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,...] $\rightarrow$ thrills

projection of features up to a bigger phrase

VP[head=α, tense=β, num=γ...] $\rightarrow$ V[head=α, tense=β, num=γ...] NP

provided α is in the set TRANSITIVE-VERBS

agreement between sister phrases:

S[head=α, tense=β] $\rightarrow$ NP[num=γ,...] VP[head=α, tense=β, num=γ...]

provided α is in the set TRANSITIVE-VERBS

Better approach to factoring linguistic knowledge

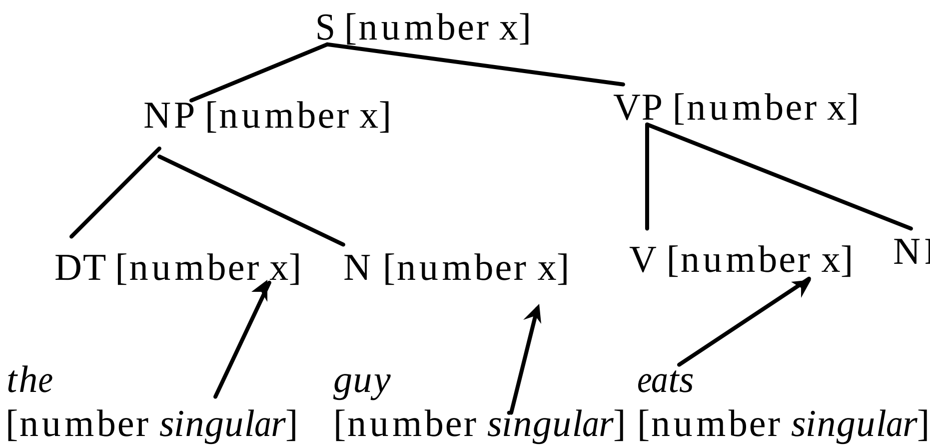
- Use the *superposition* idea: we superimpose one set of constraints on top of another:

1. Basic skeleton tree
2. Plus the added feature constraints

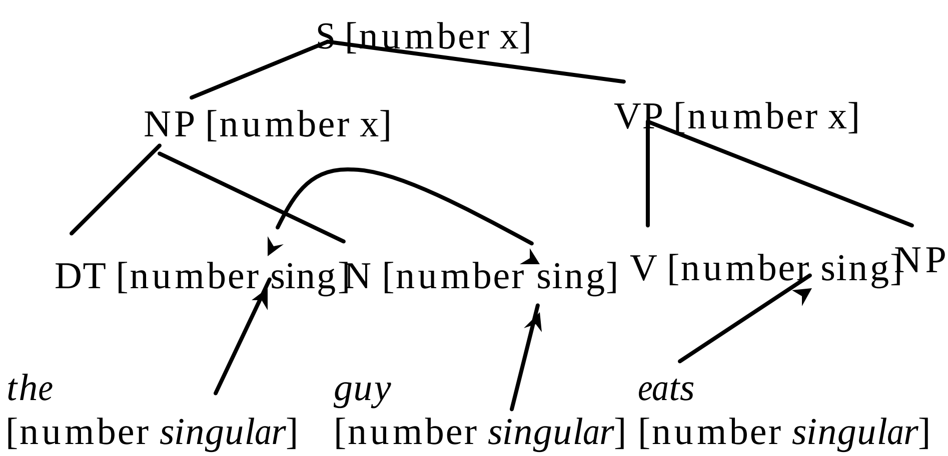
- S $\rightarrow$ NP VP

S	$\rightarrow$	NP	VP
[num x]		[num x]	[num x]
		the guy	*eats*
		[num singular]	[num singular]

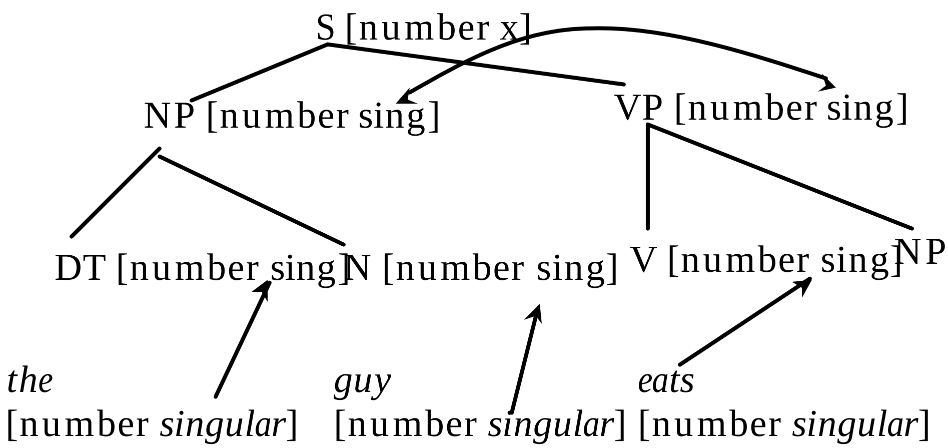
Or in tree form:



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 (1st, 2nd, 3rd);

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 micht strak 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

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

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 $<F, A, p>$
- 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

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

  - 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

$$\begin{bmatrix} Cat & S \\ Head & \begin{bmatrix} Agr & 1 \begin{bmatrix} Num & SG \\ Pers & 3 \end{bmatrix} \\ Subj & \begin{bmatrix} Agr & 1 \end{bmatrix} \end{bmatrix} \end{bmatrix}$$

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

- Number feature

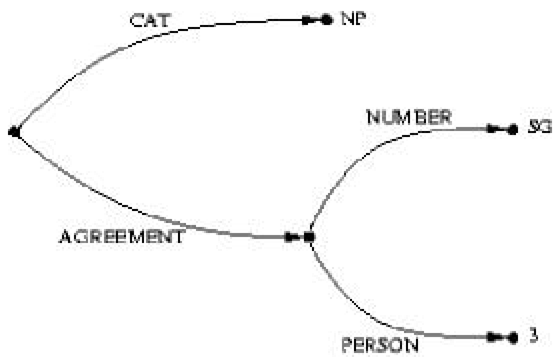
$$\begin{bmatrix} Num & SG \end{bmatrix}$$

- Number-person features

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

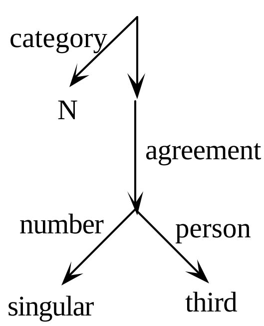
- Number-person-category features (3sgNP) $\begin{bmatrix} Cat & NP \\ Num & SG \\ Pers & 3 \end{bmatrix}$

Graphical Notation for Feature Structures

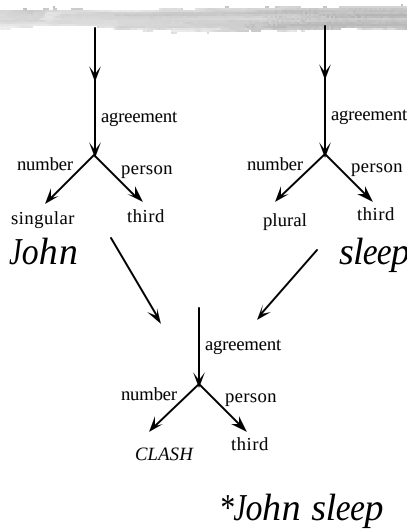


Features and grammars

[category: N
agreement: [person: third
number: singular]]



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
  - [Num SG] U [Num SG] = [Num SG]
  - [Num SG] U [Num PL] fails!
  - [Num SG] U [Num []] = [Num SG]

- [Num SG] U [Pers 3] = $\begin{bmatrix} Num & SG \\ Pers & 3 \end{bmatrix}$

- Structure 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

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

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 $\beta_0 \rightarrow \beta_1 \ldots \beta_n$
  - $<\beta_i$ feature path$>$ = Atomic value
  - $<\beta_i$ feature path$>$ = $<\beta_j$ feature path$>$
- 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 → NP VP

<NP NUM> = <VP NUM>

Head Features

- Features of most grammatical categories are copied from <u>head</u> 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>

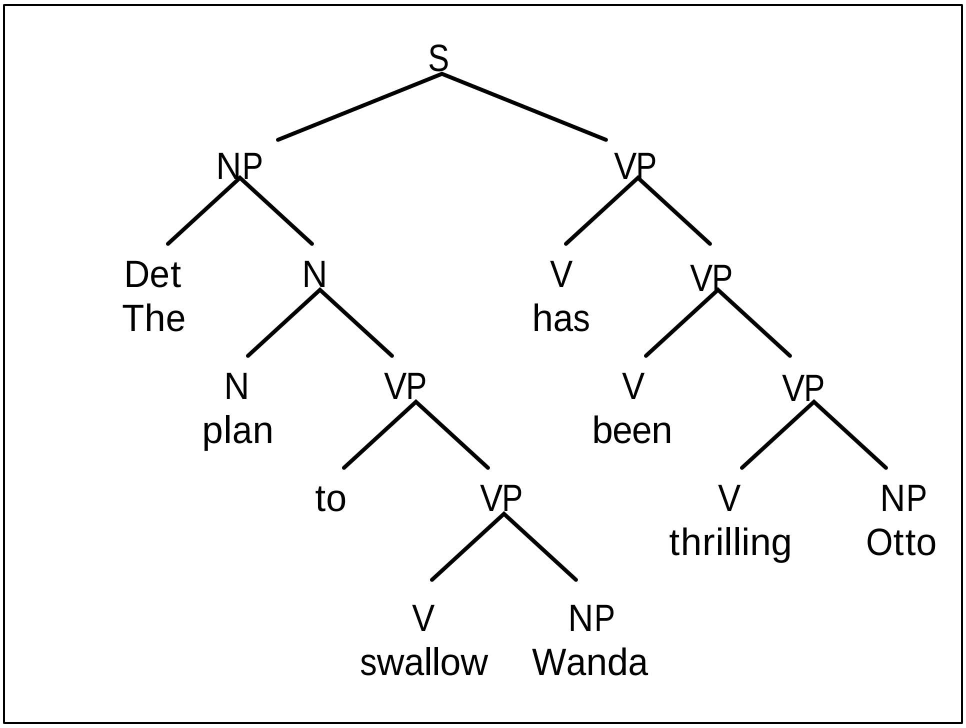
NP → Det Nom

<NP→ HEAD> = <Nom HEAD>

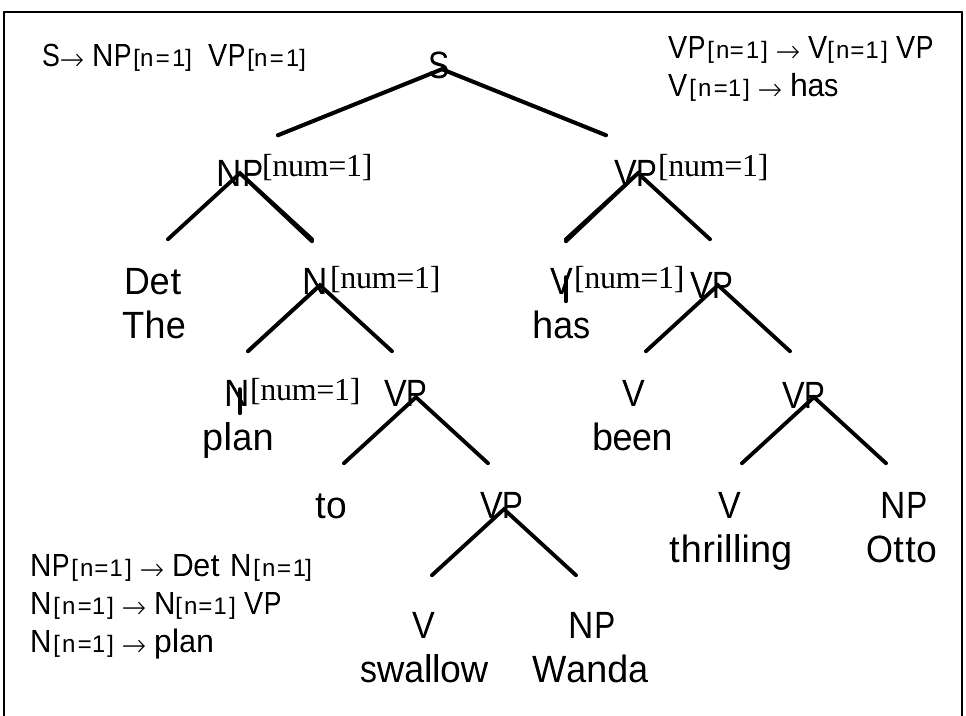
<Det HEAD AGR> = <Nom HEAD AGR>

Nom → N

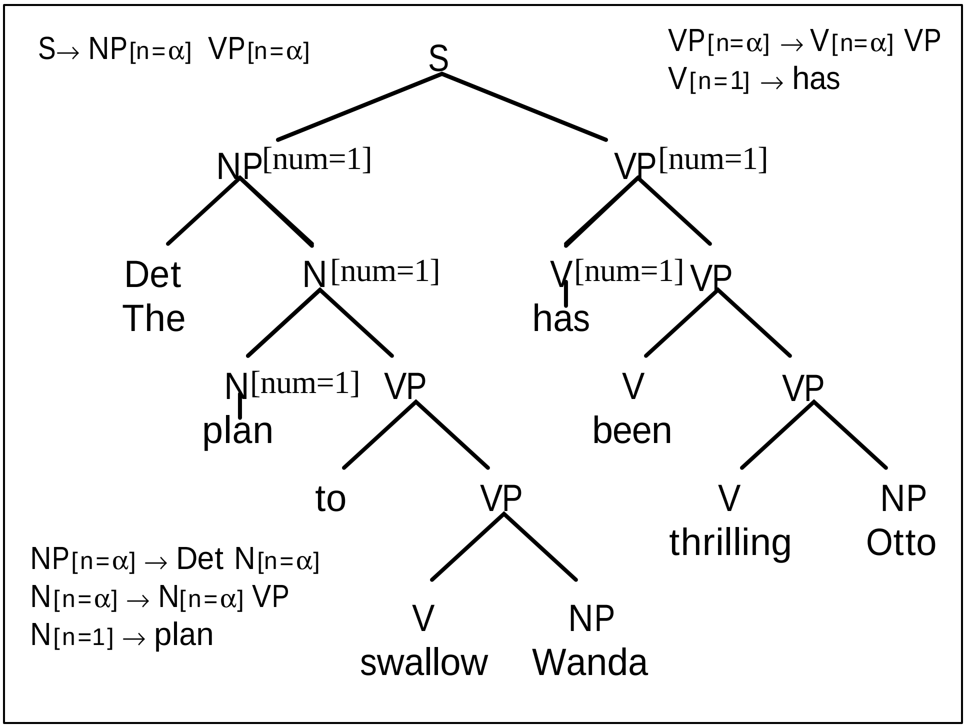
<Nom HEAD> = <N HEAD>

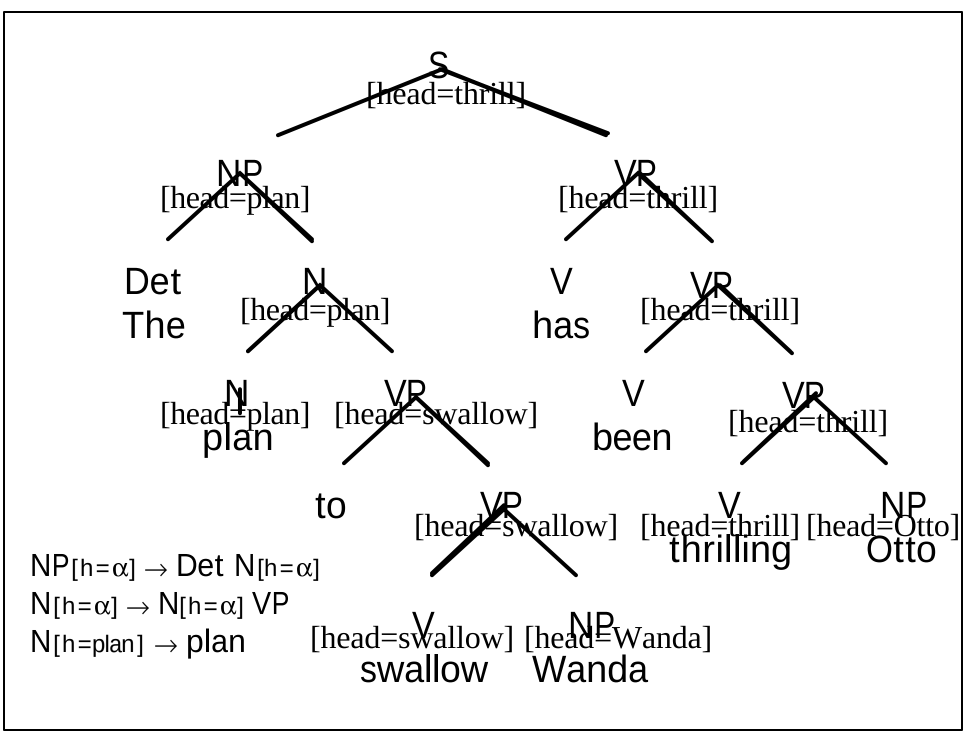


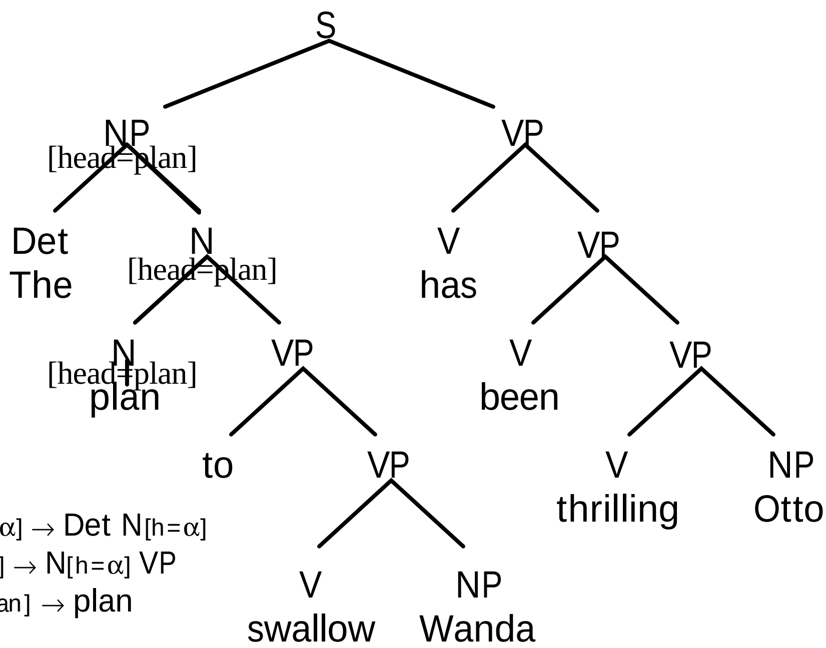
S
NP
VP
Det
The
N
V
has
VP
N
plan
VP
V
been
VP
to
VP
V
thrilling
NP
Otto
V
swallow
NP
Wanda



S→ NP[n=1] VP[n=1]
VP[n=1] → V[n=1] VP
V[n=1] → has
S
NP[num=1]
VP[num=1]
Det
The
N[num=1]
V[num=1]
has
VP
N[num=1]
plan
VP
V
been
VP
to
VP
V
thrilling
NP
Otto
NP[n=1] → Det N[n=1]
N[n=1] → N[n=1] VP
N[n=1] → plan
V
swallow
NP
Wanda

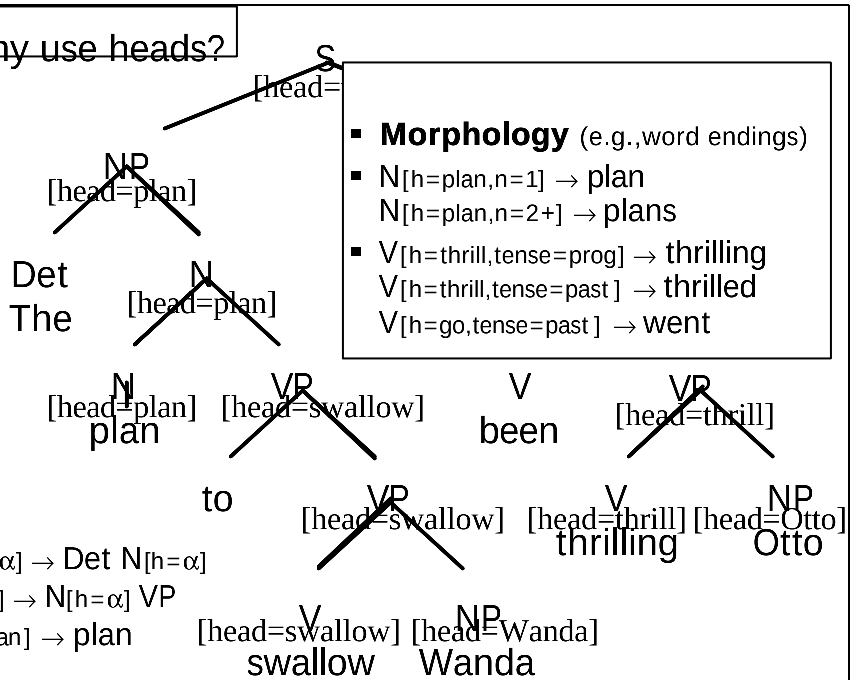

S→ NP[n=α] VP[n=α]
VP[n=α] → V[n=α] VP
V[n=1] → has
S
NP[num=1]
VP[num=1]
Det
The
N[num=1]
V[num=1]
has
VP
N[num=1]
plan
VP
V
been
VP
to
VP
V
thrilling
NP
Otto
NP[n=α] → Det N[n=α]
N[n=α] → N[n=α] VP
N[n=1] → plan
V
swallow
NP
Wanda


S
[head=thrill]
NP
[head=plan]
VP
[head=thrill]
Det
The
N
[head=plan]
V
has
VP
[head=thrill]
N
[head=plan]
plan
VP
[head=swallow]
V
been
VP
[head=thrill]
to
VP
[head=swallow]
V
[head=thrill]
thrilling
NP
[head=Otto]
Otto
NP[h=α] → Det N[h=α]
N[h=α] → N[h=α] VP
N[h=plan] → plan
V
[head=swallow]
swallow
NP
[head=Wanda]
Wanda



$NP_{[h=\alpha]} \rightarrow$ Det $N_{[h=\alpha]}$
$N_{[h=\alpha]} \rightarrow N_{[h=\alpha]}$ VP
$N_{[h=plan]} \rightarrow$ plan

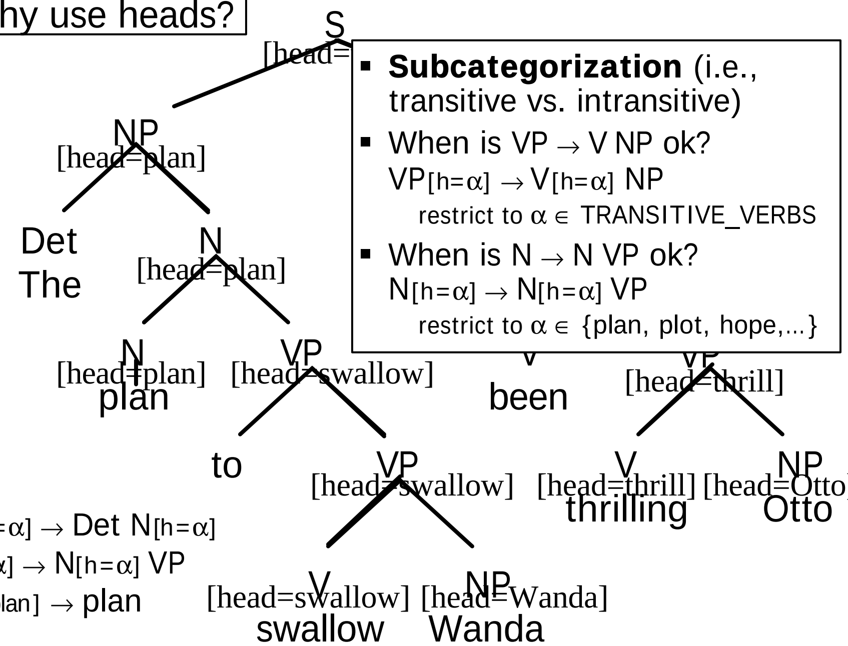
- Why use heads?



- **Morphology** (e.g.,word endings)
- $N_{[h=plan,n=1]} \rightarrow$ plan
  $N_{[h=plan,n=2+]} \rightarrow$ plans
- $V_{[h=thrill,tense=prog]} \rightarrow$ thrilling
  $V_{[h=thrill,tense=past]} \rightarrow$ thrilled
  $V_{[h=go,tense=past]} \rightarrow$ went

$NP_{[h=\alpha]} \rightarrow$ Det $N_{[h=\alpha]}$
$N_{[h=\alpha]} \rightarrow N_{[h=\alpha]}$ VP
$N_{[h=plan]} \rightarrow$ plan

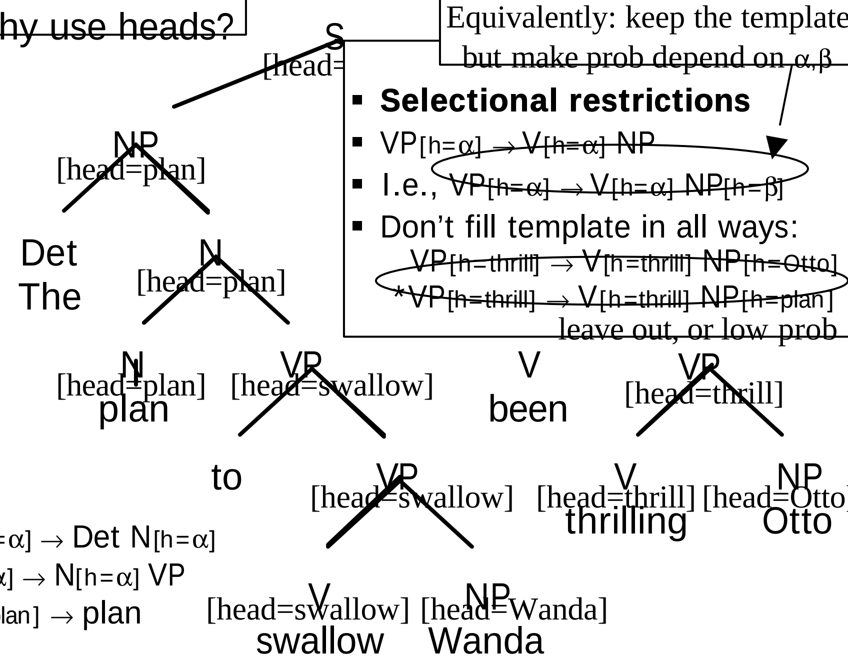
- Why use heads?



- **Subcategorization** (i.e., transitive vs. intransitive)
- When is VP → V NP ok?
  $VP_{[h=\alpha]} \rightarrow V_{[h=\alpha]}$ NP
  restrict to $\alpha \in$ TRANSITIVE_VERBS
- When is N → N VP ok?
  $N_{[h=\alpha]} \rightarrow N_{[h=\alpha]}$ VP
  restrict to $\alpha \in$ {plan, plot, hope,...}

$NP_{[h=\alpha]} \rightarrow$ Det $N_{[h=\alpha]}$
$N_{[h=\alpha]} \rightarrow N_{[h=\alpha]}$ VP
$N_{[h=plan]} \rightarrow$ plan

- Why use heads?



Equivalently: keep the template but make prob depend on α,β

- **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

$NP_{[h=\alpha]} \rightarrow$ Det $N_{[h=\alpha]}$
$N_{[h=\alpha]} \rightarrow N_{[h=\alpha]}$ VP
$N_{[h=plan]} \rightarrow$ plan

How can we parse with feature structures?

- Unification operator: takes 2 features 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_1 trying to find corresponding features in DAG_2 – 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 → NP VP

    <NP HEAD AGR> = <VP HEAD AGR>

    <S HEAD> = <VP HEAD>
  - Add field to states containing DAG representing feature structure corresponding to state of parse, e.g.

    S → • 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:

  $S \rightarrow NP\ VP$

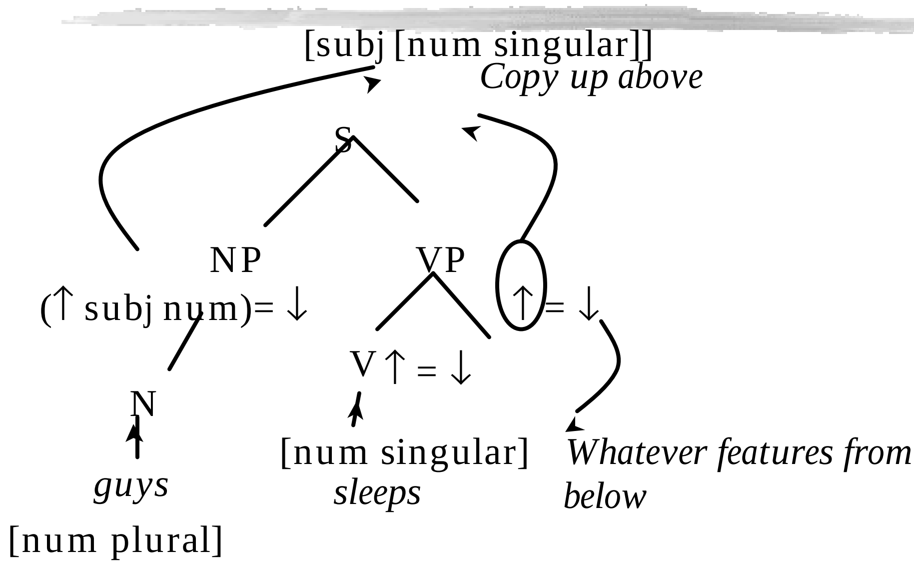
- Add corresponding 'feature checking'

  $S \rightarrow \quad NP \qquad\qquad VP$

  $\qquad (\uparrow \text{subj num}) = \downarrow \quad \uparrow = \downarrow$

- 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 int terms of increasing length in terms of # of words (what's shortest sentence in English?)
- Claim: ∃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
- $A \rightarrow A \quad A$

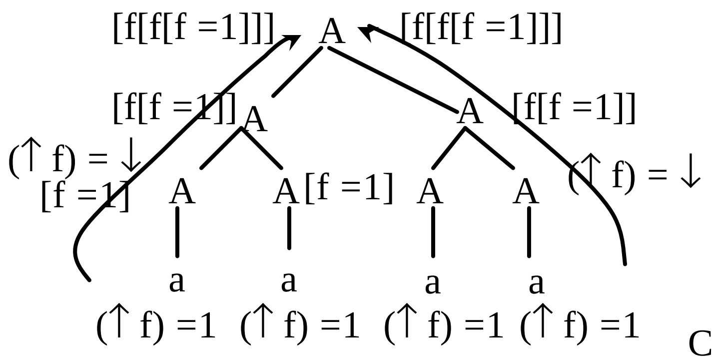
  $(\uparrow f) = \downarrow \quad (\uparrow f) = \downarrow$

  $A \rightarrow a$

  $(\uparrow f) = 1$

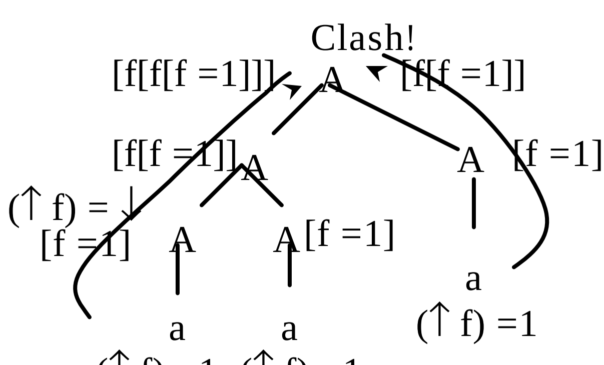
Lets us `count' the number of embeddings on the right & the left – make sure a power of 2

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

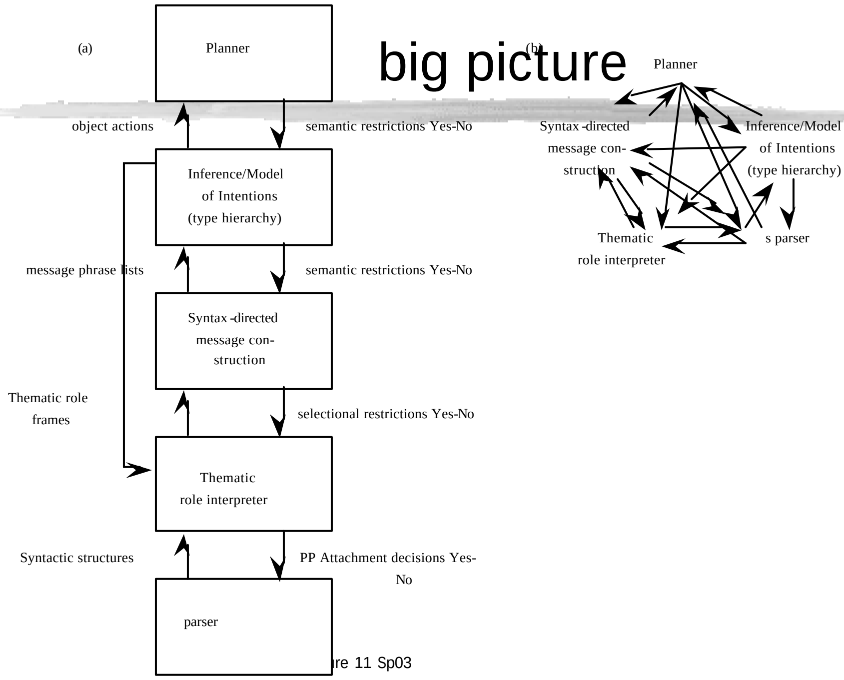
From syntax to meaning

- 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

big picture



Example of what we might do

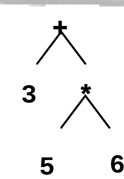
```
athena>(top-level)
Shall I clear the database? (y or n) y
sem-interpret>John saw Mary in the park
OK.
sem-interpret>Where did John see Mary
IN THE PARK.
sem-interpret>John gave Fido to Mary
OK.
sem-interpret>Who gave John Fido
I DON'T KNOW
sem-interpret>Who gave Mary Fido
JOHN
sem-interpret >John saw Fido
OK.
sem-interpret>Who did John see
FIDO AND MARY
```

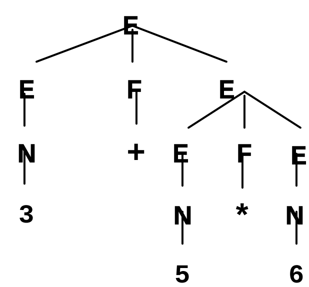
what

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

Analogy w/ prog. language

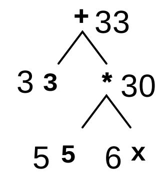
- What is meaning of 3+5*6?
- First parse it into 3+(5*6)

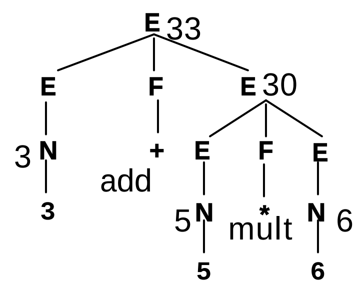




Interpreting in an Environment

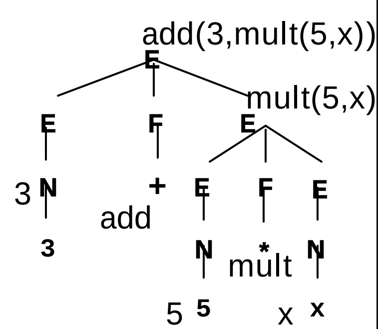
- How about 3+5*x?
- Same thing: the meaning of x is found from the environment (it's 6)
- Analogies in language?





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



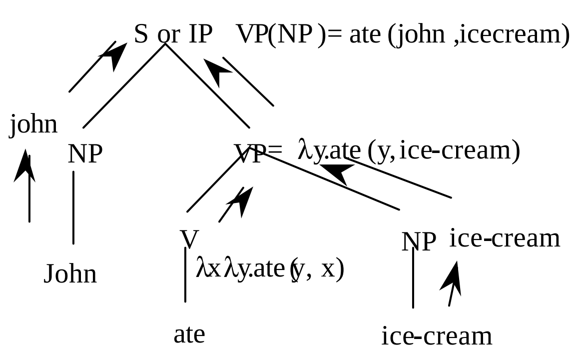
`5*(x+1)-2` is a different expression that produces *equivalent* code (can be converted to the previous code by optimization)

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

How: recover meaning from structure



What Counts as Understanding? some notions

- We understand if we can <u>respond appropriately</u>
  - 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 <u>determine its truth</u>
  - 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) ^ 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”
- Brown(x)$\wedge$ 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’etait 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) → likes(Huey,x)

  cat(Vera) ^ weird(Vera)

  sleeping(Huey) v 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)
- sibling(x,y) ^ cat(x) → cat(y)
- cat(Vera)??

- Limitations:
  - Do ‘and’ and ‘or’ in natural language really mean ‘^’ and ‘v’?

    Mary got married and had a baby.

    Your money or your life!

    He was happy but ignorant.
  - Does ‘→’ 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:

  $\lambda x\ \lambda y$ 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:

  $\lambda x\ \lambda y$ ate(y, x) (ice-cream)(John)→

  λy ate(y, ice-cream)(John)→

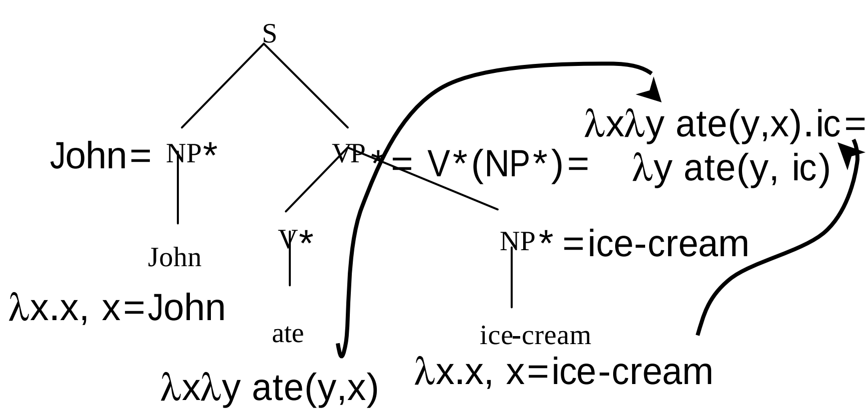
  ate(John, ice-cream)

But... where do we get the λ–forms from?

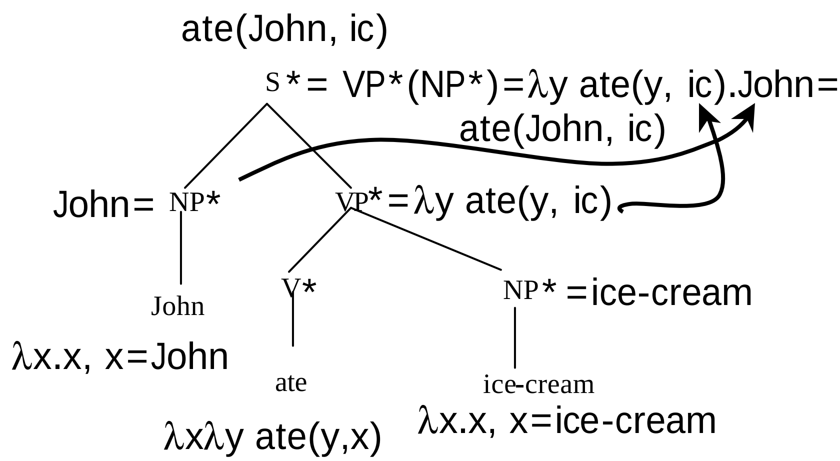
Example of what we now can do

```
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 <u>composition</u> of a function VP* on an argument NP*
- The <u>lexical entries</u> are λ forms
  - Simple nouns are just constants
  - Verbs are λ forms indicating their argument structure
- Verb phrases return <u>λ functions</u> 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 | *Semantic rule*

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

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

(vp ==> v+args)    (lambda (v+args)(lambda (subj)
                       (funcall v+args subj))))

(v+args ==> v2 np)(lambda (v2 np)
                         (lambda (subj)
                             (funcall v2 subj np))))

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

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)• 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	Semantic translation
Verb *ate*	$\lambda x \lambda y.ate(y, x)$
propN	$\lambda x.x$
V	V*= λ for lex entry
S (or CP)	S*= VP*(NP*)
NP	N*
VP	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 = λp p*p
  - Equivalent to int square(p) { return p*p; }
  - But we can talk about λp p*p without naming it
  - Format of a lambda term: λ variable expression

Logic: Lambda Terms

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

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

Logic: Multiple Arguments

- All lambda terms have one argument
- But we can fake multiple arguments ...

- Suppose we want to write times(5,6)
- Remember: square can be written as λx square(x)
- Similarly, times is equivalent to λx λy times(x,y)

- Claim that times(5)(6) means same as times(5,6)
  - times(5) = (λx λy times(x,y)) (5) = λy times(5,y)
    - If this function weren't anonymous, what would we call it?
  - times(5)(6) = (λy times(5,y))(6) = times(5,6)

Logic: Multiple Arguments

- All lambda terms have one argument
- But we can fake multiple arguments ...

- Claim that times(5)(6) means same as times(5,6)
  - times(5) = (λx λy times(x,y)) (5) = λy times(5,y)
    - If this function weren't anonymous, what would we call it?
  - times(5)(6) = (λy times(5,y))(6) = 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 times(x,y) as syntactic sugar, though

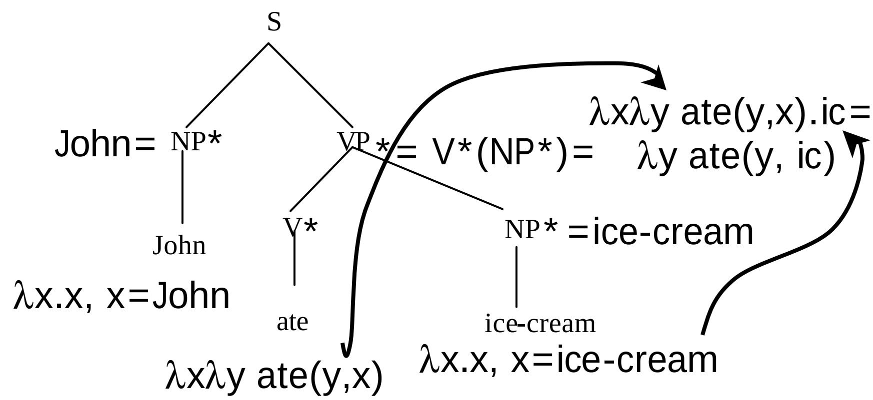
Grounding out

- 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) ?

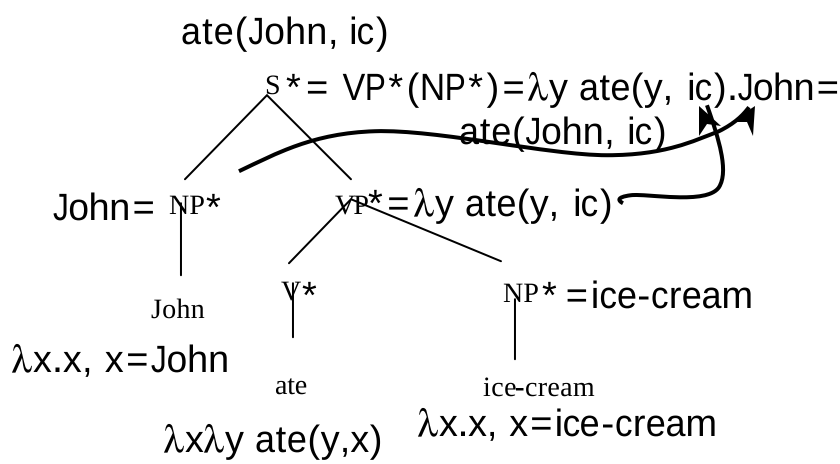
Logic: Interesting Constants

- 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: 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

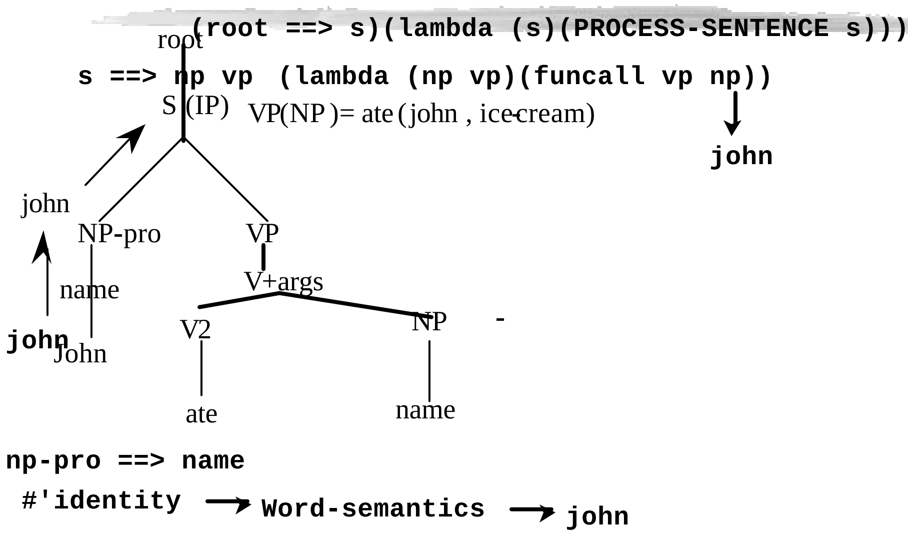
How: to recover meaning from structure



How



Construction step by step – on NP side



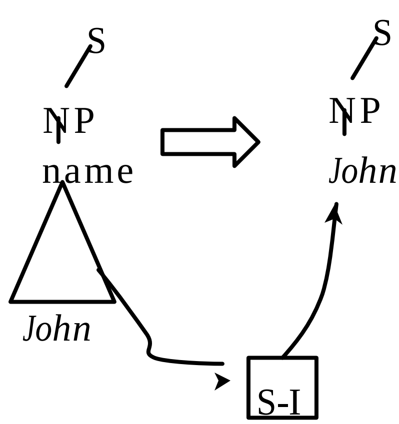
In this picture

- The meaning of a sentence is the composition of a function VP* on an argument NP*
- The lexical entries are λ forms
  - Simple nouns are just constants
  - Verbs are λ 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))
```