# Combinatory Categorial Grammar & the OpenCCG platform

Geert-Jan M. Kruijff

Language Technology Lab
DFKI GmbH

gj@dfki.de / http://www.dfki.de/~gj

- Goal

  "To present the intuitions behind Combinatory Categorial Grammar as a framework for relating the structure of a linguistic expression to the meaning it realizes; and to illustrate a development platform for CCG."

- Overview

  1. CCG & OpenCCG in the larger context of grammar engineering
  2. CCG: Analysing the structure of expressions
  3. CCG: Adding (linguistic) meaning to structure
  4. OpenCCG: Organization and implementation of grammars

- Combinatory Categorial Grammar
  - *Mildly-context sensitive* grammar framework, with provable polynomial parseability
  - Capable of handling a large range of linguistic phenomena, across a broa spectrum of language types
  - Perspicuous interface between structure ('syntax') and meaning ('semant
- OpenCCG
  - Platform for efficient parsing and realization with CCG grammars
  - Computational lexicon facilitating inheritance over syntactic and semantic common structure, enabling reuse within a single grammar and across
  - Statistics-based realization; development to include StatCCG for parsing

- Grammar engineering

  "The effort to develop natural language grammars and computational platforms for broad-coverage, robust, and efficient *interpretation* and *production* of linguistic utterances"

- Resource reuse in real-life grammar engineering
  - Large-scale grammars, moving towards multi-lingual resource reuse
  - Statistical NLP for acquiring, and processing with, large grammars
  - **But**: between broad-coverage syntax and semantics; domain-specific interpretation in and beyond NL grammar; complexity
- Various available platforms

- Formal systems
  - ... usually talk about structured objects, the *resources*
  - ... to collections of which they apply operations
  - ... in order to determine some global properties of these collections
- Resource sensitivity governs the *way* in which a system can use its resources:
  - How often resources can be used,
  - When they can be assembled into more complex resources,
  - How they can be modified

# Part I

## A combinatory categorial approach to analysing the structure of expressions

In the lexicon, words are assigned *categories*

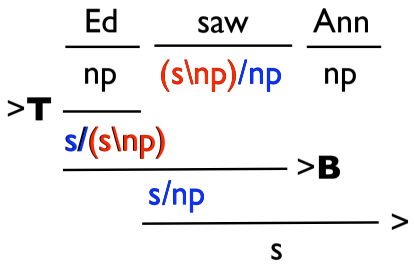
- A category specifies the grammatical use (*syntactic behavior*) of the word
- A category can be *atomic*, or a *function* using "slashes" \ (left) and / (right)
- Functional categories are written "result first": $RES \backslash ARG_1 \ldots \backslash ARG_n$

Examples

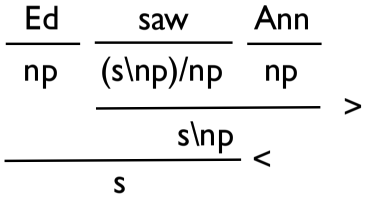
- Nouns: N
- Adjectives: N / N
- Determiners: NP / N
- Transitive verbs: (S \ NP) / NP, also written as S \ NP / NP

- Structured objects are linguistic signs
  - Monostratal, multi-level representation
- Grammatical composition is resource sensitive
  - Signs can usually be used only *once* (no arbitrary copying, wasting)
  - Signs cannot be arbitrarily combined or arranged (*linearization, dependency*)
  - Control over when operations defining grammatical composition can be applied makes it possible to avoid a collapse to "anything goes"

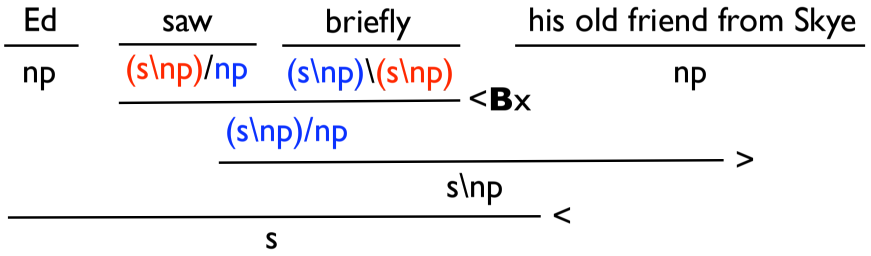
- CCG adds further rules, based on the combinators of Curry & Feys' combinatory logic
  - Forward type raising (>**T**): X ⇒ Y/(Y\X)
  - Forward harmonic composition (>**B**): X/Y Y/Z ⇒ X/Z
- These rules induce **associativity**:



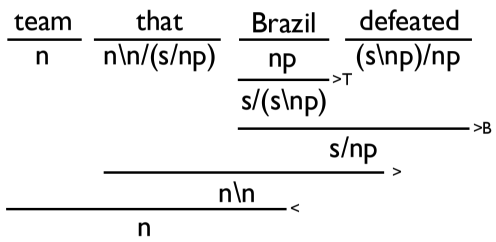

- Basic rules of combination
  - Forward application (>): X/Y Y ⇒ X
  - Backward application (<): Y X\Y ⇒ X
- Example derivation



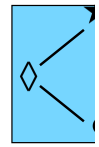

- **Permutation**-inducing rules based on the composition combinator **B**
  - Forward crossed composition (>**B**x): X/Y Y\Z ⇒ X\Z
  - Backward crossed composition (<**B**x): Y/Z X\Y ⇒ X/Z
- <**B**x needed for e.g. heavy-NP shift:




- Associativity in the structure of language
  - Associativity introduces "flexible constituency", non-associativity strict domains
  - Evidence of associativity can be found in extraction, coordination, the interaction of intonational structure and syntactic structure, ...
- Example derivation

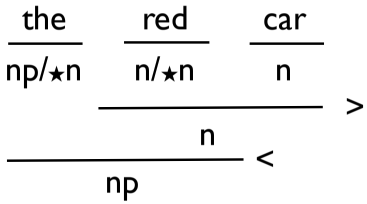


- Modal control over access to combinators
  - Modal marking *m* on connectives: $\backslash_m$ and $/_m$
  - Combinators are defined for particular modalized connectives only
  - Hierarchy over modals to inherit combinator accessibility (behavior)
- Lexically specified derivational control
  - The lexicon completely determines structure building
    - If a modalized connective, controlling the applicability of a rule, is absent from the lexicon, then the rule can never be applied
    - No more need for *ad hoc* constraints on rule applicability
  - Universal set of combinator rules: *Combinators as formal universals*

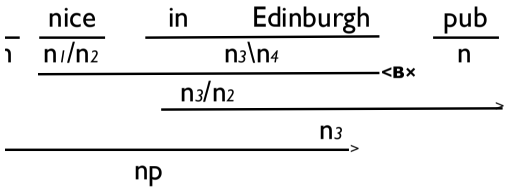


- Controlled through ★
- All modes inherit from ★, i.e. all have access to application

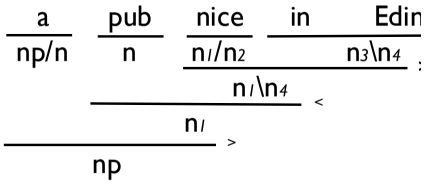
(>) $X /_\star Y \; Y \Rightarrow X$ (<) $Y \; X \backslash_\star Y \Rightarrow X$



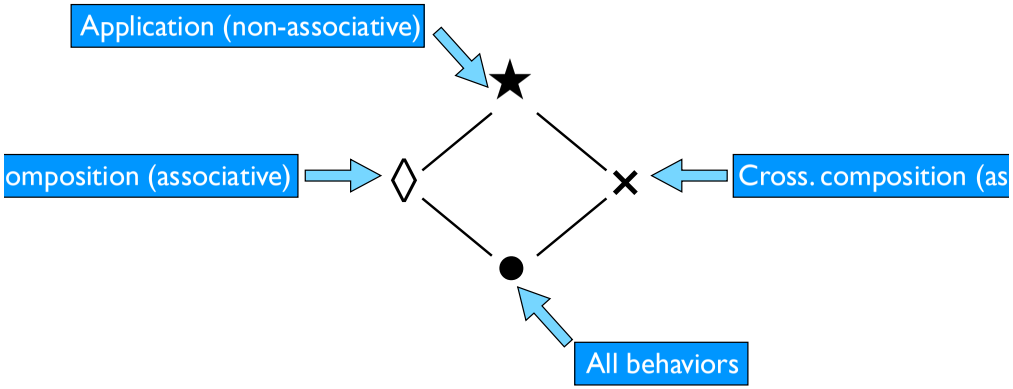

- Crossed composition allows for ungrammatical permutations



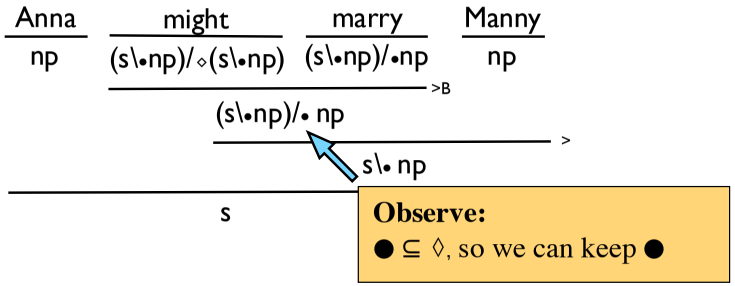

English-specific instantiation of <**B**×:
*English backward crossing composition* <**B**×:
Y/Z X\Y ⇒**B** X/Z *where* X=Y=s\$

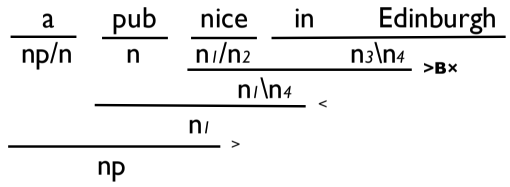


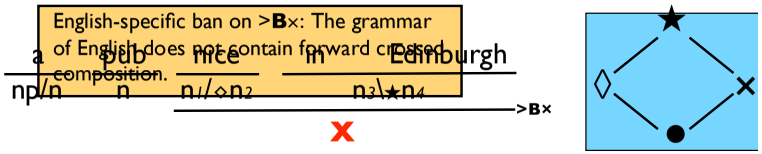

English-specific ban on >**B**×: The grammar of English does not contain forward crossed composition.



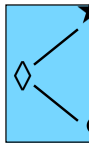

'he modalities *restrict* the upper limit, i.e. the strongest rule(s) tha an be applied. If a category contains modalities that are stronger ıan required for a rule, these modalities can be carried over in the esult ("no power loss").



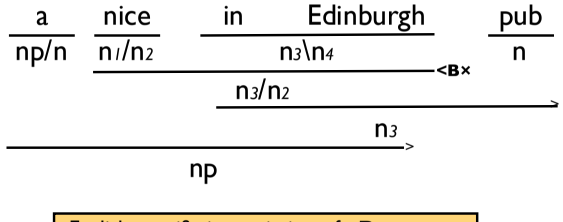

- No need to ban >**B**×
  - *man that John knew [that, s/◊s] [saw Ann, s\•np]
  - Complementizer “that” gets **B** mode ◊, which makes **B**× inapplicable



- Harmonic composition
  - Controlled through ⋄, i.e. accessible to • but not to ★
  - (>**B**) X/⋄Y Y/⋄Z ⇒B X/⋄Z (<**B**) Y\⋄Z X\⋄Y ⇒B X\⋄Z
- Examples

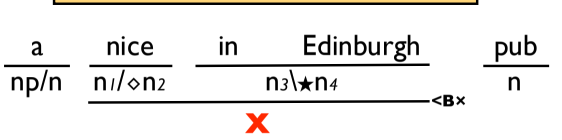
A/•B B/⋄C ⇒B A/•C ● ⊆ ⋄, observe that we remain with ● (not A/⋄

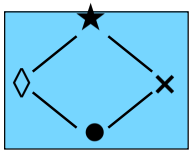
A/⋄B B/★C ⇏B A/?C ★ ⊈ ⋄, hence the rule is not applicable





English-specific instantiation of <**B**×:
*English backward crossing composition* <**B**×:
Y/Z X\Y ⇒**B** X/Z *where* X=Y=s\$



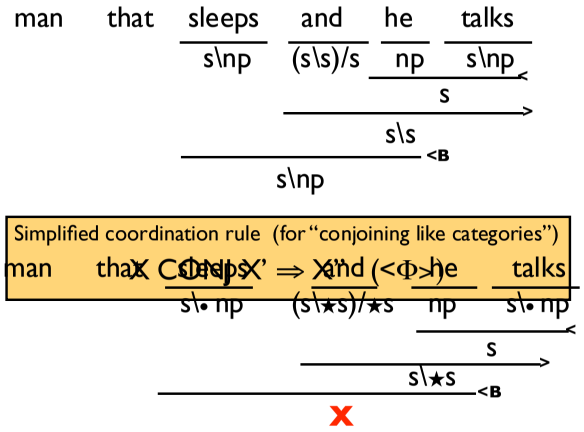
## Combinatory analysis

- Governed by Principle of Adjacency: String-based, not logic-based
- Building structures using combinatory rules

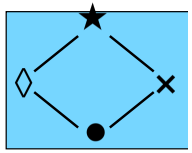
## Multi-modal combinatory analysis

- Applicability of combinatory rules is controlled through modalized slashes
- Control is thus *fully* lexicalized, the rule component is universal

- Modeling the complexity of meaning
  - Relational structure: relations indicate how parts contribute to the overa meaning of the construction (*interpretative import*)
  - Ontological richness: differentiation through ontological sorting of propositions
- *Linguistic* meaning
  - Meaning as far as linguistically realized (no inference, limited resolutior
  - Information structure: “Pragmatization of semantics”







- Category (not a rule) for coordination
  - [and, (X\★X)/★X]: * man that [sleeps, *s* \• *np*] [and he talks, s\★s]
  - Coordinator “and” gets most restrictive modes, to make **B** inapplicable

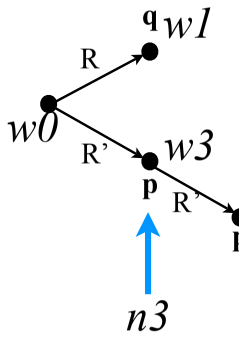
## Part 2

## Relating the structure of an expression to the meaning(s) it realizes

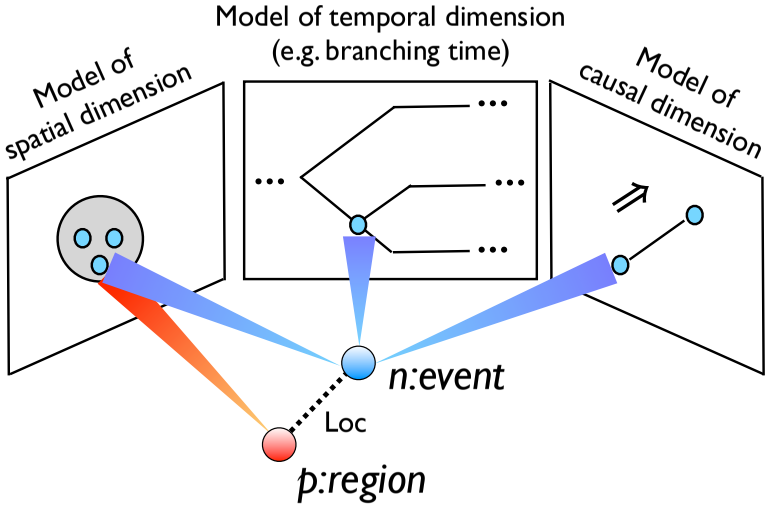
“The ability to speak does not make you intelligent.”

Qui-Gon Jinn to Obi-Wan Kenobi, talking about Jar Jar Binks
Star Wars Episode 1: The Phantom Menace

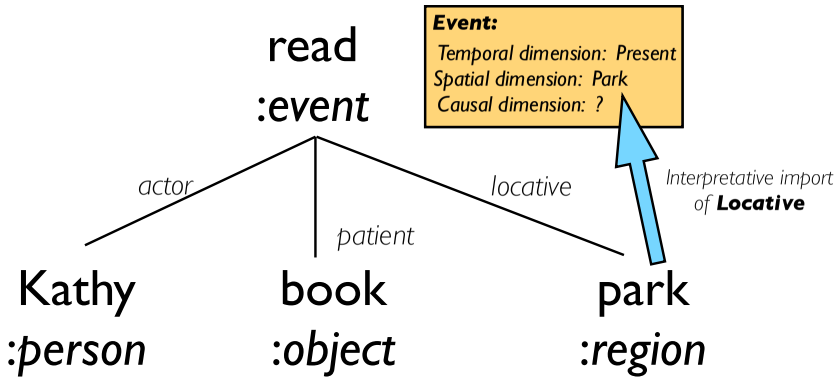
- Modal logic with a twist
  - Model theory: states, and accessibility relations between states
  - However, no reference to states *in the language*




- Hybrid logic adds state-reference
  - A nominal refers uniquely to a state in the model: @
  - Nominals are first class citizens in the language
  - `@n3◇p` means we have to evaluate at *w3*, not at *w0*
  - We can unify references: `@n m` means the nominals *n* and *m* refer to the same state in the model




"Kathy reads a book in the park"




- Reflection of ontological richness through nominals
  - Nominals can be assigned ontological sorts
  - Ontological sort of the state the nominal refers to
  - Model represents the interpretation of an ontology
- Richness through one ontology, or multiple ontologies?
  - Multiple ontologies require multiple, *linked* model theories
  - A nominal, through its sort, has reflections in different models
  - An ontology as *looking glass* on a particular aspect of meaning

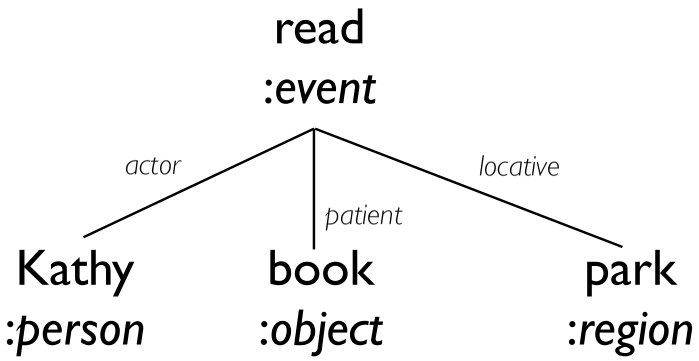
- Syntax/semantics-interface
  - Basic idea follows older proposals by Zeevat (1988), Hoffman (1995), ...
  - Co-index an argument in the meaning with the category that provides it

read

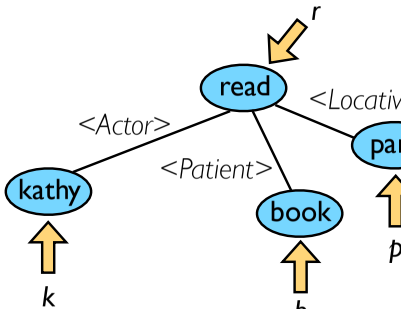
(S:h \ NP: a) / NP: b

@h **read** & @h<ACT>a & @h<PAT>b

- Constraints only get triggered through when an index on a category gets unified
  - Derivation over categories is the only source for triggering constraints




*r:event}* **read**
@r <Actor>*k* & @{k:*person*} **Kathy**
@r <Patient>*b* & @{b:*object*} **Book**
@r <Locative>*p* & @{p:*region*} **Park**




- Arguments and adjuncts have different behavior
  - The way the meaning of an argument contributes to that of the head specified by the head
  - Adjuncts use a specification that enables them to *insert* their meaning into that of the head they modify, *extending* the meaning of the head
- Adjunct meaning

yesterday

S:h \• S: h

@h p & @h <TWH>(y & **yesterday**)

  - No longer do adjuncts act like “predicates”: **yesterday**(**read**(**j,c**)), we obtain (retain) a proper relational structure

Ed

NP:e

@e **Ed**

reads

(S:r \• NP: a) /• NP:b

@r **read** & @r<ACT>a & @r<PAT>b

Mad

NP:m

@m **Mad**

(S:r \• NP: a)

@r **read** & @r<ACT>a & @r<PAT>m & @m **Mad**

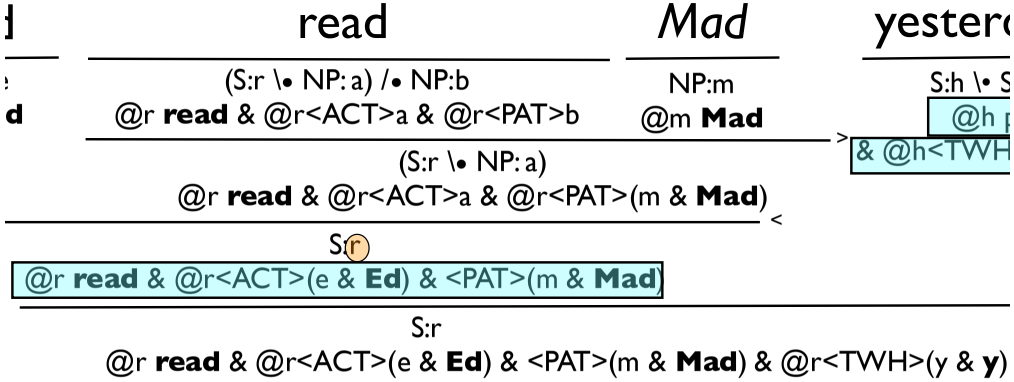
S:r

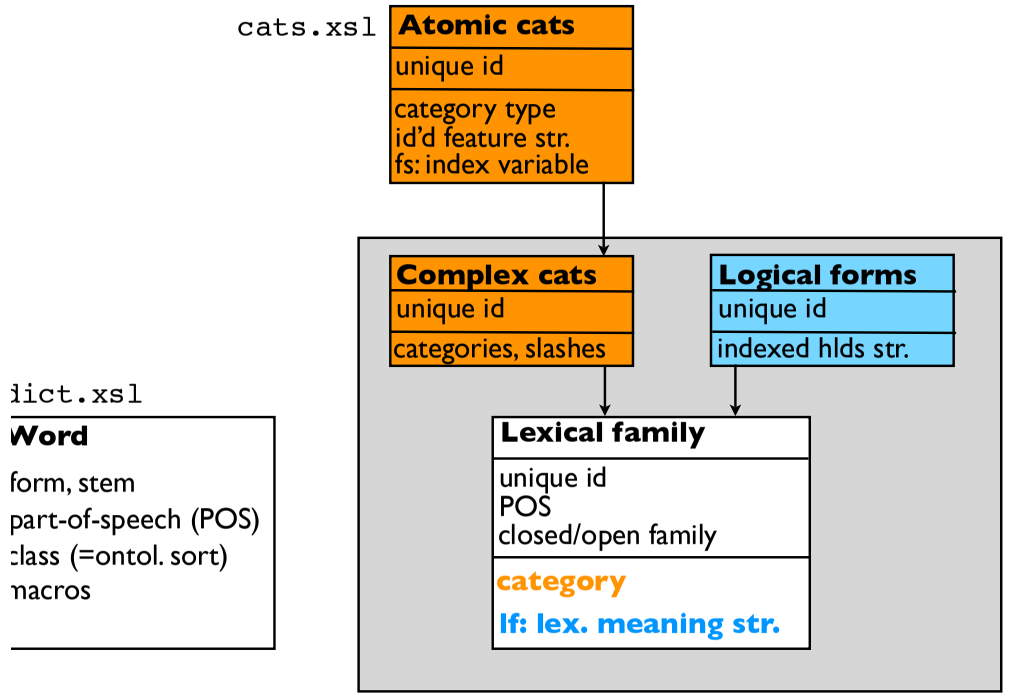
@r **read** & @r<ACT>e & @e **Ed** & <PAT>m & @m **Mad**

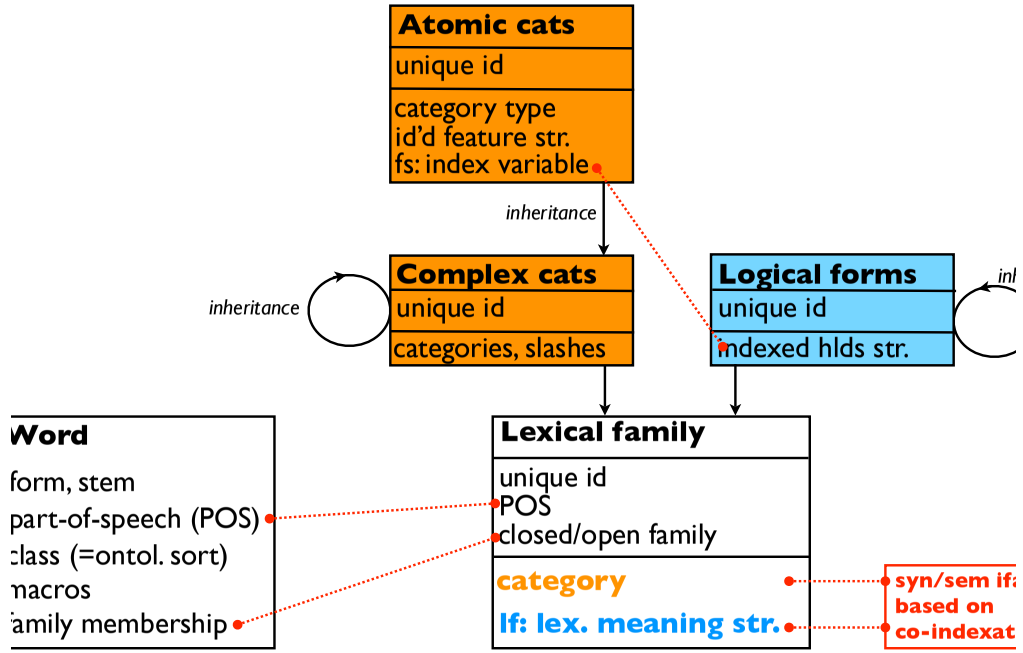
"Creative minds have always been know
to survive any kind of bad training

Anna Freu

*... Just in case you st*
*wondering about*
*joy of XSL transfor*



- Create an atomic category of type **n**

```
<xsl:variable name="n.1.X.default">
  <atomcat type="n">
    <fs id="1">
      <feat attr="num"><featvar name="NUM"/></feat>
      <feat attr="pers"><featvar name="PERS"/></feat>
      <feat attr="case"><featvar name="CASE"/></feat>
      <feat attr="index"><lf><nomvar name="X"/></lf></feat>
    </fs>
  </atomcat>
</xsl:variable>
```

- Note:
  - Feature structure has an id; this must be unique in a complex category
  - The feature structure also specifies X as the index variable (nominal) t be used in a lexical family to access the meaning of this noun

- "ball": noun,
  - category **n**,
  - @b:phys-obj(**ball**)
- "red": adjective,
  - category **n** /* **n**,
  - @x:phys-obj <Property> (r:color ^ **red**)
- "the": determiner,
  - category **np** /* **n**
  - @x:phys-obj(<Delimitation>**unique** ^ <Quantification>**specific_singu**

- Setting up the lexical family

```
<xsl:template name="add-n-families">


<family name="n.default" pos="N" closed="false">
  <entry name="default">
    <xsl:call-template name="extend">
      <xsl:with-param name="elt" select="xalan:nodeset($n.1.X.default)/
      <xsl:with-param name="ext" select="$X.Default"/>
    </xsl:call-template>
  </entry>
</family>

</xsl:template>
```

- Note:
  - Make sure that the family is specified within the **add-n-families** template

- Lexical meaning for a noun

```
<xsl:variable name="X.Default">
  <lf>
    <satop nomvar="X">
      <prop name="[*DEFAULT*]"/>
    </satop>
  </lf>
</xsl:variable>
```

- Note:
  - The nominal variable X needs to be co-indexed with the index on a category, for this meaning to be accessible
  - "[*DEFAULT*]" means using the word stem as proposition

- Specifying the entry for the word “ball”

```
<entry stem="ball" pos="N" class="thing">
      <word form="ball"  macros="@num.sg"/>
      <word form="balls" macros="@num.pl"/>
</entry>
```

- How about those macros?

```
<macro name="@num.sg">
       <fs id="1" attr="num" val="sg"/>
</macro>

<macro name="@num.pl">
      <fs id="1" attr="num" val="pl"/>
</macro>
```

  - Note: these macros only apply to feature structures with id 1!

- Imports:

```
<xsl:import href="./n.xsl"/>
<xsl:output indent="yes" xalan2:indent-amount="2"/>
<xsl:strip-space elements="*"/>
```

- Call the template to add the noun families

```
<xsl:call-template name="add-n-families"/>
```

- Note:
  - The provided lexicon-base.xsl file specifies where to put the above

- the resulting noun inherits the features from the argument

```
<xsl:variable name="n.from-1.default">
  <atomcat type="n">
    <fs inheritsFrom="1"/>
  </atomcat>
</xsl:variable>
```

  - a feature specified in the above fs overwrites the original fs material

- complex category structure

```
<xsl:variable name="adj.from-1.default">
  <complexcat>
    <xsl:copy-of select="$n.from-1.default"/>
    <slash dir="/" mode="*"/>
    <xsl:copy-of select="$n.1.X.default"/>
  </complexcat>
</xsl:variable>
```

- Building the resources
  - Use [ comsys-build ] in the grammar directory to build the resources
- Testing
  - Use [ tccg ] to load the grammar into OpenCCG
  - Type in “ball” at the prompt and see what happens!
  - Use [ testbed.xml ] for test suite; use [ ccg-test ] to do regression tests

## Basic family for adjectives

```
<family name="adj.property.default" pos="ADJ">
   <entry name="adj.property">
    <xsl:call-template name="extend">
      <xsl:with-param name="elt" select="xalan:nodeset($adj.from-1.default)
      <xsl:with-param name="ext" select="$P.default.T"/>
    </xsl:call-template>
   </entry>
</family>
```

## Lexical entry for the word “red”

```
<entry stem="red" pos="ADJ" class="color">
      <word form="red" macros=""/>
      <member-of family="adj.property.default"/>
</entry>
```

- The meaning of the adjective is added as property of the object, realized by the noun the adjective modifies

```
<xsl:variable name="P.default.X">
  <lf>
     <satop nomvar="X">
        <diamond mode="Property">
          <nomvar name="P"/>
          <prop name="[*DEFAULT*]"/>
        </diamond>
     </satop>
  </lf>
</xsl:variable>
```

- Note:
  - the X is the index of the object (interface specified in the **n** category)

## he lexical meaning of the (definite) determiner specifies unique lelimitation, about a specific singular object

```
xsl:variable name="D.DELIM.UNQ.QUANT.SPECSING.X">
 <lf>
    <satop nomvar="X">
       <diamond mode="Delimitation"><prop name="unique"/></diamond>
       <diamond mode="Quantification"><prop name="specific_singular"/></diam
    </satop>
 </lf>
/xsl:variable>
```

## Note:

- the X is the index of the object (interface specified in the **n** category)

- the resulting **np** inherits features from the **n** argument, and adds a “form” feature with value “definite” (for *the*)

```
<xsl:variable name="np.from-1.definite">
  <atomcat type="np">
    <fs inheritsFrom="1">
      <feat attr="form" val="definite"/>
    </fs>
  </atomcat>
</xsl:variable>
```

- complex category structure

```
<xsl:variable name="det.from-1.def.sg">
  <complexcat>
    <xsl:copy-of select="$np.from-1.definite"/>
    <slash dir="/" mode="*"/>
    <xsl:copy-of select="$n.1.X.default"/>
  </complexcat>
</xsl:variable>
```

- After building, **`tccg`** can parse “the red ball”

```
tccg> the red ball
1 parse found.

Parse: np{case=CASE, form=definite, index=T_4:thing, num=sg, pers=PERS} :
  @b1:thing(ball ^
        <Delimitation>unique ^
        <Quantification>specific_singular ^
        <Property>(r1:color ^ red))
```

- Add the following line to testbed.xml, and run **`ccg-test`**

```
<item numOfParses="1" string="the red ball"/>
```

## Lexical family

```
  <family name="determiner.definite" pos="DET" indexRel="Delimitation"
closed="true">
    <entry name="n.sg">
      <xsl:call-template name="extend">
        <xsl:with-param name="elt" select="xalan:nodeset($det.from-1.def.sg
        <xsl:with-param name="ext" select="$D.DELIM.UNQ.QUANT.SPECSING.X"/>
      </xsl:call-template>
    </entry>
  </family>
```

- the family is *closed* - it should only be applicable to definite determiners
- the feature **indexRel** directs the realizer to the Delimitation feature, there b no propositional content for the determiner

## Lexical entry for “the”

```
<entry stem="the" pos="DET">
      <word form="the" macros=""/>
      <member-of family="determiner.definite"/>
</entry>
```