L95: Natural Language Syntax and Parsing
8) Unification-based Grammars and Parsing

Paula Buttery

Dept of Computer Science & Technology, University of Cambridge
Reminder...

Last time we looked at lexicalisation and features to help us with:

- Modelling structural dependency across the tree as a whole
  - e.g. correctly modelling $NP$ expansion

- Modelling the structural behaviour specific to a lexical item:
  - pp-attachment
  - subcategorisation
  - co-ordination
Alternative approach represents features in **DAGs**

Re-conceptualise words, non-terminal nodes and parses as **Directed Acyclic Graphs** which may be represented as **Attribute Value Matrices**

We have **atomic categories** at each of the terminal nodes and another **AVM/DAG** at all other nodes
Some grammars allow the AVMs to be typed

Typing facilitates grammar building. Hierarchies of AVM types can be used to automatically populate features

\[ \text{NP} \rightarrow \text{AGREEMENT} \rightarrow \text{NUMBER} \rightarrow \text{num} \]

\[ \text{PERSON} \]

\[ \begin{bmatrix} \text{NP} \\
\text{AGREEMENT} \\
\text{PERSON} \\
\text{num} \\
3rd \end{bmatrix} \]

An shorthand notation uses angle bracket notation to indicate feature paths: e.g. \(<\text{NP AGREEMENT PERSON}>\) would represent the feature path leading to the atomic value 3rd.
DAGs and AVMs may exhibit **re-entrancy**

\[
\begin{array}{c}
\text{S} \\
\text{HEAD} \\
\text{AGREEMENT \{1\}} \\
\text{SUBJECT} \\
\text{AGREEMENT \{1\}} \\
\end{array}
\]

\[
\begin{array}{c}
\text{NUMBER sing} \\
\text{PERSON 3rd} \\
\text{AGREEMENT} \\
\end{array}
\]

The following shows an S type feature structure similar to J&M (page 526) with a path <S head agreement> that leads to the same feature structure as the path <S head subject agreement>:

\[
\begin{array}{c}
\text{S} \\
\text{HEAD} \\
\text{SUBJECT} \\
\end{array}
\]

1. **Non re-entrant:**

\[
\begin{array}{c}
\text{F}eature1 \\
\text{F}eature2 \\
\end{array}
\]

2. **Re-entrant:**

\[
\begin{array}{c}
\text{F}eature1 \{1\} \\
\text{F}eature2 \{1\} \\
\end{array}
\]

Notice that re-entrancy indicates that a feature structure is being encountered twice (we have two ways of getting to the same node); for the purpose of grammar design this is subtly different to simply encountering two feature paths that eventually evaluate to the same atomic categories.

Integrating Feature Structures into CFGs

We now have methods of representing feature structures; the next step then is to integrate them into our CFGs to see if we might improve the over-generation problems without inflating the rule set.
DAGs and AVMs may exhibit **re-entrancy**

1. Non re-entrant: \[
\begin{bmatrix}
\text{FEATURE1} & a \\
\text{FEATURE2} & a
\end{bmatrix}
\]

2. Re-entrant: \[
\begin{bmatrix}
\text{FEATURE1} & 1 & a \\
\text{FEATURE2} & 1
\end{bmatrix}
\]

Note that re-entrancy indicates that a feature structure is encountered twice (i.e., we have two ways of reaching the same node). For the purpose of grammar design this is subtly different to simply encountering two feature paths that eventually evaluate to the same atomic categories.

---

Paula Buttery (Computer Lab)  L95: Natural Language Syntax and Parsing  6 / 16
Parsing with DAGs involves **Unification**

- The unification of two DAGs is the most specific DAG which contains all the information in both of the original feature structures.
- Unification fails if the two DAGs contain conflicting information.
Parsin with DAGs involves **Unification**

- The unification of two DAGs is the most specific DAG which contains all the information in both of the original feature structures.
- Unification fails if the two DAGs contain conflicting information.

\[
\begin{align*}
\begin{bmatrix}
\text{PERSON} & 3rd \\
\end{bmatrix} & \sqsubset
\begin{bmatrix}
\text{NUMBER} & plural \\
\end{bmatrix} =
\begin{bmatrix}
\text{PERSON} & 3rd \\
\text{NUMBER} & plural \\
\end{bmatrix} \\
\begin{bmatrix}
\text{PERSON} & 1st \\
\text{NUMBER} & plural \\
\end{bmatrix} & \sqsubset
\begin{bmatrix}
\text{NUMBER} & [ ] \\
\end{bmatrix} =
\begin{bmatrix}
\text{PERSON} & 1st \\
\text{NUMBER} & plural \\
\end{bmatrix} \\
\begin{bmatrix}
\text{PERSON} & 1st \\
\text{NUMBER} & sing \\
\end{bmatrix} & \sqsubset
\begin{bmatrix}
\text{NUMBER} & plural \\
\end{bmatrix} = \text{unification fails} \\
\begin{bmatrix}
\text{FEATURE1} \\
\text{FEATURE2} & 1 \\
\text{FEATURE3} & 1 \\
\end{bmatrix} & \sqsubset
\begin{bmatrix}
\text{FEATURE3} & 1/a \\
\end{bmatrix} =
\begin{bmatrix}
\text{FEATURE1} \\
\text{FEATURE2} & a \\
\text{FEATURE3} & a \\
\end{bmatrix}
\end{align*}
\]
Unification examples in class
Unification **algorithm** requires **extra** graph structure

As with all recursive algorithms, the next step is to test for the various base cases of the recursion before proceeding on to a recursive call involving some part of the original arguments. In this case, there are three possible base cases:

- The arguments are identical
- One or both of the arguments has a null value
- The arguments are non-null and non-identical

If the structures are identical, then the pointer of the first is set to the second and the second is returned. It is important to understand why this pointer exchange is done in this case. After all, since the arguments are identical, returning either one would appear to suffice. This might be true for a single unification but recall that we want the two arguments to the unification operator to be truly unified. The pointer change is necessary since we want the arguments to be truly identical, so that any subsequent unification that adds information to one will add it to both.

In the case where either of the arguments is null, the pointer field for the null argument is changed to point to the other argument, which is then returned. The result is that both structures now point at the same value.

If neither of the preceding tests is true then there are two possibilities: they are non-identical atomic values, or they are non-identical complex structures. The former

From Jurafsky and Martin version 2
Unification algorithm requires extra graph structure.

Figure 15.7: The arguments after assigning the first argument’s new feature to the appropriate value in the second argument. In the case where the new feature does not match an existing feature in the second argument, a failure signal is returned. In the latter case, a recursive call is needed to ensure that the components of these complex structures are compatible. In this implementation, the key to the recursion is a loop over all the features of the second argument, \( f_2 \). This loop attempts to unify the value of each feature in \( f_2 \) with the corresponding feature in \( f_1 \). In this loop, if a feature is encountered in \( f_2 \) that is missing from \( f_1 \), a feature is added to \( f_1 \) and given the value NULL. Processing continues as if the feature had been there to begin with. If every one of these unifications succeeds, then the pointer field of \( f_2 \) is set to \( f_1 \) completing the unification of the structures and \( f_1 \) is returned as the value of the unification.

An Example
To illustrate this algorithm, let’s walk through the following example.
Unification \textbf{algorithm} requires \textbf{extra} graph structure.

Figure 15.8 The final result of unifying F1 and F2.

Figure 15.10 shows the extended representations for the arguments to this unification. These original arguments are neither identical, nor null, nor atomic, so the main loop is entered. Looping over the features of \( f_2 \), the algorithm tried to recursively attempt to unify the values of the corresponding \texttt{SUBJECT} features of \( f_1 \) and \( f_2 \).

These arguments are also non-identical, non-null, and non-atomic so the loop is entered again leading to a recursive check of the values of the \texttt{AGREEMENT} features.

In looping over the features of the second argument, the fact that the first argument lacks a \texttt{PERSON} feature is discovered. A \texttt{PERSON} feature initialized with a \texttt{NULL} value is, therefore, added to the first argument. This, in effect, changes the previous unification to the following.

\[
\text{\text{\texttt{NUMBER}} } \text{\texttt{sg}} \lor \text{\text{\texttt{PERSON}} } 3rd
\]

From Jurafsky and Martin version 2
DAGs can be straightforwardly associated with the lexicon

\[
\begin{align*}
\text{N} & \quad \rightarrow \quad \{\text{fish, rivers, pools, they}\} \\
\text{AGREEMENT} & \quad \rightarrow \quad \{\text{cans, fishes}\} \\
\text{PERSON} & \quad = \quad 3^{rd} \\
\text{NUMBER} & \quad = \quad \text{plural} \\
\text{V} & \quad \rightarrow \quad \{\text{can, fish, fishes, cans}\} \\
\text{NP} & \quad \rightarrow \quad \{\text{it, fish, rivers, pools, I, you, December, Scotland, they}\} \\
\text{NP} & \quad \rightarrow \quad \{\text{can, fish, fishes, cans}\} \\
\end{align*}
\]
DAGs can be straightforwardly associated with the lexicon

\[
\begin{bmatrix}
N \\
\text{AGREEMENT}
\end{bmatrix}
\begin{bmatrix}
\text{PERSON} & 3rd \\
\text{NUMBER} & \text{plural}
\end{bmatrix}
\rightarrow \{\text{fish, rivers, pools, they}\}
\]

\[
\begin{array}{c}
V \\
\rightarrow \{\text{cans, fishes}\}
\end{array}
\]

\[
< V \text{ AGREEMENT PERSON} > = 3rd
\]

\[
< V \text{ AGREEMENT NUMBER} > = \text{sing}
\]

\[
\langle \begin{bmatrix}
\text{N} \\
\text{AGREEMENT}
\end{bmatrix}
\begin{bmatrix}
\text{PERSON} & 3rd \\
\text{NUMBER} & \text{sing}
\end{bmatrix}
\rangle
\]

\[
\langle \text{they,} \begin{bmatrix}
\text{N} \\
\text{AGREEMENT}
\end{bmatrix}
\begin{bmatrix}
\text{PERSON} & 3rd \\
\text{NUMBER} & \text{sing}
\end{bmatrix}\rangle
\]
DAGs can be straightforwardly associated with the lexicon

\[
\begin{pmatrix}
\text{N} \\
\text{AGREEMENT}
\end{pmatrix}
\begin{pmatrix}
\text{PERSON} & 3rd \\
\text{NUMBER} & \text{plural}
\end{pmatrix} \rightarrow \{\text{fish, rivers, pools, they}\}
\]

\[
\text{V} \rightarrow \{\text{cans, fishes}\}
\]

\[
<\text{V AGREEMENT PERSON}> = 3rd
\]

\[
<\text{V AGREEMENT NUMBER}> = \text{sing}
\]

\[
\left<\text{they, N} \begin{pmatrix}
\text{AGREEMENT} \\
\text{PERSON} & 3rd \\
\text{NUMBER} & \text{sing}
\end{pmatrix}\right>
\]
We can modify CFG algorithms to parse with DAGs

- We can use any CFG parsing algorithm if:
  - associate feature constraints with CFG rules
  - unify DAGs in the states

\[ S \rightarrow NP \ VP \]
\[ < NP \ HEAD \ AGREEMENT > = < VP \ HEAD \ AGREEMENT > \]
\[ < S \ HEAD > = < VP \ HEAD > \]

- We would have items like \([X, [0, m], DAG]\) on the agenda or at each cell
Subcategorization is captured by the feature constraints

\[
\begin{align*}
[S] & \rightarrow [NP \quad [AGREEMENT \ 3] \quad [VP \quad [AGREEMENT \ 3]]] \\
[VP \quad [HEAD \ 1] \quad [SUBJ \ 3]] & \rightarrow [V \quad [HEAD \ 1] \quad [OBJ \ 2] \quad [SUBJ \ 3]]
\end{align*}
\]
Subcategorization is captured by the feature constraints

\[
\begin{align*}
[S &\rightarrow [NP \rightarrow [VP \rightarrow [V]]], \\
&[\text{HEAD } 1, \text{NP } 2, \text{VP } 2, \text{V } 2, \text{can, } 1], \\
&[\text{HEAD } 1, \text{AGREEMENT } 3], \\
&[\text{SUBJ } 3, \text{SUBJ } 3, \text{SUBJ } 3, \text{OBJ } 2], \\
&[\text{HEAD } 1, \text{AGREEMENT } 3], \\
&[\text{HEAD } 1, \text{NP } 1, \text{VP } 1, \text{head }, 1]
\end{align*}
\]
Subcategorization is captured by the feature constraints.
Alternatively use **unification as the parsing operation** instead of just for search-space reduction through feature constraining:

- \( X_0 \rightarrow X_1 X_2 \)
  - \(< X_1 \text{ HEAD AGREEMENT }> = < X_2 \text{ HEAD AGREEMENT }>\)
  - \(< X_0 \text{ HEAD }> = < X_1 \text{ HEAD }>\)

- \( X_0 \rightarrow X_1 X_2 \)
  - \(< X_0 \text{ HEAD }> < X_1 \text{ HEAD }>\)
  - \(< X_2 \text{ CAT }> = \text{ PP}\)

- \( X_0 \rightarrow X_1 \text{ and } X_2 \)
  - \(< X_0 \text{ CAT }> < X_1 \text{ CAT }>\)
  - \(< X_1 \text{ CAT }> < X_2 \text{ CAT }>\)
Alternatively use **unification as the parsing operation** instead of just for search-space reduction through feature constraining:

- $X_0 \rightarrow X_1 X_2$
  - $< X_1 \text{ HEAD AGREEMENT } >= < X_2 \text{ HEAD AGREEMENT }$
  - $< X_0 \text{ HEAD } >= < X_1 \text{ HEAD }$

- $X_0 \rightarrow X_1 X_2$
  - $< X_0 \text{ HEAD } > < X_1 \text{ HEAD }$
  - $< X_2 \text{ CAT } >= PP$

- $X_0 \rightarrow X_1 \text{ and } X_2$
  - $< X_0 \text{ CAT } > < X_1 \text{ CAT }$
  - $< X_1 \text{ CAT } > < X_2 \text{ CAT }$
Alternatively use **unification as the parsing operation** instead of just for
search-space reduction through feature constraining:

- $X_0 \rightarrow X_1 X_2$
  
  $< X_1 \text{ HEAD AGREEMENT } >= < X_2 \text{ HEAD AGREEMENT } >$
  
  $< X_0 \text{ HEAD } >= < X_1 \text{ HEAD } >$

- $X_0 \rightarrow X_1 X_2$
  
  $< X_0 \text{ HEAD } > < X_1 \text{ HEAD } >$
  
  $< X_2 \text{ CAT } >= PP$

- $X_0 \rightarrow X_1 \text{ and } X_2$
  
  $< X_0 \text{ CAT } > < X_1 \text{ CAT } >$
  
  $< X_1 \text{ CAT } > < X_2 \text{ CAT } >$
Focus on adequacy for a wide range of languages as well as tractable for parsing
Examples include **Lexical Functional Grammar, LFG** (Bresnan and Kaplan) and **Head-driven Phrase Structure Grammar, HPSG** (Pollard and Sag)
Grammars tend to incorporate aspects of morphology, syntax and compositional semantics:

\[
\langle \text{^s, [HEAD [N AGREEMENT pl]]} \rangle
\]

\[
\langle \text{fox, [HEAD [N AGREEMENT []]]} \rangle
\]

\[
\begin{align*}
\text{[HEAD [N AGREEMENT pl]]} & \sqcup \text{[HEAD [N AGREEMENT []]]} = \text{[HEAD [N AGREEMENT pl]]}
\end{align*}
\]

If you are interested see: [http://www.delph-in.net](http://www.delph-in.net)