A functional format for natural language grammars

Aarne Ranta
Xerox Research Centre Grenoble and Academy of Finland

Nancy, March 20, 1998

Logical syntax and semantics, *tectogrammar* and *phenogrammar* (Curry 1963, Montague 1974)

Type-theoretical grammar for fragments of natural languages (Ranta 1994, 1995, 1997)

Natural language analysis in typed functional programming languages such as ML, Miranda, and Haskell
(Frost & Launchbury 1989, Jones, Hudak & Shaumyan 1995)

Grammar defining the *correctness* of a piece of text, both linguistic and logical (cf. grammars of programming languages and other formalisms)

Abstract syntax + language-dependent *linear patterns*

Linearization and *parsing* derived from a declarative grammar format

$$\begin{tikzcd}
\text{parsing} \uparrow & & \text{abstract functional syntax} \arrow[dl, leftrightarrow] \arrow[d, leftrightarrow] \arrow[dr, leftrightarrow] & & \downarrow \text{linearization} \\
& \text{English} & \text{Finnish} & \text{French} &
\end{tikzcd}$$

$$\begin{tikzcd}
 & \mathrm{PredA1}(2, \mathrm{Ev}) \arrow[dl, leftrightarrow] \arrow[d, leftrightarrow] \arrow[dr, leftrightarrow] & \\
\text{2 is even} & \text{2 on parillinen} & \text{2 est pair}
\end{tikzcd}$$

$$\begin{tikzcd}
 & \mathrm{ModCN}(\mathrm{Num}, \mathrm{Ev}) \arrow[dl, leftrightarrow] \arrow[d, leftrightarrow] \arrow[dr, leftrightarrow] & \\
\text{even number} & \text{parillinen luku} & \text{nombre pair}
\end{tikzcd}$$

$$\begin{tikzcd} & \mathrm{PredA2}(\mathrm{Eq}(?), \mathrm{UseVar}(?, x), \mathrm{UseVar}(?, y)) \arrow[dl, leftrightarrow] \arrow[dr, leftrightarrow] & \\ x \text{ is equal to } y & & x \text{ est } \left\{ \begin{array}{c} \text{égal} \\ \text{égale} \end{array} \right\} \text{ à } y \end{tikzcd}$$

There exists a number x such that x is even and x is prime.

Il existe un nombre x tel que x soit pair et (que) x soit premier.

Il existe un nombre x tel que x soit pair et x est premier.

If a line contains a point then *it* does not lie outside *it*.

Si une droite contient un point alors *il* n'est pas extérieur à *elle*.

Phrase structure rule:

$$\text{Pred.}\ \ \text{S} \ \rightarrow\ \text{NP VP}$$

Inference rule:

$$\frac{Q : \text{NP} \quad F : \text{VP}}{\text{Pred}(Q, F) : \text{S}}$$

Function declaration:

$$\text{Pred} : (Q : \text{NP})(F : \text{VP})\text{S}$$

Phrase structure rule with terminals and nonterminals

$$F.\ \alpha \longrightarrow t_0\, \alpha_1\, t_1\, \alpha_2\, \ldots\ \alpha_n\, t_n$$

Function declaration + linearization rule

$$\begin{cases} F : (x_1 : \alpha_1)(x_2 : \alpha_2) \ldots (x_n : \alpha_n)\alpha \\ F(a_1, \ldots, a_n)^{\mathrm{o}} \ = \ t_0\, a_1^{\mathrm{o}}\, t_1 \ldots\ a_n^{\mathrm{o}}\, t_n \end{cases}$$

Function declaration + linear pattern

$$F : (x_1 : \alpha_1)(x_2 : \alpha_2) \cdots (x_n : \alpha_n)\alpha \ - \ t_1\, t_2\, \ldots\, t_m$$

where each t_k is either one of the variables x_i or a string

Permutation:

$$\mathrm{Mod} \;:\; (X : \mathrm{CN})(Y : \mathrm{A1}(X))\mathrm{CN},$$
$$\mathrm{Mod}(A, B)^{\mathrm{o}} \;=\; B^{\mathrm{o}}\, A^{\mathrm{o}}.$$

Reduplication:

$$\mathrm{Univ} \;:\; (X : \mathrm{CN})\mathrm{NP}(X),$$
$$\mathrm{Univ}(A)^{\mathrm{o}} \;=\; A^{\mathrm{o}}\ \textit{kuin}\ A^{\mathrm{o}}.$$

Suppression:

$$\mathrm{Def} \;:\; (X : \mathrm{CN})(Y : \mathrm{NP}(X))\mathrm{NP}(X),$$
$$\mathrm{Def}(A, x)^{\mathrm{o}} \;=\; \textit{the}\ A^{\mathrm{o}}.$$

From the pattern

$$F : (x_1 : \alpha_1) \cdots (x_n : \alpha_n)\alpha \ - \ t_1 \ \ldots \ t_m$$

we derive

$$F(a_1, \ldots, a_n)^{\mathrm{o}} = s_1 \ \ldots \ s_m$$

where

$$s_i = \begin{cases} a_k^{\mathrm{o}} & \text{if } t_i = x_k \text{ (variable)} \\ s & \text{if } t_i = s \text{ (string)} \end{cases}$$

```
data Cat  = Cat Int | Vars
data Fun  = Fun String
data Var  = Var String
data Tree = Apply Fun [Tree] | Place Var

type Function = ([Cat],Cat)
type Pattern  = [Either Int String]
type Rule     = (Function,Pattern)
type Grammar  = [(String,Rule)]
```

```
Lin    :: Grammar -> Tree -> String
Lookup :: Grammar -> Fun -> Rule

Lookup L (Fun F) =
   case lookup F L of Just c -> c
                      _  -> error ("unknown item " ++ F)
Lin L (Place (Var x)) = x
Lin L (Apply F X) =
 AP L (snd (Lookup L F)) X
  where
   AP L (Left n : P) X  = Lin L (X !! n) ++ sp (AP L P X)
   AP L (Right s : P) X = s ++ sp (AP L P X)
   AP L [] _ = ""
   sp s = if s=="" then s else " " ++ s
```

Parser combinator for category α in grammar G

$$P_G\alpha = P_1 * F_1 \ \mid \ \ldots \ \mid \ P_n * F_n$$

where $F_1, \ldots\ F_n$ are the formers for α in G. Each F_k has an elementary parser

$$P_k = p_1, \ldots\ p_{m_k}$$

where

$$p_i = \begin{cases} P_G\alpha_j & \text{if } t_i = x_j : \alpha_j \text{ (variable)} \\ \text{lit}(s) & \text{if } t_i = s \text{ (string)} \end{cases}$$

```
PC1 :: Grammar -> Cat -> Parser String Tree
PE1 :: Grammar -> Rule -> Parser String [Tree]

PE1 L (F,P) =
 case P of
   Left  n :K ->
    PC1 L (Ct n) .>. (\x ->  PE1 L (F,K) .>.
                                   (\y -> succeed (x:y)))
   Right s :K ->
    literal s .>. (\x -> PE1 L (F,K) .>. (\y -> succeed y))
   []         -> succeed []
  where Ct n = fst F !! n
PC1 L C = foldl (|||) fails
           [PE1 L ((A,K),P) ***
              (Apply (Fun F)) | (F,((A,K),P)) <- L, K == C]
```

Recursive descent parsers, built on ideas of Burge (1975) by Wadler (1985, 1995) and Hutton (1992).

Basic operations reminiscent of PROLOG, but they can be strengthened by various ways using higher-order functions.

The parser of the previous slide can be improved by techniques for circumventing *left recursion*.

Possible optimizations:

left factorization

localization of checking operations

```
type Parser a b = [a] -> [(b,[a])]

succeed v s   = [(v,s)]
fails s       = []
(p1 ||| p2) s = p1 s ++ p2 s
(p1 .>. p2) s = [(b,z) | (a,y) <- p1 s, (b,z) <- p2 a y]
(p *** f)     = p .>. (\x -> succeed (f x))
literal x l   = case l of
                  []  -> []
                  a:k -> if a == x then [(x,k)] else []
```

To adjust permutations, repetitions, and suppressions in the pattern

$$F : (x_1 : \alpha_1) \cdots (x_n : \alpha_n)\alpha \ - \ t_1 \ \dots \ t_m$$

we define

$$F(c_1, \dots, c_m))^R = F(a_1, \dots, a_n)$$

where

$$a_i = \begin{cases} c_k^R & \text{if } t_k = x_i \text{ and } x_i \text{ is consistently represented} \\ ? & \text{if } x_i \text{ is missing} \end{cases}$$

Notice that adjustment fails if different occurrences of a variable are represented by different expressions.

Each category α has, in a given language, certain *parametres* P_α and certain *inherent features* I_α. Thus

$$\alpha^{\mathrm{o}} \;=\; ((P_\alpha)\mathrm{Str}, I_\alpha).$$

For instance,

$$\mathrm{CN}^{\mathrm{o}} = \begin{cases} (\mathrm{Num})\mathrm{Str} & \text{in English} \\ ((\mathrm{Num})\mathrm{Str}, \mathrm{Gen}) & \text{in French} \\ ((\mathrm{Num}, \mathrm{Cas})\mathrm{Str}, \mathrm{Gen}) & \text{in German} \end{cases}$$

Above, we have assumed uniformly

$$\alpha^{\mathrm{o}} \;=\; \mathrm{Str}.$$

General principle:

$$\frac{F \;:\; (\alpha_1)(\alpha_2)\cdots(\alpha_n)\alpha,}{F^{\mathrm{o}} \;=\; (\alpha_1^{\mathrm{o}})(\alpha_2^{\mathrm{o}})\cdots(\alpha_n^{\mathrm{o}})\alpha^{\mathrm{o}}}.$$

For instance, in French,

$$\mathrm{PredV1}((Q,(g,n)),F)^{\mathrm{o}} = (m)(Q(\mathrm{nom})\ F(g,n,m)).$$

where

$$\begin{aligned}\mathrm{S}^{\mathrm{o}} &= (\mathrm{Mod})\mathrm{Str},\\ \mathrm{NP}^{\mathrm{o}} &= ((\mathrm{Case})\mathrm{Str},(\mathrm{Gen},\mathrm{Num}))\\ \mathrm{V1}^{\mathrm{o}} &= (\mathrm{Gen},\mathrm{Num},\mathrm{Mod})\mathrm{Str}\end{aligned}$$

Function + parametric linear pattern + inherent features

$$F : (x_1 : \alpha_1) \cdots (x_n : \alpha_n)\alpha \; - \; (p_\alpha)(t_1 \; \ldots \; t_m) \; - \; i_1, \ldots, i_{k_\alpha}$$

where

each t_j is either a variable x_p or a string-valued function applied to constant features, parametres p_α, and inherent features of x_p's

each i_j is a morphological feature, function of constant features and inherent features of x_p's

In unification grammar, the rule corresponding to

$$\mathrm{PredV1} : (Q : \mathrm{NP})(F : \mathrm{V1})\mathrm{S} \; - \; (m)(Q(\mathrm{nom})\; F(g,n,m))$$

is the rule

$$\mathrm{S}(m) \;\rightarrow\; \mathrm{NP}(g,n)\; \mathrm{V1}(g,n,m)$$

without distinction between parametres and inherent features. Such rules are weaker and more language-dependent than the format with a function and a linear pattern.

Linearization: just a more complex lookup function.

Parsing (one possibility, easy to implement): reduce to the case without morphology by permitting all forms. Check by comparison with linearization if desired.

Then the parser accepts e.g.

il se peut que tous les femmes sont amoureux de un homme

which it returns in the form

> *il se peut que toutes les femmes soient amoureuses d'un homme*

Without dependent types, type checking reduces to parsing in the above sense.

With dependent types, we have considered the special case of application. There, to check that

$$F(a_1, \ldots, a_n) : \beta$$

we just check that

$$\begin{aligned}
&F : (x_1 : \alpha_1) \cdots (x_n : \alpha_n)\alpha, \\
&a_1 : \beta_1,\ \ldots,\ a_n : \beta_n, \\
&\beta_1 = \alpha_1,\ \ldots,\ \beta_n = \alpha_n(x_1 = a_1, \ldots, x_{n-1} = a_{n-1}), \\
&\beta = \alpha(x_1 = a_1, \ldots, x_n = a_n).
\end{aligned}$$

This can be turned into type inference and localized in parsing.

To type check question marks (suppressed constituents), we must generate *constraint equations* instead of Boolean values (de Bruijn 1991, Magnusson 1994).

Constraints can sometimes be resolved automatically, by unification, but they can also lead to *interaction*, like in *proof editors* (ALF, LEGO, Coq).

We can also strengthen the language, which is now a kind of *combinatoric categorial grammar*, by a mechanism of *variable binding*.

```
Categories SI, S, NP, V1, A1, A2, CN

Parametres Num(n,sg,pl), Mod(m,ind,subj), Cas(c,from,to)

Operations
NomReg(Num)   = _,s ;
be(Num)       = is,are ;
prep(Cas)     = from,to

Ind    : (A:S)SI                              - "A" ;
NegS   : (A:S)S                       - "it is not the case that A" ;
PredV1 : (A:CN)(Q:NP(A))(F:V1(A))S            - "Q F(Num(Q))" ;
PredA1 : (A:CN)(F:A1(A))V1(A)                 - (n)"be(n) F" ;
ComplA2 : (A:CN)(B:CN)(F:A2(A,B))(Q:NP(B))A1(A) - "F prep(Cas(F)) Q" ;
ImplS  : (A:S)(B:S)S                          - "if A then B" ;

Ln     : CN          - (n)"line+NomReg(n)" ;
Pt     : CN          - (n)"point+NomReg(n)" ;
Vert   : A1(Ln)      - "vertical" ;
DiPt   : A2(Pt,Pt)   - "distinct"            - from ;
Par    : A2(Ln,Ln)   - "parallel"            - to ;
Tout   : (A:CN)NP(A) - "all A(pl)"           - sg ;
Un     : (A:CN)NP(A) - "a/an A(sg)"          - sg
```

```
Categories SI, S, NP, V1, A1, A2, CN
Parametres Gen(g,masc,fem), Num(n,sg,pl), Mod(m,ind,subj), Cas(c,de,aa)

Operations
NomReg(Num)        = _,s ;
AdjReg(Gen,Num)    = _,s,e,es ;
AdjE(Gen,Num)      = _,s,_,s ;
AdjAl(Gen,Num)     = l,ux,le,les ;
etre(Num,Mod)      = est,soit,sont,soient ;
tout(Gen,Num)      = tout,tous,toute,toutes ;
prep(Cas)          = de/d',a,avec

Ind    : (A:S)SI - "A(ind)" ;
NegS   : (A:S)S - (m)"il ne/n' etre(m) pas vrai que/qu' A(subj)" ;
PredV1 : (A:CN)(Q:NP(A))(F:V1(A))S - (m)"Q F(Gen(Q),Num(Q),m)" ;
PredA1 : (A:CN)(F:A1(A))V1(A) - (g,n,m)"etre(n,m) F(g,n)" ;
ComplA2 : (A:CN)(B:CN)(F:A2(A,B))(Q:NP(B))A1(A) - (g,n)"F(g,n) prep(Cas(F)) Q" ;
ImplS  : (A:S)(B:S)S - (m)"s'(il,ils)/si A(ind) alors B(m)" ;
Ln     : CN - (n)"droite+NomReg(n)" - fem ;
Pt     : CN - (n)"point+NomReg(n)" - masc ;
Vert   : A1(Ln) - (g,n)"vertica+AdjAl(g,n)" ;
DiPt   : A2(Pt,Pt) - (g,n)"distinct+AdjReg(g,n)" - de ;
Par    : A2(Ln,Ln) - (g,n)"parall\`ele+AdjE(g,n)" - aa ;
Tout   : (A:CN)NP(A) - "tout(Gen(A),sg) A(sg)" - Gen(A),sg ;
Un     : (A:CN)NP(A) - "un+AdjReg(Gen(A),sg) A(sg)" - Gen(A),sg
```