Semantik: Interpretation und Auswertung

Hauptseminar WS 2004/05
CIS, Universität München

Hans Leiß

3. Januar 2005

Zusammenfassung

Es werden exemplarische Grammatiken zu Nominalphrasen, einfachen Sätzen, verallgemeinerten Quantoren im Singular und Plural, Temporal- und Lokaladverbien angegeben sowie passende Interpretationen und Auswertungsfunktionen dafür definiert. Es wird gezeigt, wie man mit einem Parsergenerator daraus automatisch Programme zur Analyse und Auswertung korrekter Ausdrücke erzeugen kann.

Es sollen Teile der Peirce-Grammatik des Deutschen aus Böttner[2] implementiert werden, d.h. als semantische Werte sind nur Mengen und zweistellige Relationen erlaubt.

Alternativ können auch semantische Werte im Sinne der Montague-Semantik implementiert werden.

Inhaltsverzeichnis

1 Peirce-Grammatik

Eine Peirce-Grammatik ist eine kontextfreie Grammatik, deren Ausdrücke in einer Peirce-Algebra interpretiert werden. Wir wollen eine Peirce-Grammatik für einen Ausschnitt des Deutschen testen, bei dem die Nomina, Adjektive und Verben durch 1- und 2-stellige Relationen interpretiert werden. Dabei orientieren wir uns an Böttner [2].

1.1 Peirce'sche Algebren

Eine *Peirce'sche Algebra* besteht aus einem Bereich von „Mengen“ und einem Bereich von „Relationen“, die jeweils[1] eine Boole'sche Algebra bilden, und bestimmten Operationen, mit denen aus Relationen Mengen und aus Mengen Relationen gebildet werden. Wir fürgen hier eine dritte Sorte, die Wahrheitswerte, hinzu, obwohl man das nicht bräuchte, wenn man alle nicht-leeren Mengen mit dem Wahrheitswert `true` gleichsetzt.

Die genaue Signatur einer Peirce-Algebra ist:

⟨*PA/peirce.sig*⟩≡

```
signature PEIRCE =
   sig
        type set

        val ConS : string -> set
        val UnitS : set
        val ZeroS : set
        val SumS  : set * set -> set
        val ProdS : set * set -> set
        val CompS : set -> set

        type rel

        val ConR : string -> rel
        val I : rel
        val UnitR : rel
        val ZeroR : rel
        val SumR  : rel * rel -> rel
        val ProdR : rel * rel -> rel
        val CompR : rel -> rel
        val Minor : rel -> rel

        val Conv : rel -> rel
        val Prod : rel * rel -> rel
```

[1] Man kann die üblichen Symbole wie `+`, `.`, `;` usw. in SML leider nicht überladen.

```
        val PreIm : rel * set -> set
        val Cylin : set -> rel

        type tv

        val E : set -> tv
        val U : set -> tv
        val N : set -> tv

        val ER : rel -> tv
        val UR : rel -> tv
        val NR : rel -> tv

        val And : tv * tv -> tv
        val Or : tv * tv -> tv
        val Neg : tv -> tv
    end
```

Man unterscheidet zwischen *abstrakten* und *konkreten* Peirce-Algebren: Im ersten Fall sind die „Mengen“ und die „Relationen“ sind Elemente *irgendwelcher* (bestimmter) Boole'scher Algebren

$$\begin{aligned} \mathcal{A} = (A, +, 0, \cdot, 1, ^{-}) &\quad = \quad (\texttt{set}, \texttt{SumS}, \texttt{ZeroS}, \texttt{ProdS}, \texttt{UnitS}, \texttt{CompS}) \\ \mathcal{B} = (B, +, 0, \cdot, 1, ^{-}) &\quad = \quad (\texttt{rel}, \texttt{SumR}, \texttt{ZeroR}, \texttt{ProdR}, \texttt{UnitR}, \texttt{CompR}); \end{aligned}$$

dazu kommen geeignete Interpretationen für `MirorR,Conv,Prod,PreIm,Cylin`.

Im zweiten Fall sind die „Mengen“ und „Relationen“ wirkliche Mengen und Relationen über einem Universum U. Das Standardbeispiel ist die *volle Peirce-Algebra über der Menge* U: hier sind $\mathcal{A}$ und $\mathcal{B}$ die beiden Potenzmengenalgebren $\mathcal{A} = (2^U, +, 0, \cdot, 1, ^{-})$ und $\mathcal{B} = (2^{U\times U}, +, 0, \cdot, 1, ^{-})$ über U, jeweils mit Vereinigung als $+$, Durchschnitt als $\cdot$, $\emptyset$ als 0, U bzw. $U \times U$ als 1 und Komplement bezüglich U bzw. $U \times U$ als $^{-}$. Die weiteren Operationen werden wie folgt interpretiert:

(a) $\texttt{I} := \{\langle a, a\rangle \mid a \in U\}$, die Diagonale oder Identität

(b) $\texttt{Conv}(R) := \{\langle b, a\rangle \mid \langle a, b\rangle \in R\}$, die konverse der Relation R,

(c) $\texttt{Prod}(R, S) := \{\langle a, c\rangle \mid \exists b \in U(\langle a, b\rangle \in R \wedge \langle b, c\rangle \in S)\}$, das Produkt der Relationen R, S,

(d) $\texttt{PreIm}(R, B) := \{a \in U \mid \exists b \in B\ \langle a, b\rangle \in R\}$, das Urbild von B unter R,

(e) $\texttt{Cylin}(A) := A \times U$, der (Rechts-) Zylinder über der Menge A.

(f) und der (rekursiv definierte) „untere Teil“ einer Relation,

$$\texttt{Minor}(R) := \begin{cases} \emptyset, & \text{falls } R = R \cap (1 : R \times R : 1) \\ R \cap (\overline{1 : R} \times 1 : R) & \\ \cup\ \texttt{Minor}(R \cap (1 : R \times R : 1)), & \text{sonst,} \end{cases}$$

wobei $R:1$ der Urbildbereich `PreIm(R,UnitS)` und $1:R$ der Bildbereich `PreIm(Conv(R),UnitS)` von R ist. Siehe dazu Abschnitt 3.3.1.

Diese Interpretation wollen wir für endliches U unten programmieren. Vorher betrachten wir eine „Interpretation“ durch bloße Ausdrücke.

1.1.1 Die Term-Algebra der Peirce-Ausdrücke

In der Term-Algebra werden die Funktionen der Signatur `PEIRCE` durch Konstruktionsschritte beim Aufbau von Ausdrücken „interpretiert“, wobei eigentlich statt einzelner Terme ihre Äquivalenzklassen (modulo der hier nicht angegebenen Axiome der Peirce-Algebra) als die Objekte zu nehmen sind.

⟨*PA/peirce.sml*⟩≡

```
structure PeirceTerm =
   struct
      datatype set = ConS of string
                   | SumS of set * set
                   | ZeroS
                   | ProdS of set * set
                   | UnitS
                   | CompS of set
                   | PreIm of rel * set   (* R : B *)
      and      rel = ConR of string
                   | SumR of rel * rel
                   | ZeroR
                   | ProdR of rel * rel
                   | UnitR
                   | CompR of rel
                   | Minor of rel
                   | Id of set
                   | Prod of rel * rel
                   | Conv of rel
                   | Cylin of set
      and      tv  = E of set
                   | N of set
                   | U of set
                   | ER of rel
                   | UR of rel
                   | NR of rel
                   | And of tv * tv
                   | Or  of tv * tv
                   | Neg of tv

      val I = Id UnitS
```

```
  fun StoString ZeroS        = "0"
    | StoString UnitS        = "1"
    | StoString (ConS(w))    = w
    | StoString (SumS(A,B)) =
      "(" ^ (StoString A) ^ " + " ^ (StoString B) ^ ")"
    | StoString (ProdS(A,B)) =
      "(" ^ (StoString A) ^ " . " ^ (StoString B) ^ ")"
    | StoString (CompS(A))    = " -" ^ (StoString A)
    | StoString (PreIm(R,A)) =
      "(" ^ (RtoString R) ^ " : " ^ (StoString A) ^ ")"
  and RtoString (ConR(r))     = r
    | RtoString UnitR         = " 1' "
    | RtoString ZeroR         = " 0' "
    | RtoString (SumR(R,S))  =
      "(" ^ (RtoString R) ^ " + " ^ (RtoString S) ^ ")"
    | RtoString (ProdR(R,S)) =
      "(" ^ (RtoString R) ^ " . " ^ (RtoString S) ^ ")"
    | RtoString (CompR(R))    = "-" ^ (RtoString R)
    | RtoString (Minor(R))    = "minor" ^ (RtoString R)
    | RtoString (Id(A))       = "Id(" ^ (StoString A) ^ ")"
    | RtoString (Prod(R,S))  =
      "(" ^ (RtoString R) ^ " ; " ^ (RtoString S) ^ ")"
    | RtoString (Conv(R))     = "~" ^ (RtoString R)
    | RtoString (Cylin(A))    = "("^(StoString A) ^ " x 1)"
  and TtoString (E A) = "E(" ^ (StoString A) ^ ")"
    | TtoString (U A) = "U(" ^ (StoString A) ^ ")"
    | TtoString (N A) = "N(" ^ (StoString A) ^ ")"
    | TtoString (ER A) = "ER(" ^ (RtoString A) ^ ")"
    | TtoString (UR A) = "UR(" ^ (RtoString A) ^ ")"
    | TtoString (NR A) = "NR(" ^ (RtoString A) ^ ")"
    | TtoString (Neg A) = "Neg(" ^ (TtoString A) ^ ")"
    | TtoString (And(A,B)) = "(" ^ (TtoString A) ^ " /\\ "
                           ^ (TtoString B) ^ ")"
    | TtoString (Or(A,B)) = "(" ^ (TtoString A) ^ " \\/ "
                                 ^ (TtoString B) ^ ")"
end
```

Die Funktionen `StoString: set -> string`, `RtoString: rel -> string` und `TtoString: tv -> string` sind eingefügt, um die Peirce-Ausdrücke lesbar anzeigen zu können.

Beachte, daß die Datentypkonstruktoren Funktionen vom passenden Typ sind, z.B. kann der Konstruktor `PreIm` als Interpretation der in der Signatur `PEIRCE` verlangten Funktion `PreIm : rel * set -> set` benutzt werden.[2] Die Kon-

[2]Die Axiome der Peirce-Algebra werden allerdings hierdurch nicht erfüllt; dazu müßten wir von den Termen zu Äquivalenzklassen von Termen übergehen.

struktoren `ConR` und `ConS` dienen dazu, beliebige Konstante für Relationen oder Mengen einführen zu können.

In einer Peirce-Grammatik wollen wir den Eingabesätzen zuerst Peirce-Terme zuordnen, die noch gar nicht interpretiert sind; später werden wir diese Terme in einer vollen Peirce-Algebra auswerten. Da die Auswertung durch Fallunterscheidung nach dem Termaufbau definiert werden soll, sind Mengen- und Relationsterme als Datentyp zu definieren.

⟨*PA/peirce.sig*⟩+≡

```
signature SYN =
   sig
       datatype set = ConS of string
                     | SumS of set * set
                     | ZeroS
                     | ProdS of set * set
                     | UnitS
                     | CompS of set
                     | PreIm of rel * set   (* R : B *)
       and      rel = ConR of string
                     | SumR of rel * rel
                     | ZeroR
                     | ProdR of rel * rel
                     | UnitR
                     | CompR of rel
                     | Minor of rel
                     | Id of set
                     | Prod of rel * rel
                     | Conv of rel
                     | Cylin of set
       and      tv  = E of set
                     | N of set
                     | U of set
                     | ER of rel
                     | UR of rel
                     | NR of rel
                     | And of tv * tv
                     | Or of tv * tv
                     | Neg of tv

       val StoString : set -> string
       val RtoString : rel -> string
       val TtoString : tv -> string
   end
```

Anders gesagt, benutzen wir im ersten Schritt die Terme der Peirce-Algebra als *abstrakte Syntax* der natürlichen Beispielsprache. Die Auswertungsregeln in der

Grammatik enthalten die Funktionen aus der Signatur `PEIRCE`. Da die Startsymbole der Grammatik einen eindeutigen Ausgabetyp erfordern, müssen die in `SYN` unterschiedenen Typen für Mengen und Relationen zu *einem* Datentyp zusammengefaßt werden:

⟨*PA/absyn.sml*⟩≡

```
signature ABSYN =
   sig
      include SYN
      datatype absyn = nullary of tv | unary of set | binary of rel
   end
```

Um diesen Datentyp erweitern wie die bisherige Struktur der Peirce-Terme:

⟨*PA/absyn.sml*⟩+≡

```
structure Absyn : ABSYN =
   struct
      open PeirceTerm
      datatype absyn = nullary of tv | unary of set | binary of rel
   end
```

Wie man eine Grammatik schreibt und für diese einen Parser erzeugt, der natürlichsprachliche Ausdrücke in die abstrakte Syntax der Peirce-Terme übersetzt, werden wir in Abschnitt 1.2 und 2 sehen.

1.1.2 Die volle Peirce'sche Algebra über einer Grundmenge

Für endliche Mengen U kann man die volle Peirce'sche Algebra über U als Struktur in SML definieren. Dazu verwenden wir ein Paket `Set`, das eine einfache Implementierung (kleiner) endlicher Mengen zur Verfügung stellt.

Ein Modul für endliche Mengen

Endliche Mengen und Relationen kann man mit Hilfe von Listen implementieren. Dabei muß sichergestellt werden, daß die Anordnung der Elemente innerhalb der Liste dem Anwender unbekannt bleibt und bei der Anwendung von Mengenoperationen keinen (denotationellen) Einfluß hat. Ebenso dürfen in der Listen mehrfach vorkommende Elemente nur einmal in der Menge vorkommen.

⟨*PA/set.sml*⟩≡

```
signature SET = (* Naive implementation of small sets via lists. *)
  sig
    type 'a set
    val empty : 'a set
    val add : ''a -> ''a set -> ''a set
    val difference : ''a set -> ''a set -> ''a set
    val drop : ''a -> ''a set -> ''a set
```

```
    val equal : ''a set -> ''a set -> bool
    val intersection : ''a set -> ''a set -> ''a set
    val intersects : ''a set -> ''a set -> bool
    val list : 'a set -> 'a list
    val map : ('a -> ''b) -> 'a set -> ''b set
    val member : ''a -> ''a set -> bool
    val mk : ''a list -> ''a set
    val select : 'a set -> ('a -> bool) -> 'a set
    val size : 'a set -> int
    val singleton : 'a -> 'a set
    val subset : ''a set -> ''a set -> bool
    val union : ''a set -> ''a set -> ''a set
    val Union : ''a set list -> ''a set
  end

structure Set : SET =
    struct
        datatype 'a set = Set of 'a list

        val empty = Set []

        fun member x (Set []) = false
          | member x (Set (y::ys)) = (x = y) orelse (member x (Set ys))
        and add x (Set S) =
            let fun addLst x [] = [x]
                  | addLst x (u::U) = if x=u then (u::U) else u::(addLst x U)
            in Set (addLst x S) end
        and intersects (Set []) Q = false
          | intersects (Set (p::P)) Q =
            (member p Q) orelse (intersects (Set P) Q)
        and intersection (Set []) Q = Set []
          | intersection (Set (p::P)) Q =
            if member p Q then add p (intersection (Set P) Q)
            else intersection (Set P) Q
        and union (Set []) U = U
          | union (Set (x::Q)) U = (* x is not in Q! *)
            union (Set Q) (add x U)
        and Union sets =
            let
                fun bigU [] (Set U) = (Set U)
                  | bigU (Q::Qs) U = bigU Qs (union Q U)
            in
                bigU sets empty
            end
        and drop e (Set S) =
            let fun subtr [] e = []
```

```
                | subtr (s::S) e = if s=e then S (* s not in S! *)
                                   else (s::(subtr S e))
          in Set (subtr S e) end
      and difference (Set S) (Set T) =
          let fun diff (Set S) [] = S
                | diff (Set S) (e::E) = diff (drop e (Set S)) E
          in Set (diff (Set S) T) end

      fun size (Set S) = List.length S
      and mk L = Union (map (fn x => Set [x]) L)
      and list (Set S) = S
      and select (Set S) f =
          let fun selct [] f = []
                | selct (s::S) f = if f s then (s::(selct S f))
                                   else (selct S f)
          in Set (selct S f) end

      fun subset S T =
          let fun f x = member x T
              val U = select S f
          in size U = size S end
      and equal S T = (size S = size T) andalso (subset S T)

      fun map f (Set S) = mk (List.map f S)
      fun singleton e = (Set [e])
  end
```

Wenn man durch `Set:>SET` die Mengen zu einem abstrakten Datentyp macht, kann man die Elemente einer Menge `s` nur mit `Set.list(s)` sehen.

Universen U mit benannten Mengen und Relationen

Als nächstes wollen wir aus einem Universum U die volle Peirce-Algebra über U definieren. Ein Universum soll den (i.a. unendlichen) Typ seiner Elemente und eine konkrete (endliche!) Liste aller Elemente angeben. Außerdem soll man benannte Mengen und Relationen vorgeben können:

⟨*PA/peirce.sml*⟩+≡

```
signature UNIVERSE =
   sig
       eqtype element
       val elements : element list
       val ConS : string -> element list
       val ConR : string -> (element * element) list
       exception NoValue
   end
```

Da der Typ `element` (dessen Elemente mit `=` vergleichbar sein müssen, deshalb `eqtype`) i.a. eine unendliche Menge von SML-Ausdrücken enthält, wir aber ein endliches Universum haben wollen, braucht man die Liste `elements`.

Konstruktion einer vollen Peirce-Algebra über *U*

Die Signatur einer vollen Peirce-Algebra soll neben den Typen und Operationen aus `PEIRCE` auch den Typ der Individuen und wieder zwei Funktionen `ConS` und `ConR` enthalten, mit denen man Elemente der beiden Bereiche angeben kann:

⟨*PA/peirce.sml*⟩+≡

```
signature PEIRCEfull =
   sig
       include PEIRCE
       type element
   end

functor Peirce(U : UNIVERSE) : PEIRCEfull =
   struct
     type element = U.element
     fun ConS s = Set.mk (U.ConS s)
     fun ConR s = Set.mk (U.ConR s)

     type set = element Set.set

     val UnitS = Set.mk U.elements
     val ZeroS = Set.empty
     fun SumS(A,B) = Set.union A B
     fun ProdS(A,B) = Set.intersection A B
     fun CompS(A) = Set.difference UnitS A

     type rel = (element * element) Set.set

     fun Id(A) = Set.map (fn x => (x,x)) A
     val I = Id(UnitS)
     fun Cart(A,B) =
         let
             fun f x = List.map (fn y => (x,y)) (Set.list B)
         in
             Set.mk (List.foldr (op @) nil (List.map f (Set.list A)))
         end
     val UnitR:rel = Cart(UnitS,UnitS)
     fun Cylin(A) = Cart(A,UnitS)
     val ZeroR = Set.empty
     fun SumR(R,S) = Set.union R S
```

```
    fun ProdR(R,S) = Set.intersection R S
    fun CompR(R) = Set.difference UnitR R
    fun Conv(R) = Set.map (fn (x,y) => (y,x)) R

    fun Prod(R,S) =
        let
            fun f(a,b) = Set.map (fn (x,c) => (a,c))
                                 (Set.select S (fn (x,c) => x = b))
        in
            Set.Union (List.map f (Set.list R))
        end

    fun PreIm(R,B) =
        Set.map #1 (Set.select R (fn (a,b) => Set.member b B))
    fun Minor(R) =
        let
            val domain = PreIm(R,UnitS)
            val range = PreIm(Conv R,UnitS)
            val S = ProdR(R,Cart(range,domain))
        in
            if Set.size(R) = Set.size(S) then ZeroR else
            SumR(ProdR(R,Cart(CompS range,range)),Minor(S))
        end

    datatype tv = datatype bool

    fun E(A) = (Set.size A > 0)
    fun N(A) = (Set.size A = 0)
    fun U(A) = ((Set.size A) = (Set.size UnitS))

    fun ER(A) = (Set.size A > 0)
    fun UR(A) = ((Set.size A) = (Set.size UnitR))
    fun NR(A) = (Set.size A = 0)

    fun And(A,B) = A andalso B
    fun Or(A,B)  = A orelse B
    fun Neg(A)  = not A
  end
```

Mit dem Funktor `Peirce` und seinen Funktionen `ConS` und `ConR` kann man eine volle Peirce-Algebra mit benannten Mengen und Relationen konstruieren:[3]

Beispiel 1.1 Ein Universum `U` aus „wirklichen“ Mengen und Relationen über

[3] In der Beschreibungsdatei `.cm` muß man daher das Paket `Set` über Mengen einbauen. Statt einer Interpretation von Mengen und Relationen durch Listen könnte man sie durch ihre charakteristischen Funktionen, also Boole'schwertige Funktionen, darstellen.

einer endlichen Grundmenge und die mit Hilfe von `ConS` und `ConR` benannten Mengen und Relationen über dem Universum muß man vorgeben.

⟨*PA/pg1.sml*⟩≡
```
structure Personen =
   Peirce(datatype element = a | b | c | d | e | f | g | h | m
          val elements = [a,b,c,d,e,f,g,h,m]

          exception NoValue

          fun ConS "arbeiten" = [a,e]
            | ConS "gesund"   = [a,d]
            | ConS "Anna"     = [a]
            | ConS "Maria"    = [m]
            | ConS "Dieter"   = [d]
            | ConS "Emil"     = [e]
            | ConS "Grete"    = [g]
            | ConS "Hans"     = [h]
            | ConS "Frau"     = [a,g,m]
            | ConS "Mann"     = [d,e,h]
            | ConS "Student"  = [a,e]
            | ConS _ = raise NoValue

          fun ConR "kennen"   = [(a,b),(b,c)]
            | ConR "lieben"   = [(b,d),(c,e),(b,a),(a,a),(e,a),(h,g)]
            | ConR "begrüßen" = [(a,m),(a,g),(m,a)]
            | ConR "Geschwister" =
              let
                  val gs = [(a,d),(a,m),(a,g),(m,g),(g,d),(m,d)]
              in
                  gs @ (map (fn (x,y) => (y,x)) gs)
              end
            | ConR "Bruder"   = [(d,a),(d,g),(d,m)]
            | ConR "Schwestern" = [(m,a),(a,m),(g,a),(a,g),(m,g),(g,m)]
            | ConR "Nachbar"  = [(d,e),(e,a)]
            | ConR "älter"    = [(d,a),(d,e),(a,e)]
            | ConR _ = raise NoValue
         )
```

Die Ausnahme `NoValue` dient zur Erkennung der Fälle, in denen ein Wort zwar vom Lexer als Token erkannt wird, ihm aber in der Peirce-Algebra kein Wert zugeordnet wurde. Wir werden diese und andere Interpretationen zur Auswertung natürlichsprachlicher Ausdrücke verwenden.

1.2 Peirce'sche Grammatiken

Eine *Peirce'sche Grammatik* $PG = \langle G, P, typ, v, e\rangle$ besteht aus

(a) einer kontextfreien Grammatik $G = (T, N, S, R)$, mit Terminalsymbolen T, Nonterminalsymbolen N, einer Satzkategorie $S \in N$, und Grammatikregeln $R \subseteq N \times (T \cup N)^*$,

(b) einer Peirce'schen Algebra $P = (set, rel, \ldots)$,

(c) einer Typisierung $typ : N \cup T \to \texttt{set}, \texttt{rel}$, die jedem Symbol der Grammatik einen Bedeutungstyp von P zuordnet,

(d) einer Bewertung $v : T \to P$, die jedem Terminalsymbol X der Grammatik ein Element aus P vom Typ $typ(X)$ zuordnet,

(e) einer Zuordnung e, die jeder Grammatikregel $A \to A_1 \cdots A_k$ einen Peirce-Term $t(X_1, \ldots, X_k)$ zuordnet, so daß

$$\lambda(X_1, \ldots, X_k).t \ : \ typ(A_1) \times \cdots \times typ(A_k) \to typ(A).$$

Jedem Ausdruck w der Kategorie $A \in N$ kann in eindeutiger Weise ein *Wert* $[\![w]\!]$ aus P von der Art $typ(A)$ zugeordnet werden, indem man

$$[\![w]\!] := t^P([\![w_1]\!], \ldots, [\![w_k]\!])$$

nimmt, wenn w aus $w_1, \ldots, w_k$ mit der Regel $A \to A_1 \cdots A_k$ gebildet und t der dieser Regel mit e zugeordnete Peirce-Term ist.

Wir geben eine Peirce-Grammatik im Stil von ML-Yacc an, wo zu den Grammatikregeln

$$A \to \alpha_1 \mid \cdots \mid \alpha_n$$

bei jeder Alternative α_i der zugehörige Peirce-Term t_i in Klammern steht:

$$A \to \alpha_1\,(t_1) \mid \cdots \mid \alpha_n\,(t_n).$$

Angabe einer Peirce-Grammatik im Stil von ML-Lex/Yacc

Wir wollen eine Peirce-Grammatik so angeben, daß man daraus

(a) mit ML-Lex einen Lexer/Tokenizer erzeugen kann, der die Zerlegung der Eingabe in eine Folge von *Token*, Wörtern mit ihrer Bedeutung, besorgt,

(b) mit ML-Yacc einen Parser erzeugen kann, der die vom Tokenizer in eine Tokenfolge umgewandelte Eingabe syntaktisch gruppiert,

(c) mit dem Parser die Bedeutung der Eingabe, ihren Wert in einer gegebenen Peirce-Algebra, berechnen kann.

Wir betrachten zuerst nur (b) und (a) am Beispiel.

Beispiel 1.2 Als Grammatik nehmen wir die ersten Regeln aus Böttner[2] und als Werte die Terme der Peirce-Algebra. Die Grammatikregeln mit den Auswertungstermen werden in `pa.grm` angegeben, das Lexikon in `pa.lex`.

Sowohl im Lexikon `pa.lex` als auch in der Grammatik `pa.grm` müssen wir uns auf die Peirce-Terme beziehen. Zur besseren Lesbarkeit führen wir als Kurzname eine weitere Bezeichnung für diese Struktur ein:[4]

⟨*PA/pa.sml*⟩≡
```
structure D = Absyn : ABSYN
```

Die in `pa.lex` verwendeten lexikalischen Kategorien müssen als Terminalsymbole der Grammatik `pa.grm` definiert werden.

(a) Angabe der Grammatikregeln: Jede lexikalische und syntaktische Kategorie muß mit dem Typ ihrer semantischen Werte deklariert werden. Bei jeder Alternative einer Grammatikregel muß der Peirce-Term stehen, mit dem bei der Verwendung der Regel die Bedeutung berechnet werden soll.

⟨*PA/pa.grm*⟩≡
```
open D   (* ProdS statt D.ProdS usw.
            Trotzdem: D.set noetig fuer sig NL__Tokens *)

%%
%header (functor NL_LrValsFun (structure Token : TOKEN) : NL__LRVALS)

%term
     EOF | PUNKT
   | ist
   | ein | eines
   | PN of D.set
   | N1 of D.set
   | N2 of D.rel
   | IV of D.set
   | TV of D.rel
   | CA of D.set
   | EQ | NQ | UQ   (* Quantoren: ein, kein, jedes *)

%nonterm
   start of absyn option
   | S  of D.tv
   | NP of D.set
   | AP of D.set
   | VP of D.set
```

[4] Der Name der Struktur, `D`, ist analog zu $\mathcal{D} = (D_\sigma, c^D_\sigma)_{\sigma\in\mathrm{Typ}}$ bei Montague.

```
   | IVP of D.set
   | TVP of D.set

%start start

%eop EOF PUNKT
%noshift EOF PUNKT
%pos int

%name NL_
%verbose

%%

start : S             (print (TtoString S ^ "\n"); SOME(nullary(S)))
     | NP             (print (StoString NP ^ "\n"); SOME(unary(NP)))
     | PN             (print (StoString PN ^ "\n"); SOME(unary(PN)))
     | VP             (print (StoString VP ^ "\n"); SOME(unary(VP)))
     |                (print "Value is neiter S, PN, NP, nor VP\n"; NONE)

S :  PN VP            (E(ProdS(PN,VP)))   (* R1 *)
VP : IVP              (IVP)               (* R2a *)
     | TVP            (TVP)               (* R2b *)
IVP : IV              (IV)                (* R2 *)
     | ist AP         (AP)                (* R3 *)
     | ist ein NP     (NP)                (* R5 *)

TVP : TV PN           (PreIm(TV,PN))      (* R7 *)
     | TV UQ NP       (CompS(PreIm(Conv(CompR(Conv(TV))),NP)))  (* R19 *)
     | TV EQ NP       (PreIm(TV,NP))         (* R20 *)
     | TV NQ NP       (CompS(PreIm(TV,NP)))  (* R21 *)

AP : CA               (CA)                (* R4 *)
NP : N1               (N1)                (* R6 *)
     | N2 eines NP    (PreIm(N2,NP))      (* Bruder eines Studenten *)
```

(b) Angabe eines Lexikons: Die lexikalische Analyse einer Eingabe von Zeichen erfolgt durch einen Tokenizer und Lexer, der von ML-Lex aus der folgenden Definition erzeugt wird.

⟨*PA/pa.lex*⟩≡

```
structure Tokens = Tokens
structure Interface = Interface

open Interface
```

```
type pos = Interface.pos
type svalue = Tokens.svalue
type ('a,'b) token = ('a,'b) Tokens.token
type lexresult = (svalue,pos) token

fun eof() = Tokens.EOF(!line,!line)
fun makeInt (s : string) = s

fun N1(s:string) = Tokens.N1(Absyn.ConS s,!line,!line)
fun N2(s:string) = Tokens.N2(Absyn.ConR s,!line,!line)
fun PN(s:string) = Tokens.PN(Absyn.ConS s,!line,!line)
fun IV(s:string) = Tokens.IV(Absyn.ConS s,!line,!line)
fun TV(s:string) = Tokens.TV(Absyn.ConR s,!line,!line)
fun CA(s:string) = Tokens.CA(Absyn.ConS s,!line,!line)

%%
%header (functor NL_LexFun(structure Tokens: NL__TOKENS
                           structure Interface: INTERFACE) : LEXER);
%full

whitespace=[\t\ ]*;
%%
<INITIAL>{whitespace} => (lex());
\n          => (next_line(); lex());
"."         => (Tokens.PUNKT(!line,!line));
"ist"       => (Tokens.ist(!line,!line));
("ein" | "eine")
            => (Tokens.ein(!line,!line));
"eines"     => (Tokens.eines(!line,!line));
"einen"     => (Tokens.EQ(!line,!line));
("keinen" | "kein" | "Kein" | "Keine" | "keine")
            => (Tokens.NQ(!line,!line));
("Alle" | "alle" | "Jeder" | "jeder"
                 | "Jede" | "jede" | "Jeden" | "jeden")
            => (Tokens.UQ(!line,!line));
"Anna"      => (PN "Anna");
"Dieter"    => (PN "Dieter");
"Emil"      => (PN "Emil");
"Maria"     => (PN "Maria");
"Mann"      => (N1 "Mann");
"Frau"      => (N1 "Frau");
"Hund"      => (N1 "Hund");
("Student" | "Studenten")
            => (N1 "Student");
"Nachbar"   => (N2 "Nachbar");
"Bruder"    => (N2 "Bruder");
```

```
"gesund"   => (CA "gesund");
("arbeiten" | "arbeitet") => (IV "arbeiten");
("kennen" | "kennt")      => (TV "kennen");
("lieben" | "liebt")      => (TV "lieben");
.          => (error ("ignoring illegal character " ^ yytext,
                    !line,!line); lex());
```

Zur Beschreibung von ML-Lex und ML-Yacc siehe die Online-Dokumentation (via `smlnj.org`) oder Kapitel 2.5 und 3.4 aus [8].

2 Erstellung eines Parsers mit ML-Lex und ML-Yacc

ML-Lex erzeugt aus `*.lex` einen Lexer, der in die Datei `*.lex.sml` geschrieben wird. (Er enthält auch die am Anfang der Grammatik `*.grm` vom Benutzer definierten Funktionen.)

2.1 Schnittstelle des Lexers

Beim Lesen einer Eingabe soll der Lexer die jeweilige Zeile und die Positionen innerhalb einer Zeile mitzählen, so daß diese beim Auftreten eines Fehlers mit einer Fehlermeldung ausgegeben werden können.

Damit bei jeder neuen Initialisierung ein neuer Zähler verwendet wird, ist das folgende `Interface` keine Struktur, sondern ein 0-stelliger Funktor.

⟨*PA/nl.interface.sml*⟩≡

```
signature INTERFACE =
  sig
      type pos
      val line : pos ref
      val init_line : unit -> unit
      val next_line : unit -> unit
      val error : string * pos * pos -> unit

      type arg
      val nothing : arg
  end

functor Interface () : INTERFACE =
  struct
    type pos = int
    val line = ref 0
    fun init_line () = (line := 1)
    fun next_line () = (line := !line + 1)
    fun error (errmsg,line:pos,_) =
```

```
        print("Line " ^ (Int.toString line) ^ ": " ^ errmsg ^ "\n")

    type arg = unit
    val nothing = ()
  end
```

2.2 Verbindung von Lexer und Grammatik

Nachdem ML-Lex den Lexer und ML-Yacc den Parser erzeugt haben, müssen diese mit einander verbunden werden, um die Stuktur `NL_Parser` zu erhalten:[5]

Für den Fall, daß man `Absyn` nicht als Parameter übergeben will, genügt

⟨*PA/pa.link.sml*⟩≡

```
structure Interface : INTERFACE = Interface();

structure NL_LrVals : NL__LRVALS =
   NL_LrValsFun(structure Token = LrParser.Token);

structure NL_Lex : LEXER =
   NL_LexFun(structure Tokens = NL_LrVals.Tokens
             structure Interface = Interface);

structure NL_Parser : PARSER =
   Join(structure ParserData = NL_LrVals.ParserData
        structure Lex = NL_Lex
        structure LrParser = LrParser);
```

Für den Parser mit Auswertungsfunktion braucht man `eval : absyn -> value`, weshalb wir `Absyn` als Argument übergeben:

⟨*PA/pg.link.sml*⟩≡

```
structure Interface : INTERFACE = Interface();

structure NL_LrVals : NL__LRVALS =
   NL_LrValsFun(structure Token = LrParser.Token
                structure Absyn = Absyn);

structure NL_Lex : LEXER =
   NL_LexFun(structure Tokens = NL_LrVals.Tokens
             structure Interface = Interface);
```

[5] Der Name `NL_` wird vom Benutzer als `%name` und im `%header` in der Grammatik `*.grm` und dem Lexer `*.lex` vergeben; wir lassen ihn für alle Beispiele unverändert!

```
structure NL_Parser : PARSER =
   Join(structure ParserData = NL_LrVals.ParserData
        structure Lex = NL_Lex
        structure LrParser = LrParser);
```

Beachte, daß hier die Argumente von `NL_LrValsFun` und `NL_LexFun` zu den `%header`-Deklarationen in `*.grm` und `*.lex` passen müssen.

Der `NL_Parser` erzeugt Werte `ParserData`, die Peirce-Terme von `Absyn`.

2.3 Parser in die abstrakte Syntax der Peirce-Terme

Wenn man nur eine Syntaxanalyse, also einen Parser ohne Auswertungsfunktion braucht, kann man die Terme der Peirce-Algebra als Werte betrachten.

Den von ML-Yacc erzeugten Parser `NL_Parser` ruft man mit

`NL_Parser.parse(<lookahead>,<tokenstream>,<print_error>,<nothing>)`

auf. Dazu muß man mit `NL_Parser.makeLexer` einen Tokenstrom `lexer` erzeugen; für `pa.grm` bilden wir ihn mit einem `reader`, durch Lesen von der Konsole.

Am besten verpackt man die Hilfsfunktionen in einer eigenen Struktur, die eine Funktion `parse` und eine Ausnahme (für Lesefehler) enthält:

⟨*PA/nl.parse.sml*⟩≡
```
signature PARSE =
  sig
    type svalue
    type pos
    type ('a,'b) token
    exception Undefined
    val parse : unit -> (svalue,pos) token
  end
```

Einen Modul `NL` setzen wir unten aus der Schnittstelle `Interface`, den `Token` von `NL_LrVals` und dem Parser `NL_Parser` mit folgendem Funktor zusammen:

⟨*PA/nl.parse.sml*⟩+≡
```
functor Parse (structure Interface : INTERFACE
               structure Parser : PARSER
                  sharing type Parser.arg = Interface.arg
                  sharing type Parser.pos = Interface.pos
               structure Tokens : NL__TOKENS
                  sharing type Tokens.token = Parser.Token.token
                  sharing type Tokens.svalue = Parser.svalue
               ) : PARSE =
  struct
    type pos = Parser.pos
    type svalue = Parser.svalue
```

```
      type ('a,'b) token = ('a,'b) Parser.Token.token

      exception Undefined
      fun parse () =
          let
            fun reader (n:int) =
                case (TextIO.inputLine TextIO.stdIn) of
                  SOME s => s
                | NONE => raise Undefined
            (* Für SML 110.0.3 stattdessen:
               fun reader (n:int) = TextIO.inputLine TextIO.stdIn *)

            val _ = Interface.init_line()
            val empty = !Interface.line

            val dummyEOF = Tokens.EOF(empty,empty)

            fun loop lexer =
                let
                    val (result,lexer) =
                        Parser.parse(1,lexer,Interface.error,
                                     Interface.nothing)
                    val (nextToken,lexer) = Parser.Stream.get lexer
                in
                    if Parser.sameToken(nextToken,dummyEOF)
                    then nextToken else loop lexer
                end
          in
            loop (Parser.makeLexer reader)
          end
    end
```

Anwendung des Funktors `Parse` ohne Auswertung

Die von ML-Lex und ML-Yacc erzeugten Teilen `NL_LrVals.Tokens` und `NL_Parser` werden durch `Parse` zusammengesetzt:

⟨*PA/nl.parse.sml*⟩+≡

```
  structure NL : PARSE =
     Parse (structure Interface = Interface
            structure Parser = NL_Parser
            structure Tokens = NL_LrVals.Tokens );
```

Bem. Das Zusammenbauen der Teile, d.h. den Funktor `Parse`, kann man wohl nicht ohne `pa.grm` und `pa.lex` definieren, da dazu zumindest die Signatur `NL__TOKENS` nötig ist, die erst ML-Yacc aus `pa.grm` bildet und in `pa.grm.sig` schreibt. Sonst könnte man die Grammatik und das Lexikon später einklinken.

(Mit höherstufigen Funktoren soll das aber irgendwie gehen!)

Erstellung des `NL`-Parsers für `pa.grm` ohne Auswertung

Bem. Bisher wird nur die Interpretation `D` durch Peirce-Terme definiert; dazu braucht man die Struktur `Set` noch nicht. Sie wird nur dazugeladen, weil sie der Definition des Funktors `Peirce` in `peirce.sml` benutzt wird.

$\langle$*PA/pa.cm*$\rangle\equiv$

```
Library
  structure NL
is
  $/smlnj-lib.cm
  $/ml-yacc-lib.cm
  $/basis.cm

  set.sml

  peirce.sig
  peirce.sml
  pa.sml  (* definiert structure D = Absyn *)
  pa.lex
  pa.grm

  nl.interface.sml
  nl.parse.sml
  pa.link.sml
(*  nl.link.sml   an der LMU unter sml-110.0.42: nl.link.lmu.sml *)
```

Um `NL` zu konstruieren, ruft man nun in SML nur `CM.make PA/pa.cm";` auf.

2.4 Parser mit Werten in einer Peirce-Algebra

Um die Peirce-Terme, die der `NL_Parser` aus der Eingabe erzeugt, in einer vollen Peirce-Algebra interpretieren zu können, brauchen wir eine Auswertung `eval`, die einem Ausdruck in abstrakter Syntax einen passenden Wert in der Algebra zuordnet.[6] Weil die Funktion `eval : absyn -> value` eindeutige Argument- und Werttypen braucht, aber die Peirce-Algebren zwei Arten von Objekten

[6] Vielleicht sollte man `E,U,N : set -> set` doch zu `E,U,N : set -> boole` ändern. Bei den Termen müßten `E,U,N` Konstruktoren und `boole` ein syntaktischer Typ sein. Bei der Auswertung kann man das eingebaute `bool` benutzen und z.B. `N : set -> bool` via `N(A) = Bool.not(E(A))` auf `E:set ->`
`bool` zurückführen. Ob man das machen sollte, hängt davon ab, ob irgendwo die Mengenoperationen auf die Ausgabewerte von E, N, U angewendet werden, ohne daß man die Wahrheitswerte meint. (Werden `E,U,N` geschachtelt verwendet?)

haben, wird das etwas umständlich.

⟨*PA/semantik.sml*⟩≡
```
signature SEM = sig
                    type absyn
                    type value
                    val eval : absyn -> value
                end

functor Semantik(structure A:ABSYN
                 structure P:PEIRCEfull) : SEM =
    struct
       datatype absyn = datatype A.absyn
       datatype value = nu of P.tv | un of P.set | bi of P.rel

       fun eval(A.unary X) = un(evalS X)
         | eval(A.binary X) = bi(evalR X)
         | eval(A.nullary X) = nu(evalT X)
       and evalS(A.ConS s) = P.ConS s
         | evalS(A.UnitS) = P.UnitS
         | evalS(A.ZeroS) = P.ZeroS
         | evalS(A.SumS(A,B)) = P.SumS(evalS A,evalS B)
         | evalS(A.ProdS(A,B)) = P.ProdS(evalS A,evalS B)
         | evalS(A.CompS A) = P.CompS(evalS A)
         | evalS(A.PreIm(R,A)) = P.PreIm(evalR R,evalS A)
       and evalR(A.ConR s) = P.ConR s
         | evalR(A.Id A) = P.ProdR(P.I, P.Cylin(evalS A))
         | evalR(A.UnitR) = P.UnitR
         | evalR(A.ZeroR) = P.ZeroR
         | evalR(A.SumR(S,T)) = P.SumR(evalR S,evalR T)
         | evalR(A.ProdR(S,T)) = P.ProdR(evalR S,evalR T)
         | evalR(A.CompR T) = P.CompR(evalR T)
         | evalR(A.Minor T) = P.Minor(evalR T)
         | evalR(A.Conv T) = P.Conv(evalR T)
         | evalR(A.Prod(S,T)) = P.Prod(evalR S,evalR T)
         | evalR(A.Cylin A) = P.Cylin(evalS A)
       and evalT(A.E A) = P.E(evalS A)
         | evalT(A.U A) = P.U(evalS A)
         | evalT(A.N A) = P.N(evalS A)
         | evalT(A.ER A) = P.ER(evalR A)
         | evalT(A.UR A) = P.UR(evalR A)
         | evalT(A.NR A) = P.NR(evalR A)
         | evalT(A.And(A,B)) = P.And(evalT A,evalT B)
         | evalT(A.Or(A,B)) = P.Or(evalT A,evalT B)
         | evalT(A.Neg A) = P.Neg(evalT A)
    end
```

Neben der Auswertung `eval` der Terme in einer vorgegebenen vollen Peirce-Algebra braucht man noch Funktionen, mit denen der Parser auf Top-Level aufgerufen werden kann.

Dazu wollen wir eine Funktion `parse`, die die Eingabe in einen Peirce-Term umwandelt und diesen in lesbarer Form als Zeichenreihe ausgibt. Außerdem wollen wir eine Variante `ev` haben, die den Ausdruck in abstrakter Syntax ausgibt, d.h. einen (als `absyn` verpackten) Peirce-Term.

⟨*PA/pg.parse.sml*⟩≡

```
signature PARSE =
  sig
      structure Absyn : ABSYN
      val parse : unit -> (NL_Parser.svalue,NL_Parser.pos)
                              LrParser.Token.token
      exception Undefined
      val ev : string -> Absyn.absyn

      structure Semantik : SEM
      val eval : Absyn.absyn -> Semantik.value

      val evals : string -> Semantik.value
  end
```

Bem. Vorläufig wird der abstrakte Term und der Wert in der Algebra nur bei Eingabe einer Zeichenreihe ausgegeben. Die genaue Bedienung und damit die Signatur `PARSE` kann sich noch ändern.

⟨*PA/pg.parse.sml*⟩+≡

```
functor Parse (structure Absyn : ABSYN
               structure Algebra : PEIRCEfull
               structure Interface : INTERFACE
               structure Parser : PARSER
                  sharing type Parser.arg = Interface.arg
                  sharing type Parser.pos = Interface.pos
                  sharing type Parser.result = Absyn.absyn
               structure Tokens : NL__TOKENS
                  sharing type Tokens.token = Parser.Token.token
                  sharing type Tokens.svalue = Parser.svalue
               ) : PARSE =
  struct
    structure Absyn = Absyn

    exception Undefined

    fun parse () =
        let
```

```
        val dummyEOF = NL_LrVals.Tokens.EOF(0,0)

        fun invoke lexer =
            let fun error (s,i:int,_) =
                    print("Error, line " ^ (Int.toString i) ^
                          ", " ^ s ^ "\n")
            in
                NL_Parser.parse(0,lexer,error,())
            end

        fun loop lexer =
            let val (result,lexer) = invoke lexer
                val (nextToken,lexer) = NL_Parser.Stream.get lexer
                (* Fehler bei Absyn statt D:
                   rule domain: Absyn.absyn =/= case object: NL_Parser.result *)
                val _ = case result of
                            (D.unary s) =>
                          print("Value is: " ^ (D.StoString s) ^ "\n")
                          | (D.binary s) =>
                          print("Value is: " ^ (D.RtoString s) ^ "\n")
                          | (D.nullary s) =>
                          print("Value is: " ^ (D.TtoString s) ^ "\n")
            in if NL_Parser.sameToken(nextToken,dummyEOF)
               then nextToken else loop lexer
            end

        fun reader (n:int) =
            case (TextIO.inputLine TextIO.stdIn) of
                SOME s => s
              | NONE => raise Undefined
        (* Für SML 110.0.3 stattdessen:
           fun reader (n:int) = TextIO.inputLine TextIO.stdIn *)
    in
      loop (NL_Parser.makeLexer reader)
    end

fun ev (s:string) =
    let
        val lookahead = 5
        val _ = Interface.init_line()
        val empty = !Interface.line

        val dummyEOF = Tokens.EOF(empty,empty)

        fun invoke lexer =
            Parser.parse(lookahead,
```

```
                            lexer,
                            Interface.error,
                            Interface.nothing)
          fun loop lexer =
              let val (result,lexer) = invoke lexer
                  val (nextToken,lexer) = Parser.Stream.get lexer
              in if Parser.sameToken(nextToken,dummyEOF)
                 then result else loop lexer
              end

          fun string_reader s =
              let
                  val next = ref s
              in fn _ => !next before next := "" end
          val reader = string_reader s
      in
          loop (Parser.makeLexer reader)
      end

  structure Semantik = Semantik(structure A = Absyn:ABSYN
                                structure P = Algebra:PEIRCEfull)
  datatype value = datatype Semantik.value
  val eval = Semantik.eval

  fun evals s = eval(ev s)
end
```

Bem. Wie kann man in `parse()` dieselben Funktionen `invoke` und `loop` wie in `ev(s)` und `evals(s)` benutzen? Man muß statt `NL_Parser` den Argumentparser `Parser` verwenden, aber dann kann ich bei `case result` in `parse.loop` nicht auf die Konstruktoren `D.unary` zugreifen. (vgl. Parse bei `pa.cm`, aber dort habe ich das Drucken in die Grammatik verlagert, was ich hier nicht will.) Es ist vielleicht falsch, *zuerst* die Struktur `D = Absyn` zu definieren und erst dann Lexer und Parser zu bauen: dann wird `D` aus der Umgebung genommen.

Erstellung des Funktors `Parse` mit Auswertung

Wir isolieren die Konstruktion des Funktors `Parse`, mit dem aus einer vollen Peirce-Algebra ein Parser für eine Peirce-Grammatik `pg.grm` mit einem Lexikon `pg.lex` konstruiert werden kann.

Dann lassen sich mehrere Beispiele mit derselben Grammatik (`pg.grm` aus Abschnitt 3.1) und demselben Lexikon (`pg.lex` aus Abschnitt 3.2) behandeln, indem man die Bedeutung ausgewählter Ausdrücke in verschiedenen Peirce-Algebren definiert.

Man muß im Vergleich zu `pa.cm` hauptsächlich die etwas geänderten Funktionen aus `PA/pg.parse.sml` und natürlich den Funktor `Semantik` berücksichtigen.

⟨*PA/pg.parse.cm*⟩≡
```
Library
  structure Absyn
  functor Peirce
  functor Semantik
  signature PARSE
  functor Parse
  structure NL_Parser
  structure NL_LrVals
  structure Interface
is
  $/smlnj-lib.cm
  $/ml-yacc-lib.cm
  $/basis.cm

  set.sml           (* Set *)

  absyn.sml         (* Absyn:ABSYN *)
  peirce.sig        (* PEIRCE *)
  peirce.sml        (* PeirceTerm, Peirce *)
  pg.sml            (* structure D = Absyn *)
  pg.lex
  pg.grm

  nl.interface.sml  (* Interface *)
  semantik.sml      (* Semantik *)
  pg.link.sml       (* NL_Parser, NL_LrVals *)
  pg.parse.sml      (* Parse *)
```

Hat man mit `CM.make "pg.parse.cm";` diesen Funktor `Parse` und die angegebenen Hilfsstrukturen konstruiert, so erhält man aus einem Universum durch

⟨*Anwendung des Funktors Parse*⟩≡
```
structure NL =
  Parse(... structure Algebra = Peirce(U:UNIVERSUM):PEIRCEfull ...)
```

einen Parser mit Werten in dieser Algebra.

Anwendung des Funktors `Parse` mit Auswertung

Ein Parser, der eine Zeichenreihe in einen Peirce-Term übersetzt und diesen in einer Peirce-Algebra auswertet, wird schließlich durch Anwenden des Funktors `Parse` auf die gewünschte Peirce-Algebra konstruiert:

Beispiel 2.1 Zuerst machen wir das mit der Struktur `Personen`, die schon in dieser Datei definiert ist (vgl. Beispiel 1.1):

⟨*PA/pg1.sml*⟩+≡
```
structure NL : PARSE =
   Parse (structure Absyn = Absyn
          structure Algebra = Personen (* Algebra der Werte *)
          structure Interface = Interface
          structure Parser = NL_Parser
          structure Tokens = NL_LrVals.Tokens)
```

Ein Makefile für das Beispiel ist jetzt ganz kurz:

⟨*PA/pg1.cm*⟩≡
```
Group
   structure NL
is
   pg.parse.cm
   pg1.sml
```

Damit startet man in SML wieder `CM.make "pg1.cm";` (vom Verzeichnis `PA` aus) und kann dann die Auswertung testen:

⟨*Test der lesbaren Ausgabe der abstrakten Syntax*⟩≡
```
- NL.parse();
Anna arbeitet.
Value is: E((Anna . arbeiten))
```

⟨*Test einer Ausgabe in abstrakter Syntax*⟩≡
```
- NL.ev "Anna arbeitet";
val it = nullary (E (ProdS (ConS #,ConS #))) : ?.Absyn.absyn
```

⟨*Test einer Auswertung*⟩≡
```
- NL.evals "Anna arbeitet";
val it = nu true : NL.Semantik.value
- NL.evals "Dieter arbeitet";
val it = nu false : NL.Semantik.value
```

Beispiel 2.2 Eine andere Interpretation erhalten wir durch Anwenden des Funktors `Parse` auf eine andere Algebra. Wir erweitern die Algebra für das Beispiel von S.62 in Böttner [2], so daß auch Beispiele mit Präpositionen (S.69 in [2]) erfaßt werden.

⟨*PA/pg2.sml*⟩≡
```
structure P =
   Peirce(datatype element = Marseille | Palermo | Rom
                           | Sizilien | SP | IT | FR
          val elements = [FR, IT, Marseille, Rom, SP, Sizilien]

          exception NoValue
```

```
           fun ConS "Stadt"     = [Rom, Marseille]
             | ConS "Staat"     = [FR,IT,SP]
             | ConS "Insel"     = [Sizilien]
             | ConS "Marseille" = [Marseille]
             | ConS "Palermo"   = [Palermo]
             | ConS "Rom"       = [Rom]
             | ConS "Frankreich"= [FR]
             | ConS "Italien"   = [IT]
             | ConS "Spanien"   = [SP]
             | ConS "Sizilien"  = [Sizilien]
             | ConS _ = raise NoValue

           fun ConR "in"        = [(Rom,IT),(Marseille,FR),
                                    (Sizilien,IT),(Palermo,Sizilien)]
             | ConR "auf"       = [(Palermo,Sizilien)]
             | ConR _ = raise NoValue
           )

structure NL : PARSE =
   Parse (structure Absyn = Absyn
          structure Algebra = P
          structure Interface = Interface
          structure Parser = NL_Parser
          structure Tokens = NL_LrVals.Tokens)
```

Der passende Makefile dazu ist:

⟨*PA/pg2.cm*⟩≡

```
Library
  structure NL
is
  pg2.sml
  pg.parse.cm
```

Ein Paar Tests ergeben:

⟨*Tests von pg2*⟩≡

```
- NL.evals "Sizilien ist eine Insel";
val it = nu true : NL.Semantik.value
- NL.evals "Rom ist eine Insel";
val it = nu false : NL.Semantik.value
- NL.parse();
Rom ist eine Insel.
Value is: E((Rom . Insel))
```

3 Eine Peirce-Grammatik für deutsche Sätze: `pg.grm`

Bisher hat der Parser einen NL-Ausdrucks in einen Peirce-Term umgewandelt und diesen als Zeichenreihe am Bildschirm ausgedruckt. Normalerweise wollen wir den Ausdrücken aber (i.a. nicht druckbare) Werte zuordnen.

Dazu soll der von ML-Yacc erzeugte Parser eine natürlichsprachliche Eingabe nur in einen Peirce-Term, die *abstrakte Syntax* der Eingabe, umwandeln.

Wir betrachten statt der bisherigen Definition der Interpretation `D` (für Lexer und Parser!) aus `pa.sml`, nämlich `PeirceTerm`, die Erweiterung um diesen Datentyp `absyn`. Das definieren wir in

⟨*PA/pg.sml*⟩≡

```
structure D = Absyn
```

da die Struktur `D` sowohl in der Definition des Tokenizers `*.lex` als auch in der Grammatik `*.grm` benutzt wird.

Vorsicht: da jede Auswertung derselben Datentypdeklaration einen *anderen* Datentyp erzeugt, müssen die Funktoren `Parse` und `LrValsFun` dieselbe Argumentstruktur `Absyn` mitbekommen. (Sonst passen die Typen der Teminalsymbole nicht zu einander.)

3.1 Die Grammatik `pg.grm`

Zur Angabe der Auswertung definieren wir einige Hilfsunktionen durch die Grundoperationen: der Vorbereich `Dom` und der symmetrische Kern `SymK` einer Relation, das Bild `Im` und das universelle Urbild (oder die Involution) `UnPreIm` einer Menge unter einer Relation, das kartesische Produkt `Cart` zweier Mengen, der untere Teil `MinorS` einer Menge bezüglich einer (Vergleichs-) Relation, die Verschiedenheitsrelation `Div` auf einer Menge, die Nachbeschränkung `RestrR` einer Relation auf eine Menge.

⟨*PA/pg.grm*⟩≡

```
open D       (* ProdS statt D.ProdS usw.
                Trotzdem: D.set nötig für sig NL__Tokens *)

exception Undefined

fun Dom(r)         = PreIm(r,UnitS)
fun SymK(r)        = ProdR(r,Conv r)
fun Im(r,s)        = PreIm(Conv r,s)
fun UnPreIm(r,s)   = CompS(PreIm(CompR r,s))   (* r't alle s *)
fun Cart(s1,s2)    = ProdR(Cylin s1,Conv(Cylin s2))
fun MinorS(s,r)    = Dom(Minor(ProdR(r,Cart(s,s))))
fun Div(s)         = ProdR(Cart(s,s),CompR(Id(UnitS)))
fun PreImDiv(r,s) = Prod(Prod(r,Div s),Conv r)
```

```
fun RestrR(r,s)   = ProdR(r,Conv(Cylin s))

%%
%header (functor NL_LrValsFun (structure Token : TOKEN
                               structure Absyn : ABSYN) : NL__LRVALS)

%term
     EOF | PUNKT | Komma
   | und | oder | nicht
   | ist | sind
   | wird | werden
   | ein | eines
   | deren
   | zwar | aber | weder | noch
   | PN of D.set
   | PNgen of D.set
   | N1 of D.set  (*  N *)
   | N2 of D.rel  (* RN *)
   | IV of D.set
   | TV of D.rel
   | TVpart of D.rel
   | CA of D.set
   | IA of D.rel
   | IA_comp of D.rel
   | EQ | NQ | UQ   (* Quantoren: ein, kein, jedes *)
   | P of D.rel
   | von | als
   | N1pl of D.set
   | N2pl of D.rel
   | IVpl of D.set
   | TVpl of D.rel
   | CApl of D.set
   | IApl of D.rel
   | EQpl | NQpl | UQpl (* einige, keine, alle *)
   | SVpl of D.rel
   | RA of D.rel
   | Poss of D.rel

%nonterm
   start of absyn
   | S  of D.tv
   | NP of D.set
   | N2P of D.rel
   | AP of D.set
   | VP of D.set
   | IVP of D.set
```

```
     | NegIVP of D.set (* H.L. *)
     | NegTVP of D.set (* H.L. *)
     | TVP of D.set
     | PP of D.set
     | PassP of D.set
     | aPNP of D.set
     | cPNP  of D.set
     | pPNP  of D.rel
     | NPpl  of D.set
     | VPpl  of D.set
     | IVPpl of D.set
     | TVPpl of D.set
     | NegIVPpl of D.set
     | NegTVPpl of D.set
     | PPNP of D.rel
     | ColVPpl of D.rel
     | CNP of D.set
     | cNPune of D.set * D.set * D.set

%start start

%eop EOF PUNKT
%noshift EOF PUNKT
%pos int

%left und
%left oder

%name NL_
%verbose

%%

start : S                 (nullary S)
      | NP                (unary NP)
      | PN                (unary PN)
      | VP                (unary VP)
      | TV                (binary TV)
      | N2                (binary N2)

S :  PN VP                (E(ProdS(PN,VP)))       (* R 1 *)
      | UQ NP VP          (U(SumS(CompS(NP),VP))) (* R 13 *)
      | EQ NP VP          (E(ProdS(NP,VP)))       (* R 14 *)
      | NQ NP VP          (N(ProdS(NP,VP)))       (* R 15 *)
      | aPNP VP           (E(ProdS(aPNP,VP)))     (* R 54 *)
      | cPNP VPpl         (U(SumS(CompS(cPNP),
```

```
                                VPpl)))          (* R 53 *)
     | UQpl NPpl VPpl     (U(SumS(CompS(NPpl),VPpl))) (* R 13 *)
     | EQpl NPpl VPpl     (ER(PreImDiv(Id(VPpl),NPpl))) (* R 14b *)
     | NQpl NPpl VPpl     (N(ProdS(NPpl,VPpl)))   (* R 15 *)
(*   | S und pPNP         (SumS(S,pPNP))          (* R 59 *) FALSCH *)
     | cPNP ColVPpl       (ER(ProdR(Div(cPNP),
                                  ColVPpl)))    (* R 63a *)
     | UQpl NPpl ColVPpl  (UR(SumR(CompR(Div(NPpl)),
                                 ColVPpl)))     (* R 71a *)
     | EQpl NPpl ColVPpl  (ER(ProdR(Div(NPpl),
                                  ColVPpl)))    (* R 72a *)
     | NQpl NPpl ColVPpl  (NR(ProdR(Div(NPpl),
                                  ColVPpl)))    (* R 73a *)
     | cNPune VP          (And(U(SumS(CompS(#1 cNPune), VP)),
                              And(N(ProdS(#2 cNPune, VP)),
                                  E(ProdS(#3 cNPune, VP)))))

VP : IVP                  (IVP)                   (* R 2a *)
     | TVP                (TVP)                   (* R 2b *)
     | NegIVP             (NegIVP)                (* R 16a *)
     | NegTVP             (NegTVP)                (* R 16b *)
     | wird PassP         (PassP)                 (* R 45 *)

VPpl : IVPpl              (IVPpl)                 (* R 2a *)
     | TVPpl              (TVPpl)                 (* R 2b *)
     | NegIVPpl           (NegIVPpl)              (* R 16a *)
     | NegTVPpl           (NegTVPpl)              (* R 16b *)
     | werden PassP       (PassP)                 (* R 45 *)

IVP : IV                  (IV)                    (* R 2 *)
     | ist AP             (AP)                    (* R 3 *)
     | ist EQ NP          (NP)                    (* R 5a *)
     | ist PP             (PP)                    (* R 8 *)
     | ist NP             (NP)                    (* R 10 *)

IVPpl : IVpl              (IVpl)                  (* R 2  pl *)
     | sind AP            (AP)                    (* R 3  pl *)
     | sind NPpl          (NPpl)                  (* R 5a pl *)
     | sind PP            (PP)                    (* R 8  pl *)

NegIVP : ist nicht PP     (CompS(PP))             (* R 8b *)
     | ist NQ NP          (CompS(NP))             (* R 10b *)
     | ist nicht AP       (CompS(AP))             (* R 17a *)
     | ist nicht NP       (CompS(NP))             (* R 18 *)

TVP : TV PN               (PreIm(TV,PN))          (* R 7 *)
```

```
     | TV UQ NP           (UnPreIm(TV,NP))        (* R 19 *)
     | TV EQ NP           (PreIm(TV,NP))          (* R 20 *)
     | TV NQ NP           (CompS(PreIm(TV,NP)))   (* R 21 *)
     | TV cPNP            (UnPreIm(TV,cPNP))      (* R 55 *)
     | TV aPNP            (PreIm(TV,aPNP))        (* R 56 *)
     | TV UQpl NPpl       (UnPreIm(TV,NPpl))      (* R 19 *)
     | TV EQpl NPpl       (Dom(ProdR(PreImDiv(TV,NPpl),
                                    Id(UnitS)))) (* 20b *)
     | TV NQpl NPpl       (CompS(PreIm(TV,NPpl))) (* R 21 *)
     | TV PNgen N2pl      (UnPreIm(TV,PreIm(N2pl,PNgen)))
                                                  (* R 84a *)
     | TV Poss N2pl       (CompS(Dom(ProdR(CompR(TV),Conv(N2pl)))))
                                                  (* R 85 *)
     | TV PN und Poss N2pl (ProdS(PreIm(TV,PN),
                                  CompS(Dom(ProdR(CompR(TV),
                                                  Conv(N2pl))))))
                                                  (* R 86 *)
     | TV CNP             (UnPreIm(TV,CNP))       (* R 87a *)
     | TV Poss NPpl       (CompS(Dom(ProdR(RestrR(Poss,NPpl),
                                           CompR(TV)))))
                                                  (* R 89a *)
     | TV cNPune          (ProdS(UnPreIm(TV,#1 cNPune),
                                 ProdS(CompS(PreIm(TV,#2 cNPune)),
                                       PreIm(TV,#3 cNPune)))) (* Test *)

TVPpl : TVpl PN           (PreIm(TVpl,PN))        (* R 7 *)
     | TVpl UQ NP         (UnPreIm(TVpl,NP))      (* R 19 *)
     | TVpl EQ NP         (PreIm(TVpl,NP))        (* R 20 *)
     | TVpl NQ NP         (CompS(PreIm(TVpl,NP))) (* R 21 *)
     | TVpl cPNP          (CompS(PreIm(CompR(TVpl),
                                       cPNP)))    (* R 55 *)
     | TVpl aPNP          (PreIm(TVpl,aPNP))      (* R 56 *)
     | TVpl UQpl NPpl     (UnPreIm(TVpl,NPpl))    (* R 19 *)
     | TVpl EQpl NPpl     (PreIm(TVpl,NPpl))      (* R 20 *)
     | TVpl NQpl NPpl     (CompS(PreIm(TVpl,NPpl))) (* R 21 *)

NegTVP : TVP nicht        (CompS(TVP))            (* R 22 *)

NegTVPpl : TVPpl nicht    (CompS(TVPpl))          (* R 22 *)

AP : CA                   (CA)                    (* R 4 *)
     | IA_comp als PN     (PreIm(IA_comp,PN))     (* R 12 *)
     | IA_comp als UQ NP  (UnPreIm(IA_comp,NP))   (* R 26 *)
     | IA_comp als EQ NP  (PreIm(IA_comp,NP))     (* R 27 *)
     | IA_comp als NQ NP  (CompS(PreIm(IA_comp,NP)))
                                                  (* R 28 *)
```

```
NP : N1                  (N1)                     (* R 6 *)
     | N2 von PN         (PreIm(N2,PN))           (* R 11 *)
     | N2 eines NP       (PreIm(N2,NP))           (* R 11b *)
     | CA NP             (ProdS(CA,NP))           (* R 29 *)
     | IA NP             (MinorS(NP,IA))          (* R 31a *)

NPpl : N1pl              (N1pl)                   (* R 6 *)
     | N2pl von PN       (PreIm(N2pl,PN))         (* R 11 *)
     | N2pl eines NP     (PreIm(N2pl,NP))         (* R 11b *)
     | CApl NPpl         (ProdS(CApl,NPpl))       (* R 29 *)
     | IApl NPpl         (MinorS(NPpl,IApl))      (* R 31a *)

N2P : CA N2              (ProdR(N2,Cylin CA))     (* R 30 *)

PP : P PN                (PreIm(P,PN))            (* R 9 *)
     | P UQ NP           (UnPreIm(P,NP))          (* R 23 *)
     | P EQ NP           (PreIm(P,NP))            (* R 24 *)
     | P NQ NP           (CompS(PreIm(P,NP)))     (* R 25 *)

PassP : TVpart           (Im(TVpart,UnitS))       (* R 46 *)
     | von PN TVpart     (Im(TVpart,PN))          (* R 47 *)
     | von EQ NP TVpart  (Im(TVpart,NP))          (* R 48a *)
     | von UQ NP TVpart  (CompS(Im(CompR(TVpart),NP)))
                                                  (* R 49a *)
     | von NQ NP TVpart  (CompS(Im(TVpart,NP)))   (* R 50a *)

cPNP : PN und PN         (SumS(PN1,PN2))          (* R 51 *)
aPNP : PN oder PN        (SumS(PN1,PN2))          (* R 52 *)
pPNP : PN PN             (Cart(PN1,PN2))          (* R 61 *)
(* PPNP : PN und PN         (Cart(PN1,PN2))          (* R 64 *) *)

ColVPpl : sind N2pl      (SymK N2pl)              (* R 66 *)
     | sind RA           (SymK RA)                (* R 65 *)
     | SVpl              (SVpl)                   (* R 67a *)

CNP : PN und deren N2pl  (SumS(PN,PreIm(N2pl,PN))) (* R 88 *)

cNPune : UQ NP Komma NQ NP und EQ NP ((NP1,NP2,NP3)) (* Test *)
```

Noch nicht implementiert sind: a) Präpositionalphrasen, R 32–34, b) Relativsätze, R 35 – 42, c) Prosodie, R 43–44,

3.2 Tokenizer und Lexer `pg.lex`

Die lexikalische Analyse einer Eingabe von Zeichen erfolgt durch einen Tokenizer und Lexer, der von ML-Lex aus der folgenden Definition erzeugt wird. (Die lexikalischen Kategorien müssen als Terminalsymbole der Grammatik definiert werden, die den vom Lexer erzeugten Tokenstrom syntaktisch analysieren soll.[7])

Die folgende Definition wird (vorläufig?) für alle Grammatiken verwendet.

⟨*PA/pg.lex*⟩≡

```
structure Tokens = Tokens
structure Interface = Interface

open Interface

type pos = Interface.pos
type svalue = Tokens.svalue
type ('a,'b) token = ('a,'b) Tokens.token
type lexresult = (svalue,pos) token

fun eof() = Tokens.EOF(!line,!line)
fun makeInt (s : string) = s

fun N1(s) = Tokens.N1(D.ConS s,!line,!line)
fun N2(s) = Tokens.N2(D.ConR s,!line,!line)
fun PN(s) = Tokens.PN(D.ConS s,!line,!line)
fun PNgen(s) = Tokens.PNgen(D.ConS s,!line,!line)
fun IV(s) = Tokens.IV(D.ConS s,!line,!line)
fun TV(s) = Tokens.TV(D.ConR s,!line,!line)
fun TVpart(s) = Tokens.TVpart(D.ConR s,!line,!line)
fun CA(s) = Tokens.CA(D.ConS s,!line,!line)
fun IA(s) = Tokens.IA(D.ConR s,!line,!line)
fun IA_comp(s) = Tokens.IA_comp(D.ConR s,!line,!line)
fun  P(s) = Tokens.P(D.ConR s,!line,!line)
fun Poss() = Tokens.Poss(D.ConR "besitzen",!line,!line)

fun N1pl(s) = Tokens.N1pl(D.ConS s,!line,!line)
fun N2pl(s) = Tokens.N2pl(D.ConR s,!line,!line)
fun IVpl(s) = Tokens.IVpl(D.ConS s,!line,!line)
fun TVpl(s) = Tokens.TVpl(D.ConR s,!line,!line)
fun SVpl(s) = Tokens.SVpl(D.ConR s,!line,!line)
fun RA(s) = Tokens.RA(D.ConR s,!line,!line)

fun RestrL(r,s) = D.ProdR(D.ConR r, D.Cylin(D.ConS s))
```

[7] Durch weitere Abkürzungen wie `fun N(s) = ..` kann man es noch etwas lesbarer machen.

```
%%
%header (functor NL_LexFun(structure Tokens: NL__TOKENS
                           structure Interface: INTERFACE) : LEXER);
%full

alpha=[a-zA-ZÄÖÜäöüß];
whitespace=[\t\ ]*;
%%
<INITIAL>{whitespace} => (lex());
\n          => (next_line(); lex());
"."         => (Tokens.PUNKT(!line,!line));
","         => (Tokens.Komma(!line,!line));
"und"       => (Tokens.und(!line,!line));
"oder"      => (Tokens.oder(!line,!line));
"nicht"     => (Tokens.nicht(!line,!line));
"zwar"      => (Tokens.zwar(!line,!line));
"aber"      => (Tokens.aber(!line,!line));
"weder"     => (Tokens.weder(!line,!line));
"noch"      => (Tokens.noch(!line,!line));
"eines"     => (Tokens.eines(!line,!line));
("ein" | "eine" | "einen" | "einem")
           => (Tokens.EQ(!line,!line));
<INITIAL>"Einige" => (Tokens.EQpl(!line,!line));
"einige"    => (Tokens.EQpl(!line,!line));
("keinen" | "kein" | "Kein" | "Keine" | "keine" | "keinem")
           => (Tokens.NQ(!line,!line));
("Jeder" | "jeder" | "Jede" | "jede" | "Jeden" | "jeden")
           => (Tokens.UQ(!line,!line));
("Alle" | "alle")
           => (Tokens.UQpl(!line,!line));
"ist"       => (Tokens.ist(!line,!line));
"sind"      => (Tokens.sind(!line,!line));
"wird"      => (Tokens.wird(!line,!line));
"werden"    => (Tokens.werden(!line,!line));
"von"       => (Tokens.von(!line,!line));
"als"       => (Tokens.als(!line,!line));
("ihr" | "ihre" | "ihren" | "sein" | "seine" | "seinen")
           => (Poss());
"deren"     => (Tokens.deren(!line,!line));
"in"        => (P "in");
"auf"       => (P "auf");
"Anna"      => (PN "Anna");
"Annas"     => (PNgen "Anna");
"Berta"     => (PN "Berta");
"Clara"     => (PN "Clara");
"Dieter"    => (PN "Dieter");
```

```
"Emil"      => (PN "Emil");
"Grete"     => (PN "Grete");
"Gretes"    => (PNgen "Grete");
"Hans"      => (PN "Hans");
"Maria"     => (PN "Maria");
"Rom"       => (PN "Rom");
"Marseille"=> (PN "Marseille");
"Palermo"   => (PN "Palermo");
"Frankreich" => (PN "Frankreich");
"Italien"   => (PN "Italien");
"Spanien"   => (PN "Spanien");
"Sizilien" => (PN "Sizilien");
"Frau"      => (N1   "Frau");
"Frauen"    => (N1pl "Frau");
"Mann"      => (N1   "Mann");
"Männer"    => (N1pl "Mann");
"Spielsachen" => (N1pl "Spielsache");
"Ameise"    => (N1 "Ameise");
"Hund"      => (N1 "Hund");
"Elephant" => (N1 "Elephant");
"Tier"      => (N1 "Tier");
("Student" | "Studenten")
           => (N1 "Student");
"Stadt"     => (N1 "Stadt");
"Staat"     => (N1 "Staat");
"Insel"     => (N1 "Insel");
"Nachbar"   => (N2 "Nachbar");
"Geschwister" => (N2pl "Geschwister");
"Bruder"    =>
   (Tokens.N2(RestrL("Geschwister","Mann"),!line,!line));
"Brüder"    =>
   (Tokens.N2pl(RestrL("Geschwister","Mann"),!line,!line));
"Schwester"  =>
   (Tokens.N2(RestrL("Geschwister","Frau"),!line,!line));
"Schwestern" =>
   (Tokens.N2pl(RestrL("Geschwister","Frau"),!line,!line));
("gesund" | "gesunder" | "gesunde" | "gesundes")
           => (CA "gesund");
("weiß" | "weißer" | "weiße" | "weißes")
           => (CA "weiß");
("weiblich" | "weiblicher" | "weibliche" | "weibliches")
           => (CA "Frau");
("männlich" | "männlicher" | "männliche" | "männliches")
           => (CA "Mann");
"älter"     => (IA_comp "älter");
"größer"    => (IA_comp "größer");
```

```
"kleiner"  => (IA_comp "kleiner");
("alt" | "alter" | "alte" | "altes")
           => (IA "älter");
("groß" | "großer"| "große" | "großes")
           => (IA "größer");
("klein" | "kleiner" | "kleine" | "kleines")
           => (IA "kleiner");
"ähnlich"  => (RA "ähnlich");
"heiraten" => (SVpl "heiraten");
"arbeiten" => (IVpl "arbeiten");
"arbeitet" => (IV   "arbeiten");
"kennen"   => (TVpl "kennen");
"kennt"    => (TV   "kennen");
"lieben"   => (TVpl "lieben");
"liebt"    => (TV   "lieben");
"begrüßt"  => (TV   "begrüßen");
"begrüßen" => (TVpl "begrüßen");
"besitzt"  => (TV   "besitzen");
"geliebt"  => (TVpart "lieben");
.          => (error ("ignoring illegal character " ^ yytext,
                      !line,!line); lex());
```

Zur Beschreibung von ML-Lex und ML-Yacc siehe die Online-Dokumentation (via `smlnj.org`) oder Kapitel 2.5 und 3.4 aus [8].

Bemerkung 3.1 Beachte, daß hier *im Lexikon* die Relation `Bruder` auf die Relation `Geschwister` und das Prädikat `Mann` zurückgeführt wird; analog für `Schwester`. Dasselbe könnte man für `Eltern`, `Kind`, `Tochter`, `Sohn` usw. machen, d.h. im Lexikon gewisse Begriffe auf eine Kombination von anderen zurückführen. Dadurch wird eine Beziehung zwischen Begriffen schon im Lexikon für alle Interpretationen festgelegt. Das scheint vernünftiger, als die Gültigkeit der Beziehung in jedem Modell extensional zu sichern.

Es wäre nur konsequent, wenn man auch Eigenschaften wie die Symmetrie gewisser Verben, Relationsnomen und Adjektive im Lexikon sicherstellen würde, was bisher nicht geschieht. Wenn man es im Lexikon macht, gilt die Eigenschaft nicht nur für jedes Modell, sondern auch für alle Vorkommen in Ausdrücken. Wenn man es in der Grammatik macht, z.B. in den `ColVP`-Regeln, so gilt es zwar in allen Modellen, aber nur für bestimmte Verwendungsweisen. Wenn man es im Modell macht, so muß man zur Zeit der Konstruktion des Modells Hilfsfunktionen (wie `SymH`) definieren, die man durch die Grundfunktionen der Peirce-Algebra später leichter definieren kann.

Allerdings muß man nun neben dem Modell auch das Lexikon lesen, um zu wissen, welche Konstanten im Lexikon zusätzlich oder anders definiert werden – das ist auch nicht schön.

3.3 Beispiele

3.3.1 Klassifizierende und graduierbare Adjektive

Es wird zwischen klassifikatorischen und intensiven Adjektiven unterschieden.

Klassifikatorische Adjektive (CA) werden als Mengen interpretiert, die in der attributiven Verwendung beim Nomen zur Einschränkung, dem Durchschnitt `ProdS(A,N)`, führen; in der prädikativen Verwendung bildet `A` das einstellige Prädikat.

Intensive (oder *graduierbare*) Adjektive (IA) werden als Ordnungsrelationen interpretiert. In der prädikativen Verwendung ist `A` das zweistellige Prädikat.

Beispiel 3.2 (Prädikativer Gebrauch intensiver Adjektive) In der Interpretation `pg1.sml` erhalten wir:

⟨*Beispiel eines prädikativen, komparierten intensiven Adjektivs*⟩≡

```
- NL.evals "Anna ist älter als Emil";
val it = nu true : NL.Semantik.value
- NL.evals "Dieter ist älter als jeder Student";
val it = nu true : NL.Semantik.value
- NL.evals "Dieter ist älter als jeder Mann";
val it = nu false : NL.Semantik.value

- NL.evals "ist älter als jeder Student";
val it = un (Set [d]) : NL.Semantik.value
- NL.evals "ist älter als kein Student";
val it = un (Set [b,c,e,m]) : NL.Semantik.value
- NL.evals "ist nicht älter als jeder Student";
val it = un (Set [a,b,c,e,m]) : NL.Semantik.value
```

In der attributiven Verwendung im positiv, wie in *kleines Tier*, nehmen wir für $A\,N$ den Urbildbereich des „unteren Teils" von $A \cap N^2$, $\texttt{Minor}(A \cap N^2) : 1$.

Wie kann man den *unteren Teil* einer Relation definieren, oder zumindest den unteren Teil einer endlichen partiellen Ordnung? Ein Element a ist R-minimal, falls $a \in \mathit{dom}(R) - \mathit{range}(R) = R : 1 - 1 : R$, und R-maximal, falls $a \in 1 : R - R : 1$. Das Element a ist *R-klein*, falls es R-minimal ist, oder falls es klein bezüglich der Beschränkung von R auf $\mathit{range}(R) \times \mathit{dom}(R)$ ist. Oder, genauer gesagt, a ist R-klein, falls $a \in \mathit{minor}(R) : 1$ für

$$\mathit{minor}(R) := \begin{cases} \emptyset, & \text{falls } R = R \cdot (1 : R \times R : 1) \\ R \cdot (\overline{1 : R} \times 1 : R) & \\ +\mathit{minor}(R \cdot (1 : R \times R : 1)), & \text{sonst.} \end{cases}$$

Beachte, daß dies keine explizite, sondern nur eine rekursive Definition durch die anderen Peirce-Operationen ist. Die Rekursion terminiert für endliche, kreisfreie

R auch ohne den teuren Test im ersten Fall. Wir haben z.B.

$$\begin{aligned}
&minor(\{\langle a,b\rangle,\langle b,c\rangle,\langle a,c\rangle,\langle c,d\rangle\}) \\
&= \{\langle a,b\rangle,\langle a,c\rangle\} \cup minor(\{\langle b,c\rangle\}) \\
&= \{\langle a,b\rangle,\langle a,c\rangle\} \cup \{\langle b,c\rangle\} \cup minor(\emptyset) \\
&= \{\langle a,b\rangle,\langle a,c\rangle,\langle b,c\rangle\}
\end{aligned}$$

und dessen Urbildbereich $\{a,b\}$ ist vernünftigerweise der bezüglich der Relation kleine Teil des Bereichs $\{a,b,c,d\}$. Bei einem endlichen Baum R ist $minor(R)$ der Teil „unterhalb der Mittellinie“. Elemente auf einem R-Kreis sind nicht im unteren Teil von R.

Beispiel 3.3 (Attributiver Gebrauch intensiver Adjektive) Um den attributiven Gebrauch von intensiven Adjektiven im Positiv, wie in *großes Tier*, zu definieren, muß man zuerst die Ordnungsrelation A auf die Bezugsmenge N relativieren, d.h. zu $A \cap (N \times N)$ übergehen, und dann den „unteren Teil“ dieser Relation bestimmen; deren Urbildbereich ist die gesuchte Menge.

$\langle$*PA/pg3.sml*$\rangle\equiv$

```
structure Tiere =
   Peirce(datatype element = a1 | a2 | a3
                           | e1 | e2 | e3 | e4
                           | h1 | h2 | h3 | h4
          val elements = [a1,a2,a3,h1,h2,h3,h4,e1,e2,e3,e4]

          exception NoValue

          fun ConS "arbeiten" = [a1,a2]
            | ConS "gesund"   = [a1,a3,e2,e3,h1,h4]
            | ConS "Ameise"   = [a1,a2,a3]
            | ConS "Elephant" = [e1,e2,e3,e4]
            | ConS "Hund"     = [h1,h2,h3,h4]
            | ConS "weiß"     = [a1,a2,e1,e2,h1,h3,h4]
            | ConS "Tier"     = elements
            | ConS _ = raise NoValue

          fun ConR "größer"   =
              [(e4,e2),(e3,e2),(e2,e1),(e3,e1), (* partiell *)
               (h4,h3),(h4,h2),(h4,h1),(h3,h2),(h3,h1),(h2,h1),
               (a3,a2),(a3,a1),(a2,a1),         (* total *)
               (* transitive Huelle: *)
               (e4,h4),(e4,h3),(e4,h2),(e4,h2),(e4,h1),
               (e3,h4),(e3,h3),(e3,h2),(e3,h2),(e3,h1),
               (e2,h4),(e2,h3),(e2,h2),(e2,h2),(e2,h1),
               (e1,h4),(e1,h3),(e1,h2),(e1,h2),(e1,h1),
               (e1,a3),(e1,a2),(e1,a1),
```

```
                  (e4,a3),(e4,a2),(e4,a1),
                  (e3,a3),(e3,a2),(e3,a1),
                  (e2,a3),(e2,a2),(e2,a1),
                  (e1,a3),(e1,a2),(e1,a1),
                  (h4,a3),(h4,a2),(h4,a1),
                  (h3,a3),(h3,a2),(h3,a1),
                  (h2,a3),(h2,a2),(h2,a1),
                  (h1,a3),(h1,a2),(h1,a1)]
               | ConR "kleiner"  =
                 map (fn (x,y) => (y,x)) (ConR "größer")
               | ConR _ = raise NoValue
            )

structure NL : PARSE =
   Parse (structure Absyn = Absyn
          structure Algebra = Tiere
          structure Interface = Interface
          structure Parser = NL_Parser
          structure Tokens = NL_LrVals.Tokens)
```

Der zugehörige Makefile ist

⟨*PA/pg3.cm*⟩≡

```
Group
   structure NL
is
   pg3.sml
   pg.parse.cm
```

Die Ordnung der Tiere nach der Größe ist so gewählt, daß alle Elphanten größer als alle Hunde und diese größer als alle Ameisen sind. Innerhalb der Elephanten ist die Ordnung partiell.

Die Transitivität der Ordnung auf den Tieren ist wichtig für Aussagen wie:

⟨*Größenvergleich zwischen den Tierarten*⟩≡

```
- NL.evals "jeder Elephant ist größer als jeder Hund";
val it = nu true : Semantik.value
- NL.evals "jeder Ameise ist kleiner als jeder Hund";
val it = nu true : Semantik.value
```

Für den attributiven Gebrauch von intensiven Adjektiven im Positiv berechnen wir anscheinend richtig den Anteil der größeren bzw. kleineren Elemente einer definierbaren Menge von Tieren:[8]

[8]Die Definition bei Böttner kann das nicht: sie müßte zu *jeder definierbaren* Menge von Individuen und jeder Vergleichsrelation ein Standardelement der Menge festlegen, relativ zu dem die Anteile bestimmt werden. Das Standardelement kann aber nicht in der Auswertungsregel `R 31` auftreten, da diese für alle Vergleichsrelationen gelten muß. Außerdem existiert ein solches Standardelement nicht immer, z.B. bei den Elephanten nicht, bei den Ameisen schon.

⟨*Große Tiere*⟩≡
```
- NL.evals "großes Tier";
val it = un (Set [h3,h4,e1,e2,e4,e3]) : Semantik.value
- NL.evals "großer Elephant";
val it = un (Set [e4,e3]) : Semantik.value
- NL.evals "großer Hund";
val it = un (Set [h3,h4]) : Semantik.value
- NL.evals "große Ameise";
val it = un (Set [a3]) : Semantik.value

- NL.evals "kleines Tier";
val it = un (Set [h2,h1,a3,a2,a1]) : Semantik.value
- NL.evals "kleine Hund";
val it = un (Set [h2,h1]) : Semantik.value
- NL.evals "kleine Ameise";
val it = un (Set [a1]) : Semantik.value
```

Man kann die attributive Verwendung intensiver Adjektive iterieren:

⟨*Iteration attributiver intensiver Adjektive*⟩≡
```
- NL.evals "großes großes Tier";
val it = un (Set [e2,e4,e3]) : Semantik.value
- NL.evals "kleines großes Tier";
val it = un (Set [h4,h3]) : Semantik.value
- NL.evals "großes kleines großes Tier";
val it = un (Set [h4]) : Semantik.value
```

Es wird richtig berechnet, daß die großen Ameisen keine großen Tiere sind, sondern kleine:

⟨*Relativierung attributiver intensiver Adjektiven und Inklusion*⟩≡
```
- NL.evals "jede Ameise ist ein Tier";
val it = nu true : NL.Semantik.value
- NL.evals "jede kleine Ameise ist ein kleines Tier";
val it = nu true : NL.Semantik.value
- NL.evals "jede große Ameise ist ein großes Tier";
val it = nu false : NL.Semantik.value
- NL.evals "jeder große Elephant ist ein großes Tier";
val it = nu true : NL.Semantik.value
- NL.evals "jede große Ameise ist ein kleines Tier";
val it = nu true : NL.Semantik.value
- NL.evals "keine große Ameise ist eine kleine Ameise";
val it = nu true : NL.Semantik.value
```

Nach der Interpretation ist keine weiße Ameise groß, und kein weißer Elephant groß. In diesem Fall ist die Reihenfolge der Adjektive wesentlich:

⟨*Unvertauschbarkeit klassifikatorischer mit intensiven Adjektiven*⟩≡
```
- NL.evals "weiße Ameise";
```

```
val it = un (Set [a2,a1]) : NL.Semantik.value
- NL.evals "große Ameise";
val it = un (Set [a3]) : NL.Semantik.value

- NL.evals "weiße große Ameise";
val it = un (Set []) : NL.Semantik.value
- NL.evals "große weiße Ameise";
val it = un (Set [a2]) : NL.Semantik.value

- NL.evals "großer Elephant";
val it = un (Set [e4,e3]) : NL.Semantik.value
- NL.evals "weißer Elephant";
val it = un (Set [e2,e1]) : NL.Semantik.value

- NL.evals "großer weißer Elephant";
val it = un (Set [e2]) : NL.Semantik.value
- NL.evals "weißer großer Elephant";
val it = un (Set []) : NL.Semantik.value
```

Nach der Interpretation sind alle großen Hunde weiß, aber nicht alle weißen Hunde groß. Auch dann ist die Reihenfolge wesentlich:

⟨*Unvertauschbarkeit klassifikatorischer mit intensiven Adjektiven*⟩+≡

```
- NL.evals "großer Hund";
val it = un (Set [h3,h4]) : NL.Semantik.value
- NL.evals "weißer Hund";
val it = un (Set [h4,h3,h1]) : NL.Semantik.value

- NL.evals "weißer großer Hund";
val it = un (Set [h4,h3]) : NL.Semantik.value
- NL.evals "großer weißer Hund";
val it = un (Set [h4]) : NL.Semantik.value
```

Klassifikatorische Adjektive schränken die Bedeutung des Nomens ein:

⟨*Klassifikatorische Adjektive*⟩≡

```
- NL.evals "jede weiße Ameise ist eine Ameise";
val it = nu true : NL.Semantik.value
```

Kurioserweise ist das biologisch betrachtet falsch: „Weiße Ameisen" sind Termiten, und diese sind *keine* Ameisen.

Problem 3.4 Neben der oben behandelten *relativen Lesart* intensiver Adjektive, bei der die Ordnung A auf einen Bereich N relativiert wird, gibt es eine *funktionale* oder *dimensionale* Lesart: *Er ist gut als Vorgesetzter* im Unterschied zu *Er ist gut als Ehemann*. Hier entsteht die mit `A als N` gemeinte Vergleichsrelation nicht als Relativierung einer globalen Relation `A`.

Wie kann man das behandeln? Sind die `N` hier immer Relationsnomen, also `N2`? Handelt es sich um eine implizite *adverbiale* Modifizierung: *übt die Funktion/Tätigkeit N gut/schlecht aus*? Und wäre das dann vielleicht doch wieder eine Relativierung, allerdings die Relativierung allgemeiner Qualitätskriterien für Tätigkeiten auf spezielle Tätigkeiten?

Beispiel 3.5 (Klassifikatorische Adjektive bei Relationsnomen)

3.3.2 Aktiv und Passiv von transitiven Verben

Es gibt zwei verschiedene Auffassungen davon, was dem Passiv eines transitiven Verbs mit Bedeutung `R:rel` semantisch entspricht: (i) eine Vertauschung der Argumentstellen, d.h. $\texttt{Conv(R)} = \{\langle b, a\rangle \mid \langle a, b\rangle \in \texttt{R}\}$, (ii) die Projektion auf das zweite Argument von `R`, d.h. $\texttt{PreIm(Conv(R),UnitS)} = \{b \mid \exists a\ \langle a, b\rangle \in \texttt{R}\}$.

Wenn man (i) betrachtet, scheint offensichtlich, daß Aktiv- und Passivsatz dasselbe bedeuten. Aber das ist nur für *individuelle* Argument-Nominalphrasen plausibel, nicht für *quantifizierte*:

Beispiel 3.6 (Passiv mit quantifiziertem Subjekt/Objekt)

$\langle$*PA/pg4.sml*$\rangle\equiv$

```
structure PersonenPassiv =
   Peirce(datatype element = a | e | f | g | h
          val elements = [a,e,f,g,h]

          exception NoValue

          fun ConS "Anna"     = [a]
            | ConS "Emil"     = [e]
            | ConS "Fritz"    = [f]
            | ConS "Grete"    = [g]
            | ConS "Hans"     = [h]
            | ConS "Frau"     = [a,g]
            | ConS "Mann"     = [e,f,h]
            | ConS _ = raise NoValue

          fun ConR "lieben"   = [(e,g),(h,g),(a,e),(a,h),(g,e)]
            | ConR _ = raise NoValue
         )

structure NL : PARSE =
   Parse (structure Absyn = Absyn
          structure Algebra = PersonenPassiv
          structure Interface = Interface
          structure Parser = NL_Parser
```

```
            structure Tokens = NL_LrVals.Tokens)
```

Der Makefile für das Beispiel ist wie immer:

⟨*PA/pg4.cm*⟩≡

```
Group
   structure NL
is
   pg.parse.cm
   pg4.sml
```

Man erhält bei dieser Interpretation:

⟨*Aktiv und Passivsatz mit verschiedener Bedeutung*⟩≡

```
- NL.evals "eine Frau wird von keinen Mann geliebt";
val it = nu true : NL.Semantik.value
- NL.evals "kein Mann liebt eine Frau";
val it = nu false : NL.Semantik.value

- NL.evals "liebt eine Frau";
val it = un (Set [e,h]) : NL.Semantik.value
- NL.evals "wird von keinem Mann geliebt";
val it = un (Set [a,e,h]) : NL.Semantik.value
- NL.evals "wird von einem Mann geliebt";
val it = un (Set [g]) : NL.Semantik.value
- NL.evals "Anna wird von keinem Mann geliebt";
val it = nu true : NL.Semantik.value
```

Das Problem kommt natürlich daher, daß die Quantoren in verschiedener Reihenfolge aus dem Satz extrahiert werden:

$$\begin{array}{lcl} \exists F\ \breve{L}\ \overline{\exists} M & \mapsto & \exists x \in F\, \forall y \in M \neg\, \breve{L}(x,y) \\ \overline{\exists} M\ L\ \exists F & \mapsto & \forall y \in M \neg\, \exists x \in F\ L(y,x) \end{array}$$

Aber ist das Argument überzeugend? Wie die Wirkungsbereiche der Quantoren gemeint sind, scheint stark von der Betonung abzuhängen, und die wechselt wohl zwischen den Beispielen.

3.3.3 Koordination

Bei Eigennamen ist die Koordination noch einigermaßen verständlich: Konjunktion von Eigennamen wird wie ein ∀-Quantor bei Nomen interpretiert, Disjunktion wie ein ∃-Quantor beim Nomen. Dazu werden die Bedeutungen der Eigennamen in beiden Fällen durch Mengenvereinigung interpretiert (`R 51` und `R 52`) und erst auf der Satzebene die Quantorenlesart verwendet (`R 53` und `R 54`).

Beispiel 3.7 (Koordination von PNs im Subjekt und Objekt)

⟨*Koordination von Eigennamen*⟩≡

```
- CM.make "pg4.cs";
- NL.parse();
Emil oder Hans liebt Grete.
Value is: E(((Emil + Hans) . (lieben : Grete)))
Ein Mann liebt Grete.
Value is: E((Mann . (lieben : Grete)))

Emil und Hans lieben Grete.
Value is: U(( -(Emil + Hans) + (lieben : Grete)))
Jeder Mann liebt Grete.
Value is: U(( -Mann + (lieben : Grete)))

Grete liebt Emil und Hans.
Value is: E((Grete .  -(-lieben : (Emil + Hans))))
Grete liebt jeden Mann.
Value is: E((Grete .  -(-lieben : Mann)))

Grete liebt Emil oder Hans.
Value is: E((Grete . (lieben : (Emil + Hans))))
Grete liebt einen Mann.
Value is: E((Grete . (lieben : Mann)))
```

Das geht aber nicht mehr für kompliziertere Koordinationen, wie *Hans und Emil oder Grete*, wie Böttner bemerkt. Daher ist der ganze Ansatz nur sehr beschränkt tauglich.

Gibt es eine brauchbare Alternative? Man kann, wie in den Kategoriennamen `aPNP` und `cPNP` der disjunktiven und konjunktiven Eigennamenphrasen, die syntaktische Verbindung im Kategorienamen „speichern“ und in der Satzregel die dazu passende Auswertungsfunktion wählen. Um kompliziertere NP-Koordinierungen zu erfassen, braucht man *Listen von Mengen* $[A, B, \ldots]$.[9] Das geht über den Rahmen der Peirce-Algebra zwar hinaus, bleibt aber erststufig, mit Strukturen $(Set, Relation, Set^*, Relation^*; \ldots)$.

Dazu muß man in `pg.grm` lediglich entsprechende Kategorien und Werte vorsehen. Für jede boole'sche Kombination *bc* der durch Quantoren *qc* quantifizierten Nominalphrasen braucht man eine eigene Kategorie *bcNPqc* der entsprechenden Stelligkeit.

Beispiel 3.8 Wir betrachten die koordinierte Nominalphrase *jeder Mann, keine Frau, und ein Kind*. Die Kategorie `cNPune` soll für eine Konjunktion einer universell, einer negiert und eine existentiell quantifizierten `NP` im Singular stehen, deren Wert das Tripel der Nomenbedeutungen ist, hier also `(Mann,Frau,Kind)`.

⟨*Neue Kategorien mit Tupel von Mengen*⟩≡

[9] Entsprechend *Listen von Relationen* $[V, T, \ldots]$ für die Koordinierung von Verben.

```
% nonterm
   cNPune of D.set * D.set * D.set

cNPune : UQ NP Komma NQ NP und EQ NP  ((NP1,NP2,NP3))
```

Für Vorkommen von `cNPune` in der Subjektposition muß als Bedeutung die Kombinationen der Bedeutungen entsprechender Aussagen angegeben werden, für Vorkommen in der Objektposition die entsprechender `TVP`s:

⟨*Auswertung mit koordinieren NPs*⟩≡

```
S : cNPune VP    (And(U(SumS(COmpS(#1 cNPune), VP)),
                      (And(N(ProdS(#2 cNPune, VP)),
                          E(ProdS(#3 cNPune, VP))))))

TVP : TV cNPune  (ProdS(UnPreIm(TV,#1 cNPune),
                        ProdS(CompS(PreIm(TV,#2 cNPune)),
                              PreIm(TV,#3 cNPune))))
```

Hierbei wird mit `#i` die i-te Komponente aus dem Mengentripel herausgeholt.

⟨*Beispiel mit koordiniertem Subjekt, pg4.cm*⟩≡

```
- NL.parse();
jede Frau, kein Mann und eine Frau liebt Emil.
Value is: (U(( -Frau + (lieben : Emil)))
           /\ (N((Mann . (lieben : Emil)))
               /\ E((Frau . (lieben : Emil)))))

- NL.evals "jede Frau, kein Mann und eine Frau liebt Emil";
val it = nu true : NL.Semantik.value

Emil liebt jede Frau, keinen Mann und eine Frau.
Value is: E((Emil . ( -(-lieben : Frau)
                     . ( -(lieben : Mann) . (lieben : Frau)))))
```

Auch für zweistellige Koordinationen wie *zwar Hans, aber nicht Emil* oder *jeder Mann, aber keine Frau* kann man keine Mengenbedeutung angeben.

⟨*Versuch einer Alternative*⟩≡

```
NPun :
     UQ N1 aber NQ N1     (Cart(N11,N12))      (* Neu *)
     | PN aber nicht PN   (Cart(PN1,PN2))

S : NPun VP               (U(- NPun:1 + VP) andalso
                           not(E( 1:NPun . VP)))
```

Hier käme man mit den normalen Peirce-Operationen aus, indem man die Mengen A, B in der Relation $A \times B$ kodiert und dann auswertet wie in

$$S(A \subseteq V \wedge B \subseteq \overline{V}) \to NP(A \times B)\ \ VP(V).$$

Eine solche Übersetzung behandelt jede NP-Kombination separat, was zu einer unendlichen Grammatik führt.

In Abschnitt 4 geben wir eine Alternative hierzu an, bei der NP-Koordinierungen in Boole'sche Kombinationen von Sätzen oder Verbalphrasen übersetzt werden.

3.3.4 Plural

Bei Nominalphrasen im Plural sollte semantisch zumindest beim unbestimmten Artikel *einige* bzw. dem Existenzquantor im Plural, `EQpl`, quasi die Bedeutung

$$(EQpl\ N) = \begin{cases} \emptyset, & \text{falls } |N| \leq 2 \\ N, & \text{sonst} \end{cases}$$

herauskommen, während man für *alle* und *keine* vermutlich keinen Unterschied zum Singular erwartet. Böttner argumentiert, man solle die Auswirkung des Plurals nicht schon bei der Nominalphrase, sondern erst bei der Auswertung des Satzes einbauen, da der Plural sich auch am Verb zeigt. Das geht in der Subjektsposition nach Böttners Vorschlag (4.67) mit

$$\begin{aligned} NP \cdot_{pl} VP &:= ((NP \cdot VP) \times (NP \cdot VP) - I) : 1 \\ &= \begin{cases} \emptyset, & \text{falls } |NP \cdot VP| \leq 1, \\ NP \cdot VP, & \text{sonst,} \end{cases} \end{aligned}$$

auch wenn eine entsprechende Auswertungsregel nicht angegeben ist. Böttners Argument ist aber nicht überzeugend, da der Plural auch unabhängig von der Verbmorphologie auftritt, nämlich bei Nominalphrasen in Objektposition, die Böttner nicht behandelt.

Wir wollen *einige* sowohl in Subjekt- wie in Objektposition durch *mindestens zwei* interpretieren. Nominalphrasen `(EQpl NPpl)` interpretieren wir mit Hilfe der Verschiedenheitsrelation `Div(NP)`.

Bei der Objektposition, für `TVP : TV EQpl NPpl`, benutzen wir in `(R 20b)`

$$2 \leq |\{y \in N \mid x\ R\ y\}| \iff x\ (R\,; Div_N\,; \breve{R})\ x.$$

Bei der Subjektposition, für `S : EQpl NPpl VPpl`, benutzen wir in `(R 14b)`

$$2 \leq |\{x \in N \mid R\,x\}| \iff (Id_R\,; Div_N\,; \breve{Id_R}) \neq \emptyset,$$

wobei man dieselbe Abkürzung $PreImDiv(r, s) := r\,; Div_s\,; \breve{r}$ benutzen kann.

Beispiel 3.9 Die Semantik von Aussagen hängt bei existentiell quantifiziertem Subjekt davon ab, ob das Subjekt im Singular oder Plural steht:

⟨*EQpl im Subjekt, mit pg4.cm*⟩≡

```
  - NL.parse();
  eine Frau liebt Hans.
```

```
Value is: E((Frau . (lieben : Hans)))
einige Frauen lieben Hans.
Value is: ER((( Id((lieben : Hans))
              ; (((Frau x 1) . ~(Frau x 1)) . -Id(1)))
              ; ~Id((lieben : Hans))))

- NL.evals "liebt Hans";
val it = un (Set [a]) : NL.Semantik.value
- NL.evals "eine Frau liebt Hans";
val it = nu true : NL.Semantik.value
- NL.evals "einige Frauen lieben Hans";
val it = nu false : NL.Semantik.value

- NL.evals "liebt Emil";
val it = un (Set [a,g]) : NL.Semantik.value
- NL.evals "einige Frauen lieben Emil";
val it = nu true : NL.Semantik.value
```

Analog hängt die Bedeutung bei existentiell quantifiziertem Objekt vom Numerus des Objekts ab:

⟨*EQpl Npl in Objektposition*⟩≡

```
Hans liebt eine Frau.
Value is: E((Hans . (lieben : Frau)))
Hans liebt einige Frauen.
Value is: E((Hans . (((( lieben
                       ; (((Frau x 1) . ~(Frau x 1)) . -Id(1)))
                       ; ~lieben) . Id(1)) : 1)))

-  NL.evals "Hans liebt eine Frau";
val it = nu true : NL.Semantik.value
-  NL.evals "Hans liebt einige Frauen";
val it = nu false : NL.Semantik.value

Anna liebt einige Männer.
Value is: E((Anna . ((((lieben
                      ; (((Mann x 1) . ~(Mann x 1)) . -Id(1)))
                      ; ~lieben) . Id(1)) : 1)))
Anna liebt alle Männer.
Value is: E((Anna .  -(-lieben : Mann)))

- NL.evals "Anna liebt einige Männer";
val it = nu true : NL.Semantik.value
- NL.evals "Anna liebt alle Männer";
val it = nu false : NL.Semantik.value
```

Damit haben wir einen Unterschied zwischen *ein*, *einige* und *alle*. Wie man den behaupteten Unterschied zwischen *jeder* und *alle* in *Jeder Minister kam in einem Auto* und *Alle Minister kamen in einem Auto* modellieren könnte, bleibt unklar.

Bemerkung 3.10 Man beachte, daß *keine* und *einige nicht* verschieden sind: *Grete liebt keine Männer* und *Grete liebt einige Männer nicht*. Das ist aber im Singular genauso, sodaß die Regeln mit *nicht* wohl nur für nicht-quantifizierte Subjekte und Objekte gelten.

3.3.5 Kollektive VPs

Ein besonderer Fall, wo der Plural eine Auswirkung auf die Bedeutung hat, sind die symmetrischen Prädikate. Hierunter fallen (i) symmetrische Verben wie *heiraten*, (ii) symmetrische Relationsnomen wie *Geschwister* und (iii) symmetrische Adjektive wie *ähnlich*, denen Eigenschaften von Zweiermengen (ungeordneten Paaren) entsprechen.

Das Subjekt muß jeweils im Plural stehen; es kann u.a. durch Konjunktion von Eigennamen oder aus einem Allgemeinnamen im Plural gebildet sein.

Beispiel 3.11 Zur einfacheren Angabe einer symmetrischen Relation wird eine Hilfsfunktion `SymH` zur Berechnung der symmetrischen Hülle einer Relation definiert.

$\langle$*PA/pg5.sml*$\rangle\equiv$

```
structure Kollektive =
   Peirce(datatype element = a | b | c | d | e | f | g | h | m | s | t
          val elements = [a,b,c,d,e,f,g,h,m,s,t]

          exception NoValue

          fun ConS "Anna"      = [a]
            | ConS "Berta"     = [b]
            | ConS "Clara"     = [c]
            | ConS "Maria"     = [m]
            | ConS "Dieter"    = [d]
            | ConS "Emil"      = [e]
            | ConS "Grete"     = [g]
            | ConS "Hans"      = [h]
            | ConS "Frau"      = [a,b,c,g,m]
            | ConS "Mann"      = [d,e,h]
            | ConS "Spielsache"  = [s,t]
            | ConS _ = raise NoValue

          fun ConR "kennen"    = [(a,b),(b,c)]
            | ConR "lieben"    = [(b,d),(c,e),(b,a),(a,a),(e,a),(h,g),(h,s)]
```

```
        | ConR "Geschwister"
          = SymH [(a,e),(a,d),(a,m),(d,e),(d,m),(e,m),
                  (g,b),(g,c),(b,c)]
        | ConR "begrüßen"
          = [(a,g),(a,b),(a,c),(a,m),(d,g),(d,b),(d,c)]
        | ConR "ähnlich"  = SymH [(d,e),(g,a)]
        | ConR "heiraten" = SymH [(g,h)]
        | ConR "besitzen" = [(h,s),(h,t)]
        | ConR _ = raise NoValue
      and SymH r = r @ (map (fn (x,y) => (y,x)) r)
    )
```

Mit `SymH` wird die symmetrische Hülle einer Relation berechnet, sodaß es genügt, kleinere Kernrelationen anzugeben. (Durch `ConR` wird aus der Liste eine Menge.)

⟨*PA/pg5.sml*⟩+≡

```
structure NL : PARSE =
   Parse (structure Absyn = Absyn
          structure Algebra = Kollektive
          structure Interface = Interface
          structure Parser = NL_Parser
          structure Tokens = NL_LrVals.Tokens)
```

Der passende Makefile dazu:

⟨*PA/pg5.cm*⟩≡

```
Group
   structure NL
is
   pg.parse.cm
   pg5.sml
```

Aus dem symmetrischen Relationsnomen *Geschwister* lassen sich die unsymmetrischen Relationsnomen *Bruder* und *Schwester* ableiten: $B = G \cap (M \times 1)$ und $S = G \cap (F \times 1)$, mit M, F für *Mann* und *Frau*. Das wird im Lexikon `pg.lex` gemacht. Eigenschaften wie die Symmetrie von Nomina und Verben kann man schon im Lexikon `pg.lex` erzwingen, indem man als deren Bedeutung die symmetrische Hülle der Bedeutung eines Pseudoworts angibt (vgl. Bemerkung 3.1).

Im Plural wird `Brüder` in Verbindung mit der Kopula `sind` durch den *symmetrischen Teil*, $R \cap \breve{R}$, der `Bruder`-Relation R interpretiert. Bei anderen Vorkommen, wie in *Anna und ihre Brüder*, muß die unsymmetrische Relation R verwendet werden. Der Übergang zum symmetrischen Kern kann daher erst in den Grammatikregeln, und nicht schon im Lexikon, vorgenommen werden.

Man sieht, daß die Reihenfolge der Argumente wegen `SymH` nicht so zu sein braucht wie bei der Angabe der Interpretation:

⟨*Beispiel mit symmetrischen Prädikaten*⟩≡

```
- CM.make "pg5.cm";
```

```
- NL.evals "Emil und Dieter sind Brüder";
val it = nu true : NL.Semantik.value
- NL.evals "Emil und Dieter sind ähnlich";
val it = nu true : NL.Semantik.value
- NL.evals "Hans und Grete heiraten";
val it = nu true : NL.Semantik.value

Hans und Grete heiraten.
Value is: ER((((((Hans + Grete) x 1) .
                ~((Hans + Grete) x 1)) . -Id( 1 )) . heiraten))
Hans und Grete arbeiten.
Value is: U(( -(Hans + Grete) + arbeiten))
```

Die Kategorie `PPNP` der Paareigennamenphrase von Böttner wurde hier aufgegeben, da sie zu ambigen Lesart von `PN`-Konjuktionen führt: `(Hans und Grete)` hat als `cPNP` den Wert $\{h, g\}$, aber als `PPNP` den Wert $\{\langle h, g\rangle\}$. Da ML-Yacc keine ambigen Ausdrücke behandeln kann, geben wir statt `PPNP` die Bedeutung `cPNP x cPNP - I` an.

Das führt bei Regel 63/64 zum gleichen Ergebnis, sollte aber auch bei Gemeinnamen funktionieren: `|alle N STV|` $= N \times N \setminus I \subseteq STV \cup S\check{T}V$, sofern die symmetrische Bedeutung gemeint ist. Diese Lesart ist aber manchmal die falsche:

$\langle$*Beispiel mit unsymmetrischen Prädikaten*$\rangle\equiv$

```
alle Männer heiraten.
Value is: UR((-(((Mann x 1) . ~(Mann x 1)) . -Id( 1 )) + heiraten))
```

Natürlich heiraten nicht alle Männer *einander*, sondern verschiedene Frauen. Man müßte also entscheiden, ob man `(alle Npl Vpl)` bei symmetrischem Verb durch `(je zwei N Vpl einander)` interpretieren muß oder durch `(jedes N V jemanden)`, und entsprechend im Singular mit koordiniertem Subjekt.

3.3.6 Lücken (Gapping)

Die Regeln 59-61 funktionieren nicht. In Regel 59 wird für Sätze wie *Hans liebt Grete und Fritz Maria* die Bedeutung von *Hans liebt Grete* mit der von *Fritz Maria*, was eine Einerrelation $\{\langle f, m\rangle\}$ sein soll, vereinigt.[10] Der zweite Teilsatz hat dann gar nichts mit der Bedeutung des Verbs zu tun, und das ist natürlich unmöglich – mal abgesehen davon, daß man von der Relation noch zu einem Wahrheitswert übergehen muß.

3.3.7 Possessive Pronomen und Nominalphrasen

Vermutlich ist in

[10] Regel 60 macht Sätze der Form `PN TV PN` ambig, da nach Regel 1 und Regel 7 auch `PN VP` ein Satz und `TV PN` eine `VP` ist. Die Bedeutung ist aber bei Regel 60 eine Relation, bei Regel 1 eine Menge (Wahrheitswert).

⟨*Regel 84 bei Böttner*⟩≡
```
NP : PNgen N2           (Im(N2,PNgen))          (* R 84  *)
```

mit `NP` eine spezielle `NPposs` gemeint, die aber sonst nicht vorkommt. Denn `R 84` führt dazu, daß vor `PNgen N2` wie vor anderen `NP` noch ein Quantor stehen kann, was zu ungrammatischen Ausdrücken wie *jeder Annas Bruder* führt. Deshalb haben wir statt der NP-Regel 84 eine TVP-Regel 84a der Form `TV PNgen RN` angenommen, mit der Bedeutung `TV't alle im RN-Bild von PNgen`.

Außerdem muß man aufpassen, wie die Relationsnomen zu lesen sind: Böttner gibt für *Annas Schwestern* das Bild von $\{a\}$ unter der Relation S an, $S''\{a\}$ oder `Im(N2,PN)` in Regel 84, aber es sollte das Urbild $S : \{a\}$ oder `PreIm(N2,PN)` sein, da er an anderen Stellen $a\ S\ d$ für *Anna ist eine Schwester von Dieter* schreibt, nicht $d\ S\ a$.

Beispiel 3.12 Wir nehmen `deren` für das irreflexive und `ihre` für das reflexive Possesivpronomen.

⟨*Possesivpronomen beim Relationsnomen*⟩≡
```
- CM.make "pg5.cm";
- NL.parse();
Anna begrüßt Grete und deren Schwestern.
E((Anna .
  -(-begrüßen : (Grete + ((Geschwister . (Frau x 1)) : Grete)))))

Anna begrüßt Grete und ihre Schwestern.
E((Anna . ((begrüßen : Grete) .
           -((-begrüßen . ~(Geschwister . (Frau x 1))) :  1 ))))

- NL.evals "begrüßt Grete und deren Schwestern";
val it = un (Set [a,d]) : NL.Semantik.value
- NL.evals "begrüßt Grete und ihre Schwestern";
val it = un (Set [a]) : NL.Semantik.value
```

Damit das geht, mußte die Bedeutung bei Regel 87 geändert werden.

Beispiel 3.13 Bei Relationsnomen war die Interpretation des (reflexiven) Possesivpronomens nach Regel 84a:

⟨*Possesivpronomen beim Relationsnomen*⟩+≡
```
begrüßt seine Schwestern.
Value is:  -((-begrüßen . ~(Geschwister . (Frau x 1))) :  1 )
```

Beim absoluten Nomen ist die Bedeutung des Possesivpronomens nach Auswertungsregel 89 etwas anders:

⟨*Possessivpronomen beim absoluten Nomen*⟩≡
```
- CM.make "pg5.cm";
```

```
- NL.parse();
besitzt seine Spielsachen.
Value is:  -(((besitzen . ~(Spielsache x 1)) . -besitzen) :  1 )
liebt seine Spielsachen.
Value is:  -(((besitzen . ~(Spielsache x 1)) . -lieben) :  1 )

- NL.evals "besitzt seine Spielsachen";
val it = un (Set [a,b,c,d,e,f,g,h,m,s,t]) : NL.Semantik.value
- NL.evals "liebt seine Spielsachen";
val it = un (Set [a,b,c,d,e,f,g,m,s,t]) : NL.Semantik.value
```

Da in diesem Modell Hans nur eine seiner Spielsachen liebt, ist das korrekt.

4 Modifizierte Peirce-Grammatik

Da die Behandlung der Koordination von Nominalphrasen in der Grammatik aus Böttner [2] sehr eingeschränkt ist, wollen wir hier eine Änderung versuchen: wir lassen komplexe Nominalphrasen zu, deren Werte eine „freie“ Kombination atomarer Bestandteile (Nomen und Eigennamen) sind, die aber bei der Auswertung von Sätzen in Boole'sche Kombinationen von Aussagen oder Verbalphrasen übersetzt werden.

Alternative Behandlung komplexer Nominalphrasen (Skizze)

Das Ziel ist nun also eine rekursive Übersetzung der NP-Koordination in die Kombination entsprechender Sätze oder Verbalphrasen.

Dazu muß man die Art der Koordination und Quantifizierung einer komplexen Nominalphrase in der abstrakten Syntax durch neue Operatoren repräsentieren. Diese Operatoren werden in den Auswertungsregeln für Aussagen durch eine Übersetzung in Ausdrücke der reinen Peirce-Algebra beseitigt; man braucht deshalb also nicht unbedingt die Struktur `Semantik` zu erweitern.

⟨*Versuch einer rekursiven Behandlung koordinierter NPs*⟩≡

```
% nonterm
   cNP of D.set comb
```

Die Regeln sollten verschiedene Konstruktionen erlauben, z.B. für den Singular die folgenden, mit einer Übersetzung für die Vorkommen in der Subjektposition:

⟨*Versuch einer rekursiven Behandlung koordinierter NPs*⟩+≡

```
cNP : PN                  (Pn(PN))
    | EQ NP               (Eq(NP))
    | NQ NP               (Nq(NP))
    | UQ NP               (Uq(NP))
    | cNP und cNP         (Und(cNP1,cNP2))
    | cNP oder cNP        (Oder(cNP1,cNP2))
    | weder cNP noch cNP  (Und(Nicht(cNP1),Nicht(cNP2)))
    | zwar cNP aber cNP   (Und(cNP1,cNP2))
    | nicht cNP           (Nicht(cNP))

S : cNP VP                (distrib(cNP,VP))

fun distrib(NP,VP) = case NP of
    Pn(N)  => E(ProdS(N,VP))       (* R 1 *)
  | Eq(N)  => E(ProdS(N,VP))       (* R 14 *)
  | Nq(N)  => N(ProdS(N,VP))       (* R 15 *)
  | Uq(N)  => U(SumS(CompS(N),VP)) (* R 13 *)
  | (Und(cNP1,cNP2))  => And(distrib(cNP1,VP),distrib(cNP2,VP))
  | (Oder(cNP1,cNP2)) => Or(distrib(cNP1,VP),distrib(cNP2,VP))
```

```
  | (Nicht(cNP)) => Neg(distrib(cNP,VP))
```

Bei den `cNP` dienen `Pn,Eq,Nq,Uq,Nicht,Und,Oder` zur Kodierung der Struktur der Nominalphrase. Allerdings müssen `cNP`s formal einen semantischen Wert haben, da auch die in der `cNP` vorkommenden Bedeutungen atomarer `NP` vom Typ `set` eine Rolle bei der Übersetzung spielen. Man muß also in der Semantik neben `set` und `rel` noch einen Bereich `comb` von Pseudo-Werten zulassen, die eine Kombination durch Boole'sche Operationen und Quantoren repräsentieren:

⟨*Typ comb*⟩≡

```
datatype 'a comb = Pn of 'a | Eq of 'a | Nq of 'a | Uq of 'a
                 | Und of 'a * 'a | Oder of 'a * 'a | Nicht of 'a
```

Für komplexe Nominalphrasen und Boole'sche Kombinationen von intransitiven Verben könnte man dann `type civ = set comb` nehmen, für Boole'sche Kombinationen von transitiven Verben `type ctv = rel comb` nehmen.

Man braucht jedenfalls eine *Erweiterung der Peirce-Algebra um formale Kombinationen von Mengen und Relationen.*

Problem 4.1 Man muß aber die Kombination der NP- und der Verb-Koordination richtig behandeln, z.B. *jeder Mensch spielt oder arbeitet* nicht in *jeder Mensch spielt oder jeder Mensch arbeitet* übersetzen. Heißt das, daß man zuerst die NP-Quantoren so weit vorziehen muß, daß alle Verbkombinationen zu einer Ausdistribuierung mit Individuenargumenten führen?

4.1 Erweiterung der abstrakten Syntax

Dazu erweitern wir die abstrakte Syntax der Peirce-Terme um Kombinationen von Mengentermen und eine Funktion zum Drucken der neuen Ausdrücke:

⟨*PA/mabsyn.sml*⟩≡

```
signature ABSYN =
   sig
      include SYN
      datatype 'a comb = Pn of 'a | PnPl of 'a
                       | Eq of 'a  | EqPl of 'a
                       | Nq of 'a
                       | Uq of 'a
                       | Nicht of 'a comb
                       | Und of 'a comb * 'a comb
                       | Oder of 'a comb * 'a comb

      datatype absyn = nullary of tv | unary of set | binary of rel
                     | combi of set comb

      val CtoString : set comb -> string
```

```
    end
```

⟨*PA/mabsyn.sml*⟩+≡

```
  structure Absyn : ABSYN =
    struct
       open PeirceTerm
       datatype 'a comb = Pn of 'a | PnPl of 'a
                        | Eq of 'a  | EqPl of 'a
                        | Nq of 'a
                        | Uq of 'a
                        | Nicht of 'a comb
                        | Und of 'a comb * 'a comb
                        | Oder of 'a comb * 'a comb

       datatype absyn = nullary of tv | unary of set | binary of rel
                      | combi of set comb

       fun CtoString ((Pn s) | (PnPl s)) = StoString s
         | CtoString (Eq s)   = "E(" ^ (StoString s) ^ ")"
         | CtoString (EqPl s) = "ER(" ^ (StoString s) ^ ")"
         | CtoString (Nq s) = "N(" ^ (StoString s) ^ ")"
         | CtoString (Uq s) = "U(" ^ (StoString s) ^ ")"
         | CtoString (Nicht s) = "-(" ^ (CtoString s) ^ ")"
         | CtoString (Oder(s1,s2)) =
           "(" ^ (CtoString s1) ^ " + " ^ (CtoString s2) ^ ")"
         | CtoString (Und(s1,s2)) =
           "(" ^ (CtoString s1) ^ " . " ^ (CtoString s2) ^ ")"
    end
```

4.2 Erweiterung des Parsers

Die Funktionen `ev` und `parse`, die die Eingabe in einen Peirce-Term umwandeln und diesen in lesbarer Form als Zeichenreihe ausgeben, sollen jetzt auch die Kombinationsterme erfassen. Wir verzichten aber darauf, die Auswertung `eval` zu ändern, sodaß Kombinationsausdrücke nicht separat ausgewertet werden können; innerhalb von Sätzen sind sie aber erlaubt, da sie bei der Auswertung von Sätzen zuerst in eine Kombination von Aussagen oder Verbalphrasen übersetzt werden.[11]

Obwohl sich nur die Hilfsfunktion `loop` in `Parser.parse` ändert, geben wir die unveränderte Signatur in der geänderten Datei `mpg.parse.sml` nochmal an:

⟨*PA/mpg.parse.sml*⟩≡

[11]Die Werte der komplexen Nominalphrasen bilden hier –anders als bei Keenan und Faltz– keine sinnvolle Boole'sche Algebra: *weder Emil noch Grete* ist unserer Auffassung nach keineswegs das Komplement von $\{Fritz, Grete\}$, sondern ein bloß formaler „Zwischenwert“.

```
signature PARSE =
  sig
      structure Absyn : ABSYN
      val parse : unit -> (NL_Parser.svalue,NL_Parser.pos)
                              LrParser.Token.token
      exception Undefined
      val ev : string -> Absyn.absyn

      structure Semantik : SEM
      val eval : Absyn.absyn -> Semantik.value

      val evals : string -> Semantik.value
  end
```

Die Änderung betrifft nur die Einfügung des Falls der Kombinationsterme in der Hilfsfunktion `loop` in `parse()`:

⟨*PA/mpg.parse.sml*⟩+≡

```
functor Parse (structure Absyn : ABSYN
               structure Algebra : PEIRCEfull
               structure Interface : INTERFACE
               structure Parser : PARSER
                  sharing type Parser.arg = Interface.arg
                  sharing type Parser.pos = Interface.pos
                  sharing type Parser.result = Absyn.absyn
               structure Tokens : NL__TOKENS
                  sharing type Tokens.token = Parser.Token.token
                  sharing type Tokens.svalue = Parser.svalue
               ) : PARSE =
  struct
    structure Absyn = Absyn

    exception Undefined

    fun parse () =
        let
            val dummyEOF = NL_LrVals.Tokens.EOF(0,0)

            fun invoke lexer =
                let fun error (s,i:int,_) =
                        print("Error, line " ^ (Int.toString i) ^
                              ", " ^ s ^ "\n")
                in
                    NL_Parser.parse(0,lexer,error,())
                end
```

```
        fun loop lexer =
            let val (result,lexer) = invoke lexer
                val (nextToken,lexer) = NL_Parser.Stream.get lexer
                (* Fehler bei Absyn statt D:
                   rule domain: Absyn.absyn =/= case object: NL_Parser.result *)
                val _ = case result of
                            (D.unary s) =>
                          print("Value is: " ^ (D.StoString s) ^ "\n")
                          | (D.binary s) =>
                          print("Value is: " ^ (D.RtoString s) ^ "\n")
                          | (D.nullary s) =>
                          print("Value is: " ^ (D.TtoString s) ^ "\n")
                          | (D.combi s) =>
                          print("Value is: " ^ (D.CtoString s) ^ "\n")
            in if NL_Parser.sameToken(nextToken,dummyEOF)
               then nextToken else loop lexer
            end

        fun reader (n:int) =
            case (TextIO.inputLine TextIO.stdIn) of
                SOME s => s
              | NONE => raise Undefined
        (* Für SML 110.0.3 stattdessen:
           fun reader (n:int) = TextIO.inputLine TextIO.stdIn *)
    in
      loop (NL_Parser.makeLexer reader)
    end

fun ev (s:string) =
    let
        val lookahead = 5
        val _ = Interface.init_line()
        val empty = !Interface.line

        val dummyEOF = Tokens.EOF(empty,empty)

        fun invoke lexer =
            Parser.parse(lookahead,
                         lexer,
                         Interface.error,
                         Interface.nothing)
        fun loop lexer =
            let val (result,lexer) = invoke lexer
                val (nextToken,lexer) = Parser.Stream.get lexer
            in if Parser.sameToken(nextToken,dummyEOF)
               then result else loop lexer
```

```
                end

            fun string_reader s =
                let
                    val next = ref s
                in fn _ => !next before next := "" end
            val reader = string_reader s
        in
            loop (Parser.makeLexer reader)
        end

    structure Semantik = Semantik(structure A = Absyn:ABSYN
                                  structure P = Algebra:PEIRCEfull)
    datatype value = datatype Semantik.value
    val eval = Semantik.eval

    fun evals s = eval(ev s)
  end
```

Makefile

Interpretationen der modifizierten Grammatik brauchen ein anderes `Absyn:ABSYN`, die andere Grammatik `mpg.grm`, und eine leicht geänderte Funktion `Parse.parse`. Das laden wir durch:

⟨*PA/mpg.parse.cm*⟩≡

```
 Library
   structure Absyn
   functor Peirce
   functor Semantik
   signature PARSE
   functor Parse
   structure NL_Parser
   structure NL_LrVals
   structure Interface
 is
   $/smlnj-lib.cm
   $/ml-yacc-lib.cm
   $/basis.cm

   set.sml          (* Set *)

   mabsyn.sml       (* Absyn:ABSYN *)
   peirce.sig       (* PEIRCE *)
   peirce.sml       (* PeirceTerm, Peirce *)
   pg.sml           (* structure D = Absyn *)
```

```
pg.lex
mpg.grm

nl.interface.sml  (* Interface *)
semantik.sml      (* Semantik *)
pg.link.sml       (* NL_Parser, NL_LrVals *)
mpg.parse.sml     (* Parse *)
```

Jede Interpretation kann dann auch für die modifizierte Grammatik verwendet werden. Für die Interpretation `pg5.sml` muß man dazu in SML nur `CM.make "mpg5.cm";` mit folgendem Makefile aufrufen:

$\langle$*PA/pg5.cm*$\rangle{+}{\equiv}$

```
Group
   structure NL
is
   mpg.parse.cm
   pg5.sml
```

Entsprechende Dateien für die anderen Interpretationen lauten analog.

4.3 Erweiterung der Grammatik um koordinierte Phrasen

Wenn man bei den Subjekt- und Objekttermen zwischen Singular und Plural unterscheidet, sollte man auch bei der Koordination von Termen unterscheiden: werden nur Eigennamen, nur Gemeinnamen, adjektivisch modifizierte Gemeinnamen, quantifizierte Gemeinnamen oder eine Mischung davon koordiniert?

Außerdem muß bei der Konjunktion ein Pluralterm entstehen, bei der Disjunktion nicht, und falls ein Pluralterm beteiligt ist, sollte die gesamte Kombination ein Pluralterm sein - zumindest ist das eine plausible Approximation an den Sprachgebrauch, der in diesen Fällen beim Verb eine Pluralform vorzieht.

Man sollte daher Kategorien wie `cPN`, `cN`, `cNP`, `cQN` und `cQNP` einführen. Außerdem braucht man einfache Klammerkonventionen, um Merhdeutigkeiten zu vermeiden. Dazu wollen wir die Junktoren möglichst auf der syntaktisch einfachsten Ebene interpretieren, so daß z.B. *eurpäische Männer und Frauen* als *europäische (Männer und Frauen)* aufgefaßt wird.

Da das zu vielen Konflikten im erzeugten Parser führt[12], beschränken wir uns auf `cNP` und `cNPpl`, komplexe Nominalphrasen im Singular und im Plural. (Insbesondere kann man keine `NP` im Singular mit einer im Plural koordinieren.)

Wir brauchen drei Distributionsregeln, um Aussagen mit komplexen Nominalphrasen in Kombinationen von Aussagen und Verbalphrasen zu übersetzen: eine

[12] Mit einer Vorausschau von k Konstituenten (geschweige denn, k Symbolen) kann noch nicht einmal eine Entscheidung zwischen der Regel `NPsg : (PN,)^* PN oder PN` für den Singular und der Regel `NPpl : (PN,)^* PN und PN` für den Plural getroffen werden.

für die Subjektsposition, eine für die Objektsposition, und eine für die Subjektsposition bei kollektiver Lesart.

⟨*PA/mpg.grm*⟩≡

```
open D        (* ProdS statt D.ProdS usw.
                 Trotzdem: D.set nötig für sig NL__Tokens *)

exception Undefined

fun Dom(r)        = PreIm(r,UnitS)
fun SymK(r)       = ProdR(r,Conv r)
fun Im(r,s)       = PreIm(Conv r,s)
fun UnPreIm(r,s)  = CompS(PreIm(CompR r,s))   (* r't alle s *)
fun Cart(s1,s2)   = ProdR(Cylin s1,Conv(Cylin s2))
fun MinorS(s,r)   = Dom(Minor(ProdR(r,Cart(s,s))))
fun Div(s)        = ProdR(Cart(s,s),CompR(Id(UnitS)))
fun PreImDiv(r,s) = Prod(Prod(r,Div s),Conv r)
fun RestrR(r,s)   = ProdR(r,Conv(Cylin s))

fun distr1(NP,VP) = case NP of
    Pn(s)   => E(ProdS(s,VP))          (* R 1 *)
  | Eq(s)   => E(ProdS(s,VP))          (* R 14 *)
  | EqPl(s) => ER(PreImDiv(Id(VP),s))  (* R 14b *)  (* !! *)
  | Nq(s)   => N(ProdS(s,VP))          (* R 15 *)
  | Uq(s)   => U(SumS(CompS(s),VP))    (* R 13 *)
  | PnPl(s) => U(SumS(CompS(s),VP))    (* R 53 *)
  | (Und(cNP1,cNP2))  => And(distr1(cNP1,VP),distr1(cNP2,VP))
  | (Nicht(cNP))      => Neg(distr1(cNP,VP))
  | (Oder(cNP1,cNP2)) => Or(distr1(cNP1,VP),distr1(cNP2,VP))

fun distr2(TV,cNP) = case cNP of
    Pn(s)   => (PreIm(TV,s))           (* R 7, 56 *)
  | Eq(s)   => (PreIm(TV,s))           (* R 20 *)
  | EqPl(s) => (Dom(ProdR(PreImDiv(TV,s),Id(UnitS))))
                                      (* 20b *)
  | Nq(s)   => (CompS(PreIm(TV,s)))    (* R 21 *)
  | Uq(s)   => (UnPreIm(TV,s))         (* R 19 *)
  | PnPl(s) => (UnPreIm(TV,s))         (* R 55 *)
  | (Und(cNP1,cNP2))  => ProdS(distr2(TV,cNP1),distr2(TV,cNP2))
  | (Oder(cNP1,cNP2)) => SumS(distr2(TV,cNP1),distr2(TV,cNP2))
  | (Nicht(cNP1))     => CompS(distr2(TV,cNP1))

fun distr3(cNPpl,VP) = case cNPpl of
    PnPl(s) => ER(ProdR(Div(s),VP))       (* R 63a *)
  | EqPl(s) => ER(ProdR(Div(s),VP))       (* R 72a *)
  | Nq(s)   => NR(ProdR(Div(s),VP))       (* R 73a *)
```

```
  | Uq(s)   => UR(SumR(CompR(Div(s)),VP)) (* R 71a *)
  | (Und(cNP1,cNP2))  => And(distr3(cNP1,VP),
                            distr3(cNP2,VP))
  | (Nicht(cNP))      => Neg(distr3(cNP,VP))
  | (Oder(cNP1,cNP2)) => Or(distr3(cNP1,VP),
                           distr3(cNP2,VP))

%%
%header (functor NL_LrValsFun (structure Token : TOKEN
                               structure Absyn : ABSYN) : NL__LRVALS)

%term
     EOF | PUNKT | Komma
   | und | oder | nicht
   | ist | sind
   | wird | werden
   | ein | eines
   | deren
   | zwar | aber | weder | noch
   | PN of D.set
   | PNgen of D.set
   | N1 of D.set  (*  N *)
   | N2 of D.rel  (* RN *)
   | IV of D.set
   | TV of D.rel
   | TVpart of D.rel
   | CA of D.set
   | IA of D.rel
   | IA_comp of D.rel
   | EQ | NQ | UQ   (* Quantoren: ein, kein, jedes *)
   | P of D.rel
   | von | als
   | N1pl of D.set
   | N2pl of D.rel
   | IVpl of D.set
   | TVpl of D.rel
   | CApl of D.set
   | IApl of D.rel
   | EQpl | NQpl | UQpl (* einige, keine, alle *)
   | SVpl of D.rel
   | RA of D.rel
   | Poss of D.rel

%nonterm
   start of absyn
   | S  of D.tv
```

```
     | NP of D.set
     | N2P of D.rel
     | AP of D.set
     | VP of D.set
     | IVP of D.set
     | NegIVP of D.set (* H.L. *)
     | NegTVP of D.set (* H.L. *)
     | TVP of D.set
     | PP of D.set
     | PassP of D.set
     | aPN of D.set
     | aPNP of D.set
     | cPNP  of D.set
     | pPNP  of D.rel
     | NPpl  of D.set
     | VPpl  of D.set
     | IVPpl of D.set
     | TVPpl of D.set
     | NegIVPpl of D.set
     | NegTVPpl of D.set
     | PPNP of D.rel
     | ColVPpl of D.rel
     | CNP of D.set
     | cNP of D.set D.comb
     | pNPsg of D.set D.comb
     | nNPsg of D.set D.comb
     | cNPpl of D.set D.comb
     | pNPpl of D.set D.comb
     | nNPpl of D.set D.comb

%start start

%eop EOF PUNKT
%noshift EOF PUNKT
%pos int

%left und
%left oder

%name NL_
%verbose

%%

start : S                 (nullary S)
      | VP                (unary VP)
```

```
     | TV                (binary TV)
     | N2                (binary N2)
     | cNP               (combi cNP)
     | cNPpl             (combi cNPpl)

S : cNP VP              (distr1(cNP,VP))         (* R 1,13,14,15,54 *)
     | cNPpl VPpl        (distr1(cNPpl,VPpl))     (* R 13,14b,15,53 *)
     | cNPpl ColVPpl     (distr3(cNPpl,ColVPpl)) (* R 63a,71a,72a,73a *)

VP : IVP                (IVP)                    (* R 2a *)
     | TVP               (TVP)                    (* R 2b *)
     | NegIVP            (NegIVP)                 (* R 16a *)
     | NegTVP            (NegTVP)                 (* R 16b *)
     | wird PassP        (PassP)                  (* R 45 *)

VPpl : IVPpl            (IVPpl)                  (* R 2a *)
     | TVPpl             (TVPpl)                  (* R 2b *)
     | NegIVPpl          (NegIVPpl)               (* R 16a *)
     | NegTVPpl          (NegTVPpl)               (* R 16b *)
     | werden PassP      (PassP)                  (* R 45 *)

IVP : IV                (IV)                     (* R 2 *)
     | ist AP            (AP)                     (* R 3 *)
     | ist EQ NP         (NP)                     (* R 5a *)
     | ist PP            (PP)                     (* R 8 *)
     | ist NP            (NP)                     (* R 10 *)

IVPpl : IVpl            (IVpl)                   (* R 2  pl *)
     | sind AP           (AP)                     (* R 3  pl *)
     | sind NPpl         (NPpl)                   (* R 5a pl *)
     | sind PP           (PP)                     (* R 8  pl *)

NegIVP : ist nicht PP   (CompS(PP))              (* R 8b *)
     | ist NQ NP         (CompS(NP))              (* R 10b *)
     | ist nicht AP      (CompS(AP))              (* R 17a *)
     | ist nicht NP      (CompS(NP))              (* R 18 *)

TVP : TV cNP            (distr2(TV,cNP))         (* R 7,20,56,21,19 *)
     | TV cNPpl          (distr2(TV,cNPpl))       (* R 19,20b,21,55 *)
     | TV PNgen N2pl     (UnPreIm(TV,PreIm(N2pl,PNgen)))
                                                  (* R 84a *)
     | TV Poss N2pl      (CompS(Dom(ProdR(CompR(TV),Conv(N2pl)))))
                                                  (* R 85 *)
     | TV PN und Poss N2pl (ProdS(PreIm(TV,PN),
                                 CompS(Dom(ProdR(CompR(TV),
                                                 Conv(N2pl))))))
```

```
                                                (* R 86 *)
     | TV CNP            (UnPreIm(TV,CNP))      (* R 87a *)
     | TV Poss NPpl      (CompS(Dom(ProdR(RestrR(Poss,NPpl),
                                  CompR(TV)))))
                                                (* R 89a *)

TVPpl : TVpl cNP         (distr2(TVpl,cNP))     (* R 7,19,20,21,56 *)
     | TVpl cNPpl        (distr2(TVpl,cNPpl))   (* R 19,20b,21,55 *)

NegTVP : TVP nicht       (CompS(TVP))           (* R 22 *)

NegTVPpl : TVPpl nicht   (CompS(TVPpl))         (* R 22 *)

AP : CA                  (CA)                   (* R 4 *)
     | IA_comp als PN    (PreIm(IA_comp,PN))    (* R 12 *)
     | IA_comp als UQ NP (UnPreIm(IA_comp,NP))  (* R 26 *)
     | IA_comp als EQ NP (PreIm(IA_comp,NP))    (* R 27 *)
     | IA_comp als NQ NP (CompS(PreIm(IA_comp,NP)))
                                                (* R 28 *)

NP : N1                  (N1)                   (* R 6 *)
     | N2P von PN        (PreIm(N2P,PN))        (* R 11a *)
     | N2P eines NP      (PreIm(N2P,NP))        (* R 11b *)
     | CA NP             (ProdS(CA,NP))         (* R 29 *)
     | IA NP             (MinorS(NP,IA))        (* R 31a *)

NPpl : N1pl              (N1pl)                 (* R 6 *)
     | N2pl von PN       (PreIm(N2pl,PN))       (* R 11 *)
     | N2pl eines NP     (PreIm(N2pl,NP))       (* R 11b *)
     | CApl NPpl         (ProdS(CApl,NPpl))     (* R 29 *)
     | IApl NPpl         (MinorS(NPpl,IApl))    (* R 31a *)

N2P : CA N2              (ProdR(N2,Cylin CA))   (* R 30 *)
     | N2                (N2)                   (* R 30b *)

PP : P PN                (PreIm(P,PN))          (* R 9 *)
     | P UQ NP           (UnPreIm(P,NP))        (* R 23 *)
     | P EQ NP           (PreIm(P,NP))          (* R 24 *)
     | P NQ NP           (CompS(PreIm(P,NP)))   (* R 25 *)

PassP : TVpart           (Im(TVpart,UnitS))     (* R 46 *)
     | von PN TVpart     (Im(TVpart,PN))        (* R 47 *)
     | von EQ NP TVpart  (Im(TVpart,NP))        (* R 48a *)
     | von UQ NP TVpart  (CompS(Im(CompR(TVpart),NP)))
                                                (* R 49a *)
     | von NQ NP TVpart  (CompS(Im(TVpart,NP))) (* R 50a *)
```

```
cPNP : PN und PN          (SumS(PN1,PN2))         (* R 51 *)
     | PN Komma cPNP      (SumS(PN,cPNP))

aPN : PN                  (PN)
     | aPNP               (aPNP)
aPNP : PN oder PN         (SumS(PN1,PN2))         (* R 52 *)
     | PN Komma aPNP      (SumS(PN,aPNP))

ColVPpl : sind N2pl       (SymK N2pl)             (* R 66 *)
     | sind RA            (SymK RA)               (* R 65 *)
     | SVpl               (SVpl)                  (* R 67a *)

CNP : PN und deren N2pl   (SumS(PN,PreIm(N2pl,PN))) (* R 88 *)

cNP : pNPsg                   (pNPsg)
     | nNPsg                  (nNPsg)
     | zwar pNPsg aber nNPsg  (Und(pNPsg,nNPsg))
     | zwar nNPsg aber pNPsg  (Und(nNPsg,pNPsg))
pNPsg : aPN                   (Pn(aPN))
     | EQ NP                  (Eq(NP))
     | UQ NP                  (Uq(NP))
     | pNPsg oder pNPsg       (Oder(pNPsg1,pNPsg2))
nNPsg : nicht PN              (Nicht(Pn(PN)))
     | NQ NP                  (Nq(NP))
     | weder pNPsg noch pNPsg (Und(Nicht(pNPsg1),Nicht(pNPsg2)))

cNPpl : pNPpl                 (pNPpl)
     | nNPpl                  (nNPpl)
     | zwar pNPpl aber nNPpl  (Und(pNPpl,nNPpl))
     | zwar nNPpl aber pNPpl  (Und(nNPpl,pNPpl))
pNPpl : cPNP                  (PnPl(cPNP))
     | EQpl NPpl              (EqPl(NPpl))
     | UQpl NPpl              (Uq(NPpl))
     | pNPpl oder pNPpl       (Oder(pNPpl1,pNPpl2))
     | pNPpl und pNPpl        (Und(pNPpl1,pNPpl2))
nNPpl : nicht pNPpl           (Nicht(pNPpl))
     | NQpl NPpl              (Nq(NPpl))
     | weder pNPpl noch pNPpl (Und(Nicht(pNPpl1),Nicht(pNPpl2)))
```

Die Auflösung der Boole'schen Kombinationen geht für die Objektposition so:

⟨*Beispiel mit mpg5.cm*⟩≡

```
- NL.parse();
Anna und Hans oder Emil und Maria.
Value is: ((Anna + Hans) + (Emil + Maria))
```

```
Grete liebt Anna und Hans oder Emil und Maria.
Value is: E((Grete . ( -(-lieben : (Anna + Hans)) +
                       -(-lieben : (Emil + Maria)))))

Anna oder Hans und Emil oder Maria.
Value is: ((Anna + Hans) . (Emil + Maria))
Grete liebt Anna oder Hans und Emil oder Maria.
Value is: E((Grete . ((lieben : (Anna + Hans)) .
                      (lieben : (Emil + Maria)))))
```

Und für die Subjektposition erhalten wir eine Kombination von Aussagen:

⟨*Beispiel mit mpg5.cm*⟩+≡

```
Anna und Hans oder Emil und Maria lieben Grete.
Value is: (U(( -(Anna + Hans) + (lieben : Grete))) \/
           U(( -(Emil + Maria) + (lieben : Grete))))
```

Problem 4.2 Die Grammatik `mpg.grm` erlaubt einfachheitshalber nur NP-Koordinationen. Man müßte die Boole'schen Operationen für weitere Kategorien erlauben, z.B.

⟨*Boole'sche Koordination bei CA und N1pl*⟩≡

```
cCA : CA und CA           (ProdS(CA1,CA2))
     | CA oder CA         (SumS(CA1,CA2))

cN1pl : N1pl und N1pl     (SumS(N1pl1,N1pl2))
     | N1pl oder N1pl     (SumS(N1pl1,N1pl2))
```

Aber das führt zu vielen *shift-reduce* und *reduce-reduce*-Konflikten im erzeugten Parser. Schon die Unterscheidung in positive und negative NPs geht nur für Singular und Plural getrennt, bei der Mischung erhält man zu viele Konflikte im erzeugten Parser.

Im Übrigen hat man natürlich auch Probleme bei der Sprachbeschreibung: einige Ausdrücke sind mehrdeutig oder höchstens mit Hilfe der Betonung eindeutig zu lesen. Man sollte daher "implizite Klammerungen" annehmen, etwa:

(a) *große Hunde und Ameisen* = *große Hunde und große Ameisen*,

(b) *einige Hunde und Ameisen* = *einige Hunde und einige Ameisen*,

(c) *alle Hunde oder Ameisen* = *alle Hunde und alle Ameisen*,

(d) *große und kleine Hunde* = *große Hunde und kleine Hunde*

Außerdem sind schon die angegebenen komplexen Nominalphrasen in Verbindung mit der Negation inkorrekt: man müßte z.B. *nicht ein NP* und *nicht einige NPpl* ausschließen.

Beispiel 4.3 Noch ein Beispiel zur Koordination mit einer `ColVP`:

⟨*Beispiel mit mpg5.cm*⟩+≡

```
weder alle Männer noch Emil und Hans sind Geschwister.
Value is: (Neg(UR((-(((Mann x 1) . ~(Mann x 1)) . -Id(1))
                   + (Geschwister . ~Geschwister))))
           /\ Neg(ER((((((Emil + Hans) x 1)
                          . ~((Emil + Hans) x 1)) . -Id(1))
                         . (Geschwister . ~Geschwister)))))
```

Auch Sätze wie den folgenden kann die Grammatik von Böttner nicht behandeln:

⟨*Beispiel mit mpg5.cm*⟩+≡

```
Grete liebt zwar einen Mann aber weder Emil noch Dieter.
Value is: E((Grete . ((lieben : Mann)
                      . ( -(lieben : Emil) .  -(lieben : Dieter)))))
```

Nächste Schritte:

(a) Fehler in `Parse.loop` sauber beheben (wo fehlt `D = Absyn`?).

(b) Ausnahmebehandlung für die Ausnahmen `Undefined` und `NoValue` schreiben

(c) Zusammensetzen der von ML-Lex und ML-Yacc erzeugten Strukturen mit einem höherstufigen Funktor, d.h. *unabhängig von* der Beispielgrammatik und -Lexikon, formulierten:

⟨*Linken mit höherstufigem Funktor*⟩≡

```
functor Parse(functor LexFun    : LEXFUN_FSIG
              functor LrValsFun : LRVALS_FSIG
              structure Algebra : PEIRCEfull
              ...) : PARSE = ...
```

Das dürfte schwierig werden, weil man tief in ML-Yacc reinschauen muß, um die zur Definition von `Parse` nötigen Argumenstrukturen und Signaturen zu bekommen.

Einfacheres Verbinden der von Lexer und Parser

Versuch, höherstufige Funktoren zu benutzen:

⟨*hofun.sml*⟩≡

```
functor MyJoin(functor LrValsFun(structure Token : TOKEN) :
                 sig
                    structure ParserData : PARSER_DATA
                    structure Tokens : NL__TOKENS
                 end
               functor LexFun(structure Tokens : NL__TOKENS) : LEXER) =
   struct
      structure Interface = Interface();
      structure NL_LrVals
        = LrValsFun(structure Token = LrParser.Token)
      structure NL_Lex
        = LexFun(structure Tokens = NL_LrVals.Tokens
                    structure Interface = Interface)
      structure NL_Parser =
         Join(structure ParserData = NL_LrVals.ParserData
              structure Lex = NL_Lex
              structure LrParser = LrParser)
   end
```

Literatur

[1] George H. Boole. *An Investigation of the Laws of Thought.* Macmillan, London, 1854.

[2] Michael Böttner. *Relationale Grammatik.* Max Niemeyer Verlag, Tübingnen, 1999.

[3] Chris Brink, Katharina Britz, and Renate A. Schmidt. Peirce algebra. Technical Report MPI-I-92-229, Universität Saarbrücken, 1992.

[4] David R. Dowty, Robert E. Wall, and Stanley Peters. *Introduction to Montague Semantics.* D. Reidel, Dordrecht, 1981.

[5] Gottlob Frege. Über Sinn und Bedeutung. *Zeitschrift für Philosophie und philosophische Kritik*, NF 100:25–50, 1892.

[6] Richard Montague. The proper treatment of quantification in ordinary english. In Richmond Thomason, editor, *Formal Philosophy. Selected Papers of Richard Montague.* Yale Univ. Press, 1974.

[7] Charles Sanders Peirce. Die Logik der Relative (The Monist, VII, 1897). In C.J.W.Kloesel and H.Pape, editors, *Semiotische Schriften, Band 1*, stw 1480. Suhrkamp, 2000.

[8] David R. Tarditi and Andrew W. Appel. *ML-Yacc User's Manual. Version 2.3*, 1994.