


# 6.863J Natural Language Processing

## Lecture 11: From feature-based grammars to semantics



Robert C. Berwick  
berwick@ai.mit.edu

# The Menu Bar

- **Administrivia:**
  - Schedule alert: Lab 4 out Weds. Lab time today, tomorrow
  - Please read notes4.pdf!!
- *Agenda:*
- Feature-based grammars/parsing: unification; the question of representation
- Semantic interpretation via lambda calculus: syntax directed translation

# Features are everywhere



**morphology** of a single word:

Verb[head=thrill, tense=present, num=sing, person=3,...] → thrills

**projection** of features up to a bigger phrase

VP[head= $\alpha$ , tense= $\beta$ , num= $\gamma$ ...] → V[head= $\alpha$ , tense= $\beta$ , num= $\gamma$ ...] NP  
provided  $\alpha$  is in the set TRANSITIVE-VERBS

**agreement** between sister phrases:

S[head= $\alpha$ , tense= $\beta$ ] → NP[num= $\gamma$ ,...] VP[head= $\alpha$ , tense= $\beta$ , num= $\gamma$ ...]  
provided  $\alpha$  is in the set TRANSITIVE-VERBS

# Better approach to factoring linguistic knowledge

- Use the *superposition* idea: we superimpose one set of constraints on top of another:

1. Basic skeleton tree

2. Plus the added feature constraints

- S            →        NP                                  VP  
[num x]                [num x]                                  [num x]

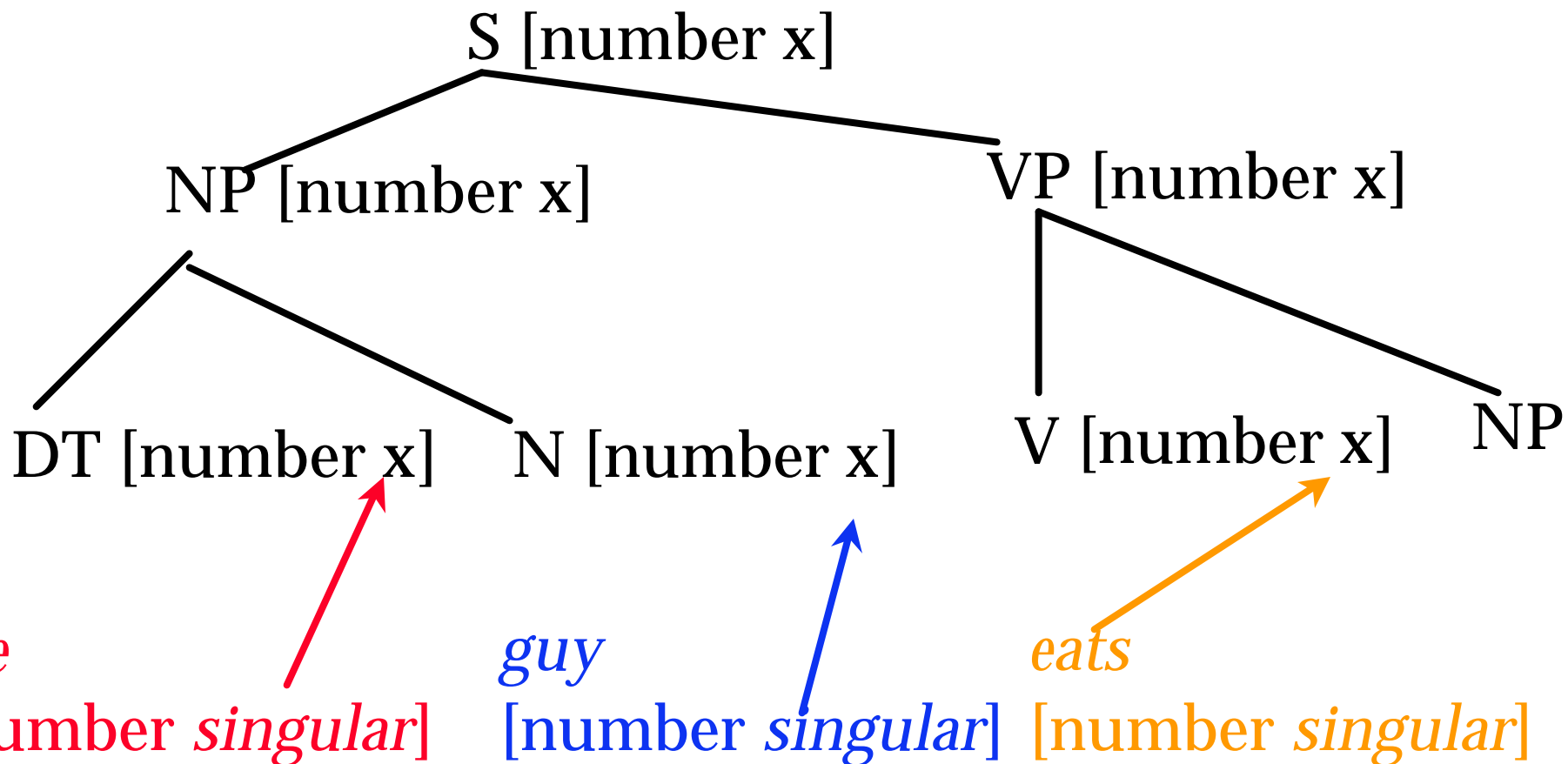
*the guy*

[num singular]

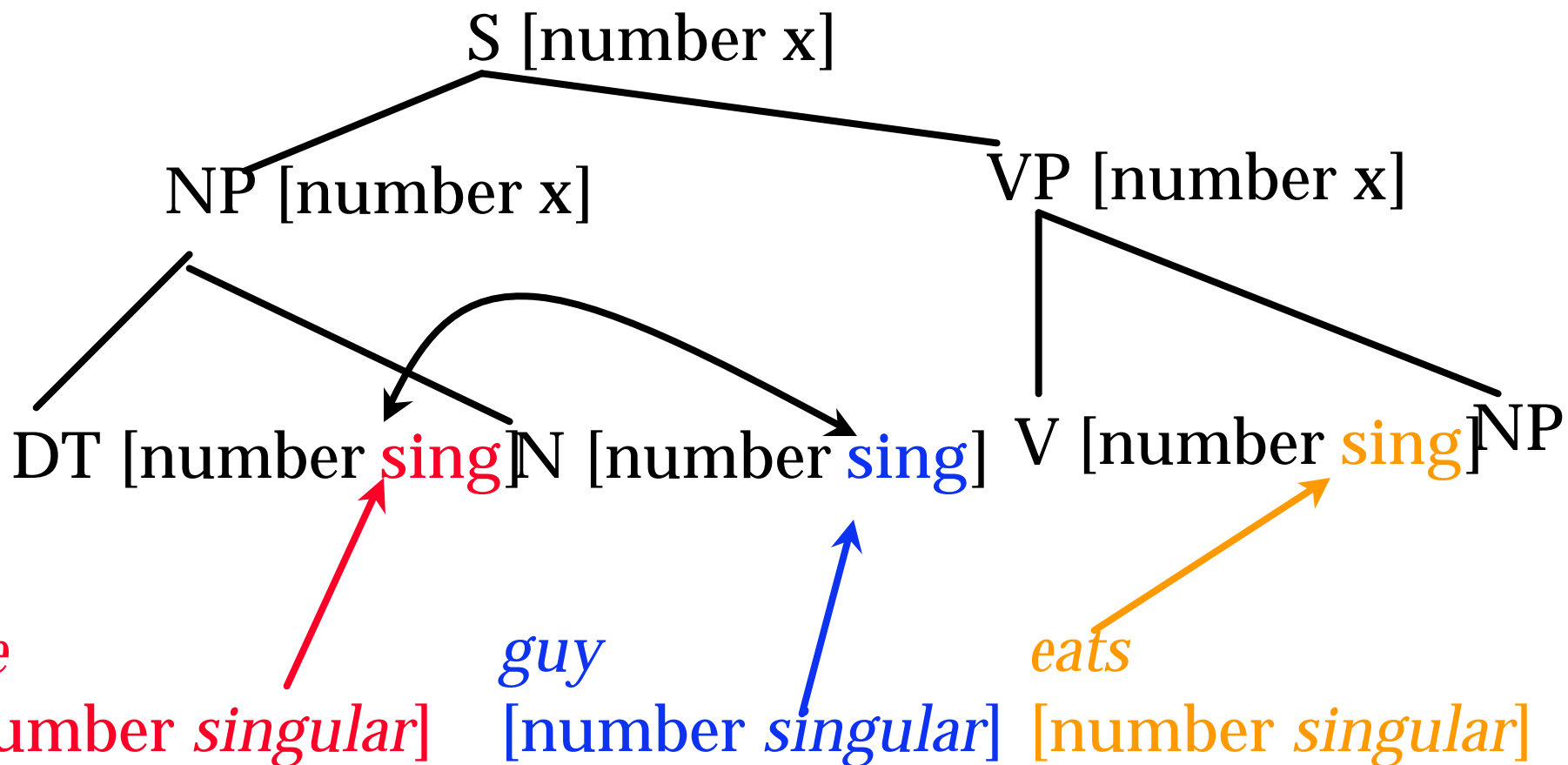
*eats*

[num singular]

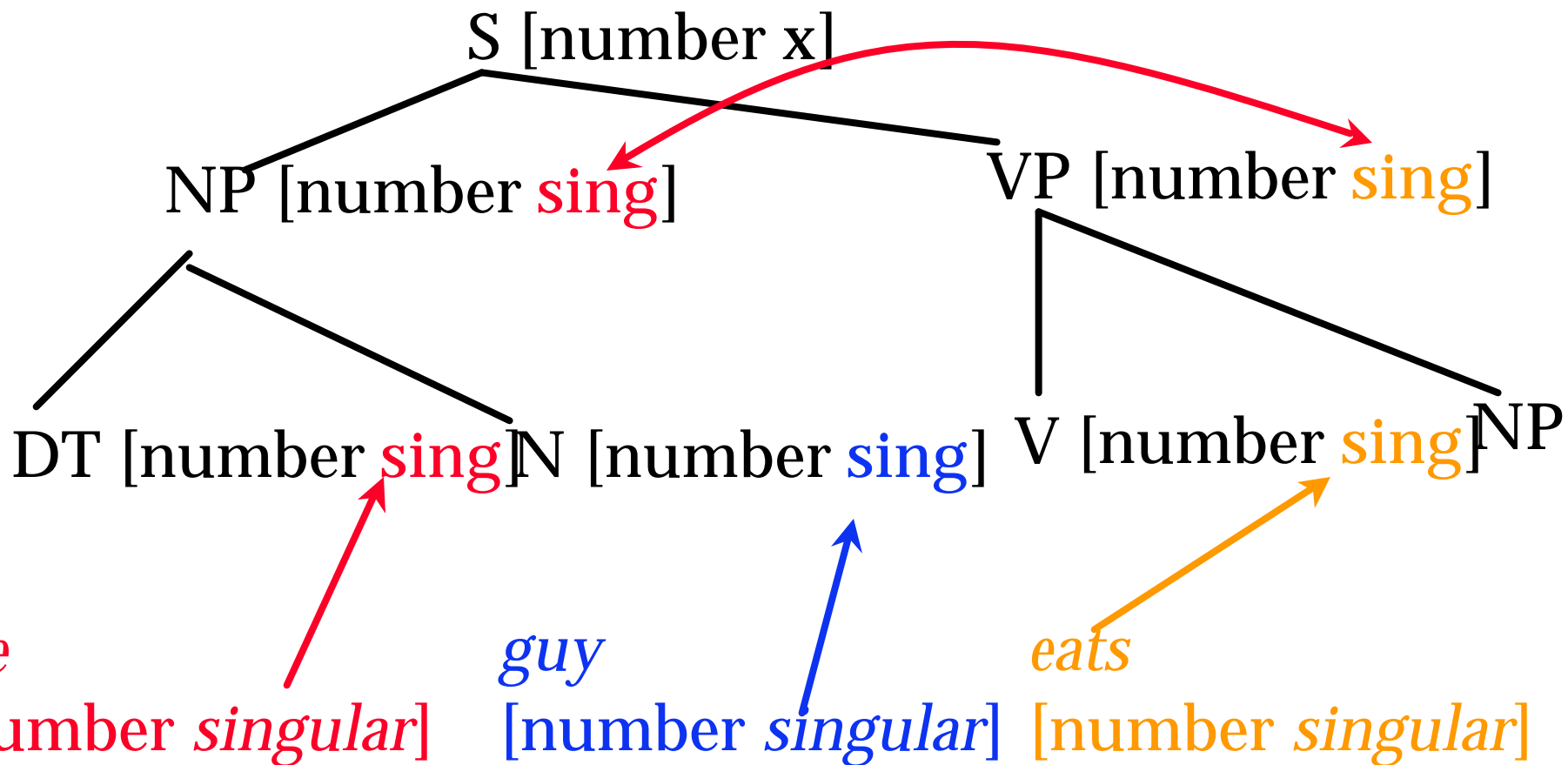
Or in tree form:



# Values trickle up



# Checking features



# What sort of power do we need here?

- We have [*feature value*] combinations so far
- This seems fairly widespread in language
- We call these *atomic feature-value combinations*
- Other examples:

1. In English:

person feature (1<sup>st</sup>, 2<sup>nd</sup>, 3<sup>rd</sup>);

Case feature (degenerate in English: nominative, object/accusative, possessive/genitive): I know *her* vs. I know *she*;

Number feature: plural/sing; definite/indefinite

Degree: comparative/superlative



# Other languages; formalizing features

- Two kinds:
  1. Syntactic features, purely grammatical function  
Example: Case in German (NOMinative, ACCusative, DATive case) – relative pronoun must agree w/ Case of verb with which it is construed

*Wer nicht stark is, muss klug sein*

*Who not strong is, must clever be*

NOM

NOM

Who isn't strong must be clever

# Continuing this example

*Ich nehme, wen du mir empfiehlst*

*I take whomever you me recommend*

ACC ACC ACC

*I take whomever you recommend to me*

*\*Ich nehme, wen du vertraust*

*I take whomever you trust*

ACC ACC DAT

# Other class of features

2. Syntactic features w/ meaning – example,  
number, def/indef., adjective degree

Hungarian

*Akart egy könyvet*

*He-wanted a book*

*-DEF*

*-DEF*

*egy könyv amit akart*

*A book which he-wanted*

*-DEF*

*-DEF*

# Feature Structures

- Sets of **feature-value pairs** where:
  - Features are atomic symbols
  - Values are atomic symbols or feature structures
  - Illustrated by **attribute-value matrix**

$$\begin{bmatrix} Feature_1 & Value_1 \\ Feature_2 & Value_2 \\ \dots & \dots \\ Feature_n & Value_n \end{bmatrix}$$

# How to formalize?

- Let  $F$  be a finite set of feature names, let  $A$  be a set of feature values
- Let  $p$  be a function from feature names to permissible feature values, that is,  
 $p: F \rightarrow 2^A$
- Now we can define a *word category* as a triple  $\langle F, A, p \rangle$
- This is a partial function from feature names to feature values

# Example

- $F = \{CAT, PLU, PER\}$

- $p$ :

$p(CAT) = \{V, N, ADJ\}$

$p(PER) = \{1, 2, 3\}$

$p(PLU) = \{+, -\}$

$sleep = \{[CAT V], [PLU -], [PER 1]\}$

$sleep = \{[CAT V], [PLU +], [PER 1]\}$

$sleeps = \{[CAT V], [PLU -], [PER 3]\}$

Checking whether features are compatible is relatively simple here

- Feature values can be feature structures themselves – should they be?
  - Useful when certain features commonly co-occur, e.g. number and person

$$\left[ \begin{array}{l} \text{Cat} \quad \text{NP} \\ \text{Agr} \quad \left[ \begin{array}{l} \text{Num} \quad \text{SG} \\ \text{Pers} \quad 3 \end{array} \right] \end{array} \right]$$

- Feature path: path through structures to value (e.g.

*Agr* → *Num* → *SG*

# Important question



- Do features have to be more complicated than this?
- More: hierarchically structured (feature structures) (directed acyclic graphs, DAGs, or even beyond)
- Then *checking* for feature compatibility amounts to *unification*
- Example



# Reentrant Structures

- Feature structures may also contain features that share some feature structure as a value

$$\left[ \begin{array}{l} \textit{Cat} \ S \\ \\ \textit{Head} \left[ \begin{array}{l} \textit{Agr} \ 1 \ \left[ \begin{array}{l} \textit{Num} \ SG \\ \textit{Pers} \ 3 \end{array} \right] \\ \\ \textit{Subj} \ \left[ \begin{array}{l} \textit{Agr} \ 1 \end{array} \right] \end{array} \right. \end{array} \right]$$

- Numerical indices indicate the shared values
- Big Question: do we need nested structures??

- Number feature

$[Num \quad SG]$

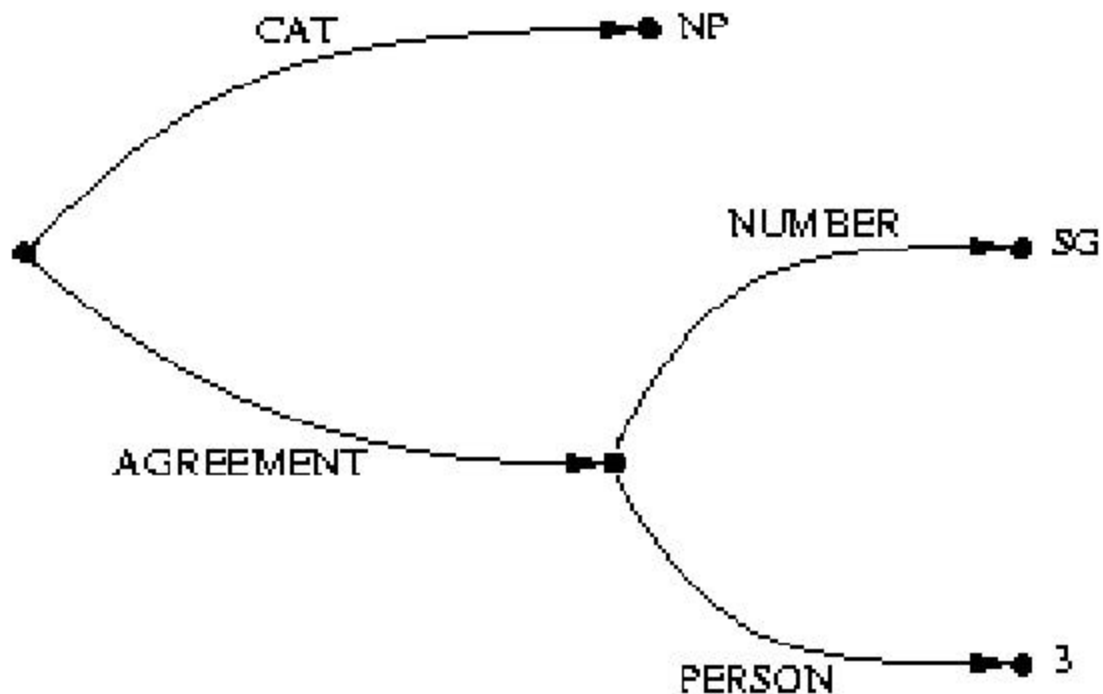
- Number-person features

$\begin{bmatrix} Num & SG \\ Pers & 3 \end{bmatrix}$

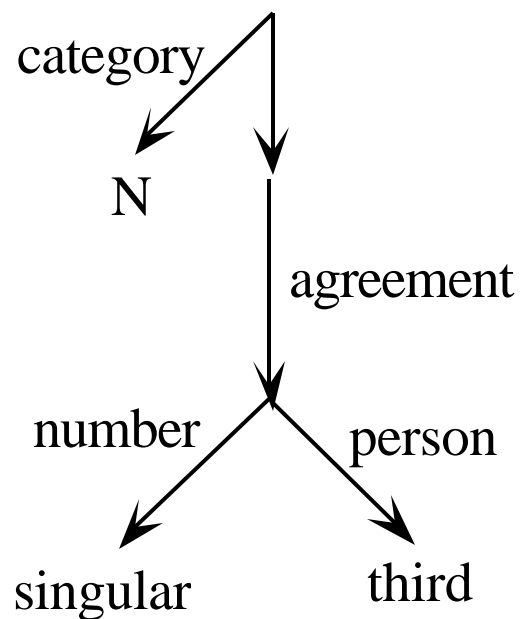
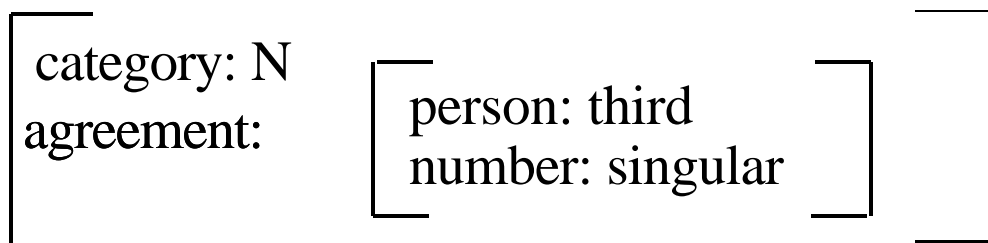
- Number-person-category features

(3sgNP)  $\begin{bmatrix} Cat & NP \\ Num & SG \\ Pers & 3 \end{bmatrix}$

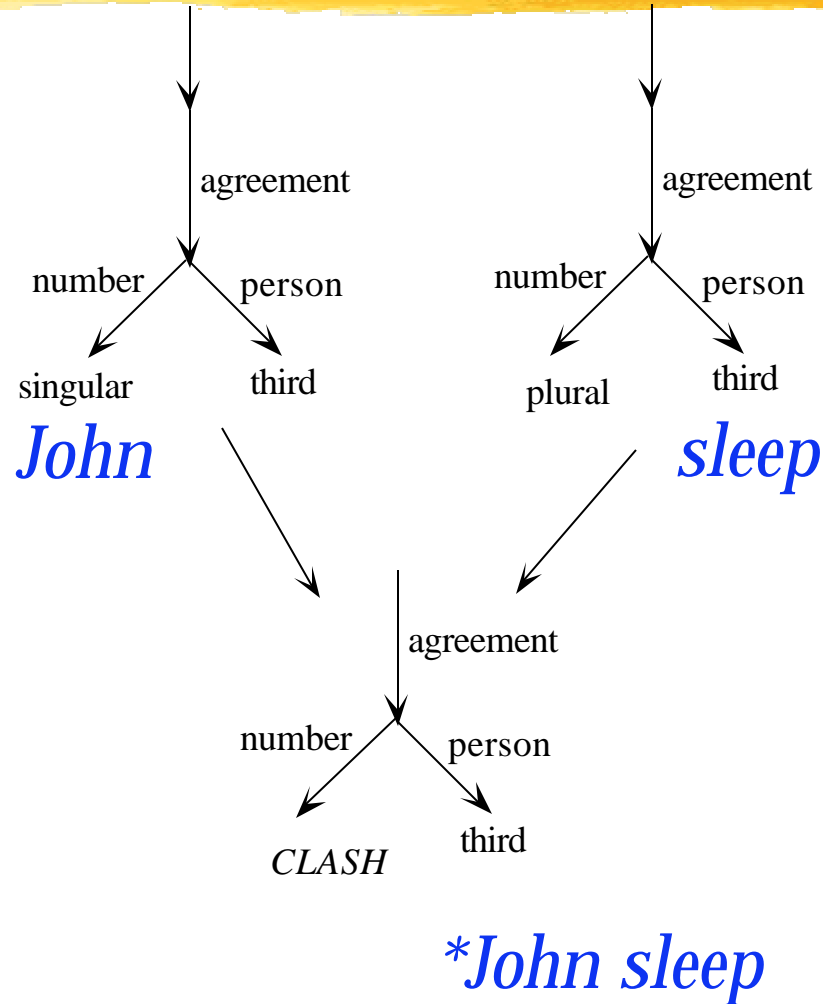
# Graphical Notation for Feature Structures



# Features and grammars



# Feature checking by unification



# Operations on Feature Structures

- What will we need to do to these structures?
  - Check the **compatibility** of two structures
  - **Merge** the information in two structures
- We can do both using **unification**
- We say that two feature structures **can be unified** if the component features that make them up are **compatible**
  - $[\text{Num SG}] \cup [\text{Num SG}] = [\text{Num SG}]$
  - $[\text{Num SG}] \cup [\text{Num PL}]$  fails!
  - $[\text{Num SG}] \cup [\text{Num } []] = [\text{Num SG}]$

- $[\text{Num SG}] \cup [\text{Pers 3}] = \begin{bmatrix} \text{Num SG} \\ \text{Pers 3} \end{bmatrix}$

- Structures are compatible if they contain no features that are incompatible

- Unification of two feature structures:

- Are the structures compatible?

- If so, return the union of all feature/value pairs

- A failed unification attempt

$$\begin{bmatrix} \text{Agr} & 1 & \begin{bmatrix} \text{Num} & \text{SG} \\ \text{Pers} & 3 \end{bmatrix} \\ \text{Subj} & \begin{bmatrix} \text{Agr} & 1 \end{bmatrix} \end{bmatrix} \cup \begin{bmatrix} \text{Agr} & \begin{bmatrix} \text{Num} & \text{Pl} \\ \text{Pers} & 3 \end{bmatrix} \\ \text{Subj} & \begin{bmatrix} \text{Agr} & \begin{bmatrix} \text{Num} & \text{PL} \\ \text{Pers} & 3 \end{bmatrix} \end{bmatrix} \end{bmatrix}$$

# Features, Unification and Grammars

- How do we incorporate feature structures into our grammars?
  - Assume that constituents are objects which have feature-structures associated with them
  - Associate sets of unification constraints with grammar rules
  - Constraints must be satisfied for rule to be satisfied
- For a grammar rule  $\beta_0 \rightarrow \beta_1 \dots \beta_n$ 
  - $\langle \beta_i \text{ feature path} \rangle = \text{Atomic value}$
  - $\langle \beta_i \text{ feature path} \rangle = \langle \beta_j \text{ feature path} \rangle$
- NB: if simple feat-val pairs, no nesting, then no need for paths



# Feature unification examples

(1) [ agreement: [ number: singular  
                  person:                  first ] ]

(2) [ agreement: [ number: singular]  
      case:          nominative ]

- (1) and (2) can unify, producing (3):

(3) [ agreement: [ number: singular  
                  person:                  first ]  
      case:          nominative ]

(try overlapping the graph structures  
corresponding to these two)

# Feature unification examples



(2) [ agreement: [ number: singular ]  
case: nominative ]

(4) [ agreement: [ number: singular  
person: third ] ]

- (2) & (4) can unify, yielding (5):

(5) [ agreement: [ number: singular  
person: third ]  
case: nominative ]

- BUT (1) and (4) cannot unify because their values conflict on <agreement person>

- To enforce subject/verb number agreement

$S \rightarrow NP VP$

$\langle NP \text{ NUM} \rangle = \langle VP \text{ NUM} \rangle$

# Head Features

- Features of most grammatical categories are copied from head child to parent (e.g. from V to VP, Nom to NP, N to Nom, ...)
- These normally written as 'head' features, e.g.

VP → V NP

<VP HEAD> = <V HEAD>

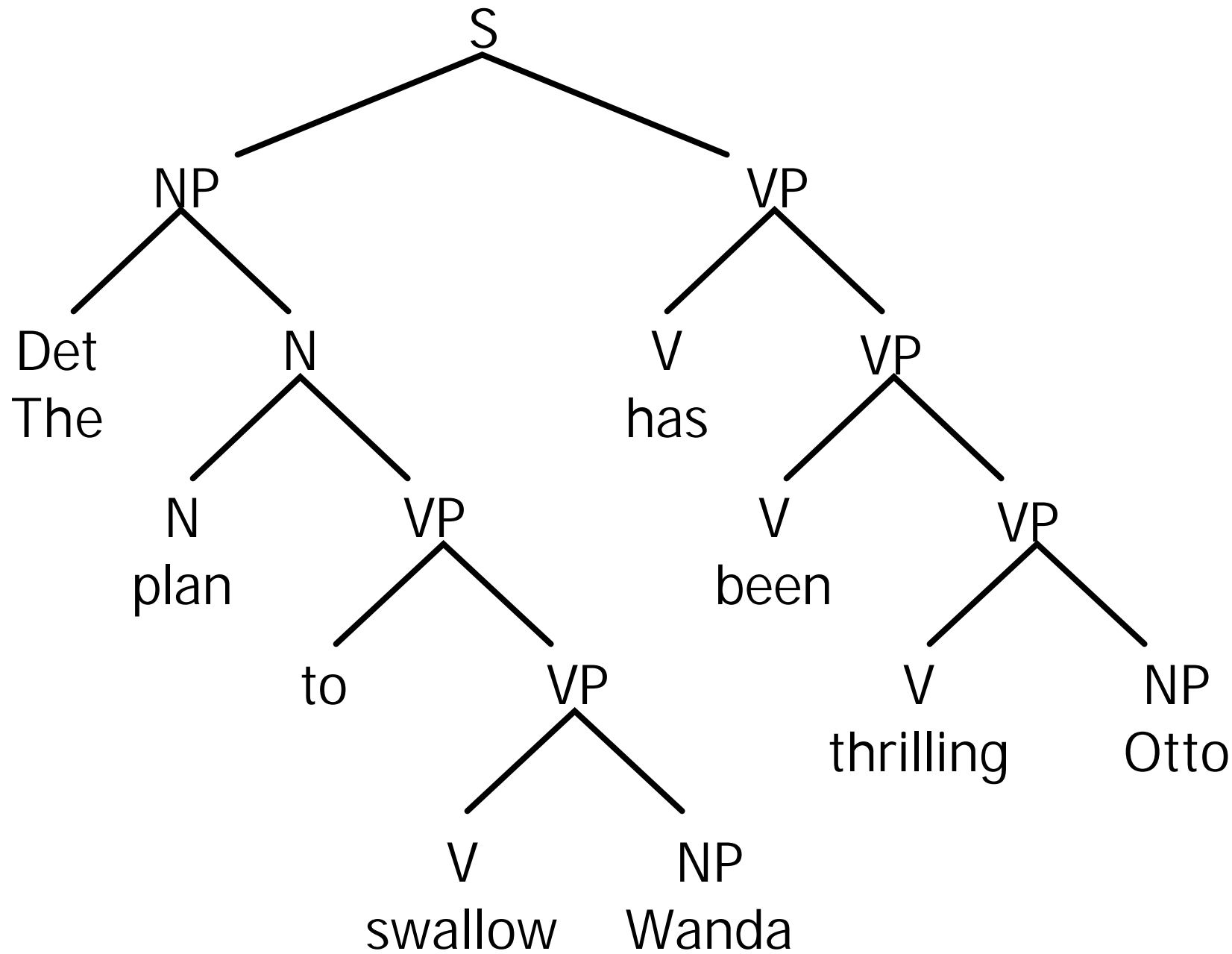
NP → Det Nom

<NP→ HEAD> = <Nom HEAD>

<Det HEAD AGR> = <Nom HEAD AGR>

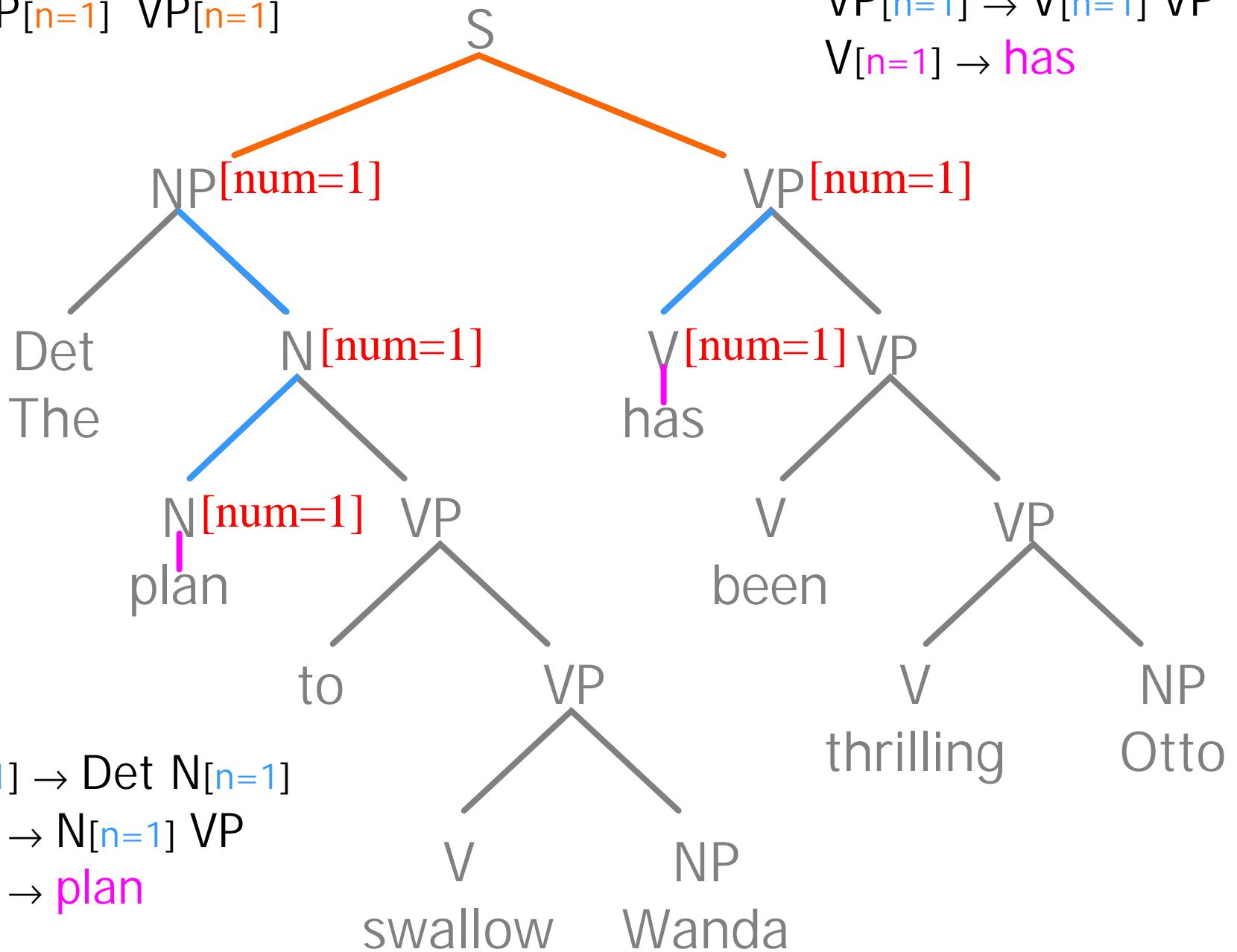
Nom → N

<Nom HEAD> = <N HEAD>



$S \rightarrow NP_{[n=1]} VP_{[n=1]}$

$VP_{[n=1]} \rightarrow V_{[n=1]} VP$   
 $V_{[n=1]} \rightarrow \text{has}$



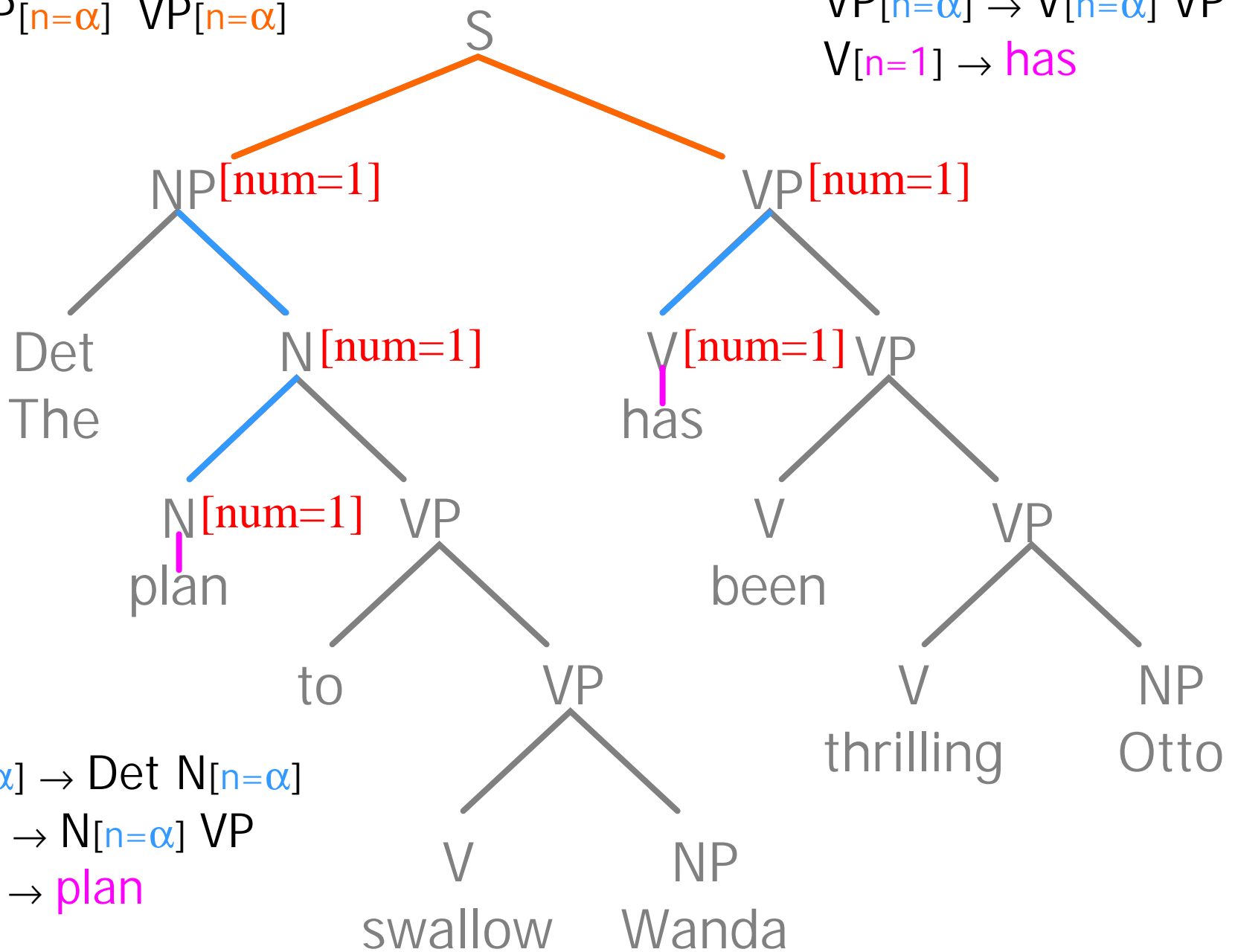
$NP_{[n=1]} \rightarrow \text{Det } N_{[n=1]}$

$N_{[n=1]} \rightarrow N_{[n=1]} VP$

$N_{[n=1]} \rightarrow \text{plan}$

$S \rightarrow NP_{[n=\alpha]} VP_{[n=\alpha]}$

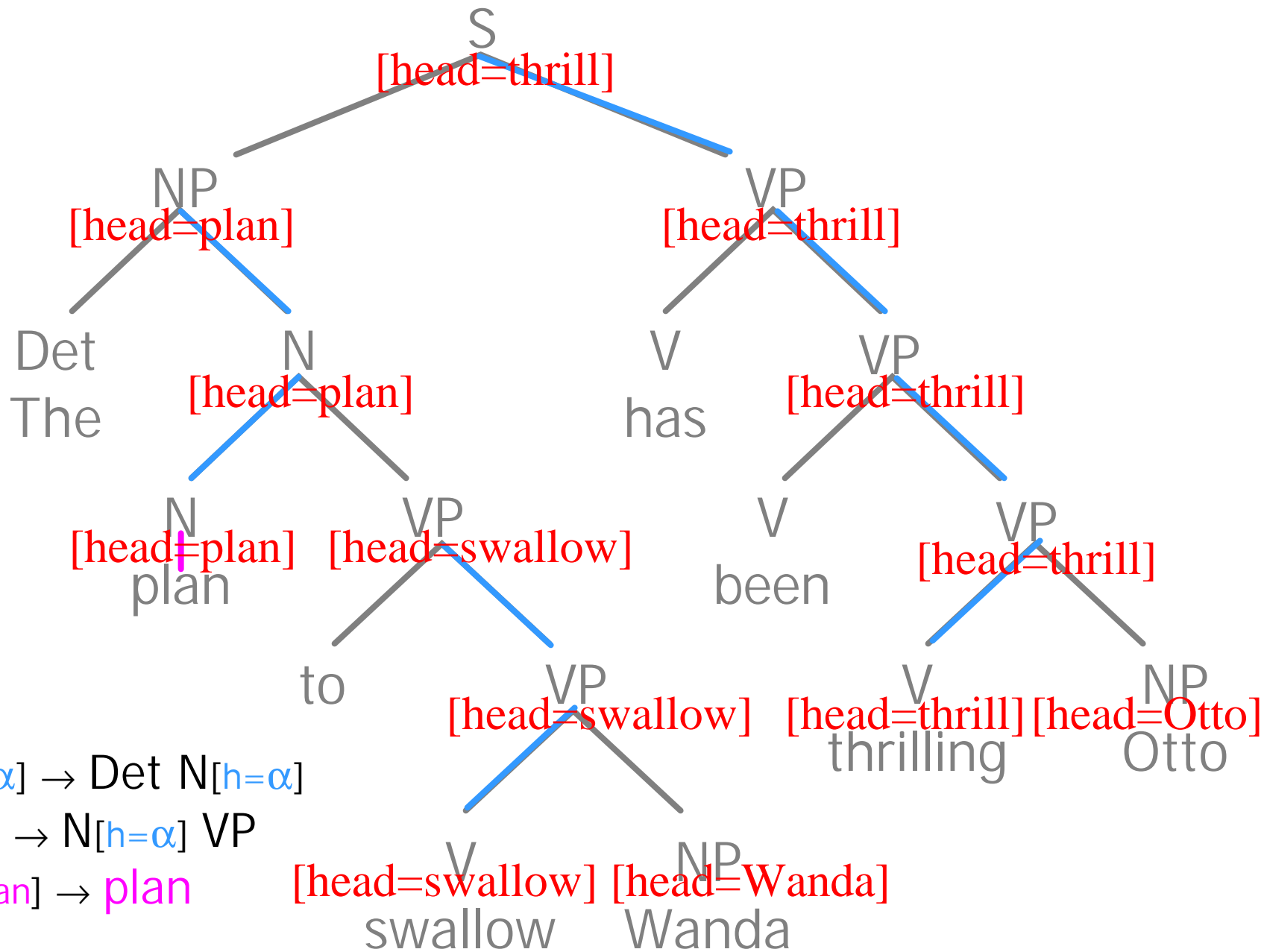
$VP_{[n=\alpha]} \rightarrow V_{[n=\alpha]} VP$   
 $V_{[n=1]} \rightarrow \text{has}$



$NP_{[n=\alpha]} \rightarrow \text{Det } N_{[n=\alpha]}$

$N_{[n=\alpha]} \rightarrow N_{[n=\alpha]} VP$

$N_{[n=1]} \rightarrow \text{plan}$



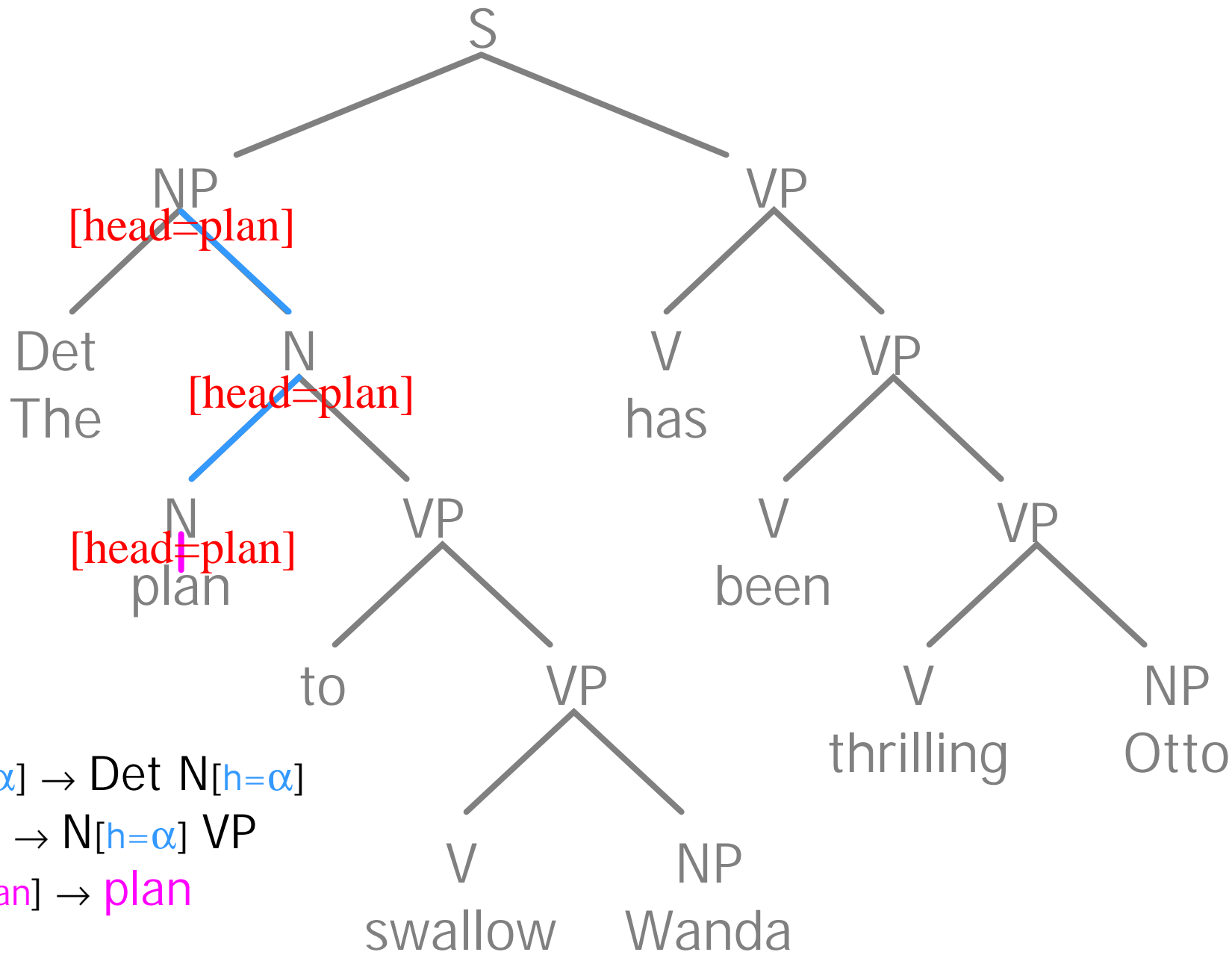
NP[h=α] → Det N[h=α]

N[h=α] → N[h=α] VP

N[h=plan] → plan

V [head=swallow] NP [head=Wanda]  
 swallow Wanda



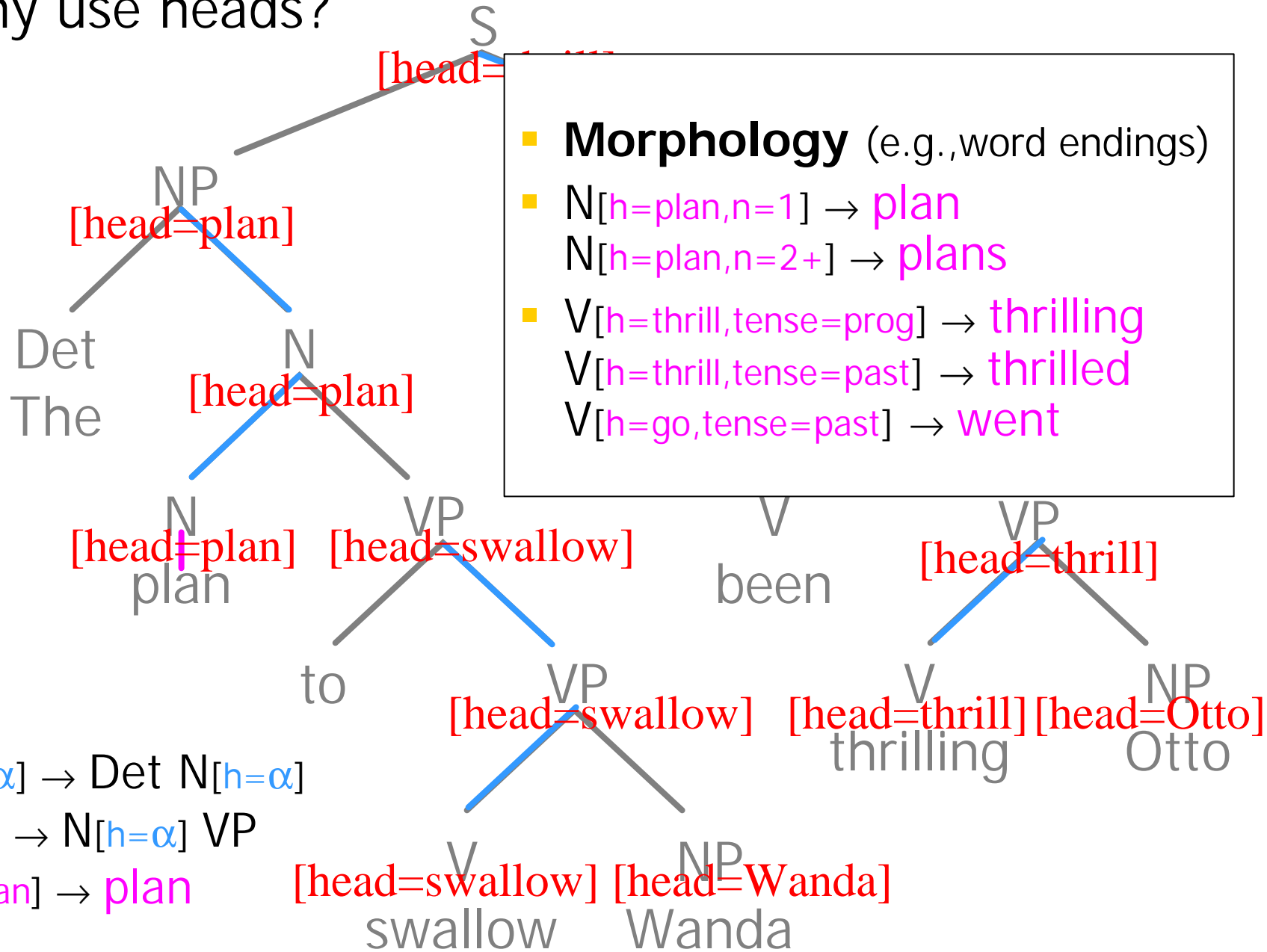


NP[h=α] → Det N[h=α]

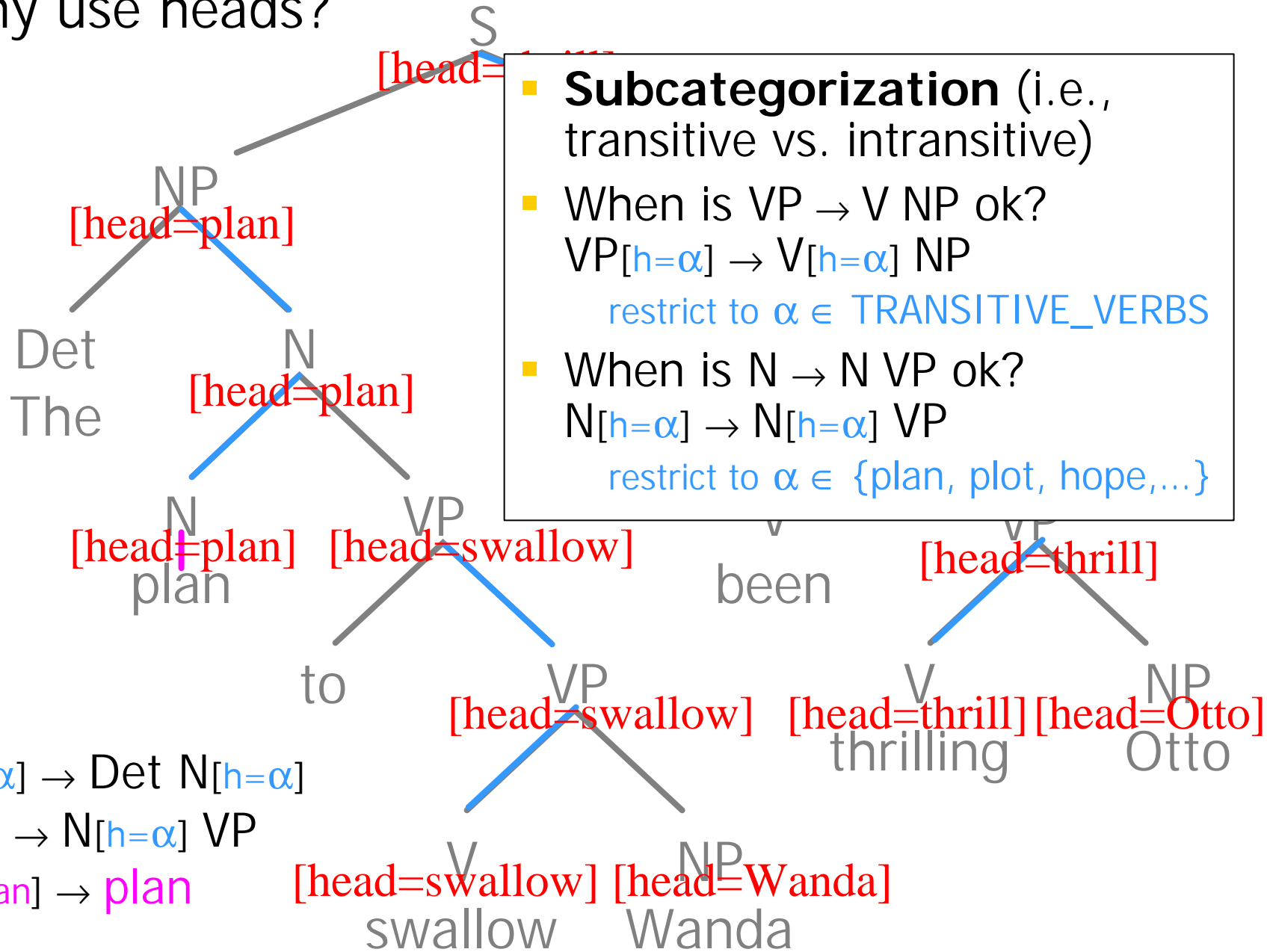
N[h=α] → N[h=α] VP

N[h=plan] → plan

■ Why use heads?

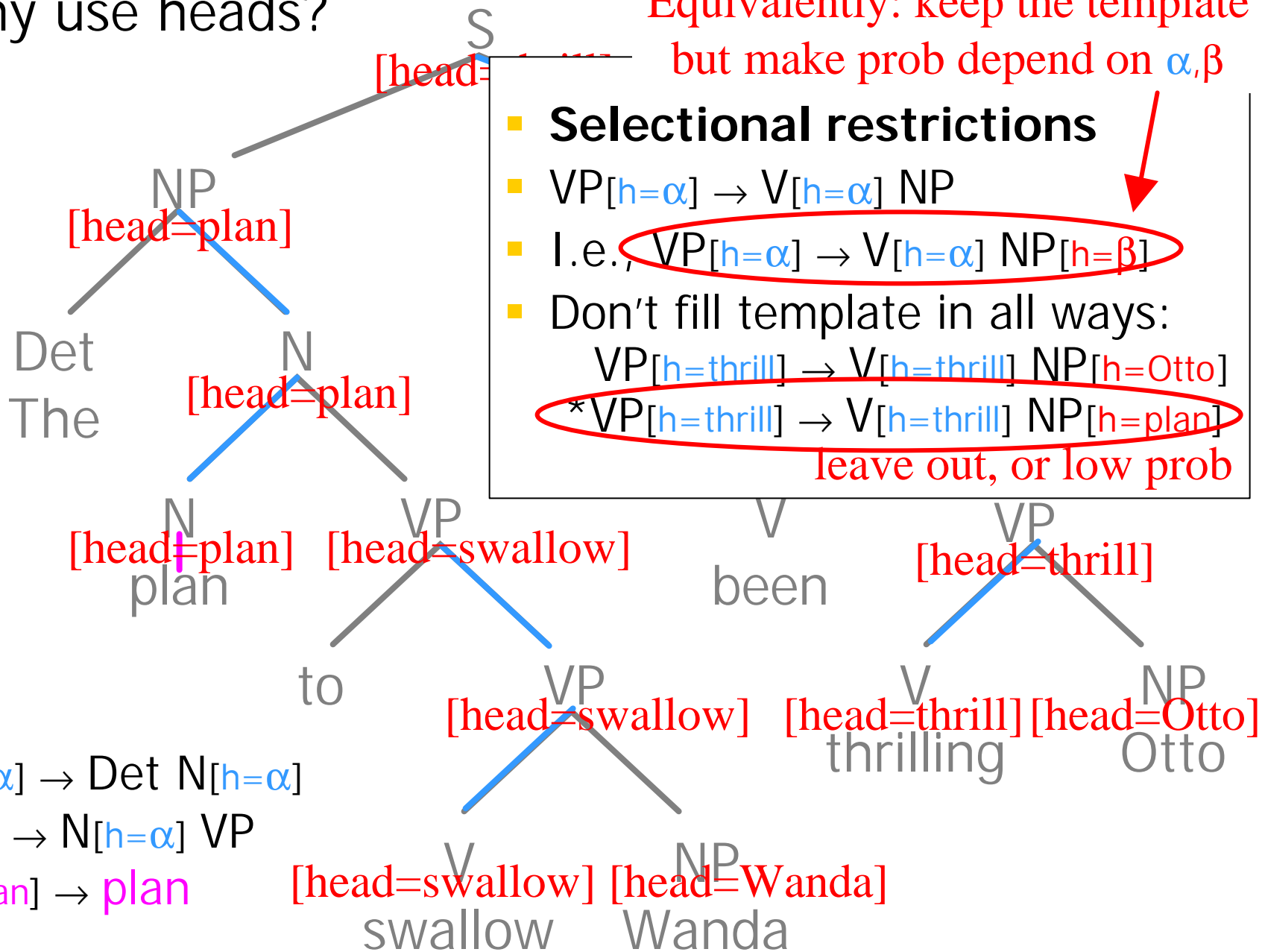


- Why use heads?



■ Why use heads?

Equivalently: keep the template but make prob depend on  $\alpha, \beta$



- **Selectional restrictions**
- $VP[h=\alpha] \rightarrow V[h=\alpha] NP$
- I.e.,  $VP[h=\alpha] \rightarrow V[h=\alpha] NP[h=\beta]$
- Don't fill template in all ways:  
 $VP[h=thrill] \rightarrow V[h=thrill] NP[h=Otto]$   
 $*VP[h=thrill] \rightarrow V[h=thrill] NP[h=plan]$   
 leave out, or low prob

NP[h= $\alpha$ ] → Det N[h= $\alpha$ ]

N[h= $\alpha$ ] → N[h= $\alpha$ ] VP

N[h=plan] → plan

V [head=swallow] [head=Wanda]  
swallow Wanda

# How can we parse with feature structures?

- Unification operator: takes 2 feature structures and returns *either* a merged feature structure or *fail*
- Input structures represented as DAGs
  - Features are labels on edges
  - Values are atomic symbols or DAGs
- Unification algorithm goes through features in one input DAG<sub>1</sub> trying to find corresponding features in DAG<sub>2</sub> – if all match, success, else fail

# Unification and Earley Parsing

- Goal:
  - Use feature structures to provide richer representation
  - Block entry into chart of ill-formed constituents
- Changes needed to Earley
  - Add feature structures to grammar rules, e.g.  
 $S \rightarrow NP VP$   
 $\langle NP \text{ HEAD AGR} \rangle = \langle VP \text{ HEAD AGR} \rangle$   
 $\langle S \text{ HEAD} \rangle = \langle VP \text{ HEAD} \rangle$
  - Add field to states containing DAG representing feature structure corresponding to state of parse, e.g.  
 $S \rightarrow \bullet NP VP, [0,0], [], DAG$

- Add new test to Completer operation
  - Recall: Completer adds new states to chart by finding states whose • can be advanced (i.e., category of next constituent matches that of completed constituent)
  - Now: Completer will only advance those states if their feature structures unify
- New test for whether to enter a state in the chart
  - Now DAGs may differ, so check must be more complex
  - Don't add states that have DAGs that are more **specific** than states in chart: is new state **subsumed** by existing states?

# General feature grammars –violate the properties of natural languages?

- Take example from so-called “lexical-functional grammar” but this applies as well to any general unification grammar
- Lexical functional grammar (LFG): add checking rules to CF rules (also variant HPSG)



# Example Lexical functional grammar

- Basic CF rule:

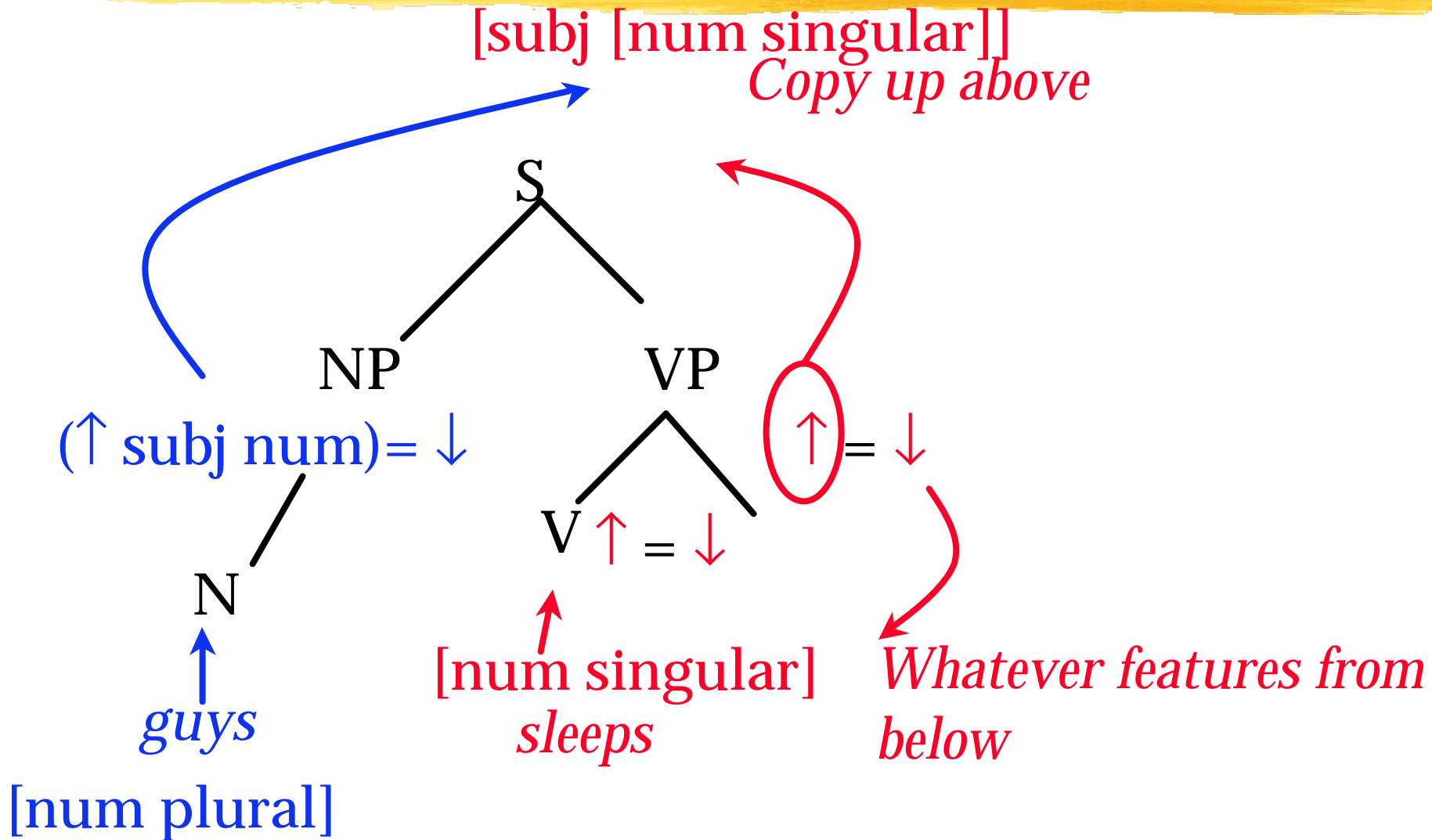
$S \rightarrow NP VP$

- Add corresponding 'feature checking'

$S \rightarrow \quad NP \quad \quad \quad VP$   
 $\quad \quad (\uparrow \text{ subj num}) = \downarrow \quad \quad \uparrow = \downarrow$

- What is the interpretation of this?

# Applying feature checking in LFG



# Evidence that you don't need this much power - hierarchy

- Linguistic evidence: looks like you just check whether features are *nondistinct*, rather than equal or not – variable *matching*, not variable substitution

- Full unification lets you generate unnatural languages:

$a^i$ , s.t.  $i$  a power of 2 – e.g.,  $a$ ,  $aa$ ,  $aaaa$ ,  $aaaaaaaa$ , ...

why is this 'unnatural' – another (seeming) property of natural languages:

Natural languages seem to obey a *constant growth* property

# Constant growth property



- Take a language & order its sentences in terms of increasing length in terms of # of words (what's shortest sentence in English?)
- Claim:  $\exists$  Bound on the 'distance gap' between any two consecutive sentences in this list, which can be specified in advance (fixed)
- 'Intervals' between valid sentences cannot get too big – cannot grow w/o bounds
- We can do this a bit more formally

# Constant growth

- Dfn. A language  $L$  is semilinear if the number of occurrences of each symbol in any string of  $L$  is a linear combination of the occurrences of these symbols in some fixed, finite set of strings of  $L$ .
- Dfn. A language  $L$  is constant growth if there is a constant  $c_0$  and a finite set of constants  $C$  s.t. for all  $w \in L$ , where  $|w| > c_0 \exists w' \in L$  s.t.  $|w| = |w'| + c$ , some  $c \in C$
- Fact. (Parikh, 1971). Context-free languages are semilinear, and constant-growth
- Fact. (Berwick, 1983). The power of 2 language is non constant-growth

# Alas, this allows non-constant growth, unnatural languages

- Can use LFG to generate power of 2 language
- Very simple to do
- $A \rightarrow A \quad A$

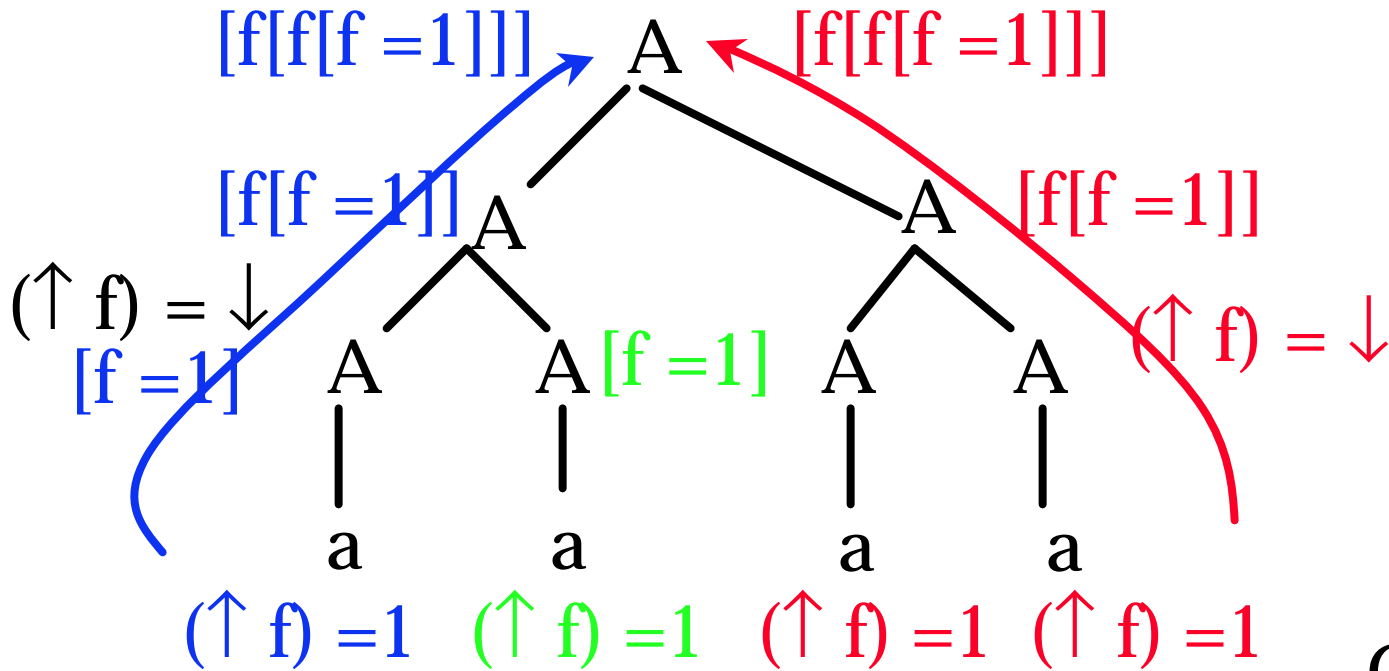
$$(\uparrow f) = \downarrow \quad (\uparrow f) = \downarrow$$

$$A \rightarrow a$$

$$(\uparrow f) = 1$$

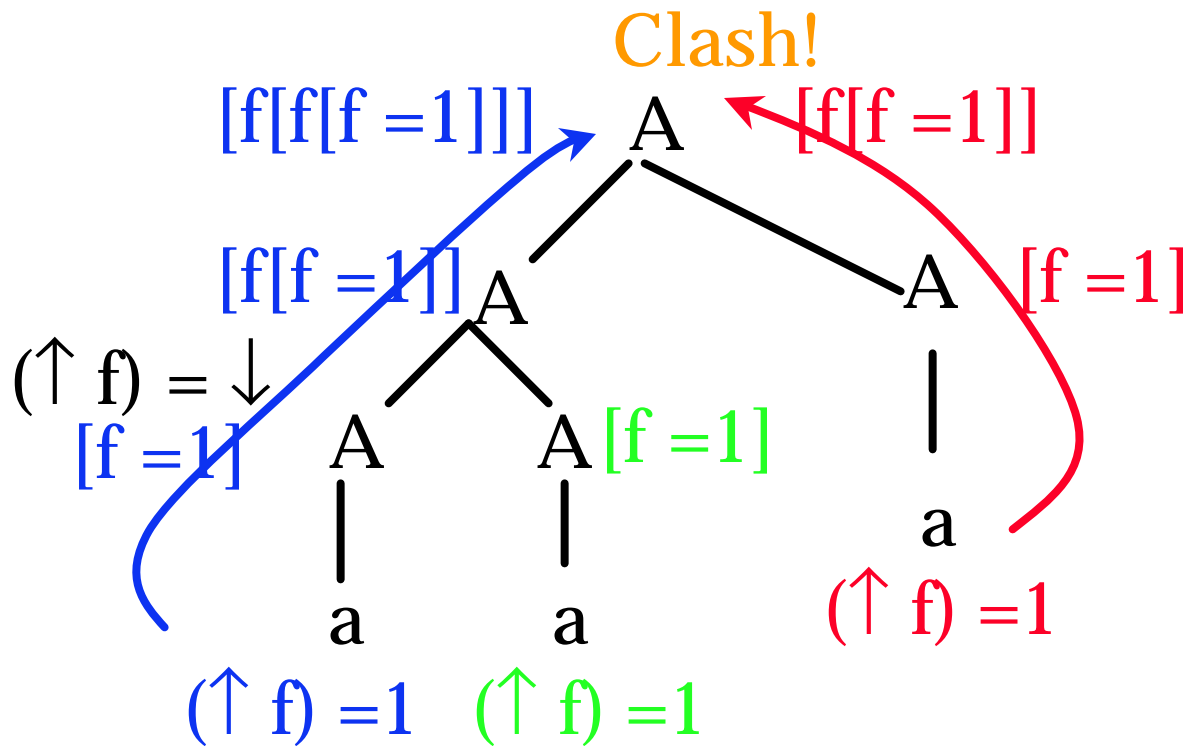
Lets us `count' the number of embeddings on the right & the left – make sure a power of 2

# Example



Checks ok

# If mismatch anywhere, get a feature clash...



**Fails!**



# Conclusion then



- If we use too powerful a formalism, it lets us write 'unnatural' grammars
- This puts burden on the person writing the grammar – which may be ok.
- However, child doesn't presumably do this (they don't get 'late days')
- We want to strive for automatic programming – ambitious goal

# Summing Up



- Feature structures encoded rich information about components of grammar rules
- Unification provides a mechanism for merging structures and for comparing them
- Feature structures can be quite complex:
  - Subcategorization constraints
  - Long-distance dependencies
- Unification parsing:
  - Merge or fail
  - Modifying Earley to do unification parsing

# From syntax to meaning



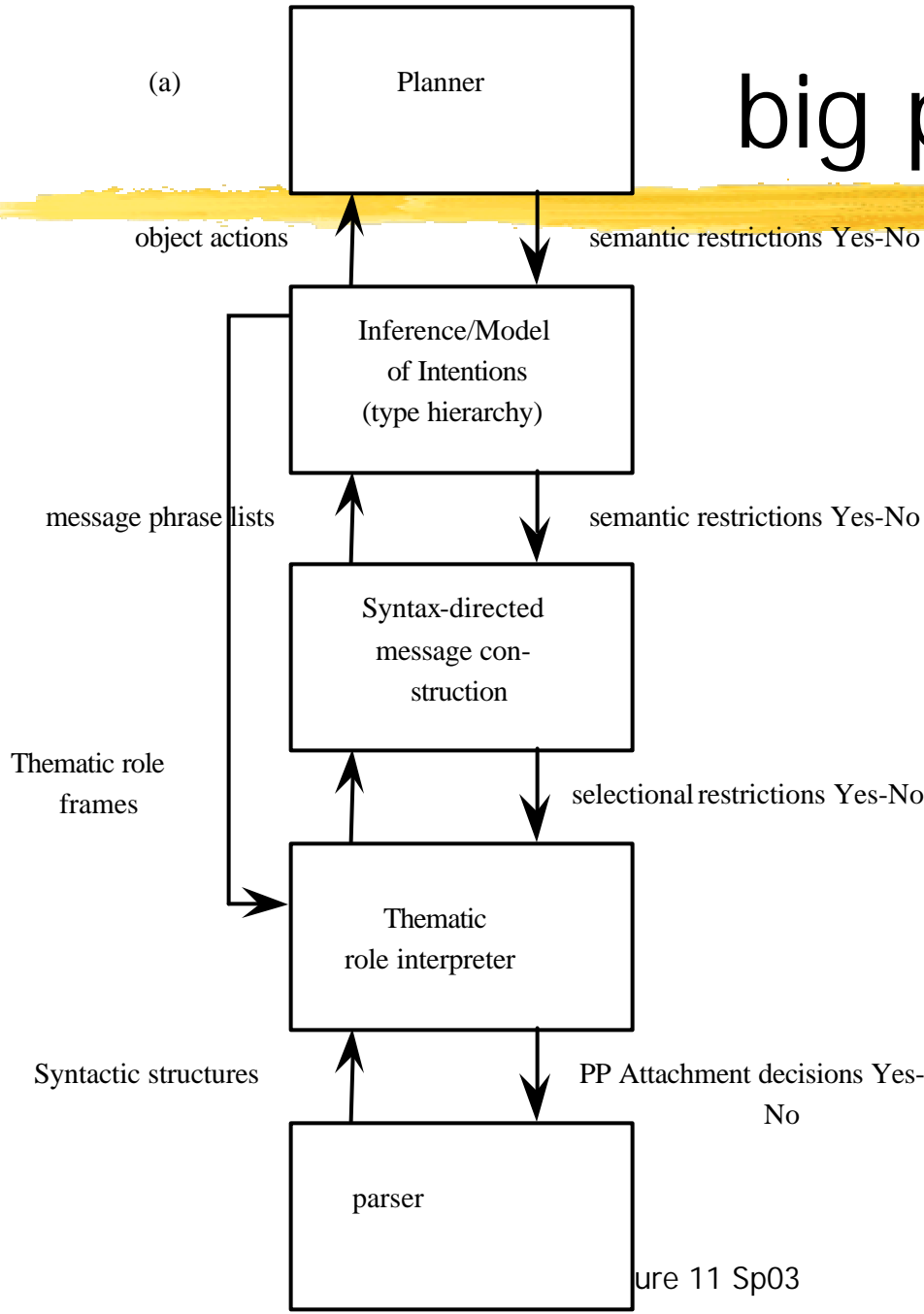
- What does 'understanding' mean
- How can we compute it if we can't represent it
- The 'classical' approach: compositional semantics
- Implementation like a programming language

# Initial Simplifying Assumptions

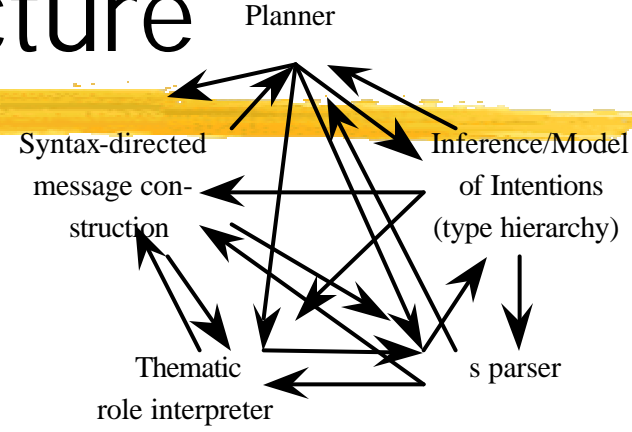


- Focus on literal meaning
  - Conventional meanings of words
  - Ignore context

(a)



# big picture



# Example of what we might do

athena>(top-level)

Shall I clear the database? (y or n) y

sem-interpret>John saw Mary in the park

OK.

sem-interpret>Where did John see Mary

IN THE PARK.

sem-interpret>John gave Fido to Mary

OK.

sem-interpret>Who gave John Fido

I DON'T KNOW

sem-interpret>Who gave Mary Fido

JOHN

sem-interpret >John saw Fido

OK.

sem-interpret>Who did John see

FIDO AND MARY

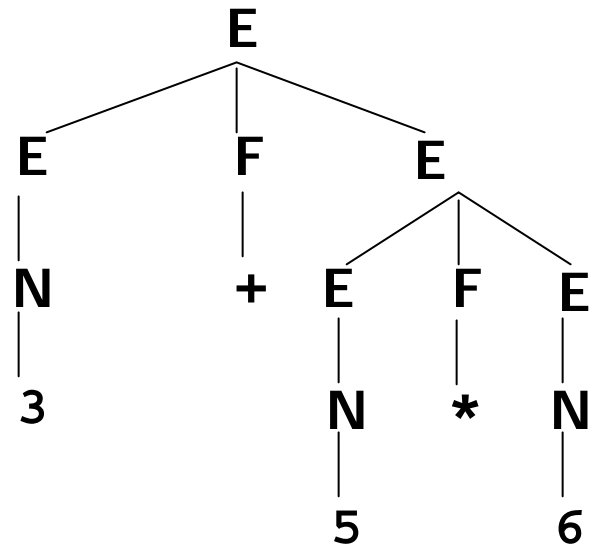
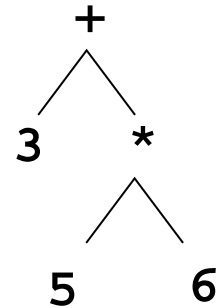
# what



- The nature (representation) of meaning representations *vs/ how* these are assembled

# Analogy w/ prog. language

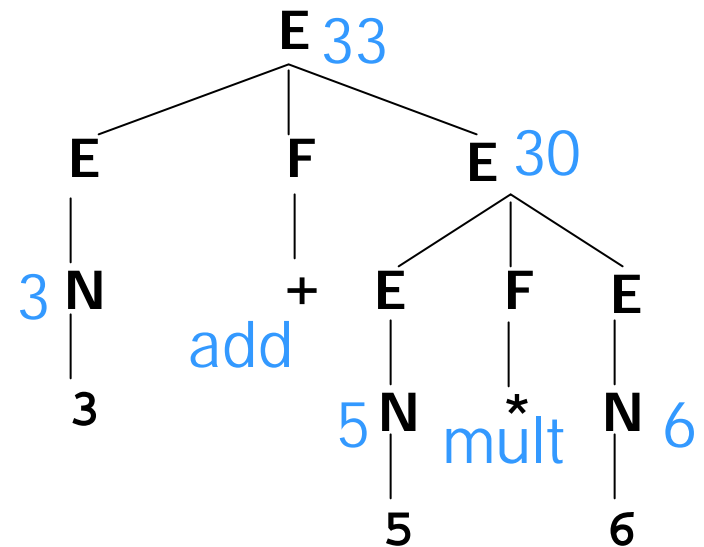
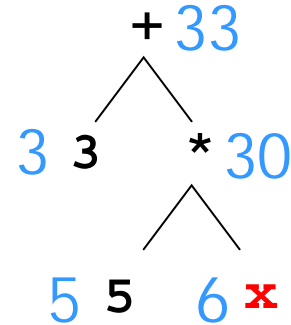
- What is meaning of  $3+5*6$ ?
- First parse it into  $3+(5*6)$





# Interpreting in an Environment

- How about  $3+5*x$ ?
- Same thing: the meaning of  $x$  is found from the environment (it's 6)
- Analogies in language?



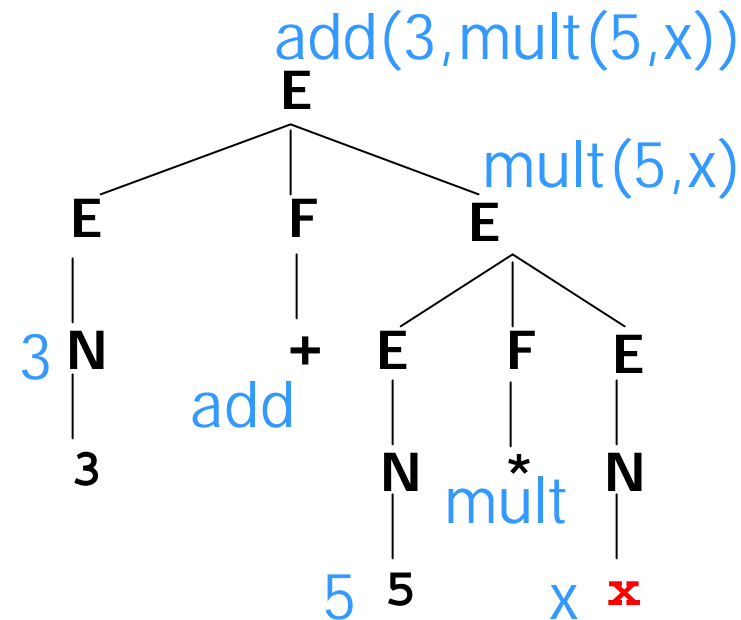
# Compiling

- How about  $3+5*x$ ?
- Don't know  $x$  at compile time
- "Meaning" at a node is a piece of code, not a number

$5*(x+1)-2$  is a different expression that produces *equivalent* code

(can be converted to the previous code by optimization)

Analogies in language?

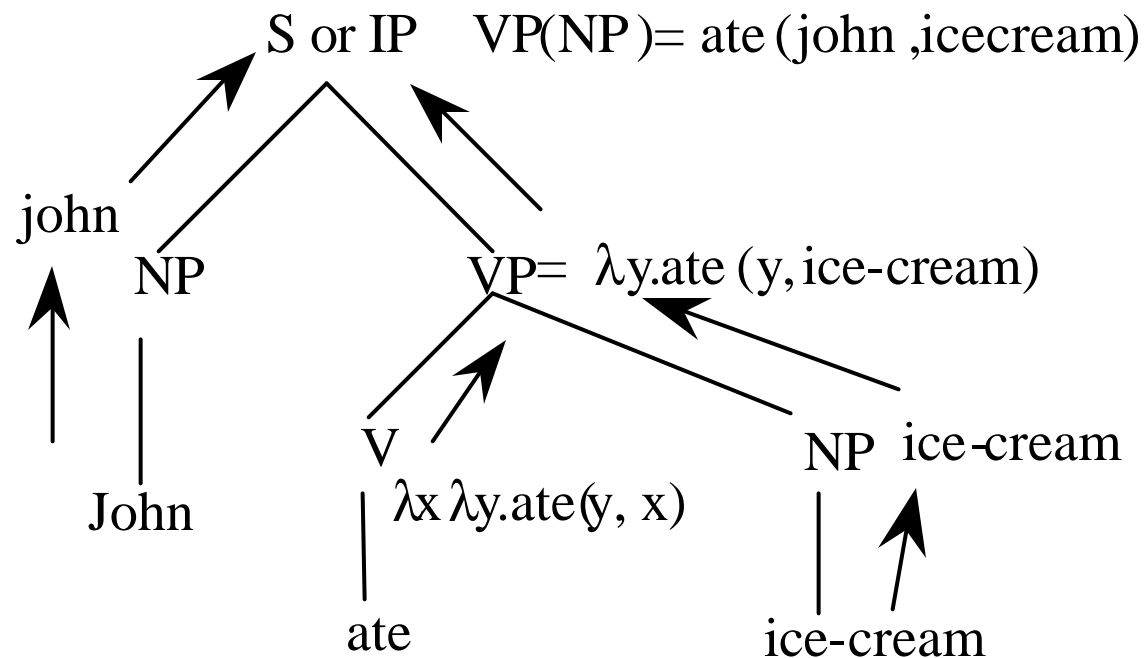


# What



- What representation do we want for something like  
John ate ice-cream →  
ate(John, ice-cream)
- Lambda calculus
- We'll have to posit something that will do the work
- Predicate of 2 arguments:  
 $\lambda x \lambda y \text{ ate}(y, x)$

# How: recover meaning from structure



# What Counts as Understanding?

## some notions

- We understand if we can respond appropriately
  - ok for commands, questions (these demand response)
  - "Computer, warp speed 5"
  - "throw axe at dwarf"
  - "put all of my blocks in the red box"
  - imperative programming languages
  - database queries and other questions
- We understand statement if we can determine its truth
  - ok, but if you knew whether it was true, why did anyone bother telling it to you?
  - comparable notion for understanding NP is to compute what the NP refers to, which might be useful

# What Counts as Understanding?

## some notions

- We understand statement if we know *how* to determine its truth
  - What are exact conditions under which it would be true?
    - necessary + sufficient
  - Equivalently, derive all its consequences
    - what else must be true if we accept the statement?
  - Philosophers tend to use this definition
- We understand statement if we can use it to answer questions [very similar to above – requires reasoning]
  - Easy: John ate pizza. What was eaten by John?
  - Hard: White's first move is P-Q4. Can Black checkmate?
  - Constructing a *procedure* to get the answer is enough

# Representing Meaning



- What requirements do we have for meaning representations?

# What requirements must meaning representations fulfill?

- Verifiability: The system should allow us to compare representations to facts in a Knowledge Base (KB)
  - Cat(Huey)
- Ambiguity: The system should allow us to represent meanings unambiguously
  - German teachers has 2 representations
- Vagueness: The system should allow us to represent vagueness
  - He lives somewhere in the south of France.



# Requirements: Inference



- Draw valid conclusions based on the meaning representation of inputs and its store of background knowledge.

Does Huey eat kibble?

thing(kibble)

Eat(Huey,x)  $\wedge$  thing(x)

# Requirements: Canonical Form



- Inputs that mean the same thing have the same representation.
  - Huey eats kibble.
  - Kibble, Huey will eat.
  - What Huey eats is kibble.
  - It's kibble that Huey eats.
- Alternatives
  - Four different semantic representations
  - Store all possible meaning representations in Knowledge Base

# Requirements: Compositionality



- Can get meaning of “brown cow” from *separate, independent* meanings of “brown” and “cow”
- $\text{Brown}(x) \wedge \text{Cow}(x)$
- I’ve never seen a purple cow, I never hope to see one...

# Barriers to compositionality



- *Ce corps qui s'appelait e qui s'appelle encore le saint empire romain n'était en aucune maniere ni saint, ni romain, ni empire.*
- This body, which called itself and still calls itself the Holy Roman Empire, was neither Holy, nor Roman, nor an Empire - *Voltaire*

# Need some kind of logical calculus

- Not ideal as a meaning representation and doesn't do everything we want - but close
  - Supports the determination of truth
  - Supports compositionality of meaning
  - Supports question-answering (via variables)
  - Supports inference
- What are its elements?
- What else do we need?

- Logical connectives permit compositionality of meaning

kibble(x)  $\rightarrow$  likes(Huey,x)

cat(Vera)  $\wedge$  weird(Vera)

sleeping(Huey)  $\vee$  eating(Huey)

- Expressions can be assigned truth values, T or F, based on whether the propositions they represent are T or F in the world
  - Atomic formulae are T or F based on their presence or absence in a DB (Closed World Assumption?)
  - Composed meanings are inferred from DB and meaning of logical connectives

- cat(Huey)
- sibling(Huey, Vera)
- $\text{sibling}(x, y) \wedge \text{cat}(x) \rightarrow \text{cat}(y)$
- cat(Vera)??
- Limitations:
  - Do 'and' and 'or' in natural language really mean ' $\wedge$ ' and ' $\vee$ '?
    - Mary got married and had a baby.
    - Your money or your life!
    - He was happy but ignorant.
  - Does ' $\rightarrow$ ' mean 'if'?
    - I'll go if you promise to wear a tutu.

- Frame

Having

Haver: S

HadThing: Car

- All represent 'linguistic meaning' of **I have a car**

*and* state of affairs in some world

- All consist of structures, composed of symbols representing objects and relations among them



# What



- What representation do we want for something like  
John ate ice-cream →  
ate(John, ice-cream)
- Lambda calculus
- We'll have to posit something that will do the work
- Predicate of 2 arguments:  
 $\lambda x \lambda y \text{ ate}(y, x)$

# Lambda application works

- Suppose John, ice-cream = constants, i.e.,  $\lambda x.x$ , the identity function
- Then lambda substitution does give the right results:

$\lambda x \lambda y \text{ate}(y, x) (\text{ice-cream})(\text{John}) \rightarrow$

$\lambda y \text{ate}(y, \text{ice-cream})(\text{John}) \rightarrow$

$\text{ate}(\text{John}, \text{ice-cream})$

But... where do we get the  $\lambda$ -forms from?

# Example of what we now can do

athena>(top-level)

Shall I clear the database? (y or n) y

sem-interpret>John saw Mary in the park

OK.

sem-interpret>Where did John see Mary

IN THE PARK.

sem-interpret>John gave Fido to Mary

OK.

sem-interpret>Who gave John Fido

I DON'T KNOW

sem-interpret>Who gave Mary Fido

JOHN

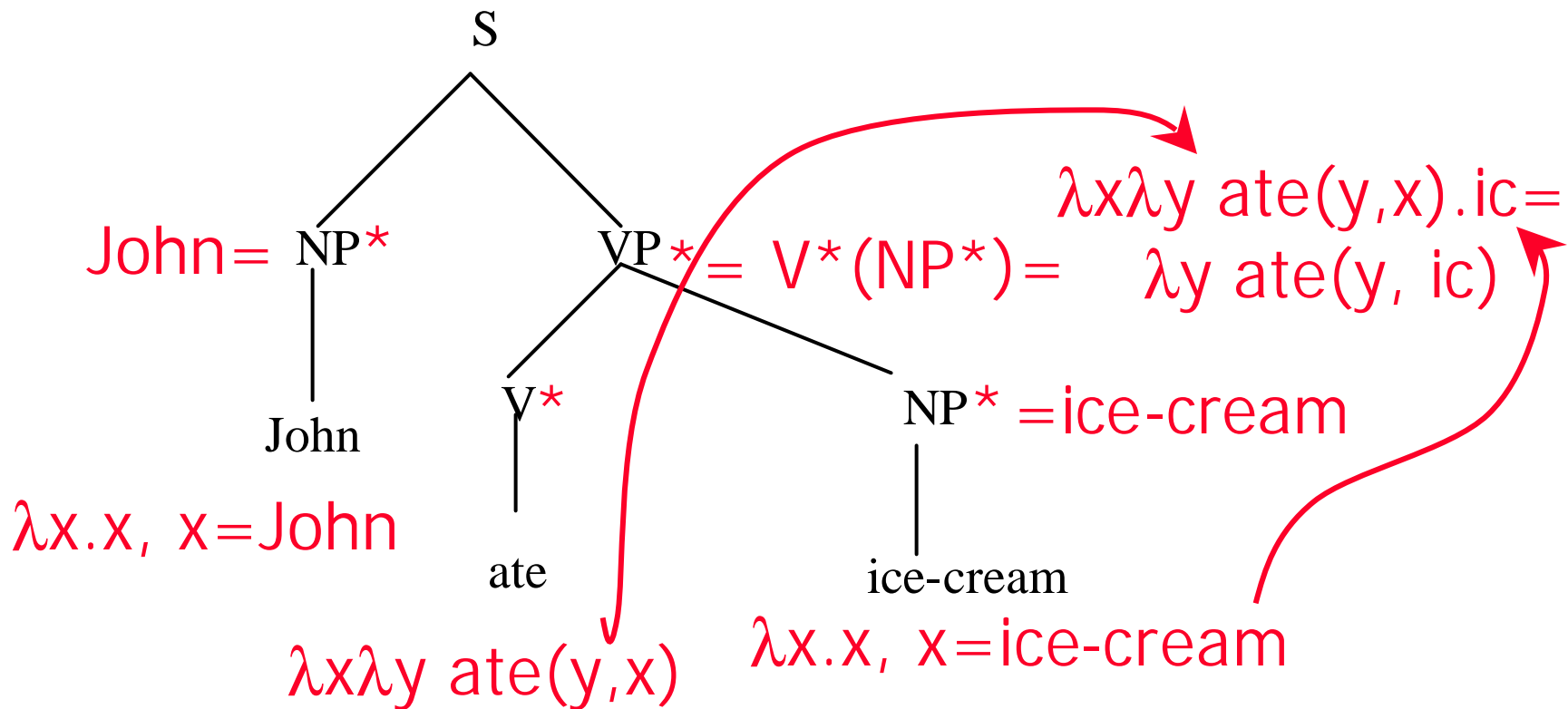
sem-interpret >John saw Fido

OK.

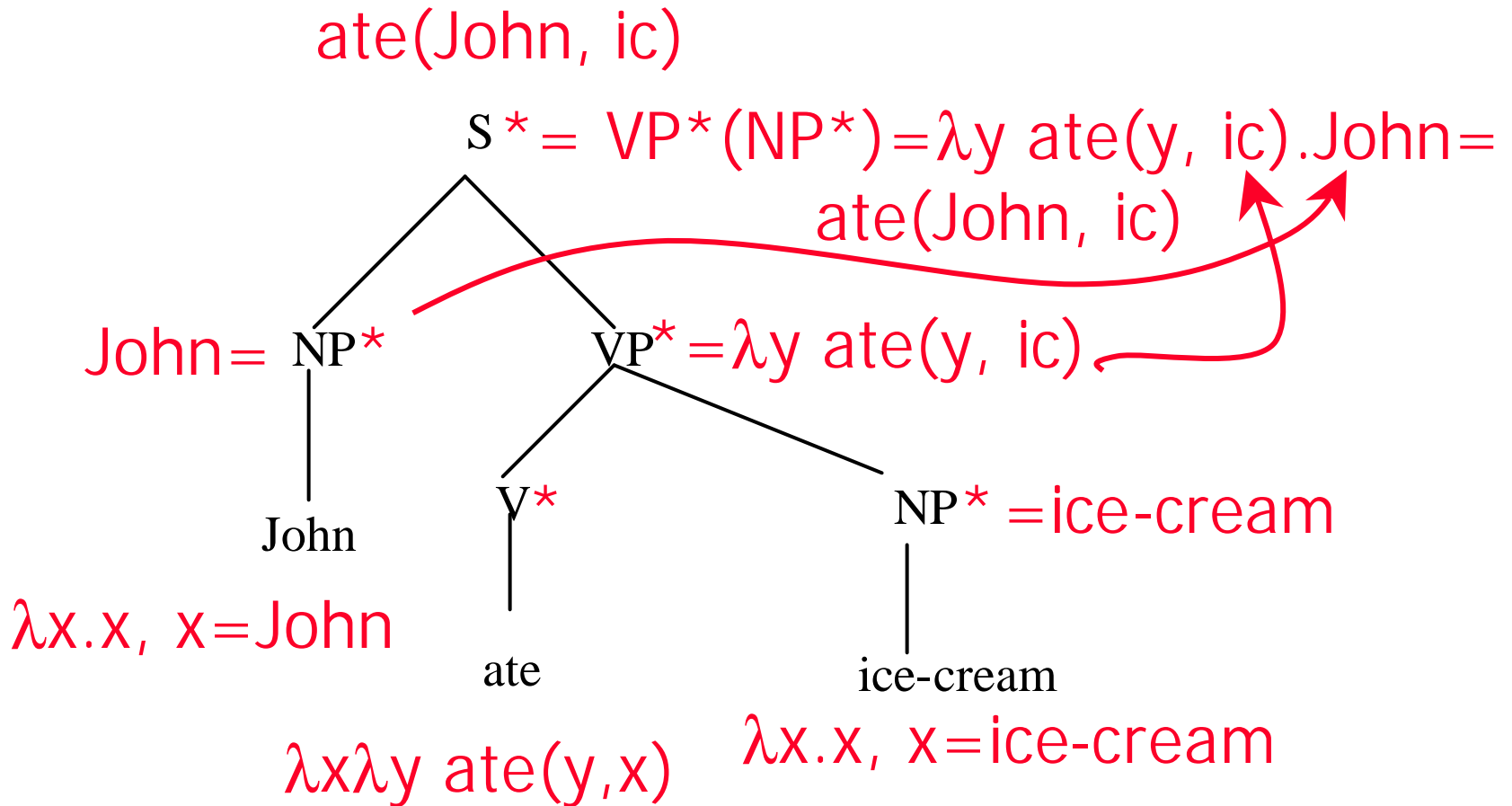
sem-interpret>Who did John see

FIDO AND MARY

# How: to recover meaning from structure



# How



# In this picture



- The meaning of a sentence is the composition of a function  $VP^*$  on an argument  $NP^*$
- The lexical entries are  $\lambda$  forms
  - Simple nouns are just constants
  - Verbs are  $\lambda$  forms indicating their argument structure
- Verb phrases return  $\lambda$  functions as their results (in fact – higher order)

# How



- Application of the lambda form associated with the VP to the lambda form given by the argument NP
- Words just return 'themselves' as values (from lexicon)
- Given parse tree, then by working bottom up as shown next, we get to the logical form *ate(John, ice-cream)*
- This predicate can then be evaluated against a database – this is *model interpretation*- to return a value, or t/f, etc.

# Code – sample rules

## Syntactic rule

(root ==> s)

(s ==> np vp)

(vp ==> v+args)

(v+args ==> v2 np)

(np-pro ==> name) #'identity)

## Semantic rule

(lambda (s) (PROCESS-SENTENCE s))

(lambda (np vp) (funcall vp np))

(lambda (v+args) (lambda (subj)  
 (funcall v+args subj))))

(lambda (v2 np)  
 (lambda (subj)  
 (funcall v2 subj np))))



# On to semantic interpretation

- Four basic principles
- 1. **Rule-to-Rule** semantic interpretation [aka “*syntax-directed translation*”]: pair syntax, semantic rules. (GPSG: pair each cf rule w/ semantic ‘action’; as in compiler theory – due to Knuth, 1968)
- 2. **Compositionality**: Meaning of a phrase is a function of the meaning of its parts *and nothing more* e.g., meaning of  $S \rightarrow NP VP$  is  $f(M(NP) \bullet M(VP))$  (analog of ‘context-freeness’ for semantics – local)
- 3. **Truth conditional meaning**: meaning of S equated with *conditions* that make it true
- 4. **Model theoretic semantics**: correlation betw. Language & world via set theory & mappings

# Syntax & paired semantics



Item or rule

Verb *ate*

propN

V

S (or CP)

NP

VP

Semantic translation

$\lambda x \lambda y. ate(y, x)$

$\lambda x. x$

$V^* = \lambda$  for lex entry

$S^* = VP^*(NP^*)$

$N^*$

$V^*(NP^*)$

# Logic: Lambda Terms

- Lambda terms:
  - A way of writing “anonymous functions”
    - No function header or function name
    - But defines the key thing: **behavior** of the function
    - Just as we can talk about 3 without naming it “x”
  - Let `square =  $\lambda p p^*p$`
  - Equivalent to `int square(p) { return p*p; }`
  - But we can talk about  `$\lambda p p^*p$`  without naming it
  - Format of a lambda term:  `$\lambda$  variable expression`

# Logic: Lambda Terms

- Lambda terms:
  - Let  $\text{square} = \lambda p \ p * p$
  - Then  $\text{square}(3) = (\lambda p \ p * p)(3) = 3 * 3$
  - **Note:  $\text{square}(x)$  isn't a function! It's just the value  $x * x$ .**
  - But  $\lambda x \ \text{square}(x) = \lambda x \ x * x = \lambda p \ p * p = \text{square}$   
(proving that these functions are equal – and indeed they are, as they act the same on all arguments: what is  $(\lambda x \ \text{square}(x))(y)$ ?)

---

- Let  $\text{even} = \lambda p \ (p \bmod 2 == 0)$  a predicate: returns true/false
- $\text{even}(x)$  is true if  $x$  is even
- How about  $\text{even}(\text{square}(x))$ ?
- $\lambda x \ \text{even}(\text{square}(x))$  is true of numbers with even squares
  - Just apply rules to get  $\lambda x \ (\text{even}(x * x)) = \lambda x \ (x * x \bmod 2 == 0)$
  - This happens to denote the same predicate as  $\text{even}$  does

# Logic: Multiple Arguments

- All lambda terms have one argument
- But we can fake multiple arguments ...
- Suppose we want to write  $\text{times}(5,6)$
- Remember:  $\text{square}$  can be written as  $\lambda x \text{ square}(x)$
- Similarly,  $\text{times}$  is equivalent to  $\lambda x \lambda y \text{ times}(x,y)$
- Claim that  $\text{times}(5)(6)$  means same as  $\text{times}(5,6)$ 
  - $\text{times}(5) = (\lambda x \lambda y \text{ times}(x,y)) (5) = \lambda y \text{ times}(5,y)$ 
    - If this function weren't anonymous, what would we call it?
  - $\text{times}(5)(6) = (\lambda y \text{ times}(5,y))(6) = \text{times}(5,6)$

# Logic: Multiple Arguments

- All lambda terms have one argument
- But we can fake multiple arguments ...
- Claim that  $\text{times}(5)(6)$  means same as  $\text{times}(5,6)$ 
  - $\text{times}(5) = (\lambda x \lambda y \text{times}(x,y)) (5) = \lambda y \text{times}(5,y)$ 
    - If this function weren't anonymous, what would we call it?
  - $\text{times}(5)(6) = (\lambda y \text{times}(5,y))(6) = \text{times}(5,6)$
- So we can always get away with 1-arg functions ...
  - ... which might return a function to take the next argument. Whoa.
  - We'll still allow  $\text{times}(x,y)$  as syntactic sugar, though

# Grounding out

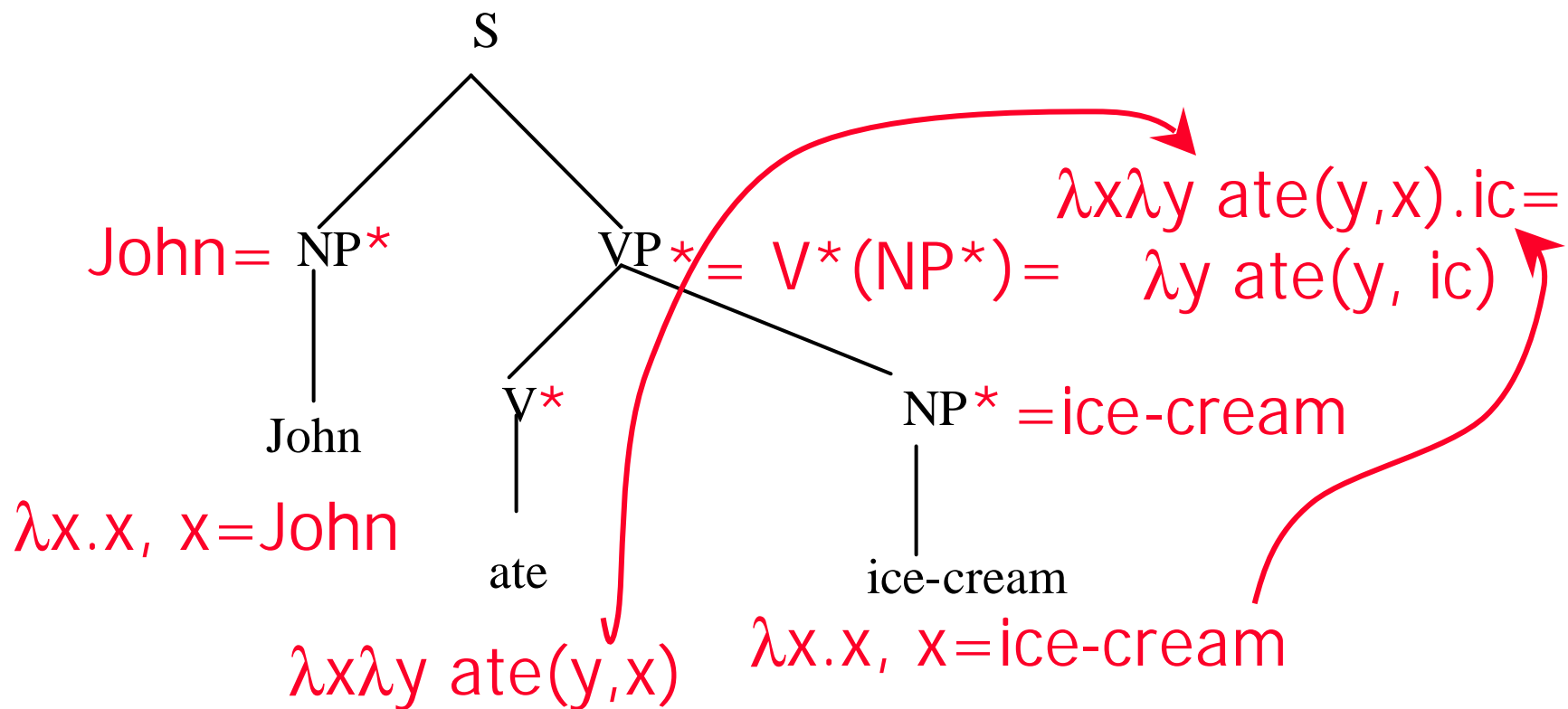
- So what does `times` actually mean???
- How do we get from `times(5,6)` to `30` ?
  - Whether `times(5,6) = 30` depends on whether symbol `times` actually denotes the multiplication function!
- Well, maybe `times` was defined as another lambda term, so substitute to get `times(5,6) = (blah blah blah)(5)(6)`
- But we can't keep doing substitutions forever!
  - Eventually we have to ground out in a **primitive term**
  - Primitive terms are bound to object code
- Maybe `times(5,6)` just executes a multiplication function
- What is executed by `loves(john, mary)` ?

# Logic: Interesting Constants

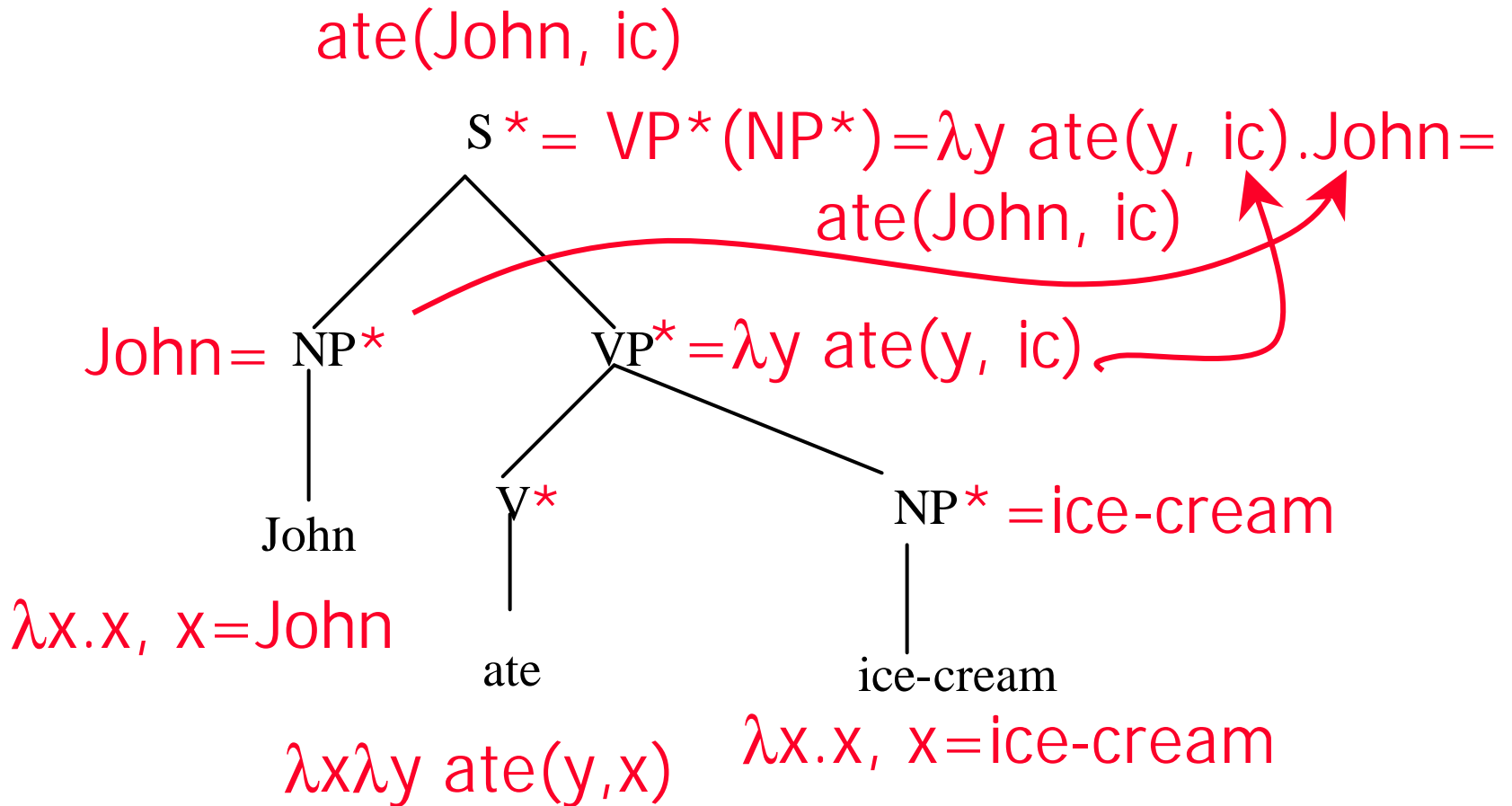
- Thus, have “constants” that name some of the entities and functions (e.g., **times**):
  - **Eminem** - an entity
  - **red** – a predicate on entities
    - holds of just the red entities:  $\text{red}(x)$  is true if  $x$  is red!
  - **loves** – a predicate on 2 entities
    - **loves(Eminem, Detroit)**
    - *Question:* What does **loves(Detroit)** denote?
- Constants used to define meanings of words
- Meanings of phrases will be built from the constants & syntactic structure



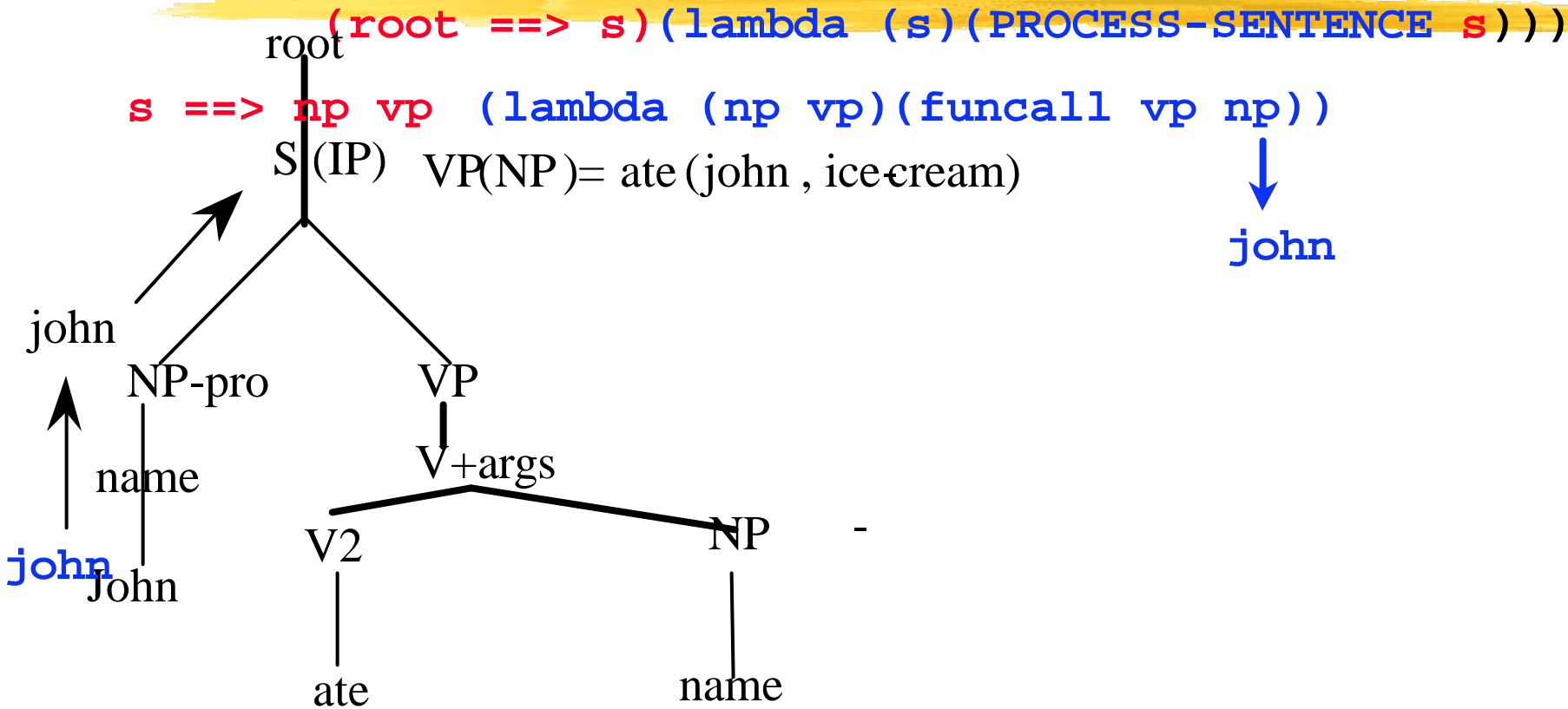
# How: to recover meaning from structure



# How



# Construction step by step – on NP side



np-pro ==> name

#'identity → word-semantics → john

# In this picture



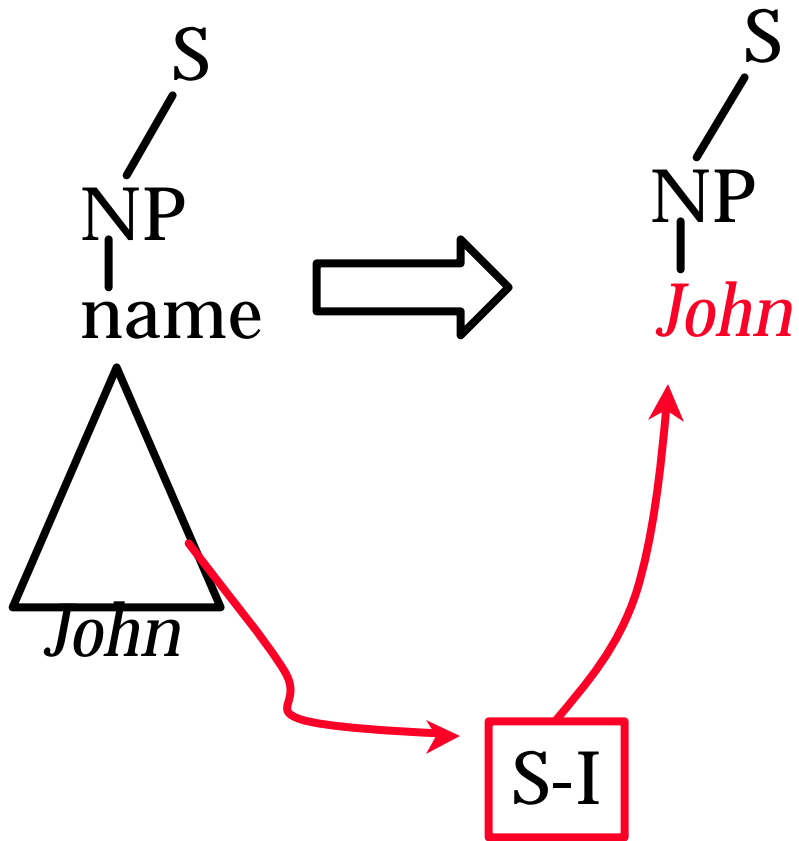
- The meaning of a sentence is the composition of a function  $VP^*$  on an argument  $NP^*$
- The lexical entries are  $\lambda$  forms
  - Simple nouns are just constants
  - Verbs are  $\lambda$  forms indicating their argument structure
- Verb phrases return a function as its result

# Processing order



- *Interpret* subtree as soon as it is built –eg, as soon as RHS of rule is finished (complete subtree)
- Picture: “ship off” subtree to semantic interpretation as soon as it is “done” syntactically
- Allows for off-loading of syntactic short term memory; SI returns with ‘ptr’ to the interpretation
- Natural order to doing things (if process left to right)
- Has some psychological validity – tendency to interpret asap & lower syntactic load
- Example: *I told John a ghost story vs. I told John a ghost story was the last thing I wanted to hear*

# Picture



# Paired syntax-semantics



```
(root ==> s)(lambda (s)(PROCESS-SENTENCE s))
```

```
(s ==> np vp)(lambda (np vp)(funcall vp np))
```