Computational Semantics, Type Theory, and Functional Programming

APPENDIX — The Functional Approach To Parsing

Jan van Eijck

CWI and ILLC, Amsterdam, Uil-OTS, Utrecht

LOLA7 Tutorial, Pecs

August 2002
Summary

• Categories
• Features, Feature Agreement, Feature Percolation
• Lexical Items
• Parse Rules for Categories
• Putting it all together
A Module for Syntactic Categories

No index information on NPs, except for pronouns. Otherwise, virtually the same as a datatype declaration for a fragment of dynamic Montague grammar. The module Cat imports the standard List module. Lists will be employed to implement a simple feature agreement mechanism.

```haskell
module Cat

where

import List
```
Define features, feature lists, indices, and numerals.

```haskell
data Feature = Masc | Fem | Neutr | Sg | Pl | Fst | Snd | Thrd | Nom | Acc
  deriving (Eq,Ord)
```
instance Show Feature
    where
    show Masc = "M"
    show Fem = "F"
    show Neutr = "N"
    show Sg = "Sg"
    show Pl = "Pl"
    show Fst = "1"
    show Snd = "2"
    show Thrd = "3"
    show Nom = "n"
    show Acc = "a"
type Agreement = [Feature]
type Idx = Int
type Numeral = Int
Selecting the gender, number, person and case part of a feature list:

```haskell
gen, nr, ps, cs :: Agreement -> Agreement
gen = filter (\x -> x == Masc || x == Fem || x == Neutr)
nr = filter (\x -> x == Sg || x == Pl)
ps = filter (\x -> x == Fst || x == Snd || x == Thrd)
cs = filter (\x -> x == Nom || x == Acc)
```
Declare a class `Cat` for categories that carry number and gender information, with a function `fs` that gives the feature list of a category, functions `gender`, `number`, `sperson` and `scase` for the syntactic gender, number, person and case features of the category, a function `combine` that computes the features of a combined category (with feature clashes reported by `[]`), and a function `agree` indicating whether there is a feature clash or not when two categories are combined.

```haskell
class (Eq a, Show a) => Cat a where
  fs :: a -> Agreement
  gender, number, sperson, scase :: a -> Agreement
  gender cat = gen (fs cat)
  number cat = nr (fs cat)
  sperson cat = ps (fs cat)
  scase cat = cs (fs cat)
```
combine :: Cat b => a -> b -> [Agreement]
combine cat1 cat2 =
    [ feats | length (gen feats) <= 1,
             length (nr feats) <= 1,
             length (ps feats) <= 1,
             length (cs feats) <= 1 ]
where
    feats = (nub . sort) (fs cat1 ++ fs cat2)
agree :: Cat b => a -> b -> Bool
agree cat1 cat2 = not (null (combine cat1 cat2))
Sentences are in class `Cat`. Set the agreement lists of ‘if then’ sentences and texts consisting of several sentences to `[]`.

```haskell
data S = S NP VP | If S S | Txt S S
    deriving (Eq,Show)

instance Cat S
    where
        fs (S np vp) = fs vp
        fs _ = []
```
Pronouns and complex NPs carry explicit feature information. The feature information of proper names depends on the name.

```
data NP = Ann | Mary | Bill | Johnny
    | PERS Agreement
    | PRO Agreement Idx
    | NP1 Agreement DET CN
    | NP2 Agreement DET RCN

deriving (Eq,Show)
```
instance Cat NP

where

fs Ann        = [Fem,Sg,Thrd]
fs Mary       = [Fem,Sg,Thrd]
fs Bill       = [Masc,Sg,Thrd]
fs Johnny     = [Masc,Sg,Thrd]
fs (PERS ftrs) = ftrs
fs (PRO ftrs i) = ftrs
fs (NP1 ftrs det cn) = ftrs
fs (NP2 ftrs det rcn) = ftrs
The entries ALL, SOME, NO, THE are for both singular and plural determiners, so they carry no number feature information. The entries LESS and MOST are for plural determiners. The number feature of MORE and EXACT depends on the numeral.

\[
\text{data DET = ALL Agreement \mid SOME Agreement} \\
\mid \text{NO Agreement \mid THE Agreement} \\
\mid \text{LESS Numeral \mid MORE Numeral} \\
\mid \text{EXACT Numeral \mid MOST}
\]

deriving (Eq, Show)
We only set the number feature.

```
instance Cat DET
    where
    fs (ALL ftrs) = ftrs
    fs (SOME ftrs) = ftrs
    fs (NO ftrs) = ftrs
    fs (THE ftrs) = ftrs
    fs (LESS i) = [Pl]
    fs (MORE 1) = [Sg]
    fs (MORE i) = [Pl]
    fs (EXACT 1) = [Sg]
    fs (EXACT i) = [Pl]
    fs MOST = [Pl]
```
We make a syntactic distinction between singular and plural versions of CNs and RCNs, although their semantic treatment will be the same.

```haskell
data CN = Man Agreement | Woman Agreement
        | Boy Agreement  | Person Agreement
        | Thing Agreement | House Agreement
        | Cat Agreement  | Mouse Agreement
        | ACN ADJ CN

deriving (Eq,Show)
```
instance Cat CN

  where

    fs (Man ftrs) = ftrs
    fs (Woman ftrs) = ftrs
    fs (Boy ftrs) = ftrs
    fs (Person ftrs) = ftrs
    fs (Thing ftrs) = ftrs
    fs (House ftrs) = ftrs
    fs (Cat ftrs) = ftrs
    fs (Mouse ftrs) = ftrs
    fs (ACN adj cn) = fs cn
data ADJ = Old | Young | Other
    deriving (Eq, Show)

instance Cat ADJ
    where
      fs Old = []
      fs Young = []
data RCN = CN1 CN VP | CN2 CN NP TV
    deriving (Eq,Show)

instance Cat RCN
    where
        fs (CN1 cn vp)   = fs cn
        fs (CN2 cn np tv) = fs cn
We make a syntactic distinction between singular and plural versions of VPs and TVs, although their semantic treatment will be the same.

```haskell
data VP = Laugh Agreement | Cry Agreement
    | Curse Agreement
    | Smile Agreement
    | VP1 TV NP | VP2 Agreement TV REFL

deriving (Eq,Show)
```
instance Cat VP
  where
  fs (Laugh ftrs) = ftrs
  fs (Cry ftrs) = ftrs
  fs (Curse ftrs) = ftrs
  fs (Smile ftrs) = ftrs
  fs (VP1 tv np) = fs tv
  fs (VP2 ftrs tv refl) = ftrs
data REFL = Self Agreement deriving (Eq, Show)

instance Cat REFL
    where fs (Self ftrs) = ftrs
Transitive verbs carry a number feature, so they are in the class \texttt{Cat}.

\begin{verbatim}
data TV = Love Agreement | Respect Agreement  
        | Hate Agreement | Own Agreement
 deriving (Eq,Show)

instance Cat TV
 where
  fs (Love ftrs)  = ftrs
  fs (Respect ftrs) = ftrs
  fs (Hate ftrs)  = ftrs
  fs (Own ftrs)   = ftrs
\end{verbatim}
A Simple CF Parser

module Parser
where
import Cat

type Words = [String]
NPs

\[\text{lexNP} :: \text{Words} \rightarrow \{(\text{NP}, \text{Words})\}\]

\[\text{lexNP} ("ann":xs) = \{(Ann, xs)\}\]

\[\text{lexNP} ("mary":xs) = \{(Mary, xs)\}\]

\[\text{lexNP} ("bill":xs) = \{(Bill, xs)\}\]

\[\text{lexNP} ("johnny":xs) = \{(Johnny, xs)\}\]

\[\text{lexNP} ("i":xs) = \{(\text{PERS} \{Sg, Fst, Nom\}, xs)\}\]

\[\text{lexNP} ("me":xs) = \{(\text{PERS} \{Sg, Fst, Acc\}, xs)\}\]

\[\text{lexNP} ("we":xs) = \{(\text{PERS} \{Pl, Fst, Nom\}, xs)\}\]

\[\text{lexNP} ("us":xs) = \{(\text{PERS} \{Pl, Fst, Acc\}, xs)\}\]

\[\text{lexNP} ("you":xs) = \{(\text{PERS} \{Snd\}, xs)\}\]
lexNP ("he":x:xs) =
    [((PRO [Masc,Sg,Thrd,Nom] (read x)),xs)]
lexNP ("him":x:xs) =
    [((PRO [Masc,Sg,Thrd,Acc] (read x)),xs)]
lexNP ("she":x:xs) =
    [((PRO [Fem,Sg,Thrd,Nom] (read x)),xs)]
lexNP ("her":x:xs) =
    [((PRO [Fem,Sg,Thrd,Acc] (read x)),xs)]
lexNP ("it":x:xs) =
    [((PRO [Neutr,Sg,Thrd] (read x)),xs)]
lexNP ("they":x:xs) =
    [((PRO [Pl,Thrd,Nom] (read x)),xs)]
lexNP ("them":x:xs) =
    [((PRO [Pl,Thrd,Acc] (read x)),xs)]
lexNP _ = []
parseNP :: Words -> [(NP,Words)]
parseNP = \xs ->
  [ (NP1 agr det cn,zs) |
    (det,ys) <- parseDET xs,
    (cn, zs) <- parseCN ys,
    agr <- combine det cn ]
++
  [ (NP2 agr det rcn,zs) |
    (det,ys) <- parseDET xs,
    (rcn, zs) <- parseRCN ys,
    agr <- combine det rcn ]
++
  [ (np,ys) | (np,ys) <- lexNP xs ]
Determiners

Note that we need a distinction in the lexicon between singular and plural *some*, *no* and *the*, because of the semantic distinction.
lexDET :: Words -> [(DET,Words)]
lexDET ("every":xs)  = [(ALL [Sg], xs)]
lexDET ("all":xs)    = [(ALL [Pl], xs)]
lexDET ("some":xs)   =
                      [(SOME [Sg], xs),(SOME [Pl], xs)]
lexDET ("no":xs)     =
                      [(NO [Sg], xs), (NO [Pl], xs)]
lexDET ("the":xs)    =
                      [(THE [Sg], xs),(THE [Pl], xs)]
lexDET ("less":"than":x:xs) = [((LESS (read x)), xs)]
lexDET ("more":"than":x:xs) = [((MORE (read x)), xs)]
lexDET ("exactly":x:xs)  = [((EXACT (read x)), xs)]
lexDET ("most":xs)     = [(MOST, xs)]
lexDET _                = []
parseDET :: Words -> [(DET, Words)]
parseDET = lexDET
ADJs

**lexADJ** :: Words -> [(ADJ,Words)]
lexADJ ("old":xs) = [(Old,xs)]
lexADJ ("young":xs) = [(Young,xs)]
lexADJ ("other":xs) = [(Other,xs)]
lexADJ _ = []

**parseADJ** :: Words -> [(ADJ,Words)]
parseADJ = lexADJ
CNs

Singular and plural CNs get distinguished by means of an appropriate number feature.
lexCN :: Words -> [(CN,Words)]
lexCN ("man":xs) = [(Man [Masc,Sg,Thrd],xs)]
lexCN ("men":xs) = [(Man [Masc,Pl,Thrd],xs)]
lexCN ("woman":xs) = [(Woman [Fem,Sg,Thrd],xs)]
lexCN ("women":xs) = [(Woman [Fem,Pl,Thrd],xs)]
lexCN ("boy":xs) = [(Boy [Masc,Sg,Thrd],xs)]
lexCN ("boys":xs) = [(Boy [Masc,Pl,Thrd],xs)]
lexCN ("person":xs) = [(Person [Sg,Thrd],xs)]
lexCN ("persons":xs) = [(Person [Pl,Thrd],xs)]
lexCN ("thing":xs) = [(Thing [Neutr,Sg,Thrd],xs)]
lexCN ("things":xs) = [(Thing [Neutr,Pl,Thrd],xs)]
lexCN ("house":xs) = [(House [Neutr,Sg,Thrd],xs)]
lexCN ("houses":xs) = [(House [Neutr,Pl,Thrd],xs)]
lexCN ("cat":xs) = [(Cat [Neutr,Sg,Thrd],xs)]
lexCN ("cats":xs) = [(Cat [Neutr,Pl,Thrd],xs)]
lexCN ("mouse":xs) = [(Mouse [Neutr,Sg,Thrd],xs)]
lexCN ("mice":xs) = [(Mouse [Neutr,Pl,Thrd],xs)]
lexCN _ = []
parseCN :: Words -> [(CN,Words)]
parseCN = \xs ->
    [ (cn,ys)| (cn,ys) <- lexCN xs ]
++
    [ (ACN adj cn, zs) | (adj,ys) <- parseADJ xs,
                      (cn, zs) <- parseCN ys ]
parseTHAT :: Words -> [Words]
parseTHAT ("that":xs) = [xs]
parseTHAT _ = []
parseRCN :: Words -> [(RCN,Words)]
parseRCN = \xs ->
  [(CN1 cn vp, us) |
    (cn,ys) <- parseCN xs,
    zs <- parseTHAT ys,
    (vp,us) <- parseVP zs,
    agree cn vp]
++
  [(CN2 cn np tv, vs) |
    (cn,ys) <- parseCN xs,
    zs <- parseTHAT ys,
    (np,us) <- parseNP zs,
    (tv,vs) <- parseTV us,
    agree np tv,
    notElem Acc (fs np) ]
parseREFL :: Words -> [(REFL,Words)]
parseREFL ("myself":xs) = [(Self [Sg,Fst], xs)]
parseREFL ("ourselves":xs) = [(Self [Pl,Fst], xs)]
parseREFL ("yourself":xs) = [(Self [Sg,Snd], xs)]
parseREFL ("yourselves":xs) = [(Self [Pl,Snd], xs)]
parseREFL ("himself":xs) = [(Self [Masc,Sg,Thrd], xs)]
parseREFL ("herself":xs) = [(Self [Fem,Sg,Thrd], xs)]
parseREFL ("itself":xs) = [(Self [Neutr,Sg,Thrd], xs)]
parseREFL ("themselves":xs) = [(Self [Pl,Thrd], xs)]
parseREFL _ = []
VPs

\[
\text{lexVP} :: \text{Words} \rightarrow [\text{(VP,Words)}]
\]

\[
\text{lexVP} ("laughs":xs) = [(\text{Laugh} [\text{Sg, Thrd}],xs)]
\]

\[
\text{lexVP} ("laugh":xs) = [(\text{Laugh} [\text{Sg, Fst}],xs),
\hspace{1cm} (\text{Laugh} [\text{Sg, Snd}],xs),(\text{Laugh} [\text{Pl}],xs)]
\]

\[
\text{lexVP} ("cries":xs) = [(\text{Cry} [\text{Sg, Thrd}],xs)]
\]

\[
\text{lexVP} ("cry":xs) = [(\text{Cry} [\text{Sg, Fst}],xs),
\hspace{1cm} (\text{Cry} [\text{Sg, Snd}],xs),(\text{Cry} [\text{Pl}],xs)]
\]
lexVP ("curses" : xs) = [(Curse [Sg, Thrd], xs)]
lexVP ("curse" : xs) = [(Curse [Sg, Fst], xs),
(Curse [Sg, Snd], xs), (Curse [Pl], xs)]
lexVP ("smiles" : xs) = [(Smile [Sg, Thrd], xs)]
lexVP ("smile" : xs) = [(Smile [Sg, Fst], xs),
(Smile [Sg, Snd], xs), (Smile [Pl], xs)]
lexVP _ = []
parseVP :: Words -> [(VP,Words)]
parseVP = \xs ->
  [ (VP1 tv np,zs) |
    (tv,ys) <- parseTV xs,
    (np,zs) <- parseNP ys,
    notElem Nom (fs np) ]
++
  [ (VP2 agr tv refl,zs) |
    (tv,ys) <- parseTV xs,
    (refl,zs) <- parseREFL ys,
    agr <- combine tv refl ]
++
  [ (vp,ys)| (vp,ys) <- lexVP xs ]
TVs

```haskell
lexTV :: Words -> [(TV,Words)]
lexTV ("loves":xs)   = [(Love [Sg,Thrd],xs)]
lexTV ("love":xs)    = [(Love [Sg,Fst],xs),
                       (Love [Sg,Snd],xs), (Love [Pl],xs)]
lexTV ("respects":xs) = [(Respect [Sg,Thrd],xs)]
lexTV ("respect":xs) = [(Respect [Sg,Fst],xs),
                       (Respect [Sg,Snd],xs), (Respect [Pl],xs)]
lexTV ("hates":xs)   = [(Hate [Sg,Thrd],xs)]
lexTV ("hate":xs)    = [(Hate [Sg,Fst],xs),
                       (Hate [Sg,Snd],xs), (Hate [Pl],xs)]
lexTV ("owns":xs)    = [(Own [Sg,Thrd],xs)]
lexTV ("own":xs)     = [(Own [Sg,Fst],xs),
                       (Own [Sg,Snd],xs), (Own [Pl],xs)]
lexTV _               = []
```
parseTV :: Words -> [(TV,Words)]
parseTV = \xs ->
      [ (tv,ys) | (tv,ys) <- lexTV xs ]
IF, THEN, ‘.’, ‘;’

parseIF :: Words -> [Words]
parseIF ("if":xs) = [xs]
parseIF _ = []

parseTHEN :: Words -> [Words]
parseTHEN ("then":xs) = [xs]
parseTHEN _ = []

parseC :: Words -> [Words]
parseC (".":xs) = [xs]
parseC (;"":xs) = [xs]
parseC _ = []
\[ \text{parseS} :: \text{Words} \rightarrow [(S,\text{Words})] \]
\[
\text{parseS} = \lambda \text{xs} \rightarrow \\
\quad [(S \text{ np vp},zs) \mid (\text{np},ys) \leftarrow \text{parseNP } \text{xs}, \\
\quad (\text{vp},zs) \leftarrow \text{parseVP } \text{ys}, \\
\quad \text{agree np vp,} \\
\quad \text{notElem Acc (fs np)} ] \\
\quad ++ \\
\quad [(\text{If s1 s2},vs) \mid ys \leftarrow \text{parseIF } \text{xs}, \\
\quad (s1,zs) \leftarrow \text{parseS } \text{ys}, \\
\quad us \leftarrow \text{parseTHEN } zs, \\
\quad (s2,vs) \leftarrow \text{parseS } us ]
\]
Since the rule $T ::= S \mid T.S$ is left-recursive, we need an extra function for splitting the input word list: `split` gives all the ways to split a list of at least two elements in two non-empty parts.

\[
\text{split} :: [a] \rightarrow \{([a],[a])\}
\]

\[
\begin{aligned}
\text{split } [x,y] &= \{([x],[y])\} \\
\text{split } (x:y:zs) &= \\
&\quad ([x],(y:zs)):\left(\text{map } \lambda (us,vs) \rightarrow ((x:us),vs)\right) \\
&\quad \quad (\text{split } (y:zs))
\end{aligned}
\]
parseTxt :: Words -> [(S,Words)]
parseTxt = \xs ->
    parseS xs
++
    [ (Txt t s,vs) | (ys,zs) <- split xs,
        us <- parseC zs,
        (t,[]) <- parseTxt ys,
        (s,vs) <- parseS us ]
The ‘scan’ function

The next function scans an input string and puts whitespace in front of punctuation marks and numerals. This can be used to convert a string like “He1 loves her2.” to “He 1 loves her 2.”

```haskell
scan :: String -> String
scan [] = []
scan (x:xs) | x == '.' || x == ';'; =
            |   ' ':x:scan xs
            | isDigit x =
            |   ' ':x: (digits ++ scan rest)
            | otherwise =
            |   x:scan xs

where (digits,rest) = span isDigit xs
```
Parse

The main parse function uses the predefined function `words` to split the input into separate words. Punctuation marks and pronoun indices should come out as separate words; we use `scan` for that. Also, for robustness, we convert everything to lowercase.

parse :: String -> [S]
parse string = [ s | (s,["."]) <- parseTxt
  (words
   (map toLower (scan string))) ]
Now try it out:

Parser> parse "Every man loves some woman."
[S (NP1 [M,Sg,3] (ALL [Sg]) (Man [M,Sg,3])))
   (VP1 (Love [Sg,3]) (NP1 [F,Sg,3]
      (SOME [Sg]) (Woman [F,Sg,3])])]

Parser> parse "All men love some woman."
[S (NP1 [M,Pl,3] (ALL [Pl]) (Man [M,Pl,3])))
   (VP1 (Love [Pl]) (NP1 [F,Sg,3]
      (SOME [Sg]) (Woman [F,Sg,3])])]

Parser> parse "All men love some women."
[S (NP1 [M,Pl,3] (ALL [Pl]) (Man [M,Pl,3])))
   (VP1 (Love [Pl]) (NP1 [F,Pl,3]
      (SOME [Pl]) (Woman [F,Pl,3])])]

Parser> parse "Bill loves more than 1 woman."
[S Bill (VP1 (Love [Sg,3])
 (NP1 [F,Sg,3] (MORE 1) (Woman [F,Sg,3]))))]

Parser> parse "Bill loves more than 1 women."
[]

Parser> parse "Bill loves more than 1 woman. He0 respects them1."
[Txt (S Bill (VP1 (Love [Sg,3])
 (NP1 [F,Sg,3] (MORE 1) (Woman [F,Sg,3]))))
 (S (PRO [M,Sg,3,n] 0)
 (VP1 (Respect [Sg,3]) (PRO [P1,3,a] 1)))]
Examples with personal pronouns:

pp1 = "I love you."
pp2 = "We respect ourselves."
pp3 = "We respect every woman that respects herself."
pp4 = "You respect yourself."
pp5 = "You respect yourselves."
Examples with singular NPs:

ex1  = "Johnny smiles."
ex2  = "Bill laughs."
ex3  = "if Bill laughs then Johnny smiles."
ex4  = "Bill laughs. Johnny smiles. Mary laughs."
ex5  = "Bill smiles. He1 loves some woman."
ex6  = "The boy loves some woman."
ex7  = "Some man loves some woman that smiles."
ex8  = "Some man respects some woman."
ex9  = "The man loves some woman."
ex10 = "Every man loves some woman."
ex11 = "Every man loves Johnny."
ex12 = "Some woman loves Johnny."
ex13 = "Johnny loves some woman."
ex14 = "Johnny respects some man that loves Mary."
ex15 = "No woman loves Bill."
ex16 = "No woman that hates Johnny loves Bill."
ex17 = "Some woman that respects Johnny loves Bill."
ex18 = "The boy loves Johnny."
ex19 = "He2 loves her1."
ex20 = "He2 respects her1."
ex21 = "If some man loves some woman then he4 respects her5." 
ex22 = "Some man loves some woman. He4 respects her5." 
ex23 = "Some woman owns some thing." 
ex24 = "Some woman owns the house." 
ex25 = "Some woman owns the house that Johnny hates." 
ex26 = "No man that cries respects himself." 
ex27 = "Some man respects himself." 
ex28 = "Exactly 1 boy curses."
Examples with plurals:

px1 = "More than 1 man laughs."
px2 = "More than 2 men laugh."
px3 = "Most men that love some woman smile."
px4 = "Some women cry. No men cry."
px5 = "No men that cry respect themselves."
px6 = "All men cry."
px7 = "The men curse. The women laugh."
px8 = "Most men curse. No women curse."
px9 = "Most men smile. They4 laugh."
px10 = "Most men cry. They4 laugh."
px11 = "More than 1 man laughs. They4 love Mary."
px12 = "Less than 4 men laugh."
px13 = "Less than 4 men laugh. They4 love Mary."