Ein Programm zur Suche mit syntaktischen Mustern

Hans Leiß, CIS

9. Mai 2003

Zusammenfassung

Es wird ein Programm dokumentiert, das aus einer Grammatik einen Parser und ein Suchprogramm erzeugt, mit dem man in getaggten Texten nach Ausdrücken suchen kann, die zu vorgegeben syntaktischen Strukturen passen.

Das Programm ist in der Programmiersprache Mercury geschrieben, einer getypten Variante von Prolog. Das Format der Grammatikregeln ist eine Modifikation der Regeln einer Definite-Clause-Grammatik, in denen optionale und iterierte Konstituenten erlaubt sind. Der Parser ist lediglich eine Modifikation der Top-Down-Suche von Prolog, allerdings so modifiziert, daß sie auch bei linksrekursiven Regeln terminiert und die gefundenen Analysen ausgibt.

Das Suchprogramm arbeitet auf einem lexikalisch markierten Text. Es verwendet als Suchmuster partielle Syntaxbäume, deren Mustervariablen bei der Suche durch geeignete Teilbäume aufgefüllt werden. Solche Suchmuster können aus einem Lexikon von Prädikaten berechnet werden,

Ordnet man einem logischen Prädikaten Liste von Musternamen, lexikalischen Stammformen, und logische Argumentstellen zu, so können daraus die passenden Suchmuster berechnet werden. In den gefundenen Ausdrücken werden dann die Teilausdrücke, die den logischen Argumenten entsprechen, markiert. In diesem Sinn ist das Programm auch ein Programm zu „semantischen“ Suche, bei der das logische Prädikat als „semantisches Muster“ vorgegeben wird und im Text nach Ausdrücke verschiedener syntaktischer Form gesucht wird, die eine sprachliche Form dieses Prädikats darstellen.

Inhaltsverzeichnis

Vorbemerkung

Dieses Dokument ist mit dem Textformatierungswerkzeug `noweb` von Norman Ramsay erstellt, einem Hilfsmittel zum „kultivierten Programmieren“ (literate programming). Es erlaubt, Erklärungen und Programmtext in einem Dokument zu erstellen und die Quellprogramme automatisch aus diesem Dokument zu extrahieren.

Programme werden aus mehreren, mit einer ⟨Marke⟩ benannten Programmfragmenten zusammengesetzt. Als Marken werden hier normalerweise die Dateinamen benutzt, in die das Programm bei der Extraktion geschrieben wird. Gelegentlich werden sinnvolle Programmteile durch Untermarken, die keine Dateinamen sind, benannt. Sie stehen dann in der Datei, die der nächst übergeordneten, mit einem Dateinamen markierten Einheit entspricht.

Alle Fragmente mit gleicher Marke werden in der Reihenfolge ihres Auftretens in diesem Dokument aneinandergehängt. Untermarken können aber auf Fragmente verweisen, die an ganz anderen Stellen des Dokuments beschrieben werden; von dort werden sie dann an die Bezugsstelle in den Programmtext eingefügt.

Danksagung

Das Programm habe ich im Rahmen des Seminars „Melčuk und Mercury“ im Wintersemester 2002/2003 entwickelt. Die Beispielgrammatik beruht auf einer Grammatik von Jörg Schuster, die ein wenig um Merkmale und ein paar syntaktische Konstruktionen erweitert wurde. Das Suchprogramm verwendet eine Sammlung von Prädikaten und syntaktischen Mustern, in denen diese Prädikate auftreten können, die ebenfalls von Jörg Schuster stammt. Es sucht in Texten, die mit dem CIS-Tagger um Wortarten- und Merkmalsinformation angereichert wurden; solche Texte und Informationen über den Tagger hat Holger Bosk bereitgestellt. Die Seminarteilnehmer Andreas Hauser und Alexander Völtz haben Teile des Programms beigetragen oder verbessert. Ich danke allen Beteiligten für diese Unterstützung.

1 Entwurfsprobleme

Das folgende Programm ist der Programmiersprache Mercury geschrieben. Mercury ist eine neue Programmiersprache, die seit 1995 an der Universität Melbourne entwickelt wird. Sie ist syntaktisch stark an Prolog angelehnt, unterstützt aber neben Relationen auch Funktionen (und höherstufige Funktionen und Prädikate), ist getypt und wird compiliert statt interpretiert. Mercury bekommt man von der Homepage des Mercury-Projekts,

`http : //www.cs.mu.oz.au/research/mercury/index.html`

Das folgende Programm ist mit der einfach zu installierenden Release-Version `mercury-0.10.1` entwickelt worden; von der neuen Version `mercury-0.11.0` gibt es eine „working version", aber noch kein Installationspaket. Es wird hier allerdings zur Erzeugung eines Lexers schon die Bibliotheksdatei `lex` von `mercury-0.11.0` verwendet, die eine etwas einfachere Syntax als die von `mercury-0.10.1` hat. Das Programm sollte problemlos unter `mercury-0.11.0` laufen.

Da Mercury eine getypte Sprache ist, müssen bei der Formulierung einer Grammatik eine ganze Reihe von Typ-, Modus- und Determinismus-Deklarationen erfolgen, die bei der Verwendung von Prolog nicht nötig sind. Das ist zwar zunächst lästig, erhöht aber die Stabilität und Korrektheit des Programms erheblich.

Am wichtigsten ist wohl die Deklaration des Typs `tree` der Syntaxbäume. Wir haben uns hier dazu entschieden, jede syntaktische Kategorie als einen Konstruktor zum Typ `tree` zu verwenden; die Merkmale und die Teilbäume der Konstituenten werden zu Argumenten des Konstruktors:

⟨*Baumdeklaration*⟩≡
```
:- type tree -->
        n(n_merkmale,string) ;
        np(np_merkmale,tree,...,tree) ;
        ... .
```

Der Grund für diese Entscheidung war, daß dann die Syntaxbäume wie Prolog-Terme als Suchmuster verwendet werden können. Da die Konstruktoren eine feste Stelligkeit haben, muß allerdings jeder Ausdruck derselben Kategorie gleich viele direkte Konstituenten haben. Dadurch kann man sich die Bedeutung der einzelnen Argumentstellen leichter merken und in den Mustern auch vorgeben, daß gewisse Konstituenten des gefundenen Ausdrücks leer sein müssen. Durch die Verwendung leerer Konstituenten in den Grammatikregeln ist diese feste Stelligkeit zwar keine Einschränkung, aber doch ungewohnt.

Eine Alternative wäre, eine uniforme Definition wie

⟨*Uniforme Baumdeklaration*⟩≡
```
:- type tree -->
        leaf(lexcat) ; branch(syncat,list(tree)).
```

zu verwenden, die auf den Typen `lexcat` der lexikalischen und `syncat` der syntaktischen Kategorien aufbaut. Der Vorteil hiervon wäre eine bessere Trennung von Lexer/Tokenizer, Grammatik und Suche, und daß man z.B., relativ zu Schreibfunktionen für die Kategorien wie `np(np_merkmale)`, leicht eine uniforme Schreibfunktion für Bäume erzeugen könnte und sie nicht dem Grammatikschreiber abverlangen müßte. Der Nachteil wäre allerdings, daß man in den Suchmustern die „unnötigen" Konstruktoren `leaf` und `branch` verwenden müßte (oder eine Übersetzung von einfacheren Mustern in entsprechend konstruierte Syntaxbäume bräuchte), und daß man im Suchmuster keine „fehlenden" Konstituenten vorgeben könnte.

Eine weitere Entwurfsentscheidung ist, daß die Grammatiken keine lexikalischen Regeln enthalten; es sollen hier ja keine Folgen von Wörtern, sondern Folgen von lexikalisch markierten Wörtern verarbeitet werden. Eine Deklaration der lexikalischen Kategorien entfällt daher; sie werden in den rechten Seiten der Grammatikregel wie konkrete Wörter in üblichen DCG's notiert, z.B. `np --> [det(Artikel)], ap, [n(Nomen)], pp`. Wenn man den Parsergenerator für normale DCG's verwenden will, muß man die Behandlung der lexikalischen Regeln (bzw. der Listen auf rechten Regelseiten) ändern. Natürlich müssen die als lexikalische Kategorien verwendeten Terme zu den Annotierungen des Taggers passen.

2 Eingeschränkte DCG-Grammatiken

2.1 Definition der EDCG-Grammatiken

Der Parsergenerator erwartet eine Definite-Clause-Grammatik, die das DCG-Format einerseits etwas einschränkt (z.B. keine Listen der Länge > 1 auf den rechten Regelseiten erlaubt) und andererseits erweitert, insbesondere die Verwendung der Iteration `*` erlaubt.

Definition 2.1 Eine Grammatik im *eingeschränkten DCG-Format* besteht aus:

(i) einer Moduldeklaration mit folgender Schnittstelle:

⟨*Deklarationsteil*⟩≡
```
:- module grammatik.

:- interface.

:- import_module list.
```

Hier muß ein Datentyp der Syntax- und Musterbäume definiert werden,

⟨*Deklarationsteil*⟩+≡
```
:- type tree -->
        ...
        ; seq(list(tree))
        ; (?) ; (+) ; (-)
        ; arg(int).
```

der

(a) zu jeder syntaktischen und lexikalischen Kategorie der Grammatik einen „passenden“ Datentypkonstruktor hat, d.h. den Mercury-Term der Kategorie, erweitert um Argumente vom Typ `tree` für die direkten Konstituenten[1],

(b) einen Konstruktor `seq(list(tree))` enthält, falls die syntaktischen Konstruktionen `*(A)` oder `seq(A)` benutzt werden,

(c) drei besondere Konstruktoren enthält, die (nur) in den Musterbäumen auftreten: `(?)` für optionale, `*` für obligatorische, und `(-)` für leere Konstituenten der Treffer.

(d) einen besonderen Konstruktor `arg(int)` enthält, der nur in Argumentmustern auftritt, wo mit dem Teilbaum `arg(n)` der entsprechende Teilbaum des Treffers (für die Ausgabe) als `n`-tes Argument des logischen Prädikats markiert wird.

Gegebenenfalls erfordert `tree` eine vorherige Deklaration von Datentypen syntaktischer Merkmalbereiche. Die Deklaration lexikalischer Kategorien (falls vorhanden) hängt vom Lexikon ab, d.h. von den im Tagger verwendeten Kategorien und Wertebereichen. (Beides kann auch durch Importieren eines Moduls mit diesen Deklarationen erfolgen.) Als Abkürzung ist

⟨*Deklarationsteil*⟩+≡
```
:- type trees == list(tree).
```

für die weiteren Deklarationen nötig, insbesondere für die Deklaration von Typ und Modus des Parsers:

⟨*Deklarationsteil*⟩+≡
```
:- pred parse(trees::in,tree::in,
            pred(tree)::((pred(in) is semidet) -> pred(in) is semidet),
            trees::in,trees::out).
```

Zur Ausgabe der gefundenen Analysen müssen Funktionen deklariert werden:

⟨*Deklarationsteil*⟩+≡
```
:- type printlevel -> ...   .

:- pred print_alltrees(printlevel::in,list(trees)::in,
                    io__state::di, io__state::uo) is semidet.
```

(ii) einem Implementationsteil aus internen Deklarationen, den Grammatikregeln und Hilfsprädikaten.

⟨*Implementationsteil*⟩≡
```
:- implementation.
```

[1] Die Anzahl der direkten Konstituenten ergibt sich aus den Grammatikregeln; aber wir setzen voraus, daß bei jeder Kategorie diese Anzahl in allen alternativen Konstruktionen gleich ist. Aus dem Datentyp `tree` könnte man also die folgende Deklaration `:- syncat .[...].` der syntaktischen Kategorien ableiten.

(a) Zur Erzeugung passender Deklarationen für die Parse-Prädikate (vgl. 3.2.4) müssen die syntaktischen Kategorien wie folgt aufgeführt werden:[2]

⟨*Syntaktische Kategorien*⟩≡

```
:- syncat .[<SynCat1>,....,<SynCatm>].
```

Für jedes dieser `<SynCat>(<Merkmaltypen>)` muß in der Definition des Typs `tree` ein Baumkonstruktor `<SynCat>(<Merkmaltypen>,tree,...,tree)` auftreten.

(b) Zur Berechnung der Rekursionstiefe braucht man etwas Arithmetik, daher:

⟨*Rekursionstiefe*⟩≡

```
:- import_module int.
```

(c) Grammatikregeln im Format der Definite-Clause-Grammars:

⟨*Regel*⟩≡

```
<Kategorie> --> <Konstruktionsmöglichkeiten>.
```

mit folgenden Besonderheiten:

i. Auf der linken Regelseite sind Kategorien folgender Art erlaubt:

⟨*Kategorie*⟩≡

```
Cat ::= <atom> | <atom>(Term,...,Term).
```

Jede Kategorie muß in allen Alternativen ihrer rechten Regelseite dieselbe Anzahl direkter Konstituenten haben, sodaß dazu nur *ein* Mercury-Prädikat deklariert zu werden braucht.[3]

ii. Auf der rechten Regelseite sind folgende Konstruktionsmöglichkeiten erlaubt:[4]

⟨*Konstruktionen*⟩≡

```
A,B ::= Cat | (A,B) | (A;B)  | {A} | [] | [Term]
            | ?(A) | seq(A) | *(A) | :(A).
```

Mit den einstelligen Operatoren[5] `?`, `seq` bzw. `*`, und `:` sollen optionale Konstituenten, nicht-leere bzw. möglicherweise leere Folgen von Konstituenten, und eine Kopfkonstituente markiert werden.

(d) Deklarationen, Hilfsprädikate und Funktionen für folgende Zwecke:

i. Zur Behandlung von `?(A)`, `*(A)`, und `[]`:

⟨*Leere Konstituenten*⟩≡

```
:- pred empty(tree::out,tree::in) is semidet.
empty(-,-).
empty(-,?).
```

---

[2] Da Mercury keine vom Benutzer eingeführten Deklarationen zuläßt, wird diese etwas seltsame Syntax verwendet; ebensogut kann man `:- [syncat, <SynCat1>,...,<SynCatn>].` verwenden.

[3] Das dient hauptsächlich der Vereinfachung, scheint mir aber praktisch notwendig, um die Ergänzung der Baumausgabe und Musterbaum-Eingabe zu vereinfachen und um sich die Stelligkeit merken zu können.

[4] In den Syntax- und Musterbäumen wird zur kompakteren Darstellung statt `[]` stets `-` verwendet.

[5] Eine Suffixschreibweise von Operatoren kann man in Mercury anscheinend nicht festlegen.

ii. Zur Behandlung der Iteration `seq(A)` brauchen wir (vgl. Abschnitt 3.2.7) die Deklaration eines Parse-Prädikats

⟨*Iteration*⟩≡
```
  :- pred seq(int::in,tree::out,tree::in,int::in,
              trees::in,trees::out) is nondet.
```

Die Eingabenummer hierin entspricht einer Kategorie `A`, die auf einer rechten Regelseite als `*(A)` oder `seq(A)` vorkommt.

iii. Zur Suche des lexikalischen Kopfs einer Phrase muß eine partielle Funktion definiert werden, die zu jeder Kategorie sagt, in welcher der direkten Konstituenten der lexikalische Kopf zu finden ist:

⟨*Bestimmung des Phrasenkopfs*⟩≡
```
  func head(tree) = tree is semidet.
  head(...) = ... .
```

Die Definition der Funktion könnte aus den Markierungen `:(A)` auf den rechten Seiten der Grammatikregeln gewonnen werden.
Außerdem muß zum Vergleich von lexikalisch markierten Wörtern mit markierten Stammformen im Muster eine partielle Funktion

⟨*Bestimmung des Kerns eines Lexems*⟩≡
```
  func core(tree) = tree is semidet.
  core(...) = ...
```
angegeben werden, die aus einem Lexem wie `n(pl,"houses","house")` den entprechenden Musterbaum `n("house")` macht.

iv. Zur Beschränkung des Parsers auf die Suche nach Analysen vorgegebener Kategorie braucht man einen „Teiltyp“ der syntaktischen Kategorien:

⟨*Typ der Startkategorien*⟩≡
```
  :- type startcat -->
          <Constructor(feature-types)>
          ; ...
```

Aus einer Liste von konkreten Startkategorien

⟨*Pseudodeklaration*⟩≡
```
  :- starts .[<Cat1>,...,<Catm>].
```

die i.a. nur ausgewählte Merkmalwerte enthalten, werden Prädikate erzeugt, die der Parser mit `start/2` aufruft.

v. Zum Aufruf der Analyse braucht man folgendes (im Interface deklariertes) Prädikat:

⟨*Parseraufruf*⟩≡
```
  parse([Leaf|Leaves],PatTree,Filter,Acc,Trees) :-
          ( startcats(StartCats),
              list__member(Cat,StartCats),
```

```
                start(Cat,ParsePred),
                list__length([Leaf|Leaves],InputLength),
                RekDepth = 2 + InputLength,
                ParsePred(Tree,PatTree,RekDepth,[Leaf|Leaves],RestLeaves),
                Filter(Tree)
            -> NewAcc = [Tree|Acc],
                parse(RestLeaves,PatTree,Filter,NewAcc,Trees)
            ; parse(Leaves,PatTree,Filter,Acc,Trees) ).

    parse([],_PatTree,_Filter,Acc,Trees) :- Trees = Acc.
```

vi. Zum Schreiben von gefundenen Phrasen müssen Funktionen definiert werden, die lexikalische und zusammengesetzte Bäume als Strings oder als Bäume ausgeben.

$\langle$*Schreiben von Syntaxbäumen*$\rangle\equiv$

```
print_alltrees(PrintLevel:printlevel,Parses:list(trees),
               In:io__state,Out:io__state) :- ...
```

Hierzu kann man z.B. den Bibliotheksmodul `pprint` importieren und den dort angegebenen Pretty-Printer ausnutzen. (Die DCG-Schreibweise kann hier nicht verwendet werden, da DCG-Regeln noch wir Grammatikregeln expandiert werden.)

Einschränkungen:

(i) Geschachtelte Und-Oder-Verbindungen auf den rechten Regelseiten sind erlaubt, aber keine geschachtelten `*` und `seq`-Verbindungen.

(ii) Die Verzweigungsbreite eines Knotens im Syntaxbaum ist die Anzahl der Konjunktionsglieder in den Disjunktionsgliedern, die bei der disjunktiven Normalform der rechten Regelseite entstünden; sie muß[6] in allen Disjunktionsgliedern gleich sein, was durch Einfügen leerer Konstituenten `[]` erreicht werden kann.

Als optional markierte Konstituenten `?(C)` sowie leere Konstituenten `[]` auf der rechten Regelseite erhöhen die Verzweigungsbreite des jeweiligen Baumknotens um 1.

(iii) Alle Konstruktionsmöglichkeiten für Ausdrücke derselben Kategorie sollten man in *einer* Regel (mit disjunktiver rechter Seite) formulieren, denn sonst wird mehrfach dieselbe Spezialisierungsregel `<Cat>(Baum,?) -> <Cat>(Baum,+)` erzeugt.

(iv) Folgen von Termen, `[Term|Terme]`, mit Länge > 1 sind auf den rechten Regelseiten nicht erlaubt, damit die Verzweigungsbreite einer syntaktischen Konstruktion an der Anzahl der Konjunktionsglieder erkennbar bleibt; statt `[a,b,c]` muß man also `[a],[b],[c]` verwenden.

(v) Konjunktions- und Disjunktionsglieder sollten rechtsgeklammert (d.h. ohne Angabe der Klammern) geschrieben werden. Eine mögliche Linksklammerung ist vermutlich nicht überall berücksichtigt.

[6] Sonst erhält man Kategorien mit gleichem Namen und verschiedener Stelligkeit und braucht entsprechend viele Deklarationen von Typen und Modi; vermutlich kommt es dann zu Typüberladungen und -konflikten.

Disjunktionen auf den rechten Regelseiten sind für die Fälle gedacht, wo in den Syntaxbäumen kein eigener Knoten als Kennzeichnung der „disjunktiven Kategorie“ erscheinen soll, sondern nur die Kategorie der jeweils verwendeten Alternative.[7]

2.2 Beispiel einer EDCG-Grammatik

Wir geben nun ein Beispiel einer solchen Datei `<grammatik>.dcg` und erläutern anschließend, was der Parsergenerator genau tun soll.

Beispiel 2.2 Zum Testen der Expansionen folgende Beispielgrammatik in der EDCG-Notation benutzt:

$\langle$*grammatik.dcg*$\rangle\equiv$
```
:- module grammatik.

:- interface.

:- import_module list.

:- type featureEN   --> sg ; gen.
:- type featureN    --> sg ; pl ; sp ; gen ; num.
:- type featureAdj  --> b ; c ; s ; crd ; ord ; prd.
:- type featureAdv  --> b ; c ; s ; part ; wh.
:- type featureV    --> inf ; pres ; past ; ger ; part ; prsg. %  ; mdl ; aux.
:- type featurePron --> idef ; pers ; refl ; wh ; rel ; poss ; def.
:- type featureDet  --> wh ; rel ; poss ; def. % def ergaenzt
:- type featureConj --> sub ; coo.

:- type featureNP --> poss ; def ; wh.

:- type tree -->
        %
        % Lexikalische Baumkonstruktoren: (Basisbäume vom Tagger)
          n(featureN,string,string)
        ; en(featureEN,string,string)
        ; adj(featureAdj,string,string)
        ; v(featureV,string,string)
        ; aux(featureV,string,string)
        ; mdl(string,string)
        ; adv(featureAdv,string)
        ; pron(featurePron,string)
        ; det(featureDet,string)
        ; cnj(featureConj,string)
```

[7] Funktionale Kategorien wie `det` vermengen den Begriff der syntaktischen Konstruktion mit dem der syntaktischen Funktion einer Konstituente; stattdessen sollte man die Alternative der Konstruktionsmöglichkeiten angeben, etwa `(art ; pers_poss ; np_gen)`.

```
        ; prep(string)
        ; infp(string)
        ; intj(string)
        ; pfx(string)
        ; sfx(string)
        ; pun(string)
        ; pul(string)
        ; pur(string)
        ; puq(string)
        ; tagerr(string,string) % Für Fehlermeldung bei Taggerfehlern
        %
        % Syntaktische Baumkonstruktoren:
        ; s(tree,tree,tree,tree,tree)
        ; np(featureNP,tree,tree,tree,tree,tree)
        ; detp(tree,tree,tree)
        ; advp(tree,tree,tree)
        ; ap(tree,tree,tree)
        ; pp(tree,tree)
        ; inp(tree,tree,tree,tree,tree)
        %
        % Hilfskonstruktor für Iteration:
        ; seq(list(tree))
        %
        % Konstruktoren nur für Musterbäume:
        ; (-)                 % leere Konstituente
        ; (+)                 % dummy: nicht-leere Konstituente
        ; (?)                 % leere oder nicht-leere Konstituente
        % Zum Matchen mit vorgegebenem Stamm:
        ; n(string)
        ; v(string)
        ; aux(string)
        ; mdl(string)
        ; adj(string)
        ; en(string)          % Zum Matchen (wegen Genitiv-Formen)
        ; det(string)         % Nur wegen Tagger-Fehlern:
        ; cnj(string)
        % Nur für Argumentbäume:
        ; arg(int)            % Konstituente entspricht n-tem log. Argument
        .

:- type trees == list(tree).

:- pred parse(trees::in,tree::in,
            pred(tree)::in(pred(in) is semidet),
            trees::in,trees::out) is nondet.

:- func head(tree) = tree is semidet.
```

```
:- func leaves(tree) = trees is det.

:- use_module io.
:- pred usage_pattern(io__state::di, io__state::uo) is det.

:- type printlevel --> flach ; tief.
:- pred printer(string::in,printlevel::out) is semidet.
:- pred print_alltrees(printlevel::in, list(trees)::in, io__state::di, io__state::uo)
        is det.

:- module muster.

:- interface.

:- type pseudo_query --> {tree, list(string), tree}.
:- type pseudo_queries == list(pseudo_query).

:- type query_code --> {string,list(int),list(string)}.
:- func dispatch(query_code) = pseudo_queries.

:- end_module muster.

:- implementation.

:- import_module int.

% Deklaration der Parseprädikate:

:- syncat .[s,np(featureNP),detp,advp,ap,pp,inp].

:- pred seq(int::in, tree::out, tree::in, int::in, trees::in, trees::out)
        is nondet.

% Grammatikregeln:

s -> (  np(def),
          (  [mdl(Vollform,Stamm)],
               :([v(inf,VVoll,VStamm)])
          ;  [],
               :([v(FeatV,Vollform,Stamm)])
          ;  [aux(FeatA,VollA,StammA)],
               :([v(part,VVoll,VStamm)] ; [v(inf,VVoll,VStamm)])
          ),
          ?(np(def)),
          *(pp)
      ).
```

```
np(poss) -> ( [det(def,Det)],
                 ?(ap),
                 *([n(sg,Vs,Sts)]),
                 :([n(gen,Voll,Stamm)]),
                 ?(pp)
             ; ?([det(def,Det)]),
                 ?(ap),
                 *([en(Fs,Vs,Sts)]),
                 :([en(gen,Voll,Stamm)]),
                 []
             ; [],
                 [],
                 [],
                 [det(poss,Det)],
                 []
             ).

np(def) -> (   ( ([det(def,Det)], { F = sg ; F = sp })
                ; (?([det(def,Det)]), { F = pl })
                ; (np(poss), { (F = sg ; F = sp ; F = pl) })
                ),
                ?(ap),
                (  *([n(sg,Vs,Ss)]),
                    :([n(F,Voll,Stamm)]), { (F = sg ; F = sp ; F = pl) }
                ; [],
                    :([en(sg,Voll,Stamm)])
                ),
                ?(pp)
             ;
                [],
                [],
                (  [],
                    :([pron(F,Pron)]),
                    { F = def ; F = idef ; F = pers }
                ;  *([en(Fs,Vs,Sts)]),
                    [en(F,Voll,Stamm)]
                ),
                []
            ).

detp -> ( :([det(FeatDet,Det)]), [], []
         ; ?([det(FeatDet,Det)]), ?(ap), :([n(gen,Voll,Stamm)])
         ; [], [], :([en(gen,Voll,Stamm)])
         ).

pp -> [prep(Prep)], :(np(def)).
```

```
advp -> (  *([adv(Fs,Advs)]),
             :([adv(FeatAdv,Adv)]),
             []
         ;  ?([adv(b,Adv)]),
             :([adv(c,AdvC)]),
             pp
         ).

ap -> (  ?(advp),
           *([adj(Fs,Vs,Ss)]),
           :([adj(FeatAdj,VollAdj,StammAdj)])
       ;  ap,
           [cnj(coo,Cnj)],
           ap
       ).

inp -> ( ?(advp), :([v(inf,VollV,StammV)]), [], ?(advp), ?(np(def) ; ap ; pp ; advp)
        ;      [], :([v(inf,VollV,StammV)]), ap, ?(advp), ?(np(def) ; pp)
        ).

% Hilfsprädikate:

:- pred empty(tree::out,tree::in) is semidet.
empty(-,-).
empty(-,?).

% Kopfpraedikat:

% Bei komplexen Phrasen ist es der Kopf der Hauptkonstituenten:
head(s(_,_,V,_,_)) = head(V).
head(np(_, _,_,_,N,_)) = head(N).
head(detp(D,_,_)) = head(D).
head(pp(_,N)) = head(N).
head(advp(_,Adv,_)) = head(Adv).
head(ap(_,_,A)) = head(A).
head(inp(_,V,_,_,_)) = head(V).

% Stamm- und Vollform sollte man separat aus dem Kopfelement entnehmen (filter.m)
head(n(F,V,S)) = n(F,V,S).
head(en(F,V,S)) = en(F,V,S).
head(adj(F,V,S)) = adj(F,V,S).
head(v(F,V,S)) = v(F,V,S).
head(aux(F,V,S)) = aux(F,V,S).
head(mdl(V,S)) = mdl(V,S).
head(adv(F,S)) = adv(F,S).
head(pron(F,S)) = pron(F,S).
head(det(F,D)) = det(F,D).
```

```
head(cnj(F,S)) = cnj(F,S).
head(prep(P)) = prep(P).
head(infp(P)) = infp(P).

% Für den Vergleich mit Musterbäumen:
:- func core(tree) = tree is semidet.

core(n(_F,_V,S)) = n(S).
core(en(_F,_V,S)) = en(S).
core(adj(_F,_V,S)) = adj(S).
core(v(_F,_V,S)) = v(S).
core(aux(_F,_V,S)) = aux(S).
core(mdl(_V,S)) = mdl(S).
core(det(_F,D)) = det(D).
core(cnj(_F,S)) = cnj(S).

% Parser (ggf. als separate Datei)

:- type startcat --> s ; np(featureNP) ; ap ; pp ; advp.

:- starts .[s,np(def),np(poss),pp,ap,advp].

parse([Leaf|Leaves],PatTree,Filter,Acc,Trees) :-
        ( startcats(StartCats),
            list__member(Cat,StartCats),
            start(Cat,ParsePred),
            list__length([Leaf|Leaves],InputLength),
            RekDepth = 2 + InputLength,
            ParsePred(Tree,PatTree,RekDepth,[Leaf|Leaves],RestLeaves),
            Filter(Tree)
        -> NewAcc = [Tree|Acc],
            parse(RestLeaves,PatTree,Filter,NewAcc,Trees)
        ; parse(Leaves,PatTree,Filter,Acc,Trees) ).

parse([],_PatTree,_Filter,Acc,Trees) :- Trees = Acc.
```

Das folgende Prädikat wird vom Suchprogramm zur Anzeige einer Bedienhilfe gebraucht und muß vom Grammatikschreiber definiert werden:

⟨*grammatik.dcg*⟩+≡

```
usage_pattern(In,Out) :-
        io__stderr_stream(StdErr,In,Out1),
        io__write_strings(StdErr, ["Patterns are: (and some more)
       Var ::= ? | + | -
        P  ::= Var | prep(<string>)
       Det ::= Var | det(<string>) | det(<wh|rel|poss>,<string>)
                   | np(poss,Det,AP,Ns,N,PP)
```

```
  V   ::= Var | v(<string>)   | v(<grnd|prsg|past|part|inf|ger|mdl|aux>,\"<surface>\",\"<stem>\")
  N   ::= Var | n(<string>)   | n(<sg|pl|sp|gen|num>,\"<surface>\",\"<stem>\")
  EN  ::= Var | en(<string>)  | en(<sg|gen>,\"<surface>\",\"<stem>\")
  A   ::= Var | adj(<string>) | adj(<b|c|s|crd|ord|pred>,\"<surface>\",\"<base>\")
 Adv  ::= Var | adv(<b|c|s|part|wh>,<string>)
  NP  ::= Var | np(Feat,Det,AP,Ns,N,PP) | EN
  AP  ::= Var | ap(Advs,As,A)
  PP  ::= Var | pp(P,NP)
  S   ::= Var | s(NP,AuxModl,V,NP,PPs)
  Xs  ::= Var | X, for X = A | N | Adv | PP. \n"],Out1,Out2),
  io__set_exit_status(1,Out2,Out).
```

Die Blattliste eines Baums braucht man später, um das Vorkommen von Wörtern in den erkannten Phrasen testen zu können:

⟨*grammatik.dcg*⟩+≡

```
leaves(Baum) = lexemliste(Baum,[]).

:- func lexemliste(tree,trees) = trees is det.

lexemliste(n(F,V,S), Acc) = [n(F,V,S)|Acc].
lexemliste(en(F,V,S),Acc) = [en(F,V,S)|Acc].
lexemliste(adj(F,V,S), Acc) = [adj(F,V,S)|Acc].
lexemliste(v(F,V,S), Acc) = [v(F,V,S)|Acc].
lexemliste(aux(F,V,S), Acc) = [aux(F,V,S)|Acc].
lexemliste(mdl(V,S), Acc) = [mdl(V,S)|Acc].
lexemliste(adv(F,S), Acc) = [adv(F,S)|Acc].
lexemliste(pron(F,S), Acc) = [pron(F,S)|Acc].
lexemliste(det(F,S), Acc) = [det(F,S)|Acc].
lexemliste(cnj(F,S), Acc) = [cnj(F,S)|Acc].
lexemliste(prep(S), Acc) = [prep(S)|Acc].
lexemliste(infp(S), Acc) = [infp(S)|Acc].
lexemliste(intj(S), Acc) = [intj(S)|Acc].
lexemliste(pfx(S), Acc) = [pfx(S)|Acc].
lexemliste(sfx(S), Acc) = [sfx(S)|Acc].
lexemliste(pun(_), Acc) = Acc.
lexemliste(pul(_), Acc) = Acc.
lexemliste(pur(_), Acc) = Acc.
lexemliste(puq(S), Acc) = [puq(S)|Acc].

lexemliste(n(S),Acc) = [n(S)|Acc].
lexemliste(v(S),Acc) = [v(S)|Acc].
lexemliste(aux(S),Acc) = [aux(S)|Acc].
lexemliste(mdl(S),Acc) = [mdl(S)|Acc].
lexemliste(adj(S),Acc) = [adj(S)|Acc].
lexemliste(en(S),Acc) = [en(S)|Acc].
lexemliste(det(S),Acc) = [det(S)|Acc].
lexemliste(cnj(S),Acc) = [cnj(S)|Acc].
```

Bei den zusammengesetzten Bäumen werden die Lexeme der Teilbäume aufgesammelt:[8]

⟨*grammatik.dcg*⟩+≡

```
lexemliste(s(NP,AuxMod,V,NP2,PPs),Acc) = lexemlisten([NP,AuxMod,V,NP2,PPs],Acc).
lexemliste(np(_FeatureNP,Det,AP,Ns,N,PP),Acc) = lexemlisten([Det,AP,Ns,N,PP],Acc).
lexemliste(detp(Det,AP,Ngen),Acc) = lexemlisten([Det,AP,Ngen],Acc).
lexemliste(advp(AdvP,Adv,PP),Acc) = lexemlisten([AdvP,Adv,PP],Acc).
lexemliste(ap(AdvP,Adjs,A),Acc) = lexemlisten([AdvP,Adjs,A],Acc).
lexemliste(pp(Prep,NP),Acc) = lexemlisten([Prep,NP],Acc).
lexemliste(inp(AdvP,Vinf,AP,AdvP2,NP),Acc) =
    lexemlisten([AdvP,Vinf,AP,AdvP2,NP],Acc).
lexemliste(seq(Trees), Acc) = lexemlisten(Trees,Acc).
```

Bei den Mustervariablen wird der Akkumulator natürlich nicht erweitert:

⟨*grammatik.dcg*⟩+≡

```
lexemliste((-),Acc) = Acc.
lexemliste((+),Acc) = Acc.
lexemliste((?),Acc) = Acc.
lexemliste(tagerr(_S,_T),Acc) = Acc.
lexemliste(arg(_N),Acc) = Acc.

:- func lexemlisten(trees,trees) = trees is det.

lexemlisten([],Acc) = Acc.
lexemlisten([T|Ts],Acc) = lexemliste(T,lexemlisten(Ts,Acc)).
```

Das waren die eigentlichen Funktionen zur Inspektion von Bäumen. Man braucht noch Funktionen zum Drucken von Bäumen.

Der Grammatikschreiber muß festlegen, wie die Bäume je nach `PrintLevel` ausgegeben werden sollen. Wir geben zwei sehr einfache Möglichkeiten an, `flach` und `tief`; weitere Ausgabeformate für Bäume, z.B. `html`, `dot`, `treetex`, könnten je nach vorhandenem Graphikprogramm ergänzt werden.[9] Den ersten Teil sollte man nicht vom Grammatikschreiber erwarten (die DCG-Regeln kann man nicht benutzen, solange alles durch `expand_terms` geht!):

⟨*grammatik.dcg*⟩+≡

```
:- import_module pprint.

printer("flach",flach).
printer("tief",tief).

:- pred print_trees(printlevel::in, trees::in, io__state::di, io__state::uo) is det.

print_trees(PrintLevel,[Tree|Trees],In,Out) :-
        pprint__write(50,tree_to_doc(PrintLevel,Tree),In,I),
```

[8]Das könnte man vielleicht aus der Pseudodeklartation `:- syncat .[...].` erzeugen.

[9]Mit `printer/2` werden die Konstruktoren benannt, damit man sie als Optionen verwenden kann.

```
        io__nl(I,J),
        print_trees(PrintLevel,Trees,J,Out).
print_trees(_,[],In,Out) :- Out = In.

:- pred print_arg_trees(printlevel::in, tree::in, trees::in,
                        io__state::di, io__state::uo) is det.

print_arg_trees(PrintLevel,ArgTree,[Tree|Trees],In,Out) :-
        pprint__write(50,marked_tree_to_doc(PrintLevel,Tree,ArgTree),In,I),
        io__nl(I,J),
        print_arg_trees(PrintLevel,ArgTree,Trees,J,Out).
print_arg_trees(_,_ArgTree,[],In,Out) :- Out = In.

print_alltrees(PrintLevel,[Ts|Rest],In,Out) :-
        (Ts = [ArgTree|Trees]
        -> (print_arg_trees(PrintLevel,ArgTree,Trees,In,J),
            %   print_trees(PrintLevel,Trees,In,J),
               io__nl(J,K),
               print_alltrees(PrintLevel,Rest,K,Out)
           )
        ; io__write_string("\nFehler in print_alltrees\n",In,J),
            print_alltrees(PrintLevel,Rest,J,Out)).
print_alltrees(_PrintLevel,[],In,Out) :- In = Out.
```

Bem.: Man sollte eine generische Lösung automatisch erzeugen[10].

⟨*grammatik.dcg*⟩+≡

```
% ----- Transform a tree to a doc for pretty-printing ------
:- func tree_to_doc(printlevel,tree) = doc is det.
:- func trees_to_doc(printlevel,trees) = doc is det.

% Flache Ausgabe des Baums (nur die Blätter):

tree_to_doc(flach,n(_,String,_)) = text(String).
tree_to_doc(flach,en(_,String,_)) = text(String).
tree_to_doc(flach,det(_,String)) = text(String).
tree_to_doc(flach,det(String)) = text(String).
tree_to_doc(flach,v(_,String,_)) = text(String).
tree_to_doc(flach,aux(_,String,_)) = text(String).
tree_to_doc(flach,mdl(String,_)) = text(String).
tree_to_doc(flach,adv(_F,String)) = text(String).
tree_to_doc(flach,adj(_F,String,_)) = text(String).
tree_to_doc(flach,cnj(_F,String)) = text(String).
```

[10]Dazu wäre es gut, die Anzahl der Merkmalargumente zu kennen; am besten sollte man vielleicht doch vom Grammatikschreiber einen Typ `syncat` (mit Merkmaltypen) verlangen und daraus `tree` systematisch konstruieren, sowie die Projektion von Baumknoten auf die entsprechenden Kategorien.

```
tree_to_doc(flach,cnj(String)) = text(String).
tree_to_doc(flach,infp(String)) = text(String).
tree_to_doc(flach,intj(String)) = text(String).
tree_to_doc(flach,pfx(String)) = text(String).
tree_to_doc(flach,sfx(String)) = text(String).
tree_to_doc(flach,pun(String)) = text(String).
tree_to_doc(flach,pul(String)) = text(String).
tree_to_doc(flach,pur(String)) = text(String).
tree_to_doc(flach,puq(String)) = text(String).
tree_to_doc(flach,prep(String)) = text(String).
tree_to_doc(flach,pron(_,String)) = text(String).

tree_to_doc(flach,s(Tree1,Tree2,Tree3,Tree4,Tree5)) =
   trees_to_doc(flach,[Tree1,Tree2,Tree3,Tree4,Tree5]).
tree_to_doc(flach,np(_FeatureNP,Tree1,Tree2,Tree3,Tree4,Tree5)) =
   trees_to_doc(flach,[Tree1,Tree2,Tree3,Tree4,Tree5]).
tree_to_doc(flach,detp(Tree1,Tree2,Tree3)) =
   trees_to_doc(flach,[Tree1,Tree2,Tree3]).
tree_to_doc(flach,pp(Tree1,Tree2)) =
   trees_to_doc(flach,[Tree1,Tree2]).
tree_to_doc(flach,advp(Tree1,Tree2,Tree3)) =
   trees_to_doc(flach,[Tree1,Tree2,Tree3]).
tree_to_doc(flach,ap(Tree1,Tree2,Tree3)) =
   trees_to_doc(flach,[Tree1,Tree2,Tree3]).

tree_to_doc(flach,inp(Tree1,Tree2,Tree3,Tree4,Tree5)) =
   trees_to_doc(flach,[Tree1,Tree2,Tree3,Tree4,Tree5]).

tree_to_doc(flach,seq(Trees)) = trees_to_doc(flach,Trees).

% Nur für Lesefehler und Musterbäume:

tree_to_doc(flach,tagerr(String1,String2)) =
   text("Tag Error: ") `<>` text(String1) `<>` text(" : ") `<>` text(String2).
tree_to_doc(flach,n(Stamm)) = text("Fehler: N ") `<>` text(Stamm).
tree_to_doc(flach,v(Stamm)) = text("Fehler: V ") `<>` text(Stamm).
tree_to_doc(flach,aux(Stamm)) = text("Fehler: AUX ") `<>` text(Stamm).
tree_to_doc(flach,mdl(Stamm)) = text("Fehler: MDL ") `<>` text(Stamm).
tree_to_doc(flach,adj(Stamm)) = text("Fehler: ADj ") `<>` text(Stamm).
tree_to_doc(flach,en(Stamm)) = text("Fehler: EN ") `<>` text(Stamm).
tree_to_doc(flach,(?)) = text("?").
tree_to_doc(flach,(+)) = text("+").
tree_to_doc(flach,(-)) = text("-").
tree_to_doc(flach,arg(_)) = text("").

% Für Baumlisten:
```

```
trees_to_doc(flach,[Tree1,Tree2|Trees]) =
   (Tree1 = (-)
   -> trees_to_doc(flach,[Tree2|Trees])
   ; Tree2 = (-)
   -> trees_to_doc(flach,[Tree1|Trees])
   ; tree_to_doc(flach,Tree1) `<>` text(" ") `<>` trees_to_doc(flach,[Tree2|Trees])).
trees_to_doc(flach,[Tree]) =
   (Tree = (-)
   -> text("")
   ; tree_to_doc(flach,Tree)).
trees_to_doc(flach,[]) = text("").

% Baumausgabe mit interner Struktur:

tree_to_doc(tief,n(_,String,_))     = text("N ") `<>` text(String).
tree_to_doc(tief,en(_,String,_))    = text("EN ") `<>` text(String).
tree_to_doc(tief,det(_,String))     = text("DET ") `<>` text(String).
tree_to_doc(tief,det(String))       = text("Det ") `<>` text(String).
tree_to_doc(tief,v(_,String,_))     = text("V ") `<>` text(String).
tree_to_doc(tief,aux(_,String,_))   = text("AUX ") `<>` text(String).
tree_to_doc(tief,mdl(String,_))     = text("MDL ") `<>` text(String).
tree_to_doc(tief,adv(_F,String))    = text("ADV ") `<>` text(String).
tree_to_doc(tief,adj(_F,String,_))  = text("ADJ ") `<>` text(String).
tree_to_doc(tief,cnj(_F,String))    = text("CNJ ") `<>` text(String).
tree_to_doc(tief,cnj(String))       = text("CNJ ") `<>` text(String).
tree_to_doc(tief,infp(String))      = text("INFP ") `<>` text(String).
tree_to_doc(tief,intj(String))      = text("INTJ ") `<>` text(String).
tree_to_doc(tief,pfx(String))       = text("PFX ") `<>` text(String).
tree_to_doc(tief,sfx(String))       = text("SFX ") `<>` text(String).
tree_to_doc(tief,pun(String))       = text(String).
tree_to_doc(tief,pul(String))       = text(String).
tree_to_doc(tief,pur(String))       = text(String).
tree_to_doc(tief,puq(String))       = text(String).
tree_to_doc(tief,prep(String))      = text("PREP ") `<>` text(String).
tree_to_doc(tief,pron(_,String))    = text("PRON ") `<>` text(String).

tree_to_doc(tief,s(Tree1,Tree2,Tree3,Tree4,Tree5)) =
   text("S ") `<>` nest(2, trees_to_doc(tief,[Tree1,Tree2,Tree3,Tree4,Tree5])).
tree_to_doc(tief,np(_Feature,Tree1,Tree2,Tree3,Tree4,Tree5)) =
   text("NP ") `<>` nest(3,trees_to_doc(tief,[Tree1,Tree2,Tree3,Tree4,Tree5])).
tree_to_doc(tief,detp(Tree1,Tree2,Tree3)) =
   text("DETP ") `<>` nest(5, trees_to_doc(tief,[Tree1,Tree2,Tree3])).
tree_to_doc(tief,pp(Tree1,Tree2)) =
   text("PP ") `<>` nest(3,trees_to_doc(tief,[Tree1,Tree2])).
tree_to_doc(tief,advp(Tree1,Tree2,Tree3)) =
   text("ADVP ") `<>` nest(5, trees_to_doc(tief,[Tree1,Tree2,Tree3])).
tree_to_doc(tief,ap(Tree1,Tree2,Tree3)) =
```

```
   text("AP ") `<>` nest(3, trees_to_doc(tief,[Tree1,Tree2,Tree3])).
tree_to_doc(tief,seq(Trees)) =
   text("SEQ ") `<>` nest(4, trees_to_doc(tief,Trees)).
tree_to_doc(tief,inp(Tree1,Tree2,Tree3,Tree4,Tree5)) =
   text("INP ") `<>` nest(4, trees_to_doc(tief,[Tree1,Tree2,Tree3,Tree4,Tree5])).

% Nur für Lesefehler und Musterbäume:

tree_to_doc(tief,tagerr(String1,String2)) =
   text("Tag Error: ") `<>` text(String1) `<>` text(" : ") `<>` text(String2).
tree_to_doc(tief,v(Stamm)) = text("Fehler: V ") `<>` text(Stamm).
tree_to_doc(tief,aux(Stamm)) = text("Fehler: AUX ") `<>` text(Stamm).
tree_to_doc(tief,mdl(Stamm)) = text("Fehler: MDL ") `<>` text(Stamm).
tree_to_doc(tief,n(Stamm)) = text("Fehler: N ") `<>` text(Stamm).
tree_to_doc(tief,en(Stamm)) = text("Fehler: EN ") `<>` text(Stamm).
tree_to_doc(tief,adj(Stamm)) = text("Fehler: ADj ") `<>` text(Stamm).
tree_to_doc(tief,(-)) = text("-").
tree_to_doc(tief,(+)) = text("+").
tree_to_doc(tief,(?)) = text("?").
tree_to_doc(tief,arg(_)) = text("").

% Liste von Bäumen:

trees_to_doc(tief,[Tree1,Tree2|Trees]) =
   (Tree1 = (-)
   -> trees_to_doc(tief,[Tree2|Trees])
   ; Tree2 = (-)
   -> trees_to_doc(tief,[Tree1|Trees])
   ; tree_to_doc(tief,Tree1) `<>` line `<>` trees_to_doc(tief,[Tree2|Trees])).
trees_to_doc(tief,[Tree]) =
   (Tree = (-)
   -> text("")
   ; tree_to_doc(tief,Tree)).
trees_to_doc(tief,[]) = text("").

% Drucken mit markierten Teilbäumen:

:- use_module string.
:- func marked_tree_to_doc(printlevel,tree,tree) = doc is det.
:- func marked_trees_to_doc(printlevel,trees,trees) = doc is det.

marked_tree_to_doc(tief,Tree,ArgTree) =
   ( ArgTree = arg(N) ->
       (text("[") `<>` text(string__int_to_string(N)) `<>` text(" ") `<>`
       tree_to_doc(tief, Tree)
       `<>` text(" ") `<>` text(string__int_to_string(N)) `<>` text("]"))
   ; (ArgTree = (+) ; ArgTree = (-) ; ArgTree = (?)) ->
```

```
      tree_to_doc(tief,Tree)
   ; (ArgTree = s(ArgT1,ArgT2,ArgT3,ArgT4,ArgT5),
      Tree = s(Tree1,Tree2,Tree3,Tree4,Tree5)) ->
      (text("S ") `<>` nest(2, marked_trees_to_doc(tief,
                                             [Tree1,Tree2,Tree3,Tree4,Tree5],
                                             [ArgT1,ArgT2,ArgT3,ArgT4,ArgT5])))
   ; (ArgTree = np(F,ArgT1,ArgT2,ArgT3,ArgT4,ArgT5),
      Tree = np(F,Tree1,Tree2,Tree3,Tree4,Tree5)) ->
      (text("NP ") `<>` nest(3,marked_trees_to_doc(tief,
                                             [Tree1,Tree2,Tree3,Tree4,Tree5],
                                             [ArgT1,ArgT2,ArgT3,ArgT4,ArgT5])))
   ; (ArgTree = detp(ArgT1,ArgT2,ArgT3),
      Tree = detp(Tree1,Tree2,Tree3)) ->
      (text("DETP ") `<>`
       nest(5, marked_trees_to_doc(tief,[Tree1,Tree2,Tree3],[ArgT1,ArgT2,ArgT3])))
   ; (ArgTree = pp(ArgT1,ArgT2),
      Tree = pp(Tree1,Tree2)) ->
      (text("PP ") `<>`
       nest(3,marked_trees_to_doc(tief,[Tree1,Tree2],[ArgT1,ArgT2])))
   ; (ArgTree = advp(ArgT1,ArgT2,ArgT3),
      Tree = advp(Tree1,Tree2,Tree3)) ->
      (text("ADVP ") `<>`
       nest(5, marked_trees_to_doc(tief,[Tree1,Tree2,Tree3],[ArgT1,ArgT2,ArgT3])))
   ; (ArgTree = ap(ArgT1,ArgT2,ArgT3),
      Tree = ap(Tree1,Tree2,Tree3)) ->
      (text("AP ") `<>`
       nest(3, marked_trees_to_doc(tief,[Tree1,Tree2,Tree3],[ArgT1,ArgT2,ArgT3])))
   ; (ArgTree = inp(ArgT1,ArgT2,ArgT3,ArgT4,ArgT5),
      Tree = inp(Tree1,Tree2,Tree3,Tree4,Tree5)) ->
      (text("INP ") `<>`
       nest(4, marked_trees_to_doc(tief,[Tree1,Tree2,Tree3,Tree4,Tree5],
                              [ArgT1,ArgT2,ArgT3,ArgT4,ArgT5])))
   ; (ArgTree = seq(_Args),
      Tree = seq(Trees)) ->
      (text("SEQ ") `<>` nest(4, trees_to_doc(tief,Trees)))
   ; text("")
   ).                            % ?

% Liste markierter Bäume:

marked_trees_to_doc(tief,[Tree1,Tree2|Trees],Args) =
   (Args = [ArgT1,ArgT2|ArgTs]
   -> (Tree1 = (-)
      -> marked_trees_to_doc(tief,[Tree2|Trees],[ArgT2|ArgTs])
      ; Tree2 = (-)
      -> marked_trees_to_doc(tief,[Tree1|Trees],[ArgT1|ArgTs])
      ; marked_tree_to_doc(tief,Tree1,ArgT1) `<>` line
```

```
        `<>` marked_trees_to_doc(tief,[Tree2|Trees],[ArgT2|ArgTs])
      )
   ; text("")).

marked_trees_to_doc(tief,[Tree],Arg) =
   (Arg = [ArgTree]
   -> (Tree = (-)
      -> text("")
      ; marked_tree_to_doc(tief,Tree,ArgTree))
   ; text("")).
marked_trees_to_doc(tief,[],_) = text("").

marked_tree_to_doc(flach,Tree,_ArgTree) = tree_to_doc(flach,Tree).
marked_trees_to_doc(flach,Tree,_ArgTree) = trees_to_doc(flach,Tree).
```

Der Submodul `muster` definiert, wie die aus der Datei `praedikate.lx` der logischen Prädikate stammenden syntaktischen Muster (genauer gesagt, ihre Beschreibungen der Form `{Mustername,Argumentnummern,Stammformen}`) in Pseudo-Anfragen übersetzt werden. Diese haben die Form `{Musterbaum,Filterbeschreibung,Argumentbaum}`, worin später die Filterbeschreibung in ein Filterprädikat übersetzt wird.[11]

Im folgenden Submodul `muster` wird keine Filterbeschreibung angegeben und daher später nur der triviale Filter `filter:id_filter` benutzt.

⟨*grammatik.dcg*⟩+≡

```
:- module muster.

:- implementation.

dispatch({F,Ints,Strgs}) = Val :-
        ( F = "np_prep_np"
        -> Val = np_prep_np(Ints,Strgs)
        ; (F = "possnp_anp" ; F = "npposs_anp")
        -> Val = possnp_anp(Ints,Strgs)
        ; F = "np_vp_np"
        -> Val = np_vp_np(Ints,Strgs)
        ; F = "id"
        -> Val = []
        ; Val = [] % throw("undefinined pattern-construction")
        ).

:- func id(list(int),list(string)) = pseudo_queries.

id(_,_) = [].
```

[11]Die Beschreibung ist nur dafür nötig, damit man Pseudo-Anfragen aus einer Datei lesen kann, was mit Prädikaten nicht geht.

```
:- func possnp_anp(list(int),list(string)) = pseudo_queries.

possnp_anp(Ints,Strings) = Result :-
        (Ints = [I,J],Strings = ["-",Stamm]
        -> Result = [{np(def,np(poss,?,?,?,?,?),?,?,n(Stamm),?),[],
                      np(def,arg(I),            ?,?,arg(J)   ,?)}
                    ]
        ; Result = []).

:- func np_prep_np(list(int),list(string)) = pseudo_queries.

np_prep_np(Ints,Strings) = PseudoQueries :-
        (Ints = [I,_,J],Strings = [Stamm,Prep,_])
        -> PseudoQueries = [{np(def,?,?,?,n(Stamm),pp(prep(Prep),+)),[],
                             np(def,?,?,?,arg(I),  pp(?,    arg(J)))},
                            {np(def,?,?,?,n(Stamm),-),[],
                             np(def,?,?,?,arg(I),  -)}]
        ; PseudoQueries = [].

:- func np_vp_np(list(int),list(string)) = pseudo_queries.

np_vp_np(Ints,Strings) = PseudoQueries :-
        (Ints = [I,J,K],Strings = [N1,Verb,N2])
        -> PseudoQueries = [{s(np(def,?,?,?,n(N1),?),
                               ?,v(Verb),
                               np(def,?,?,?,n(N2),?),-),[],
                             s(arg(I),
                               ?,arg(J),arg(K),-)}
                            ]
        ; PseudoQueries = [].

:- end_module muster.

:- end_module grammatik.
```

Eine solche Datei `<grammatik>.dcg` sollte einfacher, z.B. von einem „Linguisten", zu warten sein als die daraus erzeugte Datei von Mercury-Klauseln.

3 Erzeugung des Parsers und der Musterbäume aus einer DCG

In diesem Abschnitt wird ein Übersetzer definiert, der eine Grammatik `<grammatik>.dcg` in einen Mercury-Modul `grammatik.m` übersetzt. Dafür setzen wir die von Mercury lesbare Notation der Definite-Clause-Grammatiken im EDCG-Format voraus.

Insbesondere sollen die Parseprädikate aus einer vereinfachten Form der Grammatik berechnet werden, wobei Argumente für die Eingabe von Suchbäumen und die Ausgabe von Syntaxbäumen automatisch ergänzt werden.

Bemerkung 3.1 Der Übersetzer sollte so reorganisiert werden, daß man uniforme Teile des erzeugten Parsers nicht in der Eingabegrammatik formulieren muß; sie sollten aus konstanten Dateien in den erzeugten Modul importiert werden. Auserdem sollte der Übersetzer mit Optionen versehen werden, mit denen man die Art der Übersetzung steuern kann: z.B. sollte man optional noch Argumente zur Berechnung von semantischen Werten ergänzen, oder zwischen verschiedenen Behandlungen lexikalischer Kategorien wählen können.

3.1 Überblick über die Übersetzungsschritte

Die Übersetzung einer EDCG-Datei leistet im wesentlichen folgendes (oder soll es leisten, wenn sie abgeschlossen ist):

(i) Die Deklaration `:- syncat .[Cat1,...,Catn].` der syntaktischen Kategorien[12] wird in eine Folge von Deklarationen für Parse-Prädikate `PCat1`, ..., `PCatn` übersetzt. Zur Kategorie `cat(Merkmaltyp)` wird die Deklaration

⟨*Erzeugte Deklaration eines Parse-Prädikats*⟩≡
```
:- pred cat(Merkmaltyp::in,tree::out,tree::in,int::in,
           trees::in,trees::out) is nondet.
```

erzeugt, mit der intendierten Bedeutung:

⟨*Verwendungsweise eines Parseprädikats*⟩≡
```
cat(Merkmal,Ausgabebaum,Musterbaum,Rekursionstiefenschranke,
    Eingabeliste,Ausgabeliste).
```

(ii) Jede EDCG-Grammatikregel

⟨*Grammatikregel*⟩≡
```
cat(Merkmale) --> konstituenten(Merkmale).
```

der Quellgrammatik `<grammatik>.dcg` wird in eine DCG-Regel

⟨*übersetzte Regel*⟩≡
```
cat(Merkmale,Tree,Pat,RekTiefe) -->
        { bedingungen(RekTiefe, RekTiefe'),
          bedingungen(Tree,Pat,Trees,Pats) },
        konstitutenen(Merkmale,Trees,Pats,RekTiefe').
```

der Zieldatei `<grammatik>.m` übersetzt. Dabei werden

(a) durch das zusätzliche Argument `RekTiefe` eine Schranke an die erlaubte Rekursionstiefe übergeben, die (mit den erzeugten `{bedingungen(RekTiefe,RekTiefe')}`) verhindert, daß der Parser bei linksrekursiven Grammatikregeln divergiert,

(b) die Markierungen von optionalen und iterierten Konstituenten durch `?(A)` und `*(A)` in Disjunktionen `([] ; A)` und `([] ; seq(A))` aufgelöst,

[12]Die Darstellung hier setzt voraus, daß lexikalische Kategorien implizit dadurch gekennzeichnet sind, daß sie in den rechten Regelseiten in der Form `[Cat]` vorkommen; zu ihnen brauchen dann keine Parse-Prädikate erzeugt werden.

(c) die Markierung der Kopfkonstituenten durch `:(A)` beseitigt (bis entsprechende Klauseln für `head` daraus erzeugt werden),

(d) Argumente `Tree,Trees` für die Ausgabe der Syntaxbäume einer Phrase und ihrer direkten Teilphrasen eingefügt, sowie Argumente `Pat,Pats` für Musterbäume zur Steuerung des Parsevorgangs,

(e) Anweisungen `{bedingungen(Tree,Pat,Trees,Pats)}` erzeugt, nach denen der Syntaxbaum einer analysierten Phrase aus den Bäumen der direkten Teilphrasen und dem vorgegebenen Muster aufgebaut wird.

Außerdem werden zwei weitere Regeln erzeugt: ein Spezialfall der obigen Regel, wo `Pat` eine Mustervariable `+` ist:

⟨*übersetzte Regel mit unbestimmtem Muster*⟩≡
```
cat(Merkmale,Tree,+,RekTiefe) -->
        { bedingungen(RekTiefe, RekTiefe'),
          bedingungen(Tree,+,Trees,Pats) },
        konstitutenen(Merkmale,Trees,Pats,RekTiefe').
```

und eine

⟨*Musterspezialisierung*⟩≡
```
cat(Merkmale,Tr,?,RekTiefe) -> cat(Merkmale,Tr,+,RekTiefe).
```

Diese Regeln dienen dazu, in den Musterbäumen mit `+` und `?` vorzugeben, ob entsprechende direkte Konstituenten in den Syntaxbäumen vorhanden sein müssen oder nicht; mit `-` kann man sie ausschließen.

(iii) Kategorien `seq(A)` auf den rechten Regelseiten werden mit `n:int` numeriert und durch `seq(n)` ersetzt; zu `seq(n)` werden neue Grammatikregeln erzeugt, mit denen nicht-leere Folgen von Konstituenten der Kategorie `A` zusammengefaßt werden können.

(iv) Die Pseudodeklaration der Startkategorien

⟨*Startkategorien*⟩≡
```
:- startcat .[<Cat>(<Merkmalwerte>), ...].
```

wird in eine Liste der Startkategorien und in die Deklaration einer Hilfstabelle

⟨*Startprädikate*⟩≡
```
:- pred startcats((list(startcat) :: out)) is det.

:- pred start((startcat :: in),
              (pred(tree, tree, int, trees, trees)
              :: out(pred(out, in, in, in, out) is nondet))
             ) is semidet.
```

übersetzt, und in Klauseln, die die einzelnen Zeilen der Hilfstabelle darstellen:

⟨*Zuordung von Parseprädikaten zu den Startkategorien*⟩≡
```
start(<Cat>(<Merkmal>),
      (pred(Tree::out,PatTree::in,RekDepth::in,
```

```
            Leaves::in,RestLeaves::out) is nondet :-
        <Cat>(<Merkmal>, Tree, PatTree, RekDepth, Leaves, RestLeaves))).
```

(v) Die übrigen Deklarationen und Definitionen der Datentypen, Prädikate und Funktionen werden unverändert in die erzeugte Datei übernommen.

Bem.: Die Auflösung von `?(A)` und `*(A)` geschieht während der Adjunktion der Baumargumente. Die Ergänzung der Rekursionstiefe erfolgt anschließend, damit diese nicht im Baum erscheint. Die `RekTiefe` wird durch `seq` an die jeweiligen Argumentkategorien weitergereicht. (Das sollte man jetzt mit `adjoin_rec_depth3` machen können.)

Bemerkung 3.2 Mit etwas Aufwand könnte man wohl auch den Datentyp `tree` aus den Grammatikregeln und den Deklarationen der Merkmalbereiche erzeugen. Es scheint mir besser, daß der Grammatikschreiber sowohl die Datentypen der Merkmalbereiche als auch den Datentyp der Bäume selbst definiert.

(i) Um das Kopfprädikat `head` aus den Markierungen `:(A)` in den Grammatikregeln zu erzeugen, braucht man den Typ `tree`. Sollte man die Kopfmarkierung so lange beibehalten, bis man die Baumform erzeugt hat?

(ii) Damit man ein vollständiges Suchprogramm erzeugen kann, muß auch das Ausdrucken von Syntaxbäumen (bzw. nur der Blätter) in der Eingabegrammatik definiert sein. Welche anderen Funktionen?

3.2 Übersetzung einer EDCG-Grammatik

Wir ergänzen die Datei `mercury-extras/examples/expand_terms.m` (?) um Expansionsregeln, mit denen eine Eingabegrammatik `<Grammatik>.dcg` im EDCG-Format in eine Grammatik der gewünschten Form umgewandelt wird. Diese muß in eine Ausgabedatei mit dem Namen `grammatik.m` geschrieben werden, z.B. durch `expand_dcg <Grammatik>.dcg > grammatik.m` (vgl. den `Makefile`).

3.2.1 Term-Expansion

Zuerst wird der Teil von `expand_terms.m` übernommen, der nicht verändert werden muß:

$\langle expand_dcg.m\rangle\equiv$

```
:- module expand_dcg.

:- interface.
:- import_module io.

:- pred main(io__state::di, io__state::uo) is det.
```

```
:- implementation.
:- import_module string, list, int, term, varset, term_io.
:- import_module std_util.      % for type pair.

main ->
        io__command_line_arguments(Args),
        ( { Args = [] }
        -> term_io__read_term(Result),
            expand_terms(Result,0)
        ;  expand_terms_file_list(Args)
        ).

:- pred expand_terms_file_list(list(string)::in, io__state::di, io__state::uo)
        is det.

expand_terms_file_list([]) -> [].
expand_terms_file_list([File | Files]) ->
        expand_terms_file(File),
        expand_terms_file_list(Files).

:- pred expand_terms_file(string::in, io__state::di, io__state::uo) is det.

expand_terms_file(File) ->
        io__open_input(File, Result),
        ( { Result = ok(Stream) },
                expand_terms_stream(Stream)
        ; { Result = error(Error) },
                io__progname("expand_terms", Progname),
                { io__error_message(Error, Message) },
                io__write_strings([
                        Progname, ": ",
                        "error opening file `", File, "' for input:\n\t",
                        Message, "\n"
                ]),
                io__set_exit_status(1)
        ).

:- pred expand_terms_stream(io__input_stream::in, io__state::di, io__state::uo)
        is det.

expand_terms_stream(Stream) ->
        io__set_input_stream(Stream, _OldStream),
        term_io__read_term(Result),
        expand_terms(Result,0).
```

3.2.2 Expansion von Deklarationen und Grammatikregeln

Nun kommt der Teil, den wir selber schreiben, die Expansion der Grammatikregeln und der Deklarationen (und anderen Klauseln).

$\langle expand_dcg.m\rangle{+}{\equiv}$

```
:- pred expand_terms(read_term::in, int::in, io__state::di, io__state::uo)
        is det.

expand_terms(Result,N) ->
        ( { Result = term(VarSet0, Term0) },
                ( {is_rule(Term0)}
                -> expand_rules(VarSet0,Term0,N,M)
                ; {is_declaration(Term0)}
                -> expand_declarations(VarSet0,Term0),
                    { M = N }
                ; write_terms([pair(Term0,VarSet0)]),
                    { M = N }
                ),
                term_io__read_term(NextResult),
                expand_terms(NextResult,M)
        ; { Result = eof }
        ; { Result = error(Message, LineNum) },
                io__input_stream_name(StreamName),
                { string__format("%s:%03d: %s\n", [s(StreamName), i(LineNum),
                        s(Message)], FullMessage) },
                io__write_string(FullMessage),
                io__set_exit_status(1)
        ).
```

Deklarationen und andere Klauseln, die keine Grammatikregeln sind, werden durch folgenden Fall erfaßt:

$\langle expand_dcg.m\rangle{+}{\equiv}$

```
:- pred is_declaration(term::in) is semidet.
is_declaration(Term) :- from_term(Term,":-",_List).

:- pred expand_declarations(varset::in,term::in,
                            io__state::di,io__state::uo) is det.

expand_declarations(VarSet0,Term0) ->
        { solutions((pred(Pair::out) is nondet :-
                    expand_declaration(VarSet0,Term0,VarSet1,Term1),
                     Pair = pair(Term1,VarSet1)),
                    Pairs)
        },
        write_terms(Pairs),
```

```
        ({list__length(Pairs) > 1} -> io__nl ; []).
```

Die Grammatikregeln werden besonders behandelt:

$\langle expand_dcg.m \rangle + \equiv$

```
  :- pred is_rule(term::in) is semidet.
  is_rule(Term) :- from_term(Term,"->",List), list__length(List) = 2.

  :- pred expand_rules(varset::in,term::in,int::in,int::out,
                       io__state::di,io__state::uo) is det.

  expand_rules(VarSet0,Term0,N,M) ->
          { collect_seq(Term0,[],N,TermS,Seqs,M),                 % Ersetzt seq(A) durch seq(nA)
            solutions((pred(Pair::out) is nondet :-
                      (adjoin_trees3(TermS,VarSet0,TermR,VarSetR), % Musterinstanz: ? -> +
                          adjoin_rec_depth3(TermR,VarSetR, Term,VarSet) % Adjunktion der RekTiefe
                      ; adjoin_trees2(TermS,VarSet0,TermR,VarSetR),
                          adjoin_rec_depth(TermR,VarSetR, Term,VarSet) % Adjunktion der RekTiefe)
                      ; adjoin_trees(TermS,VarSet0, TermR,VarSetR),
                          adjoin_rec_depth(TermR,VarSetR, Term,VarSet) % Adjunktion der RekTiefe)
                      ),
                       Pair = pair(Term,VarSet)),
                      Pairs)
          },
          write_terms(Pairs),
          ({list__length(Pairs) > 1} -> io__nl ; []),
          mk_iteration_rules(Seqs,VarSet0).
```

Zu den in einer Regel gefundenen `seq(A)`'s werden eigene Regeln erzeugt:

$\langle expand_dcg.m \rangle + \equiv$

```
  :- pred mk_iteration_rules(list(term)::in, varset::in, io__state::di, io__state::uo)
          is det.

  mk_iteration_rules([Term0|Terms],VarSet0) ->
          { solutions((pred(Pair::out) is nondet :-
                      ( add_trees_iteration3(Term0,VarSet0,Term1,VarSet1)
                      ; add_trees_iteration2(Term0,VarSet0,Term1,VarSet1)
                      ; add_trees_iteration(Term0,VarSet0,Term1,VarSet1)
                      ),
                       Term = Term1,
                       VarSet = VarSet1,
                       Pair = pair(Term,VarSet)),
                      Pairs)
          },
          write_terms(Pairs),
          ({list__length(Pairs) > 1} -> io__nl ; []),
```

```
        mk_iteration_rules(Terms,VarSet0).
mk_iteration_rules([],_VarSet0) -> [].


:- pred write_terms(list(pair(term,varset))::in,
                    io__state::di,io__state::uo) is det.

write_terms([(Term - VarSet)|Ts]) ->
        term_io__write_term_nl(VarSet,Term),
        write_terms(Ts).
write_terms([]) -> [].
```

3.2.3 Hilfsfunktionen und -prädikate zur Manipulation von Termen

Zum Zerlegen und Konstruieren von Termen definieren wir (als Ersatz für den `univ`-Operator `=..` von Prolog) folgende Funktionen und Prädikate [13]:

$\langle expand_dcg.m\rangle{+}{\equiv}$

```
:- pred from_term(term::in,string::out,list(term)::out) is semidet.
from_term(Term,String,Terms) :-
        Term = term__functor(term__atom(String),Terms,_Context).

:- pred from_termlist(term::in,list(term)::out) is nondet.

from_termlist(Term,Termlist) :-
        from_term(Term,".",[T,Restlist]),
        from_termlist(Restlist,Ts),
        Termlist = [T|Ts].
from_termlist(Term,Termlist) :-
        from_term(Term,"[]",[]),
        Termlist = [].

:- func to_term(string,list(term)) = term.
to_term(String,Terms) =
        term__functor(term__atom(String),Terms,term__context_init).

:- pred to_termlist(term::out,list(term)::in) is nondet.

to_termlist(Term,[T|Ts]) :-
        to_termlist(Restlist,Ts),
        Term = to_term(".",[T,Restlist]).
to_termlist(Term,[]) :-
        Term = to_term("[]",[]).
```

[13] Eine Zusammenfassung von `from/to-term` zu einem Prädikat scheitert an einem `mode mismatch in disjunction` – vermutlich deshalb, weil die Instantiierungsskelette nur mit Variablen besetzt sind.

```
:- func to_int(int) = term.
to_int(I) = term__functor(term__integer(I),[],term__context_init).

:- func to_var(var) = term.
to_var(Var) = term__variable(Var).

:- func to_atom(string) = term.
to_atom(String) = term__functor(term__atom(String),[],term__context_init).


:- func to_vars(list(var)) = list(term).
to_vars(Vars) = Terms :-
        list__map((pred(In::in,Out::out) is det :- Out = to_var(In)),Vars,Terms).

:- func to_atoms(list(string)) = list(term).
to_atoms(Strings) = Terms :-
        list__map((pred(In::in,Out::out) is det :- Out = to_atom(In)),Strings,Terms).

:- pred add_args(term::in,list(term)::in,term::out) is semidet.

add_args(Term,Terms,TermTr) :-
        from_term(Term,F,Args),
        list__append(Args,Terms,ArgsVs),
        TermTr = to_term(F,ArgsVs).
```

Oft brauchen wir benannte Variable `Var` statt anonyme Variable `_Var`, da die letzteren bei verschiedenen Vorkommen wie verschiedene Variable behandelt würden. Um diese Variablen nicht wiederholt in Terme umwandeln zu müssen, definieren wir noch:

$\langle expand_dcg.m\rangle{+}{\equiv}$

```
:- pred new_named_var(varset::in,string::in,term::out,varset::out) is det.
new_named_var(VarSet,VarName,VarTerm,VarSetOut) :-
        varset__new_named_var(VarSet,VarName,Var,VarSetOut),
        VarTerm = to_var(Var).
```

Damit das erzeugte Programm lesbar bleibt, erzeugen wir neue Variablen mit kanonischen Namen `<Prefix>M,<Prefix>M+1,...,<Prefix>N`, die als Termliste ausgegeben werden:

$\langle expand_dcg.m\rangle{+}{\equiv}$

```
:- pred new_named_vars(varset::in,int::in,int::in,string::in,
        list(term)::out,varset::out) is det.

new_named_vars(VarSet,M,N,Prefix,NewVars,VarSetNew) :-
        (M > N
        -> VarSetNew = VarSet, NewVars = []
        ; Suffix = string__int_to_string(M),
           string__append(Prefix,Suffix,PrefixM),
```

```
            new_named_var(VarSet,PrefixM,VarM,VarSetM),
            new_named_vars(VarSetM,M+1,N,Prefix,NewVarsM,VarSetNew),
            NewVars = [VarM|NewVarsM]
        ).
```

3.2.4 Behandlung der Deklaration der Kategorien

Wir erzeugen die Deklarationen der Parse-Prädikate aus den angegebenen grammatischen Deklarationen `:- syncat .[...]. .`

Im Deklarationsteil müssen Typen und Modi für alle Prädikate angegeben sein, in die die lexikalischen und syntaktischen Kategorien der Grammatik übersetzt werden.

⟨*Deklarationsteil*⟩+≡

```
% <Cat>(Resulting Parse-Tree, Search_Pattern-Tree, Tokens, Resttokens):

:- pred <Cat>(<Merkmale>::out,tree::out,tree::in,int::in,trees::in,trees::out)
        is nondet.
   ...
```

Die Deklarationen haben mehr Argumente als die Stelligkeit der Kategorien in den Grammatikregeln, nämlich

(i) `tree::out`, dem bei einer Analyse erzeugten Syntaxbaum,

(ii) `tree::in`, einem Musterbaum, d.h. einen partiellen Syntaxbaum, mit dem der Parsevorgang gesteuert wird,

(iii) `int::in`, einer Schranke für die erlaubte Rekursionstiefe,

(iv) `trees::in`, der Folge der als elementare Syntaxbäume aufgefaßten getaggten Wörter der Eingabesatzes,

(v) `trees::out`, der Restfolge der bei der Analyse nicht verbrauchten getaggten Wörter des Eingabesatzes.

Dazu werden Ausgabemodi `::out` an alle Merkmaltypen der Kategorien angefügt sowie die wegen der Adjunktion von Baum-, Rekursionstiefe- und Differenzlistenargumenten nötigen Deklarationen `tree::out,tree::in,int::in,trees::in,trees::out`.[14]

⟨$expand_d cg.m$⟩+≡

```
:- pred expand_declaration(varset::in,term::in,varset::out,term::out) is nondet.

expand_declaration(VarSet0,Term0,VarSet,Term) :-
        (from_term(Term0,":-",[Dot])
        -> (from_termlist(Dot,[Keyword|Terms])
```

[14]Falls diese Deklarationen für bestimmte Kategorien nicht geeignet sind, kann man die betreffenden Kategorien aus der `synkat` Liste weglassen und eigene Deklarationen angeben.

```
            -> (from_term(Keyword,"syncat",[]),
                  list__member(Trm,Terms),
                  mk_declaration(Trm,"nondet",Term),
                  VarSet = VarSet0
               ; from_term(Keyword,"starts",[]),
                  (mk_startcats(Terms,Term), VarSet=VarSet0;
                  mk_startpredicates(VarSet0,Terms,VarSet,Term))
               )
            ; Term = Term0, VarSet = VarSet0
            )
        ; Term = Term0, VarSet = VarSet0). % z.B. echte Klauseln kopf :- rumpf.


:- pred mk_declaration(term::in,string::in,term::out) is semidet.

mk_declaration(Trm,Mode,Declaration) :-
        from_term(Trm,F,Args),
        add_out_modes(Args,ArgsOut),
        list__append(ArgsOut,[to_term("::",to_atoms(["tree","out"])),
                              to_term("::",to_atoms(["tree","in"])),
                              to_term("::",to_atoms(["int","in"])),
                              to_term("::",to_atoms(["trees","in"])),
                              to_term("::",to_atoms(["trees","out"]))],
                     NewArgs),
        Dec = to_term("is",[to_term(F,NewArgs), to_term(Mode,[])]),
        Declaration = to_term(":-",[to_term("pred",[Dec])]).

:- pred add_out_modes(list(term)::in,list(term)::out) is det.

add_out_modes([Arg|Args],[ArgOut|ArgsOut]) :-
        ArgOut = to_term("::",[Arg,to_atom("out")]),
        add_out_modes(Args,ArgsOut).
add_out_modes([],[]).
```

Die Übersetzung der Pseudodeklaration `:- startcat .[Cat1,...,CatN)].` in ein Hilfsprädikat `startcats(list(term))::out)` und das Parse-Prädikat `parse/5` wird in Abschnitt 3.2.8 beschrieben.

Erzeugung lexikalischer Regeln aus Deklarationen lexikalischer Kategorien

Wenn lexikalische Kategorien in den Regeln wie in `[n(Merkmal,Vollform,Stamm)]` angegeben sind, d.h. wie die Ergebnisse des Taggers, so braucht man keine Parse-Prädikate zu den lexikalischen Kategorien, nur noch Baumkonstruktoren.

Wenn man lexikalischen Regeln wie

⟨*Pseudoregel*⟩≡

```
cat(Merkmale) -> [cat(Merkmale,Wortform,Stamm)].
```

verwendet, liefert die bisherige Adjunktion der Baumargumente nicht genau das Gewünschte (vgl. Bemerkung 3.5). Man könnte aber in der Eingabegrammatik eine Deklaration

⟨*unbenutzt*⟩≡
```
  :- lexcat .[<LexCat1>,...,<LexCatk>].
```

aller lexikalischen Kategorien verlangen und statt der lexikalischen Regeln gleich geeignete Parse-Prädikate erzeugen, je nachdem, ob es um Wortarten mit oder ohne Flexionsformen geht:

⟨*Lexikalisches Regelschema für eine Wortart mit Flexionsformen*⟩≡
```
  n(T,Pat,[T|Ts],Ts) :-
          T = n(_F,_String,Stamm),
          (Pat = (+) ; Pat = (?) ; Pat = T ; Pat = n(Stamm)).
```

⟨*Lexikalisches Regelschema für eine Wortart ohne Flexionsform*⟩≡
```
  prep(T,Pat,[T|Ts],Ts) :-
          T = prep(_String),
          (Pat = (+) ; Pat = (?) ; Pat = T).
```

So wurde in der von Hand geschriebenen Version verfahren.

Aufgabe 3.1 Erzeuge solche Regeln aus einer Deklaration lexikalischer Kategorien.

3.2.5 Behandlung der Grammatikregeln

Adjunktion der Musterbaumeingabe und der Analysebaumausgabe

In jede Grammatikregel soll ein Ausgabe- und ein Eingabeargument vom Typ `tree` eingefügt werden: mit einem Musterbaum als Eingabe soll der Parser gesteuert werden, und der Ausgabebaum soll der Syntaxbaum der jeweiligen Eingabe sein. Dazu ist folgendes zu tun:

(i) Auf der linken Regelseite `Kategorie(Merkmale)` müssen zwei Baumargumente der Form

⟨*Ausgabebaum*⟩≡
```
      Kategorie(Merkmale,Konstituentenbäume)
```

und

⟨*Musterbaum*⟩≡
```
      Kategorie(Merkmale,Variable für Konstituentenbäume)
```

adjungiert werden, das Ausgabeargument des durch den Parse konstruierten Syntaxbaums und das Eingabeargument des Musterbaums.

(ii) Auf der rechten Regelseite müssen in jedem Konjunktionsglied (der disjunktiven Normalform) die passenden Baum- und Mustervariablen adjungiert werden.

Die Markierung der Kopfkonstituenten durch `:(A)` in den rechten Regelseiten wird hierbei beseitigt.[15]

Die Iterationskonstruktion `*(A)` wird bei der Baumadjunktion aufgelöst. (Beachte, daß wir nur eine Konstituente dafür erzeugen dürfen, damit eine Deklaration der Parseprädikate möglich bleibt.)

⟨*expand$_d$cg.m*⟩+≡

```
:- pred adjoin_trees(term::in,varset::in,term::out,varset::out) is semidet.

adjoin_trees(Term0,VarSet0,TermTr,VarSetTr) :-
        from_term(Term0,"->",[Head,Body]),
        arity(Body,0,Arity),
        new_named_vars(VarSet0,1,Arity,"Tr",TreeVars,VarSet1),
        new_named_vars(VarSet1,1,Arity,"Pat",PatVars,VarSetTr),
        add_headtree(Head,TreeVars,PatVars,HeadTr),
        add_trees(Body,TreeVars,PatVars,BodyTr),
        TermTr = to_term("->",[HeadTr,BodyTr]).
```

Zur Steuerung der Suche mit einem gegebenen Baummuster brauchen wir auch ein Muster, `+`, zu dem jeder Baum eine passende Instanz ist.

⟨*expand$_d$cg.m*⟩+≡

```
:- pred adjoin_trees2(term::in,varset::in,term::out,varset::out) is semidet.

adjoin_trees2(Term0,VarSet0,TermTr,VarSetTr) :-
        from_term(Term0,"->",[Head,Body]),
        arity(Body,0,Arity),
        new_named_vars(VarSet0,1,Arity,"Tr",TreeVars,VarSetTr),
        add_headtree2(Head,TreeVars,HeadTr),
        add_trees2(Body,TreeVars,BodyTr),
        TermTr = to_term("->",[HeadTr,BodyTr]).
```

Schließlich brauchen wir noch einen Fall, mit dem das „don't-care"-Muster `?` zum Muster `+` für nicht-leere Konstituenten spezialisiert wird:

⟨*expand$_d$cg.m*⟩+≡

```
:- pred adjoin_trees3(term::in,varset::in,term::out,varset::out) is semidet.

adjoin_trees3(Term0,VarSet0,TermTr,VarSetTr) :-
        from_term(Term0,"->",[Head,_Body]),
        new_named_var(VarSet0,"Tr",TreeVar,VarSetTr),
        add_args(Head,[TreeVar,to_term("?",[])],HeadTr),
        add_args(Head,[TreeVar,to_term("+",[])],BodyTr),
        TermTr = to_term("->",[HeadTr,BodyTr]).
```

[15] Das Kopfprädikat muß z.Zt. noch vom Grammatikschreiber formuliert werden; solange jede Kategorie eine feste Verzweigungsbreite hat, könnten wir es auch aus einer (mit Kopfmarkierung versehenen) Deklaration der syntaktischen Kategorie erzeugen.

Die Stelligkeit einer syntaktischen Konstruktion (Kategorie) ist die Anzahl der Konjunktionsglieder der rechten Regelseite, falls diese aus einer Konjunktion atomarer Kategorien besteht. Ist die rechte Regelseite komplexer strukturiert, so wird überprüft, ob in ihrer disjunktiven Normalform alle Konjunktionen gleich viele Glieder haben; wenn ja, ist dies die Stelligkeit, sonst wird eine Fehlermeldung erzeugt.

Die Konstruktoren `[]`, `seq(A)` und `*(A)` tragen jeweils *eine* Konstituente zur Stelligkeit bei. Aktionen `{A}` zählen natürlich nicht als Konstituenten.

⟨*expand_dcg.m*⟩+≡
```
:- import_module exception.

:- pred arity(term::in,int::in,int::out) is semidet.

arity(Term,Acc,N) :-
        ( from_term(Term,",",[A,B])
        -> arity(A,Acc,NA),
           arity(B,NA,N)
        ; from_term(Term,";",[A,B])
        -> arity(A,Acc,NA),
           arity(B,Acc,NB),
           (NA = NB
           -> N = NA
           ; throw("EDCG error: different branches have differing arities"))
        ; from_term(Term,"?",[A])
        -> arity(A,Acc,N),
           N = Acc + 1
        ; from_term(Term,"{}",_)
        ->  N = Acc
        ; N = Acc + 1  % "seq", "*", "[]", "." und ground predicates
        ).
```

Ergänzung der Baumargumente auf den linken Regelseiten: `add_headtrees`

Die folgende Funktion wird nur für den Fall benutzt, wo die beiden Listen von Eingabetermen nur in Terme umgewandelte Variablen sind:

⟨*expand_dcg.m*⟩+≡
```
:- pred add_headtree(term::in,list(term)::in,list(term)::in,term::out)
        is semidet.

add_headtree(Head,TreeVars,PatVars,HeadTr) :-
        from_term(Head,F,Args),
        add_args(Head,TreeVars,TreeArg),
        add_args(Head,PatVars, PatArg),
        list__append(Args,[TreeArg,PatArg],ArgsExtended),
        HeadTr = to_term(F,ArgsExtended).
```

Für den Fall, wo das Muster eine Mustervariable, d.h. `+`, ist, brauchen wir eine andere Ergänzung:

$\langle expand_d cg.m \rangle{+}{\equiv}$
```
:- pred add_headtree2(term::in,list(term)::in,term::out) is semidet.

add_headtree2(Head,TreeVars,HeadTr) :-
        from_term(Head,F,Args),
        add_args(Head,TreeVars,TreeArg),
        PatArg = to_term("+",[]),
        list__append(Args,[TreeArg,PatArg],ArgsExtended),
        HeadTr = to_term(F,ArgsExtended).
```

Ergänzung der Baumargumente auf den rechten Regelseiten: `add_trees`

Auf der rechten Regelseite bestehen die Baumausgabe und die Muster der Konstituenten aus je einer Variablen, die als Zusatzargumente an die jeweiligen Kategorienprädikate adjungiert werden:

$\langle expand_d cg.m \rangle{+}{\equiv}$
```
:- pred add_trees(term::in,list(term)::in,list(term)::in,term::out) is semidet.
add_trees(Term,TreeTerms,PatTerms,TermTr) :-
        add_trees(Term,TreeTerms,PatTerms,_RestTreeTerms,_RestPatTerms,TermTr).

:- pred add_trees(term::in,list(term)::in,list(term)::in,
                  list(term)::out,list(term)::out,term::out) is semidet.

add_trees(Term,TreeTerms,PatTerms,RestTreeTerms,RestPatTerms,TermTr) :-
        ( from_term(Term,",",[A,B])  % conjunction: (A,B)
        -> add_trees(A,TreeTerms,PatTerms,RestTreeTermsA,RestPatTermsA,ATr),
           add_trees(B,RestTreeTermsA,RestPatTermsA,RestTreeTerms,RestPatTerms,BTr),
           TermTr = to_term(",",[ATr,BTr])
        ; from_term(Term,";",[A,B])  % disjunction: (A;B)
        -> add_trees(A,TreeTerms,PatTerms,RestTreeTerms,RestPatTerms,ATr),
           add_trees(B,TreeTerms,PatTerms,RestTreeTerms,RestPatTerms,BTr),  % !
           TermTr = to_term(";",[ATr,BTr])
        ; from_term(Term,"{}",_A)     % action: { A }
        -> TermTr = Term,
           RestTreeTerms = TreeTerms,
           RestPatTerms = PatTerms
        ; from_term(Term,"?",[A])    % option: ?(A)
        -> TermAux = to_term(";",[A,to_term("[]",[])]),
           add_trees(TermAux,TreeTerms,PatTerms,RestTreeTerms,RestPatTerms,TermTr)
        ; from_term(Term,"*",[A])    % iteration: *(A)
        -> TermAux = to_term("?",[to_term("seq",[A])]),
           add_trees(TermAux,TreeTerms,PatTerms,RestTreeTerms,RestPatTerms,TermTr)
        ; from_term(Term,"[]",[])    % empty: []
```

```
        -> TreeTerms = [TreeTerm|RestTreeTerms],
           PatTerms = [PatTerm|RestPatTerms],
           TermTr = to_term("{}",[to_term("empty",[TreeTerm,PatTerm])])
        ; from_term(Term,":",[A])    % marked head: :(A)
        -> add_trees(A,TreeTerms,PatTerms,RestTreeTerms,RestPatTerms,TermTr)
        ; from_term(Term,".",[A|_B])    % terminal: [A]
        -> TreeTerms = [TreeTerm|RestTreeTerms],
           PatTerms = [PatTerm|RestPatTerms],
           Cond = to_term(";",[to_term("=",[PatTerm,to_atom("+")]),
                     to_term(";",[to_term("=",[PatTerm,to_atom("?")]),
                           to_term(";",[to_term("=",[PatTerm,TreeTerm]),
                                 to_term("=",[PatTerm,
                                       to_term("core",[A])])])])]),
           TermAux = to_term(",",[to_term("=",[TreeTerm,A]), Cond]),
           TermTr = to_term(",",[Term,to_term("{}",[TermAux])])
        ; from_term(Term,_F,_Args)   % categories: cat(features)
        -> TreeTerms = [TreeTerm|RestTreeTerms],
           PatTerms = [PatTerm|RestPatTerms],
           add_args(Term,[TreeTerm,PatTerm],TermTr)
        ; TermTr = Term,
           RestTreeTerms = TreeTerms,
           RestPatTerms = PatTerms).
```

Beachte daß die Adjunktion der Baumargumente auch die Konstruktion `?(A)` auflösen muß, da hier eine Disjunktion versteckt ist, deren Glieder dieselben Variablen für die Argumentbäume unterschiedlich belegen können. (Es genügt nicht, die Variablen an das eingebettete `A` zu adjungieren.)

Beispiel 3.3 Eine Regel mit einer optionalen Konstituente ist etwa:

⟨*Vor der Adjunktion der Baumargumente:*⟩≡

```
 np(Kas,sg) -> det(Kas), ?(ap(Kas,sg)), n(Kas,sg).
```

Die Variablen der Konstituenten im Ausgabe- und im Musterbaum sind korrekt aus denen der Kategorien der rechten Regelseite aufgesammelt:

⟨*Nach der Adjunktion der Baumargumente:*⟩≡

```
 np(Kas, sg, np(Kas, sg, Tr1, Tr2, Tr3), (+)) ->
         det(Kas, Tr1, (+)),
         (ap(Kas, sg, Tr2, (+)) ; { empty(Tr2, (-)) }),
         n(Kas, sg, Tr3, (+)).
 np(Kas, sg, np(Kas, sg, Tr1, Tr2, Tr3),
             np(Kas, sg, Pat1, Pat2, Pat3)) ->
         det(Kas, Tr1, Pat1),
         (ap(Kas, sg, Tr2, Pat2) ; { empty(Tr2, Pat2) }),
         n(Kas, sg, Tr3, Pat3).
 np(Kas, sg, Tr, '?') -> np(Kas, sg, Tr, (+)).
```

Für Regeln, auf deren rechter Seite ein oder mehrere (lexikalisch markierte) Wörter auftreten, werden die Baumargumente so gewählt, daß jedes der einzeln durch `[W]` in der Regel gekennzeichneten Wörter als direkte Konstituente der gebildeten Phrase zählt.

Beispiel 3.4 Gespaltene Präpositionen (oder Circumpositionen) kann man daher z.B. durch

⟨*Vor Adjunktion der Baumargumente*⟩≡
```
pp -> [prep("um")], np, [prep("herum")].
```

direkt in den Regeln angeben, wenn die Teile durch die lexikalische Markierung als `prep` gekennzeichnet werden. Daraus wird:[16]

⟨*Nach Adjunktion der Baumargumente*⟩≡
```
pp(pp(Tr1, Tr2, Tr3), (+)) ->
        ([prep("um")], { Tr1 = prep("um") }),
        np(Tr2, (+)),
        [prep("herum")], { Tr3 = prep("herum") }.
pp(pp(Tr1, Tr2, Tr3), pp(Pat1, Pat2, Pat3)) ->
        ([prep("um")],
            { Tr1 = prep("um"),
              (Pat1 = (+) ; Pat1 = '?' ; Pat1 = Tr1
              ; Pat1 = head(prep("um"))) }),
        np(Tr2, Pat2),
        [prep("herum")],
        { Tr3 = prep("herum"),
          (Pat3 = (+) ; Pat3 = '?' ; Pat3 = Tr3
          ; Pat3 = head(prep("herum"))) }.
pp(Tr, '?') -> pp(Tr, (+)).
```

Die Alternative im erzeugten Muster ist für lexikalische Regelschemata gedacht (und schadet in anderen Regeln nicht): ohne die Fälle, wo das Muster `+` oder `?` ist, wird aus der Regel

⟨*Vor Adjunktion der Baumargumente*⟩+≡
```
n(pl) -> [n(pl,"children","child")].
```

die Regel

⟨*Nach Adjunktion der Baumargumente*⟩+≡
```
n(pl, n(pl, Tr1), n(pl, Pat1)) ->
        [n(pl, "children", "child")],
        { Tr1 = n(pl, "children", "child"),
          (Pat1 = (+) ; Pat1 = '?' ; Pat1 = Tr1
          ; Pat1 = head(n(pl, "children", "child"))) }.
```

Mit ihr wird bei Vorbelegung von `Pat1` durch `n(pl,"children","child")` nur nach den Pluralformen von `"child"` gesucht wird, aber bei Vorbelegung durch `n("child")` nach allen flektierten Vorkommen.

[16]Macht die Klammerung in späteren Übersetzungsschritten Probleme?

Bemerkung 3.5 Ein Nachteil dieser Behandlung von Wörtern `[cat(W)]` auf rechten Regelseiten besteht darin, daß die jeweilige Wortkategorie im Syntaxbaum zweimal repräsentiert wird: einmal auf Grund der Kategorie der linkel Regelseite und einmal wegen der lexikalischen Markierung in `cat(W)`. Das ist der Preis dafür, daß man Grammtiken mit unterschiedlicher Kodierung lexikalischer Information verwenden kann:

(i) durch lexikalischen Regeln, in denen die jeweiligen Wortformen als Zeichenreihen angegeben werden,

(ii) durch lexikalischen Regeln, in deren Einträgen schon die lexikalischen Kategorien (und ggf. weitere Information) vorkommen, und

(iii) durch lexikalischen Regel*schemata*, die keinerlei konkrete Wortinformation enthalten.

Die erste und dritte der Regeln

⟨*Lexikalische Regeln*⟩≡
```
n(pl) -> ["children"].
n(pl) -> [n(pl,"children","child")].
n(pl) -> [n(pl,Vollform,Stammform)].
```

werden nach dem obigen Schema in folgende Regeln übersetzt, die zweite in die entsprechende Instanz der letzen:

⟨*Lexikalische Regeln nach Adjunktion der Baumargumente*⟩≡
```
n(pl, n(pl, Tr1), n(pl, Pat1)) ->
        ["children"],
        { Tr1 = "children", (Pat1 = Tr1 ; Pat1 = head("children")) }.
n(pl, n(pl, Tr1), n(pl, Pat1)) ->
        [n(pl, Vollform, Stammform)],
        { Tr1 = n(pl, Vollform, Stammform),
          (Pat1 = Tr1 ; Pat1 = head(n(pl, Vollform, Stammform))) }.
```

Die letzte Form brauchen wir, wenn die Eingabe aus getaggten Texten stammt. (Der erzeugte Syntaxbaum ist für die erste Form am natürlichsten.)

Die Ergänzung mit unbestimmtem Muster + erfolgt wieder gesondert:

⟨$expand_dcg.m$⟩+≡
```
:- pred add_trees2(term::in,list(term)::in,term::out) is semidet.
add_trees2(Term,TreeVars,TermTr) :-
        add_trees2(Term,TreeVars,_RestTreeVars,TermTr).

:- pred add_trees2(term::in,list(term)::in,
                   list(term)::out,term::out) is semidet.

add_trees2(Term,TreeVars,RestTreeVars,TermTr) :-
        ( from_term(Term,",",[A,B])  % conjunction: (A,B)
        -> add_trees2(A,TreeVars,RestTreeVarsA,ATr),
```

```
            add_trees2(B,RestTreeVarsA,RestTreeVars,BTr),
            TermTr = to_term(",",[ATr,BTr])
        ; from_term(Term,";",[A,B])  % disjunction: (A;B)
        -> add_trees2(A,TreeVars,RestTreeVars,ATr),
            add_trees2(B,TreeVars,RestTreeVars,BTr),  % !
            TermTr = to_term(";",[ATr,BTr])
        ; from_term(Term,"{}",_A)     % action: { A }
        -> TermTr = Term,
            RestTreeVars = TreeVars
        ; from_term(Term,"?",[A])    % option: ?(A)
        -> TermAux = to_term(";",[A,to_term("[]",[])]),
            add_trees2(TermAux,TreeVars,RestTreeVars,TermTr)
        ; from_term(Term,"*",[A])    % iteration: *(A)
        -> TermAux = to_term("?",[to_term("seq",[A])]),
            add_trees2(TermAux,TreeVars,RestTreeVars,TermTr)
        ; from_term(Term,"[]",[])    % empty: []
        -> TreeVars = [TreeVar|RestTreeVars],
            TermTr = to_term("{}",[to_term("empty",[TreeVar,to_term("-",[])])])
        ; from_term(Term,":",[A])    % marked head: :(A)
        -> add_trees2(A,TreeVars,RestTreeVars,TermTr)
        ; from_term(Term,".",[A|_B]) % terminal: [A]
        -> TreeVars = [TreeVar|RestTreeVars],
            TermAux = to_term("=",[TreeVar,A]),
            TermTr = to_term(",",[Term,to_term("{}",[TermAux])])
        ; from_term(Term,_F,_Args)   % categories: cat(features)
        -> TreeVars = [TreeVar|RestTreeVars],
            add_args(Term,[TreeVar,to_term("+",[])],TermTr)
        ; TermTr = Term,
            RestTreeVars = TreeVars).
```

Beispiel 3.6 Die Ergänzung mit unbestimmtem Muster + ergibt für die Regel

⟨*Vor der Adjunktion der Baumargumente:*⟩+≡

```
np(Kas,sg) -> det(Kas), ?(ap(Kas,sg)), n(Kas,sg).
```

die folgende Regel, bei der die Variablen der Konstituenten im Ausgabebaum aus denen der Kategorien der rechten Regelseite aufgesammelt sind und die Muster korrekt besetzt wurden:

⟨*Nach der Adjunktion der Baumargumente:*⟩+≡

```
np(Kas, sg, np(Kas, sg, Tr3, Tr2, Tr1), (+)) ->
        det(Kas, Tr3, (+)),
        (ap(Kas, sg, Tr2, (+)) ; { empty(Tr2, (-)) }),
        n(Kas, sg, Tr1, (+)).
```

Beispiel 3.7 Die Adjunktion der Baumargumente bei lexikalischen Regeln und Regelschemata und variablem Muster ergibt für die Regeln aus 3.4 die Regeln:

⟨*Nach der Adjunktiopn der Baumargumente:*⟩≡

```
n(pl, n(pl, Tr1), (+)) ->
        ["children"], { Tr1 = "children" }.
n(pl, n(pl, Tr1), (+)) ->
        [n(pl, Vollform, Stammform)],
        { Tr1 = n(pl, Vollform, Stammform) }.
```

Beispiel 3.8 Leere Konstituenten kann man in der Grammatikregel mit `[]` angeben:

⟨*Vor der Adjunktion der Baumargumente:*⟩+≡

```
inp -> [], vinf, ap, ?(advp), ?(np;pp).
```

⟨*Nach der Adjunktion der Baumargumente:*⟩+≡

```
inp(inp(Tr5, Tr4, Tr3, Tr2, Tr1), (+)) ->
        { empty(Tr5, (-)) },
        vinf(Tr4, (+)),
        ap(Tr3, (+)),
        (advp(Tr2, (+)) ; { empty(Tr2, (-)) }),
        ((np(Tr1, (+)) ; pp(Tr1, (+))) ; { empty(Tr1, (-)) }).
```

3.2.6 Beschränkung der Rekursionstiefe

Um auszuschließen, daß der DCG-Parser bei Grammatiken mit linksrekursiven Regeln in eine Endlosschleife gerät, adjungieren wir ein Argument an die Grundprädikate, mit dem die Rekursionstiefe beschränkt werden kann.

⟨*expand$_d$cg.m*⟩+≡

```
:- pred adjoin_rec_depth(term::in, varset::in, term::out, varset::out)
        is semidet.

adjoin_rec_depth(Term0, VarSet0, Term, VarSet) :-
        from_term(Term0,"->",[Links,Rechts]),
        new_named_var(VarSet0,"RekTiefe",Tiefe,VarSetT),
        add_rec_depth(Links,Tiefe,Kopf),
        new_named_var(VarSetT,"RekTiefe2",TiefeL,VarSet),
        add_rec_depth(Rechts,TiefeL,RumpfL),
            % Rumpf := ({TiefeL is Tiefe - 1, 0 =< TiefeL}, RumpfL),
        DefTiefeL = to_term("is",[TiefeL,to_term("-",[Tiefe,to_int(1)])]),
        PosConstraint = to_term("=<",[to_int(0),TiefeL]),
        Action =  to_term("{}",[to_term(",",[DefTiefeL,PosConstraint])]),
        Term = to_term("->",[Kopf,to_term(",",[Action,RumpfL])]).
```

Wir setzen für `add_rec_depth` voraus, daß `Tiefe` die Termform einer Variablen ist, die nicht in der `VarSet` des Eingabeterms vorkommt:

⟨*expand$_d$cg.m*⟩+≡

```
:- pred add_rec_depth(term::in, term::in, term::out) is det.

add_rec_depth(Term,Tiefe,TermTr) :-
```

```
        ( from_term(Term,",",[A,B])  % conjunction: (A,B)
        ->  add_rec_depth(A,Tiefe,ATr),
            add_rec_depth(B,Tiefe,BTr),
            TermTr = to_term(",",[ATr,BTr])
        ; from_term(Term,";",[A,B])  % disjunction: (A;B)
        ->  add_rec_depth(A,Tiefe,ATr),
            add_rec_depth(B,Tiefe,BTr),
            TermTr = to_term(";",[ATr,BTr])
        ; from_term(Term,"{}",[_A])   % action: { A }
        ->  TermTr = Term
        ; from_term(Term,"?",[A])    % option: ?(A)
        ->  add_rec_depth(A,Tiefe,ATr),
            TermTr = to_term("?",[ATr])
        ; from_term(Term,":",[A])    % marked head: :(A)
        ->  add_rec_depth(A,Tiefe,ATr),
            TermTr = to_term(":",[ATr])
        ; from_term(Term,"[]",[])    % empty: []
        ->  TermTr = Term
        ; from_term(Term,".",_Terms) % terminals: [A|B]
        ->  TermTr = Term
        ; from_term(Term,F,Args)     % categories: cat(features)
        ->  append(Args,[Tiefe],ArgsT),
            TermTr = to_term(F,ArgsT)
        ; TermTr = Term).
```

Bei der Musterspezialisierung von ? zu + sollte die Rekursionstiefe nicht verringert werden:

⟨*expand$_d$cg.m*⟩+≡

```
 :- pred adjoin_rec_depth3(term::in, varset::in, term::out, varset::out)
        is semidet.

 adjoin_rec_depth3(Term0, VarSet0, Term, VarSet) :-
        from_term(Term0,"->",[Links,Rechts]),
        new_named_var(VarSet0,"RekTiefe",Tiefe,VarSet),
        add_rec_depth(Links,Tiefe,Kopf),
        add_rec_depth(Rechts,Tiefe,Rumpf),
        Term = to_term("->",[Kopf,Rumpf]).
```

Beispiel 3.9 Die Adjunktion des Arguments zur Rekursionstiefe (ohne die der Baumargumente) ergibt z.B.:

⟨*Testbeispiel: Vor der Adjunktion der Rekursionstiefe:*⟩≡

```
 np(Kasus,pl) -> np(Kasus,Num1), [und], np(Kasus,Num2).
 np(Kasus,pl) -> det(Kasus,pl), ?(ap(Kas,pl)), n(Kasus,pl).
```

⟨*Testbeispiel: nach der Adjunktion der Rekursionstiefe:*⟩≡

```
 np(Kasus, pl, RekTiefe) ->
        { RekTiefe2 is RekTiefe - 1, 0 =< RekTiefe2 },
```

```
        np(Kasus, Num1, RekTiefe2), [und], np(Kasus, Num2, RekTiefe2).
np(Kasus, pl, RekTiefe) ->
        { RekTiefe2 is RekTiefe - 1, 0 =< RekTiefe2 },
        det(Kasus, pl, RekTiefe2),
        '?'(ap(Kas, pl, RekTiefe2)),
        n(Kasus, pl, RekTiefe2).
```

In die Deklaration der Parseprädikate wurde der Typ (und Modus) des neuen Arguments `Tiefe` eingebaut. (Das sollte aber hier mit einer separaten Klausel gemacht werden.)

Beim Aufruf der Parseprädikate muß eine Schranke der Rekursionstiefe übergeben werden; dazu kann man etwa die doppelte Länge des Eingabesatzes nehmen, plus eine Konstante, um die leere Eingabe zu erlauben.

Beispiel 3.10 (Linksrekursion) Mit einer linksrekursiven Regel wie

$\langle$*Linksrekursion*$\rangle\equiv$
```
detp -> detp, [], :([en(gen,Voll,Stamm)]).
```

erhält man ohne den Einbau des Arguments `RekTiefe` einen Laufzeitfehler:

$\langle$*Overflow*$\rangle\equiv$
```
Mercury runtime: memory zone nondetstack#0 overflowed
PC at signal: 134518521 (80496f9)
This may have been caused by a stack overflow, due to unbounded recursion.
```

Nach der Adjunktion der Rekursionstiefenschranke terminiert das Programm.

3.2.7 Behandlung der Iteration

Die Kleene'sche Iteration α^* kann mit Hilfe der regulären Gleichung $\alpha^* = 1 + \alpha^+$ auf die nicht-leere Iteration α^+ zurückgeführt werden, die ihrerseits durch die Rekursionsbedingung $\alpha^+ = \alpha; (1 + \alpha^+)$ definiert ist. Diese entspricht einer EDCG-Regel

$\langle$*Schema der nicht-leeren Iteration*$\rangle\equiv$
```
seq(A) -> A, ?(seq(A)).
```

Falls also auf einer rechten Regelseite ein `*(A)` verwendet wird, muß es in `?(seq(A))` übersetzt werden. In diesem Fall, oder wenn `seq(A)` direkt auf einer rechten Regelseite verwendet wird, muß die (übersetzte) Grammatik eine Deklaration

$\langle$*Idee*$\rangle\equiv$
```
:- pred seq(T::M,tree::out,tree::in,trees::in,trees::out) is nondet.
```
mit geeignetem Typ `T` und Modus `M` enthalten. Nach Adjunktion der Baumargumente sollte das Schema der Regel

$\langle$*Idee*$\rangle+\equiv$
```
seq(A,seq([Tr|Trs]),seq([Pat|Pats])) ->
        A(Tr,Pat),
        ?(seq(A,Trs,Pats)).
```

entsprechen. Das wirft zwei Probleme auf:

(i) Die Adjunktion der Baumargumente an `A` kann nicht mit einer *Variablen* `A` durchgeführt werden! Also werden zuerst die Vorkommen von `*(A)` in der Eingabegrammatik zu Gunsten von `?(seq(A))` beseitigt, dann alle vorkommenden Ausdrücke `seq(A)` gesammelt und schließlich für jedes dieser *konkreten* `A` die Spezialfälle von ⟨*Idee*⟩ erzeugt.

(ii) Zur Deklaration von `seq(T::M,tree::out,tree::in,trees::in,trees::out)` bräuchte man als Typ `T` eine Klasse von Mercury-Goals, eben die syntaktischen und lexikalischen Kategorien. Da es keinen Datentyp `goal` gibt, müßte man eine Umbennennung der Mercury-Konstruktoren für Goals in Konstruktoren eines Datentyps `cat` umbenennen (mit partiellen Funktionen `goal_to_cat(term) = term, cat_to_goal(term) = term`), wobei `cat` die Kategorien unterhalb von `seq` umfassen müßte.[17]

Da aber das Argument `A:T` auch freie Variablen enthält, z.B. wenn `A` eine lexikalische Kategorie darstellt, kommt als Modus `M` nur `free -> free` in Frage: `A` ist weder beim Aufruf noch bei der Beendigung des Aufrufs von `seq(A,...)` durch einen Grundterm belegt. Leider ist das Modus-System von Mercury unvollständig implementiert, so daß man `seq` nicht wie erforderlich deklarieren kann!

Als Ausweg setzen wir im ersten Argument von `seq` nicht den Term `A`, sondern eine Nummer `nA:int` ein. Zuerst muß also eine Aufzählung aller in der Eingabegrammatik auftretenden `*(A)` und `seq(A)` konstruiert werden, damit wir die Iteration durch

⟨*Realisierung*⟩≡
```
:- pred seq(int::in,tree::out,tree::in,trees::in,trees::out) is nondet.
```

deklarieren und für jedes in der Grammatik vorkommende `seq(A)` oder `*(A)` die Klausel

⟨*Realisierung*⟩+≡
```
seq(nA,seq([Tr|Trs]),seq([Pat|Pats])) ->
        A(Tr,Pat),
        ?(seq(nA,Trs,Pats)).
```

erzeugen können. Hierbei steht `A(Tr,Pat)` für eine Erweiterung der Argumente des Funktionssymbols von `A` um zwei Argumente `Tr,Pat`.

Sammlung der seq(A)-Kategorien einer Regel

Wir akkumulieren die Teile `seq(A)` einer Regel in einer Liste, gepaart mit Nummern `nA` in der Form `seq(nA,A)`, und ersetzen in der Grammatikregel `seq(A)` durch `seq(nA)`, damit die anschließenden Übersetzungsschritte (Adjunktion der Baumargumente und Rekursionstiefe) gleich Regeln mit den richtigen Aufrufen `seq(nA)` erhalten.

⟨*expand_dcg.m*⟩+≡
```
:- type terms == list(term).

:- pred collect_seq(term::in,terms::in,int::in,term::out,terms::out,int::out)
```

[17] Man kann `T` auch nicht unbestimmt lassen, etwa als `T::out`, da dies zu einer vom Compiler nicht auflösbaren Typüberladung des Kategorienkonstruktors führt.

```
        is det.

collect_seq(Term,Acc,N,NewTerm,NewAcc,NewN) :-
        (from_term(Term,"->",[Head,Body])
        -> collect_seq(Body,Acc,N,NewBody,NewAcc,NewN),
            NewTerm = to_term("->",[Head,NewBody])
        ; from_term(Term,",",[A,B])
        -> collect_seq(A,Acc,N,NewA,AccA,NA),
            collect_seq(B,AccA,NA,NewB,NewAcc,NewN),
            NewTerm = to_term(",",[NewA,NewB])
        ; from_term(Term,";",[A,B])
        -> collect_seq(A,Acc,N,NewA,AccA,NA),
            collect_seq(B,AccA,NA,NewB,NewAcc,NewN),
            NewTerm = to_term(";",[NewA,NewB])
        ; from_term(Term,"seq",[A])
        -> NewTerm = to_term("seq",[to_int(N)]),
            NewAcc = [to_term("seq",[to_int(N),A])|Acc],
            NewN is N + 1
        ; from_term(Term,"*",[A])
        -> NewTerm = to_term("?",[to_term("seq",[to_int(N)])]),
            NewAcc = [to_term("seq",[to_int(N),A])|Acc],
            NewN is N + 1
        ; NewTerm = Term,
            NewAcc = Acc,
            NewN = N
        ).
```

Hierbei wird die kleinste unverbrauchte Nummer `M` ausgegeben, damit bei der nächsten Regel mit `M` weitergezählt werden kann.

Erzeugung der Klauseln zu `seq(A)`

Die zu iterierenden Kategorien `seq(A)` einer rechten Regelseite wurden numeriert und als Liste von Ausdrücken `seq(n,A)` gesammelt. Aus jedem solchen Ausdruck erzeugen wir nun das Parse-Prädikat für die nicht-leere Iteration von `A`. (Dabei werden auch die Argumente für die Beschränkung der Rekursionstiefe eingebaut, was man abtrennen sollte.)

$\langle expand_d cg.m \rangle + \equiv$

```
:- pred add_trees_iteration(term::in,varset::in,term::out,varset::out)
        is semidet.

add_trees_iteration(Term,VarSet,TermTr,VarSetTr) :-
        from_term(Term,"seq",[N,A]),
        new_named_var(VarSet,"RekTiefe",RekDepth,VarSetR),
        new_named_var(VarSetR,"Trs",Trs,VarSet1),
        new_named_var(VarSet1,"Tr",Tr,VarSet2),
        new_named_var(VarSet2,"Pats",Pats,VarSet3),
```

```
        new_named_var(VarSet3,"Pat",Pat,VarSetTr),
        TrList = to_term(".",[Tr,Trs]),
        PatList = to_term(".",[Pat,Pats]),
        TermN = to_term("seq",[N]),
        add_args(TermN,[to_term("seq",[TrList]),
                        to_term("seq",[PatList]),
                        RekDepth],
                 Head),
        (from_term(A,".",[Token|_])
        -> ATr = to_term(",",[A,to_term("{}",
                                        [to_term(",",[to_term("=",[Tr,Token]),
                                                      to_term("=",[Pat,Token])])])])
        ; from_term(A,";",[A1,A2])
        -> add_trees(A1,[Tr],[Pat],A1Tr),
            add_trees(A2,[Tr],[Pat],A2Tr),
            add_rec_depth(A1Tr,RekDepth,A1TrR),
            add_rec_depth(A2Tr,RekDepth,A2TrR),
            ATr = to_term(";",[A1TrR,A2TrR])
        ; add_trees(A,[Tr],[Pat],ATr0),
            add_args(ATr0,[RekDepth],ATr)
        ),
        add_args(TermN,[to_term("seq",[Trs]),
                        to_term("seq",[Pats]),
                        RekDepth],
                 SeqTrBody),
        Nil = to_term("[]",[]),
        Empty = to_term("{}",[to_term(",",[to_term("=",[Trs,Nil]),
                                           to_term("=",[Pats,Nil])])]),
        Body = to_term(",",[ATr,to_term(";",[SeqTrBody,Empty])]),
        TermTr = to_term("->",[Head,Body]).
```

Und entsprechend für die Fälle, wo das Muster eine Mustervariable ist: (ebenfalls mit eingebauten Argumenten für die Beschränkung der Rekursionstiefe)

$\langle expand_d cg.m \rangle + \equiv$

```
:- pred add_trees_iteration2(term::in,varset::in,term::out,varset::out)
        is semidet.

add_trees_iteration2(Term,VarSet,TermTr,VarSetTr) :-
        from_term(Term,"seq",[N,A]),
        new_named_var(VarSet,"RekTiefe",RekDepth,VarSetR),
        new_named_var(VarSetR,"Trs",Trs,VarSetTrs),
        new_named_var(VarSetTrs,"Tr",Tr,VarSetTr),
        Pat = to_term("+",[]),
        TrList = to_term(".",[Tr,Trs]),
        TermN = to_term("seq",[N]),
```

```
        add_args(TermN,[to_term("seq",[TrList]),Pat,RekDepth],Head),
        (from_term(A,".",[Token|_])
        -> ATr = to_term(",",[A,to_term("{}",[to_term("=",[Tr,Token])])])
        ; from_term(A,";",[A1,A2])
        -> add_trees(A1,[Tr],[Pat],A1Tr),
            add_trees(A2,[Tr],[Pat],A2Tr),
            add_rec_depth(A1Tr,RekDepth,A1TrR),
            add_rec_depth(A2Tr,RekDepth,A2TrR),
            ATr = to_term(";",[A1TrR,A2TrR])
            % falsch: add_rec_depth(to_term(";",[A1Tr,A2Tr]),RekDepth,ATr)
        ; add_trees(A,[Tr],[Pat],ATr0),
            add_args(ATr0,[RekDepth],ATr)
        ),
        add_args(TermN,[to_term("seq",[Trs]),Pat,RekDepth],SeqTrBody),
        Nil = to_term("[]",[]),
        Empty = to_term("{}",[to_term("=",[Trs,Nil])]),
        Body = to_term(",",[ATr,to_term(";",[SeqTrBody,Empty])]),
        TermTr = to_term("->",[Head,Body]).
```

⟨*expand*$_d$*cg.m*⟩+≡

```
:- pred add_trees_iteration3(term::in,varset::in,term::out,varset::out)
        is semidet.

add_trees_iteration3(Term,VarSet,TermTr,VarSetTr) :-
        from_term(Term,"seq",[N,A]),
        new_named_var(VarSet,"RekTiefe",RekDepth,VarSetR),
        new_named_var(VarSetR,"Tr",Tr,VarSetTr),
        Pat = to_term("+",[]),
        Head = to_term("seq",[N,Tr,to_term("?",[]),RekDepth]),
        Body = to_term("seq",[N,Tr,to_term("+",[]),RekDepth]),
        TermTr = to_term("->",[Head,Body]).
```

Die Baumlisten werden in einen `tree`-Konstruktor `seq` verpackt, damit die Musterargumente auch vom Typ `tree` sind, wie die Pattern `?`, `+`, `-`, die in die Aufrufe hineinpropagiert werden.

Beispiel 3.11 Das übersetzt z.B. `seq(1,np(Kas))` korrekt in folgende Regeln:

⟨*Testbeispiel*⟩≡

```
seq(1, seq([Tr | Trs]), (+), RekTiefe) ->
        np(Kas, Tr, (+), RekTiefe),
        (seq(1, seq(Trs), (+), RekTiefe) ; { Trs = [] }).
seq(1, seq([Tr | Trs]), seq([Pat | Pats]), RekTiefe) ->
        np(Kas, Tr, Pat, RekTiefe),
        (seq(1, seq(Trs), seq(Pats), RekTiefe) ; { Trs = [], Pats = [] }).
seq(1, Tr, '?', RekTiefe) ->
        seq(1, Tr, (+), RekTiefe).
```

3.2.8 Erzeugung der Parse-Aufrufe mit Startkategorien

Das Suchprogramm ruft den Parser

⟨*Parseraufruf*⟩+≡
```
:- pred parse(trees::in,tree::in,
              pred(tree)::((pred(in) is semidet) -> pred(in) is semidet),
              trees::in,trees::out).
```

in der Form

⟨*Parseraufruf*⟩+≡
```
parse(Liste getaggter Wörter,Muster,Filterprädikat,[],Treffer).
```

auf, wobei die leere Liste der Anfangswert eines Akkumulators derjenigen Bäume ist, die zum Muster paßten und den Filter passierten.

Wir verlangen vom Grammatikschreiber eine Definition

⟨*Typ der Startkategorien*⟩+≡
```
:- type startcat --> ...
```

der möglichen Startkategorien[18] gegebenenfalls mit ihren Merkmalbereichen. Eine

⟨*Pseudodeklaration*⟩+≡
```
:- starts .[<Cat1>,...,<Catn>].
```

aller (tatsächlichen) Startkategorien der Eingabegrammatik soll in eine Deklaration und Definition des Parser-Aufrufs `parse/5` übersetzt werden, der die zu den Startkategorien gehörenden Parse-Prädikat aufruft.

Aus der Pseudodeklaration der Startkategorien erzeugen wir zuerst ein Hilfsprädikat[19]

⟨*Deklaration der Startkategorien*⟩≡
```
:- startcats(list(startcat)::out) is det.
startcats([<Cat1>,...,<Catn>]).
```

[18] Falls man keine lexikalischen Regeln in der Grammatik hat, müssen die Startkategorien syntaktische Kategorien sein. Wenn die Startkategorien nur ausgewählte Merkmalwerte enthalten, schränkt das die Merkmalwerte an der Wurzel der Muster implizit ein – was den Benutzer irritieren kann. Kategorien der Form `seq(A)` können keine Startkategorien sein, es sei denn, man zählt an den Regeln ab, mit welcher Nummer `n` man die Startkategorie `seq(n)` in der Liste statt `seq(A)` nennen muß.

[19] Es wird nicht geprüft, ob es zu jedem `startcat`-Konstruktor einen „passenden“ Konstruktor in `:- type tree --> ...` gibt, d.h. einen, der den `startcat`-Konstruktor um genau so viele Argumente vom Typ `tree` erweitert, wie es Konstitutenen in den zugehörigen Grammatikregeln gibt.

Dies geschieht durch die Klauseln:

⟨$expand_dcg.m$⟩+≡
```
:- pred mk_startcats(list(term)::in,term::out) is multi.

mk_startcats(_Terms,Declaration) :-
        Typ = to_term("list",[to_term("startcat",[])]),
        Mode = to_term("out",[]),
        Pred = to_term("startcats",[to_term("::",[Typ,Mode])]),
        Dec = to_term("is",[Pred, to_term("det",[])]),
        Declaration = to_term(":-",[to_term("pred",[Dec])]).
mk_startcats(Terms,Definition) :-
        to_termlist(Term,Terms),
        Definition = to_term("startcats",[Term]).
```

Außerdem wird eine Hilfsrelation[20]

⟨*Hilfstabelle*⟩≡
```
start(startcat::in,
      pred(tree,tree,int,trees,trees) :: out(pred(out,in,in,in,out)) is nondet
     ) is semidet.
```

erzeugt, die den Startkategorien mit ihren Merkmalen Parse-Prädikate zuordnet, in denen diese Merkmalwerte schon berücksichtigt sind. Der Parser kann mit dieser Hilfstabelle „uniform", d.h. ohne die Merkmalwerte der Startkategorien (oder ihre Typen), definiert werden.

⟨$expand_dcg.m$⟩+≡
```
:- pred mk_startpredicates(varset::in,list(term)::in,varset::out,term::out)
        is multi.

mk_startpredicates(VarSet0,_SynCats,VarSet,Declaration) :-
        PredMode = to_term("is",[to_term("pred",to_atoms(["out","in","in",
                                                          "in","out"])),
                                 to_term("nondet",[])]),
        Pred = to_term("start",
                       [to_term("::",to_atoms(["startcat","in"])),
                        to_term("::",[to_term("pred",
                                              to_atoms(["tree","tree","int",
                                                        "trees","trees"])),
                                      to_term("out",[PredMode])])]),
        Dec = to_term("is",[Pred, to_term("semidet",[])]),
        Declaration = to_term(":-",[to_term("pred",[Dec])]),
        VarSet = VarSet0.

mk_startpredicates(VarSet0,StartCats,VarSet,Term) :-
        list__member(Cat,StartCats),
```

[20]Da ihre Deklaration nicht von der konkreten Liste der Startkategorien abhängen, könnte man sie (wie das folgende Parser-Prädikat) auch direkt in der Eingabedatei fordern und einfach übernehmen.

```
        mk_startpredicate(VarSet0,Cat,VarSet,Term).

:- pred mk_startpredicate(varset::in,term::in,varset::out,term::out) is semidet.
mk_startpredicate(VarSet0,Term0,VarSet,Predicate) :-
        new_named_var(VarSet0,"Tree",Tree,VarSet1),
        new_named_var(VarSet1,"PatTree",PatTree,VarSet2),
        new_named_var(VarSet2,"RekDepth",RekDepth,VarSet3),
        new_named_var(VarSet3,"Leaves",Leaves,VarSet4),
        new_named_var(VarSet4,"RestLeaves",RestLeaves,VarSet),
        Pred = to_term("pred",[to_term("::",[Tree,to_atom("out")]),
                               to_term("::",[PatTree,to_atom("in")]),
                               to_term("::",[RekDepth,to_atom("in")]),
                               to_term("::",[Leaves,to_atom("in")]),
                               to_term("::",[RestLeaves,to_atom("out")])]),
        Dec = to_term("is",[Pred, to_term("nondet",[])]),
        from_term(Term0,F,Args),
        list__append(Args,[Tree,PatTree,RekDepth,Leaves,RestLeaves],ArgsNew),
        Body = to_term(F,ArgsNew),
        Klausel = to_term(":-",[Dec,Body]),
        Predicate = to_term("start",[Term0,Klausel]).
```

Beispiel 3.12 Aus der (sehr einfachen)

⟨*Pseudodeklaration*⟩+≡
```
:- starts .[np(sg)].
```

werden folgende Deklarationen und Klauseln für die Hilfstabelle erzeugt:

⟨*Ergebnis der Übersetzung*⟩≡
```
:- pred start((startcat :: in), (pred(tree, tree, int, trees, trees)
                                :: out(pred(out, in, in, in, out) is nondet))
             ) is semidet.
:- pred startcats((list(startcat) :: out)) is det.

start(np(sg), (pred((Tree :: out), (PatTree :: in), (RekDepth :: in),
                    (Leaves :: in), (RestLeaves :: out)) is nondet :-
              np(sg, Tree, PatTree, RekDepth, Leaves, RestLeaves))).

startcats([np(sg)]).
```

Das Hauptprädikat des Parsers könnte jetzt ebenfalls erzeugt werden, aber einfachheitshalber verlangen wir das (vorläufig) vom Grammatikschreiber:

⟨*Parseraufruf*⟩+≡
```
parse([Leaf|Leaves],PatTree,Filter,Acc,Trees) :-
        ( startcats(StartCats),
            list__member(Cat,StartCats),
```

```
            start(Cat,ParsePred),
            list__length([Leaf|Leaves],InputLength),
            RekDepth = 2 + InputLength,
            ParsePred(Tree,PatTree,RekDepth,[Leaf|Leaves],RestLeaves),
            Filter(Tree)
        -> NewAcc = [Tree|Acc],
            parse(RestLeaves,PatTree,Filter,NewAcc,Trees)
        ; parse(Leaves,PatTree,Filter,Acc,Trees) ).
parse([],_PatTree,_Filter,Acc,Trees) :- Trees = Acc.
```

Beachte, daß prinzipiell alle Startprädikate zur Suche verwendet werden[21] ; aber im Aufruf von `ParsePred(Tree,PatTree,...)` wird das Funktionszeichen des Musters, falls es keine Variable `?` oder `+` ist, implizit mit dem Parseprädikat verglichen; dadurch werden die irrelevanten Startprädikate relativ schnell ausgeschlossen.

3.2.9 Erzeugung der Kopffunktion

Um aus den Markierungen `:(A)` in der Grammatik die Klauseln der Funktion

⟨*Lexikalischer Kopf*⟩≡

```
:- head(tree) = tree is semidet.
```

zu erzeugen, müssen in jeder Grammatikregel alle rechten Seiten ausdistribuiert werden und für jedes Disjunktionsglied eine Klausel erzeugt werden. (Damit kann jedenfalls der Kopf in jeder Alternativen in einer anderen Konstituenten liegen.)

Aufgabe 3.2 Erzeuge die Klauseln der Kopffunktion aus den Annotationen der Grammatikregeln. Erzeuge vielleicht auch eine Stammfunktion für die lexikalischen Kategorien oder die Konstituenten `<LexCat>(<Merkmale>,<Wortformen>)`.

3.3 Ergebnis einer Grammatikübersetzung

Beispiel 3.13 Die Eingabegrammatik `grammatik.dcg` aus Beispiel 2.2 wird zu folgender Datei `grammatik.m` expandiert: (das ist nicht mehr aktuell, da `grammatik.dcg` erheblich geändert wurde!) (Umbruch von Hand)

⟨*Ergebnis der Übersetzung*⟩+≡

```
:- module grammatik.
```

[21] Schwieriger ist, aus dem Muster `F(Merkmale,Trees)` direkt das passende Parse-Prädikat zu konstruieren:

(i) aus dem Namen `F` kann man keinen Aufruf `F(Tree,...)` bilden, denn zur Parsezeit kennt man zwar `PatTree` und mit etwas Mühe auch `StartCats`, kann aber das dann konstruierbare Prädikat `ParsePred` nicht aufrufen, da Mercury keine interpretierende Sprache ist, und zur Zeit der Grammatikübersetzung kennt man zwar `StartCats`, aber nicht das Suchmuster `PatTree`,

(ii) man muß berücksichtigen, daß noch Merkmalargumente vor dem Ausgabebaum einzufügen sind.

```
:- interface.

:- import_module list.

:- type featureEN --> sg ; gen.
:- type featureN --> sg ; pl ; sp ; gen ; num.
:- type featureAdj --> b ; c ; s ; crd ; ord ; prd.
:- type featureAdv --> b ; c ; s ; part ; wh.
:- type featureV --> inf ; grnd ; past ; ger ; part ; prsg ; mdl ; aux.
:- type featurePron --> idef ; pers ; refl ; wh ; rel ; poss ; def.
:- type featureDet --> wh ; rel ; poss ; def.
:- type featureConj --> sub ; coo.

:- type tree -->
       n(featureN, string, string)
       ; en(featureEN, string, string)
       ; adj(featureAdj, string, string)
       ; v(featureV, string, string)
       ; adv(featureAdv, string)
       ; pron(featurePron, string)
       ; det(featureDet, string)
       ; cnj(featureConj, string)
       ; prep(string)
       ; infp(string)
       ; intj(string)
       ; pfx(string)
       ; sfx(string)
       ; pun(string)
       ; pul(string)
       ; pur(string)
       ; puq(string)
       ; tagerr(string, string)
       ; s(tree, tree, tree, tree)
       ; np(tree, tree, tree, tree, tree)
       ; detp(tree, tree, tree)
       ; advp(tree, tree)
       ; ap(tree, tree, tree)
       ; pp(tree, tree)
       ; inp(tree, tree, tree, tree, tree)
       ; seq(list(tree))
       ; (-) ; (+) ; '?'
       ; n(string) ; v(string) ; adj(string)
       ; en(string) ; det(string) ; cnj(string).

:- type trees == list(tree).

:- pred parse((trees :: in), (tree :: in),
```

```
              (pred tree :: (pred in) is semidet -> (pred in) is semidet),
              (trees :: in), (trees :: out)) is nondet.

:- implementation.

:- import_module int.
:- pred advp((tree :: out), (tree :: in), (int :: in),
             (trees :: in), (trees :: out)) is nondet.
:- pred ap((tree :: out), (tree :: in), (int :: in),
           (trees :: in), (trees :: out)) is nondet.
:- pred detp((tree :: out), (tree :: in), (int :: in),
             (trees :: in), (trees :: out)) is nondet.
:- pred inp((tree :: out), (tree :: in), (int :: in),
            (trees :: in), (trees :: out)) is nondet.
:- pred np((tree :: out), (tree :: in), (int :: in),
           (trees :: in), (trees :: out)) is nondet.
:- pred pp((tree :: out), (tree :: in), (int :: in),
           (trees :: in), (trees :: out)) is nondet.
:- pred s((tree :: out), (tree :: in), (int :: in),
          (trees :: in), (trees :: out)) is nondet.

:- pred seq((int :: in), (tree :: out), (tree :: in), (int :: in),
            (trees :: in), (trees :: out)) is nondet.

s(s(Tr1, Tr2, Tr3, Tr4), (+), RekTiefe) ->
        { RekTiefe2 is RekTiefe - 1, 0 =< RekTiefe2 },
        np(Tr1, (+), RekTiefe2),
        ([v(FeatV, Vollform, Stamm)], { Tr2 = v(FeatV, Vollform, Stamm) }),
        (np(Tr3, (+), RekTiefe2) ; { empty(Tr3, (-)) }),
        (seq(0, Tr4, (+), RekTiefe2) ; { empty(Tr4, (-)) }).
s(s(Tr1, Tr2, Tr3, Tr4), s(Pat1, Pat2, Pat3, Pat4), RekTiefe) ->
        { RekTiefe2 is RekTiefe - 1, 0 =< RekTiefe2 },
        np(Tr1, Pat1, RekTiefe2),
        ([v(FeatV, Vollform, Stamm)],
            { Tr2 = v(FeatV, Vollform, Stamm),
              (Pat2 = (+) ; Pat2 = Tr2
              ; Pat2 = core(v(FeatV, Vollform, Stamm))) }),
        (np(Tr3, Pat3, RekTiefe2) ; { empty(Tr3, Pat3) }),
        (seq(0, Tr4, Pat4, RekTiefe2) ; { empty(Tr4, Pat4) }).
s(Tr, '?', RekTiefe) ->
        s(Tr, (+), RekTiefe).

seq(0, seq([Tr | Trs]), (+), RekTiefe) ->
        pp(Tr, (+), RekTiefe),
        (seq(0, seq(Trs), (+), RekTiefe) ; { Trs = [] }).
seq(0, seq([Tr | Trs]), seq([Pat | Pats]), RekTiefe) ->
        pp(Tr, Pat, RekTiefe),
```

```
        (seq(0, seq(Trs), seq(Pats), RekTiefe) ; { Trs = [], Pats = [] }).
seq(0, Tr, '?', RekTiefe) ->
        seq(0, Tr, (+), RekTiefe).

np(np(Tr1, Tr2, Tr3, Tr4, Tr5), (+), RekTiefe) ->
        { RekTiefe2 is RekTiefe - 1, 0 =< RekTiefe2 },
        ((detp(Tr1, (+), RekTiefe2) ; { empty(Tr1, (-)) }),
            (ap(Tr2, (+), RekTiefe2) ; { empty(Tr2, (-)) }),
            ((seq(1, Tr3, (+), RekTiefe2) ; { empty(Tr3, (-)) }),
                [n(Feat, Voll, Stamm)], { Tr4 = n(Feat, Voll, Stamm) }
            ; { empty(Tr3, (-)) },
                [en(Feat, Voll, Stamm)], { Tr4 = en(Feat, Voll, Stamm) }
            ),
            (pp(Tr5, (+), RekTiefe2) ; { empty(Tr5, (-)) })
        ; { empty(Tr1, (-)) },
            { empty(Tr2, (-)) },
            { empty(Tr3, (-)) },
            ([pron(Feat, Pron)], { Tr4 = pron(Feat, Pron) }
            ; [en(Feat, Voll, Stamm)], { Tr4 = en(Feat, Voll, Stamm) }),
            { empty(Tr5, (-)) }
        ).

np(np(Tr1, Tr2, Tr3, Tr4, Tr5), np(Pat1, Pat2, Pat3, Pat4, Pat5), RekTiefe) ->
        { RekTiefe2 is RekTiefe - 1, 0 =< RekTiefe2 },
        ((detp(Tr1, Pat1, RekTiefe2) ; { empty(Tr1, Pat1) }),
            (ap(Tr2, Pat2, RekTiefe2) ; { empty(Tr2, Pat2) }),
            ((seq(1, Tr3, Pat3, RekTiefe2) ; { empty(Tr3, Pat3) }),
                [n(Feat, Voll, Stamm)], { Tr4 = n(Feat, Voll, Stamm),
                                          (Pat4 = (+) ; Pat4 = '?' ; Pat4 = Tr4
                                          ; Pat4 = core(n(Feat, Voll, Stamm))) }
            ; { empty(Tr3, Pat3) },
                [en(Feat, Voll, Stamm)], { Tr4 = en(Feat, Voll, Stamm),
                                           (Pat4 = (+) ; Pat4 = '?' ; Pat4 = Tr4
                                           ; Pat4 = core(en(Feat, Voll, Stamm))) }),
            (pp(Tr5, Pat5, RekTiefe2) ; { empty(Tr5, Pat5) })
        ; { empty(Tr1, Pat1) },
            { empty(Tr2, Pat2) },
            { empty(Tr3, Pat3) },
            ([pron(Feat, Pron)], { Tr4 = pron(Feat, Pron),
                                   (Pat4 = (+) ; Pat4 = '?' ; Pat4 = Tr4
                                   ; Pat4 = core(pron(Feat, Pron))) }
            ; [en(Feat, Voll, Stamm)], { Tr4 = en(Feat, Voll, Stamm),
                                         (Pat4 = (+) ; Pat4 = '?' ; Pat4 = Tr4
                                         ; Pat4 = core(en(Feat, Voll, Stamm))) }),
            { empty(Tr5, Pat5) }).
np(Tr, '?', RekTiefe) ->
        np(Tr, (+), RekTiefe).
```

```
seq(1, seq([Tr | Trs]), (+), RekTiefe) ->
        ([n(sg, Vs, Ss)], { Tr = n(sg, Vs, Ss) }),
        (seq(1, seq(Trs), (+), RekTiefe) ; { Trs = [] }).
seq(1, seq([Tr | Trs]), seq([Pat | Pats]), RekTiefe) ->
        ([n(sg, Vs, Ss)], { Tr = n(sg, Vs, Ss), Pat = n(sg, Vs, Ss) }),
        (seq(1, seq(Trs), seq(Pats), RekTiefe) ; { Trs = [], Pats = [] }).
seq(1, Tr, '?', RekTiefe) ->
        seq(1, Tr, (+), RekTiefe).


detp(detp(Tr1, Tr2, Tr3), (+), RekTiefe) ->
        { RekTiefe2 is RekTiefe - 1, 0 =< RekTiefe2 },
        ( ([det(FeatDet, Det)], { Tr1 = det(FeatDet, Det) }),
            { empty(Tr2, (-)) },
            { empty(Tr3, (-)) }
        ; ([det(FeatDet, Det)], { Tr1 = det(FeatDet, Det) } ; { empty(Tr1, (-)) }),
            (ap(Tr2, (+), RekTiefe2) ; { empty(Tr2, (-)) }),
            [n(gen, Voll, Stamm)], { Tr3 = n(gen, Voll, Stamm) }
        ; { empty(Tr1, (-)) },
            { empty(Tr2, (-)) },
            [en(gen, Voll, Stamm)], { Tr3 = en(gen, Voll, Stamm) }
        ).
detp(detp(Tr1, Tr2, Tr3), detp(Pat1, Pat2, Pat3), RekTiefe) ->
        { RekTiefe2 is RekTiefe - 1, 0 =< RekTiefe2 },
        (([det(FeatDet, Det)], { Tr1 = det(FeatDet, Det),
                                 (Pat1 = (+) ; Pat1 = '?' ; Pat1 = Tr1
                                 ; Pat1 = core(det(FeatDet, Det))) }),
            { empty(Tr2, Pat2) },
            { empty(Tr3, Pat3) }
        ; ([det(FeatDet, Det)], { Tr1 = det(FeatDet, Det),
                                  (Pat1 = (+) ; Pat1 = '?' ; Pat1 = Tr1
                                  ; Pat1 = core(det(FeatDet, Det))) }
          ; { empty(Tr1, Pat1) }),
            (ap(Tr2, Pat2, RekTiefe2) ; { empty(Tr2, Pat2) }),
            [n(gen, Voll, Stamm)], { Tr3 = n(gen, Voll, Stamm),
                                     (Pat3 = (+) ; Pat3 = '?' ; Pat3 = Tr3
                                     ; Pat3 = core(n(gen, Voll, Stamm))) }
        ; { empty(Tr1, Pat1) },
            { empty(Tr2, Pat2) },
            [en(gen, Voll, Stamm)], { Tr3 = en(gen, Voll, Stamm),
                                      (Pat3 = (+) ; Pat3 = '?' ; Pat3 = Tr3
                                      ; Pat3 = core(en(gen, Voll, Stamm))) }
        ).
detp(Tr, '?', RekTiefe) ->
        detp(Tr, (+), RekTiefe).
```

```
pp(pp(Tr1, Tr2), (+), RekTiefe) ->
        { RekTiefe2 is RekTiefe - 1, 0 =< RekTiefe2 },
        ([prep(Prep)], { Tr1 = prep(Prep) }),
        np(Tr2, (+), RekTiefe2).
pp(pp(Tr1, Tr2), pp(Pat1, Pat2), RekTiefe) ->
        { RekTiefe2 is RekTiefe - 1, 0 =< RekTiefe2 },
        ([prep(Prep)], { Tr1 = prep(Prep),
                         (Pat1 = (+) ; Pat1 = '?' ; Pat1 = Tr1
                         ; Pat1 = core(prep(Prep))) }),
        np(Tr2, Pat2, RekTiefe2).
pp(Tr, '?', RekTiefe) ->
        pp(Tr, (+), RekTiefe).


advp(advp(Tr1, Tr2), (+), RekTiefe) ->
        { RekTiefe2 is RekTiefe - 1, 0 =< RekTiefe2 },
        (seq(2, Tr1, (+), RekTiefe2) ; { empty(Tr1, (-)) }),
        [adv(FeatAdv, Adv)], { Tr2 = adv(FeatAdv, Adv) }.
advp(advp(Tr1, Tr2), advp(Pat1, Pat2), RekTiefe) ->
        { RekTiefe2 is RekTiefe - 1, 0 =< RekTiefe2 },
        (seq(2, Tr1, Pat1, RekTiefe2) ; { empty(Tr1, Pat1) }),
        [adv(FeatAdv, Adv)], { Tr2 = adv(FeatAdv, Adv),
                               (Pat2 = (+) ; Pat2 = '?' ; Pat2 = Tr2
                               ; Pat2 = core(adv(FeatAdv, Adv))) }.
advp(Tr, '?', RekTiefe) ->
        advp(Tr, (+), RekTiefe).

seq(2, seq([Tr | Trs]), (+), RekTiefe) ->
        ([adv(Fs, Advs)], { Tr = adv(Fs, Advs) }),
        (seq(2, seq(Trs), (+), RekTiefe) ; { Trs = [] }).
seq(2, seq([Tr | Trs]), seq([Pat | Pats]), RekTiefe) ->
        ([adv(Fs, Advs)], { Tr = adv(Fs, Advs), Pat = adv(Fs, Advs) }),
        (seq(2, seq(Trs), seq(Pats), RekTiefe) ; { Trs = [], Pats = [] }).
seq(2, Tr, '?', RekTiefe) ->
        seq(2, Tr, (+), RekTiefe).


ap(ap(Tr1, Tr2, Tr3), (+), RekTiefe) ->
        { RekTiefe2 is RekTiefe - 1, 0 =< RekTiefe2 },
        (advp(Tr1, (+), RekTiefe2) ; { empty(Tr1, (-)) }),
        (seq(3, Tr2, (+), RekTiefe2) ; { empty(Tr2, (-)) }),
        [adj(FeatAdj, VollAdj, StammAdj)],
            { Tr3 = adj(FeatAdj, VollAdj, StammAdj) }.
ap(ap(Tr1, Tr2, Tr3), ap(Pat1, Pat2, Pat3), RekTiefe) ->
        { RekTiefe2 is RekTiefe - 1, 0 =< RekTiefe2 },
        (advp(Tr1, Pat1, RekTiefe2) ; { empty(Tr1, Pat1) }),
```

```
        (seq(3, Tr2, Pat2, RekTiefe2) ; { empty(Tr2, Pat2) }),
        [adj(FeatAdj, VollAdj, StammAdj)],
            { Tr3 = adj(FeatAdj, VollAdj, StammAdj),
              (Pat3 = (+) ; Pat3 = '?' ; Pat3 = Tr3
              ; Pat3 = core(adj(FeatAdj, VollAdj, StammAdj))) }.
ap(Tr, '?', RekTiefe) ->
        ap(Tr, (+), RekTiefe).

seq(3, seq([Tr | Trs]), (+), RekTiefe) ->
        ([adj(Fs, Vs, Ss)], { Tr = adj(Fs, Vs, Ss) }),
        (seq(3, seq(Trs), (+), RekTiefe) ; { Trs = [] }).
seq(3, seq([Tr | Trs]), seq([Pat | Pats]), RekTiefe) ->
        ([adj(Fs, Vs, Ss)], { Tr = adj(Fs, Vs, Ss), Pat = adj(Fs, Vs, Ss) }),
        (seq(3, seq(Trs), seq(Pats), RekTiefe) ; { Trs = [], Pats = [] }).
seq(3, Tr, '?', RekTiefe) ->
        seq(3, Tr, (+), RekTiefe).


inp(inp(Tr1, Tr2, Tr3, Tr4, Tr5), (+), RekTiefe) ->
        { RekTiefe2 is RekTiefe - 1, 0 =< RekTiefe2 },
        ((advp(Tr1, (+), RekTiefe2) ; { empty(Tr1, (-)) }),
            ([v(inf, VollV, StammV)], { Tr2 = v(inf, VollV, StammV) }),
            { empty(Tr3, (-)) },
            (advp(Tr4, (+), RekTiefe2) ; { empty(Tr4, (-)) }),
            ((np(Tr5, (+), RekTiefe2) ; ap(Tr5, (+), RekTiefe2)
             ; pp(Tr5, (+), RekTiefe2) ; advp(Tr5, (+), RekTiefe2))
            ; { empty(Tr5, (-)) })
        ; { empty(Tr1, (-)) },
            ([v(inf, VollV, StammV)], { Tr2 = v(inf, VollV, StammV) }),
            ap(Tr3, (+), RekTiefe2),
            (advp(Tr4, (+), RekTiefe2) ; { empty(Tr4, (-)) }),
            ((np(Tr5, (+), RekTiefe2) ; pp(Tr5, (+), RekTiefe2))
            ; { empty(Tr5, (-)) })
        ).

inp(inp(Tr1, Tr2, Tr3, Tr4, Tr5), inp(Pat1, Pat2, Pat3, Pat4, Pat5), RekTiefe) ->
        { RekTiefe2 is RekTiefe - 1, 0 =< RekTiefe2 },
        ((advp(Tr1, Pat1, RekTiefe2) ; { empty(Tr1, Pat1) }),
            ([v(inf, VollV, StammV)], { Tr2 = v(inf, VollV, StammV),
                                        (Pat2 = (+) ; Pat2 = '?' ; Pat2 = Tr2
                                        ; Pat2 = core(v(inf, VollV, StammV))) }),
            { empty(Tr3, Pat3) },
            (advp(Tr4, Pat4, RekTiefe2) ; { empty(Tr4, Pat4) }),
            ((np(Tr5, Pat5, RekTiefe2) ; ap(Tr5, Pat5, RekTiefe2)
             ; pp(Tr5, Pat5, RekTiefe2) ; advp(Tr5, Pat5, RekTiefe2))
            ; { empty(Tr5, Pat5) })
        ; { empty(Tr1, Pat1) },
```

```
            ([v(inf, VollV, StammV)], { Tr2 = v(inf, VollV, StammV),
                                      (Pat2 = (+) ; Pat2 = '?' ; Pat2 = Tr2
                                      ; Pat2 = core(v(inf, VollV, StammV))) }),
            ap(Tr3, Pat3, RekTiefe2),
            (advp(Tr4, Pat4, RekTiefe2) ; { empty(Tr4, Pat4) }),
            ((np(Tr5, Pat5, RekTiefe2) ; pp(Tr5, Pat5, RekTiefe2))
            ; { empty(Tr5, Pat5) })
        ).
inp(Tr, '?', RekTiefe) ->
        inp(Tr, (+), RekTiefe).


:- pred empty((tree :: out), (tree :: in)) is semidet.
empty((-), (-)).
empty((-), '?').

:- func head(tree) = tree is semidet.
head(s(_1, V, _3, _4)) = head(V).
head(np(_1, _2, _3, N, _5)) = head(N).
head(detp(D, _2, _3)) = head(D).
head(pp(_1, N)) = head(N).
head(advp(_1, Adv)) = head(Adv).
head(ap(_1, _2, A)) = head(A).
head(inp(_1, V, _3, _4, _5)) = head(V).

head(n(F,V,S)) = n(F,V,S).
head(en(F,V,S)) = en(F,V,S).
head(adj(F,V,S)) = adj(F,V,S).
head(v(F,V,S)) = v(F,V,S).
head(adv(F,S)) = adv(F,S).
head(pron(F,S)) = pron(F,S).
head(det(F,D)) = det(F,D).
head(cnj(F,S)) = cnj(F,S).
head(prep(P)) = prep(P).
head(infp(P)) = infp(P).

:- func core(tree) = tree is semidet.
core(n(_F, _V, S)) = n(S).
core(en(_F, _V, S)) = en(S).
core(adj(_F, _V, S)) = adj(S).
core(v(_F, _V, S)) = v(S).
core(det(F, D)) = det(D).
core(cnj(_F, S)) = cnj(S).

:- type startcat --> s ; np ; ap ; pp.

:- pred start((startcat :: in),
```

```
               (pred(tree, tree, int, trees, trees)
               :: out(pred(out, in, in, in, out) is nondet))
              ) is semidet.
:- pred startcats((list(startcat) :: out)) is det.

start(ap, (pred((Tree :: out), (PatTree :: in), (RekDepth :: in),
                (Leaves :: in), (RestLeaves :: out)) is nondet
          :- ap(Tree, PatTree, RekDepth, Leaves, RestLeaves))).
start(np, (pred((Tree :: out), (PatTree :: in), (RekDepth :: in),
                (Leaves :: in), (RestLeaves :: out)) is nondet
          :- np(Tree, PatTree, RekDepth, Leaves, RestLeaves))).
start(pp, (pred((Tree :: out), (PatTree :: in), (RekDepth :: in),
                (Leaves :: in), (RestLeaves :: out)) is nondet
          :- pp(Tree, PatTree, RekDepth, Leaves, RestLeaves))).
start(s, (pred((Tree :: out), (PatTree :: in), (RekDepth :: in),
               (Leaves :: in), (RestLeaves :: out)) is nondet
          :- s(Tree, PatTree, RekDepth, Leaves, RestLeaves))).

startcats([s, np, pp, ap]).

parse([Leaf | Leaves], PatTree, Filter, Acc, Trees) :-
        startcats(StartCats),
        list__member(Cat, StartCats),
        start(Cat, ParsePred),
        list__length([Leaf | Leaves], InputLength),
        RekDepth = 2 + InputLength,
        ParsePred(Tree, PatTree, RekDepth, [Leaf | Leaves], RestLeaves),
        Filter(Tree)
        -> NewAcc = [Tree | Acc], parse(RestLeaves, PatTree, Filter, NewAcc, Trees)
        ; parse(Leaves, PatTree, Filter, Acc, Trees).

parse([], _PatTree, _Filter, Acc, Trees) :- Trees = Acc.

:- end_module grammatik.
```

Damit kann man eine lexikalisch korrekt markierte Eingabe syntaktisch analysieren. Es fehlen noch die Angaben der Schreibfunktionen für Bäume.

Entwurfsprobleme

Bemerkung 3.14 Die lexikalischen Kategorien führen zu einem m.M. nicht befriedigend auflösbaren Konflikt.

(i) Wenn man auf einer strikten Trennung zwischen Tagger und Grammatik besteht, sollten die lexikalischen Kategorien der Grammatik mit dem Tagger durch Regelschemata

$\langle$*lexikalisches Regelschema*$\rangle\equiv$
```
lexcat -> [tag(Merkmale,Wortform)].
```

verbunden sein. Dann enthält die Deklaration des Baumtyps eigenständige Konstruktoren wie

⟨*Lexikalische Kategorien als Baumkonstruktoren*⟩≡
```
:- type tree --> lexcat(Tagger__tag) ; ... .
```

Das ist gut, da der Typ der Bäume unabhängig von den konkreten Tags definiert ist, und Iterationen über Wortkategorien werden einfach, z.B. `*(lexcat1 ; lexcat2)`. Einen Namenszusammenhang zwischen lexikalischen Kategorien und Tags braucht man aber auf jeden Fall, also kann man auch gleich die Merkmale vererben:

⟨*lexikalisches Regelschema*⟩+≡
```
n -> [n(Merkmal,Vollform,Stammform)].
n(Merkmal) -> [n(Merkmal,Vollform,Stammform)].
```

Als Baumkonstruktor braucht man dann `n(featureN,Tagger__tag)`. Der Nachteil dieser Kodierung ist, daß man bei Mehrwortlexemen und ähnlichen Konstruktionen künstliche lexikalische Kategorien für die einzelnen Teile einführen muß.

(ii) Man verzichtet ganz auf lexikalischen Regeln, indem man die Tags des Taggers als Terminale der Grammatik erlaubt und in den Regeln als Listen kennzeichnet, z.B.

⟨*Tags als lexikalische Kategorien*⟩≡
```
s(final) -> [prep("um")], np, [infp("zu")], [v(inf,Vollform,Stamm)].
```

Dann muß man aber in der Baumdarstellung entweder diese Terminale durch einen uniformen Baumkonstruktor

⟨*Uniformer Konstruktor für lexikalische Kategorien*⟩≡
```
:- type tree --> lex(Tagger__tag) ; ...
```

verpacken. Das ist eine uniforme Lösung, und programmiertechnisch vielleicht die beste; zumindest für die Eingabe von Mustern hat sie den Nachteil, daß man dabei jedes lexikalische Element mit `lex` einpacken müßte (außer, wenn in den Mustern nur Stammformen erlaubt werden).

Die andere Möglichkeit ist, jeden Tag-Konstruktor als Baumkonstruktor zulassen:

⟨*Tagkonstruktoren als Baumkonstruktoren*⟩≡
```
:- type tree --> n(featureN,string,string) ; ...
```

wie es in der Beispielgrammatik gemacht wird. Bei jeder Änderung des Typs `tag` muß dann – da Mercury keine Subtypen `tag < tree` erlaubt – die Definition des Typs `tree` angepaßt werden, was nicht gerade modular ist.

4 Definition des Tokenizers

Im folgenden Modul `tokenize` definieren wir gemäß den Vorgaben der Bibliotheksdatei `merxury-extras-0.11.0/lex/lex.m` einen Typ `token` und eine Liste `lexemes`, aus denen im folgenden Abschnitt mit Hilfe von `lex__init` und `lex__start` ein Tokenizer erzeugt wird (vgl. `suche__reader` und Abschnitt 5.1.4).

Dieser Tokenizer zerlegt den Eingabestrom in eine Tokenfolge, die mit `token_to_lemmata` und `lemmata_to_trees` in eine Folge von atomaren Bäumen im Sinne der Grammatik umgewandelt werden können. Diese atomaren Bäume entsprechen den getaggten Eingabewörtern, wobei lediglich XML-annotierte Wörter in Mercury-Terme umgewandelt sind.

In der Schnittstelle des Tokenizers muß die Grammatik (oder zumindest die Definition der Typen `tree` und der Merkmaltypen dafür) geladen werden.[22] Außerdem braucht man den Lexer von Mercury, damit die Prädikate des Tokenizers deklariert werden können.

Die Definition eines Lexers ist in der neuen Version von Mercury etwas einfacher.[23] Sie läßt sich auch mit `mercury-0.10.1` verwenden, wenn man im `Mmakefile` festlegt, wo die entsprechenden Dateien liegen:

⟨*Mmakefile*⟩≡
```
# The following stuff tells Mmake to use the lex library:
LEX_DIR = mercury-extras-0.11.0/lex

VPATH = $(LEX_DIR):$(MMAKE_VPATH)
MCFLAGS = -I$(LEX_DIR) $(EXTRA_MCFLAGS)
MLFLAGS = -R$(LEX_DIR) $(EXTRA_MLFLAGS) \
          -L$(LEX_DIR)
MLLIBS = -llex $(EXTRA_MLLIBS)
C2INITARGS = $(LEX_DIR)/lex.init
```

Die Definition des Lexers erfolgt dann in der Datei `token.m`.

⟨*token.m*⟩≡
```
% Uses mercury-extras-0.11.0/lex/lex.m

:- module token.                   % Namensübergabe?

:- interface.

:- import_module list, string, lex.      % Bibliotheksmodule.
:- import_module grammatik.              % Eigene Module.

:- type token
```

[22] Klarer wäre es, wenn umgekehrt die Grammatik aus dem Tokenizer den Typ der Lemmata erhielte, d.h. die elementaren Parsebäume. Dann muß man aber im Parser Typumwandlungen vornehmen, wenn man einen Eingabesatz analysiert.

[23] Alexander Völtz hat dankenswerterweise die ursprünglich für `mercury-extras-0.10.1/lex` geschriebene Definition an das von `mercury-extras-0.11.0/lex` erwartete Format angepaßt.

```
    -->
            n(featureN)
    ;       adj(featureAdj)
    ;       adv(featureAdv)
    ;       v(featureV)
    ;       aux(featureV)
    ;       mdl
    ;       pron(featurePron)
    ;       det(featureDet)
    ;       det
    ;       cnj(featureConj)
    ;       cnj
    ;       prep
    ;       infp
    ;       pfx
    ;       sfx
    ;       en(featureEN)
    ;       pun
    ;       pul
    ;       pur
    ;       puq
    ;       intj
    ;       mwu
    ;       eos
    ;       word(string)
    ;       utag(string)    % unknown tag: tag not known or handled
    ;       whspace
    .

:- func lexemes = list(lexeme(token)). % 0.11.0
```

Die Wahl der Token und ihre Bedeutung ist durch die Tags des lexikalisch markierten Texts (vgl. `refcorp.tag`) vorgegeben. Merkmalkombinationen im Eingabetext werden auf ein einfaches Merkmal reduziert, da nicht klar war, welche Merkmalkombinationen erlaubt sind. Außerdem verwendet die Beispielgrammatik (noch) keine Merkmale an den syntaktischen Kategorien, abgesehen von den Nominalphrasen.[24]

⟨*token.m*⟩+≡

```
:- pred tokens_to_trees(list(token)::in,list(tree)::out) is det.

:- type lemma --> lem(token,string,string).  % Vollform, Stammform

:- pred tokens_to_lemmata(list(token)::in,list(lemma)::in,
                          list(lemma)::out) is det.
```

[24]Mit Merkmalen an den syntaktischen Kategorien könnte man die Treffergenauigkeit erhöhen. Der Hauptnachteil von Merkmalen an den Kategorien ist, daß dann die Merkmale auch in den Mustern verwendet werden müssen und diese dadurch zu fein und umständlich werden.

⟨*token.m*⟩+≡
```
:- implementation.

:- func word = regexp.
word = (+(alphanum \/
          any("ßäö:;.-,?!/”/""¼½$)(][¡&âéèø° + ∗ = _| %”))).

:- func utag = regexp. % unknown tags

utag = ("<lem cat=""n"" mor="":m:9"">"      % 0.11.0
       \/ "<lem cat=""n"" mor="":e:9"">"
       \/ "<lem cat=""n"" mor="":b"">"
       \/ "<lem cat=""n:e"" mor="""">"
       \/ "<lem cat=""kf"" mor="""">"
       \/ "<lem cat=""kf"" mor="":e"">"
       \/ "<lem cat=""-"" mor="""">"
       \/ "<lem cat=""v"" mor="":D:G"">"
       \/ "<lem cat=""adj"" mor="":5:g"">"
       \/ "<lem cat=""adv"" mor="""">"
       \/ "<lem cat=""xinc"" mor="""">"
       \/ "<lem cat=""pron"" mor="":j:g"">"
       \/ "<lem cat=""en"" mor=""en"">"
       ).

lexemes =
   [("<sen" ++ junk                    -> return(whspace)),
    ("<mwu" ++ junk                    -> return(mwu)),
    ("</mwu>"                          -> return(whspace)),
    ("<tok>"                           -> return(whspace)),
    ("</tok>"                          -> return(whspace)),
    ("<sur>"                           -> return(whspace)),
    ("</sur>"                          -> return(whspace)),
    ("</lem>"                          -> return(whspace)),
    ("<lem cat=""bos"" mor="""">"      -> return(whspace)),
    ("<READIN_CHUNK/>"                 -> return(whspace)),
    (word                              -> (func(W) = word(W))),
    (utag                              -> (func(W) = utag(W))),
    (nl                                -> return(whspace)),
    ("<lem cat=""n"" mor="":e"">"      -> return(n(sg))),
    ("<lem cat=""n"" mor="":m"">"      -> return(n(pl))),
    ("<lem cat=""n"" mor="":m:9"">"    -> return(n(pl))),
    ("<lem cat=""n"" mor="":e:m"">"    -> return(n(sp))),
    ("<lem cat=""n"" mor="":g"">"      -> return(n(gen))),
    ("<lem cat=""n"" mor="":e:g"">"    -> return(n(gen))),
    ("<lem cat=""n"" mor="":m:g"">"    -> return(n(gen))),
    ("<lem cat=""n"" mor="":4"">"      -> return(n(num))),
    ("<lem cat=""n"" mor="":4:m"">"    -> return(n(num))),
```

```
("<lem cat=""n"" mor="":4:e"">"     -> return(n(num))),
("<lem cat=""n"" mor="":4:9:m"">"   -> return(n(num))),
("<lem cat=""n"" mor="":4:9:e:m"">" -> return(n(num))),
("<lem cat=""en"" mor="""">"        -> return(en(sg))),
("<lem cat=""en"" mor="":g"">"      -> return(en(gen))),
("<lem cat=""adj"" mor="":b"">"     -> return(adj(b))),
("<lem cat=""adj"" mor="":c"">"     -> return(adj(c))),
("<lem cat=""adj"" mor="":s"">"     -> return(adj(s))),
("<lem cat=""adj"" mor="":4"">"     -> return(adj(crd))),
("<lem cat=""adj"" mor="":5"">"     -> return(adj(ord))),
("<lem cat=""adj"" mor="":B"">"     -> return(adj(prd))),
("<lem cat=""adv"" mor="":b"">"     -> return(adv(b))),
("<lem cat=""adv"" mor="":c"">"     -> return(adv(c))),
("<lem cat=""adv"" mor="":s"">"     -> return(adv(s))),
("<lem cat=""adv"" mor="":v"">"     -> return(adv(part))),
("<lem cat=""adv"" mor="":w"">"     -> return(adv(wh))),
("<lem cat=""adv"" mor="":b:g"">"   -> return(adv(b))),
("<lem cat=""adv"" mor="":b:c:s"">" -> return(adv(b))),
("<lem cat=""v"" mor="":I"">"       -> return(v(inf))),
("<lem cat=""v"" mor="":G"">"       -> return(v(pres))),
("<lem cat=""v"" mor="":G3e"">"     -> return(v(prsg))),
("<lem cat=""v"" mor="":V"">"       -> return(v(past))),
("<lem cat=""v"" mor="":C"">"       -> return(v(ger))),
("<lem cat=""v"" mor="":P"">"       -> return(v(part))),
("<lem cat=""v"" mor="":V:P"">"     -> return(v(part))),
("<lem cat=""v"" mor="":h"">"       -> return(aux(pres))),
("<lem cat=""v"" mor="":I:h"">"     -> return(aux(inf))),
("<lem cat=""v"" mor="":G:h"">"     -> return(aux(pres))),
("<lem cat=""v"" mor="":G3e:h"">"   -> return(aux(prsg))),
("<lem cat=""v"" mor="":V:h"">"     -> return(aux(past))),
("<lem cat=""v"" mor="":C:h"">"     -> return(aux(ger))),
("<lem cat=""v"" mor="":P:h"">"     -> return(aux(part))),
("<lem cat=""v"" mor="":D"">"       -> return(mdl)),
("<lem cat=""v"" mor="":D:G"">"     -> return(mdl)),
("<lem cat=""v"" mor="":D:h"">"     -> return(mdl)),
("<lem cat=""pron"" mor="""">"      -> return(pron(def))),
("<lem cat=""pron"" mor="":g"">"    -> return(pron(def))),
("<lem cat=""pron"" mor="":j"">"    -> return(pron(idef))),
("<lem cat=""pron"" mor="":q"">"    -> return(pron(pers))),
("<lem cat=""pron"" mor="":l"">"    -> return(pron(refl))),
("<lem cat=""pron"" mor="":w"">"    -> return(pron(wh))),
("<lem cat=""pron"" mor="":R"">"    -> return(pron(rel))),
("<lem cat=""pron"" mor="":R:w"">"  -> return(pron(rel))),
("<lem cat=""pron"" mor="":o"">"    -> return(pron(poss))),
("<lem cat=""det"" mor="":w"">"     -> return(det(wh))),
("<lem cat=""det"" mor="":R"">"     -> return(det(rel))),
("<lem cat=""det"" mor="":o"">"     -> return(det(poss))),
```

```
    ("<lem cat=""det"" mor="":g"">"      -> return(det)),
    ("<lem cat=""det"" mor="""">"        -> return(det(def))),
    ("<lem cat=""cnj"" mor="":K"">"      -> return(cnj(coo))),
    ("<lem cat=""cnj"" mor="":S"">"      -> return(cnj(sub))),
    ("<lem cat=""cnj"" mor="":S:w"">"    -> return(cnj(sub))),
    ("<lem cat=""cnj"" mor="""">"        -> return(cnj)),
    ("<lem cat=""prep"" mor="""">"       -> return(prep)),
    ("<lem cat=""pfx"" mor="""">"        -> return(pfx)),
    ("<lem cat=""sfx"" mor="""">"        -> return(sfx)),
    ("<lem cat=""infp"" mor="""">"       -> return(infp)),
    ("<lem cat=""pun"" mor="""">"        -> return(pun)),
    ("<lem cat=""pur"" mor="""">"        -> return(pur)),
    ("<lem cat=""pul"" mor="""">"        -> return(pul)),
    ("<lem cat=""puq"" mor="""">"        -> return(puq)),
    ("<lem cat=""intj"" mor="""">"       -> return(intj)),
    ("<lem cat=""eos"" mor="""">"        -> return(eos)),
    ("</sen>"                            -> return(eos))
].

tokens_to_trees(Tokens,Trees) :-
        tokens_to_lemmata(Tokens,[],Lemmata),
        list__reverse(Lemmata,LemmataRev),
        lemmata_to_trees(LemmataRev,[],Trees).

tokens_to_lemmata(Tokens,Acc,Lemmata) :-
        ( Tokens = [] ->
            Lemmata = Acc
        ; Tokens = [eos,word(EOS),pun,word(EOS2)|Rest] ->
            tokens_to_lemmata(Rest,[lem(eos,EOS,EOS2)|Acc],Lemmata)
        ; Tokens = [word(_Surf),_Token,mwu|Rest] ->
            tokens_to_lemmata(Rest,Acc,Lemmata)
        ; Tokens = [whspace|Rest] ->
            tokens_to_lemmata(Rest,Acc,Lemmata)
        ; Tokens = [word(Lem),Token,word(Surf)|Rest] ->
            tokens_to_lemmata(Rest,[lem(Token,Surf,Lem)|Acc],Lemmata)
        ; Tokens = [word(Lem),utag(Tag),word(Surf)|Rest] ->
            tokens_to_lemmata(Rest,[lem(utag(Tag),Surf,Lem)|Acc],Lemmata)
        ; Tokens = [Token,word(Surf)|Rest], Token \= word(_)  ->
            tokens_to_lemmata(Rest,[lem(Token,Surf,Surf)|Acc],Lemmata)
        ; Lemmata = Acc ).

:- pred lemmata_to_trees(list(lemma)::in,list(tree)::in,list(tree)::out) is det.

lemmata_to_trees([],Acc,Acc).
lemmata_to_trees([lem(n(F),Surf,Stamm)|Lemmata],Acc,Trees) :-
        lemmata_to_trees(Lemmata,[n(F,Surf,Stamm)|Acc],Trees).
lemmata_to_trees([lem(adj(F),Surf,Stamm)|Lemmata],Acc,Trees) :-
```

```
        lemmata_to_trees(Lemmata,[adj(F,Surf,Stamm)|Acc],Trees).
lemmata_to_trees([lem(adv(F),Surf,_)|Lemmata],Acc,Trees) :-
        lemmata_to_trees(Lemmata,[adv(F,Surf)|Acc],Trees).
lemmata_to_trees([lem(v(F),Surf,Stamm)|Lemmata],Acc,Trees) :-
        lemmata_to_trees(Lemmata,[v(F,Surf,Stamm)|Acc],Trees).
lemmata_to_trees([lem(aux(F),Surf,Stamm)|Lemmata],Acc,Trees) :-
        lemmata_to_trees(Lemmata,[aux(F,Surf,Stamm)|Acc],Trees).
lemmata_to_trees([lem(mdl,Surf,Stamm)|Lemmata],Acc,Trees) :-
        lemmata_to_trees(Lemmata,[mdl(Surf,Stamm)|Acc],Trees).
lemmata_to_trees([lem(pron(F),Surf,_)|Lemmata],Acc,Trees) :-
        lemmata_to_trees(Lemmata,[pron(F,Surf)|Acc],Trees).
lemmata_to_trees([lem(det(F),Surf,_)|Lemmata],Acc,Trees) :-
        lemmata_to_trees(Lemmata,[det(F,Surf)|Acc],Trees).
lemmata_to_trees([lem(cnj(F),Surf,_)|Lemmata],Acc,Trees) :-
        lemmata_to_trees(Lemmata,[cnj(F,Surf)|Acc],Trees).
lemmata_to_trees([lem(det,Surf,_)|Lemmata],Acc,Trees) :-
        lemmata_to_trees(Lemmata,[det(Surf)|Acc],Trees).
lemmata_to_trees([lem(cnj,Surf,_)|Lemmata],Acc,Trees) :-
        lemmata_to_trees(Lemmata,[cnj(Surf)|Acc],Trees).
lemmata_to_trees([lem(prep,Surf,_)|Lemmata],Acc,Trees) :-
        lemmata_to_trees(Lemmata,[prep(Surf)|Acc],Trees).
lemmata_to_trees([lem(infp,Surf,_)|Lemmata],Acc,Trees) :-
        lemmata_to_trees(Lemmata,[infp(Surf)|Acc],Trees).
lemmata_to_trees([lem(pfx,Surf,_)|Lemmata],Acc,Trees) :-
        lemmata_to_trees(Lemmata,[pfx(Surf)|Acc],Trees).
lemmata_to_trees([lem(sfx,Surf,_)|Lemmata],Acc,Trees) :-
        lemmata_to_trees(Lemmata,[sfx(Surf)|Acc],Trees).
lemmata_to_trees([lem(pun,Surf,_)|Lemmata],Acc,Trees) :-
        lemmata_to_trees(Lemmata,[pun(Surf)|Acc],Trees).
lemmata_to_trees([lem(pul,Surf,_)|Lemmata],Acc,Trees) :-
        lemmata_to_trees(Lemmata,[pul(Surf)|Acc],Trees).
lemmata_to_trees([lem(pur,Surf,_)|Lemmata],Acc,Trees) :-
        lemmata_to_trees(Lemmata,[pur(Surf)|Acc],Trees).
lemmata_to_trees([lem(puq,Surf,_)|Lemmata],Acc,Trees) :-
        lemmata_to_trees(Lemmata,[puq(Surf)|Acc],Trees).
lemmata_to_trees([lem(intj,Surf,_)|Lemmata],Acc,Trees) :-
        lemmata_to_trees(Lemmata,[intj(Surf)|Acc],Trees).
lemmata_to_trees([lem(en(F),Surf,Stamm)|Lemmata],Acc,Trees) :-
        lemmata_to_trees(Lemmata,[en(F,Surf,Stamm)|Acc],Trees).
lemmata_to_trees([lem(eos,Surf,_)|Lemmata],Acc,Trees) :-
        lemmata_to_trees(Lemmata,[pun(Surf)|Acc],Trees).
lemmata_to_trees([lem(word(_),_Surf,_)|Lemmata],Acc,Trees) :-
        lemmata_to_trees(Lemmata,Acc,Trees).
lemmata_to_trees([lem(utag(_),Surf,Tag)|Lemmata],Acc,Trees) :-
        lemmata_to_trees(Lemmata,[tagerr(Surf,Tag)|Acc],Trees).
% Nur, um das Prädikat deterministisch zu machen:
lemmata_to_trees([lem(whspace,_Surf,_Tag)|Lemmata],Acc,Trees) :-
```

```
        lemmata_to_trees(Lemmata,Acc,Trees).
lemmata_to_trees([lem(mwu,_Surf,_Tag)|Lemmata],Acc,Trees) :-
        lemmata_to_trees(Lemmata,Acc,Trees).

:- end_module token.
```

Bemerkung 4.1 Mehrwortlexeme sind in der Eingabedatei wie im Beispiel

⟨*Mehrwortlexeme*⟩≡

```
<mwu len="2">
<lem cat="n" mor=":e:m">per cent</lem>
</mwu>
<tok><sur>per</sur><lem cat="n" mor=":e:m">per</lem></tok>
<tok><sur>cent</sur><lem cat="n" mor=":e:m">cent</lem></tok>
```

dargestellt. Damit der Lexer, der z.B. bei `per cent` auf das Leerzeichen stößt, keinen Fehler meldet, sind Leerzeichen in der Definition von `word` in `token.m` erlaubt worden, was eigentlich bei Stammformen nicht sinnvoll ist. Mehrwortlexeme werden vom Tokenizer noch nicht an den Parser weitergeleitet, da weder die Kategorie des Mehrwortlexems noch die der Teilwörter überzeugend aussehen. (Der Tokenizer sollte nicht nur die `mwu`-Zeilen wegwerfen, sondern sich die darin genannte Anzahl der Teile merken und entsprechend viele folgende Token ebenfalls wegwerfen; dann könnte der Parser mit der Kategorie des Mehrwortlexems vielleicht weiterarbeiten.)

5 Erzeugung des Suchprogramms aus einer Grammatik

Zur Konstruktion des Suchprogramms sollen die Teile, die vom Tokenizer und der Grammatik abhängen, von den übrigen konstanten Teilen getrennt werden. (Aus Zeitgründen ist die Beschreibung im Folgenden etwas rudimentär geblieben.)

5.1 Das Suchprogramm

Das Suchprogramm übernimmt gewisse Hilfsfunktionen aus den vom Grammatikschreiber erstellenden Dateien.

Die Verteilung der Optionen auf den Tokenizer, die Suchverzweigung, und die Ausgabe sollte besser organisiert werden.

(i) Der Tokenizer erhält folgende der Optionen als Argumente:

⟨*Tokenizer-Optionen*⟩≡

```
Option: tokenize(bool) | lemmatize(bool)
Verbosity:  verbose(bool)
```

(ii) Die Ausgabe wird gesteuert durch:

⟨*Ausgabeoptionen*⟩≡

```
verbose(bool) | print_level(string)
```

Davon betrifft das `PrintLevel` nur den Parser; Tokenizer und Lemmatisierer benutzen nur `Verbosity`.

(iii) Das Suchprogramm unterscheidet nach

⟨*Suchoptionen*⟩≡

```
Option: -
Verbosity: verbose(bool)
File: file
```

⟨*suche.m*⟩≡

```
:- module suche.

:- interface.

:- import_module io.

:- pred main(io__state::di,io__state::uo) is det.


:- implementation.

:- import_module char, string, int, exception, array, list, assoc_list,
      std_util, term, term_io, pprint, getopt, bool, lex.

:- import_module grammatik, filter, token.
:- import_module grammatik:muster. % erklaert type query_code, pseudo_queries.

:- pred short_option(character::in, option::out) is semidet.
:- pred long_option(string::in, option::out) is semidet.
:- pred option_default(option::out, option_data::out) is nondet.

:- type option
      -->    tokenize
      ;       lemmatize
      ;       parse
      ;       file
      ;       lxfile
      ;       pattern
      ;       filter
      ;       semantic
      ;       verbose
      ;       print_level
      ;       interactive
      ;       help
      .

short_option('t', tokenize).
```

```
short_option('l', lemmatize).
short_option('a', parse).  % analyse
short_option('f', file).
short_option('x', lxfile). % logical predicates
short_option('p', pattern).
short_option('F', filter).
short_option('S', semantic).
short_option('v', verbose).
short_option('P', print_level).
short_option('i', interactive).
short_option('?', help).
short_option('h', help).

long_option("tokenize",  tokenize).
long_option("lemmatize", lemmatize).
long_option("analyze",   parse).
long_option("parse",     parse).
long_option("file",      file).
long_option("lxfile",    lxfile).
long_option("pattern",   pattern).
long_option("filter",    filter).
long_option("semantic",  semantic).
long_option("verbose",   verbose).
long_option("print_level", print_level).
long_option("interactive",   interactive).
long_option("help",      help).

option_default(tokenize, bool(bool:no)).
option_default(lemmatize, bool(bool:no)).
option_default(parse, bool(bool:yes)).
option_default(file, string("refcorp.anfang.tag")).
option_default(lxfile, string("praedikate.lx")).
option_default(pattern, string("defaultpattern")). % [np(+,?,-,+,-),[],(-)]
option_default(filter, accumulating(list:[])).     % [Name of a pred(tree,...) | Args...]
option_default(semantic, string("perform")).
option_default(verbose, bool(bool:no)).
option_default(print_level, string("flach")).
option_default(interactive, bool(bool:no)).
option_default(help, bool(bool:no)).
```

Da man bei den Optionen keine eigenen Datentypen vorgeben kann, wird mit `print_level` nur der Name eines Konstruktors des in `grammatik.m` definierten Datentyps `printlevel` übergeben. (Defaultmäßig wird jetzt die semantische Suche aufgerufen; es sollte aber wieder die syntaktische Suche werden.)

⟨*suche.m*⟩+≡

```
main ->
```

```
        io__set_exit_status(0),
        { OptionOps = option_ops(short_option,
                                 long_option,
                                 option_default) },
        io__command_line_arguments(Args),
        { getopt__process_options(OptionOps, Args, _AL, Maybe_Option_Table) },
        ( { Maybe_Option_Table = error(Message) },
            io__nl,
            io__print(Message), io__nl,
            usage
        ; { Maybe_Option_Table = ok(Option_Table),
            getopt__lookup_bool_option(Option_Table, verbose, Verbosity),
            getopt__lookup_string_option(Option_Table, print_level, Printer)
          },
            ( {grammatik__printer(Printer,PrintLevel)}
            -> ( {getopt__lookup_bool_option(Option_Table, help, yes)}
               -> usage
               ; {getopt__lookup_bool_option(Option_Table, tokenize, yes)}
               -> tokenizer(Verbosity,PrintLevel)
               ; {getopt__lookup_bool_option(Option_Table, lemmatize, yes)}
               -> lemmatizer(Verbosity,PrintLevel)
               ; { getopt__lookup_bool_option(Option_Table, interactive, no),
                   getopt__lookup_string_option(Option_Table, semantic, Predicate),
                   getopt__lookup_string_option(Option_Table, file, File),
                   getopt__lookup_string_option(Option_Table, lxfile, LxFile) }
               -> semantic_search(Verbosity,PrintLevel,Predicate,File,LxFile)
               ; { getopt__lookup_bool_option(Option_Table, interactive, Interactive),
                   getopt__lookup_string_option(Option_Table, file, File),
                   getopt__lookup_string_option(Option_Table, pattern, PatFile),
                   getopt__lookup_accumulating_option(Option_Table, filter, Filter_Args)
                 },
                   syntactic_search(Verbosity,PrintLevel,Interactive,File,
                                    PatFile,Filter_Args)
               )
            ; io__print("The option -P does not have a correct PrintLevel."),
                io__nl
            )
        ).

:- pred usage(io__state::di, io__state::uo) is det.
usage -->
        io__stderr_stream(StdErr),
        { list__append([
        "Usage: test_suche [options] -f TEXT-FILE -p PATTERN-FILE\n",
        "Current defaults:\n",
        "  TEXT-FILE    = `refcorp.anfang.tag'\n",
        "  PATTERN-FILE = `defaultpattern'\n",
```

```
"  -?, -h, -help\n",
"      Print this usage message.\n",
"  -t, -tokenize\n",
"      Tokenize tagged input line (to find bad characters etc.).\n",
"  -l, -lemmatize\n",
"      Tokenize and lemmatize tagged input (end token: </sen>).\n",
"  -f, -file\n",
"      Use the next string as filename for tagged input text.\n",
"  -p, -pattern\n",
"      Use the next string as filename for the search query.\n",
"  -i, -interactive\n",
"      Ask the user to provide a search- and argument-tree.\n",
"  -S, -semantic query\n",
"      use the next string as a semantic search predicate.\n",
"  -x, -lxfile\n",
"      use the next string as filename for the lexicon of predicates.\n",
"  -P, -print_level\n",
"      Use the next string to define the print level <flach|tief> of trees.\n",
"  -v, -verbose\n",
"      Print lemmatized input sentence.\n"],
            filter__usage,
            Usage) },
io__write_strings(StdErr, Usage),
io__set_exit_status(1).
```

Das Hauptprogramm ist eine Lese-Schleife, die mit einer Liste von Anfragen den Text durchsucht:

⟨*suche.m*⟩+≡

```
:- inst listskel(Inst) = bound( [] ; [Inst|listskel(Inst)] ).

:- pred reader(option::in,
            bool::in,
            printlevel::in,
            list(query_lex)::in(listskel(query_lex)),
            io__state::di,
            io__state::uo) is det.

reader(Option,Verbosity,PrintLevel,Queries) ->
        { Lexer  = lex__init(lexemes,
                             lex__read_from_stdin,
                             ignore(whspace)) },
        call((pred(IO0::di, IO::uo) is det :-
             State0 = lex__start(Lexer, IO0),
              tokenize_stdin(Option,Verbosity,PrintLevel,Queries,[],
                             State0, State),
              IO = lex__stop(State)
             )).
```

Um Datenterme, also Werte vom Typ `term(typ)`, in entsprechende Werte vom Typ `typ` umrechnen zu können, braucht man spezielle Leseprädikate, die mögliche Typambiguitäten der Konstruktoren und -Typvariablen beseitigen. Dazu dient:

⟨*suche.m*⟩+≡
```
:- pred read_coerce(pred(term(T),T)::in(pred(in,out) is semidet),
                    maybe(T)::out,
                    io__state::di, io__state::uo) is det.

% From term_io:
% :- type read_term(T) --> eof ; error(string, int) ; term(varset(T), term(T)).

read_coerce(Coercion,Result) ->
        term_io__read_term(Pattern),
        ( { Pattern = term(_VarSet,Term) },
            ({ Coercion(Term,Value) }
            -> { Result = yes(Value) }
            ; { Result = no }
            )
        ; { Pattern = error(Msg,N) },
            io__write(Msg),io__print(" in line "),io__write(N),
            { Result = no }
        ; { Pattern = eof },
            io__print("Error: file empty"),
            { Result = no }
        ).
```

Umwandlungsprädikate für verschiedene Typen folgen unten. (Es scheint nicht möglich zu sein, andere als Datenterme aus Dateien einzulesen. Daher können beim Einlesen keine Auswertungen vorgenommen werden.)

5.1.1 Semantische Suche

Bei der semantischen Suche ist der Suchbegriff der Name eines *logischen Prädikats*, typischerweise die infinite Form eines Vollverbs oder die Grundform eines Adjektivs – vorläufig noch nicht das Kopfnomen einer Stützverbkonstruktion.

Die möglichen logischen Prädikate werden einer Datei (mit Suffix `.lx` für Lexikon) entnommen, in der zu jedem Prädikat seine möglichen lexikalischen Darstellungen als Adjektiv, Nomen und Verb angegeben sind und eine Reihe syntaktischer Phrasenmuster, die eine der lexikalischen Darstellungsweisen mit syntaktischen Darstellungen der logischen Argumente des Prädikats wiedergeben.

⟨*suche.m*⟩+≡
```
:- pred semantic_search(bool::in,
                        printlevel::in,
                        string::in,
```

```
                     string::in,
                     string::in,
                     io__state::di,io__state::uo) is det.

semantic_search(Verbosity,PrintLevel,Praedikat,File,LxFile) ->
        io__print("\nSemantische Suche, Version 0.07"),
        % Call perl to filter LxFile with Praedikat and produce a list of Queries
        { find_patterns(Praedikat,LxFile,"TMP_Patterns",SysCall) },
        ({ Verbosity = yes }
        -> io__print("\nSystem-Aufruf: \n"),
            io__print(SysCall),io__nl
        ; []
        ),
        io__call_system(SysCall,SysCallResult),
        ( {SysCallResult = ok(0)}
        -> io__see("TMP_Patterns",OK), % open TMP_Patterns
            ( { OK = error(Message) },
                io__write_string(io__error_message(Message)),
                io__print(": "), io__print("TMP_Patterns"), io__nl
            ; { OK = ok },
                read_coerce(term_to_query_codes,QueryCodesRead),
                io__seen,              % close TMP_Patterns
                ({ QueryCodesRead = yes(QueryCodes),
                   QCs = list__map(muster__dispatch,QueryCodes),
                   list__condense(QCs,PsQueries),
                   translate_filternames(PsQueries,Queries),
                   not(Queries = [])
                 }
                -> search_with(parse,Verbosity,PrintLevel,Queries,File)
                ; io__print("\nNo patterns for this predicate available.\n")
                )
            )
        ; {SysCallResult = error(Msg1)}
        -> io__print(Msg1), io__nl
        ; []
        ).
```

Zuerst werden mit dem Suchbegriff `Praedikat` aus der Datei `File` die dazu angegebenen lexikalischen Darstellungen und Phrasenmuster herausgefiltert und in eine temporäre Datei `TMP_Patterns` geschrieben:

$\langle$*suche.m*$\rangle$+≡

```
:- pred find_patterns(string::in, string::in, string::in, string::out) is det.

find_patterns(Praedikat,File,TMP_Patterns,Call) :-
        string__to_upper(Praedikat,SUCHWORT),
        string__append_list(["/usr/bin/grep '^'",SUCHWORT,"'(' ",File,
```

```
                                  " | /usr/bin/perl lx2queries.pl > ",TMP_Patterns],
                                 Call).
```

Das geschieht durch ein Perl-Skript, das die vorgegebene Kodierung der syntaktischen Muster in `query_codes`, d.h. Tripel `{Mustername,Argumentnummern,Stammwörter}`, umrechnet:

⟨*lx2queries.pl*⟩≡

```
#!/usr/bin/perl
#
# Beispielzeile:
# GOOD(1: good entity) (goodness)n0 (good)a0 (NP0:_ofNP1)NP (PossNP1_ANP0:)NP

use warnings;
my %patterns;
my $debug = 0;

while (my $line = <>)
{
    ($name,@patterns) = split(/\s+(?=\()/,$line);
    #    if(($name,$lexeme,$logargs) = $name =~ /^([A-Z\-]+)(_\d)?(\(.*\))/)
    {   foreach my $pattern (@patterns)
        {   if(($constituents,$cat) = $pattern =~ /\((.*)\)([A-z0-9]+)/)
            {   $patterns{$name}{$cat} = [] unless($patterns{$name}{$cat});
                push @{$patterns{$name}{$cat}},$constituents;
            }
        }
    }
}
my $lexexp = '(Adv|A|N|V)P(\d)';   # je nach Grammatik
my $headarg = '([A-Z][A-Za-z]*)(\d)(:)';
my $modheadarg = '([A-Z][A-Za-z]*)-([A-Z]+)(\d)(:)';
my $freearg = '([A-Z][A-Za-z]*)-([a-z]+)';
my $argument = '([A-Z][A-Za-z]*)(\d)(~)?';

print "[";  # Begin writing a list of Mercury-Calls

foreach my $name (sort(keys(%patterns)))
{
    print "=" x 50, "\n" if($debug);
    print "$name :\n" if($debug);
    foreach my $cat(keys %{$patterns{$name}})
    {
        print "\n","$cat = ", join(", ",@{$patterns{$name}{$cat}}), "\n" if($debug);
        foreach my $pattern(@{$patterns{$name}{$cat}})
        {   my(@funname,@argnums,@argstrgs);
            # Spalte prepCat auf in Prep-prep_Cat:
            $pattern =~ s/_([a-z]+)([A-Z][a-zA-Z]*)/_Prep-$1_$2/g ;
```

```
            # Spalte das Muster in Konstituenten auf:
            my @constituents = split(/_/,$pattern);
            print "Konstituenten:  "  if($debug);
            print "@constituents\n"  if($debug);

            # Erzeuge einen Mercury-Ausdruck der Form
            # <Funktionsname>(Arg.nummern,Stammformen):
            foreach my $const (@constituents)
            {
              if(($fun, $lexfun, $argnum) = $const =~ /$modheadarg/)
              { if($const =~ /$lexfun/)
                  { $word = $patterns{$name}{lc($lexfun).$argnum}[0] };
                push @funname, lc($fun);
                push @argnums, $argnum;
                push @argstrgs,$word
              }
              elsif(($fun, $argnum) = $const =~ /$headarg/)
              {
                if($const =~ /$lexexp/)
                  { $word = $patterns{$name}{lc($1).$argnum}[0] };
                push @funname, lc($fun);
                push @argnums, $argnum;
                push @argstrgs,$word
              }
              elsif($const =~ /$freearg/)
                  { push @funname, lc($1);
                    push @argnums, "-1";
                    push @argstrgs,$2
                  }
              elsif($const =~ /$argument/)
                  { push @funname,lc($1);
                    push @argnums,$2;
                    push @argstrgs,"-"
                  }
              else { push @funname,"id"; @argnums = @argstrgs = () }
            }

            print "Aufruf:          "  if($debug);
              $out = "{"."\"".join("_",@funname)."\"".",".
                     "[".join(",",@argnums)."],".
                     "[".join(",",map(qq("$_"),@argstrgs))."]}";
            print $out,",\n";
        }
    }
}
```

```
print "{\"id\",[],[]}].";  # end writing a list of Mercury-Calls
print "\n\n" if($debug);
```

Fehler: Die Behandlung der durch lexikalische Funktionen modifizierten Köpfe (`$modheadarg` oben) konstruiert nicht die richtigen Anfragen. Das liegt zum Teil daran, daß mehrere Lexeme derselben Kategorie in der `.lx`-Datei angegeben sind, aber stets das erse genommen wird.

Was die Suchmuster genau sind, wird im Untermodul `muster` der Grammatik festgelegt. Die für die semantische Suche erzeugte Hilfsdatei enthält nicht die Anfragen, sondern nur Bezeichnungen dafür, `query_codes`.

⟨*Auszug aus grammatik.m*⟩≡
```
:- type query_code --> {string,list(int),list(string)}.
```

Um diese Einträge beim Einlesen in entsprechende Werte umrechnen zu können, braucht man ein Umwandlungsprädikat (als Argument von `read_coerce/2`):

⟨*suche.m*⟩+≡
```
:- pred term_to_query_codes(term(list(query_code))::in,
                              list(query_code)::out) is semidet.
term_to_query_codes(Term, Val) :-
        term__term_to_type(Term,Val).
```

In einem zweiten Schritt werden diese `query_codes` in `pseudo_queries` umgerechnet, d.h. die Musternamen werden interpretiert und auf die Argumentlisten angewendet. Dies macht die Funktion `dispatch`, die in der Grammatik (im Untermodul `muster`) definiert wird:

⟨*Auszug aus grammatik.m*⟩+≡
```
:- type pseudo_query --> {tree, list(string), tree}.
:- type pseudo_queries == list(pseudo_query).

:- func dispatch(query_code) = pseudo_queries.
```

Eine *Pseudo-Anfrage* besteht aus einem *Suchbaum*, d.h. einem partiellen Syntaxbaum mit Mustervariablen für Teilbäume, einem *Argumentbaum*, d.h. einer Markierung von Konstituenten im Suchbaum durch Argumentnummern des logischen Suchbegriffs, und der Kodierung eines Testprädikats, das auf gefundene Phrasen angewendet werden soll, um Bedingungen zu Prüfen, die sich nicht im Suchbaum ausdrücken lassen.

Der dritte Schritt rechnet diese Pseudo-Anfragen mit `translate_pseudoqueries` (s.u.) in wirkliche Anfragen um, die das Testprädikat statt seiner Kodierung enthalten. Mit diesen (syntaktischen) Anfragen wird dann die syntaktische Suche aufgerufen.

5.1.2 Syntaktische Suche

Eine *Anfrage* der syntaktischen Suche ist ein Tripel aus Musterbaum, Filterprädikat und Argumentmarkierung; eine Variante hat aus dem Musterbaum schon die Lexeme ausgesondert.

Solche Tupel werden zusammengefaßt in:

⟨*suche.m*⟩+≡
```
:- type query --> {tree, pred(tree), tree}.
:- inst query = bound( {ground, pred(in) is semidet, ground} ).

:- type query_lex --> {tree, trees, pred(tree), tree}.
:- inst query_lex = bound( {ground, ground, pred(in) is semidet, ground} ).
```

Die syntaktische Suche erhält ihre Anfragen entweder interaktiv oder aus einer Datei. (Im ersten Fall wird der Benutzer nach Musterbaum und Argumentmarkierung gefragt, während die Filterprädikate über die Optionen `-F` angegeben werden. Das kann man ggf. revidieren.) Bei der interaktiven Suche wird nur mit einem Muster gesucht, weil das der häufiger interessante Fall sein dürfte.

⟨*suche.m*⟩+≡
```
:- pred syntactic_search(bool::in, printlevel::in, bool::in,
                         string::in, string::in, list(string)::in,
                         io__state::di, io__state::uo) is det.

syntactic_search(Verbosity,PrintLevel,Interactive,File,PatFile,Filter_Args) -->
        io__print("\nSyntaktische Suche, Version 0.07"),
        ( { Interactive = no          % Read the search queries from a file:
          },
            io__see(PatFile, PatRes), % stdin := from File
            ( { PatRes = error(Message) },
                io__write_string(io__error_message(Message)),
                io__print(": "), io__print(PatFile), io__nl
            ; { PatRes = ok },
                read_coerce(term_to_pseudo_queries,Pattern),
                io__seen,             % close PatFile
                ({ Pattern = yes(Pseudo_Queries),
                   translate_filternames(Pseudo_Queries,Queries) }
                -> search_with(parse,Verbosity,PrintLevel,Queries,File)
                ; []
                )
            )
        ; { Interactive = yes,
            filter__get(Filter_Args,Filter) % use filters-options given
          },
            io__print("\nSearch pattern: <Tree>. (Help: ? .)\n"),
            read_coerce(term_to_tree,Pattern),
            ( { Pattern = yes(PatTree) },
                ( { PatTree = (?) }
                -> usage_pattern
                ; io__print("\nArgument pattern: <Tree>. (with arg(N) subtrees .)\n"),
                    read_coerce(term_to_tree,Arguments),
```

```
            ( { Arguments = yes(ArgTree) },
                search_with(parse,Verbosity,PrintLevel,
                            [{PatTree,Filter,ArgTree}],File)
            ; { Arguments = no },
                io__print("This is not a valid argument pattern. \n"),
                usage_pattern
            )
        )
    ; { Pattern = no },
        io__print("This is not a valid pattern. \n"),
        usage_pattern
    )
).
```

Wenn ein Musterbaum oder ein Baum zur Argumentmarkierung gelesen wird, muß er in einen Term vom Typ `tree` umgewandelt werden. Dazu dienen folgende Funktionen:

⟨*suche.m*⟩+≡

```
:- pred term_to_tree(term(tree)::in,tree::out) is semidet.
term_to_tree(Term, Val) :-
        term__term_to_type(Term,Val).
```

Wenn die Anfragen in einer Datei (wie `defaultpattern`) stehen, lesen wir gleich eine Liste von *Pseudo-Anfragen*; diese bestehen aus einem Musterbaum, einer höchstens 2-elementigen Liste von Zeichenreihen (die wie bei der Option `-F` ein Filterprädikat und sein Argument kodiert), und einem Argumentbaum.

⟨*suche.m*⟩+≡

```
% :- type pseudo_query --> {tree, list(string), tree}.  % jetzt im Modul muster|grammatik
% :- type pseudo_queries == list(pseudo_query).

:- pred term_to_pseudo_queries(term(pseudo_queries)::in,
                               pseudo_queries::out) is semidet.
term_to_pseudo_queries(Term, Val) :-
        term__term_to_type(Term,Val).

:- pred translate_filternames(pseudo_queries::in,
                              list(query)::out(listskel(query))) is semidet.

translate_filternames([{PatTree,Strings,ArgTree}|Rest],
                      [{PatTree,Filter,ArgTree}|TransRest]) :-
        filter__get(Strings,Filter),
        translate_filternames(Rest,TransRest).
translate_filternames([],[]).
```

Hier ist eine Beispieldatei mit Defaultmustern. Die Verwendung von Filterprädikaten verlangsamt die Suche *erheblich*: einerseits, weil dann der Suchbaum meistens keine lexikalischen

Elemente mehr enthält und daher Vorfilterung mit Perl unmöglich ist, und andererseits, weil in langen Sätzen oft viele Alternativen mit dem Test überprüft werden müssen.

⟨*defaultpattern.hl*⟩≡
```
% Format: {Musterbaum, Filterspezifikation, Argumentbaum}-Liste

% 1. NP mit DetP und evtl AP, kein Filter, keine Argumentmarkierung:
% [{np(def,+,?,-,+,-),[],(-)}].

% 2. NP mit DetP und evtl AP, kein Filter, Argumentmarkierung:
% [{np(def,+,?,-,+,+),[],np(def,+,?,-,arg(0),pp(+,arg(1)))}].

% 3. NP mit vorg. Kopfnomen und PP, kein Filter, Argumentmarkierung:
% [{np(def,?,?,?,n("daughter"),+),[],np(def,+,?,-,arg(0),pp(+,arg(1)))}].

% 4. NP mit PP, Filter auf Kopfnomen, Argumentmarkierung:
% [{np(def,?,?,?,+,+),["h","daughter"],np(def,+,?,-,arg(0),pp(+,arg(1)))}].

% 5. NP mit Kopfnomen und PP, kein Filter, mit Argumentmarkierung,
%    *oder* DETP mif vorg. Det, ohne Filter, ohne Argumentmarkierung
% [{np(def,?,?,?,n("daughter"),+),[],np(def,+,?,-,arg(0),pp(+,arg(1)))}, {det(def,"the"),[],(-)}].

% 6. NP mit vorg. Kopfnomen und PP mit vorg. PREP, kein Filter, mit Argumentmarkierung,
%    *oder* DETP mif vorg. Det, ohne Filter, ohne Argumentmarkierung
% [{np(def,?,?,?,n("daughter"),pp(prep("of"),+)),[],np(def,+,?,-,arg(0),pp(+,arg(1)))},
%  {det(def,"the"),[],(-)}].

% 7. NP mit PP und vorg. PREP, Filter auf Kopfnomen, Argumentmarkierung:
% [{np(def,?,?,?,+,pp(prep("of"),+)),["h","daughter"],np(def,+,?,-,arg(0),pp(+,arg(1)))}].

% 8. NP mit PP und vorg. PREP, kein Filter, Argumentmarkierung:
 [{np(def,?,?,?,+,pp(prep("of"),+)),[],np(def,+,?,-,arg(0),pp(+,arg(1)))}].
```

5.1.3 Suchen

Die Suchfunktion hat folgenden Typ und Modus:

⟨*suche.m*⟩+≡
```
:- pred search_with(option::in,bool::in,printlevel::in,
                    list(query)::in(listskel(query)),
                    string::in,io__state::di,io__state::uo) is det.
```

Die Suchfunktion überprüft zuerst, ob die angegebene Datei existiert. Falls das der Fall ist, werden die Stammformen der in den Suchmustern eventuell vorkommenden Wörter gesammelt; damit werden zuerst alle die Sätze aus der Eingabedatei extrahiert, die *alle* Stammformen mindestens *eines* Suchmusters enthalten, und in eine temporäre Datei `TMP` geschrieben.

Falls ein Suchmuster ohne konkrete Stammformen benutzt wird, muß die angegebene Gesamtdatei durchsucht werden, andernfalls wird nur die Datei `TMP` der extrahierten Sätze durchsucht. (`Option` ist hierbei stets `parse`.)

⟨*suche.m*⟩+≡

```
search_with(Option,Verbosity,PrintLevel,Queries,File) ->
        io__see(File, Res2),    % stdin := from File
        ( {Res2 = error(Message) }
        -> io__write_string(io__error_message(Message)),
            io__print(": "), io__print(File), io__nl
        ; io__print("\nSearching with pattern(s): \n"),
            { pattern_list(Queries, PatTrees) },
            io__write_list(PatTrees,",\n",io__print), io__nl,
            % Restrict to sentences containing all lexical elements of one PatTree:
            { get_all_lexemes(Queries, QueryLs, StemLists) },
            ( { nontrivial(StemLists) = yes }
            -> io__make_temp(TMP),
                { perl_call(StemLists,TMP,File,Call) },
                ({ Verbosity = yes }
                -> io__print("\nPerl-Aufruf: \n"),
                    io__print(Call), io__nl
                ; []
                ),
                io__call_system(Call,SysResult),
                ( {SysResult = ok(0)}
                -> io__seen, % close File
                    count_sentences(TMP,N),
                    ( { 0 < N }
                    -> ({ Verbosity = yes }
                        -> io__print("\nFound "),io__print(N),
                           io__print(" relevant sentences.\n")
                       ; []
                       ),
                        io__see(TMP,ResTMP),
                        ( {ResTMP = error(MessageTMP)}
                        -> io__nl,
                            io__write_string(io__error_message(MessageTMP)),
                            io__print(": "), io__print(TMP), io__nl
                        ; reader(Option,Verbosity,PrintLevel,QueryLs)
                        )
                    ; io__print("\nNo relevant sentences found.\n")
                    ),
                    io__seen,   % close TMP
                    io__remove_file(TMP, _OK)
                ; {SysResult = error(Msg)}
                -> io__print(Msg), io__nl
                ; []
```

```
                )
            ; % search the original file
                reader(Option,Verbosity,PrintLevel,QueryLs),
                io__seen % close File
            )
        ).
```

Als Rückmeldung werden alle Musterbäume, mit denen gesucht wird, angezeigt:

⟨*suche.m*⟩+≡

```
:- pred pattern_list(list(query)::in(listskel(query)),
                     trees::out) is det.

pattern_list(Queries, PatTrees) :-
    (
        Queries = [{PatTree, _Filter, _ArgTree}|Rest],
        pattern_list(Rest, PatTreesRest),
        PatTrees = [PatTree|PatTreesRest]
    ;
        Queries = [],
        PatTrees = []
    ).
```

Aus jedem Suchmuster werden alle Lexeme ausgesondert und zu einer Liste von Lexemlisten zusammengefaßt:

⟨*suche.m*⟩+≡

```
:- pred get_all_lexemes(list(query)::in(listskel(query)),
                        list(query_lex)::out(listskel(query_lex)),
                        list(list(string))::out) is det.

get_all_lexemes(Queries, QueryLs, StemLists) :-
    (
        Queries = [{PatTree, Filter, ArgTree}|Rest],
        PatLexemes = grammatik__leaves(PatTree),
        filter__stammformen(PatLexemes,Stems),
        get_all_lexemes(Rest, QueryLsRest, StemsRest),
        QueryLs = [{PatTree, PatLexemes, Filter, ArgTree} | QueryLsRest],
        StemLists = [Stems|StemsRest]
    ;
        Queries = [],
        QueryLs = [],
        StemLists = []
    ).

:- func nontrivial(list(list(string))) = bool.

nontrivial([[]|Ls]) = B :-
```

```
        nontrivial(Ls) = B.
nontrivial([[_|_]|_]) = yes.
nontrivial([]) = no.
```

Die Extraktion der höchstens relevanten Sätze der Eingabedatei erfolgt mit `Perl`, da das erheblich schneller wird als mit `Mercury`. Dabei wird die Satzende-Markierung `</sen>` als Zeilenende benutzt und dann in diesen Langzeilen nach den Stammformen gesucht:

⟨*suche.m*⟩+≡

```
:- pred perl_call(list(list(string))::in, string::in, string::in, string::out)
        is det.

perl_call(StringLists,TMP,File,Call) :-
        list__map(perl_conj,StringLists,Conjs),
        perl_disj(Conjs,String),
        string__append_list(["/usr/bin/perl -ne '$/=""</sen>""; ",
                             "print if(",String,")' ",
                             File," > ",TMP],
                            Call).

:- pred perl_disj(list(string)::in, string::out) is det.

perl_disj(Strings,Disj) :-
        (Strings = [A,B|Cs],
            perl_disj([B|Cs],BCs),
            string__append_list(["(",A,")"," || ",BCs],Disj)
        ; Strings = [A],
            string__append_list(["(",A,")"],Disj)
        ; Strings = [],
            Disj = ""
        ).

:- pred perl_conj(list(string)::in, string::out) is det.

perl_conj(Strings,Conj) :-
        (Strings = [A,B|Cs],
            perl_conj([B|Cs],BCs),
            string__append_list(["/>",A,"</"," && ",BCs],Conj)
        ; Strings = [A],
            string__append_list(["/>",A,"</"],Conj)
        ; Strings = [],
            Conj = ""
        ).
```

Bemerkung 5.1 Um die relevanten Sätze zu mehreren Mustern zu extrahieren, wird z.B. zu den Stammwörtern `far, majority` des einen und `opposition, former` des anderen Musters ein Perl-Aufruf der folgenden Form konstruiert:

⟨*perl-Aufruf für mehrere Muster*⟩≡
```
perl -ne '$/="</sen>"; print if((/>far</ && />majority</) ||
                                (/>opposition</ && />former</))'
          refcorp.anfang.tag
```

Die Datei der relevanten Sätze wird nur dann durchsucht, wenn sie nicht leer ist; andernfalls wird eine Meldung ausgegeben.

⟨*suche.m*⟩+≡
```
:- pred count_sentences(string::in,int::out,io__state::di,io__state::uo) is det.
count_sentences(File,N) ->
        { string__append_list(["grep \"<sen\" ",File,
                               " | wc -l | cat - punkt.txt > filesize.tmp"],Call) },
        io__call_system(Call,Res),
        ({ Res = ok(0) }
        -> io__see("filesize.tmp",OK),
            ( { OK = error(Msg) },
                io__nl,io__write_string(io__error_message(Msg)), io__nl,
                { N = -1 }
            ; { OK = ok },
                read_coerce(term_to_int,Int),
                ( { Int = yes(M), N = M }
                ; % io__write_string("Error in file: filesize.tmp "),
                  { Int = no, N = -1 }
                ),
                io__seen
            )
        ; { N = -1 }
        ).

:- pred term_to_int(term(int)::in,int::out) is semidet.
term_to_int(Term, Val) :-
        term__term_to_type(Term,Val).
```

Um die Anzahl der Zeilen als Zahl von Mercury einzulesen, schreibe ich umständlich einen Punkt dahinter. Das muß natürlich einfacher gehen als mit der Hilfsdatei:

⟨*punkt.txt*⟩≡
```
. % Nicht loeschen. Wird von suche:count_sentences/2 gebraucht
```

5.1.4 Tokenisierung und Lemmatisierung

Tokenisierung und Lemmatisierung rufen nur die Leseschleife `reader` mit verschiedenen Optionen auf. Sie sind hauptsächlich zur Fehlersuche gedacht.

Falls der Lexer auf unbekannte Zeichen stößt, kann man mit der Option `suche -i -tokenize` interaktiv eine Zeichenfolge eingeben und alle auftretenden Zeichen einlesen lassen. Dadurch kann man unbekannte Zeichen finden und in den Lexer einfügen.

⟨*suche.m*⟩+≡
```
:- pred tokenizer(bool::in,printlevel::in,io__state::di,io__state::uo) is det.

tokenizer(Verbosity,PrintLevel) ->
        io__print("\nSyntaktische Suche, Version 0.07"),
        io__print("\nTokenize CIS-tagged input line. Input chars ... (Stop: ^D)\n"),
        reader(tokenize,Verbosity,PrintLevel,
               [{(+),[],filter__id_filter,(-)}]).
```

Falls der Lexer die erkannten Token nicht zu einem Lemma im Sinne der Grammatik zusammenfassen kann, kann man mit der Option `suche -i -lemmatize` einen Beispielsatz der Eingabedatei (oder eine Folge lexikalisch markierter Wörter) eingeben und herausfinden, welche Tags nicht erkannt oder nicht zu einem Lemma zusammengefaßt werden können.

⟨*suche.m*⟩+≡
```
:- pred lemmatizer(bool::in,printlevel::in,io__state::di,io__state::uo) is det.

lemmatizer(Verbosity,PrintLevel) ->
        io__print("\nSyntaktische Suche, Version 0.07"),
        io__print("\nLemmatize CIS-tagged input lines. Input tags ... (End: </sen>, Stop: ^D)\n"),
        reader(lemmatize,Verbosity,PrintLevel,
               [{(+),[],filter__id_filter,(-)}]).
```

In beiden Fällen ist das Argument `PrintLevel` eigentlich irrelevant; man könnte verlangen, daß alle Grammatiken ein PrintLevel wie `flach` festlegen und dieses hier einsetzen. Statt `filter__id_filter` sollte man vielleich gleich `(pred(T) :- true) : (pred(tree) :: (pred(in) is semidet))` einsetzen, oder wie immer die richtige Syntax dafür ist.

Der Hilfsfunktion `lex__manipulate_source` aus dem Modul `lex` von `Mercury` geben wir einen kürzeren Namen:

⟨*suche.m*⟩+≡
```
:- pred lms(pred(Src, Src),
            lex__lexer_state(Tok, Src), lex__lexer_state(Tok, Src)).
:- mode lms(pred(di, uo) is det, di, uo) is det.

lms(Pred, In, Out) :- lex__manipulate_source(Pred, In, Out).
```

Die Tokenisierung benutzt geschieht wie folgt. Falls das nächste gelesene Zeichen unbekannt ist, wird eine Fehlermeldung ausgegeben. Falls das nächste gelesene Zeichen erkannt wird, aber noch nicht das Satzende ist, wird weitergelesen, sonst werden die gefundenen Token an den Lemmatisierer oder Parser weitergegeben.

⟨*suche.m*⟩+≡
```
:- pred tokenize_stdin(option::in,
                       bool::in,
                       printlevel::in,
                       list(query_lex)::in(listskel(query_lex)),
                       list(token)::in,
                       lexer_state(token, io__state)::di,
                       lexer_state(token, io__state)::uo) is det.

tokenize_stdin(Option,Verbosity,PrintLevel,QueryLs,Acc) ->
        lex__read(Result),
        ( { Option = tokenize }
        -> lms(io__print(Result)),
           lms(io__nl)
        ; []),
        ( { Result = error(_Message,Position) },
           tokenize_report_error(Verbosity,Position,Acc),
           tokenize_stdin(Option,Verbosity,PrintLevel,QueryLs,[])
        ; { Result = ok(Token) },
           ( { Token = eos } ->
               tokenize_handle_eos(Option,Verbosity,PrintLevel,
                                   [Token|Acc],QueryLs),
               tokenize_stdin(Option,Verbosity,PrintLevel,QueryLs,[])
           ; tokenize_stdin(Option,Verbosity,PrintLevel,QueryLs,[Token|Acc])
           )
        ; { Result = eof },
           tokenize_handle_eos(Option,Verbosity,PrintLevel,[eos|Acc],QueryLs)
        ).
```

Als Fehlermeldung bei einem unbekannten Zeichen wird die Position des Zeichens in der Eingabedatei ausgegeben und die Liste der bis zu diesem Zeichen erkannten Token oder Lemmata, sofern sie zu Lemmata zusammengefaßt werden können.

⟨*suche.m*⟩+≡
```
:- pred tokenize_report_error(bool::in,
                              int::in,
                              list(token)::in,
                              lexer_state(token,io__state)::di,
                              lexer_state(token,io__state)::uo) is det.

tokenize_report_error(Verbosity,Position,Acc) ->
        lms(io__print("\nLexer error at position: ")),
        lms(io__write(Position)),
        lms(io__nl),
        ( { Verbosity = yes }
```

```
        -> {tokens_to_lemmata(Acc,[],Lemmata)},
            ( { Lemmata = [] }
            -> {list__reverse(Acc,AccRev)},
                lms(io__print("Skipped tokens:\n")),
                lms(io__print(AccRev))
            ; lms(io__print("Skipped lemmata:\n")),
                lms(io__print(Lemmata))
            ),
            lms(io__nl)
        ; [] ).
```

5.1.5 Behandlung des Satzendes: Parsen oder weiter Lemmatisieren

Falls der Tokenizer ein Satzende gefunden hat, gibt er im Modus `lemmatize` die Folge der gefundenen Lemmata aus. Sonst ist er im Parse-Modus und ruft, falls alle Lexeme des Suchmusters im gefundenen Satz vorkommen, den Parser auf:

⟨*suche.m*⟩+≡

```
:- pred tokenize_handle_eos(option::in, bool::in, printlevel::in,
                            list(token)::in,
                            list(query_lex)::in(listskel(query_lex)),
                            lexer_state(token,io__state)::di,
                            lexer_state(token,io__state)::uo) is det.

tokenize_handle_eos(_Option,_Verbosity,_PrintLev,[],_QueryLs) ->
        lms(io__print("Empty input sentence. ")).
tokenize_handle_eos(Option,Verbosity,PrintLevel,[Result|Acc],QueryLs) ->
        ( { Option = lemmatize }
        -> lms(io__nl),
            ( {tokens_to_lemmata([Result|Acc],[],Lemmata)}
            -> lms(io__print("Lemmata:\n")),
                lms(io__print(Lemmata))
            ; lms(io__print("Accumulated at eos:\n")),
                lms(io__print(Acc))
            )
        ; { Option = parse,
            tokens_to_trees([Result|Acc],Trees)
          }
        -> ( { remove_non_hit_patterns(Trees, QueryLs, Querys) },
               ({ Verbosity = yes }
               -> lms(io__print("\nLemmatized Sentence: ")),
                   lms(io__print(Trees)),
                   lms(io__nl)
               ; []
               ),
               { solutions((pred(ParseTrees::out) is nondet :-
                           parse_with_patterns(Trees,Querys,[],ParseTrees)),
```

```
                    SetOfResults),
                list__condense(SetOfResults,AllParseTrees) },
              ( { Verbosity = yes }
              -> lms(io__print("\nAllParseTrees:\n")),
                 lms(io__print(AllParseTrees)),
                 lms(io__nl)
              ; []
              ),
              ( {AllParseTrees = [] }
              ->  % lms(io__print("\nNo subphrases fit to the pattern.\n"))
                 lms(io__print(":")),
                 lms((io__flush_output))
              ;   % lms(io__print("\nSubphrases fitting to the pattern: \n")),
                 lms(io__nl),
                 lms(grammatik__print_alltrees(PrintLevel,AllParseTrees))
              )
          )
       ; tokenize_handle_eos(lemmatize,Verbosity,PrintLevel,[Result|Acc],QueryLs)
       ).
```

Alle Querys, deren Lexeme im Eingabesatz `Trees` nicht vorkommen, werden entfernt, bevor der Parser aufgerufen wird:[25]

⟨*suche.m*⟩+≡

```
 :- pred remove_non_hit_patterns(trees::in,
                                 list(query_lex)::in(listskel(query_lex)),
                                 list(query)::out(listskel(query))) is det.

 remove_non_hit_patterns(Trees, QueryLs, Querys) :-
     (
         QueryLs = [{PatTree, LexList, Filter, ArgTree}|Rest],
         (
             filter__alle_vorhanden(LexList, Trees)
             -> Querys1 = [{PatTree, Filter, ArgTree}]
             ;  Querys1 = []
         ),
         remove_non_hit_patterns(Trees, Rest, Querys2),
         list__append(Querys1, Querys2, Querys)
     ;
         QueryLs = [],
         Querys = []
     ).

 :- pred parse_with_patterns(trees::in,
                             list(query)::in(listskel(query)),
```

[25]Das ist auch dann nötig, wenn mit `perl` nur die relevanten Sätze ausgefiltert wurden: verschiedene Sätze können die Lexeme verschiedener Anfragen enthalten.

```
                              list(trees)::in, list(trees)::out) is nondet.

parse_with_patterns(Trees, Queries, Acc, ParseTrees) :-
    (
        Queries = [{PatTree, Filter, ArgTree}|Rest],
        parse(Trees, PatTree, Filter, [], ParseTrees1Rev),
        ( ParseTrees1Rev = []
        -> parse_with_patterns(Trees,Rest, Acc, ParseTrees)
        ; ParseTrees1 = list__reverse(ParseTrees1Rev),
            ArgParseTrees1 = [ArgTree|ParseTrees1], % ArgTree zum Drucken
            parse_with_patterns(Trees, Rest, [ArgParseTrees1|Acc], ParseTrees)
        )
    ;
        Queries = [],
        list__reverse(Acc,ParseTrees)
    ).

:- end_module suche.
```

Bem. Der Test `alle_vorhanden` kann jetzt eliminiert werde, da mit `perl` nur die Sätze aus der Eingabedatei entnommen wurden, die die Stammformen aller Wörter im Musterbaum enthalten.

5.2 Die Filterprädikate

Da man beim Aufruf des Suchprogramms nicht beliebige Testprädikate zum Filtern der Treffer angeben kann, muß man nützliche Testprädikate vorkompilieren, unter denen beim Aufruf durch Angabe von Optionen und Parametern eines ausgewählt werden kann.

Wir sammeln einige parametrisierte Filterprädikate in einem eigenen Modul `filter`. Die Optionen `-F` des Suchprädikates werden zu einer Liste von Zeichenreihen akkumuliert, deren erstes als Name eines Prädikates und deren weitere als dessen Argumente verstanden werden. Daraus wird ein einstelliges Testprädikat für Syntaxbäume erstellt.

Zu diesen Filteroptionen muß der Grammatikschreiber hier einen Hilfstext `usage` angeben, der mit den Hilfstexten der Standardoptionen des Suchprogramms kombiniert wird.

Vorläufig werden nur Tests auf Stamm- und Vollform des Kopfes sowie das Vorkommen eines Stamms in der gefundenen Phrase implementiert.

⟨*filter.m*⟩≡

```
:- module filter.

:- interface.

:- import_module string, list.
:- import_module grammatik.
```

```
:- func usage = list(string).

:- pred get(list(string)::in, pred(tree)::out(pred(in) is semidet)) is det.

:- pred id_filter(tree::in) is semidet.
:- pred alle_vorhanden(trees::in,trees::in) is semidet.

:- pred stammformen(trees::in, list(string)::out) is det.
```

Weitere Filter sollten dazukommen, insbesondere solche, die das *Nichtvorkommen* von bestimmten Wortvollformen, -stammformen oder -arten im gefundenen Baum prüfen, beispielsweise, ob in einer gefundenen Nominalphrase kein Reflexivpronomen vorkommt.

⟨*filter.m*⟩+≡

```
:- implementation.

:- import_module std_util.

usage = ["  -F, -filter\n",
         "      Use the next character <h|H|o> as name for a filter\n",
         "      `pred(tree::in,string::in)' and the next -F argument as its argument'.\n"].

get(Liste,Pred) :-
        ( Liste = []           -> Pred = id_filter
        ; Liste = ["t"]        -> Pred = id_filter
        ; Liste = ["h",String] -> Pred = headstem(String)
        ; Liste = ["H",String] -> Pred = headform(String)
        ; Liste = ["o",String] -> Pred = occurs_stem(String)
        ; Pred = id_filter    % default, Fehlermeldung (?)
        ).

:- pred headform(string::in,tree::in) is semidet.
:- pred headstem(string::in,tree::in) is semidet.
:- pred occurs_stem(string::in,tree::in) is semidet.

id_filter(_).

headform(SurfaceString,Tree) :-
                vollform(grammatik__head(Tree)) = SurfaceString.
headstem(StemString,Tree) :-
                stammform(grammatik__head(Tree)) = StemString.

occurs_stem(String,Tree) :-
        Lexeme = grammatik__leaves(Tree),
        ( kommt_vor(n(String),Lexeme) -> true
        ; kommt_vor(v(String),Lexeme) -> true
```

```
        ; kommt_vor(adj(String),Lexeme) -> true
%       ; kommt_vor(adv(_,String),Lexeme) -> true
        ; kommt_vor(en(String),Lexeme) -> true
%       ; kommt_vor(pron(_,String),Lexeme) -> true
        ; kommt_vor(det(String),Lexeme) -> true
        ; kommt_vor(cnj(String),Lexeme) -> true
        ; kommt_vor(prep(String),Lexeme) -> true
        ; kommt_vor(infp(String),Lexeme) -> true
        ; kommt_vor(intj(String),Lexeme) -> true
        ; kommt_vor(pfx(String),Lexeme) -> true
        ; kommt_vor(sfx(String),Lexeme) -> true
        ; fail
        ).

:- func vollform(tree) = string is semidet.

vollform(n(_F,Vollform,_Stamm)) = Vollform.
vollform(v(_F,Vollform,_Stamm)) = Vollform.
vollform(adj(_F,Vollform,_Stamm)) = Vollform.
vollform(adv(_F,Vollform)) = Vollform.

:- func stammform(tree) = string is semidet.

stammform(n(_F,_V,Stamm)) = Stamm.
stammform(en(_F,_V,Stamm)) = Stamm.
stammform(adj(_F,_Vollform,Stamm)) = Stamm.
stammform(v(_F,_Vollform,Stamm)) = Stamm.
stammform(adv(_F,Stamm)) = Stamm.
stammform(pron(_F,Stamm)) = Stamm.
stammform(det(_F,Stamm)) = Stamm.
stammform(cnj(_F,Stamm)) = Stamm.
stammform(prep(Stamm)) = Stamm.

% Zum Filtern mit perl:
stammform(n(Stamm)) = Stamm.
stammform(v(Stamm)) = Stamm.
stammform(adj(Stamm)) = Stamm.
stammform(en(Stamm)) = Stamm.
stammform(det(Stamm)) = Stamm.
stammform(cnj(Stamm)) = Stamm.

:- pred kommt_vor(tree::in, list(tree)::in) is semidet.

kommt_vor(L,[T|Ts]) :-
        ( L = T -> true
        ; variante(T,L) -> true
        ; kommt_vor(L,Ts) -> true
```

```
        ; fail
        ).

:- pred variante(tree::in,tree::in) is semidet.

variante(n(_,_,Stamm), n(Stamm)).
variante(en(_,_,Stamm), en(Stamm)).
variante(adj(_,_,Stamm), adj(Stamm)).
variante(v(_,_,Stamm), v(Stamm)).

% Früher in parsetree.m:

:- pred lexemfilter(trees::in,tree::in) is semidet.

lexemfilter(Trees,PatTree) :-
        PatLexeme = grammatik__leaves(PatTree),
        alle_vorhanden(PatLexeme,Trees).

alle_vorhanden([],_).
alle_vorhanden([L|Ls],Trees) :-
        kommt_vor(L,Trees),
        alle_vorhanden(Ls,Trees).

stammformen([],[]).
stammformen([Tr|Trs],L) :-
        stammformen(Trs,Sts),
        (stammform(Tr) = St
        -> L = [St|Sts]
        ; L = Sts).

:- end_module filter.
```

Wenn die Suchmuster Wörter enthalten, wird die Suche wegen der Einschränkung auf die mit `perl` ausgesonderten Sätze erheblich schneller, als wenn man nur im Nachhinein die Treffer daraufhin prüft, ob entsprechende Wörter vorkommen.

⟨*Beispiel zur Suche mit Wörtern im Suchbaum*⟩≡

```
Mercury-Kurs > suche -i -frefcorp.xaa.tag

Syntaktische Suche, Version 0.07
Search pattern: <Tree>. (Help: ? .)
np(def,?,?,?,n("minister"),-).
```

⟨*Beispiel zur Suche mit Filtern der Treffer*⟩≡

```
Mercury-Kurs > suche -i -Fh -Fminister -frefcorp.xaa.tag

Syntaktische Suche, Version 0.07
Search pattern: <Tree>. (Help: ? .)
```

```
np(def,?,?,?,+,-).
```

6 Installation: Makefile zur Erzeugung des Suchprogramms

Warnung: Das jetzige Programm `suche.m` ist nur ein Teil des von Hand für die Beispielgrammatik erstellten Programms `suche.m`. Im jetzigen `suche.m` werden die Module `grammar,token` und `filter` des Grammatikschreibers importiert. Mercury-0.10.0 verlangt, daß Modulnamen und Dateinamen übereinstimmen; die Eingangsgrammatik `<Grammatik>.dcg` muß zwar den Modul `:- modul gammatik` deklarieren, braucht aber nicht `grammatik.dcg` zu heißen (aber bei `token` und `filter` geht das nicht).

Die jetzt mit `expand_dcg` aus `<Grammatik>.dcg` erzeugte Datei `grammatik.m` überschreibt ggf. eine frühere Übersetzung `grammatik.m` einer anderen `<Grammatik-vorher>.dcg`. Durch die Erzeugung des Suchprogramms aus `suche.m,grammatik.m,token.m,filter.m` wird auch die Binärdatei `suche` jeweils überschrieben. Wenn man die Suchprogramme zu verschiedenen Grammatiken vergleichen will, muß man also die Binärdateien (und die selbstgeschriebenen `token.m,filter.m`) erst mit einem selbstgewählten Namen umbenennen.

Erzeugung des Suchprogramms mit `expand_dcg,suche.m`

Um ein Suchprogramm aus drei (zu einander passenden) Dateien

- `grammatik.dcg`,
- `token.m`,
- `filter.m`

zu erzeugen, benutzt man einen Makefile mit den Zielen

```
depend:
        mmake expand_dcg.depend
        mmake expand_dcg
suche:
        expand_dcg grammatik.dcg > grammatik.m
        mmake suche.depend
        mmake suche
```

und ruft in der Shell `make depend` und `make suche` auf. Dabei werden drei Module `grammatik,token,filter` importiert, die aus den Dateien `token.m` und `filter.m` sowie dem erzeugten `grammatik.m` kompiliert werden.

Erzeugung des Suchprogramms mit `notangle` und der Dokumentation

Wenn man das Programm `noweb` bzw. das Teilprogramm `notangle` installiert hat und die `noweb`-Quellen der Dokumentation im Unterverzeichnis `Doku` liegen, benutzt man einen Makefile mit den Zielen

```
suchprogramm:
        notangle -Rexpand_dcg.m Doku/parsergenerator.nw | cpif expand_dcg.m
        notangle -Rgrammatik.dcg Doku/parsergenerator.nw Doku/suchgenerator.nw \
                 | cpif grammatik.dcg
        mmake expand_dcg
        expand_dcg grammatik.dcg | cpif grammatik.m
        notangle -Rtoken.m Doku/suchgenerator.nw | cpif token.m
        notangle -Rfilter.m Doku/suchgenerator.nw | cpif filter.m
        notangle -Rsuche.m Doku/suchgenerator.nw | cpif suche.m
        mmake suche.depend
        mmake suche
```

und ruft in der Shell `make suchprogramm` auf.

Bemerkung 6.1 In Mercury-0.11.0 ist die starre Kopplung von Dateiname und Modulname aufgehoben; man kann dann also zueinander passende Dateien `<Grammatik>.dcg`, `<Token>.m`, `<Filter>.m` an ihren Namen kenntlich machen. Die darin vorkommenden Deklarationen können und müssen dann `:- module grammatik.`, `:- module token.`, und `:- module filter.` heißen. Beim Linken muß dann festgelegt werden, aus welchen Dateien die Module erstellt werden sollen.

Entsprechend geht die Erzeugung des Suchprogramms aus den Dateien für Mercury-0.11.0. Allerdings ist in das entsprechende `suche.m` noch nicht die Extraktion von Sätzen der Eingabedatei mit Hilfe von `perl` eingebaut. (Kann man hoffentlich mit Vergleichen der Dateien leicht selbst machen.)