11-722: Grammar Formalisms

Parsing Overview

Alon Lavie

January 20, 2004

References:

Hopcroft and Ullman, “Introduction to Automata Theory, Languages and Computation”.
James Allen, “Natural Language Understanding”, 2nd edition.
Jurafsky and Martin, “Speech and Language Processing”.

Formal Languages

- A mathematical abstraction of *real* languages: Natural Languages and Computer Languages
- A language is no more than a set of items called *words* (the equivalent of sentences in a natural language)
- Languages can be defined declaratively, descriptively or computationally
- Formal Language Theory: The study of properties of the various types and classes of languages using formal mathematical proofs
- Fundamental problem - word membership:
  Given a word w and a language L - is $w \in L$?
- what algorithm or computational device is necessary to answer this question depends on the class of the language

Basic Definitions

- $\boldsymbol{\Sigma}$: the **alphabet** - a finite (non-empty) set of atomic symbols
  - each symbol σ in the set is a **letter**
  - letters are denoted by lower case Latin letters a, b, c,...
- a **word** is a string of letters from a given alphabet Σ
- $|w|$ denotes the length of word w
- ϵ denotes the empty word: $|\epsilon| = 0$
- we only consider words of finite length
- **Def:** a language L is a set (finite or infinite) of words constructed from a given alphabet Σ
- Examples: $L_1 = \{\epsilon\} \quad L_2 = \emptyset \quad L_3 = \{a^n b^n | n \geq 0\}$
- Set Theory and operations apply to formal languages:
  - union, intersection, complementation, membership
  - $\overline{L} = \{w \in \Sigma^* | w \notin L\}$
- Important notation:
  $\boldsymbol{\Sigma^*}$ $=$ set of all finite words over the alphabet Σ
  $\boldsymbol{\Sigma^i}$ $=$ set of all words of length i over the alphabet Σ

Language Classes

- Sets of formal languages that can be defined using a particular descriptive definition or abstraction of a computational framework
- Examples:
  - The set of languages that can be described by Regular Expressions
  - The set of languages for which we can construct a Finite State Automaton
  - The set of languages that can be defined using a Context-free Grammar
- Knowing the class to which a language belongs will allow us to develop efficient algorithms for processing the language or deciding membership in the language

Deterministic FSA

- **Formal Definition of a DFSA:** $A = (Q, \Sigma, \delta, q_0, F)$ where:
  - Q is a finite set of states
  - Σ is a finite alphabet
  - $q_0 \in Q$ is an initial (start) state
  - $F \subseteq Q$ is a set of *final* states
  - $\delta : Q \times \Sigma \rightarrow Q$ is the complete transition function
- The language accepted by a DFSA A is defined to be:
  $L(A) = \{w \in \Sigma^* |$ after computing on w, A is in a state $q \in F\}$
- based on the function δ, we define $\hat{\delta}$, the function on words that models the computation of a DFSA recursively as follows:
  - $\hat{\delta} : Q \times \Sigma^* \rightarrow Q$
  - $\hat{\delta}(q, \epsilon) = q \;\; \forall q \in Q$
  - $\hat{\delta}(q, x\sigma) = \delta(\hat{\delta}(q, x), \sigma)$
- Formal definition of $L(A)$: $L(A) = \{w \in \Sigma^* | \, \hat{\delta}(q_0, w) \in F\}$
- **Def: Regular Language:** a language $L \subseteq \Sigma^*$ is called *regular* if there exists some DFSA A such that $L = L(A)$
- Examples of regular languages: $L = \Sigma^* \;\; L = \{\epsilon\} \;\; L = (aab)^*$

Context-Free Grammars

- A descriptive generative formalism for specifying the set of words in a language using production rules
- **Formal Definition:** a *context-free grammar* $G = (V, T, P, S)$
  - V is a finite set of variables
  - T is a finite set of terminal symbols (similar to Σ for FSAs)
  - P is a set of *context-free* production rules, each of the form $A \rightarrow \alpha$, where $\alpha \in (V \cup T)^*$
  - S is a start non-terminal ($S \in V$)
- Notations:
  - we denote elements of V by S, A, B, C...
  - we denote elements of T by a, b, c...
  - we denote strings over T^* by w, x, y...
  - we denote strings over $(T \cup V)^*$ by α, β, γ...
  - we denote single variables or terminals by X, Y, Z...

Context-Free Grammars

- Example: $L = \{a^n b^n \mid n \geq 1\}$

```
G:    S --> a S b
      S --> a b
```

- in this case the language $L(G)$ could be specified in a succinct mathematical form - often this is difficult or not possible

CFG Derivations

- derivations describe the process of using the context-free rules to derive a string of terminal symbols

- **Definition:** let $\varphi_1, \varphi_2 \in (V \cup T)^*$.
  φ_1 *directly derives* φ_2, denoted by: $\varphi_1 \Longrightarrow_G \varphi_2$,
  if $\varphi_1 = \alpha A \beta, \quad \varphi_2 = \alpha\gamma\beta$ and $A \rightarrow \gamma$ is a rule in P_G

- φ_1 *derives* φ_2, denoted by $\varphi_1 \overset{*}{\Longrightarrow}_G \varphi_2$,
  if there exists a finite sequence of direct derivations such that
  $\varphi_1 \Longrightarrow_G \varphi_1' \Longrightarrow_G \varphi_2' \Longrightarrow_G \varphi_3' \Longrightarrow_G \cdots \Longrightarrow_G \varphi_2$

- $\varphi_1 \overset{i}{\Longrightarrow}_G \varphi_2$ denotes that φ_1 derives φ_2 in exactly i derivation steps

- a *rightmost* derivation is a derivation in which at each step, the rightmost non-terminal in the string is picked for expansion

- similarly for a *leftmost* derivation

Context Free Languages (CFLs)

- **Formal Definition:** the language of a CFG G is defined as:
  $L(G) = \{w \in T^* \mid S \overset{*}{\Longrightarrow}_G w\}$
- a language L is *context-free* if there exists a grammar G such that $L = L(G)$
- the set of all such languages is called the set of context-free languages (CFLs)
- two grammars G_1 and G_2 are called *equivalent* if $L(G_1) = L(G_2)$

Parse Trees

- a Parse Tree is a graphical representation of a derivation
- the leaves (yield) of the tree correspond to a terminal string in $L(G)$
- the tree does not represent the derivation order of the non-terminals
- the tree does reflect the structure of the input string - what rules were used to derive the various substrings of the input
- a parse tree constitutes a proof that a given input string is in $L(G)$
- a grammar G is called *ambiguous* if there exists a word $w \in L(G)$ that has two or more different parse trees
- There exist CFLs that are inherently ambiguous

Pushdown Automata

- An extension of a FSA that is powerful enough to accept CFLs
- The FSA is augmented with a memory storage device in the form of a stack
- **Formal Definition:** a PDA $M = (Q, \Sigma, \Gamma, \delta, q_0, Z_0, F)$ where:
  - Q, Σ, q_0, F are similar to those of a FSA
  - Γ is a finite set of stack symbols
  - Z_0 is a start stack symbol
  - $\delta : Q \times (\Sigma \cup \{\epsilon\}) \times \Gamma \rightarrow 2^{Q \times \Gamma^*}$
    $\delta(q, \sigma, Z) = \{(q_1, \gamma_1), (q_2, \gamma_2), ..., (q_m, \gamma_m)\}$
- Note that a PDA is *non-deterministic*: it can make ϵ-moves on the input
- It can also: replace the top element of the stack, "push" an element onto the stack, and "pop" an element from the stack

Recognition and Parsing of CFLs

- the recognition problem:
  given a grammar G and a word w, is $w \in L(G)$
- the parsing problem:
  given a grammar G and a word w, if $w \in L(G)$, find a parse tree (or all possible parse trees) for w
- there exist a variety of algorithms for parsing CFLs and their variants

Context Sensitive and Unrestricted Grammars

- CFGs are called *context-free* because the form of the grammar rules allows them to be used in a derivation regardless of the context in which a non-terminal appears
- there exist less restricted forms of grammars:
- **Context Sensitive** grammars are grammars where the rules have the form $\alpha \rightarrow \beta$, with the restriction that $|\alpha| \leq |\beta|$
- in order to be applied in a derivation, the entire left-hand side of the rule must match a substring of the current derived string
- **Unrestricted** grammars are grammars where the rules are unrestricted in form - $\alpha \rightarrow \beta$, where α contains one or more grammar symbols, and β contains zero or more grammar symbols
- more powerful computation devices are required in order to recognize the languages defined by these types of grammars

The Chomsky Hierarchy

- Chomsky was one of the pioneers in identifying the correspondence between the different types of grammars and the formal computational models that are required to recognize them:
- **Type-0 Grammars** are unrestricted grammars, correspond to recursively enumerable languages, require Turing Machines to recognize them
- **Type-1 Grammars** are context-sensitive grammars, correspond to context-sensitive languages and require a type of automata called *linear-bounded automata* to recognize them
- **Type-2 Grammars** are context-free grammars, correspond to CFLs and require PDAs to recognize them
- **Type-3 Grammars** are regular grammars, correspond to regular languages and require FSAs to recognize them
- The syntax of natural languages is often described by phrase structure rules that are "extended" CFGs. Algorithms for parsing them are often based on extensions of parsers for CFGs.

Parsing Algorithms

- Clear distinction in all grammar formalisms:
  - **The Grammar:** a *declarative* (usually generative) *finite* description of what structures in the language are grammatical
  - **The Language:** the (possibly infinite) set of all strings that are derivable according to the grammar
  - **The Parser:** an algorithm that for a given input, decides membership in the language, and determines it's structure according to the grammar
- In many grammar formalisms CFGs are basis for describing the constituent structure of NL sentences
- Recognition vs. Parsing:
  - *Recognition* - deciding the membership in the language:
    For a given grammar G, an algorithm that given an input w decides: is $w \in L(G)$?
  - *Parsing* - Recognition $+$ producing a parse tree for w
- Is parsing more "difficult" than recognition? (time complexity)
- Ambiguity - *a* parse for w or *all* parses for w?
  - Identifying the "correct" parse
  - Ambiguity representation - an input may have exponentially many parses

CFL Parsing Algorithms

Parsing General CFLs vs. Limited Forms

- Efficiency:
  - Deterministic (LR) languages can be parsed in *linear time*
  - A number of parsing algorithms for general CFLs require $O(n^3)$ time
  - Asymptotically best parsing algorithm for general CFLs requires $O(n^{2.376})$, but is not practical
- Utility - why parse general grammars and not just CNF?
  - Grammar intended to reflect actual structure of language
  - Conversion to CNF completely destroys the parse structure
- Parsing Unification-based grammars is quite a different story...

Top-Down vs. Bottom-Up Parsing

Top-Down Parsing:

- Construct the parse-tree starting from the root (“S”) of the grammar
- At each step, expand a non-terminal using one selected grammar rule
- match terminal nodes with the input
- backtrack when tree is inconsistent with input
- Advantage: only constructs partial trees that can be derived from the root “S”
- Problems: efficiency, handling ambiguity, left-recursion

Bottom-Up Parsing:

- Construct a parse starting from the input symbols
- Build constituents from sub-constituents
- When all constituents on the RHS of a rule are matched, create a constituent for the LHS of the rule
- Advantage: only creates constituents that are consistent with the input
- Problems: efficiency, handling ambiguity

Top-Down vs. Bottom-Up Parsing

- Various CFG parsing algorithms are a hybrid of Top-Down and Bottom-Up
- Attempt to combine the advantages of both
- A *Chart* allows storing partial analyses, so that they can be shared or memorized
- Ambiguity Packing allows efficient storage of ambiguous analyses

The Earley Parsing Algorithm

General Principles:

- A clever hybrid *Bottom-Up* and *Top-Down* approach
- *Bottom-Up* parsing completely guided by *Top-Down* predictions
- Maintains sets of "dotted" grammar rules that:
  - Reflect what the parser has "seen" so far
  - Explicitly predict the rules and constituents that will combine into a complete parse
- Time Complexity $O(n^3)$, but better on particular sub-classes
- First efficient parsing algorithm for general context-free grammars.

The Earley Parsing Method

- Main Data Structure: The *“state”* (or *“item”*)
- A state is a “dotted” rule and starting position:
  $[A \rightarrow X_1 ... \bullet C ... X_m, p_i]$
- The algorithm maintains sets of states, one set for each position in the input string (starting from 0)
- We denote the set of states for position i by S_i

The Earley Parsing Algorithm

Three Main Operations:

- **Predictor:** If state $[A \rightarrow X_1 ... \bullet C ... X_m, j] \in S_i$ then for every rule of the form $C \rightarrow Y_1 ... Y_k$, add to S_i the state $[C \rightarrow \bullet Y_1 ... Y_k, i]$
- **Completer:** If state $[A \rightarrow X_1 ... X_m \bullet, j] \in S_i$ then for every state in S_j of form $[B \rightarrow X_1 ... \bullet A ... X_k, l]$, add to S_i the state $[B \rightarrow X_1 ... A \bullet ... X_k, l]$
- **Scanner:** If state $[A \rightarrow X_1 ... \bullet a ... X_m, j] \in S_i$ and the next input word is $x_{i+1} = a$, then add to S_{i+1} the state $[A \rightarrow X_1 ... a \bullet ... X_m, j]$

The Earley Recognition Algorithm

- Simplified version with no lookaheads and for grammars without epsilon-rules
- Assumes input is string of grammar terminal symbols
- We extend the grammar with a new rule $S' \rightarrow S\ \$$
- The algorithm sequentially constructs the sets S_i for $0 \leq i \leq n+1$
- We initialize the set S_0 with $S_0 = \{[S' \rightarrow \bullet S\ \$, 0]\}$

The Earley Recognition Algorithm

The Main Algorithm: parsing input $x = x_1 ... x_n$

1. $S_0 = \{[S' \rightarrow \bullet S \ \$, 0]\}$
2. For $0 \leq i \leq n$ do:
   Process each item $s \in S_i$ in order by applying to it the *single* applicable operation among:
   (a) Predictor (adds new items to S_i)
   (b) Completer (adds new items to S_i)
   (c) Scanner (adds new items to S_{i+1})
3. If $S_{i+1} = \phi$, *Reject* the input
4. If $i = n$ and $S_{n+1} = \{[S' \rightarrow S \ \$\bullet, 0]\}$ then *Accept* the input

Parsing with an Earley Parser

- We need to keep back-pointers to the constituents that we combine together when we complete a rule
- Each item must be extended to have the form $[A \rightarrow X_1(pt_1) ... \bullet C ... X_m, j]$, where the pt_i are "pointers" to the already found RHS sub-constituents
- the constituents and the pointers can be created during Scanner and Completer
- At the end - reconstruct parse from the "back-pointers"

Efficient Representation of Ambiguities

- a Local Ambiguity - multiple ways to derive the *same* substring from a non-terminal A
- What do local ambiguities look like with Earley Parsing?
  - Multiple items in the constituent chart of the form $[A \rightarrow X_1(pt_1)...X_m(pt_m)](p_k, p_j)$, with the same A, p_j and p_k.
- Local Ambiguity Packing: create a *single* item in the Chart for $A(p_j, p_k)$, with pointers to the various possible derivations.
- $A(p_j, p_k)$ can then be a sufficient "back-pointer" in the chart
- Allows to efficiently represent a very large number of ambiguities (even exponentially many)
- Unpacking - producing one or more of the packed parse trees by following the back-pointers.

Time Complexity of Earley Algorithm

- Algorithm iterates for each word of input (i.e. n iterations)
- How many items can be created and processed in S_i?
  - Each item in S_i has the form $[A \rightarrow X_1 ... \bullet C ... X_m, j]$, $0 \leq j \leq i$
  - Thus $O(n)$ items
- The *Scanner* and *Predictor* operations on an item each require constant time
- The *Completer* operation on an item adds items of form $[B \rightarrow X_1 ... A \bullet ... X_k, l]$ to S_i, with $0 \leq l \leq i$, so it may require up to $O(n)$ time for each processed item
- Time required for each iteration (S_i) is thus $O(n^2)$
- Time bound on entire algorithm is therefore $O(n^3)$

Time Complexity of Earley Algorithm

Special Cases:

- *Completer* is the operation that may require $O(i^2)$ time in iteration i
- For unambiguous grammars, Earley shows that the completer operation will require at most $O(i)$ time
- Thus time complexity for unambiguous grammars is $O(n^2)$
- For some grammars, the number of items in each S_i is bounded by a *constant*
- These are called *bounded-state* grammars and include even some ambiguious grammars.
- For bounded-state grammars, the time complexity of the algorithm is linear - $O(n)$

Earley Parsing - Example

The Grammar:

$$
\begin{array}{llcl}
(1) & S & \rightarrow & NP\,VP \\
(2) & NP & \rightarrow & art\ adj\ n \\
(3) & NP & \rightarrow & art\ n \\
(4) & NP & \rightarrow & adj\ n \\
(5) & VP & \rightarrow & aux\ VP \\
(6) & VP & \rightarrow & v\ NP
\end{array}
$$

The original input: “$x =$ The large can can hold the water”

POS assigned input: “$x =$ art adj n aux v art n”

Parser input: “$x =$ art adj n aux v art n $”

Earley Parsing - Example

The input: “$x =$ art adj n aux v art n $”

Earley Parsing - Example

The input: “$x =$ **art** adj n aux v art n \$”

S_0: $[S' \rightarrow \bullet S \,\$\,,\ 0]$

$[S \rightarrow \bullet NP\ VP\ ,\ 0]$

$[NP \rightarrow \bullet art\ adj\ n\ ,\ 0]$

$[NP \rightarrow \bullet art\ n\ ,\ 0]$

$[NP \rightarrow \bullet adj\ n\ ,\ 0]$

S_1: $[NP \rightarrow art_1\ \bullet adj\ n\ ,\ 0]$ 1 art (0,1)

$[NP \rightarrow art_1\ \bullet n\ ,\ 0]$

Earley Parsing - Example

The input: "$x =$ art **adj** n aux v art n \$"

S_1: $[NP \rightarrow art_1 \bullet adj\ n\ ,\ 0]$

$[NP \rightarrow art_1 \bullet n\ ,\ 0]$

S_2: $[NP \rightarrow art_1\ adj_2 \bullet n\ ,\ 0]$ 2 adj (1,2)

Earley Parsing - Example

The input: "$x =$ art adj **n** aux v art n $"

S_2: $[NP \rightarrow art_1\ adj_2\ \bullet n\ ,\ 0]$

S_3: $[NP_4 \rightarrow art_1\ adj_2\ n_3 \bullet\ ,\ 0]$	3	n (2,3)
	4	$NP \rightarrow art_1\ adj_2\ n_3$ (0,3)

Earley Parsing - Example

The input: “$x =$ art adj n **aux** v art n $”

S_3: $[NP_4 \rightarrow art_1\ adj_2\ n_3 \bullet,\ 0]$
$[S \rightarrow NP_4\ \bullet VP,\ 0]$
$[VP \rightarrow \bullet aux\ VP,\ 3]$
$[VP \rightarrow \bullet v\ NP,\ 3]$

S_4: $[VP \rightarrow aux_5\ \bullet VP,\ 3]$ 5 aux (3,4)

Earley Parsing - Example

The input: "$x =$ art adj n aux **v** art n \$"

S_4: $[VP \rightarrow aux_5 \ \bullet VP \ , \ 3]$

$[VP \rightarrow \bullet aux \ VP \ , \ 4]$

$[VP \rightarrow \bullet v \ NP \ , \ 4]$

S_5: $[VP \rightarrow v_6 \ \bullet NP \ , \ 4]$ 6 v (4,5)

Earley Parsing - Example

The input: "$x =$ art adj n aux v **art** n \$"

S_5: $[VP \rightarrow v_6 \bullet NP\ ,\ 4]$

$[NP \rightarrow \bullet art\ adj\ n\ ,\ 5]$

$[NP \rightarrow \bullet art\ n\ ,\ 5]$

$[NP \rightarrow \bullet adj\ n\ ,\ 5]$

S_6: $[NP \rightarrow art_7 \bullet adj\ n\ ,\ 5]$ 7 art (5,6)

$[NP \rightarrow art_7 \bullet n\ ,\ 5]$

Earley Parsing - Example

The input: “$x =$ art adj n aux v art **n** \$”

S_6: $[NP \rightarrow art_7 \bullet adj\ n\ ,\ 5]$
$[NP \rightarrow art_7 \bullet n\ ,\ 5]$

S_7: $[NP_9 \rightarrow art_7\ n_8 \bullet\ ,\ 5]$

8 n (6,7)
9 $NP \rightarrow art_7\ n_8$ (5,7)

Earley Parsing - Example

The input: “$x =$ art adj n aux v art n **$**”

S_7:	$[NP_9 \rightarrow art_7\ n_8 \bullet,\ 5]$	
	$[VP_{10} \rightarrow v_6\ NP_9 \bullet,\ 4]$	10 $VP \rightarrow v_6\ NP_9$ (4,7)
	$[VP_{11} \rightarrow aux_5\ VP_{10} \bullet,\ 3]$	11 $VP \rightarrow aux_5\ VP_{10}$ (3,7)
	$[S_{12} \rightarrow NP_4\ VP_{11} \bullet,\ 0]$	12 $S \rightarrow NP_4\ VP_{11}$ (0,7)
	$[S' \rightarrow S \bullet \$,\ 0]$	

S_8: $[S' \rightarrow S\ \$ \bullet,\ 0]$

Augmenting CFGs with Features

- Certain linguistic constraints are not naturally described via CFGs
- Example: *Number Agreement* between constituents - “`a boys`”
- Possible to describe using refined CF rules:

```
NP-Sing --> ART-Sing N-Sing
NP-Plu  --> ART-Plu  N-Plu
```

- Much more natural to describe via a *single* feature-augmented CF rule:

```
NP --> ART  N
   ((x1 number = x2 number))
```

- Describing a large set of such feature constraints using only CF rules is not practical

Feature Structures

- *Constituents* can be viewed as *structures* (collections) of *features* that have assigned *values*
- Features can be *shared between constituents*
- Linguistic constraints express rules about how the feature-structure of a constituent is formed from its sub-constituents
- Some basic features for English:
  - Number, Gender and Person agreement
  - Verb form features and sub-categorizations
- Complex Feature Structures: Feature values can themselves be feature structures

Unification of Feature Structures

- Unification Grammars (such as HPSG) establish a complete linguistic theory for a language via a set of relationships between feature structures of constituents
- Key concept - *subsumption* relationship between two FSs:
  $F_1 \ subsumes \ F_2$ if every feature-value pair in F_1 is also in F_2
- Two FSs F_1 and F_2 *unify* if there exists a FS F that both F_1 and F_2 subsume.
- *The Most General Unifi er* is the minimal FS F that both F_1 and F_2 subsume.
- The *Unifi cation* operation allows easy expression of grammatical relationships among constituent feature structures

Unification of Feature Structures

Example:

- F_1 *subsumes* F_2:

```
F1 = ((cat *v))    F2 = ((cat *v)
                         (root *cry))
```

- F_3 is MGU of F_1 and F_2:

```
F1 = ((cat *v)     F2 = ((cat *v)          F3 = ((cat *v)
      (root *cry))       (vform *pres))          (root *cry)
                                                 (vform *pres))
```

- F_1 and F_2 do not unify:

```
F1 = ((cat *v)      F2 = ((cat *v)
      (agr *3s))          (agr *3p))
```

Unification-based Grammars

- Grammar rules can be completely specified using unification
- Example:

```
X0 --> X1 X2
   ((x0 cat = S)
    (x1 cat = NP)
    (x2 cat = VP)
    (x1 agr = x2 agr)
    (x0 subj = X1))
```

- If a feature (such as `cat`) is always specified, it can be associated with the non-terminal of a CFG rule
- Example:

```
S --> NP  VP
   ((x1 agr = x2 agr)
    (x0 subj = x1))
```

Unification-based Grammars

Example:

- The grammar rule:

```
NP --> ART  N
   (((x1 agr) = (x2 agr))
    ((x0 spec) = (x1 spec))
    (x0 = x2))
```

- The Feature Structures:

```
ART: ((agr *3s,*3p)  N: ((agr *3s)     NP:((agr *3s)
      (root *the)        (root *boy))      (spec *def)
      (spec *def))                         (root *boy))
```

CFG Parsing with Feature Unification

- *Back-bone* CFG is augmented with a functional description that describes unification constraints between grammar constituents
- The FS corresponding to the "root" of the grammar is constructed compositionally during parsing
- This is called *Interleaved Unifi cation*
- Other approaches are also possible

Unification Augmented Earley Parsing

- CFG is augmented with unification equations
- During parse time - the parser maintains a FS associated with each constituent in the chart
- Whenever COMPLETER applies (for rule i) - the unification operations associated with rule i are applied to the given FSs of the RHS constituents
- If unification succeeds, the FS associated with the LHS constituent of the rule is returned and attached to the new constituent created for the LHS of the rule.
- If the unification function fails - the rule completion “fails” - LHS constituent is *not created*

Ambiguity Packing and Unification Grammars

- Complex interaction between ambiguity detection and packing and unification
- Unification creates non-local chains of dependencies
- Pure unification grammars cannot always be parsed efficiently
- in unification-augmented CFG parsing with interleaved unification:
  - Unification can interfere with efficient ambiguity packing
  - f-structures must also be efficiently represented and packed
  - Parsing algorithms can be optimized to achieve maximal ambiguity packing [Lavie and Rose 2000]
- Strategies other than interleaved unification are possible:
  - Compute packed c-structure first, then solve unification constraints
  - multi-pass strategies for computing c-structure and f-structure can improve parsing efficiency [Placeway 2002]
  - In some pure unification grammars, subsumption can replace ambiguity packing