CMPT 413
Computational Linguistics

Anoop Sarkar
http://www.cs.sfu.ca/~anoop

Semantics

From Syntax to Meaning!

Adapted from slides by Jason Eisner
used in: 600.465 - Intro to NLP - JHU

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
  - Truth can be determined by checking a model
  - A model stores facts about the world, beliefs, etc.
  - There are well-known equivalences between a formula in a logic and model for that formula, so **we map NL into logic**

What Counts as Understanding?
some notions

- We understand a statement if we know *how* to determine its truth
  - A logic is an abstract language of statements such that:
    - Every statement has a model, and
    - A statement can be converted into another statement iff both statements are equivalent according to the same model
  - A statement is true iff it is **satisfiable** in a model
  - What are exact conditions under which it would be true? necessary + sufficient
  - Equivalently, derive all its consequences

Logic: Some Preliminaries

Three major kinds of objects

Booleans

- Roughly, the semantic values of sentences

Entities

- Values of NPs, e.g., objects like this slide
- Maybe also other types of entities, like times

Functions of various types

- A function returning a boolean is called a "predicate"
  – e.g., frog(x), green(x)
- Functions might return other functions!
- Function might take other functions as arguments!

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 **λx** 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)
  - 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.
  - 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):
  - Gilly - 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(Gilly, Lilly)
    - *Question:* What does loves(Lilly) denote?
- Constants used to define meanings of words
- Meanings of phrases will be built from the constants

Logic: Interesting Constants

- most – a predicate on 2 predicates on entities
  - most(pig, big) = "most pigs are big"
    - Equivalently, most(λx pig(x), λx big(x))
  - returns true if most of the things satisfying the first predicate also satisfy the second predicate
- similarly for other quantifiers
  - all(pig,big) (equivalent to ∀x pig(x) ⇒ big(x))
  - exists(pig,big) (equivalent to ∃x pig(x) AND big(x))
  - can even build complex quantifiers from English phrases:
    - "between 12 and 75"; "a majority of"; "all but the smallest 2"

A reasonable representation?

- `Gilly swallowed a goldfish`
- First attempt: swallowed(Gilly, goldfish)

- Returns true or false. Analogous to
  - prime(17)
  - equal(4,2+2)
  - loves(Gilly, Lilly)
  - swallowed(Gilly, Jilly)
- … or is it analogous?

A reasonable representation?

- `Gilly swallowed a goldfish`
  - First attempt: swallowed(Gilly, goldfish)
- But we're not paying attention to `a`!
- `goldfish` isn't the name of a unique object the way `Gilly` is

- In particular, don't want
  `Gilly swallowed a goldfish and Milly swallowed a goldfish`
  to translate as
  swallowed(Gilly, goldfish) AND swallowed(Milly, goldfish)
  since probably not the same goldfish …

Use a Quantifier

- `Gilly swallowed a goldfish`
  - First attempt: swallowed(Gilly, goldfish)
- Better: ∃g goldfish(g) AND swallowed(Gilly, g)
- Or using one of our quantifier predicates:
  - exists(λg goldfish(g), λg swallowed(Gilly,g))
  - Equivalently: exists(goldfish, swallowed(Gilly))
    - "In the set of goldfish there exists one swallowed by Gilly"
- Here goldfish is a predicate on entities
  - This is the same semantic type as red
  - But `goldfish` is noun and `red` is adjective .. #@!?

Tense

- `Gilly swallowed a goldfish`
  - Previous attempt: exists(goldfish, λg swallowed(Gilly,g))
- Improve to use tense:
  - Instead of the 2-arg predicate swallowed(Gilly,g) try a 3-arg version swallow(t,Gilly,g) where t is a time
  - Now we can write: ∃t past(t) AND exists(goldfish, λg swallow(t,Gilly,g))
  - "There was some time in the past such that a goldfish was among the objects swallowed by Gilly at that time"

(Simplify Notation)

- `Gilly swallowed a goldfish`
  - Previous attempt: exists(goldfish, swallowed(Gilly))
- Improve to use tense:
  - Instead of the 2-arg predicate swallowed(Gilly,g) try a 3-arg version swallow(t,Gilly,g)
  - Now we can write:
    ∃t past(t) AND exists(goldfish, swallow(t,Gilly))
  - "There was some time in the past such that a goldfish was among the objects swallowed by Gilly at that time"

Event Properties

- `Gilly swallowed a goldfish`
  - Previous: ∃t past(t) AND exists(goldfish, swallow(t,Gilly))
- Why stop at time? An event has other properties:
  - `[Gilly] swallowed [a goldfish] [on a dare] [in a telephone booth] [with 30 other freshmen] [after many bottles of vodka had been consumed].`
  - Specifies who what why when …
- Replace time variable t with an event variable e
  - ∃e past(e), act(e,swallowing), swallower(e,Gilly), exists(goldfish, swallowee(e)), exists(booth, location(e)), …
    - As with probability notation, a comma represents AND
    - Could define past as λe ∃t before(t,now), ended-at(e,t)

Quantifier Order

- `Gilly swallowed a goldfish in a booth`
  - ∃e past(e), act(e,swallowing), swallower(e,Gilly), exists(goldfish, swallowee(e)), exists(booth, location(e)), …
- `Gilly swallowed a goldfish in every booth`
  - ∃e past(e), act(e,swallowing), swallower(e,Gilly), exists(goldfish, swallowee(e)), all(booth, location(e)), …

∃g goldfish(g), swallowee(e,g) ∀b booth(b)⇒location(e,b)

- Does this mean what we'd expect??

says that there's only one event
with a single goldfish getting swallowed
that took place in a lot of booths ...

Quantifier Order

- Groucho Marx celebrates quantifier order ambiguity:
  - In this country a woman gives birth every 15 min. Our job is to find that woman and stop her.

  ∃woman (∀15min gives-birth-during(woman, 15min))

  ∀15min (∃woman gives-birth-during(15min, woman))
  - Surprisingly, both are possible in natural language!
  - Which is the joke meaning (where it's always the same woman) and why?
- What about:
  - Every prof admires, and every student detests, some course.

Quantifier Order

- `Gilly swallowed a goldfish in a booth`
  - ∃e past(e), act(e,swallowing), swallower(e,Gilly), exists(goldfish, swallowee(e)), exists(booth, location(e)), …
- `Gilly swallowed a goldfish in every booth`
  - ∃e past(e), act(e,swallowing), swallower(e,Gilly), exists(goldfish, swallowee(e)), all(booth, location(e)), …

∃g goldfish(g), swallowee(e,g) ∀b booth(b)⇒location(e,b)

- Does this mean what we'd expect??
  - It's ∃e ∀b which means same event for every booth
  - Probably false unless Gilly can be in every booth during her swallowing of a single goldfish

Quantifier Order

- `Gilly swallowed a goldfish in a booth`
  - ∃e past(e), act(e,swallowing), swallower(e,Gilly), exists(goldfish, swallowee(e)), exists(booth, location(e)), …
- `Gilly swallowed a goldfish in every booth`
  - ∃e past(e), act(e,swallowing), swallower(e,Gilly), exists(goldfish, swallowee(e)), all(booth, λb location(e,b))
- Other reading (∀b ∃e) involves quantifier raising:
  - all(booth, λb [∃e past(e), act(e,swallowing), swallower (e,Gilly), exists(goldfish, swallowee(e)), location(e,b)])
  - "for all booths b, there was such an event in b"

Nouns and Their Modifiers

- expert
  - λg expert(g)
- big fat expert
  - λg big(g), fat(g), expert(g)
  - But: bogus expert
    - Wrong: λg bogus(g), expert(g)
    - Right: λg (bogus(expert))(g) … bogus maps to new concept
- Baltimore expert (white-collar expert, TV expert …)
  - λg Related(Baltimore, g), expert(g) – expert from Baltimore
  - Or λg (Modified-by(Baltimore, expert))(g) – expert on Baltimore
  - Can't use Related for that case: law expert and dog catcher
    = λg Related(law,g), expert(g), Related(dog, g), catcher(g)
    = dog expert and law catcher

Nouns and Their Modifiers

- the goldfish that Gilly swallowed
- every goldfish that Gilly swallowed
- three goldfish that Gilly swallowed

λg [goldfish(g), swallowed(Gilly, g)]

like an adjective!

- three swallowed-by-Gilly goldfish

Or for real: λg [goldfish(g), ∃e [past(e), act(e,swallowing), swallower(e,Gilly), swallowee(e,g)]]

Adverbs

- `Lilly passionately wants Billy`
  - Wrong?: passionately(want(Lilly,Billy)) = passionately(true)
  - Better: (passionately(want))(Lilly,Billy)
  - Best: ∃e present(e), act(e,wanting), wanter(e,Lilly), wantee(e, Billy), manner(e, passionate)
- `Lilly often stalks Billy`
  - (often(stalk))(Lilly,Billy)
  - many(day, λd ∃e present(e), act(e,stalking), stalker(e,Lilly), stalkee(e, Billy), during(e,d))
- `Lilly obviously likes Billy`
  - (obviously(like))(Lilly,Billy) – one reading
  - obvious(likes(Lilly, Billy)) – another reading

Speech Acts

- What is the meaning of a full sentence?
  - Depends on the punctuation mark!?
  - Billy likes Lilly. → **assert**(like(B,L))
  - Billy likes Lilly? → **ask**(like(B,L))
    - or more formally, “Does Billy like Lilly?”
  - Billy, like Lilly! → **command**(like(B,L))

- Let’s try to do this a little more precisely, using event variables etc.

Speech Acts

- `What did Gilly swallow?`
  - **ask**(λx ∃e past(e), act(e,swallowing), swallower(e,Gilly), swallowee(e,x))
  - Argument is identical to the modifier “that Gilly swallowed”
  - Is there any common syntax?
- `Eat your fish!`
  - **command**(λf act(f,eating), eater(f,Hearer), eatee(…))
- `I ate my fish.`
  - **assert**(∃e past(e), act(e,eating), eater(f,Speaker), eatee(…))

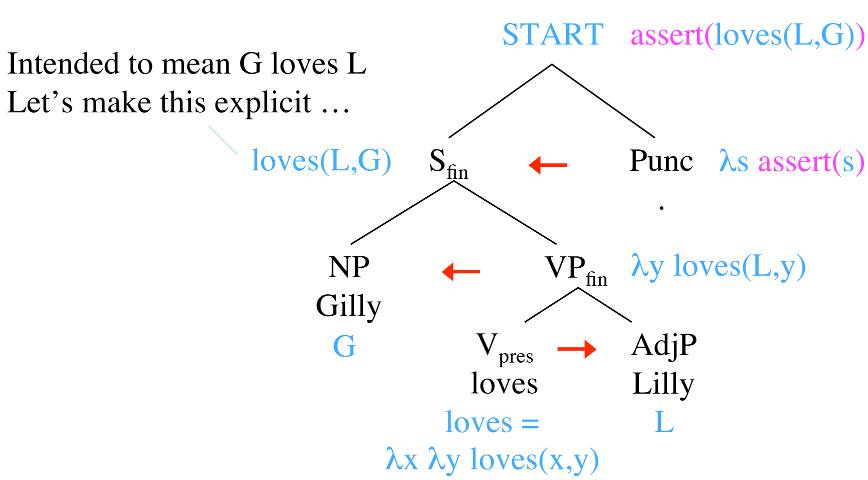
Compositional Semantics

- We’ve discussed what semantic representations should look like.
- **But how do we get them from sentences???**
- First - parse to get a syntax tree.
- Second - look up the semantics for each word.
- Third - build the semantics for each constituent
  - Work from the bottom up
  - The syntax tree is a “recipe” for how to do it

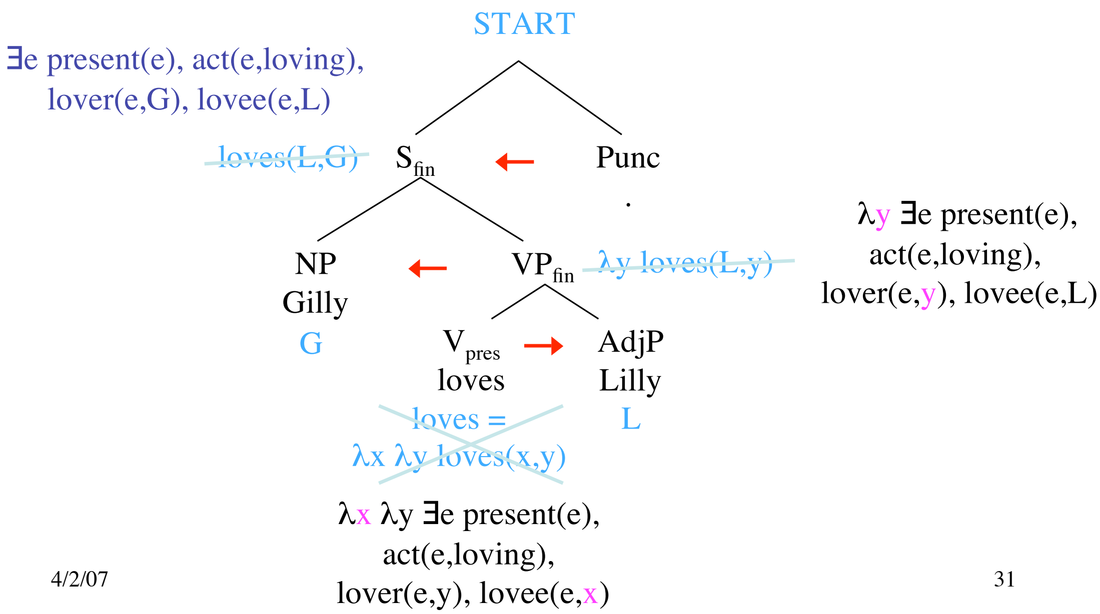
Compositional Semantics

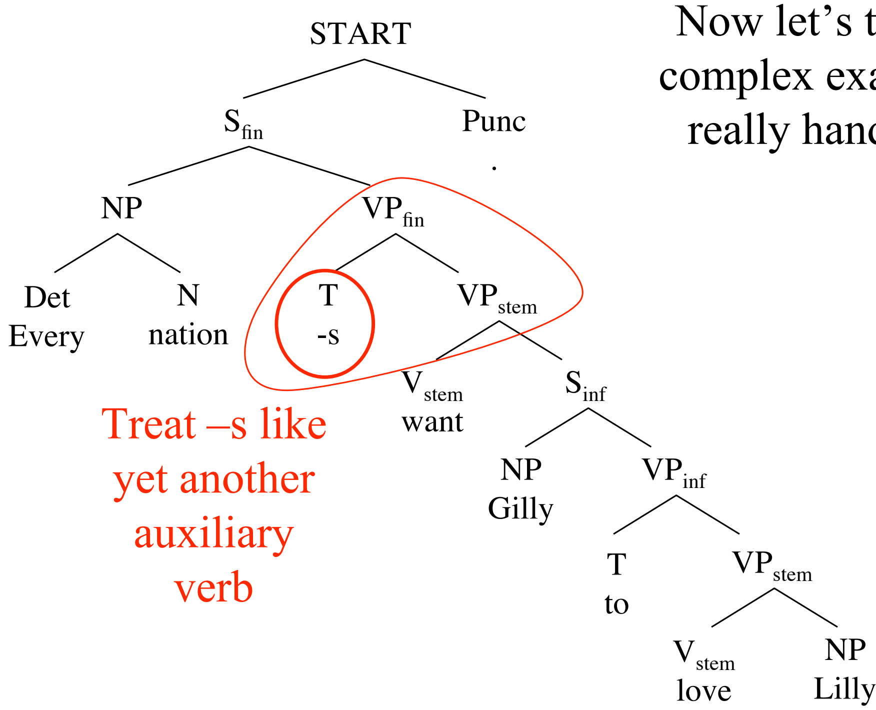
- Instead of S → NP loves NP
  - S[sem=loves(x,y)] → NP[sem=x] loves NP[sem=y]
- might want general rules like S → NP VP:
  - V[sem=loves] → loves
  - VP[sem=v(obj)] → V[sem=v] NP[sem=obj]
  - S[sem=vp(subj)] → NP[sem=subj] VP[sem=vp]
- Now `Gilly loves Lilly` has sem=loves(Lilly)(Gilly)
- In this manner we'll sketch a version where
  - Still compute semantics bottom-up
  - Grammar is in Chomsky Normal Form
  - So each node has 2 children: 1 function & 1 argument
  - To get its semantics, apply function to argument!

Compositional Semantics

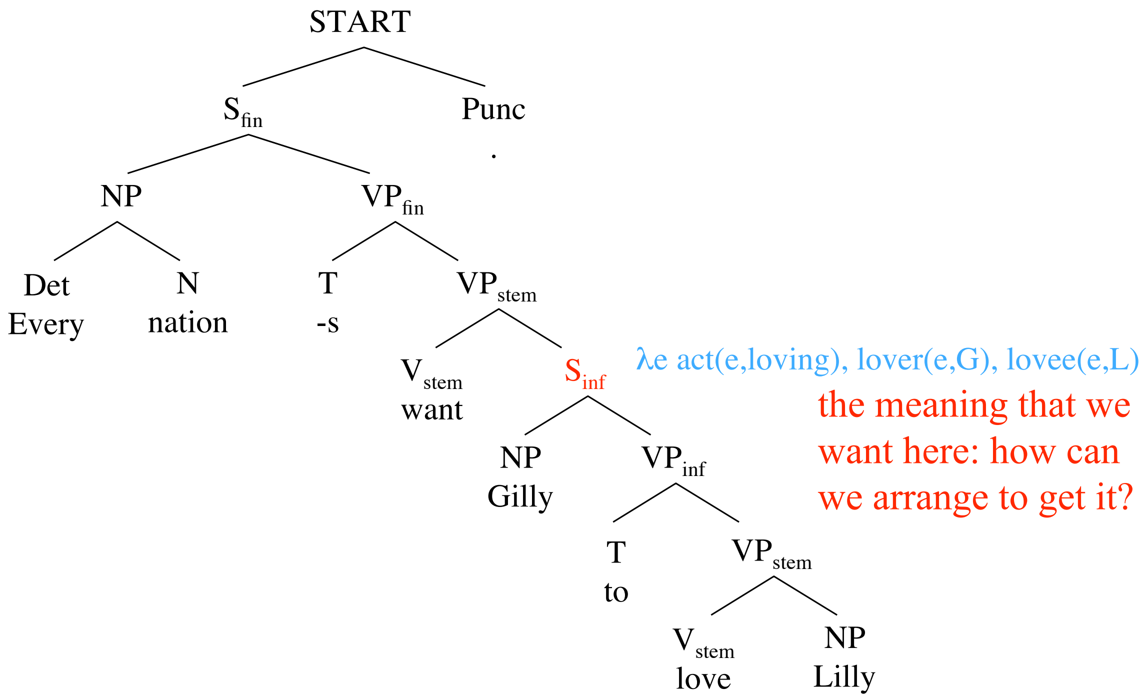


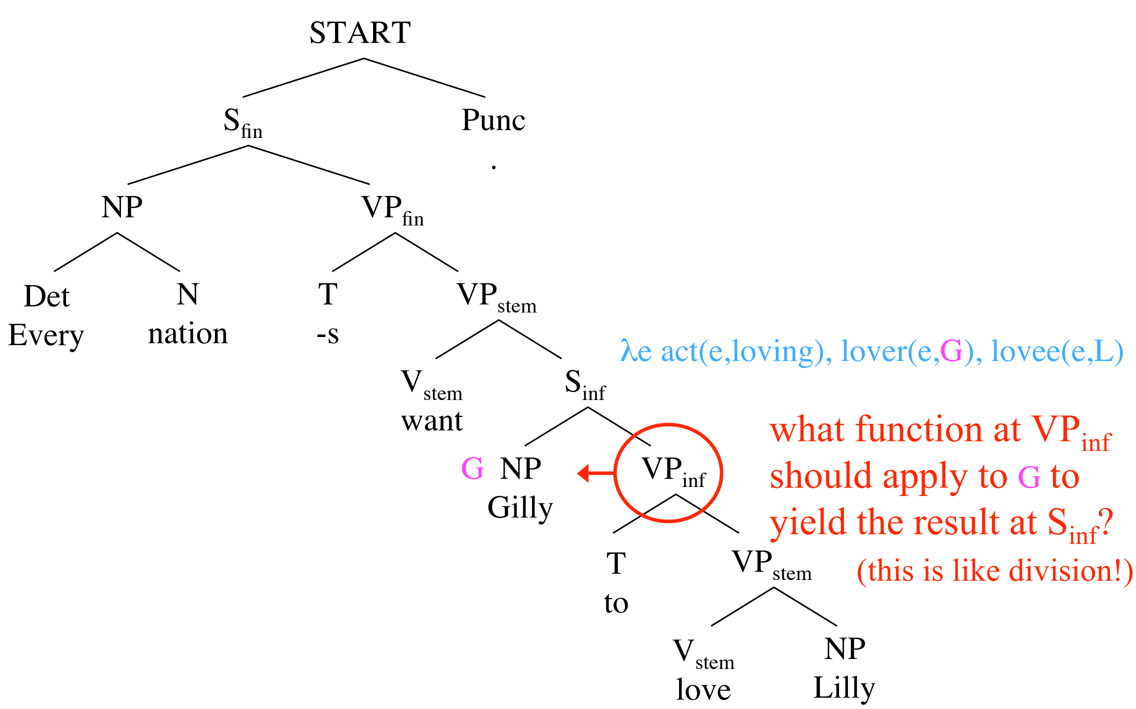
Compositional Semantics

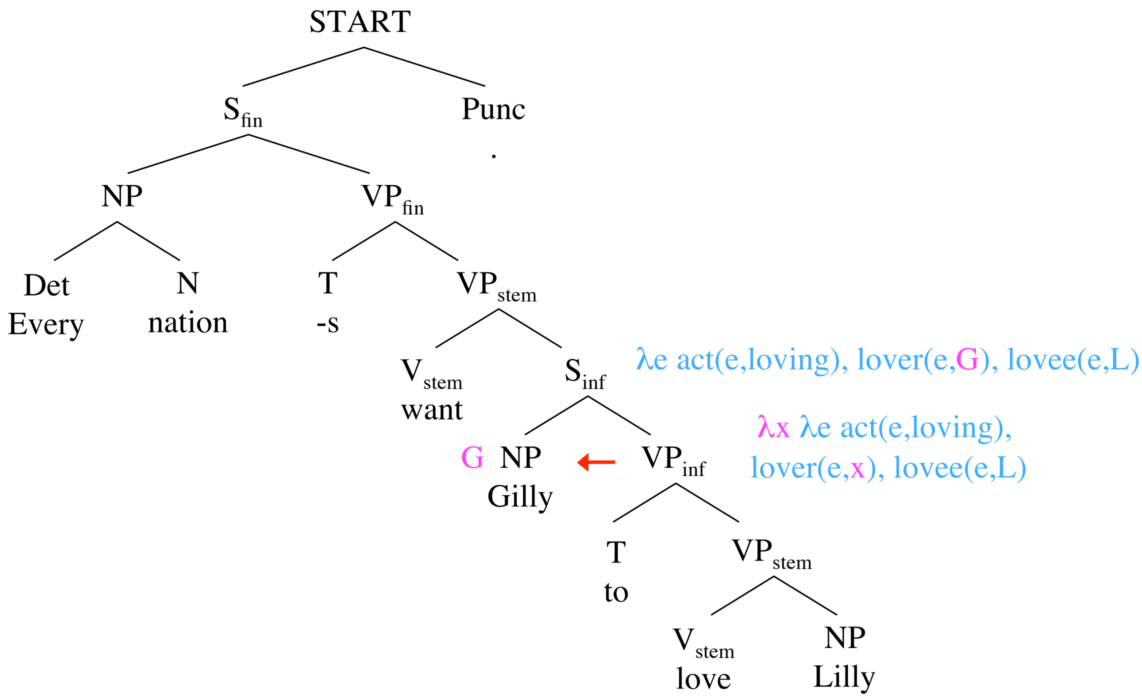


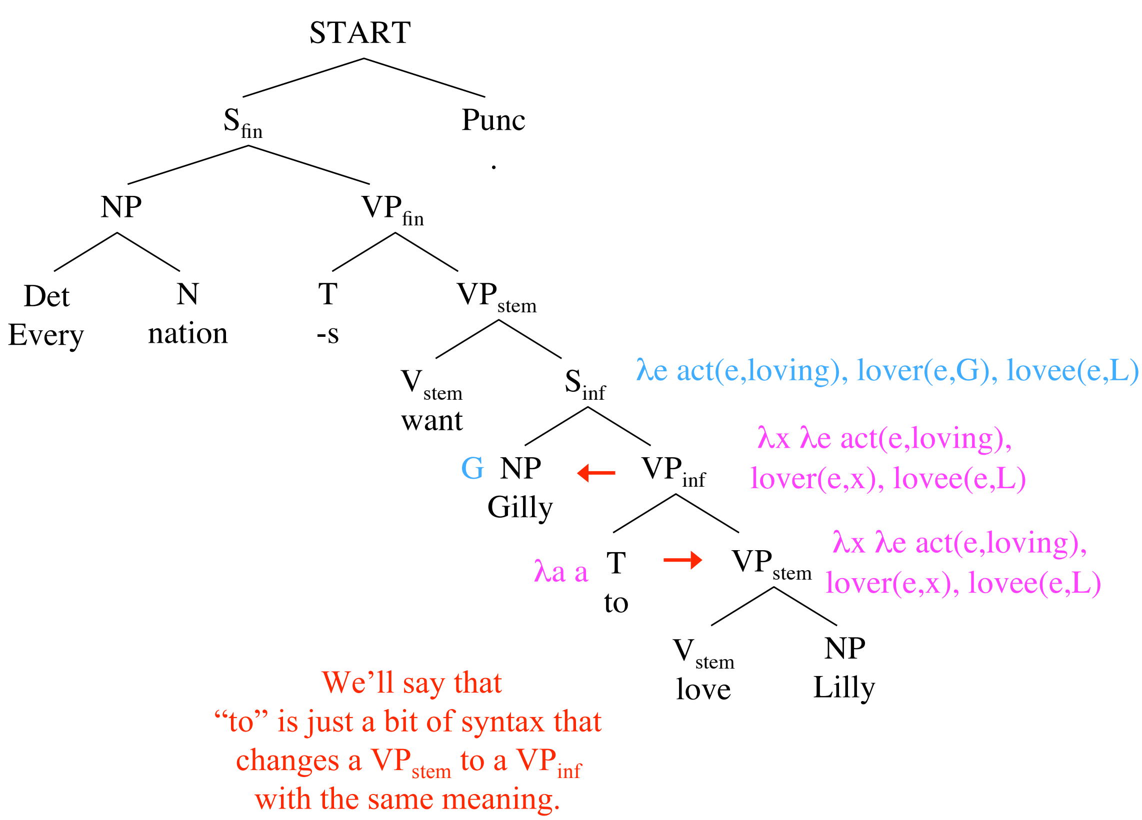


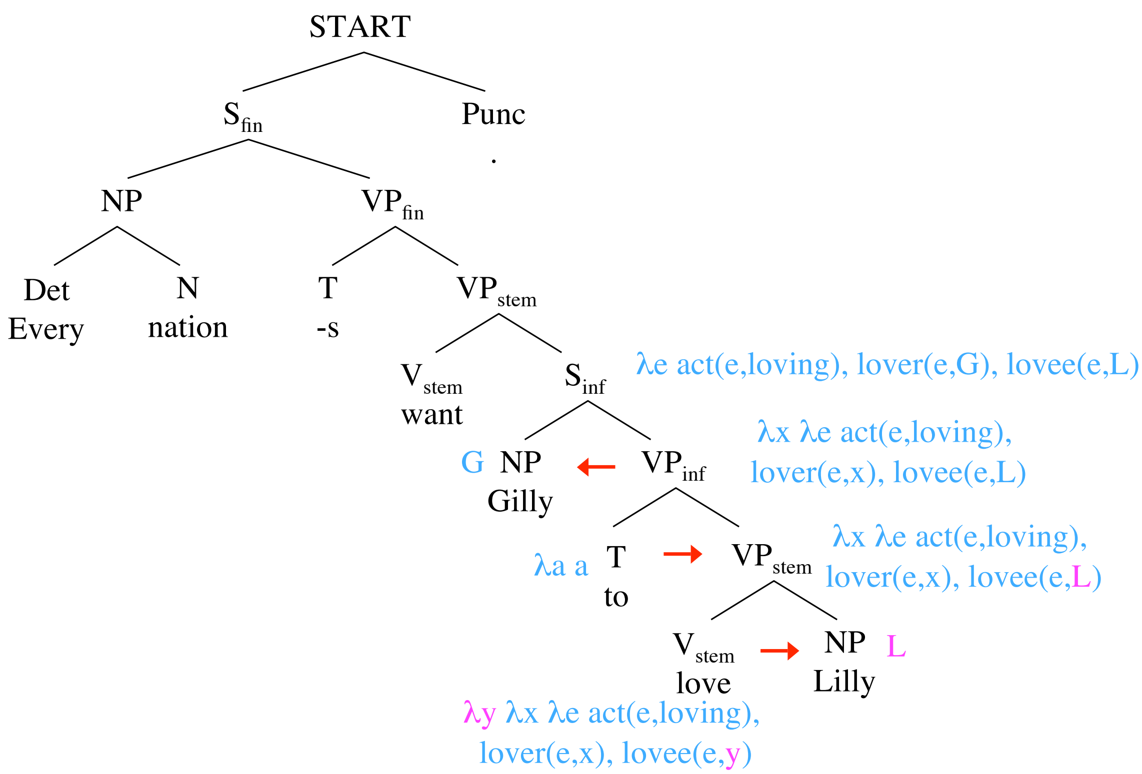
Now let's try a more complex example, and really handle tense.

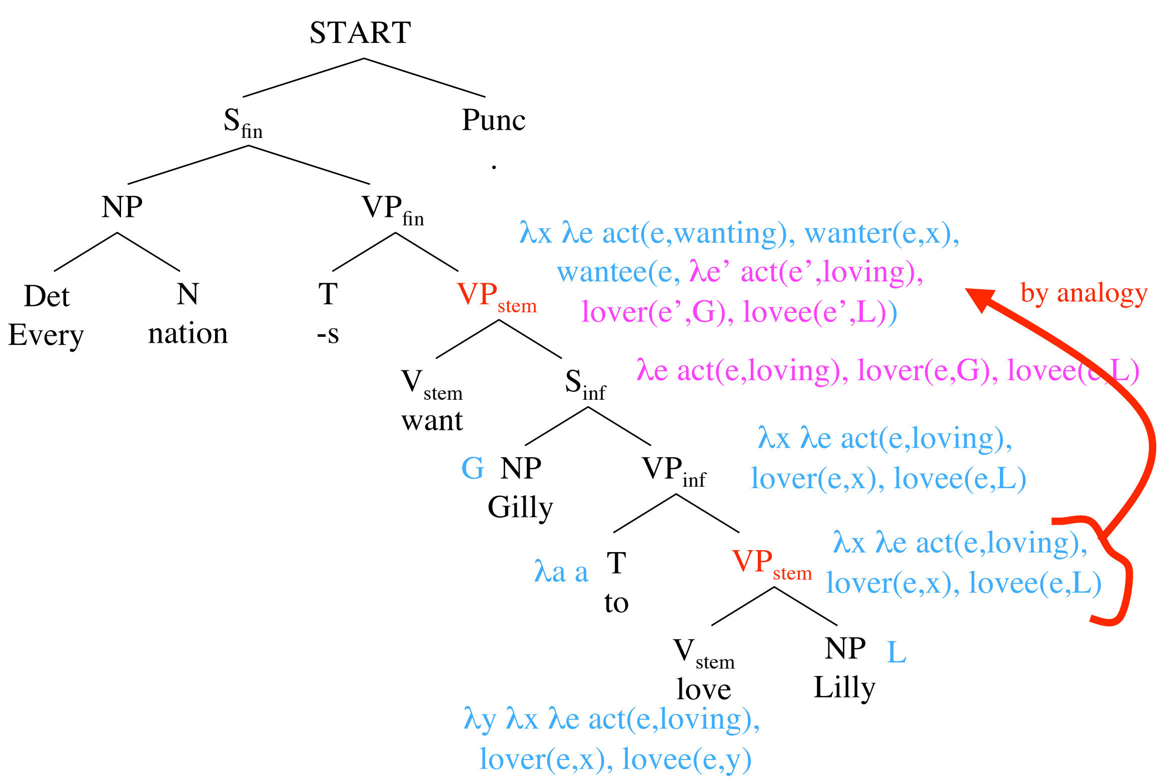

START
S_{fin}
Punc
.
NP
VP_{fin}
Det
Every
N
nation
T
-s
VP_{stem}
V_{stem}
want
S_{inf}
NP
Gilly
VP_{inf}
T
to
VP_{stem}
V_{stem}
love
NP
Lilly
λe act(e,loving), lover(e,G), lovee(e,L)
the meaning that we want here: how can we arrange to get it?

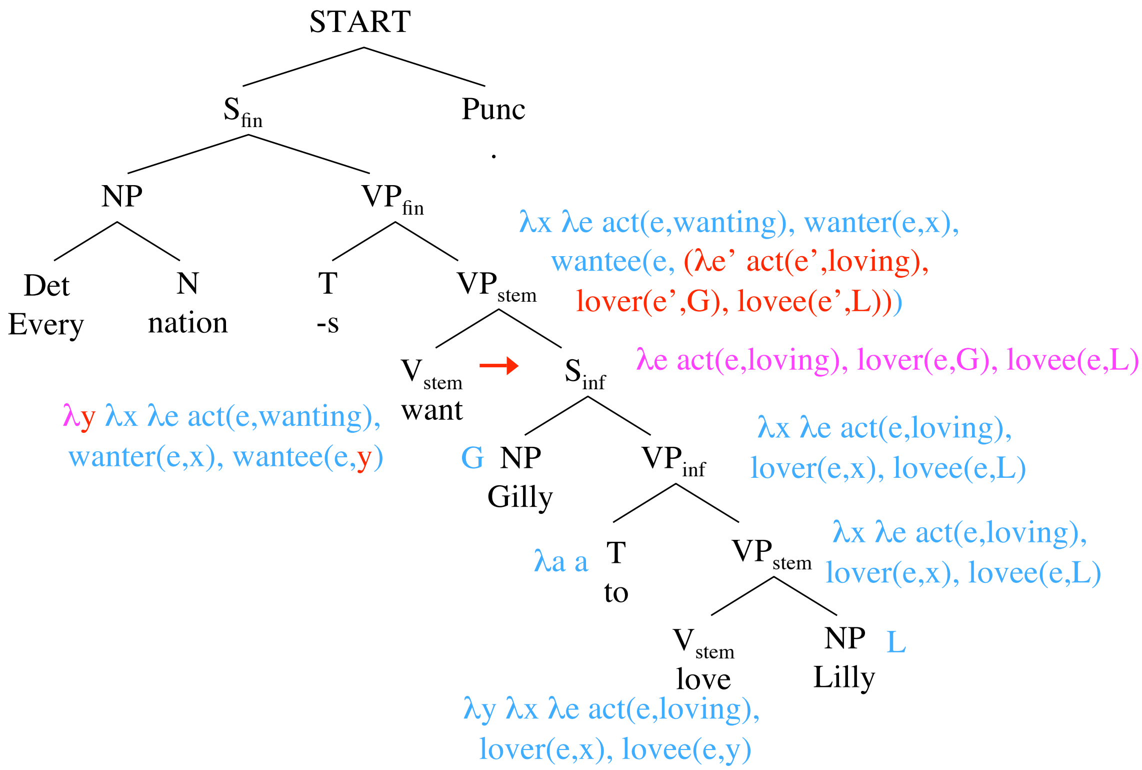

START
S_{fin}
Punc
.
NP
VP_{fin}
Det
Every
N
nation
T
-s
VP_{stem}
V_{stem}
want
S_{inf}
G
NP
Gilly
VP_{inf}
T
to
VP_{stem}
V_{stem}
love
NP
Lilly
λe act(e,loving), lover(e,G), lovee(e,L)
what function at VP_{inf} should apply to G to yield the result at S_{inf}?
(this is like division!)

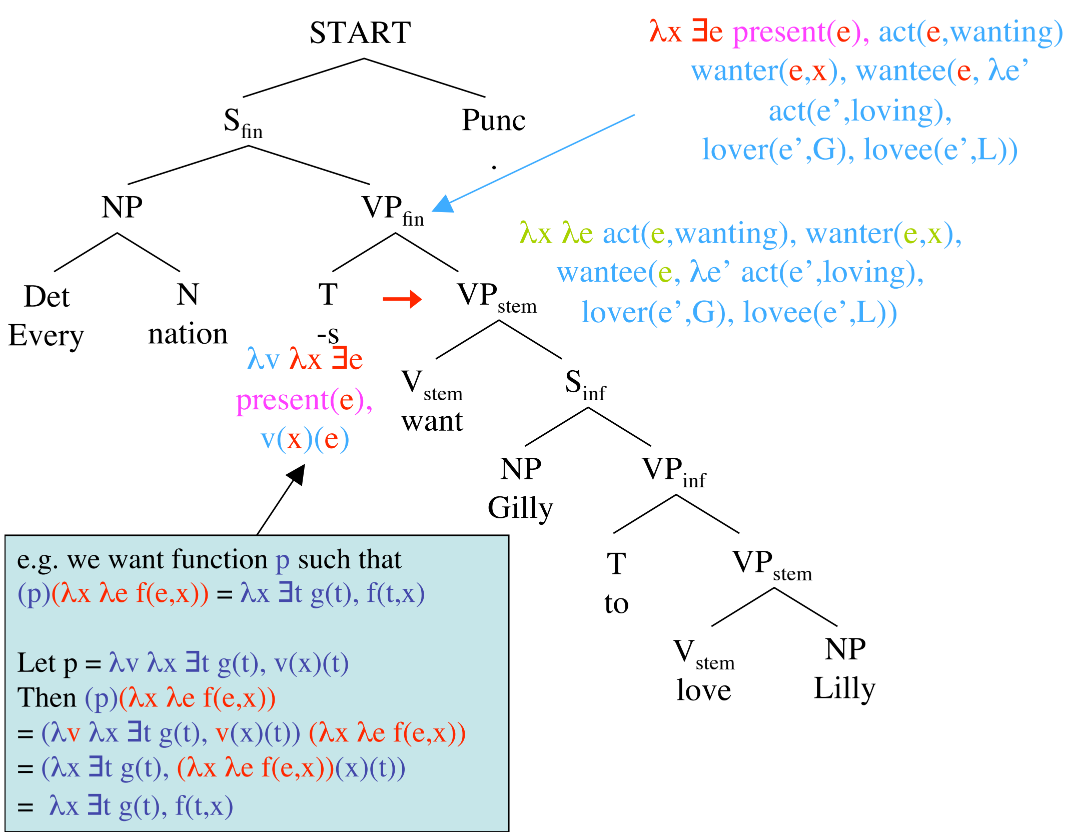

START
S fin
Punc
.
NP
VP fin
Det
Every
N
nation
T
-s
VP stem
V stem
want
S inf
λe act(e,loving), lover(e,G), lovee(e,L)
G NP
Gilly
VP inf
λx λe act(e,loving),
lover(e,x), lovee(e,L)
T
to
VP stem
V stem
love
NP
Lilly

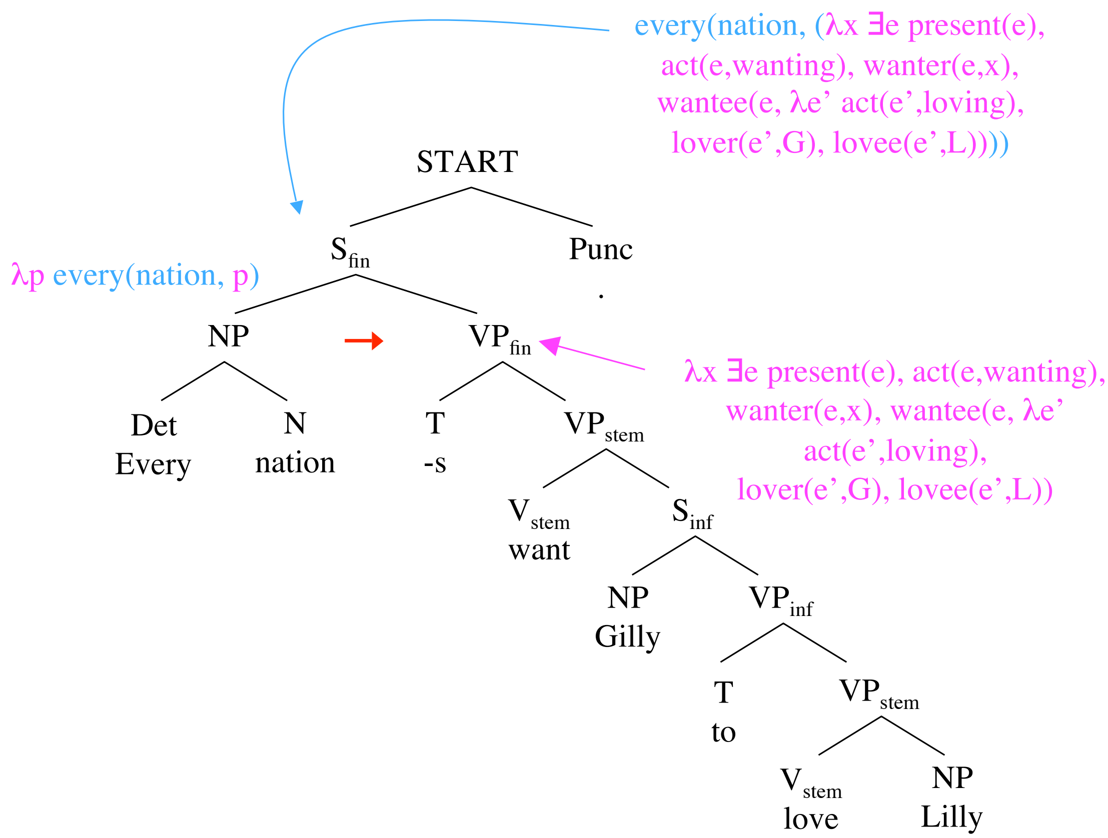

START
S fin
Punc
.
NP
VP fin
Det
Every
N
nation
T
-s
VP stem
V stem
want
S inf
λe act(e,loving), lover(e,G), lovee(e,L)
G NP
Gilly
VP inf
λx λe act(e,loving),
lover(e,x), lovee(e,L)
λa a T
to
VP stem
λx λe act(e,loving),
lover(e,x), lovee(e,L)
V stem
love
NP
Lilly
We'll say that
"to" is just a bit of syntax that
changes a VP stem to a VP inf
with the same meaning.

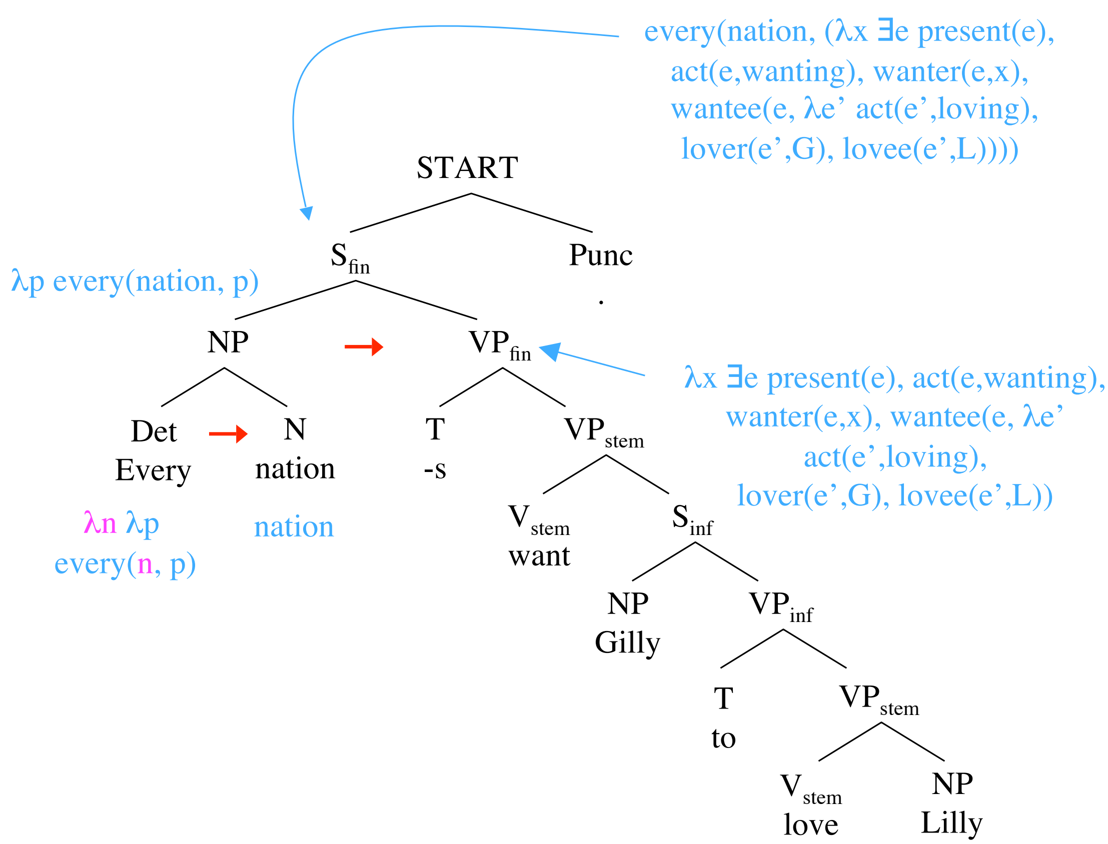

START
Sfin
Punc
.
NP
VPfin
Det
Every
N
nation
T
-s
VPstem
Vstem
want
Sinf
λe act(e,loving), lover(e,G), lovee(e,L)
G NP
Gilly
VPinf
λx λe act(e,loving),
lover(e,x), lovee(e,L)
λa a T
to
VPstem
λx λe act(e,loving),
lover(e,x), lovee(e,L)
Vstem
love
NP L
Lilly
λy λx λe act(e,loving),
lover(e,x), lovee(e,y)

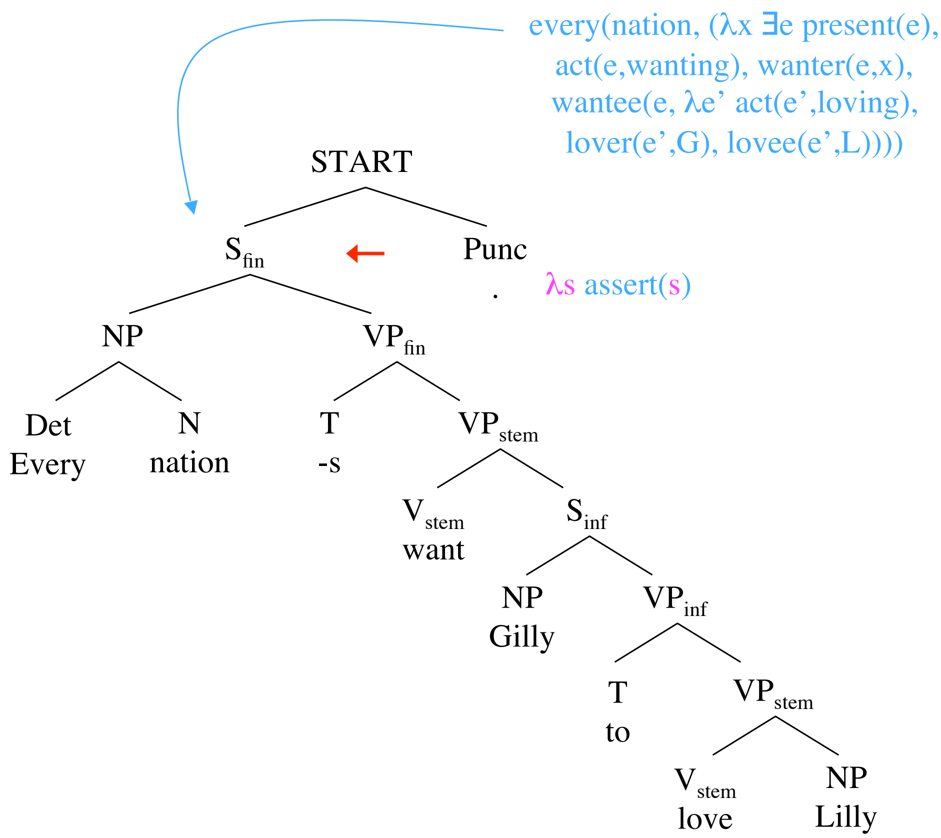

START
Sfin
Punc
.
NP
VPfin
Det
Every
N
nation
T
-s
VPstem
λx λe act(e,wanting), wanter(e,x),
wantee(e, λe' act(e',loving),
lover(e',G), lovee(e',L))
by analogy
Vstem
want
Sinf
λe act(e,loving), lover(e,G), lovee(e,L)
G NP
Gilly
VPinf
λx λe act(e,loving),
lover(e,x), lovee(e,L)
λa a T
to
VPstem
λx λe act(e,loving),
lover(e,x), lovee(e,L)
Vstem
love
NP L
Lilly
λy λx λe act(e,loving),
lover(e,x), lovee(e,y)


START
S fin
Punc
.
NP
VP fin
Det
Every
N
nation
T
-s
VP stem
λx λe act(e,wanting), wanter(e,x),
wantee(e, (λe' act(e',loving),
lover(e',G), lovee(e',L)))
V stem
want
S inf
λe act(e,loving), lover(e,G), lovee(e,L)
λy λx λe act(e,wanting),
wanter(e,x), wantee(e,y)
G NP
Gilly
VP inf
λx λe act(e,loving),
lover(e,x), lovee(e,L)
λa a T
to
VP stem
λx λe act(e,loving),
lover(e,x), lovee(e,L)
V stem
love
NP L
Lilly
λy λx λe act(e,loving),
lover(e,x), lovee(e,y)

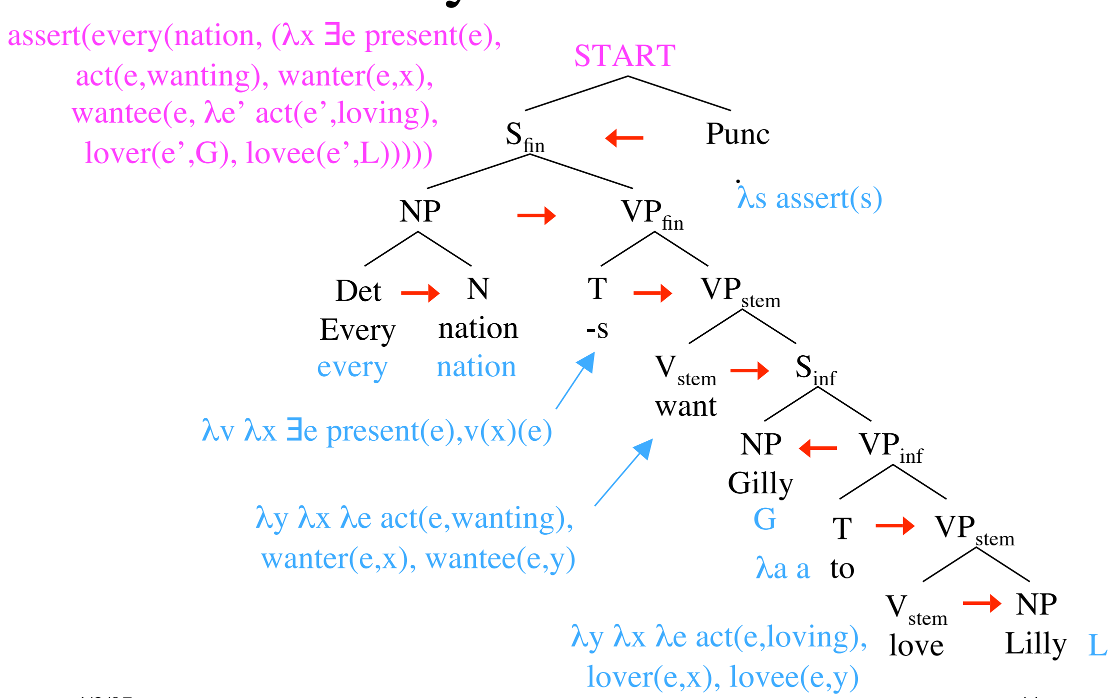

START
S fin
Punc
.
λx ∃e present(e), act(e,wanting),
wanter(e,x), wantee(e, λe'
act(e',loving),
lover(e',G), lovee(e',L))
NP
VP fin
λx λe act(e,wanting), wanter(e,x),
wantee(e, λe' act(e',loving),
lover(e',G), lovee(e',L))
Det
Every
N
nation
T
-s
VP stem
λv λx ∃e
present(e),
v(x)(e)
V stem
want
S inf
NP
Gilly
VP inf
T
to
VP stem
V stem
love
NP
Lilly
e.g. we want function p such that
(p)(λx λe f(e,x)) = λx ∃t g(t), f(t,x)
Let p = λv λx ∃t g(t), v(x)(t)
Then (p)(λx λe f(e,x))
= (λv λx ∃t g(t), v(x)(t)) (λx λe f(e,x))
= (λx ∃t g(t), (λx λe f(e,x))(x)(t))
= λx ∃t g(t), f(t,x)


every(nation, (λx ∃e present(e), act(e,wanting), wanter(e,x), wantee(e, λe' act(e',loving), lover(e',G), lovee(e',L))))
START
S fin
Punc
.
λp every(nation, p)
NP
VP fin
λx ∃e present(e), act(e,wanting), wanter(e,x), wantee(e, λe' act(e',loving), lover(e',G), lovee(e',L))
Det
Every
N
nation
T
-s
VP stem
V stem
want
S inf
NP
Gilly
VP inf
T
to
VP stem
V stem
love
NP
Lilly


every(nation, (λx ∃e present(e), act(e,wanting), wanter(e,x), wantee(e, λe' act(e',loving), lover(e',G), lovee(e',L))))
START
S fin
Punc
.
λp every(nation, p)
NP
VP fin
λx ∃e present(e), act(e,wanting), wanter(e,x), wantee(e, λe' act(e',loving), lover(e',G), lovee(e',L))
Det
Every
λn λp every(n, p)
N
nation
nation
T
-s
VP stem
V stem
want
S inf
NP
Gilly
VP inf
T
to
VP stem
V stem
love
NP
Lilly



In Summary: From the Words



Intensional Arguments

- `Willy wants a unicorn`
  - ∃e act(e,wanting), wanter(e,Willy), exists(unicorn, λu wantee(e,u))
    - "there is a unicorn u that Willy wants"
    - here the wantee is an individual entity
  - ∃e act(e,wanting), wanter(e,Willy), wantee(e, λu unicorn(u))
    - "Willy wants any entity u that satisfies the unicorn predicate"
    - here the wantee is a <u>type</u> of entity
- `Willy wants Lilly to get married`
  - ∃e present(e), act(e,wanting), wanter(e,Willy),
    wantee(e, λe' [act(e',marriage), marrier(e',Lilly)])
  - "Willy wants any event e' in which Lilly gets married"
  - Here the wantee is a <u>type</u> of event
  - Sentence doesn't claim that such an event exists
- Intensional verbs besides `want`: `hope`, `doubt`, `believe`,…

Intensional Arguments

- `Willy wants a unicorn`
  - ∃e act(e,wanting), wanter(e,Willy), wantee(e, λg unicorn(g))
    - "Willy wants anything that satisfies the unicorn predicate"
    - here the wantee is a type of entity
- Problem (a fine point I'll gloss over):
  - λg unicorn(g) is defined by the actual <u>set</u> of unicorns ("<u>extension</u>")
  - But this set is empty: λg unicorn(g) = λg FALSE = λg dodo(g)
  - Then `wants a unicorn` = `wants a dodo`. Oops!
  - So really the wantee should be <u>criteria</u> for unicornness ("<u>intension</u>")
- Traditional solution involves "possible-world semantics"
  - Can imagine other worlds where set of unicorn ≠ set of dodos
  - Other worlds also useful for: `You must pay the rent`
    `You can pay the rent`
    `If you hadn't, you'd be homeless`

Control

- `Willy wants Lilly to get married`
  - ∃e present(e), act(e,wanting), wanter(e,Willy), wantee(e, λf [act(f,marriage), marrier(f,Lilly)])

- `Willy wants to get married`
  - Same as `Willy wants Willy to get married`
  - Just as easy to represent as `Willy wants Lilly …`
  - The only trick is to construct the representation from the syntax. The empty subject position of "to get married" is said to be controlled by the subject of "wants."

Other Fun Semantic Stuff

- Temporal logic
  - `Gilly had swallowed eight goldfish before Milly reached the bowl`
  - `Billy said Jilly was pregnant` (sequence of tense)
  - `Billy said, "Jilly is pregnant."`
- Generics (not quite the same as plurals)
  - `Typhoons arise in the Pacific`
  - `Children must be carried`
- Presuppositions
  - `The king of France is bald.` (if there is no such king is this true?)
  - `Have you stopped beating your wife?` (what is presupposed?)
- Pronoun-Quantifier Interaction ("bound anaphora")
  - `Every farmer who owns a donkey beats it.`
  - `If you have a dime, put it in the meter.`
  - `The woman who every Englishman loves is his mother.`
  - `I love my mother and so does Billy.`