Grammatical Framework

Hauptseminar SS 2007
CIS, Universität München

Hans Leiß, Martin Hofmann

18. Juli 2007

Aufgabe 1 Schreibe eine GF-Grammatik `Ap0`, die folgende Ausdrücke erkennt:

E1	`the big mistake`
D1	`der große Fehler`
E2	`this mistake is an error`
D2	`dieser Fehler ist ein Irrtum`
E3	`this mistake is big`
D3	`dieser Fehler ist groß`

Verwende als Kategorien: Artikel `Art`, Adjektive `A`, Nomen `CN`, Kopulaverb `CopV`, Adjektivphrasen `AP`, attributiv erweiterte Nomen `N1`, Nominalphrasen `NP` und Sätze `S`.

Benutze `This` und `Det` als Konstruktor von Nominalphrasen, `PredNP`, `PredAP` als Konstruktor von Sätzen durch die prädikative Verwendung einer Nominal- bzw. Adjektivphrase, `AttrAP` als Name für die attributive[1] Verwendung von Adjektivphrasen.

Lösung 1 Wir müssen drei Module schreiben:

(a) Die abstrakte Syntax mit den Kategoriendeklarationenen und Konstruktoren für die Phrasenkategorien `NP`, `S`, `AP`, `N1` und die lexikalischen Kategorien `CN`, `A`, `Art`, `CopV`:

⟨*Beispiele/Ap0.gf*⟩≡

```
abstract Ap0 = {
  cat
    Art ; A ; CN ; CopV ; N1 ; NP ; AP ; S ;

  fun
    This : CN -> NP ;                -- this CN
    Det : Art -> N1 -> NP ;          -- the AP CN | a AP CN
```

[1] ggf. `AttrS` für attributive Verwendung von Sätzen, d.h. als Relativsätze

```
    PredNP : NP -> CopV -> NP -> S ; -- NP1 is NP2
    PredAP : NP -> CopV -> AP -> S ; -- NP  is AP

    AttrAP : AP -> CN -> N1 ;        -- AP CN

    UseA   : A -> AP ;               -- A

    Mistake, Error : CN ;            -- Lexicon:
    Big : A ;
    Is : CopV ;
    The, An : Art ;
}
```

Diese Grammatik erlaubt die (abstrakten Versionen der) Ausdrücke `the error` und `this big error` nicht. Um sie zuzulassen, könnte man die `CN` mit einer leeren Adjektivphrase `APempty:AP` durch `AttrAP APempty CN` formal zu `N1`-Ausdrücken machen und das Demonstrativpronomen durch `This:Art` wie einen Artikel behandeln.

Allerdings könnte die `APempty:AP` nicht nur attributiv verwendet werden, wie in `the error`, sondern auch prädikativ, wie in `this error is` , was ungrammatisch ist. Um das zu vermeiden, sollte man statt `APempty:AP` eine Einbettung `AttrAPempty : CN -> N1` erlauben:

⟨*Beispiele/Ap0b.gf*⟩≡

```
abstract Ap0b = {
  cat
    Art ; A ; CN ; CopV ; N1 ; NP ; AP ; S ;

  fun
    Det : Art -> N1 -> NP ;          -- the N1 | a N1 | this N1

    PredNP : NP -> CopV -> NP -> S ; -- NP1 is NP2
    PredAP : NP -> CopV -> AP -> S ; -- NP  is AP

    AttrAP : AP -> CN -> N1 ;        -- AP CN
    AttrAPempty : CN -> N1 ;         -- CN

    UseA   : A -> AP ;               -- A

    Mistake, Error : CN ;            -- Lexicon:
    Big : A ;
    Is : CopV ;
    The, An, This : Art ;
}
```

(b) Die konkrete Syntax für das Englische mit einfachen Linearisierungskategorien:

⟨*Beispiele/Ap0Eng.gf*⟩≡

```
concrete Ap0Eng of Ap0 = {
lincat
   Art, A, CN, CopV, N1, NP, AP, S = {s : Str} ;
lin
   This cn      = {s = "this" ++ cn.s} ;
   Det art n1   = {s = art.s ++ n1.s} ;

   PredNP np1 cop np2 = {s = np1.s ++ cop.s ++ np2.s} ;
   PredAP np cop ap = {s = np.s ++ cop.s ++ ap.s} ;

   AttrAP ap cn  = {s = ap.s ++ cn.s} ;
   UseA a         = {s = a.s} ;

   Mistake      = {s = "mistake"} ;
   Error        = {s = "error"} ;
   Big          = {s = "big"} ;
   Is           = {s = "is"} ;
   The          = {s = "the"} ;
   An           = {s = "a"} ;
}
```

In der Reihenfolge, in der die Konstruktoren unter `fun` ihre Argumente erhalten, verketten die Linearisierungsfunktionen unter `lin` deren Linearisierungen, die Zeichenreihen ihrer `s`-Felder. Diese konkrete Grammatik für das Englische ist daher bis auf die Schreibweise eine kontextfreie Grammatik.

Die Unterschiede sind: eine kontextfreien Grammatikregel wie `S -> NP CopV AP` entspricht hier (1) einem Konstruktor `PredAP : NP -> CopV -> AP -> S`, mit dem abstrakte syntaktische Objekte der Kategorie `S` aufgebaut werden, und (2) einer Linearisierungsfunktion `PredAP^o : NP^o -> CopV^o -> AP^o -> S^o`, für die der Grammatikübersetzer Argument- und Ergebnistypen berechnet, die mit den zu den Kategorien erklärten Linearisierungstypen übereinstimmen müssen.

(c) Die konkrete Syntax für das Deutsche mit etwas anderen Linearisierungskategorien: für `A` und `AP` nehmen wir String-Tabellen oder vom Gebrauch abhängende Zeichenreihen, da bei der attributiven Verwendung eine Endung an das Adjektiv angehängt wird, bei der prädikativen Verwendung nicht. (Weil ich nicht weiß, wie man GF dazu bringt, ein `ß` zu lesen, wird im Folgenden `ss` statt `ß` benutzt.)

⟨*Beispiele/Ap0Ger.gf*⟩≡

```
concrete Ap0Ger of Ap0 = {
param
```

```
      Use = attr | pred ;
   lincat
      Art, CN, CopV, N1, NP, S = {s : Str} ;
      A, AP = {s : Use => Str} ;
   lin
      This cn      = {s = "dieser" ++ cn.s} ;
      Det art n1   = {s = art.s ++ n1.s} ;

      PredNP np1 cop np2 = {s = np1.s ++ cop.s ++ np2.s} ;
      PredAP np cop ap = {s = np.s ++ cop.s ++ ap.s!pred} ;

      AttrAP ap cn  = {s = ap.s!attr ++ cn.s} ;
      UseA a        = {s = a.s} ;

      Mistake      = {s = "Fehler"} ;
      Error        = {s = "Irrtum"} ;
      Big          = {s = table{attr => "grosser" ; pred => "gross"}} ;
      Is           = {s = "ist"} ;
      The          = {s = "der"} ;
      An           = {s = "ein"} ;
   }
```

⟨*Laden und Übersetzen*⟩≡

```
> i Beispiele/Ap0Ger.gf
+ reading Beispiele/Ap0.gfc 201 msec
- compiling Beispiele/Ap0Ger.gf...   wrote file Beispiele/Ap0Ger.gfc 211 msec
```

Die Behandlung der Adjektive im Deutschen geht darüber hinaus, hat aber noch Fehler.

⟨*Fehler: die Form des attributiven Adjektivs hängt nicht von Genus und Artikel ab*⟩≡

```
> gr -number=3 | l
dieser Irrtum ist gross
der grosser Irrtum ist gross
ein grosser Fehler ist der grosser Fehler
```

Aufgabe 2 Modifiziere und erweitere die Grammatik `Ap0` zu `Ap1`, damit auch folgende Vergleichsausdrücke erkannt werden:

E1	`this mistake is bigger than this mistake`
D1	`dieser Fehler ist grösser als dieser Fehler`
E2	`a bigger mistake than this mistake`,
D2	`ein grösserer Fehler als dieser Fehler`
E3	`this mistake is as big as this error`
D3	`dieser Fehler ist so gross wie dieser Irrtum`
E2	`a as big a mistake as this error`,
D2	`ein so grosser Fehler wie dieser Irrtum`

Es geht also einerseits darum, daß man Adjektivphrasen als Konstituenten betrachtet, die im attributiven Gebrauch unzusammenhängend, im prädikativen Gebrauch zusammenhängend sind. Andererseits geht es darum, daß man Adjektive von einer Komparationsstufe abhängig machen soll.

Für das Deutsche ignoriere man vorerst die Abhängigkeit des Adjektivs vom Genus und der Adjektivdeklination. (Das kommt demnächst durch Morphologieprogramme.)

a) Linearisiere eine `AP` als gespaltenen String (als Record mit zwei String-Feldern), stelle bei der attributiven Verwendung das Nomen zwischen die Teile, und bei der prädikativen Verwendung verkette die Teile.

b) Mache das Adjektiv `A` von einer Komparationsstufe abhängig, einem Parameterbereich `Comp` mit den Werten `pos`, `cmp`, `sup`.

Sollte man auch die Adjektivphrasen von einer Komparationsstufe abhängig machen, und wie wird diese Stufe berücksichtigt?

Lösung 2 Man kann die abstrakten `AP`-Ausdrücke so anlegen, daß erst bei ihrer Verwendung nach einer Komparationsstufe unterschieden wird. Dazu spaltet man `PredAP` und `AttrAP` in verschiedene Konstruktoren auf, für jede Komparationsstufe einen. (Das scheint mir eine bessere Abstraktion zu sein, als die `AP` in drei Unterklassen aufzuteilen.)

Für a) brauchen wir in der abstrakten Syntax neue Konstruktoren, die ein Adjektiv (unabhängig von Komparationsstufen) mit einer Vergleichsnominalphrase zu einer Adjektivphrase verbinden:

⟨*Beispiele/Ap1.gf*⟩≡

```
abstract Ap1 = {
  cat
    Art ; A ; CN ; CopV ; N1 ; NP ; AP ; S ;

  fun
    This : CN -> NP ;                     -- this CN
    Det : Art -> N1 -> NP ;               -- the AP CN | a AP CN
```

```
    PredNP : NP -> CopV -> NP -> S ;     -- NP1 is NP2
    PredAPpos : NP -> CopV -> AP -> S ; -- NP  is AP.pos
    PredAPcmp : NP -> CopV -> AP -> S ; -- NP  is AP.cmp
    PredAPsup : NP -> CopV -> AP -> S ; -- NP  is AP.sup

    AttrAPpos : AP -> CN -> N1 ;        -- AP.pos CN
    AttrAPcmp : AP -> CN -> N1 ;        -- AP.cmp CN
    AttrAPsup : AP -> CN -> N1 ;        -- AP.sup CN

    UseA   : A -> AP ;                  -- A
    CompA  : A -> NP -> AP ;            -- as A.pos as NP | A.cmp than NP
                                        -- | A.sup of all NP.pl

    Mistake, Error : CN ;               -- Lexicon:
    Big : A ;
    Is : CopV ;
    The, An : Art ;
}
```

Mit `UseA` ist der Gebrauch der Adjektive als einstellige Prädikate gemeint, mit `CompA` der Gebrauch als zweistellige Prädikate in Vergleichen, bei denen das Vergleichsobjekt schon bekannt ist.

Für b), also in der konkreten Syntax, soll das Adjektiv `A` von der Komparationsstufe abhängen, weshalb das Feld den Typ `Comp => Str` statt `Str` erhält. Die Kategorie `AP` soll Ausdrücke enthalten, die ebenfalls von der Komparationsstufe abhängen, aber i.a. *unzusammenhängende* Zeichenreihen sind (engl.:*discontinuous constituents*). Statt einem `Str` nehmen wir ein Paar `{s1:Str ; s2:Str}` von Zeichenreihen, zwischen dessen Teile bei der attributiven Verwendung das Bezugsnomen eingefügt wird; bei der prädikativen Verwendung werden die Teile miteinander verkettet. Die Linearisierungstypen sind also bei `A` und `AP` von einander und von denen der anderen Kategorien verschieden.

Problem: Bei der attributiven Verwendung einer `AP` muß man wissen, ob sie zusammenhängend (ein einfaches Adjektiv) oder gespalten (eine Vergleichskonstruktion) ist, weil die Linearisierung nicht bei `big mistake = AP.s1 ++ CN`, sondern nur bei `as big a mistake as NP = AP!pos.s1 ++ a ++ CN ++ AP!pos.s2` einen Artikel `a` einschieben darf. Man kann in den Linearisierungsfunktionen die Argumente inspizieren, aber der

⟨*Test auf eine leere s2-Komponente*⟩≡
```
   AttrAPpos ap cn =
     {s = (ap.s!pos).s1 ++ (case (ap.s!pos).s2 of {"" => "" ; _ => "a"})
                        ++ cn.s ++ (ap.s!pos).s2 } ;
```

führt zu einem Fehler (anscheinend ist `""` kein Token, über das man eine Fallunterscheidung mit `case` machen kann). Deshalb führen wir bei den `AP` noch ein Feld `c : Cont` ein, in dem der Unterschied kodiert wird, sodaß man ihn später dort ablesen kann.

⟨*Beispiele/Ap1Eng.gf*⟩≡

```
concrete Ap1Eng of Ap1 = in {
param
   Comp = pos | cmp | sup ;
   Cont = simpl | split ;
lincat
   Art, CN, CopV, N1, NP, S = {s : Str} ;
   A  = {s : Comp => Str} ;
   AP = {s : Comp => {s1 : Str ; s2 : Str} ; c : Cont} ;
lin
   This cn      = {s = "this" ++ cn.s} ;
   Det art n1   = {s = art.s ++ n1.s} ;

   PredNP np1 cop np2 = {s = np1.s ++ cop.s ++ np2.s} ;
   PredAPpos np cop ap =
     {s = np.s ++ cop.s ++ (ap.s!pos).s1 ++ (ap.s!pos).s2} ;
   PredAPcmp np cop ap =
     {s = np.s ++ cop.s ++ (ap.s!cmp).s1 ++ (ap.s!cmp).s2} ;
   PredAPsup np cop ap =
     {s = np.s ++ cop.s ++ "the" ++ (ap.s!sup).s1 ++ (ap.s!sup).s2} ;

   AttrAPpos ap cn =
     {s = (ap.s!pos).s1 ++ (case ap.c of {simpl => "" ; split => "a"})
                         ++ cn.s ++ (ap.s!pos).s2 } ;
   AttrAPcmp ap cn = {s = (ap.s!cmp).s1 ++ cn.s ++ (ap.s!cmp).s2} ;
   AttrAPsup ap cn = {s = (ap.s!sup).s1 ++ cn.s ++ (ap.s!sup).s2} ;

   UseA a       = {s = table{deg => {s1 = a.s!deg ; s2 = ""}} ;
                     c = simpl} ;
   CompA a np   =
     {s = table{pos => {s1 = "as"++ (a.s!pos) ; s2 = "as" ++ np.s} ;
                cmp => {s1 = (a.s!cmp) ; s2 = "than" ++ np.s} ;
                sup => {s1 = (a.s!sup) ; s2 = "of_all" ++ np.s}} ;
      c = split} ;

   Mistake      = {s = "mistake"} ;
   Error        = {s = "error"} ;
   Big          =
     {s = table{pos => "big" ; cmp => "bigger" ; sup => "biggest"}} ;
   Is           = {s = "is"} ;
   The          = {s = "the"} ;
   An           = {s = "a"} ;
```

```
}
```

Hier ein paar Beispiele:

⟨*Nach Laden von Beispiele/Ap1Eng.gf*⟩≡

```
> gr -number=10 | l -lang=Ap1Eng
this error is this mistake
this mistake is a bigger mistake
a as big a error as the biggest error of_all the bigger mistake is big
this mistake is this error
a bigger mistake is the biggest of_all this mistake
a biggest error is bigger than the biggest error
this error is big
a bigger error is this error
a bigger mistake than a as big a mistake as a bigger error
   than this mistake is the biggest
a bigger error than this mistake is this mistake
```

Offenbar muß man noch einbauen, daß nach `of all` nur eine Nominalphrase im Plural verwendet werden darf.

Bei der deutschen Version sieht es grob wie folgt aus, wobei die Morphologie noch nachzutragen ist, d.h. die Form des attributiv verwendeten Adjektivs hängt noch nicht von Artikel, Numerus, Genus und Kasus des Bezugsnomens ab:

⟨*Beispiele/Ap1Ger.gf*⟩≡

```
concrete Ap1Ger of Ap1 = {
param
   Comp = pos | cmp | sup ;
lincat
   Art, CN, CopV, N1, NP, S = {s : Str} ;
   A  = {s : Comp => Str} ;
   AP = {s : Comp => {s1 : Str ; s2 : Str}} ;
lin
   This cn      = {s = "dieser" ++ cn.s} ;
   Det art n1   = {s = art.s ++ n1.s} ;

   PredNP np1 cop np2 = {s = np1.s ++ cop.s ++ np2.s} ;
   PredAPpos np cop ap =
     {s = np.s ++ cop.s ++ (ap.s!pos).s1 ++ (ap.s!pos).s2} ;
   PredAPcmp np cop ap =
     {s = np.s ++ cop.s ++ (ap.s!cmp).s1 ++ (ap.s!cmp).s2} ;
   PredAPsup np cop ap =
     {s = np.s ++ cop.s ++ "der" ++ (ap.s!sup).s1 ++ (ap.s!sup).s2} ;

   AttrAPpos ap cn = {s = (ap.s!pos).s1 ++ cn.s ++ (ap.s!pos).s2} ;
```

```
    AttrAPcmp ap cn = {s = (ap.s!cmp).s1 ++ cn.s ++ (ap.s!cmp).s2} ;
    AttrAPsup ap cn = {s = (ap.s!sup).s1 ++ cn.s ++ (ap.s!sup).s2} ;

    UseA a        = {s = table{pos => {s1 = a.s!pos ; s2 = ""} ;
                               cmp => {s1 = a.s!cmp ; s2 = ""} ;
                               sup => {s1 = a.s!sup ; s2 = ""}}} ;

    CompA a np     =
      {s = table{pos => {s1 = "ebenso"++ (a.s!pos) ; s2 = "wie" ++ np.s} ;
                 cmp => {s1 = (a.s!cmp) ; s2 = "als" ++ np.s} ;
                 sup => {s1 = (a.s!sup) ; s2 = "aller" ++ np.s}}} ;

    Mistake      = {s = "Fehler"} ;
    Error        = {s = "Irrtum"} ;
    Big          = {s = table{pos => "gross" ; cmp => "grösser" ; sup => "grösst"}} ;
    Is           = {s = "ist"} ;
    The          = {s = "der"} ;
    An           = {s = "ein"} ;
 }
```

Das ist noch zu verbessern (Übung!): z.Zt erhält man z.B.

⟨*Beispiel nach Laden von Beispiele/Ap1Ger.gf*⟩≡

```
> gr -number=10 | l
der grösst Irrtum aller dieser Irrtum ist der grösser Irrtum
dieser Irrtum ist grösser als dieser Fehler
dieser Fehler ist dieser Irrtum
ein grösst Irrtum aller dieser Irrtum ist der gross Fehler
dieser Fehler ist ebenso gross wie dieser Fehler
ein grösst Irrtum ist dieser Fehler
ein ebenso gross Fehler wie der grösser Fehler ist der grösst
  aller der grösser Fehler als dieser Irrtum
der gross Fehler ist grösser
dieser Fehler ist grösser
ein grösser Fehler als der ebenso gross Fehler wie dieser Fehler
  ist der grösst aller der grösser Fehler
```

Wie bei `Ap1Eng.gf` wird im Superlativ noch nicht verlangt, daß die Vergleichsnominalphrase im Plural ist. Außerdem sieht man, daß man nach `aller` keinen Artikel mehr zulassen darf, bzw. nur `dieser`, nicht `der`.

Aufgabe 3 Erweitere die konkreten Grammatiken um eine morphologische Komponente. Die Linearisierung einer `NP` hängt im Englischen vom Numerus ab, im Deutschen zusätzlich noch vom Kasus. (Wir betrachten nur Nominalphrasen der dritten Person, sodaß keine Abhängigkeit von der Person einzubauen ist.)

Man verwende einen Resource-Modul, mit dem man die morphologischen Varianten (bei der regulären Deklination) für alle Nomina und Adjektive uniform definieren kann.

(a) Was sind geeignete Linearisierungskategorien ?

(b) In welcher Reihenfolge erwartet man geschickterweise die Genus, Numerus, Kasus, Komparationswerte, nach denen sich die Form ändert?

(c) Wie sehen die Flexionsprogramme für das Deutsche genauer aus?

Lösung 3 Die Resource-Module sind natürlich sprachabhängig. Man braucht in ihnen schon Definitionen von Parameterbereichen, die daher in den sie ladenden konkreten Grammatiken entfallen können.

Erstmal bauen wir den Numerus in die Nominalphrasen ein. Wenn man die Nominalphrasen vom Numerus abhängig machen will, sollte der Linearisierungstyp etwa `{s : Number => Str}` sein. Dann muß man bei der Prädikation wählen, ob man die Subjektnominalphrase im Singular oder Plural formulieren will, also die `Pred`-Konstruktoren nach dem Numerus des Subjekts in `PredSg`, `PredPl` aufspalten. Aber da der Numerus bei verschiedenen Sprachen verschiedene Werte annehmen kann, kann diese Aufspaltung nicht in der abstrakten Syntax geschehen.[2]

Wenn man die Nominalphrase nicht vom Numerus abhängig macht, also als nicht im Numerus variierbar ansieht, sollte sie einen festen Numerus haben, der Linearisierungstyp alse etwa `{s : Str ; n : Number}` sein. Das ist für die Beispielgrammatik weniger plausibel, aber es gibt i.a. Nominalphrasen, die nicht im Numerus variieren, z.B. Eigennamen, aber auch gewisse `CN` erlauben keine Variation im Numerus, z.B. `das Gold`. Wenn man das macht, muß man den Numerus irgendwann beim Aufbau einer Nominalphrase festlegen, und damit verschiebt sich das Problem nur auf andere Konstruktoren, z.B. auf die Konstruktoren, die eine `N1` erzeugen.

Ich lege im Folgenden den Numerus bei der Bildung eines `N1`-Ausdrucks fest. Dafür muß die abstrakte Grammatik geändert werden, sodaß die Konstruktoren zur Bildung von `N1`-Ausdrücken nach dem Numerus aufgespalten werden; gleichzeitig verlagern wir `This` unter die Artikel, zu Gunsten einer Attributierung mit leerer Adjektivphrase.

⟨*Beispiele/Ap2.gf*⟩≡

[2] Man müßte einen Konstruktor der abstrakten Syntax in einer konkreten Grammatiken in mehrere (sprachspezifisch parametrisierte) Konstruktoren bzw. Linearisierungsfunktionen aufspalten; ob das mit abhängigen Typen geht, wäre zu überlegen (ggf. `Number` als Kategorie und `cat S Number` als davon abhängig?). Aber wie würde die Übersetzung von einer Sprache in eine andere die passenden parametrisierten Konstruktoren wählen?

```
abstract Ap2 = {
  cat
    Art ; A ; CN ; CopV ; N1 ; NP ; AP ; S ;

  fun
    Det : Art -> N1 -> NP ;              -- the N1 | a N1 | this N1

    PredNP : NP -> CopV -> NP -> S ;     -- NP1 is/are NP2
    PredAPpos : NP -> CopV -> AP -> S ;  -- NP  is/are AP.pos
    PredAPcmp : NP -> CopV -> AP -> S ;  -- NP  is/are AP.cmp
    PredAPsup : NP -> CopV -> AP -> S ;  -- NP  is/are AP.sup

    AttrAPemptySg : CN -> N1 ;           -- CN.sg
    AttrAPemptyPl : CN -> N1 ;           -- CN.pl

    AttrAPposSg : AP -> CN -> N1 ;       -- AP.pos CN.sg
    AttrAPcmpSg : AP -> CN -> N1 ;       -- AP.cmp CN.sg
    AttrAPsupSg : AP -> CN -> N1 ;       -- AP.sup CN.sg

    AttrAPposPl : AP -> CN -> N1 ;       -- AP.pos CN.pl
    AttrAPcmpPl : AP -> CN -> N1 ;       -- AP.cmp CN.pl
    AttrAPsupPl : AP -> CN -> N1 ;       -- AP.sup CN.pl

    UseA   : A -> AP ;                   -- A
    CompA  : A -> NP -> AP ;             -- as A.pos as NP | A.cmp than NP
                                         -- | A.sup of all NP.pl
    Mistake, Error : CN ;                -- Lexicon:
    Big : A ;
    Is : CopV ;
    The, An, This : Art ;                -- Def, Indef, Dem ?
}
```

Und nun zu den konkreten Grammatiken.

Englisch Für die Morphologie brauchen wir nur die Bildung des Plurals bei Nomina und der Komparationsstufe bei Adjektiven. Das machen wir in einer eigenen Datei, in der zwei Funktionen aus `GF-2.7/lib//prelude/Prelude.gfc` benutzt werden: `init : Str -> Str` liefert den Anfang außer dem letzten Buchstaben und `last : Str -> Str` den letzten Buchstaben einer Zeichenreihe:

⟨*Beispiele/DeclEng.gf*⟩≡

```
resource DeclEng = open Prelude in {
--# -path=.:prelude
-- Compiler-Instruktion zur Suche von $GF-Lib-Path/prelude/Prelude.gf
```

```
  param Number = sg | pl ;
  param Comp = pos | cmp | sup ;

  oper regCN : Str -> {s : Number => Str}
         = \x -> {s = table { sg => x ; pl => x + "s" }} ;

  oper regA : Str -> {s : Comp => Str} =
             \str -> let tab = case last str of
                          { "g" => table{ pos => str ;
                                          cmp => str+"g"+"er" ;
                                          sup => str+"g"+"est" } ;
                            "y" => table{ pos => str ;
                                          cmp => (init str) +"i"+"er" ;
                                          sup => (init str) +"i"+"est" } ;
                            _ => table{ pos => str ;
                                        cmp => str+"er" ;
                                        sup => str+"est" }
                          } in {s = tab} ;
}
```

Nun zur Grammatik. Die Resource-Datei `DeclEng.gf` wird benutzt, um die Paradigmen der Nomen und Adjektive zu bilden. Für das Englische variieren die `Art`, `CN` und die `CopV` im Numerus, die `N1`, `NP`, `A`, `AP` und `S` nicht.

⟨*Beispiele/Ap2Eng.gf*⟩≡

```
concrete Ap2Eng of Ap2 = open DeclEng in {
--# -path=.:prelude
param
   Cont = simpl | split ;  -- Comp, Number in DeclEng
lincat
   S = {s : Str} ;
   Art, CopV, CN = {s : Number => Str} ;
   N1, NP = {s : Str ; n : Number} ;
   A  = {s : Comp => Str} ;
   AP = {s : Comp => {s1 : Str ; s2 : Str} ; c : Cont} ;
lin
   Det art n1   = {s = art.s!(n1.n) ++ n1.s ; n = n1.n} ;

   PredNP np1 cop np2 = {s = np1.s ++ cop.s!(np1.n) ++ np2.s} ;

   PredAPpos np cop ap =
     {s = np.s ++ cop.s!(np.n) ++ (ap.s!pos).s1 ++ (ap.s!pos).s2} ;
   PredAPcmp np cop ap =
     {s = np.s ++ cop.s!(np.n) ++ (ap.s!cmp).s1 ++ (ap.s!cmp).s2} ;
   PredAPsup np cop ap =
```

```
    {s = np.s ++ cop.s!(np.n) ++ "the" ++ (ap.s!sup).s1 ++ (ap.s!sup).s2} ;

  AttrAPposSg ap cn =
    {s = (ap.s!pos).s1 ++ (case ap.c of {simpl => "" ; split => "a"})
                       ++ cn.s!sg ++ (ap.s!pos).s2 ;  n = sg} ;
  AttrAPcmpSg ap cn =
    {s = (ap.s!cmp).s1 ++ cn.s!sg ++ (ap.s!cmp).s2 ; n = sg} ;
  AttrAPsupSg ap cn =
    {s = (ap.s!sup).s1 ++ cn.s!sg ++ (ap.s!sup).s2 ; n = sg} ;

  AttrAPposPl ap cn =
    {s = (ap.s!pos).s1 ++ cn.s!pl ++ (ap.s!pos).s2 ; n = pl} ;
  AttrAPcmpPl ap cn =
    {s = (ap.s!cmp).s1 ++ cn.s!pl ++ (ap.s!cmp).s2 ; n = pl} ;
  AttrAPsupPl ap cn =
    {s = (ap.s!sup).s1 ++ cn.s!pl ++ (ap.s!sup).s2 ; n = pl} ;

  AttrAPemptySg cn = {s = cn.s!sg ; n = sg} ;
  AttrAPemptyPl cn = {s = cn.s!pl ; n = pl} ;

  UseA a       = {s = table{deg => {s1 = a.s!deg ; s2 = ""}} ;
                    c = simpl} ;
  CompA a np   =
    {s = table{pos => {s1 = "as"++ a.s!pos ; s2 = "as" ++ np.s} ;
               cmp => {s1 = a.s!cmp ; s2 = "than" ++ np.s} ;
               sup => {s1 = a.s!sup ;
                       s2 = (case np.n of {pl => "of" ++ np.s ;
                                           sg => variants {} })}} ;
     c = split} ;

  Mistake      = regCN "mistake" ;
  Error        = regCN "error" ;
  Big          = regA "big" ;
  Is           = {s = table {pl => "are" ; sg => "is" }} ;
  The          = {s = table Number { _ => "the"}} ;
  An           = {s = table {sg => "a" ; pl => ""}} ;
  This         = {s = table {sg => "this" ; pl => "these"}} ;
}
```

Bei Vergleichen im Superlativ mit einer Vergleichsnominalphrase im Singular wird mit Hilfe von `variants{}` eine Ausgabe unterdrückt und so die Linearisierung partiell gemacht:

⟨*Linearisierung mit Vergleichsglied im Plural ist erlaubt, im Singular nicht*⟩≡

```
> l -tr PredAPsup (Det This (AttrAPemptySg Mistake)) Is
```

```
                    (CompA Big (Det This (AttrAPemptyPl Error)))
this mistake is the biggest of these errors

> l -tr PredAPsup (Det This (AttrAPemptySg Mistake)) Is
                    (CompA Big (Det This (AttrAPemptySg Error)))
```

⟨*Beispiele mit Numerus*⟩≡

```
gr -number=10 | l
not completed
the biggest mistake of these biggest mistakes is big
these bigger errors than these bigger errors than these errors
  are this error
bigger errors than big errors are bigger than this biggest mistake
a bigger error is big
this big mistake is the biggest
this as big a mistake as this bigger mistake is the biggest
biggest mistakes of biggest mistakes are bigger than this
  as big a mistake as this big mistake
the bigger mistakes than the bigger mistake are the biggest
```

Fehler Auch wenn das Subjekt im Singular steht, wird eine Vergleichsnominalphrase im Plural akzeptiert:

⟨*Fehler im Numerus*⟩≡

```
> p "this error is mistakes"
PredNP (Det This (AttrAPemptySg Error)) Is (Det An (AttrAPemptyPl Mistake))
```

(Plural im Subjekt und Singular in der Vergleichsnominalphrase sollte erlaubt bleiben.)

Deutsch Zuerst zur Morphologie im Deutschen. Für die Nomina braucht man die Parameterbereiche Genus, Numerus und Kasus, wobei das Genus nicht variabel ist. Wir geben nur ein Beispiel einer reguläre Deklinationsklasse an (die für `Fehler`, aber nicht für `Irrtum` paßt und insbesondere keine Umlaute im Nomenstamm vornimmt).

Für die Adjektive braucht man die Komparation sowohl für den attributiven wie den prädikativen Gebrauch. Beim attributiven Gebrauch muß man die Formen nach Adjektivdeklination, Genus, Numerus und Kasus unterscheiden, wobei das Genus nur im Singular nötig ist; beim prädikativen Gebrauch gibt es keine morphologischen Varianten. Wir geben die reguläre Adjektivflexion an, aber ohne die Umlautbildungen im Komparativ und Superlativ.

⟨*Beispiele/DeclGer.gf*⟩≡

```
resource DeclGer = {

  param Number = sg | pl ;
  param Gender = mask | fem | neut ;
```

```
param Case   = nom | gen | dat | akk ;

oper regCN : Str -> Gender -> {s : Number => Case => Str ; g : Gender}
          = \x,y -> {s = table{ sg => table{gen => x + "s" ; _ => x} ;
                                pl => table{dat => x + "n" ; _ => x} } ;
                     g = y} ;
```

⟨*reg2N*⟩

```
param Comp = pos | cmp | sup ;
param Adekl  = st | sw | ge ;
param FormA  = sgA Adekl Gender Case | plA Adekl Case ;
param UseAP = attr FormA | pred ;

oper regA : Str -> {s : (Comp => UseAP => Str)} =
       \str -> {s = table{ pos => mkFormsA pos str  ;
                           cmp => mkFormsA cmp (str+"er")  ;
                           sup => mkFormsA sup (str+"st") }} ;

oper mkFormsA : Comp -> Str -> (UseAP => Str) =
      \c -> \str ->
              table{ pred => case c of {sup => str+"en" ;
                                        _   => str } ;
                     --  st = pronominal = Endungen wie bei dies-.
                     attr (sgA st mask nom) => str+"er" ;
                     attr (sgA st mask gen) => str+"es" ;
                     attr (sgA st mask dat) => str+"em" ;
                     attr (sgA st mask akk) => str+"en" ;
                     --
                     attr (sgA st fem nom) => str+"e"  ;
                     attr (sgA st fem gen) => str+"er" ;
                     attr (sgA st fem dat) => str+"er" ;
                     attr (sgA st fem akk) => str+"e"  ;
                     --
                     attr (sgA st neut nom) => str+"es" ;
                     attr (sgA st neut gen) => str+"es" ;
                     attr (sgA st neut dat) => str+"em" ;
                     attr (sgA st neut akk) => str+"es" ;
                     --  sw = nominal = nach bestimmtem Artikel 'der'
                     attr (sgA sw mask nom) => str+"e"  ;
                     attr (sgA sw mask gen) => str+"en" ;
                     attr (sgA sw mask dat) => str+"en" ;
                     attr (sgA sw mask akk) => str+"en" ;
                     --
                     attr (sgA sw fem nom) => str+"e"  ;
```

```
                    attr (sgA sw fem gen) => str+"en" ;
                    attr (sgA sw fem dat) => str+"en" ;
                    attr (sgA sw fem akk) => str+"e"  ;
                    --
                    attr (sgA sw neut nom) => str+"e"  ;
                    attr (sgA sw neut gen) => str+"en" ;
                    attr (sgA sw neut dat) => str+"en" ;
                    attr (sgA sw neut akk) => str+"e"  ;
                    --  ge = nach unbestimmtem Artikel 'ein'
                    attr (sgA ge mask nom) => str+"er" ;
                    attr (sgA ge mask gen) => str+"en" ;
                    attr (sgA ge mask dat) => str+"en" ;
                    attr (sgA ge mask akk) => str+"en" ;
                    --
                    attr (sgA ge fem nom) => str+"e"  ;
                    attr (sgA ge fem gen) => str+"en" ;
                    attr (sgA ge fem dat) => str+"en" ;
                    attr (sgA ge fem akk) => str+"e"  ;
                    --
                    attr (sgA ge neut nom) => str+"es"  ;
                    attr (sgA ge neut gen) => str+"en" ;
                    attr (sgA ge neut dat) => str+"en" ;
                    attr (sgA ge neut akk) => str+"es" ;
                    --
                    attr (plA sw _) => str+"en" ;
                    attr (plA _ nom) => str+"e" ;
                    attr (plA _ gen) => str+"er" ;
                    attr (plA _ dat) => str+"en" ;
                    attr (plA _ akk) => str+"e" } ;
 }
```

Nun zur Grammatik. Sie sollte für das Deutsche neben dem Plural auch erfassen, daß Artikel und attributiv verwendete Adjektive dem Genus (und dem bei der Attributierung festzulegenden Numerus) des Bezugsnomen angepaßt werden; außerdem muß die Endung des Adjektivs vom Artikel abhängen, und alles in jeder Komparationsstufe.

Nominalphrasen und damit auch attributiv verwendete Adjektivphrasen sollten im Kasus variieren können, da beim Subjekt der Nominativ und bei der Vergleichsnominalphrase der Genitiv gebraucht wird.

Die Unterscheidung, ob eine Adjektivphrase attributiv oder prädikativ (i.a.: oder adverbial) gebraucht wird, muß beibehalten werden, da nur im attributiven Gebrauch die Deklination nötig ist. Von den beiden String-Komponenten der Adjektivphrase machen wir die erste, die das Adjektiv enthält, vom Gebrauch abhängig, und die zweite, die ggf. ein Vergleichsnomen enthält, vom Kasus, da dieser beim attributiven Gebrauch an den Kasus des Bezugsnomens angepaßt wird; beim prädikativen Gebrauch ist der Nominativ zu nehmen.

⟨*Beispiele/Ap2Ger.gf*⟩≡

```
concrete Ap2Ger of Ap2 = open DeclGer in {
param
  Cont = simpl | split ;  -- Comp, Number, Case, UseA in DeclGer
lincat
  S = {s : Str} ;
  CopV = {s : Number => Str} ;
  CN = {s : Number => Case => Str ; g : Gender} ;
  N1 = {s : Adekl => Case => Str ; g : Gender ; n : Number} ;
  NP = {s : Case => Str ; g : Gender ; n : Number} ;
  Art = {s : Gender => Number => Case => Str ; d : Adekl } ;
  A  = {s : Comp => UseAP => Str} ;
  AP = {s : Comp => {s1 : UseAP => Str ; s2 : Case => Str} ; c : Cont} ;
lin
  Det art n1 =
    {s = table Case {c => art.s!(n1.g)!(n1.n)!c ++ (n1.s!(art.d))!c } ;
     g = n1.g ; n = n1.n} ;

  PredNP np1 cop np2 = {s = np1.s!nom ++ cop.s!(np1.n) ++ np2.s!nom} ;

  PredAPpos np cop ap =
    {s = np.s!nom ++ cop.s!(np.n) ++ (ap.s!pos).s1!pred ++ (ap.s!pos).s2!nom} ;
  PredAPcmp np cop ap =
    {s = np.s!nom ++ cop.s!(np.n) ++ (ap.s!cmp).s1!pred ++ (ap.s!cmp).s2!nom} ;
  PredAPsup np cop ap =
    {s = np.s!nom ++ cop.s!(np.n) ++ "am" ++ (ap.s!sup).s1!pred
                                               ++ (ap.s!sup).s2!nom} ;

  AttrAPposSg ap cn =
    {s = table{ad => table{c => (ap.s!pos).s1!(attr (sgA ad cn.g c))
                                ++ (cn.s!sg)!c ++ (ap.s!pos).s2!c}} ;
     g = cn.g ; n = sg} ;
  AttrAPcmpSg ap cn =
    {s = table{ad => table{c => (ap.s!cmp).s1!(attr (sgA ad cn.g c))
                                ++ (cn.s!sg)!c ++ (ap.s!cmp).s2!c}} ;
     g = cn.g ; n = sg} ;
  AttrAPsupSg ap cn =
    {s = table{ad => table{c => (ap.s!sup).s1!(attr (sgA ad cn.g c))
                                ++ (cn.s!sg)!c ++ (ap.s!sup).s2!c}} ;
     g = cn.g ; n = sg} ;
```

```
  AttrAPposPl ap cn =
    {s = table{ad => table{c => (ap.s!pos).s1!(attr (plA ad c))
                                ++ (cn.s!pl)!c ++ (ap.s!pos).s2!c}} ;
     g = cn.g ; n = pl} ;
  AttrAPcmpPl ap cn =
    {s = table{ad => table{c => (ap.s!cmp).s1!(attr (plA ad c))
                                ++ (cn.s!pl)!c ++ (ap.s!cmp).s2!c}} ;
     g = cn.g ; n = pl} ;
  AttrAPsupPl ap cn =
    {s = table{ad => table{c => (ap.s!sup).s1!(attr (plA ad c))
                                ++ (cn.s!pl)!c ++ (ap.s!sup).s2!c}} ;
     g = cn.g ; n = pl} ;

  AttrAPemptySg cn = {s = table{ad => cn.s!sg} ; g = cn.g ; n = sg} ;
  AttrAPemptyPl cn = {s = table{ad => cn.s!pl} ; g = cn.g ; n = pl} ;

  UseA a =
    {s = table{deg => {s1 = table{form => a.s!deg!form} ;
                       s2 = table{_ => ""}}} ;
     c = simpl} ;
  CompA a np   =
    {s = table{pos => {s1 = table{form => "so"++ (a.s!pos)!form} ;
                       s2 = table{c => "wie" ++ np.s!c}} ;
               cmp => {s1 = table{form => (a.s!cmp)!form} ;
                       s2 = table{c => "als" ++ np.s!c}} ;
               sup => {s1 = table{form => (a.s!sup)!form} ;
                       s2 = table{_ => (case np.n of {pl => "von" ++ np.s!dat ;
                                                      sg => variants {} })}}} ;
     c = split} ;

  Mistake      = regCN "Fehler" mask ;
--  Error        = regCN "Irrtum" mask ;
  Error        = reg2N "Irrtum" "Irrtümer" mask ;
  Big          = regA  "klein" ;
  Is           = {s = table {pl => "sind" ; sg => "ist" }} ;

  The  = {s = table {mask => table {sg => table {nom => "der" ;
                                                  gen => "des" ;
                                                  dat => "dem" ;
                                                  akk => "den" } ;
                                    pl => table {nom => "die" ;
                                                  gen => "der" ;
                                                  dat => "den" ;
                                                  akk => "die" }} ;
```

```
                   fem  => table {sg => table {nom => "die" ;
                                               gen => "der" ;
                                               dat => "der" ;
                                               akk => "die" } ;
                                  pl => table {nom => "die" ;
                                               gen => "der" ;
                                               dat => "den" ;
                                               akk => "die" }} ;
                   neut => table {sg => table {nom => "das" ;
                                               gen => "des" ;
                                               dat => "dem" ;
                                               akk => "das" } ;
                                  pl => table {nom => "die" ;
                                               gen => "der" ;
                                               dat => "den" ;
                                               akk => "die" }}} ;
        d = sw };
An   = {s = table {mask => table {sg => table {nom => "ein" ;
                                               gen => "eines" ;
                                               dat => "einem" ;
                                               akk => "einen" } ;
                                  pl => table {nom => "einige" ;
                                               gen => "einiger" ;
                                               dat => "einigen" ;
                                               akk => "einige" }} ;
                   fem  => table {sg => table {nom => "eine" ;
                                               gen => "einer" ;
                                               dat => "einer" ;
                                               akk => "eine" } ;
                                  pl => table {nom => "einige" ;
                                               gen => "einiger" ;
                                               dat => "einigen" ;
                                               akk => "einige" }} ;
                   neut => table {sg => table {nom => "ein" ;
                                               gen => "eines" ;
                                               dat => "einem" ;
                                               akk => "ein" } ;
                                  pl => table {nom => "einige" ;
                                               gen => "einiger" ;
                                               dat => "einigen" ;
                                               akk => "einige" }}} ;
        d = ge };
This = {s = table {mask => table {sg => table {nom => "dieser" ;
                                               gen => "dieses" ;
```

```
                                                dat => "diesem" ;
                                                akk => "diesen" } ;
                                 pl => table {nom => "diese" ;
                                                gen => "dieser" ;
                                                dat => "diesen" ;
                                                akk => "diese" }} ;
                      fem  => table {sg => table {nom => "diese" ;
                                                gen => "dieser" ;
                                                dat => "dieser" ;
                                                akk => "diese" } ;
                                 pl => table {nom => "diese" ;
                                                gen => "dieser" ;
                                                dat => "diesen" ;
                                                akk => "diese" }} ;
                      neut => table {sg => table {nom => "dieses" ;
                                                gen => "dieses" ;
                                                dat => "diesem" ;
                                                akk => "dieses" } ;
                                 pl => table {nom => "diese" ;
                                                gen => "dieser" ;
                                                dat => "diesen" ;
                                                akk => "diese" }}} ;
          d = sw };

}
```

Im Deutschen wird hier `Big` mit `klein` wiedergegeben, damit man die Umlaute bzw. die Verschmelzung der `s` im Komparativ bzw. Superlativ nicht programmieren muß.

⟨*Beispiele, deutsch und englisch*⟩≡

```
> gr -number=10 | l -multi
not completed

die kleinen Fehler sind kleiner
the big mistakes are bigger


einige so kleine Irrtum wie diese kleinsten Irrtum sind dieser Irrtum
as big errors as these biggest errors are this error


der kleinere Fehler als dieser kleinere Fehler als diese Fehler ist
   der so kleine Irrtum wie ein Fehler
the bigger mistake than this bigger mistake than these mistakes is
```

the as big a error as a mistake

die kleinen Fehler sind am kleinsten von diesen kleinsten Irrtumn
von diesen so kleinen Fehlern wie diesem Irrtum
the big mistakes are the biggest of these biggest errors
of these as big mistakes as this error

ein Fehler ist kleiner
a mistake is bigger

dieser Irrtum ist so klein wie dieser so kleine Fehler wie die kleineren Fehler
this error is as big as this as big a mistake as the bigger mistakes

der Fehler ist am kleinsten
the mistake is the biggest

Aufgabe 4 Zur Morphologie:

(a) Im Tutorium wird eine Operation `artIndef:Str` angegeben, die den unbestimmten Artikel `a` bzw. `an` des Englischen in Abhängigkeit davon ausgibt, was der erste Buchstabe des nachfolgenden Tokens ist.

Verbessere mit `artIndef` die konkrete Grammatik `Ap1Eng.gf` so, daß man `"an error"` als Linearisierung von `Det An Error` bekommt, aber `"a mistake"` als Linearisierung von `Det An Mistake`.

(b) Schreibe eine Deklination `reg2N`, die zur Nomendeklinationen aus einer Singular- und einer Pluralform (die den Umlaut schon enthält) das Paradigma aller Vollformen erzeugt, sodaß man z.B. im Plural `Irrtümer` statt `Irrtum` bekommt.

(c) Verbessere mit Hilfe eines regulären Ausdrucks die Nomendeklination im Deutschen so, daß der (erste?) Vokal unter `a`, `o`, `u` im Nomenstamm des Singulars bei den Pluralformen durch den entsprechenden Umlaut ä, ö, ü ersetzt wird. Analog für die Adjektivdeklination im Komparativ und Superlativ.

Lösung 4 b) Für `reg2N` geben wir zwei Formen und das Genus vor; die Definition wird in obige Datei `Beispiele/DeclGer.gf` eingefügt:

⟨*reg2N*⟩≡

```
oper reg2N : Str -> Str -> Gender -> {s : Number => Case => Str ; g : Gender} =
                     \x,y,Gen -> {s = table{sg => table{gen => x +"s" ; _ => x} ;
                                           pl => table{gen => y +"n" ; _ => y}} ;
                                  g = Gen} ;
```

Zum Testen der vorhandenen Funktionen aus einer Resource-Grammatik kann man die entsprechende Datei laden (wobei die GF-Pfad-Umgebungsvariable nichts nützt und daher der relative Pfadname benutzt wird):

⟨*Laden der Flexionsbildungen für das Deutsche*⟩≡

```
> i -retain ../../GF-2.7/lib/resource-1.0/german/ParadigmsGer.gf
...
  wrote file ../../GF-2.7/lib/resource-1.0/german/./ParadigmsGer.gfc 1721 msec

1663 msec
> cc regN "Fehler"
{
  g : Gender = Masc ;
  s : Number => Case => Str = table Number {
    Sg => table Case {
      Nom => "Fehler" ;
      Acc => "Fehler" ;
      Dat => "Fehler" ;
```

```
      Gen => "Fehlers"
    } ;
    Pl => table Case {
      Nom => "Fehler" ;
      Acc => "Fehler" ;
      Dat => "Fehlern" ;
      Gen => "Fehler"
    }
    } ;
  lock_N : {
    }
  = < >
}

> cc reg2N "Irrtum" "Irrtuemer" masculine
{
  g : Gender = Masc ;
  s : Number => Case => Str = table Number {
    Sg => table Case {
      Nom => "Irrtum" ;
      Acc => "Irrtum" ;
      Dat => variants {
        "Irrtum" ;
        "Irrtume"
      } ;
      Gen => variants {
        "Irrtums" ;
        "Irrtumes"
      }
      } ;
    Pl => table Case {
      Nom => "Irrtuemer" ;
      Acc => "Irrtuemer" ;
      Dat => "Irrtuemern" ;
      Gen => "Irrtuemer"
    }
    } ;
  lock_N : {
    }
  = < >
}
```

(Immer noch weiß ich nicht, wie man die Lexer dazu bekommt, die Umlaute beim Einlesen vom Terminal zu erkennen.)

Aufgabe 5 Versuche, die Morphologie der konkreten `Ap1`-Grammatiken mit Hilfe der Resource-Grammatiken für das Englische und Deutsche einfacher aufzubauen.

Z.B. enthält die Resource-Grammatik für das Deutsche (`GF-2.7/lib/resource-1.0/german/`) schon ein `reg2N` wie in der vorigen Aufgabe geplant. Wie muß man es in die eigene Grammatik einbauen?

Lösung 5

Aufgabe 6 Erweitere das im Tutorium angegebene `FoodComments`-Beispiel um eine konkrete Grammatik `FoodCommentsGer`.

Lösung 6 Das Beispiel `FoodComments` ist lediglich eine Erweiterung von `Comments` um bestimmte Konstruktoren:

⟨*Kopie von /GF-2.7/doc/tutorial/applications/FoodComments.gf*⟩≡

```
abstract FoodComments = Comments ** {

  fun
    Wine, Cheese, Fish, Pizza : Kind ;
    Fresh, Warm, Italian, Expensive, Delicious, Boring : Quality ;

}
```

Wir müssen also erst den Kern, `CommentsGer`, bauen. In `Comments.gf` im Verzeichnis `GF-2.7/doc/tutorial/applications/` ist die abstrakte Syntax definiert, die man durch eine eigene Datei `CommentsGer.gf` konkretisieren muß.

⟨*Kopie von GF-2.7/doc/tutorial/applications/Comments.gf*⟩≡

```
abstract Comments = {

  cat
    S ; Item ; Kind ; Quality ;

  fun
    Is : Item -> Quality -> S ;
    This, That, These, Those : Kind -> Item ;
    QKind : Quality -> Kind -> Kind ;
    Very : Quality -> Quality ;

}
```

Das im Tutorial angegebene Beispiel `FoodCommentsEng.gf` kann man aus einem anderen Verzeichnis unter seinem absoluten Pfad laden, wozu Hilfsdateien aus den Unterverzeichnissen `applications`, `prelude`, und `resources` von `$GF_LIB_PATH/../doc/tutorial` gebraucht werden. Diese findet der Compiler, wenn man beim Aufruf die Verzeichnisse in `-path=...` (und relativ zu `$GF_LIB_PATH`) angibt:

⟨*Test der Pfadangaben*⟩≡

```
i -v -path=:prelude:../doc/tutorial/applications:../doc/tutorial/prelude:../doc/tutorial/reso
  /home/leiss/GF-2.7/doc/tutorial/applications/FoodCommentsEng.gf

module search path:
["",
```

```
"prelude",
"../doc/tutorial/applications",
"../doc/tutorial/prelude",
"../doc/tutorial/resource",
"",
"/home/leiss/GF-2.7/lib/",
"/home/leiss/GF-2.7/lib/prelude",
"/home/leiss/GF-2.7/lib/../doc/tutorial/applications",
"/home/leiss/GF-2.7/lib/../doc/tutorial/prelude",
"/home/leiss/GF-2.7/lib/../doc/tutorial/resource",
"/home/leiss/GF-2.7/lib/"]
all modules:
["LexEng","MorphoEng","Predef","Prelude","SyntaxEng","Syntax","CommentsI",
"CommentsEng","Comments","FoodComments","FoodCommentsEng"]
```

Der Suchpfad enthält *sowohl* die Angaben in `-path=...` *als auch* ihre um das Präfix `$GF_LIB_PATH` erweiteren Varianten, sodaß ggf. statt der Bibliotheksdateien eigene Dateien gleichen Namens (relativ zum aufrufenden Verzeichnis) gefunden werden. (Ob GF nicht besser zwei getrennte Pfade verwalten sollte?)

Das Beispiel aus `GF-2.7/doc/tutorial/applications` ist durch einen Funktor `CommentsI`, eine „unvollständige konkrete" Grammatik, definiert:

⟨*Kopie von GF-2.7/doc/tutorial/applications/CommentsI.gf*⟩≡

```
--# -path=.:prelude

incomplete concrete CommentsI of Comments = open Syntax in {

  lincat
    S = Syntax.S ;
    Quality = AP ;
    Kind = CN ;
    Item = NP ;

  lin
    Is item quality = PosVP item (PredAP quality) ;
    This  = DetCN this_Det ;
    That  = DetCN that_Det ;
    These = DetCN these_Det ;
    Those = DetCN those_Det ;
    QKind = ModCN ;
    Very  = AdAP very_AdA ;

}
```

Um die konkrete Grammatik `CommentsGer` zu bilden, soll dieser Funktor `CommentsI` auf eine deutsche Konkretisierung `SyntaxGer` von `Syntax` angewendet werden, die wir in `Beispiele/SyntaxGer.gf` definieren werden. Das Ganze soll aus dem Oberverzeichnis von `Beispiele/` (oder aus `Beispiele`) aufgerufen werden.

⟨*Beispiele/CommentsGer.gf*⟩≡
```
--# -path=.:Beispiele:../doc/tutorial/resource:../doc/tutorial/applications:../doc/tutorial/p

concrete CommentsGer of Comments =
    CommentsI with (Syntax = SyntaxGer) ;
```

In `# -path=...` ist neben Unterverzeichnissen von `../doc/tutorial` für die Dateien wie `Syntax.gf`, `Comments.gf` auch das Verzeichnis `$GF_LIB_PATH/present` angegeben, das übersetzte Versionen von Dateien aus der Resource-Grammatik `resource-1.0/german` und den allen Resource-Grammatiken gemeinsamen Verzeichnissen `resource-1.0/common` und `resource-1.0/abstract` enthält.

Nun müssen wir noch eine zur abstrakten Syntax `Syntax.gf` passende konkrete Grammatik `SyntaxGer.gf` bilden. Die abstrakte Syntax sieht so aus:

⟨*Kopie von GF-2.7/doc/tutorial/resource/Syntax.gf*⟩≡
```
abstract Syntax = {

  flags startcat=Phr ;

  cat
    S ;    -- declarative sentence  e.g. "this pizza is good"
    NP ;   -- noun phrase           e.g. "this pizza"
    CN ;   -- common noun           e.g. "pizza"
    Det ;  -- determiner            e.g. "this"
    AP ;   -- adjectival phrase     e.g. "very good"
    AdA ;  -- adadjective           e.g. "very"
    VP ;   -- verb phrase           e.g. "is good"
    V ;    -- intransitive verb     e.g. "boil"
    V2 ;   -- two-place verb        e.g. "eat"

  fun
    PosVP, NegVP : NP -> VP -> S ;
    PredAP : AP -> VP ;
    PredV  : V  -> VP ;
    PredV2 : V2 -> NP -> VP ;
    DetCN  : Det -> CN -> NP ;
    ModCN  : AP -> CN -> CN ;
    AdAP   : AdA -> AP -> AP ;

  -- entries of the closed lexicon
```

```
    this_Det  : Det ;
    that_Det  : Det ;
    these_Det : Det ;
    those_Det : Det ;
    every_Det : Det ;
    theSg_Det : Det ;
    thePl_Det : Det ;
    a_Det     : Det ;
    plur_Det  : Det ;
    two_Det   : Det ;

    very_AdA  : AdA ;
    too_AdA   : AdA ;

}
```

Bem. Wieso kann das nicht deklarierte `Phr` als Startkategorie benutzt werden?

Um `SyntaxGer.gf` analog zu `SyntaxIta.gf`, also

⟨*Auszug aus GF-2.7/doc/tutorial/resources/Syntax.Ita.gf*⟩≡

```
--# -path=.:prelude
concrete SyntaxIta of Syntax = open Prelude, MorphoIta in { ... }
```

zu bilden, brauchen wir noch eine Morphologie `MorphoGer`. Wir übernehmen die Morphologie der Resource-Grammmatik des Deutschen und definieren uns nur mit `ResGer.mkA` und `ResGer.appAdj` die nötigen Konstruktoren der Kategorie `Det`.

Um `SyntaxGer.gf` zu definieren, könnte man das Beispiel `SyntaxEng.gf` (oder, da das Italienische wegen der Morphologie dem Deutschen ähnlicher ist, besser `SyntaxIta.gf`) aus `GF-2.7/doc/tutorial/resources/` studieren und an das Deutsche anpassen. Dazu kann man aber nicht einfach die Kategorien übernehmen, sondern muß sich die für das Deutsche eigenen Abhängigkeiten überlegen und passende Felder vorsehen. Da das in der Resource-Grammatik für das Deutsche schon gemacht ist, wenn auch viel genauer als für das Beispiel nötig, benutzen wir für `NP` und `AP` die Kategorien und Regeln der Resource-Grammatik, und definieren `V`, `VP` und `S` einfacher als in der Resource-Grammatik (z.B. brauchen wir die verschiedenen Verbstellungen im Satz nicht zu berücksichtigen).

Wir entnehmen die in `GF-2.7/lib/present` enthaltenen übersetzten Dateien der Resource-Grammatiken aus `resource-1.0/german`:

⟨*Beispiele/SyntaxGer.gf*⟩≡

```
--# -path=../doc/tutorial/resource:present:prelude

concrete SyntaxGer of Syntax =
  open Prelude, ResGer, ParadigmsGer, MorphoGer,
```

```
     CatGer, NounGer, AdjectiveGer
in {

lincat
  Det = CatGer.Det ;
  CN  = CatGer.CN ;
  NP  = CatGer.NP ;
  AP  = CatGer.AP ;
  AdA = CatGer.AdA ;
  -- Einfacher als CatGer.V, CatGer.VP, CatGer.V2, CatGer.S:
  V   = {s : Number => Str} ;
  VP  = {s : Bool => Number => Str} ;
  V2  = {s : Number => Str ; c : Case} ;
  S   = {s : Str} ;

lin
  DetCN = NounGer.DetCN ;
  ModCN = NounGer.AdjCN ;

  AdAP  = AdjectiveGer.AdAP ;
  very_AdA = {s = "sehr"} ** {lock_AdA = <>} ;
  too_AdA = {s = "zu"}  ** {lock_AdA = <>} ;

  PredV v      = {s = \\b,n => posneg b (v.s ! n)} ;
  PredV2 v2 np = {s = \\b,n => posneg b (v2.s ! n) ++ (np.s ! v2.c)} ;
  PredAP ap    = {s = \\b,n => copula b n ++ ap.s ! APred} ;

  PosVP  np vp = {s = np.s ! Nom ++ vp.s ! True  ! np.a.n} ;
  NegVP  np vp = {s = np.s ! Nom ++ vp.s ! False ! np.a.n} ;

  this_Det  = mkDet "dies" Sg Weak ;
  these_Det = mkDet "dies" Pl Weak ;
  that_Det  = mkDet "jen" Sg Weak ;
  those_Det = mkDet "jen" Pl Weak ;
  every_Det = mkDet "jed" Sg Weak ;

  theSg_Det = mkArt DefArt Sg ;
  thePl_Det = mkArt DefArt Pl ;
  a_Det     = mkArt IndefArt Sg ;
  plur_Det  = mkArt IndefArt Pl ;

  two_Det = mkDet "zwei" Pl Strong ; -- falsch; nur testhalber

oper
```

```
    mkDet : Str -> Number -> Adjf -> Det = \dies,n,a ->
      let Adj = (ResGer.mkA dies dies "dcomp" "dsup") in
      {s = appAdj Adj ! n ; n = n ; a = a} ** {lock_Det = <>} ;

    mkArt : Quant -> Number -> Det =  \q,n ->
      {s = q.s ! n ; n = n ; a = q.a} ** {lock_Det = <>} ;

    posneg : Bool -> Str -> Str = \b,do -> case b of {
      True => do ;
      False => do ++ "nicht"
      } ;

    copula : Bool -> Number -> Str = \b,n -> case n of {
      Sg => posneg b "ist" ;
      Pl => posneg b "sind"
      } ;
}
```

Bem. Anscheinend muß man `lock`-Felder nur bei den 0-stelligen Konstanten angeben.

Ob die hier definierten Hilfsfunktionen die gewünschten Formen erzeugen, kann man z.B. wie folgt testen:

⟨*Testen eigener Hilfsfunktionen*⟩≡

```
cc -res=SyntaxGer this_Det
```

Um die Anwendung `FoodCommentsGer` für das Deutsche zu implementieren, muß `CommentsGer` um das benötigte Vokabular erweitert werden.

In der Resource-Grammatik `resource-1.0/german/ParadigmsGer.gf` werden `regN, regA, mkA` definiert. Diese kann man separat laden und testen:

⟨*Laden und testen*⟩≡

```
> i -retain /home/leiss/GF-2.7/lib/resource-1.0/german/ParadigmsGer.gf
> cc regN "Wein"
```

In der Beispielgrammatik sollen die Wörter allerdings keine `N` und `A`-Ausdrücke sein, sondern von den Kategorien `CN` und `AP`. Daher müssen die von `ParadigmsGer.gf` konstruierten Ausdrücke noch umgewandelt werden:

⟨*Beispiele/FoodCommentsGer.gf*⟩≡

```
--# -path=.:Beispiele:../doc/tutorial/resource:../doc/tutorial/applications:../doc/tutorial/p
concrete FoodCommentsGer of FoodComments =
  CommentsGer ** open ParadigmsGer, AdjectiveGer in {
  lin
    Wine = mkCN (regN "Wein") ;
    Cheese = mkCN (reg2N "Käse" "Käse" masculine);
```

```
    Fish = mkCN (regN "Fisch") ;
    Pizza = mkCN (reg2N "Pizza" "Pizzen" feminine) ;
    -- die Adjektivflexion ist fehlerhaft bei Gen
    Fresh = mkAP "frisch" "frischer" "frischste" ;
    Warm = mkAP "warm" "wärmer" "wärmste" ;  -- Umlautausgabe?
    Italian = mkAP "italienisch" "italienischer" "italienischste" ;
    Expensive = mkAP "teuer" "teurer" "teuerste" ;
    Delicious = mkAP "schmackhaft" "schmackhafter" "schmackhafteste" ;
    Boring = mkAP "fade" "fader" "fadeste";

  oper
    mkCN : N -> CN =
      \noun -> {s = \\_ => noun.s ; g = noun.g ; lock_CN = <>} ;

    mkAP : Str -> Str -> Str -> AP =
      \pos,cmp,sup -> PositA (mkA pos cmp sup) ;
}
```

Wenn man alles übersetzt hat, kann man Beispiele erzeugen:

⟨*Erzeugte Beispielsätze*⟩≡

```
> gr -number=12 | l
dieser Fisch ist teuer
jene Fische sind frisch
diese Pizzen sind frisch
jener italienische Wein ist sehr frisch
jene italienischen Pizzen sind sehr teuer
diese Pizzen sind fade
jener sehr italienische Fisch ist frisch
jene Pizza ist italienisch
jener Wein ist warm
dieser Käse ist fade
dieser italienische Käse ist italienisch
diese Pizza ist fade
```

Beachte, daß die `SyntaxGer`-Grammatik mehr kann als die speziellere `FoodComments`, in der z.B. keine unbestimmten Artikel erlaubt sind.

Übungsaufgabe: Es wäre klüger, die in `SyntaxGer.gf` definierten `mkDet`, `mkArt` und die in `FoodCommentsGer.gf` definierten `mkCN`, `mkAP` in eine Datei `LexGer.gf` auszulagern, um damit das Vokabular für andere Beispiele aufbauen zu können. (Es gibt entsprechende `LexIta.gf` und `LexEng.gf` in `../doc/tutorial/resources`.)

Aufgabe 7 („Grammar writing by example“) Unter `GF-2.7/examples/animal` gibt es eine Beispielgrammatik zu Fragen mit der abstrakten Syntax

⟨*Kopie von GF-2.7/examples/Questions.gf*⟩≡
```
-- Simple questions and answers, in present tense.

abstract Questions = {
  cat
    Phrase ; Entity ; Action ;
  fun
    Who    : Action -> Entity -> Phrase ;           -- who chases X
    Whom   : Entity -> Action -> Phrase ;           -- whom does X chase
    Answer : Entity -> Action -> Entity -> Phrase ; -- X chases Y
}
```

Diese wird dann so konkretisiert, daß man als Linearisierungstypen geeignete Kategorien der Resource-Grammatiken wählt und die Linearisierung der abstrakten Konstruktoren, d.h. die Grammatikregeln, nicht explizit angibt, sondern *durch das Parsen mit den Resourcegrammatiken* automatisch erstellt.

Dazu wird (sprachübergreifend) ein Funktor in einem besonderen `.gfe`-Format definiert:

⟨*Kopie von GF-2.7/examples/QuestionsI.gfe*⟩≡
```
--# -resource=../../lib/present/LangEng.gfc
--# -path=.:present:prelude

-- to compile: gf -examples QuestionsI.gfe
-- or use directly gf <mkAnimals.gfs

incomplete concrete QuestionsI of Questions = open Lang in {
  lincat
    Phrase = Phr ;
    Entity = N ;
    Action = V2 ;

  lin
    Who  love_V2 man_N           = in Phr "who loves men" ;
    Whom man_N love_V2           = in Phr "whom does the man love" ;
    Answer woman_N love_V2 man_N = in Phr "the woman loves men" ;

}
```

GF übersetzt diese Datei `QuestionI.gfe` in den eigentlichen Funktor `Questions.gf`. Dabei wird die Definition der Konstruktoren `Who`, `Whom`, `Answer` aus einer Analyse des jeweiligen Beispielsatzes errechnet, indem man von den Wörtern des Beispielsatzes abstrahiert.

Hierbei wird unter mehreren Lesarten des Beispiels eine ausgewählt:

⟨*GF-2.7/examples/QuestionsI.gf, lesbar formatiert*⟩≡

```
-- File generated by GF from ./QuestionsI.gfe
incomplete concrete QuestionsI of Questions = open Lang in {
  lincat Action = V2 ;
  lin Answer = \woman_N -> \love_V2 -> \man_N ->
        PhrUtt NoPConj
          (UttNP (DetCN (DetSg (SgQuant DefArt) NoOrd)
                        (ApposCN
                           (ApposCN (UseN woman_N)
                                    (DetCN (DetPl (PlQuant IndefArt) NoNum NoOrd)
                                           (UseN love_N)))
                           (DetCN (DetPl (PlQuant IndefArt)NoNum NoOrd)
                                  (UseN man_N)))
                 )
          ) NoVoc ;
  lincat Entity = N ;
  lincat Phrase = Phr ;
  lin Who = \love_V2 -> \man_N ->
        PhrUtt NoPConj
           (UttAdv
             (AdvSC
               (EmbedQS
                 (UseQCl TPres ASimul PPos
                         (QuestVP
                            whoSg_IP
                            (ComplV2 love_V2
                                     (DetCN (DetPl (PlQuant IndefArt) NoNum NoOrd)
                                            (UseN man_N)))))))
           ) NoVoc ;

  lin Whom = \man_N -> \love_V2 ->
        PhrUtt NoPConj
          (UttQS (UseQCl TPres ASimul PPos
                    (QuestSlash whoPl_IP
                        (SlashV2 (DetCN (DetSg (SgQuant DefArt) NoOrd)
                                        (UseN man_N))
                                 love_V2)))
          ) NoVoc ;
  }
{-
AMBIGUOUS in Who
... -}  -- andere Lesarten der Beispielsätze, H.L.
```

Bem. Die für `Answer` und `Who` gewählten Lesarten sind fehlerhaft, s.u.

Dann kann man diesen Funktor auf eine der Sprachen anwenden:

⟨*Beispiele/QuestionsGer.gf*⟩≡

```
--# -path=.:Beispiele:../examples/animal:present:prelude
--  Variante Beispiele/QuestionsI.gf(e) berücksichtigen

concrete QuestionsGer of Questions = QuestionsI with
  (Lang = LangGer) ;
```

Das Ganze wird dann spezialisiert auf ein Anwendungsgebiet, Fragen über Tiere:

⟨*Kopie von GF-2.7/examples/Animals.gf*⟩≡

```
abstract Animals = Questions ** {

  flags startcat=Phrase ;

  fun
    -- a lexicon of animals and actions among them
    Dog, Cat, Mouse, Lion, Zebra : Entity ;
    Chase, Eat, See : Action ;
}
```

Um das anzuwenden, muß man noch ein Lexikon für diese abstakten Wörter angeben, wozu man wieder geeignete Funktionen der Resourcegrammatik verwendet:

⟨*Beispiele/AnimalsGer.gf*⟩≡

```
--# -path=.:Beispiele:../examples/animal:present:prelude

concrete AnimalsGer of Animals = QuestionsGer **
  open LangGer, ParadigmsGer, IrregGer in {

  lin
    Dog = regN "Hund" ;
    Cat = regN "Katze" ;
    Mouse = reg2N "Maus" "Mäuse" feminine ;
    Lion = reg2N "Löwe" "Löwen" masculine ;
    Zebra = reg2N "Zebra" "Zebras" neuter;
    Chase = dirV2 (regV "jagen") ;
    Eat = dirV2 fressen_V ;
    See = dirV2 sehen_V ;
}
```

Die übersetzte Grammatik kann man mit `print_multigrammar` in eine Datei schreiben:

⟨*Ausgabe der übersetzten Grammatik*⟩≡

```
> i Beispiele/AnimalsGer.gf | pm | wf Beispiele/animals.gfcm
```

Aufgaben:

(a) Man variiere die Beispielsätze in `QuestionsI.gfe`, z.B. nehme man andere Artikel, Quantoren oder Numerus bei den Nominalphrasen, und erzeuge eine neue Variante.

(b) Man konstruiere ein `FoodCommentsGer`, indem man nur Katergorien der Resource-Grammatik für das Deutsche verwendet, und definiere die Konstruktoren durch das Parsen geeigneter Beispielen (aus dem Englischen).

Lösung 7 Der erste Teil sollte nicht schwierig sein. Wir legen eine neue `.gfe`-Datei mit anderen Beispielsätzen (und passenden Pfaden!) an:

⟨*Beispiele/QuestionsI.gfe*⟩≡

```
--# -resource=present/LangEng.gfc
--# -path=.:Beispiele:present:prelude:../examples/animal

-- to compile: gf -examples QuestionsI.gfe

incomplete concrete QuestionsI of Questions = open Lang in {
  lincat
    Phrase = Phr ;
    Entity = N ;
    Action = V2 ;

  lin
    Who  love_V2 man_N           = in Phr "who loves a man" ;
    Whom man_N love_V2           = in Phr "whom does the man love" ;
    Answer woman_N love_V2 man_N = in Phr "the woman loves every man" ;

}
```

Das kann man wie in `examples/animal/mkAnimals` mit dem Vokabular einer konkreten Grammatik aufrufen:

⟨*Aufruf zur Erzeugung*⟩≡

```
> i -ex Beispiele/AnimalsGer.gf
file Animals.gfc not found
File Animals.gfc does not exist.
+ reading /home/leiss/GF-2.7/lib/present/Common.gfc 138 msec
...
```

Aber es werden die Beispiele aus `examples/animal/QuestionsI.gfe` genommen, nicht die aus `Beispiele/QuestionsI.gfe`, obwohl das Verzeichnis zuerst im Pfad ist.

⟨*Noch nicht richtig:*⟩≡

```
> i -v -path=.:Beispiele:../examples/animal:present:prelude -ex /home/leiss/Vorlesungen/Gramm
all modules: ["IrregGerAbs","IrregGer","LexiconGer","StructuralGer","ParadigmsGer","IdiomGer"
Going to preprocess examples in /home/leiss/GF-2.7/lib/../examples/animal/QuestionsI.gfe
Compiling resource ../../lib/present/LangEng.gfc
File ../../lib/present/LangEng.gfc not found.
gf: ../../lib/present/LangEng.gfc: openFile: does not exist (No such file or directory)
```

Das Problem ist die Reihenfolge, in der die Verzeichnisse durchsucht werden! (Wenn man andere Beispiele in `examples/animal/QuestionsI.gfe` einsetzt, geht es.)

1 Testen der vorhandenen Resource-Grammatiken

(a) Die Umgebungsvariable `$GF_LIB_PATH` kann man durch

⟨*Umgebungsvariable setzen, für bash-Shell in /.bashrc oder /.profile*⟩≡

```
export GFHOME=$HOME/GF-2.7
export GF_LIB_PATH=$HOME/GF-2.7/lib
```

auf –bei mir– `/home/leiss/GF-2.7/lib` setzen. Beim Laden muß der absolute Pfadname der zu ladenden Datei angegeben werden; die `-path=...`-Angabe dient nur zum Suchen der benötigten zusätzlichen Module:

⟨*Laden mit Pfad zum Lib-Verzeichnis*⟩≡

```
> i -v -path=present:prelude /home/leiss/GF-2.7/lib/present/LangGer.gfc
module search path: ["present","prelude","",
                       "/home/leiss/GF-2.7/lib/present",
                       "/home/leiss/GF-2.7/lib/prelude",
                       "/home/leiss/GF-2.7/lib/"]
```

Analog lädt man `LangEng` und weitere Sprachen.

(b) Zum Parsen aus dem Englischen braucht `gf` beim ersten Verwenden deutlich länger als bei der Wiederholung:

⟨*Beipiel* ⟩≡

```
> p -mcfg "I see her"

PhrUtt NoPConj (UttS (UseCl TPres ASimul PPos (PredVP (UsePron i_Pron)
           (ComplV2 see_V2 (UsePron she_Pron))))) NoVoc

3927 msec
> p -mcfg "I see her"

PhrUtt NoPConj (UttS (UseCl TPres ASimul PPos (PredVP (UsePron i_Pron)
           (ComplV2 see_V2 (UsePron she_Pron))))) NoVoc

127 msec
> p -mcfg "I see her" | l -multi

I see her

Ich sehe sie
```

(c) Bei Parsen aus dem Deutschen dauert es wesentlich länger als beim Englischen:

⟨*Parsen aus dem Deutschen, mit 2 Analysen*⟩≡

```
p -mcfg -lang=LangGer "Ich sehe sie"
```

```
PhrUtt NoPConj (UttS (UseCl TPres ASimul PPos (PredVP (UsePron i_Pron)
           (ComplV2 see_V2 (UsePron she_Pron))))) NoVoc
PhrUtt NoPConj (UttS (UseCl TPres ASimul PPos (PredVP (UsePron i_Pron)
           (ComplV2 see_V2 (UsePron they_Pron))))) NoVoc

39784 msec
> p -mcfg -lang=LangGer "Ich sehe sie"

PhrUtt NoPConj (UttS (UseCl TPres ASimul PPos (PredVP (UsePron i_Pron)
           (ComplV2 see_V2 (UsePron she_Pron))))) NoVoc
PhrUtt NoPConj (UttS (UseCl TPres ASimul PPos (PredVP (UsePron i_Pron)
           (ComplV2 see_V2 (UsePron they_Pron))))) NoVoc

2329 msec
```

Auch wenn man die Zweideutigkeit ausschließt, dauert es wegen der Morphologie länger als im Englischen:

⟨*Parsen aus dem Deutschen, eindeutige Analyse*⟩≡

```
> p -lang=LangGer "Ich sehe ihn"

PhrUtt NoPConj (UttS (UseCl TPres ASimul PPos (PredVP (UsePron i_Pron)
           (ComplV2 see_V2 (UsePron he_Pron))))) NoVoc

2418 msec
> p -lang=LangGer "Ich sehe ihn"

PhrUtt NoPConj (UttS (UseCl TPres ASimul PPos (PredVP (UsePron i_Pron)
           (ComplV2 see_V2 (UsePron he_Pron))))) NoVoc

861 msec
```

Es sieht also so aus, als erfordere die Morphologie im Deutschen einen Faktor 10 - zumindest bei so kurzen Beispielen!

(d) Problematischer ist, daß der Parser (auch bei winzigen Beispielen) divergieren oder zumindest an die Speichergrenze kommen kann:

⟨*Fehler beim Parsen aus dem Deutschen:*⟩≡

```
> p -lang=LangGer "Ich kann ihn sehen"

stack overflow
11391 msec
```

Das passiert auch bei Angabe von `-mcfg -parser=topdown` und anderen Optionen. (Es liegt wohl an einem Fehler im Parser, s.u., und sollte in GF-2.8 behoben sein.)

2 Änderungen von GF-Quelldateien

(a) Zum Compilieren der Resourcegrammatiken muß man mit

```
chmod ua+x GF-2.7/lib/resource-1.0/mkPresent
```

diese Datei ausführbar machen, damit `make present` usw. gehen.

(b) Korrektur von `resource-1.0/german/LexiconGer.gf`: Der Eintrag für `"Flugzeug"` muß lauten:

```
fun airplane_N = reg2N "Flugzeug" "Flugzeuge" neuter ;
```

(c) Das Lexikon für das Englische läßt sich erst übersetzen, wenn man die `ParamX`-Dateien in `../common` findet. Dazu wird die Pfadangabe in `LexiconEng.gf` ergänzt zu:

```
--# -path=.:prelude:../common:
```

Dann läßt sich zwar `LangEng` übersetzen, aber mit `ParadigmGer` bekommt man Schwierigkeiten

⟨*Fehler* ⟩≡

```
> i -retain /home/leiss/GF-2.7/lib/resource-1.0/german/ParadigmsGer.gf
- compiling /home/leiss/GF-2.7/lib/resource-1.0/german/../common/CommonX.gf...
     unknown qualified constant R.Anter
OCCURRED IN
renaming definition of AAnter
OCCURRED IN
renaming module CommonX
```

Deshalb habe ich `GF-2.7/lib/present` aus `GF-grammars-1.0.tgz` neu extrahiert. Vorher muß man alte `.gfc`-Versionen in den Quellverzeichnissen löschen, sonst werden diese gefunden und die `.gf` gar nicht neu kompiliert.

Man sollte aber besser nicht `LexiconEng.gf` ändern, nur weil man es separat übersetzen will, sondern mit `-path=..` beim Aufruf die nötigen Verzeichnisse angeben.

(d) Soweit ich sehe, ist die Adjektivflexion der eingebauten Deutschgrammatik unvollständig: sie sollte von der `Adjf` des Artikels oder Quantors abhängen: `der frische Fisch, ein frischer Fisch`. (Vgl. `DeclGer` aus Aufgabe 3, wo `mkFormsA` von `Adkl` in `UseA` abhängt).

⟨*Die Adjektivflexion der Resource-Grammatik*⟩≡

```
> cc -res=ResGer adjForms
\teuer_82 -> \teur_83 -> table AForm {
  APred => teuer_82 ;
```

```
    AMod (GSg Masc)Nom => teur_83 + "er" ;
    ...
    AMod GPl Gen => teur_83 + "er"
  }
```

Da die Abhängigkeit von `Adjf` in `LangGer` funktioniert, muß es irgendwo in der Resource-Grammatik nachgeholt werden; A.Ranta schrieb, beim Agreement würden die passenden Endungen aus dem vorhandenen Paradigma entnommen.

(e) In `GF-2.7/examples/animal/QuestionI.gf` wurde bei `lin Answer = ...` eine Lesart des Beispiels `the woman loves men` ausgewählt, in der `love_N` benutzt wird, was aber keine abstrahierte Variable ist. Das führt (bei meiner deutschen Version) zu falschen Übersetzungen wie

⟨*Falsche Ergebnisse*⟩≡

```
> l Answer Dog Chase Mouse
den Hund Lieben Mäuse
```

Die richtige Lesart findet man, wenn man die Beispiele in zwei Sprachen übersetzt:

⟨*Analysen mit LangEng.gfc und LangGer.gfc*⟩≡

```
> p "the woman loves men" | l -multi

The woman loves men
Die Frau Lieben Männer

The woman loves men
Die Frau Lieben Männer

The woman loves men
Die Frau liebt Männer
```

Man muß in `QuestionsI.gf` die letze der auskommentierten Lesarten des Beispiels in `lin Answer = ...` einsetzen und mit `gf < mkAnswers` neu übersetzen. Dann erhält man korrekt:

⟨*Ergebnisse nach Korrigieren von QuestionsI.gf*⟩≡

```
> l Answer Dog Chase Mouse
der Hund jagt Mäuse
```

Die Verbstellung bei den `Who`-Fragen ist ebenfalls falsch:

⟨*Fehlerhafte Verbstellung*⟩≡

```
> l Who See Zebra
wer Zebras sieht
```

Auch hier muß man die auskommentierte Lesart von `who loves men` in der Definition von `lin Who = ...` benutzen, um die Verbzweitstellung zu bekommen.

(f) Die Grammatik `lib/present/LangRus.gfc` des Russischen (aus `resource-1.0`) erzeugt nur Sätze im Konjunktiv, obwohl sie die im Indikativ erzeugen sollte. Der Grund liegt darin, daß die Auskommentierungen durch

⟨*Auskommentierung für die Beschränkung auf present tense*⟩≡

```
GF-2.7/lib/resource-1.0$ grep notpresent common/*.gf
common/CommonX.gf:    TPast = {s = []} ** {t = R.Past} ;   --# notpresent
common/CommonX.gf:    TFut  = {s = []} ** {t = R.Fut} ;    --# notpresent
common/CommonX.gf:    TCond = {s = []} ** {t = R.Cond} ;   --# notpresent
common/CommonX.gf:    AAnter = {s = []} ** {a = R.Anter} ; --# notpresent
common/ParamX.gf:     | Anter  --# notpresent
common/ParamX.gf:     | Past   --# notpresent
common/ParamX.gf:     | Fut    --# notpresent
common/ParamX.gf:     | Cond   --# notpresent

GF-2.7/lib/resource-1.0$ grep notpresent abstract/*.gf
abstract/Common.gf:    TPast, TFut, TCond : Tense ; -- I slept/will sleep/would sleep --
abstract/Common.gf:    AAnter : Ant ;               -- I have slept                   --
```

einige Datentypkonstruktoren beseitigt. Das führt aber dazu, daß an anderen Stellen die Fallunterscheidungen nicht mehr funktionieren, z.B. bei den `lin`-Definitionen aus

⟨*GF-2.7/lib/resource-1.0/russian/SentenceRus.gf* ⟩≡

```
    UseCl  t a p cl = {s = case t.t of {
      Cond => cl.s! p.p ! ClCondit ;
      _ => cl.s! p.p ! ClIndic (getTense t.t) a.a}};

    UseQCl t a p qcl= {s = case t.t of {
      Cond => qcl.s! p.p ! ClCondit ;
      _ => qcl.s!p.p! ClIndic (getTense t.t) a.a }};

    UseRCl t a p rcl ={s = \\gn,c,anim => case t.t of {
      Cond => [", "] ++ rcl.s! p.p ! ClCondit ! gn !c !anim ;
      _ => [", "] ++ rcl.s! p.p ! ClIndic (getTense t.t) a.a !gn !c !anim}};
```

Da der Datentyp `Tense` in `common/ParamX` den Konstruktor `Cond` bei der `present`-Version nicht mehr hat, wird in diesen `lin`-Definitionen *immer der erste Fall* genommen, da `Cond` *als Variable* interpretiert wird! (Weil deshalb der `_ => ...`-Fall unerreichbar ist, meldet der Compiler auch eine Warnung 'pattern not reached'!)

Da man die Datentypdefinitionen für den `present`-Fall auf weniger Konstruktoren beschränken will, *muß* man in allen `case`-Unterscheidungen die Fälle für die beseitigten Konstruktoren auch als `# notpresent` markieren. (Und wenn man anlog

Einschränkungen wie `# notpast` usw. einführen möchte?)

⟨*Korrektur von GF-2.7/lib/resource-1.0/russian/SentenceRus.gf*⟩≡

```
    UseCl  t a p cl = {s = case t.t of {
      Cond => cl.s! p.p ! ClCondit ;                 --# notpresent
      _ => cl.s! p.p ! ClIndic (getTense t.t) a.a}};

    UseQCl t a p qcl= {s = case t.t of {
      Cond => qcl.s! p.p ! ClCondit ;                --# notpresent
      _ => qcl.s!p.p! ClIndic (getTense t.t) a.a }};

    UseRCl t a p rcl ={s = \\gn,c,anim => case t.t of {
      Cond => [", "] ++ rcl.s! p.p ! ClCondit ! gn !c !anim ;   --# notpresent
      _ => [", "] ++ rcl.s! p.p ! ClIndic (getTense t.t) a.a !gn !c !anim}};
```

Nun übersetzt man die Resource-Grammatik neu (mit absolutem Pfad!):

⟨*Neue Übersetzung*⟩≡

```
> i /home/leiss/GF-2.7/lib/resource-1.0/russian/LangRus/.gf
...
wrote file /home/leiss/GF-2.7/lib/resource-1.0/russian/./LangRus.gfc 54804 msec
```

Das gab einen Fehler 'pattern not found for ParamX.Anter' (da nicht auf 'present' eingeschränkt wurde), weshalb ich analog auch die Grammatik `LangEng.gf` neu übersetzt habe. Dann geht es anscheinend.

Klüger wäre, mit `resource-1.0 > make present` alle Grammatiken für das Präsens neu zu übersetzen und mit `make install` in `lib/present` zu schieben.

(g) **Fehlersuche und Ausgabe der Chart**: Es gibt zwar eine `Control.Monad` und ein `GF.System.Tracing`, aber wie man die nutzt, ist nicht dokumentiert.

Man muß GF mit dem Glasgow Haskell Compiler aus `GF-2.7/src/` mit `make tracing` übersetzen und mit `make install` neu installieren. Dann werden die im Code angegebene `tracePrt <msg> <expr>` bei der Ausführung ausgegeben. Weitere in `GF/Parsing/GFC.hs` und `Formalism/Utilities.hs` eingefügte `tracePrt ..` werden unten genannt.

Bei deutschen Sätzen mit Modalverben divergiert der Parser:

⟨*Fehler im Deutschen*⟩≡

```
> p "er kann ihn sehen"
{Parsing.GFC - input tokens: input [0-1: Er][0-1: er][1-2: kann][2-3: ihn][3-4: sehen]}
{Parsing.GFC - nr. unfiltered forests: 1}
{Printing a list of SyntaxForests: {_[0] can8know_VV[] | _[0] can_VV[]}}

thread killed
21619 msec
```

Um das Problem des stack-overflows bei Modalverben in `LangGer` zu isolieren, wäre eine Ausgabe der Parsetabelle bzw. ein Protokoll der Einfügungen in die Parsetabelle sehr nützlich.

Das Problem muß an der Behandlung der Modal- oder Hilfsverben liegen (beim Parsen, denn das Linearisieren geht auch für Deutsch). Die abstrakte Konstruktion ist `ComplVV`, und aus dem Englischen wird `I can see her` auch erkannt.

⟨*aus abstract/Verb.gf und english/VerbEng.gf*⟩≡
```
  --  VV = Verb ** {isAux : Bool} ;
  --  ComplVV  : VV  -> VP -> VP ;        -- want to run

      ComplVV v vp = insertObj (\\a => infVP v.isAux vp a) (predVV v) ;
```
Für das Deutsche ist die Linearisierung komplizierter:

⟨*aus german/VerbGer.gf und german/ResGer.gf*⟩≡
```
      ComplVV v vp =
        let
          vpi = infVP v.isAux vp
        in
        insertExtrapos vpi.p3 (
          insertInf vpi.p2 (
            insertObj vpi.p1 (
              predVGen v.isAux v))) ;  -- v is a VV = Verb **{isAux:Bool},
                                       -- but predVGen expexts a Verb
        -- das geht aber auch bei V3 = Verb**{c2,c3:Prep} mit v.c3 und predV v
```

Die beteiligten Funktionen und Typen aus `ResGer`, die Linearisierungstypen aus `CatGer` u.a. scheinen zu stimmen (die Linearisierung funktioniert ja!).

Nach `IrregGer` sind (nur) `können`, `müssen` und `wollen` Hilfsverben (d.h. mit `auxVV` gebildet und daher `VV`s); aber ihre Erkennung funktioniert offenbar nicht; warum, verstehe ich nicht.

⟨*Auszug aus StructuralGer.gf*⟩≡
```
    want_VV = auxVV (mkV "wollen" ... "wollte" "gewollen" [] VHaben) ;
```

Hierin setzt `auxVV` nur `isAux = True`, und natürlich müßte es `gewollt` statt `gewollen` heißen, `gekonnt` statt `gekonnen`, usw.

Um die Probleme mit den Modalverben im Deutschen zu beheben, braucht man weitere Trace-Ausgaben. Mit folgender Ausgabe von Zwischenergebnissen

⟨*Einfügung in GF-2.7/src/GF/Formalism/Utilities.hs* ⟩≡
```
  instance (Print s) => Print (SyntaxForest s) where
      prt (FNode s []) = "(" ++ prt s ++ " - ERROR: null forests)"
      prt (FNode s [[]]) = prt s
      prt (FNode s [forests]) =
          -- tracePrt "Printing a SyntaxForest with 1 child" prt $  -- H.L.
```

```
        "(" ++ prt s ++ prtBefore " " forests ++ ")"
    prt (FNode s children) =
        -- tracePrt "Printing a SyntaxForest with > 1 children" prt $  -- H.L.
        "{" ++ prtSep " | " [ prt s ++ prtBefore " " forests |
                                                     forests <- children ] ++ "}"
    prt (FString s) = show s
    prt (FInt    n) = show n
    prt (FFloat  f) = show f
    prt (FMeta)     = "?"
    prtList = prtAfter "\n"
```

und einer Anzeige doppelter Kanten (durch `<LOOP>`, s.u.) erhält man Analysen für die deutschen Sätze mit Modalverben:

⟨*Beispiel* ⟩≡

```
16588 msec
> p "ich kann nicht schlafen"
{Parsing.GFC - starting FCF categories: [Phr{.s}]}
{Parsing.GFC - input tokens: input [0-1: Ich][0-1: ich][1-2: kann][2-3: nicht][3-4: schl
{Parsing.GFC - nr. unfiltered forests: 1}
{Printing a SyntaxForest with 1 child: (UsePron[0] i_Pron[])}
{Printing a SyntaxForest with > 1 children: {_[0] can8know_VV[] | _[0] can_VV[]}}
{Printing a SyntaxForest with 1 child: (AdVVP[?,0] "<LOOP>")}
{Printing a SyntaxForest with 1 child: (_[0] sleep_V[])}
{Printing a SyntaxForest with 1 child: (UseV[0] (_[0] sleep_V[]))}
{Printing a SyntaxForest with > 1 children: {_[0] (AdVVP[?,0] "<LOOP>") | _[0] (UseV[0]
{Printing a SyntaxForest with 1 child: (ComplVV[0,1] {_[0] can8know_VV[] | _[0] can_VV[]
{Printing a SyntaxForest with 1 child: (_[0] (ComplVV[0,1] {_[0] can8know_VV[] | _[0] ca
{Printing a SyntaxForest with 1 child: (PredVP[0,1] (UsePron[0] i_Pron[]) (_[0] (ComplVV
{Printing a SyntaxForest with 1 child: (UseCl[0,1,2,3] TPres[] ASimul[] PNeg[] (PredVP[0
{Printing a SyntaxForest with 1 child: (UttS[0] (UseCl[0,1,2,3] TPres[] ASimul[] PNeg[]
{Printing a SyntaxForest with 1 child: (PhrUtt[0,1,2] NoPConj[] (UttS[0] (UseCl[0,1,2,3]
{Parsing.GFC - unfiltered forests: (PhrUtt[0,1,2] NoPConj[] (UttS[0] (UseCl[0,1,2,3] TPr
}
{Parsing.GFC - nr. forests: 1}
{Parsing.GFC - nr. trees: 4}

PhrUtt NoPConj (UttS (UseCl TPres ASimul PNeg (PredVP (UsePron i_Pron) (ComplVV can8know
PhrUtt NoPConj (UttS (UseCl TPres ASimul PNeg (PredVP (UsePron i_Pron) (ComplVV can8know
PhrUtt NoPConj (UttS (UseCl TPres ASimul PNeg (PredVP (UsePron i_Pron) (ComplVV can_VV (
PhrUtt NoPConj (UttS (UseCl TPres ASimul PNeg (PredVP (UsePron i_Pron) (ComplVV can_VV (
```

Wenn man sich die fehlerhafte Kante (für das Beispiel `Ich kann schlafen`) anzeigen läßt (s.u.), erhält man:

⟨*Fehlerhafte AdVVP-Kante*⟩≡
```
{Printing a SyntaxForest with 1 child:
 (AdVVP[?,0] "(VP{.a2;.ext;.inf;
                  .s!{g=Neutr; n=Sg; p=P3}!(VPInfinit Simul).inf;
                  .n2!{g=Fem; n=Sg; p=P1};
                  .isAux=False},
                [(?),(?),(?),(2-3),(?)])")}
```

Beim Linearisieren erhält man die Fehlermeldung:

⟨*Fehler* ⟩+≡
```
Unknown label a1 in {s="\"(VP{.a2;.ext;.inf;.s!{g=Neutr; n=Sg; p=P3}!(VPInfinit Simul).i
OCCURRED IN linearization of Lang.AdVVP: VP ?0: AdV "(VP{.a2;.ext;.inf;.s!{g=Neutr; n=Sg
...
OCCURRED IN linearization of Lang.PhrUtt: Phr {String<>VP} Lang.NoPConj: PConj (Lang.Utt
```

Wie man sieht, fehlt tatsächlich eine `a1`-Komponente, d.h. es ist keine korrekte `VP`. Das heißt wohl, daß die Regel `AdVVP adv vp : VP` mit *leerem* `adv` das Problem verursacht. Es kann daran liegen, daß die Kettenableitungen

⟨*VP aus leerer Adv + VP*⟩≡
```
VP -> Adv VP -> epsilon VP -> VP -> ...
```

nicht richtig abgefangen werden (aber wieso nur im Kontext von Modalverben?), oder daß der reine Infinitiv einen korrupten `VP`-Eintrag erzeugt.

Man muß also den Aufbau einer unendlichen Chart verhindern. Peter Ljunglöf hat dazu die Funktion `chart2forests` in `GF/Formalism/Utilities.hs` so geändert, daß ein weiteres Argument `edges` mitgeführt wird und die darin schon vorhandenen Kanten nicht nochmals eingetragen werden.

Zum Tracen muß man `(Print _)`-Anteile nicht in den Typ, sondern in den Kontext vor dem Typ einbauen! `Fname :: Kontext => Typ`

⟨*Änderung von GF-2.7/src/GF/Formalism/Utilities.hs*⟩≡
```
-- for tracing, (Print n, Print e) added, HL
chart2forests :: (Ord n, Ord e, Print n, Print e) =>
                 SyntaxChart n e  -- ^ The complete chart
              -> (e -> Bool)      -- ^ When is an edge 'FMeta'?
              -> [e]              -- ^ The starting edges
              -> SList (SyntaxForest n) -- ^ The result has unique keys, ie. all 'n' are
                                        -- In essence, the result is a map from 'n' to f

chart2forests chart isMeta
       | trace2 "\nChart:" (prtBefore "\n "
           [ prt e ++ " :=> " ++ prt n ++ " : " ++ prt ees |
             (e,nodes) <- aAssocs chart,
```

```
              (SNode n ees) <- nodes --,
--              e ‘elem‘ concat ees      -- show only edges already found
            ]) True
      = concatMap (edge2forests [])
      where edge2forests edges edge
--    | trace2 "edge" (prt edge) True
               | isMeta edge       = [FMeta]
               | edge ‘elem‘ edges = [] -- don’t add a known edge
--               | edge ‘elem‘ edges = [FString (prt edge)] -- trace
               | otherwise         =
                   map (item2forest (edge:edges)) $ chart ? edge
--            item2forest _ x | trace2 "item" (prt x) False = undefined
            item2forest edges (SMeta)               = FMeta
            item2forest edges (SNode name children) =
               FNode name $ children >>= mapM (edge2forests edges)
            item2forest edges (SString s)           = FString s
            item2forest edges (SInt    n)           = FInt    n
            item2forest edges (SFloat  f)           = FFloat  f

-- chart2forests chart isMeta = concatMap (edge2forests [])
--      where edge2forests edges edge
--                | isMeta edge       = [FMeta]
--                | edge ‘elem‘ edges = [] -- or [FString "<LOOP>"] to find the loop
--                | otherwise
--                    = map (item2forest (edge:edges)) $ chart ? edge
--             item2forest edges (SMeta)               = FMeta
--             item2forest edges (SNode name children)
--                    = FNode name $ children >>= mapM (edge2forests edges)
--             item2forest edges (SString s)           = FString s
--             item2forest edges (SInt    n)           = FInt    n
--             item2forest edges (SFloat  f)           = FFloat  f
```

Wir testen das Ausgeben der Parsetabelle an folgendem Beispiel:

⟨*Beispiele/Fool.gf*⟩≡

```
abstract Fool = {
   cat NBar ; AP ; A ; CN ; VGL ; P ; PN ;

   fun attrAP : AP -> CN -> NBar ;
       mkAP : A -> VGL -> AP ;
       mkVGL : P -> PN -> VGL ;

       duemmerer : A ;
       mann : CN ;
       als : P ;
```

```
        emil : PN ;
  }
```

⟨*Beispiele/FoolGer.gf*⟩≡

```
  concrete FoolGer of Fool = {
     lincat
         NBar, A, VGL, P, PN, CN = {s : Str} ;
         AP = {s1 : Str ; s2 : Str} ;

     lin attrAP ap cn = {s = ap.s1 ++ cn.s ++ ap.s2} ;
         mkAP   a vgl = {s1 = a.s ; s2 = vgl.s} ;
         mkVGL  p pn  = {s = p.s ++ pn.s} ;
         duemmerer = {s = "duemmerer"} ;
         mann = {s = "Mann"} ;
         als  = {s = "als"} ;
         emil = {s = "Emil"} ;
  }
```

Hier ist die Ausgabe für den normalen Parser (FCFG):

⟨*Parsen mit FCFG*⟩≡

```
  > p -cat=NBar "duemmerer Mann als Emil"
  {Parsing.GFC - input tokens: input [0-1: duemmerer][1-2: Mann][2-3: als][3-4: Emil]}
  {Options: Opts []}
  {GFCtoSimple - concrete language: FoolGer}
  {GFCtoSimple - simpleGFC rules: 7}
  {MCFG.PInfo - parser info: [ allRules=9; tdRules=9/9; epsilonRules=0; lcCats=4/5; lcToke
  {Chart:
   (P{.s},[(2-3)]) :=> als[] : [[]]
   (NBar{.s},[(0-4)]) :=> attrAP[0,1] : [[(AP{.s1;.s2},[(0-1),(2-4)]),(CN{.s},[(1-2)])]]
   (CN{.s},[(1-2)]) :=> mann[] : [[]]
   (AP{.s1;.s2},[(0-1),(2-4)]) :=> mkAP[0,1] : [[(A{.s},[(0-1)]),(VGL{.s},[(2-4)])]]
   (A{.s},[(0-1)]) :=> duemmerer[] : [[]]
   (PN{.s},[(3-4)]) :=> emil[] : [[]]
   (AP{.s2},[(2-4)]) :=> mkAP[?,0] : [[(VGL{.s},[(2-4)])]]
   (VGL{.s},[(2-4)]) :=> mkVGL[0,1] : [[(P{.s},[(2-3)]),(PN{.s},[(3-4)])]]
   (AP{.s1},[(0-1)]) :=> mkAP[0,?] : [[(A{.s},[(0-1)])]]}
  {Parsing.GFC - starting FCF categories: [NBar{.s}]}
  {Parsing.GFC - nr. unfiltered forests: 1}
  {Parsing.GFC - unfiltered forests: (attrAP[0,1] (mkAP[0,1] duemmerer[] (mkVGL[0,1] als[]
  }
  {Parsing.GFC - nr. forests: 1}
  {Parsing.GFC - nr. trees: 1}

  attrAP (mkAP duemmerer (mkVGL als emil)) mann
```

```
5 msec

> p -cat=NBar -parser=topdown "duemmerer Mann als Emil"
{Parsing.GFC - input tokens: input [0-1: duemmerer][1-2: Mann][2-3: als][3-4: Emil]}
{Parsing.GFC - starting FCF categories: [NBar{.s}]}
{Chart:
 (P{.s},[(2-3)]) :=> als[] : [[]]
 (NBar{.s},[(0-4)]) :=> attrAP[0,1] : [[(AP{.s1;.s2},[(0-1),(2-4)]),(CN{.s},[(1-2)])]]
 (CN{.s},[(1-2)]) :=> mann[] : [[]]
 (AP{.s1;.s2},[(0-1),(2-4)]) :=> mkAP[0,1] : [[(A{.s},[(0-1)]),(VGL{.s},[(2-4)])]]
 (A{.s},[(0-1)]) :=> duemmerer[] : [[]]
 (PN{.s},[(3-4)]) :=> emil[] : [[]]
 (VGL{.s},[(2-4)]) :=> mkVGL[0,1] : [[(P{.s},[(2-3)]),(PN{.s},[(3-4)])]]}
{Parsing.GFC - nr. unfiltered forests: 1}
{Parsing.GFC - unfiltered forests: (attrAP[0,1] (mkAP[0,1] duemmerer[] (mkVGL[0,1] als[]
}
{Parsing.GFC - nr. forests: 1}
{Parsing.GFC - nr. trees: 1}

attrAP (mkAP duemmerer (mkVGL als emil)) mann

3 msec
```

Nach `GF-2.7/src/GF/Parsing/MCFG.hs` kann man zwischen verschiedenen Parsestrategien wählen, die durch `-parser=strategie` anzugeben sind:

⟨*Optionen laut Parsing/MCFG.hs*⟩≡

```
strategies = words "bottomup topdown n an ab at i rn ran rab rat ri ft fb"
```

Außer bei den Optionen `topdown` und `bottomup` erhält man allerdings eine Fehlermeldung, daß die Parsestrategie nicht existiere; das muß man ignorieren. Beim MCFG-Parser ist allerdings die Anzeige der Kategorien schlechter:

⟨*Parsen mit MCFG*⟩≡

```
> p -mcfg -cat=NBar -parser=bottomup "duemmerer Mann als Emil"
{Parsing.GFC - input tokens: input [0-1: duemmerer][1-2: Mann][2-3: als][3-4: Emil]}
{MCFG.PInfo - Restricting the parser information: [(0-1),(1-2),(2-3),(3-4)]}
{Chart:
 (A{}[.s],[(.s,(0-1))]) :=> duemmerer[] : [[]]
 (AP{}[.s1],[(.s1,(0-1))]) :=> mkAP[0,?] : [[(A{}[.s],[(.s,(0-1))])]]
 (AP{}[.s1,.s2],[(.s1,(0-1)),(.s2,(2-4))]) :=> mkAP[0,1] : [[(A{}[.s],[(.s,(0-1))]),(VGL
 (AP{}[.s2],[(.s2,(2-4))]) :=> mkAP[?,0] : [[(VGL{}[.s],[(.s,(2-4))])]]
 (CN{}[.s],[(.s,(1-2))]) :=> mann[] : [[]]
 (NBar{}[.s],[(.s,(0-4))]) :=> attrAP[0,1] : [[(AP{}[.s1,.s2],[(.s1,(0-1)),(.s2,(2-4))])
```

```
 (P{}[.s],[(.s,(2-3))]) :=> als[] : [[]]
 (PN{}[.s],[(.s,(3-4))]) :=> emil[] : [[]]
 (VGL{}[.s],[(.s,(2-4))]) :=> mkVGL[0,1] : [[(P{}[.s],[(.s,(2-3))]),(PN{}[.s],[(.s,(3-4)
{Parsing.GFC - starting MCF categories: [NBar{}[.s]]}
{Parsing.GFC - final chart edges: [(NBar{}[.s],[(.s,(0-4))])]}
{Parsing.GFC - nr. unfiltered forests: 1}
{Parsing.GFC - unfiltered forests: (attrAP[0,1] (mkAP[0,1] duemmerer[] (mkVGL[0,1] als[]
}
{Parsing.GFC - nr. forests: 1}
{Parsing.GFC - nr. trees: 1}

attrAP (mkAP duemmerer (mkVGL als emil)) mann

4 msec
> p -mcfg -cat=NBar -parser=topdown "duemmerer Mann als Emil"
{Parsing.GFC - starting MCF categories: [NBar{}[.s]]}
{Parsing.GFC - input tokens: input [0-1: duemmerer][1-2: Mann][2-3: als][3-4: Emil]}
{MCFG.PInfo - Restricting the parser information: [(0-1),(1-2),(2-3),(3-4)]}
{Chart:
 (A{}[.s],[(.s,(0-1))]) :=> duemmerer[] : [[]]
 (AP{}[.s1,.s2],[(.s1,(0-1)),(.s2,(2-4))]) :=> mkAP[0,1] : [[(A{}[.s],[(.s,(0-1))]),(VGL
 (CN{}[.s],[(.s,(1-2))]) :=> mann[] : [[]]
 (NBar{}[.s],[(.s,(0-4))]) :=> attrAP[0,1] : [[(AP{}[.s1,.s2],[(.s1,(0-1)),(.s2,(2-4))])
 (P{}[.s],[(.s,(2-3))]) :=> als[] : [[]]
 (PN{}[.s],[(.s,(3-4))]) :=> emil[] : [[]]
 (VGL{}[.s],[(.s,(2-4))]) :=> mkVGL[0,1] : [[(P{}[.s],[(.s,(2-3))]),(PN{}[.s],[(.s,(3-4)
{Parsing.GFC - final chart edges: [(NBar{}[.s],[(.s,(0-4))])]}
{Parsing.GFC - nr. unfiltered forests: 1}
{Parsing.GFC - unfiltered forests: (attrAP[0,1] (mkAP[0,1] duemmerer[] (mkVGL[0,1] als[]
}
{Parsing.GFC - nr. forests: 1}
{Parsing.GFC - nr. trees: 1}

attrAP (mkAP duemmerer (mkVGL als emil)) mann
```

⟨*Ausgabe der Chart bei Parsen mit Ap2Ger.gf*⟩≡

```
 (NP{.g=mask;.n=sg;}[.s!nom],[(.s!nom,(4-6))]) :=>
    Det[0,1] : [[(Art{.d=sw;}[.s!mask!sg!nom],[(.s!mask!sg!nom,(4-5))]),
                 (N1{.g=mask;.n=sg;}[.s!sw!nom],[(.s!sw!nom,(5-6))])]]

 (AP{.c=split;}[.s!cmp.s2!nom],[(.s!cmp.s2!nom,(3-6))]) :=>
    CompA[?,0] : [[(NP{.n=sg;}[.s!nom],[(.s!nom,(4-6))])]]
```

Nun ein Beispiel mit zwei verschränkten gespaltenen Konstituenten:

⟨*Beispiele/ApExtract.gf*⟩≡

```
abstract ApExtract = {
   cat Det ; A ; N ; Pron ; V ; AP; CN ; NP ; P ; S ;

   fun
     DetCN   : Det -> CN -> NP ;
     UsePron : Pron -> NP ;
     UseN : N -> CN ;
     UseA : A -> NP -> AP ;
     AdjCN   : AP -> CN  -> CN ;
     NpVp : NP -> V -> NP -> S ;

     besseres : A ;
     buch : N ;
     als : P ;
     ich : Pron ;
     das, dein, ein : Det ;
     lesen : V ;
}
```

Die konkrete Version soll eine gespaltene Adjektivphrase und ein gespaltenes Verb enthalten:

⟨*Beispiele/ApExtractGer.gf*⟩≡

```
concrete ApExtractGer of ApExtract = {
  lincat
    V, AP, CN, NP = {s1 : Str; s2 : Str} ;
    A, P, N, Pron, Det, S = {s : Str} ;

  lin
    DetCN det cn = {s1 = det.s ++ cn.s1 ; s2 = cn.s2} ;
    UsePron pr = {s1 = pr.s ; s2 = ""} ;
    UseN n = {s1 = n.s ; s2 = ""} ;
    UseA a np = {s1 = a.s ; s2 = als.s ++ np.s1} ;
    AdjCN ap cn = {s1 = ap.s1 ++ cn.s1 ; s2 = cn.s2 ++ ap.s2} ;
    NpVp np1 v np2 = {s = np1.s1 ++ v.s1 ++ np2.s1 ++ v.s2 ++ np2.s2 } ;

    buch = {s = "Buch"} ;
    ich  = {s = "ich"} ;
    als = {s = "als"} ;
    das = {s = "das"} ;
    dein = {s = "dein"} ;
    ein = {s = "ein"} ;
    besseres = {s = "besseres"} ;
    lesen = {s1 = "habe";  s2 = "gelesen"} ;
```

```
}
```

⟨*Parsetabelle* ⟩≡

```
> p "ich habe ein besseres Buch gelesen als dein Buch"
Parsing.GFC - input tokens: [0-1: ich][1-2: habe][2-3: ein][3-4: besseres]
                 [4-5: Buch][5-6: gelesen][6-7: als][7-8: dein][8-9: Buch]
Chart:
 (CN{.s2},[(6-9)]) :=> AdjCN[0,1] : [[(AP{.s2},[(6-9)]),(CN{.s2},[(?)])]]
 (CN{.s2},[(?)]) :=> UseN[?] : [[]]
 (AP{.s2},[(6-9)]) :=> UseA[?,0] : [[(NP{.s1},[(7-9)])]]
 (CN{.s1},[(3-5)]) :=> AdjCN[0,1] : [[(AP{.s1},[(3-4)]),(CN{.s1},[(4-5)])]]
 (CN{.s1},[(4-5)]) :=> UseN[0] : [[(N{.s},[(4-5)])]]
 (CN{.s1},[(8-9)]) :=> UseN[0] : [[(N{.s},[(8-9)])]]
 (AP{.s1},[(3-4)]) :=> UseA[0,?] : [[(A{.s},[(3-4)])]]
 (NP{.s2},[(6-9)]) :=> DetCN[?,0] : [[(CN{.s2},[(6-9)])]]
 (NP{.s2},[(?)]) :=> DetCN[?,0] : [[(CN{.s2},[(?)])]]
 (NP{.s2},[(?)]) :=> UsePron[?] : [[]]
 (NP{.s1},[(0-1)]) :=> UsePron[0] : [[(Pron{.s},[(0-1)])]]
 (NP{.s1},[(2-5)]) :=> DetCN[0,1] : [[(Det{.s},[(2-3)]),(CN{.s1},[(3-5)])]]
 (NP{.s1},[(7-9)]) :=> DetCN[0,1] : [[(Det{.s},[(7-8)]),(CN{.s1},[(8-9)])]]
 (Det{.s},[(2-3)]) :=> ein[] : [[]]
 (Det{.s},[(7-8)]) :=> dein[] : [[]]
 (S{.s},[(0-9)]) :=> NpVp[0,1,2] : [[(NP{.s1},[(0-1)]),(V{.s1;.s2},[(1-2),(5-6)]),(NP{.s
 (NP{.s1;.s2},[(0-1),(?)]) :=> UsePron[0] : [[(Pron{.s},[(0-1)])]]
 (NP{.s1;.s2},[(2-5),(6-9)]) :=> DetCN[0,1] : [[(Det{.s},[(2-3)]),(CN{.s1;.s2},[(3-5),(6
 (NP{.s1;.s2},[(7-9),(?)]) :=> DetCN[0,1] : [[(Det{.s},[(7-8)]),(CN{.s1;.s2},[(8-9),(?)]
 (V{.s1;.s2},[(1-2),(5-6)]) :=> lesen[] : [[]]
 (A{.s},[(3-4)]) :=> besseres[] : [[]]
 (N{.s},[(4-5)]) :=> buch[] : [[]]
 (N{.s},[(8-9)]) :=> buch[] : [[]]
 (Pron{.s},[(0-1)]) :=> ich[] : [[]]
 (P{.s},[(6-7)]) :=> als[] : [[]]
 (V{.s2},[(5-6)]) :=> lesen[] : [[]]
 (V{.s1},[(1-2)]) :=> lesen[] : [[]]
 (CN{.s1;.s2},[(3-5),(6-9)]) :=> AdjCN[0,1] : [[(AP{.s1;.s2},[(3-4),(6-9)]),(CN{.s1;.s2}
 (CN{.s1;.s2},[(4-5),(?)]) :=> UseN[0] : [[(N{.s},[(4-5)])]]
 (CN{.s1;.s2},[(8-9),(?)]) :=> UseN[0] : [[(N{.s},[(8-9)])]]
 (AP{.s1;.s2},[(3-4),(6-9)]) :=> UseA[0,1] : [[(A{.s},[(3-4)]),(NP{.s1},[(7-9)])]]
Parsing.GFC - starting FCF categories: [S{.s}]
Parsing.GFC - nr. unfiltered forests: 1
Parsing.GFC - unfiltered forests: (NpVp[0,1,2] (UsePron[0] ich[]) lesen[] (DetCN[0,1] ei

Parsing.GFC - nr. forests: 1
Parsing.GFC - nr. trees: 1
```

```
NpVp (UsePron ich) lesen (DetCN ein (AdjCN (UseA besseres (DetCN dein (UseN buch))) (Use

12 msec
```

Bleibe zu ergänzen, daß man die Einträge in der Reihenfolge, wie sie gemacht werden, bekommt.