6.863J Natural Language Processing
Lecture 13: Semantics II

Robert C. Berwick

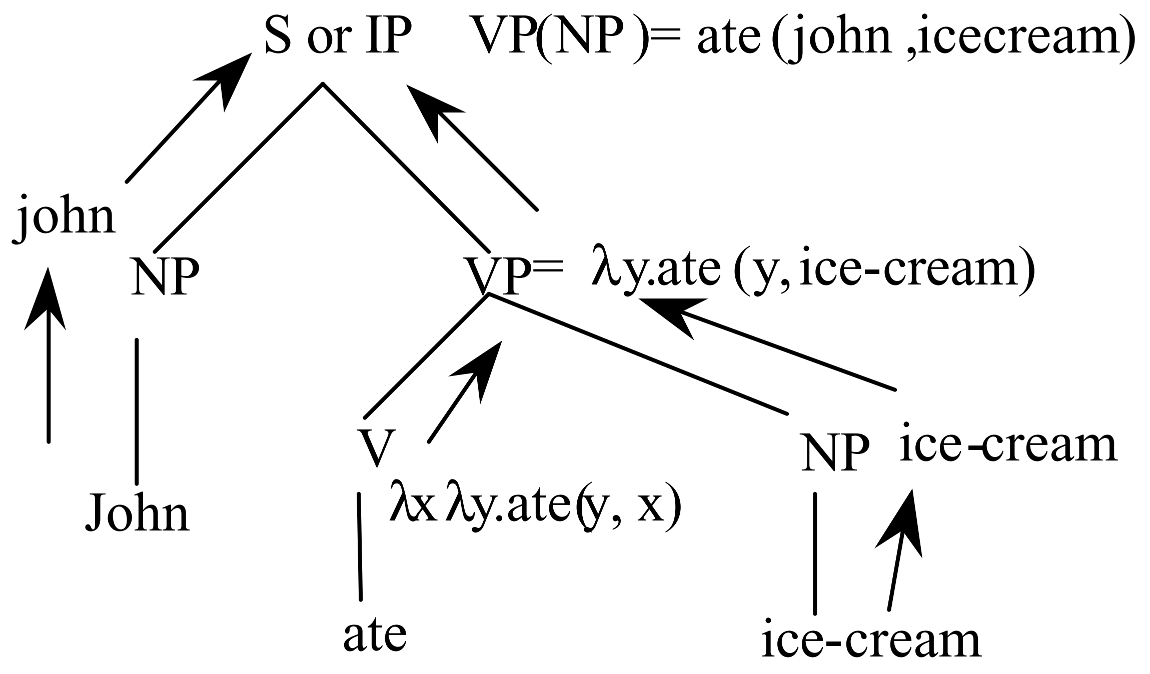
The Menu Bar

- Administrivia:
  - Schedule alert: Lab 3 due today
  - Lab 4: posted; due April 9
- *Agenda:*
- Semantics: the model-theoretic, composition-based view of meaning; example system
- Noun phrase interpretation and quantification
- Details of quantification, semantic representation & evaluation
- Lexical semantics: the meanings of words
- Tense and time

Example of what we might do: text understanding via q-answering

```
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: recover meaning from structure



"Logical" 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

Model theoretic semantics

More specifically, a *model*

1. Consists of a set *D* (the *domain*) and
2. A set of variables, *V;*
3. A function *F* (the *interpretation function*)
4. F assigns to each individual constant a member of *D*;
5. Assigns to each one-place predicate (arity 1) a subset of *D;* to each 2-place predicate (eg, *eat*) a subset of *D* x *D,* etc.

- Our lambda calculus version merely makes use of lambda functions to serve as these functions

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 λ functions as their results (in fact – higher order)

Example

- *John ate ice-cream*
- Top level process-sentence routine used, with the (eventually constructed) interpretation of the S (built from below):

```
(lambda (s)(process-sentence s)
  '(ATE :AGENT JOHN :PATIENT ICE-CREAM
                :TENSE PAST))
```

- `process-sentence` actually does the job of retrieving fact from db, adding fact to db, carrying out an inference, carrying out a robot interface, etc.

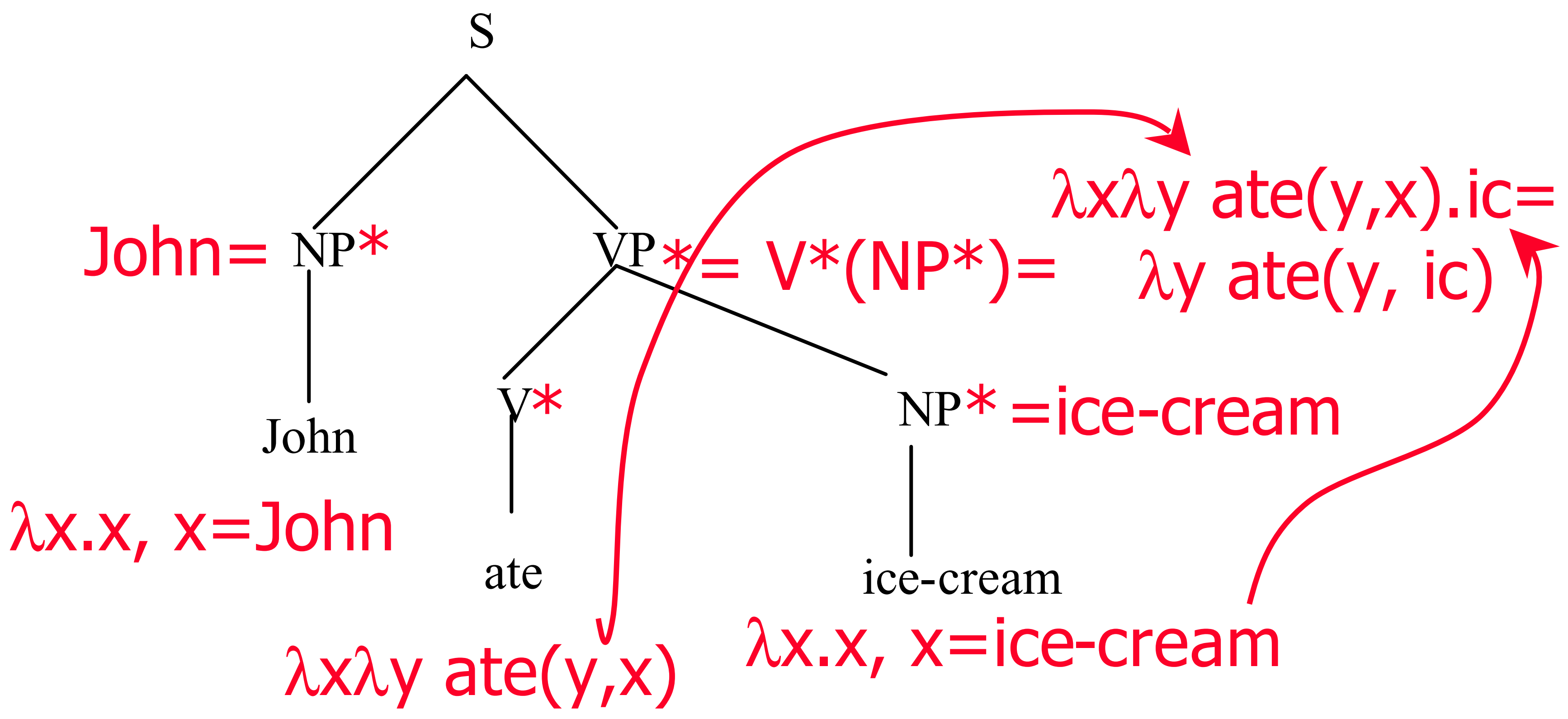
Event structure representation

- Essentially ‘verb frames’
- Needs multiple arity predicate-argument structures, semantic labeling of arguments from predicates, and semantic constraints on the fillers of the arguments
- Existing system in lab has just 3 sorts of ‘process sentence’ dispatches:
  - Assert
  - Retrieve yes-no
  - Retrieve wh question

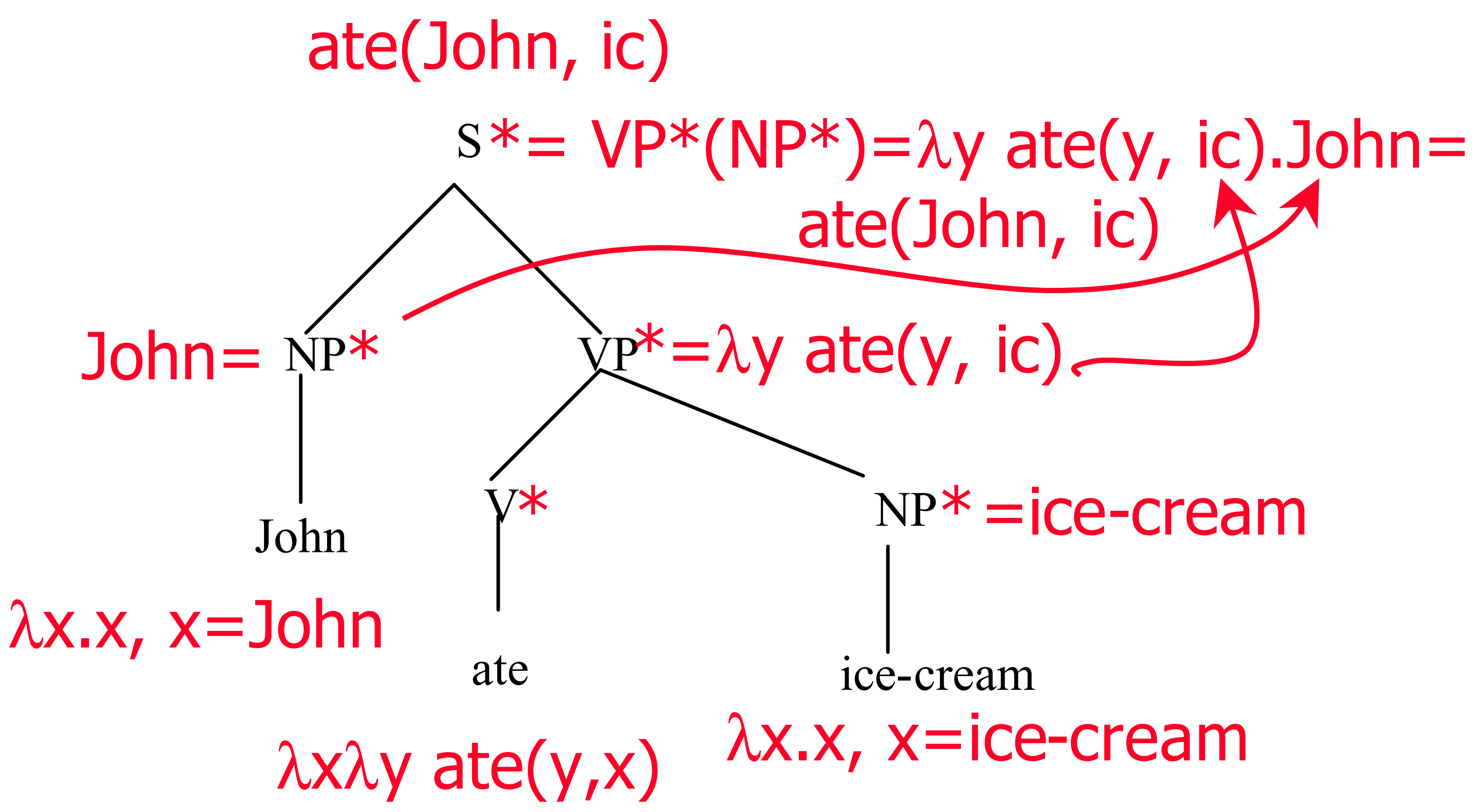
Wh questions

- Part of process-sentence
- Wh form is placed by semantics in template as, eg, ?which or ?who
- This will then correspond to the “for which x, x a person” typed lambda calculus form we wanted – explicitly in a procedural way
- Procedure prompts a *search* through db for matching sets of items that can align w/ the template

How: to recover meaning from structure



How



ate(John, ic)
S*= VP*(NP*)=λy ate(y, ic).John=
ate(John, ic)
John= NP*
VP*=λy ate(y, ic)
V*
NP* =ice-cream
John
λx.x, x=John
ate
ice-cream
λx.x, x=ice-cream
λxλy ate(y,x)

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.

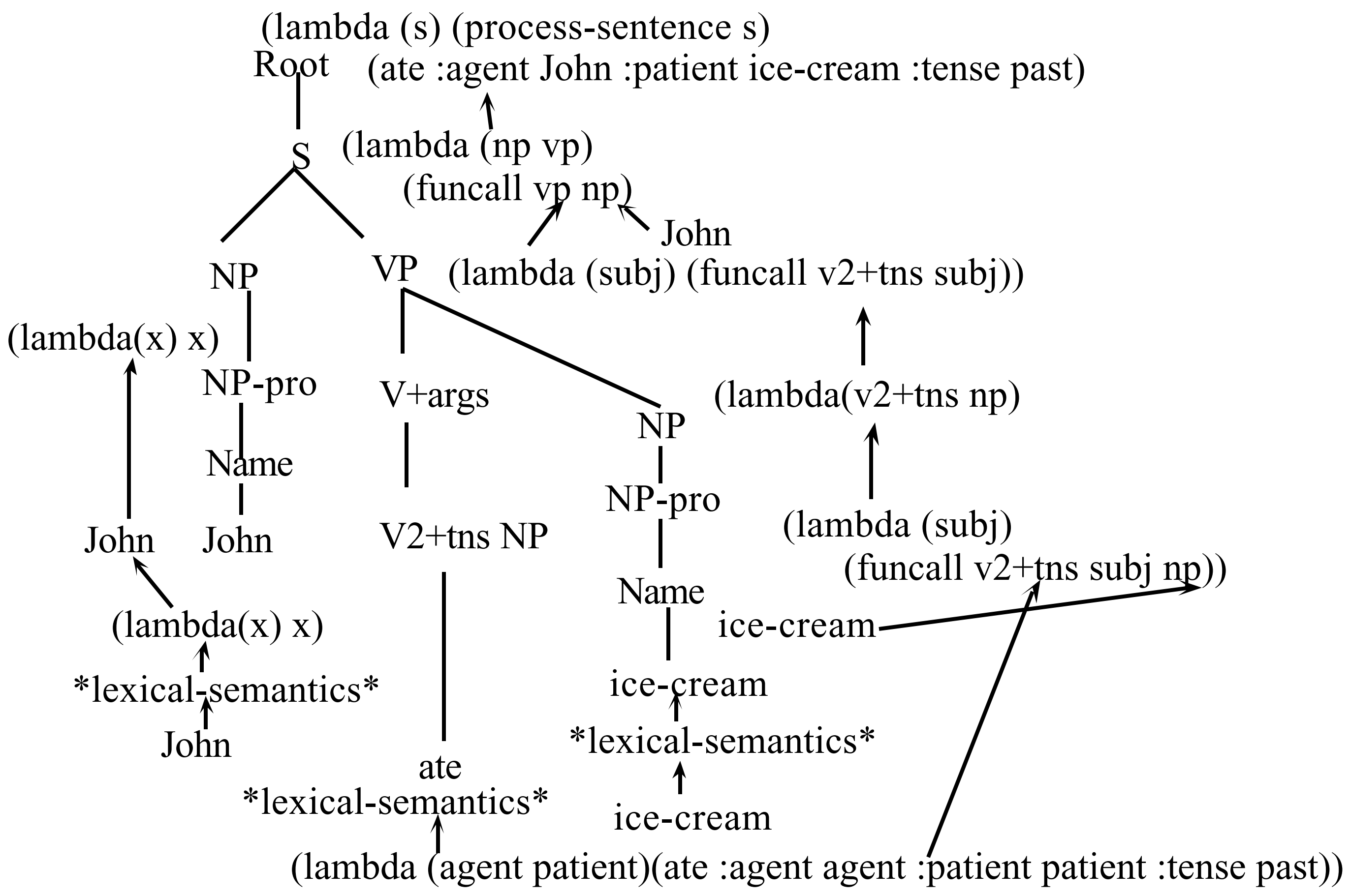
Code – sample rules

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))))

(v kiss)             (lambda (agent beneficiary affcted-obj))

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

Verb arguments

The semantic interpreter procedure



How does this work?

- Top level lambda says to call procedure named VP (whose value will be determined “from below”, ie, S-I of VP) by using the *arg* NP (again whose meaning will be provided “from below)
- In other words, to find the meaning of S, we call the procedure VP using as an argument the subject NP
- These two values will be supplied by the (recursive) semantic interpretation of the NP and VP nodes.
- At the very bottom, individual words must also contain some paired ‘semantic’ value
- This is almost enough to do the code for the whole example!

Code – sample rules

Syntactic rule

Semantic rule

```
add-rule-semantics '(root ==> s)
                    '(lambda (s)
                        (PROCESS-SENTENCE s)))

(add-rule-semantics '(s ==> np vp)
                    #'(lambda (np vp)
                        (funcall vp np)))

(add-rule-semantics '(vp ==> v+args)
                #'(lambda (v+args)
                  #'(lambda (subj)
                      (funcall v+args subj))))

(add-rule-semantics '(v+args ==> v2 np)
                    #'(lambda (v2 np)
                       #'(lambda (subj)
                            (funcall v2 subj np))))

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

Code – the interpreter

```
;;Parse rules into syntactic/semantic parts, recursively
(defun phrase-semantics (phrase)
  (cond ((atom (second phrase))  ; find phrase name –a word?
         (word-semantics (second phrase) (first phrase))) ; o.w.
        (t (rule-apply (rule-semantics (first phrase)   ; recurse
                                       (mapcar
                                              #’first(rest phrase)))
                                        (mapcar #'phrase-semantics
                                              (rest phrase))))))

;; now apply-eval loop for the semantic rules
(defun rule-apply (head args)
  (let ((result (apply head args)))
    (if (and (consp result)
                 (eq (first result) 'lambda))
      (eval (list 'function result))
      result)))
```

Code for this

```
(defun word-semantics (word sense)
  (let ((x (lookup2 word sense *lexical-semantics*)))
    (if (and (consp x)
             (eq (first x) 'lambda))
      (eval (list 'function x))
      x)))

(defun rule-semantics (head args)
  (let ((x (lookup2 head args *phrasal-semantics*)))
    (if (and (consp x)
             (eq (first x) 'lambda))
      (eval (list 'function x))
      x)))
```

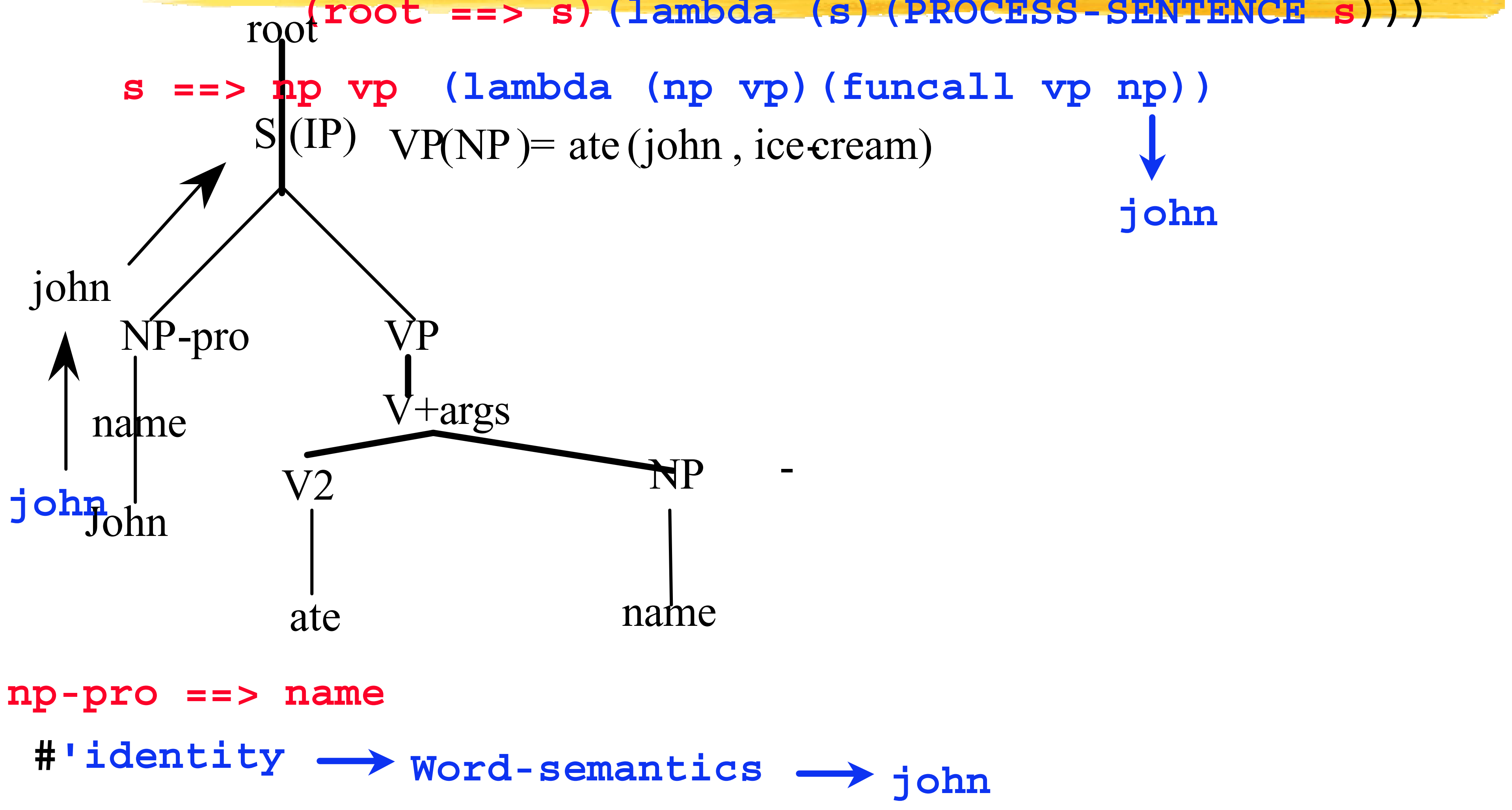
Database that ‘grounds out’

Looks up the lambda form associated w/ a word
Indexed by the head word and then its grammatical category

<u>Example:</u> "eat" is indexed under "eat" and V1 (verb in subcategory 1)
Then the function actually runs the associated lambda
Procedure associated, if any.

<u>Example:</u> (lookup2 'who 'PRONP+wh *lexical-semantics*)
returns ?WHO

Construction step by step – on NP side



(root ==> s)(lambda (s)(PROCESS-SENTENCE s)))
root
s ==> np vp (lambda (np vp)(funcall vp np))
S (IP)
VP(NP)= ate(john , ice-cream)
john
john
NP-pro
VP
name
V+args
john
John
V2
NP
ate
name
np-pro ==> name
#'identity
Word-semantics
john

Example of logical form construction

- *John ate ice-cream*
- Top level `process-sentence` routine used, with the (eventually constructed) interpretation of the S (built from below):

```
(lambda (s)(process-sentence s)
   `(ATE :AGENT JOHN :PATIENT
ICE-CREAM :TENSE PAST))
```

Construction step by step

Let’s elaborate

- What is the interpretation of S?

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

- This needs 2 values: one for the VP, one for NP
- These 2 vals are supplied from below, and substituted via evaluation done by `rule-apply`
- Let’s see where the values come from

Filling in values from below

- The Subject NP value – this is built syntactically as,

```
(NP (NP-PRO (Name John)))
```

- If we look at the 3 rules for these items (NP, NP-pro, Name) we find:

```
(add-r-s '(np ==> #'identity)
(add-r-s '(np-pro ==> #'identity)
(add-r-s 'John 'name 'John)
```

- So the call to `phrase-semantics` just leads to the composition of two identity functions, followed by the constant *John*, I.e, 'John'.

The VP meaning is a <u>higher-order</u> function

- Note that the VP procedure returns a procedure – namely, that procedure object which takes 1 object (the semantic value of the Subject), and calls the function produced by **`V+args`** on it
- Note that the actual semantic value is supplied 'higher up' by the **`funcall`** of **`vp`** on **`np`**
- For each verb subcategory, **`V+args`** constructs a lambda procedure that is a function of the verb and its arguments (excluding the subject); this lambda procedure in turn constructs a basic thematic frame whose values are filled in by the subject and the arguments to the verb

V+args construction

- For *ate,* need this syntactic rule + corresponding paired semantic rule:

verb (a func)

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

arg to verb

- Note what this says: it *expects* 2 arguments to follow

1. The semantic value of **`v2+tns`**
2. The semantic value of an **`np`** (the object np)

- It *returns* a *procedure* that requires one argument as its value: the subject; with the values of **`v2+tns`** **and** **`np`** filled in

What are the values for `v2+tns` and `np`, and Subj?

- **`Np`**: value is ice-cream
- **`Subj:`** value filled in by lambda substitution higher up, as mentioned
- **`V2+tns:`** lexical entry for *ate:*

```
(ate :agent ,agent :patient
            ,patient :tense
                     :past))
```

- Paste this all back together as we unwind to the tops

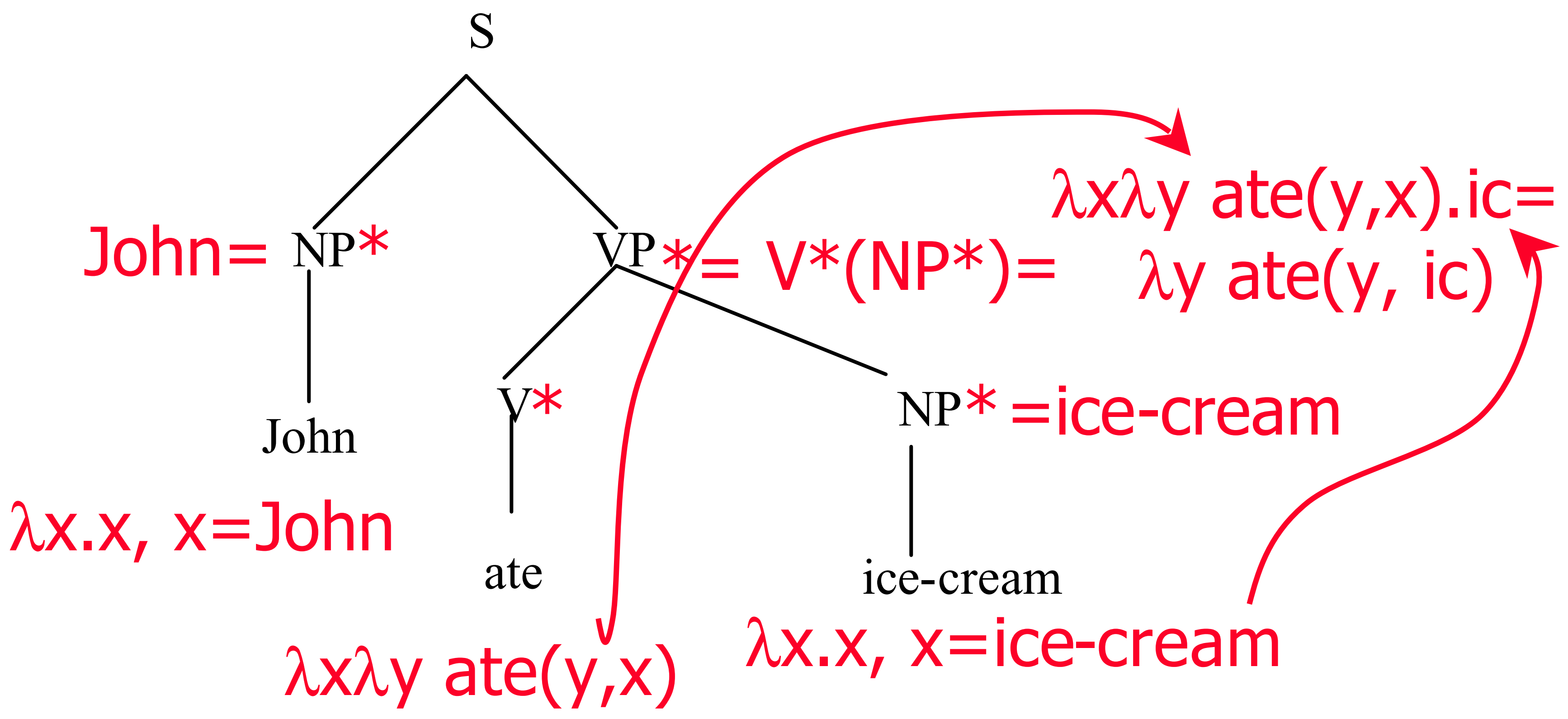
In this picture

- 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

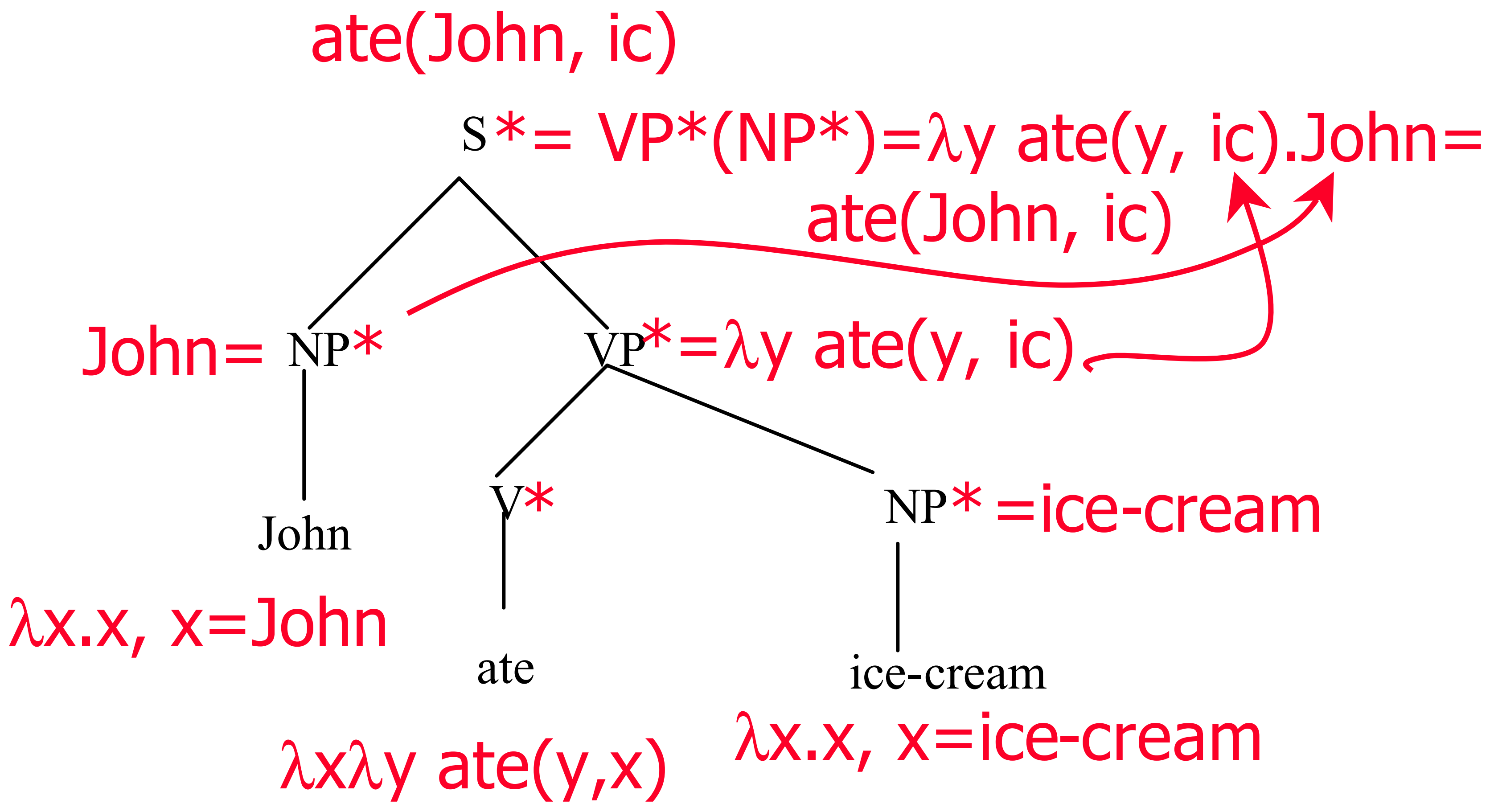
Syntax & paired semantics

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*)

How: to recover meaning from structure



How


ate(John, ic)
S*= VP*(NP*)=λy ate(y, ic).John=
ate(John, ic)
John= NP*
VP*=λy ate(y, ic)
V*
NP* =ice-cream
John
λx.x, x=John
ate
ice-cream
λx.x, x=ice-cream
λxλy ate(y,x)

Processing options

- Off-line vs. on-line
- Off-line: do all syntax first, then pass to semantic interpretation (via pass on syntax tree(s))
- On-line: do it as each phrase is completed

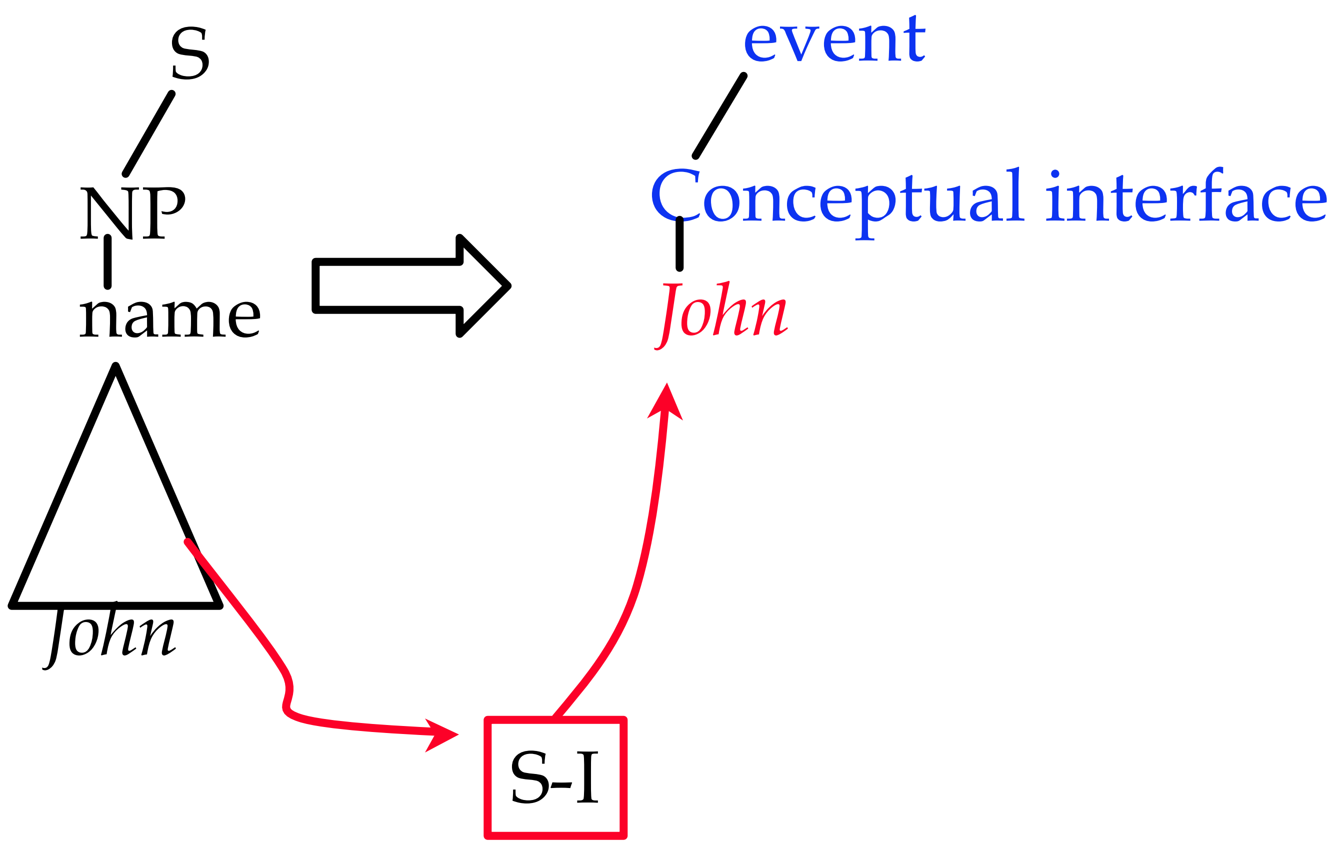
On-line

S → NP VP {VP*(NP*)}

- VP* has been stored in state representing VP
- NP* stored with the state for NP
- When rule completed, go get value of VP*, go get NP*, and apply VP* to NP*
- Store result in S*.

- As fragments of input parsed, semantic fragments created
- Can be used to block ambiguous representations

Picture



Processing order: online

- *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*

Drawback

- You also perform semantic analysis on orphaned constituents that play no role in final parse
- Worst case:
- Jump out the window,
  - But not before you put on your parachute
- Hence, case for pipelined approach: Do semantics *after* syntactic parse

Doing Compositional Semantics

- To incorporate semantics into grammar we must
  - Figure out right representation for a single constituent based on the parts of that constituent (e.g. Adj)
  - Figuring out the right representation for a category of constituents based on other grammar rules making use of that constituent (e.g NP→ Adj Noun)
- This gives us a set of function-like semantic attachments incorporated into our CFG
  - E.g. NP → Adj Noun* {λx Noun*(x) ^ Isa(x,Adj*)}

Non-Compositional Language

- What do we do with language whose meaning isn't derived from the meanings of its parts
  - Metaphor: You're the cream in my coffee.
  - She's the cream in George's coffee.
  - The break-in was just the tip of the iceberg.
  - This was only the tip of Shirley's iceberg.
  - Idioms: The old man finally kicked the bucket.
  - The old man finally kicked the proverbial bucket.
  - (? The bucket was kicked by the old man)
- Solutions?
  - Mix lexical items with special grammar rules?

What do we do with them?

- As we did with feature structures:
  - Alter an Earley-style parser so when constituents (dot at the end of the rule) are completed, the attached semantic function applied and meaning representation created and stored with state
- Or, let parser run to completion and then walk through resulting tree running semantic attachments from bottom-up

What can we do with this machinery?

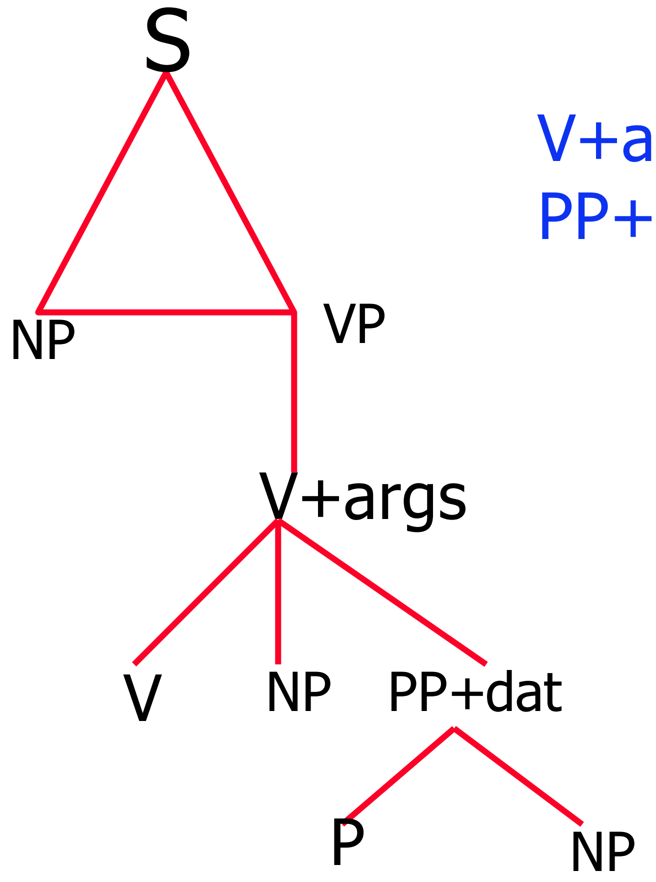
- A lot (almost all): start adding phenomena (figure out the representation) – and see
- To begin: adjs, PPs, wh-moved NPs (which book…), which act *just like* other quantifiers

The lab – an example adding PPto

- John gave Fido to Mary
- We have to add 2 new syntactic rules
- We have to add 2 new semantic rules, and dictionary semantics for any new word meanings

The syntactic rule

- Read off from the corresponding tree



V+args$\rightarrow$ V NP PP+dat
PP+dat$\rightarrow$ P NP

The semantic rules

- Look at what output should be

```
(give :agent john :patient fido
         :beneficiary mary :tense past)
```

- This tells us what verb template should look like

The semantic rules

- PP+dat rule is easy – because we already have semantics for NP

```
'(lambda (p np)
          (funcall p np))


(add-rule-sem    '(pp+dat ==> p np)
                  '(lambda (p np)
                     (funcall p np)))
nb: (add-r-s …)
```

What about dictionary entry for ‘to’

```
(add-word-semantics 'to 'p 'identity)
```

1 semantic rule for v3+tns

- Takes 3 args – namely, rhs of syntax rule
- Args are v3+tns, np, pp+dat
- Want to *return* a function as result
- Template thus:

```
'(lambda (v3+tns np pp+dat)
   `(lambda (subj)
     (funcall subj ,v3+tns ,pp+dat ,np)))
```

1 rule for v+args

```
'(v+args ==> v3+tns np pp+dat)
          '(lambda (v3+tns np pp+dat)
             `(lambda (subj)
               (funcall ,v3+tns subj
                        ',pp+dat ',np)))
```

The semantic rules

- Look at what output should be

```
(give :agent john :patient fido
         :beneficiary mary :tense past)
```

- This tells us what verb template should look like

Last: rule for the verb gave

```
(give :agent john :patient fido
             :beneficiary mary :tense past)


`(lambda (agent beneficiary patient)
      `(',v-tns :agent ,agent
             :patient ,patient
                        :beneficiary ,beneficiary
                              :tense past)))
```

Adjectives & Prepositions

- We take these as essentially restrictions on *sets* (pick out an item from a set)
- We'll elaborate this shortly

Simple Adjectives

- Adjectives modifying nouns: *red book*
- *Add* an NP "frame" as follows, a 'search' template:
- Add slot for :mod as follows:

  (book :mod (:color red))
- To actually *retrieve* objects in database, call *match* function (of some kind)
- Adjectives can be a *list* of modifiers, eg *big red book*
- *Semantics* of this still simple: a filter on the db (conjoined) $big \wedge red = red \wedge big$

Nouns as predicates; adjectives

- wedding
  - λg wedding (g)
- Greek wedding
  - λg greek(g), wedding (g)
- big fat Greek wedding
  - λg big(g), fat(g), Greek(g), wedding(g)
- But: `fake gun' is not the intersection of 'fake' and 'gun'

Beyond simple adjectives

- Still very limited: allows only *one* order and no ambiguity (*adult library card*)
- What could you do to fix this?

Primitive determiners

- *The red book vs. a red book*
- *Add* :number and ?definite components – make it part of search template
- (book ?definite ?det :number singular :mod (:color red))
- This will do a search in the db for matching set of objects that are definite, singular, red

Complex quantifiers

- Logically, quantifier is an ‘iteration’ over a (possibly infinite) Domain, sweeping up objects into a set
- Eg, $\forall$ *x, x green cheese* – iterate over set of cheese, find set of green cheese
- Naturally implemented as a loop
- But we can have nested quantifiers:

Put the block on a pyramid

Quantifiers & scope - briefly

- Everybody loves somebody sometime
- How many people?? Times??

- We will see there are 2 parts to this – representation in the syntax, and the logical representation of determiners/quantifiers
- Why important? To get the right answer
- Each person on a key congressional committee voted against the bill…

The notion of LF

- In general: notion of a *logical form* (LF) that is distinct (perhaps!) from syntax – or, at least, the *apparent surface syntax*
- Evidence: quantifier scope ambiguity; ambiguity in form maps to representational ambiguity (at every level, so at LF too

Everybody loves somebody (sometime)…

$\forall\ \exists$ *vs.* $\exists\ \forall$

Notion of LF

- This has (at least) 2 interpretations (corresponding to 2 different scopings of the quantifiers ∀ and ∃):

  ∀x, x a person, ∃ y, y a person s.t. loves(x, y)

  ∃ y, y a person, ∀ x, x a person s.t. loves(x, y)

PPs

- Again extend NP template (adjectives/modifiers) – insert at end; equivalent to relative clauses
- Example: *book on the table, book which is on the table*

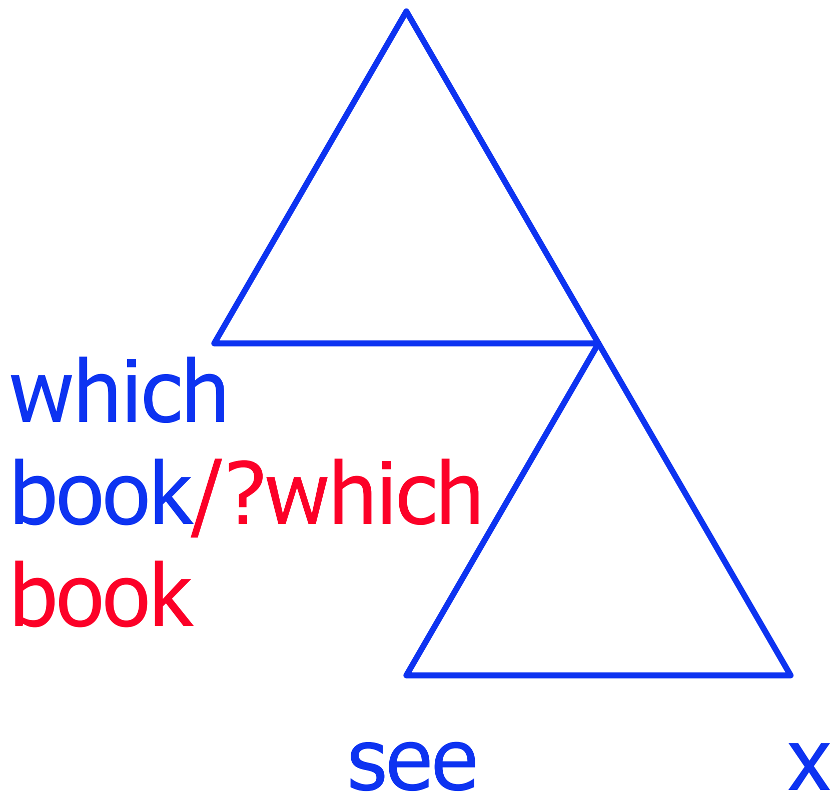
  (book :mod (:support on :mod (:loc table))

(of course, *table* itself would be a recursive, structured object, not just *table*)

Wh questions

- Part of process-sentence
- Wh form is placed by semantics in template as, eg, ?which or ?who
- This will then correspond to the “for which x, x a person” typed lambda calculus form we wanted – explicitly in a procedural way
- Procedure prompts a *search* through db for matching sets of items that can align w/ the template

Picture – wh-NP & variable x exactly in <u>correct</u> configuration



Semantic event forms – more sophisticated events

- Bob put the book on the shelf

```
(cause :agent (bob) :effect (go :theme (book)
       :path (path :oper (on) :terminal+ (shelf)))
                    :tense past))
```

- What did Bob put on the shelf

```
(cause :agent (bob) :effect (go :theme (? (what))
       :path (path :oper (on) :terminal+ (shelf)))
                            :tense past))
```

Linguistic representation: events

- So far: assumed that the predicate representation meaning of a verb has the same # of args as in the verb's subcategorization frame
- Problems:
- Determining the correct # of thematic roles
- Representing facts about these roles
- Ensuring that correct inferences can be derived directly from the representation of an even
- Ensuring that no *incorrect* inferences can be derived

How many verb classes do we need?

- We have to look and see!

Verb Subcategorization

- Intransitive: (1 arg)
  - The light glowed.
  - * He glowed the light.
- Transitive: (2 args)
  - * He devoured.
  - He devoured the apple.
- Intransitive/Transitive: (1 or 2)
  - The door opened.
  - He opened the door.
- Ditransitive: (3 args)
  - * He put.
  - * He put the book.
  - He put the book on the table.
- Ditransitive: (2 or 3 args)
  - * Water poured.
  - Water poured into the sink.
  - * Water poured with the sink.
  - He poured water into the sink.
  - * He poured water with the sink.

… How do we encode knowledge of verb subcategorization?

Traditional cfg (w/ or w/o features)

```
S->NP VP
VP-> V0 Pploc
PPloc -> Ploc NP
V1 -> put
Ploc -> on| ...
```

```
VPass -> V0 PPloc
VP/NP -> V0 NP/NP PPloc
VP/NP -> V0 NP PPloc/NP
PPloc/NP -> Ploc NP/NP
```

Link syntax to ‘lexical conceptual’ structure - Thematic Roles (theta roles)

- Agent Patient Theme Goal Location Source Recipient Experiencer Force (nonvolitional: the wind) Instrument
- Or, does the verb specify its own: Love has a Lover and a Lovee
- Linking theory: mapping between conceptual structure and grammatical function
- Separate out syntax from ‘semantics’ (good?)
- Where do the features come from?
- How do we assign thematic roles?

A universal set of features?

```
put
V
___ NP_j PP_k
[Event CAUSE([Thing   ]_i,
                  [Event GO([Thing   ]_j,
                               [Path
                              TO([Place]_{k})]_{k})]
```

Lexical-Conceptual Structure: Jackendoff (1983)

Indices = ‘links’

Linking = mapping from syntax to thematic roles

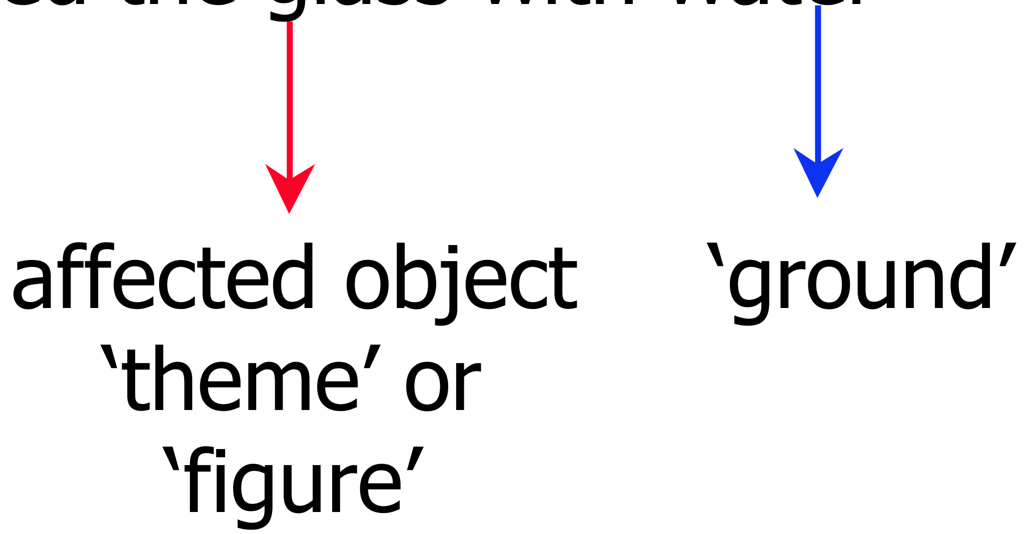
- I poured water into the glass

affected object ‘theme’ or ‘figure’

‘ground’

Linking = mapping from syntax to thematic roles

- I filled the glass with water



Where do thematic roles come from?

- Verbs don't have many arguments. Why not (in principle there could be many - schoenfinkelization)
- The list of so-called universal thematic roles is short (6-10, depending on who you are)
- Is there a principled reason why no verb has more than 3 thematic (theta) roles?
- Why should this be an autonomous linguistic system? If so, why would it be need to be hierarchical?

Verb alternations as litmus test for verb classes

SPRAY/LOAD alternation

- John sprayed the wall with paint
- John sprayed paint on the wall

- John filled the glass with milk
- *John filled milk in the glass

- *John poured the glass with milk
- John poured milk into the glass

- John covered the wall with paint
- *John covered paint on the wall

What is relevant here? Telicity - e.g. finishing the job? Foreground/vs Background?

Verb differences: alternations

- Similar verbs link differently
- So, why

  I poured water into the glass but

  **I filled water into the glass*

- Identical verbs have different alternations

  I gave the book to John/ I gave John the book

  *I donated the book to the library / *I donated the library the book*

  John faxed the message to me/ John faxed me the message

  *John whispered the message to me / *John whispered me the message*

How many classes?

This many… at least

Section9.1 (put arrange immerse install lodge mount place position set situate sling stash stow)
…
Section9.5 (pour dribble drip slop slosh spew spill spurt)

Section9.7 (spray brush cram crowd cultivate dab daub drape drizzle dust hang heap inject jam load mound pack pile plant plaster prick pump rub scatter seed settle sew shower slather smear smudge sow spatter splash splatter spread sprinkle spritz squirt stack stick stock strew string stuff swab vest wash wrap)

Section9.8 (fill adorn anoint bandage bathe bestrew bind blanket block blot bombard carpet choke cloak clog clutter coat contaminate cover dam dapple deck decorate deluge dirty dot douse drench edge embellish emblazon encircle encrust endow enrich entangle face festoon fleck flood frame garland garnish imbue impregnate infect inlay interlace interlard interleave intersperse interweave inundate lard lash line litter mask mottle ornament pad pave plate plug pollute replenish repopulate riddle ring ripple robe saturate season shroud smother soak soil speckle splotch spot staff stain stipple stop up stud suffuse surround swaddle swathe taint tile trim veil vein wreathe)

Levin, 1993 “English verb classes & Alternations” (EVCA)

Verb classes

- 183 Verb Classes
  - 1 entry: 141 classes `(/put/,  /fill/, /butter/, /open/)`
  - 2 entry: 32 classes `(/load/, /give/)`
  - 3+ entries: 10 classes `(/email/)`
  - … `173/183 classes reduced to <`3 entries

Verb classes

- 1.1.2 Causative
  2.4.3/2.4.4 Total Transformation
  5.1 Verbal Passive
  5.2 Prepositional Passive
  1.1.1 Middle (+effect)
  1.3 Conative (+motion, +contact)
  2.12 Body-Part Possessor Ascension Alternation
  7.1 Cognate Object Construction
  7.2 Cognate Prepositional Phrase Construction

- 1.1.3 Substance / Source Alternation
  1.2 Unexpressed Object Alternation
  1.4. Preposition Drop Alternation
  2.1 Dative (give)
  2.2 Benefactive (carve)
  2.3 Locative Alternation
  2.4.1/2.4.2 Material/Product Alternation
  2.6 Fulfilling Alternation
  2.7 Image Impression Alternation
  2.8 With/Against Alternation
  2.9 Through/With Alternation
  2.10 Blame Alternation
  2.11 Search Alternation
  2.14 As Alternation

And more

2.5 Reciprocal Alternations
2.13 Possessor-Attribute Factoring Alternations
3.1 Time Subject Alternation
3.2 Natural Force Subject Alternation
3.3 Instrument Subject Alternation
3.4 Abstract Cause Subject Alternation
3.5 Locatum Subject Alternation
3.6 Location Subject Alternation
3.7 Container Subject Alternation
3.8 Raw Material Subject
3.9 Sum of Money Subject Alternation
3.10 Source Subject Alternation
4.1 Virtual Reflexive Alternation
4.2 Reflexive of Appearance
5.3/5.4 Adjectival Passive
6.1 There-insertion
7.3 Reaction Object Construction
7.4 X's Way Construction
7.5 Resultative Construction
7.6 Unintentional Interpretation of Object
7.7 Bound Nonreflexive Anaphor as Prepositional Object

Summing Up

- Hypothesis: Principle of Compositionality
  - Semantics of NL sentences and phrases can be composed from the semantics of their subparts
- Rules can be derived which map syntactic analysis to semantic representation (Rule-to-Rule Hypothesis)
  - Lambda notation provides a way to extend FOPC to this end
  - But coming up with rule2rule mappings is hard
- Idioms, metaphors perplex the process

PPs

- Again extend NP template (adjectives/modifiers) – insert at end; equivalent to relative clauses
- Example: *book on the table, book which is on the table*

  (book :mod (:support on :mod (:loc table))

(of course, *table* itself would be a recursive, structured object, not just *table*)

Primitive determiners

- *The red book vs. a red book*
- *Add* :number and definite? components – make it part of search template
- (book ?definite ?det :number singular :mod (:color red))
- This will do a search in the db for matching set of objects that are definite, singular, red
- But where do these terms come from in general???
- What does event structure look like???