Lecture 17: Parsing

CS 181O
Spring 2016
Kim Bruce

Parsing Problem

- Given a grammar G and a string s, the parsing problem answers the question whether or not s ∈ L(G). If s ∈ L(G), the answer to this question may include either a parse tree or a derivation.

Parse Tree

- A parse tree for a grammar G is a tree where
  - the root is the start symbol for G
  - the interior nodes are the nonterminals of G and the children of a node N correspond to the symbols on the right hand side of some production rule for T in G
  - the leaf nodes are the terminal symbols of G
- Every string generated by a grammar has a corresponding parse tree that illustrates a derivation for that string.

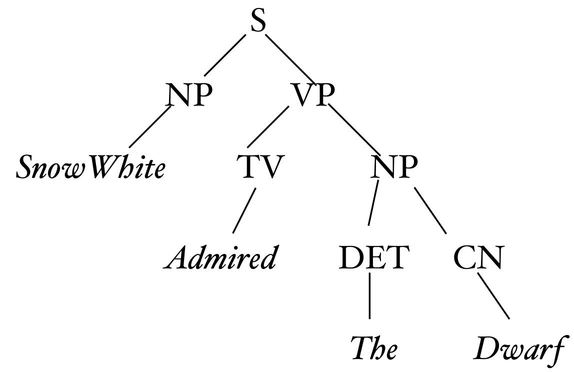
A Fragment of English

- S → NP VP
- NP → Snow White | Alice | Dorothy | Goldilocks | DET CN | DET RCN
- DET → the | every | some | no
- CN → girl | boy | princess | dwarf | giant | sword | dagger
- RCN → CN that VP | CN that NP TV
- VP → laughed | cheered | shuddered | TV NP | DV NP NP
- TV → loved | admired | helped | defeated | caught
- DV → gave

Derivation

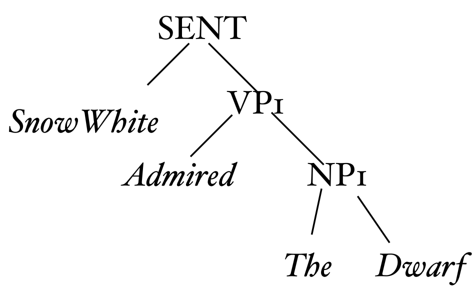
- S $\Rightarrow$ NP VP $\Rightarrow$ Snow White VP
  $\Rightarrow$ Snow White TV NP
  $\Rightarrow$ Snow White admired NP
  $\Rightarrow$ Snow White admired DET CN
  $\Rightarrow$ Snow White admired the CN
  $\Rightarrow$ Snow White admired the dwarf
- Parse tree on next slide

Parse Tree



Abstract Syntax Tree

Is generally simpler, but preserves structure



SENT *Snow White* (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"]]]]
```

Showing tree

```
instance (Show a, Show b) => Show (ParseTree a b) where
  show Ep          = "[]"
  show (Leaf t)      = show t
  show (Branch l ts) = "[." ++ show l  ++ " "
                     ++ show ts ++ "]"
```

Parsing

- Want function parse:: String → [ParseTree a b]
  - If result empty, then failed
  - If more than one, then ambiguous.
  - Generally hope for singleton list
- Problem: Stuff left over (not used)!
  - When looking for S, first look for NP, then VP
  - After finding NP, will be some input left over

Parser

- type Parser a b = [a] → [(b,[a])]
  - where a is type of input, b is type of parse tree
- For us, input is a list of strings (tokens), while b is ParseTree String Category.
  - Want parser:: Parser String (ParseTree String Category)
  - Equiv to [String] → [(ParseTree String Category,[String])]
  - type PARSER a b = Parser a (parseTree a b)
    use PARSER as an abbreviation
    E.g., PARSER String Category

Example

Leaving out all Branch, Leaf tags:

[“All”, “Girls”, “Laugh”] ⇒

[(DET “All”), [“Girls”, “Laugh”])] ⇒

[(NP [(DET “All”),(CN “Girls”)], [“Laugh”])] ⇒

[(S [NP [(DET “All”),(CN “Girls”)],

VP [“Laugh”]], [])]

Parser Combinators

- Functions that combine parsers into a new parser, or transform a parser into a different parser.
- Start with parsers that recognize simple languages and then build up more complex.

Parsing Context Free Languages in P.hs

Start on line 151

Warning

- New Haskell prelude includes definitions of <*> and <$>, which are also defined in P.hs.
- To eliminate conflicts MUST include:
  import Prelude hiding ((<*>),(<$>))
  at top of every file using those symbols!!

Input to parser

- Assume tokenizer has reduced input to list of strings:
  "Hello there Joe" ⇒ ["Hello", "there", "Joe"]
- Parsers will have type Parser String String *for now! I.e., not yet getting parse tree!*
  - *We will get there!*

Parser Combinators

- Simplest parsers: succeed, fails
- Recognize character: `symbol c input` looks to see if first item of input is c
  - E.g., symbol "Alice" "AliceSally" ⇒ [("Alice",["Sally"])]
  - symbol "Alice" "DorothySally" returns []
- Recognize String: `token cs input` looks to see if first items in input match with cs

More Combinators

- (p1 <|> p2) input returns (p1 input) ++ (p2 input)
  I.e., return list of all parses with p1 or p2
- (p1 <*> p2) input returns
  [(r1++r2,rest) | (r1, rest') <- p1 input,
  (r2, rest) <- p2 rest']

Examples

- Let:
  - p1 = (symbol "Alice" <|> symbol "Dorothy")
  - p2 = p1 <*> (symbol “Sally”)
- Then
  - p1 “AliceSally” = [(“Alice”,”Sally”)], p1 “MaryAnn” = []
    p1 “DorothySally” = [(“Dorothy”,”Sally”)]
  - p2 “AliceSallyMary” = [(“AliceSally”,Mary)]

Define Parser for English

```
pS, pNP, pVP, pD, pN :: Parser String String

pS  = pNP <*> pVP
pNP = symbol "Alice"  <|> symbol "Dorothy" <|>
          symbol "SnowWhite" <|> symbol "Goldilocks" <|>
          symbol "LittleMook" <|> symbol "Atreyu" <|>
          (pD <*> pN)
pVP = symbol "cheered" <|> symbol "laughed" <|> symbol "shuddered"
pD  = symbol "every"  <|> symbol "some"    <|> symbol "no"
pN  = symbol "dwarf"  <|> symbol "wizard"
```

Examples

- pS [“every”,"dwarf","cheered"] ⇒
  [(“everydwarfcheered",[])]
- But we want a parse tree!!

Questions?