contain infinitely many number words, e.g. *neunhunderttausendvierundfünzig* in German. Second, not every concatenation of letters gives a word in the language, and not every concatenation of words gives a sentence or other expression of the language; the concatenation comes with modification of its elements, like spelling modifications or inflectional modifications, so it is not a free operation. Third, not every set of sentences (or other word sequences) over a vocabulary makes a language; a natural language has to be closed under certain variations of sentences and has to exclude others.

While the power set monoid or semiring of all formal languages over Σ is too wide a class to be considered, there are submonoids or subsemirings that are large enough to at least contain the ususal programming languages and the formula languages of mathematical logic.

Since the combination of expressions in natural languages is not free, we now switch from free monoids Σ^* to arbitrary monoids M. Given a monoid $M = (M, \cdot, 1)$, its power set monoid $(\mathcal{P}M, \cdot, \{1\})$, with elementwise product $A \cdot B = \{\, a \cdot b \mid a \in A, b \in B \,\}$ for $A, B \in \mathcal{P}M$, has a submonoid $\mathcal{F}M$ of all ***finite subsets of*** M. As $\emptyset$ is a finite set and the union of two

finite sets is finite, $\mathcal{F}M = (\mathcal{F}M, \cup, \cdot, \emptyset, \{1\})$ is actually a subsemiring of the power set semiring $\mathcal{P}M = (\mathcal{P}M, \cup, \cdot, \emptyset, \{1\})$ of all subsets of M.

The semiring $\mathcal{F}M$ is too small to study formal or natural languages: if languages are closed under some form of iterative or inductive constructions of expressions[2], we cannot avoid infinite languages. The most important extension of $\mathcal{F}M$ is the subsemiring $\mathcal{C}M \subseteq \mathcal{P}M$ of all ***context-free subset***s of M of the power set semiring $\mathcal{P}M$. The set $\mathcal{C}M$ is the smallest set $\mathcal{S} \subseteq \mathcal{P}M$ of subsets of M such that (i) $\mathcal{F}M \subseteq \mathcal{S}$, (ii) if $A, B \in \mathcal{S}$, then $A \cdot B \in \mathcal{S}$ and $A \cup B \in \mathcal{S}$, and (iii) if $n \in \mathbb{N}$ and $p_1(X_1, \ldots, X_n), \ldots, p_n(X_1, \ldots, X_n)$ are built from variables $X_1, \ldots, X_n$ and elements of $\mathcal{S}$ using $\cdot$ and $\cup$, then the least sets $A_1, \ldots, A_n \in \mathcal{P}M$ satisfying

$$\begin{array}{rcl} A_1 & \supseteq & p_1(A_1, \ldots, A_n), \\ & \vdots & \\ A_n & \supseteq & p_n(A_1, \ldots, A_n), \end{array} \tag{1}$$

belong to $\mathcal{S}$. These sets $A_1, \ldots, A_n$ can be obtained by starting from $A_{1,0} = \ldots = A_{n,0} := \emptyset \in \mathcal{F}M \subseteq \mathcal{S}$, and given $A_{1,k}, \ldots, A_{n,k} \in \mathcal{S}$, putting $A_{i,k+1} := p_i(A_{1,k}, \ldots, A_{n,k}) \in \mathcal{S}$ by (ii), and finally showing that the sets $A_i := \bigcup\{ A_{i,k} \mid k \in \mathbb{N} \}$ for $i = 1, \ldots, n$ satisfy (1), for which one first shows $A_{i,k} \subseteq A_{i,k+1}$ by induction on k.

Since the semiring properties hold in $\mathcal{P}M$, any expression $p_i(X_1, \ldots, X_n)$ can be written as a finite sum $\alpha_{i,1} \cup \ldots \cup \alpha_{i,r_i}$ of products $\alpha_{i,j}$ of variables $X_1, \ldots, X_n$ and the members of $\mathcal{S}$ occurring in $p_i(X_1, \ldots, X_n)$. By induction, these members of $\mathcal{S}$ can be assumed to be $\emptyset$ or singleton sets $\{m\}$ with $m \in M$. When M is the free monoid Σ^*, a system $X_1 \supseteq p_1(X_1, \ldots, X_n), \ldots, X_n \supseteq p_n(X_1, \ldots, X_n)$ is therefore nothing else than a ***context-free grammar*** with "*terminal symbols*" Σ, "*nonterminal symbols*" $X_1, \ldots, X_n$, "*start symbol*" X_1 and "*grammar rules*" $X_i \supseteq \alpha_{i,j}$, with $1 \leq i \leq n$ and $1 \leq k \leq r_i$. A grammar rule like $X_i \supseteq \{v_1\}X_1\{v_2\} \cdots X_n\{v_{n+1}\}$ with $v_1, \ldots, v_{n+1} \in \Sigma^*$ is usually written as $v_1 X_1 v_2 \cdots X_n v_{n+1} \to X_i$. It can be seen as a function $f : \Sigma^* \times \ldots \times \Sigma^* \to \Sigma^*$ given by $f(w_1, \ldots, w_n) = v_1 w_1 v_2 \cdots w_n v_{n+1}$, or rather as the restriction $f : A_1 \times \ldots \times A_n \to A_i$ of this function to the sets or types $A_1, \ldots, A_n \subseteq \Sigma^*$ of strings defined by the grammar.

A ***residuated partially ordered monoid*** $(N, \cdot, 1, \leq, \backslash, /)$ is a partially ordered monoid with two binary division operations $\backslash, / : N \times N \to N$ such that for all $k, m, n \in N$,

$$m \cdot k \leq n \iff k \leq m\backslash n \quad \text{and} \quad k \cdot m \leq n \iff k \leq n/m.$$

It follows that $m\backslash n$ is the largest k such that $m \cdot k \leq n$ and n/m is the largest k such that $k \cdot m \leq n$,[3] and if $\cdot$ is commutative, $m\backslash n = m/n$. Since multiplication is monotone,

$$m \cdot m\backslash n \leq n \quad \text{and} \quad n/m \cdot m \leq n.$$

For example, on $(\mathbb{N}, \cdot, 1, \leq)$ with ordinary, commutative multiplication and standard order $\leq$, the two divisions coincide, with e.g. $14/3 = 4 = 3\backslash 14$. On the partially ordered monoid $(\mathcal{P}M, \cdot, \{1\}, \subseteq)$ there are two quotient operations

$$\begin{array}{rcl} A/B & = & \text{the largest } C \in \mathcal{P}M \text{ such that } C \cdot B \subseteq A, \\ B\backslash A & = & \text{the largest } C \in \mathcal{P}M \text{ such that } B \cdot C \subseteq A. \end{array}$$

[2] which of course is an idealization

[3] One says the monotone function $k \mapsto m \cdot k$ is ***residuated*** and has $n \mapsto m\backslash n$ as its ***residual*** function, likewise for $k \mapsto k \cdot m$ and $n \mapsto n/m$.

It is easily seen that $A/B = \{\, m \in M \mid \{m\} \cdot B \subseteq A \,\}$ and $B\backslash A = \{\, m \in M \mid B \cdot \{m\} \subseteq A \,\}$. (The existence and uniqueness of a largest $C \in \mathcal{P}M$ such that $C \cdot B \subseteq A$ follows from the closure of $\mathcal{P}M$ under arbitrary unions. Is $\mathcal{C}M$ a residuated semiring? Is $\{\, \{m\} \mid m \in M, \{m\} \cdot B \subseteq A \,\} \in \mathcal{C}\mathcal{C}M$ and hence it's union $A/B \in \mathcal{C}M$, for $A, B \in \mathcal{C}M$? By Pentus' theorem?)

The ***residual function***s or ***language division***s $B\backslash\cdot$ and $\cdot/B$ play a minor role in formal language theory, but are important for the theory of natural languages.

Todo 3: Movement and partial phrases: Verb phrases as sentences missing an initial (subject) noun phrase in English, i.e. $VP = NP\backslash S$, fronting of object-noun phrase in wh-movement and relative clauses: $RS = RelPron \cdot S/NP$. Yes/No-questions as variations of sentences (by intonation and word order), Wh-questions as sentences with a noun or adverbial phrase substituted by an interrogative noun phrase or interrogative adverbial: $QNP \cdot NP\backslash S \leq QS$, *John's car is broken* $\mapsto$ *whose car is broken*?, or $S/Adv \cdot QAdv \leq QS$ *John arrives today* $\mapsto$ *John arrives when*?

First, consider ***context-free language***s and grammars, then ***categorial grammar***s.

- Explain the `Slash`-categories and their use to describe extraction phenomena.
- complements B are arguments to predicates A/B or $B\backslash A$ and reduce the arity[4]:

$$(A/B) \cdot B \leq A, \qquad B \cdot (B\backslash A) \leq A.$$

- adverbials are pre- resp. post-modifiers P/P resp. $P\backslash P$ of predicates P that don't change the predicate category:[5]

$$P/P \cdot P \leq P, \qquad P \cdot (P\backslash P) \leq P.$$

Other aspects of natural langages (in contrast to formal languages):

- unspecific scope of quantifiers (in NPs) and operators (in adverbials)
- Coordination as abbreviation mechanism (on various levels, not just `Cl`)

2.1. Parallel Multiple Context-Free Grammars (PMCFG)

Developed by Seki e.a. 1991 [10], [5], [7]

- as extension of context-free grammars, with categories of split-strings
- uses of split strings: paradigms and split-constituents
- difficulties to write context-free grammars: possible merging, extractions

Example of non-context-freeness in natural language: *day by day*, *line by line* versus *N by N*.

A ***parallel multiple context-free grammar*** (PMCFG) $G = (\Sigma, N, S, F, P)$ over the monoid M consists of a finite *alphabet* $\Sigma \subseteq M$, a finite set N of *syntactic categories* or *nonterminals* A of *dimension* $d(A) \in \mathbb{N}$, a distinguished *start* or *sentence category* $S \in N$ (of dimension 1),

[4] A/B has arity $n+1$, if A has arity n.

[5] unless we view predicates as modifiable in various dimensions and treat the adverb as dimension argument. But adverbs are largely optional, which complicates the treatment as dimension argument.

a finite set F of *concatenation functions* $f : M^{d_1} \times \ldots \times M^{d_n} \to M^{d_r}$ of dimensions $d(f) = d_1 \times \ldots \times d_n \to d_r$ with $d_1, \ldots, d_n, d_r \in \mathbb{N}$ (depending on f), such that each component of

$$f((w_{1,1}, \ldots, w_{1,d_1}), \ldots, (w_{n,1}, \ldots, w_{n,d_n})) \in M^{d_r}$$

is a product of elements from $\{w_{1,1}, \ldots, w_{n,d_n}\} \cup \Sigma^*$, and a finite set P of *grammar rules* $(A \to f[A_1, \ldots, A_n])$ with $f \in F$ and $A, A_1, \ldots A_n \in N$ such that $d(f) = d(A_1) \times \ldots \times d(A_n) \to d(A)$.

Each concatenation function $f : d_1 \times \ldots \times d_n \to d_r$ lifts to a monotone function $\mathcal{P}(f) : \mathcal{P}(M^{d_1}) \times \ldots \times \mathcal{P}(M^{d_n}) \to \mathcal{P}(M^{d_r})$ on the set level, defined by

$$\mathcal{P}(f)(X_1, \ldots, X_n) = \{\, f(\vec{w}_1, \ldots, \vec{w}_n) \mid \vec{w}_1 \in X_1, \ldots, \vec{w}_n \in X_n \,\} \subseteq M^{d_r}$$

for $X_n \in \mathcal{P}(M^{d_1}), \ldots, X_n \in \mathcal{P}(M^{d_n})$. Therefore, there are least sets $L(A) \subseteq M^{d(A)}$, $A \in N$, such that for all grammar rules $(A \to f[A_1, \ldots, A_n]) \in P$

$$\{\, f(\vec{w}_1, \ldots, \vec{w}_n) \mid \vec{w} \in L(A_1), \ldots, \vec{w}_n \in L(A_n) \,\} \subseteq L(A).$$

Notice that $L(A)$ belongs to $\mathcal{P}(\langle\Sigma\rangle^{d(A)})$ for the submonoid $\langle\Sigma\rangle$ of M generated by Σ.

A syntax rule $(A \to f[A_1, \ldots, A_n])$ can be seen as a typed function symbol $f : A_1 \times \ldots \times A_n \to A$.

Q1: Is the family *pm-*$\mathcal{C}M$ of parallel, multiple context-free subsets of $M^{<\omega}$ a semiring, even a residuated semiring, or why can we use the `Slash`-categories?

Since $\mathcal{P}(M^r)$ is a partial order, each construction $\mathcal{P}(f) : \mathcal{P}(M^{d_1}) \times \ldots \times \mathcal{P}(M^{d_n}) \to \mathcal{P}(M^{d_r})$ has residuals, i.e. for any $L_1 \in \mathcal{P}M^{d_1}, \ldots, L_n \in \mathcal{P}(M^{d_n})$ and $L \in \mathcal{P}(M^{d_r})$, there is a largest set $X \in \mathcal{P}M^{d_i}$ with $\mathcal{P}(f)(L_1, \ldots, L_{i-1}, X, L_{i+1}, \ldots, L_n) \subseteq L$, namely

$$X = \{\, \vec{w} \in M^{d_i} \mid \mathcal{P}(f)(L_1, \ldots, L_{i-1}, \{\vec{w}\}, L_{i+1}, \ldots, L_n) \subseteq L \,\}.$$

If all syntax rules $f : A_1 \times \ldots \times A_n \to A$ with the same result type A have a (unique?) common argument type B, we can define A/B as ??? *The* `VP` *has a field* `nn : Agr => Str * ... * Str` *for its nominal object; it is filled with the paradigm of empty strings in a* `VPSlash = VP **` `{c2:Prep}`. *A construction of a* `VPSlash`*, like*

```
Slash2V3 :  V3 -> NP -> VPSlash ; -- give it (to her)
```

contributes to `VPSlash = VP/NP` *by ...*

There are several advantages in using a PMCFG to express the grammar of a natural language, resp. in interpreting categories by sets of string tuples rather than by sets of strings:

- ***paradigms***: a word can have different forms, reflecting different syntactic roles in different contexts of usage. For example, a noun in German has four singular and four plural forms, so it is best view as an 8-tuple of strings, $(w_1, \ldots, w_8)$; it is useful to distinguish eight *abstract noun forms*, even if for each noun, the nominative plural and accusative plural forms are the same string. The same applies to phrases, e.g. clauses in different tenses.
- ***discontinuous phrases***: a phrase need not be *continuous*, i.e. a sequence of consecutive words in a sentence, but can be split in two or more parts. For example, in German a relative clause (or infinitival object) of a nominal object is often moved behind the infinite part of the verb, as in "*Wir haben die Warnung mißachtet, die auf der Packung stand*", where both the nominal object "*die Warnung, die auf der Packung stand*", and the predicate "*haben mißachtet*" are split in two parts.

- ***alternative linearization***s: it may be useful to distinguish a standard from non-standard forms of a phrase. For example, an initial part of an expression may be glued with a preceding word, e.g. "*in dem warmen Zimmer*" ↦ "*im warmen Zimmer*", "*in das warme Zimmer*" ↦ "*ins warme Zimmer*", so we may distinguish the standard form "*das warme Zimmer*" from a shortened one without definite article, "*warme Zimmer*", and use the second in combination with certain prepositions.

  More complicated: "*der Angeklagte hat Angaben nicht gemacht*" ↦ "*der Angeklagte hat keine Angaben gemacht*". "*ich trinke ein Bier nicht*" ↦ "*ich trinke kein Bier*". (For this, we apparantly have to separate the predicate from the indefinite nominal subject and objects, and be able to omit the indefinite article to perform sentence negation.)

Remark 2: The choice between different forms or alternative linearizations has to be implemented in GF by a *table*, a finite function from abstract forms to strings, e.g. $Gender \times Number \to String$. (For example, see p. 66 for the possible contraction of prepositions with definite articles.) (One can also have alternative tree constructions in application grammars.)

Problem 1. *The main category of a PMCFG should have dimension 1, so that parsers take ordinary strings as input. With context-free grammars, word order is encoded in the tree structure*[6]*; for languages with relatively free word order, this leads to a high number of grammar rules.*

For a PMCF-grammar, the same holds for each dimension. If A is an n-dimensional, M a 1-dimensional category and $f : M \to A \to A$ a modification rule that attaches the modifying string to the right of the i-th component, i.e. for $(m) \in M$, $\vec{a} = (a_1, \ldots, a_n) \in A$,

$$f((m), (a_1, \ldots, a_n)) = (a_1, \ldots, a_{i-1}, a_i m, a_{i+1}, \ldots, a_n) \in A,$$

iterated modifications of $\vec{a}$ by $m_1, \ldots, m_k \in M$ lead to $\vec{a}' \in A$ with different component $\vec{a}'_i = a_i m$, $m \in M^k$, that are constructed by different trees. So, for each component, as in a context-free grammar the word order is reflected by tree structure. However, the trees also reflect different orders of modifier applications in different components a_i, a_j of $\vec{a}$, and these are not to be seen in the resulting tuple

$$\vec{a}' = (a_1, \ldots, a_{i-1}, a_i m, a_{i+1}, \ldots, a_{j-1}, a_j m', a_{j+1} \ldots, a_n) \in A.$$

If a sentence $s \in S$ is constructed from an $\vec{a} \in A$ by $g(\vec{a}) = s$, then g can only concatenate the components of $\vec{a}$ (and additional constant strings). But is this any more than the ambiguity problem for CFGs? [In `LangGer`, we insert different kinds of complements of a verb in different components of a `vp:VP`, and the ordering in which the parser found them in the input s is irrelevant for $\vec{a}$, though remembered in the tree structure; but different trees with the same linearization in `Ger` might have different linearizations in `Eng` or some other language. So, the question is how far does, should and can the abstract grammar encode the word order of its languages?]

Todo 4: Discuss the problems to write a PMCFG:

1. Tuples can be used either for *merge* operations $\vec{a} \cdot \vec{b} = (a_1, b_1, a_2, b_2, \ldots)$, or for *permutation* operations, $\vec{b} = (a_{\pi 1}, \ldots, a_{\pi n})$, or as mixtures of both, including copying and concatenating of components,

[6] though not uniquely: if the grammar is ambiguous, the same string may be the linearization of different trees

2. In GF, tupels are also used for alternatives, e.g. use pre- and post-positions as circumpositions $p = \{$`s1:Str; s2:Str`$\}$ and then concatenate both components `appPrep p np` = `p.s1 ++ np.s ++ p.s2`. It is not obvious if in such cases it is *always* assumed that at least one component is empty. Should there be a discipline when to use several components, and when to use a tuple $\{$`s : Str * Str`$\}$? It's hard to know what the fields of a linearization record are used for, e.g. complements or modifiers?
3. If several modifying elements may be combined with a head element, as for noun modification by possessive, adjective, adverb and relative clause, these can either be stored in separate fields of the implementation record, or concatenated in a single field. The first possibility is useful to fix a relative order of the modifiers, via `cn.ap ++ cn.s ++ cn.poss ++ cn.adv ++ cn.rel` when using the `cn`, but the order in which the fields are filled –coded in the tree– is irrelevant, hence we pay by ambiguity. The second possibility is useful to allow for different orders of the modifiers, say `cn.s` = `ap.s ++ n.s ++ adv.s ++ poss.s` and `cn.s` = `ap.s ++ n.s ++ poss.s ++ adv.s`, probably including some that are not correct in a concrete language.
4. Is it plausible that Currying is a good idea for syntax? For example, if binary "verbs" can be obtained by adding a complement to ternary verbs, then passive constructions may not work for the binary verbs obtained that way.

2.2. The Grammatical Framework (GF)

A ***GF-grammar*** consists of an ***abstract grammar***, containing declarations of syntactic categories and syntactic constructions, and a number of ***concrete grammars***, which provide implementation types of the syntactic categories and linearization functions of the syntactic constructions declared in the abstract grammar. GF-grammars are multilingual in the sense that the same abstract grammar can have several concrete grammars.

2.2.1. Abstract grammars

An abstract grammar can be split into different (abstract) modules (c.f. Section 3.1). Typically, there is a module containing the declarations of syntactic categories, and several extensions of this module containing syntactic constructions to build expressions of specific categories.

A ***syntactic category*** (or: abstract type) `C` is declared by[7]

```
cat C ;
```

a ***syntactic construction*** (or: grammar rule name) `f` of arity k is declared by

```
fun f : C1 -> ... -> Ck -> C ;
```

where `C1`, ..., `Ck`, `C` are syntactic categories. The declared constructions can be combined to well-typed terms, called ***trees***: if `fun f:C` is a 0-ary construction, then `f` is an ***atomic tree*** of category `C`; if $t_1, \ldots, t_k$ are trees of categories `C1`,...,`Ck` and `fun f : C1 -> ...-> Ck -> C` is a syntactic construction, then `f` $t_1 \ldots t_k$ is a ***compound tree*** of category `C`.

[7] We here only sketch a fragment where the abstract language is a language of *simply typed* terms and ignore that GF more generally admits *dependent types*, although these can be very useful for natural languages, e.g. to define categories `NP Sg` and `NP Pl` of noun phrases in singular and plural, respectively.

2.2.2. Concrete Grammars

A concrete grammar `CG` of an abstract grammar `G` provides a linearization category for each syntactic category and a linearization function for each syntactic construction of `G`. These are types and terms of a certain programming language.

The ***linearization category*** (or: implementation type) of a syntactic category `C` is a record type associated to `C` by a declaration

```
lincat C = {s1 : sigma1 ; ... ; sn : sigman ; p1 : tau1 ; ... ; pm : taum} ;
```

where `s1`,...,`sn` and `p1`,...,`pm` are labels and `sigma1`,...,`sigman` and `tau1`,...,`taum` are types of a specific implementation (programming) language.

The ***linearization function*** (or: grammar rule implementation) of a syntactic construction `f : C1 -> ... -> Ck -> C` is defined in the form

```
lin f x1 ... xk = t ;
```

where `t` is a term of the implementation type of `C` in the programming language and is built from variables `x1`,...,`xk` of the implementation types of `C1`,...,`Ck`.

Todo 5: check! The underlying programming languages has basic types like `Bool` and `Str` for boolean values and strings, parameter types `Ty` (with finitely many values) declared by

```
param
  Ty = F1 ty_11 ... ty_1k | ... | Fl ty_l1 ... Fl ty_lk'
```

where `F1`,...,`Fl` are type constructors and `ty_11`, ..., `ty_lk'` are parameter types; moreover, there are record types

```
{l1 : ty_1 ; ... ; lk : ty_k}
```

with pairwise different labels `l1`, ..., `lk`, and types

```
ty => ty'
```

of functions from a parameter type `ty` to a type `ty'`. Types `ty` can be given names `T` by type declarations

```
oper
  T : Type = ty ;
```

The implementation type

```
{s1 : sigma1 ; ... ; t1 : tau1 ; ... }
```

of a syntactic category typically has some fields `s1`, ... of type `Str` or `ty1 => ... => tyk => Str` and some fields `t1`, ... of parameter types. Records of this type contain fixed strings or tables with result type `Str`, e.g. inflection paradigms for nouns in German, of type `Number => Case => Str`, or word order variations of an expression, of type `Order => Str` for some parameter type `Order`. In general, a subexpression of a sentence in natural language is not a substring of

the sentence, but can be split into several, non-adjacent substrings, so there will be several such fields `s1`, ...`sk`. Besides these, there typically are some fields `t1`, ..., `tl` of parameter types, holding fixed, inherent parameters, e.g. fields `t1:Gender` and `t2:Person` holding the gender and person values of a personal pronoun.

Remark 3: Different syntactic categories can have the same implementation type, e.g. particles and adverbs might have the implementation type {`s:Str`} of a record with a single field of type string. But the declaration `lincat C = ty` implicitly makes the implementation type unique by adding a (hidden) field whose label `lock_C` contains the name `C` of the syntactic category: the declaration `lincat C = ty` is equivalent to

```
oper C : Type = ty ** {lock_C : {}} ;  -- or: oper C : Type = lin C ty ;
```

Todo 6: values of these types, records {`s1 = w1 ; ... ; t1 = x1 ; ...`}, `\\x => t`, field selection `r.s` and table selection `t!s`, overwriting record extension `r ** s`, opers $\backslash x -> t$, ...

The GF-Book: Ranta [9], the overview for programmers: Ranta [8].

2.2.3. Grammar compilation (todo)

The GF compiler: Angelov [1]

Compilation of GF-grammars to PMCFG-grammars: Inherent parameters p of a category C are compiled to PMCFG-categories C_p. Grammar rules $f : C \to D \to E$ to PMCFG-rules $f_{p,q,r} : C_p \to Q_q \to E_r$ for parameter values p, q, r. (Compilation of `TestLangGer` with generation of PMCFG: 324164 msec)

Formal language theory: Switch from syntactic categories as sets of strings and grammar rules as inclusions $A \cdot X \cdot Y \subseteq X$ to types and typed functions $f : A \to X \to Y \to X$, while maintaining an interpretation of types by sets of n-tuples of strings (and parameter values).

Result 1: Linearization of trees

Notice that linearization cannot be done by case distinction on the abstract form of the arguments. Explain why this is necessary for parsing.

GF-book, p.147: "So, when is ***run-time transfer*** needed? The general answer follows from a fundamental property of GF: that linearization is ***compositional***. Compositionality means that the linearization of any tree is a function of the *linearizations* of its subtrees. In other words,

$$(f\ x_1\ \ldots\ x_n)^* = f^*\ x_1^*\ \ldots\ x_n^*$$

where t^* is the linearization of a tree t, and f^* is the linearization function of a function f. This means that linearization cannot inspect the structure of the subtrees themselves, but only of their linearizations. Hence, whenever an operation is not compositional, it cannot be encoded as linearization in GF, and hence needs run-time transfer."

Result 2: The GF parser: Angelov [1].

Remark 4: The parser operates on a category S and a string $w \in \Sigma^*$ and tries to find trees t that linearize to w. However, if S has dimension $n > 1$, it accepts w if there is a partial tree $t(x_1, \ldots, x_k) : S$ with some unknown subtrees $x_1, \ldots, x_k$ represented by ***metavariables*** `?1,?2,...,?k`, such that w is a component a_i of the linearization $(a_1, \ldots, a_n) \in (\Sigma^*)^n$ of some completion of t to a tree $t(s_1, \ldots, s_k)$ without free variables.

Conversely, the linearization of a grammar rule $f : B \to A$ with m-dimensional category B ought to be a term $t : A$ where *all* components of arguments $v = (b_1, \ldots, b_m)$ are used in the string

fields of t; otherwise, a partial tree $t = f(\ldots x_j \ldots) : A$ will be produced and metavariables $?m$ appear in the syntactic structure.

3. Grammars in the Resource Library

The Resource Grammar Library of the GF provides

(i) an abstract grammar `Grammar` consisting of a set of 106 syntactic categories together with a set of 278 abstract syntactic constructions[8],

(ii) a library of concrete grammars `GrammarEng`, `GrammarSpa`, ... for about 50 natural languages like English, Spanish, ... implementing this abstract grammar `Grammar`, and

(iii) abstract lexicons `Lexicon` and `Structural` of about 350 content words and 120 structural words, with implementations `LexiconEng`, `LexiconSpa`, ... and `StructuralEng`, `StructuralSpa`, ... for the 50 natural languages.

The grammars and lexicons are combined to abstract modules `Lang` and their language specific concrete modules `LangEng`, `LangSpa`, ..., intended for testing the grammars. Some of the concrete grammars are under development and only partial implementations of `Grammar` and `Lang`.

3.1. The Abstract Resource Grammar `Lang`

Todo7: This can only be a repetition of `gf-rgl/src/abstract/`, but besides the single examples given there, a few lines of explanatory text for each rule could be helpful. There may also be proposals to improve `Lang`.

Abstract syntax as a (free) algebra of simply typed terms (abstract `trees`). Types correspond to expression categories (syntactic types), a term- or tree-constructor can be seen as the name of a syntactic construction, turning expressions of input categories to an expression of result category.

`Lang` consists of modules `Grammar` and `Lexicon`[9] using the same module `Cat` of syntactic categories. The `Grammar` consists of modules

```
abstract Grammar =
  Noun,
  Verb,
  Adjective,
  Adverb,
  Numeral,
  Sentence,
  Question,
  Relative,
  Conjunction,
  Phrase,
  Text,
  Structural,
  Idiom,
  Tense,
```

[8] 113 data declarations + 273 fun declarations in `Grammar` - 108 fun declarations in `Structural`

[9] We skip some additional modules `Construction`, `Documentation` and `Markup`.

```
  Transfer
  ;
```

declaring syntactic constructions as function symbols whose types are simple types with syntactic categories as base types. Each of these modules extends a module `Cat` that declares syntactic categories as base types, from which function types can be built that serve as types of grammatical constructions, i.e. declared grammar rules in the above modules.

3.1.1. Categories

Todo 8: Maybe I should not give all categories in one sweep, as `Cat.gf` does, but give them in connection with the construction rules, i.e. the determiner and noun categories in `Noun`, etc., but these need `AP`, `RS`, etc., which are not explained in `Noun`.

Preliminary extraction from `abstract/Cat.gf`:

```
-- Sentences and clauses (Sentence.gf, Idiom.gf)

    S ;     -- declarative sentence                e.g. "she lived here"
    QS ;    -- question                            e.g. "where did she live"
    RS ;    -- relative                            e.g. "in which she lived"
    Cl ;    -- declarative clause, with all tenses e.g. "she looks at this"
    ClSlash;-- clause missing NP (S/NP in GPSG)    e.g. "she looks at"
    SSlash ;-- sentence missing NP                 e.g. "she has looked at"
    Imp ;   -- imperative                          e.g. "look at this"
```

It seems the `ClSlash` and `SSlash` are meant to be categories of clauses and sentences missing an *object* noun phrase. (At least, the implementation categories show a field `c2 : Prep` in `Eng`.)

```
-- Questions and interrogatives (Question.gf)

    QCl ;   -- question clause, with all tenses    e.g. "why does she walk"
    IP ;    -- interrogative pronoun               e.g. "who"
    IComp ; -- interrogative complement of copula  e.g. "where"
    IDet ;  -- interrogative determiner            e.g. "how many"
    IQuant; -- interrogative quantifier            e.g. "which"

-- Relative clauses and pronouns (Relative.gf)

    RCl ;   -- relative clause, with all tenses    e.g. "in which she lives"
    RP ;    -- relative pronoun                    e.g. "in which"

-- Adjectival phrases (Adjective.gf)

    AP ;    -- adjectival phrase                   e.g. "very warm"

-- Nouns and noun phrases (Noun.gf, Structural.gf)

    CN ;     -- common noun (without determiner)    e.g. "red house"
    NP ;     -- noun phrase (subject or object)     e.g. "the red house"
    Pron ;   -- personal pronoun                    e.g. "she"
```

The syntactic category `Det` of determiners is, in a sense, a collection of expression categories that satisfy the same syntactic function, without common structure (or build from a head component); in formal language theory terms it is a sum `Det = Num + Card + Ord + Pron`.

```
-- The determiner structure is: Predet (QuantSg | QuantPl Num) Ord.

    Det ;    -- determiner phrase                   e.g. "those seven"
    Predet ; -- predeterminer (prefixed Quant)      e.g. "all"
    Quant ;  -- quantifier ('nucleus' of Det)       e.g. "this/these"
    Num ;    -- number determining element          e.g. "seven"
    Card ;   -- cardinal number                     e.g. "seven"
    ACard ;  -- adjective like cardinal             e.g. "few", "many"
    Ord ;    -- ordinal number (used in Det)        e.g. "seventh"
    DAP ;    -- determiner with adjective           e.g. "three small"

-- Numerals (Numeral.gf)

    Numeral ; -- cardinal or ordinal in words       e.g. "five/fifth"
    Digits ;  -- cardinal or ordinal in digits      e.g. "1,000/1,000th"

-- Structural words (Structural.gf)

    Conj ;  -- conjunction                          e.g. "and"
    Subj ;  -- subjunction                          e.g. "if"
    Prep ;  -- preposition, or just case            e.g. "in"
```

Astonishingly, the abstract grammar has a category `Prep` (that includes cases), though clearly, prepositions are language-specific. As far as they are used to connect a complement to verbs, nouns or adjectives, they come with the implementation of the verb, noun or adjective in concrete grammars and do not show up in the abstract grammar.

```
-- Verb phrases (Verb.gf)

    VP ;      -- verb phrase                        e.g. "is very warm"
    Comp ;    -- complement of copula, such as AP   e.g. "very warm"
    VPSlash ; -- verb phrase missing complement     e.g. "give to John"
```

The category `VP` is the category of (basic, noncoordinated) clauses missing a (nominal) subject, consisting of a finite verb with all complements except a subject, and possibly some modifying adverbials (and appositions?). It corresponds to the category `S/NP` of categorial grammar and GPSG, or rather to `Cl/NP`, except that it is not assumed that the missing `np:NP` can only be added at the end of a `vp:Cl/NP` to give a clause `cl:Cl`. The category `VPSlash` corresponds to the category `VP/NP`, i.e. of clauses missing a nominal subject and a nominal object.

```
-- Words of open classes (Lexicon.gf, additional lexicon modules)

    V ;     -- one-place verb                       e.g. "sleep"
    V2 ;    -- two-place verb                       e.g. "love"
```

```
V3 ;    -- three-place verb                    e.g. "show"
VV ;    -- verb-phrase-complement verb         e.g. "want"
VS ;    -- sentence-complement verb            e.g. "claim"
VQ ;    -- question-complement verb            e.g. "wonder"
VA ;    -- adjective-complement verb           e.g. "look"
V2V ;   -- verb with NP and V complement       e.g. "cause"
V2S ;   -- verb with NP and S complement       e.g. "tell"
V2Q ;   -- verb with NP and Q complement       e.g. "ask"
V2A ;   -- verb with NP and AP complement      e.g. "paint"

A ;     -- one-place adjective                 e.g. "warm"
A2 ;    -- two-place adjective                 e.g. "divisible"

N ;     -- common noun                         e.g. "house"
N2 ;    -- relational noun                     e.g. "son"
N3 ;    -- three-place relational noun         e.g. "connection"
PN ;    -- proper name                         e.g. "Paris"
```

Some additional categories are inherited from `Common.gf`. They are defined there since they have the same implementation in all languages in the resource grammar library (typically, just a string). These categories are `AdA, AdN, AdV, Adv, Ant, CAdv, IAdv, PConj, Phr, Pol, SC, Tense, Text, Utt, Voc, Interj`. Moreover, the list categories `ListAdv, ListAP, ListNP, ListS` are defined in `Conjunction.gf` and only used locally there.

There are no categories for verbs of arity greater than three. Of course, missing categories like these can be added to `Cat.gf`, suitable lexical entries to `Lexicon.gf` and new constructors to `Verb.gf`, `Noun.gf` and `Adjective.gf`, at least. However, higher arities of verbs can easily lead to complexity issues in grammar compilation and parsing.

Remark 5. *There are no categories for nouns with non-nominal complements, like `NV` for nouns with an infinitival complement, e.g. "belief:`NV` to become a millionaire", or `N2V` for nouns with a nominal and an infinitival complement, e.g. "advice:`N2V` to an exhausted colleague to work less". There are no categories for adjectives with non-nominal complements, like `AV` for adjectives with an infinitival complement, e.g. "eager:`AV` to become the boss".*

Sentential, interrogative and infinitival (subject or object) complements can be added to nouns and adjectives via the rules `SentCN : CN -> SC -> CN` and `SentAP -> AP -> SC -> AP`, and sentential subjects via `PredSCVP : SC -> VP -> Cl`. Such complements are constructed by

```
EmbedS    : S  -> SC ;          -- that she goes
EmbedQS   : QS -> SC ;          -- who goes
EmbedVP   : VP -> SC ;          -- to go
```

But these cannot be reflexive, I guess: "(seine) Versuche, sich zu bessern"

The one-, two- and three-place verbs `V`, `V2`, `V3`, nouns `N`, `N2`, `N3` and adjectives `A`, `A2` take nominal objects, which are related to the verb, noun or adjective by a specific case or preposition with case. These are language-dependent and specified in the implementation of the verb, noun, or adjective, e.g. *eng. to believe in* versus *ger. glauben an*.

Remark 6: It is somewhat unclear which of two nominal objects plays which syntactic roles, and whether the role is the same in all concrete languages. For example, are `direct` and `indirect`

objects coded by complement `c2` and `c3` uniformly? Are english di-transitive verbs like *give sb sth* and binary verbs like *give sth to sb* correctly related?

3.1.2. Noun

From `Noun.gf`: (but in different order)

Construction of Determiners, Quantifiers and Numerals

```
-- The determiner has a fine-grained structure, in which a 'nucleus'
-- quantifier and an optional numeral can be discerned.

    DetQuant    : Quant -> Num ->        Det ;  -- these five
    DetQuantOrd : Quant -> Num -> Ord -> Det ;  -- these five best

-- Whether the resulting determiner is singular or plural depends on the
-- cardinal.

-- All parts of the determiner can be empty, except Quant, which is
-- the "kernel" of a determiner. It is, however, the Num that determines
-- the inherent number.

    NumSg   : Num ;  -- [no numeral, but marked as singular]
    NumPl   : Num ;  -- [no numeral, but marked as plural]
    NumCard : Card -> Num ; -- one/five [explicit numeral]
```

Notice that determiners generally depend on number, as can be seen from the types of `DetQuant`, `DetQuantOrd` and the constants `someSg_Det`, `somePl_Det`. The number of a noun phrase, e.g. when used as subject of a clause, can (almost) always be chosen freely by the speaker[10], so the it is a design decision of `Grammar` to give determiners an inherent number (which is inherited to noun phrases and then determines the form of verbs in subject-verb combinations).

```
-- Card consists of either digits or numeral words.

  data
    NumDigits  : Digits  -> Card ;  -- 51
    NumNumeral : Numeral -> Card ;  -- fifty-one

-- The construction of numerals is defined in [Numeral Numeral.html].

-- A Card can  be modified by certain adverbs.

  fun
    AdNum : AdN -> Card -> Card ;   -- almost 51

-- An Ord consists of either digits or numeral words.
```

[10] except that a reciprocal pronoun can enforce plural subject, e.g. *they talk to each other*, and some verbs demand plural objects, e.g. *to collect stamps* (or mass nouns: *to collect money*).

```
-- Also superlative forms of adjectives behave syntactically like ordinals.

    OrdDigits  : Digits  -> Ord ;  -- 51st
    OrdNumeral : Numeral -> Ord ;  -- fifty-first
    OrdSuperl  : A       -> Ord ;  -- warmest

-- One can combine a numeral and a superlative.

    OrdNumeralSuperl : Numeral -> A -> Ord ; -- third largest

-- Definite and indefinite noun phrases are sometimes realized as
-- neatly distinct words (Spanish "un, unos ; el, los") but also without
-- any particular word (Finnish; Swedish definites).

    IndefArt   : Quant ;  -- a/an
    DefArt     : Quant ;  -- the
```

Construction of Common Nouns `CN`

```
-- Simple nouns can be used as nouns outright.

    UseN : N -> CN ;              -- house

-- Relational nouns take one or two arguments.

    ComplN2 : N2 -> NP -> CN ;    -- mother of the king
    ComplN3 : N3 -> NP -> N2 ;    -- distance from this city (to Paris)

-- Relational nouns can also be used without their arguments.
-- The semantics is typically derivative of the relational meaning.

    UseN2   : N2 -> CN ;          -- mother
    Use2N3  : N3 -> N2 ;          -- distance (from this city)
    Use3N3  : N3 -> N2 ;          -- distance (to Paris)
```

One may expect relational nouns to also have *prepositional complements*, but although `Cat` has a category `Prep` of prepositions, the argument category in `ComplN2` and `ComplN3` is `NP`, not `Prep`. Likewise, complementation rules for relational adjectives `A2` and verbs `V2`, `V3` do not have `Prep` as argument category, but `NP`. The preposition used to combine a complement with a relational noun, adjective or verb is specific to (and must be derived from) this noun, adjective or verb.

Remark 7. *The various ways that a preposition can be used in a language does not allow for a uniform translation of prepositions, nor can a verb be combined with an arbitrary preposition. So one should not expect* `Prep` *as argument category. However, there are such rules in* `Grammar`*:*

`Adverb.PrepNP : Prep -> NP -> Adv`	*for: in the house (see Remark 19, p. 29),*
`Extend.PrepCN : Prep -> CN -> Adv`	*for: by accident,*
`Extend.AdvRNP : NP -> Prep -> RNP -> RNP`	*for: a dispute with his wife*[11]*,*
`Question.PrepIP : Prep -> IP -> IAdv`	*for: with whom,*
`Relative.FunRP : Prep -> NP -> RP -> RP`	*for: the mother of whom*[12]
`Verb.VPSlashPrep : VP -> Prep -> VPSlash`	*for: live in (the city)*
`Sentence.SlashPrep : Cl -> Prep -> ClSlash`	*for: (with whom) he walks.*

Since we cannot expect a one-to-one translation of prepositions from one language to another, we here have to consider `Prep` *as a category of abstract prepositions fulfilling specific semantic purposes, like specifying positions or directions in space, relative to the speaker or hearer.*

Modification of common nouns

```
-- Nouns can be modified by adjectives, relative clauses, and adverbs.

    AdjCN   : AP -> CN  -> CN ;   -- big house
    RelCN   : CN -> RS  -> CN ;   -- house that John bought
    AdvCN   : CN -> Adv -> CN ;   -- house on the hill
```

The modification rule `AdvCN` is overgenerating, since `Adv` subsumes adverbial clauses, e.g. **house, because the weather was fine*. The rule is apparently meant to be used only with pro-adverbs, e.g. *over there* = *dahinten*, and adverbs built from (meaningful) prepositions, e.g. *under* : `Prep =< Adv/NP`. Q3: Can this restriction be implemented, or is it better to rely on syntactically correct input to parsing?[13]

```
-- Nouns can also be modified by embedded sentences, questions and infinitives.
-- For some nouns this makes little sense, but we leave this for applications
-- to decide.

    SentCN  : CN -> SC  -> CN ;   -- question where she sleeps
```

The rule `SentCN` only makes sense for nouns derived from (or at least related to) verbs or adjectives with suitable (and similar) complement frame, and a few other nouns like *fact*, *question*, *command*. Extensions of nouns by sentences, questions and infinitives are complementations rather than modifications. E.g., from verbs `know:VQ`, `know:VS`, `believe:VS`, but *`believe:VQ`, one might derive nouns of suitable noun categories, "*knowledge (who VP)*", but not "*belief (who VP)*". Since `SentCN` operates on arbitrary common nouns `cn:CN` and arbitrary sentential complements `sc:SC`, and can be applied repeatedly, it is highly overgenerating.

Remark 8. *(c.f. Remark 5). Instead of the category* `SC` *and the modification rule* `SentCN`*,* `Grammar` *better had categories like* `NS`*,* `NQ`*,* `NV`*,* `AS`*,* `AQ`*,* `AV` *and complementation rules* `ComplNS : NS -> S -> CN` *etc. to combine nouns and adjectives of the appropriate category with sentential, interrogative and infinitival complements, and categories like* `N2V`*, e.g. for "advice to John to work less". Since the non-nominal objects seem optional, we'd also need embedding rules* `UseNS : NS`

[13] A common noun modified by several of these modifications has several trees differing in the relative order of modification. To reduce spurious ambiguities, one can implement a tree transformation `nfCN : CN -> CN` that collapses the different modification orders. See Remark 34 and p. 190.

`-> CN` etc. The same noun might have different complement frames, e.g. "belief in God" and "belief that God exists", or "Hoffnung auf Erlösung" and "Hoffnung, zu überleben". Since complements are attached "closer" to the noun than modifiers, some spurious ambiguities would be avoided, i.e. we only had `(AdjCN ap (ComplNS n s))` instead of the two constructions `(AdjCN ap (SentCN (UseN n) s))` and `(SentCN (AdjCN ap (UseN n)) s)`.

Modification of `CN` by apposition, possessive and partitive noun phrases:

```
-- Apposition. This is certainly overgenerating.

    ApposCN : CN -> NP -> CN ;    -- city Paris (, numbers x and y)

-- Possessive and partitive constructs

    PossNP  : CN -> NP -> CN ;     -- house of Paris, house of mine
    PartNP  : CN -> NP -> CN ;     -- glass of wine
```

Remark 9: The examples for `ApposCN` are rather examples for a more restricted apposition by (a conjunction of) names, *the apostles Peter and Paul*, so maybe the type should be `ApposCN : CN -> [PN] -> CN`, where `[PN]` is the category of lists of names `PN` (c.f. category `Conjunction`). Apposition by a full noun phrase ought to be separated by commata: *Paris, the capital of France,* or *Peter and Paul, my favorite apostles,* or *the Sophists, a group of philosphers in ancient greece,*.

Proposal 1: Let `ApposCN` embed the apposition in commata and add a separate apposition `ApposPN : CN -> [PN] -> CN` without commata. Or add `ExtApposCN` for post-nominal appositions in commata. Todo 9: Compare with `Extend.ApposNP`. ◁

Q4: Is there a German version of `PossNP` when the noun phrase is a personal pronoun, e.g. `PossNP she_Pron` = *Haus von ihr*?

Construction of Noun Phrases `NP`

```
-- The three main types of noun phrases are
-- - common nouns with determiners
-- - proper names
-- - pronouns

  fun
    DetCN   : Det -> CN -> NP ;   -- the man
    UsePN   : PN -> NP ;          -- John
    UsePron : Pron -> NP ;        -- he

-- Pronouns are defined in the module Structural.gf.

-- Determiners can form noun phrases directly.

    DetNP   : Det -> NP ;          -- these five
```

In some languages, determiners have special forms for such "stand-alone" usages.

```
-- Nouns can be used without an article as mass nouns. The resource does
-- not distinguish mass nouns from other common nouns, which can result
-- in semantically odd expressions.

    MassNP  : CN -> NP ;           -- beer
```

Remark 10: The rule is massively overgenerating, so one better omits it for parsing, i.e. uses `Grammar - [MassNP]` for parsing. It seems better to add a lexical category `MN` of mass nouns and a more limited construction `MassNP : MN -> NP`. Then lexical entries `fun n:MN` can specify which nouns `n` can be used as mass nouns. For example, a mass noun in singular can be used as a noun phrase, e.g. *life is not easy*, while nouns in general cannot. Some quantifiers cannot be used with mass nouns, others can only be used with mass nouns: **much child*, *much time*, *many children*, **many time*. However, ordinary count nouns can also be used without determiner: *er hat Haus und Hof verloren und Frau und Kind verlassen*. Q5: What can be done to restrict determinerless usage of CN as NP and limit the ambiguities in parsing?

Modification of `NP` and `DAP`

```
-- A noun phrase already formed can be modified by a predeterminer.

    PredetNP : Predet -> NP -> NP ; -- only the man

-- A noun phrase can also be postmodified by the past participle of a
-- verb, by an adverb, or by a relative clause

    PPartNP : NP -> V2  -> NP ;    -- the man seen
    AdvNP   : NP -> Adv -> NP ;    -- Paris today
    ExtAdvNP: NP -> Adv -> NP ;    -- boys, such as ..
    RelNP   : NP -> RS  -> NP ;    -- Paris, which is here

-- This is different from the partitive, as shown by many languages.

    CountNP : Det -> NP -> NP ;    -- three of them, some of the boys

-- Conjoinable determiners and ones with adjectives

    AdjDAP : DAP -> AP -> DAP ;    -- the large (one)
    DetDAP : Det -> DAP ;          -- this (or that)
```

3.1.3. Adjective

Construction of adjective phrases

The file `gf-rgl/src/abstract/Adjective.gf` declares constructions for adjective phrases.

```
abstract Adjective = Cat ** {
  fun

-- The principal ways of forming an adjectival phrase are positive,
```

```
-- comparative, relational, reflexive-relational, and elliptic-relational.

    PositA  : A  -> AP ;        -- warm
    ComparA : A  -> NP -> AP ;  -- warmer than I
    ComplA2 : A2 -> NP -> AP ;  -- married to her
    ReflA2  : A2 -> AP ;        -- married to itself
    UseA2   : A2 -> AP ;        -- married

    UseComparA : A  -> AP ;     -- warmer
    CAdvAP  : CAdv -> AP -> NP -> AP ; -- as cool as John

-- The superlative use is covered in Ord.

    AdjOrd  : Ord -> AP ;       -- warmest
```

Remark 11: The rule `ComplA2` might be replaced by rules `SlashA2 : A2 -> APSlash` and `ComplAPSlash : APSlash -> NP -> AP`, with `APSlash = AP ** {c2:Preposition}`.

Todo 10: Discuss why `Grammar` has no categories `APSlash` and `NPSlash`.

Modification of adjective phrases

The first modification rule `SentAP : AP -> SC -> AP` adds a sentential complement, i.e. sentence, infinitive or question to an adjective phrase:

```
-- Sentence and question complements defined for all adjectival phrases,
-- although the semantics is only clear for some adjectives.

    SentAP  : AP -> SC -> AP ;  -- good that she is here
```

Remark 12: The semantics is not clear at all. `SentAP` combines an adjective with a sentence or an infinitive, e.g. *good that she is here* or *good to sleep*. But these are *not* adjective phrases and don't express properties! Otherwise they could be turned into a verb phrase and combined with a noun phrase to a clause[14], but **John is good that she is here* is not a clause. Even when combined with *it*, the clauses *it is good that she is here* or *it is good to sleep* do not have subject *it* and predicates *good that she is here* or *good to sleep*, but (moved) sentential and infinitival subjects *that she is here* and *to sleep*, respectively, subject-correlate *it*, and predicate *to be good*. Similarly: the subject of *a good man is hard to find* is not *a good man*, but *to find a good man*. (Recall also Chomsky's examples *John is easy to please* and *John is eager to please*.)

As remarked for `SentCN`, the modification rule `SentAP` seems to be a substitute for missing subcategories `AS`, `AV`, `AQ` of (binary) adjectives and missing complementation rules, like

```
    ComplAS : AS -> S -> AP ;    -- (we are) glad that she is here
    ComplAV : AV -> VP -> AP ;   -- (we are) happy to be alive
    ComplAQ : AQ -> QS -> AP ;   -- (we are) uncertain, whether they arrived
```

Such complementations `(ComplAS as s):AP` apparently are only used predicatively (hence might have result category `Comp`).[15] Moreover, subcategories of unary adjectives that can be used

[14] by `UseComp o CompAP : AP -> VP` and `PredVP : NP -> VP -> Cl`

[15] Can we attributively say *der gelobt zu werden begierige Schüler*, eng. *the student eager to be praised*?

predicatively with sentential, infinitival or interrogative subject may be necessary, e.g. *false, unlikely, unbelievable, plausible.* (Attributive usage of such adjectives are restricted to specific nouns, e.g. *false statement*, *unbelievable claim*, *plausible assumption.*)

```
-- An adjectival phrase can be modified by an *adadjective*, such as "very".

    AdAP    : AdA -> AP -> AP ; -- very warm

-- It can also be postmodified by an adverb, typically a prepositional phrase.

    AdvAP   : AP -> Adv -> AP ; -- warm by nature

-- The formation of adverbs from adjectives (e.g. "quickly") is covered
-- in Adverb.gf; the same concerns adadjectives (e.g. "extremely").
}
```

3.1.4. Verb

Construction rules for `VP` and `VPSlash`.

A ***verb phrase***, `VP`, is a clause missing a nominal subject complement. An ***incomplete verb phrase***, `VPSlash`, i.e. a verb phrase missing a nominal object. The simplest verb phrase construction is to use a unary (full) verb:

```
    UseV     : V   -> VP ;             -- sleep
```

A verb phrase can also be built be adding a nominal complement to an incomplete verb phrase:

```
    ComplSlash : VPSlash -> NP -> VP ; -- love it
```

The next group of constructions are the complementation of binary and ternary verbs by complements of suitable types. Binary verbs expecting an infinitival, sentential, interrogative or adjectival complement can be combined with appropriate complements to a verb phrase `VP`:

```
    ComplVV  : VV  -> VP -> VP ;       -- want to run
    ComplVS  : VS  -> S  -> VP ;       -- say that she runs
    ComplVQ  : VQ  -> QS -> VP ;       -- wonder who runs
    ComplVA  : VA  -> AP -> VP ;       -- they become red
```

The simplest incomplete verb phrases are binary verbs expecting two nominal complements:

```
    SlashV2a : V2        -> VPSlash ;  -- love (it)
```

Further incomplete verb phrases are obtained by combining ternary verbs with complements of suitable type, where a nominal phrase can be used as either missing complement:

```
    Slash2V3 : V3  -> NP -> VPSlash ;  -- give it (to her)
    Slash3V3 : V3  -> NP -> VPSlash ;  -- give (it) to her

    SlashV2V : V2V -> VP -> VPSlash ;  -- beg (her) to go
    SlashV2S : V2S -> S  -> VPSlash ;  -- answer (to him) that it is good
    SlashV2Q : V2Q -> QS -> VPSlash ;  -- ask (him) who came
    SlashV2A : V2A -> AP -> VPSlash ;  -- paint (it) red
```

An incomplete verb phrase can also be built by combining a verb expecting an infinitival object with an incomplete verb phrase:

```
SlashVV    : VV  -> VPSlash -> VPSlash ;       -- want to buy
SlashV2VNP : V2V -> NP -> VPSlash -> VPSlash ; -- beg me to buy
```

These "incomplete" complementations correspond to the "complete" ones by `ComplVV` and `ComplV2V`. Q6: Is it intended that `SlashVV` corresponds to `ComplVV` in the sense that

```
(ComplVV vv (ComplSlash vps np)) == (ComplSlash (SlashVV vv vps) np)
```

i.e. that these trees are equivalent, i.e. have the same implementation records?

Remark 13. *`ComplSlash` generalizes the rule `VP/NP` · `NP` ≤ `VP` of categorial grammar; it combines an incomplete verb phrase `vps:VPSlash` with a noun phrase `np:NP` to a verb phrase, using a preposition or case inferred from `vps` to inflect the `np`. However, for ternary verbs `v3:V3` an ambiguity arises, as for any `n2,n3:NP`, these two constructions give the same linearization*[16]*:*

```
(CompSlash (Slash2V3 v3 np2) np3) = (ComplSlash (Slash3V3 v3 np3) np2)
```

It may therefore be reasonable to replace `ComplSlash` by special complementation rules for ternary verbs (and likewise for verbs of higher arity):

```
ComplV2   : V2 -> NP -> VP ;       -- love it              --HL
Compl23V3 : V3 -> NP -> NP -> VP ; -- give it to her       --HL
Compl32V3 : V3 -> NP -> NP -> VP ; -- give to her the book --HL
```

Can the ambiguitiy be avoided and the two complementation rules be used to provide two different word orders, and how can these be matched in different languages? The drawback would be that modification rules for `VPSlash` could not be applied to `(Slash2V3 v3 np2)` before combining with `np3`. (The function `Compl23V3 v3 np2 : NP -> VP` is not the same as the expression `(Slash2V3 c3 np2) : VPSlash`, or `NP -> VP` not the same as `VPSlash`.)

```
-- Verb phrases can also be constructed reflexively.

ReflVP   : VPSlash -> VP ;         -- love himself
```

Remark 14. *`ReflVP` constructs a verb phrase by using a reflexive personal pronoun. Another way to construct a verb phrase reflexively, missing in `Grammar`, is to use a reflexive possessive pronoun, e.g. "to love one's parents" or "to blow one's nose" = "sich schneuzen".*

`Grammar` misses the "indefinite" personal resp. reflexive pronoun "one" resp."oneself", as used in or "one should not hate oneself" or in reflexive infinitival subject sentences: "to love oneself is better than to love nobody". (Just add `one_Pron : Pron` to `Structural.gf` (Ger: "man"), and add further cases to `reflPron, possPron : Agr => Str`, say `AgP0 Sg => "oneself" | AgP0 Pl => "each other"` (Ger: "sich" and "einander"), and `AgP0 Sg => "one's" | AgP0 Pl => "each other's"` (Ger: "sein" and "von einander")?)

[16] in `Eng`. Are the linearizations necessarily equal in all languages? No, in `Ger` they are different.

```
-- Passivization of two-place verbs is another way to use them. In many
-- languages, the result is a participle that is used as complement to a
-- copula.

    PassV2   : V2 -> VP ;               -- be loved

-- *Note*. the rule can be overgenerating, since the V2 need not take a
-- direct object.
```

Remark 15: More generally, one can build verb phrases by passive constructions from incomplete verb phrases rather than from binary verbs. (See extensions of `Lang` by `Extra.gf`.) These incomplete verb phrases can be obtained from ternary verbs; so, implicitly there are passive constructions from n-ary verbs for $n > 2$ (see `TestLang`). ◁

Finally, verb phrases can consist of a copula verb alone,

```
    UseCopula : VP ;                    -- be
```

or by combining a copula verb with a suitable complement,

```
    UseComp  : Comp -> VP ;             -- be warm
```

A copula verb can combine with different complements to a verb phrase, e.g. *to be old*, *to be here*, *to be king*, *to be the next president*. These complements are built by embedding adjective phrases, noun phrases, adverbs or common nouns to the category `Comp`:

```
-- Adjectival phrases, noun phrases, and adverbs can be used.

    CompAP   : AP  -> Comp ;            -- (be) small
    CompNP   : NP  -> Comp ;            -- (be) the man
    CompAdv  : Adv -> Comp ;            -- (be) here
    CompCN   : CN  -> Comp ;            -- (be) a man/men
```

Remark 16: The only copula verb in `Lang` is "*to be*", but one can extend the lexicon by others, like "*to become*" or "*to remain*". But the constructions `UseCopula` and `UseComp` only admit the copula verb "*to be*"; `Lang` has no category of copula verbs. ◁

Modification rules.

Verb phrases and incomplete verb phrases can be modified by adverbs of type `Adv` or `AdV`.

```
-- Adverbs can be added to verb phrases. Many languages make a distinction
-- between adverbs that are attached at the end vs. next to (or before) the verb.

    AdvVP    : VP -> Adv -> VP ;            -- sleep here
    ExtAdvVP : VP -> Adv -> VP ;            -- sleep , even though ...
    AdVVP    : AdV -> VP -> VP ;            -- always sleep
```

Remark 17 The difference between a category `Adv` for "adverbs attached at the end of the verb phrase" and a category `AdV` for "adverbs inserted before (or near) the verb" is unplausible: why should the same abstract adverb behave the same way in all languages, and why should the

same adverb positions correspond to each other in all languages? (Some concrete grammars of `Lang` insert the adverb before or after the negation adverb, not before or after the verb.) For German, I don't see such a difference: "*we always park our car here*" – "*wir parken unseren Wagen immer hier*". ◁

```
    AdvVPSlash : VPSlash -> Adv -> VPSlash ;  -- use (it) here
    AdVVPSlash : AdV -> VPSlash -> VPSlash ;  -- always use (it)

    VPSlashPrep : VP -> Prep -> VPSlash ;     -- live in (it)
```

Q7: Why is it useful to have both `AdvVP` and `AdvVPSlash`? In `LangEng`, we have a difference:

```
Lang> p -cat=S "I read today the book"
UseCl (TTAnt TPast ASimul) PPos (PredVP (UsePron i_Pron)
      (ComplSlash (AdvVPSlash (SlashV2a read_V2) today_Adv)
                  (DetCN (DetQuant DefArt NumSg) (UseN book_N))))

Lang> p -cat=S "I read the book today"
UseCl (TTAnt TPres ASimul) PPos (PredVP (UsePron i_Pron)
      (AdvVP (ComplSlash (SlashV2a read_V2)
                         (DetCN (DetQuant DefArt NumSg) (UseN book_N)))
             today_Adv))
```

Is there a difference in relativising an adverbially modified noun phrase. i.e. *the book here, which I bought yesterday* from *I bought (the book here) yesterday*, versus *the book, which I bought here yesterday* from *I bought (the book) here yesterday.*

Remark 18. *The rule* ***`VPSlashPrep`*** *is meant to be used to construct relative clauses by extracting the noun phrase of an adverbial of type* ***`Prep =< Adv/NP`****. For example, from the sentence "we (live (in the city):**`Adv`**):`VP`" we can obtain a relativization of the noun phrase in the adverb: "the city we (live (in:**`Adv/NP`**)):`VP/NP`". This construction is available in English, but certainly not in German, so it is questionable whether it can belong to the multilingual grammar* ***`Lang`****.*[17] *But the rule leads to ambiguities in German:*

```
PredVP (UsePron he_Pron)
  (AdvVP (UseV sleep_V)
         (PrepNP in_Prep (DetCN (DetQuant DefArt NumSg) (UseN house_N))))
PredVP (UsePron he_Pron)
  (ComplSlash (VPSlashPrep (UseV sleep_V) in_Prep)
              (DetCN (DetQuant DefArt NumSg) (UseN house_N)))
er schläft im Haus
er schläft im Haus
```

Aarne: c.f. ***`ExtraEng|Swe`*** *for use of the rule for preposition stranding. Well, in German we have a limited form of preposition stranding for pronominal adverbs "dafür", "damit", "daran", "darauf", "davor" etc., e.g. "da arbeite ich nicht für|mit|dran", "da warte ich nicht drauf", "da warne ich vor". But while we can say "auf den warte ich nicht", we cannot say "den warte ich nicht drauf", the separation is only possible with da+prep ≃ prep+das.*

[17] In German, the preposition and relative pronoun combine to a relative adverb: "*die Stadt, in der wir leben*" or "*die Stadt, worin wir leben*".

3.1.5. Adverb

According to `gf-rgl/src/Common.gf`, there are the following adverb categories:

```
    Adv ;   -- verb-phrase-modifying adverb        e.g. "in the house"
    AdV ;   -- adverb directly attached to verb    e.g. "always"
    AdA ;   -- adjective-modifying adverb          e.g. "very"
    AdN ;   -- numeral-modifying adverb            e.g. "more than"
    IAdv ;  -- interrogative adverb                e.g. "why"
    CAdv ;  -- comparative adverb                  e.g. "more"
```

The difference between `Adv` and `AdV` seems unplausible; a language-independent difference can arise from the meaning, not from the position in English (unless these correspond to each other). Q8: : Or is the difference related to questions of scope? I.e., whether a (quantified) complement of the verb is within or outside the scope of the adverb? But this also seems not fixed in the syntax.

Construction rules. The full constructions of `Adverb.gf` are:

```
-- The two main ways of forming adverbs are from adjectives and by
-- prepositions from noun phrases.

    PositAdvAdj : A -> Adv ;                -- warmly
    PrepNP      : Prep -> NP -> Adv ;       -- in the house
```

Remark 19: The adverb construction `PrepNP : Prep -> NP -> Adv` uses the category `Prep` in the abstract grammar. Such a preposition `p:Prep` may have a language-independent meaning as a binary spatial, directional or temporal relation, hence a translation of `p:Prep` to various concrete languages is not excluded. But even in these cases, the usage of a preposition seems rather language-dependent; the German preposition *um* in *um vier Uhr* and *um Mitternacht* corresponds to different prepositions in English in *at four o'clock* and *around midnight*.[18] Hence, the meaning of an adverb `PrepNP p np` cannot be expected to be compositional, and its translation by *linearize*$\circ$*parse* may be incorrect. At least, the prepositions in `Structural` and `Lexicon` have to be thought of as abstract prepositions, not as English ones.

```
-- Comparative adverbs have a noun phrase or a sentence as object of
-- comparison.

    ComparAdvAdj  : CAdv -> A -> NP -> Adv ; -- more warmly than John
    ComparAdvAdjS : CAdv -> A -> S  -> Adv ; -- more warmly than he runs

-- Subordinate clauses can function as adverbs.

    SubjS  : Subj -> S -> Adv ;             -- when she sleeps

-- Like adverbs, adadjectives can be produced by adjectives.

    PositAdAAdj : A -> AdA ;                -- extremely
```

[18] or: *in English* versus *auf Deutsch*.

```
-- Comparison adverbs also work as numeral adverbs.

    AdnCAdv : CAdv -> AdN ;                  -- less (than five)
```

Remark 20: The use of `CAdv` in `ComparAdvAdj` and `AdnCAdv` is dubious. In English, we get *less well than John* and *less than five*, but *as well as John* and **as as five*. In German, one can say *weniger gut als Johann*, but uses *besser als Johann*, not **mehr gut als Johann*.

Modification rules

```
-- Adverbs can be modified by 'adadjectives', just like adjectives.

    AdAdv  : AdA -> Adv -> Adv ;             -- very quickly
```

Example: `noch:AdAdv` in *noch schnell*

Q9: can't comparative adverbs also be used with infinitives? E.g., *he hoped to write a paper more easily than to paint a picture*

Q10: what about adverb negation? *not likely* = *unlikely*, *not often* = *rarely*, etc. Should this be treated using `(AdAdv not:AdA adv:Adv):Adv`, where `AdAdv` concatenates the two strings of its arguments? (For `CAdv`, `Eng` has `s : Polarity => Str`.)

3.1.6. Numerals

Todo 11: extract from CatGer and explain the intended rules

```
cat Numeral ;

fun digits2num : Digits -> Numeral ;
fun num : Sub1000000 -> Numeral ;

cat Digits ;

fun IDig : Dig -> Digits ;
fun IIDig : Dig -> Digits -> Digits ;
fun dconcat : Digits -> Digits -> Digits ;
fun nd10 : Sub10 -> Digits ;
fun nd100 : Sub100 -> Digits ;
fun nd1000 : Sub1000 -> Digits ;
fun nd1000000 : Sub1000000 -> Digits ;
fun num2digits : Numeral -> Digits ;
```

3.1.7. Sentences, Clauses and Imperatives

```
abstract Sentence = Cat ** {

-- Clauses

-- The predication rule form a clause whose linearization gives a table of
-- all tense variants, positive and negative.  Clauses are converted to
```

```
-- sentences (with fixed tense and polarity) with the UseCl function below.

    PredVP    : NP -> VP -> Cl ;         -- John walks

-- Using an embedded sentence as a subject is treated separately.  This can
-- be overgenerating. E.g. "whether you go" as subject is only meaningful
-- for some verb phrases.

    PredSCVP  : SC -> VP -> Cl ;         -- that she goes is good
```

While an infinitival subject is recognized correctly,

```
  Lang> p -cat=Cl "to sleep is good"
  PredSCVP (EmbedVP (UseV sleep_V)) (UseComp (CompAP (PositA good_A)))
```

when the infinitival subject is moved and replaced by a correlate *it*, we get wrong trees. (See Remark 12.)

```
-- Clauses missing object noun phrases

-- This category is a variant of the 'slash category' S/NP of GPSG and
-- categorial grammars, which in turn replaces movement transformations in
-- the formation of questions and relative clauses. Except SlashV2, the
-- construction rules can be seen as special cases of function composition,
-- in the style of CCG.

    SlashVP   : NP -> VPSlash -> ClSlash ;      -- (whom) he sees
    AdvSlash  : ClSlash -> Adv -> ClSlash ;     -- (whom) he sees today
    SlashPrep : Cl -> Prep -> ClSlash ;         -- (with whom) he walks
    SlashVS   : NP -> VS -> SSlash -> ClSlash ; -- (whom) she says that he loves

-- *Note* the set is not complete and lacks e.g. verbs with more than 2 places.

-- Imperatives

-- An imperative is straightforwardly formed from a verb phrase. It has
-- variation over positive and negative, singular and plural. To fix these
-- parameters, see Phrase.gf.

    ImpVP     : VP -> Imp ;             -- love yourselves
    AdvImp : Adv -> Imp -> Imp ;        -- please love yourselves

-- Embedded sentences

-- Sentences, questions, and infinitival phrases can be used as subjects
-- and (adverbial) complements.

    EmbedS    : S  -> SC ;              -- that she goes
    EmbedQS   : QS -> SC ;              -- who goes
```

```
    EmbedVP   : VP -> SC ;                -- to go

-- Sentences

-- These are the 2 x 4 x 4 = 16 forms generated by different combinations
-- of tense, polarity, and anteriority.

    UseCl    : Temp -> Pol -> Cl  -> S ;   -- she had not slept
    UseQCl   : Temp -> Pol -> QCl -> QS ;  -- who had not slept
    UseRCl   : Temp -> Pol -> RCl -> RS ;  -- that had not slept
    UseSlash : Temp -> Pol -> ClSlash -> SSlash ; -- (that) she had not seen

-- An adverb can be added to the beginning of a sentence, either with comma
-- ("externally") or without:

    AdvS     : Adv -> S  -> S ;            -- then I will go home
    ExtAdvS  : Adv -> S  -> S ;            -- next week, I will go home

-- This covers subjunctive clauses, but they can also be added to the end.

    SSubjS    : S -> Subj -> S -> S ;       -- I go home, if she comes

-- A sentence can be modified by a relative clause referring to its contents.

    RelS     : S -> RS -> S ;              -- she sleeps, which is good
}
```

Notice that `SSubjS` does not use `Adverb.SubjS : Subj -> S -> Adv` to combine its arguments `Subj` and `S` and then add the adverbial sentence at the end. But `AdvVP, ExtAdvVP : VP -> Adv -> VP` add an adverb at the end of the `a2` field of a verb phrase, so this can be used before adding the subject by `PredVP`. Q11: Does this lead to spurious ambiguities?

(Scope problems: "*ich will nicht (A, weil B)*" versus "*ich will (nicht A), weil B*" = "*weil B, will ich nicht A*".)

The rule `PredSCVP` can use any sentence, question or infinitive as subject with a verb phrase. This is massively overgenerating. Interrogative subjects may occur as complements of adjective phrases with copula verbs, e.g. *why this is the case, is unknown*, or by passivization from verbs with interrogative objects, e.g. *why this is the case, was asked by many*. It seems there are no verbs that need an interrogative subject, though there are verbs that need sentential or infinitival subjects, e.g. *that John is a fool, doesn't shock us* or *to not get troubles pleased him*. Yet, it may be reasonable to not classify verbs according to their subject category, as this seems systematically overloaded: a subject sentence like *that John is a fool* can be transformed into a nominal subject for the same verb, e.g. *the fact that John is a fool*. If we had verb categories like `SV`, `QV`, `IV` for verbs with sentential, interrogative or infinitival subject, we should also have noun categories like `NS`, `NQ` and `NV` for nouns that take these subjects as objects (and perhaps a nominalization operation that derives such nouns from the corresponding verbs).

3.1.8. Questions and Interrogative Pronouns

```
abstract Question = Cat ** {
```

```
-- A question can be formed from a clause ('yes-no question') or
-- with an interrogative.

  fun
    QuestCl     : Cl -> QCl ;            -- does John walk
    QuestVP     : IP -> VP -> QCl ;      -- who walks
    QuestSlash  : IP -> ClSlash -> QCl ; -- whom does John love
    QuestIAdv   : IAdv -> Cl -> QCl ;    -- why does John walk
    QuestIComp  : IComp -> NP -> QCl ;   -- where is John

-- Interrogative pronouns can be formed with interrogative determiners,
-- with or without a noun.

    IdetCN    : IDet -> CN -> IP ;       -- which five songs
    IdetIP    : IDet       -> IP ;       -- which five

-- They can be modified with adverbs.

    AdvIP     : IP -> Adv -> IP ;        -- who in Paris

-- Interrogative quantifiers have number forms and can take number modifiers.

    IdetQuant : IQuant -> Num -> IDet ;  -- which (five)

-- Interrogative adverbs can be formed prepositionally.

    PrepIP    : Prep -> IP -> IAdv ;     -- with whom

-- They can be modified with other adverbs.

    AdvIAdv   : IAdv -> Adv -> IAdv ;    -- where in Paris

-- Interrogative complements to copulas can be both adverbs and pronouns.

    CompIAdv  : IAdv -> IComp ;          -- where (is it)
    CompIP    : IP   -> IComp ;          -- who (is it)

-- More IP, IDet, and IAdv are defined in Structural.gf

-- Wh questions with two or more question words require a new, special category.

  cat
    QVP ;                                -- buy what where
  fun
    ComplSlashIP  : VPSlash -> IP -> QVP ;   -- buys what
    AdvQVP        : VP  ->   IAdv -> QVP ;   -- lives where
    AddAdvQVP     : QVP ->   IAdv -> QVP ;   -- buys what where
```

```
    QuestQVP      : IP -> QVP -> QCl ;       -- who buys what where
}
```

3.1.9. Relative Clauses and Relative Pronouns

```
-- The simplest way to form a relative clause is from a clause by a pronoun
-- similar to "such that".

    RelCl    : Cl -> RCl ;            -- such that John loves her

-- The more proper ways are from a verb phrase or a sentence with a missing
-- noun phrase.

    RelVP    : RP -> VP -> RCl ;      -- who loves John
    RelSlash : RP -> ClSlash -> RCl ; -- whom John loves

-- Relative pronouns are formed from an 'identity element' by prefixing
-- or suffixing (depending on language) prepositional phrases or genitives.

    IdRP  : RP ;                      -- which
    FunRP : Prep -> NP -> RP -> RP ;  -- the mother of whom
```

So, `RelVP` is used to relativize the nominal subject of a clause, `RelSlash` to relativize a nominal object of a clause. What is called here a relative pronoun `RP` can be any relativizing noun phrase or prepositional phrase; e.g. the relativization may come from a relativizing possessive, like *mit dessen Freunden.*

3.1.10. Conjunction

(todo)

3.1.11. Phrase

```
--1 Phrase: Phrases and Utterances

abstract Phrase = Cat ** {

-- When a phrase is built from an utterance it can be prefixed
-- with a phrasal conjunction (such as "but", "therefore")
-- and suffixing with a vocative (typically a noun phrase).

  fun
    PhrUtt   : PConj -> Utt -> Voc -> Phr ; -- but come here, my friend

-- Utterances are formed from sentences, questions, and imperatives.

    UttS      : S   -> Utt ;               -- John walks
    UttQS     : QS  -> Utt ;               -- is it good
    UttImpSg  : Pol -> Imp -> Utt ;        -- (don't) love yourself
    UttImpPl  : Pol -> Imp -> Utt ;        -- (don't) love yourselves
```

```
    UttImpPol : Pol -> Imp -> Utt ;         -- (don't) sleep (polite)

-- There are also 'one-word utterances'. A typical use of them is
-- as answers to questions.
-- *Note*. This list is incomplete. More categories could be covered.
-- Moreover, in many languages e.g. noun phrases in different cases
-- can be used.

    UttIP     : IP   -> Utt ;              -- who
    UttIAdv   : IAdv -> Utt ;              -- why
    UttNP     : NP   -> Utt ;              -- this man
    UttAdv    : Adv  -> Utt ;              -- here
    UttVP     : VP   -> Utt ;              -- to sleep
    UttCN     : CN   -> Utt ;              -- house
    UttCard   : Card -> Utt ;              -- five
    UttAP     : AP   -> Utt ;              -- fine
    UttInterj : Interj -> Utt ;            -- alas

-- The phrasal conjunction is optional. A sentence conjunction
-- can also be used to prefix an utterance.

    NoPConj   : PConj ;                    -- [plain phrase without conjunction in front]
    PConjConj : Conj -> PConj ;            -- and

-- The vocative is optional. Any noun phrase can be made into vocative,
-- which may be overgenerating (e.g. "I").

    NoVoc   : Voc ;                        -- [plain phrase without vocative]
    VocNP   : NP -> Voc ;                  -- my friend

}
```

Remark 21: The category `PConj` is dubious: why should it be possible to prefix any utterance with any conjunction in all languages? And why is the example *therefore* a phrasal conjunction? (see also Remark 101).

3.1.12. Text

```
-- Texts are built from an empty text by adding Phrases,
-- using as constructors the punctuation marks ".", "?", and "!".
-- Any punctuation mark can be attached to any kind of phrase.

abstract Text = Common ** {
  fun
    TEmpty     : Text ;                 -- [empty text, no sentences]
    TFullStop  : Phr -> Text -> Text ;  -- John walks. ...
    TQuestMark : Phr -> Text -> Text ;  -- Are they here? ...
    TExclMark  : Phr -> Text -> Text ;  -- Let's go! ...
}
```

3.1.13. Structural

3.1.14. Idiom

```
--1 Idiom: Idiomatic Expressions

abstract Idiom = Cat ** {

-- This module defines constructions that are formed in fixed ways,
-- often different even in closely related languages.

  fun
    ImpersCl  : VP -> Cl ;        -- it is hot
    GenericCl : VP -> Cl ;        -- one sleeps

    CleftNP   : NP  -> RS -> Cl ; -- it is I who did it
    CleftAdv  : Adv -> S  -> Cl ; -- it is here she slept

    ExistNP   : NP -> Cl ;        -- there is a house
    ExistIP   : IP -> QCl ;       -- which houses are there

-- 7/12/2012 generalizations of these

    ExistNPAdv : NP -> Adv -> Cl ;    -- there is a house in Paris
    ExistIPAdv : IP -> Adv -> QCl ;   -- which houses are there in Paris

    ProgrVP   : VP -> VP ;        -- be sleeping

    ImpPl1    : VP -> Utt ;       -- let's go

    ImpP3     : NP -> VP -> Utt ; -- let John walk

-- 3/12/2013 non-reflexive uses of "self"

    SelfAdvVP : VP -> VP ;        -- is at home himself
    SelfAdVVP : VP -> VP ;        -- is himself at home
    SelfNP    : NP -> NP ;        -- the president himself (is at home)

}
```

3.1.15. Tense

```
--1 Common: Structures with Common Implementations.

-- This module defines the abstract parameters of tense, polarity, and
-- anteriority, which are used in Phrase.gf to generate different
-- forms of sentences. Together they give 4 x 2 x 2 = 16 sentence forms.

-- These tenses are defined for all languages in the library. More tenses
-- can be defined in the language extensions, e.g. the "passe simple" of
-- Romance languages in ../romance/ExtraRomance.gf.
```

```
abstract Tense = Common ** {

  fun
    TTAnt : Tense -> Ant -> Temp ;  -- [combination of tense and anteriority,
                                   --  e.g. past anterior]

    PPos : Pol ;           -- I sleep  [positive polarity]
    PNeg : Pol ;           -- I don't sleep [negative polarity]

    TPres  : Tense ;       -- I sleep/have slept [present]
    ASimul : Ant ;         -- I sleep/slept [simultaneous, not compound]
    TPast  : Tense ;       -- I slept [past, "imperfect"]      --# notpresent
    TFut   : Tense ;       -- I will sleep [future]            --# notpresent
    TCond  : Tense ;       -- I would sleep [conditional]      --# notpresent
    AAnter : Ant ;         -- I have slept/had slept           --# notpresent
                           --      [anterior, "compound", "perfect"]
}
```

3.1.16. Transfer

A ***structural transfer function*** maps (abstract) trees to trees and is defined by pattern matching over tree constructors. The pattern matcher of `GF` recognizes a syntactic construction as tree constructor only if an abstract module declares it by a `data` declaration instead of a `fun` declaration. `Transfer` contains mainly two structural transfer functions[19],

```
  active2passive : Cl -> Cl
  digits2numeral : Card -> Card
```

The first one transfers clauses in active voice into clauses in passive voice and is defined in

```
abstract Transfer = Sentence, Verb, Adverb, Structural, NumeralTransfer ** {

fun
  active2passive : Cl -> Cl ;
def
  active2passive (PredVP subj (ComplSlash (SlashV2a v) obj)) =
    PredVP obj (AdvVP (PassV2 v) (PrepNP by8agent_Prep subj)) ;
  active2passive (PredVP subj (AdvVP (ComplSlash (SlashV2a v) obj) adv)) =
    PredVP obj (AdvVP (AdvVP (PassV2 v) (PrepNP by8agent_Prep subj)) adv) ;
  active2passive (PredVP subj (AdVVP adv (ComplSlash (SlashV2a v) obj))) =
    PredVP obj (AdVVP adv (AdvVP (PassV2 v) (PrepNP by8agent_Prep subj))) ;
  active2passive cl = cl ;
}
```

The submodules `Sentence`, `Verb`, `Adverb` have `data` declarations for the syntactic constructions `PredVP`, `ComplSlash`, `SlashV2a`, `AdvVP` and `AdVVP` used in the patterns here, while `Structural`

[19]The `GF`-book presents a further transfer function `aggr : S -> S` for aggregation on p. 147, 148.

has just a `fun` declaration for `by8agent_Prep`, for example. (All binary verbs `v:V2` can be used in `PassV2 v`, not just transitive binary verbs in the linguistic sense.)

The other main transfer function converts between represenations of numbers and numeral words and is defined in the submodule `NumericalTransfer` by

```
fun digits2numeral : Card -> Card ;
def
  digits2numeral (NumDigits d) = NumNumeral (digits2num d) ;
  digits2numeral n = n ;
```

where the embeddings of digits, decimals and numerals into `Card` are `data` declarations

```
    NumDigits  : Digits  -> Card ;  -- 51
    NumDecimal : Decimal -> Card ;  -- 3.14, -1, etc
    NumNumeral : Numeral -> Card ;  -- fifty-one
```

in `Noun` and `digits2num : Digits -> Numeral` is one of many auxiliary transfer functions of `NumeralTransfer` dealing with representations of various kinds of numbers and number words (cf. Subsection 5.1.3).

Structural transfer functions have no linearizations, so there are no corresponding concrete modules `TransferGer.gf` and `NumericalTransferGer.gf`.

The original (unshortened) file `Transfer.gf` also gives some examples like

```
  > p "she sees him" | pt -transfer=active2passive | l
  he is seen by her
  > p -cat=NP "3 cats with 4 dogs" | pt -transfer=digits2numeral | l
  three cats with four dogs
```

However, the flag `-transfer` to `pt` is no longer supported (since version `gf-3.9`); it used to recursively go down a tree and apply the given transfer function to subtrees. One can still use the flag `-compute` and provide a tree containing transfer functions:

```
Lang> pt -tr -compute (UseCl (TTAnt TFut AAnter) PPos
  (active2passive (PredVP (UsePron she_Pron)
                      (ComplSlash (SlashV2a see_V2) (UsePron he_Pron))))) | l
UseCl (TTAnt TFut AAnter) PPos
  (PredVP (UsePron he_Pron) (AdvVP (PassV2 see_V2)
                                (PrepNP by8agent_Prep (UsePron she_Pron))))
he will have been seen by her
```

Thus, transfer functions can only be used as part of a tree supplied to `pt -compute`. The flag `transfer` of `pt` being removed, we cannot easily apply transfer functions to a parse result.

Remark 22: To overcome this, in Section 6 we show how to re-introduce a version of `pt -transfer=f` to the `gf`-shell and provide a variation `DGrammar` of `Grammar` which makes all syntactic constructions usable in tree patterns.

3.1.17. Extra and Extend

Reflexive noun phrase constructions from `Extra`:

```
  cat
    RNP ;     -- reflexive noun phrase, e.g. "my family and myself"
    RNPList ; -- list of reflexives to be coordinated, e.g. "my family, myself, everyone"

-- Notice that it is enough for one NP in RNPList to be RNP.

  fun
    ReflRNP : VPSlash -> RNP -> VP ;   -- support my family and myself

    ReflPron : RNP ;                   -- myself
    ReflPoss : Num -> CN -> RNP ;      -- my family

    PredetRNP : Predet -> RNP -> RNP ; -- all my brothers

    ConjRNP : Conj -> RNPList -> RNP ; -- my family, John and myself

    Base_rr_RNP : RNP -> RNP -> RNPList ;       -- my family, myself
    Base_nr_RNP : NP  -> RNP -> RNPList ;       -- John, myself
    Base_rn_RNP : RNP -> NP  -> RNPList ;       -- myself, John
    Cons_rr_RNP : RNP -> RNPList -> RNPList ;   -- my family, myself, John
    Cons_nr_RNP : NP  -> RNPList -> RNPList ;   -- John, my family, myself
```

Some more constructions using `RNP` are declared in `Extend`:

```
    AdvRNP : NP -> Prep -> RNP -> RNP ;  -- a dispute with his wife
    AdvRVP : VP -> Prep -> RNP -> VP ;   -- lectured about her travels
    AdvRAP : AP -> Prep -> RNP -> AP ;   -- adamant in his refusal

    ReflA2RNP : A2 -> RNP -> AP ;        -- indifferent to their surroundings
                                         -- NOTE: generalizes ReflA2
    PossPronRNP : Pron -> Num -> CN -> RNP -> NP ;
                                -- his abandonment of his wife and children
```

Remark 23: The examples given for the rules `AdvRNP`, `AdvRVP` and `AdvRAP` are unfortunate: *to dispute sth with sb*, *to lecture about sth*, *(to be) admant in sth* seem to be a noun, verb and adjective with fixed prepositions to combine with a complement, i.e. these examples should be handled by complementation rules. For other examples like *dispute in their office*, *recommended on the day of his resign*, *prevented by her own nature*, a category `RAdv` of ***reflexive adverb*** with (only?) constructor

```
  PrepRNP : Prep -> RNP -> RAdv ;
```

seems appropriate, and `AdvRNP`, `AdvRVP` and `AdvRAP` should be replaced by constructions

```
  RAdvNP : NP -> RAdv -> RNP ;
  RAdvVP : VP -> RAdv -> VP ;
  RAdvAP : AP -> RAdv -> AP ;
```

But as for adverbs constructed by `PrepNP`, the meaning of reflexive adverbs is non-compositional, since prepositions usually don't have the same meaning in different languages.

3.2. Limitations, Deficits and Problems

Todo 12: check any claimed problem!

3.2.1. n-ary Verbs and Predicates

The predicates of arity $n > 1$ can be *atomic*, i.e. n-ary verbs or n-ary adjectives with auxiliary verb, or *compound*, i.e. arise from $(n+1)$-ary atomic predicates combined with an object, or be a sub- or coordination of n-ary predicates.

`GF` has categories `V` (unary verbs), `V2` (binary verbs), `V3` (ternary verbs), `VP` (verb phrase = unary predicat), `Comp` (complement of a copula verb), `VPSlash` (verb phrase missing an a complement = binary predicate), and some further categories like `VS` (verb taking a nominal and a sentential complement), `V2V` (verb taking two nominal and an infinite complement),

Problem 2. *In order to be able to translate verbs (or nouns, adjectives of arity 2) from one language to another, we not only need a common constant* `v`*, but also need to map the semantic roles properly between languages. For example, to map English "give sb sth" to German "jmdm etwas geben", the implementation type of* `V3` *in both languages has record fields* `c2` *and* `c3` *to store the case or preposition needed when attaching the first or second complement of* `give:V3`*. But what is intended to be the first resp. second complement?* `GF` *has no notion of direct vs. indirect object, only the* `c`*-slots.*

Suppose `ci` *is a complement type, and* `vn : cn -> ... -> (c2 -> VP)` *is the type of an n-ary verb. (So* `VP` *corresponds to* `c1 -> S` *and* `VPSlash` *to* `c2 -> VP`*, where* `c1` *is the type of the subject,* `c2` *the type of the object.) From* `V3` *on, there is no obvious inherent ordering of complements. Even for ditransitive verbs* `V2` *of English,* `mkV3 give noPrep noPrep` *only by convention defines* `give sb sth` *and* `mkV3 give noPrep to_Prep` *defines* `give sth to sb`*. (There is the further mess with the pronoun switch in "give it her"!) Can we enforce that* `c3` *represents "the indirect",* `c2` *"the direct" object? Can we force users to use* `mkV3` *always in the sense of* `mkV3 v c2 c3`*, to guarantee (*`give sth:c2 to-sb:c3 =) give sb:c3 sth:c2 = jmdm:c3 etwas:c2 geben`*, or can we rely on fixing this via application grammars?*

Only with such conventions like `vn : cn -> ... (c2 -> VP)` *can we use the* slash*-rules*

```
SlashiVn : Vn -> NP -> cn -> .... [-> ci] -> ... (c1-> t)
```

consistently, and translate properly. (`SlashiVj v np` *stores the* `np` *into the* `ci`*-field of the* `j`*-ary verb* `v:Vj`*.) Not cases or prepositions must match accross languages, but argument roles (or GF-argument numbers).*

Problem 3. *There is the annoying ambiguity of complementizing a* `V3` *one by one,*

`ComplSlash (Slash3V3 v3 np3) np2` *vs.* `ComplSlash (Slash2V3 v3 np2) np3`

which in `LangEng` *construct the same implementation record. A* `VPSlash` *can be* used *in different ways, so there is no doubt about the construction rules for* `VPSlash`*:*

```
Slash2V3 : V3  -> NP -> VPSlash ;  -- give it (to her)
SlashV2V : V2V -> VP -> VPSlash ;  -- beg (her) to go
...
```

Can we –to get rid of the ambiguities with `(ComplSlash (SlashiV3 v npi) np(5-i))`– *replace the single complementation rule*

```
ComplSlash : VPSlash -> NP -> VP ; -- love it
```

by different complementation rules (for each construction) that build a `VP` *from the underlying ternary verb directly, like*

```
ComplV3 : V3 -> NP -> NP -> VP ;    -- give it to her
ComplV2V : V2V -> VP -> NP -> VP ;  -- beg her to go
... ?
```

It seems that the modifications of `VPSlash` *also are available as modifications of* `VP`*. (The same would be needed for verbs with higher arity.).*

Remark 24 : First, the ambiguity does not hold in all languages: in `Ger` *of* `gf-3.3`*, the linearizations of* `(ComplSlash (SlashiV3 v3 npi) np(5-i))` *give* `np3 ++ np2` *for* $i = 2$*, but* `np2 ++ np3` *for* $i = 3$*. (The* `SlashiV3` *and* `ComplSlash` *use* `insertObj`*, which adds* `obj!a` *to the left of* `vp.n2`*. More precisely, an* `np` *with* `isPron = True` *was inserted at the front of* `VP.n0`*, those with* `isPron = False` *at the front of* `VP.n2`*, and the clauses order complements as in* `n0 < neg < n2 < ap`*.) In* `Eng`*,* `(ComplSlash (SlashiV3 v3 npi) np(5-i))` *for* $i = 2, 3$ *both linearize to* `v3 ++ np2 ++ np3`*, the indirect object last. But then the same tree has different meanings in* `Ger` *and* `Eng`*, at least if* `v.c2` *and* `v.c3` *are the same.*

Second, collect a set of examples for trees using `ComplSlash`*. Then we can test the above suggestion and see if we don't lose other trees we wanted to keep. (Can we write a normalization of existing trees to avoid* `ComplSlash`*? But still, GF-external programs for tree transformations had to be adjusted to such a change in the RGL.)*

Problem 4. *There are similar ambiguities between the following trees:*

```
(ComplVV vv (ComplSlash vps np)) =?= (ComplSlash (SlashVV vv vps) np)
(ComplSlash (SlashV2VNP v2v np1 vps) np2) =?=
(ComplSlash (SlashV2V v2v (ComplSlash vps np2) np1)
```

For example, we get four trees for the following example (trees hidden):

```
TestLang> p -cat=VP -tr "promise him to let my wife read the book" | l
promise him to let my wife read the book
promise him to let my wife read the book
promise him to let my wife read the book
promise him to let my wife read the book
```

arising from

```
ComplSlash (SlashV2VNP versprechen him
              (SlashV2VNP lassen (my wife) read)) (the book)
ComplSlash (SlashV2V versprechen
              (ComplSlash (SlashV2VNP lassen (my wife) read) (the book))) him
ComplSlash (SlashV2V versprechen
   (ComplSlash (SlashV2V lassen (ComplSlash read (the book))) (my wife))) him
ComplSlash (SlashV2V versprechen
      (ComplSlash (SlashV2V lassen (ComplV2 read (the book))) (my wife))) him
```

Ok, the final tree is from my added rule `ComplV2` *and does not belong to* `Lang`*.*

3.2.2. Ambiguities in Common Nouns

Problem 5. *A similar cause of ambiguities are the modifications of a* `CN`*. The RGL-rules modify a* `CN` *by an* `AP` *attribute, an* `RCl` *relative clause, or an* `Adv` *adverbial, etc. The string obtained depends on the ordering of modifiers in the tree, if the modifier extends the same* `cn.s` *field, but if different fields* `cn.rel`*,* `cn.adv`*,* `cn.ext` *are modified, one obtains spurious ambiguities depending on the ordering of modifications. (Emptyness tests of the fields, or arbitrary extensions?)*

Proposal 2: Since there are no noun categories of `Lang` with sentential or infinitival objects, certain noun phrases, e.g. "*der Glaube, daß ein Gott die Welt erschaffen hat*" or "*die Hoffnung, das Spiel zu gewinnen*", cannot be recognized (Nonsense: Use `SentCN`) `Lang` ought to be extended by noun categories `NS`, `NQ`, `NV` and `NA`, with a systematic way to infer the category of nouns derived from verbs and adjectives. (See also Remark 5 and Remark 8. And Section 5.11.2.) ◁

Problem 6. *Nouns of category* `N2` *and* `N3` *may attach their objects via prepositions. Such prepositional objects are also parsed as adverbial attributes, e.g. "das Warten auf die Abfahrt" (using* `UseN2` *and* `AdvCN`*). Can a parse as prepositional object be enforced by a kind of binding precedence (in the parser), or is this rare enough because of different cases in the prepositions of objects versus those of adverbs?*

3.2.3. Prepositions and Adverbial Dimensions in a Multilingual Grammar

Q12: Can a multilingual grammar like `Lang` have pre- or postpositions at all?

Where prepositiitions are used, like cases, to express which complement function a constituent realizes, they are semantically empty; the multilingual grammar must only be able to identify the complement function across languages. In this sense, a nominal object in instrumental case like `np.s ! v.c2` = `np.s ! instr` and a prepositional object like `v.c2.s ++ np.s ! vp.c2.c` = `"mit" ++ np.s ! dative` amount to the same and are identified by `v.c2`. (For this, we need a language-specific `v.c2 : Prep` with `oper Prep : Type = {s:Str ; c:Case}`, but no multilingual lexical category `cat Prep`.)

But: what to do with moved prepositional objects, as in

```
SlashPrep : Cl -> Prep -> ClSlash ;          -- (with whom) he walks
```

Where ***prepositions*** have semantic content, i.e. where they are used to construct adverbials, like *under the table*, we have a chance that a limited, language-independent number of *adverbial dimensions* like `param AdvDim = loc | temp | dir | ...` might be enough to define `Prep = Adv/NP` in the sense of `fun in_Prep : Prep` with `lin in_Prep = {s = "in"; d = loc}` and `UsePrep : Prep -> NP -> Adv`. This seems to me the better solution. The drawback is that since some prepositions like *in* can construct adverbs in several dimensions, which may lead to a number of ambiguities, at least if we had an `Adv` category for each dimension. (Moreover, an instrumental adverbial in one language may be represented by an instrumental `Case` in another language, etc.) In any case, the prepositions added to `Lang` are *English* prepositions and don't fit very well to other languages; the abstract grammar is not abstract enough.

There are verbs that need an adverbial of a specific dimension to build a verb phrase, i.e. *to stay at a place*, or *an einem Ort wohnen* and *in einem Raum übernachten*. So, at least some adverbials should be treated as complements of verbs rather than modifications of verb phrases.

This would need categories `VAdv d` and `Adv d` of verbs and adverbs depending on a dimension `d:AdvDim` and a dimension-specific rule

```
ComplVAdvDim : (d:AdvDim) -> VAdv d -> Adv d -> VP.
```

(Since an adverbial has an inherent dimension, adverbs cannot adapt to a dimension specified by the verb, in contrast to noun phrases which can adapt to a case specified by the verb.)

3.2.4. Missing Types of Pronouns and Numbers

Demonstratives; Cardinals and Ordinals are there, but what about *n-fold* (`threefold:A,Adv, dreifach:A,dreimal:Adv`), *the n-th*, (`third, dritte`) ?

3.2.5. Missing Notion of Modalities

There are modal verbs as auxiliary verbs, but no notion of modalized adjectives or participles, like `(un)lesbar` = `kann (nicht) gelesen werden` or `das zu lesende Buch` = `das Buch, das gelesen werden muß`

3.2.6. Iterated Modifications

Some modification rules can be used iteratively, like adding an adverb to a clause or verb phrase, or adding a relative clause to a common noun or noun phrase. In these cases, when added constituents are "stacked" like `vp.a2 ++ adv.s` or `np.s ++ np.rc ++ rc.s`, a leading comma in `adv.s` or `rc.s` may be disturbing, but a leading comma in the first of the stacked elements may be necessary. In my opinion, instead of stacking one ought to use a form of coordination of adverbials or relative clauses, and the coordinated constituent could have an introductory comma.

3.2.7. Bounded Embedding Depth

Embedding of verbal prases is limited (with problems for modal verbs and `V2V` in `Ger`); extraction from subconstituents is limited. What else?

Do we need a parameter `vptype = VPactive | VPrefl | VPpass` in `VP` and `VPSlash`? What do we need to implement different passives in `Ger`?

Generally: which properties of subconstituents have to be stored in parameter values?

3.2.8. Overgeneration Due to Empty Constituents

In particular, empty determiners and empty prepositions cause unnecessary or strange trees. `MassNP` (or `PassV2`) is known from the beginning to be overgenerating, as it is applicable to *any* `np:NP` (resp. `v:V2`), independent of inherent parameters of its head noun (resp. verb). But: constructions are *total* functions, so parameters cannot limit their applicability.

But certainly, having categories `NS`, `NA`, `NV` of nouns with restricted kinds of complements could eliminate arbitrary combinations of `N` with complements `SC`.

4. A Sketch of German

Before describing the implementation of a grammar for German in `GF`, we give an overview of main properties of German in standard linguistic notions.

We here only sketch properties of the main lexical and phrasal categories, omitting prepositions, determiners, coordination etc. (What about subordination? Only simple sentences?)

To see how to describe the language by a PMCFG grammar, we have to check in how many pieces a discontinuous phrase can be split and where the pieces can be moved to. Extraposition to the right is the normal choice, but fronting is another.[20]

4.1. Noun and Noun Phrase

Noun phrases can be ***proper names***, e.g. *Johann*, ***personal pronouns***, e.g. *ich*, *wir*, definite and indefinite ***common names***[21] e.g. *das Haus* and *ein Haus*, ***quantified noun phrases***, e.g. *all my children* and ***coordinations of noun phrases***, e.g. *neither my wife nor my children.*

Morphologically, all noun phrases inflect by case, i.e. have forms for *Nominativ*, *Akkusativ*, *Dativ*, *Genitiv*. In addition, common names and pronouns inflect for number, i.e. have *Singular* and *Plural* forms, and personal pronouns have additional possessive forms.[22]

Articles

Number	Singular			Plural	Singular			Plural
Gender	Masc.	Fem.	Neutr.		Masc.	Fem.	Neutr.	
Nominative	der	die	das	die	ein	eine	ein	(einige)
Accusative	den	die	das	die	einen	eine	ein	—
Dative	dem	der	dem	den	einem	einer	einem	—
Genitive	des	der	des	der	eines	einer	eines	—

Proper names can be modified by nominal attributes, e.g. *Johann, ein netter Bursche* or *Karl der Große*, and by relative clauses, e.g. *Caesar, der von Brutus ermordet wurde*; with a definite article or possessive pronoun they can also be modified by adjectival or participial attributes, e.g. *die|unsere kluge Maria* or *die|deine dich liebende Maria* or *der von Brutus ermordete Caesar*). In vocatives, they can also be modified by adjectival attributes, e.g. *lieber Johann.*

Common names have a syntactic arity. Most of them are of syntactic arity 0, i.e. take no complements, but denote unary predicates (in combination with a copula verb), e.g. *Präsident werden*; *der Chef sein*, or *ein Narr sein.* Others take one nominal complement and denote binary predicates, e.g. *Mutter von Johann*, *Glaube an die Freiheit*, or two nominal (resp. prepositional) complements, i.e. *Fahrt von Paris nach London* or *Division von 20 durch 5.*

Nouns derived from adjectives: *dumm* $\mapsto$ *Dummheit* (objects: *neugierig auf* $\Longleftrightarrow$ *Neugier auf?*)

Nouns derived from verbs: *entfernen von* $\mapsto$ *Entfernung von*, *hoffen auf* $\mapsto$ *Hoffnung auf*, (*ich:*`nom` *entferne A:*`acc` *von B:*`dat` $\mapsto$ *das|mein Entfernen des A:*`gen` *von B ?*)

(*schlage A:*`dat` *vor, Inf-zu* $\mapsto$ *Vorschlag an A:*`acc`*, Inf-zu*)

Common names can be modified to ***common nouns*** by adjectival attributes, e.g. *kleines Haus*, adverbial attributes, e.g. *Haus auf dem Berg*, and relative clauses, e.g. *Haus, dem ein Dach fehlt* or *Haus, in dem ich wohnte.* Common nouns inflect, like common names, for number and case,

[20] `LangGer` seems to hande fronting with special rules, but extraposition via the `ext`-field of phrase records.

[21] I use *common name* for the lexical noun categories and *common noun* for their extensions by complement, adjectival or adverbial attribute, and relative clause, i.e. the basic noun phrase missing a determiner. Both `N`, `N2`, `N3` and `CN` are called *common noun* in `GF`'s resource grammar `Lang`.

[22] Should we consider a ***reflexive function*** of nominal objects in clauses?

but in addition also for adjective forms, e.g. *ein kleines Haus*, but *das kleine Haus*; the adjective form depends on the determiner attached when the common noun is extended to a noun phrase.

Pronouns: reflexive, reciprocal, demonstrative, relative, interrogative, possessive, indefinite (*jemand*, *man*)

Functions: possessive, determinative (*du alter Narr*, *wir ahnungslose(n) Esel*)

Also: reflexive possessive: *sein|ihr eigener* vs. *sein|ihr*. See also Remark 14.

The indefinite pronoun *man* can occur in the comparison part of an adjective phrase: *es war einfacher als man erwartet hatte* oder *Johann is dümmer als man sein sollte*. Strangely, *man* agrees with singular verb form, but with singular and plural reflexive pronoun: *man soll sich nicht ärgern* und *man soll einander helfen*, or *man versprach, sich zu bessern* and *man versprach, einander zu helfen*. There are infinitival subjects with indefinite implicit subject, e.g. *sich zu ärgern ist ungesund*, but *einander zu helfen ist gut*. (The indefinite pronoun *jemand* does not agree with reflexive pronoun in plural.)

The difference between reflexive and personal pronoun is to indicate referential identity or difference. For binary verbs, a reflexive object complement often refers to the individual referred to by subject, e.g. *er hat sich selbst gelobt* versus *er hat ihn gelobt*. In imperatives, the individual referred to may be the adressee (i.e. the implicit subject): *help yourself, and God will help you*. With ternary verbs, the reflexive indirect object may refer to the direct object, e.g. *er hat ihn sich selbst überlassen*, or to the subject, e.g. *er hat ihn sich selbst untergeordnet*.[23]

But with complex noun phrases, the it is less clear which referential identities and differences are expressed by the various pronouns: *(sein$_1$ Vater)$_2$ hat ihn$_1$|sich$_2$ gelobt*, and *(jeder Freund (meines Vaters)$_2$)$_1$ half (sich selbst)$_1$|*ihm$_1$|ihm$_2$|mir$_0$|*(mir selbst)$_0$*. Likewise with reflexive possessive pronouns, e.g. *ein Freund meines Bruders hat meinen|seinen|(seinen eigenen)|dessen Kollegen beleidigt*? Or, der Freitag, 10.3.2022, Literatur V: *Kurz danach veröffentlichte Krug ein Buch mit den vielen, wunderbaren Postkarten Beckers an seine Frau Ottilie und ihn.* (seine,er = Krug)

Q14: what about *selbst* combined with personal pronoun: *das schadet ihm selbst* versus *das schadet ihm*? (And differently: *er hat ihm selbst|selber geholfen = er selbst|selber hat ihm geholfen*. Is *selbst* just used to mark an emphasis?)

Functions (and uses of noun phrases)

Predicative function: indefinite common names can, combined with a copula verb, function as unary predicates: *Johann wurde ein Mann*.

Possessive function of noun phrases by *Gen*, resp. by possessive pronoun. Nouns derived from verbs (or adjectives) and the systematic change of arities.

Possessive function of noun phrases in common nouns: *Johanns Haus* or *das|ein Haus von|des Johann* for proper names, by possessive forms *sein Haus* for pronouns, by post-attribute in genitive for common names, *Haus einer|der Frau*, by post-attrivbute in genitive for quantified noun phrases, *Haus jeder Frau*, *Eigentum vieler Frauen*, *Eigentum von vielen Frauen*, by preposition *von* for coordinated noun phrases: *Haus von Johann und seiner Frau* (or pre-attribute in genitive: *weder Johanns noch seiner Frau Haus*)

[23] Q13: For reflexive verbs, it seems that the short reflexive pronoun *sich* instead of *sich selbst* is preferred, e.g. *er hat ihn sich vorgeknöpft*, or *sich schämen*, not *sich selbst schämen*. Do we use *sich selbst* only for the reflexive usage of non-reflexive verbs? (To test the implementation, it is useful to differ between personal *mich,dich* and reflexiv *mich selbst, dich selbst*, while for third person, the difference between *ihn,sie,es* and *sich* is apparent.)

Complement (*subject* or *object*) of verbs, adjectives and common names: as nominal subject of unary common names (= possessive?)[24] *Glaube der Kinder*, *Behauptung von Johann*; as nominal (or prepositional) object of binary or ternary common names: *Mord des Brutus an Caesar*; *Achtung vor dem Gesetz*; *Rücksicht auf die Kranken.* (Common names can have complements of sentential or infinitive form: *Versuch, einen Beweis zu finden.* The complements of a common name are optional, i.e. the common name alone can combine with a copula verb to the predicate of clauses, e.g. *Johannes ist ein Mann*; *Johann ist der Chef*; *er wurde Arzt.*

`GF` has no category `NV` for nouns with (nominal subject and) infinitival complement, e.g. *Versuch Fermat's, seine Behauptung zu beweisen*, or `N2V` for nouns with nominal object and infinitive complement, e.g. *Rat des Johann an uns, weniger zu arbeiten.*

Ordering of complements and modifications:

unmodified common noun: (N2 ++ subject ++ object ++ complement).

modified common noun: (AP ++ (N2 ++ subject ++ object ++ complement)) ++ RelS)
Mutter eines kleinen Sohns, den ich nicht kenne and *Mutter eines kleinen Sohns, die ich kenne* (2 relative extractions?)

basic noun phrase: `np = {s = PreDet ++ Det ++ AP ++ N, ext = Rel ++ Appos}` in
`ich habe np.s getroffen, np.ext`.
{*s = alle meine alten Freunde, ext = die mich nicht vergessen haben, die Guten,*}
{*s = alle meine alten Freunde, ext = die Guten, die mich nicht vergessen haben,*}

Todo 13: discuss syntactic functions only for immediate constituents, i.e. functions in common nouns (or: as complement of common names), in verb phrase (or: as complement of verbs) etc.

Q15: Discuss how the syntactic functions should be accounted for in a `GF`-grammar like `LangGer`. Now there is a `PossNP : CN -> NP -> CN` in `Noun.gf`, not only `PossPron : Pron -> Quant`.

Agreement within noun phrases

Agreement of determiner, adjective and noun in number and case; dependence of determiner and adjective on the gender of the noun; dependence of the adjective on the determiner type.

4.2. Adjective and Adjective Phrase

Morphological adjective:

Usage: Adjectives are used in attributive function and then inflect, e.g. *die schwarze Nacht*, or are used in predicative function and then don't inflect, e.g. *die Nacht war schwarz.* In adverbial function, adjectives are not inflected, e.g. *er hat die Drogen schwarz gekauft* or *der Wagen ist sehr schnell gefahren.* Adjectives can also be used as objects of verbs, e.g. *wir streichen die Fenster blau.*

The attributively used adjective inflects in all degrees according to number and case, and in singular also to gender. Moreover, it inflects according to one of three ***adjective inflection types***. The endings are shown in Table 1 below (from Duden[2] 475-477).

The *strong* (or determinating) type is used for noun phrases missing an article, e.g. *junger Mann*, *junge Frau*, *junges Kind.* The *weak* type is used for noun phrases with definite article, e.g. in accusative *den jungen Mann*, *die junge Frau*, *das junge Kind.* The third, *mixed* type is used

[24] The possessive function corresponds closely to the subject function of the auxiliary verb *haben*, i.e. *er hat eine Frau* ↦ *seine Frau*, and to the subject function of full verbs, e.g. *er versucht, einzuschlafen* ↦ *sein Versuch, einzuschlafen.*

for noun phrases with indefinite article (or possessive pronoun as determiner), i.e. *(d)ein junger Mann*, *(d)eine junge Frau*, *(d)ein junges Kind*. It combines the strong forms in nominative and accusative singular with the weak forms in dative and genitive singular and in plural.

		Strong			Weak			Mixed		
Number	Case	Masc	Fem	Neuter	Masc	Fem	Neuter	Masc	Fem	Neuter
Singular	Nom	-er	-e	-es	-e	-e	-e	-er	-e	-es
	Acc	-en	-e	-es	-en	-e	-e	-en	-e	-es
	Dat	-em	-er	-em	-en	-en	-en	-en	-en	-en
	Gen	-en	-er	-en	-en	-en	-en	-en	-en	-en
Plural	Nom		-e			-en			-en	
	Acc		-e			-en			-en	
	Dat		-en			-en			-en	
	Gen		-er			-en			-en	
		without det			after definite article			after kein,mein		

Table 1: Ending tables of adjective inflection

Adjective phrases may be discontinuous: with a comparison adadjective, e.g. *so|ähnlich – wie* a predicative adjective phrase may be continuous, e.g. *Fritz war so stark wie Johann*, or discontinuous, e.g. *Fritz ist so stark gewesen wie Johann*, and similarly, when the adjective phrase is built from the comparative of an adjective, it may be continuous, e.g. *Fritz ist stärker als Johann*, or discontinuous, e.g. *Fritz ist stärker gewesen als Johann*. In attributive usage, comparative adjective phrases are split by a common name, e.g. *ein stärkerer Junge als Johann*, unless the comparative part is missing, e.g. *ein älterer Herr*.

Todo 14: Internal word order: `Obj ++ AdA ++ A2`: *(ein) seiner Frau sehr treuer (Mann)* or *(der) auf das Ergebnis ziemlich neugierige (Forscher)*.

For split adjective phrases: $\langle$*(ein) so großer (Fehler)*, *wie (deiner)*$\rangle$, as in *das wäre ja ein ebenso großer Fehler gewesen wie deiner*. The comparison part varies in case: "*(ich fand) einen größeren Fehler als deinen*".

4.3. Verb and Verb Phrase

Morphological verbs, i.e. verbs restricted to their inflectional behaviour, have finite, imperative and infinite forms. Finite forms inflect according to

- tense (*Präsens*, *Präteritum*),
- mood (*Indikativ*, *Konjunktiv*),
- number (*Singular*, *Plural*), and
- person (*Erste*, *Zweite*, *Dritte*).

Imperative forms inflect according to

- number (*Singular*, *Plural*).

Infinite forms are

- infinitive (*Infinitiv*, *zu-Infinitiv*),
- participle (*Partizip Präsens*, *Partizip Perfekt*).

The participles can be used in adjectival function and then inflect like adjectives (in *Positiv*), e.g. *das spielende Kind* and *ein gekochtes Ei*, or uninflected in predicative or adverbial function, e.g. *das Kind hat gespielt* and *das Ei ist gekocht* or *er hat ihn spielend überholt* and *sie hat gelassen reagiert.* (Objektprädikativ: *er hat das Ei gekocht gegessen.*)

German has two kinds of ***prefix verb***s: those where the prefix is (not emphasized and) always glued to the stem, e.g. the prefix *um* of the verb *umfáhren* in *er umfährt den Pfosten*, and those where the prefix is (emphasized and) sometimes split from the stem, e.g. the prefix *um* of the verb *úmfahren* in *er fährt den Pfosten um.* The *Partizip Perfekt* and *Infinitiv-zu* of these kinds of prefix verbs also differ: *umfáhren* vs. *úmgefahren*, and *zu umfáhren* vs. *úmzufahren.*

Syntactic verb classification

With respect to their syntactic behaviour, one distinguishes ***full verb***s, which have a meaning and correspond to the logical notion of relation with number (and semantic type) of arguments, from ***auxiliary verb***s and ***copula verb***s, which have no meaning, although some can also be used as a full verb, e.g. *haben* in *ich habe kein Geld.*

The ***syntactic arity*** (resp. *verb frame*) of a full verb specifies which kinds of complements it can combine with to form a clause (resp. a verbal phrase or unary predicate). The number of possible complements can range form zero to almost ten. The verbs of arity 0 can only be combined with the formal (non-denoting or ***expletive***) subject *es*; these are mainly the so-called weather verbs, e.g. *es regnet*, but also a few others, e.g. *es scheint so.*[25] Many complements[26] are ***nominal***, i.e. have the form of noun phrases, including the intransitive verbs, those that take a single, nominal complement (as subject)[27], e.g. *sleep*, almost all of which have their subject in nominative case; but subject in dative or accusative case is also possible, e.g. *mir schwindelt* and *mich friert.* (Subject in genitive case is possible for passive sentences, e.g. *der Toten wird gedacht.*)

There are verbs with several nominal complements, up to almost ten[28]; the order in which the complements have to appear in a clause is at best partially fixed, so they are distinguished by case or a preposition with case, e.g. nominative subject and accusative object: *ich füttere den Hund nicht*, or *den Hund füttere ich nicht*, or nominative subject and two objects with prepositions: *der Lehrer spricht mit dem Schüler über den Aufsatz*, or *über den Aufsatz spricht er mit ihm.* There are verbs with a ***sentential*** complement: *er glaubt, dass der Hund beißt* or *er verspricht ihr, dass er zurückkommt*, and verbs with an ***interrogative*** complement: *er fragt, wer das angeordnet hat* or *er fragt, in welcher Richtung der Bahnhof liegt.* There are verbs with ***infinitival*** complements[29]: *sie will arbeiten*, or *er glaubt, das Spiel zu gewinnen.* Some verbs take a sentential subject and nominal object, e.g. *dass wir gesund sind, freut uns*, some take an infinitival subject and nominal object, e.g. *zu erkranken, betrübt uns.* (Of course, sentential, interrogative and infinitival complements may have a head verb with sentential, interrogative

[25] There are also verbs with very few possible objects, e.g. *es*|*das*|*nichts*|*etwas*|*manches*|*vieles*|*alles tun*

[26] Todo 15: ***prepositional*** complements

[27] We count the subject as first complement, the objects as second, third etc. to be specified by `c1`, `c2`,... in `GF`.

[28] C.S.Peirce's: *person x0 rents person x1 a thing x2 in place x3 for an amount x4 from time x5 to time x6.*

[29] i.e. verb phrases as defined below and denoted by *Inf* or *Inf-zu* here, consisting of a verb in *Infinitiv* or *Infinitiv-zu* form, combined with objects and adverbials.

or infinitival complement, e.g. *er behauptet, daß er nicht wußte, daß|ob er wiederkommt* or *sie wundert sich, warum er fragt, wohin sie geht* or *er verspricht ihr, zu versuchen, ihr zu helfen.*)

Similarly, there are verbs with an ***adjectival*** complement, e.g. *der Wein schmeckt sehr gut* or *du siehst schlechter aus als gestern*, and verbs that take both a nominal and an adjectival complement, e.g. *ich male die Wand blau* or *sie wirkt auf mich kompetenter als du.* (Besides these full verbs with adjectival complements, copula verbs combine with adjective phrases, e.g. *ich bin zufrieden* or *du wirst unglücklich.* Strangely, `GF` declares `become_VA : VA` to be a full verb.)

A classification of verbs therefore ought to tell the number and kind of complements they can take, given in some standard order. Using a case c to name the nominal phrases in this case c, and letting the subject come last in the syntactic arity, the verbs could be classified by

$$\begin{aligned}
\textit{schwindeln} &: \textit{Dat} \to \textit{Clause}\\
\textit{schlafen} &: \textit{Nom} \to \textit{Clause}\\
\textit{füttern} &: \textit{Acc} \to \textit{Nom} \to \textit{Clause}\\
\textit{sprechen} &: \textit{über-Acc} \to \textit{mit-Dat} \to \textit{Nom} \to \textit{Clause}\\
\textit{glauben} &: \textit{dass-S} \to \textit{Nom} \to \textit{Clause}\\
\textit{versprechen} &: \textit{dass-S} \to \textit{Dat} \to \textit{Nom} \to \textit{Clause}\\
\textit{wollen} &: \textit{Inf} \to \textit{Nom} \to \textit{Clause}\\
\textit{glauben} &: \textit{Inf-zu} \to \textit{Nom} \to \textit{Clause}\\
\textit{empfehlen} &: \textit{Inf-zu} \to \textit{Dat} \to \textit{Nom} \to \textit{Clause}\\
\textit{freuen} &: \textit{Acc} \to \textit{dass-S} \to \textit{Clause}\\
\textit{betrüben} &: \textit{Acc} \to \textit{Inf-zu} \to \textit{Clause}
\end{aligned}$$

where *Clause* stands for sentences depending on tense and mood, S for sentences with a fixed tense and mood. Notice that the syntactic arity of a verb is not unique, as for *glauben* above. For ternary words with nominal objects, these verb frames would mention the complements in the ordering[30]

$$\textit{indirect object} < \textit{direct object} < \textit{subject}.$$

For example, the arity of *schenken* then is

$$\textit{schenken} \quad : \quad \textit{Dat} \to \textit{Acc} \to \textit{Nom} \to \textit{Clause}.$$

In contrast to the slash-categories of categorial grammar, the position of the arguments in clauses is not fixed by the syntactic arity of its main verb. For verbs having two complements of the same case, like $\textit{nennen} : \textit{Acc} \to \textit{Acc} \to \textit{Nom} \to \textit{Clause}$, one therefore cannot definitely tell how the two object noun phrases in a sentence correspond to the syntactic arity, e.g. *er nennt seinen Freund den Weltmeister* vs. *er nennt den Weltmeister seinen Freund.* (In these cases, word order is important.) There are a few nullary verbs, like the weather verbs *regnen*, *schneien*, *dämmern* etc., which combine to a clause with the expletive subject *es*, e.g. *heute regnet es*, but also some verbs with higher arity, e.g. *um das Spiel zu gewinnen, bedarf es großer Geschicklichkeit.*

Some complements can be optional, e.g. *sie liest gerade ein Buch* vs. *sie liest gerade.*

Each syntactic arity gives a class of verbs. The ***transitive verb***s are those with a nominal subject in nominative and a single nominal object in accusative, e.g. *einen Aufsatz schreiben*,

[30] or rather $\textit{direct} < \textit{indirect} < \textit{subject}$, to make it the reverse of the typical ordering in subordinate clauses like *weil er ihr einen Blumenstrauß schenkt*? Compare `accdatV` in `ResGer|Eng`. For more than 3 complements, there is no intuitively "standard" ordering, so a convention for the arity of words in a multilingual lexicon is needed.

the ***ditransitive verb***s are those with nominal subject in nominative and two nominal objects in accusative[31], e.g. *der Wagen kostet mich einen Tausender*.

Reflexive verbs are verbs with a nominal subject in nominative and, besides further complements, a nominal object that has to be a reflexive pronoun (in accusative or dative)[32] that agrees in person and number with the subject, e.g. *sich schämen* in *ich schäme mich* or *sich etwas merken* in *ich merke mir den Termin*. The reflexive pronoun is a syntactically necessary complement, but does not correspond semantically to an argument of the predicate named by the verb. A reflexive verb therefore often translates to a non-reflexive verb of smaller arity, e.g. *sich schämen* $\mapsto$ *to be ashamed*; *sich etwas merken* $\mapsto$ *to remember sth*, and hence multilingual grammars cannot have a lexical type of reflexive verbs. Notice the distinction between a reflexive verb and a ***reflexively used*** non-reflexive verb; a non-reflexive verb with nominal (or prepositional) object can, but need not be used with reflexive pronoun as complement, e.g. *jemanden|sich ärgern* and *mit etwas|sich hadern* versus *sich|*jmdn schämen* or *sich|*jmdm etwas merken*.

The ***modal verb***s, i.e. *mögen*, *wollen*, *dürfen*, *können*, *sollen*[33], *müssen*, are the verbs of arity *Inf* $\to$ *Nom* $\to$ *Clause*. (See also p. 54.)

The infinitive is also used in ***accusative cum infinitive*** (ACI)-constructions, mainly with perception verbs: *ich höre den Hund bellen*, or *ich sehe den Hund einen Hasen jagen*. These are like compressed forms of sentential complements: *ich höre, dass|wie der Hund bellt* or *ich sehe, dass|wie der Hund einen Hasen jagt*. So it seems these verbs have a basic arity *dass-S* $\to$ *Nom* $\to$ *Clause* and an arity *Inf* $\to$ *Acc* $\to$ *Nom* $\to$ *Clause* derived by ***subject-to-object raising***: the nominal subject *Nom* of the complement *dass-S* is raised to an object *Acc* of the verb, which simultaneously is the ***implicit subject*** of the verb of the *Inf* complement.

Similarly, the verb *lassen* combines with accusative and infinitive: *ich lasse dich schlafen = ich lasse zu, dass du schläfst*.[34] E.g. *lassen* : *Inf* $\to$ *Acc* $\to$ *Nom* $\to$ *Clause* can be seen as derived from *zulassen* : *dass-S* $\to$ *Nom* $\to$ *Clause*.

Verbs of arity *Inf-zu* $\to$ *Nom* $\to$ *Clause*, e.g. *hoffen* in *ich hoffe, den Wettkampf zu gewinnen*, let their subject be the implicit subject of their *Inf-zu* complement, i.e. *ich hoffe, den Wettkampf zu gewinnen = ich hoffe, daß ich den Wettkampf gewinne*. This is syntactically observable when the complement *Inf-zu* is built with a reflexive or reflexively used verb; the reflexive pronoun then has to agree in person and number with the (implicit) subject: *ich hoffe, mich nicht zu blamieren*; *du hoffst, dich nicht zu blamieren*, etc.

Control verbs are verbs of arity *Inf-zu* $\to$ *Dat* $\to$ *Nom* $\to$ *Clause* or *Inf-zu* $\to$ *Acc* $\to$ *Nom* $\to$ *Clause*. In ***subject-control*** verbs, their subject is the implicit subject of the *Inf-zu* complement, e.g. *ich verspreche dir, mich zu beeilen*. In ***object-control*** verbs, their object is the implicit subject of the *Inf-zu* complement, e.g. *ich rate dir, dich zu beeilen* or *ich ermahne dich, dich zu beeilen*. We have to keep track of the control-feature in translation, since reflexive verbs in the target language need not be reflexive in the source language, e.g. *I promise you to hurry up* $\mapsto$ *ich verspreche dir, mich zu beeilen*, but *I advise you to hurry up* $\mapsto$ *ich rate dir, dich zu beeilen*.

The *Inf-zu* complement may be accompanied by an additional ***correlate*** *es* or *das*, e.g. *ich rate es dir, dich zu beeilen* or *dich zu beeilen, das rate ich dir*. Sometimes, the complement

[31] English: *give sb sth* as opposed to *give sth to-sb*. May, for German, one object be in dative?

[32] The reflexive pronoun *sich* in third person singular can be dative or accusative.

[33] *sollen* can be used to circumscribe indirect imperatives: *er soll schweigen!* or *sie meinen, ich soll arbeiten!*

[34] But *lassen* is also a passive auxiliary verb (*lassen-Passiv*): *ich lasse mich täuschen = ich lasse zu, dass ich getäuscht werde*, or *ich lasse mir ein Haus bauen = ich veranlasse, dass mir ein Haus gebaut wird*.

Inf-zu seems to be a restricted instance of a prepositional object, and then a *pro*-form of the preposition, like *damit*, *daran*, *dazu*, may be necessary in addition to the *Inf-zu* complement: *ich prahle mit meinem Erfolg → ich prahle damit, Erfolg zu haben*, likewise *ich erinnere dich daran, dich zu beeilen*, or *ich bringe dich dazu, dich zu beeilen.*

Q16: Control verbs `V2V` with *Inf*-complement? ACI: *ich sehe|lasse euch euch streiten*; *ich höre ihn sich rasieren*; *ich lasse dich dich schämen.* Or could `isAux:Bool` be omitted in `V2V`?

Auxiliary verbs are morphological verbs (without meaning, hence without syntactic arity) that combine with infinite forms of other verbs to function as predicate in clauses. (The auxiliary verb shows person, number, tense and mood resp. imperative.) German has the following auxiliary verbs:

- the perfect auxiliary verbs *haben* and *sein*, combine with the *Partizip Perfekt* of a verb,
- the future auxiliary verb *werden* combines with a verb's *Infinitive* to *Futur-I*, and with *Partizip Perfekt* and *haben* to build the *Futur-II*
- the passive auxiliary verbs *werden*, *bekommen*, combine with *Partizip Perfekt*, the passive auxiliary verb *lassen* combines with *Inf* (*Xerxes ließ das Meer auspeitschen*), but also with dative nominal object: *er ließ sich:Dat die Haare:Acc schneiden* vs. *er ließ sich:Acc auspeitschen*
- (*lassen* in *ich lasse dich gehen*? Is this an "*admissing*" modal verb? *Person, die sich nicht impfen lassen darf*)

Copula verbs: *sein*, *bleiben*, *werden* (building a predicate with a CN or AP) and(?) *haben* (building a predicate with a CN or NP)

We might give copula verbs the syntactic arities `sein,bleiben,werden : AP -> VP` as well as `sein,bleiben,werden : NP -> VP` and `haben : CN -> VP` as well as `haben : NP -> VP`, e.g. *Pech haben* or *eine Frau haben*, or *die besten Möglichkeiten haben.*

Raising verbs:

- ***subject-to-subject raising***: *scheinen : dass-S → es → Cl*: *es scheint, daß das Wetter sich bessert ↦ das Wetter scheint sich zu bessern.* But also with *scheinen : dass-S → Dat → Cl*: *mir scheint, daß sie Narren sind ↦ sie scheinen mir Narren zu sein*
- ***subject-to-object raising*** = ACI? *ich sehe, daß|wie du den Hund fütterst ↦ ich sehe dich den Hund füttern* or *ich höre, daß es regnet ↦ ich höre es regnen*

  (Infinitive for participle in ACI: *ich habe dich schlafen sehen* instead of *ich habe dich schlafen gesehen*?)

  he wants that we help ↦ he wants us to help; dt. *er will|erwartet von uns, daß wir helfen, ≠ er fordert uns auf, zu helfen*?
- ***object-to-subject raising***? the *Acc* object of an active sentence can be raised to the *Nom* subject in its passive sentence:

  Einen guten Mann zu finden, ist schwer ↦ Ein guter Mann ist schwer zu finden

We treat the verbal gender (*Aktiv*, *Passiv*) and the clausal tenses (*Perfekt*, *Plusquamperfekt* *Futur-I*, *Futur-II*), under verb phrase and clause. (But: don't we have to distinguish verbs that

admit a passive from those that don't? Discuss passive as argument reduction, in comparision to reflexive usage?)

There are verbs that need an adverbial to build a verb phrase, i.e. *an einem Ort wohnen* oder *in einem Raum übernachten*. Such adverbials can be viewed as complements of verbs rather than as modifications of verb phrases. (c.f. page 42 and Proposal 36.)

Verb phrases

A ***basic verb phrase*** is the combination of a verb with expressions realizing all but the subject complement functions given by the syntactic arity of the verb.[35] Basic verb phrases can be coordinated and modified by adverbials to give verb phrases. Roughly, n-ary verbs correspond to n-ary atomic logical predicates, complements to arguments, adverbials to arity-preserving predicate modifiers; then verb phrases correspond to unary complex logical predicates.[36]

In languages like English, where word order is rather rigid and the subject of a basic clause is in initial position, one may assume the remaining final part of the clause to be a constituent of the clause, i.e. its verb phrase; for example, one may then build a complex clause by combining a single subject with a coordination of several verb phrases, e.g. *John walked home and went to bed*. For languages where the ordering of verb complements as clause constituents is relatively free, as in German, a similar argument holds with object complements, e.g. *den Ring hatte er gekauft und ihr geschenkt*, but is not used to justify an "object-missing-clause" as clause constituent.

A perhaps better reason to assumme a verb phrase category is that they provide infinitival complements, so to speak clausal complements of verbs, nouns and adjectives that leave their subject *implicit* and identify it with the subject or an object of the verb, noun or adjective.[37] Even though a basic verb phrase is not a basic clause, it is useful to speak of its ***predicate*** constituent and its ***object*** constituents.

Usage as predicate of clauses: Basic verb phrases can be combined with a subject to build a basic clause. In this case, its predicate consists of two verbal parts, a finite part `vfin` and a (possibly empty) infinite part `vinf`, which in turn consist of verb forms of an auxiliary verb and a full or modal verb. The predicate of a verb phrase has four tenses in addition to the *Präsens* and *Imperfekt* of verbs, namely *Perfekt*, *Plusquamperfekt*, *Futur I*, *Futur II*. The *Perfekt* resp. *Plusquamperfekt* are expressed by the finite form of the full verb's perfect auxiliary, *sein* or *haben*, in *Präsens* resp. *Imperfekt* and the *Partizip Perfekt* of the full verb. The *Futur I* resp. *Futur II* is expressed by a finite form of the future auxilary verb *werden* in *Präsens* and the *Infinitiv* of the full verb, resp. the *Infinitiv* of the verb and followed by the *Infinitiv* of the verb's perfect auxiliary:

[35] In Head Phrase Structure Grammar (HPSG), the basic verb phrase is a *head-complement-structure*, where the verb is the *head*, the other expressions are the *complements*. With respect to the syntactic arity of the verb, these are just the object complements of the verb. So, essentially, $VP = Nom \to Clause$ or `VP = NP\S`.

[36] This correspondence is not precise, since complements and modifiers may contain quantifiers, hence don't represent individuals in the logical sense.

[37] These infinitival phrases often have a clear meaning as action ascribed to the implicit subject. The resource grammar `Lang` assumes verb phrases for all languages.

	vfin	vinf	vfin	vinf
Präsens	glaubt		geht	
Imperfekt	glaubte		ging	
Perfekt	hat	geglaubt	ist	gegangen
Plusquamperfekt	hatte	geglaubt	war	gegangen
Futur I	wird	glauben	wird	gehen
Futur II	wird	geglaubt haben	wird	gegangen sein

If, rather than from a full verb, the predicate comes from a , e.g. *wollen*, which takes *haben* as its perfect auxiliary, the predicate part `vinf` also contains the infinite verb of the infinitival complement of the modal verb. Then its *Partizip Perfekt gewollt* is replaced by its infinitive *wollen*, and in *Futur II* the infinitive of its perfekt auxiliary is put in front of the infinitive of the infinitival complement:

	vfin	vinf
Präsens	will	gehen
Imperfekt	wollte	gehen
Perfekt	hat	gehen wollen
Plusquamperfekt	hatte	gehen wollen
Futur I	wird	gehen wollen
Futur II	wird	haben gehen wollen

In a basic clause, the , as well as the ordering of the predicate parts `vfin` and `vinf`, can vary; in particular, the subject does not occupy a fixed position. Rather than considering the order of objects and adverbs within verb phrases, it makes more sense to view a verb phrase as discontinuous, or even as an unordered collection of predicate parts, object complements and adverbials that only will, together with the subject, be brought into various relative orderings in basic clauses.

In most cases, each of the `vfin` and `vinf` parts, the nominal objects, the sentential and infinitival objects, and the adverbials is internally ordered and continuous. An adjectival complement may be continuous, e.g. *heller als das Meer* in *(er) malt den Himmel heller als das Meer*, or discontinuous, i.e. *(weil er) den Himmel heller malt als das Meer*. Adjectival complements(?) of copula verbs (or: the predicative usage of adjective phrases), may be continuous, e.g. *größer als Johannes* in *(Maria) ist größer als Johannes*, but can also be discontinuous, as in *(weil Maria) größer ist als Johannes*. But nominal complements also need not be continuous; the relative clause or the infinitival object may be right-extracted: *(er) hat das Haus gekauft, das du gebaut hast*, or *(er) hatte den Plan gefaßt, das Haus zu kaufen*. (Likewise in infinitival complements: *den Plan zu fassen, das Haus zu kaufen, (war klug)*.)

If verb phrases are modified by several adverbs or adverbial clauses, are they always combined to a continuous modifier (in a fixed relative order, like: temporal before local adverbs), e.g. *ich lese das Buch morgen im Zug*, or can they be ordered in different ways in a clause, e.g. *morgen lese ich im Zug das Buch*? Q17: Is the ordering preserved, when adverbial clauses replace adverbs?

Verb phrases can also be used to build infinitival complements of verbs, nouns and adjectives, in which case their head verb is put in *Infinitive* or *Infinitive-zu*.

Usage as infinitival complement: when a verb phrase is used as infinitival complement, the order of complements and adverbials of its head verb seems rather fixed: nominal objects and adjectival objects precede the infinite head verb, sentential, infinitival or interrogative complements follow the head verb (or are extracted further to the right). E.g. *jemandem [zu] glauben,*

daß die Sonne untergeht, or *jemanden [zu] fragen, wann der Zug ankommt*, or *den Brüdern (manchmal) einen Gruß [zu] senden*, or *sich (niemals) [zu] schämen* or *(oft) mit dem Gedanken [zu] spielen, eine Weltreise zu machen*, and *das Bild (gerne) schwarz [zu] übermalen.* (Apparently, adverbs are between two nominal objects, before the non-pronominal nominal object, after the pronominal object, but not attached to the verb, as `AdV` suggests.)

Let us call an infinite verb phrase a ***reflexive infinitive***, if its infinite head verb is a reflexive or reflexively used verb. The reflexive infinitive then contains a reflexive (personal or possessive) pronoun, which has to agree in number and person with the *implicit* subject of its head verb, e.g. *sich bemühen, sich nicht zu schämen* and *sich bemühen, seine eigenen Probleme zu lösen.* If the reflexive infinitive is the infinitival object of a (matrix) verb, its implicit subject is the subject or the (direct) object complement of the matrix verb, depending on whether the matrix verb is a subject-control verb or an object-control verb.

Since the head verb of the reflexive infinite may itself be a control verb and have an infinitival object, we can get ***nested infinitival complements***, e.g. *(ich hoffe,) dich [davon] zu überzeugen, mir zu helfen*, or, more complicated, *(ich hoffe,) dich [davon] überzeugen zu können, dir helfen zu lassen* and even *(ich hoffe,) dich [davon] überzeugen zu können, ihr zu raten, sich helfen zu lassen.* In such cases, the embedded infinitival complements with *zu* apparently have to be extracted to the right and may leave a correlate in place, e.g. the *davon* above. Infinitival complements without *zu* stay in place, e.g. *eine Pause machen dürfen*, or *dich das Buch lesen lassen* or *mir helfen können wollen.* (Todo 16: Nested infinitives are complex in Futur-II and Plusquamperfekt.)

But the reflexive infinitive may also be the subject complement of its matrix verb. In this case, its reflexive personal or possessive pronoun is ungoverned, e.g. *sich mit dem Ergebnis abzufinden*, or *to close one's eyes* ↦ *seine|die Augen zu schließen.* This is the reason why verb phrases `vp` need to store nominal objects as tables `vp.nn: Agr => Str`, not as fixed strings `vp.nn: Str`. Q18: Can `Agr` be used to handle reciprocal complements as well, e.g. *ich rate euch, einander zu helfen*, where *einander* only makes use of the plural of `Agr`.

Q19: What about `VP`-negation, and negation of modal verbs: *ich will nicht sprechen* vs. *ich will schweigen.* i.e. can we negate the (full) verb, and so to speak add complements to the dual verb, the negated verb? Do we still get a positive polarity? Maybe clearer with *Inf-zu*: ***verb phrase negation*** *ich hoffe, nicht einzuschlafen* (eng. *I hope not to fall asleep*) vs. ***sentence negation*** *ich hoffe nicht, einzuschlafen* (eng. *I do not hope to fall asleep*, or *ich rate dir, nicht der Letzte zu sein* vs. *ich rate dir nicht, der Letzte zu sein*, or *ich verspreche dir, den Vortrag nicht zu halten* vs. *ich verspreche dir nicht, den Vortrag zu halten.* (Can we admit *nicht* at several positions and let the parser accept only one, by forcing `Pol = PPos` for all following positions?)

Where to put negation? How to combine negation + indefinite noun phrase – *nicht ein = kein*, or mass noun: "*nicht Geld = kein Geld*"? Highly influenced by emphasis, ok: "*nicht eine Minute*". Where is the negation of modal verbs *sie darf sich nicht impfen lassen* and where is the negation with auxiliaries, e.g. "*er hat sich nicht gefragt, ob ...*" and "*wir dürfen das Fest nicht ausfallen lassen*", or *ihr müßt [uns|das Kind|einen Narren] nicht loben*, or "*wir sollen dem|einem Kind das Spielen nicht verbieten*" Do we have negated quantifiers: "*nicht wenige*", "*nicht viele*", "*nicht alle*", "*nicht ein*"? Negated noun phrases can be focused: "*nicht viele Aufgaben konnten sie lösen*" vs. "*viele Aufgaben konnten sie nicht lösen*". But *"*nicht manche*".

Which grammar had a second way to set the polarity depending on negated quantifiers, `Fre`?

Q20: Do the modal verbs (`vv.isAux=True`) or `lassen:V2V.isAux=True` combine with a full

verb (similarly as its perfect and future) as an auxiliary or to a new verb, i.e. *das Buch [nicht] (lesen wollen)* or *[nicht] (das Buch lesen) wollen*? And likewise with passive auxiliary? *(([nicht] (gelesen werden)) wollen):V* and *[nicht] ((gelesen werden) wollen):V*?

What about *scheinen* in *er scheint ein Narr zu sein*? Is this a `VV`-verb with `isAux = False`? Is there a perfect form – *er hat ein Narr zu sein geschienen*??

4.4. Adverb

Todo 17: German has an ***adverb negation***, e.g. *Fritz arbeitet nicht gerne*, where *nicht gerne = ungerne*, likewise *nicht oft = manchmal*, *nicht immer = manchmal nicht*, *nicht jetzt = ein anderes Mal*, in contrast to ***clause negation***, e.g. *Fritz arbeitet jetzt nicht* $\neq$ *Fritz arbeitet nicht jetzt*. The first sentence has negative polarity, the second positive polarity.

`LangGer` can parse *er schläft nicht hier heute* (as `Eng`: *he doesn't sleep here today*), but not *er schläft heute nicht hier* or *er schläft heute nicht*.

er schläft nicht hier heute = er schläft woanders heute = er schläft heute woanders?

Gibt es doch `AdV` + *nicht* + `Adv`? *müssen später nicht erneut beantwortet werden* — * *erneut nicht später*, oder geht es hier einmal um ein Adverb zu *müssen*, das andere Mal um ein Adverb zu *beantworten*? *um (sie später*: ***`Adv`*** *nicht (erneut*: ***`Adv`*** *zu beantworten))*: *Inf-zu*.

How to set the sentence polarity for support verbs,. e.g. "*spielt dann nicht eine Rolle*" $\mapsto$ "*spielt dann keine Rolle*"?

Q21: Is it true that the averbs are ordered in a rather standard way (at least in the Mittelfeld? Zeit < Ort, etc.?), and if so, how do we fix the order of adverbs? Does German make a difference between `Adv` and `AdV`? Aarne: *er schläft heute*|**immer nicht*, but *er schläft nicht immer*.

(Isn't *er schläft (immer nícht) = immer (er schläft nicht) = er schläft nie*, and *ich verlasse dich nie = immer (ich verlasse dich nicht)*, so *(immer nícht) = nie*? In contrast, *(er schl´äft immer) nicht = nicht (immer (er schläft)) = manchmal (nicht (er schläft)) = manchmal (er schläft nicht) = er schläft manchmal nícht*, and there is no lexical adverb for *manchmal nícht*?)

According to Macheiner[6] (p.74): (in a declarative sentence with transitive verb,) the temporal adverbial can be put at the beginning of the sentence, at the end, or after the finite verb.

Am Anfang *schuf Gott den Himmel und die Erde.*
Gott schuf den Himmel und die Erde am Anfang.
Gott schuf am Anfang *den Himmel und die Erde.*

There may be several adverbials, which can be put at the beginning, at the end, or at the *middle* position, i.e. between subject and verbal phrase, but then hardly the temporal adverb at the end.

One distinguishes between ***referential adverbial***s, which contribute to the situation or event referred to, from ***evaluative adverbial***s, which express the speaker's evaluation of the event. (To the latter kind belong "*ja*", "*doch*", "*wohl*" etc.). Among adverbials of the same kind, the more specific one follows the more general one. ("*leider*" < "*wahrscheinlich*".) Among local adverbials, the more specific one comes before the less specific ones.

Relative order of temporal, local, causal, modal and instrumental adverbials: temporal < local ("*am Montag um 8 Uhr vor dem Dom*")

Sentence negation:

ich habe viele Bücher (nicht) gelesen	*ich lese viele Bücher (nicht)*
ich habe ein Buch nicht gelesen	*ich lese ein Buch nicht*
ich habe kein Buch nicht gelesen	*ich lese kein Buch nicht*

Prepositional phrases as special case of adverbs, via `Prep =< Adv/NP`.

There are split prepositions in German, i.e. *wenn man von hier aus den Blick über den Dnjepr hinweg nach Osten richtet* (Karl Schlögel, Entscheidung in Kiew, S.92); also *von dort her*. But are these worth a second field `s2:Str` in `Prep`, which lead to a lot of wrong trees (via `VPSlashPrep` only, c.f. Remark 18)?.

At which positions can we have an adverb (if objects are "positive")?

vz	Vorfeld	`vfin`	Mittelfeld				`vinf`
Präsens	Nom	glaubt	Dat	Acc	Adv	nicht	
Imperfekt	Nom	glaubte	Dat	Acc		nicht	
Perfekt	Nom	hat	Dat	Acc		nicht	geglaubt
Plusquamperfekt	Nom	hatte	Dat	Acc		nicht	geglaubt
Futur I	Nom	wird	Dat	Acc		nicht	glauben
Futur II		wird	Dat	Acc		nicht	geglaubt haben

Es ist niemals so simpel! Eher für atomare Aussage `s:S` (mit individuellen Objekten?) und temporalem Adverb `advT:Adv` wird am Satzende negiert, bei `v:V2`: Sind Einstellungsadverbien `AdvM` anders zu behandeln als temporale, lokale oder Wiederholungsadverbien? Z.B.

Satznegation `advT`	Adverbnegation ? `advM`
er liest das Buch heute (nicht),	*er liest das Buch (nicht) gerne*
heute liest er das Buch (nicht),	*gerne liest er das Buch (nicht)*
er liest heute das Buch (nicht),	*er liest* $^?$ *nicht gerne das Buch*

Aber: *er liest das Buch (nicht heute)*, oder *er geht nicht gerne ins Kino* und *er stellt nicht gerne eine Frage*. Darf im Skopus von *gerne* ein *negierter* Satz stehen? *er arbeitet (gerne nicht) = er unterläßt gerne das Arbeiten*. Welche Adverbien sind "mit der Negation vertauschbar", d.h.

$$\textit{nicht regelmäßig } \phi \stackrel{?}{=} \textit{regelmäßig } \neg\phi,$$

oder haben ein duales Adverb,

$$\textit{nicht immer } \phi = \textit{manchmal } \neg\phi \ ?$$

und wann hat man so etwas wie

$$\textit{selten } \phi \Leftarrow \textit{nicht oft } \phi \stackrel{?}{\Leftarrow} \textit{oft } \neg\phi \Leftarrow \textit{immer } \neg\phi = \textit{nie } \phi \ ?$$

At least, English does not put an adverb between verb and object, but German does: *I sometimes read novels – ich lese manchmal Romane.*

4.5. Clause

A ***basic clause*** is the combination of a verb with other expressions realizing all complement functions according to the syntactic arity of the verb. Thus, a basic verb phrase can be extended to a basic clause by adding a suitable subject expression.

Subject-verb agreement: nominative nominal subject and predicate (finite verb part `vfin`) agree in person and number: *ich schlafe*, *du schläfst*, *wir schlafen*, etc.; a non-nominative nominal or prepositional subject agrees with the predicate in third person singular: *mich|uns friert*, or *mir|uns ist schlecht*, or *dir|euch wird verziehen*, and *nach mir|uns wird gesucht*. Sentential, interrogative and infinitival subjects agree with the predicate in third person singular, e.g. *daß es schneit, freut uns* or *wo die Sterne stehen, ist uns bekannt* or *den Armen zu helfen ist eine Pflicht*.

Remark 25: Contractions of preposition and article cannot generally be excluded in (object) complement frames of verbs, nouns, and adjectives; they can even be used in subjects of passive clauses, e.g. *ans Christkind wird geglaubt*. (Hence we cannot avoid $9 \leq |\texttt{Prep}| \leq 15$.)

- verbal gender, (*Aktiv*, *Passiv*)
- *Vorfeld* + `vfin` + *Mittelfeld* + `vinf` + *Nachfeld*
- verb order in interrogative, main, and subordinate clauses
- extraction to the *Nachfeld*
- in-place ***correlates*** *es* and *das* and extraction of infinitives and *daß*-sentences
- relative order of adverbials in the *Mittelfeld*
- negation: VP-negation versus sentence-negation, (negation in NPs with *kein*)

Where do negation, modalization, passivization operate? On verbs/predicates: *es nicht tun*, *es tun können*, *getan werden (können)*; on adjective modifiers $un : A/A$ as in *unschön* or $bar : V\backslash A$ in *erziehbar* (also in `AGrec`), on participles `ungelesen`

There are (passive) (and passive interrogative) clauses without (even expletive) subject, like *hier wird nicht demonstriert* or *jetzt wird wieder gearbeitet* (and *muß dagegen nicht protestiert werden?*). (Apparently, there is no infinitive form of these passives. Or is *demonstriert werden* the subject of the question *demonstriert werden darf heute nicht*?)

As explained above, the predicate is split into a finite part `vfin` and an infinite part `vinf`.

Sentential objects are often right-***extracted*** (from the Mittelfeld), i.e. moved to the right behind the infinite verb part, e.g. *er hatte [es] geglaubt, daß die Erde flach sei* and *er wird [es] glauben, daß die Erde flach ist*. Like nominal objects, the sentential object can be moved to the Vorfeld, e.g. *daß die Erde flach ist, [das] wird er glauben*. Acceptable, but less used is the in-place position: *er hat, daß die Erde flach ist, nicht geglaubt*. In subordinate clauses: *(weil) daß die Erde flach ist, niemand glaubt*. The probably most used placement is in the Nachfeld. (So `LangGer` gives `VP` a field `ext:Str` to put the extracted constituent into.)

	Vorfeld	`vfin`	Mittelfeld	`vinf`	Nachfeld
	damals	hat/hatte	(es) fast jeder	geglaubt,	daß die Erde flach ist
vz	heute	glaubt	(es) fast niemand,		daß die Erde flach ist
	heute	wird	(es) kaum jemand	glauben,	daß die Erde flach ist

Word ordering Interrogative clause = verb-initial, declarative clause = verb-second, subordinate clause = verb-final

Q22: But what about the ordering of light and heavy noun phrases, the ordering of adverbs, the position of the negation adverb, the extraction depending on expression length?

...

5. The Resource Grammar `LangGer`

Let me emphasize that this is a documentation and explanation of source code developed by Aarne Ranta, Harald Hammarström and Björn Bringert (2002-2006), with some additions and changes by Erzsébet Galgóczy, Scharolta Siencnik and myself. (Scharolta Siencnik at least made noun phrases and adjective phrases discontinuous, added clauses with non-nominal subjects and added focus-rules in `ExtraGer`.) Hence I may at places have misunderstood the original intentions. The examples from German demonstrating various phenomena or constructions often cannot directly be tested with the implementation; the main reason is that the small test lexicon `LexiconGer` and lexicon of structural words `StructuralGer` miss entries of corresponding word classes (or words with properties in question) or that the available entries would give semantically inadequate examples.[38]

The full code is on `https://github.com/GrammaticalFramework/gf-rgl.git`.

As mentioned in Section 3.1, the abstract grammar `Lang` consists of modules `Grammar` and `Lexicon` and fixes a startcategory `Phr` of phrases. Accordingly, the concrete grammar `LangGer` consists of modules `GrammarGer` and `LexiconGer`. The module `GrammarGer` collects implementations for the submodules of `Grammar`.

```
concrete GrammarGer of Grammar =
  NounGer,
  VerbGer,
  AdjectiveGer,
  AdverbGer,
  NumeralGer,
  SentenceGer,
  QuestionGer,
  RelativeGer,
  ConjunctionGer,
  PhraseGer,
  TextX - [Tense,Temp,Adv,CAdv],
  IdiomGer,
  StructuralGer,
  TenseGer,
  NamesGer
  ** {

flags startcat = Phr ; unlexer = text ; lexer = text ;

} ;
```

These are all extensions of a module `CatGer` providing implementation categories for the syntactic categories of `Cat`, in particular the 1-dimensional implementation type for the start category:

```
  lincat
    Phr = {s : Str} ;
```

[38] Most of the phenomena can be demonstrated with lexicon entries, if the semantics is ignored. An extension of `Lang` by a `TestLexicon` with entries to test specific syntactic properties is under construction, as well as a file `lintest.gfs` for linearizing all constructions in `LangGer` and `LangEng`, to support regression tests.

Any concrete grammar in GF is based on a language-specific ***resource module*** which defines parameter types like gender, number, case, tense etc., as well as record types of morphological words like noun, adjective, verb, preposition, adverb, which contain inflection paradigms for these types of word that make use of the parameter types. A parameter type is defined by

```
param Typename =
  Constructor_1 ArgumentTypes_1 | ... | Constructor_k ArgumentTypes_k ;
```

where the constructors are function symbols f used to build terms ft of type `Typename` from terms t of f's argument types. In the simplest case of 0-ary constructors, these are just constants f of type `Typename`.

The type of a morphological word class is defined by

```
oper Wordclass : Type =
  {s : Form => Str ; p_1 : Type_1 ; ... ; p_m : Type_m} ;
```

where `s`, `p_1`,..., `p_m` are parameter names, including the name `s` for an inflection table, and `Form => Str` is a table type for `s`, `Type_1`, ..., `Type_m` are names of parameter types (or table types) for `p_1`, ..., `p_m`, and `Form` is the parameter type of (abstract) forms of words of the wordclass, `Str` the type of strings. The inflection table can have a more complicated type, for example `Form => Str * Str` for split words, or `Form_1 * Form_2 => Str` or `Form_1 => Form_2 => Str` if the abstract forms can be split into two parts; if there is no variation in forms, the type of `s` can be just the type `Str` or `Str * Str`.

Example 1. *For example, a type of adpositions might be defined by*

```
param Kind = Pre | Post | Circum ;
param AdCase = Genitive | Dative | Accusative ;
oper Adposition : Type = {s : Str * Str ; t : Kind ; c : AdCase} ;
```

An adposition would then be a record of this type, defined as an ***oper****ation by, for example,*

```
oper umherum : Adposition =
  {s = <"um", "herum"> ; t = Circum ; c = Accusative} ;
```

The grammar rule for combining an adposition with a noun phrase would then have to see from `umherum.t = Circum` *that the first part* `umherum.s.p1 = "um"` *has to be put in front of the noun phrase, the second part* `umherum.s.p2 = "herum"` *has to be put after the noun phrase, and from* `umherum.c = Accusative` *that the noun phrase has to be put in the accusative case.*

As we aim at explaining the implementation of the grammar rules for German, we will not explain how the inflection paradigms are built. (For this, see the resource file `ParadigmsGer`.) From the auxiliary operation of the resource file `ResGer`, we only give the parameter types and wordclass types as needed, and in doing so, often omit the keywords `param` and `oper`. Other operations of `ResGer` are often presented informally, by describing their effect on the linerization records of arguments in grammar rules.

The syntactic categories of the abstract grammar, given in `Cat`, are interpreted by ***linearization types*** or ***implementation types*** in `CatGer`. However, we shall not explain the implementation types of all syntactic categories in one sweep, but do this one by one.

The abstract grammar does not make a formal distinction between lexical and syntactic categories, but treats them alike as atomic categories of a system of simple types. Some of these are categories of word classes, like `N`, `V`, `A` etc. Their implementation type is then derived from the corresponding morphological word class, `Noun`, `Verb`, `Adjective` etc. by

```
lincat V = Verb ;
```

There is a slight distinction between the morphological type and the implementation type derived from it. Several lexical categories in the grammar, e.g. subjunctors `Subj` and conjunctors `Conj`, which consist of just uninflected words, have the same morphological type `oper M = {s : Str}`. The linearization categories

```
lincat Subj = M ; lincat Conj = M ;
```

derived from `M` have an additional invisible field and are *different* types

```
Subj = {s : Str ; lock_Subj : {}} ;   Conj = {s : Str ; lock_Conj : {}} ;
```

Notice that the name of the category is part of the additional lock-field, `lock_Subj` and `lock_Conj` respectively, so that the type checking phase of the grammar compiler can detect type incorrect uses of a subjunctor as a conjunctor, for example. Above, {} is the ***unit type***, and each subjunctor or conjunctor has the only possible value `<>` of this type in its lock-field.

5.1. Noun Phrases

Morphological noun. Nouns vary in number and case and have an inherent gender. The parameter domains are the standard two value number, four-value case and three-value gender:

```
param
  Number = Sg | Pl ;
  Case = Nom | Acc | Dat | Gen ;
  Gender = Masc | Fem | Neutr ;
```

The auxiliary type `Noun` has a field for the inflection paradigm and the inherent gender[39]:

```
Noun : Type = {
  s : Number => Case => Str ;
  g : Gender
  } ;
```

Lexical noun categories

This auxiliary type `Noun` (of `ResGer.gf`) is extended in `CatGer` to three lexical noun categories:[40]

```
lincat
  N  = Noun ;
  N2 = Noun ** {c2 : Preposition} ;
  N3 = Noun ** {c2,c3 : Preposition} ;
```

[39] We here skip two fields dealing with compound nouns

[40] For records `r` and `s`, `r ** s` is the extension of `r` by the fields of `s`, where fields with a label common to `r` and `s` get their value from `s`.

The categories `N2` of binary nouns and `N3` of ternary nouns have fields `c2` and `c3` indicating the preposition and case used to combine a nominal complement with the noun.

Prepositions The auxiliary type[41]

```
Preposition : Type =
  {s : PrepForm => Str ; s2 : Str ; c : Case ; t : PrepType} ;
```

has a field `s` for the preposition string (or "inflection" table), a field `s2` for the post-position string (or second part of a circum-position), a field `t` to distinguish between three types of prepositions, and a field `c` for the case of the nominal complement.

There are three types of "prepositions": (i) cases, (ii) pure pre-, post- and circum-positions, and (iii) prepositions contracted with the definite article in singular, e.g. *am* for *an dem* or *ins* for *in das*, or with some relative or interrogative pronouns, e.g. *worin* for *in was*, distinguished by the parameter type

```
param
  PrepType = isCase | isPrep | isContracting ;
```

To handle prepositions contracted with the definite article in singular, e.g. *zum* for *zu dem* and *zur* for *zu der*, the pre-position is not just of type `Str`, but a table of type `PrepForm => Str`, since the contracted form depends on the gender. This argument type

```
param
  PrepForm = CPl | CSg Gender | CAdvPron | CIPron ;
```

of the table type `PrepForm => Str` provides six values, `CPl`, `CSg Masc`, `CSg Fem`, `CSg Neutr`, `CAdvPron` and `CIPron`. An inflection table `s : PrepForm => Str` therefore provides six strings: the standard form of the preposition, three contractions of the preposition with the definite article in singular (in the case demanded by the preposition), the contraction of the preposition with the demonstrative *das* (used as pronominal adverb, e.g. *damit*,) and the contraction with the relative or interrogative pronoun *was*, e.g. *womit*. Thus, for prepositions, the table `s` does not contain a paradigm of word forms in the usual sense, but the preposition and some contractions with article or pronouns `das` and `was`. For example, according to the above types, the preposition *zu* can then be defined as operation

```
oper
  zu : Preposition =
    {s = table{CPl => "zu";
               CSg Masc => "zum" ; CSg Fem => "zur"; CSg Neutr => "zum" ;
               CAdvPron => "dazu" ; CIPron => "wozu"};
     s2 = "" ;
     c = Dat;
     isPrep = isContracting} ;
```

Remark 26: Some prepositions restrict the number of its complement noun phrase, e.g. *zwischen* with dative in *zwischen den Bäumen*, and *zwischen* with accusative in *zwischen die Seiten*. So, the implementation type might need a further field `n` with values `Nums = SgOrPl | PlOnly`.

[41] The implementation type of the syntactic category `Prep` is derived by `lincat Prep = Preposition ;`

Since the paradigms of pronouns, proper names, common nouns, and noun phrases not all have the same domains, we use two auxiliary functions to apply propositions. To apply a preposition `p` to a table `t : Case => Str`, the inflection table of a noun in a given number, an operation

```
appPrep0 : Preposition -> (Case => Str) -> Str = \p,t ->
   p.s ! CPl ++ t ! p.c ++ p.s2 ;
```

can be used. It concatenates the normal, uncontracted preposition string `p.s ! CPl` with the form for case `p.c` taken from the table `t`, and adds the postposition part `p.s2`. E.g., it combines a circumposition *um_herum* taking the accusative and a noun phrase *der Tisch* to *um den Tisch herum*.

To apply a preposition to a noun phrase, another operation

```
appPrepNP : Preposition -> NP -> Str
```

will be explained below. E.g., it combines the above preposition *zu* with the noun phrases *der Baum*, *die Tür*, *das Haus* by concatenating a suitable contraction of *zu* with the definite article of the noun phrase, followed by the noun phrase with article removed, giving *zum Baum*, *zur Tür*, *zum Haus*. For noun phrases not having a definite article in singular, it uses the form `zu.s ! CPl`, e.g. gives *zu den Bäumen*, *zu einem Baum* etc.[42]

To apply a preposition to an interrogative or relative pronoun, two similar operations

```
appPrepIP : Preposition -> IP -> Str ;
appPrepRP : Preposition -> RP -> RelGenNum => Str ;
```

are defined (in `ResGer`) and combined to an overloaded operation

```
appPrep = overload {
  appPrep : Preposition -> (Case => Str) -> Str = appPrep0 ;
  appPrep : Preposition -> NP -> Str = appPrepNP ; -- e.g. in dem CN => im CN
  appPrep : Preposition -> IP -> Str = appPrepIP ; -- e.g. in was    => worin
  appPrep : Preposition -> RP -> RelGenNum => Str = appPrepRP ;
  } ;
```

that can be applied to a preposition and an object of any of the four types of the second argument.

Perhaps not all the contracted prepositions can be used to combine a noun with its nominal complement, but there are some examples like *Weg ins Glück* or *Fahrt zur Hölle*. German has postpositions, e.g. *den Tag über*, *die Straße entlang*, *den Berechnungen zufolge*, *der Sage nach*, and circumpositions, e.g. *von mir aus* or *um den Tisch herum*; these are hardly used to combine a noun with its nominal complement, perhaps *die Fahrt um die Insel herum*.

Recall that `Lang` has no noun categories with non-nominal objects. Of course, German does have nouns with non-nominal objects, e.g. *Gründe, das Buch zu lesen*. (These are treated in `Lang` by the rules `EmbedVP : VP -> SC` and `SentCN : CN -> SC -> CN` that turn infinitives to sentential complements and combine these with *arbitrary* common nouns.)

[42] There are also: *ans*, *am*, *aufs*, *beim*, *hinters*, *hinterm*, *ins*, *im*, *übers*, *überm*, *unters*, *unterm*, *vom*, *vors*, *vorm*.

Categories `Pron`, `PN`, `CN` and `NP`

Pronouns have personal and possessive forms; the personal form varies in case, the possessive in gender (in third person singular) and case. This gives rise to the parameter domains

```
param
  NPForm = NPCase Case | NPPoss GenNum Case ;
  GenNum = GSg Gender | GPl ;
```

This four values of `GenNum` are suffcient, since in German word inflection, gender distinctions are only made in the singular.

Moreover, pronouns (and noun phrases generally) have an inherent agreement feature, with gender, number and person components:[43]

```
param
  Agr = Ag Gender Number Person ;
  Person = P1 | P2 | P3 ;
```

When the pronoun or noun phrase is used as subject of a sentence, its number and person determine the form of the main verb; its gender determines the form of the article and adjective of the noun phrase.

So the implementation category for pronouns is

```
Pron = {s : NPForm => Str ; a : Agr} ;
```

Proper names inflect for case (only) and have an inherent gender, so the category `PN` is

```
PN = {s : Case => Str; g : Gender} ;
```

The category of common nouns has an inherent gender parameter `g:Gender`, inherited from its head noun, an inflection paradigm `s:Adjf => Number => Case => Str` varying in adjective form, number and case, and three movable parts that can be separated from the head noun:

```
CN = {s : Adjf => Number => Case => Str ;
      rc : Number => Str ; -- Frage , [rc die ich gestellt habe]
      ext : Str ;          -- Frage , [sc wo sie schläft])
      adv : Str ;          -- Haus    [adv auf dem Hügel]
      g : Gender} ;
```

[43] This parameter type `Agr` has `|Agr| = |Gender|*|Number|*|Person| = 3*2*3 = 18` values. But distinctions in gender are only made in third person singular, so we can replace the parameter type by

```
Agr = AgSgP1 | AgSgP2 | AgSgP3 Gender | AgPl Person | AgPlPol ;
```

which only has 9 values, including a value for the polite personal pronoun *Sie*, which has special reflexive and possessive forms, e.g. *Sie sollten sich anstrengen und Ihr Bestes tun*. For the indefinite personal pronoun, in German we can use *man* with agreement value `AgSgP3 Masc`, e.g. *Man soll sich anstrengen und sein Bestes tun*, but in English, this pronoun *one* needs a special agreement value to select its reflexive possessive form *one's* from `reflPron`, e.g. in *One should try hard and do one's best*, not *his best*. As far as I see, noun phrase agreement in plural noun phrases can only use gender as in "*eine von ihnen*" vs. "*einer von ihnen*". But this is such a special case that we restricted `np.a` to `AgPl Person`.

Common nouns can be constructed from lexical nouns and can be modified by adjectival attributes, relative clauses, and adverbials. The adjectival attribute is attached to the noun from the left and part of the `s`-field; its form depends on properties of determiners, e.g. *(das) kleine Kind*, *(ein) kleines Kind*, and the common noun varies in number and case: *(dem) kleinen Kind*, *(viele) kleine Kinder*. A relative clause and a sentential complement of the noun can be separated ("moved away") from the noun by a verb, e.g. *sie hat die Frage beantwortet, die ich gestellt habe*, or *sie hat die Frage nicht beantwortet, wovon sie lebe* or *sie hat die Hoffnung aufgegeben, ein Star zu werden*; hence these are stored in separate fields `rc` and `ext`.

An adverbial modification of the noun, e.g. a local adverbial *auf dem Hügel*, is attached to the noun from the right. It can be separated from the noun by a possessive attribute, e.g. *(das) Haus von Johann auf dem Hügel*[44], and maybe therefore stored in a separate field `adv`. But since possessive attributes are also part of the common noun, this is not a good reason for having a separate `adv`-field in `CN`. It seems rather unusual to insert a participle between noun and modifying adverb, e.g. *(?) wir haben ein Haus gekauft [in Hamburg | aus Stein]*.
Todo 18: better omit the field `CN.adv:Str`.

An adjectival modification by an adjective with a comparison complement, attributively used, must be split and wrapped around the noun: in positive degree, e.g. *(ein) ebenso großer Berg wie die Zugspitze*, or in comparative degree, "*(ein) größerer Berg als die Zugspitze*". The comparative part can also be separated from the noun by a verb: "*wir haben einen größeren Berg bestiegen als die Zugspitze*".

The type `NP` of (basic) noun phrase consists of a field `s : Bool => Case => Str` for the inflection table, two fields `ext:Str` and `rc:Str` for movable parts, and inherent parameters `a:Agr` for agreement features and `w:Weight`. The `s`-field of `np:NP` combines two inflection tables: the ordinary inflection table `np.s ! False : Case => Str` of four strings for the four cases, and a special table `np.s ! True : Case => Str` where the leading definite article (of a noun phrase in singular) is dropped, to be used in `appPrepNP : Preposition -> NP -> Str`.

```
NP : Type = { -- HL 7/22: Bool = True if DefArt is dropped
   s : Bool => Case => Str ;
   rc : Str ;  -- die Frage , [rc die ich gestellt habe]
   ext : Str ; -- die Frage , [sc wo sie schläft]
   a : Agr ;
   w : Weight } ; -- light NPs come before negation in simple clauses
```

The determiner, the adjectival and the adverbial attributes cannot be moved[45] and are part of the `s`-field, e.g. *er hat die schwierige Frage von Johann, ob die Erde eine Kugel sei, beantwortet, die bisher offen geblieben war*. Apparently, the (attributive) possessive noun phrase (i.e. *von Johann*) cannot be separated from the head noun. The `rc`-field stores a relative clause, the `ext`-field stores the interrogative, infinitival or sentential object of the head noun, which can be separated from the noun: *sie hat die Frage beantwortet, wo er war*; *sie hat den Wunsch geäußert, eine Weltreise zu machen*; *wir haben die Hoffnung aufgegeben, dass der Regen aufhört*.

The ***weight*** of a noun phrase combines the properties of being a pronoun, being a light noun phrase and having a definite article to a 4-value parameter domain

[44] yielding an ambiguity: *(Haus von Johann):CN auf dem Hügel* versus *Haus von (Johann auf dem Hügel):CN*, using `MassNP : CN -> NP` and `PossNP`, if the possessive uses *von*.

[45] But: the comparision part of an adjectival attribute may have to be put to `np.ext`: *ich habe ein kleineres Haus [] gekauft als deines*.

```
param
  Weight = WPron | WLight | WHeavy | WDefArt ;
```

When used as nominal objects of a verb, the ***light noun phrases***, i.e. those with weight `WPron` or `WLight`, are placed before the sentence negation adverb *nicht* in clauses and infinitival verb phrases, the heavy ones after the negation adverb. (See `mkClause` on p. 140 and `infVP` on p. 123. Notice that *nicht* is not seen in infinitival complements, since `ComplVV`, `SlashVV` and `SlashV2V` only admit infinitival complements with positive polarity.)[46] The weight is also used to place pronouns in accusative before pronouns in dative. (See p. 120, p. 140.)

The weight `WDefArt` of an `np` indicates that the `np` has a leading definite article. This is used in the above mentioned auxiliary operation

```
appPrepNP : Preposition -> NP -> Str
```

that handles the contraction of definite article with prepositions; the code is explained below:

```
appPrepNP : Preposition -> NP -> Str = \prep,np ->
let
  g : Gender = genderAgr np.a ;
  n : Number = numberAgr np.a ;
  b = case <prep.t,n,np.w> of {
    <isContracting,Sg,WDefArt> => True ;
                    -- e.g. "zum Hof | zur Tür | zum Fenster herein"
    _ => False} ;   -- e.g. "auf dem Hof | auf der Tür | auf dem Fenster"
  f = case b of {True => CSg g ; _ => CPl} ;
in
prep.s ! f ++ np.s ! b ! prep.c ++ np.ext ++ prep.s2 ++ np.rc ;
```

If a preposition `p` has `p.t` = `isContracting`, say *zu* above, and an `np` has `np.w` = `WDefArt` and a singular agreement `np.a` in gender `g`, say *der große Berg*, then `appPrepNP p np` concatenates the contracted form `p.s ! (CSg g)` of the preposition, i.e. *zum*, with the noun phrase *without* its definite article `np.s ! True ! p.c` in the appropriate case `p.c`, i.e. *großen Berg*, yielding *zum großen Berg*. For prepositions `p` like *auf* with `p.t` = `isPrep`, `appPrepNP p np` concatenates the preposition string `p.s ! CPl`, i.e. *auf*, with the noun phrase *including* its definite article, `np.s ! False ! p.c`, i.e. *dem großen Berg*, yielding *auf dem großen Berg*.

Remark 27: The contraction of pre- or postpositions with possessive pronouns, e.g. *meiner + wegen* ⇒ *meinetwegen* (likewise *deinetwegen*,...,*deretwegen*), or with demonstrative pronouns, e.g. *seit + das* ⇒ *seitdem*, is not implemented. In `GF`, a demonstrative pronoun is a complex expression, e.g. *das* = `DetNP (DetQuant DefArt NumSg)`, hence such contractions seem to afford tree transformations.[47] The pronominal adverbs provided by the `CAdvPron` part of prepositions is not used yet, but probably useful as (i) correlate part of complex adverbs or (ii) prepositional objects, for which the preposition in `v.c2` could be used to connect a sentential object with the verb, e.g. *wir hatten damit gerechnet, dass es regnet* for a verb `v` = *rechnen mit etwas*. ◁

[46] Q23: Where is the negation put when the verb has a heavy and a light nominal object? *ich schenke dem Kind nicht einen Ball* with *nicht einen* = *keinen*? With emphasis, we can say *Ich schenke dem Kind (nicht einen)* | *keinen Ball, sondern einen Drachen*, but also: *Ich schenke dem Kind einen Ball nicht* = *Ich schenke dem Kind keinen Ball.* Q24: How does the weight depend on the quantifier in the object-np? *Ich lese diese*|*viele Bücher nicht* vs. **ich lese wenige Bücher nicht*, at least **ich lese nicht manche Bücher*

[47] So far, `LangGer > l PrepNP an_Prep (DetNP (DetQuant DefArt NumSg))` gives `am dem`, not `daran`.

Q25: Where is the adverbial attribute relative to the possessive and object-np? The default ordering of constituents seems to be

```
Det ++ AP ++ N2 ++ possessive ++ nominal ++ sentential ++ relative clause
```

This is implemented in the linearization default (in `CatGer`) for top-level usage of noun phrases:

```
linref
  NP = \np -> np.s!False!Nom ++ np.ext ++ np.rc ;
```

Q26: Which clause constructions allow us to extract a relative clause (or sentential complement) of an object noun phrase behind the infinite predicate part? (Currently, `mkClause` ignores this.)

Remark 28: The compilation complexities of the most expensive rules are now, (after reducing `Agr` to 9 values, with 15M VerbGer.gfo)

```
+ SlashV2VNP 99532800 (46080,240)
+ SlashVP 207360 (14160,136)
+ ComplSlash 207360 (150480,186)
```

5.1.1. Common Nouns

Construction of Common Nouns

Todo 19: remove field `CN.adv` and attach `adv.s` to `cn.s` in `AdvCN`.

A simple noun and a binary relational noun without an object can be used as common noun by the rules

```
UseN  : N -> CN ;           -- house
UseN2 : N2 -> CN ;          -- mother
```

The noun paradigm and gender are lifted to the paradigm and gender of the common noun, and an empty relative clause, extracted part and adverbial are added to the corresponding fields:

```
UseN, UseN2 = \n -> {
  s = \\_ => n.s ;
  g = n.g ;
  rc = \\_ => [] ;
  ext,adv = []
  } ;
```

Binary relational nouns `N2` can be lexical elements or obtained from a ternary (lexical) noun `N3` by ignoring one of its argument positions:

```
Use2N3  : N3 -> N2 ;          -- distance (from this city)
Use3N3  : N3 -> N2 ;          -- distance (to Paris)
```

This is done by dropping the field `n.c3` from a given `n:N3` and using the remaining fields `n.s`, `n.g`, and `n.c2`, or by using `n.c3` as field `c2:Preposition`:

```
Use2N3 n = n ;
Use3N3 n = n ** {
  c2 = n.c3;
  } ;
```

A common noun can also be obtained from a binary noun by adding a (nominal) complement:

```
ComplN2 : N2 -> NP -> CN ;    -- mother of the king
```

The complement is added to the paradigm field and cannot be moved from the head noun:

```
ComplN2 n2 np = {
  s = \\_,n,c => n2.s ! n ! c ++ appPrepNP n2.c2 np ;
  g = n2.g ;
  rc = \\_ => [] ;
  ext,adv = []
 } ;
```

Similarly, a binary noun can be obtained from a ternary noun by adding a nominal complement:

```
ComplN3 : N3 -> NP -> N2 ;    -- distance from this city (to Paris)
```

The complement is used as "direct object" `n3.c2` of a ternary noun `n3:N3`:

```
ComplN3 n3 np = {
  s = \\n,c => n3.s ! n ! c ++ appPrepNP n3.c2 np ;
  g = n3.g ;
  c2 = n3.c3 ;
  } ;
```

The idea seems to be that the `c2`-object stands closer to the head `n3` than the `n3`-object, i.e.

```
n3 ++ np1 ++ np2 = ComplN2 (ComplN3 np1) np3,
```

since there is no rule `ComplN3'` that inserts a complement `np` to the `c3`-field of an `n3`.

Modification of Common Nouns

Nouns can be modified by adjectives, relative clauses, and adverbs.

i) Modification by adjectives. In German, the modification by adjectives (with their complements and modifiers)

```
AdjCN   : AP -> CN  -> CN ;   -- big house
```

puts the adjective in front of the common noun it modifies, unless the adjective phrase has a sentential complement:

```
AdjCN ap cn =
  let
    g = cn.g
```

```
in cn ** {
  s = case ap.isPre of { -- HL 1/2023 False only for ap = SentAP ap' sc
    True => \\a,n,c =>   -- besserer cn als a.s2 [instead: cn, besser als a.s2,]
           (ap.c.p1 ++ ap.c.p2 ++ ap.s ! agrAdj a (gennum g n) c)
      ++   (cn.s ! a ! n ! c) ++ ap.s2 ! c ++ ap.ext ;
    False => \\a,n,c => cn.s ! a ! n ! c ++ -- postnominal ap with sc
      embedInCommas (ap.c.p1 ++ ap.c.p2 ++ ap.s ! APred ++ ap.s2 ! c ++ ap.ext)} ;
  g = g
  } ;
```

In the paradigm `ap.s`, only the forms for the `Strong` adjective inflection type are stored. The forms of the `Weak` and `Mixed` inflection types can be computed with the auxiliary operation `agrAdj` (c.f. Section 5.2) by properly selecting forms of the strong inflection, given gender, number and case.

ii) A common noun can be modified by a relative clause:

```
RelCN   : CN -> RS  -> CN ;   -- house that John bought
```

The implementation given for `RelCN cn rs` in `NounGer` extends the field `rc` of the argument `cn` by a suitable form of the given relative clause `rs`:

```
RelCN cn rs = cn ** {
   rc = \\n => (cn.rc ! n ++ embedInCommas (rs.s ! RGenNum (gennum cn.g n)))
} ;
```

This sounds not very well: instead of relativizing twice, e.g. *bekannter Maler, der in Florenz geboren wurde, der in Rom starb*, we would rather coordinate the relative clauses and then modify the noun once, i.e. *bekannter Maler, der in Florenz geboren wurde und in Rom starb,*.

iii) Modification by an adverbial:

```
AdvCN   : CN -> Adv -> CN ;   -- house on the hill
```

The implementation adds the adverb string at the end of a possibly already present adverb:

```
AdvCN cn a = cn ** {adv = cn.adv ++ a.s} ;
```

Remark 29: Iterated usage of this rule leads to ambiguities, in combination with `PrepNP : Prep -> NP -> Adv` and `DetCN : Det -> CN -> NP`, if these rules concatenate their constituents in the argument order. Then the trees

```
AdvCN cn1 (PrepNP p (DetCN det (AdvCN cn2 adv)))
AdvCN (AdvCN cn1 (PrepNP p (DetCN det cn2))) adv
```

linearize to the same string, `cn1 ++ p ++ det ++ cn2 ++ adv`, e.g. "*Tiere im Wald auf dem Berg*". (This is not the case if `p` is a postposition.)

Similarly, `AdvCN` and the rule `AdvVP : VP -> Adv -> VP` give rise to an ambiguity between attaching an adverb to the noun of an object of the verb or to the verb phrase, i.e. the `VP`-trees

```
ComplSlash (SlashV2a v2) (DetCN det (AdvCN adv cn))
AdvVP (ComplSlash (SlashV2a v2 (DetCN det cn))) adv
```

have the same linearizations, as sketched in

```
(v2 ++ (det ++ cn ++ adv))  versus  ((v2 ++ (det ++ cn)) ++ adv)
```

and "*see (the man (with the telescope))*" versus "*see (the man) (with the telescope)*". ◁

iv) Modification by embedded sentences, infinitives and questions:

According to the abstract syntax in `Noun.gf`, nouns can be modified by embedded sentences, questions and infinitives, but this modification "makes little sense" for "some" nouns. As discussed in the abstract syntax, Proposal 3.2.2, the intended modification rules are actually *complementation* rules for suitable subcategories `NS`, `NQ`, `NV` of the category `N` of nouns, i.e.

```
ComplNS : NS -> S -> CN ;
ComplNQ : NQ -> QS -> CN ;
ComplNV : NV -> VP -> CN ;
```

We view the "modification rule"

```
SentCN  : CN -> SC  -> CN ;   -- question where she sleeps
```

of `Lang` as a compensation for these missing noun categories and complementation rules.[48] This view is supported by the fact that the modifiers are collected in a category `SC` of "sentential complements" (defined in the module Verb 5.4).

The rule `SentCN : CN -> SC -> CN` adds a sentence, question or infinitive `sc.s:Str` of a complement `sc:SC` to the right of the `ext`-field of a `cn:CN`.

```
SentCN cn sc = cn ** {ext = cn.ext ++ embedInCommas sc.s} ;
```

In German, the sentential complement can be separated from the noun, e.g. *sie hat die Frage gestellt, wer das getan hat*, so it is stored the field `ext` for extractions. The complement sentence, question or infinitive `sc.s` has to be separated from the noun by a comma.

Clearly, this rule is highly overgenerating, as it applies to any (even modified) noun and any kind of sentential complement; moreover, the rule can be appplied iteratively and so add several sentential "complements" to the same noun.

v) Modification by apposition

As remarked in the abstract syntax `Noun`, the apposition rule

```
ApposCN : CN -> NP -> CN ;    -- city Paris (, numbers x and y)
```

is certainly overgenerating. For example, the number of the common noun and of the noun phrase should agree: *city Paris*, **cities Paris* and *numbers 3 and 4*, **numbers 3*. It is questionable if appositions can be attached to common nouns, or just to (coordinations of) proper names, e.g. *Johann, mein bester Freund,* or *Elisabeth die zweite* and *Karl der Große*. The implementation

[48] If one adds them, one should also add embedding rules `UseNS:NS -> CN`, `UseNQ:NQ -> CN`, `UseNV:NV -> CN`.

```
ApposCN  cn np = cn ** {
  s = \\a,n,c => cn.s ! a ! n ! c ++ np.s ! False ! c ++ np.ext ++ np.rel
} ;
```

attaches the noun phrase at the end of the common noun, in the same case, but does not ensure that common noun and noun phrase agree in number (and gender: **Elisabeth der zweite*). Moreover, sometimes the apposition should be embedded in commata, sometimes not. Although there are rules `AdvNP` and `ExtAdvNP` to attach an adverb to the end of a noun phrase, there is (so far) no rule to attach an adverb to the beginning, and hence appositions like *Johann, angeblich dein bester Freund,* cannot be parsed. So, this implementation seems somewhat too imprecise. Todo 20: at least, we should embed the attached `np` in commata[49].[50]

vi) Possessive construct:

A common noun can be modified by a possessive attribute using the rule

```
PossNP  : CN -> NP -> CN ;     -- house of Paris, house of mine
```

This is implemented in `gf-3.9` by attaching the possessive noun phrase using the preposition *von* with dative:

```
PossNP cn np = cn ** case np.w of {
  WDefArt => {s = \\a,n,c => cn.s ! a ! n ! c
                             ++ np.s ! False ! Gen ++ np.ext ++ np.rc} ;
    _     => {s = \\a,n,c => cn.s ! a ! n ! c
                   ++ appPrep P.von_Prep (np.s ! False) ++ np.ext ++ np.rc}
  } ;
```

The possessive noun phrase, including its sentential complement or relative clause, is attached closer to the common noun than an adverbial modifier, relative clause or sentential complement of the common noun, in the `s`-field. (Q28: is this usual in German?)

Remark 30: There are different ways to use a noun phrase `np` possessively in German. The attachment by *von* is perhaps not the most common one; it constructs a common noun and admits both pronouns, proper names and complex noun phrases as possessive attributes: *(der) alte Hund von ihm* or *(der) Hund von Johann* or *(der) Hund von einem fremden Mann.* Another, perhaps more frequent (post- or pre-nominal) attachment is by the genitive, i.e. *(der) alte Hund eines fremden Mannes* or *(der) alte Hund Johanns.* We only implement the postnominal genitive attribute when `np` has a definite article. A possessive prenominal genitive attribute, e.g. *eines fremden Mannes alter Hund*, replaces a definite article and constructs a noun phrase, not a common noun, c.f. `Extend.GenNP : NP -> Quant`; likewise for a possessively used proper name in genitive, e.g. *Johanns alter Hund.* A personal pronoun in genitive cannot be used as possessive attribute, as there are special possessive forms: instead of *der Hund seiner* one has to use *sein Hund.* (Q29: What about the possessive with indefinite NPs, like *einer seiner Hunde*, *einer von Johanns Hunden* or *einer der Hunde Johanns*?)

[49] But commata would disturb `glass_of_CN np = N.ApposCN (mkCN (P.mkN "Glas" "Gläser" neuter)) np` in `ConstructionsGer`. Shouldn't this be implemented by `PartNP`?. And also in *Königin Elisabeth die zweite*, `ApposCN (ApposCN (UseN queen_N) (UsePN elisabeth_PN)) (DetNP (DetQuantOrd defArt ord))`. So perhaps a separate rule `ExtApposCN` is needed that puts the apposition in commata to `cn.ext`.

[50] Q27: can we implement appositions as projections from relative clauses: *Johann, der angeblich dein bester Freund ist* $\mapsto$ *Johann, angeblich dein bester Freund*, and thereby ensure the agreement in number?

Expl: (SZ 285, 2023) *eines der wichtigsten Werke der modernen deutschen Literatur* and *eines der wichtigsten Dokumente des zerrütteten Erbes dessen, was Buber einstmals "die deutsch-jüdische Symbiose" nannte*

Q30: A common noun can be extended to a noun phrase by a prenominal possessive in genitive. In this construction[51], say `PossDetCN : NP -> CN -> NP`, we could use `np.w = WPron` to check if `np:NP` is a pronoun and then switch from genitive to possessive pronoun forms. For post-nominal possessive `np` in genitive (in `PossNP`), we have to exclude personal pronouns `p:Pron`, but we cannot implement such a restriction in `GF` given the (total) rule `UsePron : Pron -> NP`. Pronouns in German cannot be used whereever complex noun phrases can, the distributions differ. So `UsePron` is somewhat dubious.

Remark 31: Concerning `AdvCN` and `PossNP`, German has means to avoid ambiguities and misreadings: both *die Häuser aus Holz unserer Vorfahren* as well as *die Häuser unserer Vorfahren aus Holz* have unintended readings (i.e. *Holz unserer Vorfahren* resp. *Vorfahren aus Holz*). Using compound nouns, the intended reading can be expressed: *die Holzhäuser unserer Vorfahren.* (Likewise by derived adjectives: *die hölzernen Häuser*, *eng. the wooden houses.*)

vii) Partitive construct:

```
PartNP  : CN -> NP -> CN ;     -- glass of wine
```

The implementation is the same as for the possessive construction:

```
PartNP cn np = case np.w of {
  WDefArt => cn ** {s = \\a,n,c => cn.s ! a ! n ! c ++
                                   np.s ! False ! Gen ++ np.ext ++ np.rc} ;
  _       => cn ** {s = \\a,n,c => cn.s ! a ! n ! c ++
                                   appPrep von_Prep (np.s ! False) ++ np.rc}
    } ;
```

We have not enough parameters in the argument `np` to see if it is built by `MassNP` and then drop the *von* in e.g. *Glas von Wein.*

Remark 32: This is a case where we would like to be able to do constructions by induction on the abstract tree of the arguments, not its linearization records.

Remark 33: The module `Structural` has constants `possess_Prep:Prep` and `part_Prep:Prep`, both implemented using the preposition `"von"`. These constants have to be kept for backward compatibility, although they are termed "obsolete" and subsumed under `PossNP` and `PartNP`. ◁

Remark 34. *The various modification rules of `CN` can be applied in different order when parsing the same string, which leads to (I think, spurious) ambiguities:*

```
Lang> p -cat=CN "new green house of John" | ? grep -v ApposCN
AdjCN (PositA new_A) (AdjCN (PositA green_A) (PartNP (UseN house_N) (UsePN john_PN)))
AdjCN (PositA new_A) (AdjCN (PositA green_A) (PossNP (UseN house_N) (UsePN john_PN)))
AdjCN (PositA new_A) (PartNP (AdjCN (PositA green_A) (UseN house_N)) (UsePN john_PN))
AdjCN (PositA new_A) (PossNP (AdjCN (PositA green_A) (UseN house_N)) (UsePN john_PN))
PartNP (AdjCN (PositA new_A) (AdjCN (PositA green_A) (UseN house_N))) (UsePN john_PN)
PossNP (AdjCN (PositA new_A) (AdjCN (PositA green_A) (UseN house_N))) (UsePN john_PN)
```

[51] is this `Extend.GenNP`?

Similar with modification by `MassNP`, `PossNP`, `PartNP`:

```
Lang> p -cat=CN "cap of wine of John" | ? grep -v ApposCN
PartNP (UseN cap_N) (MassNP (PartNP (UseN wine_N) (UsePN john_PN)))
PartNP (UseN cap_N) (MassNP (PossNP (UseN wine_N) (UsePN john_PN)))
PartNP (PartNP (UseN cap_N) (MassNP (UseN wine_N))) (UsePN john_PN)
PartNP (PossNP (UseN cap_N) (MassNP (UseN wine_N))) (UsePN john_PN)
PossNP (UseN cap_N) (MassNP (PartNP (UseN wine_N) (UsePN john_PN)))
PossNP (UseN cap_N) (MassNP (PossNP (UseN wine_N) (UsePN john_PN)))
PossNP (PartNP (UseN cap_N) (MassNP (UseN wine_N))) (UsePN john_PN)
PossNP (PossNP (UseN cap_N) (MassNP (UseN wine_N))) (UsePN john_PN)
```

Can't we reduce this by fixing possible orders of modifiers, for example, attach `PossNP` and `PartNP` always closer to the `N` than `AdjCN` and `AdvCN`? This can in principle be done by using operator precedences to limit the extraction of trees from parse forests. A substitute is by tree transformation and collaps, see p. 190. It could also be enforced using categories depending on modifiers, c.f. slides for the GF summerschool in Riga, 2017.

5.1.2. Determiners

Construction of Determiners

The linearization category `Det` of determiners `det:Det` in `Ger` contains two inflection paradigms, the "standard" paradigm `det.s` for its use in combination with a common noun, e.g. in *der erste schöne Tag*, and a "special" paradigm `det.sp` when used as a noun phrase in itself, e.g. in *der erste*. The form of the determiner varies in gender and case, where the gender depends on the common noun it is combined with[52] and the case on the complement (or adverbial) function of the noun phrase in a clause. In order to treat contractions of prepositions with definite articles, we use a boolean flag `b:Bool` to distinguish between the usual paradigms `det.s ! False`, `det.sp ! False` and shortened versions `det.s ! True`, `det.sp ! True` where the leading definite article is missing.

```
lincat
  Det = {s,sp : Bool => Gender => Case => Str ; -- True if DefArt is dropped
         n : Number ; a : Adjf ; isDef, hasDefArt : Bool} ;
```

Moreover, the implementation type contains inherent features, the number `det.n` and the adjective inflection type `det.a`, according to which the common noun is inflected, and two boolean flags `det.isDef` to distinguish definite from non-definite determiners and `hasDefArt` to distinguish those having a leading definite article (in the full paradigms `det.s ! False`, `det.sp ! False`) from those that do not.

Recall that by design, `Grammar` gives determiners an inherent number, which will be inherited to noun phrases, and the number of the subject of a clause determines the form of the main verb.

Determiners, except for atomic ones, are constructed from "quantifiers" `Quant` (including articles and possessive pronouns), cardinals `Card` and ordinals `Ord`, via an intermediate category `Num` of grammatical number, by

[52] In the stand-alone usage, the gender must be given by the construction, e.g. `DetNPMasc, DetNPFem` in `Extend`.

```
DetQuant : Quant -> Num -> Det ;
DetQuantOrd : Quant -> Num -> Ord -> Det ;

NumCard : Card -> Num ;
NumPl : Num ;
NumSg : Num ;
```

Therefore we first review the cardinals and ordinals. For the moment, keep in mind that the most complex determiners are examples like *meine zwei ersten* in *meine zwei ersten weißen Zähne*. (Q31: Is *einer meiner zwei ersten weißen Zähne*? a `Card` modified by a possessive noun phrase?)

Determiners are normally used to turn complemented and modified nouns to noun phrases (and thereby stop further noun modification possibilities, like adjectival attributes), e.g. *ich mag kein Bier*. But they can also be used stand-alone, as a noun phrase, e.g. *keines* in *ich mag keines*. As the example shows, the forms can differ in these two usages. Hence the implementation type for determiners will have two paradigms, one for nomal usage, `Det.s`, and one for stand-alone usage, `Det.sp`. Since the difference shows in the leading quantifier element, there will also be two paradigms `Quant.s` and `Quant.sp`.

Q32: What about the relative ordering between numerals and ordinals: *meine drei besten Arbeiten*, *drei meiner besten Arbeiten*, *die drei ersten Fälle*, *drei der ersten Fälle*, *die ersten drei Fälle* ? And *die drittbeste*, not *die dritte beste*.

5.1.3. Numbers and number words

Numbers, restricted to cardinal and ordinal numbers (*drei* and *dritte*, but no quotients like *drittel*, or repetion numbers *dreimal*, *dreifach* etc.), occur as ***numerals***, i.e. numbers in words of natural language, or as ***digits***, i.e. in mathematical notation, based on sequences of digits 0,...,9.

```
Numeral ; -- cardinal or ordinal in words       e.g. "five/fifth"
Digits ;  -- cardinal or ordinal in digits      e.g. "1,000/1,000th"
```

There are conversion functions[53] that convert digits to numerals and conversely,

```
digits2num : Digits -> Numeral
num2digits : Numeral -> Digits
```

so we can here limit on the construction of digits

Digits and decimals

When used as cardinals, digits don't inflect (perhaps except genitive and dative plural, *3er* and *3en*), but certainly the numeral (emphasized) *ein* inflects like an adjective in gender, number and case: in singular masculine, we have *(der) eine, (den) einen*, in plural *(die) einen (und die anderen)*; moreover, there is the stand-alone usage, nominative: *einer*, *eine*, *eines*. When used as ordinals, they inflect like adjectives. So the inflection tables use a parameter type

[53] for which computation rules of `Transfer` are needed, see Section 3.1.16

```
param
  CardOrd = NCard AForm | NOrd AForm ;
```

which is unfortunately big to cover the worst-case cardinal *ein*. The implementation categories for the syntactic categories `Digits` and `Decimal` are

```
lincat
  Digits = {s : CardOrd => Str ; n : Number ; isDig, tail1to19 : Bool} ;
  Decimal = {s : CardOrd => Str ; n : Number ; hasDot : Bool} ;
```

The inherent `n : Number` has the value `Pl` for all digits and decimals except 1, for which it has the value `Sg`. Hence, as determiners, digits and decimals govern the following common noun in number, e.g. *0 Kinder*, *1 Kind*, *2 Kinder*.

Digits

The category `Digits` is for the digital representation. The non-empty sequences of digits are constructed from the ten sequences of length 1,

```
D_0, D_1, D_2, D_3, D_4, D_5, D_6, D_7, D_8, D_9 : Dig,
```

where the implementation category of `Dig` is

```
Dig = {s : CardOrd => Str ; n : Number ; isZero : Bool} ;
```

The implementation record of `D_`i for $i \neq 1$ is obtained as in

```
  D_4 = mk3Dig "4" "4t" Pl ;
```

with the auxiliary operation

```
  mk3Dig : Str -> Str -> Number -> Dig = \c,o,n -> {
    s = table {NCard _ => c ;
               NOrd af => (regA o).s ! Posit ! af} ;
    n = n ;
    isZero = False
    } ;
```

except that for $i = 0$, it has `isZero = True`. The inflection paradigm `s : CardOrd => Str` contains the same string for all forms[54] `NCard af`, and strings with adjective endings (of the strong adjective inflection) for the forms `NOrd af`, to get *4ter Tag* for *fourth day* (and predicative form *4t*). For `D_1`, we put `n = Sg` and the cardinal forms are obtained by concatenating `"1"` with corresponding endings of the pronoun,

```
  NCard (AMod (GSg g) c) => "ein" + pronEnding ! GSg g ! c ;
```

to get the singular (accusative) forms *1en Planeten*, *1e Königin* (but *0 Planeten*, *0 Königinnen*).

Remark 35: The determiner 1 also governs the adjective inflection: *1 kleines Kind*.

[54] Todo 21: But don't we need some inflection for stand-alone usage: `(den) dreien`?

There is a subltety in constructing the ordinal forms: the ordinal endings of German numerals (in nominative singular feminine) for the numbers 0 to 19 are *-te*, while those for 20 to 100 are *-ste*, those for 101 to 119 again *-te*, etc. The same holds for digital representations, so we have *0te*,...,*19te*, *20ste*,...,*100ste*, *101te*,...,*119te*, *120ste*,...,*200ste*, *201te*,...,*219te*, etc. To produce these forms from the implementation records, the boolean flags `isDig` and `tail1to19` of `Digits` are used in the implementation of the two by constructors

```
IDig : Dig -> Digits ;
IIDig : Dig -> Digits -> Digits ;
```

The construction `IDig` is implemented by taking the inflection paradigm of a `d:Dig` and filling the fields `n`, `isDig` and `tail1to19` appropriately:

```
IDig d = {s = d.s ; n = d.n ; isDig = True ; tail1to19 = notB d.isZero} ;
```

The construction `IIDig` adds a the cardinal form of `d:Dig` in front of an ordinal form of a sequence `i:Digits`. The fields `d.isDig` and `i.tail1to19` are used to decide whether an *s* has to be inserted in front of the ordinal ending:

```
IIDig d i =
  let i = lin Digits i ; -- suppress warning "missing lock_Digits"
      isPld : Bool = case d.n of {Sg => False ; _ => True} ;
      b : Bool = case i.isDig of {True => isPld ; _ => notB i.tail1to19} ;
      i' : Digits = case b of {True => IDig (mkDig (i.s ! invNum ++ BIND ++ "s")) ;
                              _  => i }
  in {s = table {NCard af => d.s ! invNum ++ BIND ++ i.s ! NCard af ;
                 NOrd af => d.s ! invNum ++ BIND ++ i'.s ! NOrd af} ;
      n = Pl ;
      isDig = False ;
      tail1to19 = case i.isDig of {True => notB isPld ; False => i.tail1to19}
  } ;
```

If `i:Digits` has length 1, say $i = 4$, the ordinal forms *14te*, *24ste*,...,*94ste* are build; if `i:Digits` has length at least 2, the combination `DigII d i` has the same ordinal endings as `i` itself.

Remark 36: The implementation of `D_4 : Dig` via

```
D_4 = mk3Dig "4" "4t" Pl ;
```

is a simplified form of `mkDigit "4" "14" "40" "4te"`, where `mkDigit` is the construction of numerals below.

Decimals are constructed from sequences of digits by

```
NegDecimal : Digits -> Decimal ;
PosDecimal : Digits -> Decimal ;
```

depending whether one wants to add a leading `-` sign. The leading minus sign in *-20st* is glued with the digits *"20st"*, i.e. without inserting a space, by inserting the `BIND` token:

```
NegDecimal d = {
  s = \\o => "-" ++ BIND ++ d.s ! o ;
  n = Pl ;
  hasDot = False
} ;
```

A further construction

```
IFrac : Decimal -> Dig -> Decimal ;
```

allows us to put a dot `"."` before the final digit and then add further digits at the end by repeated applications:

```
IFrac d i = {
  s = \\o => d.s ! invNum ++
             if_then_Str d.hasDot BIND (BIND ++ "." ++ BIND) ++
             i.s ! o;
  n = Pl;
  hasDot = True
} ;
```

using an abbreviation

```
invNum : CardOrd = NCard (AMod (GSg Masc) Nom) ;
```

When several dots are needed, e.g. to represent dates like *24.12.2023*, separate constructions have to be implemented.

Numerals

The implementation category of numerals (i.e. number words) contains an inflection paradigm and a field for an inherent feature of (grammatical) number:

```
Numeral = {s : CardOrd => Str ; n : Number} ;
```

To construct numerals, for each digit d, special forms of $10 + d$ and $d * 10$ are simultaneously treated, e.g. *three, thirteen, thirty* or *drei, dreizehn, dreissig* (former: *dreißig*) for digit *3*.

Digit

For these, an auxiliary category `Digit` is used, with the implementation type

```
Digit = {s : DForm => CardOrd => Str} ;
```

where the parameter type `DForm` is

```
param
  DForm = DUnit  | DTeen  | DTen ;
```

The implementation records for the digits `n2:Digit`,...,`n9:Digit` are built for digit d from the numerals for d, $10+d$, $d*10$ and a specific ordinal form (i.e. feminine singular nominative) of d:

```
  lin
    n2 = mkDigit "zwei" "zwölf" "zwanzig" "zweite" ;
```

by means of auxiliary operations

```
  mkDigit : (x1,_,_,x4 : Str) -> Digit =
    \drei,dreizehn,dreissig,dritte ->
    {s = table {
           DUnit => cardOrd drei dritte ;
           DTeen => cardOrd dreizehn (dreizehn + "te") ;
           DTen  => cardOrd dreissig (dreissig + "ste")
           }
     } ;
  cardOrd : Str -> Str -> CardOrd => Str = \drei,dritte ->
    table {
      NCard a => _ => drei ; -- (regA drei).s ! Posit ! a ;
      NOrd a => (regA (init dritte)).s ! Posit ! a
      } ;
```

As mentioned above, the numbers (0 and) 1 to 19 have ordinal ending *-te*, the remaining ones have ordinal ending *-ste*, as it is implemented in `mkDigit`. Not yet: It can also be seen from `cardOrd` that all cardinal forms `NCard af` have the strong adjective endings (of table 1, p. 47), provided by adjective declension `regA` in positive degree. The ordinal forms are built from the specific ordinal form provided, shortened by the final *e*, also using `regA`.

The syntactic category `Digit` is used to build up the numerals. There are special forms for multiples of 1 to 9 by 10, 100, 1000, 1000000, i.e. *zehn*,...,*neunzig*, *(ein)hundert*, ..., *(ein)tausend*, ..., *eine Million*. The abstract module `Numeral` of `Grammar` has syntactic types `Sub10`, `Sub100`, `Sub1000`, `Sub1000000` and embeddings

```
  fun pot0 : Digit -> Sub10 ;
  fun pot1 : Digit -> Sub100 ;

  fun pot1plus : Digit -> Sub10 -> Sub100 ;
  fun pot1to19 : Digit -> Sub100 ;

  fun pot1as2 : Sub100 -> Sub1000 ;
  fun pot2as3 : Sub1000 -> Sub1000000 ;

  fun pot2 : Sub10 -> Sub1000 ;
  fun pot3 : Sub1000 -> Sub1000000 ;
  fun num : Sub1000000 -> Numeral ;
```

and conversion functions between `Digits` and `Sub1000000` to `Numerals`

```
  fun digits2num : Digits -> Numeral ;
  fun num : Sub1000000 -> Numeral ;
```

The type `Sub10` covers the numbers 1 to 9 = 10-1, the type `Sub100` the numbers 1 to 99 = 100 - 1, etc. We only describe `Sub10`.

Sub10

The implementation type for `Sub10` has a field for an inflection paradigm as in `Digit` and a field for an inherent number:

```
Sub10 = {s : DForm => CardOrd => Str ; n : Number} ;
```

The numerals in `Sub10` are obtained from the digits `n2`,...,`n9` by the construction

```
pot0 : Digit -> Sub10 ;
```

which is implemented by taking their inflection paradigm and giving them the number `Pl`:

```
pot0 d = {s = d.s ; n = Pl} ;
```

The fixed numeral for the number 1,

```
pot01 : Sub10 ;
```

is the only one where the cardinal endings vary in gender and case:[55]

```
pot01 = {
  s = \\f => table {
        NCard af => (regA "ein").s ! Posit ! af ;
        NOrd af => (regA "erst").s ! Posit ! af
        } ;
  n = Sg
  } ;
```

The linearization types for the other domains have a simpler inflection paradigm than `Sub10`:

```
Sub100, Sub1000, Sub1000000 = {s : CardOrd => Str ; n : Number} ;
```

We omit a description of the constructors for these domains and refer to the source code in `NumeralGer.gf` and `NumeralTransfer.gf`.

Cardinals

By cardinals, `Grammar` means both cardinal words, e.g. *dreiundzwanzig*, and cardinal numbers in digital notation, e.g. `23`. In German, they inflect in gender and case, at least in the exceptional case *ein* = `NumNumeral (num (pot2as3 (pot1as2 (pot0as1 pot01))))`: in nominative, we have *ein*, *eine*, *ein*, in accusative *einen*, *eine*, *eine*. Moreover, it inflects like an adjective e.g. *der eine (große Fehler)* vs. *mein einer (großer Fehler)*, which occurs often in singular *der eine – der andere* and plural *die einen – die anderen*. (Other cardinals sometimes also inflect, at least for genitive plural in strong adjective inflection, e.g. *zweier Fehler*.) The cardinals have an inherent number. So the implementation type is

```
Card = {s : AForm => Str ; n : Number} ;
```

[55] Q33: What is the reason for *not* having a digit `n1:Digit` defined by `n1 = mkDigit "ein" "elf" "zehn" "erste"` and then putting `pot01 = pot0 n1 ** {n = Sg}` ? Currently, `pot01` has `DUnit,DTeen,DTen = ein`.

and `card.s` stores the `Strong` adjective forms of `card:Card`. The inherent number governs the number of the common noun in determiner-noun constructions: e.g. *ein Hund*, *zwei Hunde*. (`GrammarGer` does not have *null Hunde*.)

The construction of cardinals by

```
NumNumeral : Numeral -> Card ;
NumDigits : Digits -> Card ;
NumDecimal : Decimal -> Card ;
```

are just extractions of parts of the paradigms of the underlying digits or numeral:

```
NumNumeral numeral = {s = \\af => numeral.s ! NCard af ; n = numeral.n} ;
NumDigits  digits  = {s = \\af =>  digits.s ! NCard af ; n = digits.n} ;
NumDecimal decimal = {s = \\af => decimal.s ! NCard af ; n = decimal.n} ;
```

Cardinals can be modified by the rule

```
AdNum : AdN -> Card -> Card ; -- more than 51
```

which just puts a numeral-modifying adverb in front of the cardinal form:

```
AdNum adn num = {s = \\af => adn.s ++ num.s ! af ; n = num.n } ;
```

The category `AdN` of numeral-modifying adverbs has the implementation type `{s : Str}` of a record with just a field `s` of type string. An `AdN` can be a lexical element, like `almost_AdN`, `at_least_AdN`, and `at_most_AdN`, or can be constructed by

```
AdnCAdv : CAdv -> AdN ;
```

from a comparision adverb `less_CAdv`, `as_CAdv` and `more_CAdv`. Thus, using `AdNum` we get cardinals like *fast|höchstens|wenigstens 5* and *weniger|mehr als 5* (resp. *fünf*), but also

```
Lang> l (AdNum (AdnCAdv as_CAdv) (NumDecimal (PosDecimal (IDig D_5))))
as as 5
genau 5
```

To obtain different linearizations of `as_CAdv` here and in `ComparAdvAdj`, we have modified the implementation type of `CAdv`. Todo 22: This needs to be done for English as well.

By convention, cardinals up to twelve are usually expressed by numerals, e.g. *die zehn Gebote* and *die zwölf Apostel*, not by digits.

Cardinals can be used predicatively, e.g. *we are nineteen*, as a special case of predicative noun phrases; in German: *die Apostel waren zwölf*, *an Teilnehmern gab es 23*.

Remark 37: There is a computation rule for cardinals in `NumeralTransfer.gf`.

Ordinals

In `Grammar`, the syntactic category `Ord` covers ordinal numbers as digitals, e.g. *19te, 20ste* or numerals, e.g. *neunzehnte, zwanzigste*, but also adjectives in superlative, e.g. *älteste*. Ordinals in German inflect like adjectives, so the implementation type is

```
Ord = {s : AForm => Str} ;
```

(As for adjectives, only the forms for the strong adjective inflection type are stored; the others can be obtained via `agrAdj`.) Ordinal numbers are constructed from digits and numerals by

```
OrdDigits  : Digits  -> Ord ;  -- 51st
OrdNumeral : Numeral -> Ord ;  -- fifty-first
```

Other `Ord`s can be constructed from adjectives (c.f. Section 5.2) by

```
OrdSuperl  : A       -> Ord ;  -- warmest
```

In German, the predicative forms of the examples shown are *am 51sten*, *am einundfünfzigsten*, and *am wärmsten*. The linearizations project the ordinal part of the paradigms of digits and numerals, slightly modified to obtain the predicative form[56]

```
OrdDigits numeral =
  {s = table{APred => "am" ++ numeral.s ! NOrd APred ++ BIND ++ "en" ;
             af => numeral.s ! NOrd af}} ;
OrdNumeral numeral =
  {s = table{APred => "am" ++ numeral.s ! NOrd APred ++ BIND ++ "en" ;
             af => numeral.s ! NOrd af}} ;
```

Likewise, the linearization of `OrdSuperl` projects the superlative forms of the underlying adjective, modifying the predicative form *wärmsten* of the adjective *warm* to *am wärmsten*:

```
OrdSuperl a =
  {s = table {APred => "am" ++ a.s ! Superl ! APred ;
              af => a.s ! Superl ! af}} ;
```

If *first*, *second* etc. are viewed as definite ordinal numbers, some adjectives like *next*, *last* can be viewed as indefinite ordinal numbers. Here, any adjective in superlative form is an `Ord`.

Finally, there is a special construction of ordinals

```
OrdNumeralSuperl : Numeral -> A -> Ord ;
```

which glues an ordinal number with an adjective in superlative, e.g. *zweitälteste* or *drittbeste*:

```
OrdNumeralSuperl n a = {
  s = table {APred => "am" ++ n.s ! NOrd APred ++ BIND
                       ++ a.s ! Superl ! APred ;
             af => n.s ! NOrd APred ++ Predef.BIND ++ a.s ! Superl ! af}
  } ;
```

Remark 38: There is no rule `OrdDigitsSuperl : Digits -> A -> Ord` to obtain *am 51stbesten*.

Remark 39: Due to the token glueing in `OrdNumeral` and `OrdNumeralSuperl`, to parse *am 10ten*, the parser input must be `am 1 &+ Ot &+ en`. To ease the parsing of adjectives, `OrdSuperl`

[56] the predicative form of the underlying digits and numeral are *51st* and *einundfünfzigst*, respectively.

does not glue the ending *-en* to the predicative of the adjective, but the predicative form of the adjective is *besten*, not *best* or *am besten* (see Section 5.2). Hence `OrdNumeralSuperl` can produce *am drittbesten*. To parse *am drittbesten*, the parse input must be `am dritt &+ besten`. (But the predicative form *besten* does not allow us to construct *bestmöglich*.)

Remark 40: The non-glued combination *das dritte beste Lied* is subsumed by `DetCN det cn` and `AdjCN ap cn` via `DetQuantOrd : Quant -> Num -> Ord -> Det`.

Ordinals `ord:Ord` can be used attributively, as determiner and as noun phrase via

```
AdjOrd ord : AP
DetQuantOrd IndefArt NumPl ord : Det
DetNP (DetQuantOrd IndefArt NumPl ord) : NP.
```

The noun phrase of the last construction can be used predicatively, if the `ord` is an adjective in superlative, e.g. *sie sind am besten* or *sie sind am grünsten*, and as nominal complement, e.g. *die ältesten schlafen*.

Q34: What about *die 7te Beethoven'sche Symphonie*, but also with cardinal: *die 7 Beethoven'schen Symphonien*? And what about predicative usage: *das ist gut* $\mapsto$ *das ist am besten*, but also: *es war 7* (Uhr) $\mapsto$ *es war am 7ten* (Tag des Monats).

Quantifiers and Numbers

The category `Quant` of `Grammar` is not identical with the category of quantifier words. For example, some quantifiers have an inherent number, e.g. *jeder* (eng. *each*) is inherently singular, while *alle* (eng. *all*) is inherently plural. Others can combine with both singular or plural nouns, e.g. *kein* (eng. *no*). One could try to combine these to quantifier expressions varying in number, so that *each* and *all* are the singular and plural forms of the same quantifier (i.e. $\forall$ in logic), but there are quantifiers like *many* that have no singular form. In `Grammar`, the determiners have an inherent number, which is provided by a separate `Num` constituent of determiners; it may be a cardinal like *1* (with number singular) or *0,2,...* (with number plural), or a token with singular or plural number, but empty string field.

The `Quant` is the kernal of a determiner, and combines with a `Num` (cardinal or token fixing a grammatical number) and possibly an additional ordinal to a determiner:

```
DetQuant    : Quant -> Num ->         Det ;  -- these five
DetQuantOrd : Quant -> Num -> Ord -> Det ;  -- these five best
```

The implementation type of the syntactic category `Num` is an extension of the implementation type of `Card` by a boolean field `isNum`. Thus, for `GrammarGer`, the implementation type is

```
Num = {s,sp : AForm => Str ; n : Number ; isNum : Bool} ;
```

The second paradigm `sp : AForm => Str` is for stand-alone usage; e.g. *wir sprachen mit zweien, kannten aber nur einen von ihnen*. This is needed in `DetNP det`, but not implemented yet for plural. (To simplify the ordinary paradigm from `Card.s : AForm => Str` to `Card.s : Str`, maybe we can use `(NumCard m).sp` for the inflection needed for cardinal 1 = *ein(s)*. But this would use the `BIND` to glue the endings to `"ein"`, a disadvantage for parsing.)

The construction

```
NumCard : Card -> Num ;
```

takes the inflection paradigm and inherent number value of the cardinal, adds an inflection paradigm for stand-alone usage and sets the `isNum` field:

```
NumCard n = n ** {
  sp = table {AMod gn c => n.s ! APred ++ BIND ++ adjEnding ! gn ! c ;
              APred => n.s ! APred} ;
  isNum = True
} ;
```

The `sp`-paradigm can be used to generate[57] the plural *2e*, *2e*, *2en*, *2er* or *zweie*, *zweie*, *zweien*, *zweier*, but stem and endings need to be given as separate tokens in parsing.

Todo 23: The `sp`-paradigm is not used yet, neither nominally, e.g. *wir erinnerten uns zweier*, nor attributively, e.g. *wir erinnerten uns zweier Fehler*. Currently, we only get

```
Lang> l PredVP (UsePron we_Pron) (ComplSlash (SlashV2a listen_V2)
                 (DetNP (DetQuant DefArt (NumCard (NumDigits (IDig D_2))))))
wir hören den 2 zu
```

Should *beide* (and *viele*) count as cardinals, like *zwei*, so that we would get *die beiden Kinder*?

Adding endings in the `s`-paradigm of cardinals would disturb the generation of numerals like *hundertfünfundzwanzig*, so we only should add endings when applying `NumCard : Card -> Num`. Q35: But maybe also to the `s`-paradigm of `NumCard card` for small cardinals only,?

The two other constructions of `Num`, i.e. `NumSg, NumPl : Num`, are implemented by an inflection paradigm of empty strings and appropriate values for number and `isNum`:

```
NumPl = {s,sp = \\af => []; n = Pl ; isNum = False} ;
NumSg = {s,sp = \\af => []; n = Sg ; isNum = False} ;
```

The quantifiers of `Grammar` are articles, demonstative and possessive pronouns, but also the negated quantifier *kein* (eng. *no*). The quantifier forms vary in grammatical number, gender (in singular) and case (c.f. the table of definite and indefinite article on p.44).

The quantifiers, possibly followed by a cardinal and an ordinal, e.g. *this one*, *these two*, *the four youngest*[58], *these four youngest*, *a fourth*, *my four youngest*, *four*, are normally used as determiners, i.e. to turn a common noun into a noun phrase. But they can also be used separately, as a substitute for a noun phrase, e.g., *we liked the four youngest*, or *the four youngest were nice*. At least the quantifiers *ein*, *kein* and *mein* have other endings in combination with a common noun than when used separately, e.g. *ein|kein|mein Hund* vs. *einer|keiner|meiner*. Hence, two inflection paradigms, `s` and `sp`, are needed in the implementation type for `Quant`. The stand-alone forms under `sp` also depend on whether a cardinal or ordinal follows the quantifier (indicated by the flag `num.isNum : Bool` in `DetQuant` and `DetQuantOrd`): *wir geben es denen* vs. *wir geben es den dreien*, not **wir geben es denen drei(en)*, and *wir geben es denen* vs. *wir geben es den dritten*. The inflection paradigm `s` also varies, but only for the indefinite article: if a cardinal follows, the indefinite article is dropped, e.g. *a dog*, but **a one dog* simplifies to *one dog*, and **a two dogs* is ungrammatical. In German, the indefinite article and the cardinal *one*

[57] with the options in `l -table -bind NumCard (NumDigits (IDig D_2))` to glue the stem with the endings
[58] The grammar accepts *the three first*, but not *the first three*.

have even the same form, *ein*, and both **ein ein Hund* and **ein zwei Hunde* are ungrammatical. The implementation type of `Quant` therefore has two inflection paradigms:[59]

```
Quant = {
  s : Bool => GenNum => Case => Str ; -- True if leading DefArtSg is dropped
  sp : GenNum => Case => Str ;        --     and contracted with preposition
  a : Adjf ;
  isDefArt : Bool ;
  delCardOne : Bool -- delete following cardinal 1 (IndefArt and no_Quant)
  } ;
```

There are three further fields. First, quantifiers govern the following common noun in the adjective inflection, e.g. *ein junger Hund* vs. *der junge Hund*, hence we need a field `a:Adjf`. Second, the linearization type must have a boolean field telling whether a `quant:Quant` is the definite article. This flag helps us in the determiner constructions to build two forms of the determiner, one with and one without the leading definite article[60], e.g. *der zweite* and *zweite*. To contract prepositions with a leading definite article in noun phrases, we can then pick the shortened determiner form and get e.g. *im zweiten alten Haus* instead of *in dem zweiten alten Haus* (using `DetQuantOrd`), likewise *im einen* instead of *in dem einen (Fall)* (using `DetQuant` with cardinal *ein(s)* and number `Sg`).[61] Third, a flag `delCardOne` is added, with value `True` for the quantifiers `IndefArt` and `no_Quant` only, as these behave differently from other quantifierts when combined with the cardinal *ein*: *der eine*, *mein einer*, *jener eine*, but *ein ein(er)* = *ein einziger*, *kein ein(er)* = *kein einziger*. [Alternatively, we could implement *ein ein(er)* = *ein* and *kein ein(er)* = *kein*, which would lead to more ambiguities.]

The construction

```
DefArt : Quant ;
```

is implemented as follows. The forms of the definite article (c.f. the table on p.44) are collected in an auxiliary table (with cases orderd as `Nom`, `Acc`, `Dat` `Gen`):

```
artDef : GenNum => Case => Str = table {
  GSg Masc => caselist "der" "den" "dem" "des" ;
  GSg Fem  => caselist "die" "die" "der" "der" ;
  GSg Neutr => caselist "das" "das" "dem" "des" ;
  GPl      => caselist "die" "die" "den" "der"
  } ;
```

This table is then used in both the normal and stand-alone usage of the article. In the stand-alone usage, the genitive singular forms and the genitive and dative plural forms *den* and *der*

[59] Their types originally were `Bool => Number => Gender => Case => Str`, which gives tables of 2*2*3*4 = 48 strings, while the types `(Bool =>) GenNum => Case => Str` used here gives (2 *) 4 * 4 = (32) 16 strings: in `DetQuant` and `DetQuantOrd`, we can see from `quant.a` = `MixedStrong` that `quant`=`IndefArt`, and then drop the `quant` part if `num.isNum = True`. (However, we need empty strings in `DefArt.s!True!(GSg g)` to avoid a metavariable being raised when parsing a contracted preposition, e.g. *im Haus*.)

[60] Namely, for `DetQuant : Quant -> Num -> Det`, we know in `DetQuant quant num` from `quant.isDefArt`, `num.isNum` and `num.n`, whether the `quant.s` *is* a definite article in singular, so we can drop it in the determiner paradigm.

[61] However, in stand-alone usage `DetNP det`, we don't want to contract preposition and definite article alone, e.g. have *in dem*, not *im*. Can we do this in `PrepNP p np` without an additional flag in `np:NP` remembering whether `np` is constructed with `DetNP (DetQuant DefArt NumSg)`? This is a fairly rare case, so we ignore it.

have to be overwritten, e.g. to get *(wir erinnern uns) dessen*, *(wir erinnern uns) derer* and *(wir geben es) denen*:

```
DefArt = {
  s = \\b,gn,c => case <b,gn> of {
                   <True,GSg _> => [] ; _ => artDef ! gn ! c} ;
  sp = \\gn,c  => case <gn,c> of {
                   <GSg Masc,Gen> => "dessen" ;
                   <GSg Fem, Gen> => "derer" ;
                   <GSg Neutr,Gen> => "dessen" ;
                   <GPl,Dat> => "denen" ; -- HL 6/2019
                   <GPl,Gen> => "derer" ; -- HL 6/2019
                    _ => artDef ! gn ! c } ;
  a = Weak ;
  isDefArt = True ;
  delCardOne = False ;
  } ;
```

The auxiliary operations

```
numGenNum : GenNum -> Number = \gn ->
  case gn of {GSg _ => Sg ; GPl   => Pl} ;

genGenNum : GenNum -> Gender = \gn ->
  case gn of {GSg g => g ; GPl => Masc} ;
```

select the number and gender (in plural, a default) from a value `gn:GenNum`.

The indefinite article

```
IndefArt : Quant ;
```

is implemented similarly. In the normal usage, the endings of the indefinite article *ein* (in singular) and the negated indefinite article *kein* are the "pronominal" endings collected in

```
pronEnding : GenNum => Case => Str = table {
  GSg Masc  => caselist "" "en" "em" "es" ;
  GSg Fem   => caselist "e" "e" "er" "er" ;
  GSg Neutr => caselist ""  ""  "em" "es" ;
  GPl       => caselist "e" "e" "en" "er"
  } ;
```

In the stand-alone usage, the singular has different forms, e.g. masculine *einer,einen,einem,eines*, collected in

```
detEnding : GenNum => Case => Str = table {
  GSg Masc  => caselist "er" "en" "em" "es" ;
  GSg Fem   => caselist "e"  "e"  "er" "er" ;
  GSg Neutr => caselist "es" "es" "em" "es" ;
  GPl       => caselist "e"  "e"  "en" "er"
  } ;
```

The indefinite article has no plural forms, but in stand-alone usage, we add forms of *einige* (eng. *some*), to avoid problems with empty noun phrases.[62]

```
IndefArt = {
  s = table {GSg g => \\c => "ein" + pronEnding ! (GSg g) ! c ;
             GPl => \\c => []} ;
  sp = table {GSg g => \\c => "ein" + detEnding ! (GSg g) ! c ;
              GPl => caselist "einige" "einige" "einigen" "einiger"} ;
  a = MixedStrong ; -- Sg Mixed, Pl Strong
  isDefArt = False ;
  delCardOne = True ; -- ein+ein(er) => ein(er)
  } ;
```

The construction of quantifiers from personal pronouns, i.e.

```
PossPron : Pron -> Quant ;
```

gives possessive pronouns.[63] As part of determiners in their normal usage, e.g. in *(ich lese) dein neues Buch* and *(ich lese) dein erstes neues Buch*, they have the same forms. But as part of determiners in their stand-alone usage, they take different forms, e.g. *(ich lese) deines* and *(ich lese) dein erstes* (eng. *yours* and *your first one*).

```
PossPron p = {
  s  = \\gn,c => p.s ! NPPoss gn c ; -- mein (dritter)
  sp = \\gn,c => p.sp ! PossF gn c ; -- meiner
  a = Mixed ;
  isDefArt = False ;
  delCardOne = False ;
  } ;
```

Remark 41: The `PossF gn c` differ only slightly from the `NPPoss gn c`, and we could probably use an auxiliary operation `spPoss : GenNum -> Case -> NPForm` to get

```
p.sp ! PossF gn c = p.s ! spPoss gn c ;
```

similar to the computation of `Weak` and `Mixed` adjective paradigms from the strong one via `agrAdj : Adjf -> GenNum -> Case -> Str`. So, we can probably get rid of `p.sp`. ◁

The quantifier

```
no_Quant : Quant ;
```

is implemented as shown

[62] Q36: Are there still cases where *einige* popped up in unwanted contexts? Yes, e.g. `glass_of_CN (DetNPFem somePl_Det)` gives *Glas einige*, but singular *Glas eines* is also nonsense. How can we block using `(DetQuant IndefArt NumPl).sp ! False` in `(DetNP det).s = det.sp`?

[63] These can be seen as determinative usage of personal pronouns, in contrast to their nominal usage provided by `UsePron : Pron -> NP`.

```
no_Quant = {
  s = \\_ => table {GSg g => \\c => "kein" + pronEnding ! GSg g ! c ;
                    GPl   => \\c => "kein" + detEnding ! GPl ! c} ;
  sp = \\gn,c => "kein" + detEnding ! gn ! c ;
  a = Mixed ;
  isDefArt = False ;
  delCardOne = True} ; -- HL kein+ein(er) => kein(er)
```

Similarly for `that_Quant` and `this_Quant`, except that their two paradigms don't differ and they enforce `Weak` adjective inflection:

```
that_Quant = let jener : GenNum => Case => Str =
     \\gn,c => "jen" + detEnding ! gn ! c
  in {s = \\_ => jener ; sp = jener ; a = Weak ; isDefArt,delCardOne = False} ;

this_Quant = let dieser : GenNum => Case => Str =
     \\gn,c => "dies" + detEnding ! gn ! c
  in {s = \\_ => dieser ; sp = dieser ; a = Weak ; isDefArt, delCardOne = False} ;
```

Determiners

Expressions of syntactic categories like noun phrase, adjective phrase or verb phrase are extensions of a head element of a lexical category, i.e. a noun, adjective or verb, by complements or attributes. Determiners, i.e. expressions of category `Det`, are *not* extensions of words of a single lexical category, but rather expressions of different structure that can be combined with common nouns to a noun phrase. (The expressions that fulfil the *determinative syntactic function* are collected in the category `Det`.) This standard or ***determining usage of determiners*** is made precise in the construction `DetCN : Det -> CN -> NP`. When a common noun is understood from the context, there is also a ***stand-alone usage of determiners***, made precise in the construction `DetNP : Det -> NP`. Determiners have different paradigms for the two usages.

The paradigms `s` and `sp` for determiners vary in gender (in singular) and case.[64] In additon, determiners take a boolean argument to distinguish between common forms (under `s ! False`) and variants (under `s ! True`) in which the possibly leading definite article in singular is dropped. The implementation type of determiners is

```
Det = {
  s,sp : Bool => Gender => Case => Str ; -- True if DefArt is dropped, HL 8/22
  n : Number ; a : Adjf ; isDef, hasDefArt : Bool
  } ;
```

The inherent parameters `n:Number` for grammatical number and `a:Adjf` adjective inflection type govern the form of adjective attributes in the common noun that may be combined with the determiner. The boolean parameters `hasDefArt` is used to produce contractions of prepositions and leading definite article, e.g. to obtain *im alten Haus* from *in dem alten Haus*. The flag `isDef` is used for the relative order of various nominal objects in a verb phrase.

[64] The dependence on gender in plural is an artefact. Since determiners have an inherent number, the paradigms ought to have a type `Bool => G n => Case => Str` depending on the number `n`, where `G Sg` $\simeq$ `Gender` and `G Pl` has a single value, respectively. Equivalently, we might have two separate categories `DetSg` and `DetPl`, with paradigms of types `Bool => Gender => Case => Str` and `Bool => Case => Str`, respectively.

The module `Structural` of structural words of `Grammar` has six atomic determiners

```
every_Det, few_Det, many_Det, much_Det, somePl_Det, someSg_Det : Det ;
```

None of these have a leading definite article in singular, so the boolean argument to their paradigms is irrelevant. Moreover, the two paradigms often don't differ at all. Three of these impose plural and strong adjective inflection:

```
few_Det = {
  s,sp = \\_,g,c => "wenig" + adjEnding ! (gennum g Pl) ! c ;
  n = Pl ; a = Strong ; isDef = False ; hasDefArt = False} ;

many_Det = {
  s,sp = \\_,g,c => "viel" + adjEnding ! (gennum g Pl) ! c ;
  n = Pl ; a = Strong ; isDef = False ; hasDefArt = False} ;

somePl_Det = {
  s,sp = \\_,g,c => "einig" + adjEnding ! (gennum g Pl) ! c ;
  n = Pl ; a = Strong ; isDef = True ; hasDefArt = False} ;  ---- isDef?
```

The adjective endings added here are the endings of the strong adjective inflection, given by

```
adjEnding : GenNum => Case => Str = table {
  GSg Masc  => caselist "er" "en" "em" "en" ;
  GSg Fem   => caselist "e"  "e"  "er" "er" ;
  GSg Neutr => caselist "es" "es" "em" "en" ;
  GPl       => caselist "e"  "e"  "en" "er"
  } ;
```

Only that part of the paradigms is relevant that corresponds to the number value. So, for these three determiners the following adjective has the same (strong) endings as the determiner (c.f. Duden[2] 496, 485, 494), e.g.

```
Lang> l -table DetCN many_Det (AdjCN (PositA old_A) (UseN wine_N))
s False Nom : viele alte Weine
s False Acc : viele alte Weine
s False Dat : vielen alten Weinen
s False Gen : vieler alter Weine
s True Nom : viele alte Weine
s True Acc : viele alte Weine
s True Dat : vielen alten Weinen
s True Gen : vieler alter Weine
ext :
rc :
```

Remark 42: In general, *viel-*, *wenig-*, *einig-* etc. can be used in singular and plural, so these ***count adjectives*** should have type `Quant` in `LangGer`, and the above determiners be constructed from them via `DetQuant`, like `someSg_Det = DetQuant some_Quant NumSg`. ◁

Of the three singular determiners, `every_Det` has the same paradigms for determining and stand-alone usage and imposes weak adjective inflection:

Memo:

		Strong			Weak			Mixed		
Number	Case	Masc	Fem	Neuter	Masc	Fem	Neuter	Masc	Fem	Neuter
Singular	Nom	-er	-e	-es	-e	-e	-e	-er	-e	-es
	Acc	-en	-e	-es	-en	-e	-e	-en	-e	-es
	Dat	-em	-er	-em	-en	-en	-en	-en	-en	-en
	Gen	-en	-er	-en	-en	-en	-en	-en	-en	-en
Plural	Nom		-e			-en			-en	
	Acc		-e			-en			-en	
	Dat		-en			-en			-en	
	Gen		-er			-en			-en	
		without det			after definite article			after kein,mein		

Table 2: Ending tables of adjective inflection

		adjEnding			pronEnding			detEnding		
Number	Case	Masc	Fem	Neuter	Masc	Fem	Neuter	Masc	Fem	Neuter
Singular	Nom	-er	-e	-es	-	-e	-	-er	-e	-es
	Acc	-en	-e	-es	-en	-e	-	-en	-e	-es
	Dat	-em	-er	-em	-em	-er	-em	-em	-er	-em
	Gen	-en	-er	-en	-es	-er	-es	-es	-er	-es
Plural	Nom		-e			-e			-e	
	Acc		-e			-e			-e	
	Dat		-en			-en			-en	
	Gen		-er			-er			-er	

Table 3: Ending tables of determiner constructions

End Memo

```
every_Det = {
  s,sp = \\_,g,c => "jed" + detEnding ! (gennum g Sg) ! c ;
  n = Sg ; a = Weak ; isDef = False ; hasDefArt = False} ;
```

The two paradigms differ for `someSg_Det`, as seen e.g. in *er kaufte ein altes Auto* versus *er kaufte eines*, and this determiner imposes mixed adjective inflection:

```
someSg_Det = {
    s = \\_,g,c => "ein" + pronEnding ! GSg g ! c ;  -- ein, eine, ein
    sp = \\_,g,c => "ein" + detEnding ! GSg g ! c ;  -- einer, eine, eines
    n = Sg ; a = Mixed ; hasNum = True ; isDef = False ; hasDefArt = False
    } ;
```

Remark 43: In a sense, the determiners `someSg_Det` and `somePl_Det` are derived from the indefinite article `IndefArt:Quant`, which leads to multiple parse results, e.g.

```
Lang> p -lang=Ger -cat=NP "ein guter Wein"
DetCN someSg_Det (AdjCN (PositA good_A) (UseN wine_N))
DetCN (DetQuant IndefArt NumSg) (AdjCN (PositA good_A) (UseN wine_N))
```

But we have implemented a slight difference in plural:

```
Lang> p -lang=Ger -cat=NP "gute Weine"
DetCN (DetQuant IndefArt NumPl) (AdjCN (PositA good_A) (UseN wine_N))

Lang> l DetCN somePl_Det (AdjCN (PositA good_A) (UseN wine_N))
einige gute Weine
```

For the singular determiner `much_Det` and several other count adjectives as singular determiners, the adjective inflection has some exceptions (c.f. Duden[2] 482 - 496), so the four possibilities of `Adjf` are somewhat crude.[65] If we use *viel* without endings, which seems useful for mass nouns at least, strong adjective inflection is imposed.

```
much_Det = {
  s = \\_,g,c => "viel" ;
  sp = \\_,g,c => "viel" + detEnding ! (gennum g Sg) ! c ;
  n = Sg ; a = Strong ; isDef = False ; hasDefArt = False} ;
```

The use of `detEndings` in the stand-alone usage is a guess (may at least be wrong in genitive).

Remark 44: In German, we can add a definite article in front of `much_Det`, e.g. *das viele Geld*. Similarly for `many_Det`, e.g. *die vielen Kinder*. This is not accepted by `GrammarGer`. (It would be, if *viele* would count as a cardinal.)

The module `Noun` has two main ways to construct determiners,

```
DetQuantOrd : Quant -> Num -> Ord -> Det ;
DetQuant : Quant -> Num -> Det ;
```

To prepare the contraction of preposition and definite articles of noun phrases, for both `DetQuant quant num` and `DetQuantOrd quant num ord` we need variants of the paradigms of `quant`: if `quant` is the definite article and `num` is in singular, the variants have empty strings: The `quant` is the leading constituent in a determiner constructed by `DetQuantOrd` or `DetQuant`, and the determiner the leading constituent in a noun phrase constructed by `DetCN det cn`. A contractable preposition then just combines with the noun phrase shortened by the leading definite article, e.g. *[in] dem (einen) (besten) Fall* $\mapsto$ *[im] (einen) (besten) Fall*.

We first consider the construction `DetQuantOrd quant num ord`. In this case, there is no difference between the normal and the stand-alone paradigm, because of the final ordinal, e.g. *dieser dritte Versuch* and *dieser dritte*, or *deine drei besten Aufsätze* and *deine drei besten*. The `quant` determines the adjective inflection of the following ordinal and common noun, e.g. *der|dieser dritte große Versuch*, *ein|mein|kein dritter großer Versuch*. If `num` is a cardinal except *ein(s)*, it is not inflected, e.g. *zwei* in nominative *die zwei dritten Zähne* or dative *den zwei dritten Zähnen*.

Whether `num` is the cardinal *ein* can be detected from `num.n = Sg` and `num.isNum = True`. This `num` needs to be inflected, e.g. *der eine große Fehler*, *den einen großen Fehler*, *mein einer großer Fehler*, *meinen einen großen Fehler*. Moreover, if the preceding `quant` is `IndefArt` or `no_Quant`, they are contracted with the `num` *ein*, e.g. *ein + ein = ein (einziger)*, *kein + ein = kein (einziger)*. The flag `quant.delCardOne` indicates whether we should drop a following numeral *one* or, as implemented below, replace it by a form of *einziger*.

[65] If `DetCN det cn` could inspect the abstract construction of `det`, this could be accounted for using slight modifications as provided by `agrAdj`.

```
DetQuantOrd quant num ord =
  let
    n = num.n ;
    a = quant.a ;
    d = quant.isDefArt ;
    isCardOne = case n of {Sg => num.isNum ; _ => False} ;
    nums : AForm => Str = \\af => case af of {
      AMod (GSg g) c => case <quant.delCardOne,isCardOne> of {
        <True,True> =>  einziger ! af ; -- (ein,kein) einziger
        <_,True> => num.sp ! af ;       -- (der,dieser) eine ; (mein) einer
        _ => num.s ! af } ;             -- (die,diese) zwei ; (meine) zwei
      _ => num.s ! APred}
  in {
    s,sp = \\b,g,c => let gn = gennum g n in
      quant.s ! b ! gn ! c ++ nums ! agrAdj a gn c ++ ord.s ! agrAdj a gn c ;
    n = n ;
    a = a ;
    isDef = case a of {Strong => False ; _ => True} ;
    hasDefArt = d
    } ;
```

Remark 45: Determiners ending in an ordinal give rise to an ambiguity, e.g. for *my third teeth.*

```
Lang> p -cat=NP "meine dritten Zähne"
DetCN (DetQuant (PossPron i_Pron) NumPl)
      (AdjCN (AdjOrd (OrdNumeral (num (pot2as3 (pot1as2 (pot0as1 (pot0 n3)))))))
             (UseN tooth_N))
DetCN (DetQuantOrd (PossPron i_Pron) NumPl
           (OrdNumeral (num (pot2as3 (pot1as2 (pot0as1 (pot0 n3)))))))
       (UseN tooth_N)
```

Here, `OrdNumeral numeral : Ord` can either be turned into an `AP` by `AdjOrd` and then used to modify the common noun, or it can be the final part of a determiner which combines with the noun. Roughly, *the three (best new books)* = `(DetCN (DetQuant quant num) (AdjCN ord cn))` versus *(the three best) new books* = `(DetCN (DetQuantOrd quant num ord) cn)`. ◁

For the construction `DetQuant quant num`, the normal and the stand-alone paradigms differ: if no cardinal follows, i.e. `num.isNum = False`, the stand-alone form of `quant` is used, e.g. *ein|kein|mein großer Fehler* versus *einer|keiner|meiner*, but also if `quant` is the definite article: *wir gaben es den Kindern* versus *wir gaben es denen*, and in singular *wir erinnern uns des Problems|der Frage* versus *wir erinnern uns dessen|derer*(?).[66]

```
DetQuant quant num =
  let
    n = num.n ;
    a = quant.a ;
    d = quant.isDefArt ;
```

[66]Sometimes, `defArt.sp` is avoided, e.g. *mit dem* ↦ *damit*, *wegen des* ↦ *deswegen*, *wegen der* ↦ *deretwegen.*

Quant	Num	Adjf	DetCN det cn (DetQuant quant num).s		DetNP det (DetQuant quant num).sp	
der	NumSg	weak	der	q.s + n.s ! a	der	
	NumPl		die		die (Dat,Gen)	q.sp
	ein		der eine	q.s + n.sp ! a	der eine	
	zwei		die zwei		die zwei	q.s + n.s(p)
ein	NumSg	mixed	ein	q.s + ϵ	einer	q.sp
	NumPl	strong	ϵ	q.s + n.s ! a	einige(?)	q.sp
	ein	mixed	ein (einziger)	q.s + einziger	ein (einziger)	q.s + einziger
	zwei	strong	zwei	ϵ+ n.s ! a	zwei(e)	q.s + n.s(p)!a
kein	NumSg	mixed	kein	q.s + n.s	keiner	q.sp
	NumPl		keine		keine	
?	ein		kein (einziger)		kein einziger	q.s + einziger
?			/ nicht ein		/ nicht einer	/ nicht + einer
	zwei		keine zwei	q.s + n.s	keine zwei	q.s + n.s
mein	NumSg	mixed	mein	q.s + n.s	meiner	q.sp
	NumPl		meine	q.s + n.s	meine	q.sp
	ein		mein einer	q.s + einer!a	mein einer	q.s + einer
	zwei		meine zwei	q.s + n.s	meine zwei	q.s + n.s
dieser	NumSg	weak	dieser		dieser	
	NumPl		diese		diese	
	ein		dieser eine		dieser eine	
	zwei		diese zwei		diese zwei	

Table 4: The paradigms `s` of determinative and `sp` of stand-alone usage of determiners

```
  quantsp : Bool => GenNum => Case => Str =
    case num.isNum of {True => quant.s ; False => quant.sp} ;
  isCardOne = case n of {Sg => num.isNum ; _ => False} ;
  nums : AForm => Str = \\af => case af of {
    AMod (GSg g) c => case <quant.delCardOne,isCardOne> of {
      <True,True> =>  einziger ! af ; -- (ein,kein) einziger
      <_,True> => num.sp ! af ;       -- (der,dieser) eine ; (mein) einer
      _ => num.s ! af } ;
    AMod GPl c => num.s ! APred ;     -- (den,diesen) zwei(en) .sp?
    APred => num.s ! APred}
in {
  s = \\b,g,c => let gn = gennum g n in
    quant.s ! b ! gn ! c ++ nums ! agrAdj a gn c ;
  sp = \\b,g,c => let gn = gennum g n in
    quantsp ! b ! gn ! c ++ nums ! agrAdj a gn c ;
  n = n ;
  a = a ;
  isDef = case a of {Strong => False ; _ => True} ;
  hasDefArt = d
} ;
```

Todo 24: If the determiner ends in a cardinal, inflection may be needed. At least the (empty)

indefinite article in plural imposes strong adjective inflection for small cardinals: *wir erinnern uns der|meiner drei Fehler* versus *wir erinnern uns drei<u>er</u> Fehler*, or *mit den zwölf Aposteln* versus *mit den zwölf<u>en</u>*. This is not implemented yet.

Remark 46: In the paradigm for stand-alone usage, the possible omission of a leading definite article in singular needs an exception: if the article is *not* followed by a cardinal, the contraction of preposition and article should be avoided: *in dem einen* $\mapsto$ *im einen*, but *in dem* $\mapsto$ **im*. (Instead, we get *im dem*. Can we obtain *in dem* $\mapsto$ *darin*?)

Determiners can be coordinated, e.g. *these three or your two (ones)*. This is handled in the module `Conjunction` (Section 3.1.10, 5.5).

Remark 47: `Grammar` does not have noun phrases in genitive as determiners, e.g. *the philosopher Plato's books*, *Lady Windermere's fan*, *my best student's ideas*. This construction is declared in `Extend.GenNP : NP -> Quant` and generalizes `PossPron : Pron -> Quant`. (But it admits strange iterations: *eines Tages*, *eines Tages Abends*, *eines Tages Abends Endes*, ...)

5.1.4. Construction of Noun Phrases

Basic noun phrases can be built from a determiner and a common noun with the rule

```
DetCN   : Det -> CN -> NP ;   -- the man
```

The paradigm of `DetCN det cn` concatenates `det.s` with `cn.s` and varies in case. The form of the determiner depends on the gender `cn.g` of the common noun and the adjective inflection of the common noun depends on the determiner, as given by `det.a : Adjf`.

The relative clause and a sentential, interrogative or infinitival complement of a `cn:CN` are lifted to the `rc`-field and `ext`-field of the noun phrase `DetCN det cn`, respectively.

```
DetCN det cn = {
  s = \\b,c => det.s ! b ! cn.g ! c
               ++ cn.s ! (adjfCase det.a c) ! det.n ! c ++ cn.adv ;
  a = agrgP3 cn.g det.n ;
  w = case det.isDef of { True => case det.hasDefArt of {True => WDefArt ;
                                                         _ => WLight } ;
                          _ => WHeavy } ;
  rc = cn.rc ! det.n ;
  ext = cn.ext
  } ;
```

The agreement value of the noun phrase is `agrgP3 = (Ag cn.g det.n P3)` and determined by the gender of the common noun and the number read off from the determiner. If the determiner is definite, the noun phrase counts as light, else as heavy. The weight of the noun phrase influences its relative position with respect to negation in clauses, e.g. *ich sehe den Mann nicht* vs. *ich sehe nicht einen Mann* =? *ich sehe keinen Mann* (see `mkClause`, p.140).[67]

A variant of `DetCN` is to construct a noun phrase from a determiner alone, i.e. the construction

[67] Notice that `rc` and `ext` are lifted from the constituent `cn`. This suggests that the independent rule `RelNP`, p. 23, to relativize a full noun phrase by a relative clause is questionable, at least if proper names and pronouns can also be relativized (before embedding them into `NP`).

```
DetNP   : Det -> NP ;  -- these five
```

where the common noun of `DetCN det cn` is empty, e.g. *diese fünf* or *dieses fünfte*:

```
DetNP det = { -- more genders in ExtraGer
  s = \\b,c => det.sp ! b ! Neutr ! c ;
  a = agrP3 det.n ;
  -- HL 6/2019: no pronoun switch: ich gebe ihr das  vs. ich gebe es ihr
  w = case det.isDef of {
        True => case det.hasDefArt of { True => WDefArt ;
                                        _ => WLight } ;
        _ => WHeavy } ;
  rc, ext = []
  } ;
```

This construction (resp. its generalization `DAP`) is the only one where `det.sp` is used. The gender `Neutr` of the determiner (and in the agreement value `agrP3`) is a default; variants `DetNPMasc` and `DetNPFem` with gender `Masc` and `Fem` are defined in `ExtendGer`.[68]

Atomic noun phrases that are obtained by using a proper name via the rule

```
UsePN   : PN -> NP ;          -- John
```

are light noun phrases with third person singular agreement and by default no relative clause or sentential, interrogative or infinitival complement.

```
UsePN pn = {
  s = \\_,c => pn.s ! c ;
  a = agrgP3 pn.g Sg ;
  w = WLight ;
  rc, ext = []
  } ;
```

Similarly, when pronouns are used as noun phrases by

```
UsePron : Pron -> NP ;        -- he
```

the inflection type `Pron.s : NPForm => Str` has to be turned into `NP.s : Bool => Case => Str`.

```
UsePron pron = {
  s = \\_,c => pron.s ! NPCase c ;
  a = pron.a ;
  w = WPron ;
  rc, ext = []
  } ;
```

[68] For parsing, either use the rules `ExtendGer.DetNP*` or the more general rules `ExtendGer.DAP*`, but not both.

Since proper names and personal pronouns are light noun phrases, nominal objects of these kinds precede sentence negation in clauses, i.e. we get *sie liebt Johann|ihn nicht* rather than *sie liebt nicht Johann|ihn*.

Q37: What about *ich Arme(r)*, *du Dumme(r)*, or *wir Studenten*?

Remark 48: Pronouns have a different distribution than complex noun phrases: the possessive of a pronoun has special forms (the ***possessive pronoun***) which are used in front of the common noun, e.g. *meine kleinen Kinder*, while the possessive usage of non-pronoun noun phrases is by its genitive form, either in front of or following the common noun. (`PartNP` checks whether its argument `np` is a pronoun, and builds *Hand des Mannes*, but *Hand von mir*. Shouldn't `PossNP` similarly distinguish between *Hund des Mannes* and *mein Hund*, and `Extend.GenNP` subsume the pre-nominal possessive pronoun?)

The construction of noun phrases from mass nouns,

```
    MassNP     : CN -> NP ;            -- (I drink) beer
```

is massively overgenerating, since `Cat` provides no category of mass nouns. Any common noun in singular can be used as mass noun to construct a noun phrase without determiner:

```
    MassNP cn = {
      s = \\_,c => cn.s ! Strong ! Sg ! c ++ cn.adv ;
      a = agrgP3 cn.g Sg ;
      w = WLight ;  -- ich trinke Bier nicht vs. ich trinke kein Bier
      rc = cn.rc ! Sg ;
      ext = cn.ext
      } ;
```

Also, some quantifiers cannot be used with mass nouns, others can only be used with mass nouns: **viel Kind*, *viel Glück*, *viele Kinder*, **viele Glück*. (Case and gender in singular: *wir wünschen euch viel Freude*, *viel Erfolg*, *viel Glück*.)

5.1.5. Modification of Noun Phrases

```
-- A noun phrase already formed can be modified by a pre-determiner.

    PredetNP : Predet -> NP -> NP ; -- only the man
```

The type of pre-determiners is supposed to be

```
  lincat
    Predet = {
      s : Number => Gender => Case => Str ;
      c : {p : Str ; k : PredetCase} ;
      a : PredetAgr -- if an agr is forced, e.g. jeder von uns ist ...
      } ;
  param
    PredetCase = NoCase | PredCase Case ;
    PredetAgr = PAg Number | PAgNone ;
```

The field `a:PredetAgr` suggests that pre-determiners can be used to build subjects of a clause, which is the case with `PredetNP : Predet -> NP -> NP`, but not with `PredetRNP : Predet -> RNP -> RNP`. The field `c:{p:Str ; k:PredetCase}` is used when the pre-determiner governs the case of its argument (reflexive) noun phrase.

```
PredetNP pred np =
  let ag = case pred.a of {PAg n => agrP3 n ; _ => np.a} in np ** {
    s = \\b,c0 =>
      let c = case pred.c.k of {NoCase => c0 ; PredCase k => k} in
      pred.s ! numberAgr ag ! genderAgr np.a ! c0 ++ pred.c.p ++ np.s ! b ! c ;
    a = ag ;
    w = WHeavy
    } ;
```

Todo 25: Examples: Discuss. Distinguish according to `isPron np`?

Remark 49: In `Grammar`, *every* is a determiner, which governs the common noun in number, but *all* is a predeterminer, which combines with any (singular or plural) noun phrase. In particular, in can precede the definite article. Hence, **every dogs sleep*|*sleeps* is not accepted, but *all (the) dogs sleep* and(!) *all (the) dog sleeps* are. (The latter may be useful for mass nouns, e.g. *all the money*. In German, we have in singular *aller Mut*, *alle Freude*, *alles Geld*, but with article only *all der*|*mein Mut*, *all die*|*meine Freude*, *all das*|*mein Geld*.)

The abstract grammar declares four post-nominal modifications of noun phrases. The post-nominal modification of noun phrases by a past participle,

```
PPartNP : NP -> V2  -> NP ;    -- the man seen
```

should in German at least have the participle in commata, so it is implemented by

```
PPartNP np v2 = np ** {
  s = \\b,c => np.s ! b ! c ++ embedInCommas (v2.s ! VPastPart APred) ;
  w = WHeavy
  } ;
```

with an auxiliary operation

```
embedInCommas : Str -> Str= \s -> bindComma ++ s ++ endComma
```

This gives, e.g. *das Buch, ungelesen,*. However, more commonly, the participle is modified, e.g. *das Buch, kaum gelesen,* or *das Buch, von einigen hoch gelobt,*.

Remark 50: There are constructions

```
Extend.PastPartAP : VPSlash -> AP ;
Extend.PastPartAgentAP : VPSlash -> NP -> AP ;
```

that provide such examples, up to the post-nominal position:

```
AllGerAbs> l DetCN (DetQuant DefArt NumSg)
  (AdjCN (PastPartAP (AdVVPSlash always_AdV (SlashV2a read_V2))) (UseN book_N))
das immer gelesene Buch
```

If we let `PastPartAP` set `isPre = False`, then `AdjCN` would give an uninflected post-nominal modification, *das Buch, immer gelesen.* ◁

The two post-nominal modifications by an adverb,

```
AdvNP   : NP -> Adv -> NP ;    -- Paris today
ExtAdvNP: NP -> Adv -> NP ;    -- boys, such as ..
```

are similarly implemented by attaching the adverb to `np.s`:

```
AdvNP np adv = np ** {
  s = \\b,c => np.s ! b ! c ++ adv.s ;
  w = WHeavy
  } ;

ExtAdvNP np adv = np ** {
  s = \\b,c => np.s ! b ! c ++ embedInCommas adv.s ;
  w = WHeavy
  } ;
```

Todo 26: In `ExtAdvNP`, it may be better to put the `adv.s` to `np.ext`, so that it can be separated from the noun, e.g. *sie hatten einige Philosophen studiert, darunter Platon und Aristoteles.*

Finally, the post-nominal modification of noun phrases by relative clauses,

```
RelNP   : NP -> RS  -> NP ;    -- Paris, which is here
```

adds the argument `rs` to the field `np.rc`:

```
RelNP np rs = np ** {
  rc = let gn = gennum (genderAgr np.a) (numberAgr np.a)
       in np.rc ++ embedInCommas (rs.s ! RGenNum gn) ;
  w = case isPron np of { True => WLight ; _ => np.w }
  } ;
```

Q38: Do we really want to relativize a noun phrase, or can we remove `RelNP` in favour of new rules `PronRel : Pron -> RS -> NP` and `RelPN : PN -> RS -> NP` in addition to the existing rule `RelCN : CN -> RS -> CN`? Is *die Stadt Paris, die ich kenne,* a good example, or is this the relativization of common noun with apposition, i.e.

```
DetCN det (RelCN (ApposCN (UseN city_N) (UsePN paris_PN)) rs  ?
```

Better may be e.g. *mindestens 15 Staaten, die zusammen 65 Prozent der EU-Bevölkerung repräsentieren,* (der Freitag, 4. Januar 2024, p.3). At least semantically, this is not the plural of a relativized common noun, `RelCN (UseN state_N) rs`. (Or e.g. *die Haltung des Gesundheitsministers, der wenig Verständnis für deren Forderungen hat.*) By `RelNP`, several relative clauses can be added, e.g. *die Stadt Paris, die ich kaum kenne, die viele Leute lieben,*. This seems strange; a single, coordinated relative sentence seems better.

Remark 51. *Relative sentences can be attached to common nouns and to noun phrases, which leads to ambiguities* `(DetCN det (RelCN cn rs))` *and* `(RelNP (DetCN det cn) rs)` *like "das Haus , das ich kenne ,". Can these ambiguities be reduced? The above post-nominal modifications can be applied in any order, which gives some flexibility for the price of ambiguities.*

```
TestLang> p -cat=NP "Paris heute , gesehen , , das schläft ,"
  PPartNP (AdvNP (RelNP (UsePN paris_PN) (UseRCl ... rs)) today_Adv) see_V2
  PPartNP (RelNP (AdvNP (UsePN paris_PN) today_Adv) (UseRCl ... rs)) see_V2
  RelNP (PPartNP (AdvNP (UsePN paris_PN) today_Adv) see_V2) (UseRCl ... rs)
```

Todo 27: Avoid the double comma, produced by `RelNP` and `PPartNP`.

Todo 28: The rule

```
CountNP : Det -> NP -> NP ;    -- three of them, some of the boys
```

is claimed to be different from the partitive in many languages.

```
CountNP det np = -- drei der Kinder | drei von den Kindern -- HL 7/22, ad-hoc TODO
                 -- det or numeral? np or rather (DefArt +) cn?  drei (einiger Kinder) ?
  let g = genderAgr np.a
  in {
    s = \\b,c => det.sp ! b ! g ! c ++ appPrep von_Prep np ;
    a = agrgP3 g det.n ;
    w = case det.isDef of { True => WLight ; _ => WHeavy } ;
    rc = np.rc ;
    ext = np.ext
  } ;
```

A problem here is that the number `det.n` cannot influence the number of the argument `np` (but it governs the number of `cn` in `DetCN det cn`). So this construction overgenerates.

```
Lang> l CountNP (DetQuant no_Quant NumSg) (UsePron we_Pron)
none of us
keiner von uns
Lang> l CountNP (DetQuant IndefArt NumPl) (UsePron we_Pron)
ones of us
einige von uns
```

There are two more rules in `Noun`, providing determiners with adjectives and coordinations of such, e.g. *ein kleines oder dein bestes*. The first extends any determiner to a determiner "with adjective phrase". Since the extended determiner can be used stand-alone, but also in combination with another adjective phrase, two paradigms are needed:

```
DetDAP det = {
  s = \\g,c => det.s ! False ! g ! c ;
  sp = \\g,c => det.sp ! False ! g ! c ;
  n = det.n ; a = det.a ; isDef = det.isDef ; hasDefArt = det.hasDefArt
  } ;
```

The second rule modifies such an extended determiner by adding another adjective phrase; the determiner governs the adjective inflection type, as in `AdjCN`, so that we get e.g. *ein kleiner*, but also *der kleine*:

```
AdjDAP dap ap = -- the large (one)
  {s,sp  = \\g,c => dap.s ! g ! c ++ ap.c.p1 ++ ap.c.p2
     ++ ap.s ! agrAdj dap.a (gennum g dap.n) c ++ ap.s2 ! c ++ ap.ext ;
   a = dap.a ; n = dap.n ; isDef = dap.isDef ; hasDefArt = dap.hasDefArt } ;
```

The relative order between sentential complement and comparision noun phrase of the adjective may be wrong (if both arise at the same time).

```
Lang> l AdjDAP (DetDAP every_Det) (ComparA old_A (UsePN john_PN))

every older than John
jeder ältere als Johann
```

Remark 52: Not every determiner may be used in this rule. E.g., we get hard to understand examples with determiner `much_Det` and comparative adjective (Q39: how to avoid?):

```
Lang> l (AdjDAP (DetDAP much_Det) (ComparA old_A (UsePN john_PN)))
much older than John
viel älterer als Johann
```

The sequence *much + older* sounds like an adjective modification, which it isn't here and would in German be *viel älter* instead of *viel älterer*. Together with `Extend.UseDAP : DAP -> NP`, these give strange apparent complements to copula verbs, e.g. *ich bin viel älterer als du.* ◁

Todo 29: Check whether the correct paradigm `s` or `sp` is used by `Extend.UseDAP : DAP -> NP`.

5.2. Adjective Phrases

Morphological Adjective

Adjectives can be used in postive, comparative and superlative ***degree***. The parameter type is

```
param
  Degree = Posit | Compar | Superl ;
```

In each degree, adjectives have forms for predicative and attributive usage. The predicatively used German adjective does not inflect (for number and person, as the predicatively used verb does), e.g. *jung* in *du bist jung* and *sie sind jung*, while the attributively used adjective inflects for number and case of the noun it modifies, e.g. *mein junger Hund* vs. *meine jungen Hunde* and *meinen jungen Hund* vs. *meine jungen Hunde*, in singular also for gender, e.g. *mein junger Hund* vs. *meine junge Katze*. Therefore, the parameter type for adjective forms is

```
param
  AForm = APred | AMod GenNum Case ;
```

Attributively used adjectives also vary according to an ***adjective inflection type***,

```
  Adjf = Strong | Weak | Mixed | MixedStrong ;
```

This inflection type of attributively used adjectives in noun phrases depends on the determiner of the noun phrase; the `Strong` form is used when there is no determiner, e.g. *kleines Kind*, the `Weak` form is used when there is a definite article, *das kleine Kind*. The `Mixed` adjective inflection combines the `Strong` form in the nominative and accusative singular with the `Weak` forms in all other inflection cases; it is used for the negation determiner `kein` and the possessive pronoun as determiner of the noun phrase. The `MixedStrong` form is used with the indefinite article, using the `Mixed` forms in singular and the `Strong` forms in plural.

The strong attributive forms of adjectives are `AMod GenNum Case`. Where the weak and mixed forms do not agree with the strong ones, they use the two endings *-e* and *-en*, which are also the endings in the strong inflection of `AMod (GSg Fem) Nom` and `AMod (GSg Masc) Acc`. Hence, the forms of the weak and mixed inflection types can be obtained by properly selecting from the strong paradigm. This is done by an auxiliary operation

```
  agrAdj : Adjf -> GenNum -> Case -> AForm = \g,a,n,c ->
    let
      gn = gennum g n ;
      e  = AMod (GSg Fem) Nom ;
      en = AMod (GSg Masc) Acc ;
    in
    case a of {
      Strong => AMod gn c ;
      Weak => case <gn,c> of {
        <GSg _,   Nom> => e ;
        <GSg Masc,Acc> => en ;
        <GSg _,   Acc> => e ;
        _              => en } ;
      Mixed => case <gn,c> of {
```

```
      <GSg g, Nom|Acc> => AMod gn c ;
      _ => en } ;
    MixedStrong => case <gn,c> of {
      <GSg _, Dat|Gen> => en ;
      _                => AMod gn c }
  } ;
```

Applying `agrAdj` whenever attributive adjective forms are needed, the type of the inflection paradigm of adjectives can be simplified from `Degree => Adjf => AForm => Str` by dropping the dependence on `Adjf`. The auxiliary type of the morphological adjective therefore is just

```
Adjective : Type = {s : Degree => AForm => Str} ;
```

and only contains the forms of the strong adjective inflection.

Lexical Adjective

There are two syntactic categories of adjectives, unary and binary adjectives:

```
lincat
  A = Adjective ;
  A2 = Adjective ** {c2 : Preposition} ;
```

Remark 53: In `Lexicon.gf`, there is an entry `easy_A2V : A2`, which seems to suggest that there is a category `A2V` of ternary adjectives with prepositional and infinitival objects, e.g. *leicht für jemanden, es zu tun*. But `easy_A2V` is typed as a binary adjective, maybe because the infinitival complement is the subject, not an object of the adjective. `ParadigmsGer` says:

```
-- Notice: categories $AS, A2S, AV, A2V$ are just $A$,
-- and the second argument is given as an adverb.
```

In fact, to add a sentential or infinitival *subject*, one first has to apply `PositA` or `UseComparA` to the adjective to obtain an adjective phrase, then apply `UseComp o CompAP` to obtain a verb phrase and finally add the subject by `PredSCVP o EmbedS` or `PredSCVP o EmbedVP`; for example, this gives *that John sleeps is probable* from `probable_AS` or *to hear music is fun* from `fun_AV`. To add a sentential or infinitival *object*, one turns it into a "*sentential complement*" `SC` by `EmbedS` or `EmbedVP` and embeds this by `SentAP` to the adjective phrase; for example, this gives *John is glad to hear music* from `glad_A:A`. The sentential object could, unplausibly, also be analysed as adverbial sentence and added to the adjective phrase (or to the derived verb phrase) by `AdvAP` (or `AdvVP`); for example, this gives a strange analysis of *John is glad that he can sleep*. ◁

Category of Adjective Phrase

Instead of letting adjective phrases vary in degree, `Grammar` has three syntactic constructions

```
PositA : A -> AP
UseComparA : A -> AP
AdjOrd o OrdSuperl : A -> AP
```

that turn a unary adjective into an adjective phrase in which the adjective is in positive, comparative and superlative degree, respectively. Therefore, the inflection paradigm of adjective phrases does not depend on `Degree`, but is just a table of type `AForm => Str`. The implementation type of adjective phrases is (an extension of the one of `gf-3.9` by the `s2`-field) is

```
AP = {s : AForm => Str ; -- (strong) adjective paradigm
      s2 : Case => Str ; -- comparison noun phrase
      isPre : Bool ; -- True unless post-nominal as attribute to CN
      c: Str * Str ; -- (ich bin) [c1 ihm] treu ; stolz [c2 auf dich]
      ext : Str} ;   -- (du bist) so klug (gewesen) [ext ihn zu lesen]
```

The field `s2 : Case => Str` is to hold a comparision noun phrase, e.g. *wie der Kirchturm* in *so hoch wie der Kirchturm* or *als der Kirchturm* in *höher als der Kirchturm*. The comparision noun phrase depends on case, e.g. *sie bauten ihre Häuser nicht höher als den Kirchturm*, and, in attributive usage, can be separated from the adjective by a noun, e.g. *größere Türme als die Kirchtürme*, or *(ein) so hoher Turm wie der Kirchturm*. Hence a separate field `s2` is needed.

The complement of a binary adjective is stored in a field `c : Str * Str`. Nominal objects are stored in the first component, prepositional objects in the second.[69] A special field `s` is needed since objects can be separated from the adjective, e.g. by the participle *gewesen* in *er war stolz gewesen auf sein Prüfungsergebnis*. A complement can of course also be fronted, e.g. *auf sein Prüfungsergebnis war er sehr stolz*. As the examples indicate, in predicatively used adjective phrases the nominal objects precede the noun, e.g. *sie blieb ihrem Ziel treu* rather than **sie blieb treu ihrem Ziel*. (But in adverbial usage, this might be the proper ordering.) With prepositional objects, both positions are common, e.g. *er war stolz auf sein Ergebnis* and *er war auf sein Ergebnis sehr stolz*.

The field `ext : Str` is intended for sentential complements, e.g. an infinitival complement in *froh [darüber], die Prüfung bestanden zu haben*, or a sentential complement in *froh [darüber], daß die Sonne schien*. As with nominal and prepositional complements, they can be separated from the adjective, e.g. by a participle, or fronted.

Q40: How are adjectives with a sentential complement used in comparisons, as in *froher, VP.inf als NP* or as in *froher als NP, VP.inf*? Generally, the comparision part need not be a noun phrase, e.g. *ein besserer Sänger als ich*, or *(Pelé war) ein ebenso guter Sambatänzer wie Fußballspieler*, but can for example be an adverb, e.g. *ein besseres Wetter als gestern*.

In contrast to other languages, German does not distinguish between adjectives that precede from those that follow the noun in their attributive usage. The attributively used adjective phrase precedes the noun (though a comparison noun phrase may follow), except when its adjective has a sentential complement. In this case, the (uninflected) attribute follows the noun, e.g. *die Kinder, froh, daß es schneit,* instead of **die frohen, daß es schneit, Kinder* (though we might say *die darüber, daß es schneit, frohen Kinder*). Hence, a flag `isPre : Bool` distinguishes adjective phrases which can be used as prenominal attribute from those which cannot (the ones constructed by `SentAP`).

Remark 54: There is a post-nominal adjectival apposition: *das Brot, hart und angeschimmelt, war ungenießbar*, or *das Buch, noch nicht gelesen, lag auf dem Tisch*. To turn an adjective phrase into such a post-nominal attribute or apposition, i.e. uninflected adjective in commata, we might add a rule `ApposCN : CN -> AP -> CN` to `Extend`, implemented roughly by `AdjCN (ap ** {isPre = False}) cn`, or a similar rule `ApposNP:NP -> AP -> NP` generalizing `PPartNP`.

Remark 55. *The complement type `AP.c : Str * Str` should be `AP.c : Agr => Str * Str` to allow for reflexive pronouns and reflexive possessives, as in "stolz auf sich" or "stolz auf seine*

[69] This can be seen from `Adjective.ComplA2`. The constructions `Verb.ComplVA` and `Verb.SlashV2A` use `insertAdj adj c ext vp` to insert the nominal complement `c.p1` to `vp.nn.p2`, but the adjective with the prepositional complement `adj.s ++ c.p2` to `vp.adj`, and `ext` to `vp.ext`.

(eigene) Leistung". Also, the sentential objects can depend on `Agr`*, e.g. "froh, sich entschuldigt zu haben". Hence, a better implementation type could be*

```
AP : Type = {
   s : AForm => Str ;         -- (strong) adjective paradigm
   isPre : Bool ;             -- True unless post-nominal as attribute to CN
   s2 : Agr => Case => Str ;  -- comparison part: klügere N als man selbst
   c  : Agr => Str * Str ;    -- nominal vs. prepositional complement
   ext: Agr => Str} ; -- sentential complement: froh, sein Ziel zu erreichen
```

See `ExtraGer.ReflA2RNP`*, p. 178.* ◁

5.2.1. Construction of Adjective Phrases

The construction

```
PositA  : A  -> AP ;        -- warm
```

to use a unary adjective in positive degree just selects the positive forms of the adjective paradigm and fills the other fields with empty comparison part and complements:

```
PositA  a = {
  s = a.s ! Posit ;
  s2 = \\_ => [] ;
  isPre = True ;
  c = <[],[]> ;
  ext = []
  } ;
```

There are two constructions to use an adjective in comparative degree. First, the adjective can be used without a comparison part, by

```
UseComparA : A  -> AP ;     -- warmer
```

E.g., *das Meer ist heute wärmer* or *wir lieben die wärmeren Tage*. Then the comparative forms of the adjective are stored in the paradigm of the adjective phrase:

```
UseComparA a = {
  s = \\af => a.s ! Compar ! af ;
  s2 = \\_ => [] ;
  isPre = True ;
  c = <[],[]> ;
  ext = []
  } ;
```

Second, the adjective can be used with a comparison part. In `Grammar`, the comparison part must be a noun phrase (hence, e.g. *today, the sea is warmer than yesterday* is not recognized[70]):

[70] Todo 30: Check if `Extend` would accept this. The original implementation of `ComparA` stores the comparison part of an adjectival attribute in the field `cn.s:Str`, too, as in `LangEng`.

```
ComparA : A  -> NP -> AP ;  -- warmer than I
```

In German, the comparison noun phrase is not concatenated with the adjective, since they can be separated by an intervening noun, e.g. *(ein) wärmeres Meer als die Nordsee*, so the comparison noun phrase is stored in a special field `s2` of the adjective phrase:

```
ComparA a np = {
  s = \\af => a.s ! Compar ! af ;
  s2 = \\c => conjThan ++ np.s ! False ! c ++ np.ext ++ np.rc ;
  isPre = True ;
  c = <[],[]> ;
  ext = []
  } ;
```

The comparison part has a leading conjunction `conjThan`, in German *als*, and it varies in case, so that the adjective phrase can be an attribute in nominal objects, e.g. *(ich kenne) ein wärmeres Meer als den Atlantischen Ozean.*

Binary adjectives can be used without complement, by

```
UseA2   : A2 -> AP ;        -- married
```

as in *eine verheiratete Frau* or *wir sind verheiratet*, which is implemented, like `PositA`, by

```
UseA2 a = {
  s = a.s ! Posit ;
  s2 = \\_ => [] ;
  isPre = True ;
  c = <[],[]> ;
  ext = []
  } ;
```

The only binary adjective in `Lexicon.gf` is `married_A2`. (Participles of verbs can be turned into adjective phrases by `Extend.PastPartAP:VPSlash -> AP`.) A better example is e.g. *proud (of sth)* in *a woman proud of her career* or *she is proud of her career.*

Remark 56: A problem with this rule is that if the adjective expects a prepositional object, the object can be read as an adverbial modification of the adjective phrase, e.g. *mit mir verheiratet*:

```
AdvAP (UseA2 married_A2) (PrepNP with_Prep (UsePron i_Pron))
```

Similarly, the adjective phrase can be turned into a `VP` by `UseComp o CompAP` and modified by `AdvVP` with the adjective's prepositional object understood as `Adv`. (And `AdvAP` or `AdvVP` can be applied iteratively, leading to acceptance of e.g. *die mit mir mit mir verheiratete Frau.*) ◁

Binary adjectives can also be used with a complement:

```
ComplA2 : A2 -> NP -> AP ;  -- married to her
```

Depending on whether the adjective expects a nominal or prepositional object, the complement string is added to the first or second component of the complement field `c`, in the case `a.c2` expected by the adjective:

```
ComplA2 a np =
  let
    obj = appPrepNP a.c2 np
  in {
    s = a.s ! Posit ;
    s2 = \\_ => [] ;
    isPre = True ;
    c = case a.c2.t of {isCase => <obj, []> ; _ => <[], obj>} ;
    ext = []
  } ;
```

Binary adjectives can also be used reflexively, by

```
ReflA2  : A2 -> AP ;        -- married to itself
```

Here, the third person singular (of any gender) of the reflexive pronoun is put into the complement field `s`:

```
ReflA2 a =
  let
    obj = appPrep a.c2 (reflPron ! agrP3 Sg) ;
  in {
    s = a.s ! Posit ;
    s2 = \\_ => [] ;
    isPre = True ;
    c = case a.c2.t of {isCase => <obj, []> ; _ => <[], obj>} ;
    ext = []
  } ;
```

E.g., with suitable adjectives, this gives *auf sich stolz* or *sich treu* and *seiner (selbst) bar*.

Remark 57: The restriction to reflexive pronoun in third person excludes examples like *wir waren stolz auf uns*. More generally, the complement could be a relflexive noun phrase `RNP`, see `Extend.ReflA2RNP` (and git branch `reflexiveNPs`).

To use an adjective in superlative degree, it first has to be turned into an ordinal by `OrdSuperl : A -> Ord`, which selects the superlative attributive forms of the adjective paradigm and adds a leading *am* to the predicative form (c.f. p. 81). An ordinal can be turned into an adjective phrase by

```
AdjOrd  : Ord -> AP ;       -- warmest
```

The implementation just selects the paradigm from the ordinal:

```
AdjOrd a = {
  s = a.s ;
  s2 = \\_ => [] ;
  isPre = True ;
  c = <[],[]> ;
  ext = []
  } ;
```

The rule `AdjOrd` does not allow us to use a binary adjective in superlative degree, e.g. *viele sind stolz auf sich, aber Johann ist am stolzesten auf dich.*

Remark 58: The transformation to an ordinal leads to ambiguities: either the adjective in superlative is the final part of the determiner or the initial part of the common noun:

```
Lang> parse -cat=NP "der wärmste Tag"
DetCN (DetQuant DefArt NumSg) (AdjCN (AdjOrd (OrdSuperl warm_A)) (UseN day_N))
DetCN (DetQuantOrd DefArt NumSg (OrdSuperl warm_A)) (UseN day_N)
```

It seems better not to transform the adjective to an ordinal, but instead have rules `UseSuperlA = AdjOrd o OrdSuperl : A -> AP` and `UseSuperlA2 : A2 -> AP`. ◁

5.2.2. Modification of Adjective Phrases

Todo 31: Check where the modification rules have to make the result depend on `ap.isPre` of its argument `ap`, and whether the ordering of complements and modifying adverb is correct.

An adjective phrase can be modified by a comparison adverb and a (nominal) comparison part:

```
CAdvAP  : CAdv -> AP -> NP -> AP ; -- as cool as John
```

Remark 59: As a modification rule, `CAdvAP` can be iterated, which seems artificial, e.g. *the water is as [as warm as the sea] as the air*. Similarly, if its argument `ap` already expresses a value like *quite warm* on the implicit (temperature) scale provided by *warm*, the resulting `CAdv adv ap np`, e.g. *as quite warm as the sea*, expresses incomparable degrees: either the temperature is *quite warm* or *as warm as the sea*, but not both. To overcome this, one could restricted the rule to a construction rule `CAdvAP : CAdv -> A -> NP -> AP`, analogous to `ComparA : A -> NP -> AP`. (Probably, the reason for GF's type of `CAdvAP` is to admit adjectival arguments with complements, e.g. *as [proud of his work] as John*, but this would apply as well to `ComparA` and e.g. *[prouder of his work] than John.*)[71]

The implementation of `CAdvAP` adds the first part of the comparative adverb *as–as*, in German *(eben)so–wie*, to the paradigm forms in `s` and the second to the comparision field in `s2`:[72]

```
CAdvAP cadv ap np =
let adv : Str * Str = cadv.s ! False in
  ap ** {
    s = \\afl => adv.p1 ++ ap.s ! afl ;
    s2 = \\c => ap.s2 ! c ++ adv.p2 ++ np.s ! False ! c ++ np.ext ++ np.rc ;
    isPre = True
  } ;
```

The `isPre` field is set to `True`, so that the resulting adjective phrase `ap` can be used as pre-nominal attribute in `AdjCN ap cn`, or rather as attribute wrapped around the noun.

Remark 60: The original implementation restricted the comparison part to be a noun phrase in nominative. But besides e.g. *ich besitze einen kleineren Wagen als du*, one can also say, e.g. *ich kaufe einen kleineren Wagen als diesen Sportwagen.* Hence the restriction is removed here.

[71] Q41: Can this be improved with incomplete adjective phrases `APSlash = AP/NP` and appropriate complentation and modification rules?

[72] Since `ap:AP`, not `ap:A`, the use of the adjective's comparison object `ap.s2` is to avoid a metavariable in the parse tree. This clearly indicates that the argument `ap` should not itself be constructed by `CAdvAP`, but it can.

As remarked earlier, the modification rule

```
SentAP  : AP -> SC -> AP ;  -- good that she is here
```

ought to be a construction rule `ComplAS : AS -> S -> AP` or `ComplAV : AV -> VP -> AP` with subcategories `AS` and `AV` of `A`. `SentAP ap sc` is intended to combine the sentential, infinitival or interrogative complement `sc:SC` of the (head) adjective of `ap` to the `ext`-field of an `ap`. (The example *good that she is here* is misleading, as it shows an adjective with a sentential subject, *[that she is here] is good*. An adjective with sentential object is *happy* in *(John is) happy that she is here* or *(John is) happy to be alive*. There are adjectives with interrogative subject, e.g. *ob es besser wird, ist unklar*, but I don't see an adjective with interrogative object.)

Typed as a modification rule, `SentAP` is overgenerating, since it can be applied iteratively and there is no restriction to the argument `ap`. The implementation `SentAP ap sc` concatenates the sentential complement `sc.s` to the extraction field `ap.ext` of the argument `ap`:

```
SentAP ap sc = ap ** {
  isPre = False ;
  ext = ap.ext ++ sc.s
  } ;
```

(It seems to be assumed here that `ap.ext` is the empty string.) The complement can be separated from the adjective, as in e.g. *Johann ist froh [darüber] gewesen, daß sie kommt*.

The sentential complement of an adjective phrase built by `SentAP` can be a subject-sentence (or -infinitive or -interrogative clause), e.g. *that S, is good* or *why S, is unknown*, or an object-sentence, e.g. *I find inacceptable, that S* or *they considered it unbelievable, why S*. But such adjective phrases can hardly be used as pre-nominal attribute, e.g. *der gelobt zu werden begierige Schüler*. Instead, they can be used as uninflected post-nominal attribute (or apposition), e.g. *der Schüler, begierig, gelobt zu werden,* or *Johann, froh, daß sie gekommen war,*. The flag `isPre` is set to `False` to indicate that the attributive usage of the adjective phrase (in `Noun.AdjCN`) is not pre-nominal, but post-nominal.

The modification of adjective phrases by "adadjectives",

```
AdAP    : AdA -> AP -> AP ; -- very warm
```

puts the adadjective in front of the forms of the adjective phrase's paradigm, but leaves the complements intact:

```
AdAP ada ap = ap ** {s = \\a => ada.s ++ ap.s ! a} ;
```

This is certainly not very precise, hence overgenerating. It ignores that some adadjectives modify adjective phrases in specific degrees, e.g. *sehr|zu dumm*, but **sehr|zu dümmer*, **sehr|zu dümmste*, or *wenig|kaum|viel dümmer als ...*, but **wenig|kaum|viel dumm wie ...*.

Remark 61: One could change the linearization types of `AP` and `AdA` and implement

```
AdAP ada ap = ap ** {s = \\deg,a => ada.s ++ ap.s ! ada.deg ! a} ;
```

The rules `PositA`, `ComparA` and `SuperlA = AdjOrd o OrdSuperl` could be combined to a rule `UseA : A -> AP`. Then all uses of adjectives had to provide a degree to be passed to the adjective

phrase. This could be done by splitting the rule `AdjCN` for attributive and `CompAP` for predicative usage into three rules each. But when a modification fixes the degree of an adjective phrase, one cannot apply another (degree-fixing) modification rule to the result. A dead end? Alternatively, one perhaps can use dependent types and `AdAP : (d:Degree) -> AdA d -> AP d -> AP d`. In the end, different subclasses of adjectives may be modified in specific ways only, depending on their meaning, e.g. scalar adjectives like *old* by absolute resp. relative scale values, e.g. *50 years old* resp. e.g. *as old as John* or *much older than John*, but colour adjectives like *blue* by other adjectives, e.g. *light blue* or *dark blue*. ◁

In English, an adjective phrase can be postmodified, say by a prepositional phrase,

```
AdvAP   : AP -> Adv -> AP ; -- warm by nature
```

In German, the modifying adverb precedes the adjective,

```
AdvAP ap adv = ap ** {s = \\a => adv.s ++ ap.s ! a} ; -- HL 1/2024
```

e.g. *von Natur aus warm*, or *(das) mit Abstand bevölkerungsreichste (Land)*, or *(ein) heutzutage selten gelesenes (Buch)*. In attributively used adjective phrases, the modifying adverb (as well as the nominal complements) precede the adjective, e.g. *ein vor Angst krankes Kind*, so that the adjective inflection comes at the end. But in predicative or adverbial usage, the adverb may follow the adjective, e.g. *krank vor Angst* or *treu seinem Auftrag*.

The attributive usage of adjectives is implemented by `AdjCN : AP -> CN -> CN` in the module `Noun`, the predicative usage by `CompAP o UseComp : AP -> VP` in the module `Verb`. Adjectives can also be used as adverbs, e.g. *der Wagen fuhr schnell*. Adjectives in superlative can be used as adadjectives, e.g. *höchst dumm* or e.g. *äußerst klug*, maybe also *schellstmöglich*. These are treated in the module `Adverb`.

Todo 32: Add test examples to `lintest.gfs`, with doubly modified adjective phrases.

5.3. Adverb Phrases

`Lang` differs between `AdV`, the "*adverb directly attached to verb*" (e.g. "*always*"), and `Adv`, the "*verb-phrase-modifying adverb*" (e.g. "*here*"). Among the verb-phrase-modifying adverbs, `Lang` distinguishes between definite adverbs `Adv`, e.g. *here*, and interrogative adverbs `IAdv`, e.g. *where*. The indicated difference between `Adv` and `AdV` by occcurence position in sentences does not hold for German: *immer = always* is not directly attached to the verb in German, but rather between two nominal objects of a `v:V3`: *Johann hat mir immer eine Zigarette angeboten* is preferred over alternatives[73]

Still, different *adverb insertion functions* may be needed for German, since adverbial clauses are ususally sentence-initial or -final. According to `Verb.gf`, `AdvVP` is to add adverbs at the end of a `vp`, while `AdVVP` is to attach them next to or before the verb. But is it plausible that the position of an adverb corresponds across languages?

Presumably, the category `AdV` corresponds to *sentence-modifying adverb* (*Satzadverb* in German). But even if the semantic difference between predicate modifiers `Adv` and sentence operators `AdV` is made, I don't see a systematic difference in the position of these adverbs in German sentences.

5.3.1. Categories of Adverbs

The resource grammar `Grammar` does not distinguish between a lexical category `Adv` and a phrasal category `AdvP` of adverbs, as one might expect. There are four categories `Adv`, `AdV`, `IAdv` and `CAdv` of adverbs at the end of verb phrases, adverbs close to the verb, interrogative adverbs and comparative adverbs (or adverb prases). Their linearization categories are the same for all languages of the library, given (in `CommonX`) as

```
Adv = {s : Str} ;
AdV = {s : Str} ;
IAdv = {s : Str} ;    -- interrogative adverb
CAdv = {s,p : Str} ;  -- comparative adverb
```

While these types are as expected for lexical adverbs[74], they are insufficient for adverbial phrases. In German, adverbs ought to be split strings. First, adverbs can have a movable comparison part: *Johann ist schneller gefahren als sein Freund* or *er ist schneller gefahren als [es] erlaubt ist*. Second, adverbs can consist of a "pronominal" adverb, e.g. *dort*, plus a movable relative clause: *Die CO_2-Anlagen sind dort geplant, wo die Klimakrise bereits besonders heftig wütet.* The relative clause can be fronted, e.g. *wo ... wütet, dort sind ... geplant*, while the comparison part apparently cannot. Hence the implementation type ought to have three fields:

```
Adv = {s : Str ; cp : Str ; rc : Str} ;
```

Q42: But what about the degree of an adverb: *you work well, he works better, she works best*? e.g. *zur Zeit des Krieges habe man besser gelebt als jetzt.* While adverbs derived from adjectives might vary in degree, others certainly do not, e.g. *in this way*. So it seems better to generate three adverbs from an adjective.

[73] *Johann hat immer mir eine Zigarette angeboten* or *Johann hat mir eine Zigarette immer angeboten.* For binary verbs, it may be different: *Johann hat immer seine Arbeit gemacht* and *Johann hat seine Arbeit immer gemacht* are used.

[74] One might distinguish pronominal adverbs, e.g. *here*, *now*, etc., from adverbial phrases in general and add a field `isPron : Bool` to the implementation type to mark the difference.

On the assumption that the category `AdV` corresponds to sentence-modifying adverbs, the following implementation type seems correct:

```
AdV = {s : Str} ;
```

There is also a (lexical) category `CAdv` of ***comparative adverb***s. Comparative adverbs have two string parts, e.g. ⟨*weniger,als*⟩ and ⟨*[genau] so,wie*⟩. To construct from `cadv:CAdv` an adverb can afford a different linearization of `cadv` than to construct a cardinal modifier (i.e. an `AdN`), e.g. *(she did it) as well as John*, but *(she has) exactly 4 (children)*, and *(she did it) better than John*, but *(she has) more than 4 children*. Hence, a `CAdv` has a table of string pairs and a field to select the degree of the adjective used when forming an adverb:

```
CAdv = {s : Bool => Str * Str ; deg : Degree} ; -- True for AdN ; False for Adv
```

The rule `AdnCAdv` selects the `True` part of the paradigm, the rule `ComparAdvAdj` the `False` part.

5.3.2. Construction of Adverbs

Construction of Comparative Adverbs

As the type of `CAdv` is changed, the operation `mkCAdv` of `common/CommonX.gf` is overwritten by an operation

```
mkCAdv : Str * Str -> Str * Str -> Degree -> CAdv
```

which creates a record of type `CAdv` by inserting the arguments in the corresponding fields of the record. Examples of comparative adverbs are given in `StructuralGer` by

```
as_CAdv = P.mkCAdv <"genau",[]> <"so","wie"> Posit ; -- genau 5 ; so gut wie np
less_CAdv = P.mkCAdv <"weniger","als"> <"weniger","als"> Posit ;
more_CAdv = P.mkCAdv <"mehr","als"> <"","als"> Compar ;
```

There may be user-defined comparative adverbs like ⟨*"anders","als"*⟩ in e.g. *(sie) hat Covid nicht anders bewertet als andere Infektionskrankheiten.*

Construction of Adverbs

Lexical adverbs are built with an auxiliary operation `mkAdv : Str -> Adv`, where

```
mkAdv str = {s = str ; cp,rc = []} ;
```

Some atomic adverbs are given in the modules `LexiconGer` and `StructuralGeg`, for example

```
already_Adv = mkAdv "schon" ;
now_Adv = mkAdv "jetzt" ;
today_Adv = mkAdv "heute" ;
everywhere_Adv = mkAdv "überall" ;
```

An often used construction of adverbs is from preposition and noun phrase,

```
PrepNP      : Prep -> NP -> Adv ;        -- in the house
```

which combines[75] the preposition with the noun phrase in the case demanded by the preposition:

```
PrepNP prep np = {s = appPrepNP prep np ; cp,rc = []} ;
```

Remark 62: The argument `np` may have a sentential object, which sometimes is an extractable part of the adverb, e.g. *sie hatten in der Absicht trainiert, das Spiel zu gewinnen.* To implement this, we could add a field `ext:Str` to the implementation type of `Adv` and use a variant of `appPrepNP` that allows us to lift `np.ext` or `np.rc` to the `ext`-field of `PrepNP prep np`. ◁

Next, there are adverbs constructed from an adjective. The construction

```
PositAdvAdj : A -> Adv ;                  -- warmly
```

uses the predicative, uninflected form of an adjective adverbially:

```
PositAdvAdj a = {s = a.s ! Posit ! APred ; cp,rc = []} ;
```

Comparative adverbs with a noun phrase as comparision,

```
ComparAdvAdj  : CAdv -> A -> NP -> Adv ; -- more warmly than John
```

add the noun phrase in nominative to the comparison field of the adverb, as in e.g. *(der Hund ist) weniger schnell (gelaufen) als der Hase*:

```
ComparAdvAdj cadv a np = let adv : Str * Str = cadv.s ! False in {
  s = adv.p1 ++ a.s ! cadv.deg ! APred ;
  cp = adv.p2 ++ np.s ! False ! Nom ++ bigNP np ;
  rc = []
  } ;
```

The comparative adverbs governs the degree of the argument adjective. This is used to obtain from `more_CAdv` *schneller als* rather than *mehr schnell als*. The example *more warmly than* in the rule declaration would in German be *wärmer als*, e.g. *ich trinke den Tee wärmer als du.*

Remark 63: While the comparative adverb governs the degree of the argument *adjective* in `ComparAdvAdj`, it cannot determine the degree of the argument *adjective phrase* in `CAdvAP`. So `CAdvAP more_CAdv (PostitA good_A)` gives **gut als*, not (*)*mehr gut als*. It seems dubious that a comparison adverb can be combined with an adjective phrase of any degree, e.g. *more good|better|best than*; on the other hand, the adjective can take complements, e.g. *more proud of herself than John.*)

As we implement `more_CAdv` to use the adjective in comparative in `ComarAdvAdj`, we get an ambiguity for adjectives from `CAdvAP`:

```
Lang> p -cat=AP "besser als er"
CAdvAP more_CAdv (UseComparA good_A) (UsePron he_Pron)
ComparA good_A (UsePron he_Pron)
```

[75] For `appPrepNP`, see p. 66.

◁

Remark 64: There are also adverbs (and adjectives) obtained by comparison of two adverbs (or adjectives) by, e.g. *rather more - than* -, in which the comparative degree is not used, as in the idiomatic *mehr recht als schlecht*, or *mehr|eher warm als kalt*. But this would be a construction of a different type.

Comparative adverbs with a sentential comparision part,

```
ComparAdvAdjS : CAdv -> A -> S  -> Adv ; -- more warmly than he runs
```

hold the sentence in subordinate word order in their comparison field:

```
ComparAdvAdjS cadv a s = let adv : Str * Str = cadv.s ! False in {
  s = adv.p1 ++ a.s ! cadv.deg ! APred ;
  cp = adv.p2 ++ s.s ! Sub ;
  rc = []
  } ;
```

e.g. *(er fuhr) schneller als [es] die Polizei erlaubt*. Again, the comparison clause can be separated from the adverb, e.g. *er stellte sich weniger dumm an [,] als wir gedacht hatten*.

Todo 33: We can also build adverbs from adjectives in superlative degree, e.g. *wir gehn am liebsten schwimmen*, by an additional rule `SuperlAdvAdj : A -> Adv`, implemented by

```
SuperlAdvAdj a = {s = a.s ! Superl ! APred ; cp,rc = []} ;
```

(In `Eng`, the adjective `good_A:A` has forms `well`, `better`, and `best`, but the adverbs `better_Adv` and `best_Adv` are independent entries in `DictEng.gf`.) There are also adverbs `AdV` resp. `Adv` derived from present participles, e.g. *das Problem ist weitgehend gelöst* resp. *daran wird laufend gearbeitet*. ◁

Finally, adverbs can be built from a subjunctor and a sentence via

```
SubjS : Subj -> S -> Adv ;
```

Adverbial sentences, e.g. *weil die Sonne scheint*, are obtained by putting the subjunctor in front of the sentence in subordinate word order, i.e. the finite verb is at the end:

```
SubjS subj s = {s = subj.s ++ s.s ! Sub ; cp,rc = []} ;
```

Todo 34: Implement the combination of a pronominal adverb with a movable relative clause, in various orderings, e.g. the local adverb *dort, wo der Pfeffer wächst* or its directional version *dorthin, wo ...* in e.g. *(sie sollen) dorthin gehen, wo der Pfeffer wächst*, or *wo der Pfeffer wächst, dorthin sollen sie gehen*. This seems to be missing and will need the (so far unused) field `Adv.rc`. So far, `wo` is an interrogative adverb only:

```
Lang> p -cat=QCl "wo regnet es"
QuestIAdv where_IAdv (ImpersCl (UseV rain_V0))
```

Do we need a category `RAdv` for relative adverbs to implement `dort, wo es regnet`, or can we misuse the `IAdv` for a construction `RelAdv : IAdv -> Cl -> RCl` ?

Remark 65: Perhaps the pronominal adverb *dorthin* is a correlate of the adverb *wo der Preffer wächst*, and instead of the fields `s:Str` and `rc:Str` we better had `s:Str` and `cor:Str` for an adverb correlate. Then `SubjS` could fill the `s`-field with the adverbial sentence *weil die Sonne scheint* and the `cor` field with the correlate *deshalb*. The correlate should be provided by the subjunctor, i.e. subjunctors would have to be correlate-subjunctor pairs, e.g. ⟨*deshalb, weil*⟩ or ⟨*damals, als*⟩ or ⟨*[erst] dann, nachdem*⟩. Can we do so also with ⟨*da*|*dort*|*hier, wo*⟩, where the local adverbial sentence is built with a relative pronoun *wo* instead of a subjunctor? ◁

Modification of Adverbs

Some adverbs can be modified by adadjectives like *very* in English, *sehr* in German:

```
AdAdv  : AdA -> Adv -> Adv ;              -- very quickly
```

As in English, the modifying adadjective is put in front of the adverb derived from an adjective:

```
AdAdv ada adv = adv ** {s = ada.s ++ adv.s} ;
```

as in e.g. *(der Zug fuhr) sehr schnell* or *zu schnell*. Clearly, the rule is overgenerating: if the adverb is an adverbial clause, an ungrammatical expression will arise, e.g. **sehr weil die Sonne scheint*. Moreover, an adverb modifier like `very_AdA` should only be used to modify an adverb in "positive degree", e.g. *sehr schnell*, but not one in "comparative degree", e.g. **sehr schneller als der Hase*. Conversely, *viel* modifies adverbs in comparative degree, e.g. *viel schneller als der Hase*, but not those in "positive degree", **viel schnell*.

Remark 66: For the category `AdV` of "adverbs directly attached to the verb" there is a an atomic adverb `always_AdV` in `Structural`, and construction and modification rules `PositAdVAdj` and `AdAdV` in `Extend`.

Todo 35: Discuss the negation adverb *nicht* and its relation to the polarity of clauses!

Todo 36: Adapt `insertAdj`, `insertAdv` and `insertAdV` to handle split adjectives and adverbs.

Proposal 3: To order adverbs in basic clauses we may need to refine the category `Adv` in `AdverbGer` to carry an additional field to classify adverbs:

```
lincat Adv = {s:Str; t:AdvType} ;
param AdvType = loc | dir | temp | mod | ... ;
```

Then the `VPSlash` and `VP` categories would need a field `adv : {loc,dir,temp,mod,...:Str}` to insert adverbs to appropriate fields (which would give spurious ambiguities according to which adverb is inserted first!) and order them by `AdvType` in clauses, perhaps in various `AdvOrder`s.

The `AdvType` would also be nice to classify prepositions of "semantic" type `Prep =< Adv/NP` (as distinguished from those in verb frames). The classification should be done by (overlapping) tests, e.g. `isLocal : Prep -> Bool`, to avoid adding parameters to `Prep`.

How to translate adverb orderings to other languages (using parameters of `cl.s : Tense => ... => Ord => AdvOrd => Str`, probably)? Should adverbial clauses be classified also? ◁

Morphological Verb

The various verb categories `V`,...,`V2V`,`V3` all are extensions of the type of ***morphological verb***:

```
Verb : Type = {
  s : VForm => Str ;
  prefix : Str ;
  particle : Str ;
  aux : VAux ;
  vtype : VType
  } ;
```

Each field of this record type gives the type of the information in the record of a verb `v:Verb`.

The field `v.s` of `v` contains the inflection paradigm of the verb. The verb form parameter `VForm` of `v.s : VForm => Str` distinguishes the infinite, finite, imperative and participle forms of `v`:

```
param VForm =
   VInf Bool           -- True = with the particle "zu"
 | VFin Bool VFormFin  -- True = prefix glued to verb
 | VImper    Number    -- prefix never glued
 | VPresPart AForm     -- prefix always glued
 | VPastPart AForm ;
```

The finite forms vary in tense, mood, number and person, so one would expect a constructor `VFin Tense Mood Number Person`. But the four mood and tense variations are here represented by four constructors `VPresInd`, `VPresSubj`, `VImpfInd`, `VImpfSubj` of a parameter type `VFormFin`, where `Pres` and `Impf` indicate the *Präsens* and *Präteritum* (or *Imperfekt*) tense, and `Ind` and `Subj` indicate the *Indikativ* and *Konjunktiv* mood:

```
param VFormFin =
   VPresInd  Number Person
 | VPresSubj Number Person
 | VImpfInd  Number Person --# notpresent
 | VImpfSubj Number Person --# notpresent
 ;
```

This grouping (presumably) is to highlight the finite forms as those varying in (at least) both number and person; there is no further use of the parameter type `VFormFin` made in `LangGer`. (The lines marked by `--# notpresent` are ignored when the grammar is compiled to a restricted version that covers the present tense only.)

The value of the parameter `Bool` in `VFin Bool VFormFin` steers whether the prefix of the verb is glued to the stem or not.[76] ...

[76] The value of `Bool` is fixed when `v:Verb` is turned to a verb of a lexical category `V`,...,`V3` and the paradigm `v.s` is extended to the paradigm of the lexical verb. Q43: Why is the dependence of the participle and infinite not marked by a `Bool` – because it is not needed in the present participle?

The participle forms are not grouped like `VPart Tense AForm`, but given by two separate constructors `VPresPart AForm` for the *Partizip Präsens* and `VPastPart AForm` for the *Partizip Perfekt*[77], where the parameter type `AForm` covers both the predicative and adjectival forms of adjectives.

The field `v.prefix` of `v:Verb` holds the prefix attached to (most of) the forms in `v.s` when `v` is extended to a verb `v:V` or some of the other verbal categories. Likewise, the field `v.particle` holds a separable particle, e.g. *Lehrgeld zahlen*. The field `v.VAux` of type

```
param VAux = VHaben | VSein ;
```

fixes which of the perfect auxiliaries *haben* or *sein* the verb needs. Finally, the field `v.VType` stores the verbtype of `v`, using the parameter values

```
param VType = VAct | VRefl Case ;
```

Actually, the only reflexive verbs seem to be of verbtype `VRefl Dat` and `VRefl Acc`, so the parameter type `VType` might be reduced somewhat.

Lexical Verb This partial classification of morphological verbs `v:Verb` is extended to the full syntactic arity when `v` is turned into a verb of the lexical categories of the grammar `Lang`, namely

```
lincat
  V, VA, VS, VQ = ResGer.Verb ;
  VV = Verb ** {isAux : Bool} ;

  V2, V2A, V2S, V2Q = Verb ** {c2 : Preposition} ;
  V2V = Verb ** {c2 : Preposition ; isAux : Bool ; objCtrl : Bool} ;

  V3 = Verb ** {c2, c3 : Preposition} ;
```

The verbs of category `VA`, `VS`, `VQ` take an adjectival, sentential and interrogative object, those of category `V2` and `V3` take one and two nominal objects, respectively. The fields `c2` and `c3` hold the preposition or just the case used to attach the nominal object to the verb. The verbs of category `V2A`, `V2S`, `V2Q` are those with a nominal and an adjectival, sentential and interrogative object respectively. (There is no category for verbs with adverbial complement, e.g. *an einem Ort wohnen* in *Johann wohnt hier nicht mehr*.) These verb categories don't restrict the subject complement, although only few verbs admit sentential, interrogative or infinitival subjects. There is also no category for nullary verbs, i.e. verbs with the expletive subject *es*, like the weather verbs, e.g. *heute regnet es*; so the grammar does not exclude *das Haus regnet*.

The verbs `v` of category `VV` and `V2V` take an infinitival complement; those with field `v.isAux = True` are auxiliary verbs, which take a pure *Inf* complement and use the infinitive as past participle form, e.g. *sie will arbeiten* and *sie hat arbeiten wollen* as well as *ich lasse sie arbeiten* and *ich hatte sie arbeiten lassen*, whereas those with `v.isAux = False` take an *Inf-zu* complement and use the participle form, e.g. *sie hofft, zu arbeiten* and *sie hat gehofft, zu arbeiten*[78] as well as *ich verspreche|rate dir, zu arbeiten* and *ich hatte dir versprochen|geraten, zu arbeiten*.

[77] Q44: What is the reason for this different grouping?

[78] The comma and the extraction to the right are less common, if the complement is short.

For `V2V`, the field `objCtrl` is to distinguish object- from subject-control verbs by values `True` and `False`. Depending on the value of `v.objCtrl` of `v:V2V`, reflexive pronouns in the infinitival complement have to agree in person and number with the nominal object or the subject of `v`.

5.4.1. Verb Phrases `VP` and Incomplete Verb Phrases `VPSlash`

Verb phrases have more "tenses" than verbs; they are built from the verb tense in combination to an anteriority parameter:

```
param VPForm =
   VPFinite  Mood Tense Anteriority
 | VPImperat Bool
 | VPInfinit Anteriority ;
```

In `gf-3.3`, the implementation category of `VP` was

```
VP : Type = {
    s  : Verb ;
    a1 : Polarity => Str ; -- nicht
    n0 : Agr => Str ;      -- dich
    n2 : Agr => Str ;      -- deine Frau
    a2 : Str ;             -- heute
    isAux : Bool ;         -- is a double infinitive
    inf : Str ;            -- sagen
    ext : Str              -- dass sie kommt
    } ;
```

Clearly, the `s`-field is intended to hold the verb of the `VP`, the `a2`-field holds an adverb (or an adverbial clause, added by `ExtAdvVP`). But why does the `a1`-field, which seems to hold the negation, depend on `Polarity`? To be able to change the polarity of a tree and then linearize correctly?

The participle perfect of a modal verb `v` exists, as in *Er hat das gewollt*, but is replaced by the infinitive, if `v` comes with an infinitive complement, like *Wir hatten aufbrechen wollen.* (According to Eisenberg [3], das "führt zum Aufeinandertreffen von zwei Infinitiven" – the *double infinitive*) So, `isAux = True` seems to mean the verb `v` is a modal verb. And the `inf`-field takes the infinitive (or `zu`-infinitive) complement of `v`, and `ext` the sentential complement. The fields `n0` and `n2` seem to separate reflexive pronominal from other reflexive nominal complements.

Remark 67. *In `gf-3.3`, `VP.a1:Polarity => Str` contained the negation `nicht`, and `insertAdV` added adverbs `adv:AdV` in front of `VP.a1`, example: `immer nicht` – but shouldn't this be `nie` ?–, while `insertAdv` added adverbs `adv:Adv` at the end of `VP.a2`. Does this distinction really exists in German, or is `gf-3.10` right with assuming it doesn't exist? At least, several adverbs sometimes occur consecutively in a clause, e.g "(die Außenminister verabschiedeten die Maßnahmen) am Freitagabend bei einem Treffen in Brüssel".*[79] *So, `insertAdv` seems useful. Was the idea of `a1 : Polarity => Str` to make certain adverbs depend on polarity, so that we might have*

[79] Notice the ambiguity in `(prep (DetCN det cn)) (prep np)` versus `(prep (DetCN det (cn prep np)))`, as for *(am Abend) (beim Treffen) (in Brüssel)* versus *(am Abend) (beim (Treffen in Brüssel))*.

`always_AdV = {s = \\p => case p of {Pos => "immer"; Neg => "nie"}}` and `insertAdV adv vp = vp ** {a1 = \\p => adv ! p ++ vp.a1 ! p}`? Can it work with non-empty `vp.a1` or several `AdVs`? At least this was not so since `gf-3.0`.

(`VP` of `gf-3.10` hidden.)

Q45: What caused the complexity in `gf-3.9`? There are complexity issues with verbs of arity 4 or higher; with contracted pronouns *am*, ..., *zum*; with case distinction of moving/extracting infinite complements etc.

Redesign: (forget the predicative `AP,CN,Adv` in `nn.p4` for the moment)

Among its fields, a verb phrase `vp:VP` should have a field `s:Verb` for its main verb, a field `nn:Agr => Str` for the verb's nominal objects (the reflexives of which depend on the agreement features of the missing subject), a field `ext` for sentential and interrogative (right-extracted) complements, and a field `inf` for the infinitival complement of `vp.s : Verb`.[80]

The infinitival complement of auxilary verbs `v:VV` or `v:V2V` is hold ***in-place***, e.g. *ich habe schlafen wollen* and *ich habe ihn schlafen lassen*, those of full verbs `v:VV` or `v:V2V` are ***extracted*** to the right, e.g. *ich habe versucht, zu schlafen* and *ich habe ihn gebeten, nicht zu schlafen*. More precisely, the infinitival complement is not continuous, but in general split into an in-place and a right-extracted *part*. We hold the in-place part of `vp`'s infinitival complement[81] in `vp.inf.inpl` and the extracted part in `vp.inf.extr`. Since an infinitival complement may contain reflexives that have to agree with the subject (or a nominal object) of the matrix verb, the components have to be of type `Agr => Str`, e.g. *ich will mir selbst helfen*, *du willst dir selbst helfen* etc., so *sich selbst helfen wollen* : `Agr => Str`.

However, for in-place infinitival complements, this type `Agr => Str` will not quite do, since the in-place part is in general a split string. Namely, in a verb phrase whose main verb is an auxiliary verb, e.g. *wollen*:`VV` or *lassen*:`V2V`, in Futur-II its infinitival complement is split by the inserted tense auxiliary *haben*, e.g. the infinitival complement *euch helfen* in *man wird (euch haben helfen) wollen* and *man wird ihn (euch haben helfen) lassen*. To separate nominal objects from the predicate (the verb in infinitive form, modified by adverbs), we need a splitted field[82]

```
vp.inf.inpl : (Agr => Str)*Str.
```

The infinitival complement may be a verbal phrase with an embedded infinitival complement. By wrapping a further `<obj,pred>` pair around its inner infinitival complement, we can have a `vp` with a nested infinitival complement, and if it is used in finite form in tense *Futur-II*, say in `PredVP np vp` for ich_0 *werde* ihn_1 dir_2 *haben* $helfen_2$ $lassen_1$ $müssen_0$, we have to insert *haben* into the gap of the innermost infinitival complement $\langle dir,\ helfen\rangle$.

The split point may be used to insert a correlate *es* for moved inf-zu infinitives, at least if the ***original position*** of an infinitival object comes before the nominal object (of a `V2V` verb), as in

[80] In `Eng, Ger` of `gf-3.9`, `vp.inf` did just hold the infinite form of the verb `vp.s` (which was not used for `eng`), but we here change this to let `vp.inf` be the infinitival complement of `vp.s`. We should also *not* store an *embedded* infinitival complement separately, but distinguish between in-place and extracted infinitival complement. In `gf-3.6`, `Ger.ComplVV v vp` added `infExt ++ vp.exp` to `rvp.ext`, but this was the embedded infinitive.

[81] assuming that there is only one; but in general, a sentence can have several: *to be is better than not to be.*

[82] The object depends on `Agr` since it may be a reflexive pronoun, e.g. *mir selbst* in *du wirst mich mir selbst haben helfen lassen wollen* instead of *du wirst mich dir haben helfen lassen sollen *(gesollt haben)*. The predicate, i.e. the verb's infinitive, modified by adverbs, does not depend on `Agr`. Q46: This is not true for adverbs: *ich werde jmdn in seinem(!) Haus haben wohnen lassen müssen*, where *haben* is inserted between adverb and verb infinitives!

ich habe (euch zu helfen) ihm empfohlen ↦ *ich habe [es] ihm empfohlen, euch zu helfen*, just as the indirect nominal object (`v.c3`) comes before the direct nominal object (`v.c2`) of a `V3` verb (in the unmarked ordering). But possibly there is a ***correlate switch*** involved that moves the *es* forward, and the original position of the infinitival object follows the nominal object?

`VP` should also have a field `inf.extr : Agr => Str` for an ***extracted*** (even direct?) infinitival complement. The two fields `inpl` and `extr` of `vp.inf` can be used simultaneously, as one of them contributes an empty string (unless extraction leaves a correlate *es* in `vp.inf.inpl`).

When the `vp` is used as the finite verb phrase of a clause, by `PredVP np vp` or `PredSCVP sc vp`, we have to decide on the relative order between `vp.inf.extr` and `vp.ext`. The content of `vp.inf.extr` might alternatively go to `vp.ext`, as there seem to be no verbs with both a sentential (or interrogative) *and* an infinitival complement.[83]

There is no need to split `inf.extr` like `inf.inpl` into `<objs,pred>`, since `inf.extr` can only be used as infinitival complement, for which there is no need for the gap for a missing temporal auxiliary *haben*. Extracted infinitival complements are nested by embedding them to the right, e.g. *ich habe (ihr empfohlen, (dich zu bitten, ihr zu helfen))*, also with auxiliary verbs in between, e.g. *ich habe ihr empfehlen müssen, dich zu bitten, ihn ihr helfen zu lassen*. Extracted infinitival complements may depend on agreement features over several embedding levels, which makes reflexive resolution non-obvious. For example, we can have *ich muß dir|euch raten, (ihr zu versprechen, (dich|euch um sie zu kümmern))*, with control verbs of different control.

Q48: Do we want to be able to *define* such complex predicates like *jmdm raten, zu versprechen, sich zu kümmern* or *jmdm empfehlen, zu versuchen, sich anzustrengen*, or *jmdn schwören lassen, (sich von jmdm fernzuhalten)*? We'd need a clear difference in defining *to let sb help himself* versus *to let sb help oneself*.

Therefore, the implementation category `VP` should be as follows:

```
VP : Type = {
   s  : Verb ;   -- schlafe:V,lese:V2, will,hoffe:VV, lasse,verspreche,rate:V2V
   -- nominal,prepositional object or comp of s
   -- HL 3/2021: nn = <refl|pron,NP,PP,AP|CN|Adv>  -- pron,light,heavy,comp
   --                 <sich|ihr,deine Frau,an sie,gut>
   nn : Agr => Str * Str * Str * Str
   adj : Str ;      -- adjectival complement of s:V(2)A, e.g. ich finde dich schön
   a1 : Str ;       -- adv before negation, adV  -- e.g. hat es heute nicht getan
   a2 : Str ;       -- adv at the end -- e.g. hat es [deshalb] nicht getan, weil S
   isAux : Bool ;   -- auxiliary, e.g. (hat es tun) wollen (*gewollt)
   ext : Str ;      -- sentential complement of s:V(2)S, e.g. dass|wann sie kommt
   -- infinitival complement of s:V(2)V, e.g. sich zu tun
   -- e.g. will:VV tun     | hoffe:VV, zu tun,
   -- lasse:V2V (dich) tun | verspreche,rate:V2V (dir), mich|dich zu bessern
   inf : {inpl : (Agr => Str)*Str ; extr : Agr => Str} ;
   c1 : Preposition -- case of subject, e.g. mich friert
```

[83] Q47: The `ext`-field should not hold an `np`-part (e.g. a postponed relative clause), if it is to contain a sentential or infinitival complement. Can we ensure this? For example, `v:V2` admits extracted relative clauses from nominal objects, e.g. *ich habe den Beweis nicht verstanden, den du skizziert hast*, but probably `v:V2V` does not: is *ich bitte dich, den Beweis nochmal zu erklären, den du skizziert hast* ok, or should it read *ich bitte dich, den Beweis, den du skizziert hast, noch einmal zu erklären*? But the first is just the infinitive of *den Beweis verstehen, den ...* Here we could glue the extraction to the verb infinitive, when using `infVP isAux vp`.

```
  } ;
```

Remark 68. *Alternatively, we could have* `inf : Agr => Str * Str * Str` *and use the first two strings to build the inplace, the third for the extracted infinitival complement, which would make reflexive resolution simpler.*

The sentential complements `VP.ext` *might also be of type* `Agr => Str`*. A sentential complement can contain an open reflexive personal or possessive pronoun that refers to the subject or object of the matrix verb: "er hat ihr|ihm gesagt, daß man ihn selbst fragen soll". One could then, for* `ext`*, too, resolve reflexive (personal or possessive) pronouns as for* `inf`*. One could possibly also merge* `inf.extr` *with* `ext : Agr => Str`*.*

If the resolution method is inadequate, having `ext:Str` *and using* `SelfNP` *might be the better solution. (But* `SelfNP` *and* `SelfAdvVP` *give a lot of trees.)*

The `nn.p4`-field of `VP` must have type `Agr => Str`, because there are reflexive complements to copula verbs, e.g. (*to be one's own boss*):`Comp`, in `Ger`: *Johann ist sein eigener Chef* and *du bist dein (eigener) Chef, er ist größer als sein eigener Vater* etc.

Remark 69. *Position of clause negation and reflexive object of infinitival constituent: [es] nicht wissen, sich zu helfen = nicht (sich zu helfen) wissen ↦ sich nicht zu helfen wissen ≠ (sich nicht zu helfen) wissen. It is hard to tell the scope of negation from the surface word order.*

P.Weiss, Marat/Sade: "So verseucht sind wir von den Gedankengängen / die Generation von Generation übernahm / daß auch die besten von uns / sich immer noch nicht zu helfen wissen".

The category `VPSlash` extends `VP` by a field `c2:Preposition` used to add a prepositional or nominal object, and by a field `objCtrl:Bool` used to resolve reflexive pronouns:

```
VPSlash =
  VP ** {c2 : Preposition ;
         objCtrl : Bool } ; -- True = embedded reflexives agree with object
```

A third category involved in the construction of verb phrases is the category `Comp` of complements to copula verbs. It consists of a paradigm depending on agreement features and a field for an extracted part:

```
  Comp = {s : Agr => Str ; ext : Str} ;
```

The inflection paradigm varies on `Agr`, since its copula verb has to agree with the subject when used as a predicate. Moreover, reflexive pronouns and reflexive possessives in the complement refer to the implicit subject of the copula verb, e.g. e.g. *sich treu sein*, or *sein (eigener) Chef werden* or *älter als ihr|sein (eigener) Bruder sein*. Part of a complement can be extracted, e.g. part of the complement *der Chef, der es allen recht macht* is extracted in *du kannst nicht der Chef sein, der es allen recht macht*. But in general, the extracted part could also depend on `Agr`: *sie wird besser sein als ihr Bruder*. (Todo 37: So `AP` has to be changed to make also the `ext` part of an adjective phrase depend on `Agr`.)

Remark 70. *The default implementation of reflexive possessives by* `../common/ExtendFunctor.gf : ReflPossPron = PossPron he_Pron : Quant` *doesn't get the dependece on agreement right:*

```
TestLang> l (PredVP (UsePron she_Pron) (UseComp (CompAP
    (ComparA old_A (DetCN (DetQuant ReflPossPron NumSg) (UseN2 brother_N2))))))
she is older than his brother
sie ist älter als [ReflPossPron] Bruder
```

Remark 71: missing optional dative: *du bist mir ein schöner Trottel*

Most constructions of verb phrases first build an initial `vp:VP` from a verb `v:V`, with `v` in `vp.s` and default values in the remaining fields (see `predV : V -> VP`, p. 122 below), and then insert complements of type `Str` or `Agr => Str` to the fields `vp.nn`, . . . , `vp.inf`. Some of these insertion operations just append a string to the left or right of an existing string value:

```
  insertAdV : Str -> VP -> VP = \adv,vp -> vp ** {
    a1 = adv ++ vp.a1 } ;

  insertAdv : Str -> VP -> VP = \adv,vp -> vp ** {
    a2 = vp.a2 ++ adv } ;

  insertExtrapos : Str -> VP -> VP = \ext,vp -> vp ** {
    ext = vp.ext ++ ext } ;
```

These operations can be used iteratively (e.g. via `AdvVP : VP -> Adv -> VP` or `SlashVV : VV -> VPSlash -> VPSlash`), so that several adverbs or extraposed elements occur consecutively at one position in a clause. (Q49: when to add a comma in front of an extraposed element?)

Remark 72: currently, `insertAdV` is not used in `LangGer`, all adverbs are collected in `vp.a2`.

Complements of a copula verb are inserted into the fourth component of the field `vp.nn` of the verb phrase `vp` spanned by the copula verb, using

```
  insertObj : (Agr => Str) -> VP -> VP = \obj,vp ->  -- obj:Comp A|Adv|CN
    vp ** { nn = \\a => let vpnn = vp.nn ! a in
                        <vpnn.p1, vpnn.p2, vpnn.p3, obj ! a ++ vpnn.p4> } ;
```

Nominal and prepositional complements of a verb are inserted into the first three components of the field `vp.nn` of the initial verb phrase spanned by the verb. The main insertion operation is the following `insertObjNP`, but there is a further one, `insertObjRefl`, below.[84]

```
  insertObjNP : NP -> Preposition -> VPSlash -> VPSlash = \np,prep,vp ->
    let obj = appPrep prep np ;
        b : Bool = case prep.t of {isPrep | isContracting => True ; _ => False} ;
        w = np.w ;
        c = prep.c
    in insertObj' obj b w c vp ;

  insertObj' : Str -> Bool -> Weight -> Case -> VPSlash -> VPSlash =
    \obj,isPrep,w,c,vp -> vp ** {
      nn = \\a =>
```

[84] *Remark 73*: `insertObjc` to insert an `np` into a `VPSlash` is no longer used, except in `ParseGer`.

```
      let vpnn = vp.nn ! a in
      -- HL 11/6/19: rough object NP order (expensive):
      -- vfin < accPron < refl < (gen|dat)Pron < lightNP
      --                       < neg < heavyNP|PP < vinf|comp
      case <isPrep, w, c> of { -- 2 * 3 * 4 = 24 cases
        <True, _,_> =>       -- <prons, light, heavy++pp, compl>
          <vpnn.p1, vpnn.p2, vpnn.p3 ++ obj, vpnn.p4> ;
        <False,WPron, Acc> => -- <ihn ++ sich, light, heavy, comp>
          <obj ++ vpnn.p1, vpnn.p2, vpnn.p3, vpnn.p4> ;
        <False,WPron, _  > => -- <sich ++ ihm|seiner, light, heavy, comp>
          <vpnn.p1 ++ obj, vpnn.p2, vpnn.p3, vpnn.p4> ;
        <False,WLight,Dat> => -- (assuming v.c2=acc) nonPron: dat < acc|gen
                              -- <prons, dat ++ np, heavy, comp>
          <vpnn.p1, obj ++ vpnn.p2, vpnn.p3, vpnn.p4> ;
        <False,WLight,_  > => -- <prons, np ++ gen|acc, heavy, comp>
          <vpnn.p1, vpnn.p2 ++ obj, vpnn.p3, vpnn.p4> ;
        <False,WHeavy|WDefArt,Dat> => -- <prons, light, dat ++ np, comp>
          <vpnn.p1, vpnn.p2, obj ++ vpnn.p3, vpnn.p4> ;
        <False,WHeavy|WDefArt,_  > => -- <prons, light, np ++ gen|acc, comp>
          <vpnn.p1, vpnn.p2, vpnn.p3 ++ obj, vpnn.p4> }
    } ; -- the ordering of objects of v:V3 also depends on Slash?V3
```

The nominal and prepositional complements are inserted into different components of `vp.nn` so that when extending the verb phrase to a clause (see `mkClause`, p. 140), one can order the complements according to their structure, weight and case. In `vp.nn.p1` the pronouns are collected, with accusative pronoun leftmost, to implement the ***pronoun switch*** for ternary verbs: *ich schicke der Behörde einen Brief* versus *ich schicke ihn ihr* (**ich schicke ihr ihn*). Given the relatively free word order in the German Mittelfeld, no such order by structure, weight and case of the complements can be satisfying in all circumstances.[85]

Remark 74. *It is not clear if in* ***`insertObjNP np prep vp`*** *the complete nominal or prepositional object* ***`appPrepNP prep np`*** *built by*

```
appPrepNP : Preposition -> NP -> Str = \prep,np ->
      prep.s ++ np.s ! False ! prep.c ++ bigNP np ++ prep.s2 ;
bigNP : NP -> Str = \np -> np.ext ++ np.rc ;
```

should be inserted into one of the four components of the field ***`vp.nn`****, or if* ***`np.rc`*** *and* ***`np.ext`*** *better go to* ***`vp.ext`****. In the latter case, how can we know if* ***`vp.ext`*** *already contains a sentential or interrogative complement, and how should these be ordered relative to* ***`np.rc`*** *and* ***`np.ext`****?*

Constructions of `VP`

The simplest verb phrase construction is to use a unary (full) verb. This rule

```
    UseV     : V   -> VP ;        -- sleep
```

[85] See `gf-rgl/src/english/RefineEng(Abs).gf` for an experimental implementation of pronoun switch in `Eng`.

is implemented by

```
UseV v = predV v ;
```

where `predV` turns a morphological verb `v : Verb` into a verb phrase `vp = predV v : VP` by inserting `v` into `vp.s`, noting in `vp.isAux` that the verb (by default) is not an auxiliary verb, and if it is a reflexive verb, as seen from `v.vtype = VRefl c` with case `c`, inserts a suitable form of the reflexive pronoun in `vp.nn ! p1`; the remaining fields of `vp` are filled with empty constituents or default values as follows:

```
predV : Verb -> VP = predVGen False ;

predVGen : Bool -> Verb -> VP = \isAux, verb -> {
  s = verb ;
  isAux = isAux ;
  a1,a2,adj,ext : Str = [] ;
  nn  : Agr => Str * Str * Str * Str = case verb.vtype of {
    VAct => \\_ => <[],[],[],[]> ;
    VRefl c => \\a => <reflPron ! a ! c,[],[],[]>
    } ;
  inf = {inpl = <\\_ => [], []>; extr = \\_ => []} ;
  c1 = PrepNom
  } ;
```

Of the four complementation rules for binary verbs with infinitival, sentential, interrogative or adjectival complement, i.e.

```
ComplVV  : VV  -> VP -> VP ;  -- want to run
ComplVS  : VS  -> S  -> VP ;  -- say that she runs
ComplVQ  : VQ  -> QS -> VP ;  -- wonder who runs
ComplVA  : VA  -> AP -> VP ;  -- they become red
```

the latter three are simply adding the sentential, interrogative or adjective argument phrase to the corresponding fields `ext:Str` or `adj:Str` of the implementation record `predV v = vp : VP` of the verbal phrase built from the argument verb `v`:

```
ComplVS v s =
  insertExtrapos (comma ++ conjThat ++ s.s ! Sub) (predV v) ;
ComplVQ v q =
  insertExtrapos (comma ++ q.s ! QIndir) (predV v) ;
ComplVA v ap =
   insertAdj (ap.s ! APred) ap.c ap.ext (predV v) ;
```

In `ComplVS`, the object sentence `s` is used in its form `s.s ! Sub` for subordinate sentences, which has its verb at the end.[86] In `ComplVQ`, the interrogative phrase `q` is used in indirect form, which also means that the verb is at the end.

[86] We need a second rule that adds the object sentence in conjunctive form: *er behauptete, die Erde sei flach*?

In `ComplVA`, from the adjective phrase `ap` the predicative form `ap.s ! APred` of the adjective together with the post-adjective complement `ap.c.p2` is appended to `vp.adj`, the pre-adjective complement `ap.c.p1` to `vp.nn.p2` and the extracted part `ap.ext` to `vp.ext`:

```
insertAdj : Str -> Str * Str -> Str -> VP -> VP = \adj,c,ext,vp -> vp ** {
  nn = \\a =>
    let vpnn = vp.nn ! a in <vpnn.p1, vpnn.p2 ++ c.p1, -- der Frau treu
                             vpnn.p3, vpnn.p4> ;
  adj = vp.adj ++ adj ++ c.p2 ;               -- neugierig auf das Buch
  ext = vp.ext ++ ext} ;
```

So far, reflexives are not handled in complements `ap.c : Str*Str` of adjectives, c.f. Remark 55.

Q50: `insertAdj` is intended to insert the adjectival complement of a verb `v:VA`, like *er malt die Wand blau*. Can these adjectives have complements like those in the comment of `insertAdj`? Maybe they can: *wir halten dich für (ihr treu | ihm überlegen | älter als deinen Bruder)*?

The complementation rule `ComplVV` for adding infinitival complements is more difficult. The difficulty comes from the fact that a verb phrase `vp:VP` has two rather different uses, one as a predicate (employing a finite form of the verb) in a clause, and another as infinitival complement of a verb, noun or adjective (employing an infinite form of the verb). Since in German, the verb's complements in a clause can be ordered relatively freely, the fields of `vp` holding its constituents are *not* combined to a single (split) string in `vp.s`. But in an infinitival complement, the verb's complements are orderd in a rather fixed way. So, to use `vp` as infinitival complement, we first have to combine its infinite verb form and its complements into a suitable (split) string.

Hence, an application `(ComplVV v vp)` of the rule

```
ComplVV    : VV  -> VP -> VP ;                  -- want to run
```

first has to turn its argument `vp:VP` into an infinitival form and then insert this into the `inf`-field of the verb phrase `vps = (predVGen v):VP` opened by the verb `v:VV` to give the resulting verb phrase `rvp = (ComplVV v vp):VP`.

```
ComplVV v vp =                 -- HL 3/22: inf-complement in-place,
  let                          --          infzu-complement extracted
    vps = predVGen v.isAux v ; -- e.g. will.isAux=True | wagt.isAux=False
    inf = mkInf v.isAux Simul Pos vp
  in
  insertExtrapos vp.ext (
    insertInf inf vps) ;
```

Here, `mkInf` uses an auxiliary operation `infVP` to turn `vp` into an infinitival complement of `rvp`. This operation `infVP` combines `vp`'s nominal objects to `objs`, combines its verb `vp.s` in suitable infinite form with adverbs `vp.a1` and `vp.a2` to `pred`, and extracts its infinitival complement `inf`. In general, the infinitival complement of `rvp` depends on a chosen polarity and anteriority of the form of `vp.s`, e.g. *(nicht) lesen* and *(nicht) gelesen haben*[87], or the passive *(nicht) gelesen werden* and *(nicht) gelesen (worden) sein*, and on `v.isAux` to select between pure and *zu*-infinitive of `vp.s`.

[87] quite usual: *wir hoffen, Ihnen hiermit geholfen zu haben* or *Sie glauben, mir damit nicht geschadet zu haben?*

```
infVP : Bool -> Anteriority -> Polarity -> VP
  -> { objs:(Agr => Str) ; pred:Str;
       inpl:(Agr => Str)*Str ; extr:Agr => Str } =
    \isAux, ant, pol, vp -> let vps = useVP vp in
    { objs = \\agr => (vp.nn ! agr).p1 ++ (vp.nn ! agr).p2
        ++ negation ! pol ++ (vp.nn ! agr).p3
        ++ vp.a2 ++ (vp.nn ! agr).p4 ;  -- objects + predicative A|CN|NP
      pred = vp.a1 ++ vp.adj
             ++ (vps.s ! (notB isAux) ! agrP3 Sg ! VPInfinit ant).inf ;
      inpl = vp.inf.inpl ;
      extr = vp.inf.extr
    } ;
```

In the resource grammar `LangGer`, the module `VerbGer` fixes infinitives to be in simultaneous anteriority and positive polarity, so we here simply write `(infVP v.isAux vp)`.[88],[89]

For `mkInf`, basically, if `v.isAux = True`, the infinitival complement of `rvp` built from `vp` is put in-place, i.e. goes to `rvp.inf.inpl`. In this case, it roughly is `<objs,pred>`, the combination of `objs = vp.nn` paired with `pred`, the adverbially modified infinitive of `v.s!VPInf`.[90] The in-place infinitival complement has to be split in two parts, and if `vp` has its own in-place infinitival complement, this is embedded in the gap, e.g. *(ich) will ihn*$_1$ *den Hund*$_2$ *füttern*$_2$ *lassen*$_1$, using

```
-- embed <sich, helfen> into <ihn, lassen> = <ihn sich, helfen lassen>
embedInf : (Agr => Str) * Str -> (Agr => Str) * Str -> (Agr => Str) * Str =
   \f,g -> <\\a => g.p1!a ++ f.p1!a, f.p2 ++ g.p2> ;
```

For `v.isAux = False`, the infinitival complement built from `vp` by `mkInf` is extracted, i.e. goes to `rvp.inf.extr`, e.g. *(ich) wage, ihm*$_1$ *zu raten*$_1$*, den Hund*$_2$ *zu füttern*$_2$. (So, extraction is forced even for short *Inf-zu* complements, i.e. we get *(weil) man bittet, zu läuten* instead of *weil man zu läuten bittet*, and *(weil) ich glaube, es zu wissen* instead of *(weil) ich es zu wissen glaube.*)

More precisely, there is no clear decision between inplace and extracted placement of the infinitival complement, but there is an in-place and extracted *part* of the infinitival complement `rvp.inf` in *(ich) hoffe, versprechen zu können, das zu tun*: the verbal phrase `vp` = *(ich) kann (versprechen, das zu tun)* has part of its own infinitival complement *versprechen, das zu tun* moved to `rvp.inf.inpl = <[],`***`versprechen können`***`>`, the other part moved to `rvp.inf.extr = ,` ***`das zu tun`***. To get nested infinitival complements right, we distinguish whether the matrix verb `v` and the verb of the `vp` are auxiliary verbs or not:

```
glueInpl : (Agr => Str)*Str -> (Agr => Str) =
    \inplace -> \\agr => (inplace.p1!agr ++ inplace.p2) ;

mkInf : Bool -> Anteriority -> Polarity -> VP ->
              {inpl : (Agr => Str) * Str ; extr : (Agr => Str)} =
```

[88] But there is the more general `CompVV : VV -> Anteriority -> Polarity -> VP -> VP` in `ParseGer`.

[89] Q51: are the adverbials, negation and adjectival complement properly ordered in `obs` and `pred`?

[90] The infinitival complement `vp.inf.extr` cannot be derived from `vp.inf.inpl` as `\\agr => vp.inf.in.p1!agr ++ vp.inf.in.p2`. This caused serious troubles for nesting, when in in-place `vp.inf.inpl` has to be turned into `rvp.inf.extr` depending on `v.isAux`!

```
\isAux,ant,pol,vp ->
   let
       vpi = infVP isAux ant pol vp ;
       topInpl = <vpi.objs, vpi.pred> ;
       emptyInpl : (Agr => Str) * Str = <\\_ => [], []> ;
       comma = bindComma
   in
     case <isAux,vp.isAux> of {
       <True,True> -- 1: will {inpl=<(sich, waschen) können>, extr = []}
           => {inpl = embedInf vpi.inpl topInpl ;
               extr = \\agr => vpi.extr!agr} ;
       <True,False> -- 2: will {inpl=<[], versuchen>, extr = sich zu waschen}
           => {inpl = topInpl ;
               extr = \\agr => (glueInpl vpi.inpl)!agr ++ vpi.extr!agr} ;
       <False,True> -- 3: wagt{inpl=<[], []>, extr = (sich, waschen) zu wollen}
           => {inpl = emptyInpl ;
               extr = let moved = embedInf vpi.inpl topInpl
                      in \\agr => comma ++ (glueInpl moved)!agr ++ vpi.extr!agr} ;
       <False,False> -- 4: wagt, {inpl=<[], []>, extr = zu versuchen,
           => {inpl = emptyInpl ;                        -- (sich zu waschen)}
               extr = \\agr => comma ++ (glueInpl topInpl)!agr ++ vpi.extr!agr}
   } ;
```

The infinitival constituent `inf` is then inserted into `vps:VP` by embedding the in-place part of `inf` into the gap of `vps.inf` and by appending the extracted part of `inf` to `vp.inf.extr`, using

```
insertInf : {inpl:(Agr => Str)*Str ; extr:(Agr => Str)} -> VP -> VP =
\inf,vp -> vp ** {inf = {inpl = embedInf inf.inpl vp.inf.inpl ;
                         extr = \\agr => vp.inf.extr!agr ++ inf.extr!agr}} ;
```

Example 2. *In German, `v.isAux = wollen.isAux = True` and `v.isAux = wagen.isAux = False`. With the following (simplified) implementation records of argument verb phrases*

```
vp1 = {s=lesen; nn=ein Buch;  inf = {inpl=[];extr=[]}}
vp2 = {s=wollen; nn=[];  inf = {inpl=<[],versuchen>;extr= sich zu waschen}}
```

we get these implementation records of the resulting verb phrase `rvp`:

```
ComplVV wollen vp1 = {s = will; nn=[];
                      inf={inpl=<ein Buch, lesen>; extr=[]}}

ComplVV wagen vp1 = {s = wagt; nn=[];
                     inf = {inpl = <[],[]>; extr =, ein Buch zu lesen}}
ComplVV wagen vp2 =
       {s = wagt; nn=[];
        inf = {inpl = [];
               extr = versuchen zu wollen, sich zu waschen}}
```

The last, and most important, complementation rule for binary verbs `v:V2` with a nominal complement is subsumed by the complementation rule

```
ComplSlash : VPSlash -> NP -> VP ; -- love it
```

for an incomplete verb phrase `vps:VPSlash` by a noun phrase `np`. The incomplete verb phrase `vps` extends a verb phrase by a field `c2:Preposition` and a field `objCtrl:Bool`. These are used by `ComplSlash` to add a nominal object and bind the reflexives to this object:

```
ComplSlash vps np =
  let vp = case vps.objCtrl of { True => objAgr np vps ; _  => vps }
           ** { c2 = vps.c2 ; objCtrl = vps.objCtrl } ;
  in insertObjNP np vps.c2 vp ;
```

If the main verb in `vps` is an object-control verb, the reflexives in `vps` are forced to take the agreement of the added nominal object.[91] The auxiliary operation `insertObjNP` inserts `np.s` with preposition or case `vps.c2` into the `nn`-field of `vps`.

The operation `objAgr np vp` instantiates `vp.nn` and `vp.inf` fields to agreement `np.a`. So `objAgr` implements the following ***method to resolve reflexive pronouns***. For a verbal phrase `vp:VP` or an incomplete verbal phrase `vp:VPSlash`, the nominal objects in `vp.nn` and `vp.inf` depend on agreement features `a : Agr`. When a clause is built from `vp` by `(PredVP np vp)`, we bind the reflexives to this subject `np` by letting the nominal objects be constantly `vp.nn!(np.a)`, i.e. replace `vp` by

```
vp ** {nn = \\a => vp.nn ! np.a}
```

before using the `vp` in finite form as predicate of the clause; we proceed similarly with its infinitival object `vp.inf`. For a clause `(PredVPSC sc vp)` with sentential subject, instead of `np.a` we use agreement with third person singular. So to speak, all (unbound) reflexives in `vp:VP` are viewed as subject-controlled. If a nominal complement `np` is added to `vp:VPSlash` in `(ComplSlash np vp)`, then, as shown above, if `vp.objCtrl = True` we bind the reflexives in `vp.nn` to this nominal object `np` by instantiating `vp.nn` with `np.a`, otherwise leave them unbound, i.e. subject-controlled.[92] To build an infinitive from `vp` with unknown implicit subject, e.g. to be used as an infinitival subject as in "*to blow one's nose in the tablecloth is unpolite*", we can just instantiate reflexives to third[93] person singular.

The rule to build a verb phrase by using an incomplete verb phrase reflexively, i.e.

```
ReflVP   : VPSlash -> VP ;          -- love himself
```

generalizes the reflexive usage of binary verbs. It is implemented by inserting a reflexive pronoun as nominal or prepositional object according to the case or preposition in `vp.c2`:

```
ReflVP vp = insertObjRefl vp ; -- HL, 19/06/2019
```

[91] Todo 38: shouldn't this instantiation be done by `insertObjNP`, so that it applies as well to `SlashV2VNP` in *beg me to buy* in *to beg sb to love his (own!) neighbours*? Would the default `objCtrl=False` conflict with `insertObjNP` used in `Slash?V3`'s to insert an object to `predVc v:V3`? Or should this be reserved to reflexive noun phrases `RNP` of `Extra/Extend`? Are and should reflexive possessives in `vps` also be instantiated to `np.a`?

[92] Rethink this when adding a reflexive noun phrase `rnp`: *to promise sb. to wash one's car* vs. *to ask sb. to wash his (own) car*, so the `rnp` needs its own agreement `rnp.a`, c.f. *I promised*| *asked my wife to wash my*| *her hair.*

[93] or rather, the agreement features of the indefinite personal pronoun *man* (to be added). Maybe `objAgr` should differ between instantiating `vp.nn` and `vp.inf`: object-control is about the implicit subject of `vps.inf`; at least if the `vps:VPSlash` is obtained by adding complements to verbs of arity $n \geq 3$, the `vps.nn` should perhaps not be specialized by `objAgr`?

The auxiliary operation inserts a pure reflexive pronoun in field `vp.nn.p1`, and a reflexive pronoun with preposition in field `vp.nn.p2`:

```
insertObjRefl : VPSlash -> VPSlash = \vp ->
  let prep = vp.c2 ;
      obj : Agr => Str = \\a => prep.s ! CPl ++ reflPron ! a ! prep.c ++ prep.s2
  in vp ** {
    nn = \\a =>
      let vpnn = vp.nn ! a in
      case prep.t of {
        isCase => <obj ! a ++ vpnn.p1, vpnn.p2, vpnn.p3, vpnn.p4> ;
        _      => <vpnn.p1, obj ! a ++ vpnn.p2, vpnn.p3, vpnn.p4> }
  } ;
```

When a clause is formed by applying `mkClause np vp` (p. 140) to the resulting verb phrase, the objects in `vp.nn.p1` are placed in front of the negation adverb *nicht*, those in `vp.nn.p2` after the negation adverb.

Remark 75. *In `LangGer`, a reflexive verb prhase `ReflVP vps` can be part of a generalized clause, `GenericCl (ReflVP vps)`, i.e. the general subject "man" is correctly referred to by the reflexive pronoun "sich". But it is wrong in `LangEng`, where "oneself" ought to be used:*

```
TestLang> l GenericCl (ReflVP (SlashV2a know_V2))
one knows itself
man kennt sich
```

`ReflVP` does not allow to form infinitives with reflexive possessives, like "man soll seine Angelegenheiten selber regeln". For this, use `ReflPoss`,`ReflRNP` of `ExtraGer`, see p. 175.

Copula Verbs Of the two constructions of verb phrases from copula verbs, i.e.

```
UseCopula : VP ;                    -- be
UseComp   : Comp -> VP ;            -- be warm
```

the first is implemented by the default verb phrase spanned by the copula verb *sein*:[94]

```
UseCopula = predV sein_V ;
```

Q52: Is the default `isAux = False` correct for `UseCopula`, or do we need `predVGen True sein_V`?

The second construction first builds the initial verb phrase `vp` spanned by *sein* and then inserts the inflection paradigm `comp.s` of the complement into the object field `vp.nn.p4` and the extracted part `comp.ext` into the field `vp.ext`:

[94] It is unclear why this deserves a special construction, but other copula verbs *bleiben* or *werden* don't. True, we can say *hier sein—bleiben*, but not *hier werden*, but this is not a strong reason to exclude *bleiben* and *werden* from being copula verbs.

```
UseComp comp =
   insertExtrapos comp.ext (
     insertObj comp.s (predV sein_V)) ; -- agr not used
```

Remark 76: Apparently, Scharolta added the field `vp.adj` for adjectival complements of verbs, as the example *ich finde dich schön* showed. The complement of a copula verb is inserted in `vp.nn.p4`. Todo 39: check if the pre-adjective complement `ap.c.p1` of an adjectival complement `ap` of `ComplVA va ap` inserted by `insertAdj` into the `vp.nn.p2`-field is correct. Can there be a problem, as we are talking about different verb phrases `vp`, i.e. `predV sein_V` and `predV va`?

Copula-preceded Complements

The four constructions to build complements to copula verbs,

```
CompAP   : AP  -> Comp ;            -- (be) small
CompNP   : NP  -> Comp ;            -- (be) the man
CompAdv  : Adv -> Comp ;            -- (be) here
CompCN   : CN  -> Comp ;            -- (be) a man/men
```

are implemented by filling the fields `comp.s` and `comp.ext` of the resulting `comp:Comp` by suitable values. In `CompAdv a`, the adverb is inserted in `comp.s` and the empty string in `comp.ext`:

```
CompAdv a = {s = \\_ => a.s ; ext = []} ;
```

In `CompAP ap = comp`, the predicative form of an adjective phrase with its pre- and post-adjective complements and the comparision part is inserted in `comp.s` and `ap.ext` in `comp.ext`

```
CompAP ap = {s = \\_ => ap.c.p1 ++ ap.s ! APred ++ ap.c.p2 ++ ap.s2 ! Nom ;
             ext = ap.ext} ;
```

In `CompNP np`, the predicatively used noun phrase `np` in nominative case is inserted in `comp.s`, together with its relative clause `np.rc`:

```
CompNP np = {s = \\_ => np.s ! False ! Nom ++ np.rc ; ext = np.ext} ;
```

Similarly with `CompCN cn`, where the predicative usage needs an indefinite article in the singular and a strong adjective inflection:

```
CompCN cn = {s = \\a => case numberAgr a of {
                            Sg => "ein" + pronEnding ! GSg cn.g ! Nom ++
                                  cn.s ! Strong ! Sg ! Nom ++ cn.rc ! Sg ;
                            Pl => cn.s ! Strong ! Pl ! Nom ++ cn.rc ! Pl
                          } ;
             ext = cn.adv ++ cn.ext
  } ;
```

The resulting inflection tables `comp.s : Agr => Str` are constant, except for `CompCN cn`. Hence reflexives in the verbal phrases built from `comp` are excluded, e.g. *ist in seinen besten Jahren*, *ist sich treu*, *ist klüger als sein Bruder*, *ist die Hoffnung seines Vereins*, *ist ein geachteter Bürger seiner Heimatstadt*, *ist sein eigener Herr*.

Todo 40: This should be fixed by changing `AP.c : Str * Str` to `AP.c : Agr => Str * Str`.

Passive Constructions

```
-- Passivization of two-place verbs is another way to use them. In many
-- languages, the result is a participle that is used as complement to a
-- copula.

    PassV2   : V2 -> VP ;               -- be loved

-- *Note*. the rule can be overgenerating, since the V2 need not take a
-- direct object.
```

The construction `PassV2 v` is implemented by modifying the initial verb phrase `vp` spanned by the passive auxiliary verb *werden*. The predicatively used past partiple `v.s ! VPastPart APred` is inserted into the field `vp.nn.p4` for the complement of copula verbs, and the subject case is set depending on `v.c2`:

```
    PassV2 v = -- acc object -> nom subject; all others: same Case
      let vp = predV werdenPass in
        insertObj (\\_ => v.s ! VPastPart APred) vp
        ** { c1 = subjPrep v.c2 } ;
```

If `v:V2` expects a nominal object in accusative, the default case for the subject, `vp.c1`, is changed to nominative; if `v` expects a nominal object in some other case, or a prepositional object, the corresponding value `v.c2` is used as subject case `vp.c1`. This is done by the auxiliary operation

```
  subjPrep : Preposition -> Preposition = \prep ->
    case <prep.c,prep.t> of {
      <Acc,isCase> => prep ** {c = Nom} ;
      _ => prep
    } ;
```

Q53: What about the difference between *getan werden* and *getan sein*? Some other forms of passive are implemented in `TestLangGer`.

Remark 77. *`PassV2` is generalized to `PassVPSlash : VPSlash -> VP` in `ExtraGer`, which does not work properly if the `vps` is obtained from ternary verbs. `PassVPSlash vps` is incorrect for `vps = Slash2V3 v[c2:acc,c3:dat] np.acc`: we get "*ihr.dat wird einen.acc Brief geschickt" instead of "sie.nom bekommt einen.acc Brief geschickt". But: "ihr.dat wird mißtraut:V2[dat]", not "*sie.nom bekommt mißtraut". That is, `PassVPSlash vps` needs to know whether its argument `vps` is built from a `V3[acc,dat]` or a `V2[dat]`; this would need an inspection of the abstract tree of the argument `vps`, which is impossible in `GF`. May we should not passivize predicates (`VPSlash`), but only verbs (`V2`, and separately `V3`) – `PassVPSlash` is too general and thereby wrong. But `GF` has no notion of transitive verb, i.e. verb of category `V2` that can be passivized.*

Constructions of `VPSlash`

The basic constructions of an incomplete verb phrase are to use a binary verb or to combine a ternary verb with one complement of the expected category.

```
    SlashV2a : V2        -> VPSlash ;  -- love (it)
    Slash2V3 : V3  -> NP -> VPSlash ;  -- give it (to her)
    Slash3V3 : V3  -> NP -> VPSlash ;  -- give (it) to her
```

These are easily implemented by inserting the complement in the appropriate field of the record `vps = (predVc v):VPSlash` opend by the verb `v`, where

```
predVc : Verb ** {c2 : Preposition} -> VPSlash = \v ->
  predV v ** {c2 = v.c2 ; objCtrl = False} ;
```

fills all fields of `(predV v):VP` and then adds the preposition `v.c2` and a default that the embedded verb is not an object-control verb:

```
SlashV2a v = (predVc v) ;

Slash2V3 v np = insertObjNP np v.c2 (predVc v) ** {c2 = v.c3} ;
Slash3V3 v np = insertObjNP np v.c3 (predVc v) ;
```

Notice that, by convention, `c2` should be used to combine the verb with its direct, `c3` to combine it with its indirect nominal object.[95] (Q54: What does this tell about the linear order of objects? `insertObjNP` adds pronouns to the left, other nominal or prepositional objects to the right (of the corresponding `nn`-component). So we should not have equivalent linearizations of the trees

```
ComplSlash (Slash3V3 v np3) np2 == ComplSlash (Slash2V3 np2) np3
```

but we do! Todo 41: check again, and see how `mkClause` orders its objects.)

For ternary verbs with a non-nominal complement, an incomplete verb phrase is obtained by combining the non-nominal complement to the verb by the rules

```
SlashV2V : V2V -> VP -> VPSlash ;  -- beg (her) to go
SlashV2S : V2S -> S  -> VPSlash ;  -- answer (to him) that it is good
SlashV2Q : V2Q -> QS -> VPSlash ;  -- ask (him) who came
SlashV2A : V2A -> AP -> VPSlash ;  -- paint (it) red
```

The implementations of these are easy for sentential, interrogative and adjectival complements, which are inserted into `(predV v)` as with `ComplVS`, `ComplVQ`, `ComplVA`, and then the field for `c2` is filled by `v.c2` and the one for `objCtrl` by a default:

```
SlashV2S v s =
  insertExtrapos (comma ++ conjThat ++ s.s ! Sub) (predV v)
                                          ** {c2 = v.c2; objCtrl = False} ;
SlashV2Q v q =
  insertExtrapos (comma ++ q.s ! QIndir) (predV v)
                                          ** {c2 = v.c2; objCtrl = False} ;
SlashV2A v ap =
  insertAdj (ap.s ! APred) ap.c ap.ext (predV v)
                                          ** {c2 = v.c2; objCtrl = False} ;
```

The implementation of `SlashV2V` is a simple extension of `ComplVV`:

```
SlashV2V v vp = -- (jmdn) bitten, sich zu waschen | sich waschen lassen
  ComplVV v vp ** {c2 = v.c2 ; objCtrl = v.objCtrl} ;
```

[95] For correct translation between ternary verbs, this convention is essential.

An application of `SlashV2V`, just like `ComplVV`, uses `infVP` to turn its argument `vp:VP` into the infinitival complement of its argument `v:V2V` and inserts it into the `inf`-field of the verb phrase `vps = (predVGen v.isAux v)` opened by `v`; it then adds `v.c2` and `v.objCtrl` to give the resulting partial verb phrase `rvp := (SlashV2V v vp):VPSlash`. Since `v:V2V` is a control verb, reflexives in `rvp.inf` depend on its missing subject or nominal object, so we have to remember `v.objCtrl` in `rvp` to be able to instantiate reflexives properly when an object noun phrase is added to `rvp` by `ComplSlash`. (The argument-`vp` may itself be reflexive, e.g. `vpi.inf` = *sich vornehmen, etwas zu tun.*)[96]

Reflexive resolution works with this in `Ger` (`Eng` considers `v:V2V` to be an object-control verb):

```
TestLang> gr -number=4 (PredVP (UsePron i_Pron)
         (ComplSlash (SlashV2V ? (ReflVP (SlashV2a wash_V2))) (UsePron ?))) | l
I let me wash myself
ich lasse mich mich selbst waschen
I warn it to wash itself
ich warne es , sich selbst zu waschen
I promise us to wash ourselves                -- wrong
ich verspreche uns , mich selbst zu waschen
I let her wash herself
ich lasse sie sich selbst waschen
```

Due to a mistake in `SlashVP`, reflexive resolution doesn't work in relative clauses (p. 145). (Also, `SelfNP`, `SelfAdVVP` and `SelfAdvVP` give non-reflexive readings with *selbst*. To highlight the difference between personal and reflexive pronoun, `ReflVP` uses `ResGer.reflPronSelf`, which adds *selbst* to all forms of `ResGer.reflPron`.)

See the extension of `Extend.RNP` to reflexive (incomplete) predicates in `ReflGer`|`Eng`.

But: "reflexive resolution" cannot be as simple as implemented by `objCtrl` and `objAgr`: in *I:NP advise (my brother/sister):RNP to help (him/her)self:ReflPron!rnp.a and (my:PossPron!np.a child)*, the infinitival complement of `advise:V2V` refers to the object `rnp:RNP` as its implicit subject and to the subject `np:NP` as referent of the possessive. The infinitival complement of a verb `v:V2V` can depend on the two agreement values `np.a` and `rnp.a` of the subject and object of the main verb `v:V2V`. (8/23)

Incomplete verb phrases can also be obtained by adding an incomplete infinitival complement to a verb `v:VV` or `v:V2V` by the rules

```
SlashVV    : VV  -> VPSlash -> VPSlash ;       -- want to buy
SlashV2VNP : V2V -> NP -> VPSlash -> VPSlash ; -- beg me to buy
```

Applications of both rules have to turn their argument `vp:VPSlash` into an *incomplete* infinitival complement of their argument `v:VV`. This can be done using the same operation `vpi = infVP v.isAux vp`, because the `vp` just has no nominal `c2`-object under `vp.nn` (i.e. an empty string).

An ad-hoc implementation of `SlashVV` uses `ComplVV` and adds the `c2` and `objCtrl` fields of its argument `vp:VPSlash` to the result `rvp:VPSlash`.

[96] We may have `vp.s.vtype = VRefl c`, but generally, a "reflexive" `vp:VP` (built with `ReflVP : VPSlash -> VP`) just has a `reflPron` inserted in `vp.nn.p1`, so we can't check if it is a reflexive `vp`, unless we add a field `vp.vtype` or can check if `vp.nn.p1` is empty. Or can we use reflexive noun phrases `Extra.RNP`, which are inserted to `nn.p4`, or misuse `vp.s.vtype`?.

```
SlashVV v vp =
  ComplVV v vp ** {c2 = vp.c2 ; objCtrl = vp.objCtrl} ;
```

The idea is that adding a nominal object `np:NP` to this resulting partial verb phrase gives the same as adding `np` as a complement to the argument `vp:VPSlash`, i.e. that

```
(ComplSlash (SlashVV v vp) np) == (ComplVV v (ComplSlash vp np))     (*)
```

are equivalent in the sense that these trees have the same linearizations. In `Eng`, `Romance`, and perhaps other languages, this seems to be the case, even for iterated uses of `SlashVV`, since an object added to the innermost `vp:VPSlash` is rightmost in the linearization and hence can be added likewise to the topmost partial verb phrase:

(want to dare to try to read) (the book) = want to dare to try to (read the book)

The topmost partial verb phrase can both be completed to first a verp phrase and then a clause, or to first an incomplete clause and then a relative clause:

(we:NP ((want to dare to try to read):VPSlash (the book):NP):VP):Cl,
(which:RP (we:NP ((want to dare to try to read):VPSlash):ClSlash)):RCl.

For `Ger`, the ad-hoc implementation of `SlashVV` doesn't work well, since an *Inf-zu*-complement is ususally extracted, but less so when its nominal object is missing: *(ich) will (nicht) wagen, das Buch zu lesen*, but rather *das Buch, das ich (nicht) zu lesen wagen will*, than *das Buch, das ich (nicht) wagen will, zu lesen*

Complementizing the innermost partial verb phrase *zu lesen* first to the verb phrase *das Buch zu lesen* and then using `ComplVV` iteratively, we obtain

(wir:NP (wollen (wagen (, zu versuchen (, das Buch zu lesen):VP):VP):VP):VP):Cl.

If we can use `SlashVV` iteratively to the partial verb phrase *zu lesen*, there is certainly no extraction of partial *Inf-zu* complements involved, but we would rather get

(((zu lesen):VPSlash zu versuchen):VPSlash wagen):VPSlash.

One can then both complete this first to a verb phrase and then to a clause, as well as complete it first to an incomplete clause and then to a relative clause:

(wir:NP (wollen ((das Buch):NP (zu lesen zu versuchen wagen):VPSlash):VP):VP):Cl
(das:RP (wir:NP (zu lesen zu versuchen wagen):VPSlash):ClSlash):RCl

It follows that for `Ger`, the equivalence `(*)` does not hold: applications of `SlashVV` combine partial verb phrases in-place, while applications of `ComplVV` combine *Inf-zu* complements by extractions, e.g. *(ich) habe das Buch (zu kaufen gewagt)* versus *(ich) habe gewagt(, das Buch zu kaufen)*.[97]

[97] So, when is an implementation of `VPSlash` correct? When a `vp:VPSlash` can be completed to a verb phrase (by `ComplSlash`), or to an incomplete clause (by `SlashVP` and `RelSlash`), or when it behaves "well" as top-level construct?

Question 55. *But if the two trees in* `(*)` *linearize differently, how to know which of the constructions should be used to yield which word order in which language? It is not true that the different trees have the same linearization in* `Eng`*, as can be seen from*

```
TestLang> p -lang=Ger -cat=Cl -tr
            "ich will nichts zu meinem Buch hinzufügen müssen" | l -lang=Eng
PredVP (UsePron i_Pron)
  (ComplVV want_VV (ComplVV must_VV (ComplSlash
     (Slash2V3 add_V3 nothing_NP)
        (DetCN (DetQuant (PossPron i_Pron) NumSg) (UseN book_N)))))
PredVP (UsePron i_Pron)
  (ComplVV want_VV (ComplSlash (SlashVV must_VV
     (Slash3V3 add_V3 (DetCN (DetQuant (PossPron i_Pron) NumSg) (UseN book_N)))
          ) nothing_NP))
PredVP (UsePron i_Pron)
  (ComplSlash (SlashVV want_VV (SlashVV must_VV
     (Slash3V3 add_V3 (DetCN (DetQuant (PossPron i_Pron) NumSg) (UseN book_N)))
          )) nothing_NP)

I want to have to add nothing to my book
I want to have to nothing add to my book
I want nothing to have to add to my book
```

The three trees have the same linearization in `Ger`*, because of the separation of objects in the four components of the* `VP.nn`*-field, it seems.*

Can we make `(SlashVV wagen_VV (SlashV2a read_V2))` be a `vp:VPSlash` with `vp.inf.inpl` = *zu lesen wagen* and not extract this under `must_VV`? In the default implementation, `(SlashVV wagen_VV lesen)` embeds "*, zu lesen*" in `vp.inf.extr`. Does an embedding without comma in `vp.inf.inpl` give a correction, which handles nested `SlashVV` properly? Incomplete trial:

```
    SlashVV v vp =
      let
        vps = predVGen v.isAux v ; -- e.g. will.isAux=True|wage.isAux=False
        vpi = infVPSlash v.isAux Simul Pos vp ;
        inf : {inpl: (Agr => Str) * Str ; extr : (Agr => Str)} =
          let
            topInpl = <vpi.objs, vpi.pred> ;
            emptyInpl : (Agr => Str) * Str = <\\_ => [], []> ;
            glue : (Agr => Str)*Str -> (Agr => Str) =
                   \i -> \\agr => (i.p1!agr ++ i.p2) ;
          in
          case <v.isAux,vp.isAux> of {
            <True,_   >  -- 1. will lesen können | 2. will zu lesen wagen
                => {inpl = embedInf vpi.inpl topInpl ;
                    extr = vpi.extr} ;
            <False,True> -- 3. wagt lesen zu wollen
                => {inpl = emptyInpl ;
                    extr = let moved = (embedInf vpi.inpl topInpl)
```

```
                        in \\agr =>
                           (comma ++ (glue moved)!agr ++ (vpi.extr!agr))} ;
          <False,False> -- 4. wagt zu lesen zu versuchen
              => {inpl = embedInf vpi.inpl topInpl ;
                  extr = vpi.extr} } ;
      in
      insertExtrapos vp.ext
        (insertInf inf vps) ** {c2 = vp.c2 ; objCtrl = vp.objCtrl};
```

We use a slight modification `infVPSlash` of `infVP` to build the infinitive *zu lesen wagen* from the partial verb phrase *wage zu lesen*, instead of *wagen, zu lesen*. We can then iteratively build inplace nested infinitives of partial verb phrases followed by adding a nominal complement, i.e.

```
  (ComplSlash (SlashVV vn (...  (SlashVV v1 vp) ...)) np),
```

or add the nominal complement to the innermost partial verb phrase and iteratively extract nested infinitives of verb phrases, i.e.

```
  (ComplVV vn ( ... (ComplVV v1 (ComplSlash vp np)) ...)),
```

but cannot add a nominal complement in between, as for `ComplVV o ComplSlash o SlashVV`:

```
TestLang> p -tr -cat=Cl -lang=Eng "I must dare to read the book" | l -lang=Ger
PredVP (UsePron i_Pron)
  (ComplSlash (SlashVV must_VV (SlashVV wagen_VV (SlashV2a read_V2)))
              (DetCN (DetQuant DefArt NumSg) (UseN book_N)))
PredVP (UsePron i_Pron)
  (ComplVV must_VV (ComplSlash (SlashVV wagen_VV (SlashV2a read_V2))
                               (DetCN (DetQuant DefArt NumSg) (UseN book_N))))
PredVP (UsePron i_Pron)
  (ComplVV must_VV (ComplVV wagen_VV (ComplSlash (SlashV2a read_V2)
                               (DetCN (DetQuant DefArt NumSg) (UseN book_N)))))

ich muss das Buch zu lesen wagen
ich muss das Buch wagen zu lesen   -- wrong
ich muss wagen , das Buch zu lesen
```

The reason is that `ComplSlash` inserts a nominal object at top-level, while the object is missing in an embedded infinitival, but we can't know how deeply embedded.

Finally, the construction of incomplete verb phrases by[98]

```
    SlashV2VNP : V2V -> NP -> VPSlash -> VPSlash ; -- beg me to buy
```

can preliminarily be implemented by

```
 SlashV2VNP v np vp = -- jmdn bitten zu kaufen | jmdn kaufen lassen
   insertObjNP np v.c2 (ComplVV v vp ** {c2 = v.c2 ; objCtrl = v.objCtrl}) ;
```

[98]See Problem **??**, p. **??**, on the effect of `VPSlash.objCtrl:Bool` and `VPSlash.c1|c2:Preposition` on the complexity of compilation of `SlashV2VNP`.

As intended, this gives *(dich waschen lasse):VPSlash* with auxiliary *lassen*:V2V in

```
TestLang> l (RelSlash IdRP (SlashVP (UsePron i_Pron)
               (SlashV2VNP lassen_V2V (UsePron youSg_Pron) (SlashV2a wash_V2))))
that I let you wash
den ich dich waschen lasse
```

but it doesn't embed the infinitival complement properly in Ger:

```
TestLang> l  (RelSlash IdRP (SlashVP (UsePron i_Pron)
                  (SlashV2VNP beg_V2V (UsePron youSg_Pron) (SlashV2a wash_V2))))
that I beg you to wash
den ich dich bitte , zu waschen
```

instead of *den zu waschen ich dich bitte*. Using SlashVV instead of ComplVV to obtain an in-place infinitival complement causes a memory problem. (In Eng, the definitions are

```
    SlashVV vv vp = vp **
      insertObj (\\a => infVP vv.typ vp False Simul CPos a) (predVV vv) ;
    SlashV2VNP vv np vp = vp **
      insertObjPre (\\_ => vv.c2 ++ np.s ! NPAcc)
        (insertObjc (\\a => vv.c3 ++ infVP vv.typ vp False Simul CPos a)
                    (predVc vv)) ;
```

The infinitive is inserted as string into the s2:Str field for complements of Eng.VP.

And it gets reflexive resolution and the object order wrong under SlashVP

```
TestLang> l  (PredVP (UsePron he_Pron)
    (ComplSlash (SlashV2VNP beg_V2V (UsePron i_Pron) (SlashV2a listen_V2))
                (UsePron we_Pron)))
he begs me to listen to us
er bittet mich uns , zuzuhören

TestLang> l DetCN (DetQuant DefArt NumSg) (RelCN (UseN woman_N)
  (UseRCl (TTAnt TPres ASimul) PPos (RelSlash IdRP
     (SlashVP (UsePron they_Pron)
         (SlashV2VNP beg_V2V (UsePron i_Pron) (SlashV2a listen_V2))))))
the woman that they beg me to listen to
die Frau , die sie mich bitten , zuzuhören    ==>..., der sie mich bitten, ...
```

Without SlashVP: *er bittet mich, auf ihn|*sich|mich zu hören.* The object order is wrong in Eng as well under ReflVP:

```
TestLang> l (PredVP (UsePron he_Pron)
   (ReflVP (SlashV2VNP beg_V2V (UsePron i_Pron) (SlashV2a listen_V2))))
he begs to himself me to listen
```

Remark 78: (SlashV2VNP beg_V2V (UsePron he_Pron) (SlashV2a listen_V2)) = *bitte ihn , (jmdm) zuzuhören* should be made to work for relative clauses like *die Frau , der zuzuhören*

ich ihn bitte. But `Eng` *the woman whom I (beg him to listen to)* is simpler: `v2v np vps` can be used to build a clause under `np ((v2v np vps) np):VP` using `ComplSlash`, or to build a relative clause under `RP np (v2v np vps)` under `RelSlash` and `SlashVP`.

```
TestLang> l (UseCl (TTAnt TPres ASimul) PPos (PredVP (UsePron i_Pron)
     (ComplSlash (SlashV2VNP beg_V2V (UsePron he_Pron) (SlashV2a listen_V2))
                 (UsePron she_Pron))))
I beg him to listen to her
ich bitte ihn ihr , zuzuhören
```

We want to get: *die Frau , der er mich bittet zuzuhören*, just like *das Haus , das er (mich bittet zu kaufen):VPSlash*

Q56: Can we implement ACI as lexical transformation `sehen:VS` ↦ `sehen:V2V`? ...

Modification of Verb Phrases

According to `Verb.gf`, `AdvVP` is to add adverbs at the end of a `vp`, while `AdVVP` is to attach them next to or before the verb.

```
-- Adverbs can be added to verb phrases. Many languages make a distinction
-- between adverbs that are attached at the end vs. next to (or before) the
-- verb.

    AdvVP    : VP -> Adv -> VP ;        -- sleep here
    ExtAdvVP : VP -> Adv -> VP ;        -- sleep , even though ...
    AdVVP    : AdV -> VP -> VP ;        -- always sleep
```

An adverbial clause should be added by `ExtAdvVP`, so that it is embedded in commata:

```
    ExtAdvVP vp adv = insertAdv (embedInCommas adv.s) vp ;
```

The auxiliary operation `insertAdv` (p. 120) adds `adv.s` to the right end of `vp.a2`. But since `Adverb.gf` (p. 29) does not make a distinction between lexical adverbs and adverbial clauses[99], we cannot restrict `adv` to adverbial clauses in `ExtAdvVP vp adv` or to lexical adverbs in the other rules.

The difference between adverbs occurring in front of the verb and those following the verb (or the negation adverb, as in `gf-3.2` of `LangGer`), suggesting the other two rules `AdVVP` and `AdvVP`, does not seem to exist in German, so these both insert `adv` to `vp.a2` by `insertAdv`:

```
    AdvVP vp adv = insertAdv adv.s vp ;
    AdVVP adv vp = insertAdv adv.s vp ; -- not AdV 27/5/2012: nicht immer
```

Remark 79: In simple clauses in main verb order and unary verb, the adverb follows the verb and precedes the sentence negation: *wir arbeiten heute – wir arbeiten heute nicht*. But besides sentence negation, there is an adverb negation: *wir arbeiten nicht heute (, sondern morgen)*, or *wir arbeiten nicht gern*, e.g. *wir arbeiten ungern*. Is this different with adverbs like *immer*, *oft*, which involve quantification over time and hence relate to the tense of the verb form? *wir*

[99] it only makes a distinction between adverbs `Adv` and comparison adverbs `CAdv`

arbeiten manchmal nicht = wir arbeiten (nicht immer), or *wir arbeiten oft nicht = Oft arbeiten wir nicht* $\neq$ *wir arbeiten nicht oft = wir arbeiten selten*, but **wir arbeiten nicht manchmal*, only: *wir arbeiten manchmal nicht = manchmal (arbeiten wir nicht)*? Is it similar with adverbs quantifying over positions where an action can take place? *hier (arbeiten wir nicht)* vs. *(hier arbeiten wir) nicht*? Is this really clearer in English? *here, (we don't work)* vs. *we don't (work here)*

Remark 80. *The word order in sentences is the one of the corresponding clause, fixed by* `mkClause` *and* `UseCl` *below (p. 140 and 142; also* `TestLangGer.mkClSlash ExtraGer.mkVPS`*). But* `mkClause` *gets the position of adverbs wrong in inverted and subordinate clauses: the adverbs* `vp.a2` *should follow the subject* `subj`*, not be at the end of* `obj3` *(c.f. Remark 84).*

1. *The difference between preverbal and postverbal adverbs does not seem to exist in German: all adverbs are postverbal "John doesn't sleep here|often|today = Johann schläft hier|oft|heute nicht". (This is sentence negation* `np doesn't v adv = neg (np does (v adv))`*, not verb phrase negation* `np does neg(v adv)`*, it seems.)*
   If this is true, then with unary verbs, adverbs should appear in front of negation: "Johann schläft hier nicht", "Johann wird hier nicht schlafen". The different "Johann schläft heute (nicht hier)" would then be a kind of adverbial negation: "nicht hier = anderswo".
   Do the English preverbal adverbs correspond to a different scope? "John doesn't always sleep =? John (does not always) sleep = (not always) does John sleep = Johann schläft (nicht immer)".[100]
   In some cases, the "standard"(?) negation is expressed by a different adverb: "Johann schläft immer nicht = Johann schläft nie". But for which adverbs do we have a contracted (and strongly preferred) form like "immer nicht = nie"?
2. `AdvVP (become red) today`*: "ich bin nicht rot heute geworden" works with* `pfin ++ neg ++ nn4 ++ advs ++ pinf`*, but "ich bin heute nicht rot geworden" with* `pfin ++ advs ++ neg ++ nn4` *does not. Likewise "ich werde nicht bereit heute sein" instead of "ich werde heute nicht bereit sein".*
   This concerns `AdvVP vp adv` *for verb phrases of the form* `(ComplVA v ap)`*,* `(UseComp (CompAP ap))`
3. *For verb phrases of the form* `ComplSlash (SlashV2a v) np`*, the adverb should follow the object, "ich hatte sie heute nicht gewaschen" instead of "ich hatte sie nicht heute gewaschen", i.e. we need* `subj ++ pfin ++ nn1 ++ adv ++ neg ++ ...++ pinf` *instead of* `subj ++ pfin ++ nn1 ++ neg ++ (nn3 ++ nn4 ++ adv) ++ pinf`
4. *Can we simply put the adverbs in* `vp.a2` *before* `neg`*, even if other objects follow the negation? With light objects: "ich habe dir das Buch heute nicht geschickt" and with heavy objects: "ich habe dir heute nicht ein Buch geschickt" = "ich habe dir heute kein Buch geschickt". Looks good. Is the order then*

   ```
   Main => subj ++ pfin ++ light ++ adv ++ neg ++ heavy ++ comp ++ pinf
   ```

   But then adverbial complements *have to be put to* `comp` *(i.e.* `nn4`*): "sie wohnt nicht in Berlin", not "sie wohnt in Berlin nicht".*
   Remark 81: in `gf-3.3` *the order was*

   ```
   Main => subj ++ verb.fin ++ compl ++ inf ++ extra
   ```

[100] And what happens with modal verbs: "*kann nicht immer schlafen*" – "*(kann nicht) (immer schlafen)*" or "*(kann (nicht immer)) schlafen*"? Is this resolved by intonation?

with `compl = obj0 ++ vp.a1!pol ++ obj ++ vp.a2`*, where* `vp.a1 = pre-verbal advs ++ neg` *and* `obj0 = pronouns`*,* `obj = non-pron objects`*. This doesn't place the adverbs correctly either, I think. Do some tests with* `tests/german/vpadv.trees`*.*

5. *Should we use* `vp.a1` *for post-verbal (post-*`vfin`*), and post-negation adverbs (*`AdV`*), and* `vp.a2` *for post-verbal, pre-negation adverbs* `Adv`*? Does such a distinction exist in German? Q57: Can there be a correct translation between languages that make the difference between* `Adv` *and* `AdV` *and those that don't (likewise if it is just a difference in adverb positioning, not in adverb kind)?*

Examples for adverb ordering (from journal der Freitag Nr48, S.5/6, Nov.2023)

1. *Es gibt bis heute kein ausgereiftes technisches Verfahren, das flächendeckend und langfristig garantieren könnte, dass das CO_2 für immer unter der Erde bleibt. (*`adv.tmp < adv.loc`*)*
2. *Kurz davor hatte es dort lange und heftig geregnet. (*`adv.tmp < adv.loc < adv.mod`*)*
3. *Seine Vorstellung war, dass die schwere Lohnarbeit irgendwann vollständig durch Technisierung ersetzt werden könnte. (*`adv.tmp < adv.mod`*)*
4. *Er wird 1930 in Sachsen-Anhalt als Sohn einer jüdischen Mutter und eines evangelischen Pfarrers geboren. (*`adv.tmp < adv.loc < adv.mod`*)*
5. *Kurze Zeit später sind sie auch dort nicht mehr sicher. (*`adv.tmp < adv.loc < adv.neg`*)*

There also is a construction `AdvS` resp. `ExtAdvS` to add an adverb resp. an adverbial sentence to the front of a sentence (p. 143); a rule `SSubjS : S -> Subj -> S -> S` to add a subjunctive clause at the end of a sentence. This must lead to multiple analyses when the adverbial clause is at the end (in main verb order), like

```
SSubjS (PredVP np vp) subj s = PredVP np (AdvVP vp (SubjS subj s))
```

In any case, an adverbial modification of verb phrases is needed, e.g. to build adverbially modified infinitival complements: *we recommend you to always be polite.*

5.4.2. Clauses and Sentences

Categories of Clauses and Sentences

The implementation type of clauses consists of a field `s` for an inflection paradigm.

```
lincat Cl =
  {s : Mood => ResGer.Tense => Anteriority => Polarity => Order => Str} ;
```

The parameter types `Mood`, `Tense`[101], `Anteriority`, `Polarity` and `Order` come with the values

```
param
  Mood        = MIndic | MConjunct ;
  Tense       = Pres | Past | Fut | Cond ;
  Anteriority = Simul | Anter ;
  Polarity    = Pos | Neg ;
  Order       = Main | Inv | Sub ;
```

[101] The parameter type `Tense = ResGer.Tense` is not the syntactic category `cat Tense` in Section 5.10.

The *mood* has a value `MIndic` for *indicative mood* and a value `MConjunct` for *conjunctive mood*, which shows at the form of the finite main verb. In *tense*, three values for *present tense*, *past tense* and *future tense* are distinguished, and a value `Cond` for *conditional*. The clause tense is realized by the present or imperfect form of its finite main verb, or by combining a finite form of a temporal auxiliary verb *sein*, *haben* or *werden* with an infinite form of the main verb. The *anteriority* expresses relative temporal order between the tense of the clause and the event mentioned; for each tense there, these are either *simultaneous* or *anterior*, in which case the event mentioned is past relative to the tense level of the clause.

Polarity indicates the absence or presence of the negation adverb *nicht*: a clause without the negation adverb has *positive polarity* `Pos`, a clause with the negation adverb has *negative polarity*. (Polarity does not fully correspond to the difference between atomic and negated atomic formulas in logic: in natural languages, negation may be incorporated in quantifiers like *kein* (eng. *no*), and then does not influence polarity. *wir trinken kein Bier* has positive polarity, but *Bier trinken wir nicht* has negative polarity. So, it seems unclear what use should be made of the positive polarity of *we don't drink beer* when translating this sentence from English to German.)

The *order* refers to the position of the finite verb: in main clauses, with order `Main`, the finite verb follows the first, clause-initial complement (or adverb), in subordinate clauses, with order `Sub`, the finite verb comes at the end, and in questions, at the beginning, i.e. the order is inverted. For example, of the possible 96 forms of the simple clause *wir gehen*, we show those for positive polarity and main order:

```
MIndic Pres Simul Pos Main : wir gehen
MIndic Pres Anter Pos Main : wir sind gegangen
MIndic Past Simul Pos Main : wir gingen
MIndic Past Anter Pos Main : wir waren gegangen
MIndic Fut Simul Pos Main : wir werden gehen
MIndic Fut Anter Pos Main : wir werden gegangen sein
MIndic Cond Simul Pos Main : wir würden gehen
MIndic Cond Anter Pos Main : wir würden gegangen sein
MConjunct Pres Simul Pos Main : wir gehen
MConjunct Pres Anter Pos Main : wir seien gegangen
MConjunct Past Simul Pos Main : wir gingen
MConjunct Past Anter Pos Main : wir wären gegangen
MConjunct Fut Simul Pos Main : wir werden gehen
MConjunct Fut Anter Pos Main : wir werden gegangen sein
MConjunct Cond Simul Pos Main : wir würden gehen
MConjunct Cond Anter Pos Main : wir würden gegangen sein
```

Sentences are obtained from clauses by fixing a value for mood, tense, anteriority and polarity, so their implementation type consists of a paradigm inflecting with respect to order only:

```
S  = {s : Order => Str} ;
```

Similar categories for relative and interrogative clauses and sentences will be handled in sections 5.4.3 and 5.4.4

Construction of Clauses

The subject in a German clause can be nominal in any of the four cases, or prepositional, or sentential, interogative or infinitival. To handle the various types uniformly, an auxiliary

operation `mkClause` is used that takes a subject, split into a string `subj` and agreement features `agr` (gender, number, and person), and a verb phrase `vp`, instantiates the fields `vp.nn` of nominal and `vp.inf` of infinitival objects to `agr` and returns a clause `(mkClause subj agr vp):Clause`.

In this auxiliary operation `mkClause`[102], the initial `useVP` builds the `VPForms` of the `vp`.

```
mkClause : Str -> Agr -> VP -> Cl = \subj,agr,vp ->
  let vps = useVP vp in {
    s = \\m,t,a,b,o =>
      let
        ord   = case o of {
          Sub => True ;  -- glue prefix to verb
          _ => False
          } ;
        verb  = vps.s  ! ord ! agr ! VPFinite m t a ;
        haben = verb.inf2 ;
        neg = negation ! b ;
        obj1  = (vp.nn ! agr).p1 ++ (vp.nn ! agr).p2 ; -- refl ++ pronouns ++ light nps
        obj2  = (vp.nn ! agr).p3 ;                     -- pp-objects and heavy nps
        obj3  = (vp.nn ! agr).p4 ++ vp.adj ++ vp.a2 ;  -- pred.AP|CN|Adv, via useComp
        compl = obj1 ++ neg ++ obj2 ++ obj3 ;          -- HL 6/2019
        infObjs = (vp.inf.inpl.p1)!agr ;
        infPred = vp.inf.inpl.p2 ;
        -- leave inf-complement of +auxV(2)V in place,
        -- extract infzu-complement of -auxV(2)V: (ComplVV, SlashV2V)
        infCompl : Str = case <t,a,vp.isAux> of {
               <Fut|Cond,Anter,True> => [] ;  _ => infObjs ++ infPred } ;
        pred : {inf, infComplfin : Str} = case <t,a,vp.isAux> of {
           <Fut|Cond,Anter,True>  =>                                  --# notpresent
             {inf    = infObjs ++ haben ++ infPred ++ verb.inf ;      --# notpresent
              infComplfin = -- es ++ wird ++ haben ++ tun ++ wollen   --# notpresent
                 infObjs ++ verb.fin ++ haben ++ infPred ++ verb.inf} ; --# notpresent
           <_,Anter,True> =>                                          --# notpresent
             {inf    = verb.inf ++ haben ;                            --# notpresent
              infComplfin = -- es ++ wird/hat/hatte ++ tun ++ wollen  --# notpresent
                 infObjs ++ verb.fin ++ infPred ++ verb.inf ++ haben} ; --# notpresent
            _ =>
             {inf    = verb.inf ++ haben ;
              infComplfin = -- es zu tun ++ versucht ++ [] ++ hat
                            infCompl ++ verb.inf ++ haben ++ verb.fin}
            } ;
        extra = vp.inf.extr!agr ++ vp.ext ;
      in
      case o of {
Main => subj ++ verb.fin ++ compl ++ infCompl ++ pred.inf ++ extra ;
Inv  => verb.fin ++ subj ++ compl ++ infCompl ++ pred.inf ++ extra ;
Subj =>             subj ++ compl ++   pred.infComplfin   ++ extra
```

[102]For simplicity, we here identify the result type `Clause:Type` of `MkClause` with the linearization type `Cl`.

```
      }
  } ;
```

In subordinate clauses the `verb.fin` is not at the end even in `<Fut|Cond,Simul,True>` and `<_,Anter,True>`, since we say *(weil) er das Buch hatte lesen wollen* instead of *(weil) er das Buch lesen wollen hatte.* (The position of adverbs `vp.a2` has to be changed, see Remark 80.)

Remark 82. *A fixed order of nominal, prepositional and other complements is built in, together with the position of the (sentence) negation. This order depends on the four components of* `vp.nn`*, where in each component there is an ordering depending on the constructions used to fill these components, so a minor word order flexibility seems possible. (But the strict separation between light and heavy nominal objects and between nominal and prepositional object is dubious.)*

As we insert complements in separate record fields, we loose the relation between word order and the sequence of insertions that is coded in a tree. So, in a sense, the implementation record can be "more abstract" than the tree: different trees having the same implementation record represent an ambiguity that is independent of how the record is linearized.

The relative order of complements and adverbs in German is rather free, e.g.

heute trifft Maria ihre Kollegen beim Essen nicht
Maria trifft heute ihre Kollegen beim Essen nicht
Maria trifft ihre Kollegen beim Essen heute nicht
ihre Kollegen trifft Maria heute beim Essen nicht
ihre Kollegen trifft Maria beim Essen heute nicht
beim Essen trifft Maria ihre Kollegen heute nicht

heute hat Maria ihre Kollegen beim Essen nicht getroffen
Maria hat heute ihre Kollegen beim Essen nicht getroffen
ihre Kollegen hat Maria heute beim Essen nicht getroffen
ihre Kollegen hat Maria beim Essen heute nicht getroffen
beim Essen hat Maria ihre Kollegen heute nicht getroffen
(getroffen hat Maria ihre Kollegen beim Essen heute nicht)

In particular, the subject need not be in front of a verb phrase (consisting of finite and infinite verb part, complements and adverbs), but can be exchanged with an object or adverb in the Mittelfeld (between finite verb "hat" and infinite verb part "getroffen").

How can we admit more relative orderings of subject, complements and adverbs? By a change of the type of clause constructions, for example by adding to `PredVP` *an additional argument* `p:Perm` *(linearized to the empty string as for* `t:Temp` *in* `UseCl`*), a specific permutation of these complements, adverbs and the subject could be fixed. But besides a change in the abstract grammar, this would at least need information about which of these complements and adverbs in* `vp.nn` *and* `vp.a2` *are nonempty; moreover, a component of* `vp.nn` *may concatenate two or more objects, which blocks a permutation of those. In general, if all objects and adverbs were stored in separate fields of* `vp`*, the sequence of filling these fields (with nonempty(?) strings) could be remembered in a parameter field* `p:Param` *of* `vp`*, and the clause paradigm might cover some permutations different from the one in* `vp.p`*. By making the parameter type* `Order` *depend on* `Param`*, the clause paradigm would contain all the permutations. But to select a specific one, the type of* `UseCl` *would still need a separate argument of type* `Param`*.*

Notice that `GrammarEng` *concatenates nominal objects, prepositional objects, adverbs and some adjectival and infinitival complements in* `vp.s2`*, so different trees of category* `VP` *may have different implementation records in* `GrammarEng`*, but the same ones in* `GrammarGer`*. It is not clear*

how word orders of different languages are related to each other by the abstract grammar. As translations ought to preserve meaning, the relative scope of quantifiers in objects and adverbs may be coded differently by the word orders of different languages, which may be a problem for the common abstract syntax ***Grammar****.*

The predication rule

```
PredVP    : NP -> VP -> Cl ;          -- John walks
```

for clauses with nominal (and prepositional) subject can be implemented by reading off from a verb phrase `vp:VP` the case (and preposition) `vp.c1` for the subject, putting a noun phrase `np` into this case (and preposition) to get a string `np.s ! vp.c1` and letting `mkClause` build the clause `PredVP np vp` with the help of the agreement features depending on `np` and `vp.c1`:

```
PredVP np vp =
  let subj = mkSubject np vp.c1
  in mkClause subj.s subj.a vp ;
```

The auxiliary operation `mkSubject : NP -> Preposition -> {s:Str ; a:Agr}`, when applied to `np:NP` and `p:Preposition`, returns the string selected from the inflection paradigm `np.s` by the preposition or case `p` and the agreement features `np.a`, when `p` is nominative, else the agreement of third person singular.

The predication rule for clauses with sentential, interrogative or infinitival subjects,

```
PredSCVP  : SC -> VP -> Cl ;          -- that she goes is good
```

is similarly implemented with `mkClause`, using the string of its subject argument and the agreement of third person singular:

```
PredSCVP sc vp = mkClause sc.s (agrP3 Sg) vp ;
```

Remark 83: The default value `vp.c1 = prepNom:Preposition` for nominal or prepositional subjects is ignored by `PredSCVP`. Can we distinguish *on the type level* between verb phrases taking a nominal from those taking a sentential subject? (With dependent categories one can do it, but what is the price in grammar writing, and what is the gain in correctness?)

Todo 42: Discuss that many extractions (focussing) of partial phrases will not be covered in `LangGer`, like "*gern gelesen hat er den Brief nicht*" and "*gelesen hat er den Brief nicht gern*".

Construction of Sentences

The rules to construct sentences

```
UseCl    : Temp -> Pol -> Cl  -> S ;   -- she had not slept
```

is simply implemented by selecting, from the paradigm `cl.s` of a clause `cl`, the strings for given values `t:Temp` for tense and `p:Pol` for polarity:

```
UseCl t p cl = {
  s = \\o => t.s ++ p.s ++ cl.s ! t.m ! t.t ! t.a ! p.p ! o
  } ;
```

The values `t:Temp` and `p:Pol` provide empty strings `t.s` and `p.s` as well as values `t.m : Mood`, `t.t : Tense`, `t.a : Anteriority` and `p.p : Polarity`. These are used to select from the paradigm

```
cl.s : Mood => ResGer.Tense => Anteriority => Polarity => Order => Str
```

a corresponding table of type `Order => Str` as paradigm of the sentence (see Section 5.10).

Modification of Sentences

A sentence can be modified by adding an adverb, using

```
AdvS     : Adv -> S  -> S ;            -- then I will go home
```

The adverb is added at the beginning. To obtain a sentence in inverted or main verb order, the argument sentence in inverted verb order follows (giving a sentence in main order); to obtain a sentence in subordinate order, the argument sentence in subordinate order follows:

```
AdvS a s = {s = table {Sub => a.s ++ s.s ! Sub ; o => a.s ++ s.s ! Inv}} ;
```

Remark 84. *This is too simple; in inverted and subordinate clauses, the adverb ought to be in third position, following the finite verb and the subject, e.g. "würde man heute nicht bereit sein" and "(wenn) man heute nicht bereit sein würde", but we get*

```
TestLang> l -table (AdvS today_Adv
                      (UseCl (TTAnt TCond ASimul) PNeg (GenericCl ready_VP)))
s : today one wouldn't be ready
s Main : heute würde man nicht bereit sein
s Inv : heute würde man nicht bereit sein
s Sub : heute man nicht bereit sein würde
```

It seems this construction should only be used with ***`Main`*** *order. (For inverted and subordinated order, use* ***`AdvVP`*** *or* ***`AdvVPSlash`*** *to adverbially modify the verb phrase, not the sentence.)*

Remark 85: Adverbs can be split, e.g. *dort, wo S*, and the parts are often separated, e.g. *wir wollen dort wohnen, wo es schön ist*, Q58: Is *dort* the correlate of the adverb *wo S*, or *dort* the adverb and *wo S* a relative clause as in *in Bayern, wo es schön ist*, i.e. do we want `Adv = {s:Str ; cor:Str}` or `Adv ={s:Str ; rc:Str}`?

To modify a sentence by an adverbial clause, a separate construction

```
ExtAdvS  : Adv -> S  -> S ;            -- next week, I will go home
```

is used that attaches a comma to the sentence-initial adverbial clause:

```
ExtAdvS a s =
  {s = table {Sub => a.s ++ "," ++ s.s ! Sub ;
              o => a.s ++ "," ++ s.s ! Inv}} ;
```

This covers subjunctive clauses, but these can also be added to the end, using

```
SSubjS    : S -> Subj -> S -> S ;       -- I go home, if she comes
```

The implementation attaches to the end of the given sentence `a : S` a modifying sentence `b : S` in subordinate verb order with leading comma and subjunction `s : Subj`:

```
SSubjS a s b = {s = \\o => a.s ! o ++ "," ++ s.s ++ b.s ! Sub} ;
```

The modification of a sentence by a relative clause, i.e. the construction

```
RelS     : S -> RS -> S ;              -- she sleeps, which is good
```

appends, in each of the three orders, a comma-separated relative clause at the end, introduced by the relative pronoun *was*:

```
RelS s r = {s = \\o => s.s ! o ++ "," ++ r.s ! RSentence} ; --- "was"
```

Incomplete Clauses and Incomplete Sentences

An ***incomplete clause*** is a clause missing a nominal object. Incomplete clauses are used to construct relative clauses (p. 145) and (object-) questions (p. 147): these have a clause-initial relativizing or interrogative noun (or prepositional) phrase in the case of the missing object, followed by the incomplete clause. An ***incomplete sentence*** is a sentence missing a nominal object.

Categories `ClSlash` and `SSlash`

The implementation categories are those of clauses and sentences, extended by a field `c2` for the (preposition and) case of the missing nominal object:

```
ClSlash = {
  s : Mood => ResGer.Tense => Anteriority => Polarity => Order => Str ;
  c2 : Preposition
  } ;

SSlash = {s : Order => Str} ** {c2 : Preposition} ;
```

The paradigm type of `ClSlash.s` ends in `Order => Str` rather than `Str`, because both relative clauses and object questions are derived from incomplete clauses, and these use different order: from the clause *er hat ihr einen Brief geschickt* we can relativize the object by *(der Brief), den er ihr geschickt hat* with order `Sub` of the incomplete clause *er hat ihr _ geschickt*, and we can form an object-question *welchen Brief hat er ihr geschickt*, with order `Inv` in the incomplete clause.

Remark 86: The category `ClSlash` should have a result type `Agr => Str` or `RelGenNum => Case => Str` instead of `Str` in field `s`, as `RCl` already has. Then we could implement `SlashVP` by a modification `mkClSlash` of `mkClause` that resolves reflexives against the (yet unknown) object, or at least does *not* resolve them against the inserted subject. (The values of `GenNum` have to be extended by a value for the relative pronoun *was* for clauses

```
RelGenNum = RGenNum GenNum | RSentence ;
```

See the Section 5.4.3 on relative clauses for details.)

Construction and Modification of Incomplete Clauses `ClSlash`

The main constructions are:

```
SlashVP   : NP -> VPSlash -> ClSlash ;      -- (whom) he sees
AdvSlash  : ClSlash -> Adv -> ClSlash ;     -- (whom) he sees today
SlashPrep : Cl -> Prep -> ClSlash ;         -- (with whom) he walks
SlashVS   : NP -> VS -> SSlash -> ClSlash ; -- (whom) she says that he loves
```

Currently, the rule

```
SlashVP   : NP -> VPSlash -> ClSlash ;      -- (whom) he sees
```

is implemented by `mkClause`:

```
SlashVP np vp =
   let subj = mkSubject np vp.c1
   in mkClause subj.s subj.a vp ** {c2 = vp.c2} ;
```

Problem 7. *This implementation of `SlashVP` assumes that open reflexives in its argument `vp` refer to the subject and so dependencies on `Agr` are instantiated to the agreement of the subject. But the resulting `(SlashVP np vp)` is an incomplete clause, missing a nominal* object *to which the reflexives may have to refer, depending on `vp.objCtrl`.*

```
TestLang> l (RelSlash IdRP (SlashVP (UsePron i_Pron)
                (SlashV2V lassen_V2V (ReflVP (SlashV2a wash_V2)))))
that I let wash myself
den ich mich selbst waschen lasse
```

In the definition of `SlashVP`, we must modify `mkClause` to `mkClSlash: ... => Agr => Str`, obtained like `mkClause` with open constituents, so to speak. We seem to need

```
RelSlash : RP -> ClSlash -> RCl ; -- whom John loves
```

such that `RelSlash rp cls = rp.s ++ cls.s!(rp.agr)`. But this would make `ClSlash` rather expensive, extending the table `ClSlash.s` by a factor of `|Agr|` $= 3 * 2 * 3 = 18$. But if we let `ClSlash.s` end in `RelGenNum => Str` instead of `Agr => Str`, due to `|RelGenNum|` $= 5$ this is reasonably efficient and good enough to have reflexive resolution with the following modified `SlashVP` and `RelSlash`:

```
SlashVP np vp =
  let subj = mkSubject np vp.c1
  in mkClSlash subj.s subj.a vp ** { c2 = vp.c2 } ;

RelSlash rp slash = {
  s = \\m,t,a,p,gn =>
    (appPrep slash.c2 rp) ! gn ++ slash.s ! m ! t ! a ! p ! Sub ;
  c = slash.c2.c
  } ;
```

For `SlashVP` and `RelSlash`, we use the following modification of `ClSlash` and `mkClSlash` in `TestLangGer.gf`. The point is to let `ag : Agr` depend on `objCtrl`:

```
lincat
  ClauseSlash = {
    s : Mood => ResGer.Tense => Anteriority => Polarity => Order
                                                    => RelGenNum => Str ;
    c2 : Preposition
    } ;


oper
  gnToAgr : RelGenNum -> Agr = \gn ->
     case gn of {RGenNum (GSg g) => AgSgP3 g ;
                 RGenNum GPl     => AgPl P3 ;
                 RSentence       => AgSgP3 Neutr} ;

  mkClSlash : Str -> Agr -> ResGer.VPSlash -> ClauseSlash = \subj,agr,vp ->
  let vps = useVP vp in lin ClauseSlash {
    c2 = vp.c2 ;
    s = \\m,t,a,b,o,gn =>
      let
        ord   = case o of {
          Sub => True ;  -- glue prefix to verb
          _ => False
          } ;
        verb  = vps.s  ! ord ! agr ! VPFinite m t a ;
        haben = verb.inf2 ;
        neg = negation ! b ;
        ag : Agr = case vp.objCtrl of {True => gnToAgr gn ; _ => agr} ;
        obj1  = (vp.nn ! ag).p1 ++ (vp.nn ! ag).p2 ; -- refl ++ pronouns ++ light nps
        obj2  = (vp.nn ! ag).p3 ;                    -- pp-objects and heavy nps
        obj3  = (vp.nn ! ag).p4 ++ vp.adj ++ vp.a2 ; -- pred.AP|CN|Adv, via useComp HL 6/201
        compl : Str = obj1 ++ obj2 ++ neg ++ obj3 ;
        infObjs = vp.inf.inpl.p1 ! ag ;
        infPred = vp.inf.inpl.p2 ;
        infCompl : Str = case <t,a,vp.isAux> of {
               <Fut|Cond,Anter,True> => [] ;  _ => infObjs ++ infPred } ;
        pred : {inf, infComplfin : Str} = case <t,a,vp.isAux> of {
           <Fut|Cond,Anter,True>  =>                                  --# notpresent
             {inf    = infObjs ++ haben ++ infPred ++ verb.inf ;      --# notpresent Duden
              infComplfin = -- es ++ wird ++ haben ++ tun ++ wollen   --# notpresent
                 infObjs ++ verb.fin ++ haben ++ infPred ++ verb.inf} ; --# notpresent
           <_,Anter,True> =>                                          --# notpresent
             {inf    = verb.inf ++ haben ;                            --# notpresent
              infComplfin = -- es ++ wird/hat/hatte ++ tun ++ wollen  --# notpresent
                 infObjs ++ verb.fin ++ infPred ++ verb.inf ++ haben} ; --# notpresent
            <Pres,_,_> =>
             {inf    = verb.inf ++ haben ;
              infComplfin = -- es zu tun ++ [] ++ [] ++ versucht
```

```
                infCompl ++ verb.inf ++ haben ++ verb.fin}
                                                                    ; --# notpresent
          _ =>                                                        --# notpresent
           {inf    = verb.inf ++ haben ;                              --# notpresent
            infComplfin = -- es zu tun ++ versucht ++ [] ++ hat       --# notpresent
                          infCompl ++ verb.inf ++ haben ++ verb.fin}  --# notpresent
          } ;
      extra : Str = (vp.inf.extr) ! ag ++ vp.ext ;
    in
    case o of {
      Main => subj ++ verb.fin ++ compl ++ infCompl ++ pred.inf ++ extra ;
      Inv  => verb.fin ++ subj ++ compl ++ infCompl ++ pred.inf ++ extra ;
      Subj =>             subj ++ compl ++   pred.infComplfin   ++ extra
    }
} ;
```

Q59: Is the position of the adverbs `vp.a2` in `obj3` correct or dubious?

Todo 43: To resolve reflexive possessive against the missing nominal object, we may have to use `rp.a : RAgr = RNoAg | RAg Number Person` to compute a different agreement value for specializing `vp.nn`. (For example: "wir, die wir(!) dies unser(!) Schicksal so gut erkannten")

The other uses of `ClSlash` are adapted to this `ClauseSlash` in `TestLangGer.gf`. (The `RelGenNum` (and `ip.a`?) plays a role in forming object questions from incomplete clauses, to resolve reflexives and reflexive possessives: "wer hat seinem (eigenen!) Kind versprochen, sich(!) zu bessern?".)

```
QuestSlash ip slash =
  let gn : GenNum = case ip.n of {Sg => GSg Masc ; _ => GPl}
  in {
      s = \\m,t,a,p =>
        let
          cls = slash.s ! m ! t ! a ! p ;
          who = appPrep slash.c2 ip.s ;
        in table {
          QDir   => who ++ cls ! Inv ! (RGenNum gn);
          QIndir => who ++ cls ! Sub ! (RGenNum gn)
          }
  } ;

AdvSlash slash adv = {
  s  = \\m,t,a,b,o,gn => slash.s ! m ! t ! a ! b ! o ! gn ++ adv.s ;
  c2 = slash.c2
} ;

SlashPrep cl prep = {
  s = \\m,t,a,p,o,gn => cl.s ! m ! t ! a ! p ! o ;
  c2 = prep
} ;

SlashVS np vs slash =
  let subj = mkSubject np PrepNom ;
```

```
      vp = insertExtrapos (conjThat ++ slash.s ! Sub) (predV vs) -- comma?
  in mkClSlash subj.s subj.a (vp ** {c2 = slash.c2 ;
                                     objCtrl = False}) ;  -- pseudo

UseSlash t p cl = {
  s = \\o => t.s ++ p.s ++ cl.s ! t.m ! t.t ! t.a ! p.p ! o ;
  c2 = cl.c2
  } ;
```

Some occurrences of `ClSlash` in `Extra/Extend` remain to be adapted to `ClauseSlash`:

Construction of Incomplete Sentences `SSlash`

```
UseSlash : Temp -> Pol -> ClSlash -> SSlash ; -- (that) she had not seen
```

Construction of Embedded Sentences `SC`

The three constructions for sentential, interrogative and infinitival complements

```
EmbedS    : S  -> SC ;              -- that she goes
EmbedQS   : QS -> SC ;              -- who goes
EmbedVP   : VP -> SC ;              -- to go
```

are implemented by

```
EmbedS  s  = {s = conjThat ++ s.s ! Sub} ;  -- no leading comma, if
EmbedQS qs = {s = qs.s ! QIndir} ;          -- sentence-initial
EmbedVP vp = {s = useInfVP False vp} ;
```

As noted in the comment, a leading comma is not part of the complements, because they can also be used as subjects, in sentence-initial position. (Q60: Is then the first letter capitalized by token `Predef.CAPIT:Str`?)

Remark 87: `useInfVP` has been changed and needs to be adapted here and in `CatGer`.

5.4.3. Relative Clauses and Relative Sentences

Categories `RP` of Relative Pronoun and `RCl` of Relative Clause

A ***relative clause*** basically is obtained from a clause (in subordinate, verb final order) by dropping a nominal complement of its verb and putting a relative pronoun (or relativizing noun phrase or prepositional phrase) in front of the resulting incomplete clause. Usually, the dropped nominal complement is in third person, but first and second person are possible as well: *du, dem ich oft geholfen habe*, or *wir, die wir(!) dies unser Schicksal so gut erkannten* (G. Seferis). The relative pronoun inflects for case, as the nominal complement, but also for the gender of the (head) noun of the dropped nominal complement; an additional form is used to relativize a sentential complement, e.g. *die Sonne schien, was uns freute.*

```
RelGenNum = RGenNum GenNum | RSentence ;
```

The 4-valued parameter domain `GenNum` was introduced for noun phrases (p.64). The relative pronoun agrees in person and number with the finite verb, and an additional agreement value `RNoAg` is used when the relative pronouns represents a sentential complements of the verb.[103]:

```
RAgr = RNoAg | RAg Number Person ;
```

Therefore, the implementation category `RP` of relative pronoun is:

```
RP = {s : RelGenNum => Case => Str ; a : RAgr} ;
```

The category of relative clauses is similar to that of incomplete clauses, `ClSlash`, but instead of `c2:Preposition` we have the simpler field `c:Case` (since contractions like *an dem* = *am* only occur with article *dem*, not with relative pronoun *dem*) and the paradigm `RCl.s` varies in `RelGenNum`, which determines the form of the relative pronoun:

```
RCl = {s : Mood => Tense => Anteriority => Polarity => RelGenNum => Str ;
       c : Case} ;
```

As subordinate clauses, relative clauses have the fixed order `Sub`, i.e. the verb is at the end.

Q61: The flag `RCl.c` classifies relative clauses and relative sentences. It is set in `RelCl`, `RelVP` and `RelSlash` below. As far as I see, it is *used* only in `UseRCl`, to set `RS.c`, and `RS.c` is *used* only in `ConjunctionGer`, to set the flag `c` of the category of lists of relative clauses:

```
lincat
  [RS] = {s1,s2 : RelGenNum => Str ; c : Case} ;
lin
  BaseRS x y = twoTable RelGenNum x y ** {c = y.c} ;
  ConsRS xs x = consrTable RelGenNum comma xs x ** {c = xs.c} ;
```

As there is no comparison of `c` in different list members, these flags could be removed, I think.

Remark 88: In principle, a relative clause ought to inflect not for `RelGenNum`, but for `Agr`, because possessives in the clause depend on person, e.g. *du, der|die du deine Kinder liebst*. But this was too complex for `ClSlash`, so it probably will be too complex for `RCl`, too.

Construction of Relative Clauses

A somewhat rare way to turn a clause into a relative clause,

```
RelCl    : Cl -> RCl ;            -- such that John loves her
```

is to use the clause in subordinate order, introduced by *derart, daß*:

```
RelCl cl = {
  s = \\m,t,a,b,_ => "derart" ++ conjThat ++ cl.s ! m ! t ! a ! b ! Sub ;
  c = Nom
  } ;
```

[103] Check that this is really the usage of `RNoAg`.

(The English clause relativization by *such that* does not correspond to the German *so daß*, which introduces a "consecutive clause". But *derart dass* or Kleist's *dergestalt daß* are also in little use; is this a clause relativization at all? The clause relativization *Johann liebt Maria, was mich freut* is different, `RelVP rp vp` below.)

The two standard ways to relativize are by relativizing the nominal subject or object of a clause. If the subject of a clause `PredVP np vp` is relativized using

```
RelVP    : RP -> VP -> RCl ;      -- who loves John
```

we build a clause from the verb phrase `vp` with a suitable form of the relative pronoun `rp` as subject; the verb in the relative clause has to agree in person and number with the omitted subject, so we need the agreement features `RP.a` to choose the proper form of the verb `vp.s`:

```
RelVP rp vp = {
  s = \\m,t,ant,b,rgn =>
    let
      gn = case rgn of {
        RGenNum gf => gf ;
        RSentence => GSg Neutr
        } ;
      agr = case rp.a of {
        RNoAg => agrP3 (numGenNum gn) ;
        RAg n p => Ag Neutr n p
        } ;
      cl = mkClause (rp.s ! rgn ! Nom) agr vp
    in
    cl.s ! m ! t ! ant ! b ! Sub ;
  c = Nom
  } ;
```

Remark 89: First and second person subjects can be relativized as well, using "*ich, der|die ich ... tue*", "*du, der|die du ... tust*", etc., but not with `RelNP : NP -> RS -> NP`, since `RS` and `RCl` deliver a table `...=> RelGenNum => Str` intended to relativize a subject in third person.

Todo44: Add a special rule `RelPronVP : Pron -> VP -> NP` that builds, so to speak, `pron, rp ++ (mkClause pron vp)!Sub`. (From `pron.a` we get gender, number and person, so we can use a suitable value for `RelGenNum` in `rp`. But in `StructuralGer`, `i_Pron` has inherent gender `Masc`, `youSg_Pron` inherent gender `Fem`, so one needs `iFem_Pron` and `youSgMasc_Pron` in `Extend`.) ◁

If a nominal object of a clause is relativized, we combine the incomplete clause obtained by dropping the object with a suitable form of the relative pronoun, using

```
RelSlash : RP -> ClSlash -> RCl ; -- whom John loves
```

Only the direct object (`ClSlash.c2`) can be relativized by `RelSlash`; an indirect object of a ternary verb has to be turned into a direct one by the `Slash2V3`-rule. The implementation is:

```
RelSlash rp slash = {
  s = \\m,t,a,p,gn =>
      (appPrep slash.c2 rp) ! gn ++ slash.s ! m ! t ! a ! p ! Sub ;
  c = slash.c2.c
  } ;
```

See the correction of this above (p. 145) concerning the dependence on `RelGenNum`.

The relativization of a nominal object combines the preposition used to attach the object with the relative pronoun: *ich warte auf den Tag* $\mapsto$ *der Tag, auf den ich warte*.

```
TestLang> l (RelSlash IdRP (SlashVP (UsePron i_Pron) (SlashV2a wait_V2)))
that I wait for
auf den ich warte
```

This is done by the `appPrepIP` case of `appPrep`, which also implements the contraction *auf was* = *worauf*:[104]

```
Lang> l -table (UseRCl (TTAnt TPres ASimul) PPos
    (RelSlash IdRP (SlashVP (UsePron i_Pron) (SlashV2a wait_V2))))
s (RGenNum (GSg Masc)) : auf den ich warte
s (RGenNum (GSg Fem)) : auf die ich warte
s (RGenNum (GSg Neutr)) : auf das ich warte
s (RGenNum GPl) : auf die ich warte
s RSentence : worauf ich warte
```

Relative clauses can also be obtained by relativising the noun phrase of an adverbial:

```
Lang> l DetCN (DetQuant DefArt NumSg) (RelCN (UseN house_N)
         (UseRCl (TTAnt TPres ASimul) PPos (RelSlash IdRP
          (SlashPrep (PredVP (UsePron we_Pron) (UseV sleep_V)) in_Prep))))
das Haus , in dem wir schlafen
```

Here, `SlashPrep : Cl -> Prep -> ClSlash` considers `Prep` as a partial adverb, `Prep = Adv/NP`.

Construction of Relative Pronouns

There are two constructions of relative pronouns. The first one,

```
IdRP  : RP ;                        -- which
```

is implemented by

```
IdRP = {s = relPron ; a = RNoAg} ;
```

where `relPron : RelGenNum => Case => Str` is a paradigm of the four cases of *der*, *die*, *das* in singular and *die* in plural, and the four cases *was*, *was*, *was*, *wessen* for `RSentence`, and `RNoAg` the special value for agreement of relative pronouns.[105]

Ambiguity: *die Frau , mit der er verheiratet ist*:

```
RelSlash IdRP (SlashVP np (VPSlashPrep vp prep))
= RelSlash IdRP (SlashPrep (PredVP np vp) prep)
```

[104] Since the first value of `RelGenNum`, i.e. `RGenNum (GSg Masc)`, is used for parsing a relative sentence, we cannot parse *worauf ich warte*, only *auf den ich warte*.

[105] more precisely?

The other construction of relative pronouns is by

```
FunRP : Prep -> NP -> RP -> RP ;  -- the mother of whom
```

An application `FunRP p np rp` is a relative pronoun that, for `gn:RelGenNum` and `c:Case`, extends the `np` in case `c` by the given `rp` applied to `gn`, and stores number and person of the `np.a` in its agreement field:

```
FunRP p np rp = {
  s = \\gn,c => np.s ! False ! c ++ appPrep p (rp.s ! gn) ;
  a = RAg (numberAgr np.a) (personAgr np.a)
  } ;
```

(Can the construction reasonably be iterated?) *der Aufstieg auf den Berg war schwierig* $\mapsto$ *(der Berg), der Aufstieg auf den schwierig war* [wouldn't we rather say: *(der Berg), auf den der Aufstieg schwierig war*?]

This calls for a discussion: `FunRP` seems to be intended for relativizing an embedded nominal in a nominal object: *I know the mother of John* $\mapsto$ *John, the mother of whom I know.* For a possessive genitive, we need a different linearization, `Extend.GenRP`: *die Mutter von|des Johann* $\mapsto$ *(Johann), dessen Mutter.* Todo 45: Implement `GenIP` and `GenRP`.

The most complicated cases are relativizations of nominals in infinitival or sentential objects. *he promised us to provide a proof of the claim* $\mapsto$ *(the claim) he promised us to provide a proof of*, `Ger` *(die Behauptung), von der er uns einen Beweis zu liefern versprach.* These are only addressed in `Extra/Extend`.

There are no modification rules for relative clauses and relative sentences.

Relative Sentences

A ***relative sentence*** is obtained from a relative clause by fixing values for mood, tense, anteriority and polarity. Its category therefore consists of a field `s` for an inflection paradigm that simplifies the one for relative clauses by dropping dependencies on mood, tense and antiority, and of a field `c` for a case (holding the case of the relative pronoun?):

```
RS = {s : RelGenNum => Str ; c : Case} ;
```

The only way to construct a relative sentence is by specializing a relative clause using

```
UseRCl   : Temp -> Pol -> RCl -> RS ;  -- that had not slept
```

This rule is implemented by selecting from the paradigm `cl.s` of a given relative clause `cl` the string for given values `t:Temp` and `p:Pol`:

```
UseRCl t p cl = {
  s = \\r => t.s ++ p.s ++ cl.s ! t.m ! t.t ! t.a ! p.p ! r ;
  c = cl.c
  } ;
```

The values `t:Temp` and `t:Pol` contribute empty strings `t.s` and `p.s` to the strings in the paradigm and provide parameter values `t.m`, `t.t`, `t.a` and `t.p` for mood, tense, anteriority and polarity for the selcection from `cl.s`.

5.4.4. Interrogative Clauses and Questions (improve!)

Categories of Interrogative Clause and Question

The implementation type of interrogative clauses is similar to that of (definite) clauses. It consists of a field `s` of an inflection paradigm varying in mood, tense, anteriority, polarity and position of the finite verb:

```
QCl = {s : Mood => Tense => Anteriority => Polarity => QForm => Str} ;
```

However, while in (definite) clauses the finite verb position varies according to the 3-valued paramter type `Order`, in interrogative clauses there are only two values,

```
param
  QForm = QDir | QIndir ;
```

In direct interrogative clauses, e.g. *hast du gut geschlafen*, the finite verb is in initial position, i.e. `QDir` corresponds to the value `Inv` of `Order`, while in subordinate interrogative clauses, e.g. *ob du gut geschlafen hast*, the finite verb is in final position, i.e. `QIndir` corresponds to `Sub`.

The implementation category for questions consists of a field for an inflection paradigm varying only according to `QForm`:

```
QS = {s : QForm => Str} ;
```

Interrogative clauses are either Yes-No-questions (built from an ordinary clause by intonation or word order) or build from a verb phrase by adding an interrogative noun phrase or adverb at the beginning, or an interrogative complement of a copula verb. We first introduce these interrogative categories (`IP`, `IAdv`, `IComp`) and their construction and modification rules, and then come back to interrogative clauses.

5.4.5. Interrogative Noun Phrases

`Lang` has a limited notion of ***interrogative noun phrase***, a category `IP` called *interrogative pronoun* in GF. I prefer to use the more general name[106], since `IP` also covers noun phrases with interrogative determiners, e.g. *welcher bekannte Autor* or *wessen bedeutendes Buch*[107], and noun phrases with interrogative attributes, e.g. *die Bücher welcher Autoren* or *die Weine aus welchem Land*. I will use "*interrogative pronoun*" in the traditional sense, i.e. for *wer* and *was*.

Interrogative noun phrases are always in third person, e.g. *wer* or *welche Studenten*, vary in case and have an inherent number:

```
IP = {s : Case => Str ; n : Number} ;
```

The field `n:Number` is the analogon of the field `NP.a:Agr` for agreement features. The interrogative pronouns *wer* and *was* are singular, but other interrogative noun phrases are plural, so as subjects, they determine the number of the finite verb, e.g. *welcher Student bestand das Examen* vs. *welche Studenten bestanden das Examen*.

[106] Maybe the category name should be `INP`, similar to the category `RNP` of "*reflexive noun phrases*" in `Extend`.
[107] provided as `IQuant` in `ExtraGer`

Remark 90: The linearization type `IP` needs corrections. Instead of `n:Number`, a more general agreement field `a:GenNum` is needed: as subject, an interrogative noun phrase also determines reflexive possessive attributes of objects, e.g. *welches Kind hat seine Mutter gesucht* vs. *welche Mutter hat ihr Kind gesucht*. We may also have to add a field `isPron : Bool` to distinguish interrogative pronouns from more general interrogative noun phrases. E.g., the possessive can be expressed by a pre-nominal interrogative possessive pronoun, e.g. *wessen Wagen*, or by a post-nominal interrogative complex noun phrase in genitive, e.g. *der Wagen welcher Person*, but hardly the other way round, at least not **der Wagen wessen*. ◁

Interrogative noun phrases are constructed from interrogative determiners by

```
IdetCN    : IDet -> CN -> IP ;       -- which five songs
IdetIP    : IDet       -> IP ;       -- which five
```

e.g. *wie viele Brüder von Johann*, *welches kleine Kind*, *welche drei Kinder*, *welche drei*. These constructions correspond to the constructions

```
DetCN : Det -> CN -> NP ;
DetNP : Det -> NP ;
```

of noun phrases in Section 4.1. So we first consider interrogative determiners.

Interrogative determiners and quantifiers

In analogy to the construction of determiners by `DetQuant`, interrogative determiners are build from an interrogative quantifier or possessive pronoun and a cardinal number, e.g. *welcher eine (große Fehler)* or *wessen drei (kleine Kinder)*.

An ***interrogative determiner*** varies in gender (in singular) and case and has an inherent number. They also determine the adjective inflection in interrogative noun phrases: *welches eine gute Buch liebst du*, *wessen gutes Buch liebst du*. So, the linearization type is

```
IDet = {s : Gender => Case => Str ; n : Number ; a : Adjf} ;
```

An ***interrogative quantifier***, e.g. *welche*, *welcher*, *welches*, varies in gender (in singular)[108], number and case, and determines the adjective inflection:

```
IQuant = {s : GenNum => Case => Str ; a : Adjf} ;
```

Construction of Interrogative Quantifiers and Determiners

The only interrogative quantifier in `Lang` is the lexical element *welcher*, *welche*, *welches*, defined in `StructuralGer` by

```
which_IQuant = {s = \\gn,c => "welch" + detEnding ! gn ! c ; a = Strong} ;
```

An interrogative determiner is either a lexical element, e.g. *wieviel*, *wieviele* (eng. *how much*, *how many*), the second of which defined by

[108] I changed the type by combining the `Number` and `Gender`, as I did for `Quant`, and by adding `a:Adjf`.

```
  how8many_IDet = {
    s = \\g,c => "wie viel" + detEnding ! (gennum g Pl) ! c ;
    n = Pl ; a = Strong
  } ;
```

in `StructuralGer`, or constructed with

```
    IdetQuant : IQuant -> Num -> IDet ;  -- which (five)
```

from an interrogative quantifier by fixing the grammatical number obtained from the cardinal:

```
    IdetQuant iquant num =
      let
        n = num.n
      in {
      s = \\g,c => iquant.s ! (gennum g n) ! c ++ num.s ! g ! c  ;
      n = n
      } ;
```

For example, this gives *welche drei* and *welche 13*.

```
    Lang> p -cat=IDet "welche drei"
    IdetQuant which_IQuant
              (NumCard (NumNumeral (num (pot2as3 (pot1as2 (pot0as1 (pot0 n3)))))))
```

Construction of Interrogative Noun Phrases

An interrogative noun phrase can be a lexical element, i.e. an interrogative pronoun, e.g. *was* and *wer*, or a noun phrase constructed with an interrogative determiner and possibly a cardinal, e.g. *wie viele Brüder von Johann*, *welches kleine Kind*, *welche drei Kinder*, *welche drei*.

Within an interrogative noun phrase constructed by

```
    IdetCN    : IDet -> CN -> IP ;       -- which five songs
```

the interrogative determiner agrees (in singular) with the gender of the common noun, governs the adjective declination and number of the common noun, e.g. *welcher junge Hund, welche junge Katze, welches junge Tier, wie viele alte Hunde*; the interrogative noun phrase inflects according to case only:

```
    IdetCN idet cn =
      let
        g = cn.g ;
        n = idet.n
      in {
        s = \\c => idet.s ! g ! c ++ cn.s ! idet.a ! n ! c
                   ++ cn.adv ++ cn.rc ! n ++ cn.ext ;
        n = n
      } ;
```

The rule

```
IdetIP    : IDet       -> IP ;       -- which five
```

is similar, but with fixed gender `Neutr` and without a common noun.

Remark 91: To ask for the possessor of something, *whose car*, *wessen Wagen*, use `Extend.GenIP` or `Extend.GenModIP`. There seems to be no rule to ask with post-nominal interrogative possessive, e.g. *the car of whom*, *der Wagen von wem*. We also cannot ask with *indefinite* interrogative noun phrases, like *what (kind of) a car (is this)*, *was für ein Wagen (ist das)*? [But: the contraction *mit was* $\mapsto$ *womit* does not apply to *mit was für einem N.neutr*.]

Modification of Interrogative Noun Phrases

An interrogative noun phrase can be modified by

```
AdvIP : IP -> Adv -> IP ;
```

This just expands an interrogative noun phrase by an adverb at the end:

```
AdvIP ip adv = {
  s = \\c => ip.s ! c ++ adv.s ;
  n = ip.n
  } ;
```

e.g. *wer hier* or *wer in unserer Stadt*. Probably, the construction should not be used with adverbial sentences `adv`, e.g. **wer, weil die Sonne scheint*.

Remark 92: Interrogative noun phrases built by `IDetCN id cn` can have relative clauses in the `cn` constituent. But relative clauses can in German also be combined with interrogative pronouns, e.g. *wer, der bei Verstand ist, würde das tun?* Todo 46: Add a rule `RelIP : IP -> RelS -> IP`, an interrogative version of `RelNP`, to `ExtendGer` or `ExtraGer`.

5.4.6. Interrogative Adverbs

An ***interrogative adverb*** is an adverb which turns a clause into an interrogative clause or question. Semantically, it questions when or where the fact expressed by the clause holds, or how or why the action expressed is performed, etc.

Interrogative adverbs, e.g. *wann* or *an welchem Ort*, do not vary at all, so their implementation type is

```
IAdv = {s : Str} ;
```

Q62: Maybe we need a second field `s2:Str`, e.g. *wann (wollen wir arbeiten), wenn nicht jetzt*?

Construction of Interrogative Adverbs

An interrogative adverb can be a lexical element like *wie*, *wann*, *wo*, *warum*, defined by

```
how_IAdv = {s = "wie"} ;
when_IAdv = {s = "wann"} ;
where_IAdv = {s = "wo"} ;
why_IAdv = {s = "warum"} ;
```

in `StructuralGer`. Interrogative adverbs can also be constructed with

```
PrepIP    : Prep -> IP -> IAdv ;     -- with whom
```

by applying a preposition to an interrogative noun phrase:

```
PrepIP p ip = {
  s = appPrep p ip.s ;
  } ;
```

e.g. *mit was*, *mit wem* or *mit welchem Freund*. However, since the clause `cl` in `QuestIAdv iadv cl` can already have an adverb, the same adverbial function of the clause may get filled twice, e.g. *warum arbeitet Johann, weil er Geld braucht*. (While there is a category `ClSlash = Cl/NP` for clauses missing a nominal object, there is no category `Cl/Adv` for a clause missing an adverb, hence there are no specific constructions of type `IAdv -> Cl/Adv -> QCl`.)

Remark 93: In German there are also many contractions of *wo* with a preposition, e.g. *womit* = *mit was*, *wofür*, *wohin*, *worin*; some of these are interrogative pronominal adverbs, e.g. *womit* in *womit habt ihr das gemacht*, but some can (also or only) be used to introduce a relative clause: *das Ereignis, woran wir denken*. (Todo 47: implement these adverbs, by opers of types `Prep -> IPron -> IAdv` and `Prep -> RelPron -> RAdv`, similar to the contraction of prepositions with definite article singular or as tree transformations.)

Remark 94: In German (and other languages), one may ask for the value on the implicit scale provided by an adjective, as in *wie alt*, *wie lang*, *wie kalt*. These can be used as interrogative complement to a copula, e.g. *wie kalt ist das Wasser* (c.f. `IComp`), but also attributively, e.g. *in wie kaltem Wasser schwimmst du*? So we may need a construction `HowA : A -> IAP` to build an interrogative adjective phrase from an adjective (possibly derived from a participle: e.g. *ein wie berühmter Autor ist er?*). So far, there is no category `IAP` in `Grammar` or `Extend`. ◁

Modification of Interrogative Adverbs

An interrogative adverb can be modified by

```
AdvIAdv : IAdv -> Adv -> IAdv ;
```

which attaches an adverb at the end of the interrogative adverb, e.g. *wo in diesem Land*:

```
AdvIAdv i a = {s = i.s ++ a.s} ;
```

This may be applied to some atomic adverbs, e.g. *wo überall*, but not to all, e.g. **wo nirgends*. Similarly for *wie gern*, *wie oft*. The construction can also be used to ask for the value of an adverbially used adjective, e.g. *wie gut* or *how well* in *how well did you sleep*:

```
Lang> l AdvIAdv how_IAdv (PositAdvAdj good_A)
how well
wie gut
```

(In particular, we don't need an additional rule `HowAdv : A -> IAdv`.)

Remark 95: One may also ask for the difference of values on the implicit scale provided by an (adverbially used) adjective, e.g. *wie viel besser geht es dir heute (als gestern)?* or *wieviel lieber trinkst du Wein als Bier*? There is an interrogative adverb

```
how8much_IAdv = {s = "wieviel"} ;
```

in `StructuralGer`, which at least gives

```
Lang> l AdvIAdv how8much_IAdv (ComparAdvAdj more_CAdv good_A (MassNP (UseN bread_N)))
how much more well than bread
wieviel besser als Brot
```

Todo 48: So, we might need a rule `HowmuchA : A -> Adv` that uses the comparative degree of the adjective to form an adverb, e.g. *how much better than – as*. Or else the above modification `AdvIAdv iadv adj : IAdv` should be able to use a degree of `adj` depending on the `iadv`. Also, it should be possible to drop the comparison part. ◁

Q63: It seems that we need to distinguish (interrogative) adverbs in positive from adverbs in comparative degree, and both are split phrases: *Johann ist genau so schnell gefahren wie du* and *Johann ist schneller gefahren als du*, as well as *wieviel schneller ist Johann gefahren als du?*

5.4.7. Interrogative Verb Phrases

Todo 49: Implement the four constructions of interrogative verb phrases `QVP`, `ComplSlashIP`, `QuestQVP`, `AdvQVP` and `AddAdvQVP`. (Lincat `QVP` = ?)

5.4.8. Interrogative Complements to Copula Verbs

Finally, ***interrogative complements of copula verbs*** vary in agreement, so they can be adapted to the agreement features of the nominal subject, e.g. *was für ein Narr ist Johann*, *was für Narren waren wir* for number, but in some contexts also for gender (or rather sex), e.g. **was für eine Mutter ist Johann*, though *was für eine Bestie|Flasche ist er* is grammatical.

```
IComp = {s : Agr => Str ; ext : Str} ;
```

In `Lang`, there are only two ways to construct interrogative complements:

```
CompIAdv  : IAdv -> IComp ;          -- where (is it)
CompIP    : IP   -> IComp ;          -- who (is it)
```

implemented by

```
CompIAdv a = {s = \\_ => a.s ; ext = ""} ;
CompIP ip = {s = \\_ => ip.s ! Nom ; ext = "" } ;
```

The second one also allows interrogative complements as e.g. *which car* in *which car do you like*.

A further construction from an interrogative adjective phrase

```
ICompAP : AP -> IComp ;   -- "how old"
```

is given in `Extend` resp. `Extra`. However, it can also be used with adjective phrases in comparative or superlative degree and then gives poor results, e.g. *how much older* as an `IComp`.

`Extend` also provided a construction from an interrogative quantifier,

```
CompIQuant : IQuant -> IComp ;   -- which (is it) [agreement to NP]
```

Constructions of Interrogative Clauses

First, interrogative Yes-No-clauses, in all moods, tenses, anteriorities and polarities, are built from a clause by

```
QuestCl     : Cl -> QCl ;            -- does John walk
```

They begin with a finite verb (and should end in a question mark *?*), but as subordinate clauses, start with *ob* (eng. *whether*) and have the finite verb at the end:

```
QuestCl cl = {
  s = \\m,t,a,p =>
        let cls = cl.s ! m ! t ! a ! p
        in table {
          QDir   => cls ! Inv ;
          QIndir => "ob" ++ cls ! Sub
          }
  } ;
```

The second construction of an interrogative clause

```
QuestVP     : IP -> VP -> QCl ;      -- who walks
```

puts an interrogative pronoun as nominal or prepositional subject in front of a verb prase:

```
QuestVP ip vp = {
  s = \\m,t,a,p =>
    let
      who = appPrep vp.c1 ip.s ;
      cl = (mkClause who (agrP3 ip.n) vp).s ! m ! t ! a ! p ;
    in table {
        QDir   => cl ! Main ;
        QIndir => cl ! Sub
        }
  } ;
```

Remark 96: In `QuestionGer`, the subject string was `ip.s ! Nom`, so the rule did not work for (passive) verb phrases with prepositional subject, like *auf wen wird gewartet*, or with (active) verbs with non-nominative subject, e.g. *wen friert*, *wem ist kalt*. (But *frieren* can also have nominative subject: *ich friere*, *man fror*.)

Third, an interrogative clause can be built with

```
QuestSlash  : IP -> ClSlash -> QCl ; -- whom does John love
```

by adding an interrogative nominal or prepositional object in front of an incomplete clause:

```
QuestSlash ip slash = {
  s = \\m,t,a,p =>
        let
          cls = slash.s ! m ! t ! a ! p ;
```

```
            who = appPrep slash.c2 ip.s ;
          in table {
            QDir   => who ++ cls ! Inv ;
            QIndir => who ++ cls ! Sub
            }
    } ;
```

This rule (and the original implementation of `SlashVP` using `mkClause`) is used in parsing e.g. *wen liebt ihr*, *welche Bücher habt ihr gelesen* or *auf was wartet ihr*.

```
Lang> l (QuestSlash whoSg_IP (SlashVP (UsePron youPl_Pron) (SlashV2a love_V2)))
whom do you love
wen liebt ihr
Lang> l (QuestSlash whatSg_IP (SlashVP (UsePron youPl_Pron) (SlashV2a wait_V2)))
what do you wait for
auf was wartet ihr
```

Todo 50: Does the modified implementation of `SlashVP` in `TestLangGer` using `mkClSlash` allow us to parse reflexives correctly, i.e. examples like *welches seiner (eigenen) Werke liebt man am meisten*, *wer liebt seine (eigenen) Kinder nicht*?

Fifth, interrogative clauses can also consist of an interrogative adverb followed by a clause:

```
QuestIAdv   : IAdv -> Cl -> QCl ;    -- why does John walk
```

In direct questions, the finite verb follows the interrogative adverb, e.g. *warum geht Johann*, in subordinate question, the finite verb is at the end, e.g. *(ich weiß,) warum Johann geht*:

```
QuestIAdv iadv cl = {
  s = \\m,t,a,p =>
        let
          cls = cl.s ! m ! t ! a ! p ;
          why = iadv.s
        in table {
          QDir   => why ++ cls ! Inv ;
          QIndir => why ++ cls ! Sub
          }
  } ;
```

Copula verbs can be combined to clauses with complements of category `Comp`, which can be an `AP`, `NP`, `Adv` or `CN`, and nominal subjects. Accordingly, there is a rule

```
QuestIComp  : IComp -> NP -> QCl ;   -- where is John
```

to construct an interrogative clause by combining a copula verb and its nominal subject with an ***interrogative complement*** `IComp`, which can be an interrogative `AP`, e.g. *wie klug bist du*, an interrogative `NP` resp. `IP`, e.g. *wer bin ich*, an interrogative `Adv` resp. `IAdv`, e.g. *wo seid ihr*, an interrogative `CN`, e.g. *was für ein Mensch ist Johann*, *ein wie alter Junge ist Johann*.[109] The

[109] Recall that `Lang` has no class of copula verbs; the verbal phrase is always formed with *to be*. Aarne: Other copula verbs, e.g. *to become* or *to remain*, may not be combined with an arbitrary `Comp` or `IComp`: *here|where is John*, but **here|where becomes John*.

interrogative complement can be splittable and admits extraction of a part, e.g. *wieviel älter bist du, als wir vermutet haben.*

The implementation of `QuestIComp` puts the extractable part of the given interrogative complement `icomp` into the `ext`-field of the verb phrase `vp` built from the copula verb, combines the nominative form of the given noun phrase `np` with the `vp` to a clause `cls`, and prefixes this clause with the non-extracted part of `icomp` adapted to the agreement features of the `np`:

```
QuestIComp icomp np = {
  s = \\m,t,a,p =>
        let
          vp  = predV sein_V ** {ext = icomp.ext};
          subj = mkSubject np vp.c1 ;
          cls = (mkClause subj.s subj.a vp).s ! m ! t ! a ! p ;
          why = icomp.s ! np.a
        in table {
          QDir   => why ++ cls ! Inv ;
          QIndir => why ++ cls ! Sub
          }
  } ;
```

Construction of Questions

The only way to construct a ***question*** or ***interrogative sentence*** is by fixing the tempus and polarity arguments of an interrogative clause:

```
UseQCl : Temp -> Pol -> QCl -> QS ;  -- who had not slept
```

This construction is implemented, like the corresponding one for sentences on p.142, by selecting from the paradigm `cl.s` of an interrogative clause `cl` the strings for given values `t.m` of mood, `t.t` of tense, `t.a` of anteriority and `t.p` of polarity, which are selected from arguments `t:Temp` and `p:Pol`:

```
UseQCl t p cl = {
  s = \\q => t.s ++ p.s ++ cl.s ! t.m ! t.t ! t.a ! p.p ! q
  } ;
```

Recall from p.142 that `t.s` and `p.s` are empty strings.

There are no modification rules for interrogative clauses and questions.

5.4.9. Imperatives

Category

The implementation type of the category `Imp` of imperatives

```
Imp = {s : Polarity => ImpForm => Str} ;
```

consists of an inflection paradigm varying in polarity and imperative form. There are singular and plural forms, and polite and familiar forms, (given in `common/ParamX.gf`):

```
param
  ImpForm = ImpF Number Bool ;  -- True = polite, False = familiar.
```

There are two familiar forms in each polarity, e.g. *schäme dich (nicht)*, *schämt euch (nicht)*, and two polite forms, e.g. *schämen Sie sich (nicht)*, *helfen Sie einander (nicht)*. Notice that for polite forms, a difference in number is apparent in the use of reflexive versus reciprocal pronoun in the verb phrase.

Construction of Imperatives

```
-- An imperative is straightforwardly formed from a verb phrase.
-- It has variation over positive and negative, singular and plural.

    ImpVP     : VP -> Imp ;              -- love yourselves
```

A nicer example would be *love each other*, but reciprocal pronouns are not part of `Lang`.

To build an imperative `ImpVP vp` from a verb phrase `vp`, the function `ImpVP` first generates the finite verb forms of the verb phrase, in particular the imperative verb forms (contained in `vps = useVP vp`), and then constructs a paradigm varying in polarity `pol` and imperative form `n`: for familiar imperative form, the second person imperative verb form in the number given by `n` is chosen, with separable verb prefixed separated. The imperative verb form comes in front, the negation, objects, adverbs and infinite verb part follow.

```
ImpVP vp =  let vps = useVP vp in {
  s = \\pol,n =>
    let
      ps = case n of {
        ImpF _ True => <P3,"Sie",True> ; -- setzen Sie sich
        _ => <P2,[],False>
        } ;
      agr  = Ag Fem (numImp n) ps.p1 ; --- g does not matter
      verb = vps.s ! False ! agr ! VPImperat ps.p3 ;
      inf  = vp.inf.inpl.p2 ++ verb.inf ;  -- HL .s/.inpl.p2
      obj  = (vp.nn ! agr).p2 ++ (vp.nn ! agr).p3 ++ (vp.nn ! agr).p4 ++ vp.adj
    in
    verb.fin ++ ps.p2 ++ (vp.nn ! agr).p1
             ++ vp.a1 ++ negation ! pol ++ obj ++ vp.a2 ++ inf ++ vp.ext
} ;
```

The parameters `pol:Polarity` of imperatives is fixed by turning imperatives into utterances, using `UttImpSg` and `UttImpPl` from the module `Phrase`, p.168. In particular, since `CatGer.linref VP` uses positive polarity as default, imperatives with positive polarity can be parsed as `Imp`, but those with negative polarity can only be parsed as `Utt`.

```
Lang> p -cat=Imp "komm"
ImpVP (UseV come_V)
Lang> p -cat=Imp "komm nicht"
The parser failed at token 2: "nicht"
Lang> p -cat=Utt "komm nicht"
UttImpSg PNeg (ImpVP (UseV come_V))
```

Remark 97: Gender `g` does not matter, because the singular polite imperative form `ImpF Sg True` is never used. If it were, we had `agr = Ag g Sg P3`, so a possessive object `(vp.nn ! agr).p2` depended on gender, e.g. *seine Pflicht* versus *ihre Pflicht* (eng. *one's duty*). For the plural polite imperative form `ImpF Pl True` we need a "polite possessive pronoun", which `Lang` does not provide: *tu deine Pflicht*, *tut eure Pflicht*, *tun Sie Ihre Pflicht*. Actually, number is irrelevant for the polite imperative in German; the parameter type could be a 3-valued domain:

```
ImpForm = Familiar Number | Polite
```

Todo 51: Maybe the adverb `vp.a2` comes too late, or `always_AdV` has to be inserted into `vp.a1`. More testing is needed. Some mistakes are

1. A predicate with copula verb and reflexive nominative complement like *to always be oneself* = *immer man selbst sein* has the personal pronoun as part of the complement, so the imperatives are *sei immer du selbst*, *seid immer ihr selbst*, *seien Sie immer Sie selbst*.

   ```
   Lang> gr -tr -cat=Imp ImpVP ? | l
   ImpVP (SelfAdvVP (AdVVP always_AdV UseCopula))
   always be yourself
   sei immer selbst
   ```

   For `vp = SelfAdvVP (...)`, we have `vp.a2 = immer selbst`. To fix this, we would need `vp.a2 : Agr => Str` instead of `vp.a2 : Str`.

Remark 98: Modal verbs do not have imperatives. Instead of *müsse singen*, a sentence (with emphasis on the infinite verb) is used: *du mußt singen*, *du kannst gehen*, *du darfst schlafen*, *du willst arbeiten*.[110] But copula verbs do have imperatives: *sei still*, *bleibe hier*, *werde erwachsen*.

Modification of Imperatives

The abstract grammar of `Lang` has a rule to modify imperatives by adverbs:

```
AdvImp : Adv -> Imp -> Imp ;          -- please love yourselves
```

However, the *please* in the example shown is not an adverb, but a vocative: `please_Voc:Voc`. Vocatives are added at the end, not at the beginning, of an utterance by a phrase construction `PhrUtt : PConj -> Utt -> Voc -> Phr` in the module `Phrase`. It might be reasonable to allow for a vocative at the beginning or in the midst of imperatives, i.e. to introduce rules

[110] A tree transformation can handle this, see Section **??**.

```
VocImp : Voc -> Imp -> Imp ;     ImpVocVP : Voc -> VP -> Imp ;
```

to get *bitte, Herr Ober, bringen Sie mir ein Bier* and *bringen Sie mir, bitte, ein Bier.*

The rule `AdvImp` is implemented for `Eng`, `Bul`, `Romance`, and `Scand` only, mostly by

```
AdvImp adv imp = {
  s = \\pol,impform => adv.s ++ imp.s ! pol ! impform
} ;
```

Since the verb phrase argument of `ImpVP : VP -> Imp` may already contain an adverb, a rule to add an adverb to an imperative seems unnecessary.

But: there are imperatives with a leading conditional: *wenn du Hilfe brauchst, ruf mich an*, and imperatives where the adverb follows the verb: *üb' immer Treu' und Redlichkeit* or *quäle nie ein Tier im Scherz.*

5.5. Conjunction (incomplete)

For many categories, a list of two or more expressions of this category can be conjoined to an expression of the category, mainly by putting a conjunction like `and_Conj:Conj` between the final two elements and a comma between the remaining elements of the list.

For any conjoinable category `C` there is a category `ListC` and a conjoin construction

```
ConjC : Conj -> ListC -> C ;
```

and two list constructors

```
BaseC : C -> C -> ListC ;
ConsC : C -> ListC -> ListC ;
```

Implementations of these are provided in `ConjunctionGer.gf` or `ExtendGer.gf`. A conjunction word is a split string, e.g. ⟨*entweder,oder*⟩ or ⟨*sowohl,als auch*⟩. The implementation category for conjunctions therefore is

```
Conj = {s1,s2 : Str ; n : Number} ;
```

where the field `n:Number` is relevant only for noun phrase coordination. Some examples of conjunction words are given in `StructuralGer`,

```
and_Conj = {s1 = [] ; s2 = "und" ; n = Pl} ;
or_Conj = {s1 = [] ; s2 = "oder" ; n = Sg} ;
if_then_Conj = {s1 = "wenn" ; s2 = comma ++ "dann" ; n = Sg} ;
```

We begin with conjunctions for a category with simple implementation type `{s : Str}`.

Conjunction of interrogative adverbs

Because the implementation category of interrogative adverbs is `IAdv = {s : Str}`, the implementation category of `ListIAdv`, with special notation `[IAdv]`, is

```
[IAdv] = {s1,s2 : Str} ;
```

Here, `s2` is the field for the final element of the list. The constructors are implemented by

```
BaseIAdv x y = {s1 = x.s ; s2 = y.s} ;
ConsIAdv x xs = {s1 = x.s ++ comma ++ xs.s1 ; s2 = xs.s2} ;
ConjIAdv conj ss = {s = conj.s1 ++ ss.s1 ++ conj.s2 ++ ss.s2} ;
```

For example, a conjunction of three interrogative adverbs is as follows:

```
TLang> l (ConjIAdv and_Conj (ConsIAdv where_IAdv (BaseIAdv when_IAdv how_IAdv)))
where , when and how
wo , wann und wie
```

Conjunction of adverbs

The conjunction of adverbs is implemented similarly. (GF provides some support to ease such implementations, which we don't use here.)

```
TLang> l ConjAdv neither7nor_DConj
  (BaseAdv (PrepNP in8front_Prep (DetCN (DetQuant DefArt NumSg) (UseN house_N)))
           (PrepNP in_Prep (DetCN (DetQuant DefArt NumSg) (UseN garden_N))))
neither in front of the house nor in the garden
weder vor dem Haus noch im Garten
```

So far, there is no possibility to separate the initial from the final part of the conjoined adverb, as e.g. in *wir haben weder im Haus gearbeitet noch im Garten.*

Conjunction of prepositions (todo)

A coordination of prepositions expecting the same case is quite common, e.g. *das Buch lag weder auf noch unter dem Tisch.* But since the category `Prep` does not fix the case governed by prepositions, this restriction cannot be implemented in the coordination of prepositions.

Q64: Since there are several types of `Preposition`, i.e. cases, pre-, post-, circumpositions, and contracting prepositions, a coordination rule would probably be overgenerating in unexpected ways. Better leave it?

Conjunction of determiners

It seems that for `GF`, roughly, the initial part of a (simple) noun phrase obtained by dropping the noun is considered as a determiner with adjective phrase, or `DAP = NP/N`, e.g. *the third best book* $\mapsto$ *the third best.* Since `DAP` has paradigms `s`,`sp` for usage with following adjective phrase and for stand-alone usage, the implementation type for `ListDAP` is

```
[DAP] = {s1,sp1,s2,sp2 : Gender => Case => Str ; n : Number ; a : Adjf} ;
```

(We ignore here the contraction of prepositions with a leading definite article in singular and hence omit the flags `hasDefArt` and `isDef`.) The list constructors concatenate forms for each paradigm separately:

```
BaseDAP x y = {
  s1 = x.s ; sp1 = x.sp ; s2 = y.s ; sp2 = y.sp ; n = y.n ; a = y.a} ;
```

```
ConsDAP x xs = {
  s1 = \\g,c => x.s!g!c ++ comma ++ xs.s1!g!c ;
  sp1 = \\g,c => x.sp!g!c ++ comma ++ xs.sp1!g!c ;
  s2 = xs.s2 ; sp2 = xs.sp2 ; n = xs.n ; a = xs.a} ;
```

The conjoin construction is (preliminarily) implemented by

```
ConjDet conj ss = {
  s = \\b,g,c => conj.s1 ++ ss.s1!g!c ++ conj.s2 ++ ss.s2!g!c ;
  sp = \\b,g,c => conj.s1 ++ ss.sp1!g!c ++ conj.s2 ++ ss.sp2!g!c ;
  n = conj.n ;
  a = Weak ; isDef,hasDefArt = False} ; -- ad hoc
```

It may be better to use the values of the adjective inflection from the last list element and the definiteness values of the first list element. Perhaps, conjunctions like the following hardly occur:

```
TLang> l ConjDet and_Conj
           (ConsDAP (DetDAP (DetQuant DefArt NumSg))
             (ConsDAP (AdjDAP (DetDAP somePl_Det) (PositA brown_A))
               (BaseDAP (DetDAP (DetQuant this_Quant NumSg))
                        (DetDAP few_Det))))

der , einige braune , dieser und wenige
```

Conjunction of adjective phrases (todo)

How should split adjectives `ap = {s:AForm => Str; s2:Case => Str}` be coordinated? A problem here is the conjunction of comparison parts `s2`: *colder than water and warmer than ice* is fine predicatively, but how to use the conjunction attibutively in German? *eine kältere Flüssigkeit als Wasser und wärmere als Eis*? Or rather as in English, with post-nominal attribute: *eine Flüssigkeit, kälter als Wasser und wärmer als Eis*? What if some, but not all conjuncts are split? E.g. *eine rote, aber flüssigere Farbe als Blut*, or *eine rote Farbe, aber flüssiger(e?) als Blut*? If we had a boolean flag to test whether one of the `s2`-fields is non-empty, we could force a postnominal coordinated attribute.

As a compromise, we currently combine the comparison noun phrase (in nominative!) to the inflection part of each conjunct. This gives a wrong case for adjectival objects (of a `V2A`) in

male die Erde sowohl kleiner als die Sonne als auch größer als der Mond

and a strange, but comprehensible prenominal attribute in

viele sowohl kleinere als die Sonne als auch größere als der Mond Sterne

Conjunction of sentences

In the coordination of sentences, it seems that the conjunction can restrict the order of the component sentences. For example, *und* uses the same order for all sentences of the list,

`Main`	*sie liest das Buch und er trinkt das Bier*
`Inv`	*liest sie das Buch und trinkt er das Bier (?)*
`Sub`	*(wenn) sie das Buch liest und er das Bier trinkt*

but ⟨*weder,noch*⟩ or ⟨*einerseits,andererseits*⟩ demand order `Inv` on both conjunct sentences,

`Main`	*weder liest sie das Buch, noch trinkt er das Bier*
`Inv`	*weder liest sie das Buch noch trinkt er das Bier (?) – liest weder ...*
`Sub`	*(wenn) weder sie das Buch liest noch er das Bier trinkt*

Moreover, `if_then_Conj` = ⟨*wenn,dann*⟩ expects `Sub` in the first and `Inv` in the second conjunct:

wenn sie das Buch liest, dann trinkt er das Bier

Remark 99: It seems to me that ⟨*wenn,dann*⟩ is not a conjunction, but a subjuctor *wenn* or *falls* (constructing an adverbial sentence) and an adverb correlate *dann*, so that *falls sie das Buch liest, dann* is an adverb in a basic sentence. ◁

The current implementation of sentence conjunction is:

```
BaseS x y = { -- twoTable Order ;
  s1 = x.s ;
  s2 = table {Inv => y.s ! Main ; o => y.s ! o}
  } ;
ConsS x xs = { -- consrTable Order comma ;
  s1 = \\o => x.s ! Inv ++ comma ++ xs.s1 ! case o of {Inv => Main ; _ => o} ;
  s2 = xs.s2
  } ;
ConjS conj ss = conjunctDistrTable Order conj ss ;
```

But this gives, for example, at least an incorrect inverse order:

```
TLang> p -cat=S "Mary reads the book and John drinks the beer" | l -table
s : Mary reads the book and John drinks the beer
s Main : Maria liest das Buch und Johann trinkt das Bier
s Inv : liest Maria das Buch und Johann trinkt das Bier
s Sub : Maria das Buch liest und Johann das Bier trinkt
```

So it seems that in `Sub` resp. `Inv` order, both component sentences have to be in `Sub` resp. `Inv` order, but in `Main` order, the order of the component sentences is determined by the conjunct.

Q65: Do we have to change the implementation type of `Conj` to

```
Conj = {s1,s2 : Str ; n : Number ; o1,o2 : Order}
```

and adapt the constructors for sentence coordination?

Conjunction of infinitives (todo)

Expl. *Um die Region zu mythisieren, musste Kundera nicht nur deren Gegenwart verkennen, sondern auch ihre Geschichte kräftig schönen.* (der Freitag, 18. Januar 2024)

Conjunction of noun phrases

Nominal objects can be coordinated, but to parse them as noun phrases, they have to be given in nominative (by the restriction in `linref NP`). For example, *nicht nur die Frauen , sondern auch die Männer* has the tree

```
ConjNP notonly_butalso_Conj
       (BaseNP (DetCN (DetQuant DefArt NumPl) (UseN woman_N))
               (DetCN (DetQuant DefArt NumPl) (UseN man_N)))
```

Q66: How can we implement a coordination of prepositional objects, e.g. *(wir warten) weder auf dich noch auf ihn* or *(er arbeitet) sowohl für die Firma als auch für sich selbst*? Since GF has no category of prepositional phrases, it is not obvious how to handle this.

Conjunction of imperatives

Since the linearization category of imperatives is `Imp = {s : Polarity => ImpForm => Str}`, the implementation of conjoined imperatives is clear:

```
lincat
  [Imp] = {s1,s2 : Polarity => ImpForm => Str} ;
lin
  BaseImp x y = {s1 = \\p,f => x.s ! p ! f ; s2 = \\p,f => y.s!p!f} ;
  ConsImp x xs =
    {s1 = \\p,f => x.s ! p ! f ++ comma ++ xs.s1 ! p ! f ; s2 = xs.s2} ;
  ConjImp conj xs =
    {s = \\p,f => conj.s1 ++ xs.s1 ! p ! f ++ conj.s2 ++ xs.s2 ! p ! f} ;
```

Here is an example:

```
TLang> l ConjImp and_Conj
  (BaseImp (AdvImp now_Adv (ImpVP (UseComp (CompAP (PositA froh_A)))))
           (ImpVP (UseV laugh_V)))

jetzt sei froh und lache
```

5.6. Phrase

The module `Phrase` contains, among other things, constructions of ***utterance***s. These are expressions of category `Utt`, which has the implementation type

```
Utt = {s : Str} ;
```

There are imperatives to a single or to several persons, and polite imperatives in plural:

```
UttImpSg  : Pol -> Imp -> Utt ;         -- (don't) love yourself
UttImpPl  : Pol -> Imp -> Utt ;         -- (don't) love yourselves
UttImpPol : Pol -> Imp -> Utt ;         -- (don't) sleep (polite)
```

These are implemented by

```
UttImpSg pol imp = {s = pol.s ++ imp.s ! pol.p ! ImpF Sg False} ;
UttImpPl pol imp = {s = pol.s ++ imp.s ! pol.p ! ImpF Pl False} ;
UttImpPol pol imp = {s = pol.s ++ imp.s ! pol.p ! ImpF Sg True} ;
```

5.7. Text

The module `Text` of `Grammar` is implemented in `GrammarGer` by a restriction

```
TextX - [Tense,Temp,Adv,CAdv],
```

of a language-independent concrete module `TextX` which extends a module `CommonX` of implementation types like

```
Text = {s:Str} ;
Phr = {s:Str} ;
```

and implementation types for `Tense`, `Temp`, `Adv`, `CAdv` that are changed for German in `CatGer`. The text constructors

```
TEmpty : Text ;
TFullStop, TQuestMark, TExclMark : Phr -> Text -> Text ;
```

are implemented by

```
concrete TextX of Text = CommonX ** open Prelude in {
  lin
    TEmpty = {s = []} ;
    TFullStop x xs = {s = x.s ++ SOFT_BIND ++ "." ++ xs.s} ;
    TQuestMark x xs = {s = x.s ++ SOFT_BIND ++ "?" ++ xs.s} ;
    TExclMark x xs = {s = x.s ++ SOFT_BIND ++ "!" ++ xs.s} ;
}
```

Here, the token `SOFT_BIND` instructs the parser to glue the period, question mark or exclamation mark to the string `x.s` of the argument `x:Phr` without inserting a space.

5.8. Structural (todo)

Remark 100. *The implementations of the pre-determiners in* ***`StructuralGer`***

```
all_Predet  = {s = appAdj (regA "all") ; c = noCase ; a = PAgNone} ;
most_Predet = {s = appAdj (regA "meist") ; c = noCase ; a = PAgNone} ;

oper noCase : {p : Str ; k : PredetCase} = {p = [] ; k = NoCase} ;
```

do not work properly, yielding "all myself" = "aller ich" and "most myself" = "meister ich". These combinations do not make much sense. At least we should have a plural agreement in

```
all_Predet = {s = appAdj (regA "all") ; c = noCase ; a = PAg Pl} ;
```

(or do we want to allow "all mein schönes Geld", too?) Then we get "ich liebe alle meine jungen Kinder", although likewise "alle mein junges Kind". Maybe we could let ***`all_Predet`*** *be empty in combination with singular forms of* ***`ReflPron`****, and "wir|ihr|sie alle" in plural.*

It seems that ***`most`****, "die meisten" in German, is more a quantifier than a pre-determiner.*

```
most_Predet = {
  s = \\n,g,c => let gn = MorphoGer.gennum g n ;
                    adj = (P.mkA "viel" "mehr" "meiste").s ! Superl
                in
                    MorphoGer.artDef ! gn ! c ++ adj ! (agrAdj Weak gn c) ;
  c = {p = [] ; k = PredCase Gen} ;
  a = PAg Pl} ;
```

By the test for pronoun in `PredetRNP`*, this gives "die meisten von uns", but "die meisten meiner kleinen Kinder". (*`MorphoGer` *subsumes* `ResGer`*.)*

The prepositions – c.f. Remark 33

```
possess_Prep = P.von_Prep ; -- mkPrep "von" P.dative ;
part_Prep    = P.von_Prep ; -- mkPrep "von" P.dative ;
```

are obsolete; they can be used with `PrepNP prep np` to build an adverbial, which is incorrect. Better use the possessive and partitive constructions `PossNP,PartNP : CN -> NP -> CN` and load `Structural - [possess_Prep, part_Prep]` in a grammar used for parsing.

The determiner `someSg_Det:Det` (*eng. some*) is implemented by

```
someSg_Det = {
    s = \\_,g,c => "ein" + pronEnding ! GSg g ! c ;  -- ein, eine, ein
    sp = \\_,g,c => "ein" + detEnding ! GSg g ! c ;  -- einer, eine, eines
    n = Sg ; a = Mixed ; isDef = False ; hasDefArt = False
} ;
```

It has the same forms as the construction `DetQuant IndefArt NumSg` (*eng. a/one*) from the indefinite article in singular (up to the irrelevant `aPl:Adjf`). This causes unnecessary ambiguities in parsing; perhaps we should indicate intonation, using `"eín"` here, to avoid these ambiguities.

Remark 101: There are three "*phrase-beginning conjunctions*" in `Structural`,

```
  but_PConj = {s = "aber"} ;
  otherwise_PConj = {s = "sonst"} ;
  therefore_PConj = {s = "deshalb"} ;
```

This cannot work for sentences, since *aber* expects `Main` sentence order, the other two expect `Inv` sentence order. But `UttS` forces `Main` order, leading to wrong orders for *deshalb* and *sonst*:

```
Lang> p -tr -lang=Eng "therefore she laughs" | l
PhrUtt therefore_PConj (UttS (UseCl (TTAnt TPres ASimul) PPos
                          (PredVP (UsePron she_Pron) (UseV laugh_V)))) NoVoc
therefore she laughs
deshalb sie lacht
```

In German, *deshalb* is an adverb, not a `PConj`. Q67: Does `PConj` make sense for German, and can we expect a strict correspondence between `PConj` expressions of different languages? One can put *aber* or *und* in front of a sentence in `Main` order (so these are `PConj`s), e.g. the first sentence of [4], *Aber Jakob ist immer quer über die Gleise gegangen.* ◁

5.9. Idiom (todo)

Discuss at least

```
ImpersCl vp = mkClause "es" (agrP3 Sg) vp ;  -- it is cold
GenericCl vp = mkClause "man" (agrP3 Sg) vp ;
```

For German, we don't need to add an ***indefinite personal pronoun*** *man* in the lexicon, since reflexive and possessive forms are the same as those of *er* and obtained from the agreement value `agrP3 Sg`. (But in English, there are special forms *oneself* and *one's*, so at least a separate agreement value is needed.) Notice that although *man* has singular agreement, it can be used with plural reciprocal pronoun: *man hilft einander*.

Check: A generic clause `GenericCl vp` is a simple clause; sentences like *wenn man ..., dann ... man ...* have to be built using `SubjS` and two generic clauses.

Q68: What about *man=einer*, in *das können die doch nicht mit einem machen!*

Q69: Are there rules in `Grammar` (or `Extend`) that care about correlates? i.e. sentences with an additional (non-complement) *es* or *das* at the "original position" from where a sentential complement is moved? It may be a subject sentence, as in *es ist seltsam, dass die Erde nicht flach ist*, or an object senctence, as in *wir glauben es|das nicht, dass die Erde eine Scheibe ist.* Likewise with infinitival complements: *es ist schön, im Meer zu schwimmen* and *wir glauben es kaum, euch schon wieder zu treffen.*

Bug 1: There is no difference between the *it* in an `ImpersCl` and the personal pronoun *it*. While an infinitival subject is recognized correctly,

```
Lang> p -cat=Cl "to sleep is good"
PredSCVP (EmbedVP (UseV sleep_V)) (UseComp (CompAP (PositA good_A)))
```

when the infinitival subject is moved and replaced by a correlate *it*, we get the wrong trees

```
Lang> p -cat=Cl "it is good to sleep"
ImpersCl (UseComp (CompAP (SentAP (PositA good_A) (EmbedVP (UseV sleep_V)))))
PredVP (DetNP (DetQuant DefArt NumSg))
         (UseComp (CompAP (SentAP (PositA good_A) (EmbedVP (UseV sleep_V)))))
PredVP (UsePron it_Pron)
         (UseComp (CompAP (SentAP (PositA good_A) (EmbedVP (UseV sleep_V)))))
```

What is wrong here is that `SentAP` builds an adjective phrase *good to sleep*, which can then be turned into a verb phrase

```
(UseComp (CompAP (SentAP (PositA good_A) (EmbedVP (UseV sleep_V)))))
```

which can be combined with the expletive *it* of *it rains* (and `ImpersCl`) as well as the pronoun *it*, as well as any noun phrase as subject. This would similarly allow a predicate *hard to follow* in *your argument is hard to follow*, which must be analyzed as a version of *it is hard to follow your argument* or *to follow your argument is hard*. Similarly: *a good man is hard to find*.

But this gives Chomsky's *John is easy to please* and *John is eager to please* examples:

```
Lang> p -cat=Cl "John is good to see"
```

```
PredVP (UsePN john_PN)
  (UseComp (CompAP (SentAP (PositA good_A)
                     (EmbedVP (ComplSlash (SlashV2a see_V2)
                                          (DetNP (DetQuant IndefArt NumPl)))))))
Lang> p -cat=Cl "John is good to sleep"
PredVP (UsePN john_PN)
  (UseComp (CompAP (SentAP (PositA good_A)
                      (EmbedVP (UseV sleep_V)))))
```

The number agreement in *good men are hard to find* indicates that this is a version of object-to-subject raising.

5.10. Tense

Recall from Section 3.1.15 that the grammar `Lang` has syntactic catetegories `Temp`, `Tense`, `Ant` and `Pol` that are used to select 16 = 4 x 2 x 2 forms of clauses for all languages in the library. Each clause form is defined by a selection of one out of four values for tense, two for anteriority and two for polarity. Eight "expressions" of the syntactic category `Temp` are constructed by

```
TTAnt : Tense -> Ant -> Temp ;
```

from the four constants `TPres`, `TPast`, `TFut`, `TCond` of `Tense` and two constants `ASimul`, `AAnter` of `Ant`. The module `abstract/Tense.gf` also provides two constants `PPos`, `PNeg` of `Pol`.

For German, the implementation categories of `Temp`, `Tense`, `Ant` and `Pol` are

```
lincat
  Temp  = {s : Str ; t : ResGer.Tense ; a : Anteriority ; m : Mood} ;
  Tense = {s : Str ; t : ResGer.Tense ; m : Mood} ;
  Ant   = {s : Str ; a : Anteriority} ;
  Pol   = {s : Str ; p : Polarity} ;
```

Besides a field `s` for a string, the records of these types have fields `t`, `a`, `m` or `p` for a value of the parameter types (c.f. Section 5.4.2)

```
param
  Tense = Pres | Past | Fut | Cond ;
  Anteriority = Simul | Anter ;
  Mood  = MIndic | MConjunct ;
  Polarity    = Pos | Neg ;
```

The constants of categories `Ant` and `Pol` are interpreted by empty strings `[] = ""` and corresponing parameters (via a module `TenseX`):

```
lin
  ASimul = {s = [] ; a = Simul} ;
  AAnter = {s = [] ; a = Anter} ; --# notpresent
  PPos  = {s = [] ; p = Pos} ;
  PNeg  = {s = [] ; p = Neg} ;
```

In `Grammar`, the syntactic categories `Tense` and `Ant` are used only as argument categories of `TTAnt : Tense -> Ant -> Temp`, and only `Temp` and `Pol` are argument categories of other syntactic constructions (namely, `UseCl`, `UseQCl`, `UseRCl` and `UseSlash`). The four constants of category `Tense` and the construction `TTAnt` to interpret the 16 values of category `Temp` are implemented as follows:

```
concrete TenseGer of Tense =
  CatGer [Tense,Temp], TenseX [Ant,Pol,AAnter,ASimul,PNeg,PPos]
  ** open ResGer in {

  lin
    TTAnt t a = {s = t.s ++ a.s ; t = t.t ; a = a.a ; m = t.m} ;

    TPres = {s = [] ; t = Pres ; m = MIndic} ;
    TPast = {s = [] ; t = Past ; m = MIndic} ;   --# notpresent
    TFut  = {s = [] ; t = Fut  ; m = MIndic} ;   --# notpresent
    TCond = {s = [] ; t = Cond ; m = MIndic} ;   --# notpresent

}
```

The paradigm `temp.s` of any expression `temp = TTAnt t a : Temp` is the empty string, `temp.s = t.s ++ a.s`. Likewise, the paradigm `p.s` of the two expressions `p:Pol` is the empty string. Hence, an application of `UseCl : Temp -> Pol -> Cl -> S` can turn any clause `cl:Cl` into a sentence `s:S` by parsing empty substrings of the input to any of the eight values of `Temp` and two values of `Pol`. For example, we have

```
Lang> p -cat=S "wir schlafen nicht"
UseCl (TTAnt TPres ASimul) PNeg (PredVP (UsePron we_Pron) (UseV sleep_V))
```

Notice that the four values of category `Tense` are implemented by four values of the linearization category `TenseGer` that have indicative mood `MIndic` built in, but there are four more values with conjunctive mood `MConjunct`. The paradigm of a clause,

```
    Cl = {s : Mood => ResGer.Tense => Anteriority => Polarity => Order => Str} ;
```

covers 2 x 4 x 2 x 2 = 32 sentences, i.e. paradigms of type `Order => Str`. To parse sentences in conjunctive mood, one can add values like (c.f. `ExtraGer`)

```
    TImpfSubj  = {s = [] ; t = Past ; m = MConjunct} ;   --# notpresent
```

to the category `Tense`, but these would not be linearized to other languages in the library.

5.11. Extension of `LangGer` to `AllGer`

There is a module `abstract/Extend.gf` with declarations of constructions that are available in several languages, but perhaps cannot be implemented for all languages of the resource grammar library. Implementations of the common abstract language `Grammar,Extend` can be obtained by means of a default implementation `common/ExtendFunctor.gf` of the declarations

in `Extend`.[111] The language-specific implementation `ExtendGer` overwrites those default implementations where needed.

There is also a module `ExtraGer` of constructions declared in `ExtraGerAbs` that are specific for German.[112] This leads to a grammar

```
abstract AllGerAbs =
  Lang,
  IrregGerAbs,
  Extend --, Extra
  ** {} ;
```

with implementation

```
concrete AllGer of AllGerAbs =
  LangGer,
  IrregGer,
  ExtendGer
  ** open ExtraGer in {} ---- to force compilation
  ;
```

Notice that in order to be able to linearize trees obtained with `AllGer` to other languages of the library, the target languages must share the same abstract grammar `AllGerAbs`, although they need not provide implementations for the irregular verbs of `IrregGerAbs`.

5.11.1. `ExtendGer` (partial)

The construction

```
GenNP       : NP -> Quant ;       -- this man's
```

generalizes `PossPron : Pron -> Quant`. It is implemented in `ExtendGer` as

```
GenNP np = {
  s,sp = \\gn,c => np.s ! False ! Gen ++ np.ext ++ np.rc ;
  a = Strong ;
  isDefArt = False ;
  delCardOne = False
  } ;
```

BUT: the rule leads to many trees. Since `UseDAP` is intended to replace `DetNP`, let us compare the number of trees with either rule, after correcting `DefArt.sp`:

```
AllGerAbs> p -cat=NP "des Mannes Wagen" | ? wc -l                ==> 182
AllGerAbs> p -cat=NP "des Mannes Wagen" | ? grep -v DAP | wc -l  ==> 56
AllGerAbs> p -cat=NP "des Mannes Wagen" | ? grep -v DetNP | wc -l ==> 56
```

[111] See also `gf-rgl/doc/DocExtend.hs` for a script to create a documentation `GF-RGL-Extend.html` showing which functions of `Extend` are implemented for which languages of the RGL.

[112] `Extra` was a previous version of `Extend`. For many languages of the library, the concrete grammars for `Extra` and `Extend` overlap. For German, we have moved all constructions in `Extra` that are also declared in `Extend` to `ExtendGer`, so that `ExtraGer` is small and can be loaded in combination with `ExtendGer`. The plan is to put some constructions providing word-order variations to `ExtraGer`, in particular extractions of sentential complements.

And `dessen alter Wagen` has 228 trees!

```
AllGerAbs> p -cat=NP "dessen alter Wagen" | ? wc -l                 ==> 228
AllGerAbs> p -cat=NP "dessen alter Wagen" | ? grep -v DetNP | wc -l ==> 66
AllGerAbs> p -cat=NP "dessen alter Wagen" | ? grep -v DAP | wc -l   ==> 2
```

The linearization of trees seems correct (all versions with endings -es and -e not shown):

```
AllGerAbs> l -table
DetCN (DetQuant (GenNP (DetCN (DetQuant (PossPron i_Pron) NumSg) (UseN man_N))) NumSg)
      (AdjCN (PositA old_A) (UseN dog_N))
s False Nom : meines Manns alter Hund
s False Acc : meines Manns alten Hund
s False Dat : meines Manns altem Hund
s False Gen : meines Manns alten Hunds
ext :
rc :
```

But one cannot parse all forms, and the number of trees is rather absurd:

```
AllGerAbs> p -cat=Utt "meines Manns alter Hund" | ? wc -l   ==> 2700
AllGerAbs> p -cat=Utt "meines Manns alten Hund" | ? wc -l   ==>  120
AllGerAbs> p -cat=Utt "meines Manns altem Hund" | ? wc -l   ==>    0
AllGerAbs> p -cat=Utt "meines Manns alten Hunds" | ? wc -l ==> 12240

AllGerAbs> p -cat=Utt "meines Manns alte Hunde" | ? wc -l   ==>   60
AllGerAbs> p -cat=Utt "meines Manns alten Hunden" | ? wc -l ==>    0
AllGerAbs> p -cat=Utt "meines Manns alter Hunde" | ? wc -l  ==> C-c C-c thread killed
```

Reflexive Noun Phrases

The abstract module `Extend` (and `Extra`) contains a category `RNP` of ***reflexive noun phrase***s, which is implemented in `ExtendGer` by

```
    RNP = {s : Agr => Case => Str ; rc,ext : Str ; isPron : Bool} ;
```

Reflexive noun phrases admit a verb phrase construction `ReflRNP : VPSlash -> RNP -> VP` that generalizes the construction `ReflVP : VPSlash -> VP` of `Verb`. An `rnp:RNP` cannot be used as subject of a clause, as it has no inherent agreement value.[113] While the inflection table `np.s : Bool => Case => Str` of a noun phrase `np` varies in `Bool` and `Case` to handle contractions of preposition and definite articles, the inflection table `rnp.s : Agr => Case => Str` of a reflexive noun phrase `rnp` varies in agreement and case. The first construction of reflexive noun phrases,

```
    ReflPron : RNP ;                    -- myself (oneself)
```

[113] But then the comment on `a:PredetAgr` in the type of `Predet` below makes no sense! Subjects like *die meisten meiner Brüder* need a fixed agreement value `Ag Masc Pl P3`, so the possessive cannot be constructed with `ReflPoss`, but must be built by `PossPron i_Pron`.

is just the reflexive (non-nominative) usage of personal pronouns, with special form *sich* instead of *ihn, sie, es* in third person singular:

```
ReflPron = {            -- personal pronoun, with "sich" in P3 Sg
  s = ResGer.reflPron ; rc,ext = [] ; isPron = True } ;
```

The second construction is the possessive usage of personal pronouns,

```
ReflPoss : Num -> CN -> RNP ;      -- my family, one's nose
```

A proper implementation needs a case distinction on the construction of `num` by `NumSg`, `NumPl`, or `NumCard card`: *mein kleines Kind*, *meine kleinen Kinder*, but instead of *wenigstens 3 meine Kinder* we need *wenigstens 3 meiner Kinder*. Since this case distinction is impossible in `GF`, to get at least the tolerable *meine wenigstens 3 Kinder* we put the possessive in front:

```
ReflPoss num cn = {            --  HL 5/2022, mixed adjf, Duden 477
   s = \\a,c => let adjf = case num.n of {Sg => Strong | Pl => Weak}
                in possPron a num.n cn.g c ++ num.s ! cn.g ! c
                   ++ cn.s ! adjfCase adjf c ! num.n ! c ++ cn.adv ;
   ext = cn.ext ;
   rc = cn.rc ! num.n ;
   isPron = False } ;
```

It is not quite clear to me what is intended with the third construction

```
PredetRNP : Predet -> RNP -> RNP ; -- all my brothers
```

In `ExtraEng`, an application `PredetRNP pdet rnp` puts the fixed string `pdet.s:Str` in front of the strings of the paradigm `rnp.s : Agr => Str`. This works with the pre-determiner *only*, but for *all* and the reflexive pronoun, is *all we* meant to be replaced by *all of us* or *we all* (and what about *all I* or *most we*)? For `Ger`, we use a flag `RNP.isPron` to allow for a special combination of pre-determiners with `ReflPron` in the implementation by

```
PredetRNP pred rnp = rnp ** {
  s = \\a,c => let n = case pred.a of {PAg n => n ; _ => numberAgr a} ;
                  g = genderAgr a ;
                  d = case pred.c.k of {NoCase => c ; PredCase k => k}
    in case rnp.isPron of {
      True => pred.s ! Pl ! Masc ! c ++ "von" ++ rnp.s ! a ! Dat ;
      _ => pred.s ! n ! g ! c ++ pred.c.p ++ rnp.s ! a ! d} ;
  isPron = False} ;
```

The special case with "*von*" makes only sense for "*die meisten von uns*" or "*alle von uns*", but is wrong for other pre-determines, e.g. gives "*nur von uns*" instead of "*nur wir*". Perhaps, as the name pre-determiner indicates, they should not be used in combination with `ReflPron`.

The construction `PredetRNP` can be used iteratively, which makes sense for *nur nicht ich*, *nicht nur ich*, *nicht alle meine Freunde*, but less or not so for e.g. *die meisten nur meiner Freunde*.

A final construction

```
ConjRNP : Conj -> RNPList -> RNP ; -- my family, John and myself
```

builds a reflexive noun phrase from a list of reflexive (and non-reflexive) noun phrases by a conjunction like *und*, *oder*, or a split conjunction like *sowohl _ als auch _*. The category of lists of reflexive noun phrases is a record of two paradigms,

```
RNPList = {s1,s2 : Agr => Case => Str} ;
```

and the conjunction rule is implemented by a built-in operation `conjunctDistrTable2`:

```
ConjRNP conj rnps = conjunctDistrTable2 Agr Case conj rnps
  ** {isPron = False ; ext,rc = []} ;
```

which constructs a paradigm `s : Agr => Case => Str` by putting a comma or the conjunction `conj` between the strings of the paradigms in the list of $n + 2$ elements. The list constructors

```
Base_rr_RNP : RNP -> RNP -> RNPList ;       -- my family, myself
Base_nr_RNP : NP  -> RNP -> RNPList ;       -- John, myself
Base_rn_RNP : RNP -> NP  -> RNPList ;       -- myself, John
Cons_rr_RNP : RNP -> RNPList -> RNPList ;   -- my family, myself, John
Cons_nr_RNP : NP  -> RNPList -> RNPList ;   -- John, my family, myself
```

are implemented by turning the non-reflexive noun phrases into a constant paradigm (containing sentential or infinitival objects and relative clauses) before combining the paradigms:

```
Base_rr_RNP x y = twoTable2 Agr Case x y ;
Base_nr_RNP x y =
  twoTable2 Agr Case {s = \\_,c => x.s ! False ! c ++ x.ext ++ x.rc} y ;
Base_rn_RNP x y =
  twoTable2 Agr Case x {s = \\_,c => y.s ! False ! c ++ y.ext ++ y.rc} ;

Cons_rr_RNP x xs = consrTable2 Agr Case comma x xs ;
Cons_nr_RNP x xs = consrTable2 Agr Case comma
                     {s = \\_,c => x.s ! False ! c ++ x.ext ++ x.rc} xs ;
```

The rule `Verb.ReflVP : VPSlash -> VP` inserts a reflexive pronoun into the field `vp.nn.p1` of an incomplete verb phrase `vp`. A more general rule is needed that can insert any reflexive noun phrase instead. This rule

```
ReflRNP : VPSlash -> RNP -> VP ;   -- support my family and myself
```

is so far implemented by

```
ReflRNP vps rnp =
  insertObj (\\a => appPrep vps.c2 (rnp.s ! a)) vps ;
```

Here, `insertObj obj vps` adds `obj` to `vps.nn.p4`, ignoring whether the `obj` is derived from a pronoun `rnp` or not. Now that we have added fields `isPron`, `rc` and `ext` to `RNP`, we can improve `ReflRNP` to make it generalize `ReflVP`. At least closer to a generalization of `ReflVP` is

```
ReflRNP vps rnp = insertObjReflNP vps rnp ;
```

where the operation `insertObjReflNP vps rnp` distinguishes between (reflexive) pronouns and non-pronons, and considers all non-pronouns as light nominal objects inserted into `vps.nn.p2`.

```
insertObjReflNP : RNP -> ResGer.VPSlash -> ResGer.VP =
  \rnp,vp -> insertObjRNP rnp vp.c2 vp ;

insertObjRNP : RNP -> Preposition -> ResGer.VPSlash -> ResGer.VP =
  \rnp,prep,vp ->
  let obj : Agr => Str =
        \\a => prep.s ! CPl ++ rnp.s ! a ! prep.c ++ rnp.ext ++ rnp.rc
  in vp ** {
    nn = \\a =>
      let vpnn = vp.nn ! a in  -- acc-pron < pron < non-pron nominal < prep.
        case <prep.t, rnp.isPron, prep.c> of {
          <isCase,True,Acc> => <obj ! a ++ vpnn.p1, vpnn.p2, vpnn.p3, vpnn.p4> ;
          <isCase,True,_>   => <vpnn.p1 ++ obj ! a, vpnn.p2, vpnn.p3, vpnn.p4> ;
          <isCase,False,_>  => <vpnn.p1, vpnn.p2 ++ obj ! a, vpnn.p3, vpnn.p4> ;
          <_,_,_>           => <vpnn.p1, vpnn.p2, vpnn.p3 ++ obj ! a, vpnn.p4> }
  } ;
```

To prove that this `ReflRNP` generalizes `ReflVP : VPSlash -> VP`, do we need more than showing that `ReflRNP vps RelfPron` amounts to `ReflVP vps` for any `vps:VPSlash`?

Of the additional constructions using `RNP` declared in `abstract/Extend`, the first one,

```
AdvRNP : NP -> Prep -> RNP -> RNP ;   -- a dispute with his wife
```

is a variation of `AdvNP : NP -> Adv -> NP` in which the adverb is built by an implicit reflexive version of `PrepNP : Prep -> NP -> Adv`. The construction is implemented (in `ExtraGer`) by

```
AdvRNP np prep rnp = {s = \\a,c => np.s ! False ! c
                          ++ appPrep prep (rnp.s ! a) ++ rnp.ext ++ rnp.rc ;
                       ext = np.ext ; rc = np.rc ; isPron = False} ;
```

It is not quite clear whether a sentential, infinitival or interrogative complement `np.ext` of the argument noun phrase `np` should be separated by a sentential, infinitival or interrogative complement and relative clause in the modifying adverbial, as e.g. in *(wir haben) den Beweis für seine Behauptung, dass ihm alle glauben, (nicht abgewartet), den er angekündigt hatte.*

As mentioned earlier, in Remark 23, the example *(a) dispute with his wife* appears to be a complementation construction (say, `RComplN2 : N2 -> RNP -> RCN`). But modifications by reflexive adverbials are possible, e.g. *ein Baum in seinem Garten*, or by a reflexive possessive, e.g. Virginia Woolf's title *a room of one's own.*

The construction

```
ReflA2RNP : A2 -> RNP -> AP ;         -- indifferent to their surroundings
```

ought to build a *reflexive* adjective phrases, in which the complement depends on `Agr`.[114] As long as `AP.c : Str * Str` does not depend on `Agr`, we can only use a default agreement value:

```
ReflA2RNP adj rnp =                  -- would need AP.c : Agr => Str * Str
  let                                           -- without reflexive APs,
    compl = appPrep adj.c2 (rnp.s ! agrP3 Sg) ;  -- use a fixed agreement
  in {
    s = adj.s ! Posit ;
    isPre = True ;
    c = case adj.c2.isPrep of {False => <compl, []> ; True => <[], compl>} ;
    ext = rnp.ext ++ rnp.rc
  } ;
```

The construction of modifying an adjective phrase by a reflexive adverb `RAdv = Prep + RNP`

```
AdvRAP : AP -> Prep -> RNP -> AP ;    -- adamant in his refusal
```

can preliminarily be implemented by

```
AdvRAP ap prep rnp =
  let                                          -- ? Ger: adv ++ ap.s ! af
    adv = appPrep prep (rnp.s ! agrP3 Sg) ;    -- bug: fixed agreement
  in ap ** { s = \\af => ap.s ! af ++ adv } ; -- e.g. unknown in one's youth
```

Remark 102: It might be better if `AP` had a field `AP.adv : Agr => Str` where a reflexive adverb could be inserted. Todo 52: order by `adv ++ ap.s`: *(eine) in meinen Augen gute (Lösung)*. The relative order of complements and adverb in an adjective phrase needs to be considered.

Remark 103: When using a reflexive `ap : AP`, one has to concatenate `ap.s ! (AMod gn c)` or `ap.s ! APred` with `ap.c ! agr`. E.g. *ein auf seine(!) Taten stolzes Kind* or *sie ist stolz auf ihre(!) Taten*, and, with reflexive adverb, *sie war in ihrer Jugend stolz auf ihre Taten.*

The construction of modifiying a verb phrases `vp` by a reflexive adverb `prep + rnp`,

```
AdvRVP : VP -> Prep -> RNP -> VP ;    -- lectured about her travels
```

would force us to make `vp.a2:Str` depend on `Agr` and relate it with object-control: *er traf sich mit ihr in seinem Haus* vs. *er traf sich mit ihr in ihrem Haus.* An implementation would make verb phrases more complex, and probably lead to memory problems in grammar compilation, so we don't implement `AdvRVP`. As mentioned earlier, the example suggests a complementation rule, and reflexive nominal (and prepositional) objects can already be inserted into `vp.nn`.

Finally, an implementation of

```
PossPronRNP : Pron -> Num -> CN -> RNP -> NP ;
                                  -- his abandonment of his wife and children
```

uses the possessive of the given pronoun and the given numeral to build a determiner, which then combines the common noun with a possessively used reflexive noun phrase:

[114] In `Eng`, we have `AP.s : Agr => Str` and `RNP.s : Agr => Str`, so there is no need to introduce `RAP`.

```
PossPronRNP pron num cn rnp =
  DetCN (DetQuant (PossPron pron) num)
    (PossNP cn (lin NP {s = \\_,c => rnp.s ! pron.a ! c ;
                        a = pron.a ;
                        w = WLight ;
                        ext = rnp.ext ;
                        rc = rnp.rc})) ;
```

This gives (randomly generated) examples like

```
AllGerAbs> l PossPronRNP we_Pron NumPl (UseN student_N)
                              (ReflPoss NumPl (UseN camera_N))
unsere Studenten von unseren Kameras
```

But the provided English example suggests that a complementation rule is intended, taking a reflexive noun phrase as complement and yielding a reflexive common noun, e.g. *abandonment of one's family* (called `RComplN2 : N2 -> RNP -> RCN` above). We would then probably have

```
(PossPronRNP pron num (UseN2 n2) rnp).s =
  (ReflPoss num (RComplN2 n2 rnp)).s ! pron.a
```

to build *one's abandonment of one's family.*

Reflexive Predicates

Q70: How far do we want to push reflexive noun phrases? Besides the reflexive pronouns, the reflexive noun phrases are of the form `predet ++ (reflposs ++ cn)`, where a sentential object or relative clause of the common noun `cn` does not contain reflexives. For example, since `rnp.ext:Str`, the infinitival object in

```
TestLang> p -cat=Cl "ich fürchte meine Gründe , meinen Hund zu fürchten ,"
PredVP (UsePron i_Pron) (ReflRNP (SlashV2a fear_V2)
  (ReflPoss NumPl (SentCN (UseN reason_N) (EmbedVP
     (ComplSlash (SlashV2a fear_V2)
         (DetCN (DetQuant (PossPron i_Pron) NumSg) (UseN dog_N)))))))
```

contains `PossPron i_Pron`, not `ReflPoss`. But of course relative clauses and infinitival complements can contain reflexives in German, e.g. *man soll seine Anstrengungen, seine (eigenen) Fehler zu korrigieren, nicht übertreiben.*

In preliminary modules `german/Refl.gf`, `german/ReflGer.gf` and `german/ReflEng.gf` I have implemented some extensions to use reflexive noun phrases. In general, all constructions `f:A -> NP -> B` call for a modification `Rf:A -> RNP -> RB`, where fields `d:D => Str` of `B` have to be changed in `RB` to `d:Agr => D => Str`, if `f(a,np)` embeds strings `np.s ! c : Str` obtained from a parameter `c:Case` to field `B.d`. For example, `PrepNP : Prep -> NP -> Adv` needs a modification to `PrepRNP : Prep -> RNP -> RAdv`, to turn `Adv.s : Str` into `RAdv.s : Agr => Str`, so that we get reflexive adverbs like *in my (own) house*, more precisely, *in one's house.*

How many "reflexive" versions of categories do we need to add to `GrammarGer`, and how many can we add within the limits of grammar compilation? `RNP`, `RAdv`, `AP` with reflexive object `AP.c`, `CN` with reflexive adverb `CN.adv` or `RCN` with reflexive nominal object incorporated in `RCN.s` etc.? So far, `VP` and `VPSlash` have reflexive nominal objects `VP.nn` and reflexive infinitival

complemente `VP.inf`, but sentential complements or relative clauses are not dependent on `Agr`. (Infinitival complements should be reflexive because of control verbs.)

Remark 104. *How can an application grammar define a unary predicate "to brush one's teeth"? Should we be able to distinguish between the possessive "his" and the reflexive possessive "his own", as in "he explained it to his (own?) child". Should we write "sein eigener" to distinguish the reflexive possessive from the personal possessive "sein", or rather "sein" and "dessen"?*

For `Eng`, `ReflRNP` has to insert a reflexive noun phrase like "one's teeth" or "one's own car" of type `Agr => Str` into an incomplete verb phrase `vps` and must update `vps.s2 : Agr => Str` to `s2 = \\a => vps.s2!a ++ rnp!a`. We need a new value `AgP0:Agr` with `persPron!AgP0`="one", `possPron ! AgP0` ="one's", `reflPron ! AgP0` = "oneself" and `(ReflPoss num cn).s ! AgP0 = one's ++ num.s ++ cn.s ! num.n`, and should define `linref RNP` using `AgP0` [instead of adding a `one_Pron:Pron` with `one_Pron.a = AgP0:Agr` and `one_Pron.s ! Nom` = "one"]. Since `Ger` uses `Ag Masc Sg P3` as agreement of the implicit subject of the infinitive in `EmbedVP` (and of the subject "man" in `GenericCl`), we get the intended infinitives in `Ger`, but not in `Eng`:

```
AllGerAbs> l EmbedVP (ReflRNP (SlashV2a love_V2) (ReflPoss NumSg (UseN dog_N)))
seinen Hund zu lieben
AllEngAbs> l EmbedVP (ReflRNP (SlashV2a love_V2) (ReflPoss NumSg (UseN dog_N)))
to love its dog
```

For `Eng`, all tables `Agr => Str` have to be extended. `GenericCl` has to use "one" as subject and must update reflexives in `vp` by instantiating `vp.s2` to `AgP0`, as `EmbedVP` has to, in order to make the reflexives depend on the missing subject's agreement. (See also `RelPronVP`, p. 44.)

Remark 105: There is a kind of "semireflexive predicates", i.e. predicates with reflexive reference to an object: *jmdn ermahnen, sich anzustrengen* in *sie ermahnten uns, uns anzustrengen.*

Todo 53: Explain more of the constructions in `Extend`. At least

- adjust `mkClause`, `ComplVPIVV`, `MkVPS`, `DisToCl` in `Extra` and `Extend`

5.11.2. `ExtraGer` (todo)

Constructions specific to German may be lexical items, e.g. the modal verb

```
moegen_VV : VV ;          -- ich mag/möchte singen
```

or forms of passive from ternary verbs, like

```
Pass3V3 : V3 -> VPSlash ; -- wir bekommen den Beweis erklärt
```

or constructions with correlate *es* for sentential or infinitival complements (or for adverbs).

```
EsVV : VV -> VP -> VP ;         -- ich genieße es zu schlafen
EsV2A : V2A -> AP -> S -> VP ; -- ich finde es schön, dass ...
```

Correlates

Many verbs with a sentential complement can as well have an infinitival complement, and conversely. For example, *ich genieße es, zu schlafen*, but also *ich genieße es, dass die Sonne scheint.*

Likewise, *ich finde es schön, daß die Sonne scheint* and *ich finde es schön, im See zu schwimmen.* In a sense, *es* and *das* are correlates for "direct" sentential complements (analogs to nominal objects in accusative), in contrast to the following "prepositional" sentential complements (analogs to prepositional objects).

As we can use prepositions to combine complements with a verb, we can use the `CAdvPron` field of contracting prepositions[115] to insert a correlate for an infinitival or sentential complement.

Correlates for infinitival and sentential complements of nouns (sketch)

For noun complements, a possible rule could be

```
SentN2 : N2 -> SC -> CN ;
```

with implementation (so far, in `gf-rgl/tests/german/TestLexiconGer.gf`)

```
SentN2 n2 sc =
  let cor : Str = case n2.c2.t of {
    R.isContracting => n2.c2.s ! R.CAdvPron ; _ => []}
  in {
    s = \\_,n,c => n2.s ! n ! c ++ cor ++ P.bindComma ;
    ext = sc.s ;
    rc = \\_ => [] ;
    adv = [] ;
    g = n2.g
  } ;
```

With `hope_NV:N2` and

```
hope_NV = mkN2 (mkN "Hoffnung" feminine) aufs_Prep ;
```

this would give, for example:

```
TLang> gr -tr -cat=NP (DetCN ? (SentN2 hope_NV (EmbedVP ?))) | l
DetCN few_Det (SentN2 hope_NV (EmbedVP ready_VP))

few hopes to be ready
wenige Hoffnungen darauf , bereit zu sein
```

This seems better than `SentCN`. First, `SentN2 : N2 -> SC -> CN` applies to binary nouns only. Second, the correlate *darauf* depends on the noun's preposition `hope_NV.c2 = aufs_Prep`. Third, we can have a special rule `CorN2 : N2 -> CN` which uses just the correlate as complement, e.g. *die Hoffnung darauf.* Forth, instead of a sentential complement we can always also add a nominal complement, by `ComplN2 : N2 -> NP -> CN`[116]. But since `SentN2` applies to *all* binary nouns, we would still get bad examples, e.g. *Bruder davon, zu schlafen.*[117] A drawback is that since both the complement and the correlate can be dropped by `UseN2 : N2 -> CN` and `PrepNP auf np` is an adverb, we get three trees for *die Hoffnung auf np*,

[115] More generally, non-contracting prepositions should also have such a field to be used for correlates.
[116] This is quite common: *ein Grund (für np | dafür, dass s)*, *der Glaube (an np | daran, dass s)* etc.
[117] But *ich bin kein Freund davon, zu träumen* is good.

```
DetCN defArt (ComplN2 hope np)
DetCN defArt (AdvCN (UseN2 hope) (PrepNP auf np))
AdvNP (DetCN defArt (UseN2 hope)) (PrepNP auf np)
```

Q71: How can the readings as adverb be suppressed by an existing reading as complement?

Remark 106: English seems to make less use of sentential correlates: *That she spoiled the game ..., there can be little doubt* ↦ *There can be little doubt that she spoiled the game ...* Ger: *Daß S, daran kann es kaum Zweifel geben* ↦ *Es kann kaum Zweifel (daran) geben, daß S.*

Alternatively, instead of combining a sentential complement `sc:SC` with a binary noun, we could be more specific and use categories `NS` of nouns with sentential object, `NV` of nouns with infinitival object, and `NQ` of nouns with interrogative objects. We could then use

```
lincat NS = Noun ** {c2 : Preposition} ;
```

and use the prepositions in `ns.c2` to add a correlate in the complementation rule

```
ComplNS : NS -> S -> CN ;
```

implemented by

```
ComplNS ns s =
  let p = ns.c2 ;
      cor = case p.t of {isContracting => p.s ! CAdvPron ; _ => []}
  in {
    s = \\a,n,c => ns.s ! n ! c ++ cor ++ comma ;
    rc = \\n => [] ;
    ext = conjThat ++ s.s ! Sub ;
    adv = [] ;
    g = ns.g
  } ;
```

E.g., to add the correlate *daran* in *(der) Glaube daran, dass die Erde eine Kugel ist.* Clearly, such correlates are strongly related to the prepositions used in corresponding verbs, i.e. *glauben an etwas.* Without a separate construction to add a nominal object, *(der) Glaube an das Christkind* would not be accepted. Analogously to `UseN2 : N2 -> CN`, we would need a rule

```
UseNS : NS -> CN ;
```

to use the noun without its sentential complement (and without its correlate, of course).

Sentential complements in conjunctive

An object sentence can also take the form of a sentence in subjunctive mood (Konjunktiv), e.g. *(der) Glaube , die Erde sei eine Kugel.* Such a construction can however not be implemented with argument type `S` of indicative sentences. We need a new syntactic category `SConj` of sentences in conjunctive, with linearization category

```
SConj = {s : Order => Str} ;
```

The construction of such sentences by

```
UseConjCl : Temp -> Pol -> Cl -> SConj ;
```

can be implemented by a variation of `UseCl`, ignoring the value `t.m` of its argument `t:Temp`:

```
UseConjCl t p cl = {
  s = \\o => t.s ++ p.s ++ cl.s ! MConjunct ! t.t ! t.a ! p.p ! o
  } ;
```

We can then implement the complementation by sentences `s:SConj`,

```
ComplConjNS : NS -> SConj -> CN ;
```

ignoring the correlate in `ns.c2.s ! CAdvPron` and using conjunctive mood in the object clause:

```
ComplConjNS ns s = {
    s = \\a,n,c => ns.s ! n ! c ++ comma ;
    rc = \\n => [] ;
    ext = s.s ! Main ;
    adv = [] ;
    g = ns.g
  } ;
```

For example, with this we can parse both kinds of sentence complements:

```
TLang> p -cat=NP "der Glaube , es gebe keine Hoffnung"
DetCN (DetQuant DefArt NumSg)
  (ComplConjNS belief_NS (UseConjCl (TTAnt TPres ASimul) PPos
      (ExistNP (DetCN (DetQuant no_Quant NumSg) (UseN2 hope_NV)))))

TLang> p -cat=NP "der Glaube daran , dass es keine Hoffnung gibt"
DetCN (DetQuant DefArt NumSg)
  (ComplNS belief_NS (UseCl (TTAnt TPres ASimul) PPos
      (ExistNP (DetCN (DetQuant no_Quant NumSg) (UseN2 hope_NV)))))
```

Similarly, one could add complement sentences in conjunctive mood for verbs and adjectives. ◁

Remark 107: However, such complementation rules by sentential complements need more lexical work to specify the categories of nouns (and adjectives), and some computational overhead (at least six rules `UseNS`, `UseNV`, `UseNQ`, `ComplNS`, `ComplNV`, `ComplNQ`) instead of a single overgenerating rule `SentN2`.

Correlates for infinitival and sentential complements of verbs (sketch)

An implementation of a rule `SentVS : VS -> SC -> VP` similar to `SentN2` above can insert the correlate to `vp.nn` as prepositional object and the `sc` to `vp.ext`, so that correlate and sentential object can be separated, e.g. *wir hatten nicht daran geglaubt, dass es möglich ist.* However, the argument category `SC` in `SentVS` is too crude: not every verb that takes an infinitival object also takes a sentential object, e.g. *manche versuchen, den Ärmelkanal zu durchschwimmen* ↛ *manche versuchen, dass sie den Ärmelkanal durchschwimmen.*

`Grammar` distinguishes between verbs `VS` taking a sentential object, verbs `VV` taking an infinitival object, and verbs `VQ` taking an interrogative object. In particular, `Grammar` has rules `ComplVS :`

`VS -> S -> VP` and `SlashV2S : V2S -> S -> VPSlash` to add complement sentences to verbs expecting an object sentence. So we only need to add complementation rules that also (or only) add a correlate for the object sentence (infinitive or interrogative, respectively).

To do so, take `lincat VS = V2` instead of `VS = Verb`[118] in `CatGer`, change `ParadigmsGer.mkVS` accordingly and add a field `cor:Str` to the implementation type `VP`. Then add a construction

```
ComplCorVS : VS -> S -> VP ; -- glaube daran|es, dass S
```

that inserts a correlate for the sentential object to the correlate field of the resulting verb phrase. From prepositions, we derive correlates by

```
oper
  mkCor : Preposition -> Str = \p ->
    case p.t of {isContracting => p.s ! CAdvPron ; _ => "es" | "das"} ;
```

If `vs.c2` contains a contracting preposition, the correlate can be *daran*, *darauf* etc.

```
ComplCorVS vs s =
  insertExtrapos (comma ++ conjThat ++ s.s ! Sub)
    (predV vs ** {c2 = vs.c2 ; cor = mkCor vs.c2}) ;
```

(This violates the convention that resource grammar rules should avoid alternatives like `"es"|"das"`.) A separate construction

```
CorVS : VS -> VP ;           -- glaube daran
```

is needed to *only* add the correlate, e.g. to get *wir hatten daran nicht geglaubt*.

```
CorVS vs =
  predV vs ** {c2 = vs.c2 ; cor = mkCor vs.c2} ;
```

Changing `StructuralGer` to

```
hope_VS = mkVS (regV "hoffen") ** {c2 = aufAcc_Prep} ;
```

we get for example:

```
TLang> p -cat=S "wir hatten darauf nicht gehofft"
UseCl (TTAnt TPast AAnter) PNeg (PredVP (UsePron we_Pron) (CorVS hope_VS))
```

The correlate would better follow the negation adverb. Similarly, we can have correlates for infinitival complements:

```
CorVV vv = predV vv ** {c2 = vv.c2 ; cor = mkCor vv.c2} ;

ComplCorVV vv vp =
  let inf = mkInf False Simul Pos vp ;   -- False = force extraction
  in insertExtrapos vp.ext (insertInf inf (CorVV vv)) ;
```

[118] Similarly, one could change the type of ternary verbs `V2S` by adding a field `c3:Preposition`.

Again, the corrolate ought to follow the adverb *jeden Montag* in

```
TLang> p -cat=Cl "ich denke daran jeden Montag , dein Buch zu lesen"
```

Todo 54: Maybe add rules for correlates with ternary verbs

```
SlashCorV2S : V2S -> S -> VPSlash ;   -- glaube es (dir) , dass ...
SlashCorV2V : V2V -> VP -> VPSlash ;  -- rate es (dir) , mehr zu arbeiten
```

And for correlates for interrogative complements, e.g. *worauf|auf was hoffen wir*?

There is no syntactic category for correlates[119], and correlates cannot generally be used to replace a prepositional object, e.g. *sie starren an die Wand* ↛ *sie starren daran*. (Correlates are pro-forms of sentential complements; if the complement sentence is combined with the verb without a preposition, e.g. *ich sehe (es) ein, dass 5 ungerade ist*, there seems to be no difference between a correlate *es* and a pronoun *es* in *ich sehe es ein*?)

Todo 55: Correct the two constructions `EsVV : VV -> VP -> VP` and `EsV2A : V2A -> AP -> S -> VP` that add a correlate *es* for an infinitival or sentential object. The correlate should not be treated as an object, which is sometimes not put before, but after the negation adverb *nicht*. And rename the rules to `ComplCorVV` etc.? At least, `EsV2A` should rather be called `EsVSA:VSA -> S -> A -> VP`, as there is no nominal object.

Correlates for adverbs and adverbial clauses (todo)

How can we deal with correlates for adverbs, e.g. *ich gehe deshalb schlafen, weil ich müde bin* for *weil ich müde bin, (deshalb) gehe ich schlafen* and *deshalb gehe ich schlafen.*[120]

What can we do? In `TCatGer`, we can add a category `PronAdv` of pronominal adverbs, relating interrogative and definite adverbs with subjunctions for adverbial clauses. A linearization type with several fields of type `Str` would not do, because at most two of these (the definite adverb and the subjunction) are used in a single sentence (hence the third component raises a metavariable in parsing). But a table with result type `Str` might do.

```
param AdvType = AdvI | AdvD | AdvS ;

lincat PronAdv = {s : AdvType => Str} ;
lin deshalb_PronAdv =
  {s = table AdvType {AdvI => "weshalb" ; AdvD => "deshalb" ; AdvS => "weil"} ;
```

We can then use several of the three strings, but perhaps not as adverbs or subjunctors. To make this possible, we can let `PronAdv` have components of other syntactic categories, i.e.

```
lincat PronAdv =
  {s : AdvType => Str ; i : IAdv ; d : Adv ; sj : Subj} ;
```

with `padv.s ! AdvI = padv.i.s` etc., and define from `S = StructuralGer`

[119] The correlate *daran* in *wir glauben daran* is not the pronominal adverb `daran_Adv`, as it represents an object sentence, not an adverbial sentence.

[120] Q72: Do we have the same *double* realisation of a syntactic role in *ich sah ihn kommen, den Hund*?

```
da_wo : PronAdv = let adv : CatGer.IAdv * CatGer.Adv * CatGer.Subj =
                 <S.where_IAdv, S.there_Adv, {s = [] ; lock_Subj=<>}>
  in {s = table{AdvI => adv.p1.s ; AdvD => adv.p2.s ; AdvSj => adv.p3.s} ;
      i = adv.p1 ; d = adv.p2 ; sj = adv.p3
  } ;
```

We would like to get *dort, wo der Pfeffer wächst* as a split adverb, using *dort* as a correlate, but both components being adverbs in themselves, or the second one a "*freier Fragesatz*" [3] as in *ich weiß, wo der Pfeffer wächst.* This can be done with a rule

```
PronAdvS : PronAdv -> S -> Adv ;
```

implemented by

```
PronAdvS p s = {s = p.i.s ++ s.s!Sub ; cor = p.d.s ; cp = []} ;
```

A comma is omitted, so that both *sie sollen dort wohnen, wo der Pfeffer wächst* and *wo der Pfeffer wächst, dort sollen sie wohnen* are possible. But: since the resulting `Adv` has two string-fields, whence we always have to use both fields.

Q73: Can we instead just add a correlate in a rule `CorExtAdvS : Adv -> S -> S`, and give up the constraint implemented by `PronAdv`? (How to compute a correlate from `adv:Adv`, if `adv.isPron = True`?)

In analogy to `QuestIAdv : IAdv -> Cl -> QCl`, we might have a relative clause construction

```
RelAdv : PronAdv -> Cl -> RCl ; -- RAdv = ReflAdv ?

RelAdv adv qs = {s = qs.s ! QIndir ; cor = adv.s ; cp = []} ;

-- RelAdv is bad: accepts "die Frage , wo sie war" by
-- ExtAdvNP with metavariable for cor = []
```

Note: *wo* can be used as temporal adverb: *heutzutage, wo jeder ein Auto hat*

Moving extractions (todo)

Some implementation categories have a field `ext:Str` for sentential, infinitival or interrogative complements. These fields, like the field for a relative sentence in noun phrases or for a comparison noun phrase in adjective or adverb phrases, contain parts of a phrase that may be separated from the other part. Typically, the past participle of a verb may be inserted between a noun and a relative clause, or between an adjective and its comparison noun phrase.

So far, such constructions are not implemented yet.

6. Improving Translation by Structural Transfer

It has sometimes been suggested that structural transfer should be done outside of `GF`, by exporting the data type of GF-trees to haskell or some other programmig language and writing tree transformers in the host programming language, suited to particular applications.

In my opinion, this is misguided in that there are or may be structural transfers useful for many applications and common to many concrete languages. To make better use of structural transformations inside `GF`, two possibilities are needed: (i) to declare more syntactic constructions `fun f:C` by `data f:C`, and (ii) to apply structural transformations to (subtrees of) trees resulting from parsing an input string.

6.1. The grammar `DGrammar`

To allow for more transfer functions to be defined *without* changing the abstract grammar `Grammar`, we introduce a variant `DGrammar` consisting of variants `DNoun.gf` of `Noun.gf` etc. where (almost) all syntactic constructions are `data` constructors and then use the original concrete modules to define concrete modules of the variants.

Using functors `D<module>F` for each module of `Grammar`, the second step can be done uniformly for several languages. For example, we define a functor `DNounF` by

```
incomplete concrete DNounF of DNoun = open Noun in {
  lin DetCN = Noun.DetCN ;  ... }
```

and the language specific concrete module `DNounGer` for German by

```
concrete DNounGer of DNoun = CatGer ** DNounF with (Noun = NounGer) ;
```

Then `DGrammar` can be extended by the example lexicon `Lexicon` (or a version `DLexicon` where all entries are declared by `data`) and a module of transfer functions to `DLang`. (See Section 6.6.) This is done in (my) `~/GF/gf-rgl/tests/german/DLang.gf`. To explore parallelism between definite, interrogative and relative phrases, additionally some categories are turned into dependent categories, in `~/GF/gf-rgl/tests/german/DepLang.gf` (not using functors).

6.2. Adding flags `normalize` and `transfer` to `put_tree`

To make the use of tree transfer functions available in the gf-shell again, we have to modify two files of the GF-compiler,

```
gf-core/src/compiler/GF/Command/Commands.hs
gf-core/src/compiler/GF/Command/TreeOperations.hs
```

We distingish two possibilities: first, we just want to apply a transfer function `f:C -> D` to a parse result `tree : C` and obtain a transformed tree of possibly different type `D`. This ought to be done by

```
> parse -cat=C "..." | put_tree -transfer=f
```

Second, we want to apply a transfer function `f:C -> C` to replace all maximal subtrees `s:C` of a given tree by their transforms `f(s):C` of the same type. This ought to be done by

```
  > parse -cat=C "..." | put_tree -normalize=f
```

The transformed tree can be linearized to a target language `Tgt` by piping it to `linearize -lang=Tgt`. Since `def`-rules can be recursive and can call other transfer functions, `f(s)` may apply a transfer `g:C' -> C'` to some of its subtrees of type `C'`.

To install these possibilities, first add the following definitions to the file `TreeOperations.hs`:

```
-- Apply transfer function f:C -> D to tree e, if e:C, else return e.  HL 12/2024

transfer :: PGF -> CId -> Expr -> Expr
transfer pgf f e = case inferExpr pgf (appf e) of
  Left _err -> e
  Right _ty -> compute pgf (appf e)
 where
  appf = EApp (EFun f)

-- Apply transfer function f:C -> C to the tree e's maximal subtrees s:C,
-- and replace these s by the normalized values of f(s):C.

normalize :: PGF -> CId -> Expr -> Expr
normalize pgf f e = case inferExpr pgf (appf e) of
     Left _err -> case e of
                    EApp g a -> EApp (normalize pgf f g) (normalize pgf f a)
                    _ -> e
     Right _ty -> case (compute pgf (appf e)) of
                    v | v /= appf e ->
                    case v of  -- v may contain transfer functions
                      EApp (EFun g) a | v /= e -> normalize pgf g a
                      _ -> v
                    _ -> e     -- default case of f, or f has no computation rule
 where
  appf = EApp (EFun f)
```

Moreover, edit the function `allTreeOps` by entries for `transfer` and `normalize` like

```
   ("transfer",("wrap this transfer function around tree and compute",
      Right $ \f -> map (transfer pgf f))),
```

so that `help pt` will show

```
flags:
 -transfer wrap this transfer function around tree and compute
 -normalize apply this type-preserving transfer function to all suitable subtrees
```

Second, in the file `Commands` under `pgfCommands`, edit the list of examples for `pt` by two entries for the new flags (similar to that for the flag `compute`), so that `help pt` will show

```
examples:
  p "the 4 dogs" | pt -normalize=digits2numeral | l    -- "the four dogs"
  p -cat=S "they played the opossum" | pt -transfer=idiomS | l -lang=Ger
                                                 -- "sie stellten sich tot"
```

Todo 56: Instead of `idiomS`, use a type-changing example like `nominalize:VP -> NP`.

Finally, recompile and install the modified GF-compiler by calling

```
cabal install
```

from a shell in the directory `gc-core`.

6.3. Improving translations

The translation method offered by GF is *linearize* ∘ *parse*, i.e. the translation of a text `src:Str` from a source language `Src` to a text (of type `Str`) in a target language `Tgt` is done by

```
parse -lang=Src src | linearize -lang=Tgt
```

Ideally, each abstract tree obtained from parsing represents a meaning of the source text. However, even if the same trees are useful to structure expressions of source and target language, a structure-preserving translation often results in possible, but rarely used wordings. Sometimes the source and target languages even use different structures to express the same meaning. For example, Ancient Greek has 11 participles, while English or German only have two. To translate the participles of Ancient Greek, we *must* use differently structured expressions (like relative or adverbial clauses).

To improve the structure-preserving translation offered by *linearize* ∘ *parse*, we want to apply transfer functions in between, i.e. use the more general translation by *linearize* ∘ *transfer* ∘ *parse*, expressed in the gf-shell by

```
parse -lang=Src src | put_tree -transfer=f | linearize -lang=Tgt
```

It may seem that the transfer function `f` used here ought to depend on the source and target languages `Src` and `Tgt`. But in general, a tree is linearized to several target languages, and transfer functions in `GF` operate on the abstract syntax, so there are no transfer functions specific to a given source and a given target language (unless `pt -compute` is modified).

Let us call a type-preserving transfer function $f : C \to C$ a ***normalizing transfer function***. Such a transfer function can be used to substitute subtrees of a given type C in a tree. Several applications of normalizing transfers come to mind:

Collapsing different structures to reduce ambiguities

When expressions of a category C can be modified by different modification constructions $f_i : C \to C$, the relative order in which the modifiers are attached to the head element is often irrelevant (in particular, if the modifiers are stored in different fields of the implementation record). For example, there are six ways to modify a simple common noun by `AdjCN`, `AdvCN` and `RelCN`, e.g. *small dog in Paris that jumps*, although the adjective is pre-nominal and there are at most two relative orderings of the postnominal adverbial and postnominal relative clause (in `Eng`), and the different constructions seem not to correspond to semantic differences. To give the adverbial modifiers narrowest scope and the relative clauses the widest scopes, one can define a normalizing transfer `nfCN` by

```
fun nfCN : CN -> CN ;
def nfCN (RelCN cn rs) = insertRS (nfCN cn) rs ;
```

```
    nfCN (AdvCN cn adv) = insertAdv (nfCN cn) adv ;
    nfCN (AdjCN ap cn) = insertAdj ap (nfCN cn) ;
    nfCN cn = cn ;
```

The axiliary functions presuppose that their `CN`-argument already is in normal form, i.e. that modifying adverbs are innermost, modifying relative clauses outermost:

```
fun insertAdv : CN -> Adv -> CN ; -- with cn in normal form !
def insertAdv (RelCN cn rs) adv = RelCN (insertAdv cn adv) rs ;
    insertAdv (AdjCN ap cn) adv = AdjCN ap (insertAdv cn adv) ;
    insertAdv cn adv = AdvCN cn adv ;
fun insertAP : AP -> CN -> CN ; -- with cn in normal form !
def insertAP ap (RelCN cn rs) = RelCN (insertAP ap cn) rs ;
    insertAP ap cn = AdjCN ap cn ;
fun insertRS : CN -> RS -> CN ; -- with cn in normal form !
def insertRS (AdjCN ap cn) rs = RelCN (AdjCN ap cn) rs ;
    insertRS (AdvCN cn adv) rs = RelCN (AdvCN cn adv) rs ;
    insertRS cn rs = RelCN cn rs ;
```

The last clause might be replaced by first conjoining several relative clauses:

```
    -- conjoin multiple relativizations:
    insertRS (RelCN cn (ConjRS and_Conj rse)) rs
      = RelCN cn (ConjRS and_Conj (rconsRS rse rs)) ;
    insertRS (RelCN cn rs1) rs
      = RelCN cn (ConjRS and_Conj (BaseRS rs1 rs)) ;
    insertRS cn rs = RelCN cn rs ;
fun rconsRS : ListRS -> RS -> ListRS ;
def rconsRS (BaseRS rs1 rs2) rs = ConsRS rs1 (BaseRS rs2 rs) ;
    rconsRS (ConsRS rs1 rse) rs = ConsRS rs1 (rconsRS rse rs) ;
```

Using `pt -normalize=nfCN -nub`, the six different constructions of the example are mapped to the same "normal form" before translation:

```
DLang> p -lang=Eng -cat=NP "a small dog in Paris that jumps"
       | pt -normalize=nfCN -nub -tr | l -lang=Ger
DetCN (DetQuant IndefArt NumSg)
      (RelCN (AdjCN (PositA small_A)
             (AdvCN (UseN dog_N) (PrepNP in_Prep (UsePN paris_PN))))
       (UseRCl (TTAnt TPres ASimul) PPos (RelVP IdRP (UseV jump_V))))

ein kleiner Hund in Paris , der springt
```

Similarly, one might identify special uses of `RelNP` and `AdvNP` with uses of `RelCN` and `AdvCN`:

```
fun nfNP : NP -> NP ;
def nfNP (RelNP (DetCN det cn) rs) = DetCN det (RelCN cn rs) ;
    nfNP (AdvNP (DetCN det cn) adv) = DetCN det (AdvCN cn adv) ;
```

Perhaps the last clause should better be

```
nfNP (AdvNP (DetCN det cn) adv) = DetCN det (nfCN (AdvCN cn adv)) ;
```

And perhaps `nfNP` should also give `PossNP` narrower scope than `AdvNP`.

Clearly, we can't reduce arbitrary uses of `RelNP` to uses of `RelCN`, since a relativization of noun phrases is possible, e.g. *Johann und sein Bruder, die ich gut kenne*. This combination seems mostly function as apposition, not as restriction. (Q74: Should we distinguish between restrictions, by `CN+RC`, `Det+RC`, and appositions, by `PN+RC`, `Pron+RC`, `NP+RC`? Both noun phrases and relative clauses can be appositions, so is the difference `Extend.ApposNP : NP -> NP -> NP` and `RelNP : NP -> RS -> NP` the best we can do?)

Improve wordings

We might also want to translate or normalize negations of an indefinite noun phrase, e.g.

```
nfNP (PredetNP not_Predet something_NP) = nothing_NP ;
nfNP (PredetNP not_Predet nothing_NP) = something_NP ;
nfNP (PredetNP not_Predet somebody_NP) = nobody_NP ;
nfNP (PredetNP not_Predet nobody_NP) = somebody_NP ;
nfNP (PredetNP not_Predet (DetCN (DetQuant IndefArt num) cn)) =
     DetCN (DetQuant no_Quant num) cn ;
nfNP (PredetNP not_Predet (DetCN someSg_Det cn)) =
     DetCN (DetQuant no_Quant NumSg) cn ;
```

These are sometimes unused under intonation, e.g. *sie machten nicht eínen Fehler*, which is preserved, e.g. *sie machten keínen Fehler*.

It is less clear if normalizations of determiners like (c.f. `StructuralGer`)

```
fun nfDet : Det -> Det ;
def nfDet someSg_Det = DetQuant IndefArt NumSg ;
    nfDet det = det ;
```

are language independent or rather restricted to specific languages like German. `Eng`:*some* vs. *a/one*.

Stylistic changes

Sometimes a structural change does not preserve the category of the tree, in which case we cannot substitute an embedded expression by its transformed version.

Here is an example for such a ***non-normalizing transfer function***. Modal verbs, i.e. verbs of category `VV`, do not have imperative forms in English or German, but they can be used in imperatives in GF:

```
Lang> l AdvImp now_Adv (ImpVP (ComplVV must_VV (UseV go_V)))

now have to go
jetzt müsse gehen
```

The correct way to express the intended command is by a suitable indicative clause. To transfer such incorrect imperatives to clauses, we can define a transfer function:

```
fun trImp : Imp -> Cl ;
def trImp (ImpVP (ComplVV modalV VP))
      = PredVP (UsePron youSg_Pron) (ComplVV modalV VP) ;
    trImp (AdvImp adv (ImpVP (ComplVV modalV VP)))
      = PredVP (UsePron youSg_Pron) (AdvVP (ComplVV modalV VP) adv) ;
```

and then translate the incorrect modal imperatives to clauses:

```
DLang> p -cat=Imp -lang=Eng "now have to go" | pt -transfer=trImp -tr | l
PredVP (UsePron youSg_Pron) (AdvVP (ComplVV must_VV (UseV go_V)) now_Adv)

you must go now
du musst jetzt gehen
```

To allow for the plural *ihr*, the polite *Sie*, use `youPl_pron` and `youPol_Pron` as pronoun. Q75: How can we admit negated imperatives, i.e. variants for `UttImpSg : Pol -> Imp -> Utt` etc.? We may write binary transfers `f`, but can we pipe an `imp:Imp` to `pt -transfer="f PNeg"` ?

Other candidates for non-normalizing transfer functions:

Reduction to a core language (todo)

Mappings from language-specific constructions to a core common to several languages, $toCore : E \to C$, and conversely, mappings $toExtra : C \to E$ from the common core to a language-specific extension. We need an artificial abstract language combining the (disjoint) `Extra` parts.

What about tree transformations `toAllEng : Grammar -> AllEngAbs` to replace constructions in `Grammar` by language-specific ones in `AllEngAbs`, or converse transformations `fromAllEng: AllEngAbs -> Grammar`, `fromAllGer : AllGerAbs -> Grammar` to a common core `Grammar`?

For example, `AllEngAbs` might exclude `Verb.ReflVP`, and we could translate

```
fun nfVP : VP -> VP ;
def nfVP (ReflVP vps) = ReflRNP vps ReflPron ;
    nfVP vp = vp ;
```

No, this is still a normalizing transfer! The transfer module must contain the tree constructor `Grammar.ReflVP` *and* the syntactic constructor `AllEngAbs.ReflRNP:VPSlash -> RNP -> VP`. But why should one want to include the obsolete `ReflVP` and transfer it away in trees, rather than excluding it in the abstract language, i.e. use `AllEngAbs`?

Todo 57 Another non-normalizing transfers could be `nominalize : VP -> NP`, as in e.g. *to read a book* $\mapsto$ *the reading of a book*. It can perhaps roughly be implemented by

```
fun nominalize : VP -> NP ;
def nominalize ComplSlash (SlashV2a v2) np) =
    DetCN (DetQuant DefArt NumSg) (ComplCN (PresPartCN (UseV v2)) np)
```

with so far undefined constructions `ComplCN:CN -> NP -> CN` and `PresPartCN:VP -> CN`.

Similarly, one can transfer (some) relative clauses into participle constructions and conversely: *low hanging (fruits)* $\mapsto$ *(fruits) that hang low*, roughly

```
ap + cn  =>  cn + RelVP rp (UseComp (CompAP ap))
```

6.4. Translation of idiomatic expressions

Idiomatic expressions are grammatically well-formed expressions, but are used with a meaning that is not composed of the literal meanings of their words in the standard way. Often, they also can be used with the compositional meaning, like *at the end of the day*. To translate the idiomatic reading of an expression in the source language to one or more target languages, transfer its grammatical construction tree to a ***idiom constant*** and linearize the constant to the linearizations of trees of expressions in the target languages.[121] Consider the following constants

```
fun idiom_early_bird_Cl : Cl ;       -- the early bird catches the worm
    idiom_simulate_dead_VP : VP ;    -- play the opossum
```

In a grammar for English, the linearization is given by the tree of the English idiom:

```
lin idiom_early_bird_Cl =
    PredVP (DetCN (DetQuant DefArt NumSg)
              (AdjCN (PositA early_A) (UseN bird_N)))
      (ComplSlash (SlashV2a catch_V2)
        (DetCN (DetQuant DefArt NumSg) (UseN worm_N)))
```

In a grammar for German, the linearization can be the tree of a corresponding German idiom with similar idiomatic meaning, i.e. *Morgenstund' hat Gold im Mund*:

```
lin idiom_early_bird_Cl =
    PredVP (MassNP (UseN morgenstunde_N))
      (AdvVP (ComplSlash (SlashV2a have_V2) (MassNP (UseN gold_N)))
        (PrepNP in_Prep (DetCN (DetQuant DefArt NumSg) (UseN mouth_N))))
```

For `idiom_simulate_daed_VP`, the linearization in English is the tree of the expression,

```
lin idiom_simulate_dead_VP =
    (ComplSlash (SlashV2a play_V2)
       (DetCN (DetQuant DefArt NumSg) (UseN opossum_N))) ;
```

while the linearization in German uses a special reflexive Verb *sich totstellen*:

```
lin idiom_simulate_dead_VP =
    UseV (reflV (prefixV "tot" (regV "stellen")) accusative) ;
```

Notice that parsing the idomatic expression `src` gives both its standard tree and the idiom constant, hence, if all words in the tree have linearizations in the target language `Tgt`, a translation by *linearize* ∘ *parse* gives a literal translation as well as an idiomatic translation:

```
DLang> p -tr -lang=Eng -cat=Cl "the early bird catches the worm" | l -lang=Ger
PredVP (DetCN (DetQuant DefArt NumSg) (AdjCN (PositA early_A) (UseN bird_N)))
  (ComplSlash (SlashV2a catch_V2)
    (DetCN (DetQuant DefArt NumSg) (UseN worm_N)))
```

[121] If there is one target language only, the transfer function can map the idiom's tree to the target expression's tree, and skip the idiom constant.

```
idiom_early_bird_Cl

der frühe Vogel fängt den Wurm
Morgenstund hat Gold im Mund
```

To suppress the non-idiomatic reading of an idiom `src`, we can use a transfer function `f` to map its tree to the idiom constant and identify both, using

```
parse -lang=Src -cat=S src | pt -normalize=f -nub | l -lang=Tgt
```

Idiomatic expressions can be sentences, but more often they can vary in tense and combine with different subjects and objects, i.e. are of type `Cl`, `VP` or `VPSlash`, or they can be adverbs.

Let `idiomS` be a transfer function intended to handle idioms of type `S`. As default case, we let the transfer function go down its input tree to subtrees of the same or similar kinds, and transfer these with transfer functions of appropriate type:

```
fun idiomS : S -> S ;
def idiomS (UseCl tmp pol cl) = UseCl tmp pol (idiomCl cl) ;
    idiomS s = s ;
```

Special cases, which might fix `tmp` or `pol` in `idiomS`, have to be inserted before the default case. To map the tree of *the early bird catches the worm* of type `Cl` to the constant, we insert a special case before the default of `idiomCl`:

```
fun idiomCl : Cl -> Cl ;
def idiomCl (PredVP (DetCN (DetQuant DefArt NumSg)
                           (AdjCN (PositA early_A) (UseN bird_N)))
                 (ComplSlash (SlashV2a catch_V2)
                   (DetCN (DetQuant DefArt NumSg) (UseN worm_N)))
               = idiom_early_bird_Cl ;
    idiomCl (PredVP np vp) = PredVP np (idiomVP vp) ;
    idiomCl cl = cl ;
```

The syntactic constructors in the tree, including the words, have to be introduced by `data` declarations, so we need to declare some words of the idioms that are not in `DLexicon`:

```
data catch_V2 : V2 ; early_A : A ;
     morgenstunde_N : N ; opossum_N : N ;
```

To be able to translate in the reverse direction, the tree for the German corresponding idiom, *Morgenstund hat Gold im Mund*, must also be mapped to `idiom_early_bird_VP` in `idiomCl`.[122]

Parsing the English idiom then gives two trees, the ordinary parse of the expression and the one containing the idiom constant. These are identified[123] and translated to the German idiom by

[122] These pattern trees contains language-specific words, so the transfer function can be applied to trees obtained from parsing any language: only pattern trees with words of the input language can match the input tree.

[123] But not when using `pt -normalize=idiomS`, since then the second tree is mapped to a tree `UseCl tmp pol (idiomCl idiom_early_bird_Cl)`. Strange bug?: if `data ok : Cl`, then `pt -compute (idiomCl ok)` gives `ok`, but if `fun ko : Cl`, then `pt -compute (idiomCl ko)` only gives `idiomCl ko`, ignoring the default `idiomCl cl = cl`. Sure, we need that `pt -compute (f ko)` gives `(f ko)` if `f` has *no* computation rule (a syntactic constructor).

```
DLang> p -lang=Eng -cat=S "the early bird catches the worm"
          | pt -tr -normalize=idiomCl -nub | l -lang=Ger
UseCl (TTAnt TPres ASimul) PPos idiom_early_bird_Cl

Morgenstund hat Gold im Mund
```

While idioms of type `Cl` can be varied in tense and polarity, idioms of type `VP` can additionally be combined with any noun phrase to a clause. The example `idiom_simulate_dead_VP` can be treated as special case in

```
fun idiomVP : VP -> VP ;

def idiomVP (ComplSlash (SlashV2a play_V2)
               (DetCN (DetQuant DefArt NumSg) (UseN opossum_N)))
      = idiom_simulate_dead_VP ;
```

The idiomatic verb phrase can be part of sentences and provide literal and idiomatic translations:[124]

```
DLang> p -lang=Eng "John would have played the opossum" | l -lang=Ger
Johann würde das Opossum gespielt haben
Johann würde sich totgestellt haben
```

Similarly, we can have idiomatic expressions of type `VPSlash` that combine with two complements, e.g. *jmdm den Kopf waschen*.

Remark 108: In some cases, binary predicates can be translated by *linearize* ∘ *parse* via constants of category `VPSlash` or even `V2` in the lexicon, e.g. *to be ashamed of* ↦ *sich schämen für*.

6.5. Translation of multi-word-expressions

So far, we can translate expressions of *basic* syntactic types, using abstract constants of categories `N`, `A`, `V`, `Adv`, or `VP`, `V2`, and `VPSlash` (to translate predicates like *be ashamed* ↦ *sich schämen*).[125]

More difficult to handle are multi-word-expressions, e.g. to translate the predicate *give a talk* into *einen Vortrag halten*: the nouns *talk* and *Vortrag*, of common meaning, are combined with specific verbs *give* and *halten* of different and here irrelevant meanings. But *a talk* and *einen Vortrag* here are not fixed complements, but just represent a range of noun phrases with fixed head nouns, as in *gave two interesting talks on birds*. Depending on which parts of the noun phrase complement are missing, we need constants of *non-basic* syntactic types like `Det -> VP` and `Det -> AP -> VP`. Corresponding special cases have to be inserted into `idiomVP`:

```
-- multi-word-expressions: more variation
idiomVP (ComplSlash (SlashV2a give_V2) (DetCN det (UseN2 talk_N2)))
  = give_talk_VP det ;
idiomVP (ComplSlash (SlashV2a give_V2) (DetCN det (ComplN2 talk_N2 np)))
  = give_talk_on_VP det np ;
```

[124] *If S, two birds may be killed with the one stone* ↦ *Falls S, könnte man zwei Fliegen mit einer Klappe schlagen.*

[125] Q76: Can we handle reflexive predicates like *to do one's best*, *to care about the health of one's children* etc., or *to hurt one's leg* ↦ *sich das Bein verletzen*.

```
    idiomVP (ComplSlash (SlashV2a give_V2)
                (DetCN det (AdjCN ap (UseN2 talk_N2))))
      = give_AP_talk_VP det ap ;
    idiomVP (ComplSlash (SlashV2a give_V2)
                (DetCN det (AdjCN ap (ComplN2 talk_N2 np))))
      = give_AP_talk_on_VP det np ;
```

Again, these special cases should be followed by cases going down to subtrees and applying transfer functions of appropriate type, until a final default case:

```
    idiomVP (AdvVP vp adv) = AdvVP (idiomVP vp) adv ;
    idiomVP (AdVVP adv vp) = AdVVP adv (idiomVP vp) ;
    idiomVP (ProgrVP vp) = ProgrVP (idiomVP vp) ;
    idiomVP (ComplVV vv vp) = ComplVV vv (idiomVP vp) ;
    idiomVP (ComplVS vs s) = ComplVS vs (idiomS s) ;
    idiomVP vp = vp ;
```

Sentences with this multi-word-expression can be correctly parsed and translated:

```
DLang> p -lang=Eng -cat=S -tr "John will give a good talk on birds" | l -lang=Ger
...
UseCl (TTAnt TFut ASimul) PPos (PredVP (UsePN john_PN)
   (give_AP_talk_on_VP (DetQuant IndefArt NumSg) (PositA good_A)
      (DetCN (DetQuant IndefArt NumPl) (UseN bird_N))))
...
Johann wird einen guten Vortrag über Vögel halten
```

However, there will be several different readings where *on birds* is viewed as modifying adverb, and these cannot be collapsed to a single tree by `pt -normalize=idiomVP` any more.

6.6. The modules `DIdiomTransfer` and `DLang`

A module of transfer functions using any syntactic constructors of `Lang` in tree patterns can then be built by extending `DGrammar` and `DLexicon`. We let it contain the above `data` and `fun` declarations and `def` computation rules:

```
--# -path=.:../../src/abstract
-- 12/2024, needs modified gf-core/src/compiler/GF/Commands|TreeOperations

abstract DIdiomTransfer = DGrammar, DLexicon ** {

  data
    opossum_N : N ; ...
    give_AP_talk_on_VP : Det -> AP -> AP -> NP -> VP ;

  fun
   idiom_simulate_dead_VP : VP ; ...

  fun
   idiomS : S -> S ;
```

```
  def
   idiomS (UseCl tmp pol cl) = UseCl tmp pol (idiomCl cl) ;
   ...

  }
```

We can then build a version of `Lang` in which all syntactic constructions can be used in patterns of transfer functions, by combining `DGrammar` and (for convenience) `DLexicon` with the above module of transfer functions:

```
--# -path=.:../../src/abstract:../../common:
abstract DLang =
  DGrammar,
  DLexicon,
  DIdiomTransfer      -- HL 12/2024
  ** {
  flags startcat=Phr ;
  } ;
```

The module `DIdiomTransfer` declares transfer functions (which have no linearizations), but also some syntactic constructions that need language-specific linearizations. Hence, concrete modules implementing `DIdiomTransfer` are needed in concrete languages implementing `DLang`.

```
--# -path=.:../../src/abstract:../../src/common:../../src/prelude:../../src/german

concrete DIdiomTransferGer of DIdiomTransfer =
 CatGer ** open ParadigmsGer, DGrammarGer, DLexiconGer in {

  lin
    idiom_early_bird_Cl =
      (PredVP (MassNP (UseN morgenstunde_N))
         (ComplSlash (AdvVPSlash (SlashV2a have_V2)
                        (PrepNP in_Prep (DetCN (DetQuant DefArt NumSg) (UseN mouth_N))))
            (MassNP (UseN gold_N)))) ;

    idiom_simulate_dead_VP =
      UseV (reflV (prefixV "tot" (regV "stellen")) accusative) ;

    opossum_N = reg2N "Opossum" "Opossums" neuter ;
    early_A = mkA "früh" "früher" "früheste" ;
    catch_V2 = dirV2 (irregV "fangen" "fängt" "fing" "finge" "gefangen")  ;
    morgenstunde_N = mkN  "Morgenstund" "Morgenstunden" feminine ;

    give_V2 = dirV2 (irregV "geben" "gibt" "gab" "gebe" "gegeben")  ;
    talk_N2 = mkN2 (mkN "Vortrag" "Vorträge" masculine)
                   (mkPrep "über" "über den" "über die" "übers" accusative);

    give_talk_VP det =
      (ComplSlash (SlashV2a hold_V2) (DetCN det (UseN2 talk_N2))) ;
```

```
    give_AP_talk_VP det ap =
      (ComplSlash (SlashV2a hold_V2) (DetCN det (AdjCN ap (UseN2 talk_N2)))) ;
    give_talk_on_VP det np =
      (ComplSlash (SlashV2a hold_V2) (DetCN det (ComplN2 talk_N2 np))) ;
    give_AP_talk_on_VP det ap np =
      (ComplSlash (SlashV2a hold_V2) (DetCN det (AdjCN ap (ComplN2 talk_N2 np)))) ;
}
```

Linearizations for words in the English idioms are just provided to support the literal translations. Notice that in the linearizations of `give_talk_VP` etc., the verb `give_V2` is replaced by `hold_V2`, so that *give a talk* can be translated to *einen Vortrag halten*.

7. Lose Ends

7.1. How to Prove Properties of a Resource Grammar?

We can do testing, but how can we prove correctness, or any other property of GF's resource grammars? This is a difficult question.

- First, all grammars are incomplete: for example, the resource grammars have no verbal category of verbs of arity greater than 3, but languages have such verbs.
- Second, the prepositions of the RGL are English prepositions, and neither do they cover all prepositions, nor do the given ones map one-to-one to prepositions of other languages, likewise for adverbs (`then/when/when?` in `Eng` corresponds to `dann/als|sobald|nachdem/wann?` in `Ger`). So, correct translation of adverbials calls for a comprehensive set of adverb constructors, possibly organized in several *adverbial dimensions*, i.e. temporal, local, directional, causal, etc. prepositions or pro-adverbs.

  Similarly, categories such as `PConj` for "*phrase-beginning conjuncts*" will not contain "the same" words in all languages, and hence they cannot have a simple translation.
- Third, different sequences of rule applications may produce the same record in one language, but different ones in another language. For example, the sequence may be relevant for the word order in one language but not in the other. How can the word orders be made corresponding between those language where order matters?

On the other hand, the advantage of writing grammar *rules* is that we can improve and correct them, so we can reason about the constructions, and guarantee certain properties. For example, we would like to prove that the two trees

```
ComplSlash (Slash3V3 v np3) np2  ===  ComplSlash (Slash2V3 v np2) np3
```

have the same linarizations, for all `v:V3` and `np2:NP`, `np3:NP`, in all languages (or are different trees intended for different word orders, c.f. below). Putting the implementation issues aside, can we prove such properties using the intended mathematical interpretation of abstract terms (trees) by records with string and parameter fields?

A better example may be

```
ImpersCl vp === PredVP (UsePron it_Pron) vp
```

since

```
IdiomEng.gf:    ImpersCl vp = mkClause "it" (agrP3 Sg) vp ;
SentenceEng:    PredVP np vp = mkClause (np.s ! npNom) np.a vp ;

IdiomGer.gf:    ImpersCl vp = mkClause "es" (agrP3 Sg) vp ;
SentenceGer:    PredVP np vp = let subj = mkSubject np vp.c1
                               in mkClause subj.s subj.a vp ;
ResGer.mkSubject (UsePron it_Pron) Nom = {s = "es" ; a = agrP3 Sg}
```

So it seems that `ImpersCl` can be reduced to the "core" language without `Idiom`. This could be expected, but in fact the implementations in `IdiomEng`, `IdiomGer` are based on `ResEng`, `ResGer`,

not only on constructors of other grammar modules (which contain `it_Pron`). However, the use of `it_Pron` here is in fact a ***correlate*** for the moved infinitival subject, or sentential subject: *es ist gut, dass S*, or *es ist gut, ... zu tun.*

More generally, it would be nice to implement *normalization* of trees (say, to give a normal form to realize a possessive function, or relate passive clauses to active ones, or translate adjectival attributes to relative clauses, etc. (This should be part of `Transfer`.)

Having such proofs, it would be nice to record the assumptions used, and re-establish them after grammar modifications. (i.e. do regression tests of general properties of the grammars, not just of individual parse results.)

Word order and parsing: do we want two trees, or use a new `ComplV3` instead?

```
Lang> p -tr -lang=Eng -cat=Cl "I sell the dog to the man" | l
PredVP (UsePron i_Pron) (ComplSlash (Slash2V3 sell_V3
(DetCN (DetQuant DefArt NumSg) (UseN dog_N)))
(DetCN (DetQuant DefArt NumSg) (UseN man_N)))
PredVP (UsePron i_Pron) (ComplSlash (Slash3V3 sell_V3
(DetCN (DetQuant DefArt NumSg) (UseN man_N)))
(DetCN (DetQuant DefArt NumSg) (UseN dog_N)))

I sell the dog to the man
ich verkaufe dem Mann den Hund
I sell the dog to the man
ich verkaufe dem Mann den Hund
```

If we insert objects into the same `nn:Str` field, the order of objects must be reflected in the tree. If we insert the objects in separated components of `nn:Str *...* Str`, different trees have the same record, and the record is more abstract than the trees. We then need a default relative ordering of the `nn` fields. Or some (invisible) parameters that choose between the possible relative orders, similar to the `t:Temp` with empty string component.

Q77: What is needed of `RNP` to be able to prove that `ReflRNP : VPSlash -> RNP -> VP` is a generalization of `ReflVP : VPSlash -> VP`?

Generally, when is an abstract rule implemented correctly? For example, are there criteria how a constituent constructed by a `Slash*`-rule has to behave? If the constituent embedded in a relative clause (with the relativizing element extracted) or if it embedded in a complementation rule works correctly?

Proposal 4: There ought to be a set of abstract, semantically motivated prepositions as adverb constructors. (c.f. Remark 7.) ◁

7.2. Problems

Todo 58: There are some problems with the existing `LangGer`.

- Nesting depth of `VV` or `V2V` complements:
- Sometimes, a comparison of `Ger` with `Eng` is useful or needed to understand the motivation of constructions in the `RGL`. For example, the difference between `AdvVP` and `AdVVP` apparently makes little sense for `Ger`, because only English has a distinction between adverbs coming before and adverbs coming after the verb.

- Modification rules can be applied iteratively, which may be ok for `AdvCN:CN -> Adv -> CN`, but is dubious for `RelCN:CN -> RelS -> CN`. In some cases, only a single modification is correct. However, `GF` does not allow us to have partial rules that are not applicable to a `CN` already modified (though the iterated modification might return its argument `cn` unchanged). See `nfCN` in `DIdiomTransfer.gf`.

- Inflection paradigms `s : Parameters => Str`

- Adjectives with object sentences or object infinitives have category `A`, e.g. `probable_AS` and `fun_AV`. Hence, an object sentence can also be analysed as an adverbial sentence, which makes no sense, e.g. *John is glad that he can sleep*, c.f. the remark to lexical adjectives in Section 53.

- Compound nouns in `DictGer` are often not inflected correctly: if they are composed of adjective and noun, like `grasgruene_taeubling`, the adjective is not inflected: e.g. **des Grasgrüne_ Täublings*.

- `CAdv` is used in combination with an adjective in

  ```
  ComparAdvAdj  : CAdv -> A -> NP -> Adv ; -- more warmly than John
  ComparAdvAdjS : CAdv -> A -> S  -> Adv ; -- more warmly than he runs
  ```

  to form an adverb, and with

  ```
  CAdvAP  : CAdv -> AP -> NP -> AP ; -- as cool as John
  ```

  to form an adjective phrase. In the first usage, the comparative adverb can govern the degree of the adjective, e.g. *more + good = good ! Compar = better*, in the second usage, it cannot, as the adjective phrase has a fixed degree. Q78: Can we let `CAdv` determine the degree of an adjective and still add complements? If we had

  ```
  CAdvAP : CAdv -> A -> NP -> AP ;
  CAdvA2P : CAdv -> A2 -> NP -> NP -> AP ;
  ```

  we could build *genau so stolz auf np1 wie np2* and *stolzer auf np1 als np2*. If we let `AP.s : Degree => AForm => Str`, each usage of an `ap` has to somehow choose a degree.

  There is yet no `ComplComparA2 : A2 -> NP -> AP` for adding an object to an `A2` in comparative, e.g. *stolzer auf sein Kind (als wir)*.

- Sometimes, prefixverbs have a modifiable prefix, e.g. `wohlfühlen_rV:V` = *sich wohlfühlen*. But how can we modify the separable prefix, as in "*der Pazifische Makrelenhecht, der sich bei 15 bis 18 Grad im Wasser am wohlsten fühlt*" (SZ 16.Februar 2024, Nr.39, S.14)

- How to parse *damals war es für sie so, als sei ... der Fall*, or *so, als ob ... der Fall sei*? Do we have an adverb construction `as_if : Cl -> Adv`?

- It seems that `most_Predet` is specific to `Eng`, e.g. *most of [the children | my old articles]*. In `Ger`, we'd rather need a quantifier `most_Quant`, e.g. *die meisten [der Kinder | meiner alten Artikel]* (c.f. Remark 100.) So we seem to need a transfer `Eng.most_Predet` $\mapsto$ `Ger.most_Quant`. Maybe `most:Predet` is incorrect even for English. Was it intended for *most small houses*, not for *most of the small houses* only?

Remark 109: The rule `ExtraGer.PassVPSlash : VPSlash -> VP` is overwritten in `TestLangGer`.

Example problems:

- Left extraction: SZ 26.2.2022 *hat die EU gegen Russland das härteste Sanktionspaket in der Geschichte des Wirtschaftsblocks erlassen* from *Paket von Sanktionen gegen Russland.*
- `PossNP` and `AdvNP`: Notice the complement *auf die Ukraine* of *Angriff*, in contrast to the adverbial modification (or complement?) *für die Ukraine* of *Folgen* in
  - *die wirtschaftlichen Folgen des russischen Angriffs auf die Ukraine*
  - *die wirtschaftlichen Folgen des russischen Angriffs für die Ukraine*

  It seems that objects are closer to N than possessive genitives, and these closer than adverbial modifiers.

  GF has `PossNP` and `PartNP` to attach a noun phrase (in genitive resp. with *of*) to a common noun. It should also have `ComplSubjN2` besides `ComplN2`, for subject and object complement to binary nouns: *the education of children by adults*, *die Ermordung des Cäsars durch Brutus*, *der Mord des Brutus an Cäsar*. The concrete type of `N2` seems to need fields `c1,c2:Preposition` to relate the subject- and object-complement to the noun (at least for nouns derived from binary verbs).
- It also seems that objects (of nouns) are closer than adjective or adverbial modifiers, but should we identify *(nice ((talk (on birds)) in Paris))* with *(nice (talk (in Paris)) (on birds))* by tree normailzations, while keeping the difference to *nice talk (on (birds in Paris))*? Or should we use different scopes to represent possibly different word orders? How could this work across languages?

Q79: Can we accept variants and generate a normal form of an expression? This would be very useful to let less used variants be accepted in parsing, but only a specific form be generated in linearization. For example, `ConstructionGer` defines

```
timeunitAdv n time =
  let n_hours_NP : NP = mkNP n time
  in  SyntaxGer.mkAdv (for_Prep | P.accPrep) n_hours_NP ;
```

with alternative tree constructions. Both "*elf Wochen*" and "*für elf Wochen*" are parsed to the same tree

```
timeunitAdv (NumNumeral (num (pot2as3 (pot1as2 pot111)))) week_Timeunit
```

But the first construction seems to be the "normal form" for generation:

```
Lang> p -tr -cat=Adv "elf Wochen" | l
timeunitAdv (NumNumeral (num (pot2as3 (pot1as2 pot111)))) week_Timeunit

für elf Wochen
```

The other form is also generated when linearizing the full inflection table (using `l -table`):

```
Lang>  p -cat=Adv "elf Wochen" | l -table
s : für elf Wochen
s : elf Wochen
```

A resource grammar perhaps ought not to have such alternatives. For application grammars, however, they seem rather useful, provided this works similarly for nested alternatives and the selection of the "normal form" is stable under compiler optimizations. ◁

Remark 110: `ApposCN`, `DetNP` and `MassNP` cause rather incorrect trees, e.g. for *er lebt in seinem Haus seiner Frau*:

```
PredVPRAdv (UsePron he_Pron) (UseV live_V)
  (PrepRNP in_Prep (ReflPoss NumSg (ApposCN (ApposCN (UseN house_N)
     (DetNPFem (DetQuant (PossPron i_Pron) NumSg))) (MassNP (UseN woman_N)))))
```

A *lot* (78%) of incorrect trees are obtained by

```
ReflPredicates>
p -cat=VP "seinem Chef seinen Hund zu verkaufen" | ? wc -l                => 79
p -cat=VP "seinem Chef seinen Hund zu verkaufen" | ? grep MassNP | wc -l  => 62
p -cat=VP "seinem Chef seinen Hund zu verkaufen" | ? grep ApposCN | wc -l => 62
p -cat=VP "seinem Chef seinen Hund zu verkaufen"
  | ? grep -v ApposCN | grep -v he_Pron | grep -v it_Pron
ComplRSlash (SlashR3V3 sell_V3 (ReflPoss NumSg (UseN boss_N)))
            (ReflPoss NumSg (UseN dog_N))
ReflRNP (SlashR3V3 sell_V3 (ReflPoss NumSg (UseN boss_N)))
            (ReflPoss NumSg (UseN dog_N))
```

Generally, a `CN` can be possessively post-modified by an `NP` in genitive, like *Haus seiner Frau*, but then should not additionally be possessively pre-modifiable by an `NP` in genitive or a possessive pronoun, like *mein Haus seiner Frau* or *your house of mine*. In principle, syntactic constructions are typed *partial* functions, i.e. 'good' arguments should satisfy properties not encoded in their type, but the syntactic constructions of GF are *total*. To fix this, the linearization of a syntactic construction of GF could set an error-field in the constructed expression if its (type-correct) argument is not 'good' according to the values of its fields.

Q80: Noun complements and compound nouns: there is a difference between *(finite automata) theory = theory of finite automata* and *modern (automata theory) = modern theory of automata*. Using words of `Lexicon.gf`, we get two trees for *new paper industry*:

```
Lang> p -lang=Eng -tr -cat=NP "the new paper industry" | l
DetCN (DetQuant DefArt NumSg) (AdjCN (PositA new_A)
  (ApposCN (UseN paper_N) (MassNP (UseN industry_N))))
DetCN (DetQuant DefArt NumSg) (ApposCN (AdjCN (PositA new_A) (UseN paper_N))
                                       (MassNP (UseN industry_N)))
the new paper industry
das neue Papier Industrie
the new paper industry
das neue Papier Industrie
```

The trees are clearly wrong (at least in German, where the head noun *Industrie* is feminine). It seems that *paper* in *paper industry* is a modifier or object of *industry*. In German, *endliche Automatentheorie* is ungrammatical, it has to be *Theorie endlicher Automaten*. (Duden: **der chemische Fabrikbesitzer*, but: *der klassische Gitarrenunterricht*.) These constructions involve compound nouns in German. `ParseEngAbs` has declarations `fun CompoundSgCN, CompoundPlCN : CN -> CN -> CN`.

7.3. ExtraGer: what to do to improve parsing

Todo 59: We should write a new version of the parsing grammar `ParseGer` that omits some overgenerating rules of `Grammar` and adds, besides rules of `ExtendGer`, further rules specific to German, to be collected in `ExtraGer`.

1. Exclude `Noun.DetNP` and replace it by `DetNPMasc, DetNPFem, DetNPNeutr : Det -> NP`, using an operation `oper DetNP : Det -> Gender -> NP` with parameter type `Gender`.
2. Exclude `MassNP : CN -> NP`, and replace it by a category `MN` of mass nouns, a category `CMN` of common mass nouns, and rules `UseMN : MN -> CMN` and `DetCMN : Det -> CMN -> NP` to generate `water_MN:MN`, `klares Wasser:CMN` and `viel klares Wasser:NP`. However, count nouns are also used without article, e.g. *Überfischung und Klimawandel sind die Hauptgründe für die Entwicklung*.
3. Remove rules `AdvRNP`, `AdvRVP` and `AdvRAP` of `Extend` and replace them by `RAdvNP`, `RAdvVP` and `RAdvAP` using a category `RAdv` of reflexive adverbs, see Remark 23.
4. Implement splittable adjective and adverb phrases, e.g. *weil die Inflation größer war als die Lohnerhöhungen* and *dass die Bürger weniger konsumieren als erwartet*
5. Add complement frames to nouns and adjectives in `LexiconGer`, e.g. *Grund für etwas*, *Hoffnung auf etwas*. (But if we can't suppress adverbial readings of these prepositional objects, what do we gain?)
6. Implement sentential correlates: *(Gründe) dafür, daß sich die Wirtschaft nur zaghaft erholt*, and allow them as alternative to prepositional objects.
7. Implement the extraction of relative clauses of nominal objects behind the non-finite verb part.

7.3.1. Structural ambiguiuties

1. Replace the rules to modify a noun by `AP`, `Adv`, `RelS` to a joint modification, so that different orders of modifiers are ignored. (With dependent category `(CN am advm rsm)` as in the Riga Summerschool, or by reduction to a normal form and identification of trees?) Can we similarly collaps modifications of `NP`s, or identify modifications of `DetCN det cn` with modifications of the embedded `cn`?
2. German adjectives can be read as adverbs, in context where they shouldn't:

   ```
   TLang> p -cat=Cl "ich bin jung"
   PredVP (UsePron i_Pron) (UseComp (CompAP (PositA young_A)))
   PredVP (UsePron i_Pron) (UseComp (CompAdv (PositAdvAdj young_A)))
   PredVP (UsePron i_Pron) (AdvVP UseCopula (PositAdvAdj young_A))
   ```

   Can we collaps the three readings to the first one (by `pt -transfer` and `pt -nub`) ?
3. For parsing from German, can we omit `someSg_Det` from the abstract syntax, or normalize it to `DetQuant IndefArt NumSg`? Rather not; they differ in Engslish: *some* vs. *a/one*.) Can we ignore, i.e. normalize away, the cardinal 1 used as number in determiners?

4. A binary adjective `a2` or noun `n2` that combines with a nominal object `np` by a preposition `a2.c2, n2.c2 : Preposition` can always be used without complement and the resulting `AP` or `CN` be modified by the adverb `adv = PrepNP p np` via

```
AdvVP (UseComp (CompAP (UseA2 a2))) adv
AdvVP (UseComp (CompCN (UseN2 n2))) adv
```

   Q81: How can one enforce that complementation precedes modification? A (language-independent?) *operator precedence declaration* should instruct the parser to only extract one of the two structures from the parse table. Can the parser compare the preposition `p` of the adverb `adv` in the trees above with the preposition `p' = a2.c2` in the complement construction `ComplA2 a2 np` resp. `ComplN2 n2 np`? Sure, on the abstract language level, we cannot compare the trees

```
(AdvCN (UseN2 n2) (PrepNP p np)) = (ComplN2 n2 np)
```

   and see if `p` is the preposition `n2.c2` used to bind the complement `np` to `n2`.
5. An adverb may be moved from a subordinate clause to the main (or superordinate) clause, and can be read as adverb of the main clause: *dass die Forscher wegen des Rückgangs der Inflation damit rechnen, dass die Zentralbanken [?] die Zinsen wieder senken*

Todo 60: Add testfiles `*.gfs` or `*.gftest` ?

Remarks: 110, Todos: 60, Questions: 81, Proposals: 4

References

[1] K. Angelov. *The Mechanics of the Grammatical Framework*. PhD thesis, Chalmers University of Technology, Gothenburg, Sweden, 2011.

[2] G. Drosdowski, editor. *DUDEN "Grammatik der deutschen Gegenwartssprache"*, volume 4., völlig neu bearbeitete und erweiterte Auflage. Bibliographisches Institut, Mannheim/Wien/Zürich, 1984.

[3] P. Eisenberg. *Grundriß der deutschen Grammatik*. J.B.Metzlersche Verlagsbuchhandlung, Stuttgart, 1986.

[4] U. Johnson. *Mutmassungen ber Jakob*. Suhrkamp Verlag, Frankfurt a.M., 1959.

[5] Y. Kaji, R. Nakanishi, and H. Seki. The computational complexity of the universal recognition problem for parallel multiple context-free grammars. *Computational Intelligence*, 10(4):440–452, 1994.

[6] J. Macheiner. *DAS GRAMMATISCHE VARIETÉ oder Die Kunst und das Vergnügen, deutsche Sätze zu bilden*. Eichborn Verlag, Frankfurt am Main, 1998.

[7] R. Nakanishi, K. Takada, and H. Seki. An efficient recognition algorithm for multiple context-free languages. In *MOL 5: 5th Meeting on the Mathematics of Language*, pages 119–123, Saarbrücken, Germany, 1997.

[8] A. Ranta. Grammatical framework, a type-theoretic grammar formalism. *Journal of Functional Programming*, 14(2):145–189, 2004.

[9] A. Ranta. *Grammatical Framework: Programming with Multilingual Grammars*. CSLI Publications, Stanford, 2011. ISBN-10: 1-57586-626-9 (Paper), 1-57586-627-7 (Cloth).

[10] H. Seki, T. Matsumura, and T. Kasami. On multiple context-free grammars. *Theoretical Computer Science*, 88(2):191–229, 1991.

Index

Status of Dictionaries

There are 16 dictionaries under `gf-rgl/src/`:

```
~/GF/gf-rgl/src$ ls -l */Dict???Abs.gf
 1347458 Sep 11  2018 english/DictEngAbs.gf
 1684580 Sep 11  2018 estonian/DictEstAbs.gf
     556 Sep 11  2018 finnish/DictFinAbs.gf
 2372417 Sep 11  2018 french/DictFreAbs.gf
  956238 Aug  5 18:09 german/DictGerAbs.gf
  138051 Feb  3  2022 icelandic/DictIceAbs.gf
 3041777 Feb  7  2022 latin/DictLatAbs.gf
 1586314 Jul  5  2019 latvian/DictLavAbs.gf
  126834 Sep 11  2018 maltese/DictMltAbs.gf
  491611 Sep 11  2018 mongolian/DictMonAbs.gf
 3020372 Jul  5  2019 portuguese/DictPorAbs.gf
   82468 Feb  7  2022 russian/DictRusAbs.gf
 1010377 Jul  5  2019 spanish/DictSpaAbs.gf
     336 Sep 11  2018 swedish/DictSweAbs.gf
  541219 Sep 11  2018 turkish/DictTurAbs.gf

~/GF/gf-rgl/src$ ls -l */Dict???.gf
 2985699 Sep 11  2018 english/DictEng.gf
 8163388 Sep 11  2018 estonian/DictEst.gf
     561 Sep 11  2018 finnish/DictFin.gf
 5015320 Aug  7 16:37 french/DictFre.gf
 2583201 Aug  5 18:09 german/DictGer.gf
  719729 Feb  3  2022 icelandic/DictIce.gf
 4065682 Feb  7  2022 latin/DictLat.gf
 3350709 Jul  5  2019 latvian/DictLav.gf
  293722 Sep 11  2018 maltese/DictMlt.gf
  968876 Sep 11  2018 mongolian/DictMon.gf
 8310232 Jul  5  2019 portuguese/DictPor.gf
  269269 Feb  7  2022 russian/DictRus.gf
 1811727 Jul  5  2019 spanish/DictSpa.gf
     344 Sep 11  2018 swedish/DictSwe.gf
  876085 Jul  5  2019 turkish/DictTur.gf
  634522 Sep 11  2018 urdu/DictUrd.gf
```

How many entries of a given category `C` of `Grammar` do exist in these dictionaries? Counted by `gf-rgl/src> grep "_C " */Dict*Eng*Abs.gf | wc -l` for language Eng etc.

There are also 5 dictionaries in `gf-rgl/src/morphodict`, for `Eng`, `Fin`, `Ger`, `Ita`, `Swe`.

	Eng	Est	Fin	Fre	Ger	Ice	Lat	Lav	Mlt	Mon	Por	Rus	Spa	Swe	Tur	Urd
	4016	7211	0	8751	4698	860	3417	20420	4146	7415	0	1008	5218	7322	2657	3551
2	6730	616	0	0	3870	0	3529	0	0	1378	0	0	2	554	0	7040
3	75	0	-	-	742	-	0	-	-	99	-	-	0	0	0	0
	112	18	0	0	55	0	0	0	0	4	0	119	0	0	0	8
	72354	42392	60130	59297	61825	5257	19116	33325	0	9413	84457	952	29010	74833	18890	23428
2	3	0	0	3	0	0	0	0	0	1	0	3	0	0	0	0
	25852	4044	2932	22525	15774	784	8822	0	0	5394	44072	984	5571	18638	659	18573
2	7	0	0	1	0	0	0	0	0	1	0	1	0	0	0	0
	2375	2418	10899	1570	1462	4	2162	0	0	0	0	993	1130	1162	0	941
5	367	0	0	0	0	0	87	0	0	0	0	0	27	179	0	32

Differences of GF from what the GF-book says

These differences occurred with `Grammatical Framework (GF) version 3.11.0`

1. The gf-shell command `pt = put_tree` no longer supports the option `-paraphrase` mentioned on p. 144 of the F-book
2. The gf-shell command `pt = put_tree` no longer supports the option `-transfer=f` that is mentioned on p.195 of the GF-book. (So we can no longer parse and apply the tree transform function `f` to the result, `p "..." | pt -transfer=f | l`, which was useful for translation to a single target language.)
3. Contrary to C.3.6 (p.267) of the GF-book, one cannot separate a declaration `data f : A -> C;` to `fun f : A -> C; data C = f;`. A conflict results when adding `data NP = DetCN;` to `Noun`. According to C.3.6, "all these constructor definitions must appear in the same module in which the category is itself defined." While above `NP` is defined in the module `Cat` which is not "the same" module as `Noun = Cat ** { fun DetCN : Det -> CN -> NP; ... }`, the conflict also results if a category `C`, a declaration `fun f : A -> C`, and the separate declaration `data C = f` appear in the same module.
   However, a declaration `data f : A -> C` need not be in the module where `C` is defined, so replacing `fun DetCN : Det -> CN -> NP;` in `Noun` by `data DetCN : Det -> CN -> NP;` works. This is heavily used in `Lang`, e.g. for `data UseV : V -> VP` in `Verb`.
4. The GF-compiler does not type-check in

   ```
   table (Ints n) [value_0,...,value_n] ! k
   ```

   whether $k \leq n$ and raises an `Internal error in Compute.ConcreteNew`. But it detects a type mismatch between `Case` and `Int` in `table (Case) [value_0,...,value_n] ! 4`.

Appendix: What needs to be added or improved?

1. a better test lexicon: for *all* lexical categories and inflection classes (for a given language), the (language-specific) text lexicon should have an instance.
   (c.f. `GF/gf-rgl/tests/german/TestLexicon.gf`)
2. a more complete lexicon of structural words: all pronominal adverbs, all pronominal adjectives (*so*, *solch*, *ander*, *welch*), subjunctions like *bevor*, *ehe*, *als*, *während*, *nachdem*
   (Bem. *eher-als+am ehesten*:Adv?)
3. a semantically organized lexicon of all? prepositions as adverb constructors, so that adverbs can be translated in a meaning-preserving way
4. a large treebank of abstract trees, organized according to the syntax modules. Some cases that show all variations, some that show the tree structure in typical case. (See
   - `GF/gf-rgl/treebanks/rgl-exx.txt` 15k From trees, treebanks for Eng, Swe, Bul are generated by the gf-script `GF/gf-3.6/testsuite/libraries/exx-resource.gfs`
   - `GF/gf-rgl/treebanks/rgl-api-trees.txt` 991 trees, 104k, 2018
   - `GF/gf-rgl/treebanks/ud-rgl-trees.txt` 64 trees, 12k, 2019
   - `GF/gf-rgl/treebanks/numeral-trees.txt` 10 trees
   - `GF/gf-rgl/treebanks/pron-ordering-trees.txt.*V3`
   - `GF/gf-rgl/tests/german/object-order.README`
   - `GF/gf-rgl/tests/german/infinitives*.trees` 16975+1266
   - `GF/gf-rgl/tests/german/trees.eng.tmp` 16443
   - `GF/gf-rgl/tests/german/parsetrees.eng.tmp` 9895
   - `GF/gf-rgl/tests/german/passive.pos|neg|dub.trees` 5366,768,744
   - `GF/gf-rgl/tests/german/relativeClause.trees` 1952
   - `GF/gf-rgl/tests/german/rnp.trees` 2880
   - `GF/gf-rgl/tests/german/s.eng.trees.gf-3.9` 16358
   - `GF/gf-rgl/tests/german/todo.trees` 2085
   - `GF/gf-rgl/tests/german/vp.eng.trees` 8939
   - `GF/gf-rgl/tests/german/vpadv.trees` 4919
   - `GF/gf-rgl/tests/german/vp.trees` 9172
   - `Vorlesungen/GrammaticalFramework-14-15/grammar.trees`, 9106 K,
   - `Vorlesungen/GrammaticalFramework-12-13/exx-resourceGer.txt`)
   - `Magisterarbeiten/Baatarkhuu/gf-trees/examples.abs`, 7235 K
   - `Magisterarbeiten/Erdenebadrakh/mongolian/test.trees` 10755
   - `GF/gf-3.7.1/lib/src/german/examples.abs`
   - `gf-3.8` = `GF/gf/lib/src/german/trees|trees.AllGer|.abs`

   See also the `.txt` files in these directories for example input to the parser. Can we have text examples with acceptance bit, as in `passive.txt` ?
5. movement of extractable parts: specific rules may give expressions with moved extractions, for example `ComplSlashExt` for nominal objects with relative clause extracted behind the infinite verb part, or `ApposCNExt` for post-nominal apposition in commata.
6. grammar rules for participle constructions (c.f. Magisterarbeiten/Derkatcheva)

Files with bugs:

1. Vorlesungen/GrammaticalFramework-12-13/gf-errors.txt
2. Magisterarbeiten/Azimi/germanNeu/BUGs.txt
3. Magisterarbeiten/Azimi/germanNeu/TestLexAllAbs.gf
4. GF/gf-bugs.txt
5. GF/gf-rgl/tests/german/bug.adjective - ../bug.sentence
6. GF/gf-3.7.1/lib/src/german/Bugs.hl (20 Fehler, z.T. korrigiert)
7. GF/gf-3.7.1/lib/src/german/Bug.AdvVP-ComplSlash.txt

Changes made:

1. GF/gf-3.7.1/ChangesHL

GF-scripts: see also GF/gf/lib/tests/run.hs+readme.rst, GF/gf/lib/doc/Test.hs

1. GF/gf-3.0/testsuite/libraries/exx-resource.gfs
2. GF/gf-3.7.1/lib/src/german/examples.gfs
3. GF/gf-3.7.1/lib/src/german/examplesLang.gfs
4. GF/gf/lib/src/german/dictnouns|nouns|paradigms|trees*.gfs 2017-2021

Older .gfo-files

1. GF/gf-3.6/lib/src/german/LangGer.gfo
2. GF/gf/lib/src/german/LangGer.gfo 2017-2021

Changes made in older versions of gf-*/lib/src/german/:

```
diff GF/gf-3.6/lib/src/german/*Ger.gf GF/gf-3.6/lib/src/german/*Ger.gf~
<     CleftNP np rs = mkClause "es" np.a -- HL (agrP3 Sg)
>     CleftNP np rs = mkClause "es" (agrP3 Sg)
diff ~/GF/gf-3.6/lib/src/german/CatGer.gf ~/GF/gf-3.6/lib/src/german/CatGer.gf~
<     VP = \vp -> useInfVP False vp ++ vp.ext ;
>     VP = \vp -> useInfVP False vp ;
diff ~/GF/gf-3.6/lib/src/german/LexiconGer.gf  ~/GF/gf-3.6/lib/src/german/LexiconGer.gf~
<   -- talk_V3 = mkV3 (regV "reden") datPrep von_Prep ;
<   talk_V3 = mkV3 (regV "reden") (mkPrep "mit" dative) (mkPrep "über" accusative) ; -- HL
>   talk_V3 = mkV3 (regV "reden") datPrep von_Prep ;
diff ~/GF/gf-3.6/lib/src/german/SentenceGer.gf  ~/GF/gf-3.6/lib/src/german/SentenceGer.gf~
<     EmbedVP vp = {s = useInfVP False vp ++ vp.ext} ;  -- 10/7/16 vp.ext, HL
>     EmbedVP vp = {s = useInfVP False vp} ;

GF/gf-3.2/lib/src/german/StructuralGer.gf:
  whatPl_IP = {s = caselist "was" "was" "was" "wessen" ; n = Sg} ; -- HL: Sg!
  whoPl_IP = {s = caselist "wer" "wen" "wem" "wessen" ; n = Sg} ;  -- HL: Sg for Subj-Verb agr

<   whatPl_IP = {s = caselist "was" "was" "was" "wessen" ; n = Sg} ; -- HL: Sg!
>   whatPl_IP = {s = caselist "was" "was" "was" "wessen" ; n = Pl} ; ----
<   whoPl_IP = {s = caselist "wer" "wen" "wem" "wessen" ; n = Sg} ;  -- HL: Sg for Subj-Verb ag
<                                                                    -- But: Wer(Pl) taeuscht(S
>   whoPl_IP = {s = caselist "wer" "wen" "wem" "wessen" ; n = Pl} ;
```

```
-- The only changes in gf-3.1/*/germans are:
---gf-3.1/lib/src/german/README.hl
For testing, see lib/src/Make.hs

Changes made in LexiconGer:

  airplane_N = reg2N "Flugzeug" "Flugzeuge" neuter ;

  become_VA = mkVA IrregGer.werden_V ;
  close_V2 = dirV2 (IrregGer.schließen_V) ;
  blow_V = IrregGer.blasen_V ;
  burn_V = IrregGer.brennen_V ;
  dig_V = IrregGer.graben_V ;
  fall_V = seinV (IrregGer.fallen_V) ;
  float_V = seinV (IrregGer.treiben_V) ;
  flow_V = seinV (IrregGer.fließen_V) ;
  fly_V = seinV (IrregGer.fliegen_V) ;
  freeze_V = IrregGer.frieren_V ;
  lie_V = IrregGer.lügen_V ;
  sing_V = IrregGer.singen_V ;
  smell_V = IrregGer.riechen_V ;
  stand_V = IrregGer.stehen_V ;
  swim_V = seinV (IrregGer.schwimmen_V) ;
  think_V = IrregGer.denken_V ;

-- No changes in gf-3.0/*/germans (except Lexicon: Flugzeuge)
```

Todo-files:

```
Feb  3  2022 GF/todo.improvements.txt
4369 Jul 13  2017 GF/gf/lib/src/german/README.hl
```

Concerning translation, see Aarne's remarks of 2014 in `GF/gf/lib/doc/translation.txt`

Concerning the gf-shell commands, see `GF/gf-core/src/compiler/GF/Command/Commands2.hs`

To create `.gfo` files for earlier versions, change `../../Setup.hs` by telling haskell/build where to look for the RGL sources:

```
-- | RGL source directory
sourceDir :: FilePath
-- sourceDir = "src"
-- HL, to build original *.gfo's
sourceDir = "../gf-3.9/lib/src"
```

```
From Aarne's mail 22 Feb '22 to the gf-list.

The most systematic unit test treebank is the one displayed in the RGL synopsis,

http://www.grammaticalframework.org/lib/doc/synopsis/
```

with source code in

https://github.com/GrammaticalFramework/gf-rgl/blob/master/doc/synopsis/api-examples.txt

meant to address each RGL structure individually.

However, some systematic work has been done at least in German:

https://github.com/GrammaticalFramework/gf-rgl/tree/master/tests/german

where we have also used Stefan Müller's HPSG test sets.