Lecture 18: Building Parse Trees

CS 181O
Spring 2016
Kim Bruce

Last Time

- Saw how to recognize language
  - Parser returned string of everything recognized, paired with the remaining input that wasn't used

```
pS  = pNP <*> pVP
pNP = symbol "Alice"  <|> symbol "Dorothy" <|> (pD <*> pN)
pVP = symbol "smiled" <|> symbol "laughed"
pD  = symbol "every"  <|> symbol "some"   <|> symbol "no"
pN  = symbol "dwarf"  <|> symbol "wizard"

*P> pNP ["every","dwarf","laughed"]
[("everydwarf",["laughed"])]

*P> pS ["every","dwarf","laughed"]
[("everydwarflaughed",[])]
```

Building Parse Tree

- Instead want to return parse tree (or AST)

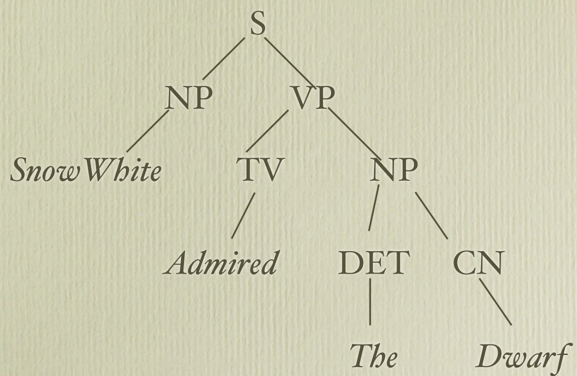
```
-- f<$>p returns a parser that behaves like p, but transforms the
-- first argument of each pair returned by applying f to it.
(<$>) :: (a -> b) -> Parser s a -> Parser s b
(f <$> p) xs = [ (f x,ys) | (x,ys) <- p xs ]

digitize = f <$> digit    — digit is parser recognizing digits
  where f c = ord c - ord '0'

*P> digitize "57a"
[(5,"7a")]
```

Strategy

- Modify each parser to return part of parse tree with appropriate label as branch

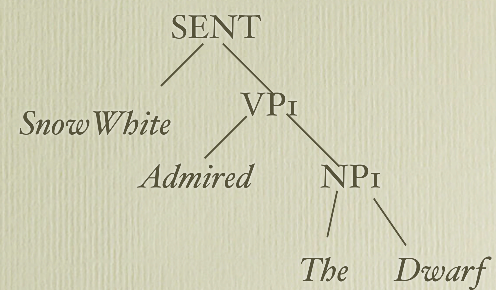
```
data ParseTree a b =  Ep | Leaf a | Branch b [ParseTree a b]
              deriving Eq

type PARSER a b = Parser a (ParseTree a b)
```

ParseTree



Abstract SyntaxTree

Is generally simpler, but preserves structure



SENT *SnowWhite* (VP1 *Admired* (NP1 *The Dwarf*))

Preserved subtree structure!

Strategy

- Build parse tree, then apply function to get AST (or, equivalently, term in Haskell)

Parse Trees

- A parse tree is either empty, or a leaf, or a branching node with information on its subtrees. (*Nodes and leaves can hold different info*)
- data ParseTree a b = Ep | Leaf a |
  Branch b [ParseTree a b]
  deriving Eq

Parse Trees

Leaf info type *Branch info type*

data Category = S | NP | VP | DET | N | V | ADJ

tree :: ParseTree String Category

tree = Branch S [Branch NP [Leaf "SnowWhite"],

Branch VP [Branch TV [Leaf "admired"],

Branch NP

[Branch DET [Leaf "The"],

Branch N [Leaf "Dwarf"]]]]

Parsing

- In P2.hs defined
  - sent, np, vp, det, cn :: PPARSER
  - where PPARSER = PARSER String Category
    = Parser String (ParseTree String Category)
    = [String] →[(ParseTree String Category, [String])
  - Applying sent to list of words results in list of pairs of parse trees and remaining words of input.
- Want to take successful parses and write ADT
  - e.g., element of type Sent

ParseTree ⇒ Sent

- See my file TreeToSyntax.hs in sample programs.
  - stringToNP :: String → NP,
    stringToVP :: String -> VP,
    ...
    converts words to primitives of appropriate type
  - treeToSent :: ParseTree String Category -> Sent
    treeToNP :: ParseTree String Category -> NP
    ...
    converts parse tree to Haskell rep of phrase

String → Sent

- Convert from input string to list of terms of type Sent, corresponding to different parses
  - Function pts takes input and returns list of its parse trees.
  - Function sentences takes input and returns list of elements of type Sent corresponding to parses.
  - main program allows interactive input to translate sentences
  - Leave to you (on next homework) to extend to full language (with adjectives!)
  - *Alternatively could translate to predicate logic.*

Features and Categories

Features

- So far have ignored complexities due to features, e.g., gender, number, person, case, tense, ...
- Can add features to cfg to require agreement

Modifying Grammar

- Replace S → NP VP by
  - $S_{\varnothing} \rightarrow NP_{\{Sg\}}\ VP_{\varnothing}$
  - $S_{\varnothing} \rightarrow NP_{\{Sg\}}\ VP_{\{Sg\}}$
  - $S_{\varnothing} \rightarrow NP_{\{Pl\}}\ VP_{\varnothing}$
  - $S_{\varnothing} \rightarrow NP_{\{Pl\}}\ VP_{\{Pl\}}$
- If start w/cfg, then end with cfg

Features

- Should group them, but simpler to include all in same type.

```
data Feat = Masc  | Fem  | Neutr | MascOrFem   — gender
          | Sg    | Pl                          — number
          | Fst   | Snd  | Thrd                 — person
          | Nom   | AccOrDat                    — case
          | Pers  | Refl | Wh                   — pronoun type
          | Tense | Infl                        — tense
          | On    | With | By | To | From       — prep type
          deriving (Eq,Show,Ord)

type Agreement = [Feature]
```

Functions

- gender, number, person, ... check for kind of feature
- prune function eliminates redundancy
  - Want at most one feature in each category
  - Function combine lets add features together as long as at most one in each group in final.

Category

- List of features associated with a lexical item
  - data Cat = Cat Phon CatLabel Agreement [Cat] deriving Eq
  - type Phon = String — string representing word
  - type CatLabel = String — part of speech
  - Agreement is list of features
  - Last arg is subcategorization list
    - list of items can be combined with. E.g., transitive verb needs np with feature AccOrDat, ditransitive also needs prep phrase with To feature.

Imposing Roles

- Syntactic rules impose features on components when recognized.
  - E.g., S → NP VP, imposes Nom on NP
  - Function assign :: Feat Cat [Cat]
  - assign f oldCat tries to add feature f to oldCat.
    - If compatible gives list with that new category
    - If not compatible gives empty list

Lexicon

- lexicon :: String →[Cat]
  - Associates words with the possible categorizations for them.
  - Look through definitions in text & P.hs
  - Esp, see pronouns, determiners (all vs every), verbs (esp subcategorization lists)