Predication with Sentential Subject in GF

Hans Leiß
leiss@cis.uni-muenchen.de

Retired from:
Ludwig-Maximilians-Universität München
Centrum für Informations- und Sprachverarbeitung

LACompLing 2018
Stockholm, August 28–31, 2018

Background

- A. Ranta's "Predication Grammar" in GF (Proc. EACL 2014)
  - reuses most of the syntactic constructions of GF's *resource grammar library* (RGL) with non-dependent categories, but implements predication and complementation rules differently
  - categories V, VP are abstract types *depending on arguments*; the implementation types of V, VP are record types
  - categories V, VP, A, AP distinguish between sentential and nominal *object* arguments only

Goal

- Refine the predication grammar by also distinguishing between sentential and nominal *subject* arguments

Contents

- Grammatical Framework's (GF) Resource Grammars (RG)
- A. Ranta's experimental "Predication Grammar"
- Extension by sentential/interrogative/infinitive subjects
- Complexity/Examples

Grammars in GF's Resource Grammar Library (RGL)

Abstract Grammar: declarations of

- syntactic categories as (non-dependent) abstract types
- syntactic constructions as typed function symbols

⟨*SGram.gf*⟩≡

```
abstract SGram = {
  cat S ; NP ; V2 ; VP ;          -- syntactic categories
  fun Pred : NP -> VP -> S ;      -- syntax rules
      Compl: V2 -> NP -> VP ;
      John, Mary : NP ;           -- lexicon (words)
      like : V2 ;
}
```

Function type = context-free rule: `NP -> VP -> S` = `S -> NP VP`

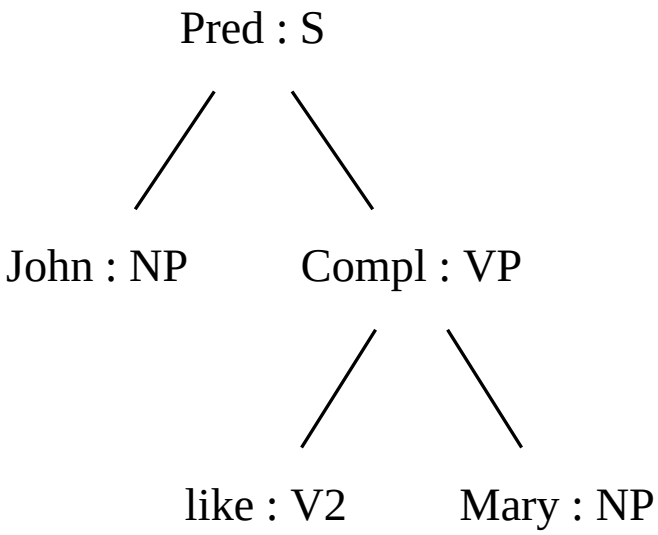
Concrete Grammar: implementations of

- syntactic categories by record types
- syntactic constructions by functions between records

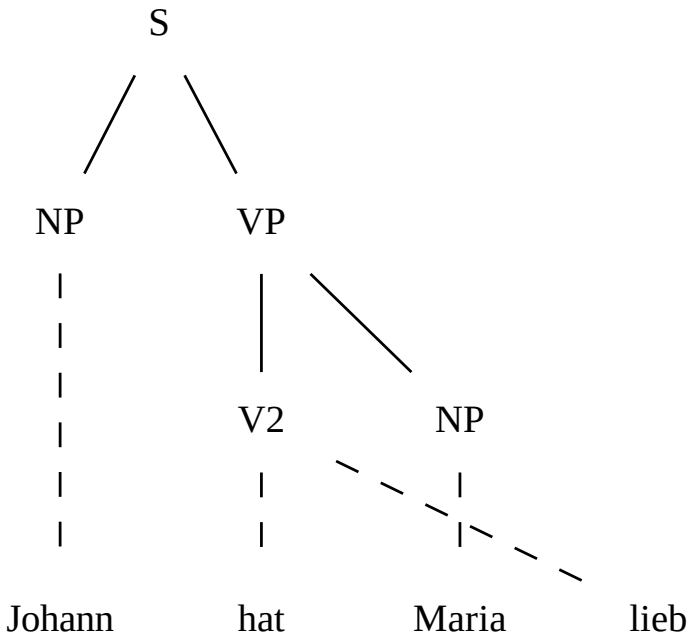
⟨*SGramGer.gf*⟩≡

```
concrete SGramGer of SGram = {
  lincat S = { s : Str } ; NP = { s : Str ; a : Agr } ;
         VP = { s : Agr => Str } ;
         V2 = { s : Agr => Str ; s2 : Str } ;
  lin Pred np vp = { s = np.s ++ vp.s ! np.a } ;
      Compl v2 np =
             { s = \\a => v2.s ! a ++ np.s ++ v2.s2 } ;
      like = { s = table Agr { Sg => "hat" ;
                               Pl => "haben" } ;
               s2 = "lieb" } ;
      John = { s = "Johann" ; a = Sg } ;
      Mary = { s = "Maria" ; a = Sg } ;
  param Agr = Sg | Pl ;              -- parameter type
}
```

```
SGram> p "Johann hat Maria lieb" | vt -view=eog -format=eps
```



abstract tree



parse tree

GF's resource grammars have different categories of verbs:

- `V2`: binary verbs with nominal object (`read,like,know`)
- `VS`: binary verbs with sentential object (`fear,know,hope`)
- `VQ`: binary verbs with interrogative object (`know,wonder`)
- `VV`: binary verbs with infinitival object (`can,want,must`)
- `V3`: ternary verbs with nominal objects (`give,sell`)
- `V2S`: ternary verb with nominal and sentential object (`answer`)
- `V2Q`: ternary verb with nominal and interrogative object (`ask`)
- `V2V`: ternary verb with nominal and infinitival object (`beg`)

There are complementation rules for non-nominal objects, like

- `ComplVS : VS -> S  -> VP` (say that she runs)
- `ComplVQ : VQ -> QS -> VP` (wonder who runs)
- `ComplVV : VV -> VP -> VP` (want to run)

Complementation by NP and for ternary verb uses auxiliary categories

$$\begin{array}{rcll} \texttt{VP} & \simeq & \texttt{NP} \to \texttt{S} & \simeq \text{ unary predicate (sentence missing subject)} \\ \texttt{VPSlash} & \simeq & \texttt{NP} \to \texttt{VP} & \simeq \text{ binary predicate (VP missing nom.object)} \end{array}$$

Complementation by nominal object `ComplV2 = ComplSlash`:

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

Complementation for ternary verbs:

```
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
SlashV2Q : V2Q -> QS -> VPSlash ;  -- ask (him) who came
```

In summary: grammars of GF's resource grammar library have

- binary and ternary verbs distinguishing objects of nominal, sentential, interrogative and infinitival kind
- no such distinction for the subject argument of verbs.

In summary: grammars of GF's resource grammar library have

- binary and ternary verbs distinguishing objects of nominal, sentential, interrogative and infinitival kind
- no such distinction for the subject argument of verbs.

But of course, such a distinction (for verbs/adjectives) is necessary:

- sentential subject: `That this is the case, surprised us`
- interrogative subject: `What caused this is obvious`
- infinitival subject: `To go swimming may help you`

Passive constructions give predicates with non-nominal subject:

- `That the earth is flat was commonly believed`
- `How long the earth exists was not known`
- `To do your homework was often recommended to you`

A. Ranta's "Predication Grammar"

GF admits to declare syntactic categories as *dependent types*.

A. Ranta (EACL 2014) uses this to reimplement predication and complementation rules in terms of dependent categories in order to

- obtain schematic rules abstracting over complement frames
- fix anteriority, tense and polarity of predicates earlier than RGs

```
RGL:  VP.s : VFin Tense Ant Pol VAgr => Str
Pred: VP.s : VFin VAgr => Str
```

Sources: `gf/lib/src/experimental/Pred.gf`

Abstract Predication Grammar

- Verb category depending on types of arguments

```
cat Arg ;      -- argument type lists (HPSG subcat list)
    PrV Arg ; -- dependent verb category
```

- Construction of argument type lists:

```
fun aNone, aS, aV, aQ : Arg ; -- basic lists
    aNP : Arg -> Arg ;        -- list extension
```

RG verb categories correspond to dependent verb categories as

`V`	$\simeq$	`PrV aNone`	`V3`	$\simeq$	`PrV (aNP (aNP aNone))`
`V2`	$\simeq$	`PrV (aNP aNone)`	`V2S`	$\simeq$	`PrV (aNP aS)`
`VS`	$\simeq$	`PrV aS`	`V2V`	$\simeq$	`PrV (aNP aV)`

- Predicates, sentences, questions depending on arguments:

```
cat PrVP Arg ;    -- finite incomplete predicate
    PrVPI Arg ;   -- infinite incomplete predicate
    PrCl Arg ;    -- clause (incomplete sentence)
    PrQCl Arg ;   -- interr.clause (incomplete question)
```

(PrVP a) = predicates missing (object) arguments of type a:ARG:

$$\begin{array}{rcl@{\qquad}rcl} \texttt{PrVP aNone} & \simeq & \texttt{VP}, & \texttt{PrVP (aNP aS)} & \simeq & \texttt{NP} \to \texttt{S} \to \texttt{VP} \\ \texttt{PrVP (aNP aNone)} & \simeq & \texttt{VPSlash}, & \texttt{PrVP (aNP aQ)} & \simeq & \texttt{NP} \to \texttt{QS} \to \texttt{VP} \end{array}$$

Verbs of any type (PrV a) are predicates of that type (PrVP a):

```
fun UseV     : (a:Arg) -> Ant -> Tense -> Pol
                              -> PrV a       -> PrVP a ;
    PassUseV : (a:Arg) -> Ant -> Tense -> Pol
                              -> PrV (aNP a) -> PrVP a ;
```

Complementation rules now combine a (binary) predicate with a possibly incomplete object to a likewise incomplete predicate:

```
ComplV2 : (a:Arg) -> PrVP (aNP a) -> NP      -> PrVP a ;
ComplVS : (a:Arg) -> PrVP aS      -> PrCl  a -> PrVP a ;
ComplVV : (a:Arg) -> PrVP aV      -> PrVPI a -> PrVP a ;
ComplVQ : (a:Arg) -> PrVP aQ      -> PrQCl a -> PrVP a ;
```

Similarly for ternary predicates combined with second complement:

```
SlashV3  : (a:Arg) -> PrVP (aNP (aNP a))  -> NP
                                          -> PrVP (aNP a) ;
SlashV2S : (a:Arg) -> PrVP (aNP aS) -> PrCl a
                                          -> PrVP (aNP a) ;
SlashV2V : (a:Arg) -> PrVP (aNP aV) -> PrVPI a
                                          -> PrVP (aNP a) ;
SlashV2Q : (a:Arg) -> PrVP (aNP aA) -> PrQCl a
                                          -> PrVP (aNP a) ;
```

Concrete Predication Grammars

The concrete grammars `PredEng.gf`, `PredChi.gf`, etc. mostly share implementation types of syntactic categories.
Implementation type of category `(C a)` is *independent of* `a:Arg`.

Verb categories `(PrV a)` are implemented by the record type

```
lincat
  PrV = {
    s  : VForm => Str ;          -- verb paradigm
    p  : Str ;                   -- verb particle
    c1 : ComplCase ;             -- prep+case for 1st compl.
    c2 : ComplCase ;             -- prep+case for 2nd compl.
    isSubjectControl : Bool ;    -- subj.of embedded infiniti
    vtype : VType ;              -- auxiliary, reflexive etc.
    vvtype : VVType ;            -- pure|zu-infinitive compl.
    } ;
oper
  ComplCase : Type ; -- language specific, e.g. preposition
```

Verb phrases have parts of the verb paradigm and the verb's objects:

```
PrVP = {
    v : VAgr => Str * Str * Str ;   -- would,have,slept
    inf : VVType => Str ;           -- ((to) sleep | sleeping
    imp : ImpType => Str ;
    c1 : ComplCase ;
    c2 : ComplCase ;
    part  : Str ;                   -- verb part.: (look) up
    adj   : Agr => Str ;            -- predicative adjective
    obj1  : (Agr => Str) * Agr ;    -- agr for object control
    obj2  : (Agr => Str) * Bool ;   -- subject control = True
    vvtype : VVType ;               -- type of infinitive com
    adv : Str ;                     -- adverbial
    adV : Str ;                     -- negation adverb
    ext : Str ;                     -- right-extracted parts
    } ;
```

Clauses have less (but more informed) fields, plus a subject:

```
PrCl = {
  v : Str * Str * Str ;
  adj,obj1,obj2 : Str ;
  adv : Str ;
  adV : Str ;
  ext : Str ;
  subj : Str ;        -- subject
  } ;

PrQCl = PrCl ** {
  foc : Str ;         -- focus: *who* does she love
  focType : FocusType ;  -- if foc is filled, inplace:
                         -- who loves *who* } ;
```

Notice: word order of clauses is not completely fixed

Implementation functions of syntactic constructions fill these slots with suitable combinations of slots of argument records.

For example, `UseV` selects active verb forms depending on given values `a:Ant`, `t:Tense`, `p:Pol` to fill slots of the `PrVP` type:

```
UseV x a t p v = {
  v    = \\agr => tenseV t a p active agr v ;
  inf  = \\vt => tenseInfV a p active v vt ;
  imp  = \\it => imperativeV p it v ;
  c1   = v.c1 ; c2   = v.c2 ;
  part = v.p ;
  obj1 = <case isRefl v of {True => \\a => reflPron a ;
                                _ => \\_ => []},
                  defaultAgr> ;
  obj2 = <noObj, v.isSubjectControl> ;
  vvtype = v.vvtype ;
  adV = negAdV p ;
  adv, ext = [] } ;
```

The dependent categories and grammar rules for predication make a partial grammar

```
Pred = Cat[Ant,NP,..] + (Pr-categories and constructions)
```

It has to be completed by

- constructions of the RGL independent of predication:
  `UseN : N -> CN`,
  `DetCN : Det -> CN -> NP`, etc.
- lifting the verb categories of the RGL to Pr-categories of Pred:
  `LiftV : V -> PrV aNone`,
  `LiftV2S : V2S -> PrV (aNP aS)`, etc.

The extended grammar

```
Lift = Pred + RGLBase + (Lift* : Cat -> dep.Pr-cats)
```

can be equipped with the example lexicon of the RGL:

```
Test = Lift + Lexicon + Structural
```

Generation and parsing with dependent categories

One can generate abstract trees of given category, for example

Complete clause:

```
Test> generate_random -tr -cat="PrCl aNone" | linearize
PredVP aNone (DetNP many_Det)
       (AgentPassUseV aNone AAnter TFut PPos
                      (LiftV2 understand_V2) something_NP)

many will have been understood by something
```

Incomplete clause:

```
Test> generate_random -tr -cat="PrCl aS" | linearize
PredVP aS (UsePN john_PN) (UseV aS ASimul TFut PNeg
                                   (LiftVS say_VS))
John will not say
```

Dependent types are not fully integrated into the parser of GF, but checked by post-processing. This slows down parsing.

```
Test> parse -cat=PrS "John hopes that the bird flies"

UseCl (PredVP aNone (UsePN john_PN)
         (ComplVS aNone (UseV aS ... (LiftVS hope_VS))
                  (PredVP aNone (DetCN .. (UseN bird_N))
                           (UseV aNone ... (LiftV fly_V)))))

Test> parse -cat=PrS "John hopes the bird"

The parsing is successful but the type checking failed:

  Couldn't match expected type PrCl (aNP ?1)
         against inferred type PrCl aS
  In the expression:
       PredVP aS (UsePN john_PN)
         (UseV aS ASimul TPres PPos (LiftVS hope_VS))
```

- The *advantage* of using dependent categories for predication is that syntactic constructions can be written at a higher level, abstracting from irrelevant parts of complement frames
- the *disadvantage* is that the GF-parser does not check the dependent arguments at parse time, but by post-processing constraint solving

To combine the advantages of dependent categories and parsing with non-dependent categories,

- translate the schematic rules using categories `C (a:Arg)` to instances with non-dependent categories `C_a`

See `gf/lib/src/experimental/NDPred.gf`, `NDTestEng.gf`

Extension by Sentential Subjects

Both the existing RGL and the Predication grammar

- have no verbs with sentential/interrogative/infinitival subject
- have `PassV2 : V2 -> VP`, but cannot passivize verbs of type `VS`, `VQ`, `VV`, `V2S`, `V2Q`, `V2V`.

To add sentential subjects, we have to

A introduce new categories of verbs, adjectives, predicates

B extend the lexicon by suitable new verbs and adjectives

C introduce new constructions (`VP`,`Cl` with sent.subject)

In order not to delay checking of the subject's type to the post-processing, we do not add a further subject-`Arg` as in

```
PrV aS a,   PrVP aS a,
```

but use new categories `(PrSV a)`, `(PrSVP a)` etc.

Abstract grammar SPred

A1 New (non-dep.) lexical categories with sentential subject:

```
cat
  -- bin.verb with sentential subject and NP,S,Q object
  SV2 ;  -- that S, surprises/enjoys/disappoints NP
  SVS ;  -- that S1 causes/proves/implies that S2
  SVQ ;  -- that S explains (why|when|where S2)
  SVV ;  -- that S must/seems/is able ((to) VP)

  -- unary adjective with S subject
  SA ;   -- that S is plausible/unlikely
  -- binary adjective with S subject and NP object
  SA2 ;  -- that S is good for NP

  -- 0-ary verbs (expletive subject)
  V0 ;   -- it rains
```

A2 Dependent categories for verbs/predicates with sent.subject

```
cat
  PrSV Arg ;    -- (that S) surprises (NP)
  PrSVP Arg ;   -- (that S) surprises us
  PrSVPI Arg ;  -- (that S) (must|seems|is able)
                            (to) surprise her

  PrSAP Arg ;   -- (that S) is plausible
```

We don't need a category of clauses with sentential subject: i.e. can use `PrCl` for clauses with `S`, `Q`, `V` subject.

A1' Lexical categories with interrog./infinitival subject

```
-- QV Arg ; -- omit, no verbs v:QV exist
QA Arg ;    -- (why S) (is) uncertain | unknown

-- bin.verb with infinitival subject and NP object:
VPV2 ; -- to VP pleases NP

-- adjective with infinitival subject:
VPA ;  -- to VP is healthy | difficult
```

A2' Categories for predicates with interrog./infinit. subject

```
PrQA Arg  ;  -- why S, is uncertain (to me)
PrQVP Arg ;  -- why S, was unknown | explained to me
PrVVP Arg ;  -- to VP is healthy | was suggested to me
```

Remark: To form predicative sentences with sentential subjects like

- `that S is just a belief`,
- `why S is the question`,
- `to VP was our hope`

we also need nouns with `S`, `Q`, `V` object, related to verbs `VS`,`VQ`,`VV`.

```
cat
  NS ; -- belief|fact|claim (that S)
  NQ ; -- question (why|when|where S)
  NV ; -- hope|wish|fear (to VP)
fun
  belief_NS : NS ;
  question_NQ : NQ ;
  hope_NV : NV ;
```

In contrast: `TestEng` has `PredVP` *(a belief)*^{NP} *is (that she sleeps)*^{VP}

B Lexicon: verbs and adjectives with sentential subject

```
fun
  surprise_SV2 : SV2 ;
  cause_SVS : SVS ;
  explain_SVQ : SVQ ;

  plausible_SA : SA ;
  unlikely_SA : SA ;

  good_SA2 : SA2 ;

  rain_V0 : V0 ;        -- for 0-ary predication
  explain_V2S : V2S ;   -- for passive: V2S -> SV2
  explain_V2Q : V2Q ;   --               V2Q -> QV2
```

We lift these independent lexical categories to dependent ones:

```
fun
  LiftOV   : VO   -> PrOV ;

  LiftSV2 : SV2 -> PrSV (aNP aNone) ;
  LiftSVS : SVS -> PrSV aS ;
  LiftSVQ : SVQ -> PrSV aQ ;
  LiftSVV : SVV -> PrSV aV ;

  LiftSA   : SA   -> PrSAP aNone        ;
  LiftSA2 : SA2 -> PrSAP (aNP aNone) ;

  LiftV2Q : V2Q -> PrV (aNP aQ) ; -- from Lift* for Pred
  LiftV2S : V2S -> PrV (aNP aS) ;
```

New constructions with sentential subject

C1 Predicates with sent.subject `(PrSVP a)` can be built

(i) from a verb of category `SV2`, `SVS`, `SVQ` $\simeq$ `PrSV aNone|aS|aQ`

```
fun
  UseSV : (a : Arg) -> Ant -> Tense -> Pol ->
                       PrSV a       -> PrSVP a ;
```

(ii) by passivizing a verb of category `VS`, `V2S` $\simeq$ `PrV aS|aNp aS`:

```
  PassUseVS  :  Ant -> Tense -> Pol ->
                       PrV aS       -> PrSVP aNone ;
  PassUseV2S :  Ant -> Tense -> Pol ->
                       PrV (aNP aS) -> PrSVP (aNP aNone)
```

and likewise `AgentPassUseVS` and `AgentPassUseVS2` for passives with an additional agent-`NP`.

(iii) by complementation of binary predicates with a suitable object:

```
ComplSV2  : (a : Arg)
            -> PrSVP (aNP a) -> NP  -> PrSVP a ;
ComplSVS  : (a : Arg)
            -> PrSVP aS -> PrCl a   -> PrSVP a ;
ComplSVQ  : (a : Arg)
            -> PrSVP aQ -> PrQCl a  -> PrSVP a ;
ComplSVV  : (a : Arg)
            -> PrSVP aV -> PrSVPI a -> PrSVP a ;
```

(iv) by complementation of ternary predicates with suitable object:

```
-- SlashSV2: (whom) that S surprises _
--           (whom) it surprises _ that S
-- SlashSV3: (whom) that S was explained to _ by me
--           (whom) it was explained to _ by me that S
```

(iv) from an adjective phrase with sentential subject:

```
UseSAP : (a : Arg) -> Ant -> Tense -> Pol
                   -> PrSAP a -> PrSVP a ;
```

C2 Clause formation by predication with predicates `PrSVP a`

```
PredSVP : (a : Arg) -> Place
          -> PrCl aNone -> PrSVP a -> PrCl a ;
```

Subject sentences can be positioned either in place or moved:

- *that a bird flies* is plausible
- *it* is plausible *that a bird flies*

The difference is marked by empty dummy constituents

```
fun InPlace, Moved : Place ;
```

C1' Analogous constructions are (partly) possible and implemented for predication with interrogative subjects:

```
-- UseQV is not needed: there are no verbs v : PrQV a
UseQAP     : (a : Arg) -> Ant -> Tense -> Pol
                       -> PrQAP a -> PrQVP a ;
PassUseVQ  :              Ant -> Tense -> Pol
                       -> PrV aQ -> PrQVP aNone ;
ComplQV2   : (a : Arg)
             -> PrQVP (aNP a) -> NP -> PrQVP a ;
```

C2' with corresponding predication rule:

```
PredQVP : (a : Arg) -> Place
          -> PrQCl aNone -> PrQVP a -> PrCl a ;
```

Likewise for predicates and predication with infinitival subject.

Concrete grammar SPredEng

For English, the implementation of the concrete grammar was easy:

- the predication grammar `PredEng` provides constructions for predicates with *nominal* subject, and these can be reused:

```
lin
  UseSV = UseV ;
  PassUseVS = PassUseV aNone ;
  PassUseV2S = PassUseV aNone ; -- (!) V2S -> SVP
  ComplSV2 = ComplV2 ; ComplSVQ = ComplVQ ; ...
  UseSAP = UseAP ;
```

  This works since `lincat V,VP` don't care about the subject.

- exception: in the predication rule `PredSVP`, the subject may occupy its standard place or be moved to the right.

- exception: in the predication rule `PredSVP`, the subject may occupy its standard place or be moved to the right.

```
PredSVP _ p s vp =
  let subj = that_Compl ++ p.s ++ declSubordCl s ;
       agr = defaultAgr ; vagr = defaultVAgr
  in
  case p.moved of {
    False => predVP subj agr vagr vp ;
    True  => predVP "it" agr vagr (vp ** {ext = subj})
    } ;
```

where `predVP` fills the fields of the resulting `PrCl`-record:

```
predVP : Str -> Agr -> VAgr -> PrVP -> PrCl =
\s,agr,vagr,vp ->
  { subj = s ; ... ; ext = vp.ext } ;
```

Concrete grammar SPredGer

Since SPred is an extension of Pred by new categories and constructions, we first need a concrete grammar PredGer.

This is facilitated by a functor

```
PredFunctor : PredInterface -> Pred
```

which provides a *partial* concrete Pred-grammar in terms of types and operations declared (or even implemented) in PredInterface.

So we get PredGer by applying the functor

```
concrete PredGer of Pred =
  PredFunctor - [PredVP, ...] -- omit what to override
  with (PredInterface = PredInstanceGer)
in { lin PredVP np vp = ... } -- implement missing parts
```

to an implementation PredInstanceGer of PredInterface.

`PredInterface` contains declarations of parameter types and defaults that may need a language-specific implementation, like

```
Gender : PType ;
Agr : PType ;          -- full agreement, inherent in NP
NPCase : PType ;       -- full case of NP

subjCase : NPCase ;
ComplCase : Type ;  -- e.g. preposition
```

but also language-independent implementations of categories, like

```
NounPhrase : Type = {s : NPCase => Str ; a : Agr} ;

PrV : Type = {
  s  : VForm => Str ;
  c1 : ComplCase ; c2 : ComplCase ;
  ... } ;
```

For German, the main changes we had to make were

- add a field for the perfect-auxiliary to `PrV`
- add a field `c0:ComplCase` in `PrVP` for non-nom. subjects
- replace `AAgr` and `Agr` by a simpler `ObjAgr` in `PrVP`
- use `ComplCase = Prep` in `PrV` and `PrVP`

The last two items were needed for complexity reasons:

i) the GF-compiler instantiates the parameters in a record type by all possible values; only $8 = |\texttt{ObjAgr}|$ of the $18 = |\texttt{AAgr}| = |\texttt{Agr}| = |\texttt{Gender * Number * Person}|$ values are needed

ii) prepositional complements of verbs/adjectives are common, but: we must simplify RGL's category

```
Preposition = {s : Str ; c : PCase ; isPrep : Bool} ;
```

to `Prep` by excluding 6 glued versions `Det+Prep` like `AnDat` from `PCase`.

The implementation category of (PrVP a) then is

```
PrVP = {
  v : VAgr => Str * Str * Str ;  -- würde,geschlafen,haben
  inf : VVType => Str ;          -- (zu) schlafen
  imp : ImpType => Str ;
  c0 : Prep ;       -- subject case in passive of prepV2's
  c1 : ComplCase ; -- = Prep (in Eng: = Str)
  c2 : ComplCase ;
  part  : Str ;
  adj   : AAgr => Str ;
  obj1  : (ObjAgr => Str) * ObjAgr ; -- agr for control
  obj2  : (ObjAgr => Str) * Bool ;
  vvtype : VVType ;
  adv : Str ;
  adV : Str ;
  ext : Str ;
  } ;
```

Without these reductions of `Agr` and `Preposition`, it was impossible to compile the grammar using the `PredFunctor`

```
-- these need 7,5G memory or kill gf, without them 2,7G
-- ComplVA  15116544 (46656,1) Eng: 60 (60,1)
-- ComplVN  15116544 (46656,1)      0,4G memory
-- SlashV2A 15116544 (46656,1) Fin: 7375872 (37632,1)
-- SlashV2N 15116544 (46656,1)      5,6G memory
```

Their types are the most complex types of the `Pred`-functions: they have two argument types `PrVP` and `PrAP/PrCN`, each with two `ComplCase` slots and two `ObjAgr => Str` slots:

```
ComplVA/V2A  : (a : Arg) -> PrVP aA/(aNP aA)
                         -> PrAP a -> PrVP a/(aNP aA) ;
SlashV2A/V2N : (a : Arg) -> PrVP (aNP aA/aN)
                         -> PrAP/PrCN a -> PrVP (aNP a) ;
```

The syntactic constructions like

```
UseV : (a : Arg) -> ... -> PrV a -> PrVP a ;
```

are implemented in `PredFunctor` using auxiliary functions like `tenseV` that fill the field `v : VAgr => Str * Str * Str` of `PrVP` by selecting from the verb paradigm `s : VForm => Str` in `PrV`
Such auxiliary functions had to provided for German.

To allow for (non-nominative) subjects, as in

wir helfen euch^{2,Pl,Dat} $\mapsto$ euch^{2,Pl,Dat} wird^{3,Sg} geholfen
wir denken an euch $\mapsto$ an euch wird^{3,Sg} gedacht

an override of `PassUseV : ... -> PrV (aNP a) -> PrVP a` was needed to put the value in `c1:ComplCase` of the argument verb into into the `c0:Prep` field of the resulting `PrVP`.

So far for `PredGer`.

The new constructions of `SPred` with sentential subject can then be implemented mainly as for English, using those of `PredGer`:

```
lin
  UseSV = UseV ;
  ComplSV2 = ComplV2 ; ComplSVQ = ComplVQ ; ...
  UseSAP = UseAP ;
```

An exception is `PassUseV2S`, which has to fill the subject-field `c0:Prep`

```
  PassUseV2S a t p v = {
    v   = \\agr => tenseV t a p passive agr v ;
    c0 = case <v.c2.isPrep, v.c2.c> of { -- subj <- obj2
      <False, R.Acc> => subjPrep ;       -- i.e. nominative
      _              => v.c2 } ;
    ...
    } ;
```

Conclusion

- We have implemented an extension `SPred` of `Pred` to handle predication with sentential (and interrogative, infinitival) subjects for languages `Eng` and `Ger`
- Missing parts are relative clauses, clause coordination
- Much of `Pred` can be reused in `SPred`, the filtering of unwanted combinations depends on the *abstract* types only
- A complexity issue arises for `PredGer` from instantiating parameters to all possible values in the implementation records of `PrV`, `PrVP`
- Parsing is too slow

Lessons learned:

- naming convention of the RGL (for categories/constructions) relies on *few* verb classes, not suited to indicate subject types
- better integration of dependent categories in the parser would permit to
  1. combine different passive rules
     `PassV2 : V2 -> VP`, `PassVS : VS -> SVP` etc. by abstracting over the subject and object types:
     `PassV : V x (y a) -> VP y a`
  2. combine different `VP`-modifiers
     `NP must_VV VP`, `S must_VSV SVPI` etc. by abstracting over the subject type
     `must_VV : VPI x y -> VP x y`

Demo-Howto

1. parse examples:

   ```
   > i -v STestEng.gf
   STest> p -cat=Utt "it surprised John that it rained"
   ```

2. `STest> parse and visualize analysis trees (typed terms)`

   ```
   > i -v STestEng.gf
   STest> p -cat=Utt "it surprised him that it rained"
            | wf | rf -tree -lines | vt
            | ? dot -Tps -otrees.tmp.ps | gv trees.tmp.ps
   ```

   Likewise for parse trees with `vp` instead of `vt`

3. Testfiles:
   - `examplesEng.txt`
   - `schemata.out` generated via
     `shell> make -fMakefileS example-schemata`