

Erzeugung von Parsern für Peirce-Grammatiken
Seminar „Relationale Grammatik“
CIS, WS 2009/10

Hans Leiß

Universität München
Centrum für Informations- und Sprachverarbeitung

30.11./7.12. 2009

Quellen

1. SML of New Jersey, ML-Lex, ML-Yacc: Binär+Quelldateien
 - <http://www.smlnj.org> (siehe Downloads)
2. User's Guide to ML-Lex and ML-Yacc:
 - <http://rogerprice.org>oder in den Quelldateien von SML:
 - `smlnj/ml-lex/lexgen.tex`
 - `smlnj/ml-yacc/doc/mlyacc.tex`
3. ML-GLR: Quelldateien und Installationsanweisung
 - Subversion-Archiv (mit Leserechten; Benutzernamen und Passwort wurden im Seminar verteilt)
<https://svn.cip.ifi.lmu.de/~leiss/svn/hl-yacc>
4. User's Guide to ML-GLR:
 - Homepage des Seminars (.pdf), oder im Archiv unter `doc/ml-glr.pdf`

Bisher implementiert:

1. Endliches U : UNIVERSUM aus Elementen und benannten Mengen und Relationen: `PersonenU`
2. Erzeugung der vollen Peirce-Algebra $\mathcal{P}(U)$ über der Menge U durch den Funktor `Peirce(U:UNIVERSUM) : PEIRCEfull`
3. Endliche Peirce-Algebra: `Personen` = `Peirce(PersonenU)`
4. Grammatiken im ML-GLR-Format mit
 - 4.1 Auswertung zu Werten eines Datentyps
 - 4.2 Auswertung durch SML-Funktionen als λ -Termen

Jetzt:

1. Parser mit Peirce-Termen als Werten (abstrakte Syntax)
2. Auswertung von Peirce-Termen in einer Peirce-Algebra
3. Parse-Funktionen

Algebra der Peirce-Terme

In der Term-Algebra werden die Funktionen der Signatur PEIRCE durch Konstruktionsschritte beim Termaufbau „interpretiert“:

$\langle PA-GLR/peirce.sml \rangle \equiv$

```
structure PeirceTerm =  
  struct  
    datatype set = ConS of string (* Mengennamen *)  
                | SumS of set * set  
                | ZeroS  
                | ProdS of set * set  
                | UnitS  
                | CompS of set  
                | PreIm of rel * set (* R : B *)
```

So werden "Konstruktoren" `ConS : string -> set` eingeführt.

Beispiel: `PreIm(ConR "Bruder", ConS "Student") : set`

Relationsterme:

$\langle PA-GLR/peirce.sml \rangle + \equiv$

```
and      rel = ConR of string
          | SumR of rel * rel
          | ZeroR
          | ProdR of rel * rel
          | UnitR
          | CompR of rel
          | Id of set
          | Prod of rel * rel
          | Conv of rel
          | Cylin of set
          | Minor of rel      (* HL *)
          | Tc of rel         (* HL *)
```

Beispiel: `ProdR(Id(ConS "Mensch"),ConR "jünger") : rel`

Wahrheitswert-Terme:

$\langle PA\text{-}GLR/peirce.sml \rangle + \equiv$

```
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 (* Identitätsrelation *)
```

Beispiel: $E(\text{ProdS}(\text{ConS "Student"}, \text{ConS "arbeiten"}))$: tv

Für eine bessere Lesbarkeit drucken wir Peirce-Terme mit den üblichen (überladenen) Symbolen +, *, ... aus:

Ausgabe von Mengentermen: StoString : set -> string

$\langle PA-GLR/peirce.sml \rangle + \equiv$

```
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) ^ ")"
```

Beispiel: PreIm(ConR "Bruder", ConS "Student")

\mapsto "(Bruder : Student)"

Ausgabe von Relationstermen: RtoString : rel -> string

$\langle PA-GLR/peirce.sml \rangle + \equiv$

```
and RtoString (ConR(r))      = r
  | RtoString UnitR = " 1' " | RtoString ZeroR = " 0' "
  | RtoString (SumR(R,S)) =
    "(" ^ (RtoString R) ^ " + " ^ (RtoString S) ^ ")"
  | RtoString (ProdR(Cylin A,Conv(Cylin B))) =
    "(" ^ (StoString A) ^ " x " ^ (StoString B) ^ ")"
  | RtoString (ProdR(R,S)) =
    "(" ^ (RtoString R) ^ " . " ^ (RtoString S) ^ ")"
  | RtoString (CompR(R)) = "-" ^ (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)"
  | RtoString (Tc(R)) = "tc(" ^ (RtoString R) ^ ")"
  | RtoString (Minor(R)) = "minor" ^ (RtoString R)
```

Ausgabe von Wahrheitswerttermen: TtoString: tv -> string

$\langle PA-GLR/peirce.sml \rangle + \equiv$

```
and TtoString (E A) = "0 < " ^ (StoString A)
  | TtoString (U A) = "1 = " ^ (StoString A)
  | TtoString (N A) = "0 = " ^ (StoString A)
  | TtoString (ER A) = "0' < " ^ (RtoString A)
  | TtoString (UR A) = "1' = " ^ (RtoString A)
  | TtoString (NR A) = "0' = " ^ (RtoString A)
  | TtoString (Neg A) = "-(" ^ (TtoString A) ^ ")"
  | TtoString (And(A,B)) = "(" ^ (TtoString A) ^
                          " /\ " ^ (TtoString B) ^ ")"
  | TtoString (Or(A,B)) = "(" ^ (TtoString A) ^
                          " \/ " ^ (TtoString B) ^ ")"
```

end

Beispiel: E(ProdS(ConS "Student", ConS "arbeiten"))

\mapsto "0 < (Student . arbeiten)"

Bemerkung:

- PeirceTerm erfüllt die Axiome der Peirce-Algebra *nicht*, z.B.

$$\text{SumS}(\text{ZeroS}, \text{ZeroS}) \neq \text{ZeroS}.$$

- Um eine PA zu erhalten, müßte man beweisbar gleiche Terme in Äquivalenzklassen

$$[t]_{\equiv} := \{s \mid PA \vdash t = s\}$$

zusammenfassen und die Operationen wie in

$$\text{SumS}([s]_{\equiv}, [t]_{\equiv}) := [\text{SumS}(s, t)]_{\equiv}$$

definieren.

Peirce-Terme als abstrakte Syntax

Da eine ML-GLR-Grammatik ein Startsymbol mit eindeutigem Werttyp haben muß, verpacken wir die Peirce-Terme der Typen `set`, `rel` und `tv` in einen eindeutigen Typ `absyn`:

```
<PA-GLR/absyn.sml>≡  
signature ABSYN =  
  sig  
    datatype set = ConS of string | ...  
                | PreIm of rel * set  
    and rel = ConR of string | ... | Tc of rel  
    and tv =      E of set | ... | Neg of tv  
  
    datatype absyn = nullary of tv  
                  | unary of set  
                  | binary of rel  
    val toString : absyn -> string  
  end
```

Um diesen Datentyp erweitern wir die Struktur der Peirce-Terme:

$\langle PA\text{-}GLR/absyn.sml \rangle + \equiv$

```
structure Absyn : ABSYN =
  struct
    open PeirceTerm
    datatype absyn = nullary of tv
                    | unary of set | binary of rel
    fun toString (unary s) = SttoString s
      | toString (binary s) = RttoString s
      | toString (nullary s) = TttoString s
  end
```


Die Grammatik pa.glr

$\langle PA-GLR/pa.glr \rangle \equiv$

open Absyn (* ProdS statt Absyn.ProdS usw. *)

%%

<pa.glr-header>

%term

EOF | PUNKT

| PN of set | CN of set | RN of rel

| CopV | IV of set | TV of rel

| CA of set

| ein | eines | EQ | NQ | UQ (* ein, kein, jedes *)

%nonterm

start of absyn option

| S of tv | N' of set | AP of set

| VP of set | IVP of set | TVP of set

%start start

$\langle PA\text{-}GLR/pa.glr \rangle + \equiv$

%eop EOF PUNKT

%pos int

%name NL_

%%

start : S (print (TtoString S ^ "\n"); SOME(nullary(S)))
| N' (print (StoString N' ^ "\n"); SOME(unary(N')))
| PN (print (StoString PN ^ "\n"); SOME(unary(PN)))
| VP (print (StoString VP ^ "\n"); SOME(unary(VP)))
| (print "Value is neither S, PN, N', nor VP\n"; NONE)

S : PN VP (E(ProdS(PN,VP))) (* R1 *)

VP : IVP (IVP) (* R2a *)

| TVP (TVP) (* R2b *)

IVP : IV (IV) (* R2 *)

| CopV AP (AP) (* R3 *)

| CopV ein N' (N') (* R5 *)

$\langle PA-GLR/pa.glr \rangle + \equiv$

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

AP : CA              (CA)                    (* R4 *)
N' : CN              (CN)                    (* R6 *)
      | RN eines N' (PreIm(RN,N')) (* NPgen-Attribut *)
```

Um große Terme zu vermeiden, kann man in den UserDeclarations Abkürzungen einführen und in den Regeln benutzen, z.B.

```
fun Exp(R,A) = CompS(PreIm(Conv(CompR(Conv(R))),A))
%%
TVP : TV UQ N' (Exp(TV,N'))
```

Angabe eines passenden Lexers `pa.lex`

Wir müssen hier die von ML-GLR erwarteten Lexer-Ergebnisse vom Typ `token list * string` abliefern:

$\langle PA\text{-}GLR/pa.lex \rangle \equiv$

```
structure T = Tokens
structure I = Interface
structure A = Absyn
```

```
open I
type pos = I.pos
type svalue = T.svalue
type ('a,'b) token = ('a,'b) T.token
type lexresult = (svalue,pos) token list * string
```

```
fun eof() = (T.EOF(!line,!line)::nil,"")
```

Der %header muß verlangen, daß Absyn:ABSYN ein Argument von NL_LexFun ist, da die Token Peirce-Konstante enthalten:

$\langle PA\text{-}GLR/pa.lex \rangle + \equiv$

```
fun CN(s:string) = ([T.CN(A.ConS s,!line,!line)],s)
fun RN(s:string) = ([T.RN(A.ConR s,!line,!line)],s)
fun PN(s:string) = ([T.PN(A.ConS s,!line,!line)],s)
fun IV(s:string) = ([T.IV(A.ConS s,!line,!line)],s)
fun TV(s:string) = ([T.TV(A.ConR s,!line,!line)],s)
fun CA(s:string) = ([T.CA(A.ConS s,!line,!line)],s)
```

%%

%header (

functor NL_LexFun(structure Absyn: ABSYN

structure Tokens: NL__TOKENS

sharing type Absyn.set = Tokens.set

structure Interface: INTERFACE):LEXER);

%full

whitespace=[\t\]*;

%%

$\langle PA\text{-}GLR/pa.lex \rangle + \equiv$

```
<INITIAL>{whitespace} => (lex());  
\n          => (next_line(); lex());  
"."       => (T.PUNKT(!line,!line)::nil,yytext);  
"ist"     => (T.CopV(!line,!line)::nil,yytext);  
("ein" | "eine") => (T.ein(!line,!line)::nil,yytext);  
"eines"   => (T.eines(!line,!line)::nil,yytext);  
"einen"   => (T.EQ(!line,!line)::nil,yytext);  
("keinen" | "kein" | "keine")  
          => (T.NQ(!line,!line)::nil,yytext);  
("alle" | "jeder" | "jede" | "jeden")  
          => (T.UQ(!line,!line)::nil,yytext);
```

Die Ergebnisse des Lexers für bedeutungstragende Wörter werden mit den Abkürzungen CN,...,CA von oben angegeben:

$\langle PA-GLR/pa.lex \rangle + \equiv$

```
"Anna"      => (PN "Anna");
"Emil"      => (PN "Emil");
"Mann"      => (CN "Mann");
"Frau"      => (CN "Frau");
"Hund"      => (CN "Hund");
("Student" | "Studenten") => (CN "Student");
"Nachbar"   => (RN "Nachbar");
"Bruder"    => (RN "Bruder");
"gesund"    => (CA "gesund");
("arbeiten" | "arbeitet") => (IV "arbeiten");
("kennen" | "kennt")      => (TV "kennen");
("lieben" | "liebt")      => (TV "lieben");
. => (error ("ignoring illegal character " ^ yytext,
           !line,!line); lex());
```

Schnittstelle des Lexers

Beim Lesen der Eingabe soll der Lexer (nur) die Zeilen mitzählen, damit man Zeilennummern in Fehlermeldungen ausgeben kann.

Damit jede Initialisierung des Lexers einen neuen Zähler hat, wird Interface durch eine Funktoranwendung jeweils neu erzeugt:

$\langle PA\text{-}GLR/interface.sml \rangle \equiv$

```
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,l:pos,_) =
    print("Error, line " ^ (Int.toString l) ^ ": "
      ^ errmsg ^ "\n")
  type arg = unit
  val nothing = ()
end
structure Interface = Interface()
```

Verbinden von Lexer und Parser

Eine Anwendung von `ml-glr` auf `pa.glr` (via `CM.make`) erzeugt

`<PA-GLR/pa.grm.lnk.sml>`≡

```
structure NL_Vals =
  NL_ValsFun(structure Token = ChartParser.Token
              structure Grammar = ChartParser.Grammar
              structure Absyn = Absyn)
structure NL_Lex =
  NL_LexFun(structure Absyn = Absyn
             structure Tokens = NL_Vals.Tokens
             structure Interface = Interface)
structure NL_Parser =
  Join(structure ParserData = NL_Vals.ParserData
        structure Lex = NL_Lex
        structure ChartParser = ChartParser)
```

NL_Parser : PARSE

```
signature PARSE =
  sig
    ...
    type pos      (* token positions *)
    type svalue   (* values at nodes of parse trees *)
    type result   (* extracted from the root's value *)
    type lexresult = (svalue,pos) Token.token list * string

    val makeLexer : (int -> string) -> lexresult Stream.stream
    val parse : (lexresult Stream.stream) *
                (string * pos * pos -> unit) * arg
                -> result list * (lexresult Stream.stream)
    val debug : bool ref
  end
```

Die Struktur NL_Parser muß man um Bedienfunktionen erweitern.

```
reader: int -> string
```

Den von ML-GLR erzeugten Parser `NL.Parser` ruft man auf mit
`NL.Parser.parse(<tokenstream>, <error>, <pa.glr %arg>)`

Den Tokenstrom `lexer` erzeugt `NL.Parser.makeLexer` aus einem
`reader: int -> string`, hier dem Lesen von der Konsole.

<parse für pa.glr> \equiv

```
fun reader n = (* von der Konsole lesen *)
  case (TextIO.inputLine TextIO.stdIn) of
    SOME s => s | NONE => ""
```

Zum Parsen erzeugt man einen Tokenstrom

```
val lexer = Parser.makeLexer reader
```

und liest davon so lange, bis EOF gefunden wurde:

parse: unit -> result list

$\langle \text{parse für pa.gr} \rangle + \equiv$

```
structure I = Interface
```

```
fun parse () =
```

```
  let
```

```
    val (_,e) = (I.init_line(),!I.line)
```

```
    fun loop lexer =
```

```
      let val (result,lexer) =
```

```
          Parser.parse(lexer,I.error,I.nothing)
```

```
          val ((nextTokens,_),lexer) =
```

```
              Parser.Stream.get lexer
```

```
      in
```

```
        if Parser.sameToken(hd nextTokens,
```

```
                              Tokens.EOF(e,e))
```

```
        then result else loop lexer
```

```
      end
```

```
in loop (Parser.makeLexer reader) end
```

Parse: INTERFACE*PARSER*NL_TOKENS -> PARSE

Die Bedienfunktionen setzen wir aus Interface, den Tokens von NL_Vals und dem Parser NL_Parser mit einem Funktor Parse zu einer Struktur NL:PARSE zusammen, wobei

<PA-GLR/pa.parse.sml>≡

```
signature PARSE =
```

```
  sig
```

```
    type svalue
```

```
    type pos
```

```
    type ('a,'b) token
```

```
    type result
```

```
    val parse : unit -> result list
```

```
    val trees : bool ref
```

```
    val debug : bool ref
```

```
  end
```

<PA-GLR/pa.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
  type result = Parser.result
  val trees = Parser.trees
  val debug = Parser.debug
  <parse für pa.glr>
```

```
end
```

Anwendung des Funktors Parse ohne Auswertung

Die von ML-Lex und ML-GLR erzeugten Teile `NL_Vals.Tokens` und `NL_Parser` werden durch `Parse` zusammengesetzt:

`<PA-GLR/pa.parse.sml>+≡`

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

Erstellung des Parsers für pa.glr ohne Auswertung

$\langle PA\text{-}GLR/pa.cm \rangle \equiv$

Library

structure NL

is

\$/basis.cm

\$/ml-glr-lib.cm

set.sml (* für PeirceTerm unnötig *)

peirce.sig

peirce.sml (* Peirce(U):PEIRCEfull braucht Set *)

absyn.sml

pa.lex

pa.glr (* erzeugt pa.glr.sig|sml, pa.glr.lnk.sml

interface.sml

pa.interface.sml

pa.parse.sml (* definiert structure NL: PARSE *)

Konstruiere NL in SML mit `CM.make "PA-GLR/pa.cm"`;

Benutzung

Standard ML of New Jersey v110.67 [...]

```
- CM.make "PA-GLR/pa.cm";
```

```
...
```

```
- NL.parse();
```

```
Anna ist ein Student.
```

```
0 < (Anna . Student)
```

```
Emil arbeitet.
```

```
0 < (Emil . arbeiten)
```

```
Anna kennt einen Studenten.
```

```
0 < (Anna . (kennen : Student))
```

```
Anna kennt keinen Studenten.
```

```
0 < (Anna . -(kennen : Student))
```

```
Anna kennt jeden Studenten.
```

```
0 < (Anna . -(~kennen : Student))
```

```
.
```

```
Value is neither S, PN, N', nor VP
```

```
^C^C
```

Auswertung von Peirce-Termen in einer PA

Bisher wurden Peirce-Terme nur lesbar als Strings ausgegeben.
Jetzt sollen sie in einer PA ausgewertet werden.

Wir bauen die abstrakte Syntax und ein Modell $\mathcal{P}(U)$ zu einer Struktur mit einer Auswertungsfunktion zusammen:

$\langle PA-GLR/semantics.sml \rangle \equiv$

```
signature SEM = sig
    type absyn
    type value
    val eval : absyn -> value
end
```

Semantics : ABSYN * PEIRCEfull -> SEM

$\langle PA\text{-}GLR/semantics.sml \rangle + \equiv$

```
functor Semantics(structure A:ABSYN structure P:PEIRCEful
                  : SEM where type absyn = A.absyn =
```

```
  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.PreIm(R,A)) = P.PreIm(evalR R,evalS A)
```

```
    | ...
```

```
  and evalR(A.Cylin A) = P.Cylin(evalS A)
```

```
    | ...
```

```
  and evalT(A.Neg A) = P.Neg(evalT A)
```

```
    | ...
```

```
end
```

Neue Signatur der Bedienung

`<PA-GLR/pg.parse.sml>≡`

```
signature PARSE =
```

```
  sig
```

```
    structure Absyn : ABSYN
```

```
    structure Semantics : SEM
```

```
    val parse : unit -> Absyn.absyn list
```

```
    val ev : string -> Absyn.absyn list
```

```
    val parse_file : string -> Absyn.absyn list list
```

```
    val test_file : string -> unit
```

```
    val parse_seq : unit -> Absyn.absyn list list
```

```
    val eval : Absyn.absyn -> Semantics.value
```

```
    val evals : string -> Semantics.value list
```

```
    val trees : bool ref
```

```
    val debug : bool ref
```

```
  end
```

Damit man dieselbe Grammatik `pg.glr` mit verschiedenen Interpretationen benutzen kann, ist ein einfacherer Aufruf der Konstruktion des Parsers nützlich:

`<PA-GLR/pg.parse.sml>+≡`

```
functor Parse (structure Algebra : PEIRCEfull) =
  ParseAux(structure Absyn = Absyn
            structure Algebra = Algebra
            structure Interface = Interface
            structure Parser = NL.Parser
            structure Tokens = NL.Vals.Tokens)
```

```
functor ParseAux(...) : PARSE = ...
```

(* ergänzt die Bedienfunktionen an NL.Parser *)

Erstellung von Parse: PEIRCEfull -> PARSE

Dieses neue Parse wird für `pg.glr` und `pg.lex` in der folgenden Datei `pg.parse.cm` konstruiert: (vgl. `pa.cm`)

$\langle PA\text{-}GLR/pg.parse.cm \rangle \equiv$

```
Library
  structure Absyn
  functor Peirce
  functor Semantics
  signature PARSE
  functor Parse
  structure NL_Parser
  structure NL_Vals
  structure Interface
is
  $/basis.cm
  $/ml-glr-lib.cm
```

$\langle PA\text{-}GLR/pg.parse.cm \rangle + \equiv$

\$smlnj/compiler.cm (* Control.Print.printDepth *)

absyn.sml (* Absyn:ABSYN *)

peirce.sig (* PEIRCE *)

peirce.sml (* PeirceTerm, Peirce *)

set.sml (* Set *)

pg.lex

pg.glr (* NL_ValsFun, .glr.lnk.sml, NL *)

interface.sml (* Interface functor *)

pa.interface.sml (* Interface structure *)

semantics.sml (* Semantics *)

pg.parse.sml (* Parse:PEIRCEfull->PARSE *)

Hat man mit `CM.make "pg.parse.cm"`; den Funktor `Parse` konstruiert, so erhält man aus einem Universum U durch

$\langle \textit{Anwendung des Funktors Parse} \rangle \equiv$
`structure NL = Parse(structure Algebra =`
`Peirce(U:UNIVERSUM):PEIRCEfull)`

einen Parser, der die Eingabe in einen Peirce-Term analysiert und diesen in der Peirce-Algebra $\mathcal{P}(U)$ auswertet.

Mit derselben Grammatik `pg.g1r` und demselben Lexikon `pg.lex` kann man dann Ausdrücke in Peirce-Algebren $\mathcal{P}(U_1), \dots, \mathcal{P}(U_k)$ auswerten.

Man braucht für einen Parser “mit Auswertung” also :

- die Angabe einer Grammatik `pg.glr` mit `%header`

```
functor NL_ValsFun(structure Absyn: ABSYN
                  structure Token: TOKEN
                  structure Grammar: GRAMMAR
                  sharing type Token.term =
                      Grammar.term)
```

und passendem `%lexheader`,

- die Angabe eines Lexers `pg.lex` mit `%header`

```
functor NL_LexFun(structure Absyn: ABSYN
                  structure Tokens: NL__TOKENS
                  sharing type Absyn.set = Tokens.set
                  structure Interface: INTERFACE):LEXER
```

- ein Universum `U:UNIVERSE` und den “Makefile” für
`Peirce:UNIVERSE -> PEIRCEfull` und `Parse:PEIRCEfull`
`-> PARSE`.

Beispiel Auswertung in der PA der Personen:

$\langle PA\text{-}GLR/pg1.sml \rangle \equiv$

```
structure PersonenU = struct ... end
structure Personen  = Peirce(structure U = PersonenU)
structure NL1 = Parse(structure Algebra = Personen)
```

Ein Makefile für das Beispiel ist jetzt ganz kurz:

$\langle PA\text{-}GLR/pg1.cm \rangle \equiv$

```
Group
  structure NL1
is
  pg.parse.cm  (* Peirce : UNIVERSE -> PEIRCEfull,
                Parse:  PEIRCEfull -> PARSE          *)
  pg1.sml      (* PersonenU:UNIVERSE,
                NL1 = Parse(Peirce(U)) *)
```

Nun kann man in SML den Parser erzeugen:

```
⟨Erzeugen des Parsers für pg.glr mit Auswertung in Personen⟩≡  
- CM.make "pg1.cm";
```

und dann die Auswertung testen:

```
⟨Test einer Ausgabe in abstrakter Syntax⟩≡  
- NL1.ev "Anna arbeitet";  
Absyn: 0 < (Anna . arbeiten)  
val it = [nullary (E (ProdS (ConS "Anna",ConS "arbeiten"))  
: Absyn.absyn list
```

```
⟨Test einer Auswertung⟩≡  
- NL1.eval "Anna arbeitet";  
Absyn: 0 < (Anna . arbeiten)  
val it = [nu true] : Semantics.value list  
- NL1.eval "liebt einen Studenten";  
Absyn: (lieben : Student)  
val it = [un (Set [c,b,a,e])] : Semantics.value list
```

In Zukunft sollte man also nur noch:

- die Angabe `pg.lex` des Lexers erweitern;
- die Angabe `pg.glr` der Grammatik erweitern;
- in `pgn.sml` das Universum U_n und den Parser mit Bedienung in NLn : `PARSE` für die Auswertung in $\mathcal{P}(U_n)$ definieren.

Dann kann man den Parser

- erzeugen mit dem Makefile

```
 $\langle PA-GLR/pgn.cm \rangle \equiv$ 
```

```
Group structure NLn is pg.parse.cm pgn.sml
```

- und testen mit einer Datei `pgn.tests` von Beispielsätzen.

Test von Beispielen

$\langle PA-GLR/pg1.tests \rangle \equiv$

Anna arbeitet.

Dieter ist gesund.

Dieter ist ein Student.

$\langle \text{Auswertung einer Datei von Beispielen} \rangle \equiv$

```
- parse_file "PA-GLR/pg1.tests";
```

```
val it =
```

```
  [[nullary (E (ProdS (ConS "Anna", ConS "arbeiten")))],
```

```
   [nullary (E (ProdS (ConS "Dieter", ConS "gesund")))],
```

```
   [nullary (E (ProdS (ConS "Dieter", ConS "Student")))]]
```

```
: Absyn.absyn list list
```

Die Analysen/Werte jedes Satzes werden in einer Liste gesammelt, und die Liste dieser Listen ist das Ergebnis.