Implementing Minimalism
(Computationally)

Amanda Payne
May 9 2012
What is Minimalism?

- A syntactic framework meant to emphasize economy

- Developed around features which motivate movement, selection, etc.

  - Categorial feature + Selection feature
  - \( n + =n \)
  - ointment + the

  - Merge to form “the ointment”. 
Implementation

- Based on the description in Ch. 2 of Kobele 2006

- Data type:
  \[ \text{TREE} = \{\text{Spec, Head, Comp, [features]}\} \star [\text{moving sub-constituents}] \]

- For a single lexical item:
  \[ \text{TREE} = \{\emptyset, \text{word, } \emptyset, [\text{features}]\} \star [] \]
(Non-exhaustive) list of features

- Categorial
- Selectional
- Raising
- Lowering
- Licensee
- Licensor
**Merge & Move**

- **What is Merge?**
  - A function taking two constituents and merging them into one.

- **What is Move?**
  - A function moving a constituent to another position in the tree structure, by a feature that triggers either raising of the head of its complement, or lowering itself onto the head of its complement.
Ex:

(*MERGEs two trees (or lexemes) into one, based on specific feature types *)

let merge tree1 tree2 =
    let feat1 = List.hd (fst tree1).feat in
    let feat2 = List.hd (fst tree2).feat in
    match (feat1, feat2) with
        (Categ(categ), Select(select)) when categ=select -> merge_1_select tree1 tree2
        | (Select(select), Categ(categ)) when categ=select -> merge_1_select tree2 tree1
        | (Lower(lower), Categ(categ)) when lower=categ -> merge_2_lower tree2 tree1
        | (Categ(categ), Lower(lower)) when lower=categ -> merge_2_lower tree1 tree2
        | (Raise(éléver), Categ(categ)) when éllever=categ -> merge_3_raise tree2 tree1
        | (Categ(categ), Raise(éléver)) when éllever=categ -> merge_3_raise tree1 tree2
        | _ -> raise (Red_flag "MERGE error!");;
Ex:

```plaintext
# the;
- : Lexicon.elt =
  [{spec = ""; head = "the"; comp = "";
    feat = [Select "n"; Categ "d"; Licensee "k"; Licensee "q"]},
   □}
# cat;
- : Lexicon.elt =
  [{spec = ""; head = "cat"; comp = ""; feat = [Categ "n"]}, □)
# merge the cat;
- : tree * tree list =
  [{spec = ""; head = "the"; comp = " cat ";
    feat = [Categ "d"; Licensee "k"; Licensee "q"]},
   □)
```

```
Ex:

(*MOVE: first ensure that move is licensed by a existing licensor/licensee feature.
If so, delete this feature and perform the move-- *)

let move tree =
    let tree_left = fst tree in
    let tree_deriv = snd tree in
    let tree_feat = List.hd tree_left.feat in
    let match_feat = List.hd (List.hd tree_deriv).feat in
    match (tree_feat, match_feat) with
        (Licensor(licensor), Licensee(licensee)) when licensor = licensee -> move_feature_check tree_left tree_deriv
    | _ -> raise (Red_flag "MOVE error")

val premove : tree * tree list =
    ([{spec = ""; head = "";
        comp = " be -s devour -ing the ointment ";
        feat = [Licensor "q"; Categ "s"]},
        [{spec = ""; head = "John"; comp = ""; feat = [Licensee "q"]}]]
    # let move_john = move premove;;

val move_john : tree * tree list =
    ({spec = " John "; head = "";
      comp = " be -s devour -ing the ointment "; feat = [Categ "s"]},
     □)
Serial Verbs

- **Serial Verbs**: Series of 2 or more verbs within the same clause. They share arguments but are not morphological compounds.

- Main goal – what is the structure of an SVC? What features must you assign to verbs in SVC langs?

- Looking at Yoruba (and English)
  - Subordination? Adjunction?
Ex:

- *Aje wè lo.* (Aje swam away) /// *Aje lo sara.* (Aje went to run)
  
  *Aje swim go.* \hspace{1cm} Aje go run

- Multiple lexical entries (not just for serial verbs)

```javascript
let yoruba = [
  make_lexeme "Aje" [Categ("d"); Licensee("k"); Licensee("q")];
  make_lexeme "we" [Select("V"); Select("d"); Categ("V")]; (*swam*)
  make_lexeme "lo" [Select("V"); Select("d"); Categ("V")]; (*go, v1*)
  make_lexeme "lo" [Categ("V")]; (*go, v2*)
  make_lexeme "sara" [Categ("V")]; (*run*)
];
```
References
