Semantics

Natural Language Processing
CS 6120—Spring 2013
Northeastern University

David Smith
some slides from
Jason Eisner, Dan Klein, Stephen Clark & Eva Banik

Language as Structure

- So far, we've talked about **structure**
- What structures are **more probable**?
  - Language modeling: Good sequences of words/ characters
  - Text classification: Good sequences in defined contexts
- How can we recover **hidden structure**?
  - Tagging: hidden word classes
  - Parsing: hidden word relations

What Does It All Mean?

- Studying phonology, morphology, syntax, etc. independent of meaning is methodologically very useful
- We can study the structure of languages we don't understand
- We can use HMMs and CFGs to study protein structure and music, which don't bear meaning in the same way as language

What Does It All Mean?

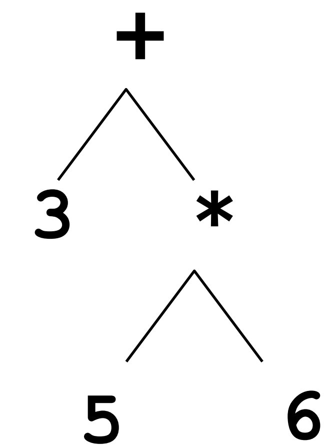
- How would you know if a computer “understood” the “meaning” of an (English) utterance (even in some weak “scare-quoted” way)?
- How would you know if a **person** understood the meaning of an utterance?

What Does It All Mean?

- Paraphrase, "state in your own words" (English to English translation)
- Translation into another language
- Reading comprehension questions
- Drawing appropriate inferences
- Carrying out appropriate actions
- Open-ended dialogue (Turing test)

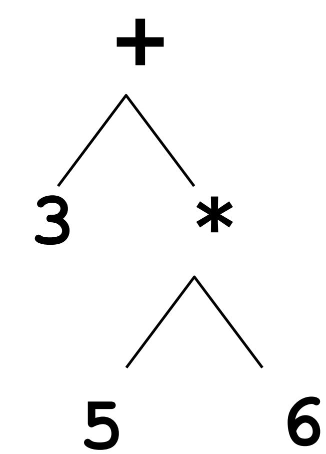
Programming Language Interpreter

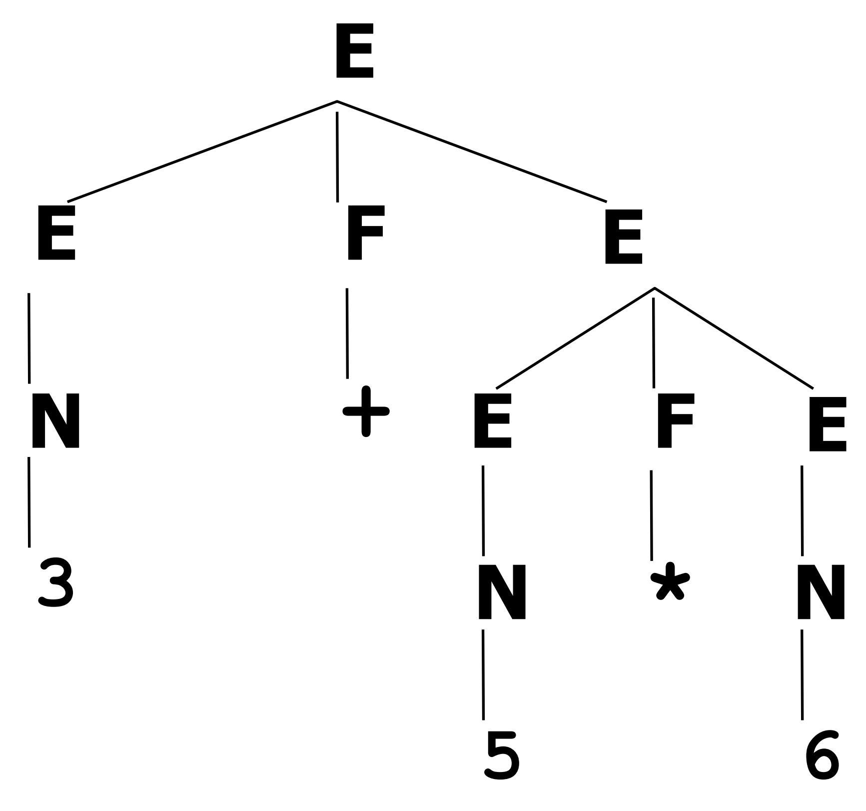
- What is meaning of `3+5*6`?
- First parse it into `3+(5*6)`



Programming Language Interpreter

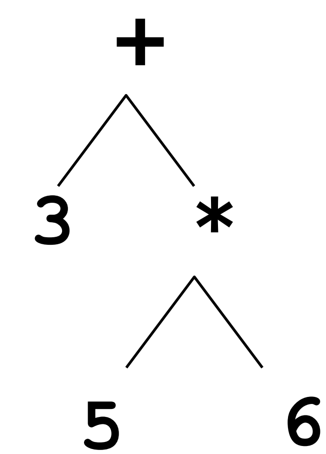
- What is meaning of `3+5*6`?
- First parse it into `3+(5*6)`

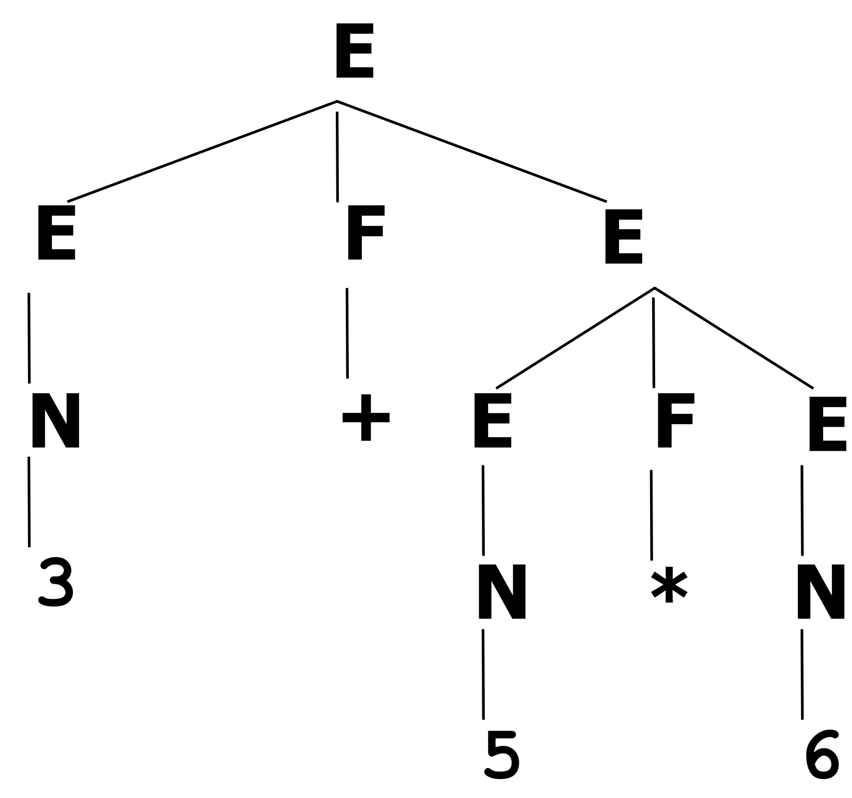




Programming Language Interpreter

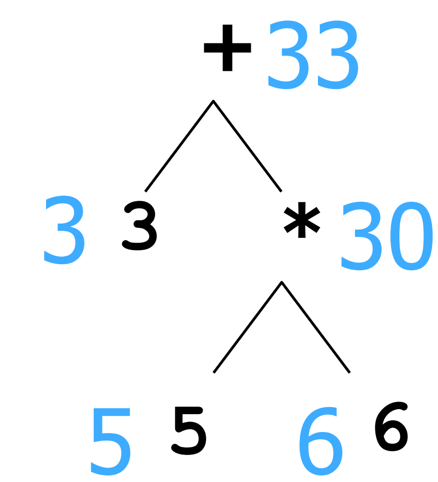
- What is meaning of `3+5*6`?
- First parse it into `3+(5*6)`
- Now give a meaning to each node in the tree (bottom-up)

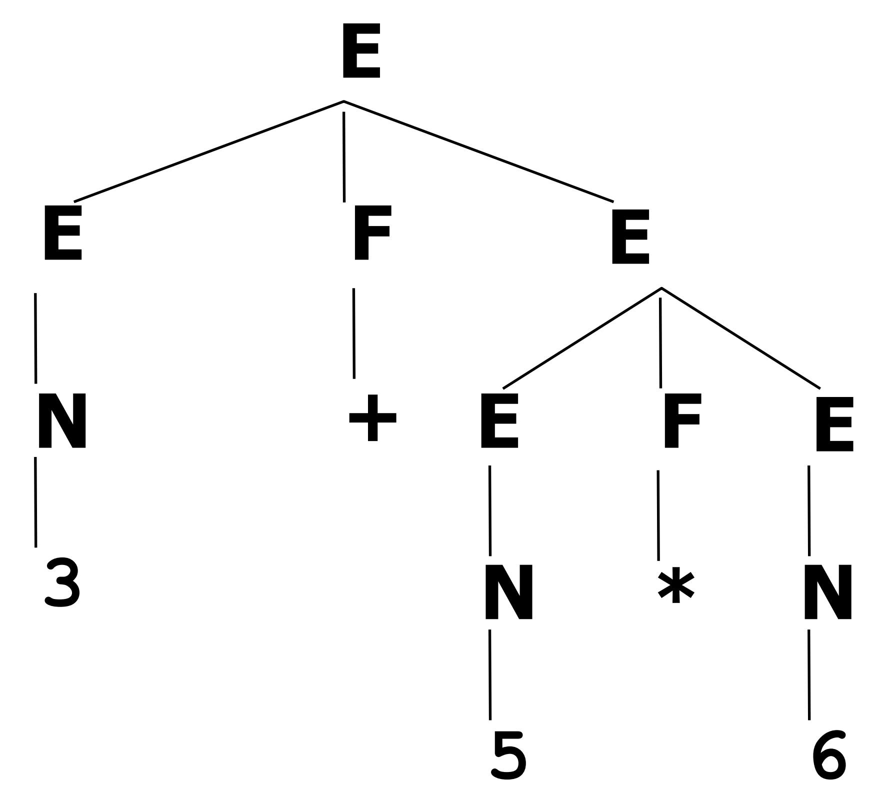




Programming Language Interpreter

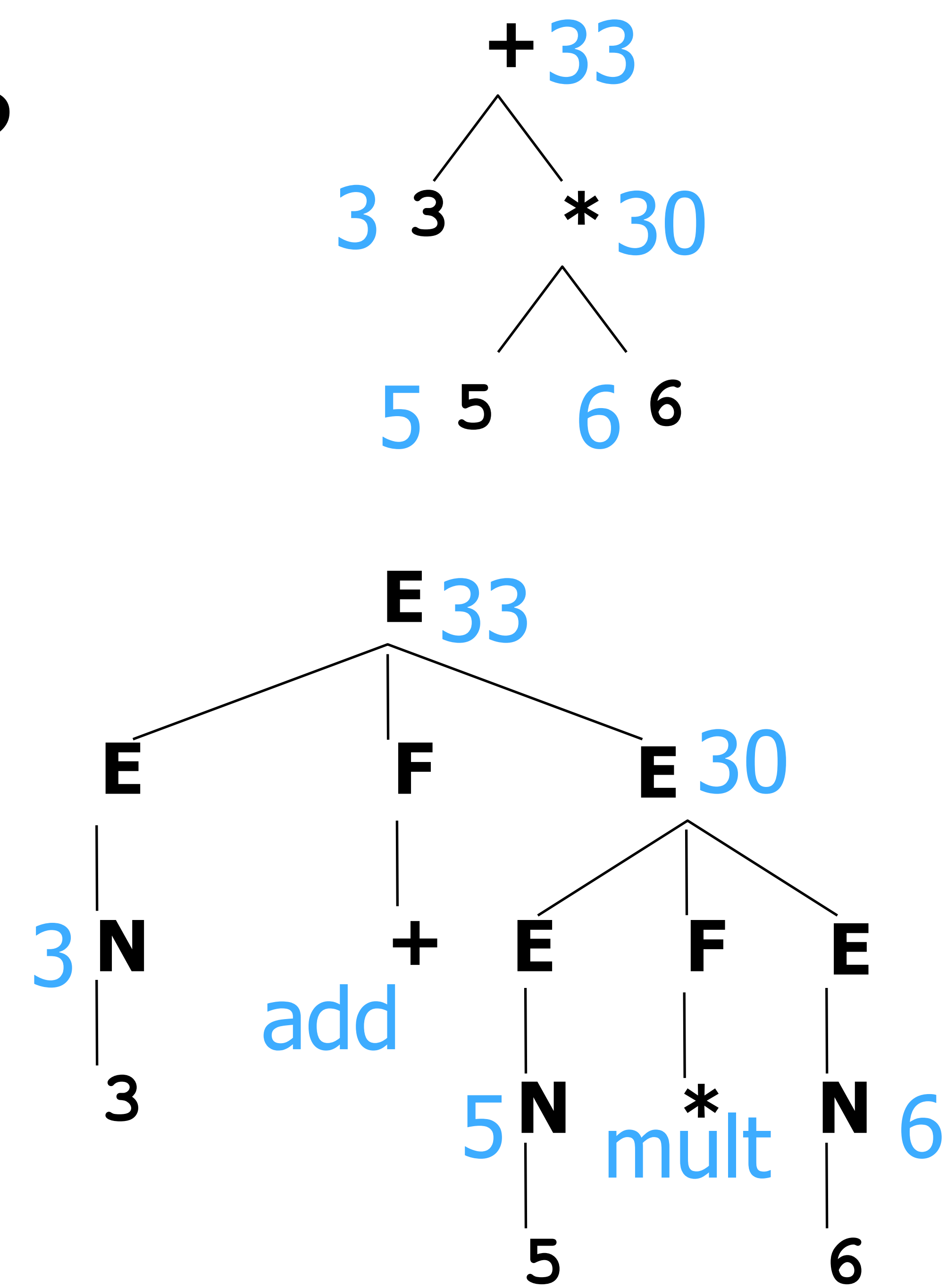
- What is meaning of 3+5*6?
- First parse it into 3+(5*6)
- Now give a meaning to each node in the tree (bottom-up)



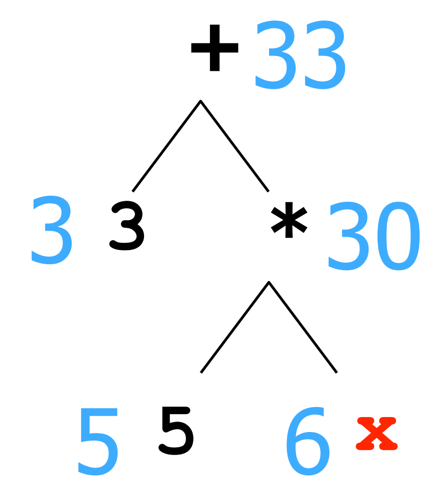


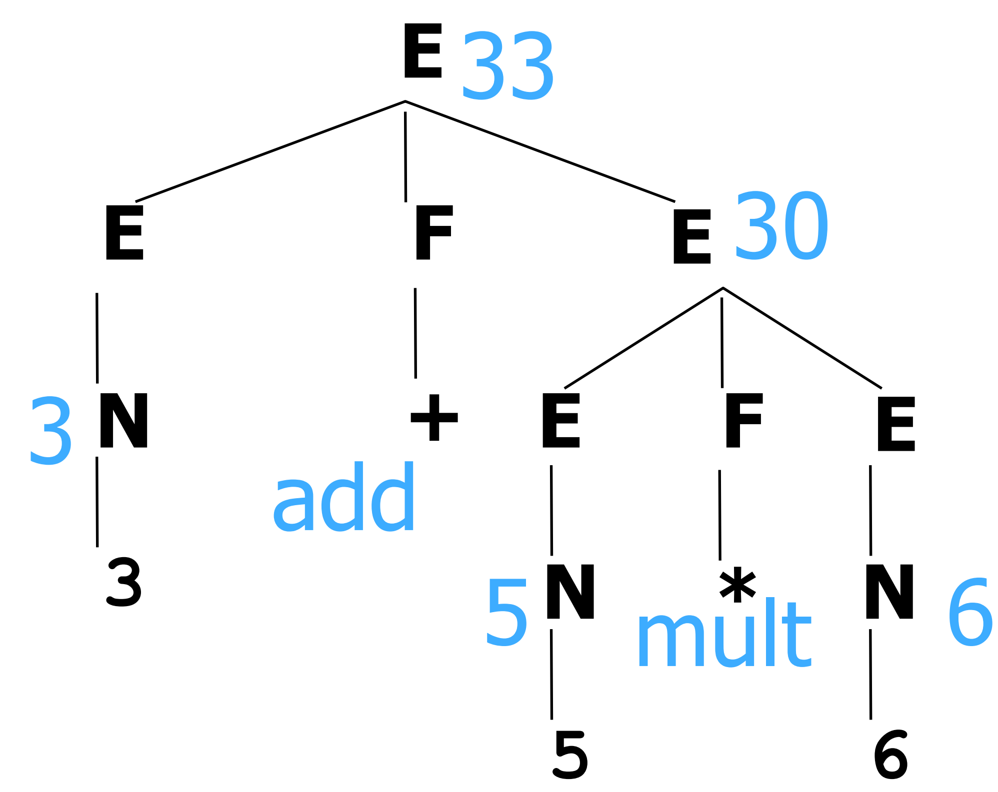
Programming Language Interpreter

- What is meaning of `3+5*6`?
- First parse it into `3+(5*6)`
- Now give a meaning to each node in the tree (bottom-up)



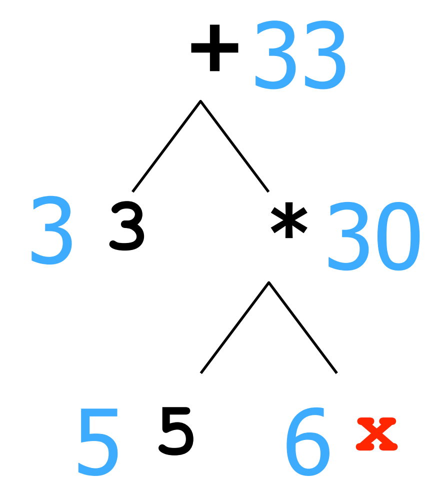
Interpreting in an Environment

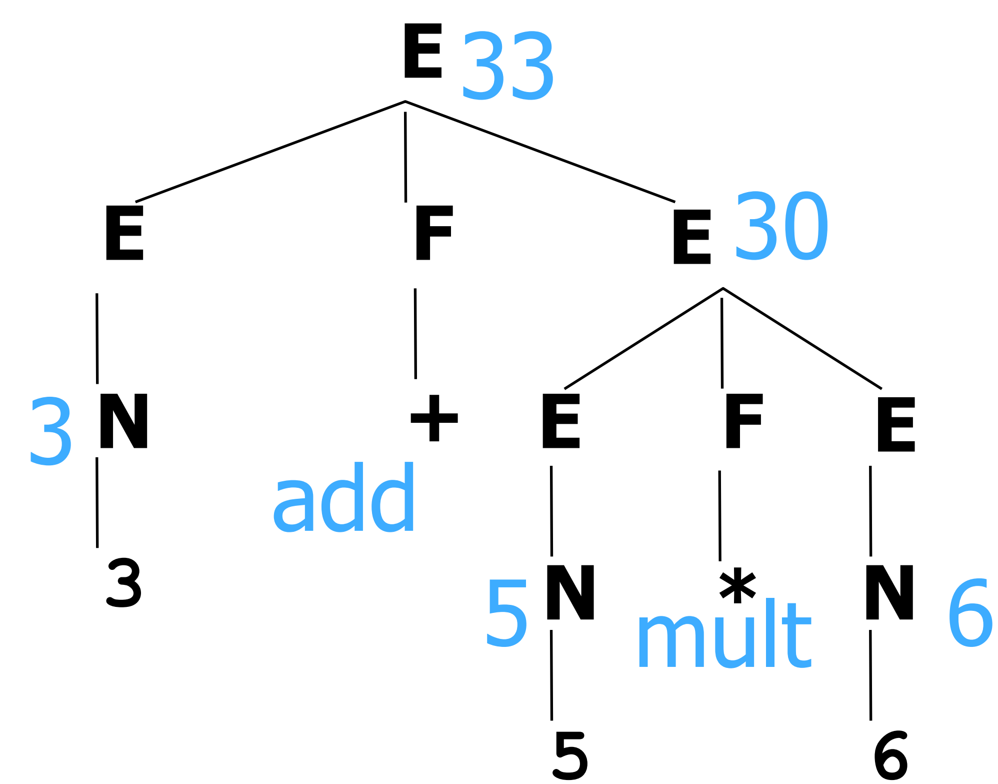




Interpreting in an Environment

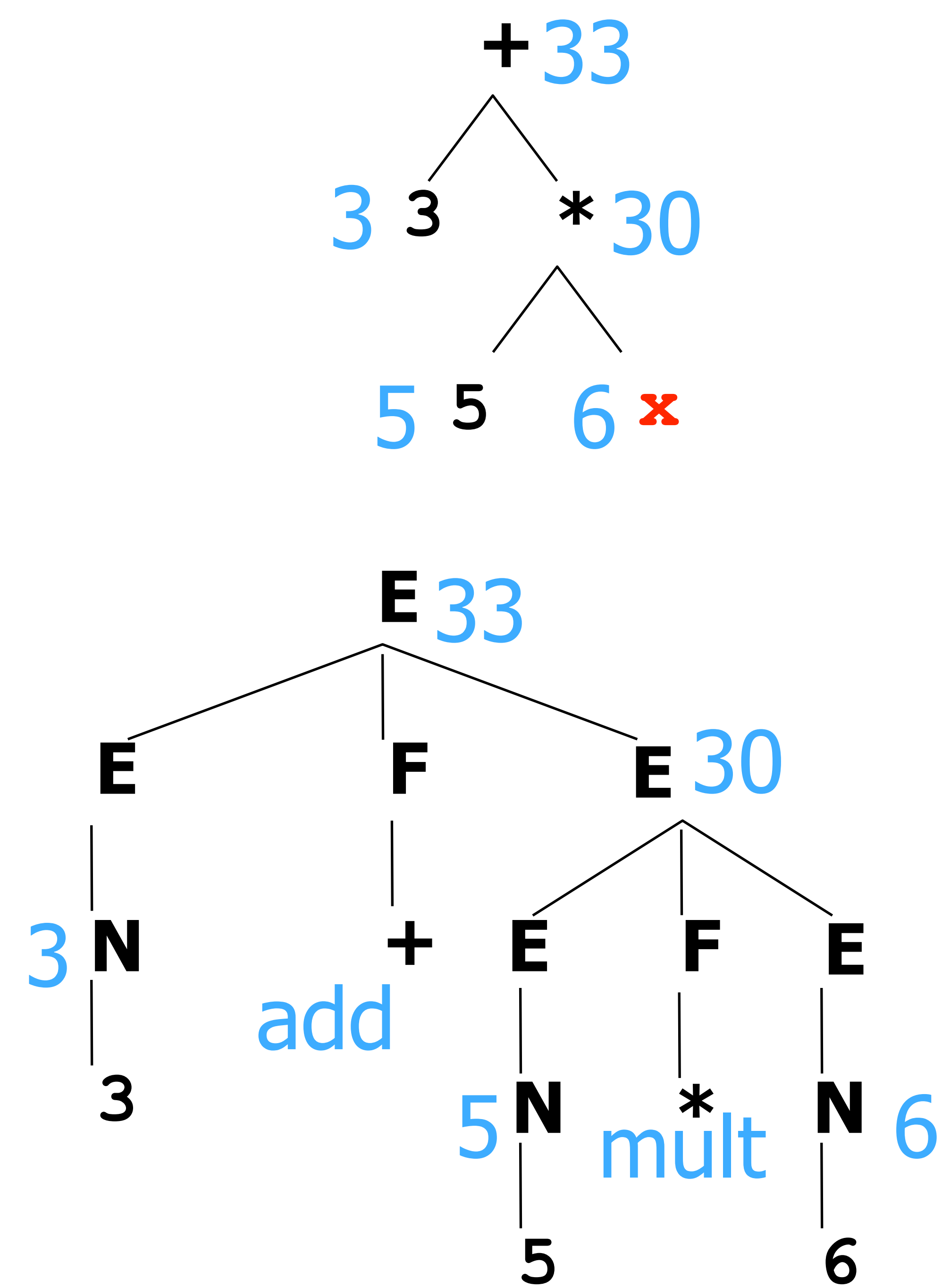
- How about `3+5*x`?





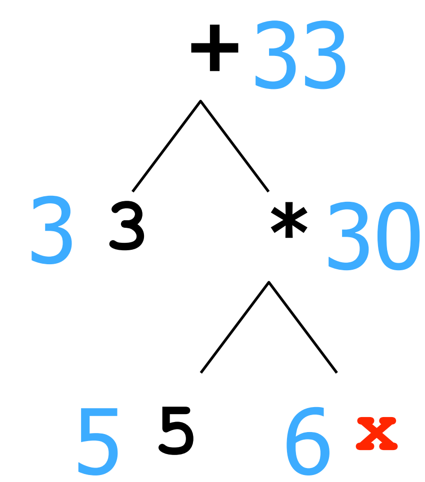
Interpreting in an Environment

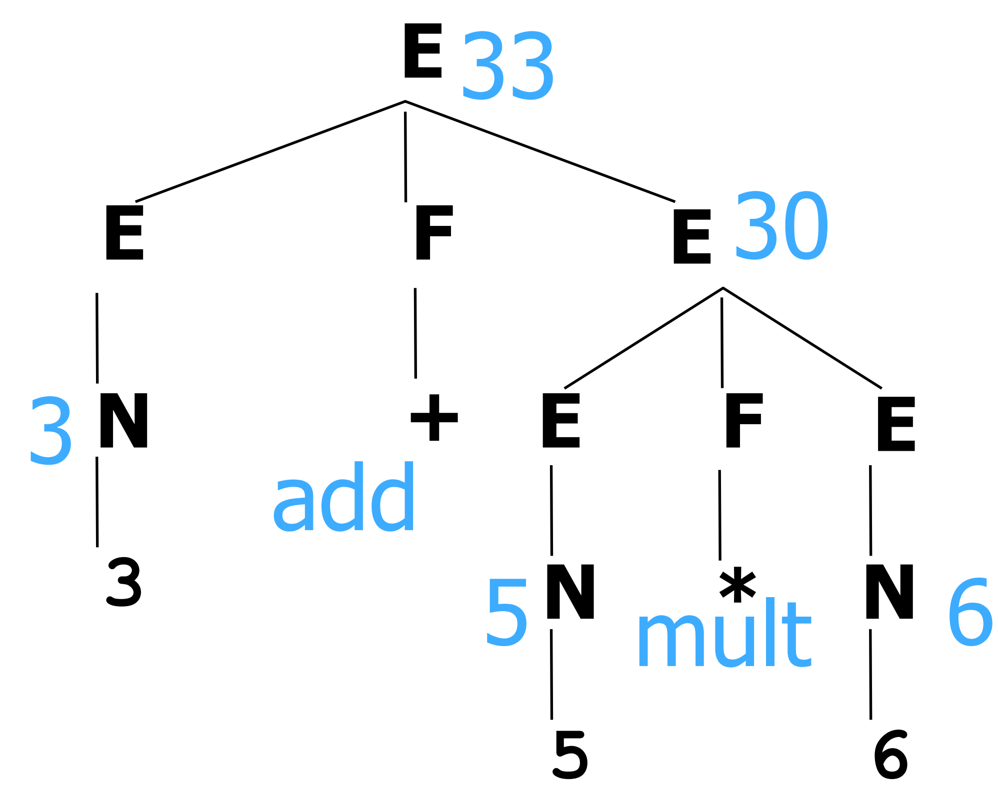
- How about `3+5*x`?
- Same thing: the meaning of `x` is found from the environment (it's 6)



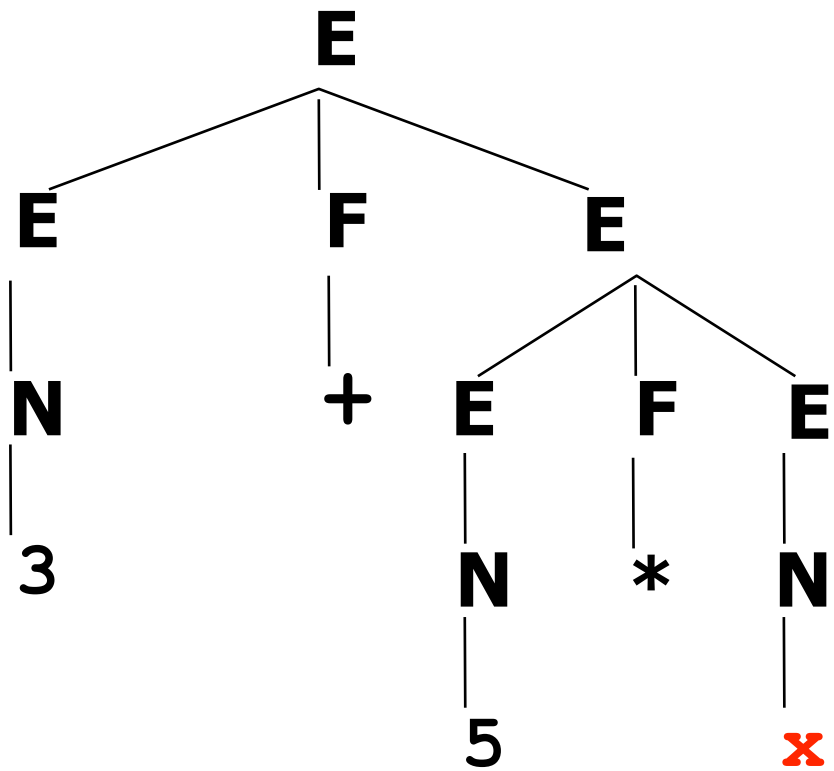
Interpreting in an Environment

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



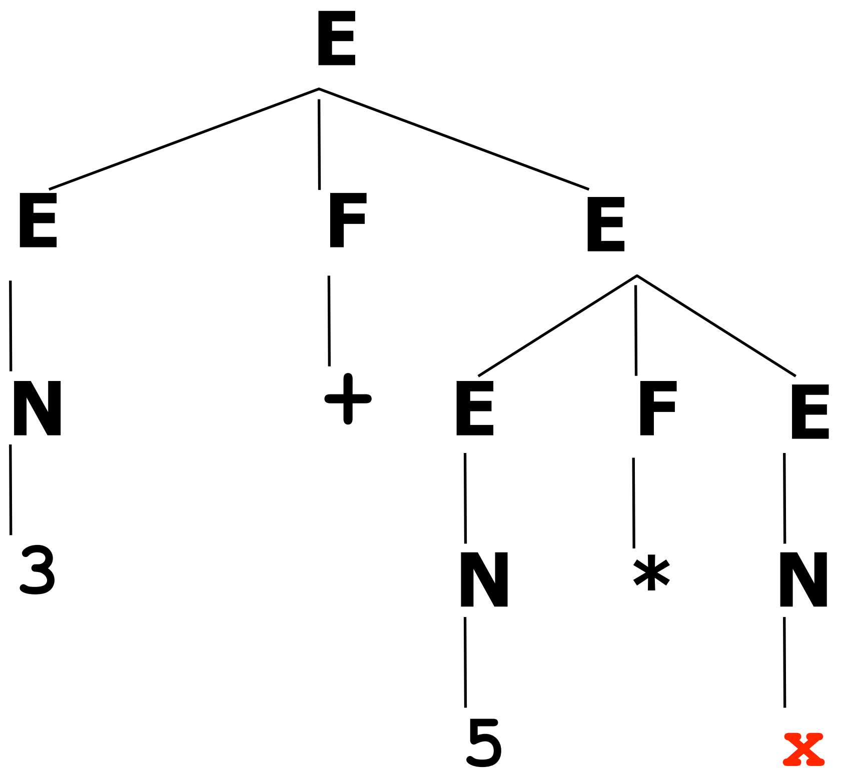


Compiling



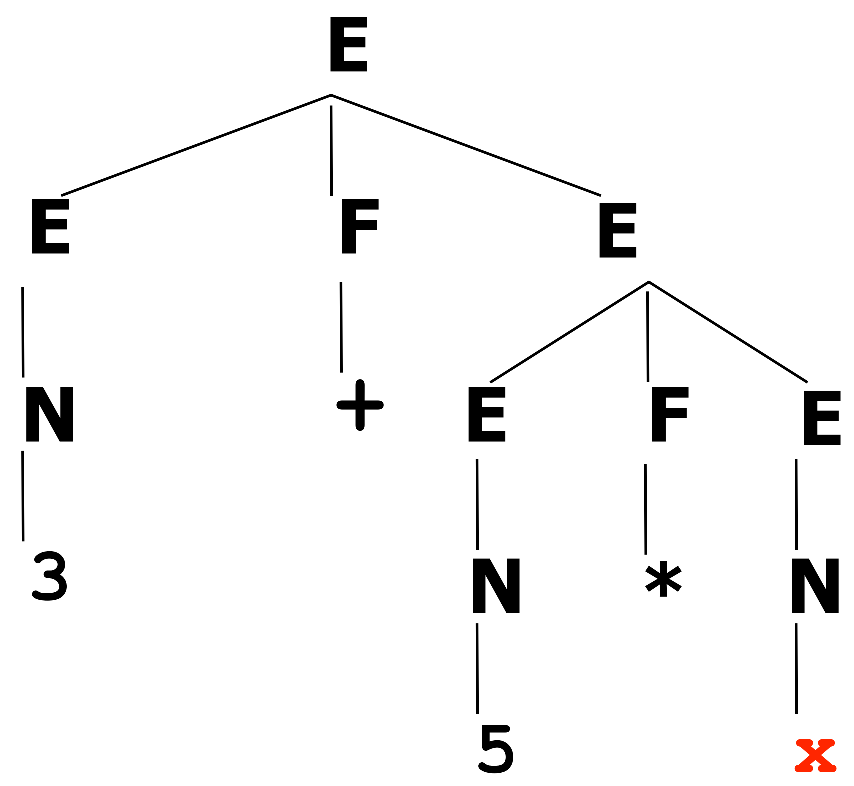
Compiling

- How about `3+5*x`?



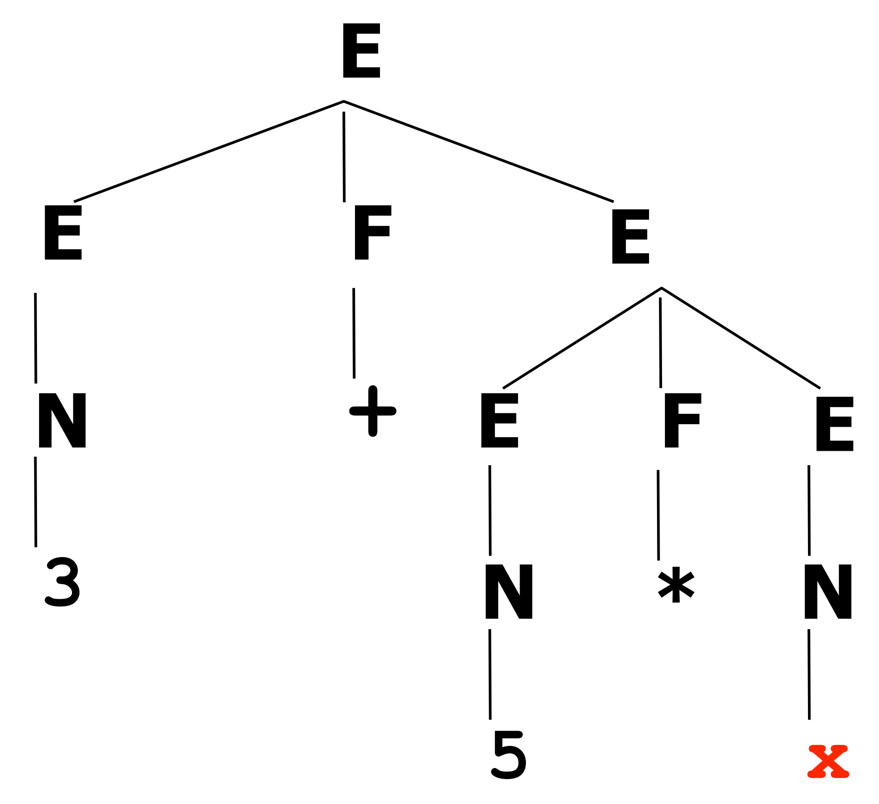
Compiling

- How about `3+5*x`?
- Don't know `x` at compile time



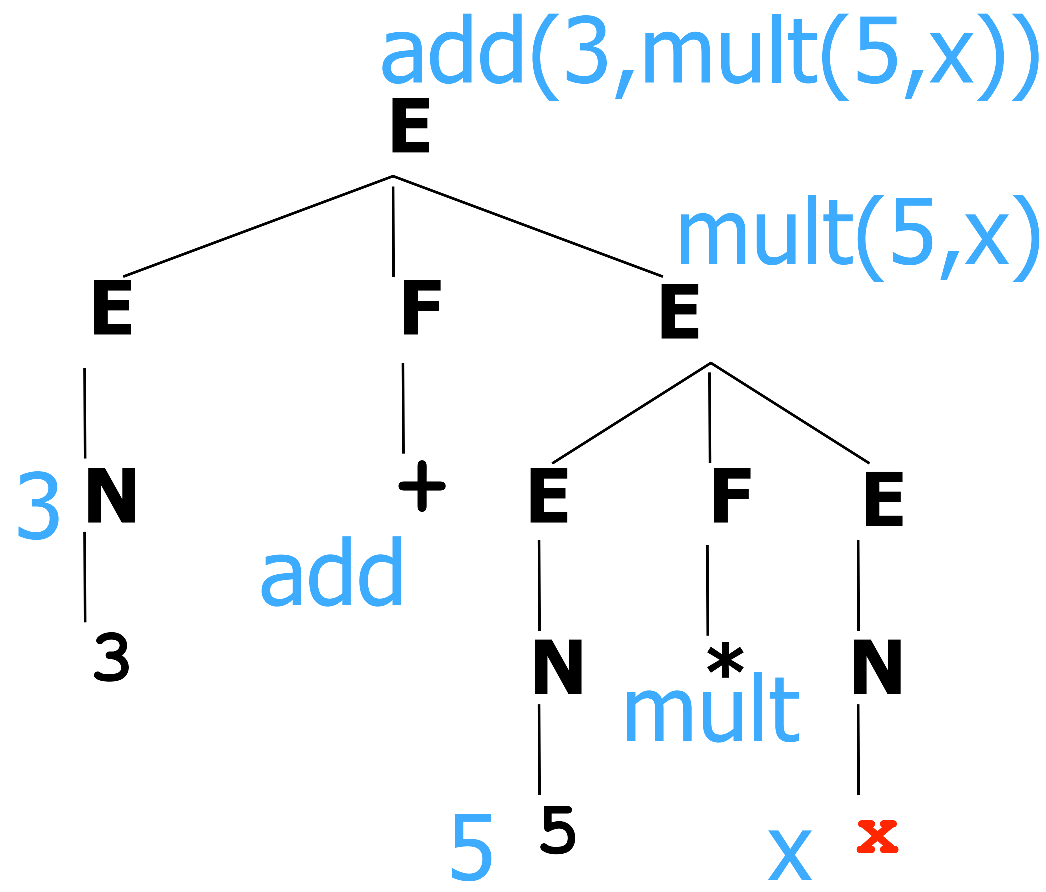
Compiling

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



Compiling

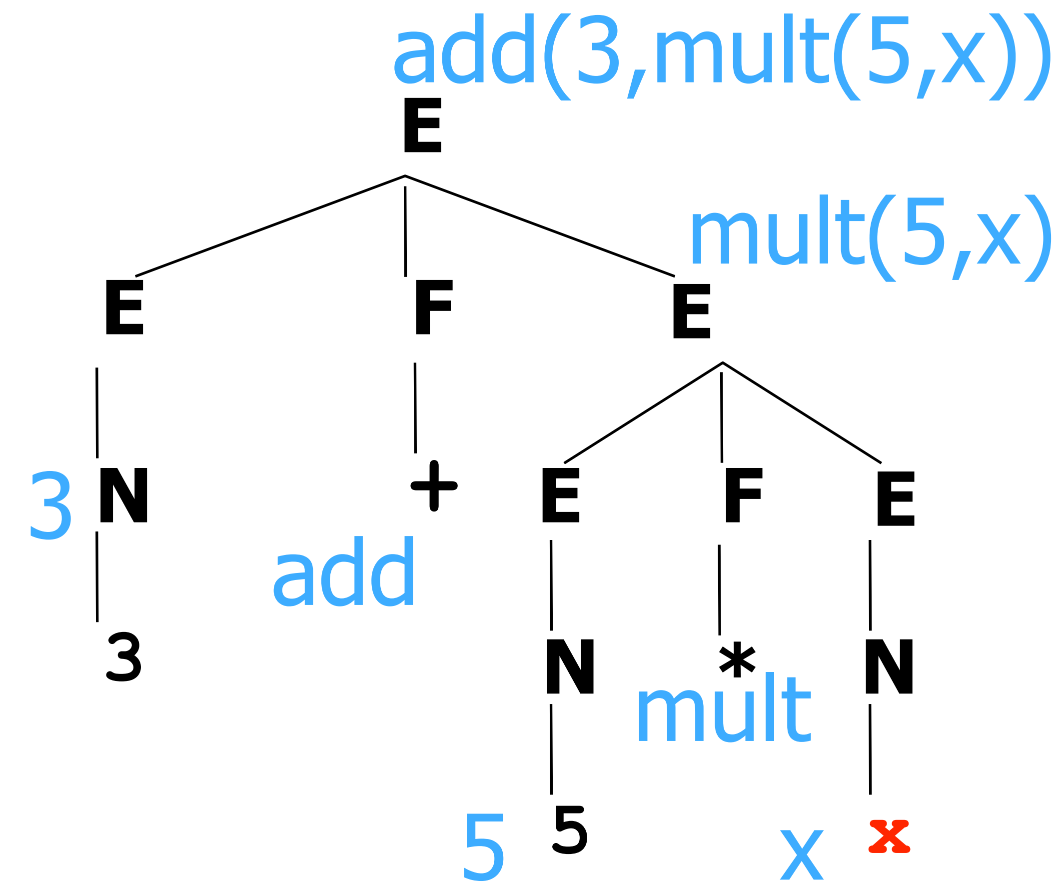
- How about `3+5*x`?
- Don't know `x` at compile time
- "Meaning" at a node is a piece of code, not a number



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

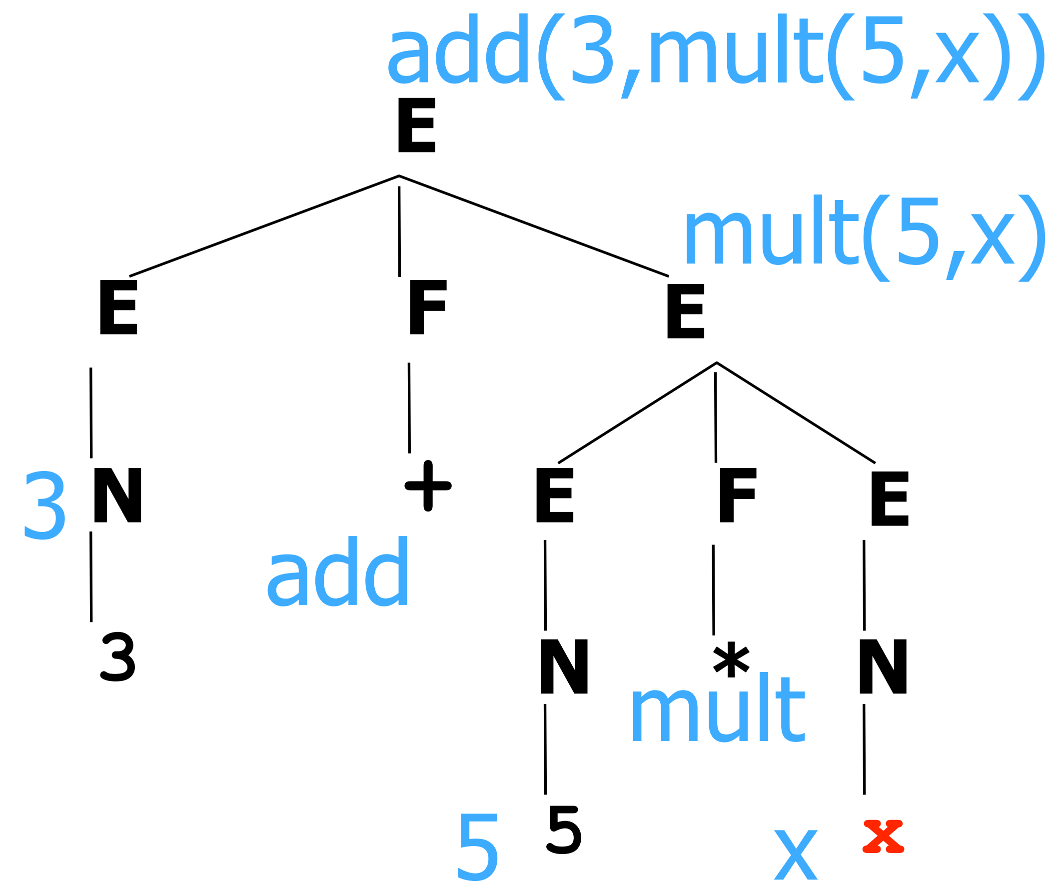
`5*(x+1)-2` is a different expression that produces equivalent code



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)

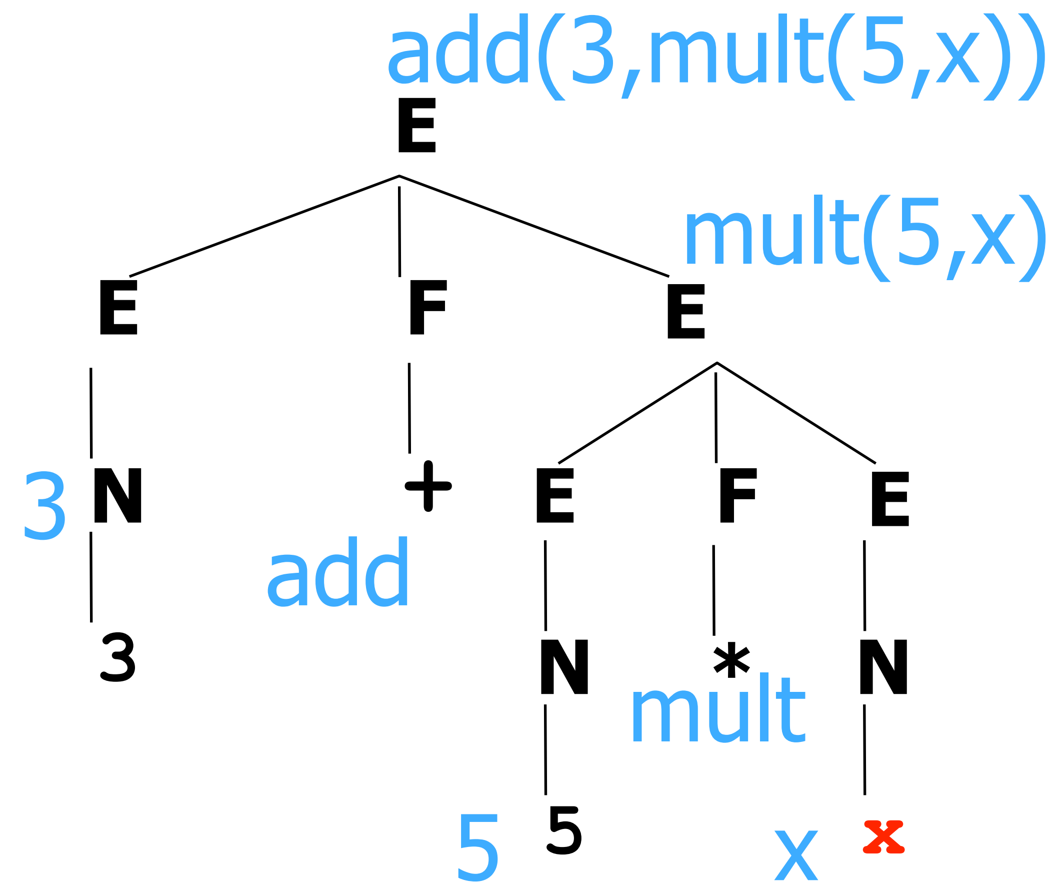


Compiling

- 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 Counts as Understanding?

some notions

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
  - SQL 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

What Counts as Understanding? some notions

- We understand statement if we know how one could (in principle) 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?
  - Match statements with a "domain theory"
  - Philosophers tend to use this definition

What Counts as Understanding?
some notions

- We understand statement if we know how one could (in principle) 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?
  - Match statements with a "domain theory"
  - 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

What Does It All Mean?

- Paraphrase, "state in your own words" (English to English translation)
- Translation into another language
- Reading comprehension questions
- Drawing appropriate inferences
- Carrying out appropriate actions
- Open-ended dialogue (Turing test)
- Translation to **logical form** that we can reason about

(First Order) Logic
Some Preliminaries

(First Order) Logic
Some Preliminaries

Three major kinds of objects

(First Order) Logic
Some Preliminaries

Three major kinds of objects

1. Booleans
   - Roughly, the semantic values of sentences

(First Order) Logic
Some Preliminaries

Three major kinds of objects

1. Booleans
   - Roughly, the semantic values of sentences
2. Entities
   - Values of NPs, e.g., objects like this slide
   - Maybe also other types of entities, like times

(First Order) Logic
Some Preliminaries

Three major kinds of objects

1. Booleans
   - Roughly, the semantic values of sentences
2. Entities
   - Values of NPs, e.g., objects like this slide
   - Maybe also other types of entities, like times
3. Functions of various types
   - Functions from booleans to booleans (and, or, not)
   - A function from entity to boolean is called a "predicate" – e.g., frog(x), green(x)
   - Functions might return other functions!

(First Order) Logic
Some Preliminaries

Three major kinds of objects

1. Booleans
   - Roughly, the semantic values of sentences
2. Entities
   - Values of NPs, e.g., objects like this slide
   - Maybe also other types of entities, like times
3. Functions of various types
   - Functions from booleans to booleans (and, or, not)
   - A function from entity to 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

Logic: Lambda Terms

- Lambda terms:

Logic: Lambda Terms

- Lambda terms:
  - Let square = λp p*p

Logic: Lambda Terms

- Lambda terms:
  - Let square = λp p*p
  - Then square(3) = (λp p*p)(3) = 3*3

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.

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 λ**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)?)

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 λ**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)?)

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 λ**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 <u>predicate:</u> returns true/false

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 λ**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 <u>predicate:</u> returns true/false
  - even(x) is true if x is even

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 λ**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 <u>predicate:</u> 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)

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 λ**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 <u>predicate:</u> 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

Logic: Multiple Arguments

- All lambda terms have one argument

Logic: Multiple Arguments

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

Logic: Multiple Arguments

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

Logic: Multiple Arguments

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

- Suppose we want to write times(5,6)

Logic: Multiple Arguments

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

- Suppose we want to write times(5,6)
- Suppose times is <u>defined</u> as λx λy (x*y)

Logic: Multiple Arguments

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

- Suppose we want to write times(5,6)
- Suppose times is <u>defined</u> as λx λy (x*y)
- Claim that times(5)(6) is 30
  - times(5) = (λx λy x*y) (5) = λy 5*y

Logic: Multiple Arguments

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

- Suppose we want to write times(5,6)
- Suppose times is defined as λx λy (x*y)
- Claim that times(5)(6) is 30
  - times(5) = (λx λy x*y) (5) = λy 5*y
    - If this function weren't anonymous, what would we call it?

Logic: Multiple Arguments

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

- Suppose we want to write times(5,6)
- Suppose times is <u>defined</u> as λx λy (x*y)
- Claim that times(5)(6) is 30
  - times(5) = (λx λy x*y) (5) = λy 5*y
    - If this function weren't anonymous, what would we call it?
  - times(5)(6) = (λy 5*y)(6) = 5*6 = 30

Logic: Multiple Arguments

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

- If we write times(5,6), it's just syntactic sugar for times(5)(6) or perhaps times(6)(5) [notation varies]
  - times(5,6) = times(5)(6)
    = (λx λy x*y) (5)(6) = (λy 5*y)(6) = 5*6 = 30

Logic: Multiple Arguments

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

- If we write times(5,6), it's just syntactic sugar for times(5)(6) or perhaps times(6)(5) [notation varies]
  - times(5,6) = times(5)(6)
    = (λx λy x*y) (5)(6) = (λy 5*y)(6) = 5*6 = 30

- So we can always get away with 1-arg functions ...

Logic: Multiple Arguments

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

- If we write times(5,6), it's just syntactic sugar for times(5)(6) or perhaps times(6)(5) [notation varies]
  - times(5,6) = times(5)(6)
    = (λx λy x*y) (5)(6) = (λy 5*y)(6) = 5*6 = 30

- So we can always get away with 1-arg functions ...
  - ... which might return a function to take the next argument. Whoa.

Logic: Multiple Arguments

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

- If we write times(5,6), it's just syntactic sugar for times(5)(6) or perhaps times(6)(5) [notation varies]
  - times(5,6) = times(5)(6)
    = (λx λy x*y) (5)(6) = (λy 5*y)(6) = 5*6 = 30

- So we can always get away with 1-arg functions ...
  - ... which might return a function to take the next argument. Whoa.
- Remember: square can be written as λx square(x)

Logic: Multiple Arguments

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

- If we write times(5,6), it's just syntactic sugar for times(5)(6) or perhaps times(6)(5) [notation varies]
  - times(5,6) = times(5)(6)
    = (λx λy x*y) (5)(6) = (λy 5*y)(6) = 5*6 = 30

- So we can always get away with 1-arg functions ...
  - ... which might return a function to take the next argument. Whoa.
- Remember: square can be written as λx square(x)
  - And now times can be written as λx λy times(x,y)

Grounding out

Grounding out

- So what does times actually mean???

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 * actually denotes the multiplication function!

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 * actually denotes the multiplication function!

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 * actually denotes the multiplication function!

- Well, maybe * was defined as another lambda term, so substitute to get *(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

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 * actually denotes the multiplication function!

- Well, maybe * was defined as another lambda term, so substitute to get *(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 *(5,6) just executes a multiplication function

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 * actually denotes the multiplication function!
- Well, maybe * was defined as another lambda term, so substitute to get *(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 *(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., *):
  - GeorgeWBush - 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(GeorgeWBush, LauraBush)
    - Question: What does loves(LauraBush) denote?
- Constants used to define meanings of words
- Meanings of phrases will be built from the constants

Logic: Interesting 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

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 $\forall x$ pig(x) $\Rightarrow$ big(x))
  - exists(pig,big) (equivalent to $\exists x$ 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(GeorgeWBush, LauraBush)
  - swallowed(Gilly, Jilly)
- … or is it analogous?

A reasonable representation?

A reasonable representation?

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

A reasonable representation?

- `Gilly swallowed a goldfish`
  - First attempt: swallowed(Gilly, goldfish)
- But we're not paying attention to `a`!

A reasonable representation?

- `Gilly swallowed a goldfish` (with "a" underlined)
  - 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

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

A reasonable representation?

- `Gilly swallowed a goldfish` (the "a" is underlined)
  - 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

Use a Quantifier

- `Gilly swallowed a goldfish` (the "a" is underlined)
  - First attempt: swallowed(Gilly, goldfish)

Use a Quantifier

- `Gilly swallowed a goldfish`
  - First attempt: swallowed(Gilly, goldfish)
- Better: ∃g goldfish(g) AND swallowed(Gilly, g)

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"

Use a Quantifier

- `Gilly swallowed a goldfish` (with "a" underlined)
  - 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

Tense

- `Gilly swallowed a goldfish`

Tense

- `Gilly swallowed a goldfish`
  - Previous attempt: exists(goldfish, λg swallowed(Gilly,g))

Tense

- `Gilly swallowed a goldfish`
  - Previous attempt: exists(goldfish, λg swallowed(Gilly,g))
- Improve to use tense:

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

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))

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

Event Properties

- `Gilly swallowed a goldfish`
  - Previous: ∃t past(t) AND exists(goldfish, swallow(t,Gilly))

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 …

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)), …

- Does this mean what we'd expect??

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)

- Does this mean what we'd expect??

Quantifier Order

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

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

- Does this mean what we'd expect??

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?

Quantifier Order

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

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

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??

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

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))

Quantifier Order

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

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)])

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"

Intensional Arguments

Intensional Arguments

- `Willy wants a unicorn`

Intensional Arguments

- Willy wants a unicorn
  - ∃e act(e,wanting), wanter(e,Willy), exists(unicorn, λu wantee(e,u))
    - "there is a particular unicorn u that Willy wants"
    - In this reading, the wantee is an individual entity

Intensional Arguments

- `Willy wants a unicorn`
  - ∃e act(e,wanting), wanter(e,Willy), exists(unicorn, λu wantee(e,u))
    - "there is a particular unicorn u that Willy wants"
    - In this reading, 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"
    - In this reading, the wantee is a type of entity
    - Sentence doesn't claim that such an entity exists

Intensional Arguments

- `Willy wants a unicorn`
  - ∃e act(e,wanting), wanter(e,Willy), exists(unicorn, λu wantee(e,u))
    - "there is a particular unicorn u that Willy wants"
    - In this reading, 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"
    - In this reading, the wantee is a <u>type</u> of entity
    - Sentence doesn't claim that such an entity exists
- `Willy wants Lilly to get married`

Intensional Arguments

- Willy wants a unicorn
  - ∃e act(e,wanting), wanter(e,Willy), exists(unicorn, λu wantee(e,u))
    - "there is a particular unicorn u that Willy wants"
    - In this reading, 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"
    - In this reading, the wantee is a type of entity
    - Sentence doesn't claim that such an entity exists
- Willy wants Lilly to get married
  - ∃e present(e), act(e,wanting), wanter(e,Willy), wantee(e, λe' [act(e',marriage), marrier(e',Lilly)])

Intensional Arguments

- `Willy wants a unicorn`
  - ∃e act(e,wanting), wanter(e,Willy), exists(unicorn, λu wantee(e,u))
    - "there is a particular unicorn u that Willy wants"
    - In this reading, 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"
    - In this reading, the wantee is a type of entity
    - Sentence doesn't claim that such an entity exists
- `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"

Intensional Arguments

- `Willy wants a unicorn`
  - ∃e act(e,wanting), wanter(e,Willy), exists(unicorn, λu wantee(e,u))
    - "there is a particular unicorn u that Willy wants"
    - In this reading, 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"
    - In this reading, the wantee is a <u>type</u> of entity
    - Sentence doesn't claim that such an entity exists
- `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

Intensional Arguments

- `Willy wants a unicorn`
  - ∃e act(e,wanting), wanter(e,Willy), exists(unicorn, λu wantee(e,u))
    - "there is a particular unicorn u that Willy wants"
    - In this reading, 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"
    - In this reading, the wantee is a type of entity
    - Sentence doesn't claim that such an entity exists
- `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 type of event
  - Sentence doesn't claim that such an event exists

Intensional Arguments

- `Willy wants a unicorn`
  - ∃e act(e,wanting), wanter(e,Willy), exists(unicorn, λu wantee(e,u))
    - "there is a particular unicorn u that Willy wants"
    - In this reading, 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"
    - In this reading, the wantee is a <u>type</u> of entity
    - Sentence doesn't claim that such an entity exists
- `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

Intensional Arguments

- `Willy wants a unicorn`

Intensional Arguments

- Willy wants a unicorn
  - ∃e act(e,wanting), wanter(e,Willy), wantee(e, λu unicorn(u))
    - "Willy wants anything that satisfies the unicorn predicate"
    - here the wantee is a type of entity

Intensional Arguments

- `Willy wants a unicorn`
  - ∃e act(e,wanting), wanter(e,Willy), wantee(e, λu unicorn(u))
    - "Willy wants anything that satisfies the unicorn predicate"
    - here the wantee is a type of entity
- Problem (a fine point I'll gloss over):

Intensional Arguments

- `Willy wants a unicorn`
  - ∃e act(e,wanting), wanter(e,Willy), wantee(e, λu unicorn(u))
    - "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 set of unicorns ("extension")

Intensional Arguments

- `Willy wants a unicorn`
  - ∃e act(e,wanting), wanter(e,Willy), wantee(e, λu unicorn(u))
    - "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)

Intensional Arguments

- `Willy wants a unicorn`
  - ∃e act(e,wanting), wanter(e,Willy), wantee(e, λu unicorn(u))
    - "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 set of unicorns ("extension")
  - But this set is empty: λg unicorn(g) = λg FALSE = λg dodo(g)
  - Then `wants a unicorn` = `wants a dodo`. Oops!

Intensional Arguments

- `Willy wants a unicorn`
  - ∃e act(e,wanting), wanter(e,Willy), wantee(e, λu unicorn(u))
    - "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 set of unicorns ("extension")
  - 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 criteria for unicornness ("intension")

Intensional Arguments

- `Willy wants a unicorn`
  - ∃e act(e,wanting), wanter(e,Willy), wantee(e, λu unicorn(u))
    - "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 set of unicorns ("extension")
  - 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 criteria for unicornness ("intension")
- Traditional solution involves "possible-world semantics"

Intensional Arguments

- `Willy wants a unicorn`
  - ∃e act(e,wanting), wanter(e,Willy), wantee(e, λu unicorn(u))
    - "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 set of unicorns ("extension")
  - 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 criteria for unicornness ("intension")
- Traditional solution involves "possible-world semantics"
  - Can imagine other worlds where set of unicorn ≠ set of dodos

Intensional Arguments

- `Willy wants a unicorn`
  - ∃e act(e,wanting), wanter(e,Willy), wantee(e, λu unicorn(u))
    - "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 set of unicorns ("extension")
  - 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 criteria for unicornness ("intension")
- 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

Control

- `Willy wants Lilly to get married`

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)])

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)])

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`

Control

- `Willy wants Lilly to get married`
  - $\exists$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`

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 …`

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 <u>controlled</u> by the subject of "wants."

Nouns and Their Modifiers

Nouns and Their Modifiers

- `expert`
  - λg expert(g)

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

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 with different intonation:
    - λg (Modified-by(Baltimore, expert))(g) – expert on Baltimore
    - Can't use Related for this 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

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)]

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

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

Adverbs

- `Lili passionately wants Billy`
  - Wrong?: passionately(want(Lili,Billy)) = passionately(true)
  - Better: (passionately(want))(Lili,Billy)
  - Best: ∃e present(e), act(e,wanting), wanter(e,Lili), wantee(e, Billy), manner(e, passionate)

Adverbs

- `Lili passionately wants Billy`
  - Wrong?: passionately(want(Lili,Billy)) = passionately(true)
  - Better: (passionately(want))(Lili,Billy)
  - Best: ∃e present(e), act(e,wanting), wanter(e,Lili), wantee(e, Billy), manner(e, passionate)
- `Lili often stalks Billy`
  - (often(stalk))(Lili,Billy)
  - many(day, λd ∃e present(e), act(e,stalking), stalker(e,Lili), stalkee(e, Billy), during(e,d))

Adverbs

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

Speech Acts

Speech Acts

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

Speech Acts

- What is the meaning of a full sentence?
  - Depends on the punctuation mark at the end. ☺
  - Billy likes Lili. → **assert**(like(B,L))
  - Billy likes Lili? → **ask**(like(B,L))
    - or more formally, "Does Billy like Lili?"
  - Billy, like Lili! → **command**(like(B,L))
    - or more accurately, "Let Billy like Lili!"

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

Speech Acts

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?

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(...))

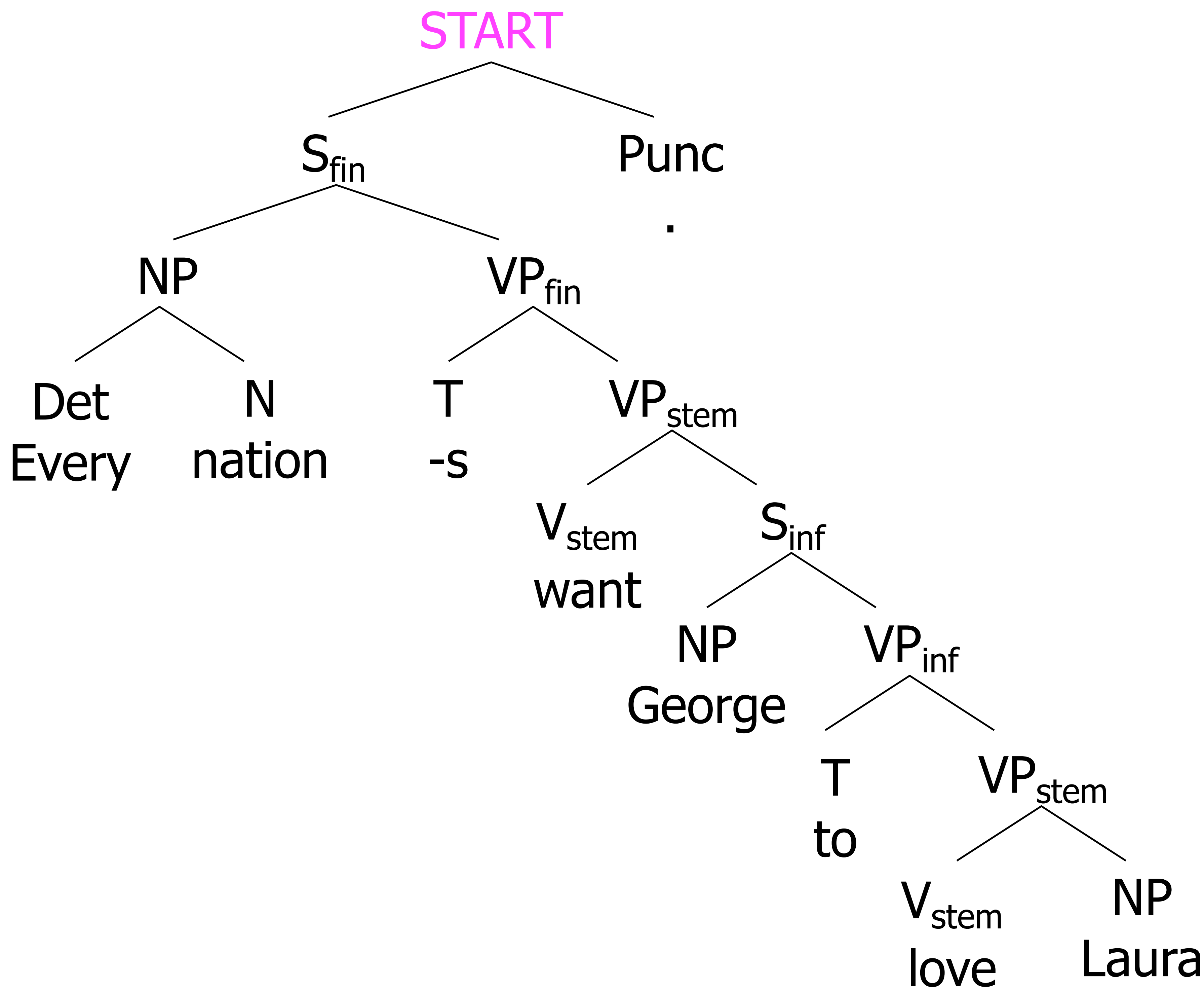
Speech Acts

- `What did Gilly swallow?`
  - **ask**($\lambda x\ \exists e$ 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**($\exists e$ past(e), act(e,eating), eater(f,Speaker), eatee(…))

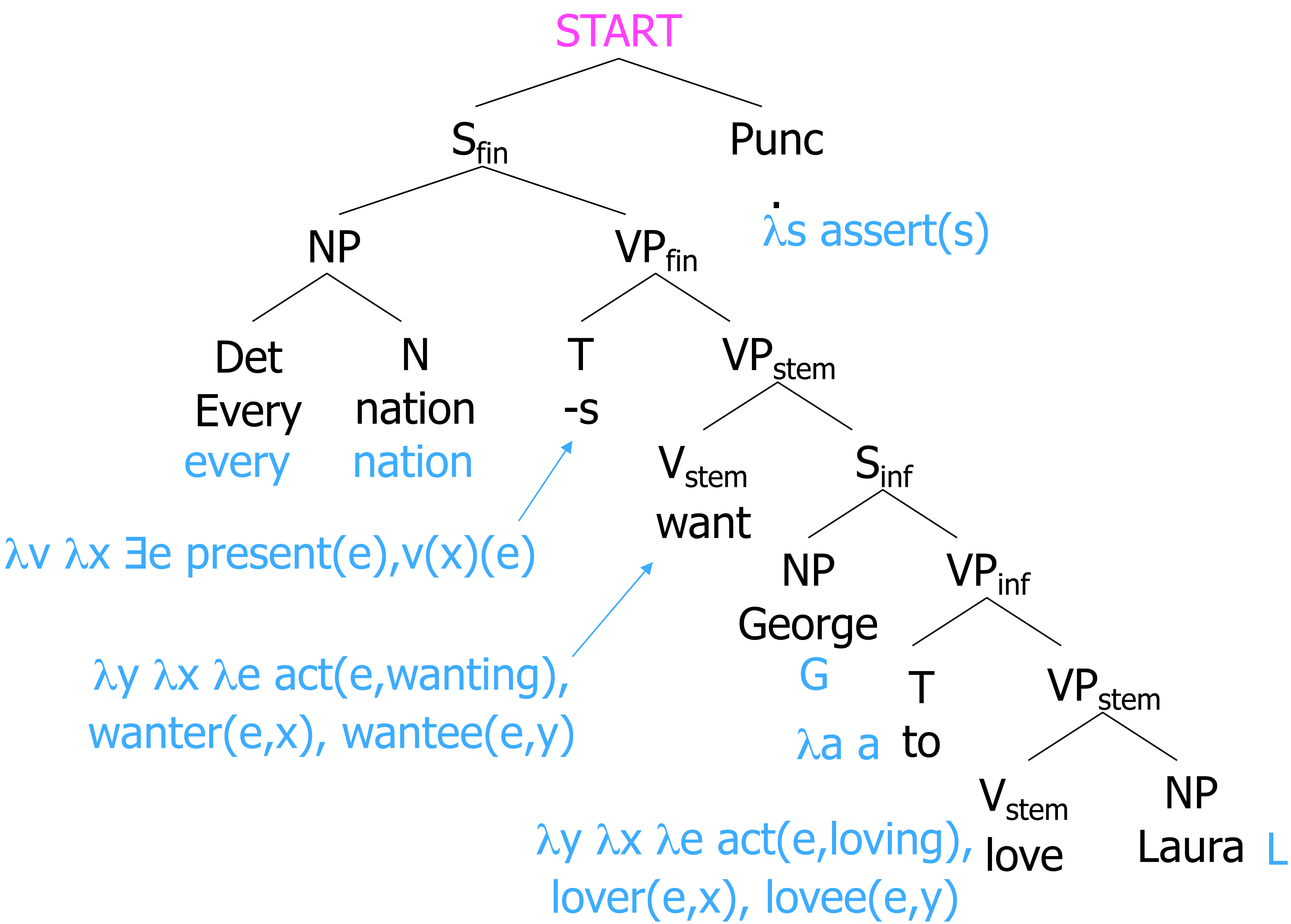
Compositional Semantics

- 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

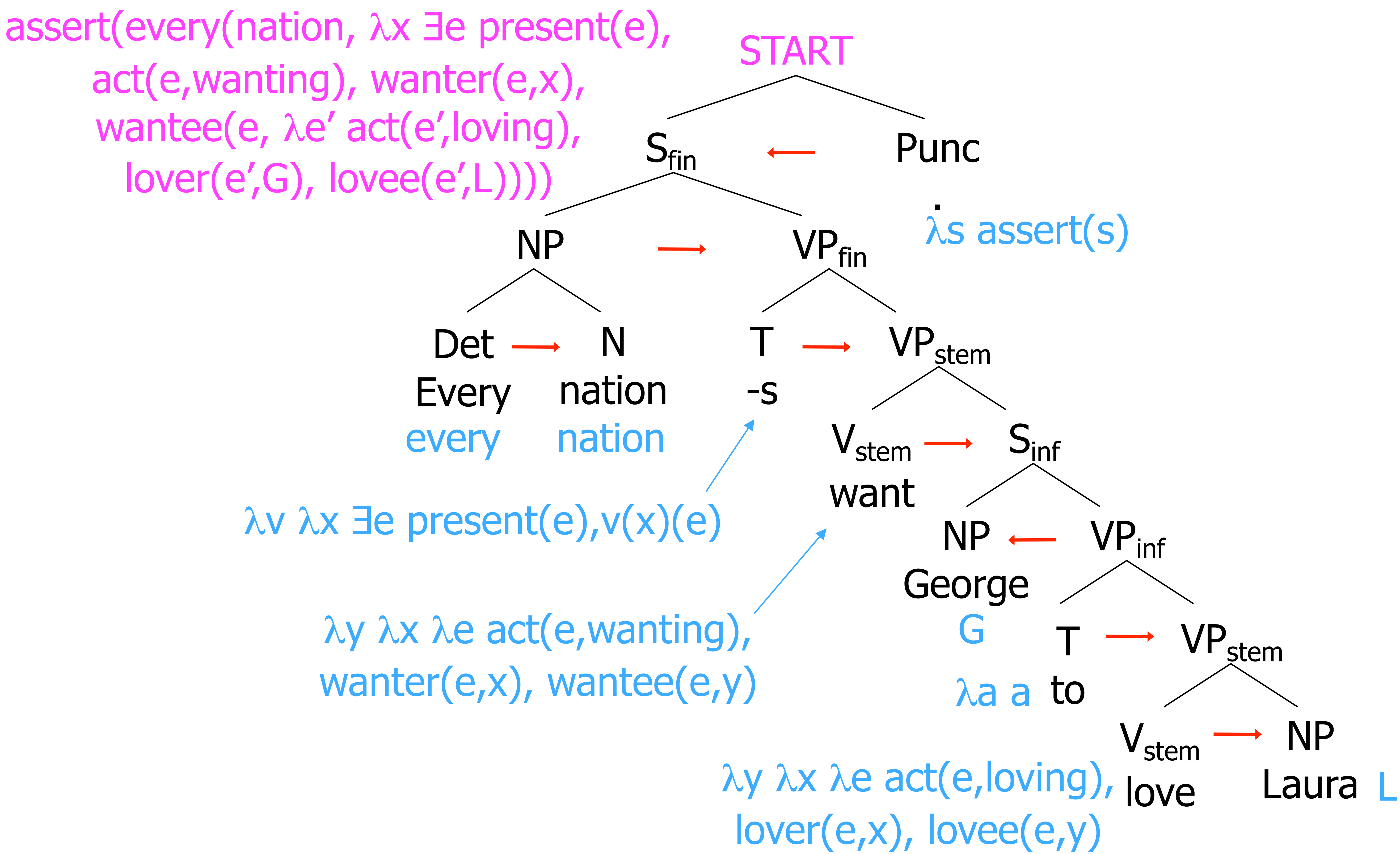
Compositional Semantics



Compositional Semantics



Compositional Semantics



Compositional Semantics

Compositional Semantics

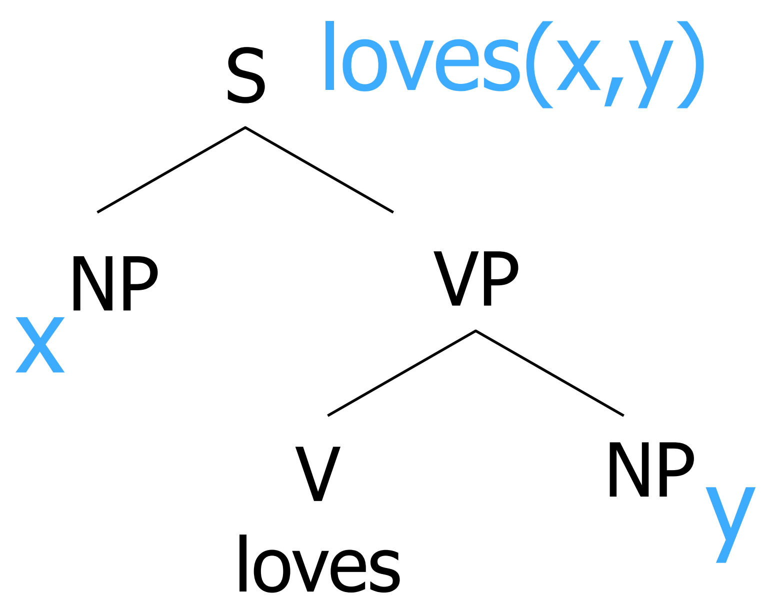
- Add a “sem” feature to each context-free rule
  - S → NP loves NP
  - S[sem=loves(x,y)] → NP[sem=x] loves NP[sem=y]
  - Meaning of S depends on meaning of NPs

Compositional Semantics

- Add a “sem” feature to each context-free rule
  - S → NP loves NP
  - S[sem=loves(x,y)] → NP[sem=x] loves NP[sem=y]
  - Meaning of S depends on meaning of NPs
- TAG version:

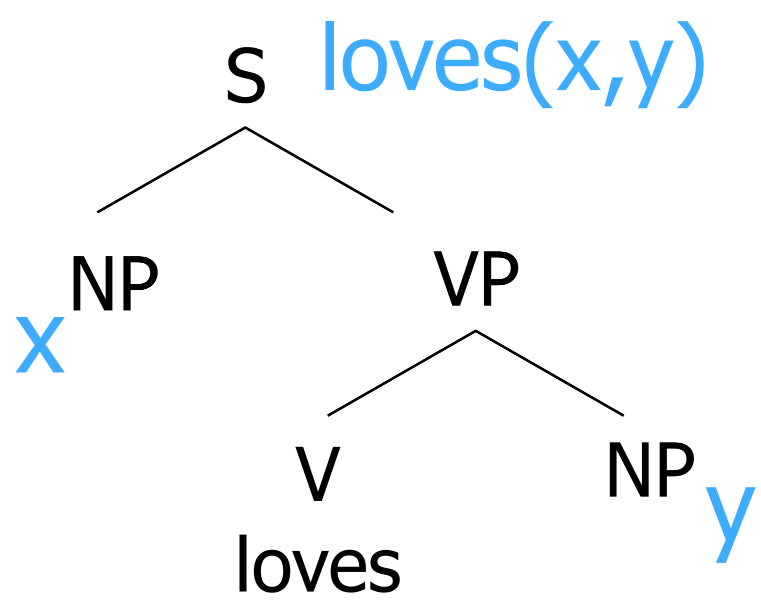
Compositional Semantics

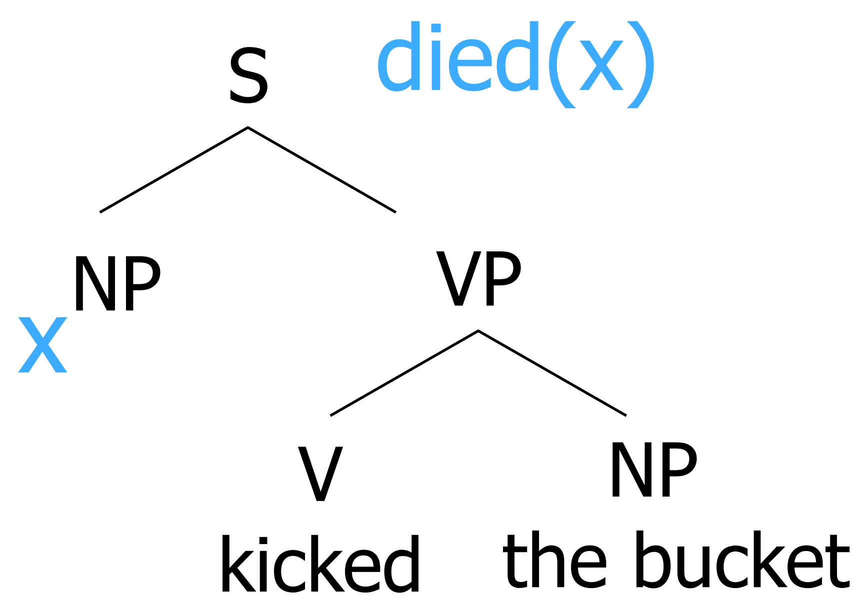
- Add a "sem" feature to each context-free rule
  - S → NP loves NP
  - S[sem=loves(x,y)] → NP[sem=x] loves NP[sem=y]
  - Meaning of S depends on meaning of NPs
- TAG version:



Compositional Semantics

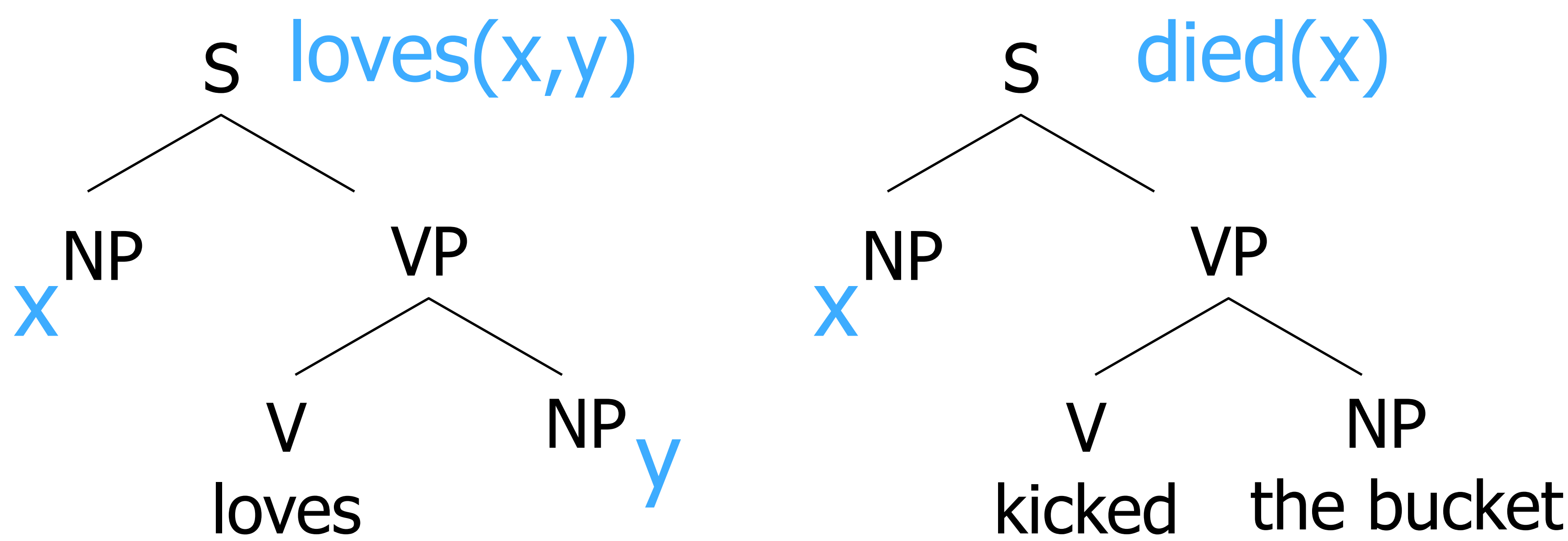
- Add a "sem" feature to each context-free rule
  - S → NP loves NP
  - S[sem=loves(x,y)] → NP[sem=x] loves NP[sem=y]
  - Meaning of S depends on meaning of NPs
- TAG version:





Compositional Semantics

- Add a "sem" feature to each context-free rule
  - S → NP loves NP
  - S[sem=loves(x,y)] → NP[sem=x] loves NP[sem=y]
  - Meaning of S depends on meaning of NPs
- TAG version:



- Template filling: S[sem=showflights(x,y)] →
  I want a flight from NP[sem=x] to NP[sem=y]

Compositional Semantics

Compositional Semantics

- Instead of S → NP loves NP
  - S[sem=loves(x,y)] → NP[sem=x] loves NP[sem=y]

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]

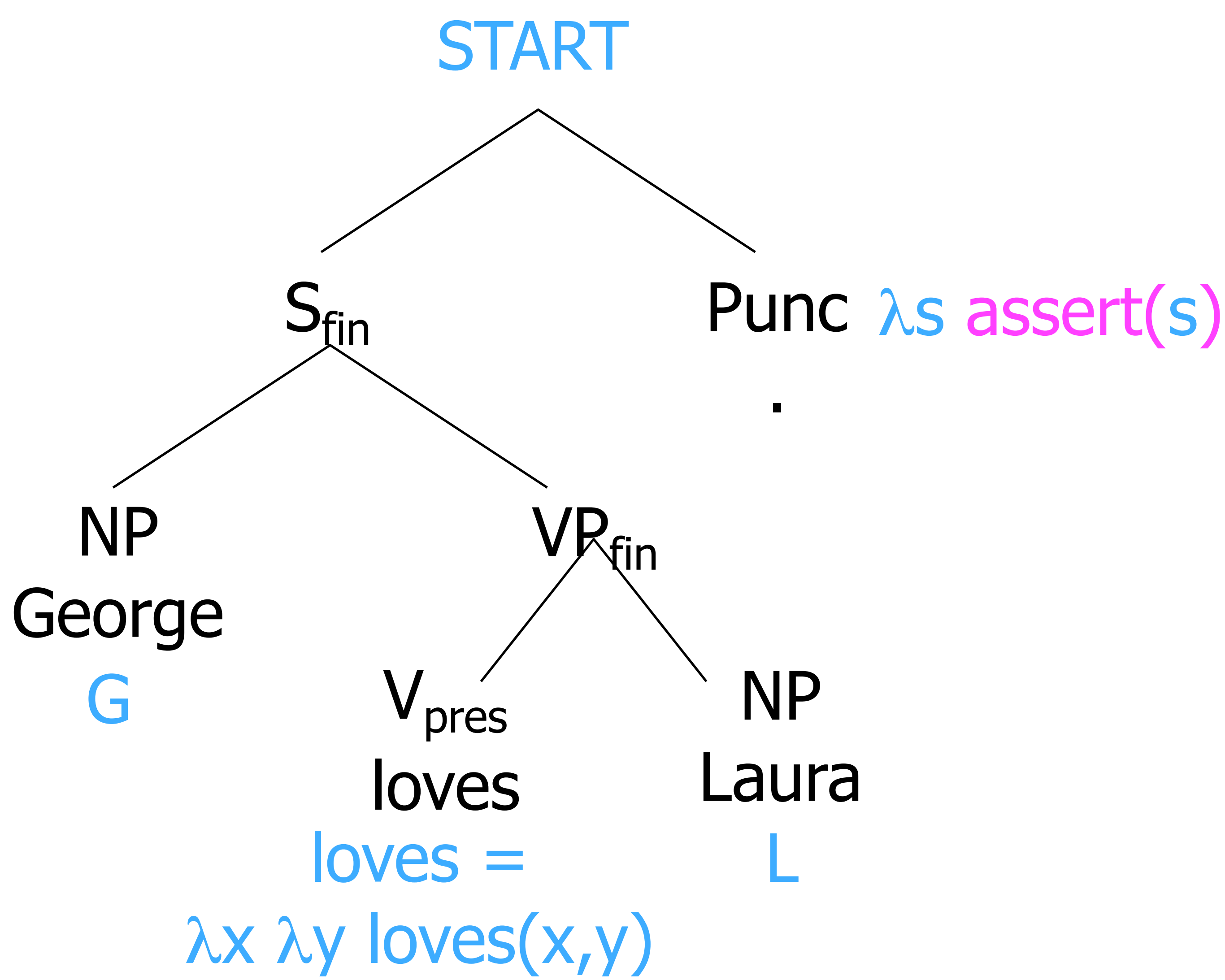
Compositional Semantics

- 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 `George loves Laura` has sem=loves(Laura)(George)

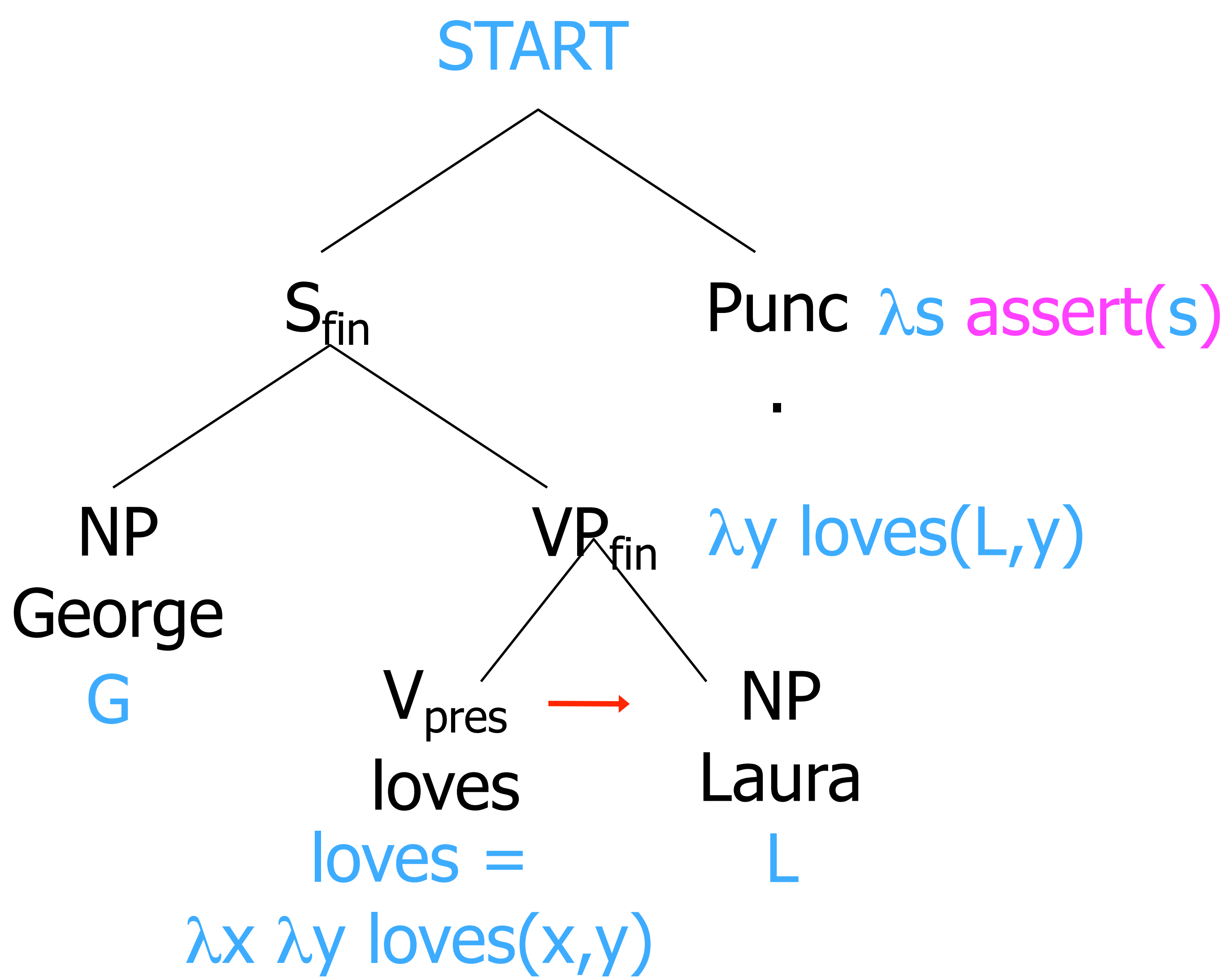
Compositional Semantics

- 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 `George loves Laura` has sem=loves(Laura)(George)
- 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!

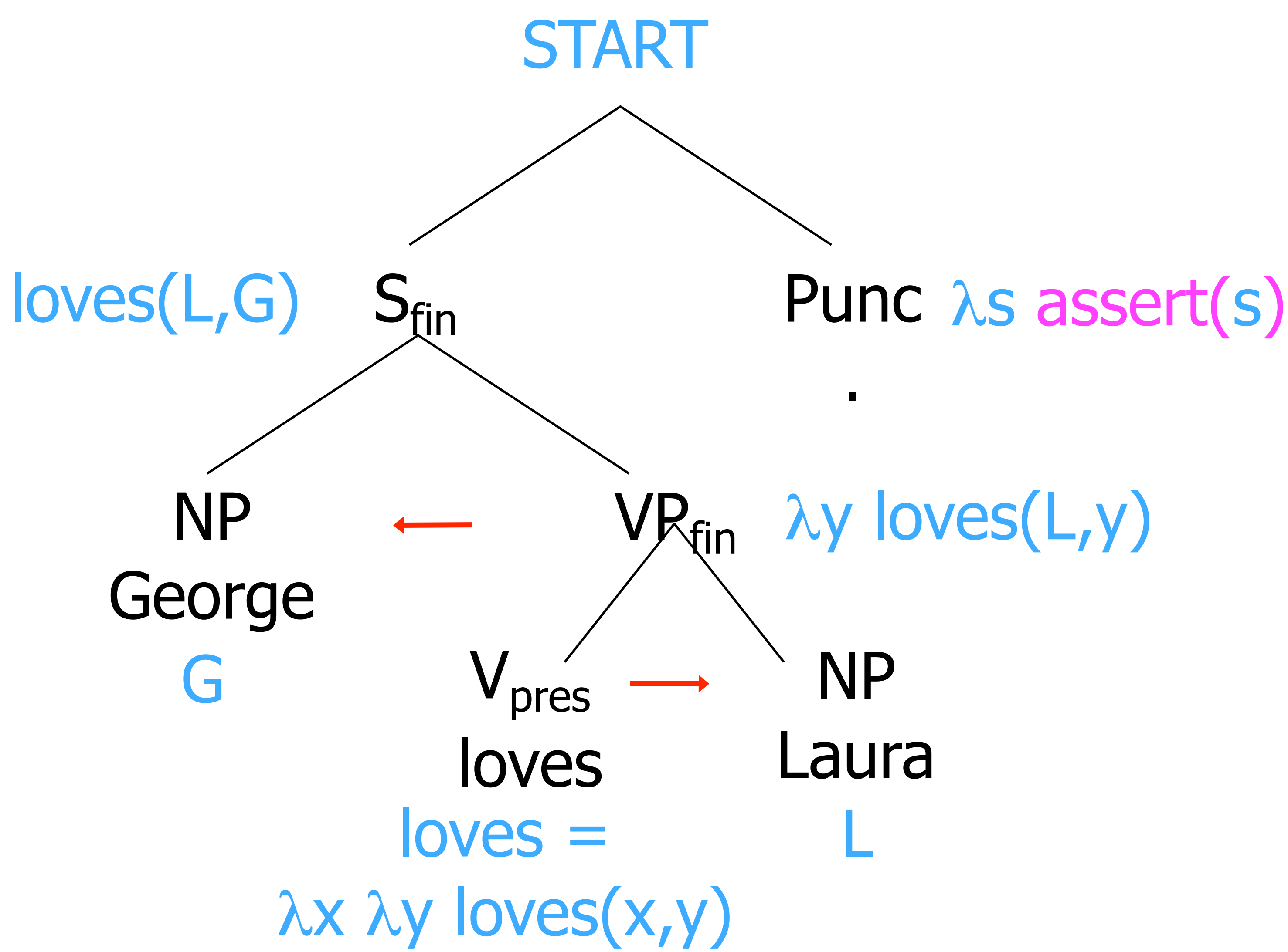
Compositional Semantics



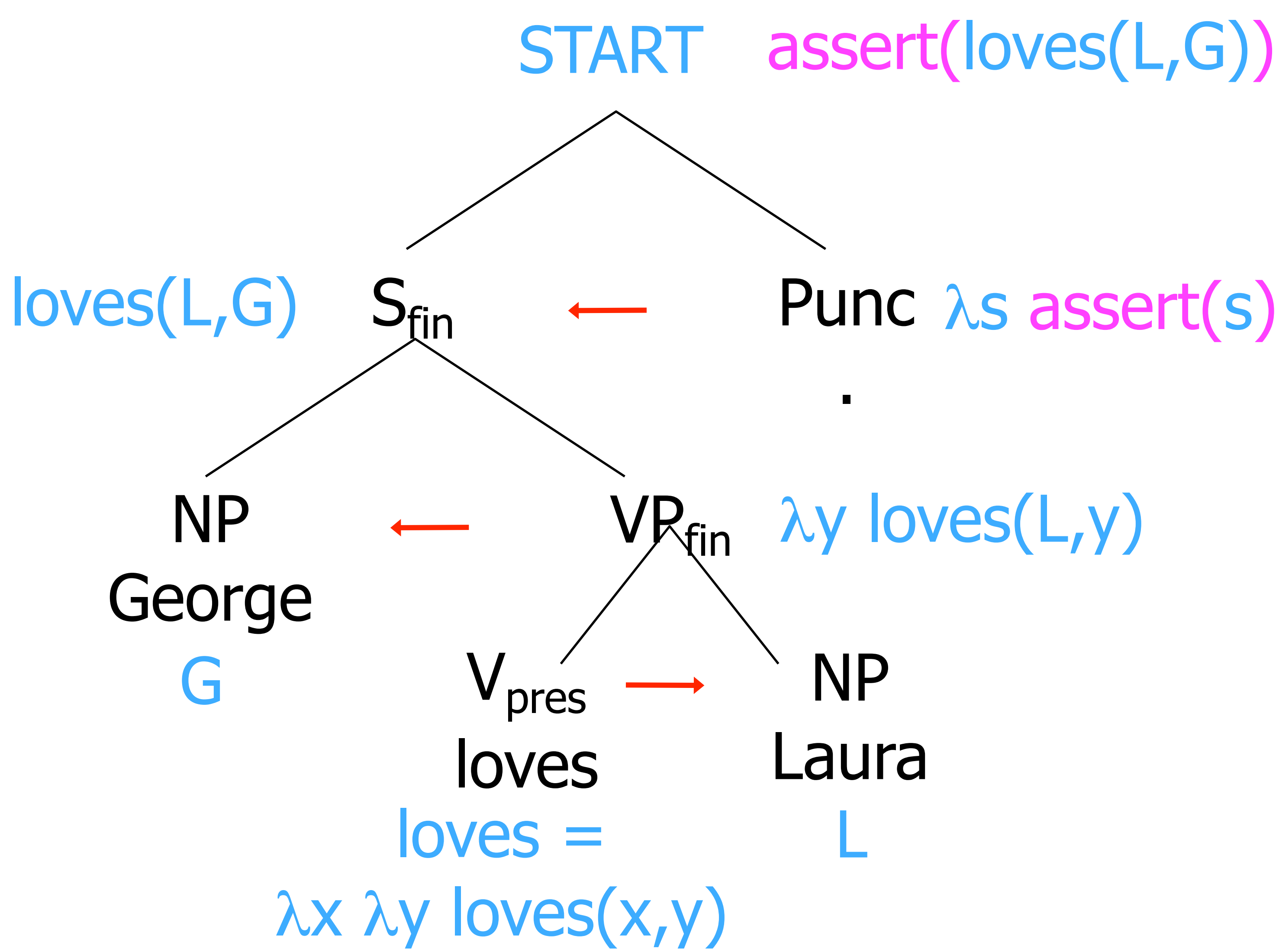
Compositional Semantics



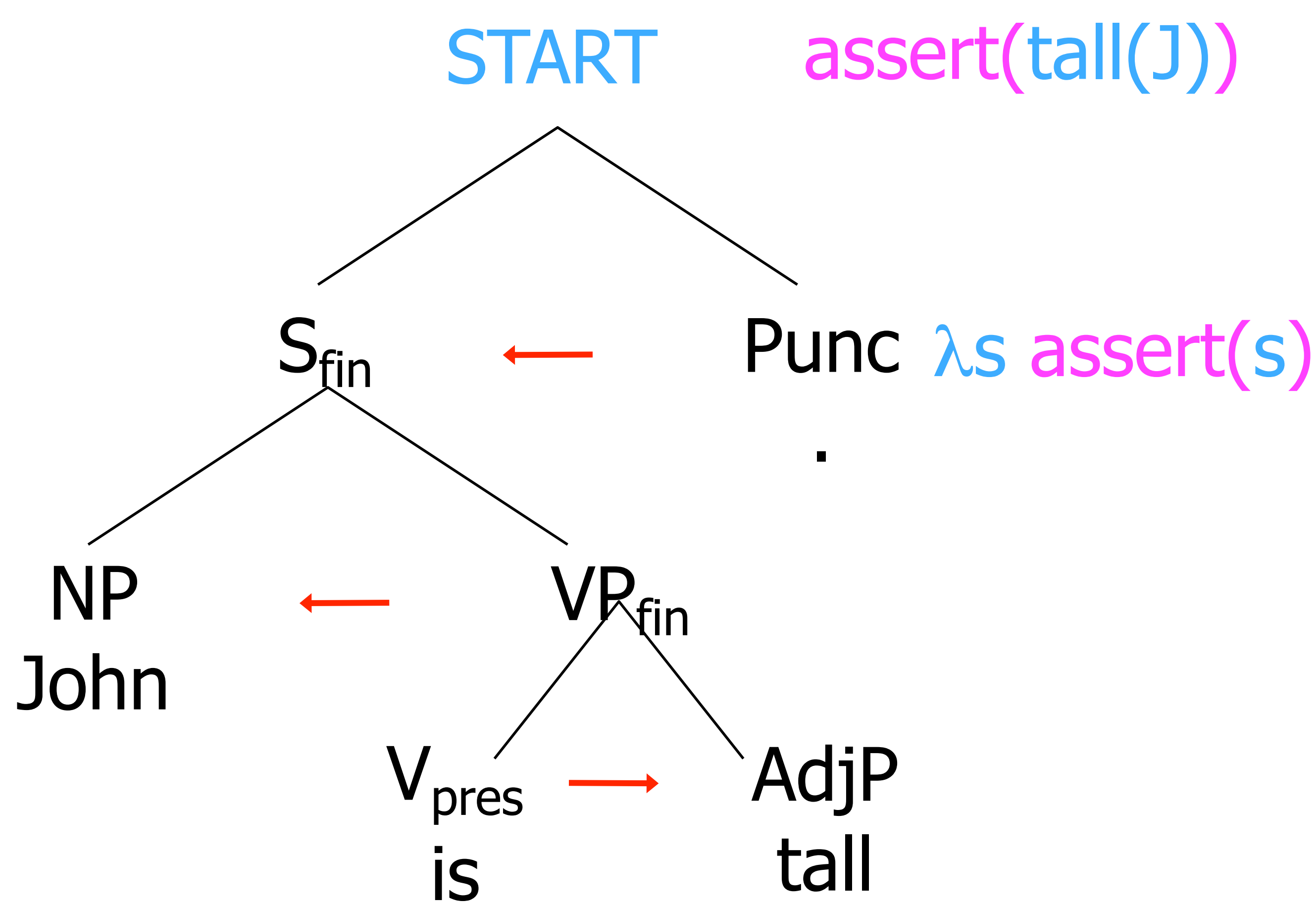
Compositional Semantics



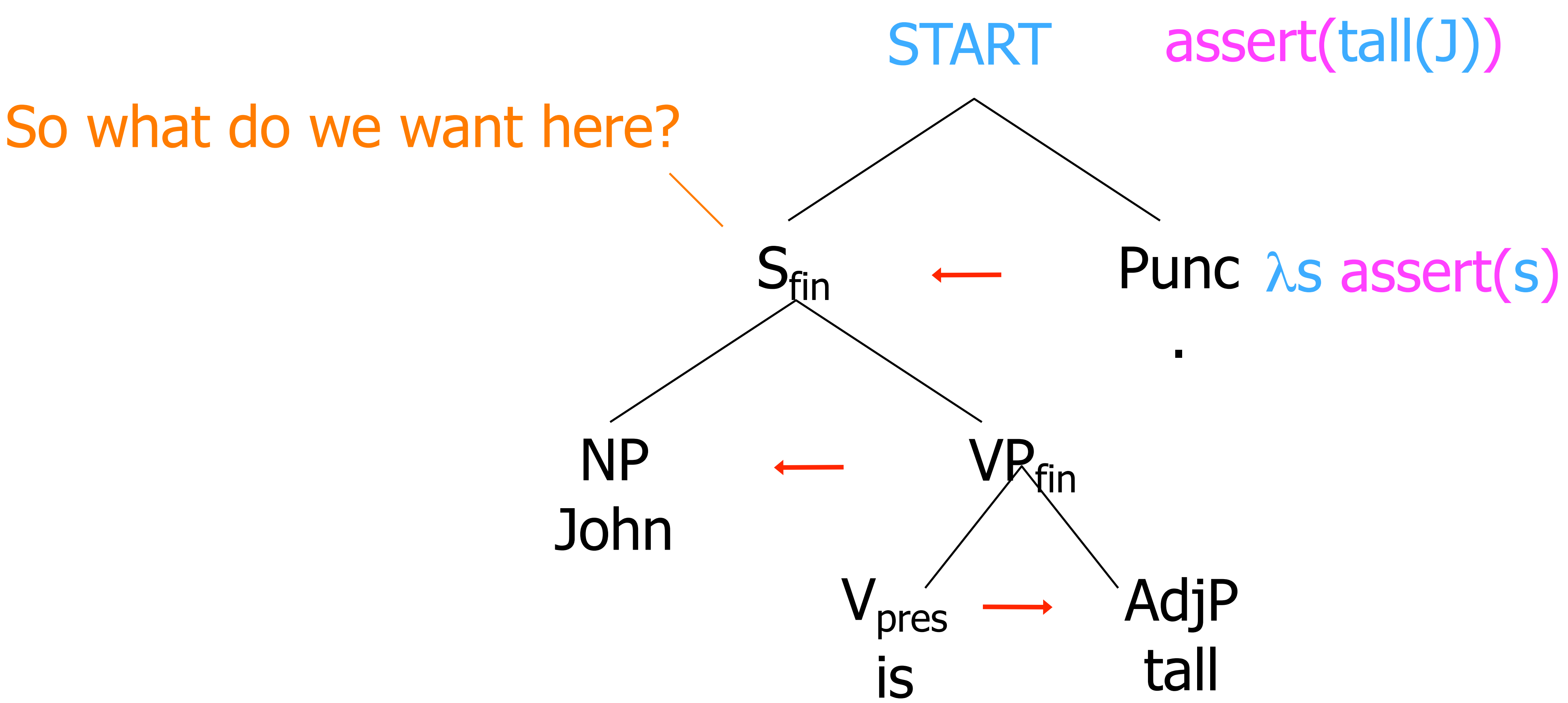
Compositional Semantics



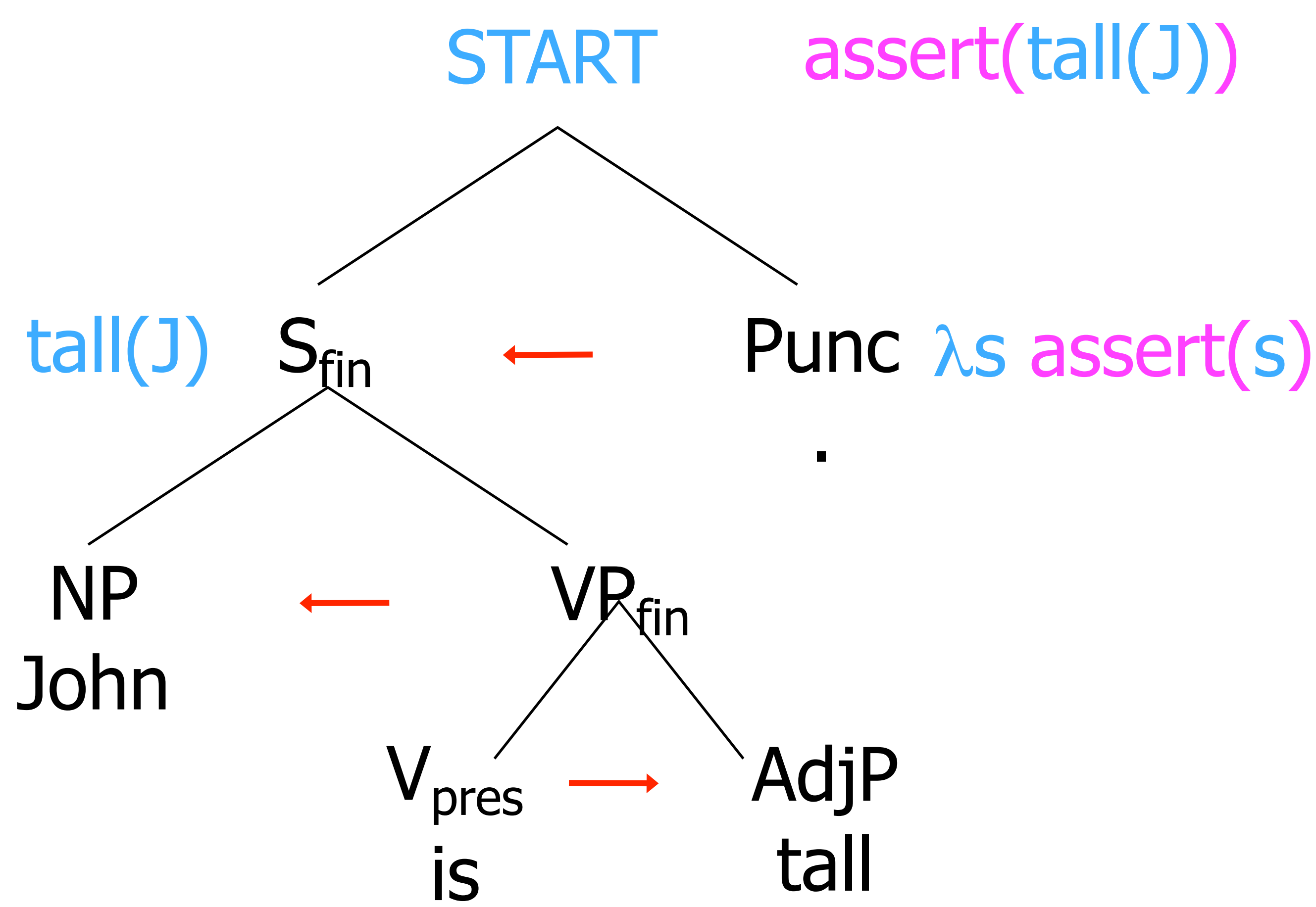
Compositional Semantics



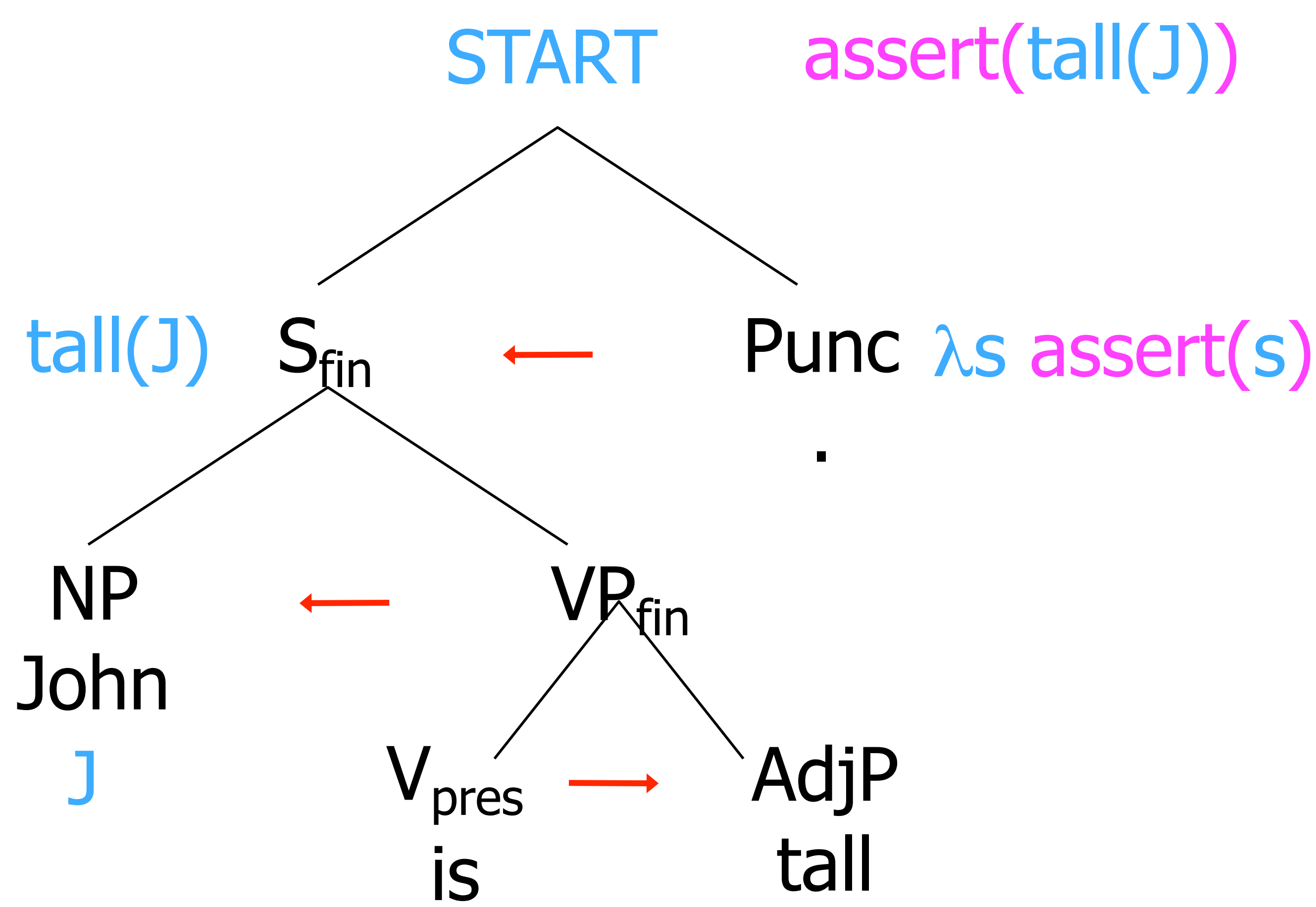
Compositional Semantics



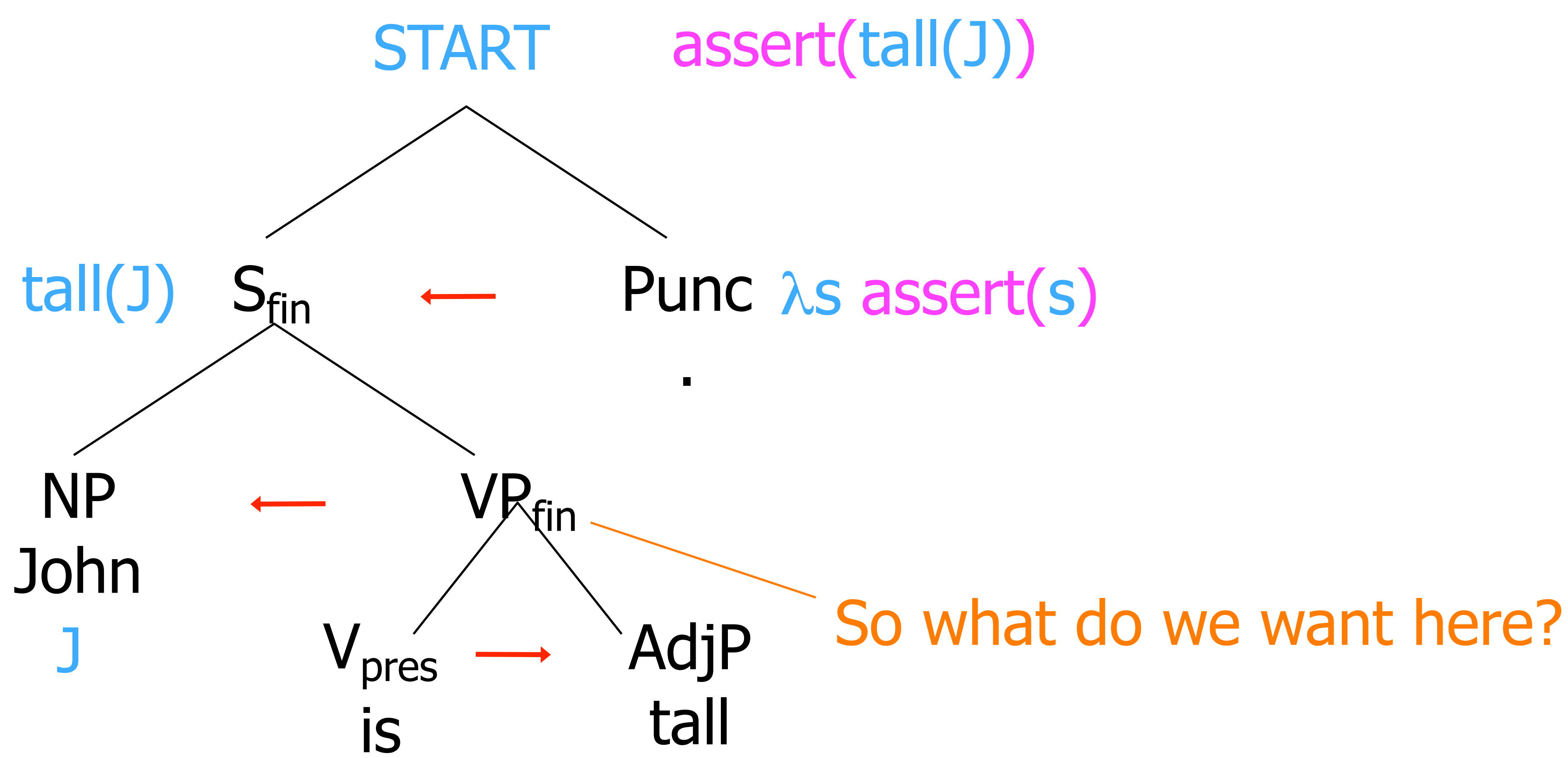
Compositional Semantics



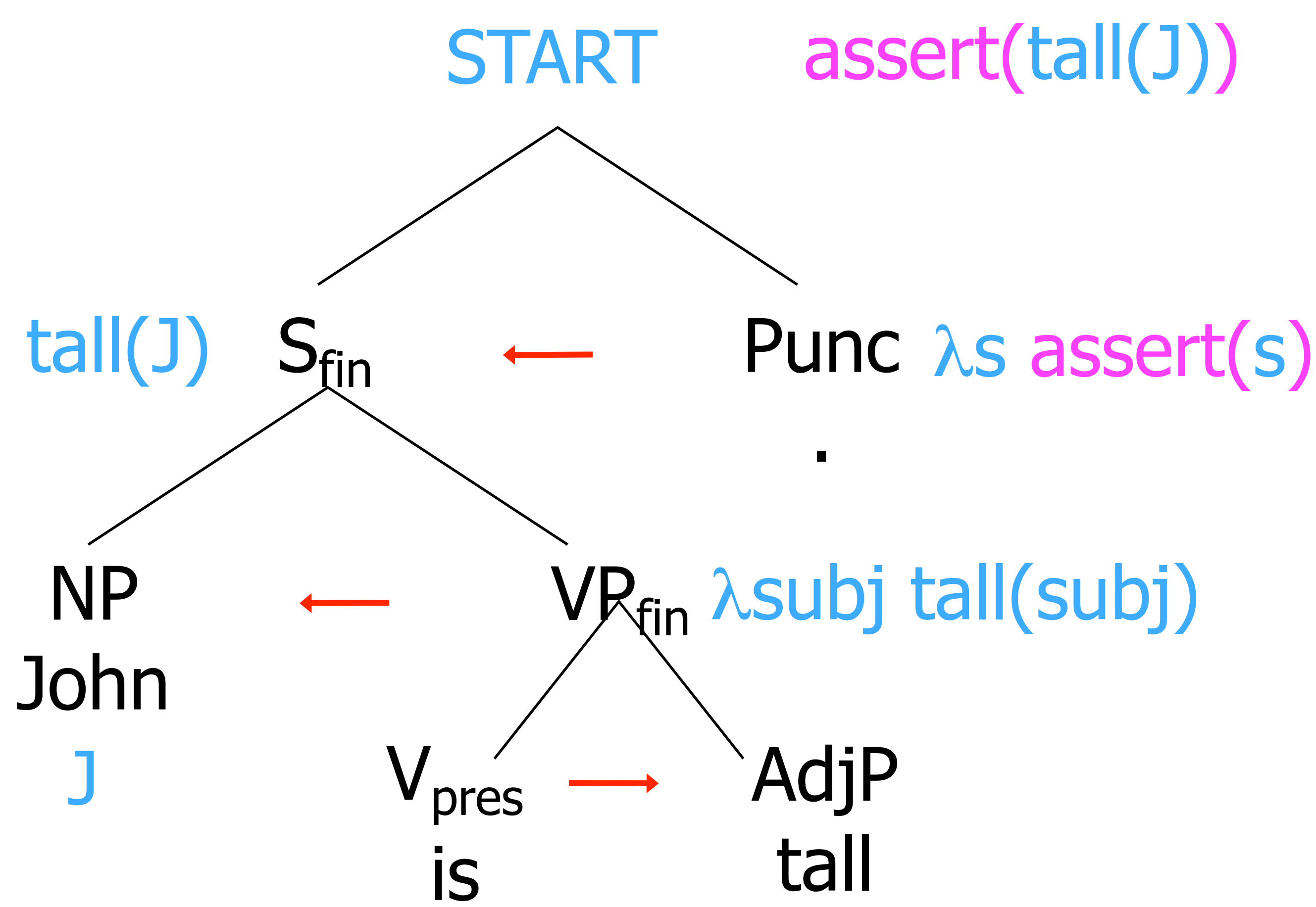
Compositional Semantics



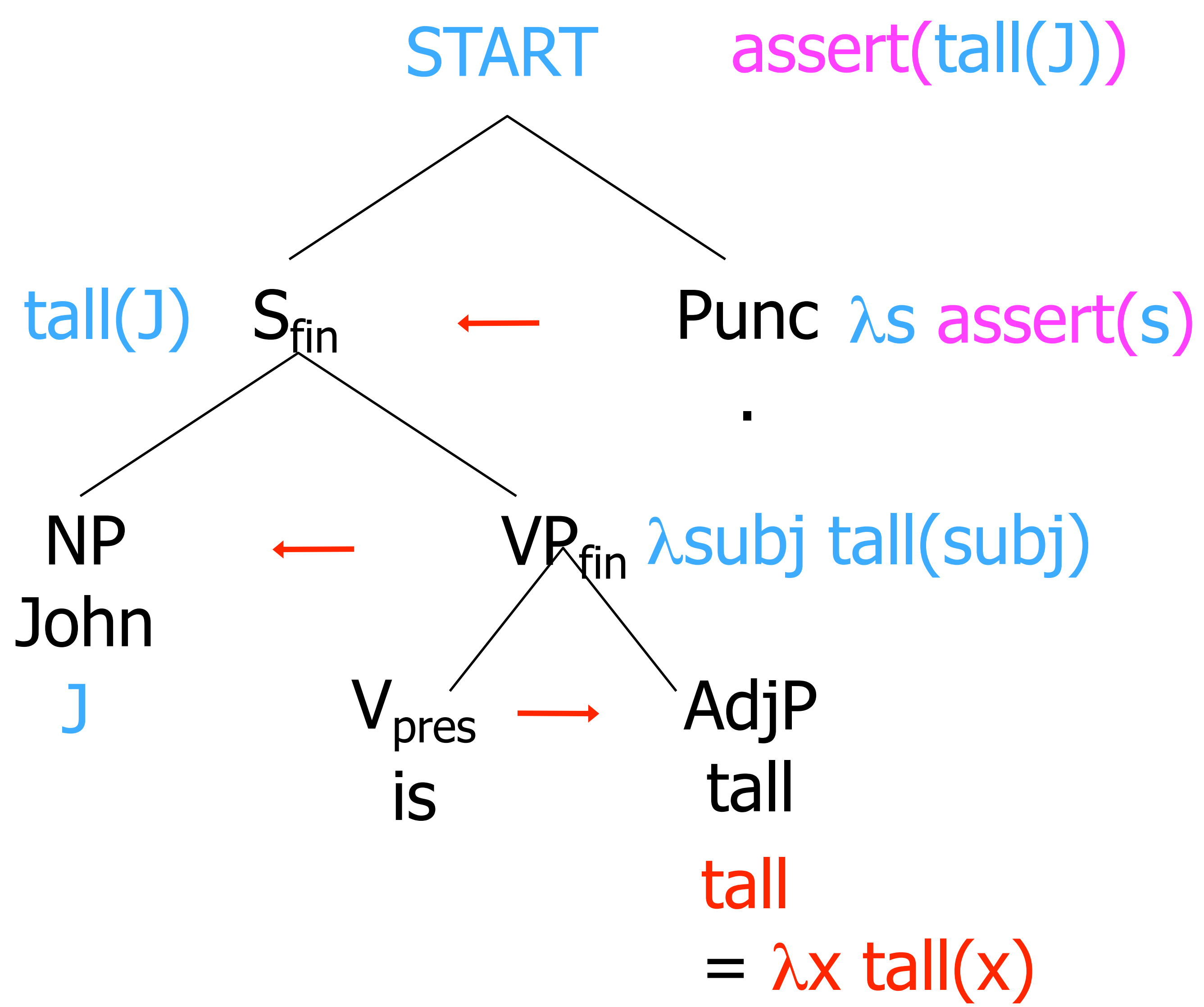
Compositional Semantics



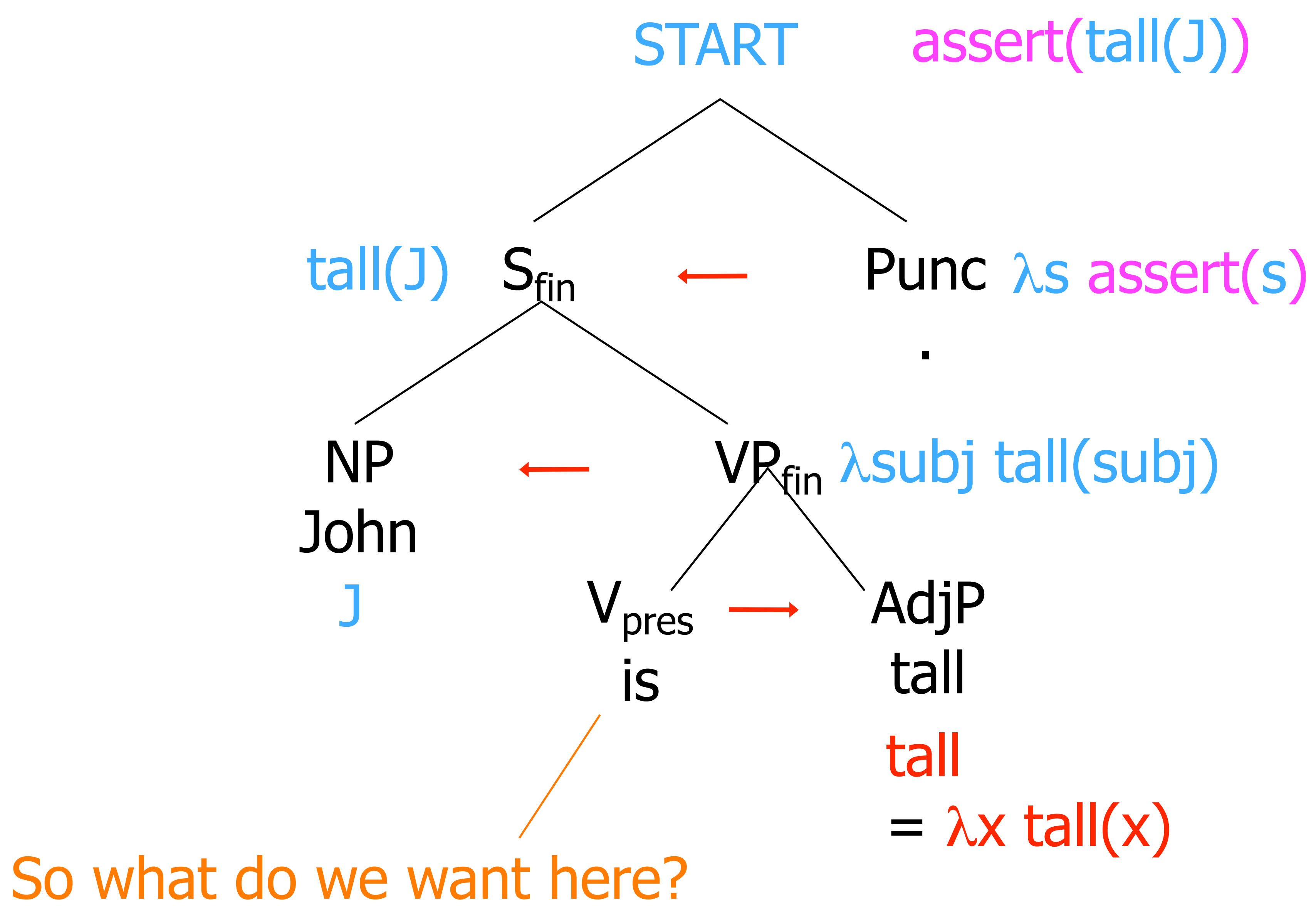
Compositional Semantics



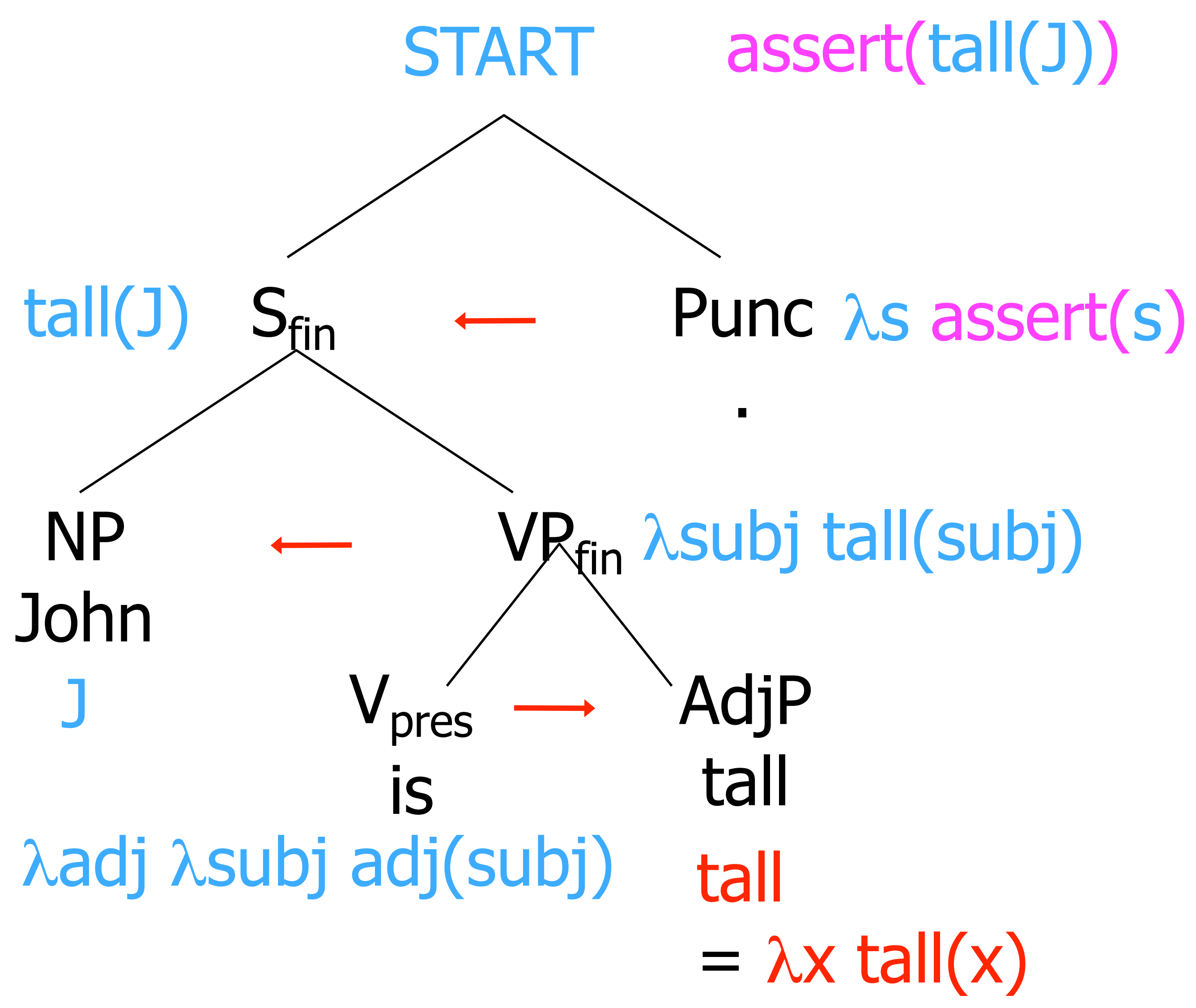
Compositional Semantics



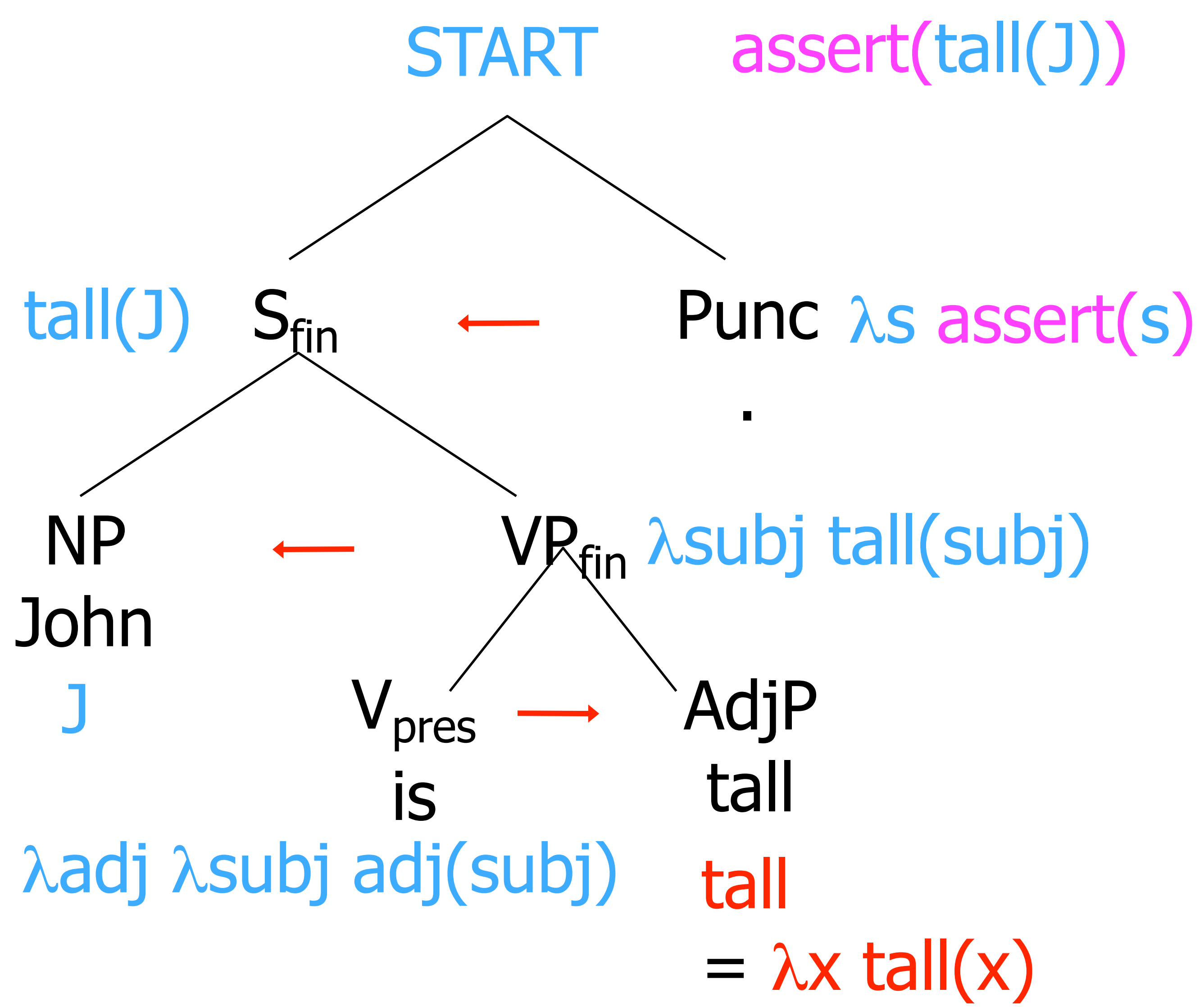
Compositional Semantics



Compositional Semantics



Compositional Semantics

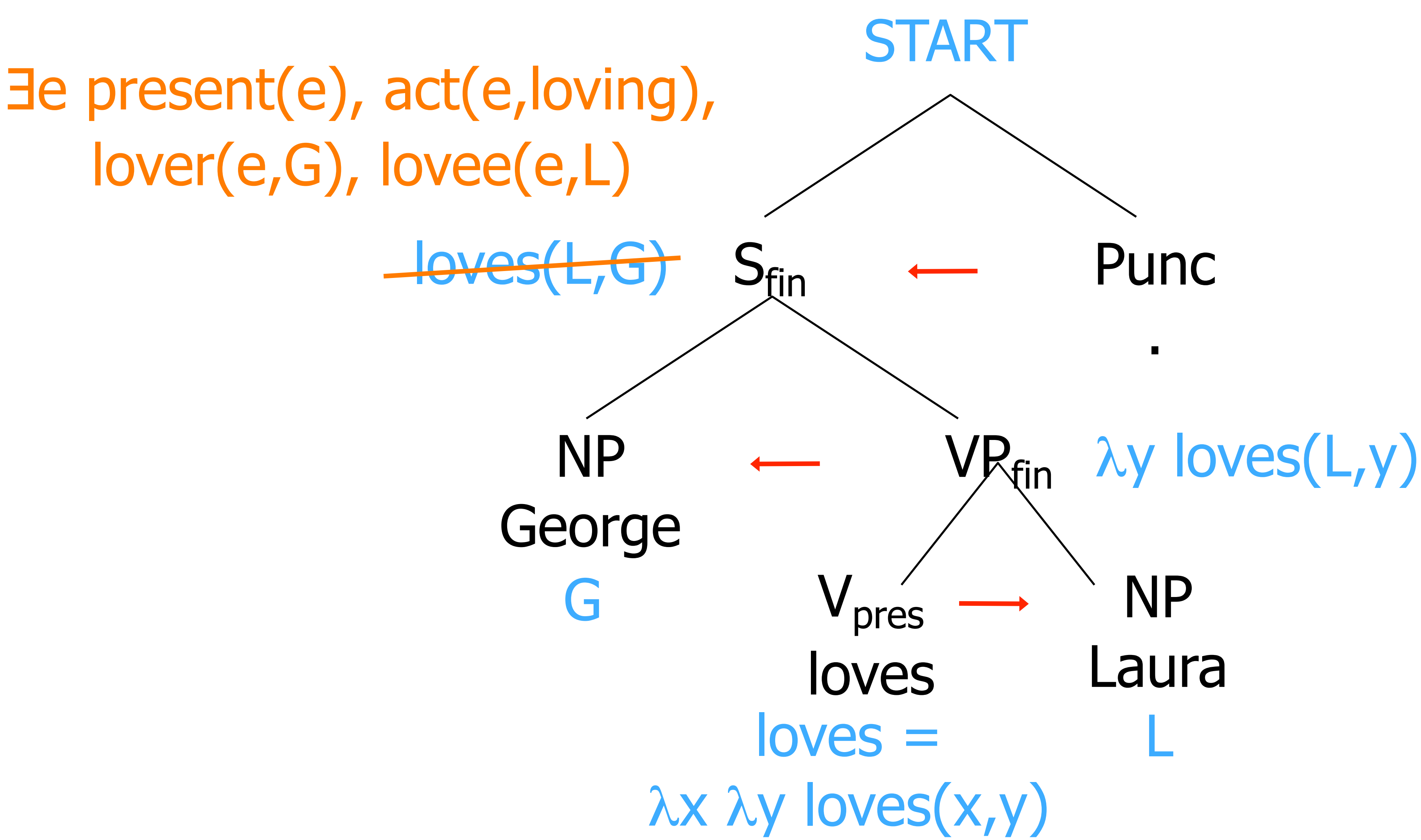


$$(\lambda adj\ \lambda subj\ adj(subj))(\lambda x\ tall(x))$$

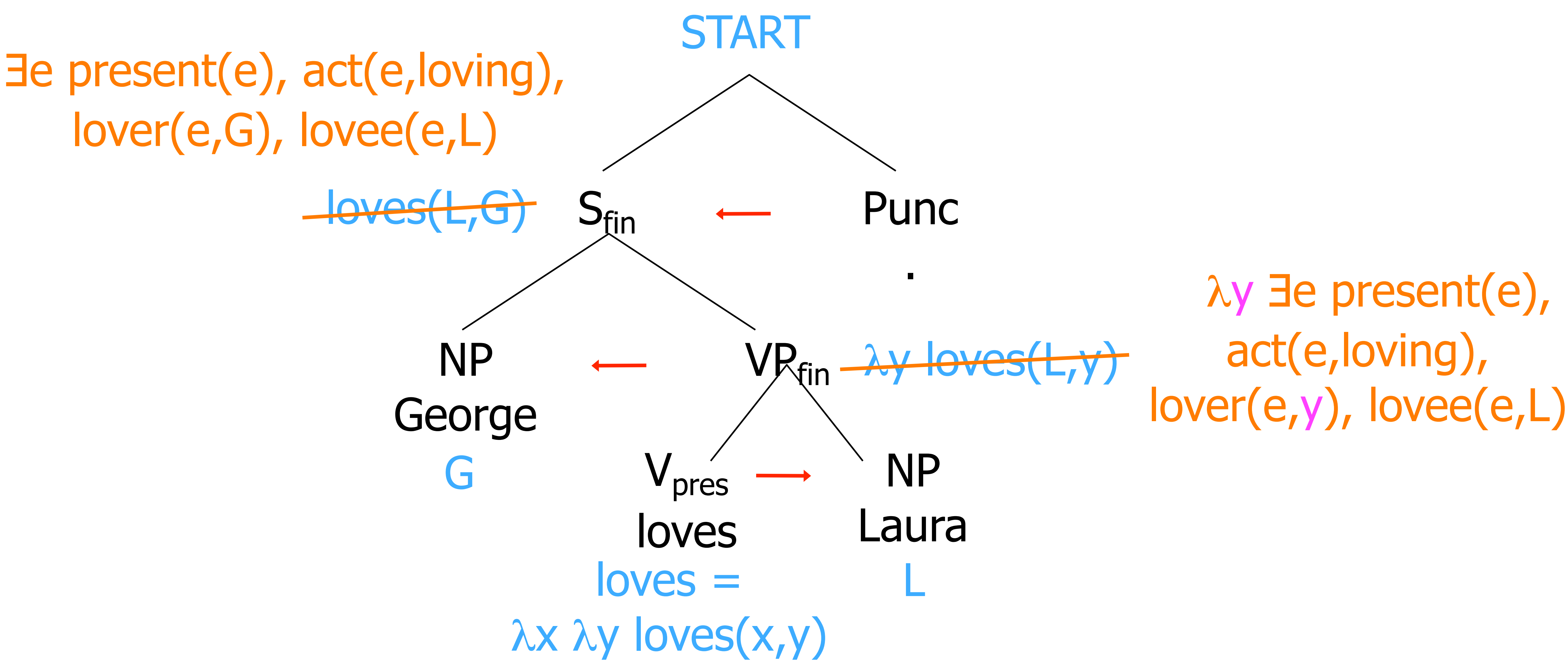
$$= \quad \lambda subj\ (\lambda x\ tall(x))(subj)$$

$$= \quad \lambda subj\ \quad tall(subj)$$

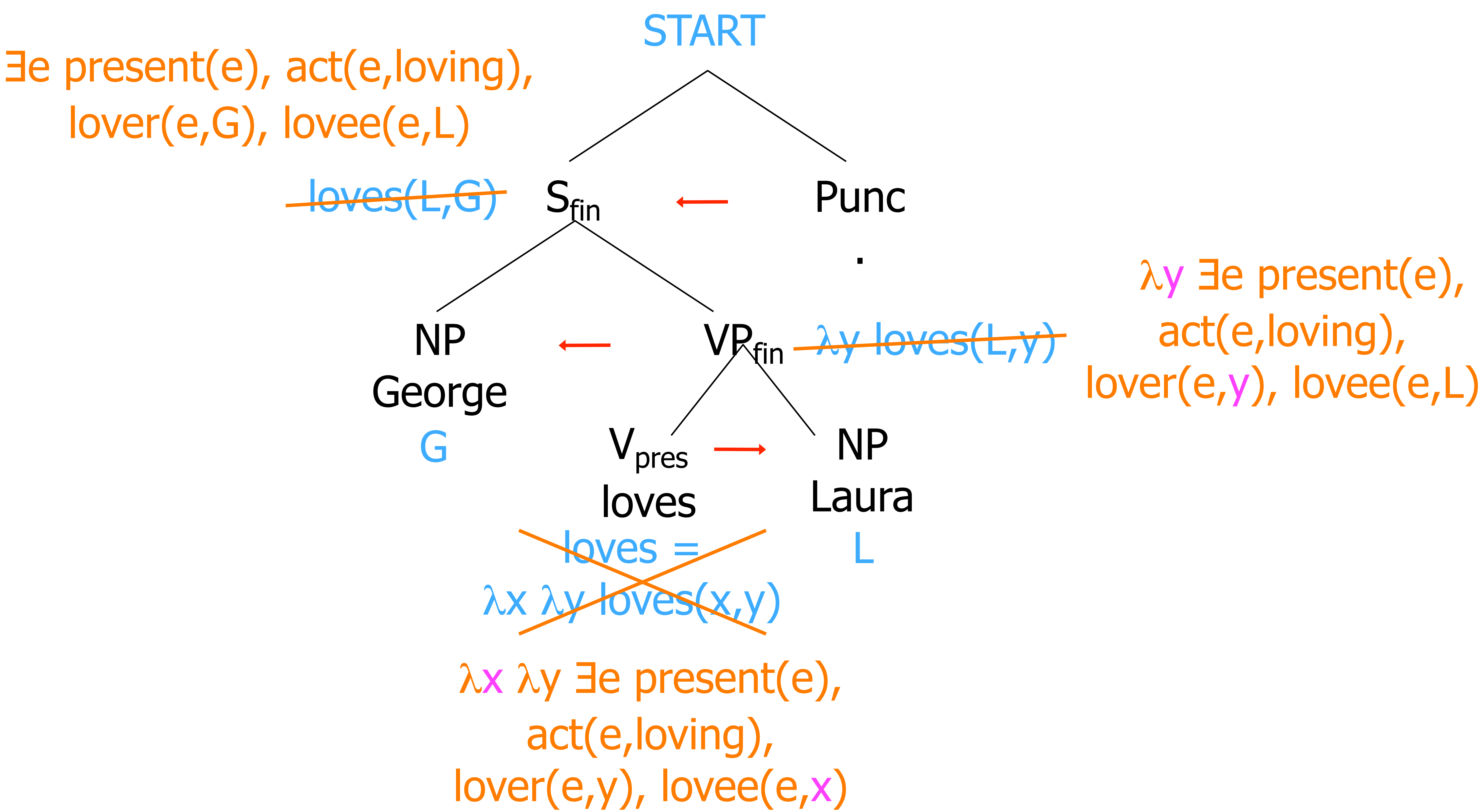
Compositional Semantics

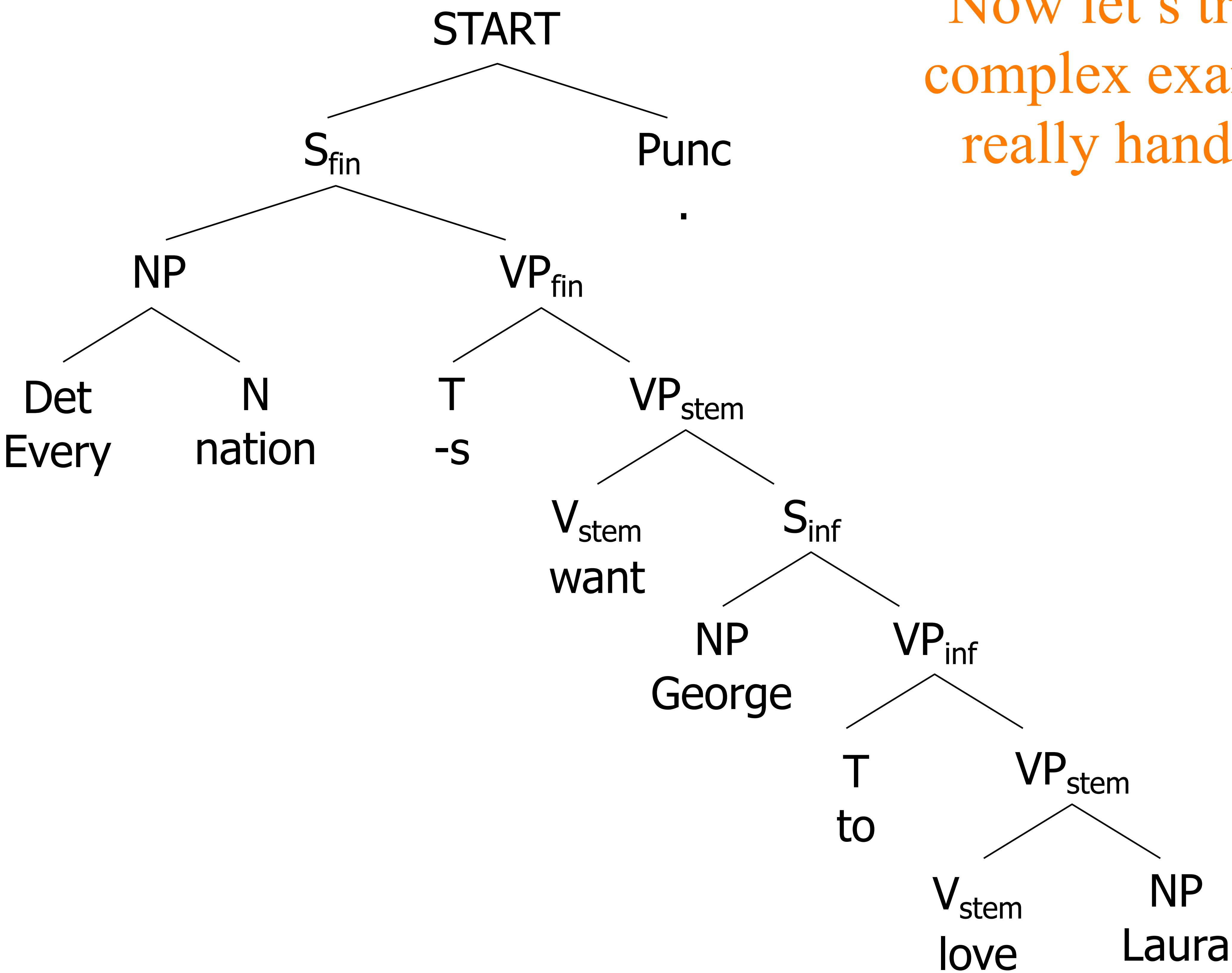


Compositional Semantics



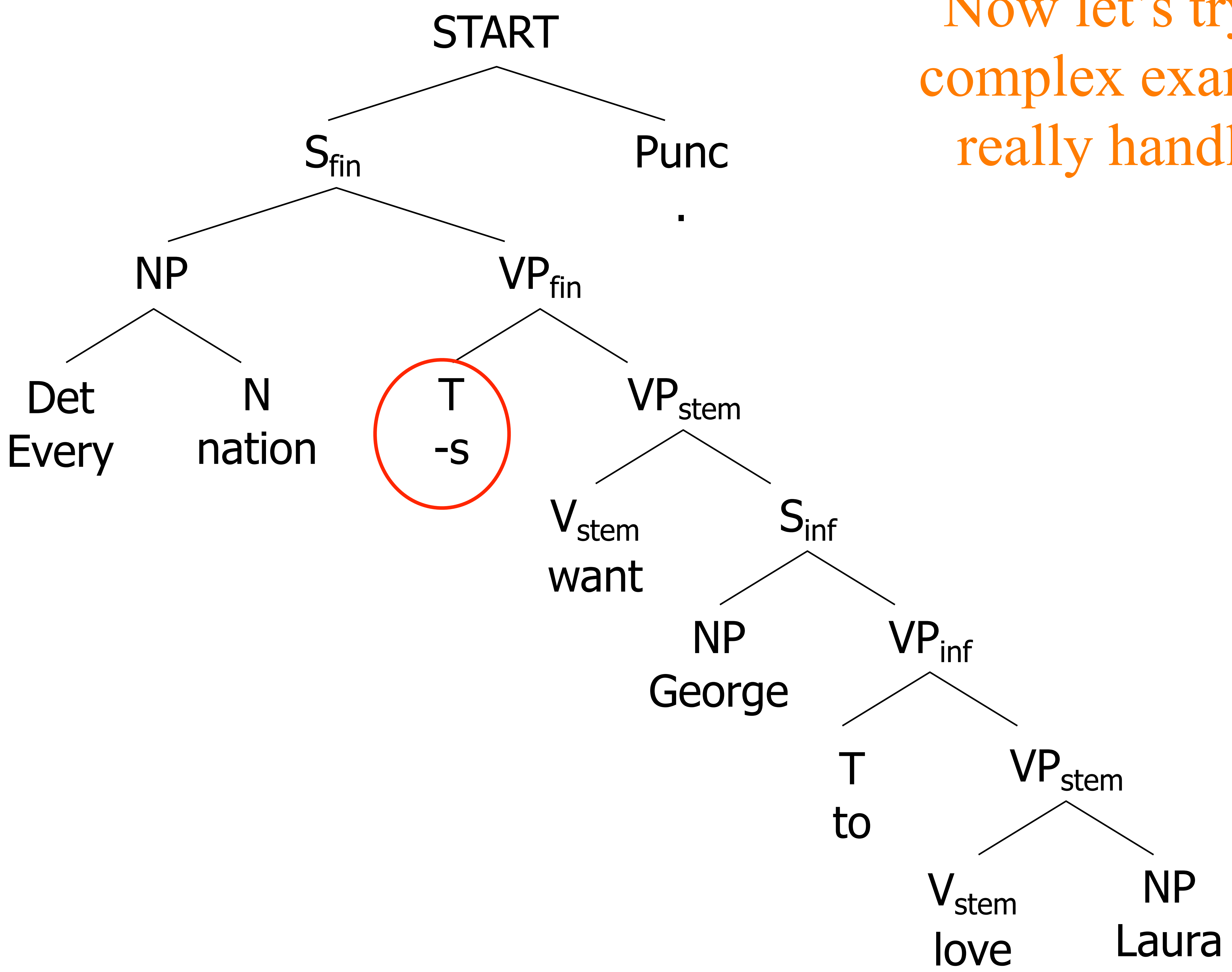
Compositional Semantics



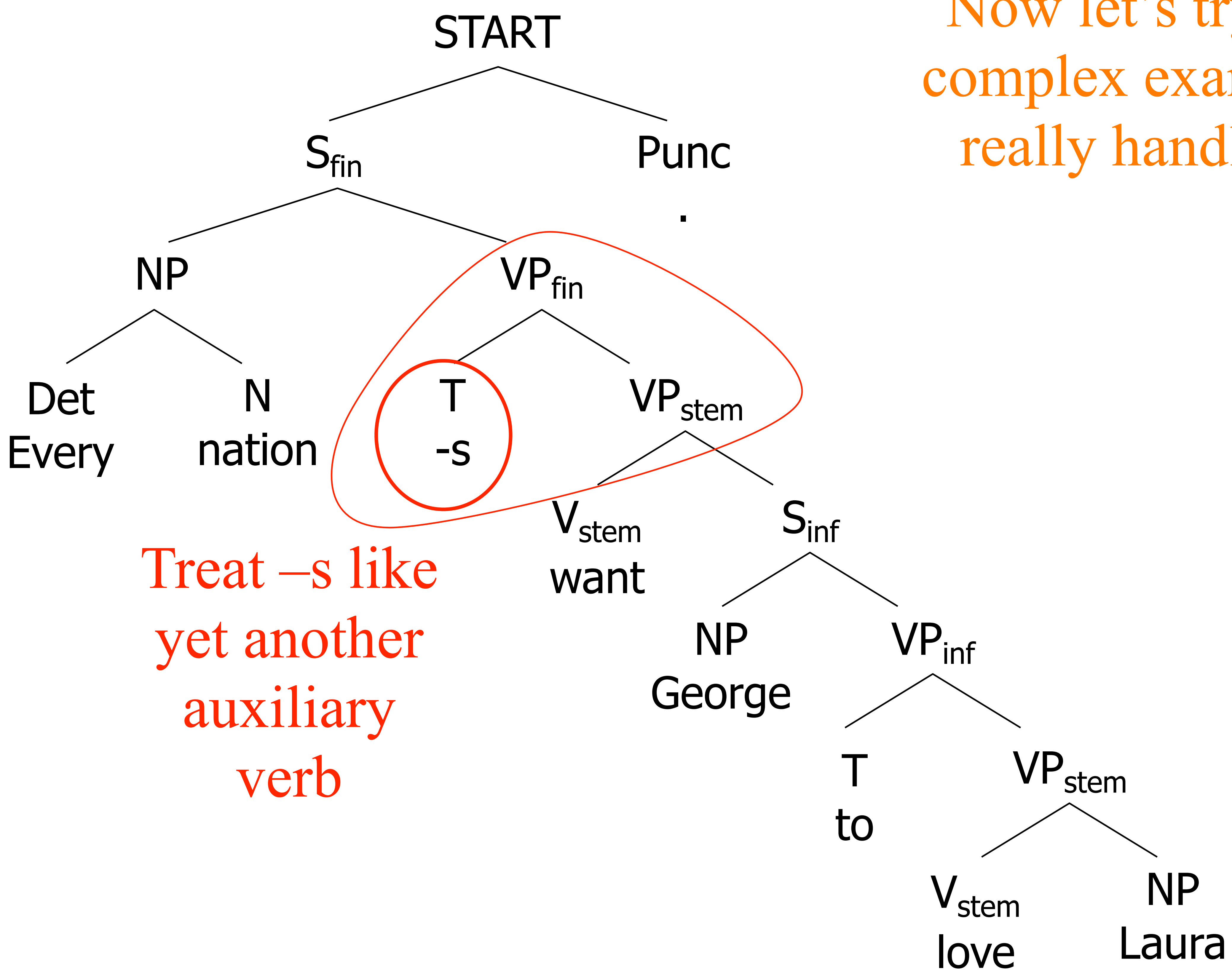


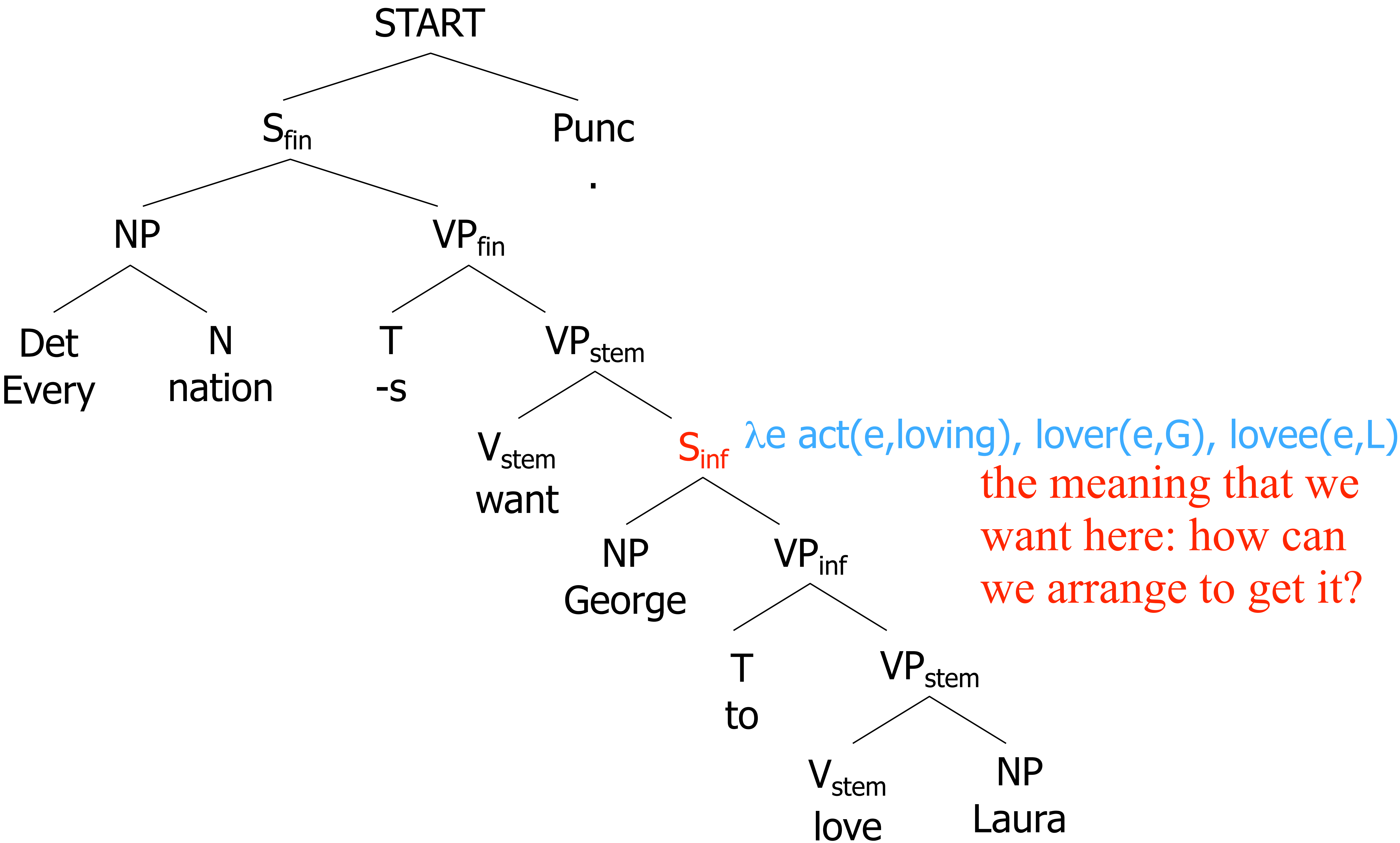
Now let’s try a more complex example, and really handle tense.

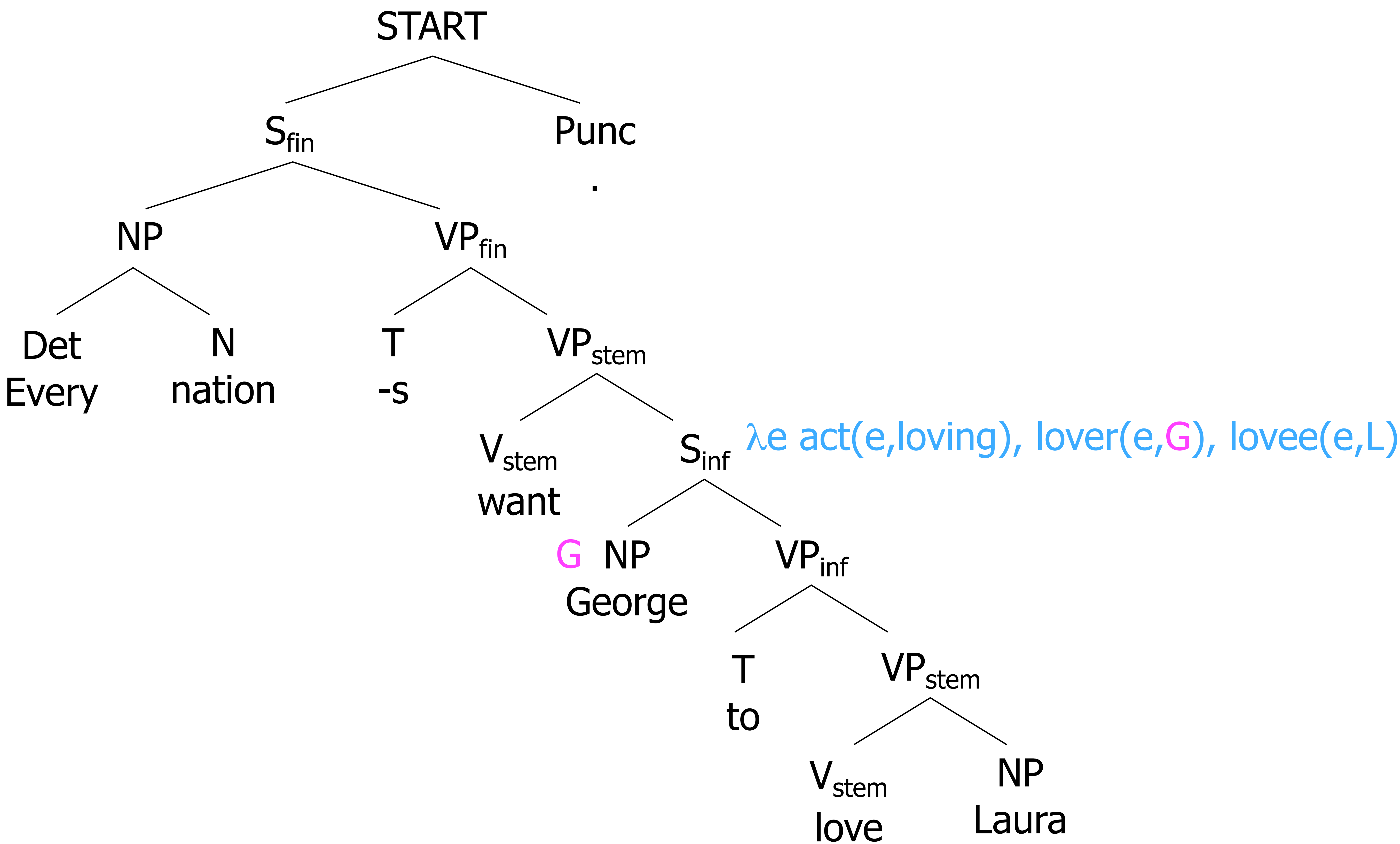
Now let's try a more complex example, and really handle tense.

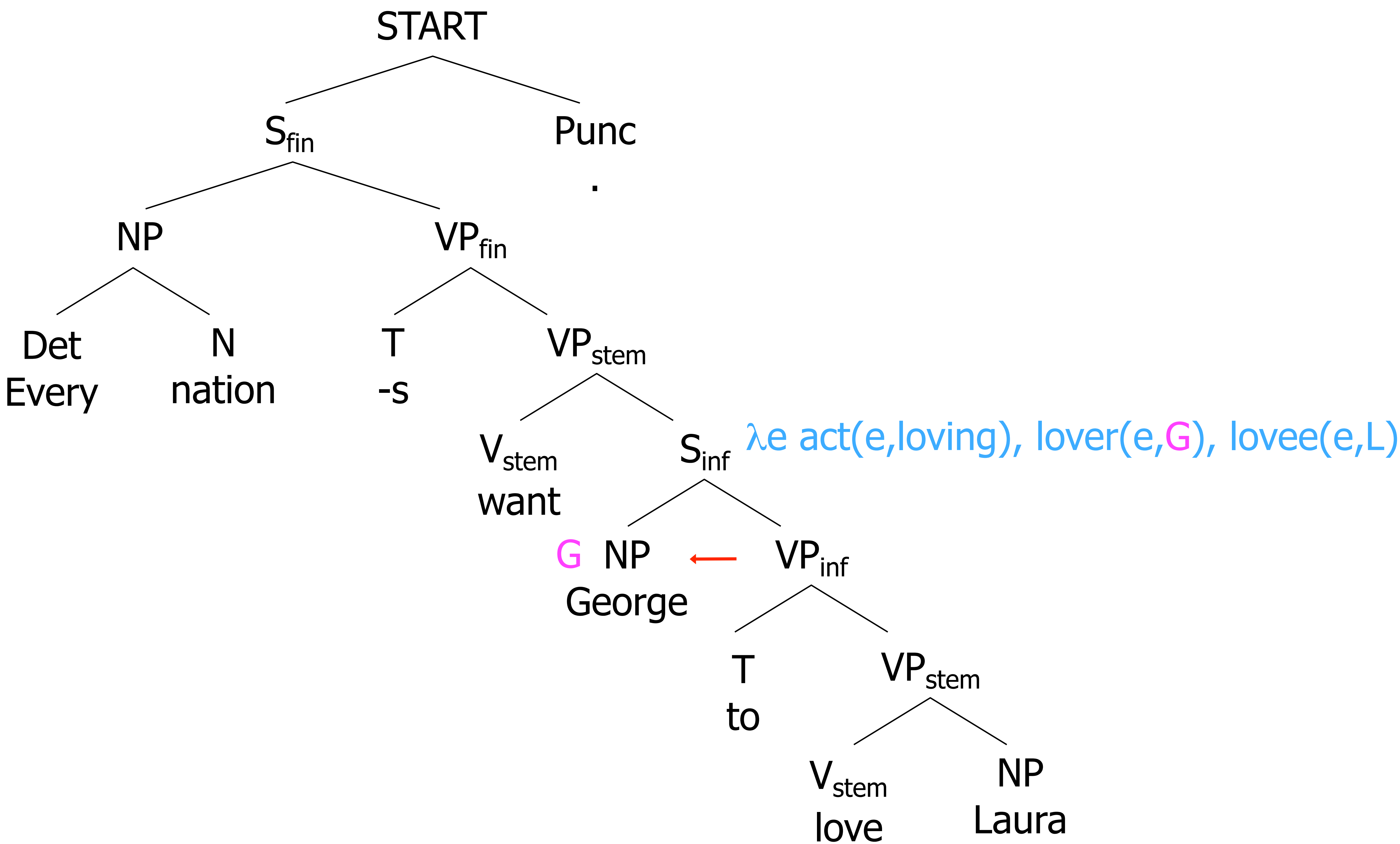


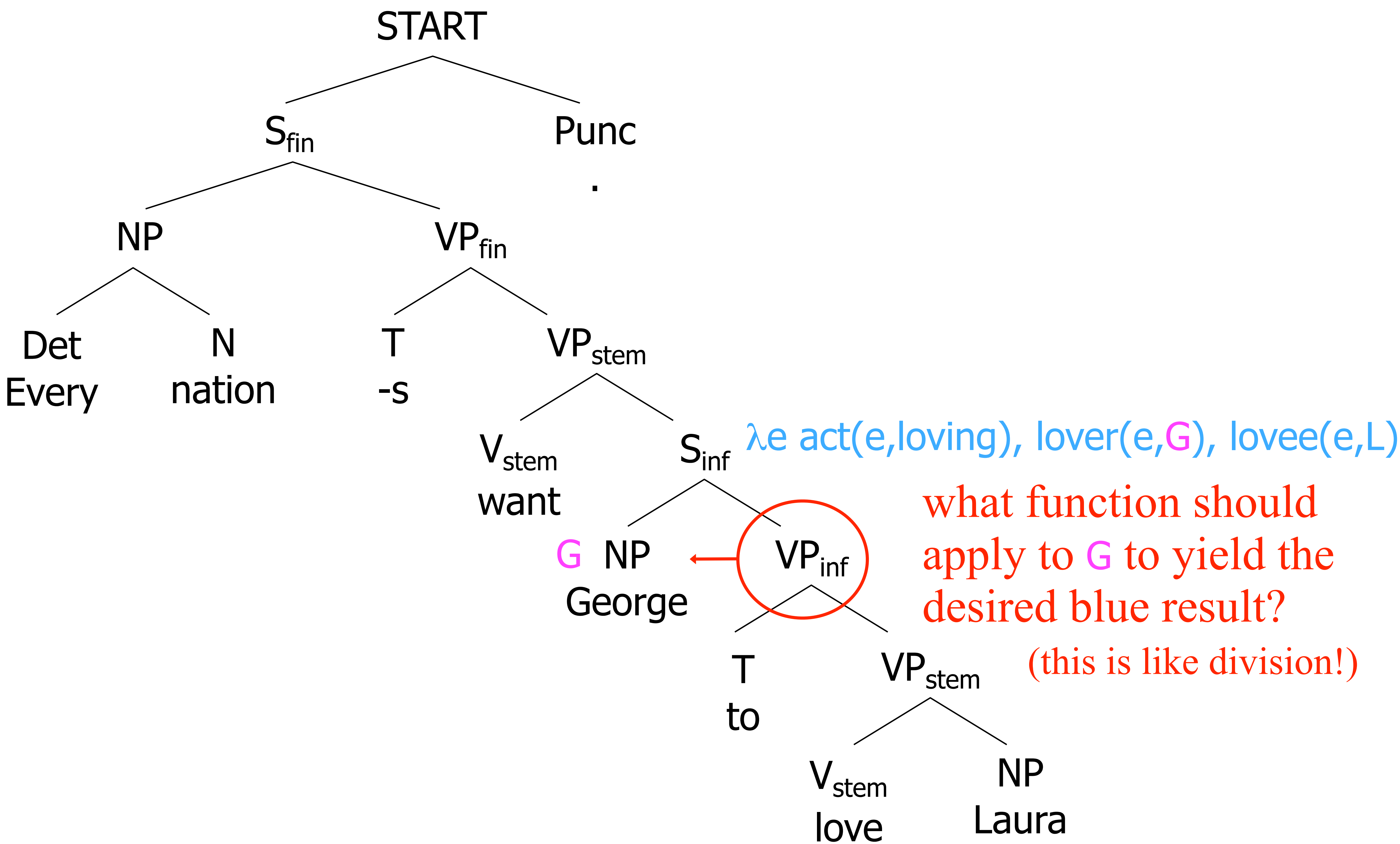
Now let's try a more complex example, and really handle tense.

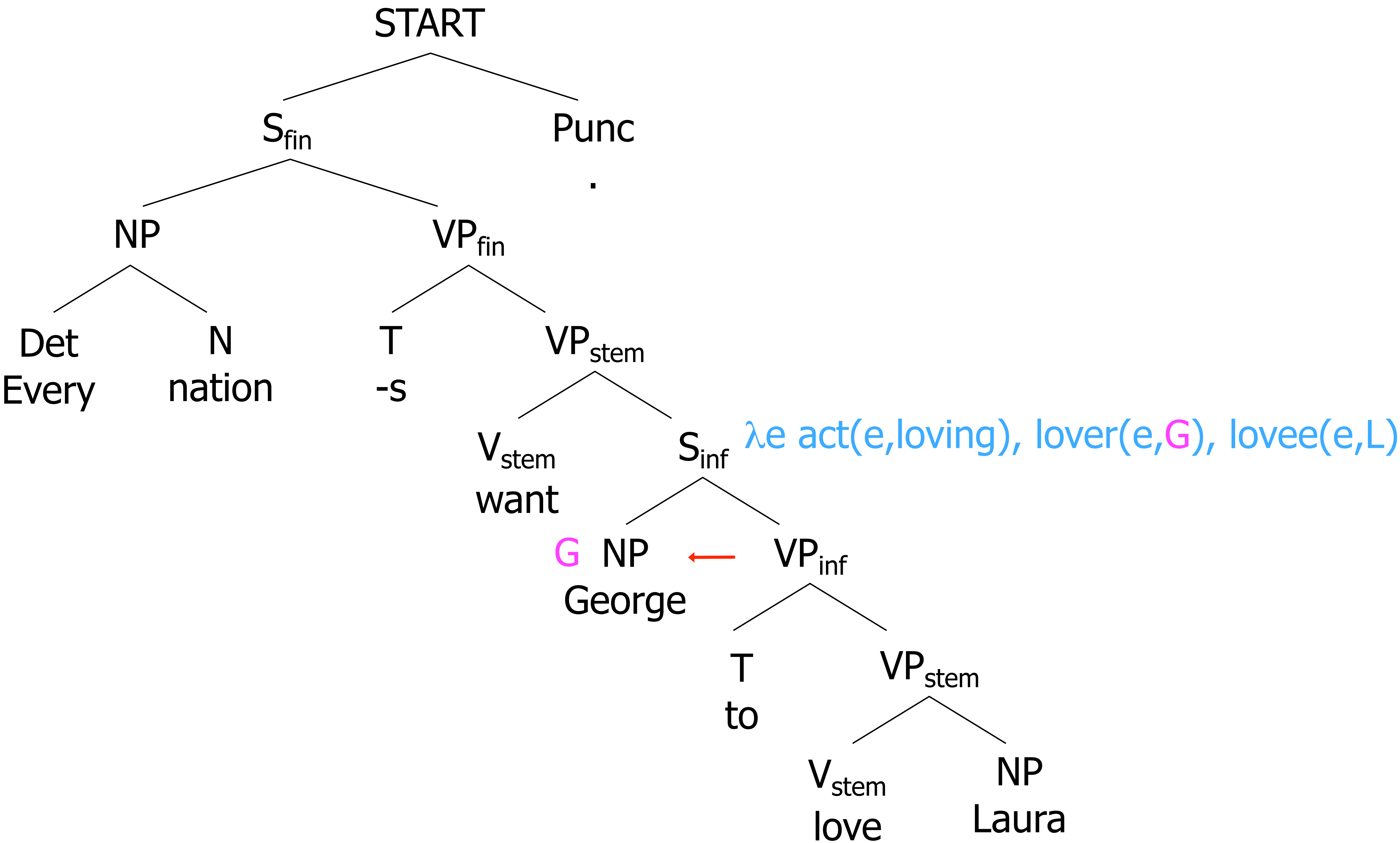


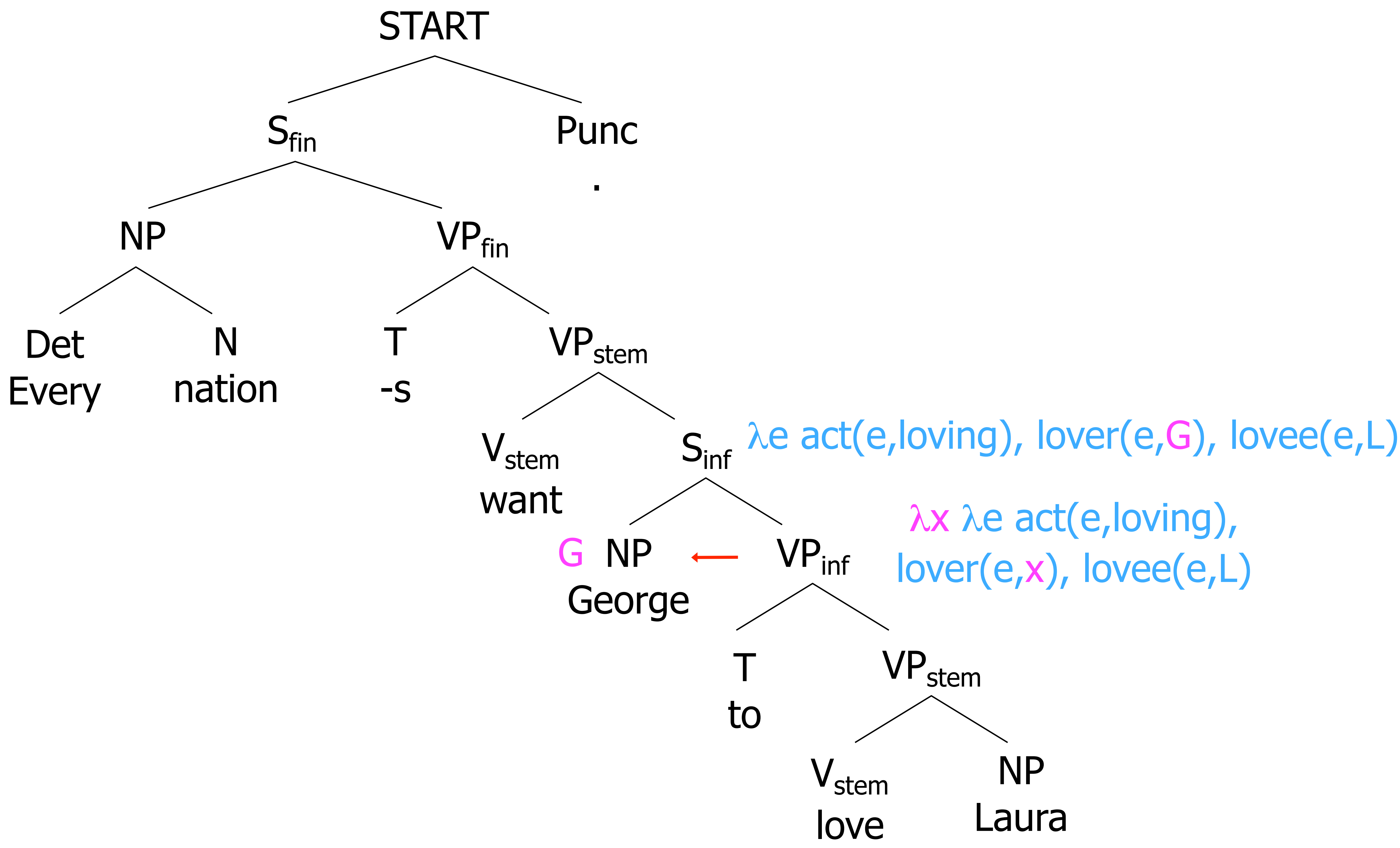

START
Sfin
Punc
.
NP
VPfin
Det
Every
N
nation
T
-s
VPstem
Vstem
want
Sinf
λe act(e,loving), lover(e,G), lovee(e,L)
the meaning that we want here: how can we arrange to get it?
NP
George
VPinf
T
to
VPstem
Vstem
love
NP
Laura

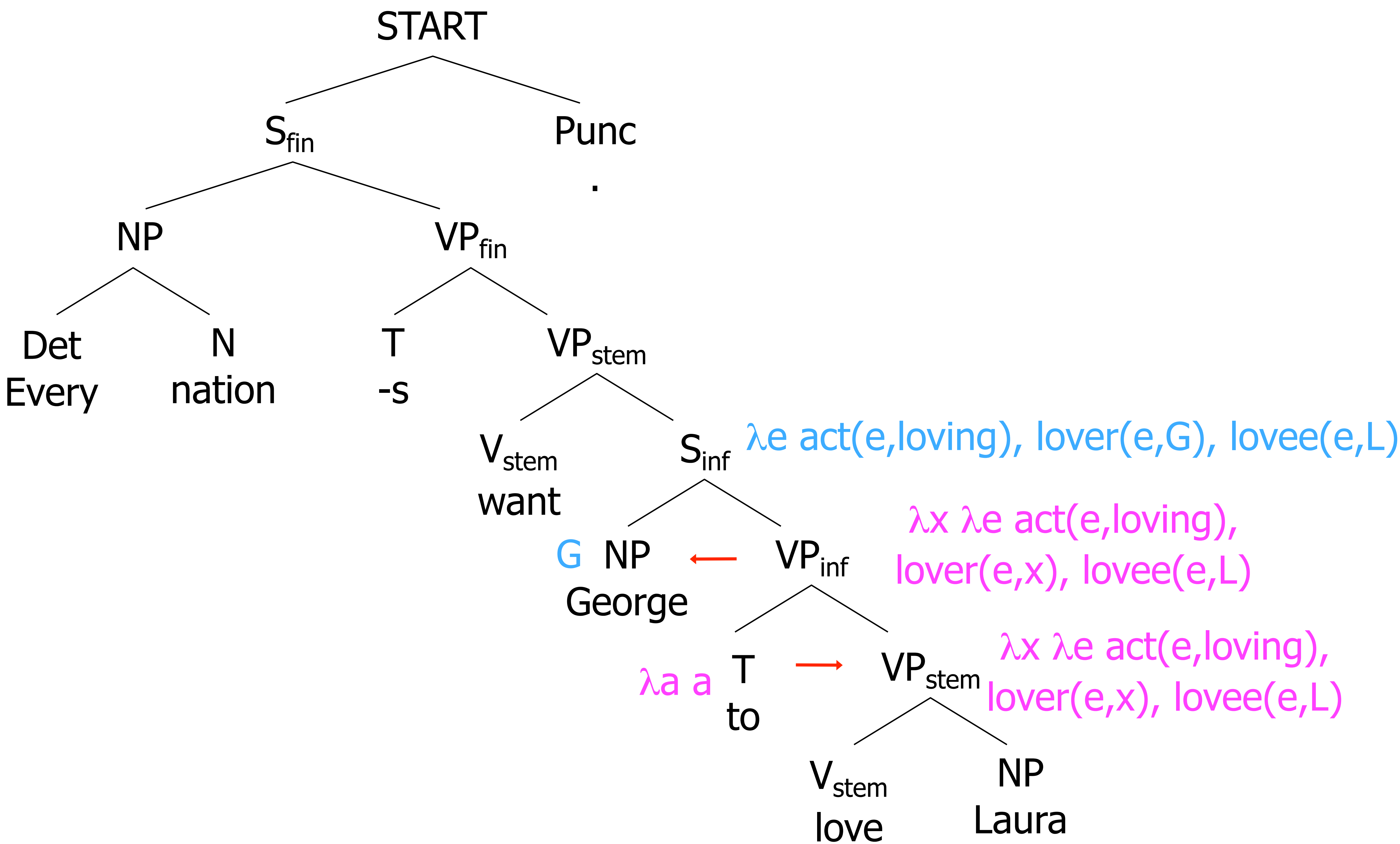

START
S_fin
Punc
.
NP
VP_fin
Det
Every
N
nation
T
-s
VP_stem
V_stem
want
S_inf
λe act(e,loving), lover(e,G), lovee(e,L)
G
NP
George
VP_inf
T
to
VP_stem
V_stem
love
NP
Laura

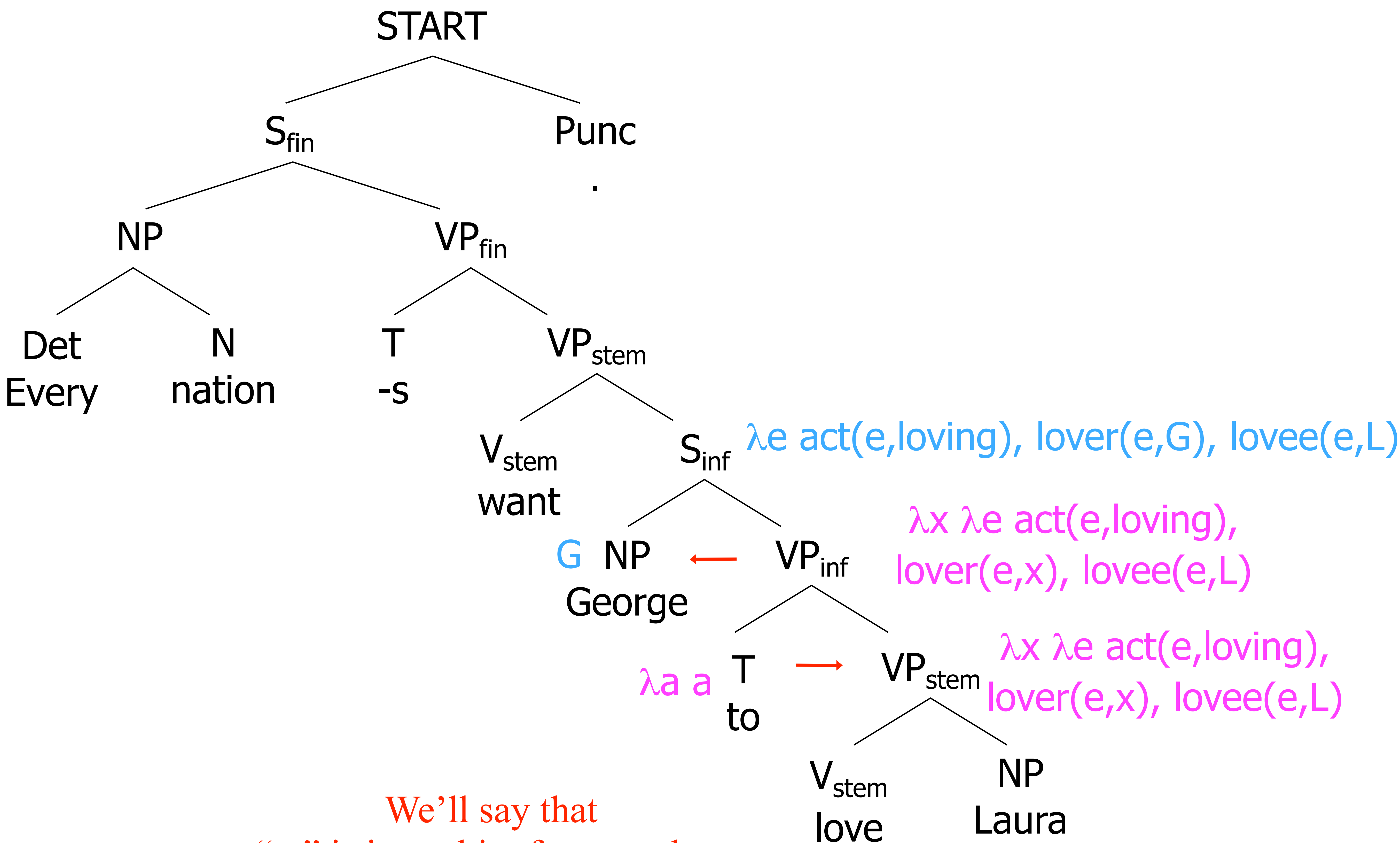

START
S fin
Punc
.
NP
VP fin
Det
Every
N
nation
T
-s
VP stem
V stem
want
S inf
λe act(e,loving), lover(e,G), lovee(e,L)
G NP
George
VP inf
T
to
VP stem
V stem
love
NP
Laura


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
George
VP inf
T
to
VP stem
V stem
love
NP
Laura
what function should apply to G to yield the desired blue result?
(this is like division!)

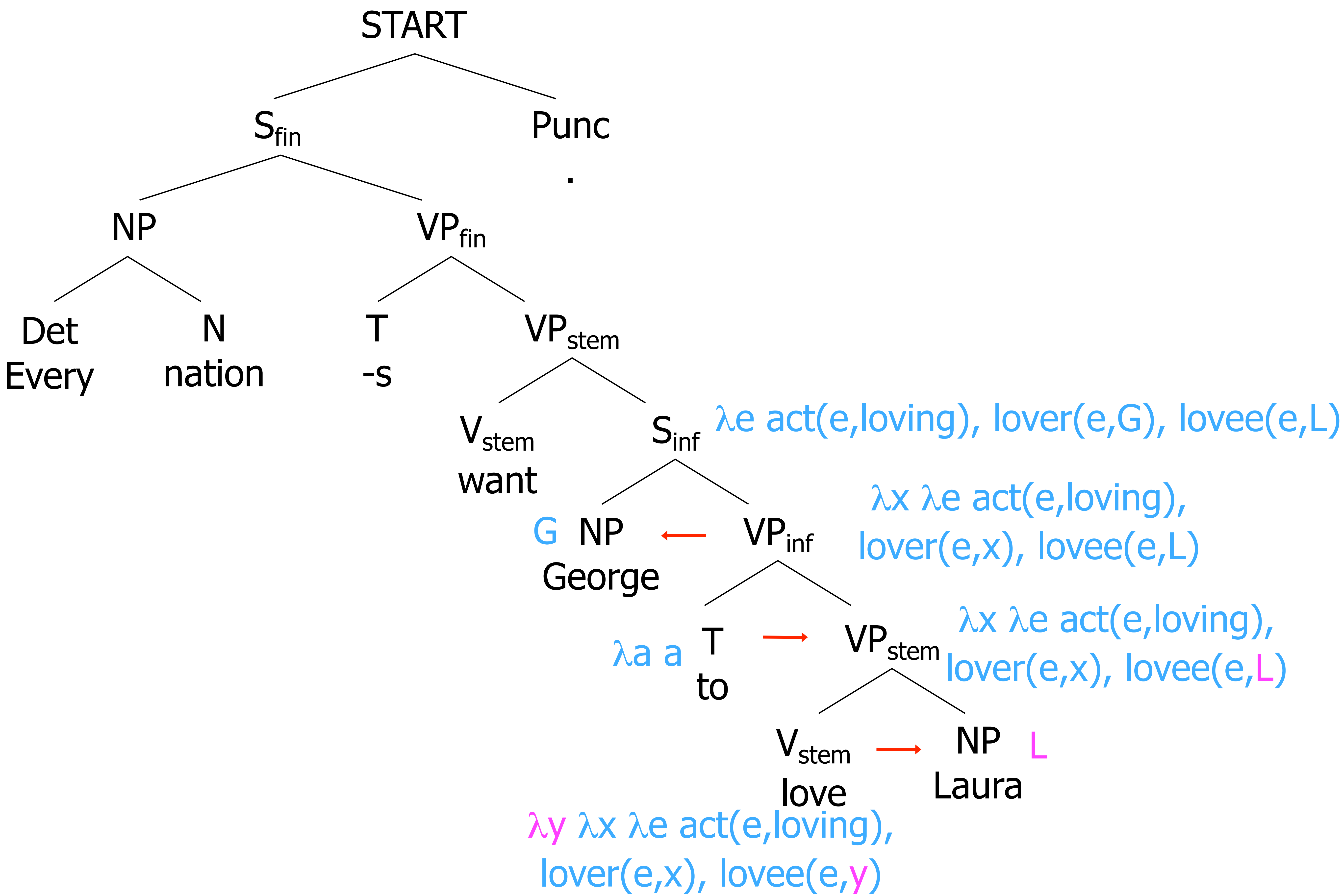

START
Sfin
Punc
.
NP
VPfin
Det
Every
N
nation
T
-s
VPstem
Vstem
want
Sinf
λe act(e,loving), lover(e,G), lovee(e,L)
G
NP
George
VPinf
T
to
VPstem
Vstem
love
NP
Laura

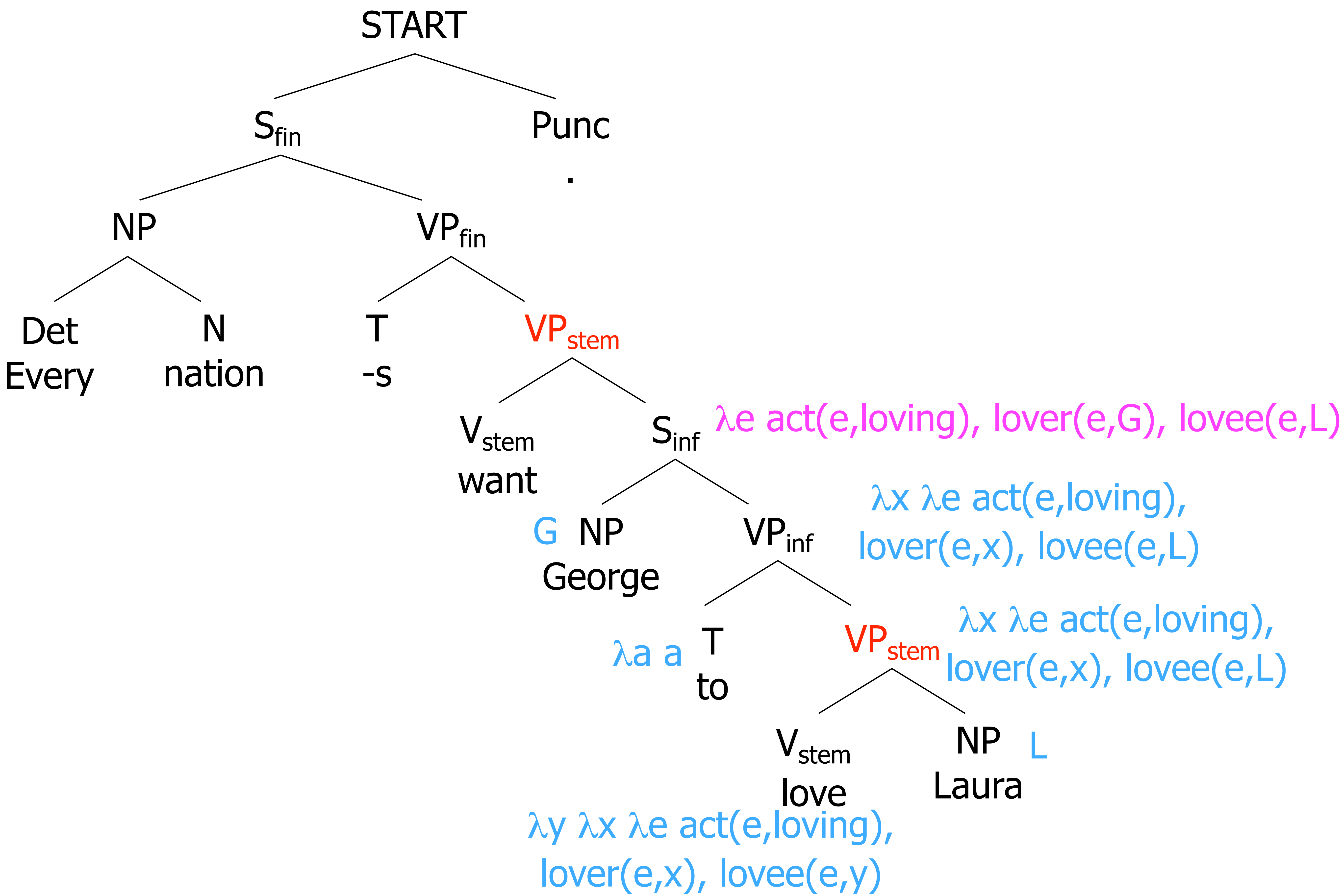

START
S_{fin}
Punc
.
NP
VP_{fin}
Det
Every
N
nation
T
-s
VP_{stem}
V_{stem}
want
S_{inf}
λe act(e,loving), lover(e,G), lovee(e,L)
G NP
George
VP_{inf}
λx λe act(e,loving), lover(e,x), lovee(e,L)
T
to
VP_{stem}
V_{stem}
love
NP
Laura

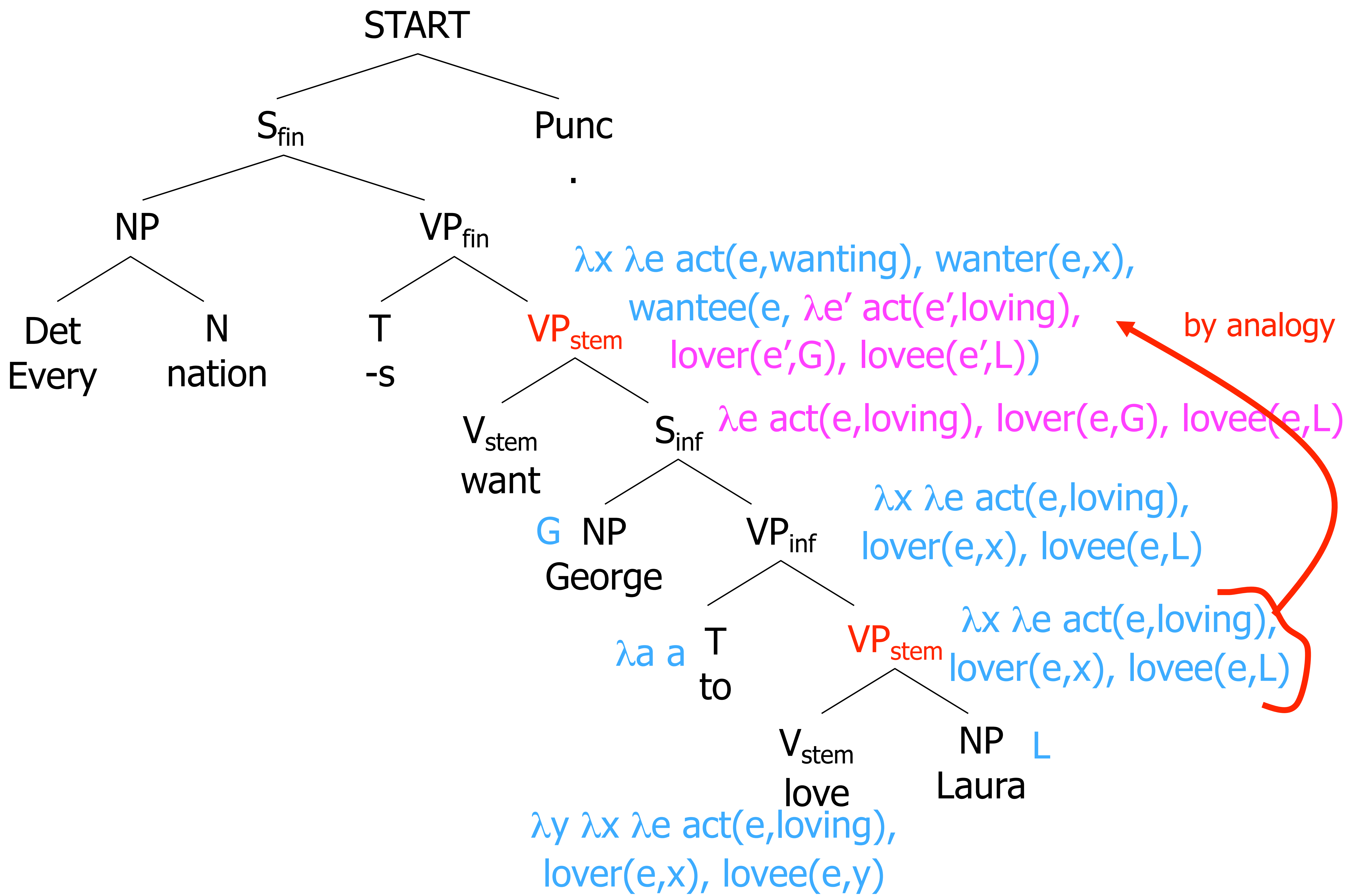

START
S fin
Punc
.
NP
VP fin
Det
Every
N
nation
T
-s
VP stem
V stem
want
S inf
λe act(e,loving), lover(e,G), lovee(e,L)
G
NP
George
VP inf
λx λe act(e,loving),
lover(e,x), lovee(e,L)
λa a
T
to
VP stem
λx λe act(e,loving),
lover(e,x), lovee(e,L)
V stem
love
NP
Laura

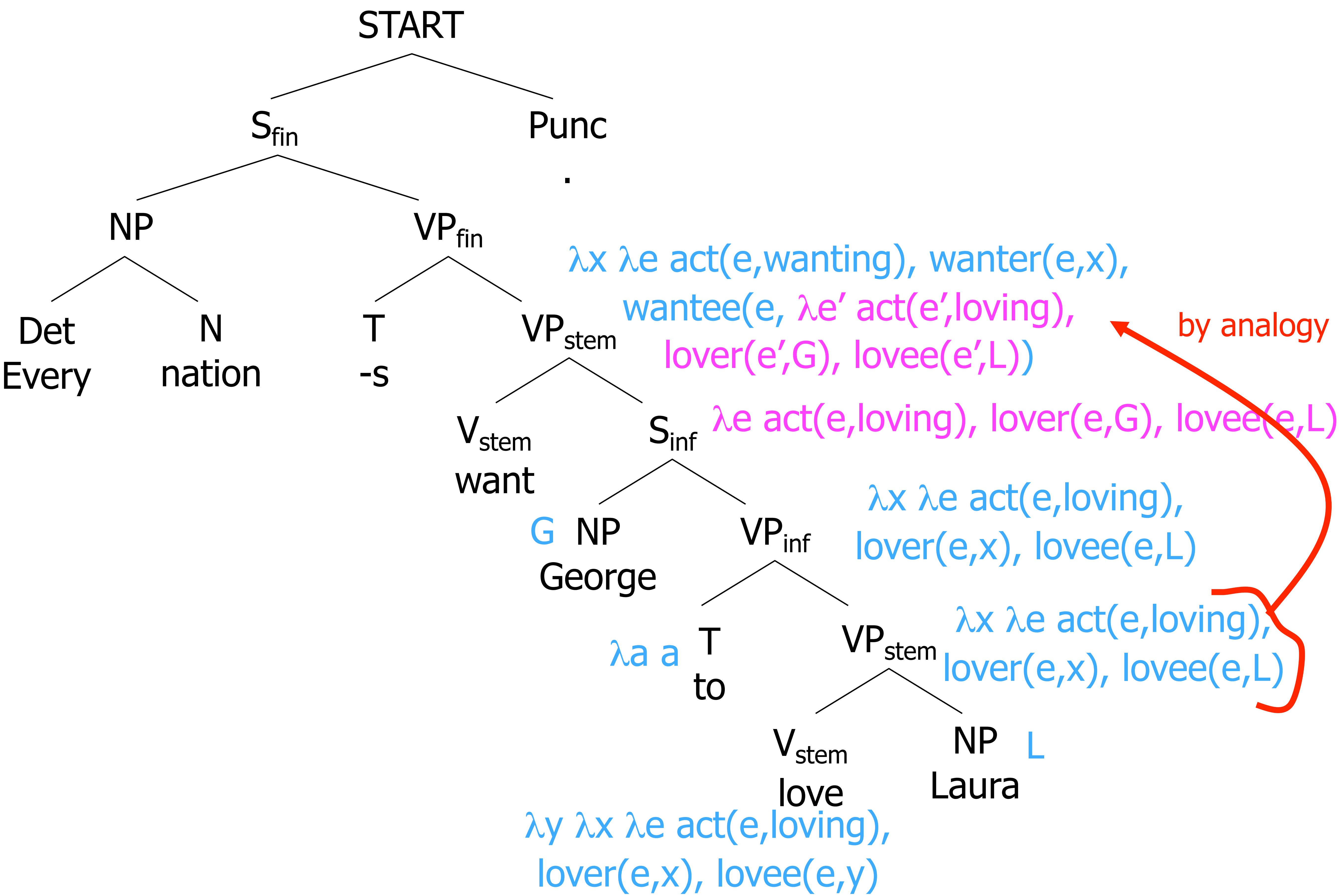


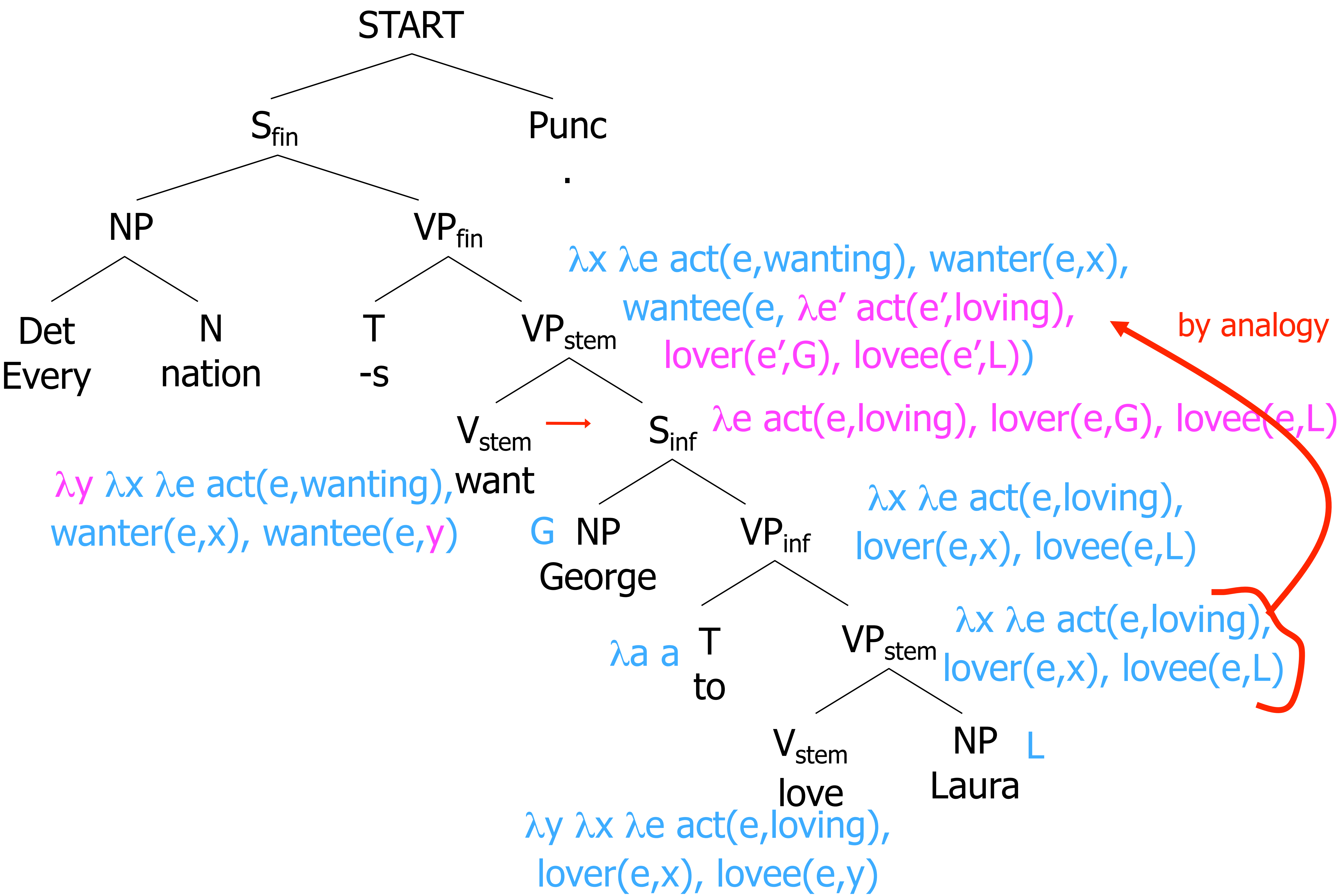
We'll say that
"to" is just a bit of syntax that
changes a VP_{stem} to a VP_{inf}
with the same meaning.

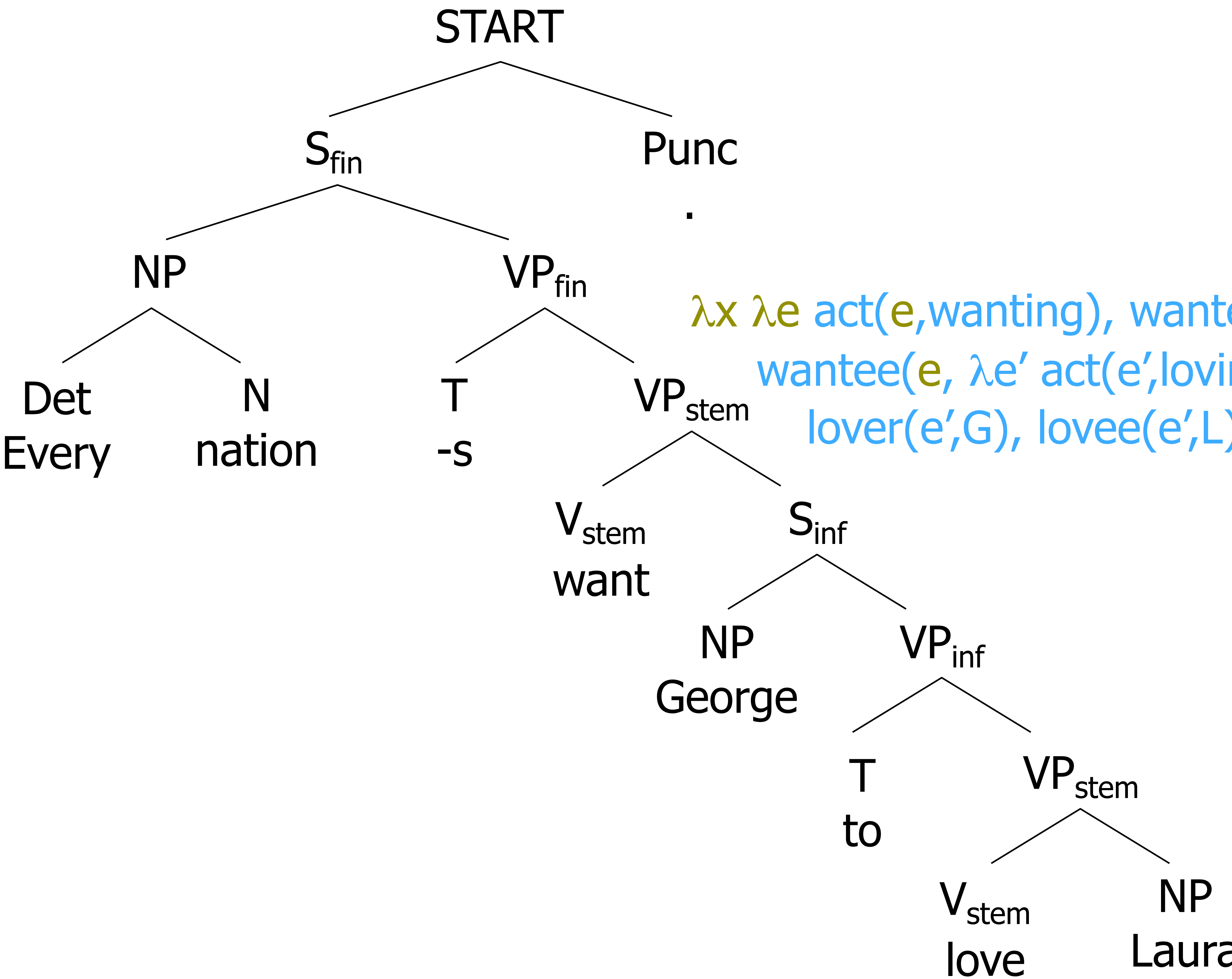

START
S fin
Punc
.
NP
VP fin
Det
Every
N
nation
T
-s
VP stem
V stem
want
S inf
λe act(e,loving), lover(e,G), lovee(e,L)
G NP
George
VP inf
λx λe act(e,loving), lover(e,x), lovee(e,L)
λa a T
to
VP stem
λx λe act(e,loving), lover(e,x), lovee(e,L)
V stem
love
NP L
Laura
λy λx λe act(e,loving), lover(e,x),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


START
S fin
Punc
.
NP
VP fin
Det
Every
N
nation
T
-s
VP stem
V stem
want
S inf
λe act(e,loving), lover(e,G), lovee(e,L)
λx λe act(e,loving),
lover(e,x), lovee(e,L)
G
NP
George
VP inf
λa a
T
to
VP stem
λx λe act(e,loving),
lover(e,x), lovee(e,L)
V stem
love
NP
Laura
L
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))
by analogy
V_stem
want
S_inf
λe act(e,loving), lover(e,G), lovee(e,L)
G
NP
George
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
Laura
L
λy λx λe act(e,loving),
lover(e,x), lovee(e,y)


START
S fin
Punc
.
NP
VP fin
λx λe act(e,wanting), wanter(e,x),
wantee(e, λe′ act(e′,loving),
lover(e′,G), lovee(e′,L))
by analogy
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
George
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
Laura
λy λx λe act(e,loving),
lover(e,x), lovee(e,y)

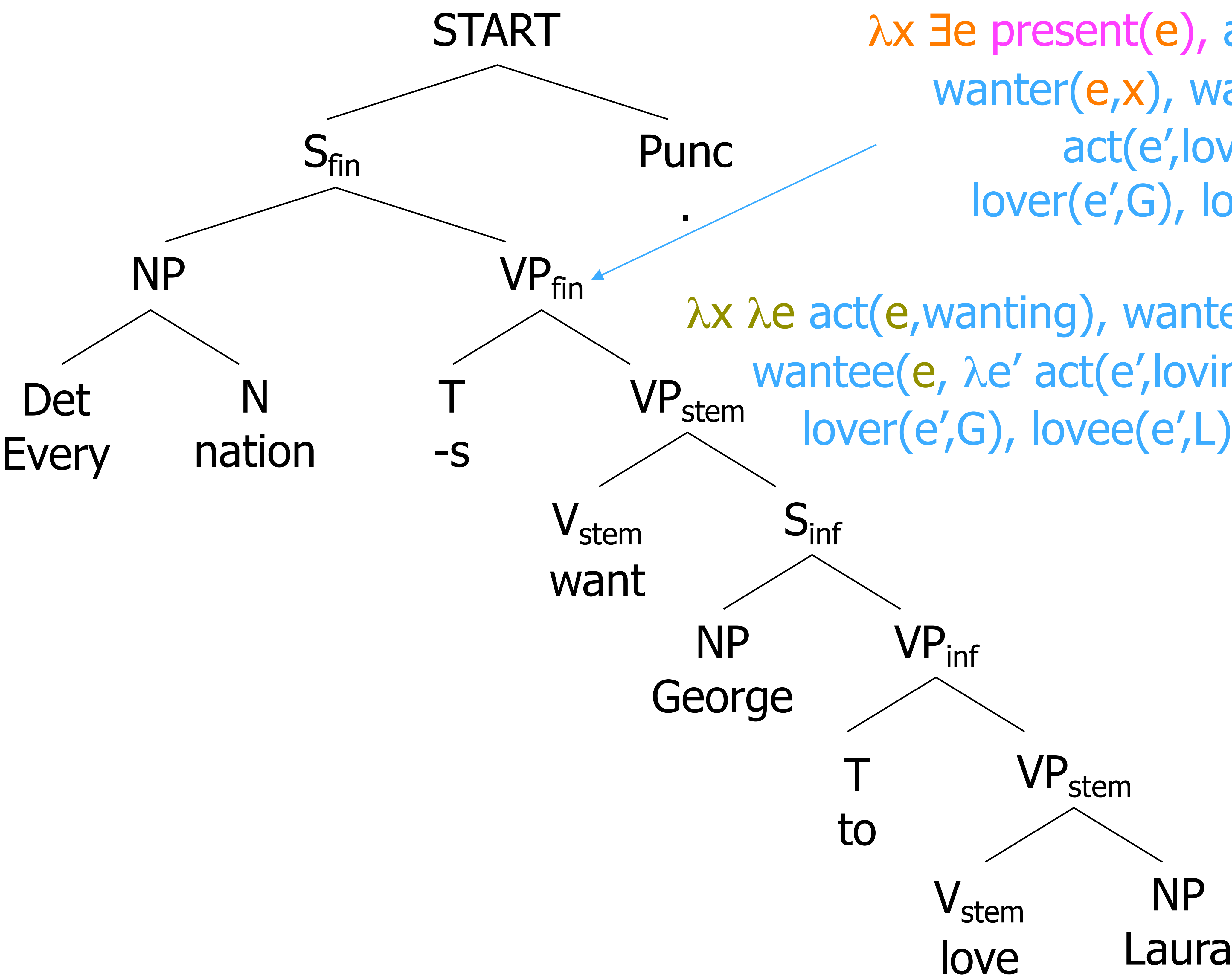

START
S fin
Punc
.
NP
VP fin
Det
Every
N
nation
T
-s
VP stem
λx λe act(e,wanting), wanter(e,x),
wantee(e, λe′ act(e′,loving),
lover(e′,G), lovee(e′,L))
by analogy
V stem
want
S inf
λe act(e,loving), lover(e,G), lovee(e,L)
λy λx λe act(e,wanting),
wanter(e,x), wantee(e,y)
G
NP
George
VP inf
λx λe act(e,loving),
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
Laura
L
λy λx λe act(e,loving),
lover(e,x), lovee(e,y)

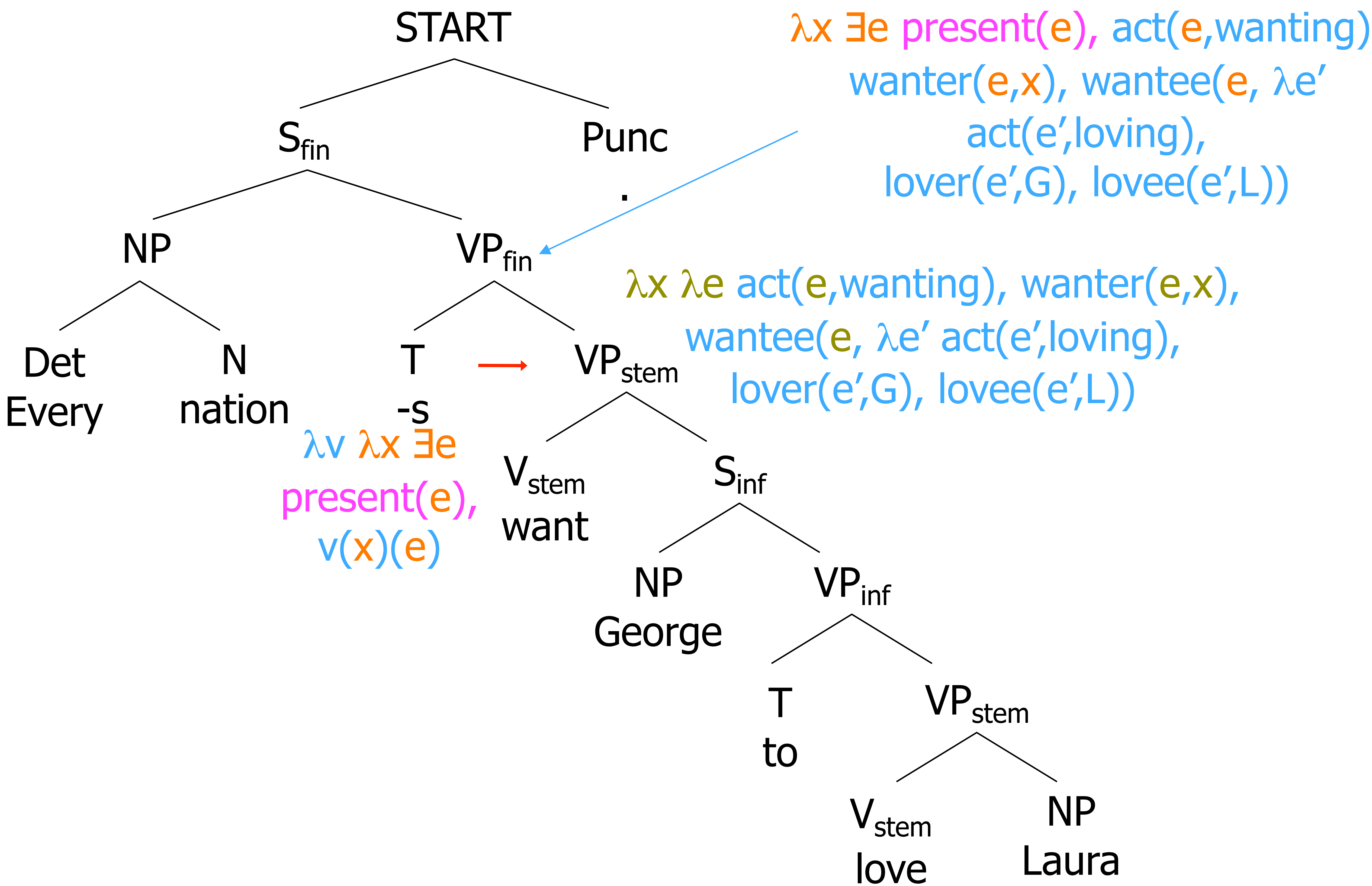


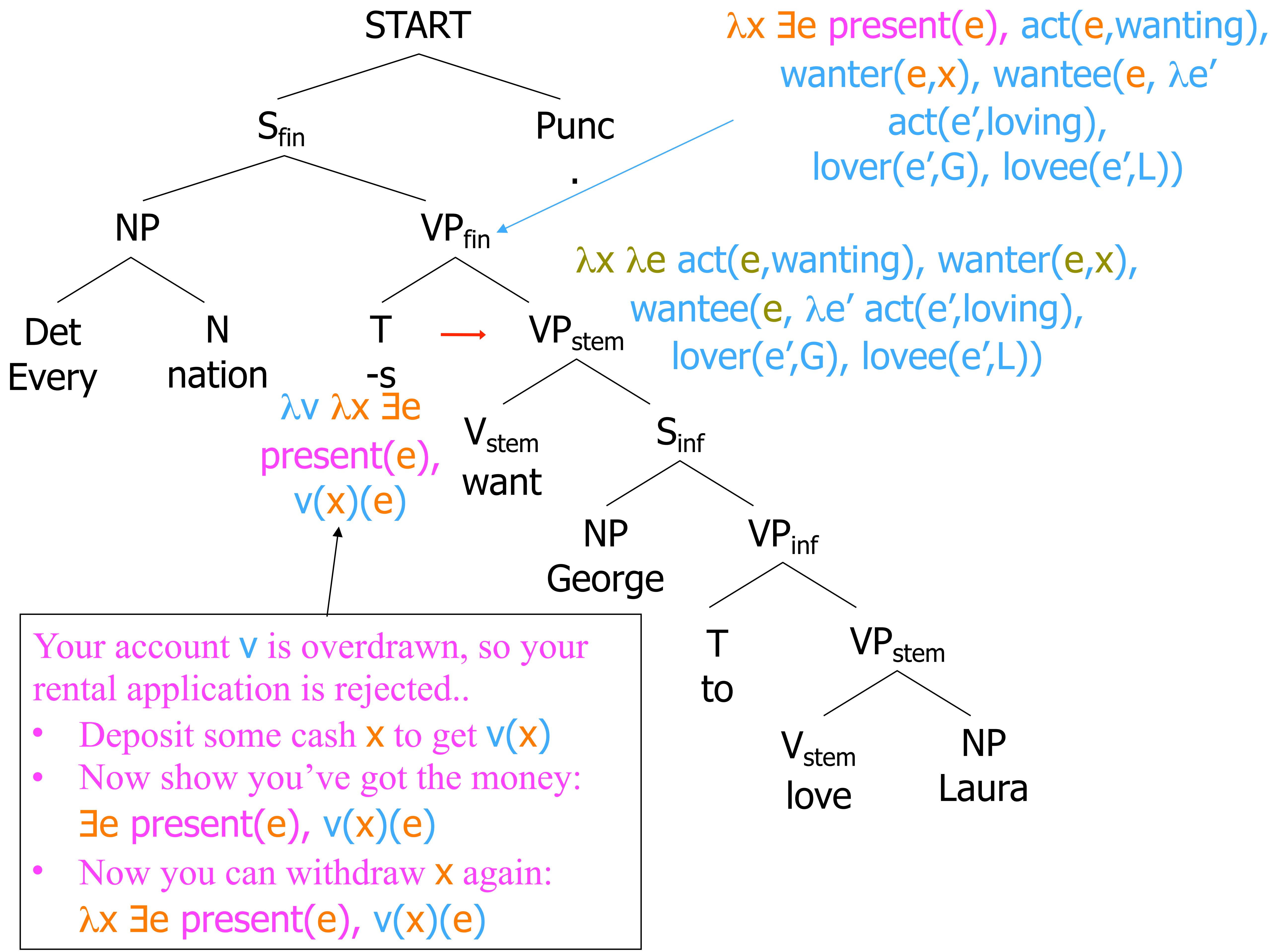
$\lambda x\ \lambda e$ act(e,wanting), wanter(e,x), wantee(e, $\lambda e'$ act(e',loving), lover(e',G), lovee(e',L))

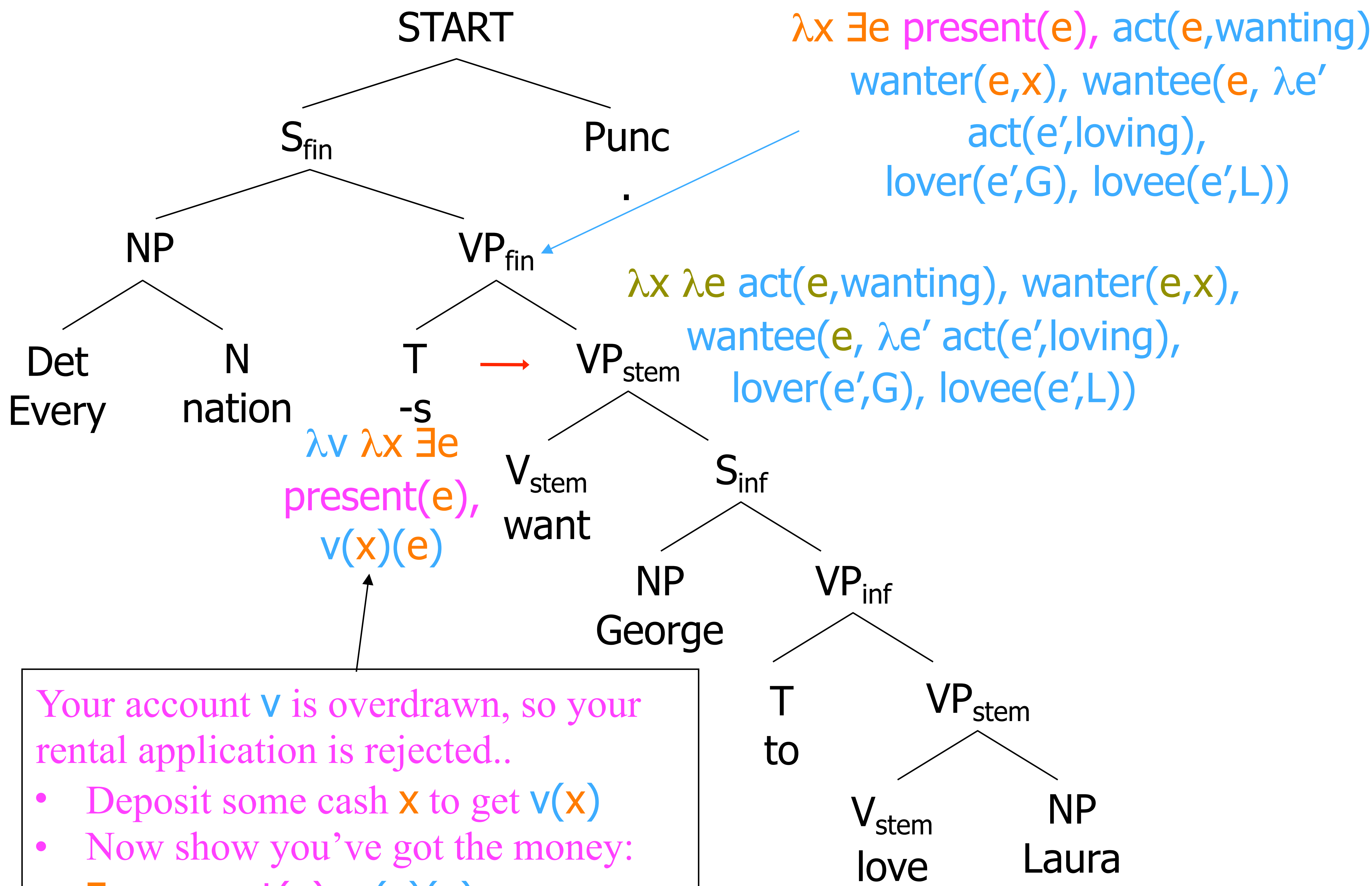
$\lambda x\ \exists e\ present(e), act(e,wanting), wanter(e,x), wantee(e, \lambda e'\ act(e',loving), lover(e',G), lovee(e',L))$

$\lambda x\ \lambda e\ act(e,wanting), wanter(e,x), wantee(e, \lambda e'\ act(e',loving), lover(e',G), lovee(e',L))$




START
S_fin
Punc
.
NP
VP_fin
Det
Every
N
nation
T
-s
VP_stem
λv λx ∃e
present(e),
v(x)(e)
V_stem
want
S_inf
NP
George
VP_inf
T
to
VP_stem
V_stem
love
NP
Laura
λx ∃e present(e), act(e,wanting),
wanter(e,x), wantee(e, λe′
act(e′,loving),
lover(e′,G), lovee(e′,L))
λx λe act(e,wanting), wanter(e,x),
wantee(e, λe′ act(e′,loving),
lover(e′,G), lovee(e′,L))

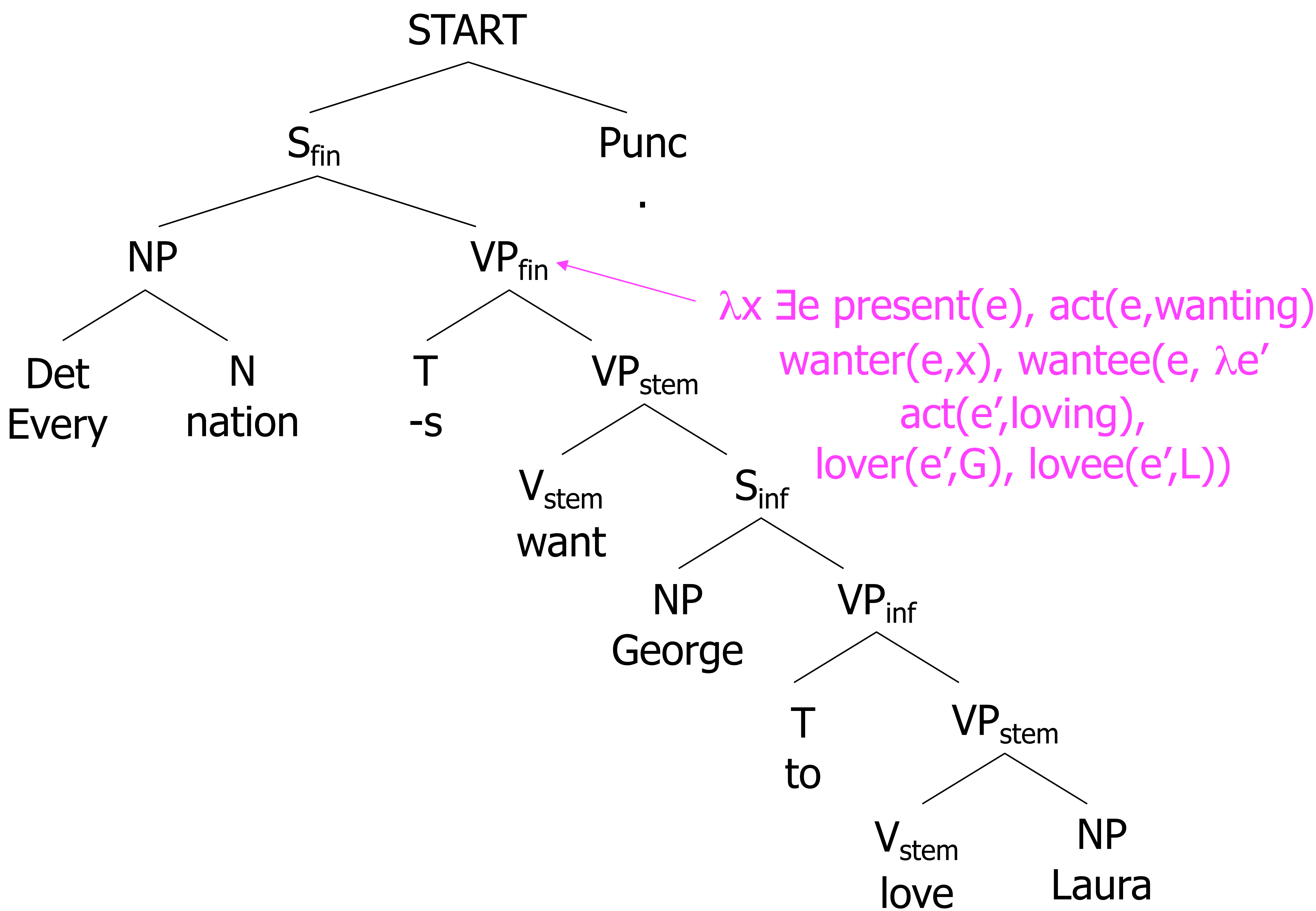

START
S fin
Punc
.
NP
VP fin
Det
Every
N
nation
T
-s
VP stem
V stem
want
S inf
NP
George
VP inf
T
to
VP stem
V stem
love
NP
Laura
λx ∃e present(e), act(e,wanting), wanter(e,x), wantee(e, λe′ act(e′,loving), lover(e′,G), lovee(e′,L))
λx λe act(e,wanting), wanter(e,x), wantee(e, λe′ act(e′,loving), lover(e′,G), lovee(e′,L))
λv λx ∃e present(e), v(x)(e)
Your account v is overdrawn, so your rental application is rejected..
Deposit some cash x to get v(x)
Now show you’ve got the money: ∃e present(e), v(x)(e)
Now you can withdraw x again: λx ∃e present(e), v(x)(e)



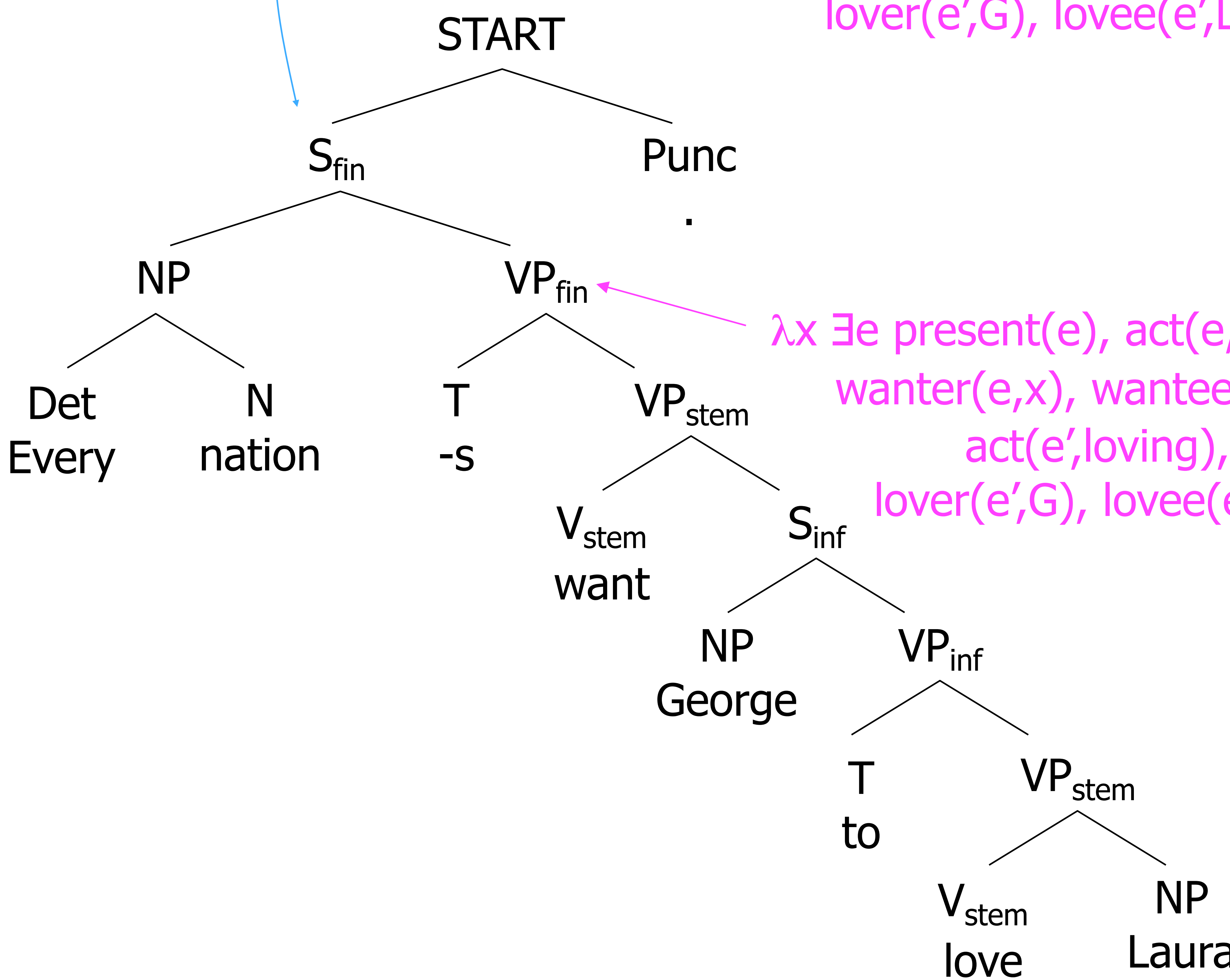
Your account v is overdrawn, so your rental application is rejected..

- Deposit some cash x to get v(x)
- Now show you've got the money: ∃e present(e), v(x)(e)
- Now you can withdraw x again: λx ∃e present(e), v(x)(e)

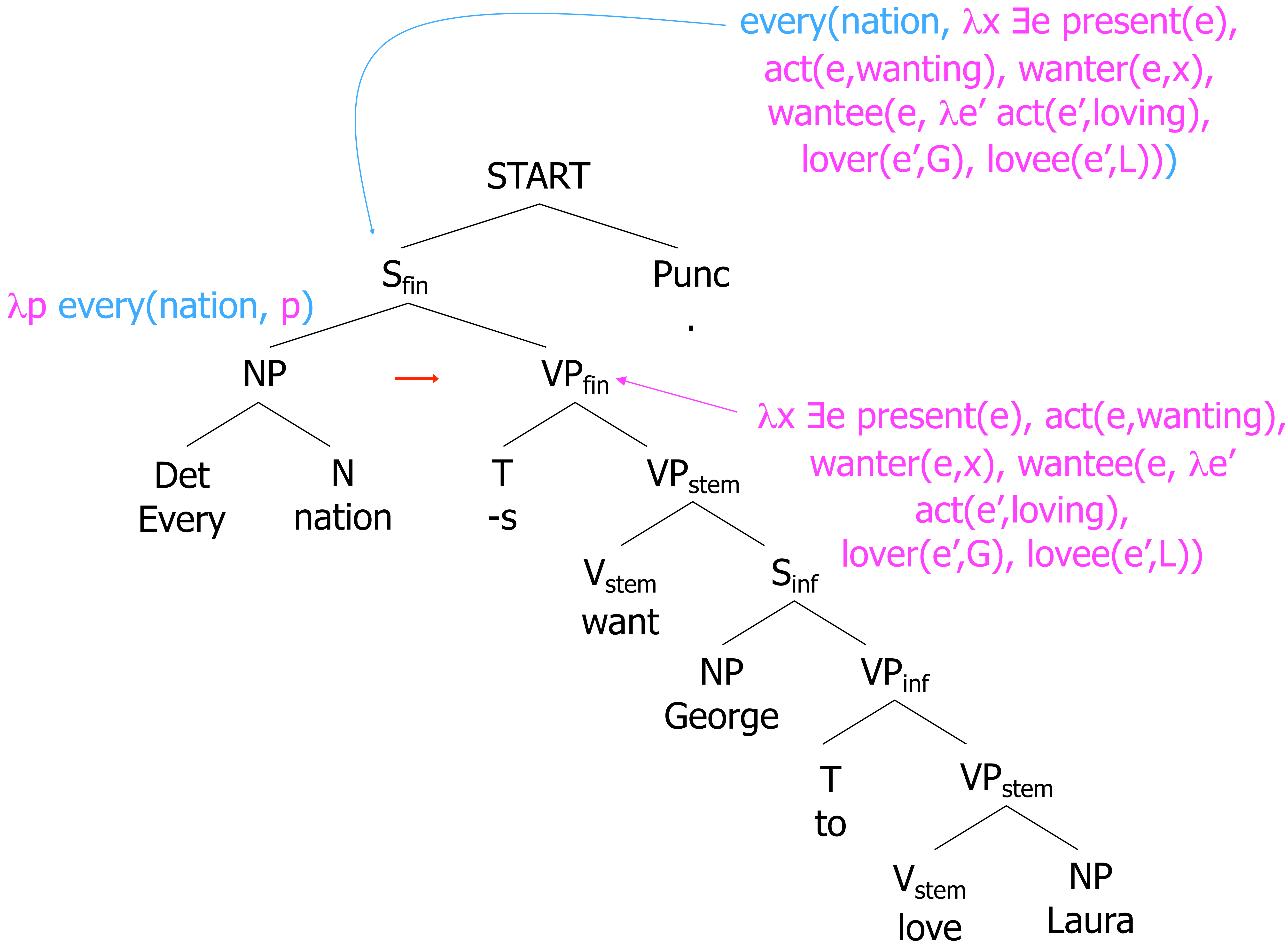
Better analogy: How would you modify the <u>second</u> object on a stack (λx,λe,act…)?


START
S fin
Punc
.
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
George
VP inf
T
to
VP stem
V stem
love
NP
Laura

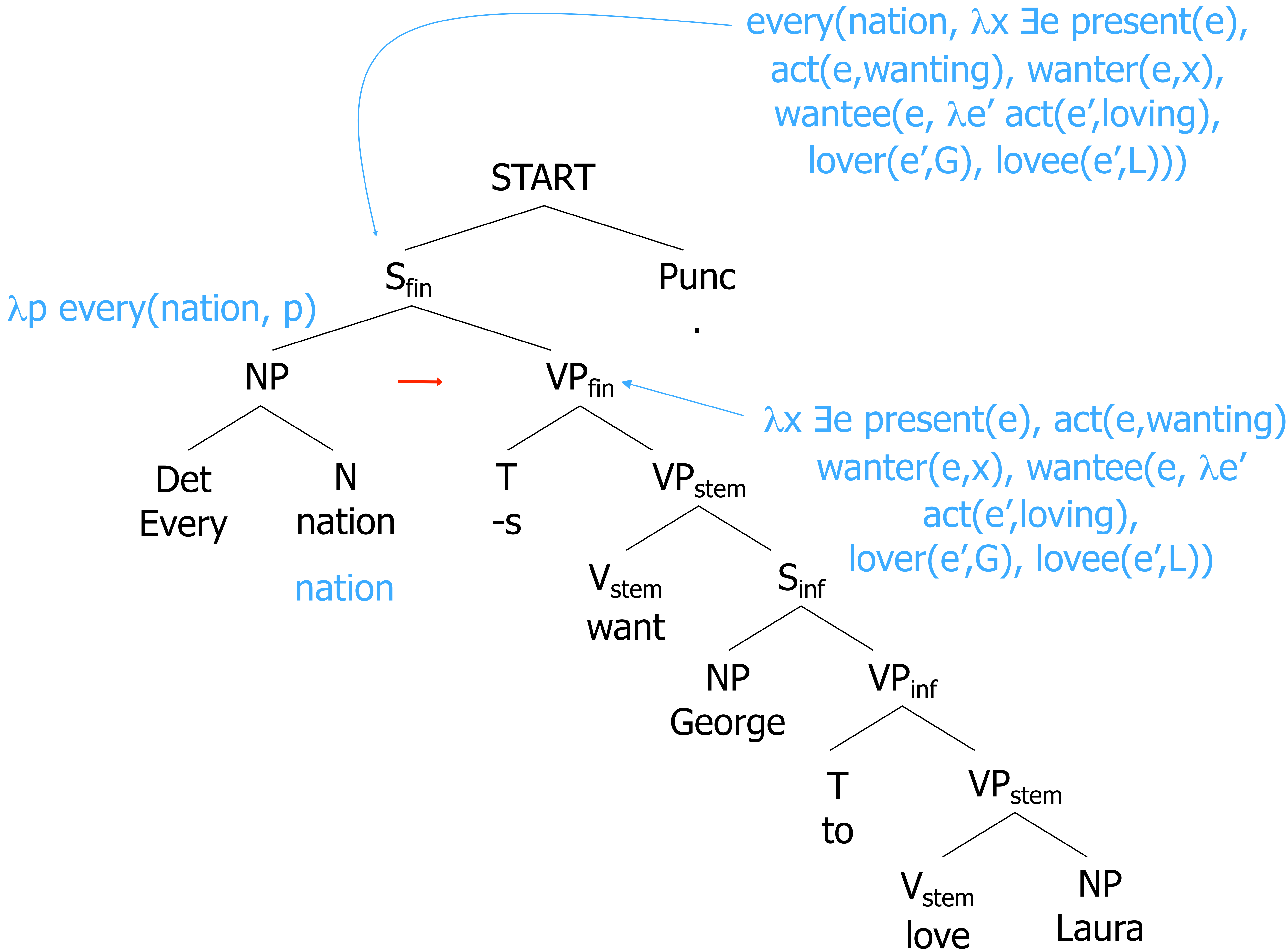
every(nation, λx ∃e present(e), act(e,wanting), wanter(e,x), wantee(e, λe′ act(e′,loving), lover(e′,G), lovee(e′,L)))

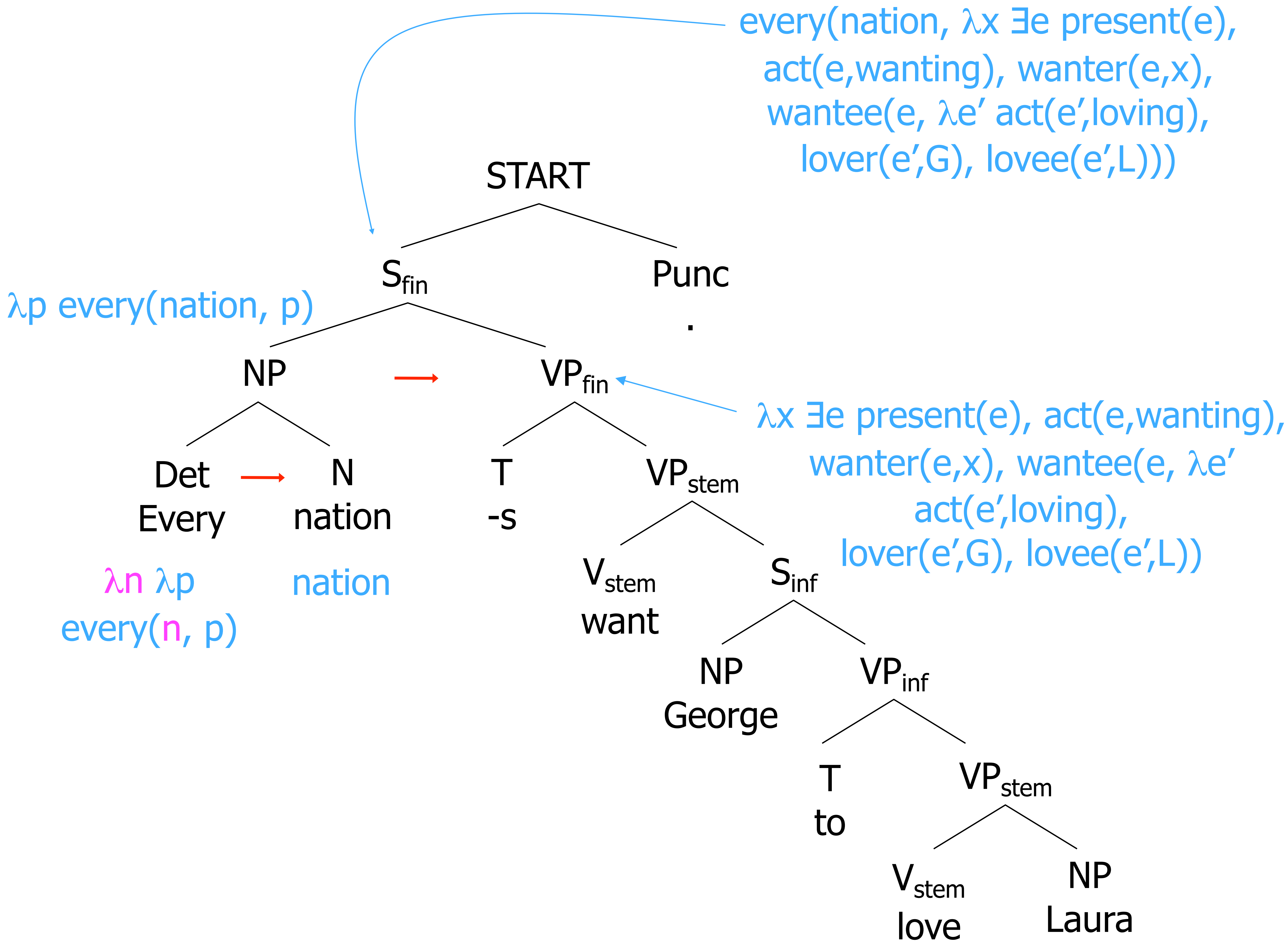


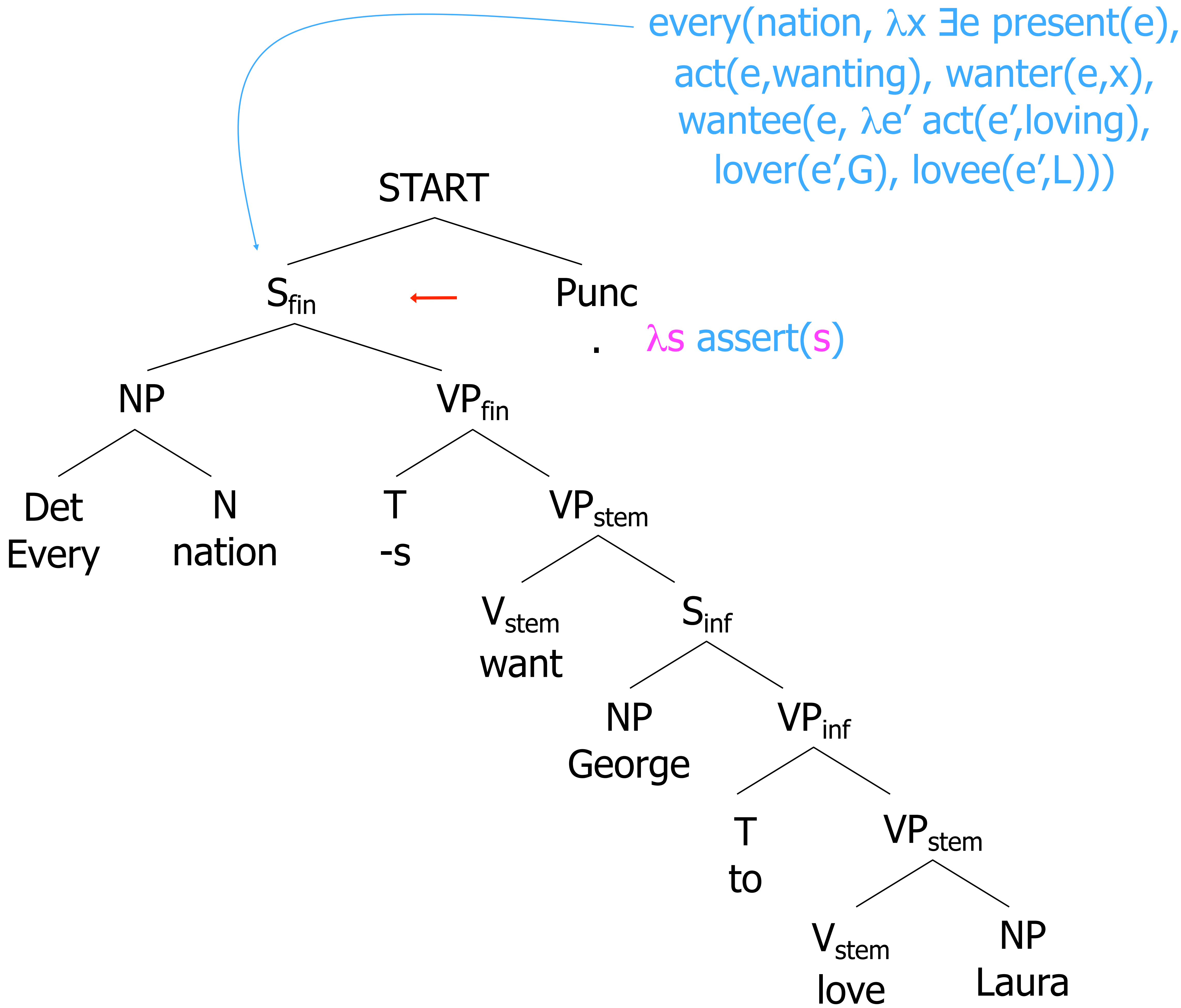
λx ∃e present(e), act(e,wanting), wanter(e,x), wantee(e, λe′ act(e′,loving), lover(e′,G), lovee(e′,L))



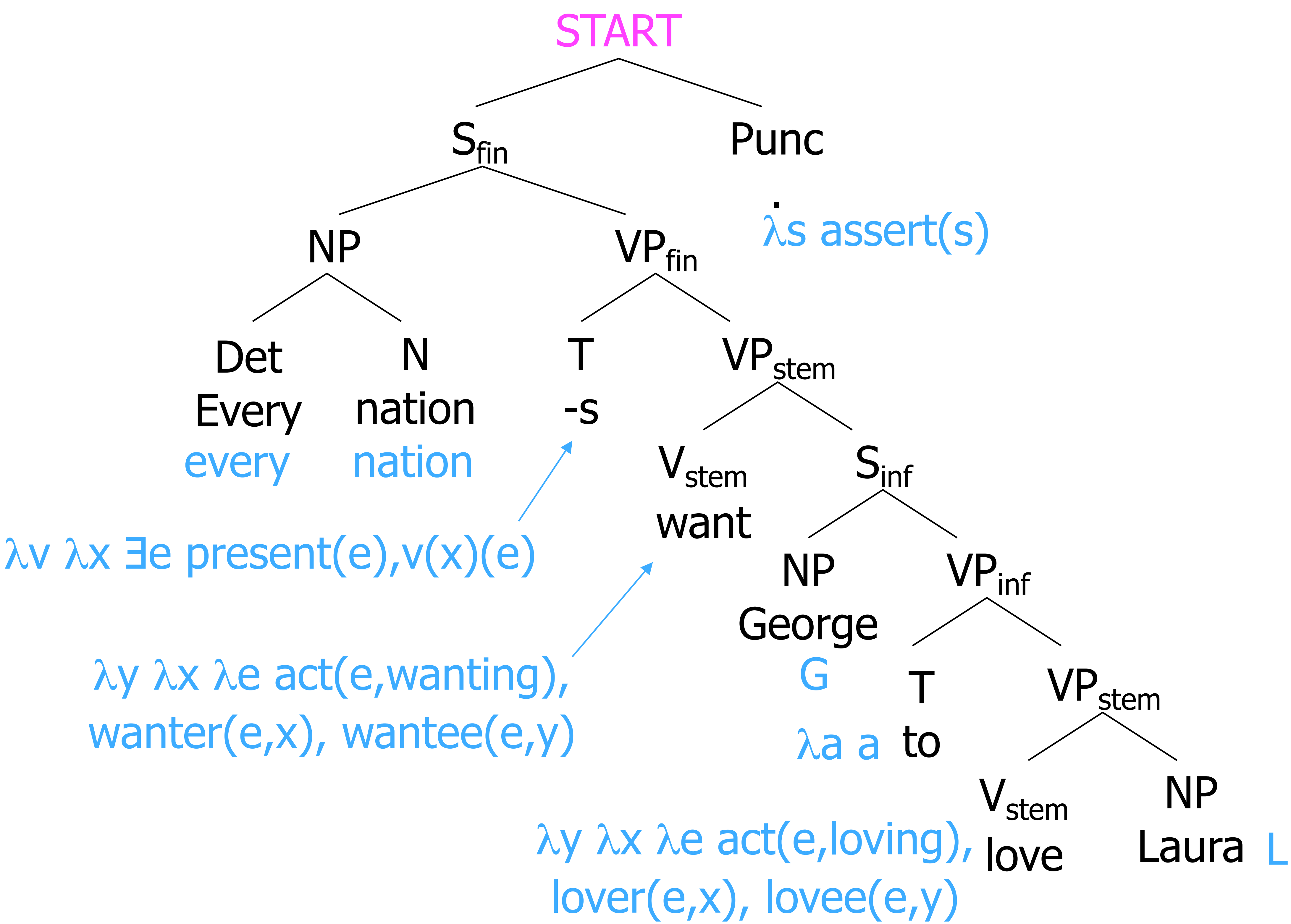
every(nation, λx ∃e present(e), act(e,wanting), wanter(e,x), wantee(e, λe′ act(e′,loving), lover(e′,G), lovee(e′,L)))
START
λp every(nation, p)
S fin
Punc
.
NP
VP fin
λx ∃e present(e), act(e,wanting), wanter(e,x), wantee(e, λe′ act(e′,loving), lover(e′,G), lovee(e′,L))
Det
Every
N
nation
T
-s
VP stem
V stem
want
S inf
NP
George
VP inf
T
to
VP stem
V stem
love
NP
Laura


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
Sfin
Punc
.
λp every(nation, p)
NP
VPfin
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
nation
T
-s
VPstem
Vstem
want
Sinf
NP
George
VPinf
T
to
VPstem
Vstem
love
NP
Laura

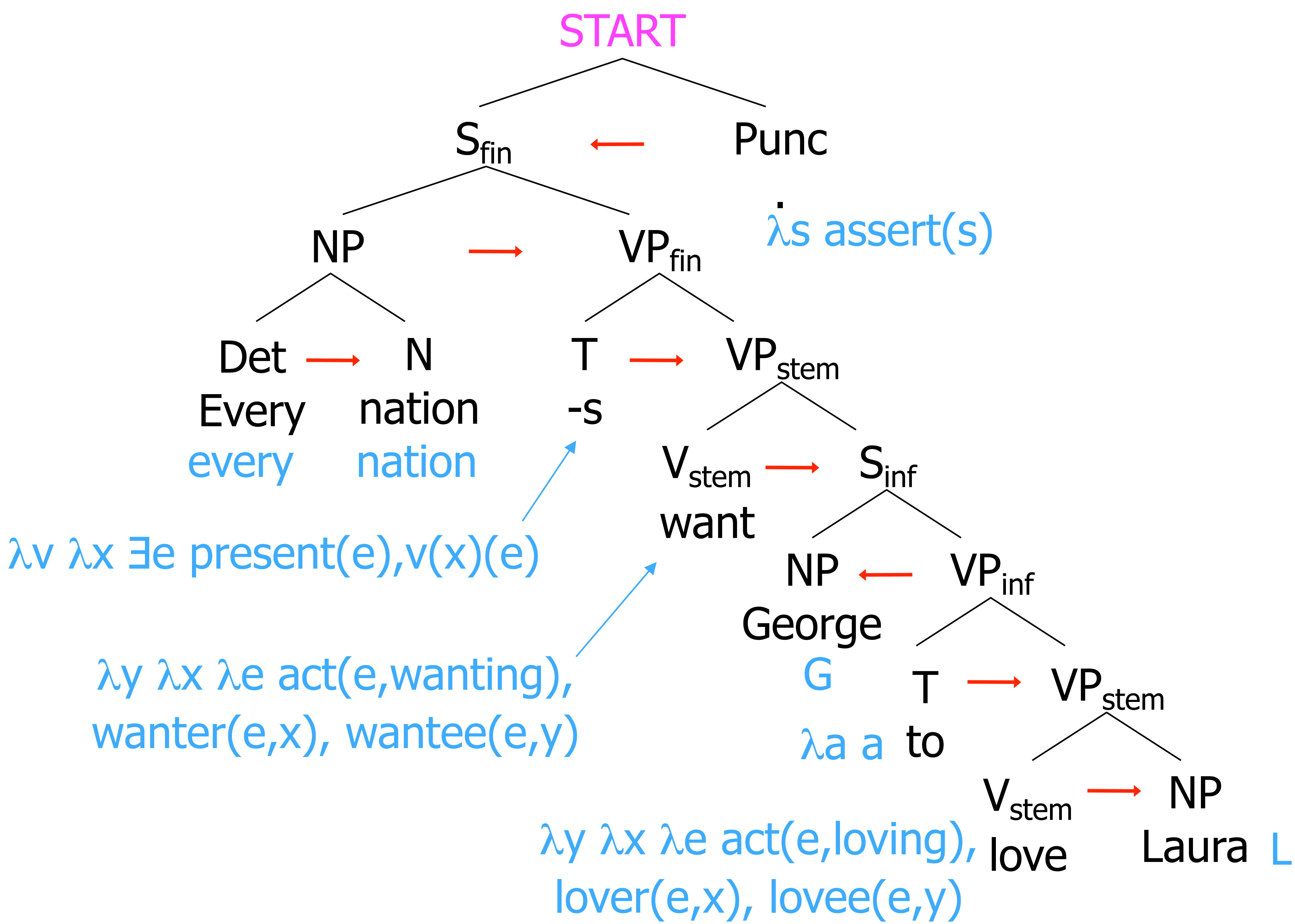

every(nation, λx ∃e present(e), act(e,wanting), wanter(e,x), wantee(e, λe′ act(e′,loving), lover(e′,G), lovee(e′,L)))
START
S fin
Punc
.
λp every(nation, p)
NP
VP fin
λx ∃e present(e), act(e,wanting), wanter(e,x), wantee(e, λe′ act(e′,loving), lover(e′,G), lovee(e′,L))
Det
Every
N
nation
T
-s
VP stem
λn λp every(n, p)
nation
V stem
want
S inf
NP
George
VP inf
T
to
VP stem
V stem
love
NP
Laura


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
.
λs assert(s)
NP
VP fin
Det
Every
N
nation
T
-s
VP stem
V stem
want
S inf
NP
George
VP inf
T
to
VP stem
V stem
love
NP
Laura

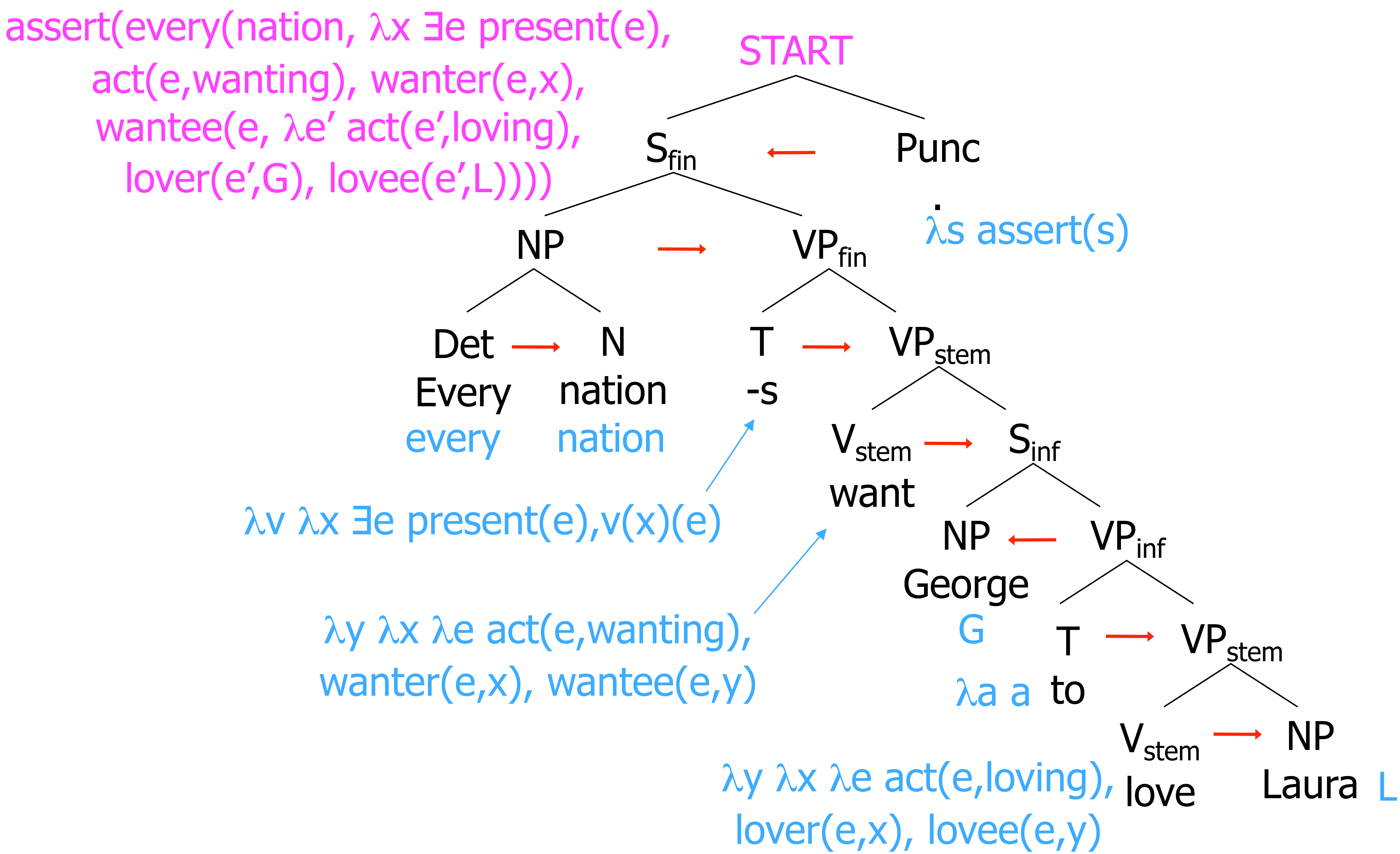
In Summary: From the Words



In Summary: From the Words



In Summary: From the Words



Other Fun Semantic Stuff: A Few Much-Studied Miscellany

- Temporal logic
  - `Gilly had swallowed eight goldfish before Milly reached the bowl` (underlined: "had swallowed", "reached")
  - `Billy said Jilly was pregnant` (underlined: "was")
  - `Billy said, "Jilly is pregnant."` (underlined: "is")
- Generics
  - `Typhoons arise in the Pacific`
  - `Children must be carried`
- Presuppositions
  - `The king of France is bald.`
  - `Have you stopped beating your wife?`
- Pronoun-Quantifier Interaction ("bound anaphora")
  - `Every farmer who owns a donkey beats it.` (underlined: "it")
  - `If you have a dime, put it in the meter.` (underlined: "it")
  - `The woman who every Englishman loves is his mother.` (underlined: "his")
  - `I love my mother and so does Billy.` (underlined: "so")

In Summary

- How do we judge a good meaning representation?
- How can we represent sentence meaning with first-order logic?
- How can logical representations of sentences be **composed** from logical forms of words?
- Next time: can we train models to recover logical forms?

Computational Semantics

Overview

- So far: What is semantics?
  - First order logic and lambda calculus for compositional semantics
- Now: How do we infer semantics?
  - Minimalist (not in Chomskyan sense) approach
    - Semantic role labeling
  - Semantically informed grammar
    - Combinatory categorial grammar (CCG)
    - Tree adjoining grammar (TAG)

Semantic Role Labeling

- Characterize predicates (e.g., verbs, nouns, adjectives) as *relations* with *roles* (slots)

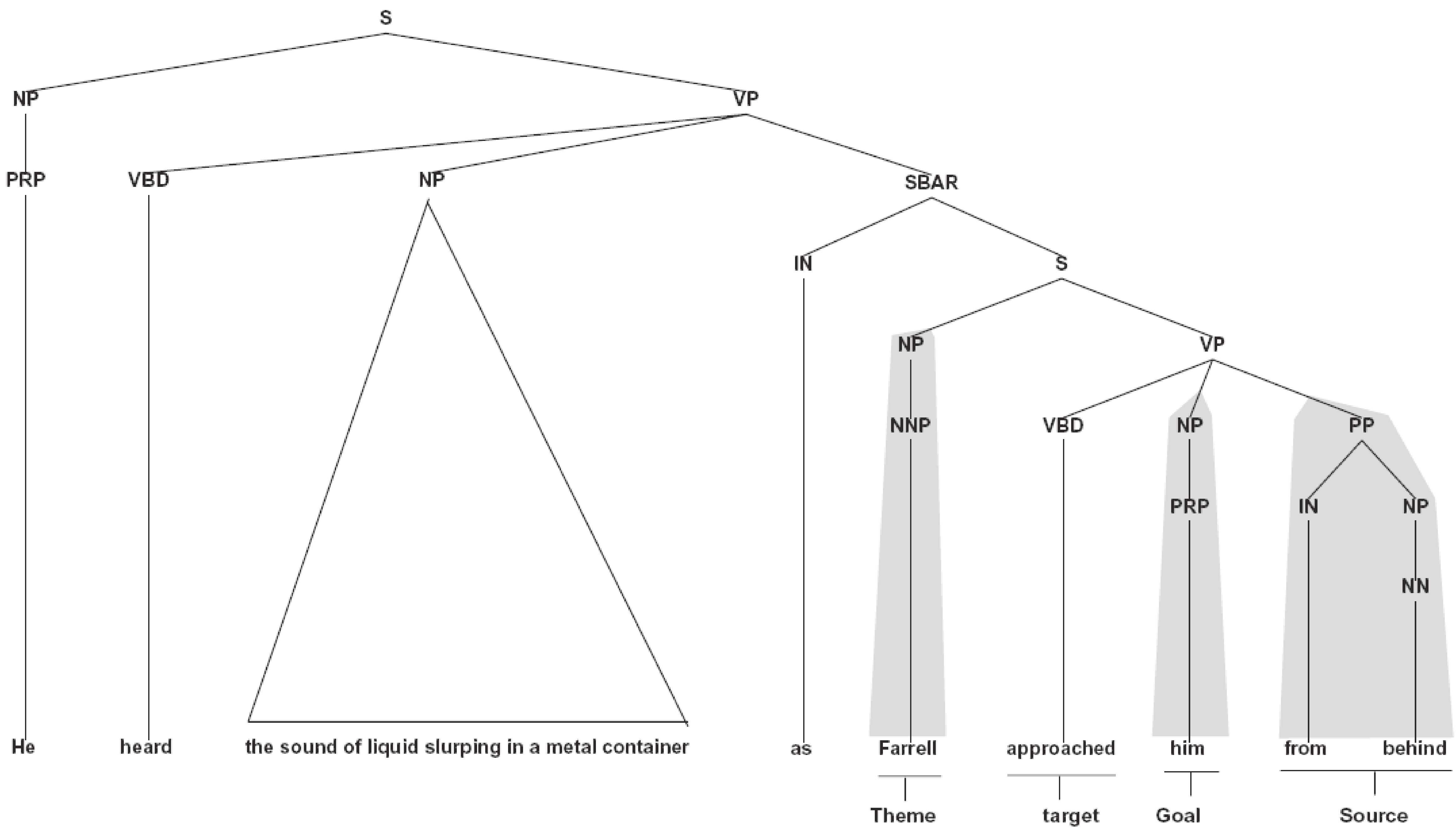
  [$_{Judge}$ She] **blames** [$_{Evaluee}$ the Government] [$_{Reason}$ for failing to do enough to help] .

  Holman would characterize this as **blaming** [$_{Evaluee}$ the poor] .

  The letter quotes Black as saying that [$_{Judge}$ white and Navajo ranchers] misrepresent their livestock losses and **blame** [$_{Reason}$ everything] [$_{Evaluee}$ on coyotes] .

- We want a bit more than which NP is the subject (but not much more):
  - Relations like subject are syntactic, relations like agent or experiencer are semantic (think of passive verbs)
- Typically, SRL is performed in a pipeline on top of constituency or dependency parsing and is much easier than parsing.

SRL Example


S
NP
VP
PRP
VBD
NP
SBAR
IN
S
NP
VP
NNP
VBD
NP
PP
PRP
IN
NP
NN
He
heard
the sound of liquid slurping in a metal container
as
Farrell
approached
him
from
behind
Theme
target
Goal
Source

PropBank Example

fall.01		sense: move downward	
	roles:	Arg1:	thing falling
		Arg2:	extent, distance fallen
		Arg3:	start point
		Arg4:	end point

Sales fell to $251.2 million from $278.7 million.

arg1:	Sales
rel:	fell
arg4:	to $251.2 million
arg3:	from $278.7 million

PropBank Example

rotate.02		sense: shift from one thing to another	
	roles:	Arg0:	causer of shift
		Arg1:	thing being changed
		Arg2:	old thing
		Arg3:	new thing

Many of Wednesday's winners were losers yesterday as investors quickly took profits and rotated their buying to other issues, traders said. (wsj_1723)

arg0:	investors
rel:	rotated
arg1:	their buying
arg3:	to other issues

PropBank Example

aim.01 sense: intend, plan
roles: Arg0: aimer, planner
Arg1: plan, intent

The Central Council of Church Bell Ringers aims *trace* to improve relations with vicars. (wsj_0089)

arg0: The Central Council of Church Bell Ringers
rel: aims
arg1: *trace* to improve relations with vicars

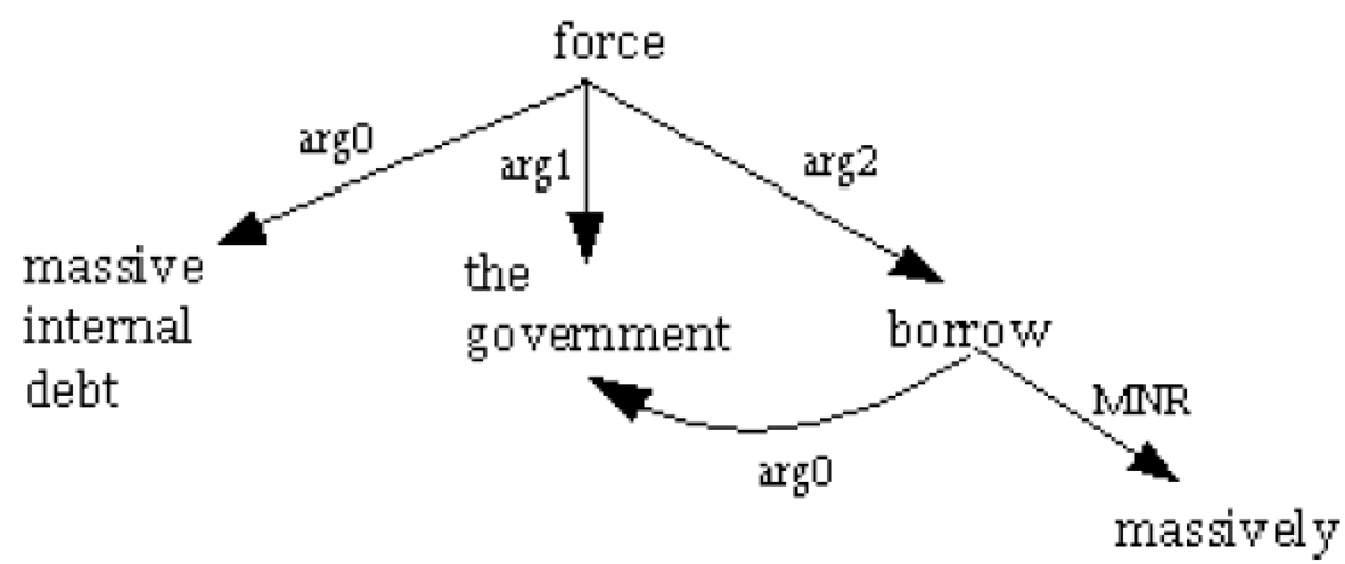
aim.02 sense: point (weapon) at
roles: Arg0: aimer
Arg1: weapon, etc.
Arg2: target

Banks have been aiming packages at the elderly.

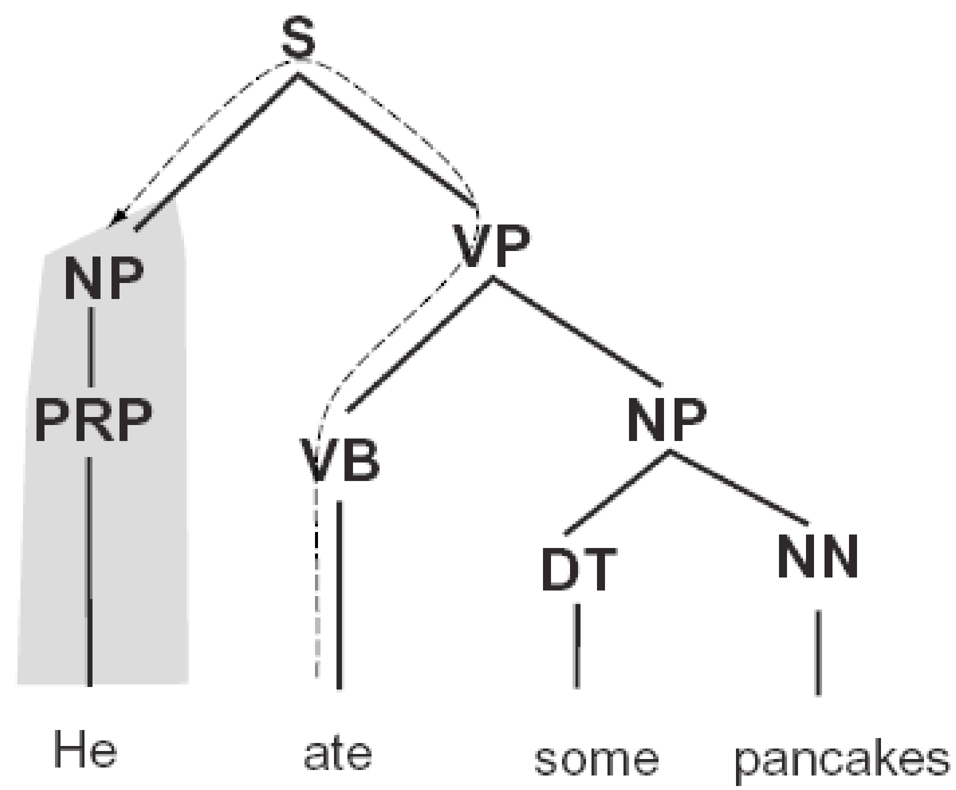
arg0: Banks
rel: aiming
arg1: packages
arg2: at the elderly

Shared Arguments

```
(NP-SBJ (JJ massive) (JJ internal) (NN debt) )
  (VP (VBZ has)
   (VP (VBN forced)
    (S
     (NP-SBJ-1 (DT the) (NN government) )
     (VP
      (VP (TO to)
       (VP (VB borrow)
        (ADVP-MNR (RB massively) )...
```



Path Features



Path	*Description*
VB↑VP↓PP	PP argument/adjunct
VB↑VP↑S↓NP	subject
VB↑VP↓NP	object
VB↑VP↑VP↑S↓NP	subject (embedded VP)
VB↑VP↓ADVP	adverbial adjunct
NN↑NP↑NP↓PP	prepositional complement of noun

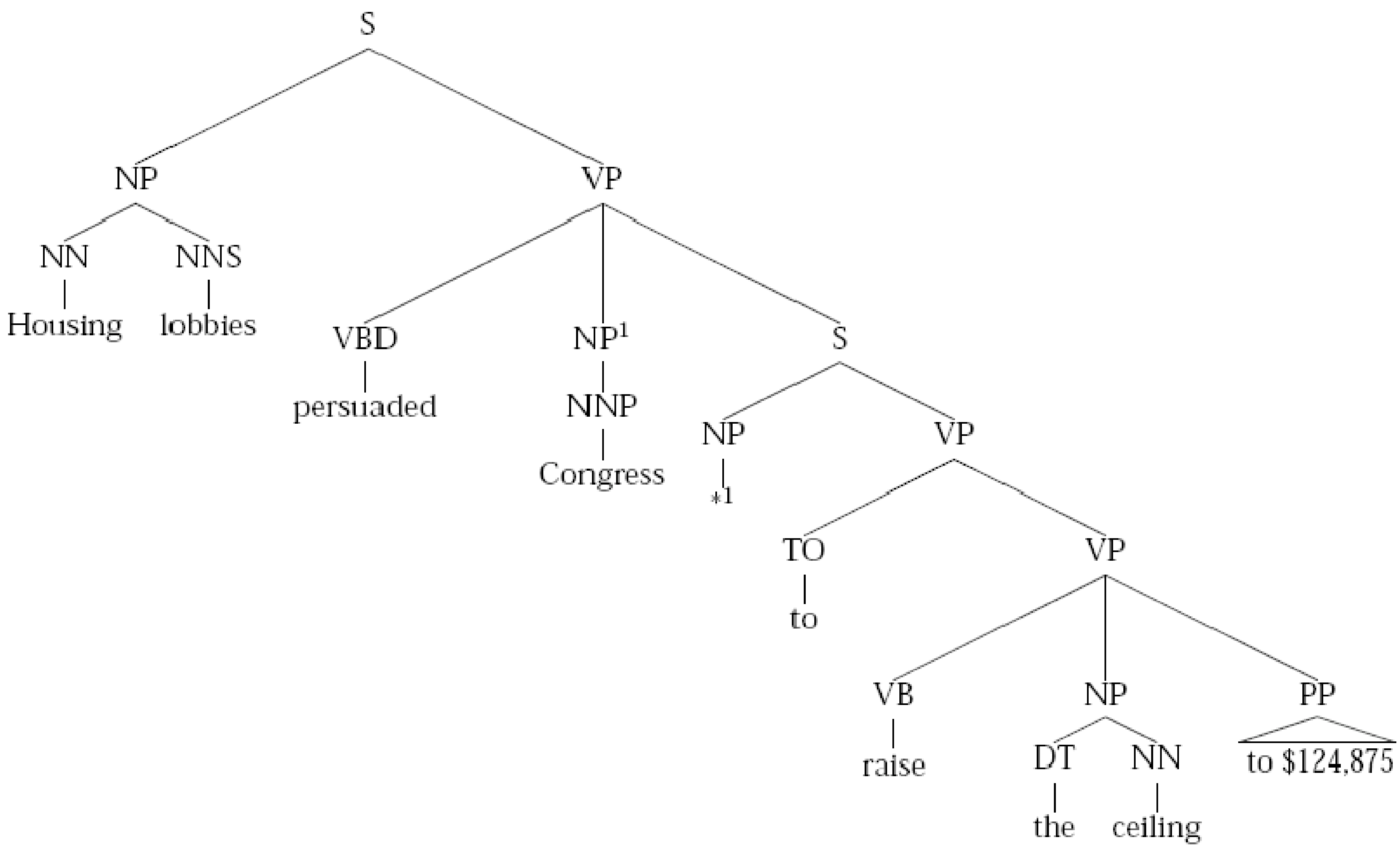
SRL Accuracy

- Features
  - Path from target to role-filler
  - Filler's syntactic type, headword, case
  - Target's identity
  - Sentence voice, etc.
  - Lots of other second-order features
- Gold vs. parsed source trees
  - SRL is fairly easy on gold trees
  - Harder on automatic parses
  - Joint inference of syntax and semantics not a helpful as expected

CORE		ARGM	
F1	Acc.	F1	Acc.
92.2	80.7	89.9	71.8

CORE		ARGM	
F1	Acc.	F1	Acc.
84.1	66.5	81.4	55.6

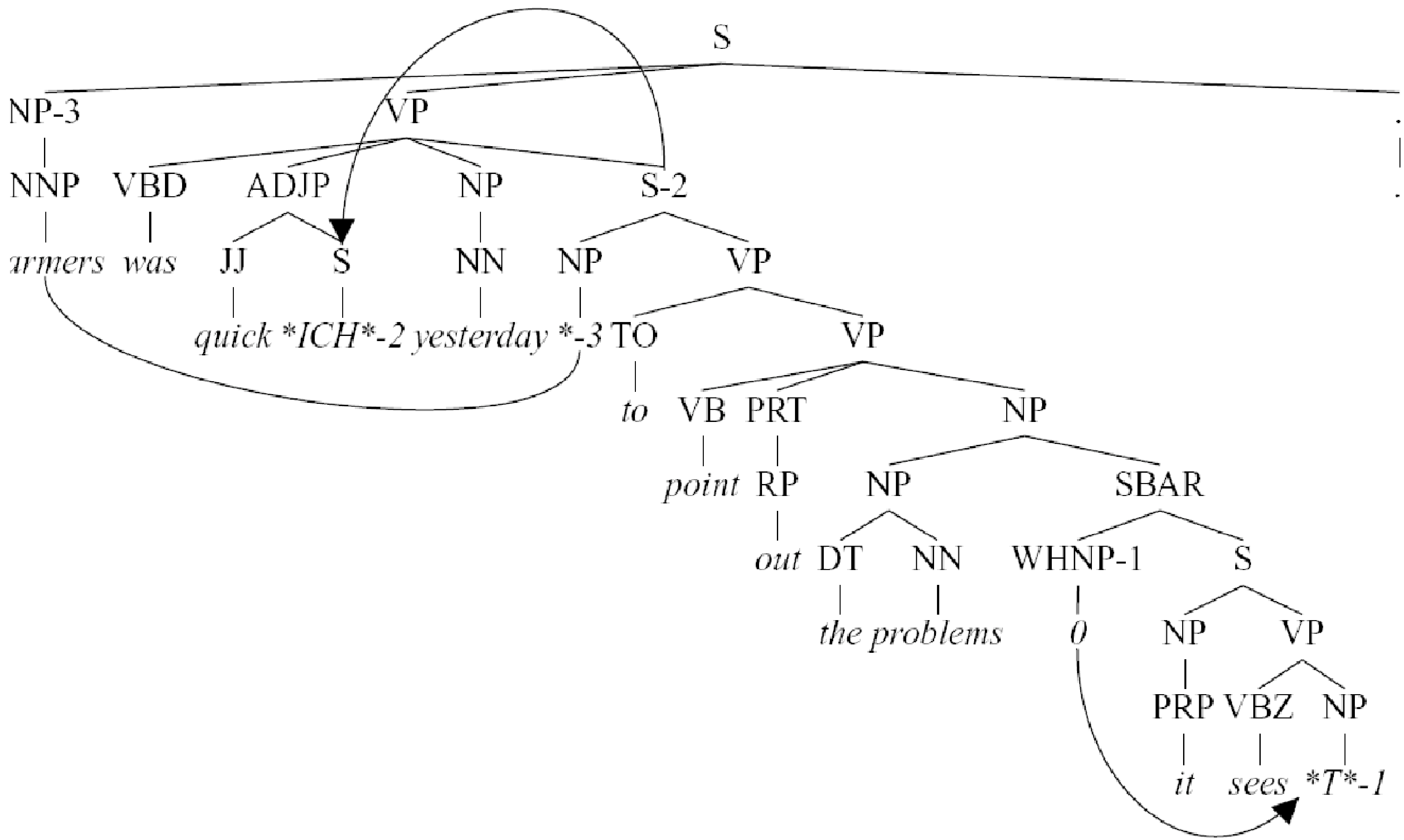
Interaction with Empty Elements



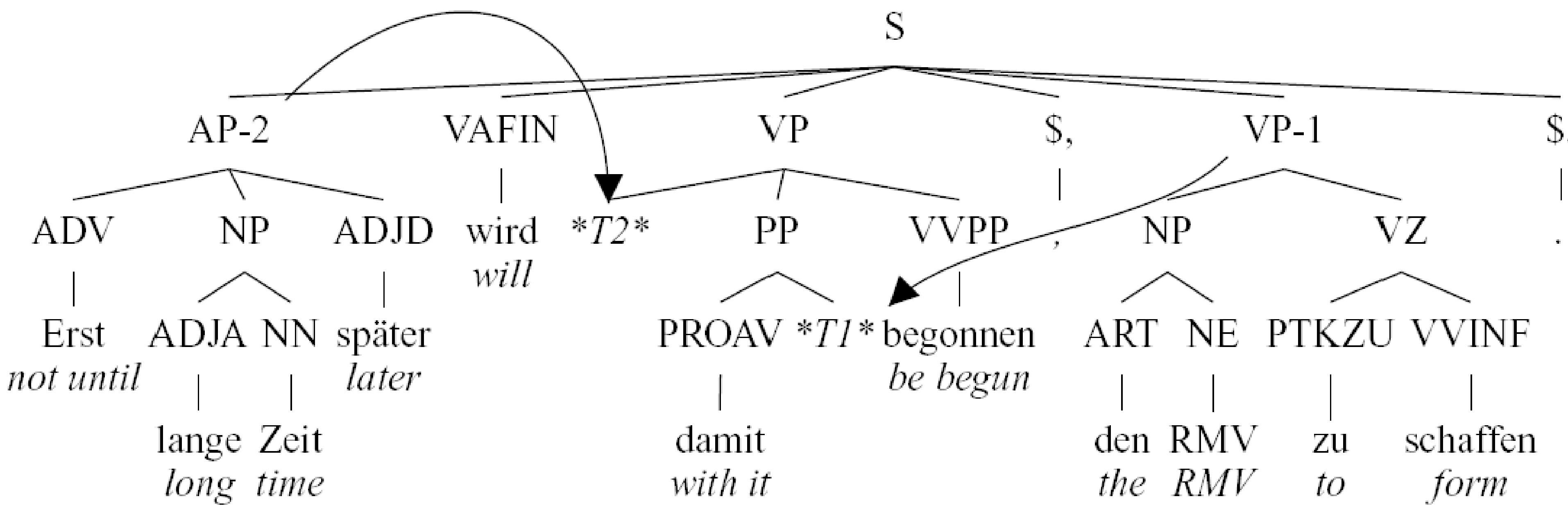
Empty Elements

- In Penn Treebank, 3 kinds of empty elem.
  - Null items
  - Movement traces (WH, topicalization, relative clause and heavy NP extraposition)
  - Control (raising, passives, control, shared arguments)
- Semantic interpretation needs to reconstruct these and resolve indices

English Example



German Example



Combinatory Categorial Grammar

Combinatory Categorial Grammar (CCG)

- Categorial grammar (CG) is one of the oldest grammar formalisms
- *Combinatory* Categorial Grammar now well established and computationally well founded (Steedman, 1996, 2000)
  - Account of syntax; semantics; prodody and information structure; automatic parsers; generation

Combinatory Categorial Grammar (CCG)

- CCG is a lexicalized grammar
- An elementary syntactic structure – for CCG a lexical category – is assigned to each word in a sentence

  walked: S\NP "give me an NP to my left and I return a sentence"
- A small number of rules define how categories can combine
  - Rules based on the combinators from Combinatory Logic

CCG Lexical Categories

- Atomic categories: S , N , NP , PP , . . . (not many more)
- Complex categories are built recursively from atomic categories and slashes, which indicate the directions of arguments
- Complex categories encode subcategorisation information
  - intransitive verb: S \NP *walked*
  - transitive verb: (S \NP)/NP *respected*
  - ditransitive verb: ((S \NP)/NP)/NP *gave*
- Complex categories can encode modification
  - PP nominal: (NP \NP)/NP
  - PP verbal: ((S \NP)\(S \NP))/NP

Simple CCG Derivation

$$
\begin{array}{ccc}
\textit{interleukin} - \textit{10} & \textit{inhibits} & \textit{production} \\
\hline
NP & (S\backslash NP)/NP & NP \\
\end{array}
$$

$$(S\backslash NP)/NP \quad NP \;\Rightarrow_{>}\; S\backslash NP$$

$$NP \quad S\backslash NP \;\Rightarrow_{<}\; S$$

$>$ forward application

$<$ backward application

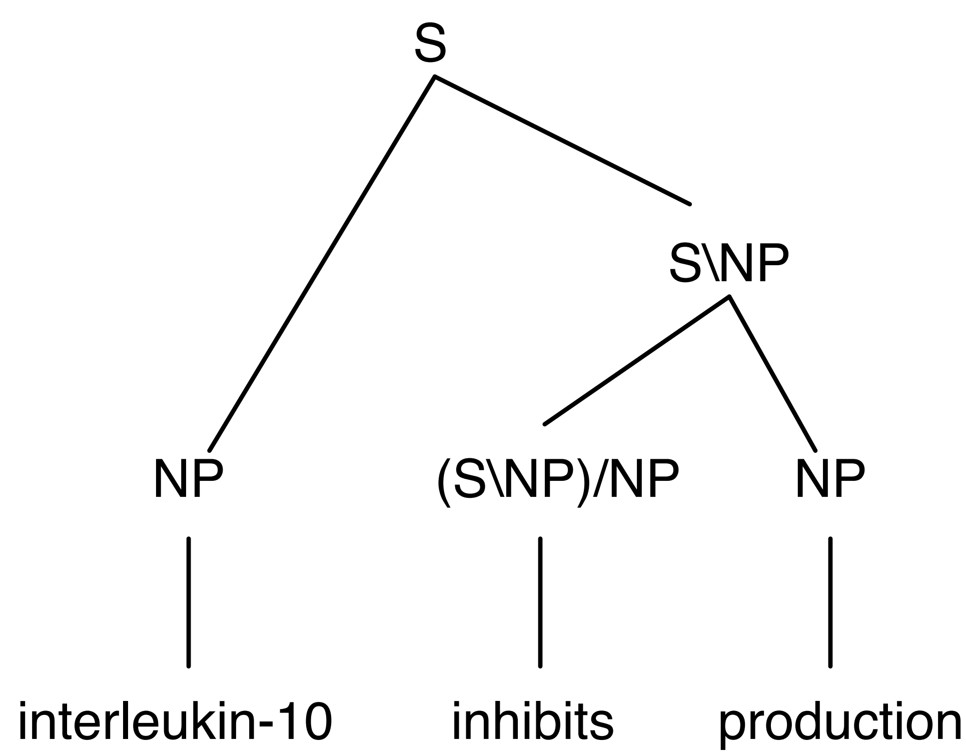
Function Application Schemata

- Forward ($>$) and backward ($<$) application:

$$\begin{array}{lllll} X/Y & Y & \Rightarrow & X & (>) \\ Y & X\backslash Y & \Rightarrow & X & (<) \end{array}$$

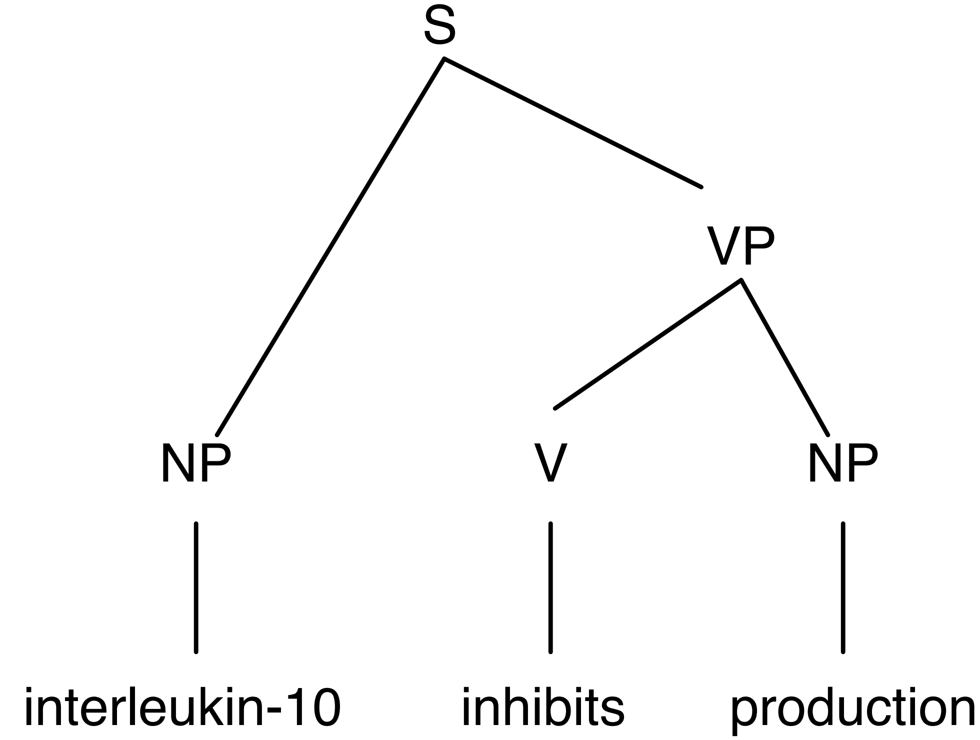
Classical Categorial Grammar

- 'Classical' Categorial Grammar only has application rules
- Classical Categorial Grammar is context free



Classical Categorial Grammar

- 'Classical' Categorial Grammar only has application rules
- Classical Categorial Grammar is context free



Extraction out of a Relative Clause

The	$company$	$which$	$Microsoft$	$bought$
NP/N	N	$(NP\backslash NP)/(S/NP)$	NP	$(S\backslash NP)/NP$

Extraction out of a Relative Clause

$$\frac{The}{NP/N} \quad \frac{company}{N} \quad \frac{which}{(NP\backslash NP)/(S/NP)} \quad \frac{\dfrac{Microsoft}{NP}}{S/(S\backslash NP)}{>}\mathbf{T} \quad \frac{bought}{(S\backslash NP)/NP}$$

$>\mathbf{T}$ type-raising

Extraction out of a Relative Clause

$$
\begin{array}{ccccc}
\textit{The} & \textit{company} & \textit{which} & \textit{Microsoft} & \textit{bought} \\
\hline
NP/N & N & (NP\backslash NP)/(S/NP) & NP & (S\backslash NP)/NP \\
 & & & \overline{S/(S\backslash NP)}^{>\mathbf{T}} & \\
 & & & \multicolumn{2}{c}{\overline{S/NP}^{>\mathbf{B}}}
\end{array}
$$

$>\mathbf{T}$ type-raising

$>\mathbf{B}$ forward composition

Extraction out of a Relative Clause

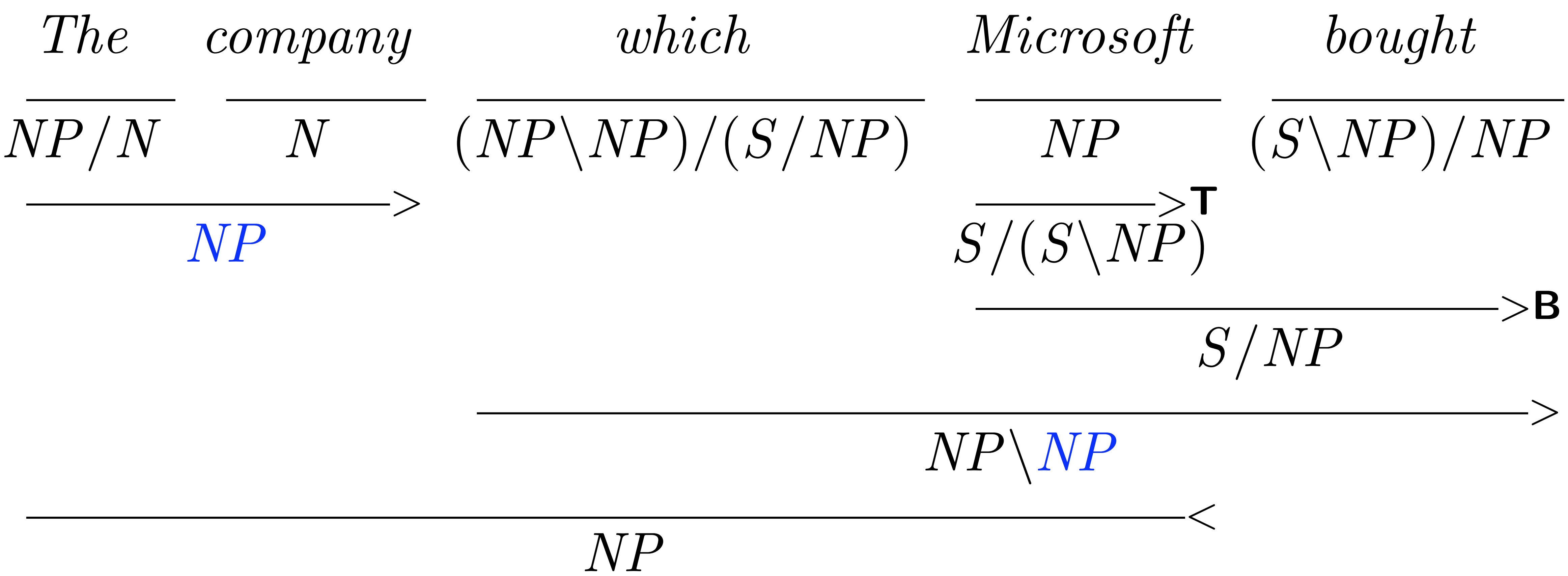
$$\dfrac{The}{NP/N} \quad \dfrac{company}{N} \quad \dfrac{which}{(NP\backslash NP)/(S/NP)} \quad \dfrac{Microsoft}{NP} \quad \dfrac{bought}{(S\backslash NP)/NP}$$

$$\dfrac{NP}{S/(S\backslash NP)}{>}\mathbf{T}$$

$$\dfrac{S/(S\backslash NP) \quad (S\backslash NP)/NP}{S/NP}{>}\mathbf{B}$$

$$\dfrac{(NP\backslash NP)/(S/NP) \quad S/NP}{NP\backslash NP}{>}$$

Extraction out of a Relative Clause



Forward Composition and Type-Raising

- Forward composition ($>_{\mathbf{B}}$):

$$X/Y \quad Y/Z \quad \Rightarrow \quad X/Z \qquad (>_{\mathbf{B}})$$

- Type-raising ($\mathbf{T}$):

$$\begin{array}{lll} X & \Rightarrow T/(T\backslash X) & (>_{\mathbf{T}}) \\ X & \Rightarrow T\backslash(T/X) & (<_{\mathbf{T}}) \end{array}$$

- Extra combinatory rules increase the weak generative power to mild context -sensitivity

"Non-constituents" in CCG – Right Node Raising

Google	*sells*	*but*	*Microsoft*	*buys*	*shares*
NP	$(S\backslash NP)/NP$	$conj$	NP	$(S\backslash NP)/NP$	NP
$\overline{}>\mathbf{T}$			$\overline{}>\mathbf{T}$		
$S/(S\backslash NP)$			$S/(S\backslash NP)$		

$>\mathbf{T}$ type-raising

"Non-constituents" in CCG – Right Node Raising

Google	*sells*	*but*	*Microsoft*	*buys*	*shares*
NP	$(S\backslash NP)/NP$	$conj$	NP	$(S\backslash NP)/NP$	NP
$S/(S\backslash NP)$ $>\mathbf{T}$			$S/(S\backslash NP)$ $>\mathbf{T}$		
S/NP $>\mathbf{B}$			S/NP $>\mathbf{B}$		

$>$ **T** type-raising

$>$ **B** forward composition

"Non-constituents" in CCG – Right Node Raising

Google	*sells*	*but*	*Microsoft*	*buys*	*shares*
NP	$(S\backslash NP)/NP$	$conj$	NP	$(S\backslash NP)/NP$	NP

$$\frac{NP}{S/(S\backslash NP)}{>}\mathbf{T} \qquad \frac{NP}{S/(S\backslash NP)}{>}\mathbf{T}$$

$$\frac{S/(S\backslash NP) \quad (S\backslash NP)/NP}{S/NP}{>}\mathbf{B} \qquad \frac{S/(S\backslash NP) \quad (S\backslash NP)/NP}{S/NP}{>}\mathbf{B}$$

$$\frac{S/NP \quad conj \quad S/NP}{S/NP}{<}\Phi{>}$$

"Non-constituents" in CCG – Right Node Raising

Google	*sells*	*but*	*Microsoft*	*buys*	*shares*
NP	$(S\backslash NP)/NP$	$conj$	NP	$(S\backslash NP)/NP$	NP

$$\frac{NP}{S/(S\backslash NP)} > \mathbf{T} \qquad \frac{NP}{S/(S\backslash NP)} > \mathbf{T}$$

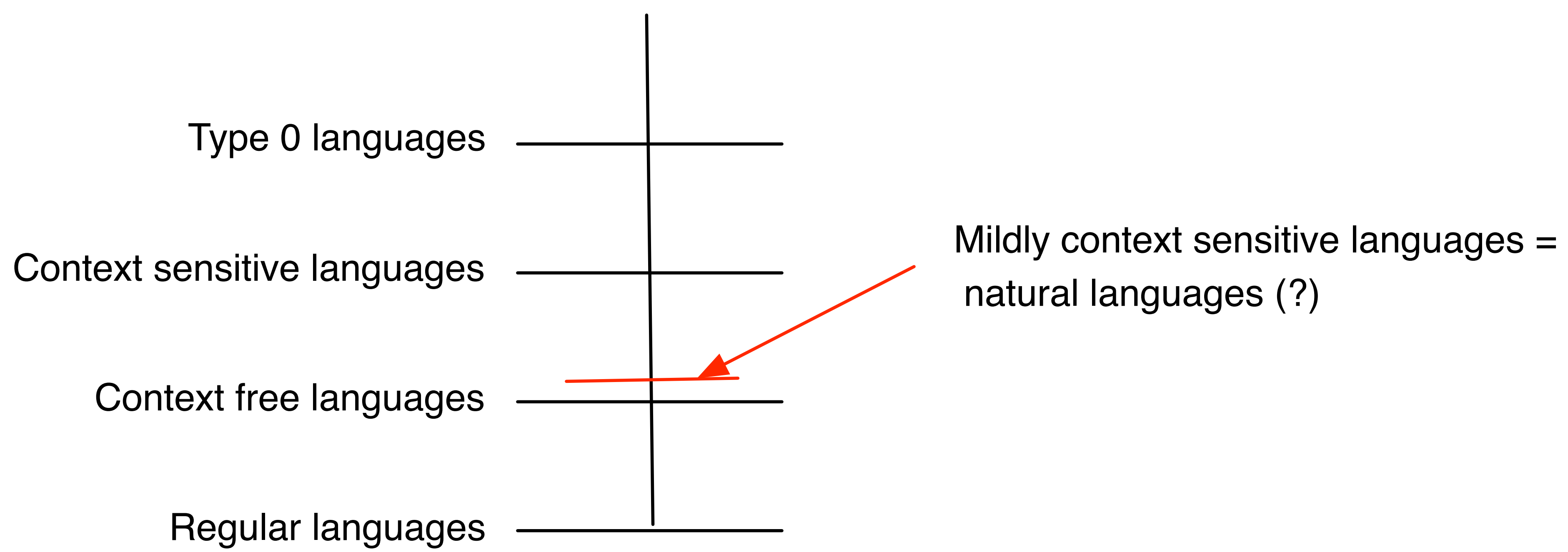
$$\frac{S/(S\backslash NP) \quad (S\backslash NP)/NP}{S/NP} > \mathbf{B} \qquad \frac{S/(S\backslash NP) \quad (S\backslash NP)/NP}{S/NP} > \mathbf{B}$$

$$\frac{S/NP \quad conj \quad S/NP}{S/NP} <\Phi>$$

$$\frac{S/NP \quad NP}{S} >$$

Combinatory Categorial Grammar

- CCG is *mildly* context sensitive
- Natural language is provably non-context free
- Constructions in Dutch and Swiss German (Shieber, 1985) require more than context free power for their analysis
  - these have *crossing* dependencies (which CCG can handle)



CCG Semantics

- Categories encode argument sequences
- Parallel syntactic combinator operations and lambda calculus semantic operations

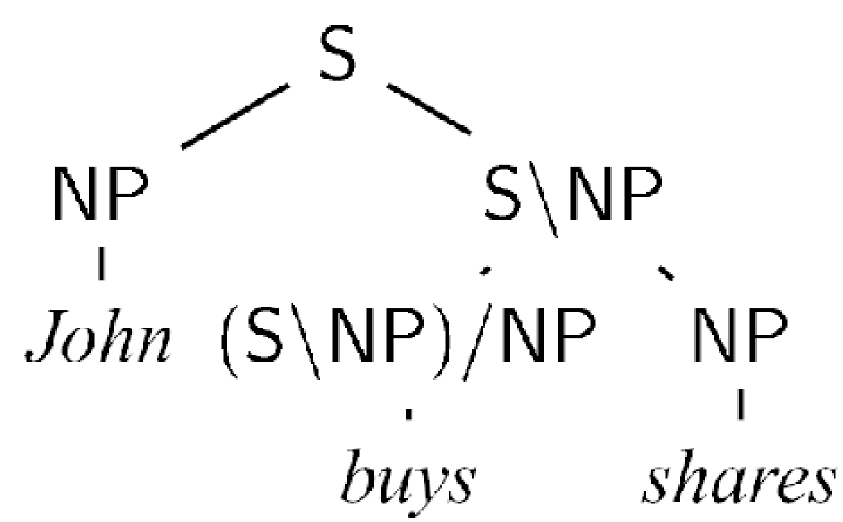
$$John \vdash \mathsf{NP} : john'$$

$$shares \vdash \mathsf{NP} : shares'$$

$$buys \vdash (\mathsf{S}\backslash\mathsf{NP})/\mathsf{NP} : \lambda x.\lambda y.buys' xy$$

$$sleeps \vdash \mathsf{S}\backslash\mathsf{NP} : \lambda x.sleeps' x$$

$$well \vdash (\mathsf{S}\backslash\mathsf{NP})\backslash(\mathsf{S}\backslash\mathsf{NP}) : \lambda f.\lambda x.well'(fx)$$



CCG Semantics

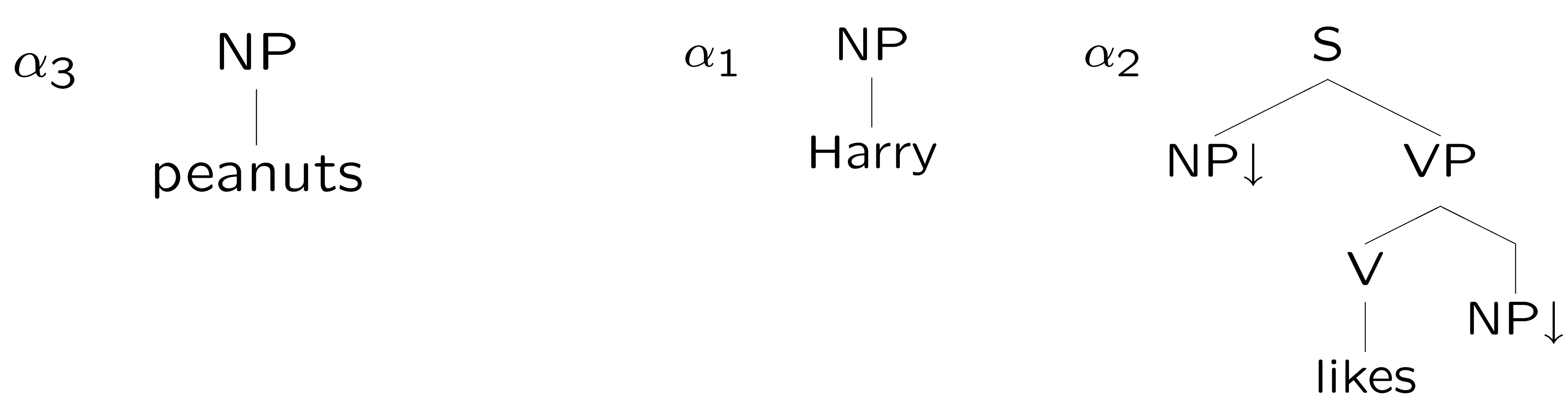
Left arg.	Right arg.	Operation	Result
X/Y : f	Y : a	Forward application	X : f(a)
Y : a	X\Y : f	Backward application	X : f(a)
X/Y : f	Y/Z : g	Forward composition	X/Z : λx.f(g(x))
X : a		Type raising	T/(T\X) : λf.f(a)

etc.

Tree Adjoining Grammar

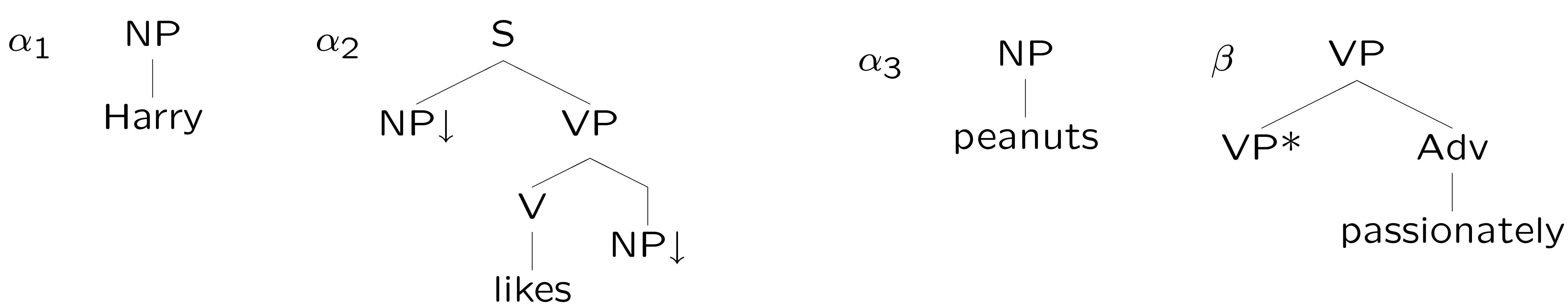
TAG Building Blocks

- Elementary trees (of many depths)
- Substitution at ↓
- Tree *Substitution* Grammar equivalent to CFG

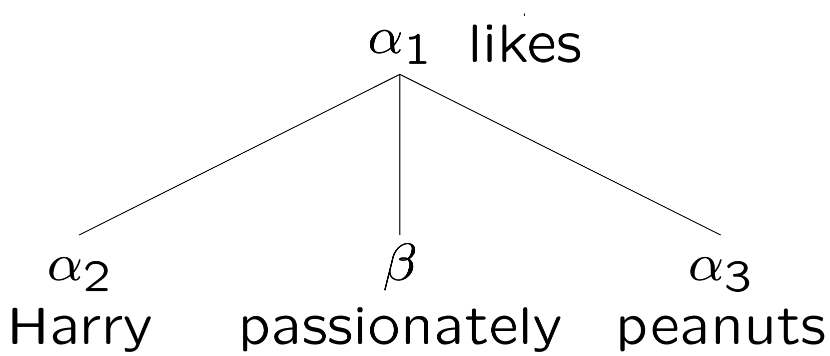


TAG Building Blocks

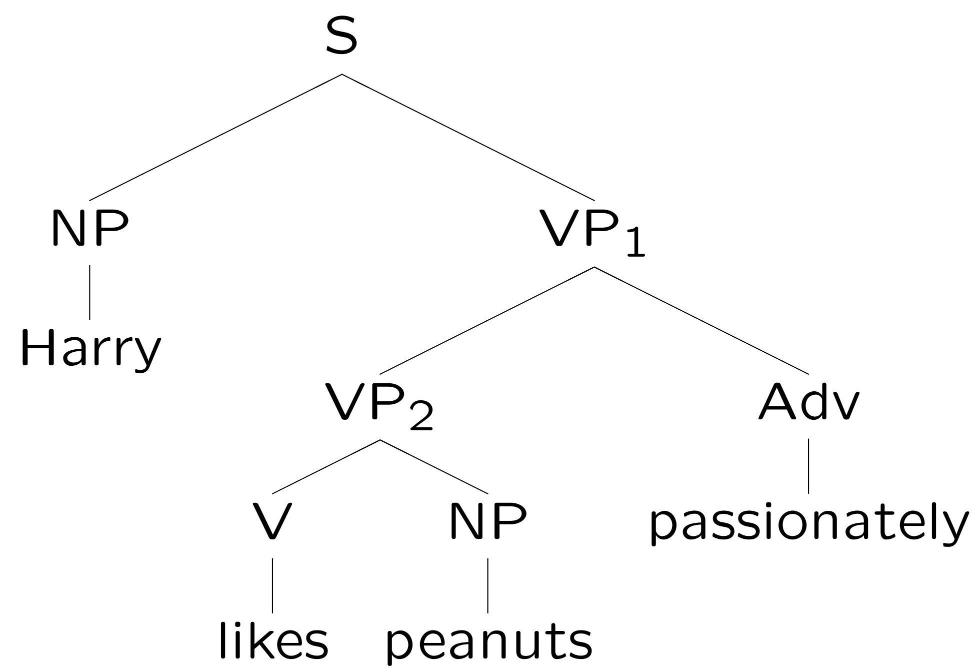
- Auxiliary trees for *adjunction*
- Adds extra power beyond CFG



Derivation Tree



Derived Tree



Semantics

$$Harry(x) \wedge likes(e,x,y) \wedge peanuts(y) \wedge passionately(e)$$

Semantic representation - derived or derivation tree?

Derived tree

- not monotonic (e.g. immediate domination)
- contains nodes that are not needed for semantics

Derivation tree in TAG shows

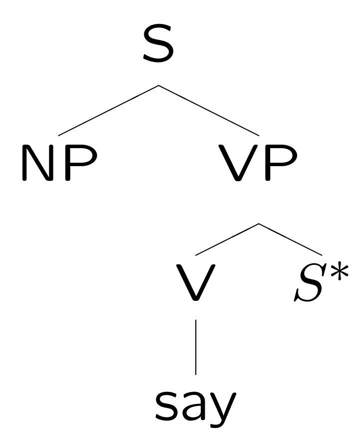
- what elementary and auxiliary trees were used
- how the trees were combined
- where the trees were adjoined / substituted

⇒ Derivation tree provides a natural representation for compositional semantics

Elementary Semantic Representations

- description of meaning (conjunction of formulas)
- list of argument variables

β_{say}



$say(e_1, x, e_2)$
arg: $< x, 00 >, < e_2, 011 >$

Composition of Semantic Representations

- sensitive to way of composition indicated in the derivation tree
- sensitive to order of traversal

Substitution: a new argument is inserted in $\sigma(\alpha)$

- unify the variable corresponding to the argument node (e.g. x in $thought(e, x)$) with the variable in the substituted tree (e.g. NP: $Peter(x_5)$)
- semantic representations are merged

Adjoining: $\sigma(\beta)$ applied to $\sigma(\alpha)$

- predicate: semantic representation of adjoined auxiliary tree
- argument: a variable in the 'host' tree

Harry likes peanuts passionately.

$Harry(x)$
arg: -

$likes(e, x, y)$
arg: $< x, 00 >, < y, 011 >$

$peanuts(y)$
arg: -

$passionately(e)$
arg: e

Result:

$likes(e, x, y) \wedge$ $Harry(x) \wedge$ $peanuts(y) \wedge$ $passionately(e)$
arg: -

Extensions and Multi-Component LTAG

To what extent can we obtain a compositional semantics by using derivation trees?

Problem: Representation of Scope

Every boy saw a girl.

(suppose there are 5 boys in the world, how many girls have to exist for the sentence to be true?)

Quantifiers have two parts:

- predicate-argument structure
- scope information

The two parts don't necessarily stay together in the final semantic representation.

Multi-Component Lexicalized Tree Adjoining Grammar

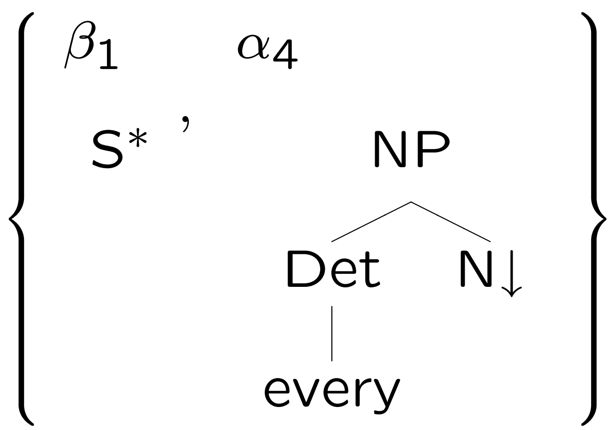
- Building blocks are sets of trees (roughly corresponding to split-up LTAG elementary trees)

- Locality constraint: a multi-component elementary tree has to be combined with only one elementary tree (tree locality; Tree local MC-TAG is as powerful as LTAG)

- We use at most two components in each set

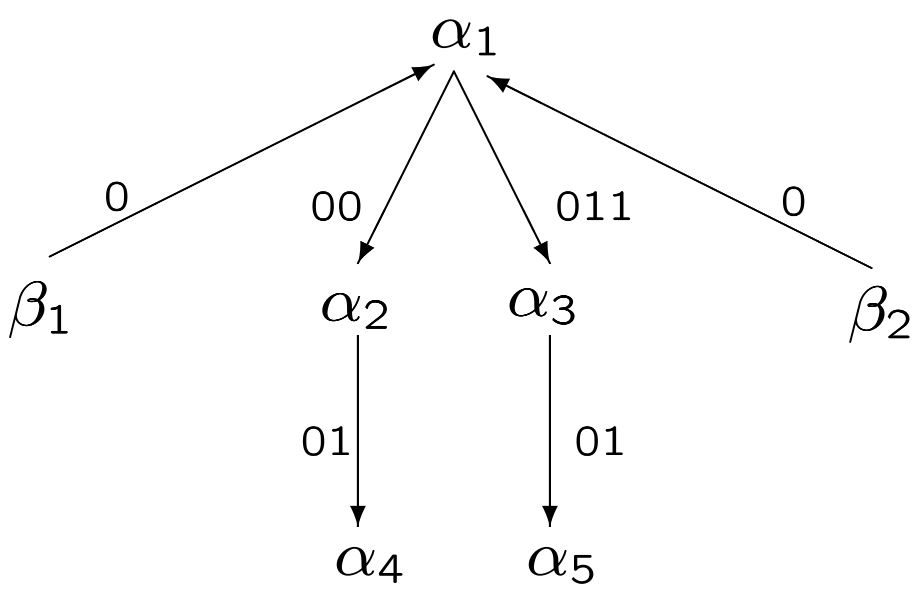
- Constraint on multiple adjunction

Representation of Quantifiers in MC-TAG



Derivation Tree with Two Quantifiers - underspecified scope

Some student loves every course.



18

CCG & TAG

- Lexicon is encoded as categories or trees
- *Extended domain of locality*: information is localized in the lexicon and “spread out” during derivation
- Greater than context-free power; polynomial-time parsing; $O(n^5)$ and up
- Spurious ambiguity: multiple derivations for a single derived tree

Lexical Semantics

Overview

- Semantics so far: compositional semantics
  - How to put together propositions from atomic meanings (lexicon)?
- Now: lexical semantics
  - What are those atomic meanings?
  - Clustering words with similar senses
  - Sense disambiguation, functional clustering

A Concordance for “party”

- thing. She was talking at a party thrown at Daphne's restaurant in
- have turned it into the hot dinner-party topic. The comedy is the
- selection for the World Cup party, which will be announced on May 1
- in the 1983 general election for a party which, when it could not bear to
- to attack the Scottish National Party, who look set to seize Perth and
- that had been passed to a second party who made a financial decision
- the by-pass there will be a street party. "Then," he says, "we are going
- number-crunchers within the Labour party, there now seems little doubt
- political tradition and the same party. They are both relatively Anglophilic
- he told Tony Blair's modernised party they must not retreat into "warm
- "Oh no, I'm just here for the party," they said. "I think it's terrible
- A future obliges each party to the contract to fulfil it by
- be signed by or on behalf of each party to the contract." Mr David N

What Good are Word Senses?

- thing. She was talking at a party thrown at Daphne's restaurant in
- have turned it into the hot dinner-party topic. The comedy is the
- selection for the World Cup party, which will be announced on May 1
- in the 1983 general election for a party which, when it could not bear to
- to attack the Scottish National Party, who look set to seize Perth and
- that had been passed to a second party who made a financial decision
- the by-pass there will be a street party. "Then," he says, "we are going
- number-crunchers within the Labour party, there now seems little doubt
- political tradition and the same party. They are both relatively Anglophilic
- he told Tony Blair's modernised party they must not retreat into "warm
- "Oh no, I'm just here for the party," they said. "I think it's terrible
- A future obliges each party to the contract to fulfil it by
- be signed by or on behalf of each party to the contract." Mr David N

What Good are Word Senses?

- thing. She was talking at a party thrown at Daphne's restaurant in
- have turned it into the hot dinner-party topic. The comedy is the
- selection for the World Cup party, which will be announced on May 1
- the by-pass there will be a street party. "Then," he says, "we are going
- "Oh no, I'm just here for the party," they said. "I think it's terrible

- in the 1983 general election for a party which, when it could not bear to
- to attack the Scottish National Party, who look set to seize Perth and
- number-crunchers within the Labour party, there now seems little doubt
- political tradition and the same party. They are both relatively Anglophilic
- he told Tony Blair's modernised party they must not retreat into "warm

- that had been passed to a second party who made a financial decision
- A future obliges each party to the contract to fulfil it by
- be signed by or on behalf of each party to the contract." Mr David N

What Good are Word Senses?

- John threw a “rain forest” party last December. His living room was full of plants and his box was playing Brazilian music …

What Good are Word Senses?

- Replace word w with sense s
  - Splits w into senses: distinguishes this token of w from tokens with sense t
  - Groups w with other words: groups this token of w with tokens of x that also have sense s

What Good are Word Senses?

- number-crunchers within the Labour party, there now seems little doubt
- political tradition and the same party. They are both relatively Anglophilic
- he told Tony Blair's modernised party they must not retreat into "warm
- thing. She was talking at a party thrown at Daphne's restaurant in
- have turned it into the hot dinner-party topic. The comedy is the
- selection for the World Cup party, which will be announced on May 1
- the by-pass there will be a street party. "Then," he says, "we are going
- "Oh no, I'm just here for the party," they said. "I think it's terrible

- an appearance at the annual awards bash , but feels in no fit state to
- -known families at a fundraising bash on Thursday night for Learning
- Who was paying for the bash? The only clue was the name Asprey,
- Mail, always hosted the annual bash for the Scottish Labour front-
- popular. Their method is to bash sense into criminals with a short,
- just cut off people's heads and bash their brains out over the floor,

What Good are Word Senses?

- number-crunchers within the Labour party, there now seems little doubt
- political tradition and the same party. They are both relatively Anglophilic
- he told Tony Blair's modernised party they must not retreat into "warm

- thing. She was talking at a party thrown at Daphne's restaurant in
- have turned it into the hot dinner-party topic. The comedy is the
- selection for the World Cup party, which will be announced on May 1
- the by-pass there will be a street party. "Then," he says, "we are going
- "Oh no, I'm just here for the party," they said. "I think it's terrible
- an appearance at the annual awards bash, but feels in no fit state to
- -known families at a fundraising bash on Thursday night for Learning
- Who was paying for the bash? The only clue was the name Asprey,
- Mail, always hosted the annual bash for the Scottish Labour front-

- popular. Their method is to bash sense into criminals with a short,
- just cut off people's heads and bash their brains out over the floor,

What Good are Word Senses?

What Good are Word Senses?

- Semantics / Text understanding
  - Axioms about TRANSFER apply to (some tokens of) `throw`
  - Axioms about BUILDING apply to (some tokens of) `bank`

What Good are Word Senses?

- Semantics / Text understanding
  - Axioms about TRANSFER apply to (some tokens of) `throw`
  - Axioms about BUILDING apply to (some tokens of) `bank`
- Machine translation

What Good are Word Senses?

- Semantics / Text understanding
  - Axioms about TRANSFER apply to (some tokens of) `throw`
  - Axioms about BUILDING apply to (some tokens of) `bank`
- Machine translation
- Info retrieval / Question answering / Text categ.
  - Query or pattern might not match document exactly

What Good are Word Senses?

- Semantics / Text understanding
  - Axioms about TRANSFER apply to (some tokens of) `throw`
  - Axioms about BUILDING apply to (some tokens of) `bank`
- Machine translation
- Info retrieval / Question answering / Text categ.
  - Query or pattern might not match document exactly
- Backoff for just about anything
  - what word comes next? (speech recognition, language ID, …)
    - trigrams are sparse but tri-meanings might not be
  - bilexical PCFGs: p(S[`devour`] → NP[`lion`] VP[`devour`] | S[`devour`])
    - approximate by p(S[EAT] → NP[`lion`] VP[EAT] | S[EAT])

What Good are Word Senses?

- Semantics / Text understanding
  - Axioms about TRANSFER apply to (some tokens of) `throw`
  - Axioms about BUILDING apply to (some tokens of) `bank`
- Machine translation
- Info retrieval / Question answering / Text categ.
  - Query or pattern might not match document exactly
- Backoff for just about anything
  - what word comes next? (speech recognition, language ID, …)
    - trigrams are sparse but tri-meanings might not be
  - bilexical PCFGs: p(S[devour] → NP[lion] VP[devour] | S[devour])
    - approximate by p(S[EAT] → NP[lion] VP[EAT] | S[EAT])
- Speaker's real intention is senses; words are a noisy channel

Cues to Word Sense

Cues to Word Sense

- Adjacent words (or their senses)

Cues to Word Sense

- Adjacent words (or their senses)
- Grammatically related words (subject, object, …)

Cues to Word Sense

- Adjacent words (or their senses)
- Grammatically related words (subject, object, …)
- Other nearby words

Cues to Word Sense

- Adjacent words (or their senses)
- Grammatically related words (subject, object, …)
- Other nearby words
- Topic of document

Cues to Word Sense

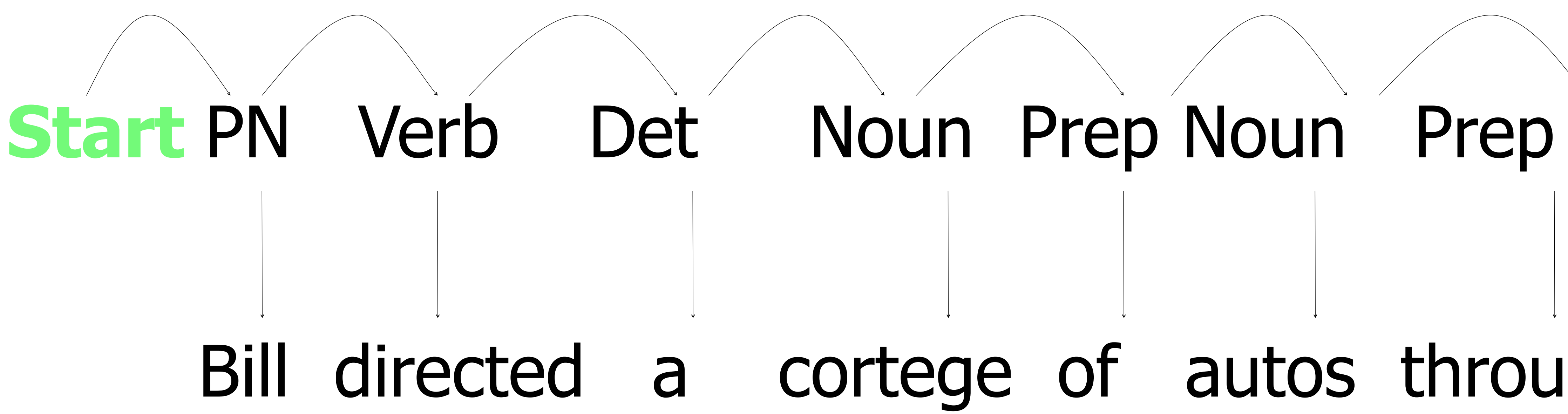
- Adjacent words (or their senses)
- Grammatically related words (subject, object, …)
- Other nearby words
- Topic of document
- Sense of other tokens of the word in the same document

Word Classes by Tagging

- Every tag is a kind of class
- Tagger assigns a class to each word token

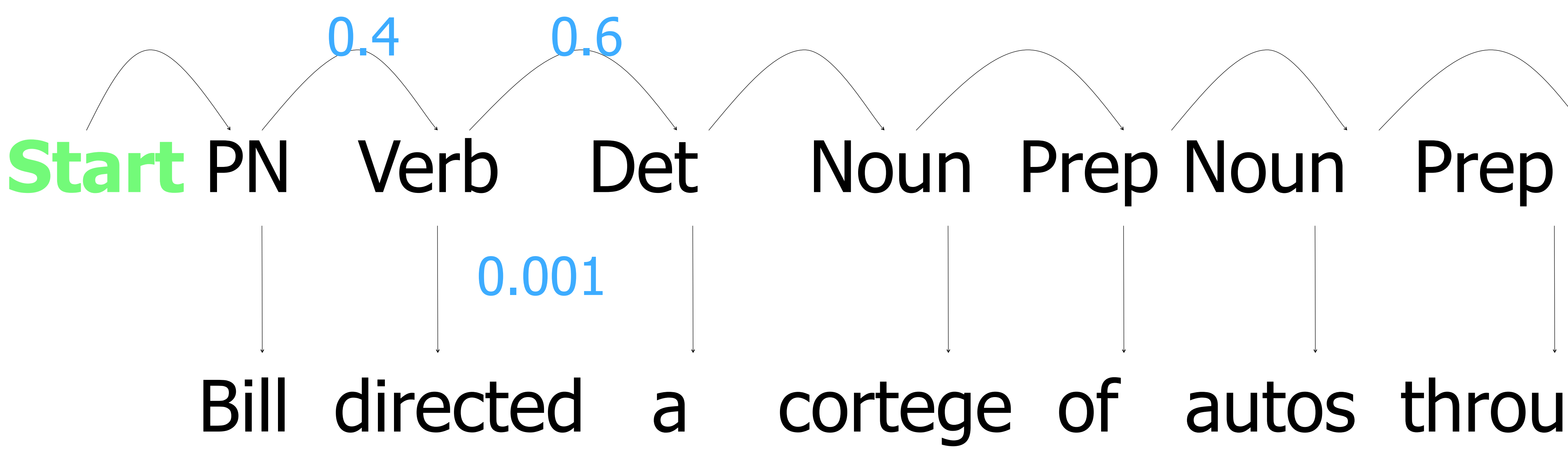
Word Classes by Tagging

- Every tag is a kind of class
- Tagger assigns a class to each word token



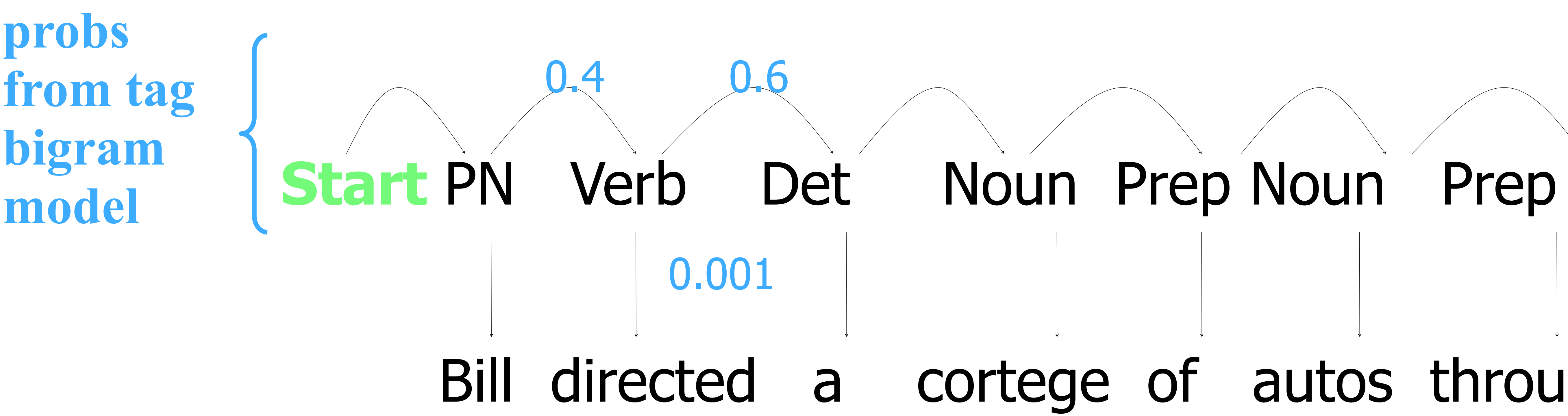
Word Classes by Tagging

- Every tag is a kind of class
- Tagger assigns a class to each word token



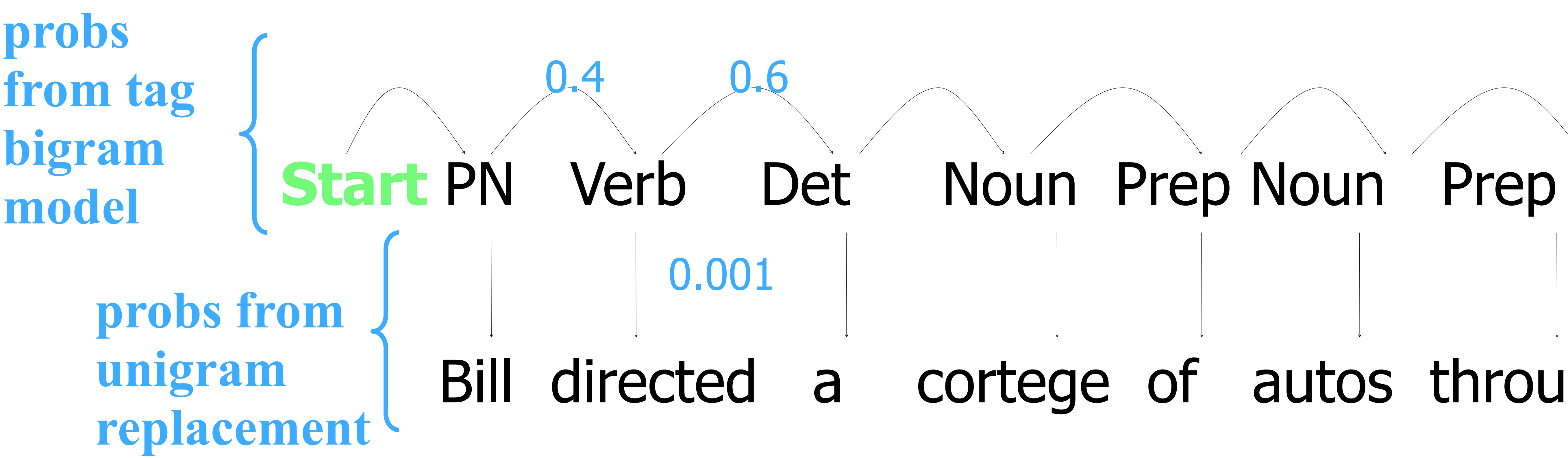
Word Classes by Tagging

- Every tag is a kind of class
- Tagger assigns a class to each word token



Word Classes by Tagging

- Every tag is a kind of class
- Tagger assigns a class to each word token



Word Classes by Tagging

- Every tag is a kind of class
- Tagger assigns a class to each word token
  - Simultaneously groups and splits words
  - “party” gets split into N and V senses
  - “bash” gets split into N and V senses
  - {party/N, bash/N} vs. {party/V, bash/V}
  - What good are these groupings?

Learning Word Classes

- Every tag is a kind of class
- Tagger assigns a class to each word token
  - {party/N, bash/N} vs. {party/V, bash/V}
  - What good are these groupings?
  - Good for predicting next word or its class!

- Role of forward-backward algorithm?
  - It adjusts classes etc. in order to predict sequence of words better (with lower perplexity)

Words as Vectors

- Represent each word **type** w by a point in k-dimensional space
  - e.g., k is size of vocabulary
  - the 17^{th} coordinate of w represents **strength** of w's association with vocabulary word 17

Words as Vectors

- Represent each word **type** w by a point in k-dimensional space
  - e.g., k is size of vocabulary
  - the 17^{th} coordinate of w represents **strength** of w's association with vocabulary word 17

= party

Words as Vectors

- Represent each word **type** w by a point in k-dimensional space
  - e.g., k is size of vocabulary
  - the 17^{th} coordinate of w represents **strength** of w's association with vocabulary word 17

= party

(0, 0, 3, 1, 0, 7, . . . 1, 0)

Words as Vectors

- Represent each word **type** w by a point in k-dimensional space
  - e.g., k is size of vocabulary
  - the 17th coordinate of w represents **strength** of w's association with vocabulary word 17

= party

aardvark abacus abandoned abbot abduct above ... zygote zymurgy

(0, 0, 3, 1, 0, 7, . . . 1, 0)

Words as Vectors

- Represent each word **type** w by a point in k-dimensional space
  - e.g., k is size of vocabulary
  - the 17th coordinate of w represents **strength** of w's association with vocabulary word 17

= **party**

aardvark abacus abandoned abbot abduct above zygote zymurgy

(0, 0, 3, 1, 0, 7, . . . 1, 0)

From corpus:

Arlen Specter **abandoned** the Republican **party**.
There were lots of **abbots** and nuns dancing at that **party**.
The **party** **above** the art gallery was, **above** all, a laboratory for synthesizing **zygotes** and beer.

Words as Vectors

- Represent each word **type** w by a point in k-dimensional space
  - e.g., k is size of vocabulary
  - the 17th coordinate of w represents **strength** of w's association with vocabulary word 17

= party

aardvark abacus abandoned abbot abduct above ... zygote zymurgy

(0, 0, 3, 1, 0, 7, . . . 1, 0)

count too high (too influential)

From corpus:

Arlen Specter **abandoned** the Republican **party**.
There were lots of **abbots** and nuns dancing at that **party**.
The **party** **above** the art gallery was, **above** all, a laboratory for synthesizing **zygotes** and beer.

Words as Vectors

- Represent each word **type** w by a point in k-dimensional space
  - e.g., k is size of vocabulary
  - the 17th coordinate of w represents **strength** of w's association with vocabulary word 17

= **party**

aardvark abacus abandoned abbot abduct above ... zygote zymurgy

(0, 0, 3, 1, 0, 7, . . . 1, 0)

count too high (too influential)

count too low

From corpus:

Arlen Specter **abandoned** the Republican **party**.
There were lots of **abbots** and nuns dancing at that **party**.
The **party** **above** the art gallery was, **above** all, a laboratory for synthesizing **zygotes** and beer.

Words as Vectors

- Represent each word **type** w by a point in k-dimensional space
  - e.g., k is size of vocabulary
  - the 17th coordinate of w represents **strength** of w's association with vocabulary word 17

= party

aardvark	abacus	abandoned	abbot	abduct	above		zygote	zymurgy
(0,	0,	3,	1,	0,	7,	. . .	1,	0)

Words as Vectors

- Represent each word **type** w by a point in k-dimensional space
  - e.g., k is size of vocabulary
  - the 17th coordinate of w represents **strength** of w's association with vocabulary word 17

= party

aardvark abacus abandoned abbot abduct above zygote zymurgy

(0, 0, 3, 1, 0, 7, . . . 1, 0)

how might you measure this?

Words as Vectors

- Represent each word **type** w by a point in k-dimensional space
  - e.g., k is size of vocabulary
  - the 17th coordinate of w represents **strength** of w's association with vocabulary word 17

= party

aardvark abacus abandoned abbot abduct above ... zygote zymurgy

(0, 0, 3, 1, 0, 7, . . . 1, 0)

how might you measure this?

- how often words appear next to each other

Words as Vectors

- Represent each word **type** w by a point in k-dimensional space
  - e.g., k is size of vocabulary
  - the 17th coordinate of w represents **strength** of w's association with vocabulary word 17

= party

aardvark abacus abandoned abbot abduct above ... zygote zymurgy

(0, 0, 3, 1, 0, 7, . . . 1, 0)

how might you measure this?

- how often words appear next to each other
- how often words appear near each other

Words as Vectors

- Represent each word **type** w by a point in k-dimensional space

  = party

  - e.g., k is size of vocabulary
  - the 17th coordinate of w represents **strength** of w's association with vocabulary word 17

aardvark abacus abandoned abbot abduct above zygote zymurgy

(0, 0, 3, 1, 0, 7, . . . 1, 0)

how might you measure this?

- how often words appear next to each other
- how often words appear near each other
- how often words are syntactically linked

Words as Vectors

- Represent each word **type** w by a point in k-dimensional space
  - e.g., k is size of vocabulary
  - the 17th coordinate of w represents **strength** of w's association with vocabulary word 17

= party

aardvark abacus abandoned abbot abduct above ... zygote zymurgy

(0, 0, 3, 1, 0, 7, . . . 1, 0)

how might you measure this?

- how often words appear next to each other
- how often words appear near each other
- how often words are syntactically linked
- should correct for commonness of word (e.g., "above")

Words as Vectors

- Represent each word **type** w by a point in k-dimensional space
  - e.g., k is size of vocabulary
  - the 17th coordinate of w represents **strength** of w's association with vocabulary word 17

aardvark abacus abandoned abbot abduct above ... zygote zymurgy

(0, 0, 3, 1, 0, 7, . . . 1, 0)

Words as Vectors

- Represent each word **type** w by a point in k-dimensional space
  - e.g., k is size of vocabulary
  - the 17th coordinate of w represents **strength** of w's association with vocabulary word 17

aardvark abacus abandoned abbot abduct above ... zygote zymurgy

(0, 0, 3, 1, 0, 7, . . . 1, 0)

- Plot all word types in k-dimensional space

Words as Vectors

- Represent **each word type** w by a point in k-dimensional space
  - e.g., k is size of vocabulary
  - the 17th coordinate of w represents **strength** of w's association with vocabulary word 17

aardvark abacus abandoned abbot abduct above zygote zymurgy

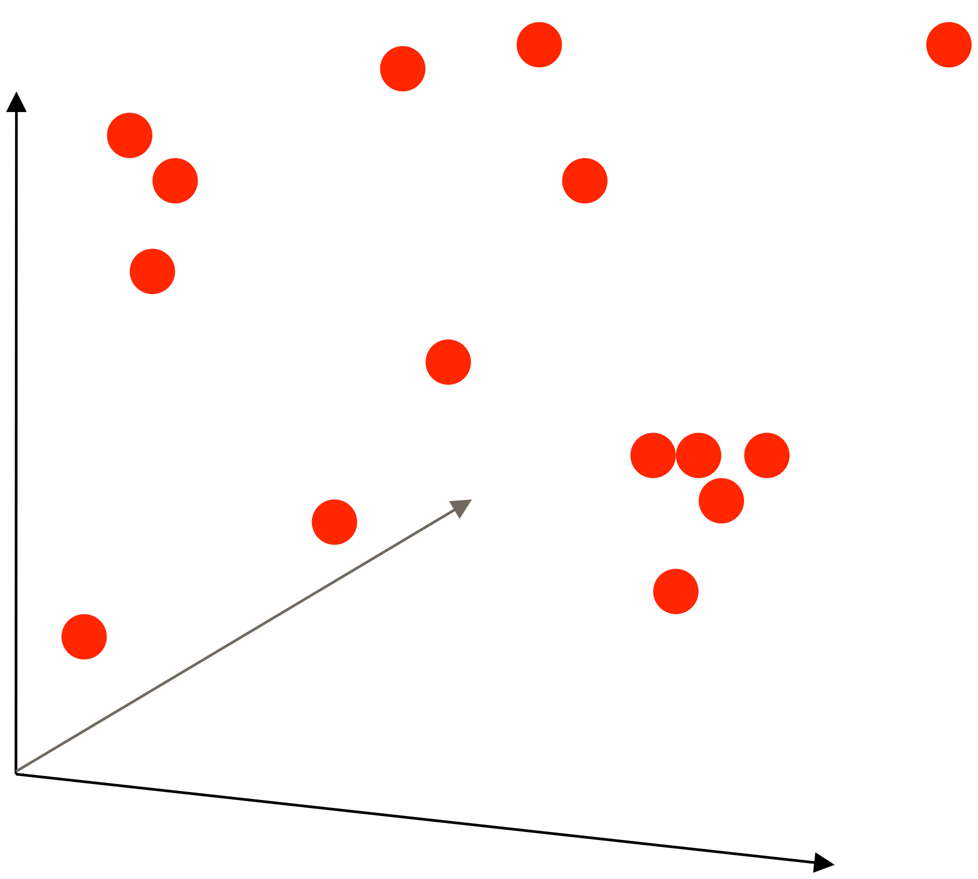
(0, 0, 3, 1, 0, 7, . . . 1, 0)

- Plot all word types in k-dimensional space
- Look for **clusters** of close-together types

Learning Classes by Clustering

- Plot all word types in k-dimensional space
- Look for **clusters** of close-together types

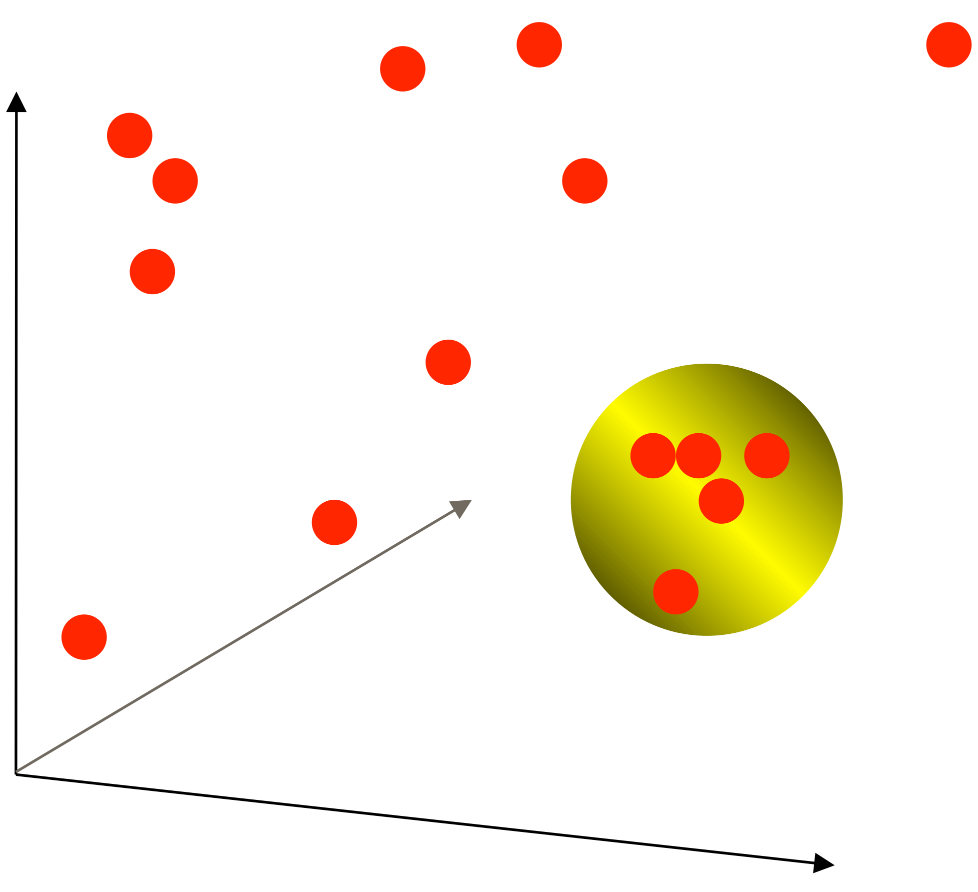
Plot in k dimensions (here k=3)



Learning Classes by Clustering

- Plot all word types in k-dimensional space
- Look for **clusters** of close-together types

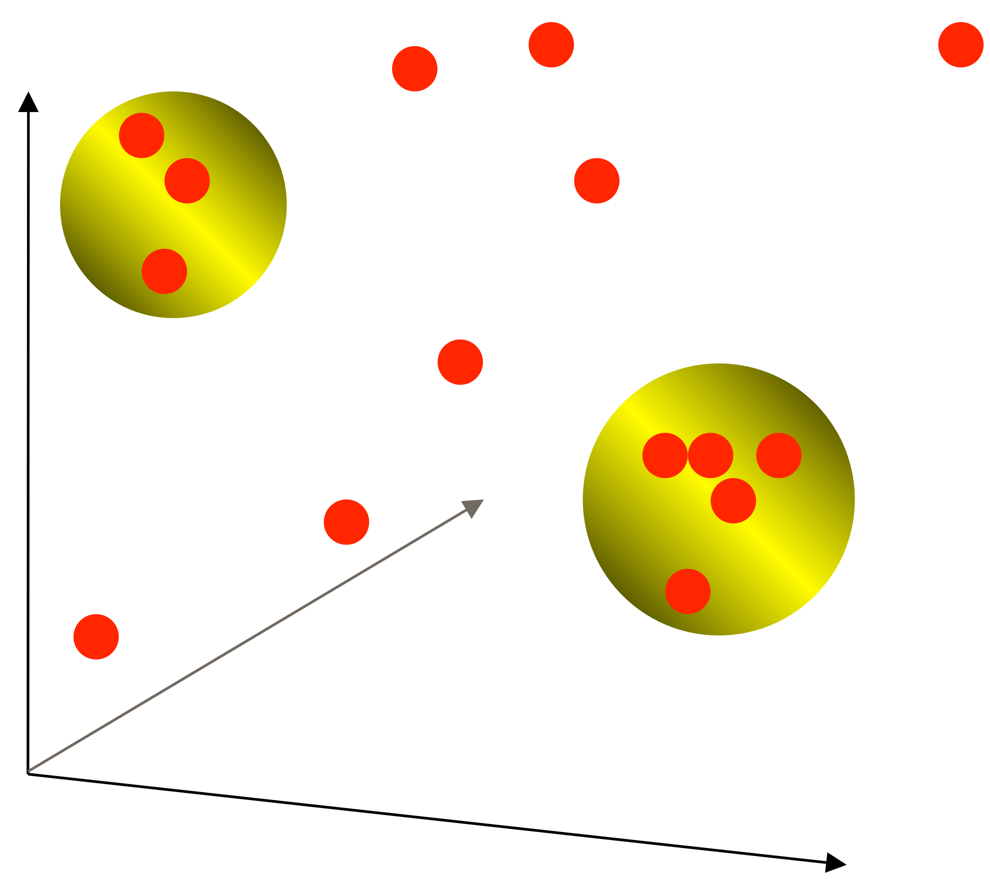
Plot in k dimensions (here k=3)



Learning Classes by Clustering

- Plot all word types in k-dimensional space
- Look for **clusters** of close-together types

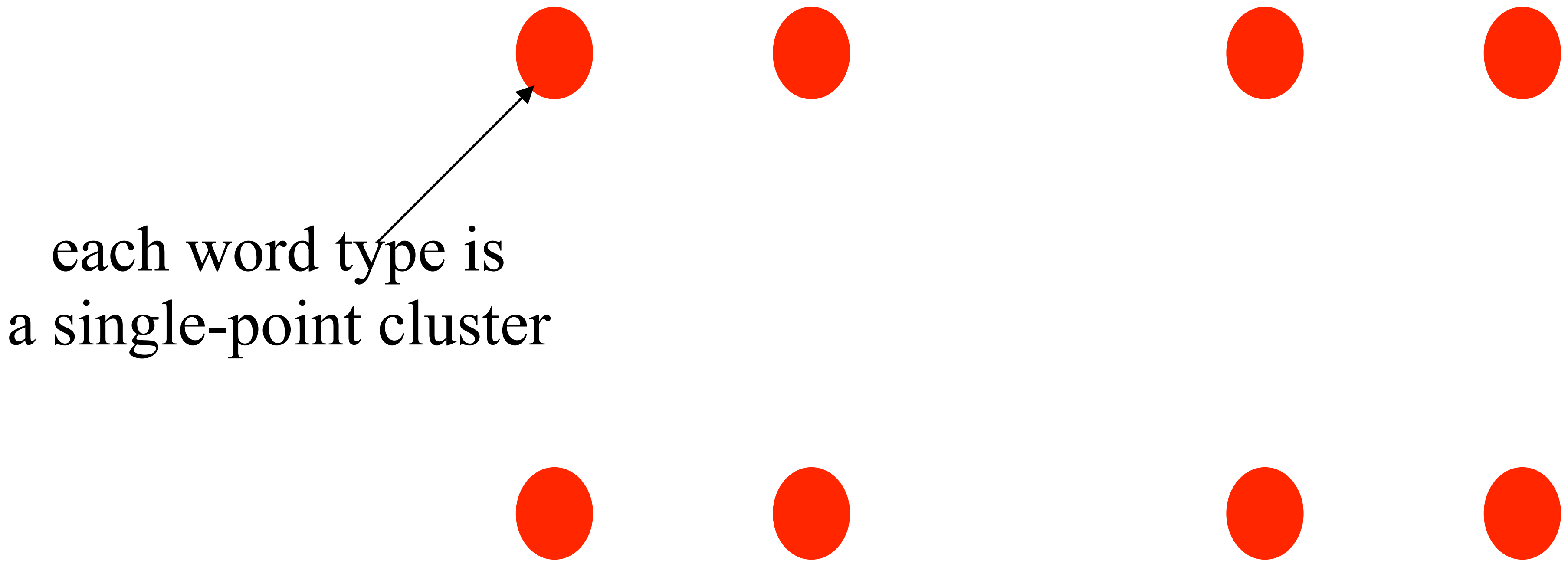
Plot in k dimensions (here k=3)



Bottom-Up Clustering

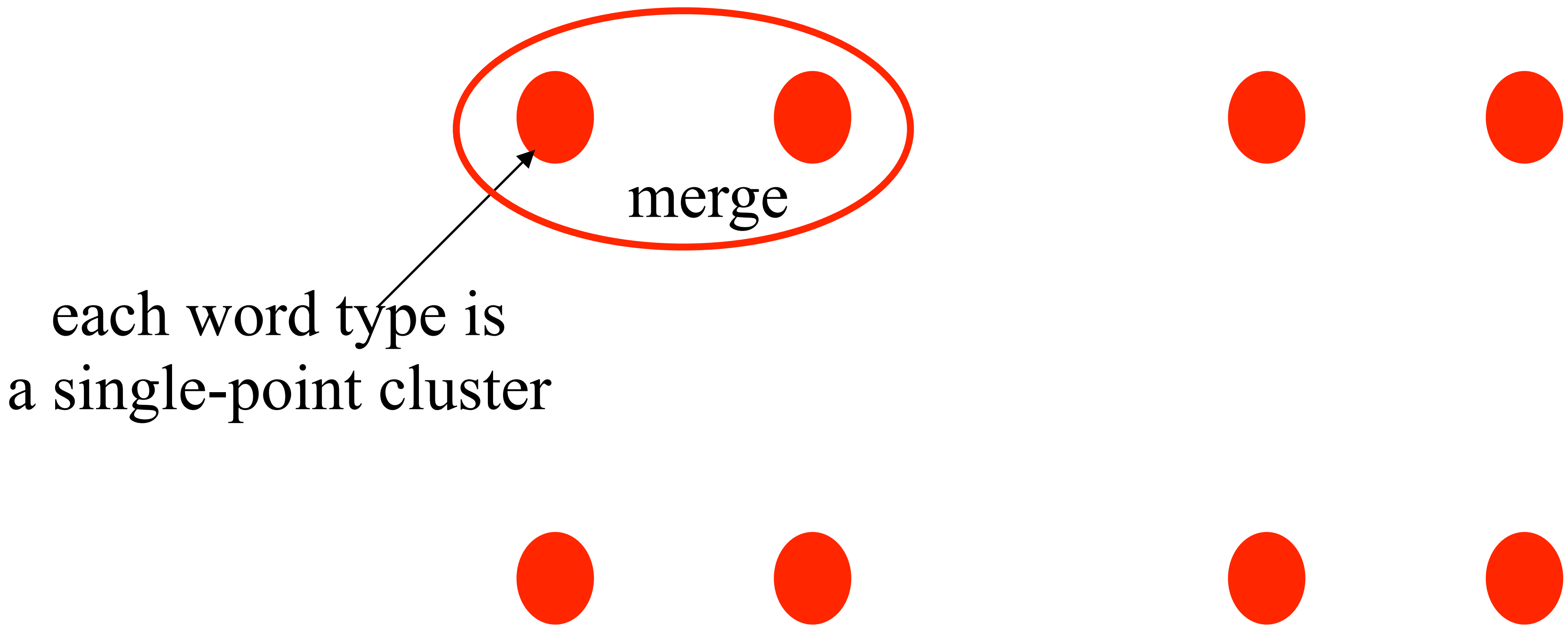
- Start with one cluster per point
- Repeatedly merge 2 closest clusters
  - Single-link: dist(A,B) = min dist(a,b) for $a \in A$, $b \in B$
  - Complete-link: dist(A,B) = max dist(a,b) for $a \in A$, $b \in B$

Bottom-Up Clustering – Single-Link

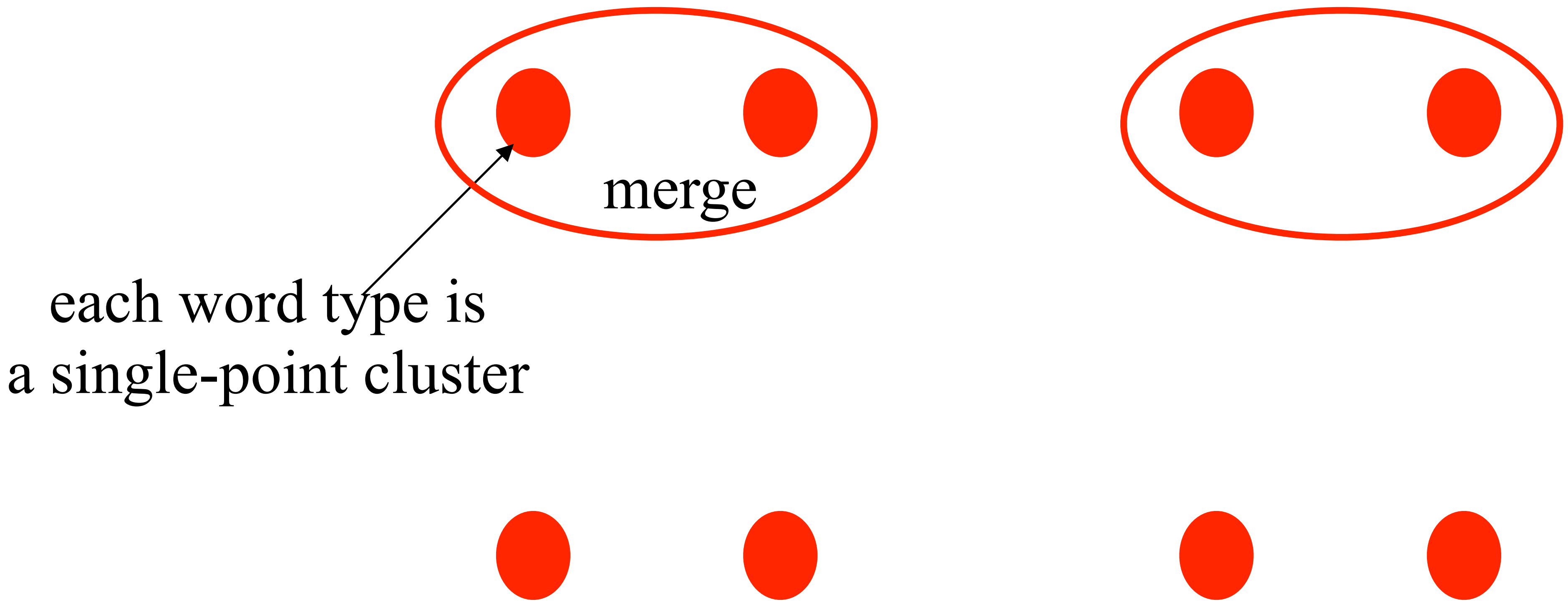


example from Manning & Schütze

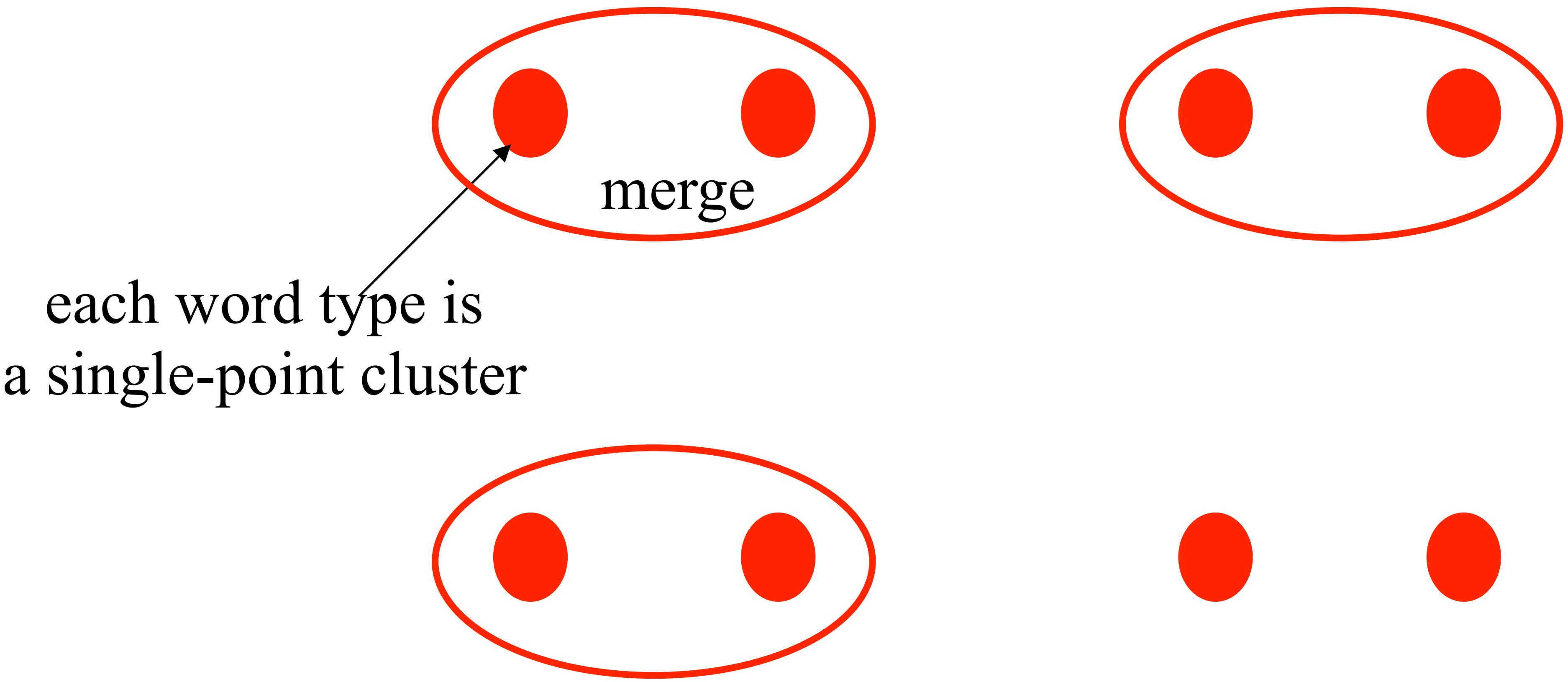
Bottom-Up Clustering – Single-Link



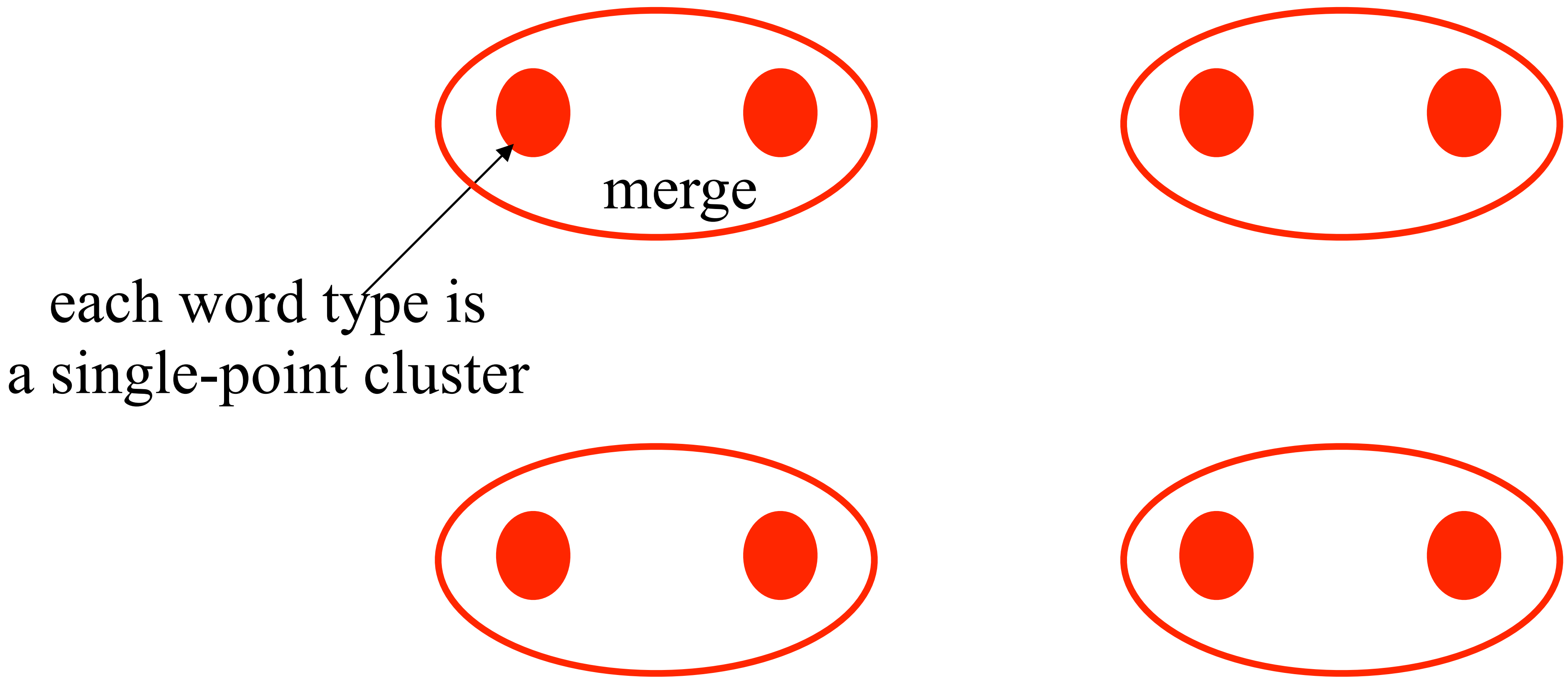
Bottom-Up Clustering – Single-Link



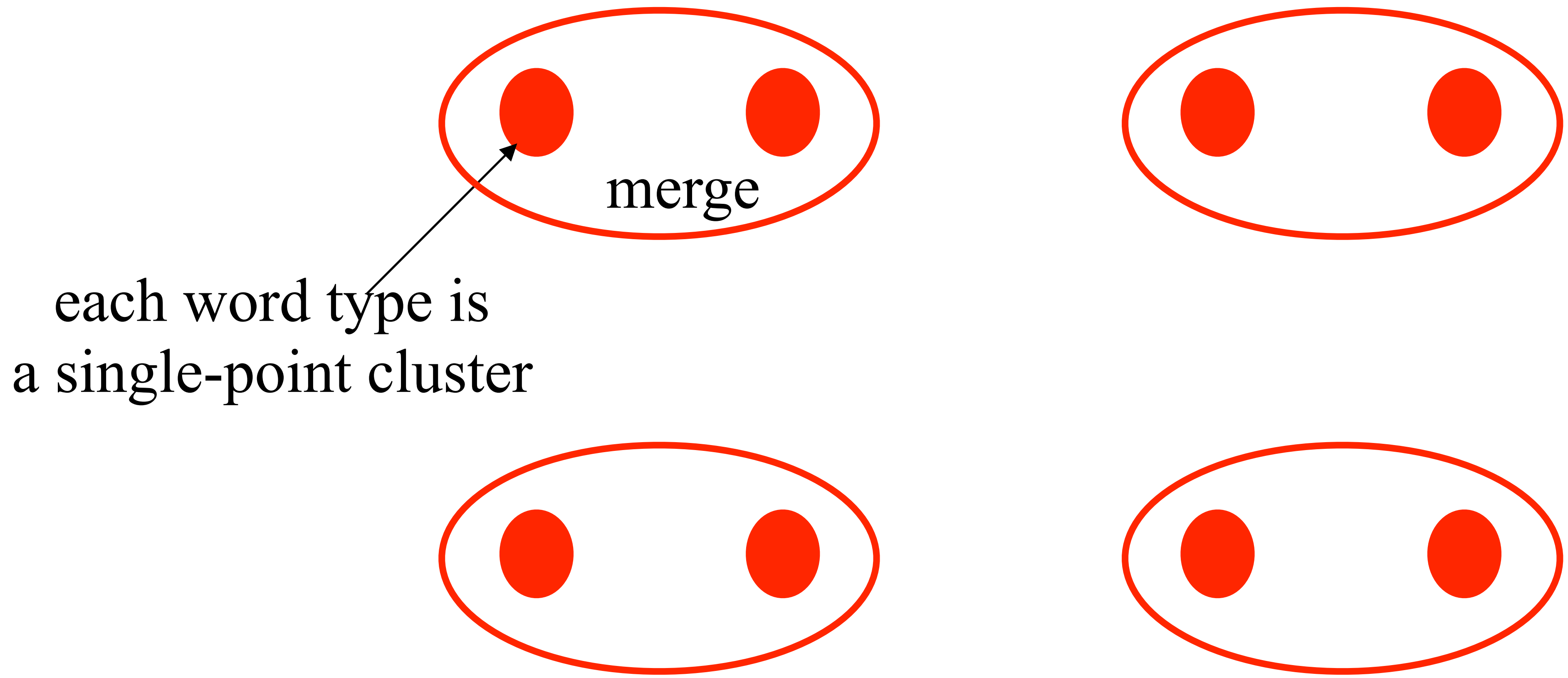
Bottom-Up Clustering – Single-Link



Bottom-Up Clustering – Single-Link



Bottom-Up Clustering – Single-Link

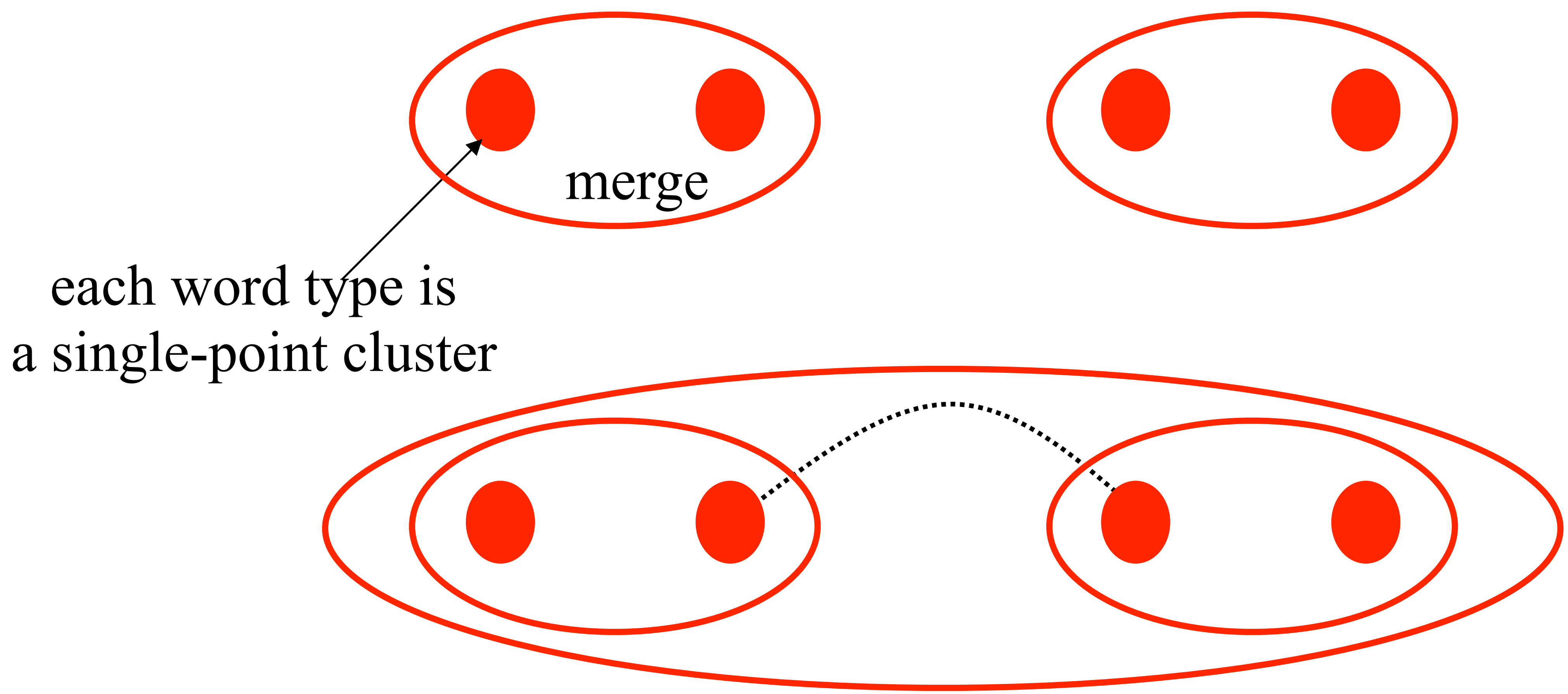


Again, merge closest pair of clusters:

Single-link: clusters are close if any of their points are

dist(A,B) = min dist(a,b) for a∈A, b∈B

Bottom-Up Clustering – Single-Link

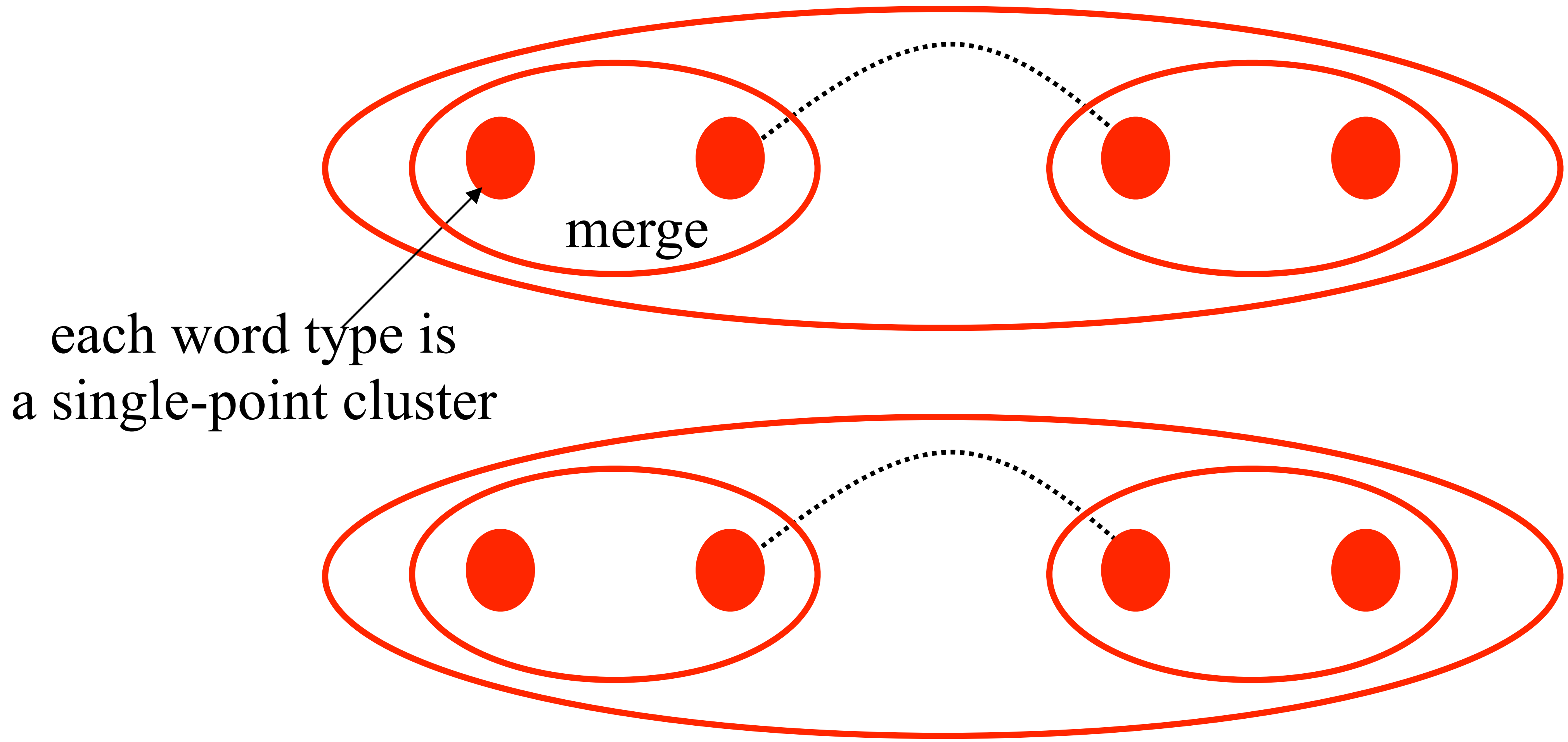


Again, merge closest pair of clusters:

Single-link: clusters are close if any of their points are

dist(A,B) = min dist(a,b) for a∈A, b∈B

Bottom-Up Clustering – Single-Link

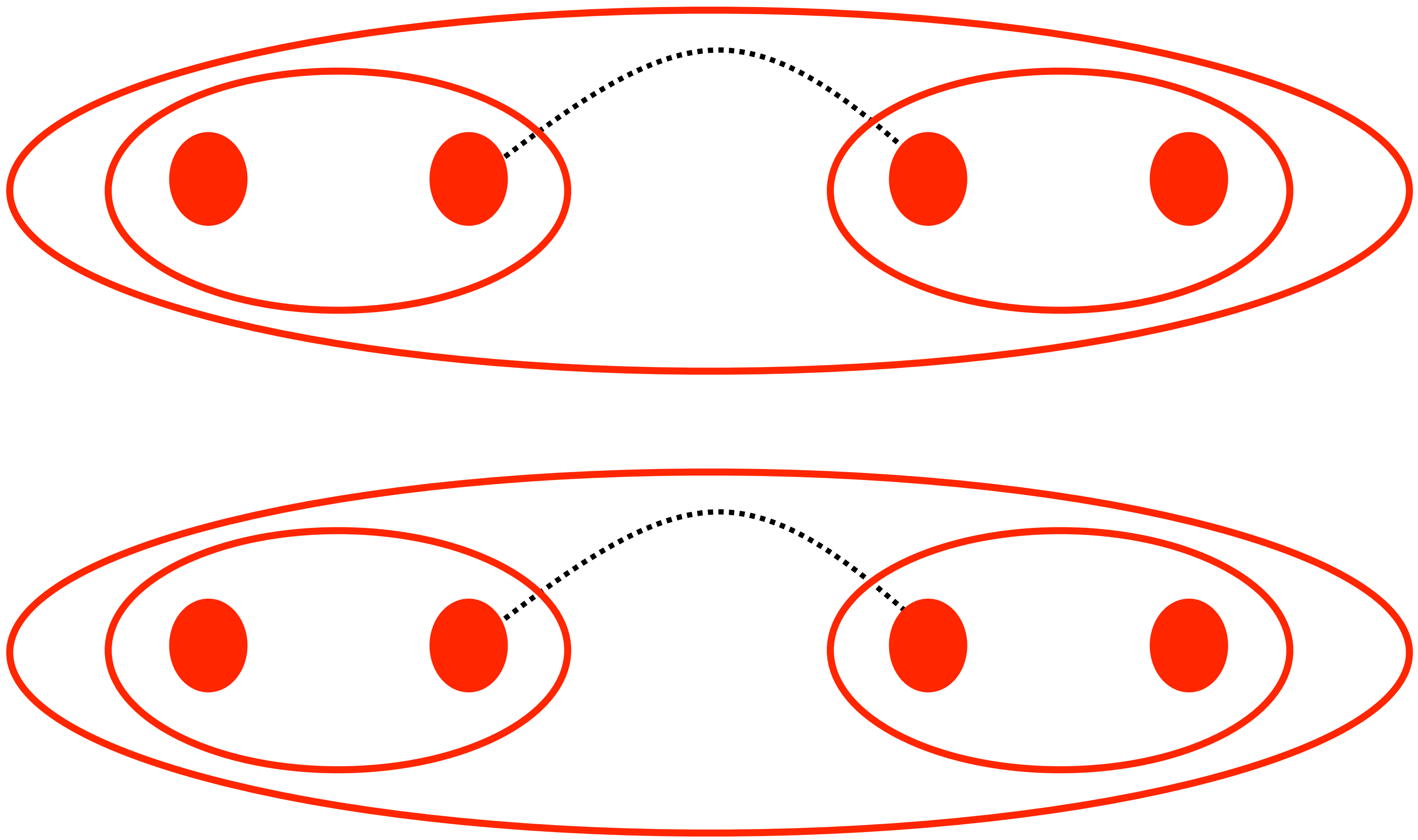


Again, merge closest pair of clusters:

Single-link: clusters are close if any of their points are

dist(A,B) = min dist(a,b) for $a \in A$, $b \in B$

Bottom-Up Clustering – Single-Link



Again, merge closest pair of clusters:

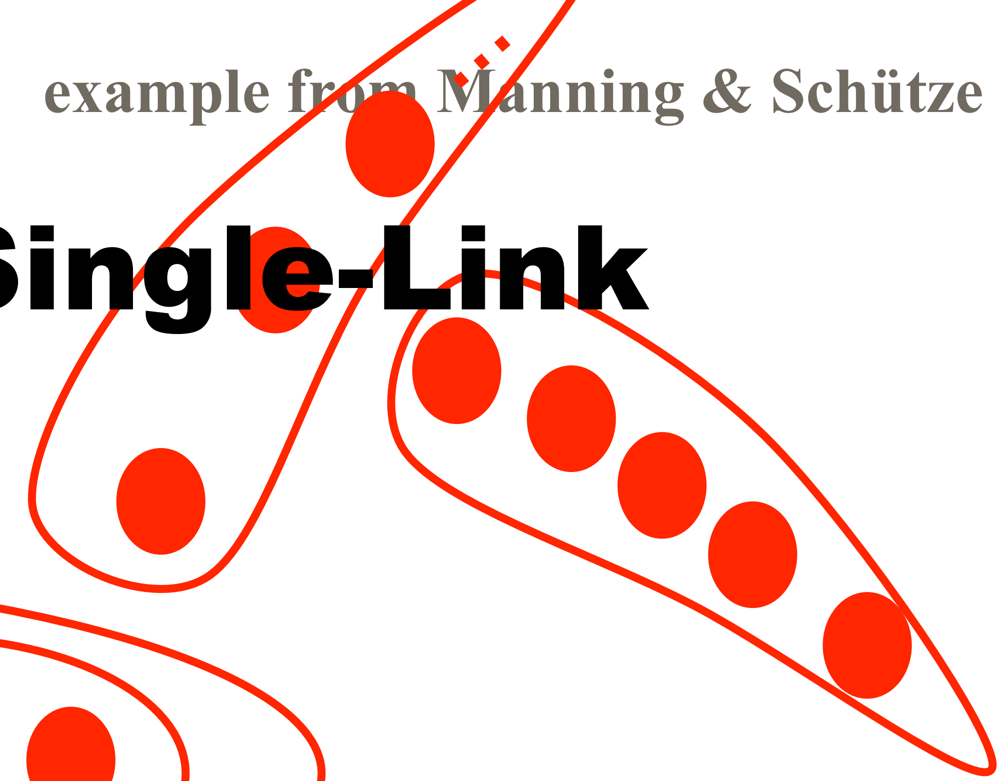
Single-link: clusters are close if any of their points are

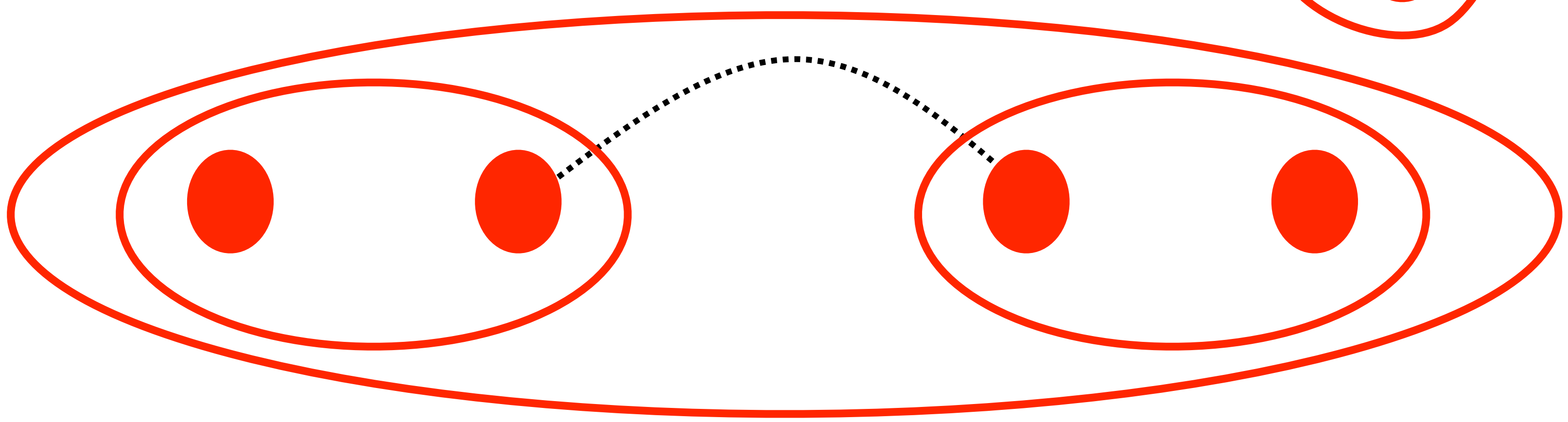
dist(A,B) = min dist(a,b) for $a \in A$, $b \in B$

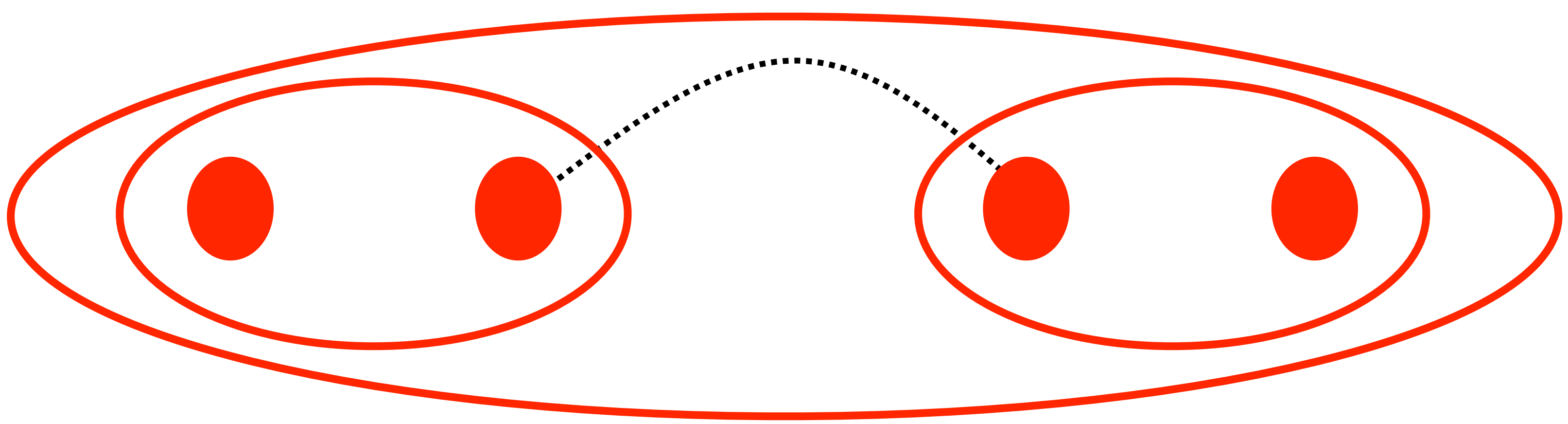
Fast, but tend to get long, stringy, meandering clusters

example from Manning & Schütze

Bottom-Up Clustering – Single-Link







Again, merge closest pair of clusters:

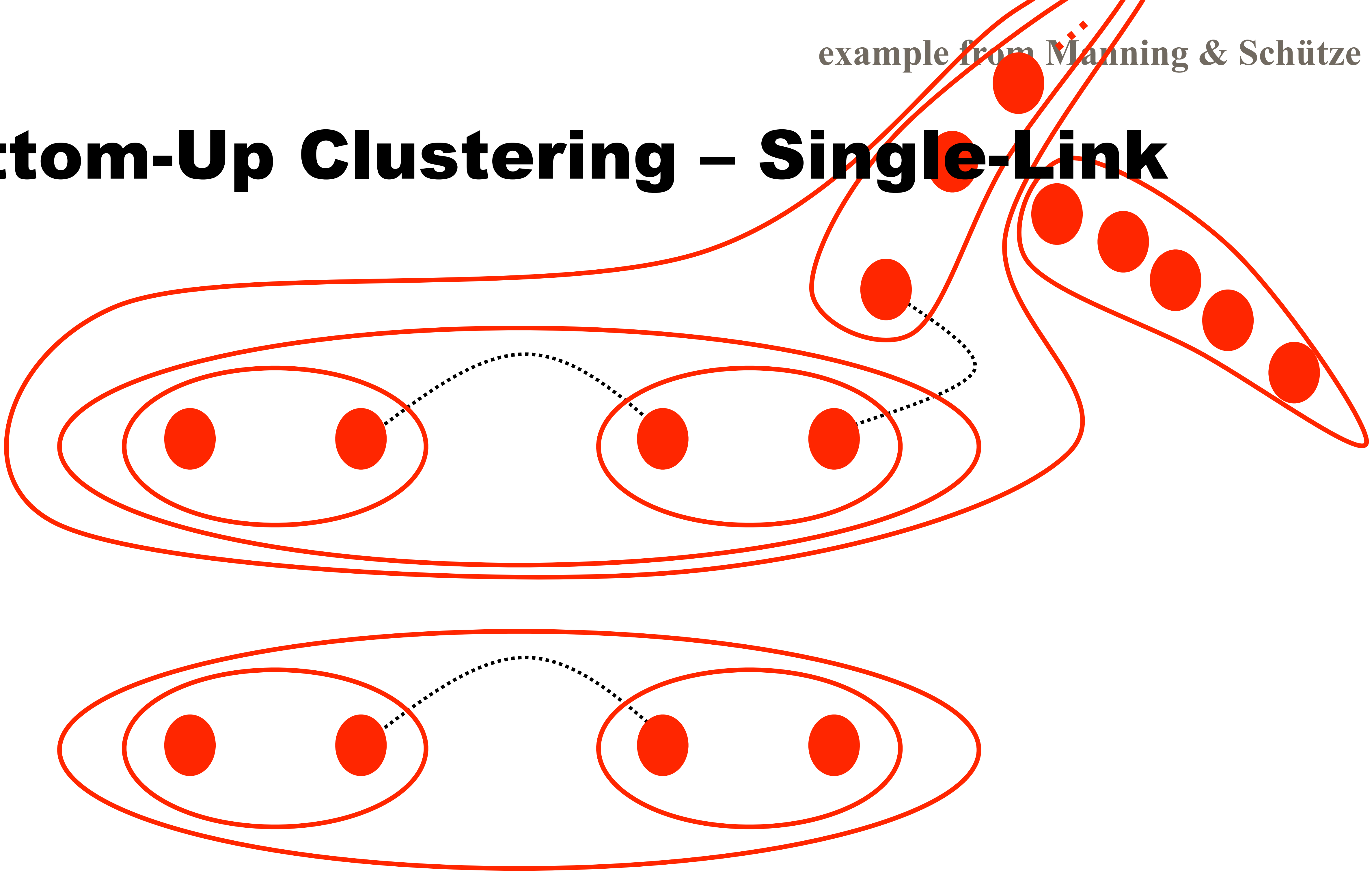
Single-link: clusters are close if any of their points are

dist(A,B) = min dist(a,b) for $a \in A$, $b \in B$

Fast, but tend to get long, stringy, meandering clusters

example from Manning & Schütze

Bottom-Up Clustering – Single-Link



Again, merge closest pair of clusters:

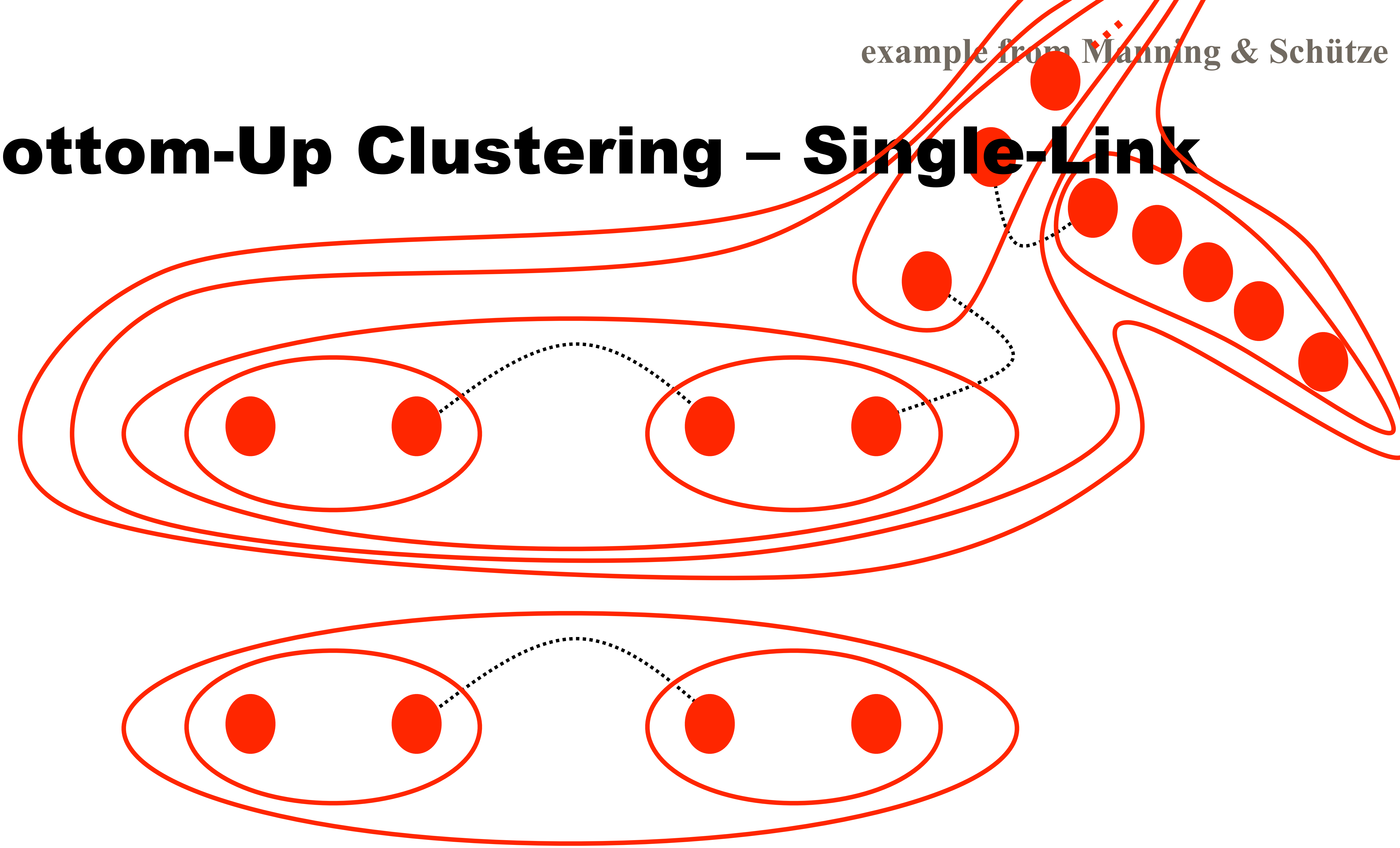
Single-link: clusters are close if any of their points are

dist(A,B) = min dist(a,b) for a∈A, b∈B

Fast, but tend to get long, stringy, meandering clusters

example from Manning & Schütze

Bottom-Up Clustering – Single-Link



Again, merge closest pair of clusters:

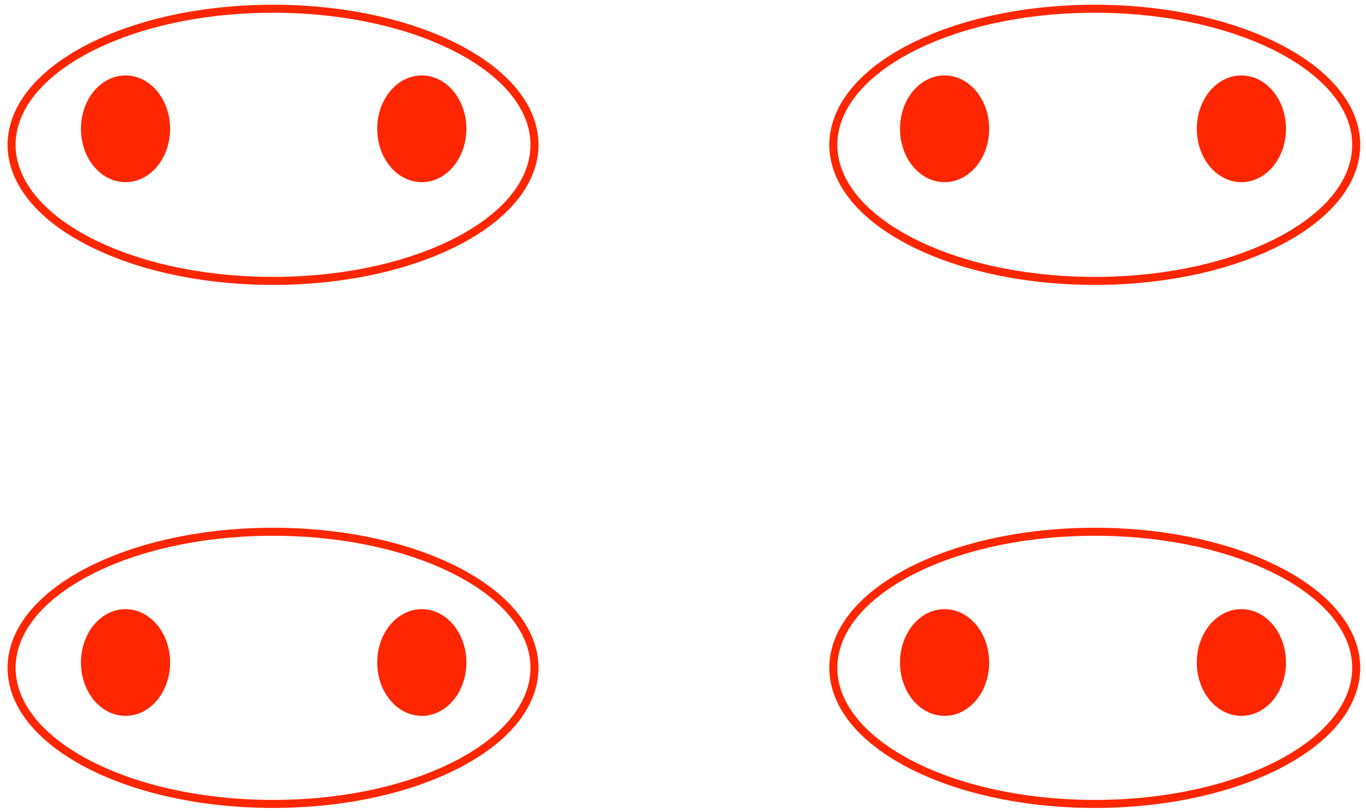
Single-link: clusters are close if any of their points are

dist(A,B) = min dist(a,b) for a∈A, b∈B

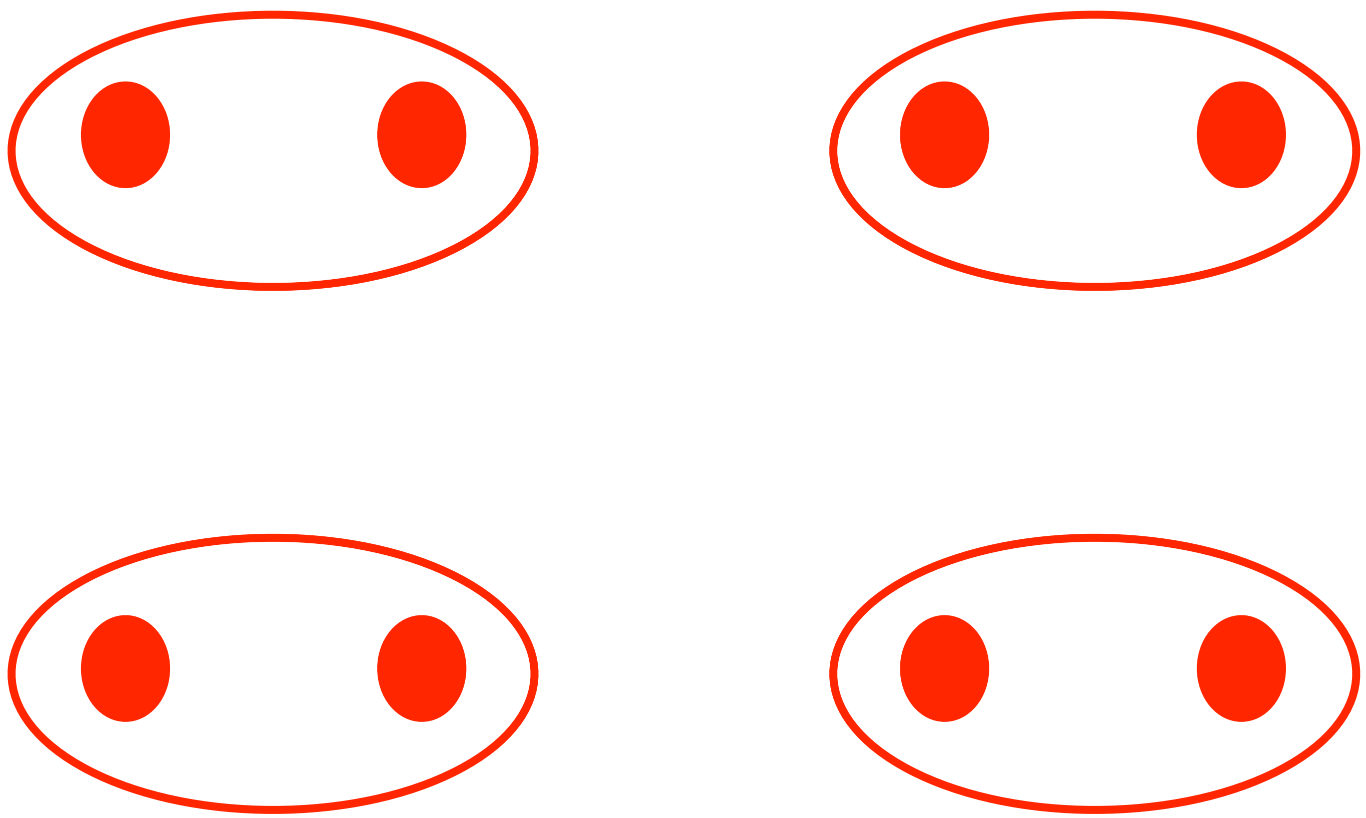
Fast, but tend to get long, stringy, meandering clusters

example from Manning & Schütze

Bottom-Up Clustering – Complete-Link



Bottom-Up Clustering – Complete-Link

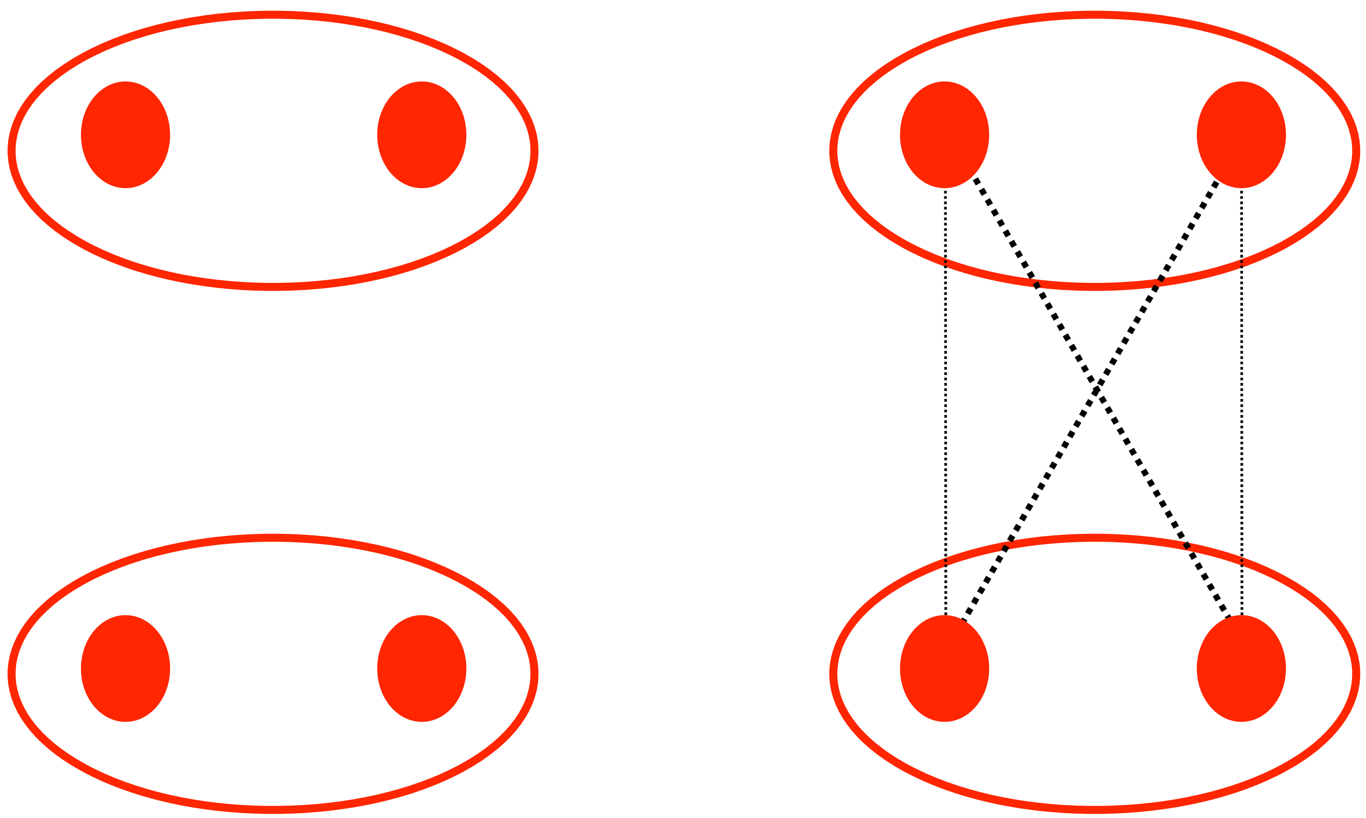


Again, merge closest pair of clusters:

Complete-link: clusters are close only if all of their points are

dist(A,B) = max dist(a,b) for $a \in A$, $b \in B$

Bottom-Up Clustering – Complete-Link

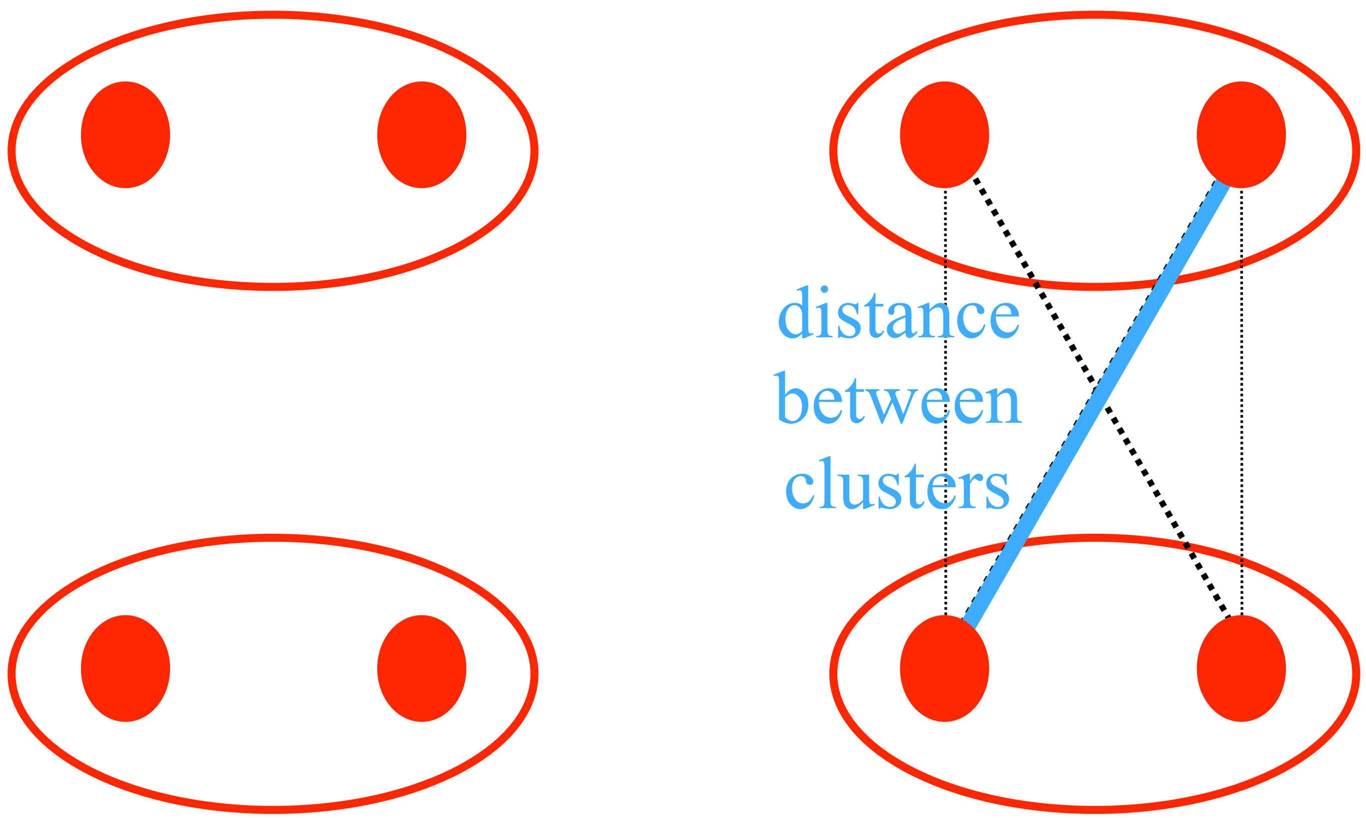


Again, merge closest pair of clusters:

Complete-link: clusters are close only if all of their points are

$$\text{dist}(A,B) = \max \text{dist}(a,b) \text{ for } a \in A,\ b \in B$$

Bottom-Up Clustering – Complete-Link

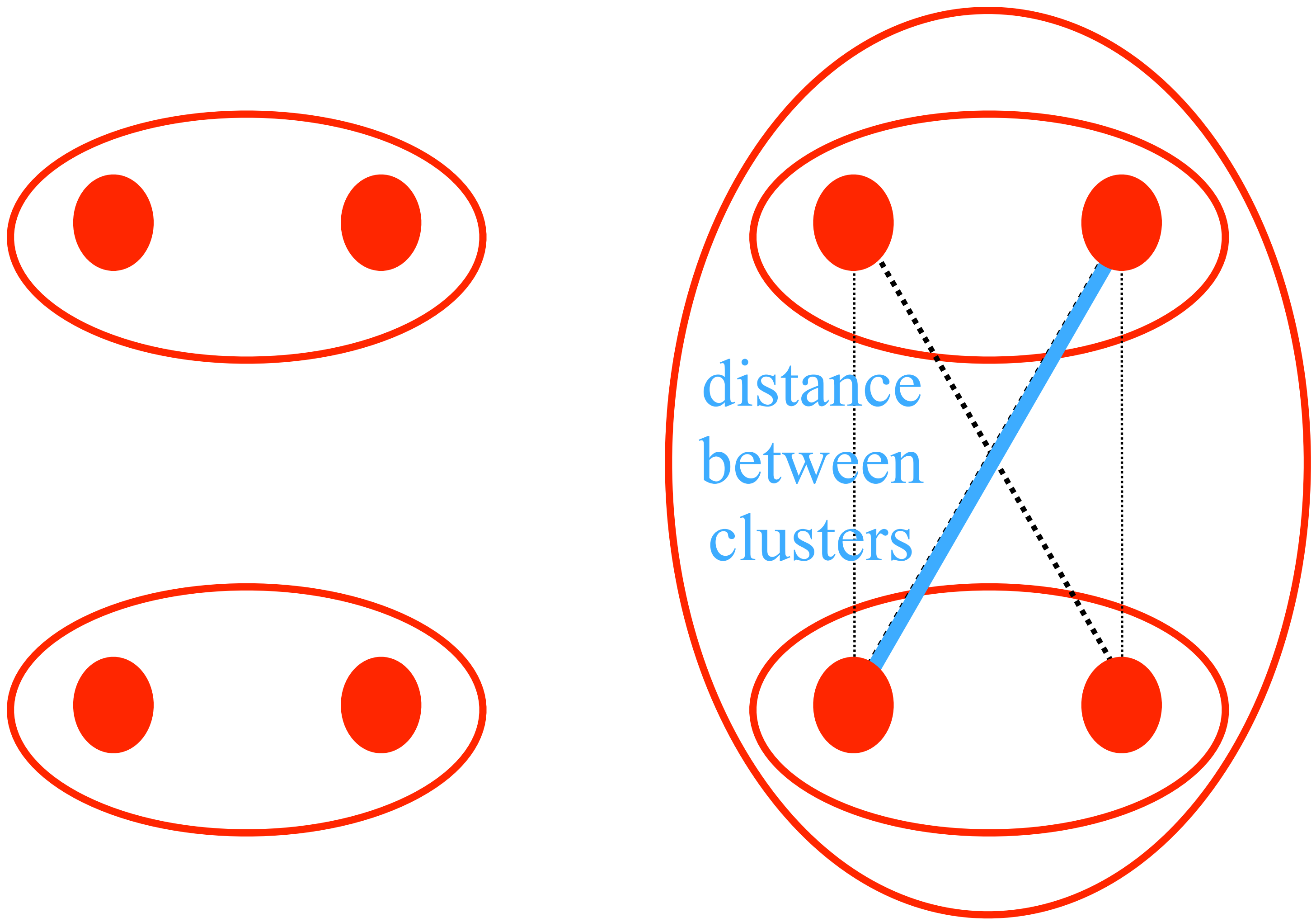


Again, merge closest pair of clusters:

Complete-link: clusters are close only if all of their points are

dist(A,B) = max dist(a,b) for $a \in A$, $b \in B$

Bottom-Up Clustering – Complete-Link



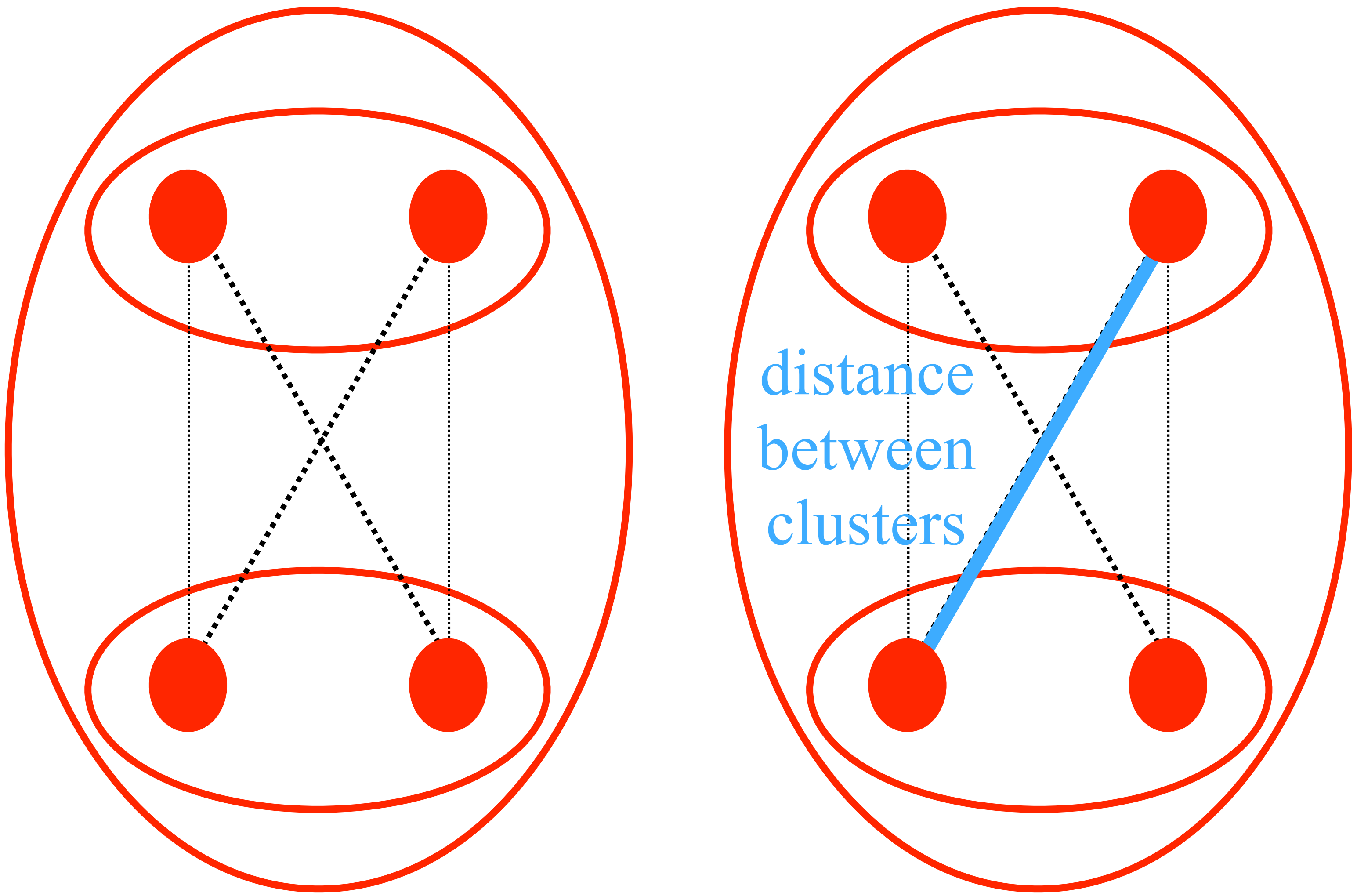
Again, merge closest pair of clusters:

Complete-link: clusters are close only if all of their points are

dist(A,B) = max dist(a,b) for a∈A, b∈B

example from Manning & Schütze

Bottom-Up Clustering – Complete-Link

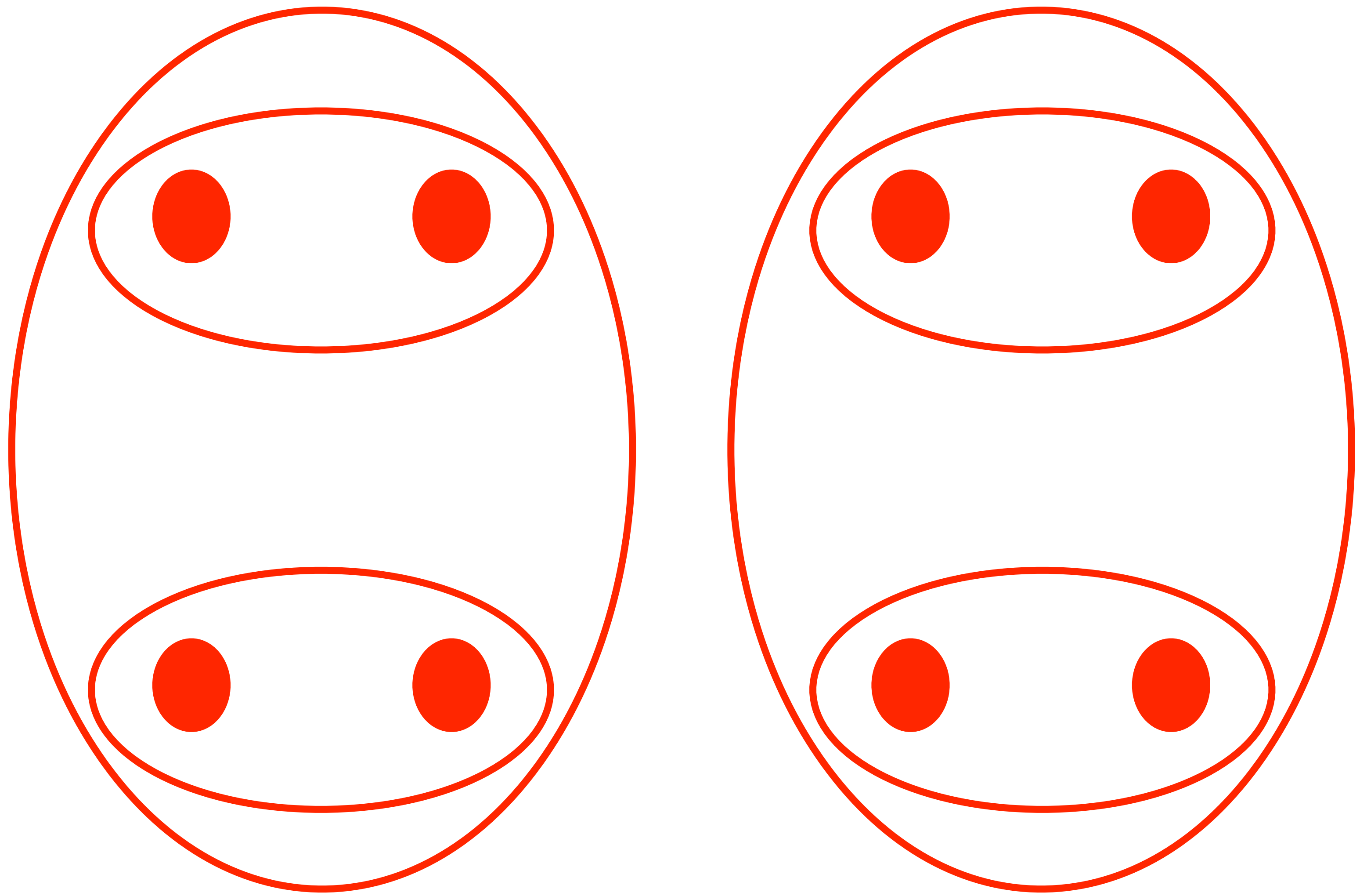


Again, merge closest pair of clusters:

Complete-link: clusters are close only if all of their points are

dist(A,B) = max dist(a,b) for a∈A, b∈B

Bottom-Up Clustering – Complete-Link



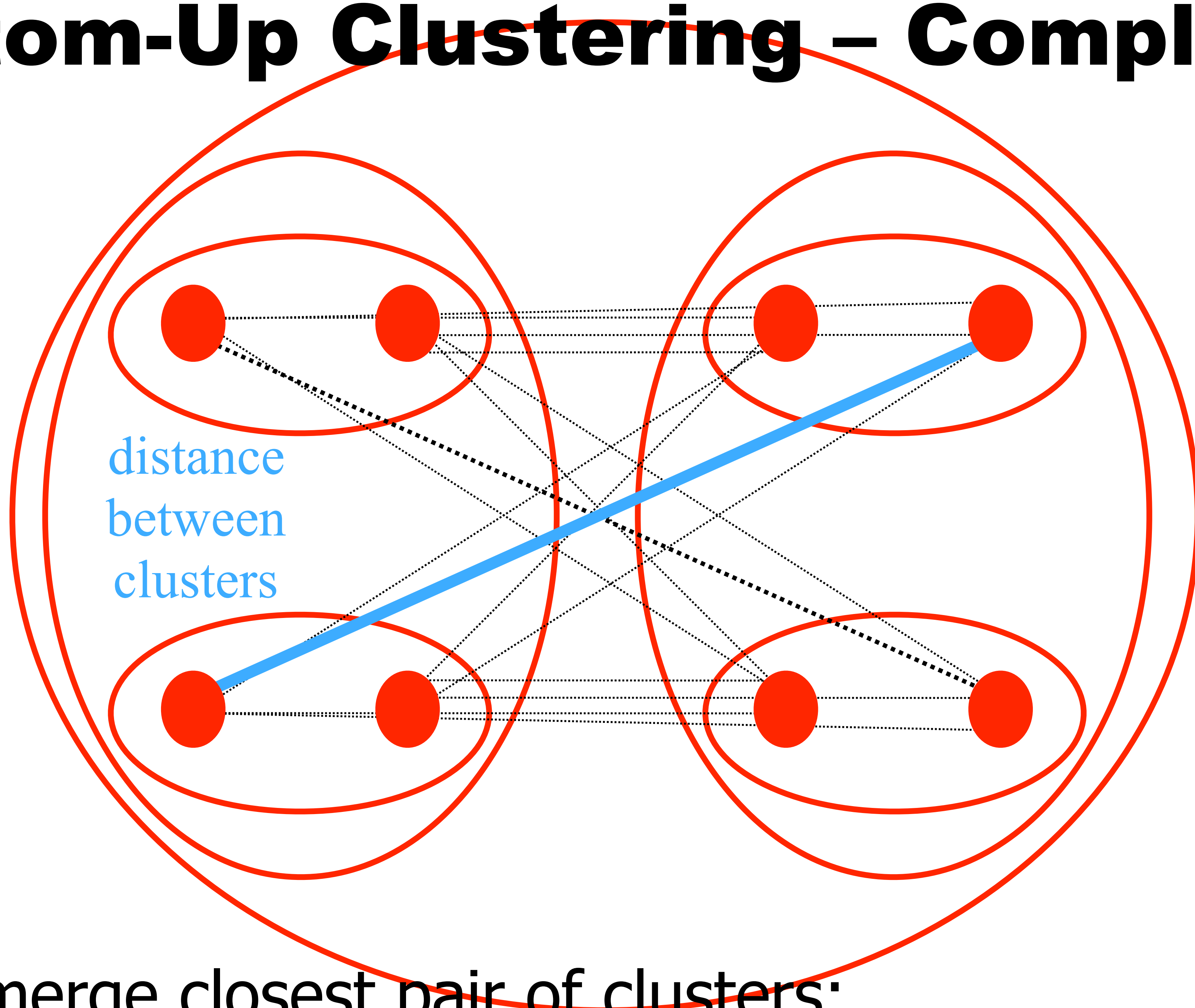
Again, merge closest pair of clusters:

Complete-link: clusters are close only if all of their points are

$\text{dist}(A,B) = \max \text{dist}(a,b)$ for $a \in A$, $b \in B$

Slow to find closest pair – need quadratically many distances

Bottom-Up Clustering – Complete-Link



Again, merge closest pair of clusters:

Complete-link: clusters are close only if all of their points are

dist(A,B) = max dist(a,b) for a∈A, b∈B

Slow to find closest pair – need quadratically many distances

Bottom-Up Clustering

- Start with one cluster per point
- Repeatedly merge 2 closest clusters
  - Single-link: dist(A,B) = min dist(a,b) for a∈A, b∈B
  - Complete-link: dist(A,B) = max dist(a,b) for a∈A, b∈B
    - too slow to update cluster distances after each merge; but ∃ alternatives!

Bottom-Up Clustering

- Start with one cluster per point
- Repeatedly merge 2 closest clusters
  - Single-link: dist(A,B) = min dist(a,b) for $a \in A$, $b \in B$
  - Complete-link: dist(A,B) = max dist(a,b) for $a \in A$, $b \in B$
    - too slow to update cluster distances after each merge; but ∃ alternatives!
  - Average-link: dist(A,B) = mean dist(a,b) for $a \in A$, $b \in B$
  - Centroid-link: dist(A,B) = dist(mean(A),mean(B))

Bottom-Up Clustering

- Start with one cluster per point
- Repeatedly merge 2 closest clusters
  - Single-link: dist(A,B) = min dist(a,b) for $a \in A$, $b \in B$
  - Complete-link: dist(A,B) = max dist(a,b) for $a \in A$, $b \in B$
    - too slow to update cluster distances after each merge; but ∃ alternatives!
  - Average-link: dist(A,B) = mean dist(a,b) for $a \in A$, $b \in B$
  - Centroid-link: dist(A,B) = dist(mean(A),mean(B))
- Stop when clusters are "big enough"
  - e.g., provide adequate support for backoff (on a development corpus)

Bottom-Up Clustering

- Start with one cluster per point
- Repeatedly merge 2 closest clusters
  - Single-link: dist(A,B) = min dist(a,b) for a∈A, b∈B
  - Complete-link: dist(A,B) = max dist(a,b) for a∈A, b∈B
    - too slow to update cluster distances after each merge; but ∃ alternatives!
  - Average-link: dist(A,B) = mean dist(a,b) for a∈A, b∈B
  - Centroid-link: dist(A,B) = dist(mean(A),mean(B))
- Stop when clusters are "big enough"
  - e.g., provide adequate support for backoff (on a development corpus)
- Some flexibility in defining dist(a,b)
  - Might not be Euclidean distance; e.g., use vector angle

EM Clustering (for k clusters)

EM Clustering (for k clusters)

- EM algorithm
  - Viterbi version – called "k-means clustering"
  - Full EM version – called "Gaussian mixtures"

EM Clustering (for k clusters)

- EM algorithm
  - Viterbi version – called “k-means clustering”
  - Full EM version – called “Gaussian mixtures”

EM Clustering (for k clusters)

- EM algorithm
  - Viterbi version – called "k-means clustering"
  - Full EM version – called "Gaussian mixtures"

- **Expectation step:** Use current parameters (and observations) to reconstruct hidden structure

EM Clustering (for k clusters)

- EM algorithm
  - Viterbi version – called "k-means clustering"
  - Full EM version – called "Gaussian mixtures"

- **Expectation step:** Use current parameters (and observations) to reconstruct hidden structure
- **Maximization step:** Use that hidden structure (and observations) to reestimate parameters

EM Clustering (for k clusters)

- EM algorithm
  - Viterbi version – called "k-means clustering"
  - Full EM version – called "Gaussian mixtures"

- **Expectation step:** Use current parameters (and observations) to reconstruct hidden structure
- **Maximization step:** Use that hidden structure (and observations) to reestimate parameters

EM Clustering (for k clusters)

- EM algorithm
  - Viterbi version – called “k-means clustering”
  - Full EM version – called “Gaussian mixtures”

- **Expectation step:** Use current parameters (and observations) to reconstruct hidden structure
- **Maximization step:** Use that hidden structure (and observations) to reestimate parameters

- Parameters: k points representing cluster centers

EM Clustering (for k clusters)

- EM algorithm
  - Viterbi version – called "k-means clustering"
  - Full EM version – called "Gaussian mixtures"

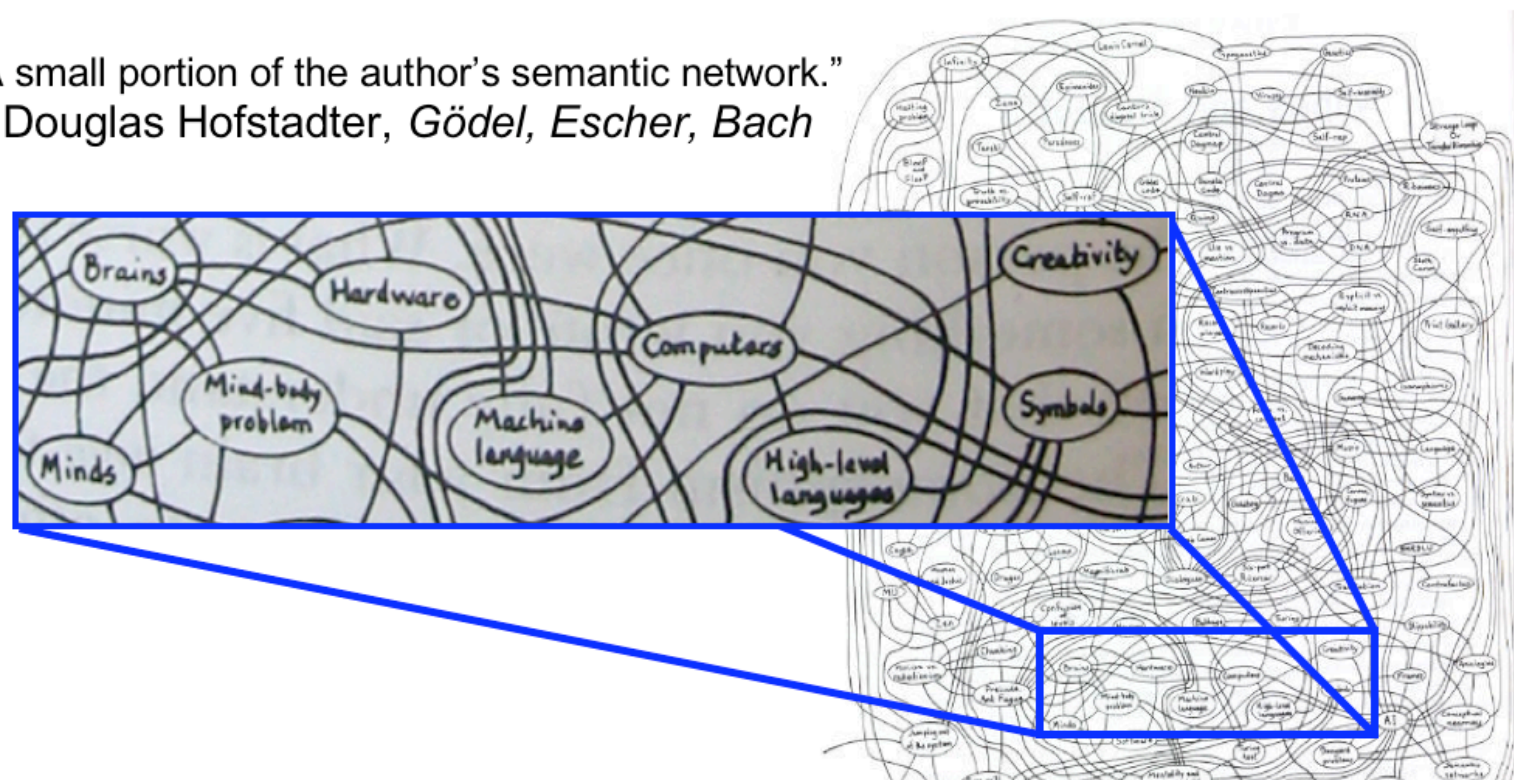
- **Expectation step:** Use current parameters (and observations) to reconstruct hidden structure
- **Maximization step:** Use that hidden structure (and observations) to reestimate parameters

- Parameters: k points representing cluster centers
- Hidden structure: for each data point (word type), which center generated it?

Learning syntactic patterns for automatic hypernym discovery

Rion Snow, Daniel Jurafsky, and Andrew Y. Ng.

- It has long been a goal of AI to automatically acquire structured knowledge directly from text, e.g, in the form of a semantic network.



"A small portion of the author's semantic network."
– Douglas Hofstadter, *Gödel, Escher, Bach*

We aim to classify whether a noun pair (X, Y) participates in one of the following semantic relationships:

Hypernymy (ancestor)

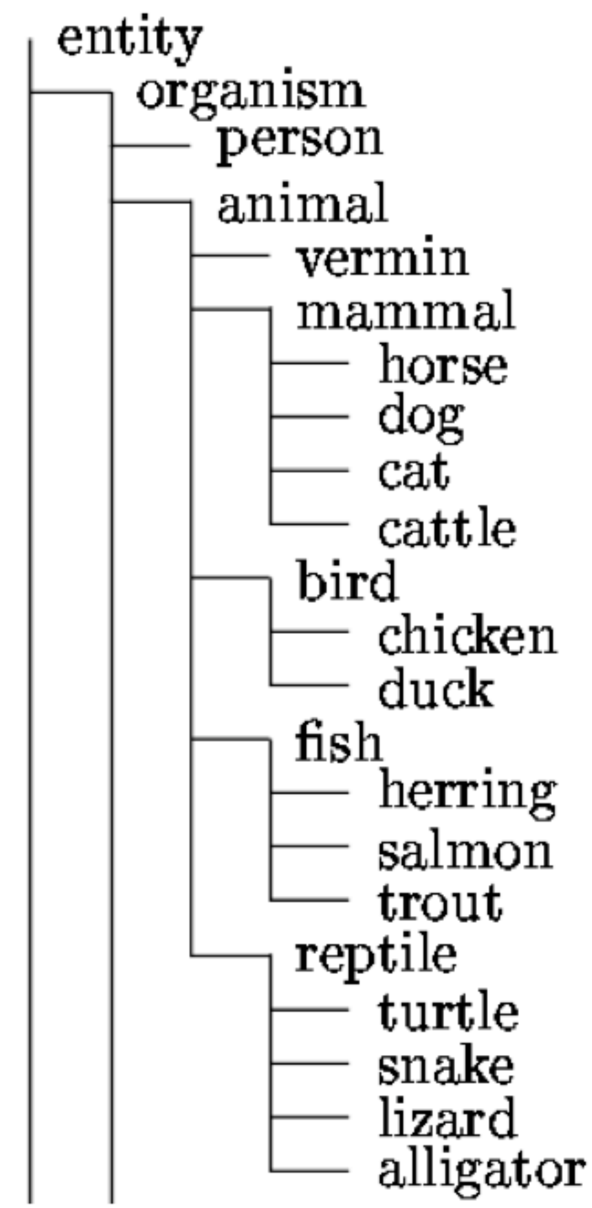
$Y \underset{H}{>} X$ if "X is a kind of Y".

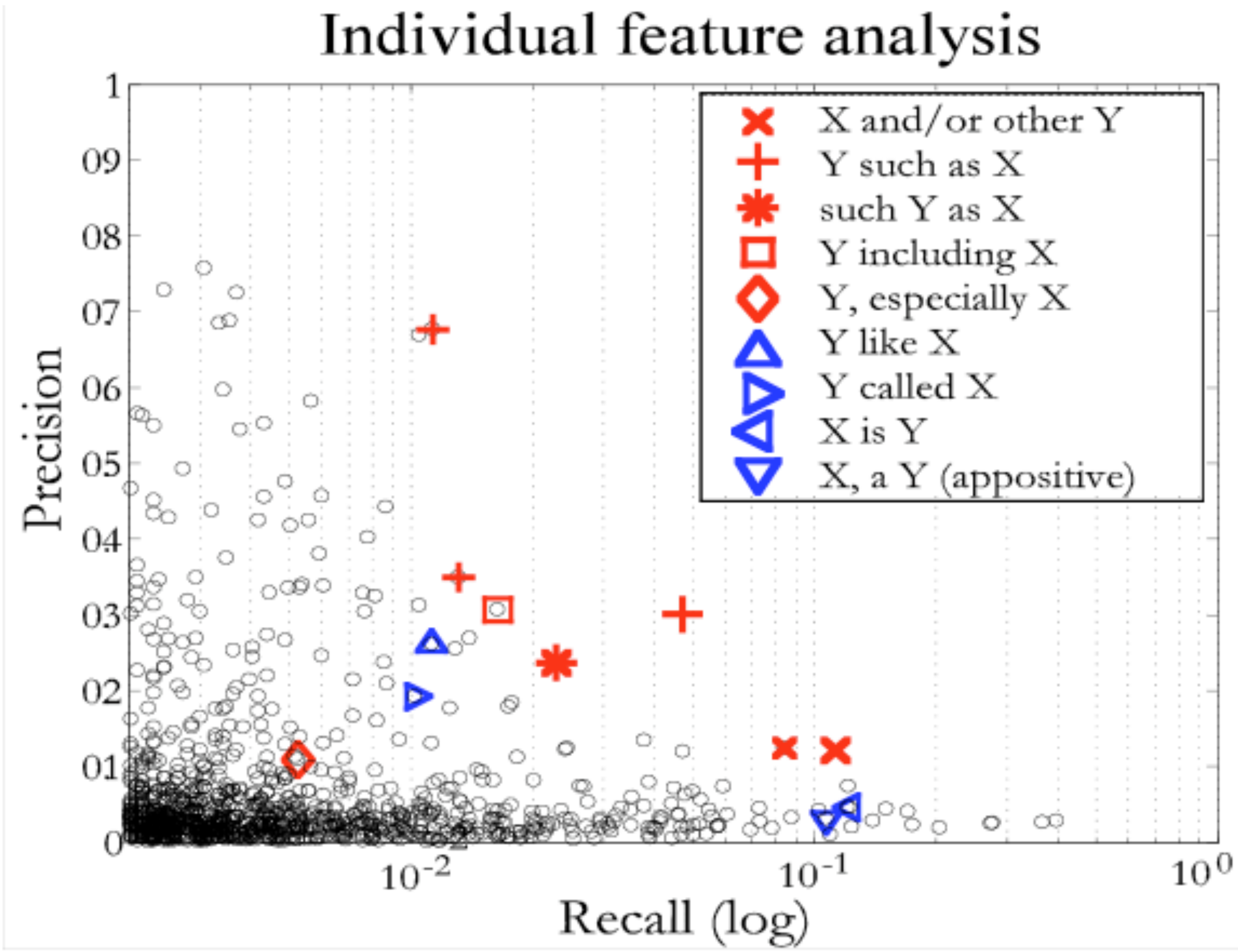
$$entity \underset{H}{>} organism \underset{H}{>} person$$

Coordinate Terms (taxonomic sisters)

$Y \underset{C}{\sim} X$ if X and Y possess a common hypernym, i.e. $\exists Z$ such that "X and Y are both kinds of Z."

$$horse \underset{C}{\sim} dog \underset{C}{\sim} cat$$

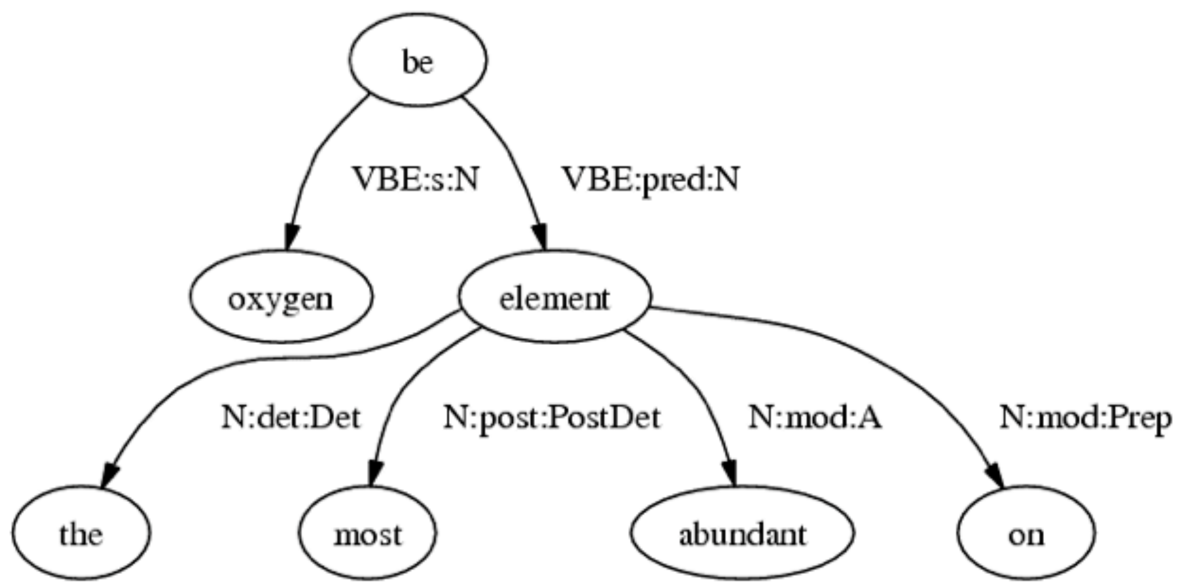




- Precision/recall for 69,592 classifiers (one per feature)
- Classifier *f* classifies noun pair ***x*** as hypernym iff $x_f > 0$
- **In red:** patterns originally proposed in (Hearst, 1992)

“Oxygen is the most abundant element on the moon.”

Dependency Graph:



Dependency Paths (for “oxygen / element”):

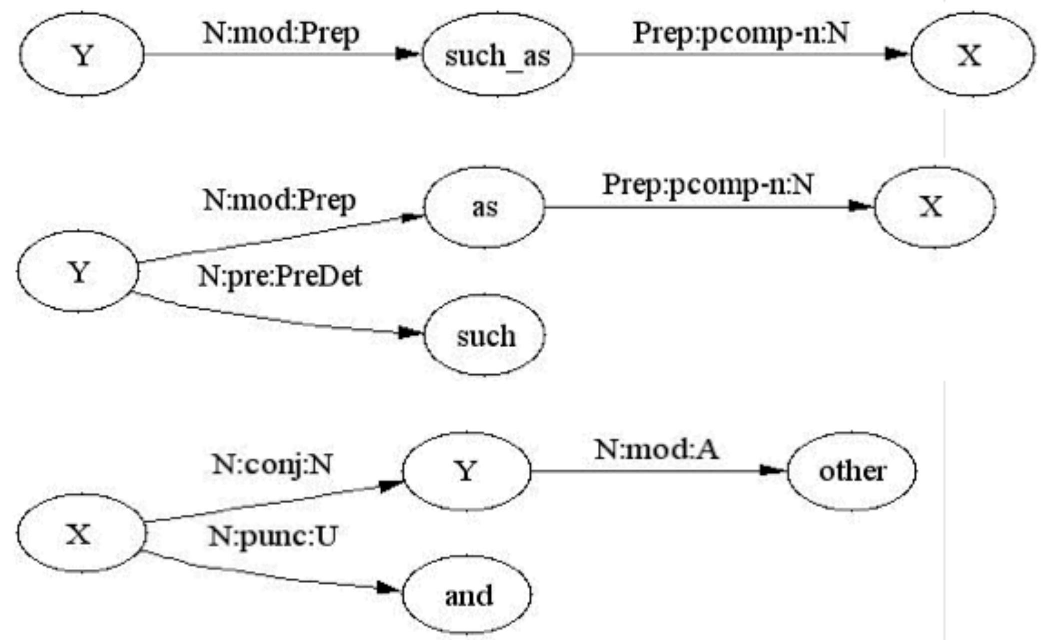
-N:s:VBE, “be” VBE:pred:N
-N:s:VBE, “be” VBE:pred:N,(the,Det:det:N)
-N:s:VBE, “be” VBE:pred:N,(most,PostDet:post:N)
-N:s:VBE, “be” VBE:pred:N,(abundant,A:mod:N)
-N:s:VBE, “be” VBE:pred:N,(on,Prep:mod:N)

Rediscovering Hearst's Patterns

Y such as X…

Such Y as X…

X… and other Y



Proposed in (Hearst, 1992) and used in (Caraballo, 2001), (Widdows, 2003), and others – but what about the rest of the lexico-syntactic pattern space?

Example: Using the "Y called X" Pattern for Hypernym Acquisition

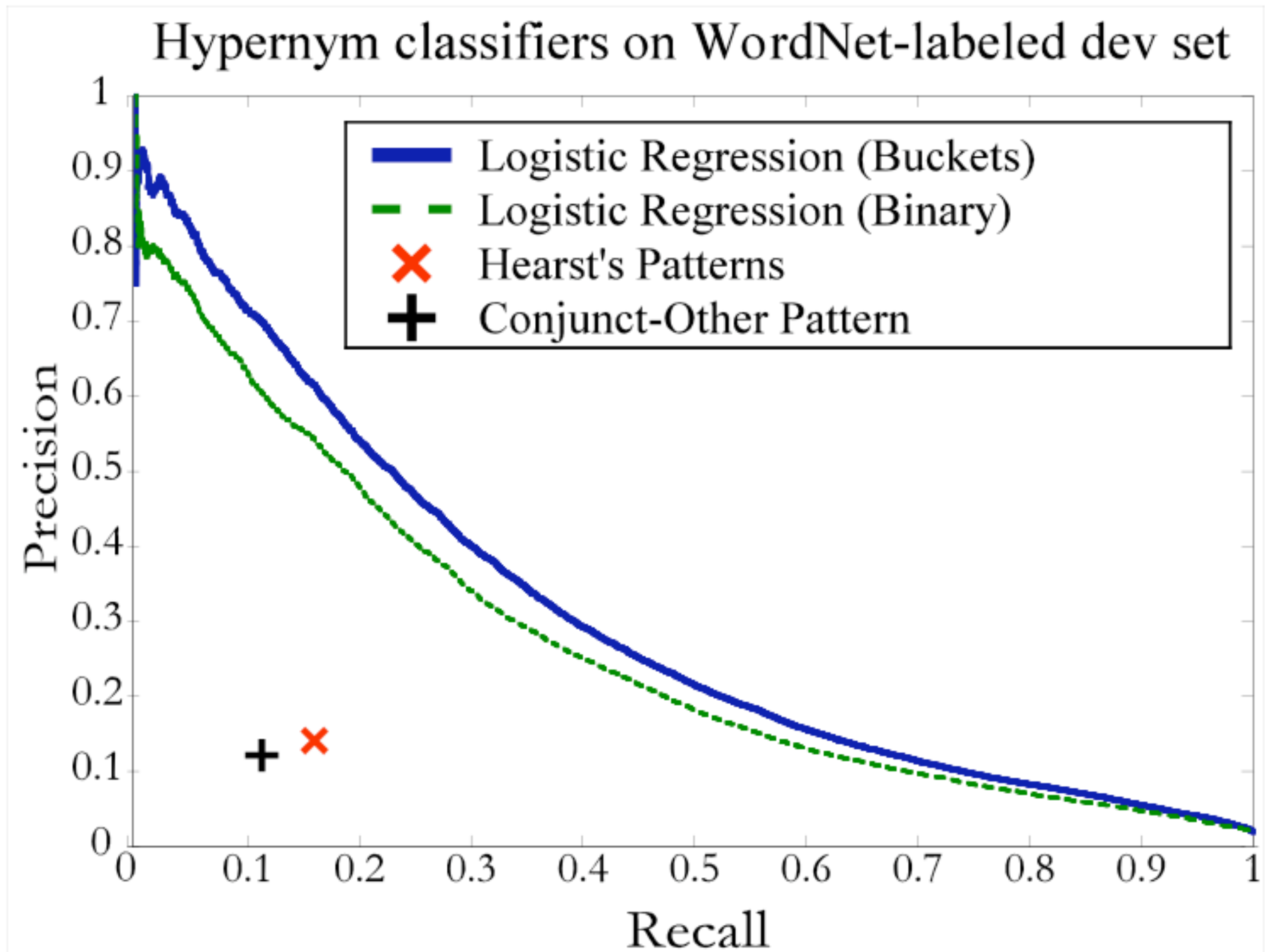
MINIPAR path: -N:desc:V,call,call,-V:vrel:N → "<hypernym> 'called' <hyponym>"

None of the following links are contained in WordNet (or the training set, by extension).

Hyponym	Hypernym	Sentence Fragment
efflorescence	condition	…and a condition called efflorescence…
'neal_inc	company	…The company, now called O'Neal Inc.…
hat_creek_outfit	ranch	…run a small ranch called the Hat Creek Outfit.
tardive_dyskinesia	problem	... irreversible problem called tardive dyskinesia…
hiv-1	aids_virus	…infected by the AIDS virus, called HIV-1.
bateau_mouche	attraction	…sightseeing attraction called the Bateau Mouche...
kibbutz_malkiyya	collective_farm	…Israeli collective farm called Kibbutz Malkiyya…

Type of Noun Pair	Count	Example Pair
NE: Person	7	"John F. Kennedy / president", "Marlin Fitzwater / spokesman"
NE: Place	7	"Diamond Bar / city", "France / place"
NE: Company	2	"American Can / company", "Simmons / company"
NE: Other	1	"Is Elvis Alive / book"
Not Named Entity:	9	"earthquake / disaster", "soybean / crop"

A better hypernym classifier



- 10-fold cross validation on the WordNet-labeled data
- **Conclusion:** 70,000 features are more powerful than 6

VERBOCEAN: Mining the Web for Fine-Grained Semantic Verb Relations

Timothy Chklovski and Patrick Pantel

Why Detect Semantic Rels between Verbs?

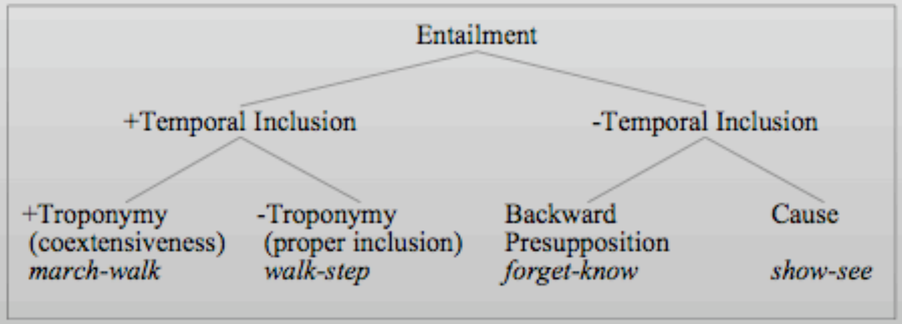
- So that we can
  - Understand the relationship when it's not stated
    - Napoleon ***fought*** and ***won*** the battle
    - During the holidays, people ***wrap*** and ***unwrap*** presents
    - Soldiers prefer to avoid getting ***wounded*** and ***killed***
  - Use the relationship when summarizing across documents (e.g. same event, preceding event)
    - The board ***considered*** the offer of $3B
    - The board ***accepted*** the offer $3.8B
    - The board ***okayed*** the offer of approximately $4B
  - Determine if two people have similar views on and event
    - "I ***nudged*** him."
    - "He ***shoved*** me."
- Hard to do manually

Why use Web? Motivating Intuition

- Small collections are tough: Semantics is often implied (Lenat, Chklovski)
- The Web's **10^{12}** is a lot of words
- So, Use small bits of more detailed text to help with mass of general text
  - Patterns issued to a search engine and their correlation

Relevant Work

- Levin's classes (similarity)
  - 3200 verbs in 191 classes
- PropBank
  - 4,659 framesets (1.4 framesets per verb)
- VerbNet
  - 191 coarse-grained groupings (with overlap)
- FrameNet
- WordNet
  - troponomy
  - antonymy
  - entailment
  - cause



Fellbaum's (1998) entailment hierarchy.

VerbOcean: Web-based Extraction of Verb Relations

- VerbOcean is a network of verb relations
  - Currently, over 3400 nodes with on average 13 relations per verb
- Detected relation types are:
  - similarity
  - strength
  - antonymy
  - enablement
  - temporal precedence (happens-before)
- Download from http://semantics.isi.edu/ocean/

Approach

- Three stages:
  - Identify pairs of highly associated verbs co-occurring on the Web with sufficient frequency using DIRT (Lin and Pantel 2001)
  - For each verb pair
    - test patterns associated with each semantic relation
      - E.g. Temporal Precedence:
        "to *X* and then *Y*", "*X*ed and then *Y*ed"
    - calculate a score for each possible semantic relation
  - Compare the strengths of the individual semantic relations and output a consistent set as the final output
    - prefer the most specific and then strongest relations

Lexical Patterns

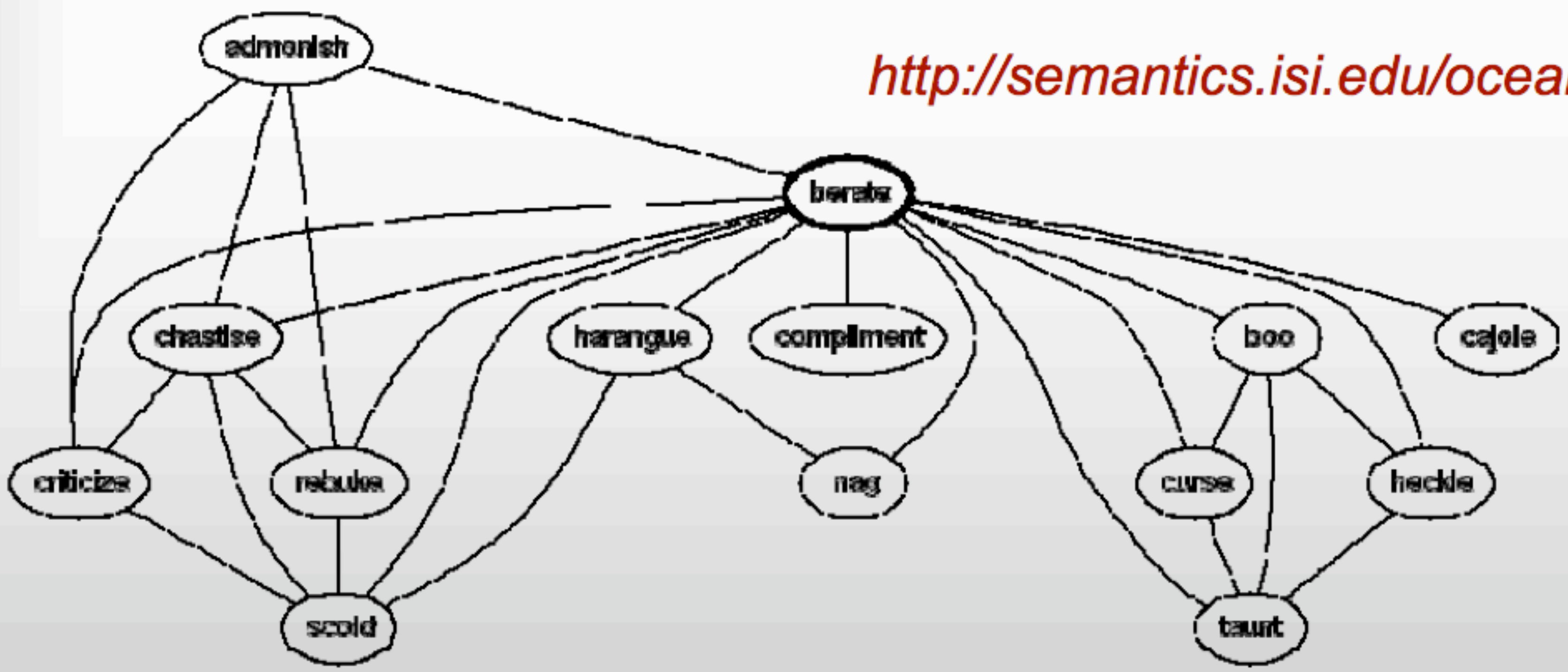
Semantic Relation	*Surface Patterns*	*Example*
similarity (4)	X ie Y Xed and Yed	*"She **heckled and taunted** the comedian."*
strength (8)	X even Y Xed even Yed Xed and even Yed not just Xed but Yed	*"He not **just harassed, but terrorized** her."*
enablement (4)	Xed * by Ying the Xed * by Ying or to X * by Ying the	*"She **saved the document by clicking the** button."*
antonymy (7)	either X or Y either Xs or Ys Xed * but Yed	*"There's something about Mary: you will **either love or hate** her."*
happens-before (12)	to X and then Y Xed * and then Yed to X and later Y to X and subsequently Y Xed and subsequently Yed	*"He **designed the prototype and then patented** it."*

Lexical Patterns Match…

- Refined to decrease capturing wrong parts of speech or incorrect semantic relations
  - Xed * by Ying the; Xed * by Ying or
    - "… waved at by parking guard …"
    - "… encouraged further by sailing lessons …"

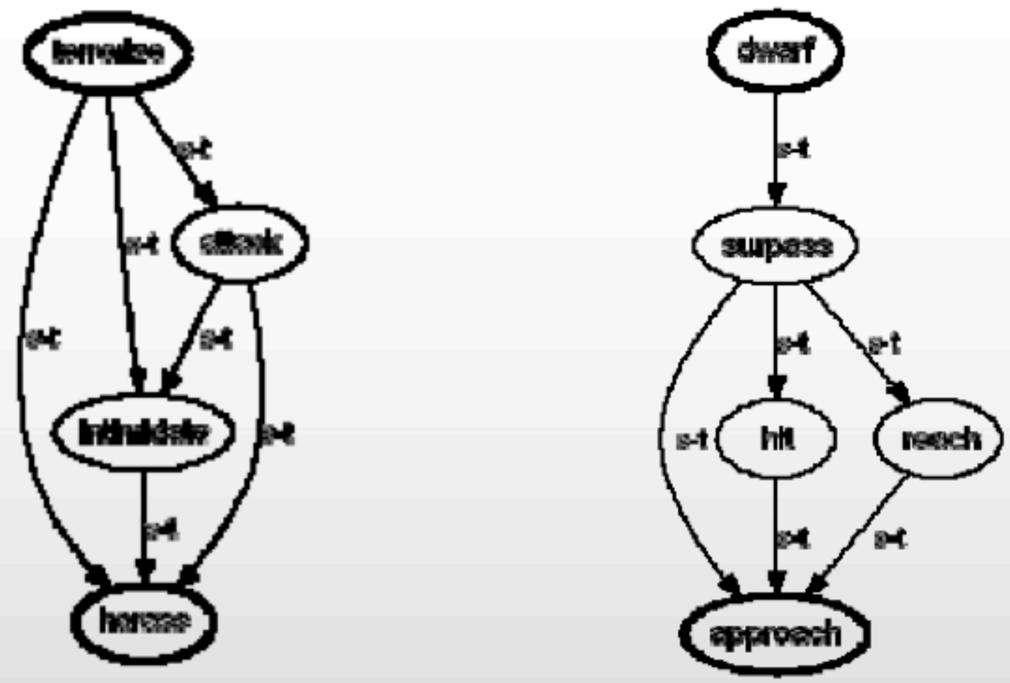
VerbOcean – Similarity

http://semantics.isi.edu/ocean/



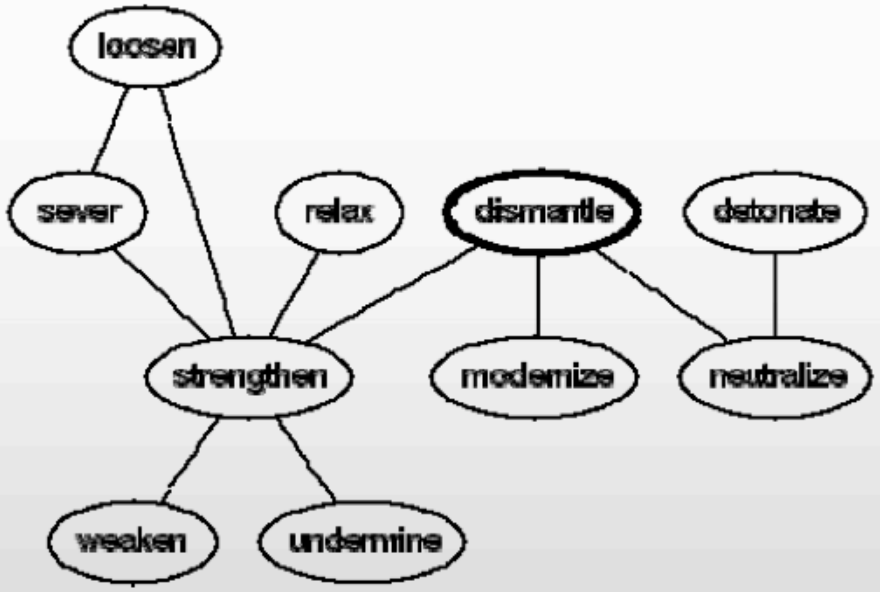
- Verbs that are similar or related
  - e.g. boo - heckle

VerbOcean – Strength



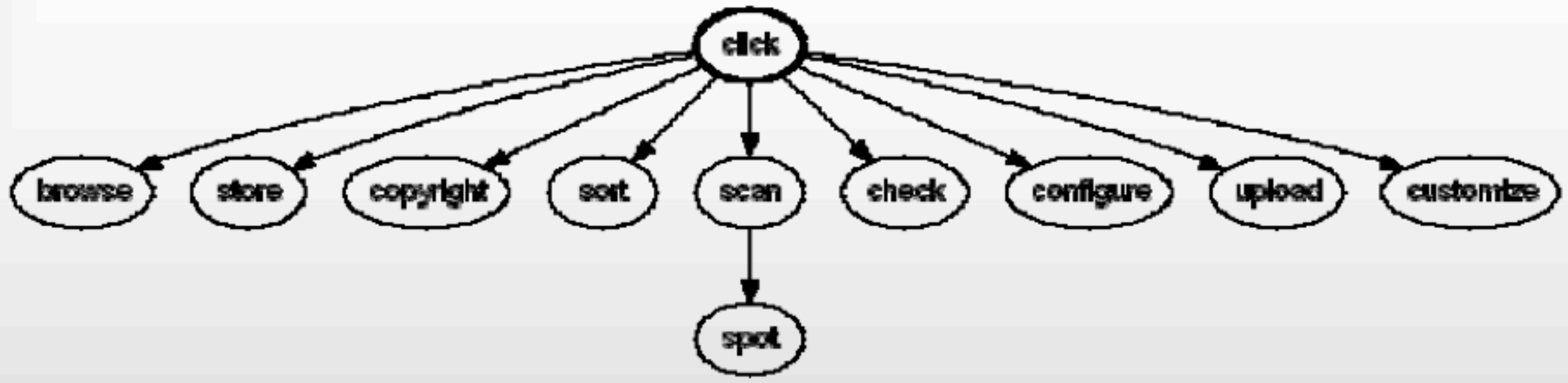
- Similar verbs that denote a more intense, thorough, comprehensive or absolute action
  - e.g. change-of-state verbs that denote a more complete change (shock → startle)

VerbOcean – Antonymy



- Semantic opposition
  - switching thematic roles associated with the verb (buy – sell)
  - stative verbs (live – die)
  - sibling verbs which share a parent (walk – run)
  - restitutive opposition: antonymy + happens-before (damage - repair)

VerbOcean – Enablement



- Holds between two verbs V_1 and V_2 when the pair can be glossed as "V_1 is accomplished by V_2" (assess - review)

Appendix. Sample relations extracted by our system.

SEMANTIC RELATION	EXAMPLES	SEMANTIC RELATION	EXAMPLES	SEMANTIC RELATION	EXAMPLES
similarity	maximize :: enhance produce :: create reduce :: restrict	enablement	assess :: review accomplish :: complete double-click :: click	happens before	detain :: prosecute enroll :: graduate schedule :: reschedule
strength	permit :: authorize surprise :: startle startle :: shock	antonymy	assemble :: dismantle regard :: condemn roast :: fry