Chart parsing with non-atomic categories

L545
Spring 2010

With thanks to Detmar Meurers

The issue

- Parsing strategies and memoization (well-formed substring tables, charts) discussed with atomic categories.
  - Example: $S \rightarrow NP \ VP$
- How about the compound terms used as categories?
  - Example: $S \rightarrow NP(\text{Per,Num}) \ VP(\text{Per,Num})$

Overview

Three options for parsing with grammars using non-atomic categories:
1. Expand the grammar into a CFG with atomic categories
2. Parse using an atomic CFG backbone with reduced information
3. Incorporate special mechanisms into the parser

Idea 1: Transform into CFG with atomic categories

If only compound terms without variables are used as categories, the rules directly correspond to rules with atomic categories.
Example:
- $S \rightarrow NP(1,\text{sg}) \ VP(1,\text{sg})$
- $S \rightarrow NP(2,\text{sg}) \ VP(2,\text{sg})$
- $S \rightarrow NP(3,\text{sg}) \ VP(3,\text{sg})$
- $S \rightarrow NP(1,\text{pl}) \ VP(1,\text{pl})$
- $S \rightarrow NP(2,\text{pl}) \ VP(2,\text{pl})$
- $S \rightarrow NP(3,\text{pl}) \ VP(3,\text{pl})$

More on Idea 1

If there are a finite set of possible values for the variables occurring in the compound terms, it is possible to replace a rule with the instances for all possible instantiations of variables.
Example:
- $S \rightarrow NP(\text{Per,Num}) \ VP(\text{Per,Num})$
- $S \rightarrow NP(1,\text{sg}) \ VP(1,\text{sg})$
- $S \rightarrow NP(2,\text{sg}) \ VP(2,\text{sg})$
- $S \rightarrow NP(3,\text{sg}) \ VP(3,\text{sg})$
- $S \rightarrow NP(1,\text{pl}) \ VP(1,\text{pl})$
- $S \rightarrow NP(2,\text{pl}) \ VP(2,\text{pl})$
- $S \rightarrow NP(3,\text{pl}) \ VP(3,\text{pl})$

Evaluation of Idea 1

- leads to a potentially huge set of rules (number of categories grows exponentially w.r.t. the number of features)
  - grammar size relevant for time and space efficiency of parsing
- doesn’t allow for variables, i.e., misses generalizations
Idea 2: Parse using atomic CFG backbone (reduced info)

• idea:
  – parse using a property defined for all categories
  – use other properties to filter solutions from set of parses

• downside:
  – parsing with partial information can significantly enlarge the search space

Idea 3: Incorporate special mechanism into parser

• How two categories are combined has to be replaced by **unification**.
• Every active and inactive edge in a chart may be used for different uses. So for each time an edge is used, a new copy needs to be made.
• Two effectiveness issues:
  – Use **subsumption** test to ensure general enough predictions
  – Using **restriction** to prevent prediction loops
• Two efficiency issues (not dealt with here):
  – intelligent **indexing** of edges in chart
  – **packing** of similar edges in chart (cf., Tomita parser)

Where we’re going

First, we need to thoroughly explore using non-atomic categories

- Feature Structures and Unification
- Unification-Based Grammars
- Chart Parsing with Unification-Based Grammars
- Type Hierarchies

Feature structures

- To address the problem of adding agreement to CFGs, we need features, e.g., a way to say:

  \[
  \begin{array}{c}
  \text{NUMBER } sg \\
  \text{PERSON } 3 \\
  \end{array}
  \]

- A structure like this allows us to state properties, e.g., about a noun phrase

  \[
  \begin{array}{c}
  \text{CAT } NP \\
  \text{NUMBER } sg \\
  \text{PERSON } 3 \\
  \end{array}
  \]

- Each feature (e.g., **NUMBER**) is paired with a value (e.g., *sg*)
  – A bundle of feature-value pairs can be put into an attribute-value matrix (AVM)

Constraints

What we’re doing is saying that each rule of the grammar is a complex bundle of constraints

- \( S \rightarrow NP \ VP \) means that an \( S \) object is constrained to be composed of an \( NP \) followed by a \( VP \)

Features allow us to add more constraints

- \( S \rightarrow NP \ VP \) only if the number of \( NP = \) the number of \( VP \)
  – Constraint 1: \( S \rightarrow NP \ VP \)
  – Constraint 2: \( NP \; \text{NUM} = VP \; \text{NUM} \)

So, what we are delving into is constraint-based processing

Feature paths

Values can be atomic (e.g., *sg* or *NP* or 3):

\[
\begin{array}{c}
\text{NUMBER } sg \\
\text{PERSON } 3 \\
\end{array}
\]

Or they can be complex, allowing for **feature paths**:

\[
\begin{array}{c}
\text{CAT } NP \\
\text{AGREEMENT } [\text{NUMBER } sg] \\
\text{PERSON } 3 \\
\end{array}
\]

The value of the path \([\text{AGREEMENT}][\text{NUMBER}]\) is *sg*

- Complex values allow for more expressivity than a CFG, i.e., can represent more linguistic phenomena
Feature structures as graphs

• Technically, feature structures are directed acyclic graphs (DAGs)
• The feature structure represented by the attribute-value matrix (AVM):

\[
\begin{bmatrix}
\text{CAT} & \text{NP} \\
\text{AGR} & \text{NUM} [s] \\
& \text{PER} [3]
\end{bmatrix}
\]

is really the graph:

\[
\text{cat} \quad \text{np} \quad \text{sg} \\
\text{agr} \quad \text{num} \\
\text{per} \quad 3
\]

Reentrancy (structure sharing)

Feature structures embedded in feature structures can share the same values

• Two features share precisely the same object as their value
  – Well indicate this with a tag like \(\text{\#} \)

\[
\begin{bmatrix}
\text{CAT} & S \\
\text{HEAD} & \text{AGR} [\text{NUM} [s]] \\
\text{SUBJ} & \text{AGR} [\text{PER} [3]]
\end{bmatrix}
\]

• In this example, the agreement features of both the matrix sentence and embedded subject are identical (same object)
  – This is referred to as reentrancy

Unification

Well often want to merge feature structures

• Unification (\(\cup\)) = a basic operation to merge two feature structures into a resultant feature structure (FS)

The two feature structures must be compatible, i.e., have no values that conflict

• Identical FSs: \(\text{[NUMBER} [s] \cup \text{[NUMBER} [s]) = \text{[NUMBER} [s]\]
• Conflicting FSs: \(\text{[NUMBER} [s] \cup \text{[NUMBER} [p]) = \text{Fail}\)
• Merging with an unspecified FS: \(\text{[NUMBER} [s] \cup \text{[]}) = \text{[NUMBER} [s]\]

Unification (cont.)

• Merging FSs with different features specified:

\[
\text{[NUMBER} [s] \cup \text{[PERSON} [3]) = \text{[NUMBER} [s] \cup \text{PERSON} [3]
\]

• More examples:

\[
\begin{bmatrix}
\text{CAT} & \text{NP} \\
\text{AGR} & \text{NUM} [s]
\end{bmatrix} \cup \begin{bmatrix}
\text{AGR} & \text{NUM} [s]
\end{bmatrix} = \begin{bmatrix}
\text{CAT} & \text{NP} \\
\text{AGR} & \text{NUM} [s]
\end{bmatrix}
\]

\[
\begin{bmatrix}
\text{agr} & \text{num} [s]
\end{bmatrix} \cup \begin{bmatrix}
\text{agr} & \text{num} [s]
\end{bmatrix} = \begin{bmatrix}
\text{agr} & \text{num} [s]
\end{bmatrix}
\]

Unification with Reentrancies

• Remember that structure-sharing means they are the same object:

\[
\begin{bmatrix}
\text{AGR} & \text{NUM} [s] \\
\text{SUBJ} & \text{AGR} [\text{PER} [3]]
\end{bmatrix} \cup \begin{bmatrix}
\text{SUBJ} & \text{AGR} [\text{PER} [3]]
\end{bmatrix} = \begin{bmatrix}
\text{AGR} & \text{NUM} [s] \\
\text{SUBJ} & \text{AGR} [\text{PER} [3]]
\end{bmatrix}
\]

• When unification takes place, shared values are copied over:

\[
\begin{bmatrix}
\text{AGR} & \text{NUM} [s] \\
\text{SUBJ} & \text{AGR} [\text{PER} [3]]
\end{bmatrix} \cup \begin{bmatrix}
\text{SUBJ} & \text{AGR} [\text{PER} [3]]
\end{bmatrix} = \begin{bmatrix}
\text{AGR} & \text{NUM} [s] \\
\text{SUBJ} & \text{AGR} [\text{PER} [3]]
\end{bmatrix}
\]
Unification with Reentrancies (cont.)

• And remember that having similar values is not the same as structure-sharing:

\[
\begin{align*}
\text{agr} &\quad \text{num}\quad \text{sg} \\
\text{subj} &\quad \text{agr} \quad \text{num}\quad \text{sg}
\end{align*}
\]

\[
\begin{align*}
\text{subj} &\quad \text{agr} \quad \text{num}\quad \text{sg} \\
\text{agr} &\quad \text{num}\quad \text{sg} \\
\text{subj} &\quad \text{agr} \quad \text{num}\quad \text{sg}
\end{align*}
\]  

\[
\sqcup
\begin{align*}
\text{subj} &\quad \text{agr} \quad \text{num}\quad \text{sg} \\
\text{agr} &\quad \text{num}\quad \text{sg} \\
\text{subj} &\quad \text{agr} \quad \text{num}\quad \text{sg}
\end{align*}
\]  

\[
\begin{align*}
\text{agr} &\quad \text{num}\quad \text{sg} \\
\text{subj} &\quad \text{agr} \quad \text{num}\quad \text{sg} \\
\text{agr} &\quad \text{num}\quad \text{sg} \\
\text{subj} &\quad \text{agr} \quad \text{num}\quad \text{sg}
\end{align*}
\]  

\[
\sqcup
\begin{align*}
\text{subj} &\quad \text{agr} \quad \text{num}\quad \text{sg} \\
\text{agr} &\quad \text{num}\quad \text{sg} \\
\text{subj} &\quad \text{agr} \quad \text{num}\quad \text{sg}
\end{align*}
\]  

\[
\begin{align*}
\text{agr} &\quad \text{num}\quad \text{sg} \\
\text{subj} &\quad \text{agr} \quad \text{num}\quad \text{sg} \\
\text{agr} &\quad \text{num}\quad \text{sg} \\
\text{subj} &\quad \text{agr} \quad \text{num}\quad \text{sg}
\end{align*}
\]  

19

• With structure-sharing, you have to make sure the values are compatible everywhere that structure-sharing is specified:

\[
\begin{align*}
\text{agr} &\quad \text{num}\quad \text{sg} \\
\text{subj} &\quad \text{agr} \quad \text{num}\quad \text{sg} \\
\text{agr} &\quad \text{num}\quad \text{sg} \\
\text{subj} &\quad \text{agr} \quad \text{num}\quad \text{sg}
\end{align*}
\]  

\[
\sqcup
\begin{align*}
\text{subj} &\quad \text{agr} \quad \text{num}\quad \text{sg} \\
\text{agr} &\quad \text{num}\quad \text{sg} \\
\text{subj} &\quad \text{agr} \quad \text{num}\quad \text{sg}
\end{align*}
\]  

\[
\begin{align*}
\text{agr} &\quad \text{num}\quad \text{sg} \\
\text{subj} &\quad \text{agr} \quad \text{num}\quad \text{sg} \\
\text{agr} &\quad \text{num}\quad \text{sg} \\
\text{subj} &\quad \text{agr} \quad \text{num}\quad \text{sg}
\end{align*}
\]  

\[
\sqcup
\begin{align*}
\text{subj} &\quad \text{agr} \quad \text{num}\quad \text{sg} \\
\text{agr} &\quad \text{num}\quad \text{sg} \\
\text{subj} &\quad \text{agr} \quad \text{num}\quad \text{sg}
\end{align*}
\]  

\[
\begin{align*}
\text{agr} &\quad \text{num}\quad \text{sg} \\
\text{subj} &\quad \text{agr} \quad \text{num}\quad \text{sg} \\
\text{agr} &\quad \text{num}\quad \text{sg} \\
\text{subj} &\quad \text{agr} \quad \text{num}\quad \text{sg}
\end{align*}
\]  

20

Subsumption

We can say that a more general feature structure (less values specified) subsumes a more specific feature structure:

(1) [num sg]
(2) [per 3]
(3) [num sg]

So, we have the following subsumption relations, where:

• (1) subsumes (3)
• (2) subsumes (3)
• (1) does not subsume (2), and (2) does not subsume (1)

Implementing Unification

How do we implement a check on unification?

• i.e., given feature structures $F_1$ and $F_2$, return $F$, the unification of $F_1$ and $F_2$

Unification is a recursive operation:

• If a feature has an atomic value, see if the other FS has that feature with the same value
  – $[F_a]$ unifies with $[,]$, and $[F_a]$

• If a feature has a complex value, follow the paths to see if they’re compatible and have the same values at bottom
  – Does $[F G1]$ unify with $[F G2]$? We have to inspect $G1$ and $G2$ to find out.

• To avoid cycles, do an occur check to see if we’ve seen a FS before or not

The need for unification

Let’s say that we hypothesize that we have:

• a verb which selects for a 3rd person singular noun subject
• a subject which is 2nd person singular

What the verb specifies for the subject has to be able to unify with what the subject is

• In this case, unification will fail (person feature doesn’t unify)

Unification-based grammars (Grammars with feature structures)

One way to encode features is to augment a CFG skeleton with feature structure path equations

• CFG skeleton
  $S \rightarrow NP\ VP$

• Path equations
  $(NP\ AGREEMENT) = (VP\ AGREEMENT)$

Conditions:

1. There can be zero or more path equations for each rule skeleton $\rightarrow$ no longer atomic
2. When a path equation references constituents, they can only be constituents from the CFG rule

Handling Linguistic Phenomena

Well look at 3 different phenomena that feature-based, or unification-based, grammars capture fairly succinctly:

1. Agreement
2. Subcategorization
3. Long-distance dependencies
1) Agreement in Feature-based Grammars

One way to capture agreement rules:

\[
S \rightarrow NP \ VP \ (S \text{ head}) = (VP \text{ head}) \\
(NP \text{ head agr}) = (VP \text{ head agr}) \\
VP \rightarrow V \ NP \\
(VP \text{ head}) = (V \text{ head}) \\
NP \rightarrow D \ Nom(nal) \\
(NP \text{ head}) = (Nom \text{ head}) \\
(Det \text{ head agr}) = (Nom \text{ head agr}) \\
Nom \rightarrow Noun \\
(Nom \text{ head}) = (Noun \text{ head}) \\
(Nom \text{ head agr num}) = pl \\
\]

2) Subcategorization

We could specify subcategorization like so:

\[
VP \rightarrow V \ NP \\
(V \text{ head}) = (V \text{ head}) \\
(V \text{ subcat}) = \text{intrans} \\
VP \rightarrow V \ NP \\
(V \text{ head}) = (V \text{ head}) \\
(V \text{ subcat}) = \text{trans} \\
VP \rightarrow V \ NP \\
(V \text{ head}) = (V \text{ head}) \\
(V \text{ subcat}) = \text{ditrans} \\
\]

But values like intrans do not correspond to anything that the rules actually look like

• To make subcat better match the rules, we can make its value a list of a verb's arguments, e.g. \(<\text{NP,PP}>\)

Subcategorization rules

\[
VP \rightarrow V \ NP \ PP \\
(VP \text{ head}) = (V \text{ head}) \\
(V \text{ subcat}) = <\text{NP, NP, PP}> \\
V \rightarrow \text{leaves} \\
(V \text{ head agr num}) = \text{sg} \\
(V \text{ subcat}) = <\text{NP, NP, PP}> \\
\]

There is also a longer, more formal way to specify lists:

\[
<\text{NP,PP}> \text{ is equivalent to:} \\
\begin{bmatrix}
\text{first} & \text{NP} \\
\text{rest} & \text{first} & \text{PP} \\
\text{rest} & \text{rest} & ()
\end{bmatrix}
\]
Handling Subcategorization

How do we ensure that an object’s subcategorization list corresponds to what we see in the actual tree?

- We need a subcategorization principle

Roughly speaking, as a tree is built, items are checked off of the subcategorization list.

- The subcategorization list must be empty at the top of a tree.

In the previous example, if we had used the rule $VP \rightarrow V \ NP$, then we would have been left with $\text{subcat} <\text{NP,PP}>$.

- Or, if we had some rule like $VP \rightarrow V \ NP \ PP \ PP$, then our subcategorization list at the top would be $<\text{NP}>$, but with a PP left over.

3) Long-distance dependencies

Long-distance dependencies are often also called movement phenomena.

- Topicalization: John she likes.
- Wh-questions: Who does she like?

But we want to capture this without movement instead, well pass features along the tree.

- Bottom: introduce a trace
- Middle: pass the trace
- Top: unify the features of the trace with some real word (e.g., John or Who above).

Well use a gap feature for this.

Altering a chart parser to handle unification

Our grammar still has a context-free backbone, so we could just parse a sentence with a CFG and use the features to filter out the ungrammatical sentences.

- But by utilizing unification as we parse, we can eliminate parses that won’t work in the end.
  - e.g., we’ll eliminate NPs that don’t match in agreement features with their VPs as we parse, instead of ruling them out later.

What’s going on

- Traces, or gaps, are allowed as items from subcategorization lists.
- When we introduce a trace, we need to make sure it gets checked off the subcategorization list, so everything works out with the subcategorization principle.
- Alternate way: the gap value of a mother of a rule is the union of the daughter’s gap values.
  - So, we wouldn’t have to write separate rules for RelClause, Nom, NP, etc.
  - When a subcategorization list is empty and there is an item which matches something in the gap set, we can remove it from gap.
**Changes to the chart representation**

Each state will be extended to include the LHS feature structure (FS), which can get augmented as it goes along

- i.e., Add a feature structure (in DAG form) to each state
  - So, $S \rightarrow NP \ VP \ [0,0]$
  - Becomes $S \rightarrow NP \ VP \ [0,0], \ FS_S$

The predictor, scanner, and completer have to pass the FS, so all three operations have to be altered

**Earley parser with atomic categories**

**Prediction**

for each $i[A \rightarrow \alpha B \beta]$ in chart
for each $B' \rightarrow \gamma$ in rules
add $j[A \rightarrow \alpha \ B \beta, \gamma]$ to chart

**Completion (fundamental rule of chart parsing):**

for each $i[A \rightarrow \alpha B \beta]$ and $k[B' \rightarrow \gamma]$ in chart
add $i[A \rightarrow \alpha \ B \beta, \gamma]$ to chart

**Earley parser with unification**

**Prediction**

for each $i[A \rightarrow \alpha B \beta]$ in chart
for each $B' \rightarrow \gamma$ in rules
add $j[A \rightarrow \alpha \ B \beta, \gamma]$ to chart

**Completion (fundamental rule of chart parsing):**

for each $i[A \rightarrow \alpha B \beta]$ and $k[B' \rightarrow \gamma]$ in chart
add $i[A \rightarrow \alpha \ B \beta, \gamma]$ to chart

**Prediction**

for each $i[A \rightarrow \alpha B \beta]$ in chart
for each $B' \rightarrow \gamma$ in rules
add $j[A \rightarrow \alpha \ B \beta, \gamma]$ to chart

The predictor takes the specification of $B$ (i.e., FS) and finds the most general unifier (mgu) of $B$ with $B'$

- If $B$ and $B'$ do not unify, then the rule for $B'$ is not added to the chart
- Initially (i.e., at position 0), all that happens is that a dotted rule with a FS is added to the chart

**How to use a chart with feature structures**

- Use unification to combine categories in completion or prediction.
- Each time a rule or an edge is used, a new copy is made.
- But how about testing whether an entry already exists in the chart?
  - Currently, we simply check to see whether a state unifies with something already in the chart and do not add a new state if it is already there
  - But a more specific or a more general state may already be in the chart
The subsumption problem (based on Covington 1994)

- S → NP VP
- NP → Det N
- VP → V'(0)
- VP → V'(X) Comps(X)
- V'(X) → V(X)
- V'(X) → Adv V(X)
- Comps(1) → NP
- Comps(2) → NP NP
- Det → the
- N → dog
- N → cat
- Adv → often
- V(0) → sings
- V(1) → chases
- V(2) → gives

The subsumption problem (2)

What happens when we try to parse the dog chases the cat?

- At position 2 (between dog and chases), from 2 to 2, the parser predicts:
  - VP → • V'(0)
  - V'(0) → • V(0)
  - V'(0) → • Adv V(0)
  - VP → • V'(X) Comps(X)

- What happens when we scan chases?
  - We have a passive V(1) edge
  - But there is no predicted V'(1) edge—only V'(0)

Using subsumption to check the chart

Subsumption check: Do not add a state to the chart if an equivalent or more general state is already there.

- So, if we want to add a singular determiner state at [x, y], and the chart already has a determiner state at [x, y] unspecified for number, then we do not add it.

- If we don’t impose a subsumption restriction, we could add two states at [x, y], one expecting to see a singular determiner, the other just a determiner.
  - On seeing a singular determiner, the parser advances the dot on both rules, creating two edges (since singular unifies with singular and with unspecified).
  - As a result, we would get duplicate edges.

- With subsumption, if either a singular or plural determiner is encountered, we advance the dot, creating only one edge (singular or plural) at [x, y]

Checking for subsumption

Case 1

Let’s define a function subsumes_chk which takes 2 arguments: more general item & more specific item

No variables:
- subsumes_chk(V'(1), V'(1)). → yes
- subsumes_chk(V'(1), V'(2)). → no

Compound terms without variables are either identical or different, i.e., here: subsumption = unification

Case 2

Variables only in more general term:
- subsumes_chk(V'(X), V'(1)). → yes
- subsumes_chk(foo(X,X), foo(1,1)). → yes
- subsumes_chk(foo(X,X), foo(1,2)). → no

Succeeds if a consistent variable assignment exists, i.e., here: subsumption = unification

Case 3

Variables in both terms:
- subsumes_chk(vbar(X), vbar(Y)). → yes
- subsumes_chk(vbar(X), vbar(foo(1,Y))). → yes
- subsumes_chk(vbar(foo(1,2)), vbar(foo(1,Y))). → no

Succeeds if terms can be unified without further instantiating more specific term; in other words:
- Unification should not require a particular instantiation of a variable in the more specific term.

- Idea: Identify each variable in more specific term with a unique, variable-free term; then subsumption = unification.
The restriction problem

Shieber et al 1995: Grammar accepting $ab^n$ with $N$ being instantiatied to the successor representation of $n$.

\[
\begin{align*}
\text{start} & \rightarrow r(0, N) \\
r(X, N) & \rightarrow r(s(X), N) b \\
r(N, N) & \rightarrow a
\end{align*}
\]

Prediction step with unification will loop:

1. \[ o[start \rightarrow \bullet, r(0, N)] \]
2. \[ o[r(0, N) \rightarrow \bullet, r(s(0), N) b] \]
3. \[ o[r(s(0), N) \rightarrow \bullet, r(s(s(0))), N) b] \]
4. \[ o[r(s(s(0)), N) \rightarrow \bullet, r(s(s(s(0))), N) b] \]
5. \[ o[r(s(s(s(0))), N) \rightarrow \bullet, r(s(s(s(s(0)))), N) b] \]

Using restriction to prevent prediction loops

- Prediction terminates for grammars with atomic categories, since a new item is only added to the chart if not already there and there is a finite number of atomic categories.
- Moving beyond atomic categories, there can be an infinite number of non-atomic categories.
- Prediction loop on left-recursive rules can be problem again.
- Solution: restrict number of predicted categories to finitely many cases

Prediction with restriction

for each \([A \rightarrow \alpha \bullet, B \beta]\) in chart
for each \([B' \rightarrow \gamma]\) in rules
add \([\sigma(B \rightarrow \bullet, \gamma)]\) with \(\sigma = \text{restriction}(\text{mgu}(B, B'))\) to chart

\text{restriction}(\text{mgu}(B, B'))\) can be any operation reducing the number of possible substitutions to finite classes:
- depth bound on term complexity
- elimination of terms that are known to grow indefinitely
- use only of selected terms known not to grow indefinitely

This is sound since prediction only creates a hypothesis to be completed!

Type hierarchies

Let's get back to our feature structure formalism ...

- Instead of simple feature structures, formalisms like Head-Driven Phrase Structure Grammar (HPSG) use typed feature structures
- Two problems right now:
  - What prevents us right now from specifying: \([\text{NUM fem}]\)?
  - How can we capture the fact that all values of \text{NUM} are the same sort of thing, i.e., make a generalization?

Solution: use \textbf{types}

Using restriction to prevent prediction loops

- Prediction terminates for grammars with atomic categories, since a new item is only added to the chart if not already there and there is a finite number of atomic categories.
- Moving beyond atomic categories, there can be an infinite number of non-atomic categories.
- Prediction loop on left-recursive rules can be problem again.
- Solution: restrict number of predicted categories to finitely many cases

Example

Grammar:

\[
\begin{align*}
\text{start} & \rightarrow r(0, N) \\
r(X, N) & \rightarrow r(s(X), N) b \\
r(N, N) & \rightarrow a
\end{align*}
\]

Parsing using a restrictor that replaces every term deeper than 2 with a variable:

1. \[ o[start \rightarrow \bullet, r(0, N)] \]
2. \[ o[r(0, N) \rightarrow \bullet, r(s(0), N) b] \]
3. \[ o[r(s(0), N) \rightarrow \bullet, r(s(s(0))), N) b] \]
4. \[ o[r(s(s(0))), N) \rightarrow \bullet, r(s(s(s(0))), N) b] \]
5. \[ o[r(s(s(s(0))), N) \rightarrow \bullet, r(s(s(s(s(0)))), N) b] \]

Type systems

1. Each feature structure is labeled by a type. \([\text{noun} \rightarrow \text{[CASE case]}]\)
2. Each type has appropriateness conditions specifying what features are appropriate for it.
   - \text{noun} \rightarrow \text{[CASE case]}
   - \text{verb} \rightarrow \text{[VPFORM vform]}
3. Types are organized into a type hierarchy.
4. Unification is modified to allow two different types to unify.
Type hierarchy

If we have specified:
• \textit{CASE} is appropriate for \textit{noun}
• the value of \textit{CASE} is \textit{case}
• we have the following type hierarchy:
  \texttt{nom acc dat}

Then, the following are possible feature structures:
\[
\begin{bmatrix}
\texttt{noun} \\
\texttt{CASE nom}
\end{bmatrix}
\begin{bmatrix}
\texttt{noun} \\
\texttt{CASE acc}
\end{bmatrix}
\begin{bmatrix}
\texttt{noun} \\
\texttt{CASE dat}
\end{bmatrix}
\]

Unification of types

Now, when we unify feature structures, we have to unify types, too:
• \[\texttt{case case} \sqcup \texttt{case nom} = \texttt{case nom}\]
• \[\texttt{case nom} \sqcup \texttt{case acc} = \text{FAIL}\]

If we specify that \texttt{acc} and \texttt{dat} have a common subtype, \texttt{obj}
• Then, we have the following unification:
  \[\texttt{case acc} \sqcup \texttt{case dat} = \texttt{case obj}\]