Formal Semantics

Natural Language Processing
CS 4120/6120—Spring 2017
Northeastern University

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

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?

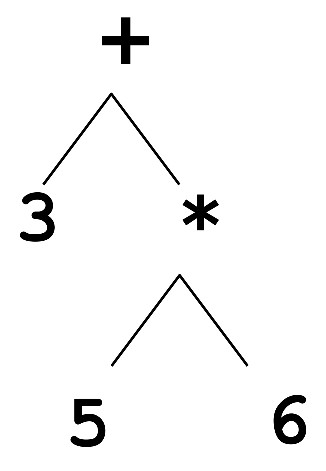
- How would you know if a computer “understood” the “meaning” of an (English) utterance (even in some weak “scare-quoted” way)?
- How would you know if a **person** understood the meaning of an utterance?

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)

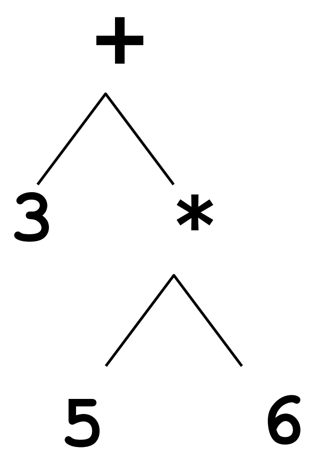
Programming Language Interpreter

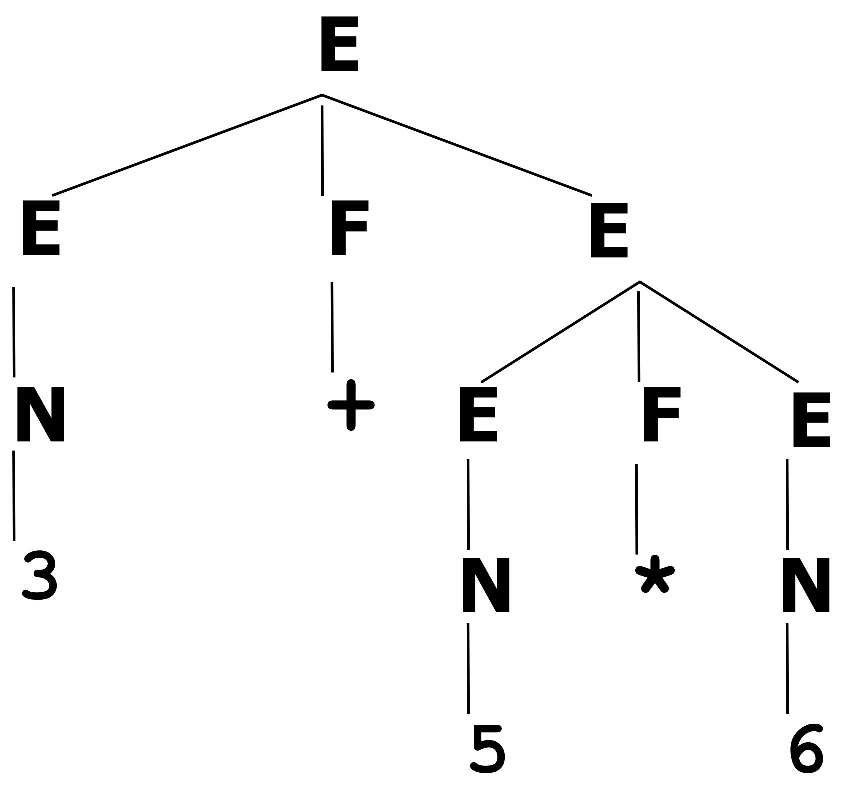
- What is meaning of `3+5*6`?
- First parse it into `3+(5*6)`



Programming Language Interpreter

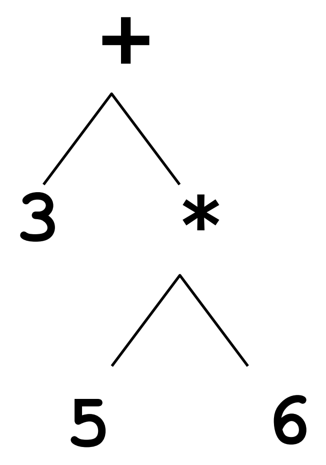
- What is meaning of `3+5*6`?
- First parse it into `3+(5*6)`

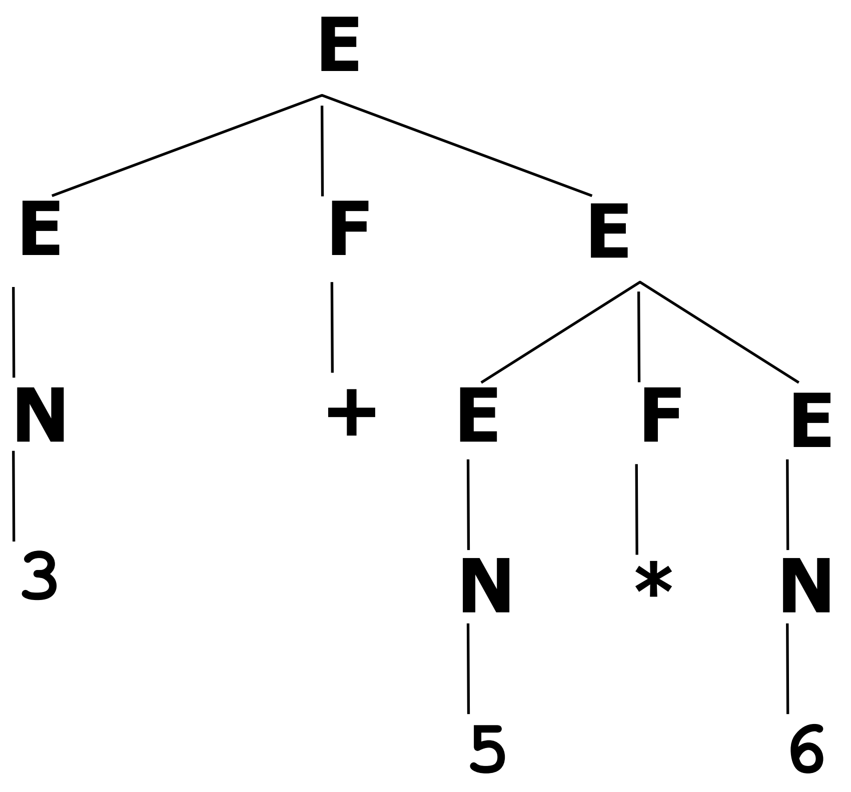




Programming Language Interpreter

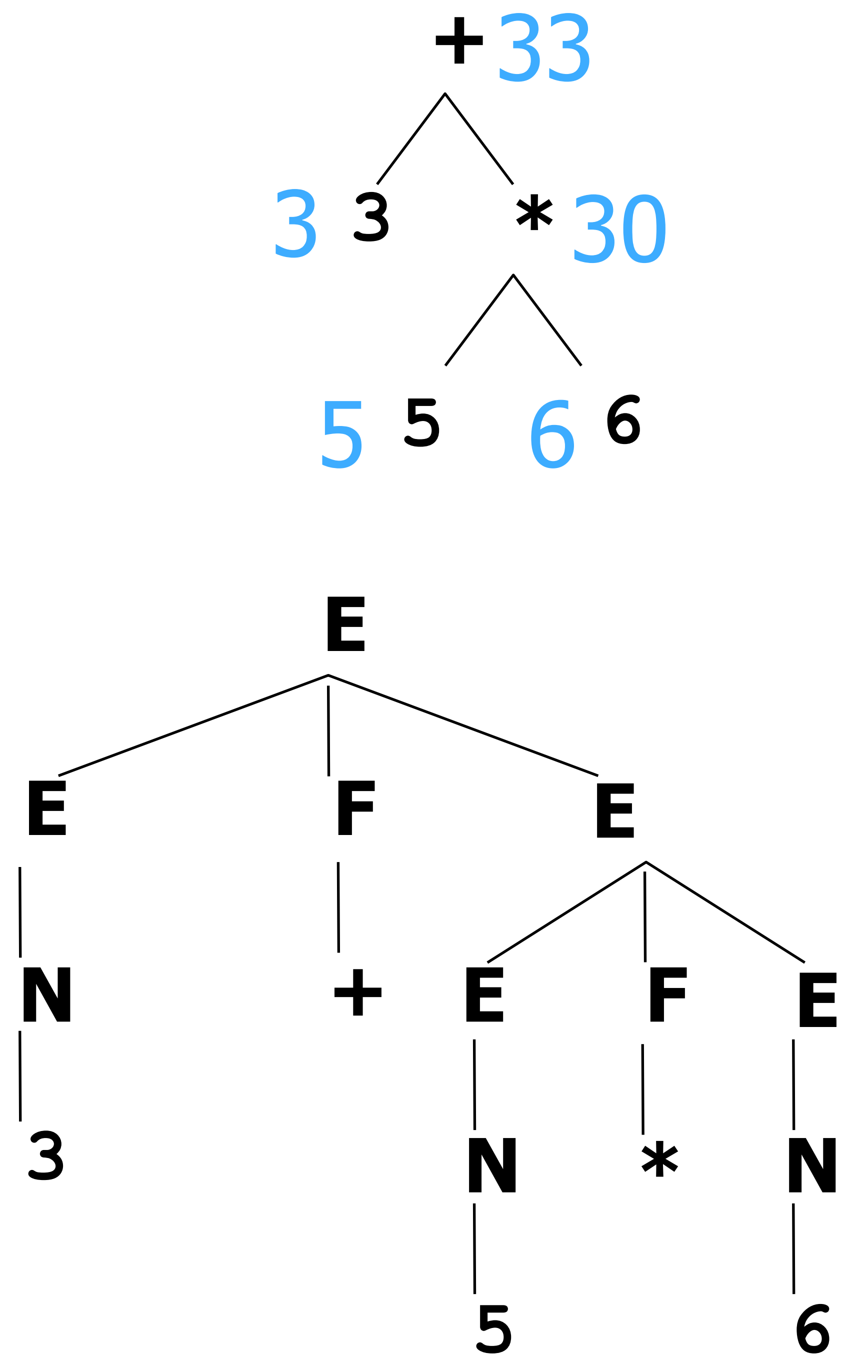
- What is meaning of `3+5*6`?
- First parse it into `3+(5*6)`
- Now give a meaning to each node in the tree (bottom-up)





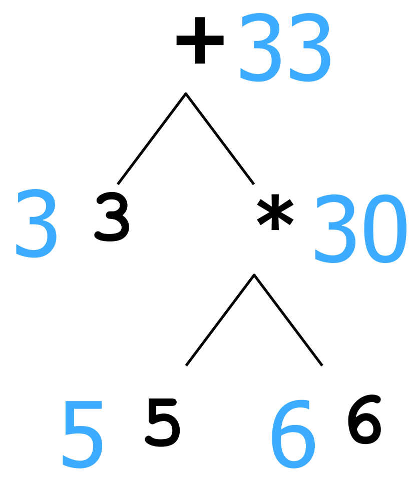
Programming Language Interpreter

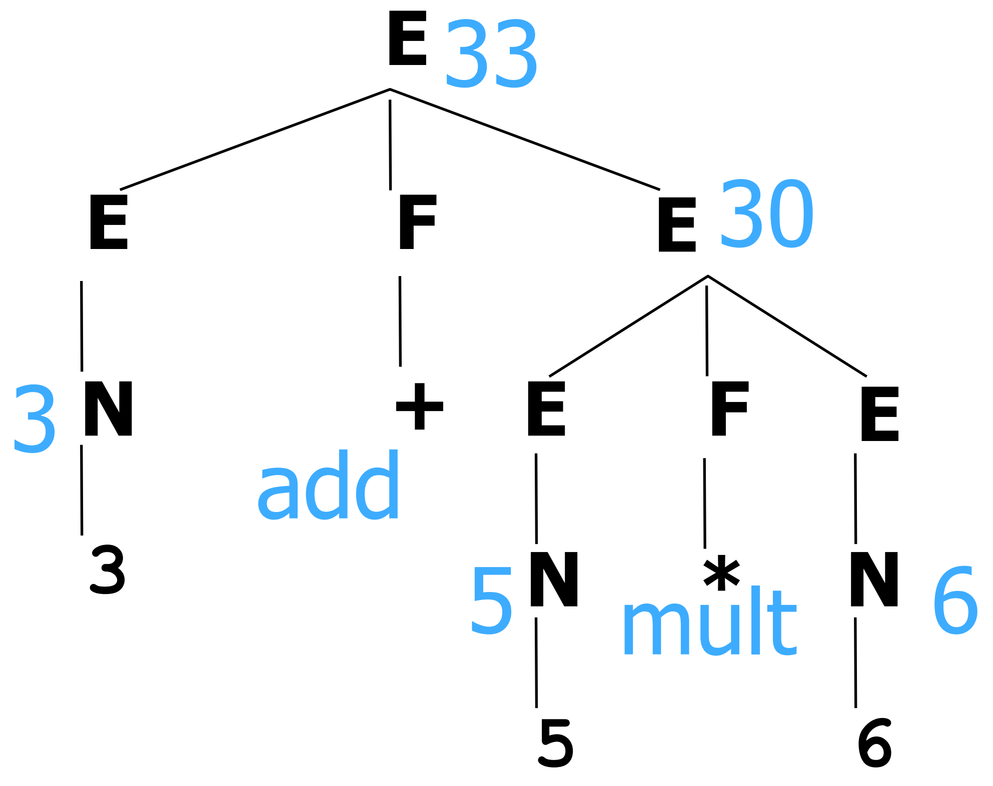
- What is meaning of `3+5*6`?
- First parse it into `3+(5*6)`
- Now give a meaning to each node in the tree (bottom-up)



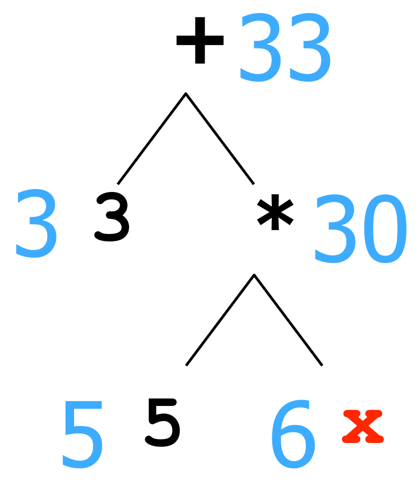
Programming Language Interpreter

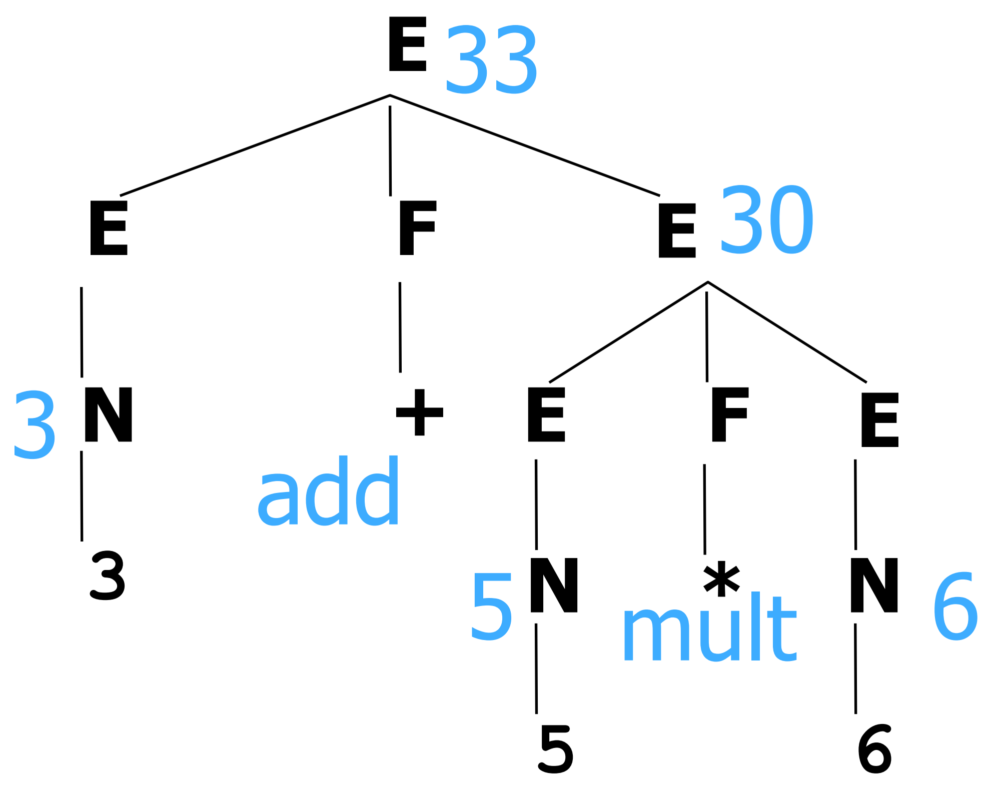
- What is meaning of `3+5*6`?
- First parse it into `3+(5*6)`
- Now give a meaning to each node in the tree (bottom-up)





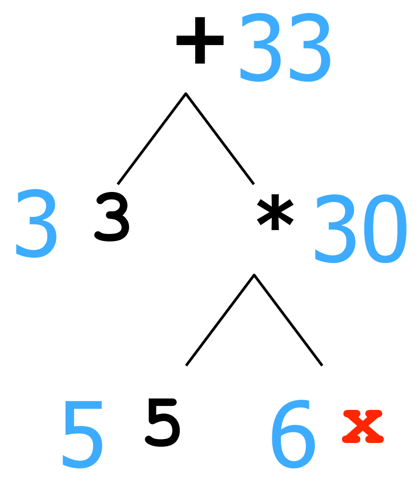
Interpreting in an Environment

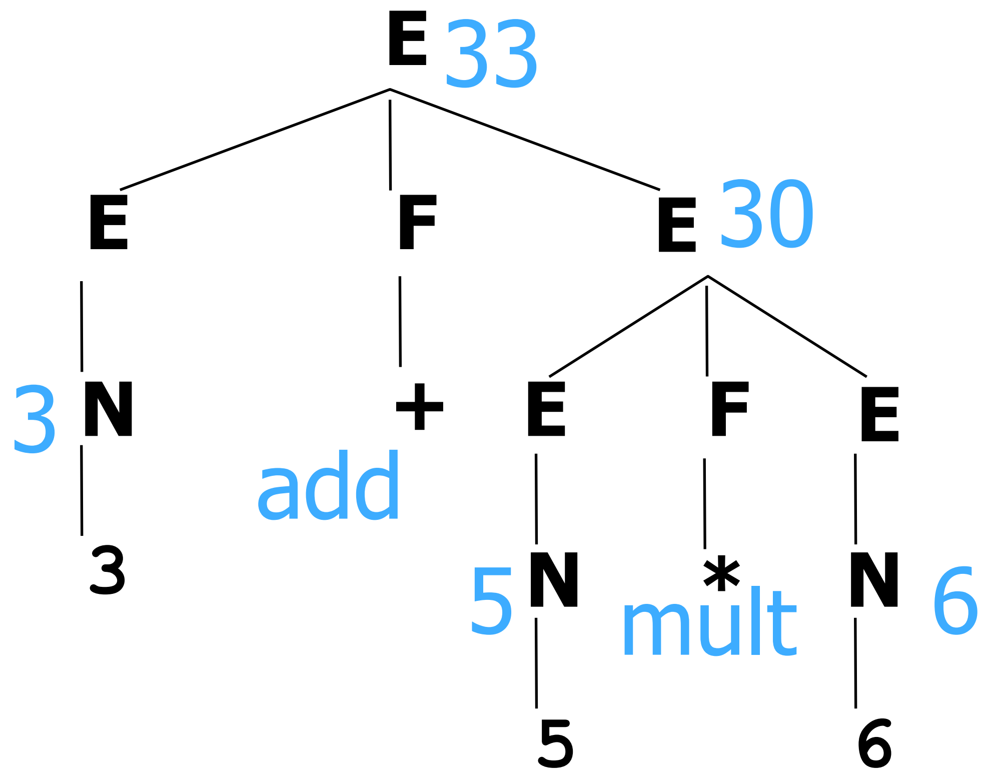




Interpreting in an Environment

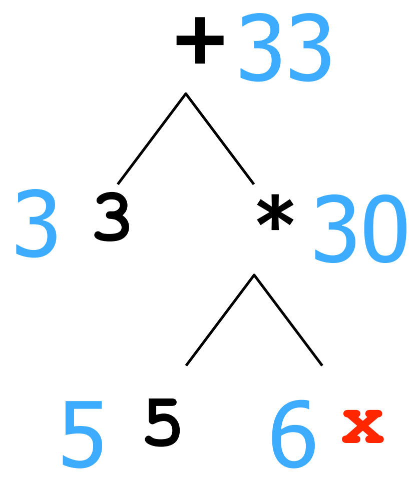
- How about `3+5*x`?

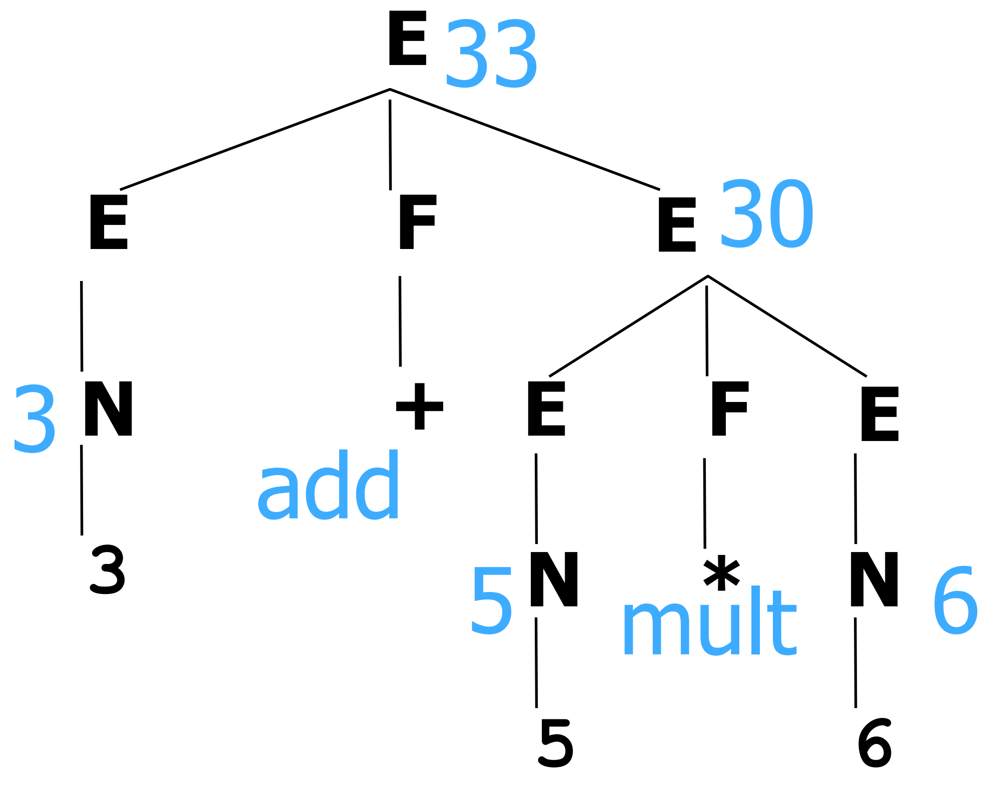




Interpreting in an Environment

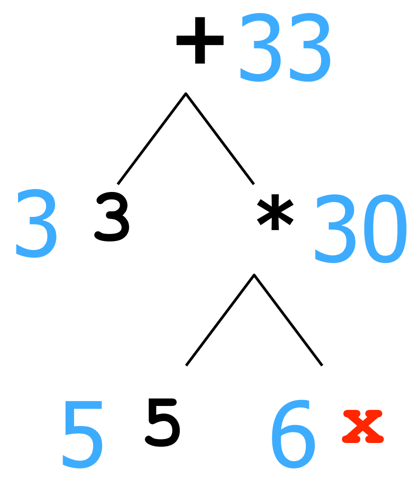
- How about `3+5*x`?
- Same thing: the meaning of `x` is found from the environment (it's 6)

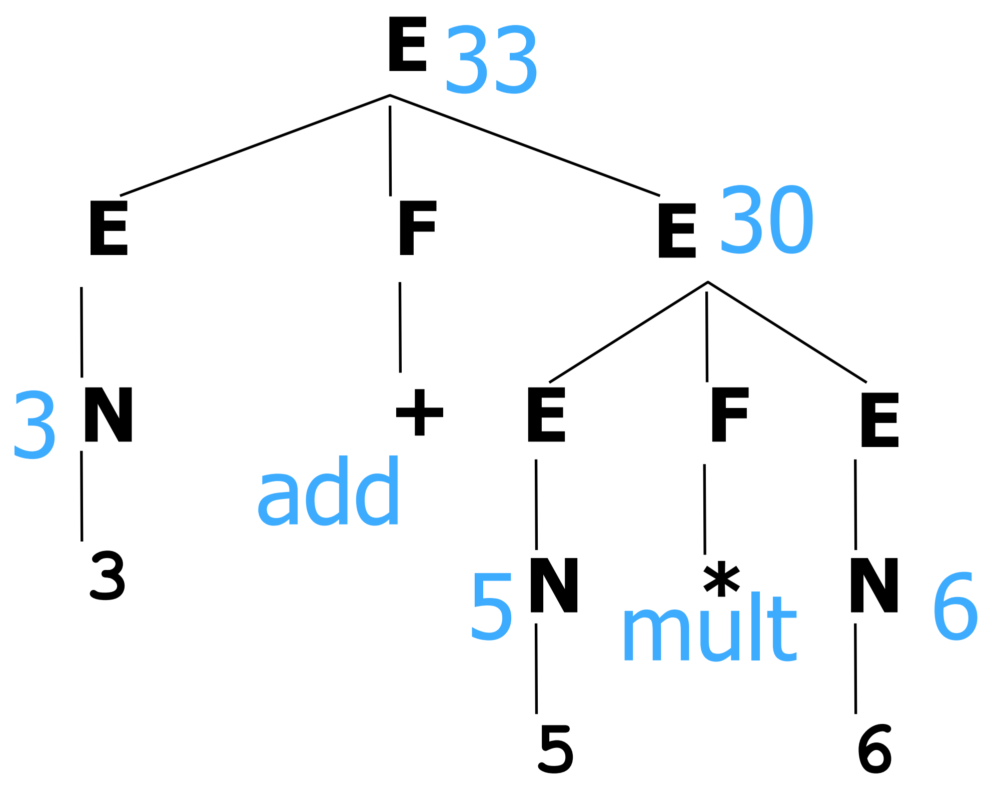




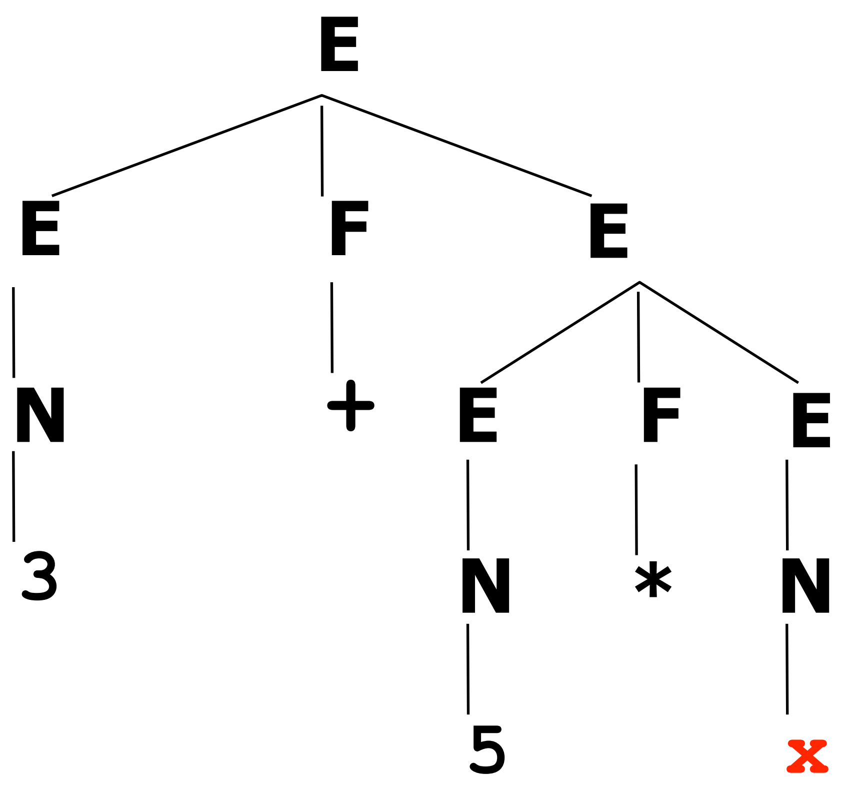
Interpreting in an Environment

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



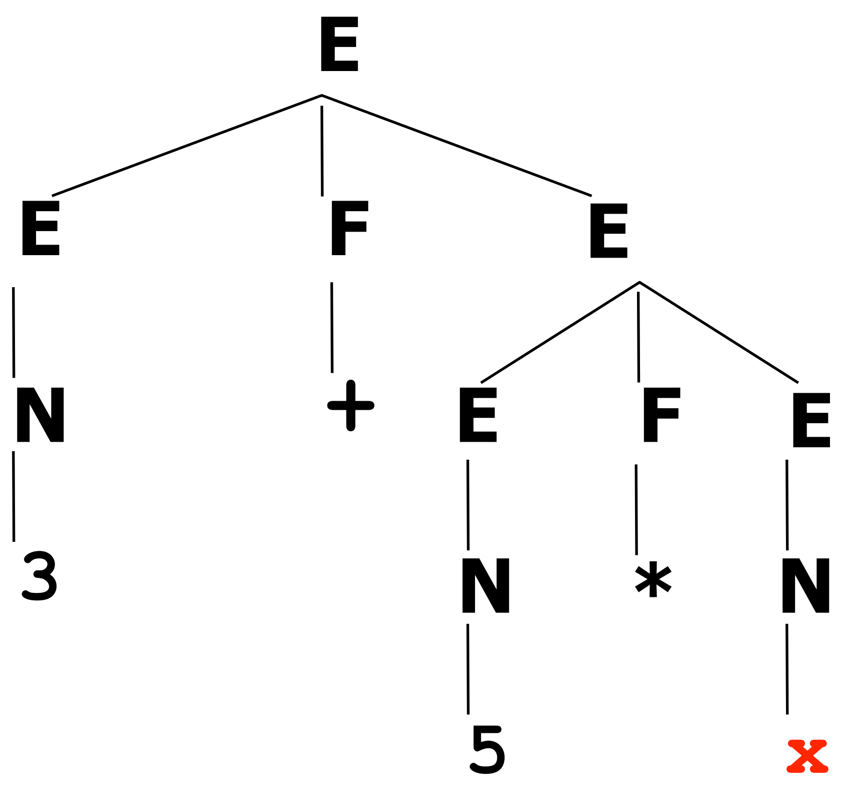


Compiling



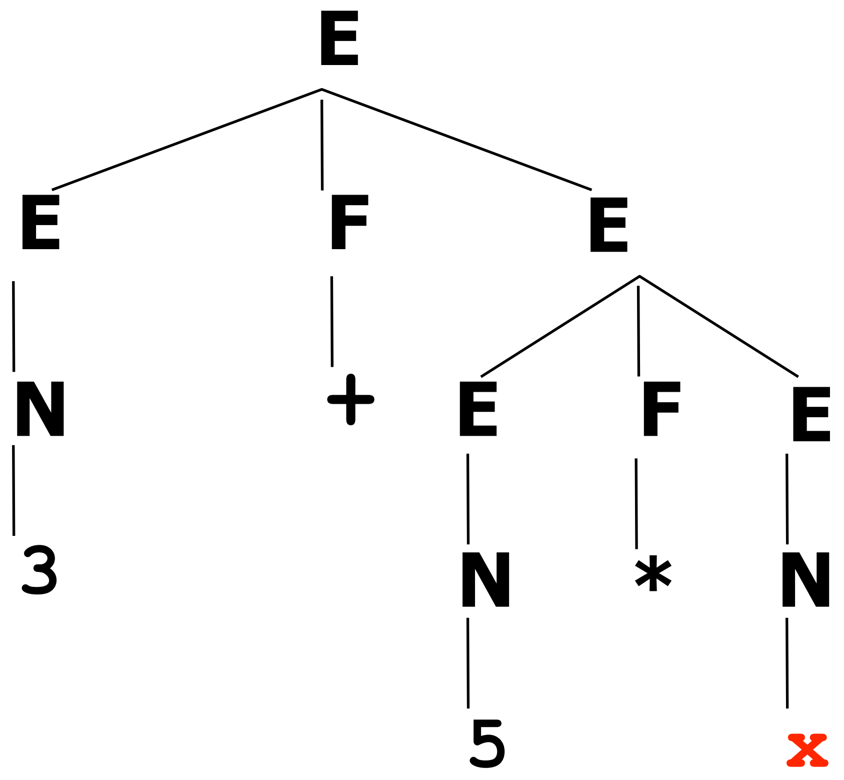
Compiling

- How about `3+5*x`?



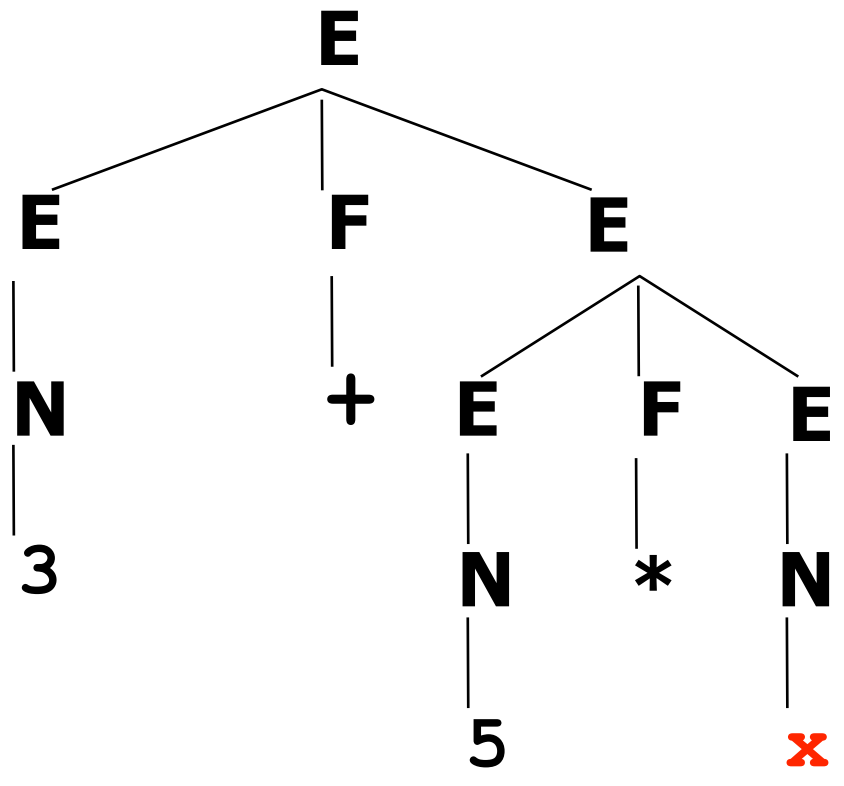
Compiling

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



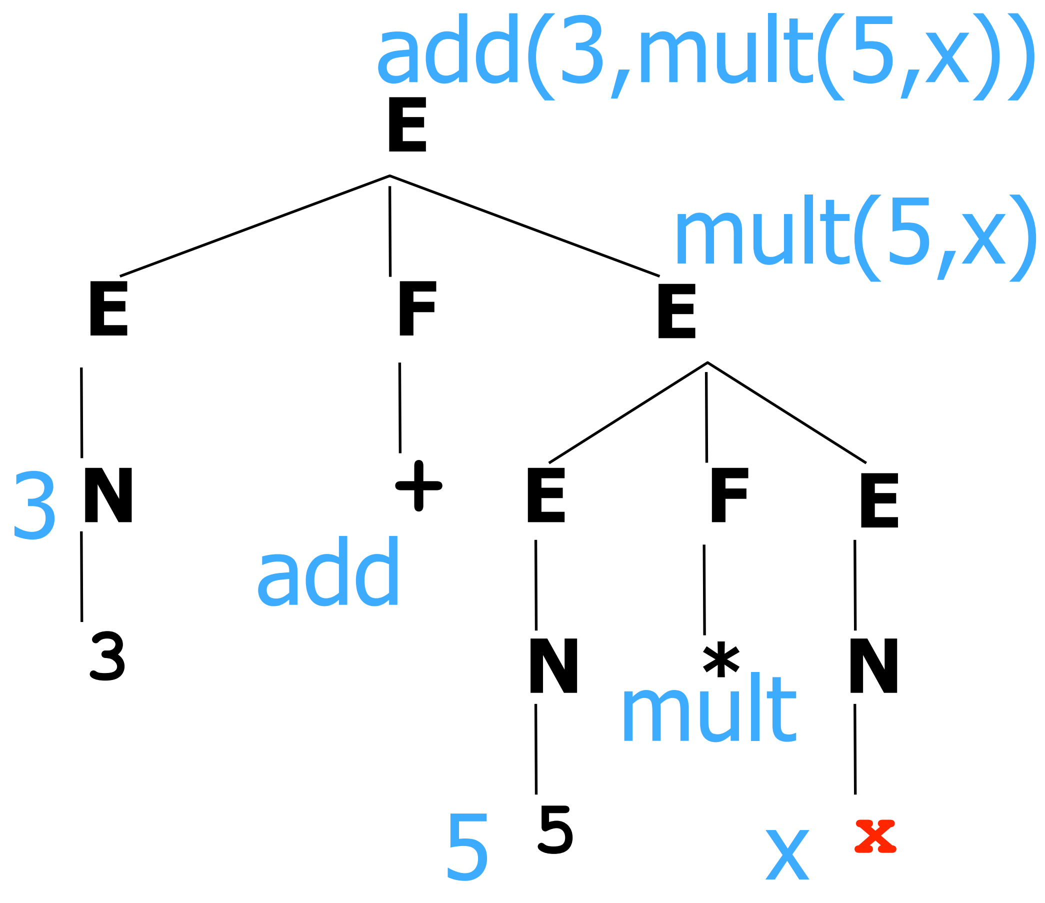
Compiling

- 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

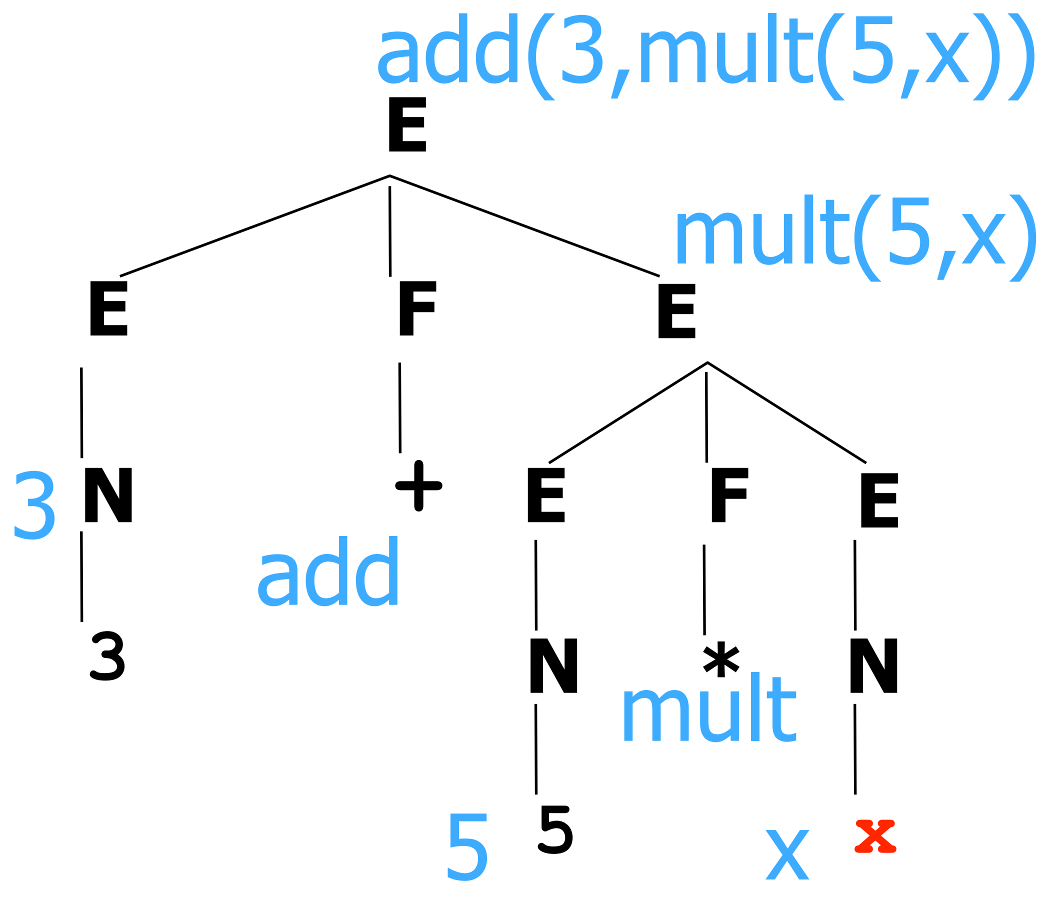
- How about `3+5*x`?
- Don't know `x` at compile time
- "Meaning" at a node is a piece of code, not a number



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

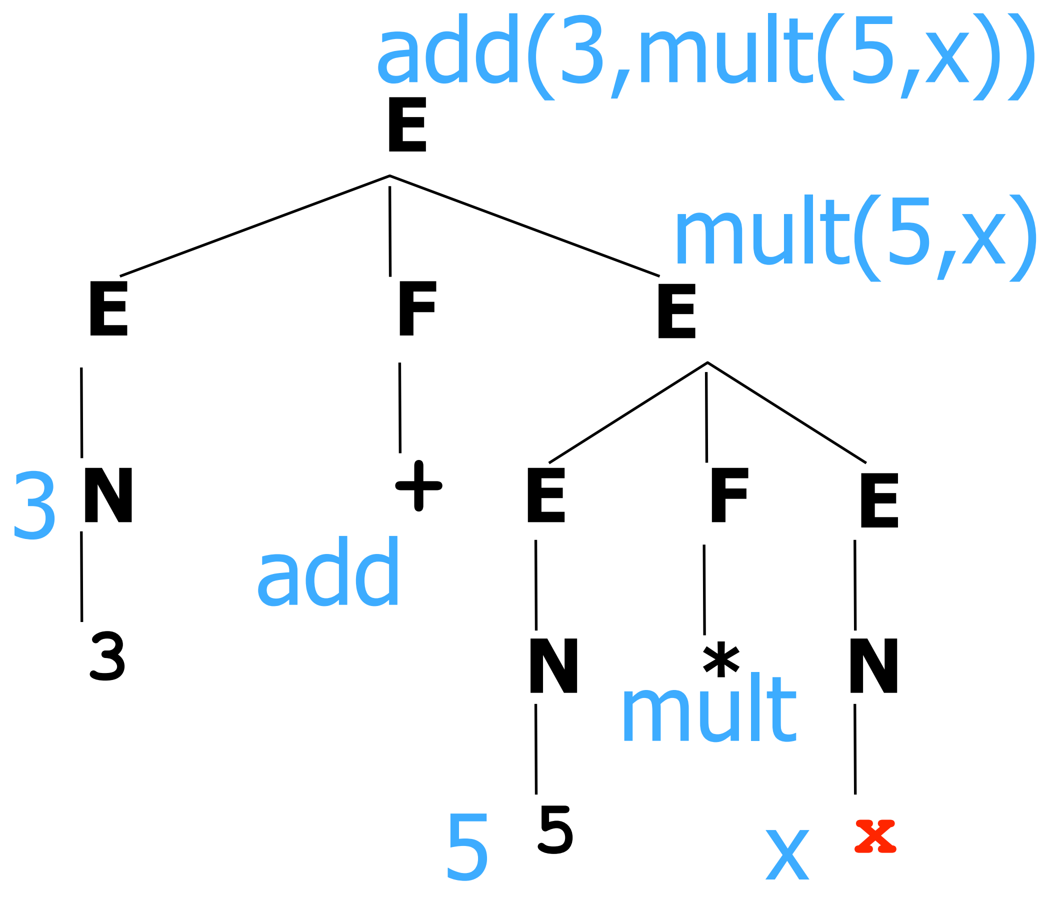
`5*(x+1)-2` is a different expression that produces equivalent code



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)

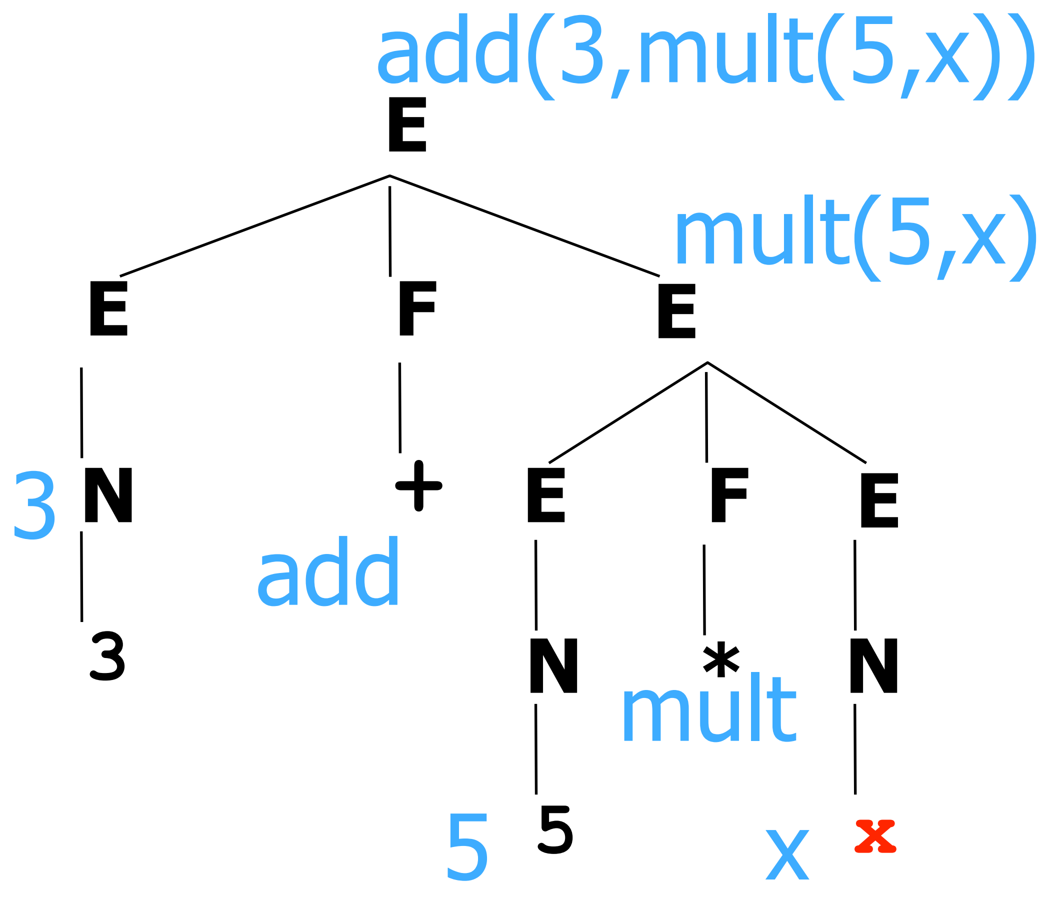


Compiling

- 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 = $\lambda p\ p*p$
  - Then square(3) = $(\lambda p\ p*p)(3) = 3*3$

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 = $\lambda p\ p*p$
  - Then square(3) = $(\lambda p\ p*p)(3)$ = 3*3
  - Note: square(x) isn't a function! It's just the value x*x.
  - But $\lambda \mathbf{x}$ square(x) = $\lambda x\ x*x$ = $\lambda p\ p*p$ = 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 predicate: returns true/false
  - even(x) is true if x is even
  - How about even(square(x))?
  - λx even(square(x)) is true of numbers with even squares
    - Just apply rules to get λx (even(x*x)) = λx (x*x mod 2 == 0)
    - This happens to denote the same predicate as even does

Logic: Multiple Arguments

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 defined 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 ∀x pig(x) ⇒ big(x))
  - exists(pig,big) (equivalent to ∃x pig(x) AND big(x))
  - can even build complex quantifiers from English phrases:
    - "between 12 and 75"; "a majority of"; "all but the smallest 2"

A reasonable representation?

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

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

A reasonable representation?

A reasonable representation?

- `Gilly swallowed a goldfish`
  - 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`
  - 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` <u>`a`</u> `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` (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
- 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`
  - First attempt: swallowed(Gilly, goldfish)

Use a Quantifier

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

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"

Use a Quantifier

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

Tense

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)), <u>exists</u>(booth, location(e)), …
- `Gilly swallowed a goldfish in every booth`
  - ∃e past(e), act(e,swallowing), swallower(e,Gilly), exists(goldfish, swallowee(e)), <u>all</u>(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), $\underbrace{\text{exists(goldfish, swallowee(e)),}}_{\exists g\ \text{goldfish(g), swallowee(e,g)}}$ 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`
  - $\exists$e past(e), act(e,swallowing), swallower(e,Gilly), $\underbrace{\text{exists(goldfish, swallowee(e)),}}_{\exists g\ \text{goldfish}(g),\ \text{swallowee}(e,g)}$ $\underbrace{\text{all(booth, location(e)),}}_{\forall b\ \text{booth}(b) \Rightarrow \text{location}(e,b)}$ …

- 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), $\underbrace{\text{exists(goldfish, swallowee(e)),}}_{\exists g\ \text{goldfish(g), swallowee(e,g)}}$ $\underbrace{\text{all(booth, location(e)),}}_{\forall b\ \text{booth(b)} \Rightarrow \text{location(e,b)}}$ …

- 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` <u>`a woman`</u> `gives birth` <u>`every 15 min`</u>`.  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 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), $\underbrace{\text{exists(goldfish, swallowee(e)),}}_{\exists g\ \text{goldfish}(g),\ \text{swallowee}(e,g)}\ \underbrace{\underline{\text{all}}\text{(booth, location(e)),}}_{\forall b\ \text{booth}(b) \Rightarrow \text{location}(e,b)}$ …

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), $\underbrace{\text{exists(goldfish, swallowee(e))}}_{\exists g\ \text{goldfish(g), swallowee(e,g)}}$, $\underbrace{\text{all(booth, location(e))}}_{\forall b\ \text{booth(b)} \Rightarrow \text{location(e,b)}}$, …
- 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),
    $$\underbrace{\text{exists(goldfish, swallowee(e))}}_{\exists g\ \text{goldfish}(g),\ \text{swallowee}(e,g)},\ \underbrace{\text{all(booth, location(e)), …}}_{\forall b\ \text{booth}(b)\Rightarrow \text{location}(e,b)}$$
- 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)), $\underline{\text{exists}}$(booth, location(e)), …
- `Gilly swallowed a goldfish in every booth`
  - $$\exists e\ \text{past}(e), \text{act}(e,\text{swallowing}), \text{swallower}(e,\text{Gilly}), \underbrace{\text{exists}(\text{goldfish}, \text{swallowee}(e))}_{\exists g\ \text{goldfish}(g),\ \text{swallowee}(e,g)}, \underbrace{\underline{\text{all}}(\text{booth}, \text{location}(e)), \ldots}_{\forall b\ \text{booth}(b) \Rightarrow \text{location}(e,b)}$$
- 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`
  - $\exists$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
  - $\exists$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

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 <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)])

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"

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 <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 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 <u>set</u> of unicorns ("<u>extension</u>")
  - But this set is empty: λg unicorn(g) = λg FALSE = λg dodo(g)
  - Then `wants a unicorn` = `wants a dodo`. Oops!
  - So really the wantee should be <u>criteria</u> for unicornness ("<u>intension</u>")
- Traditional solution involves "possible-world semantics"
  - Can imagine other worlds where set of unicorn ≠ set of dodos

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`
  - ∃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 $\underbrace{\texttt{goldfish that Gilly swallowed}}_{\lambda g\ [\text{goldfish}(g),\ \text{swallowed}(\text{Gilly},\ g)]}$

Nouns and Their Modifiers

- the goldfish that Gilly swallowed
- every goldfish that Gilly swallowed
- three $\underbrace{\text{goldfish that Gilly swallowed}}_{\lambda g\,[\text{goldfish}(g),\ \text{swallowed}(\text{Gilly}, g)]}$
- three $\overbrace{\text{swallowed-by-Gilly}}^{\text{like an adjective!}}$ goldfish

Nouns and Their Modifiers

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

$$\texttt{three } \underbrace{\texttt{goldfish that Gilly swallowed}}_{\lambda g\ [\text{goldfish}(g),\ \text{swallowed}(\text{Gilly},\ g)]}$$

- `three swallowed-by-Gilly goldfish`

$$\texttt{three } \overbrace{\texttt{swallowed-by-Gilly}}^{\text{like an adjective!}} \texttt{ goldfish}$$

Or for real: $\lambda g\ [\text{goldfish}(g),\ \exists e\ [\text{past}(e),\ \text{act}(e,\text{swallowing}),\ \text{swallower}(e,\text{Gilly}),\ \text{swallowee}(e,g)\ ]]$

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**($\lambda x\ \exists e$ 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?

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

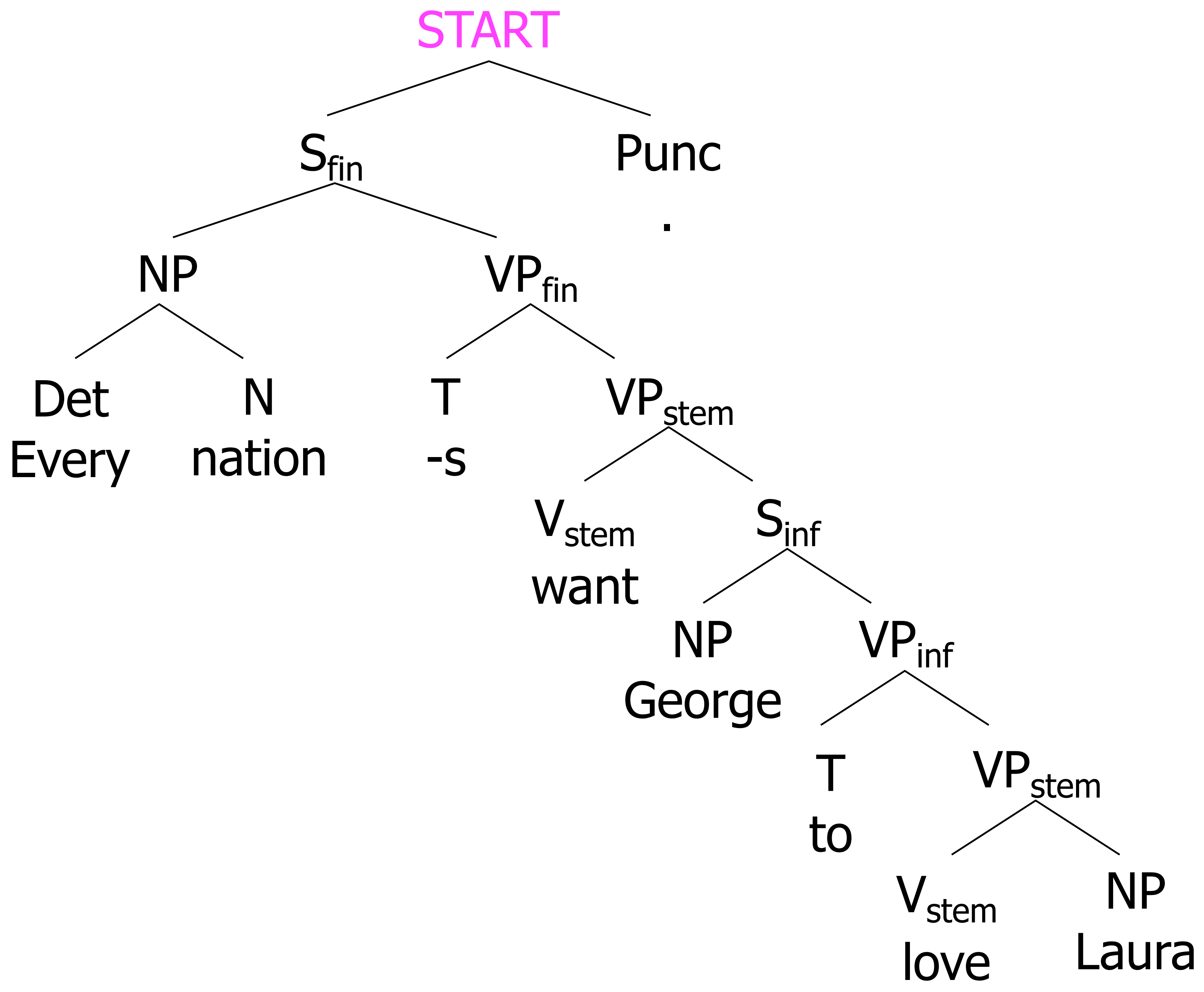
Compositional Semantics

- We've discussed what semantic representations should look like.

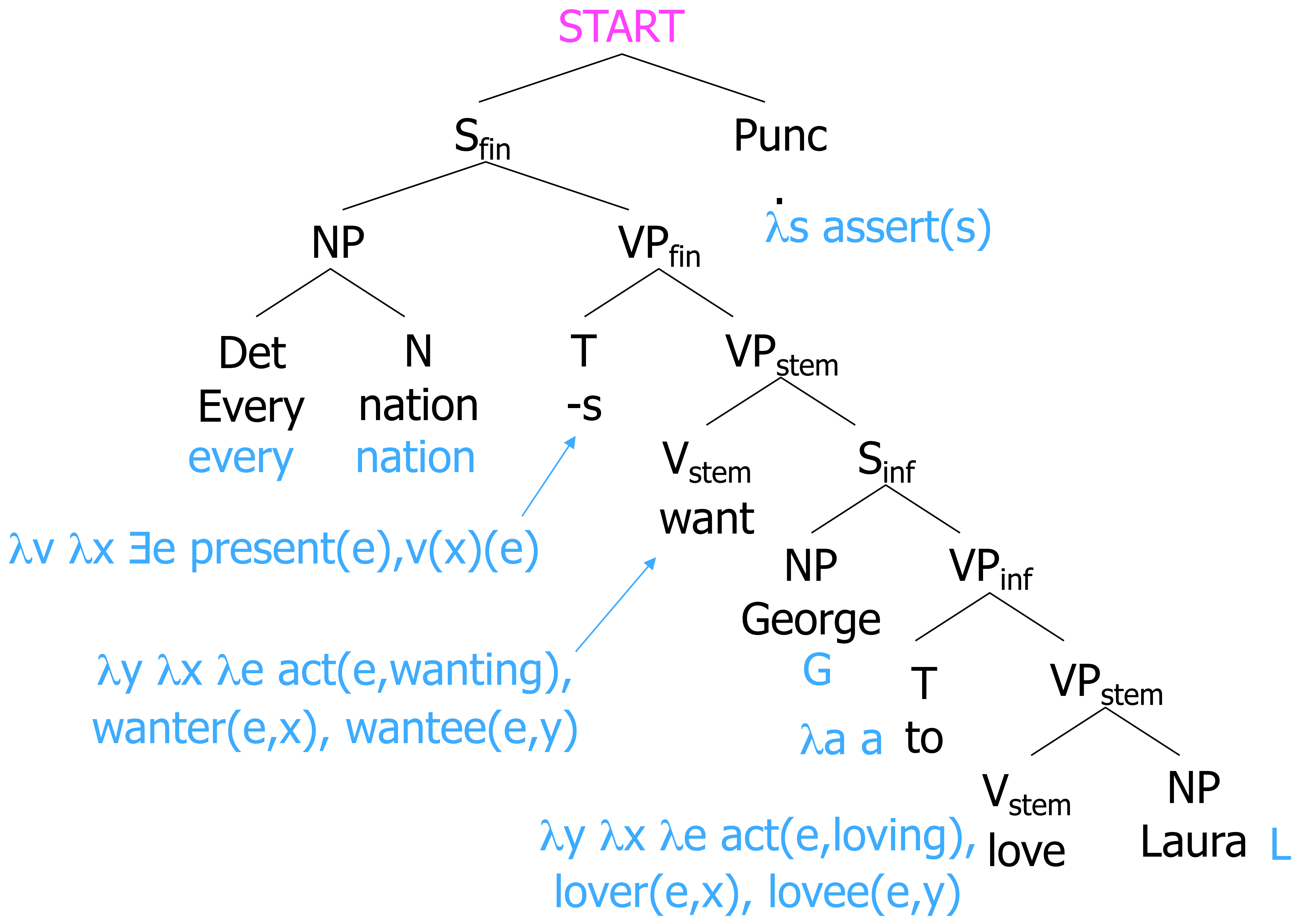
- **But how do we get them from sentences???**

- First - parse to get a syntax tree.
- Second - look up the semantics for each word.
- Third - build the semantics for each constituent
  - Work from the bottom up
  - The syntax tree is a "recipe" for how to do it

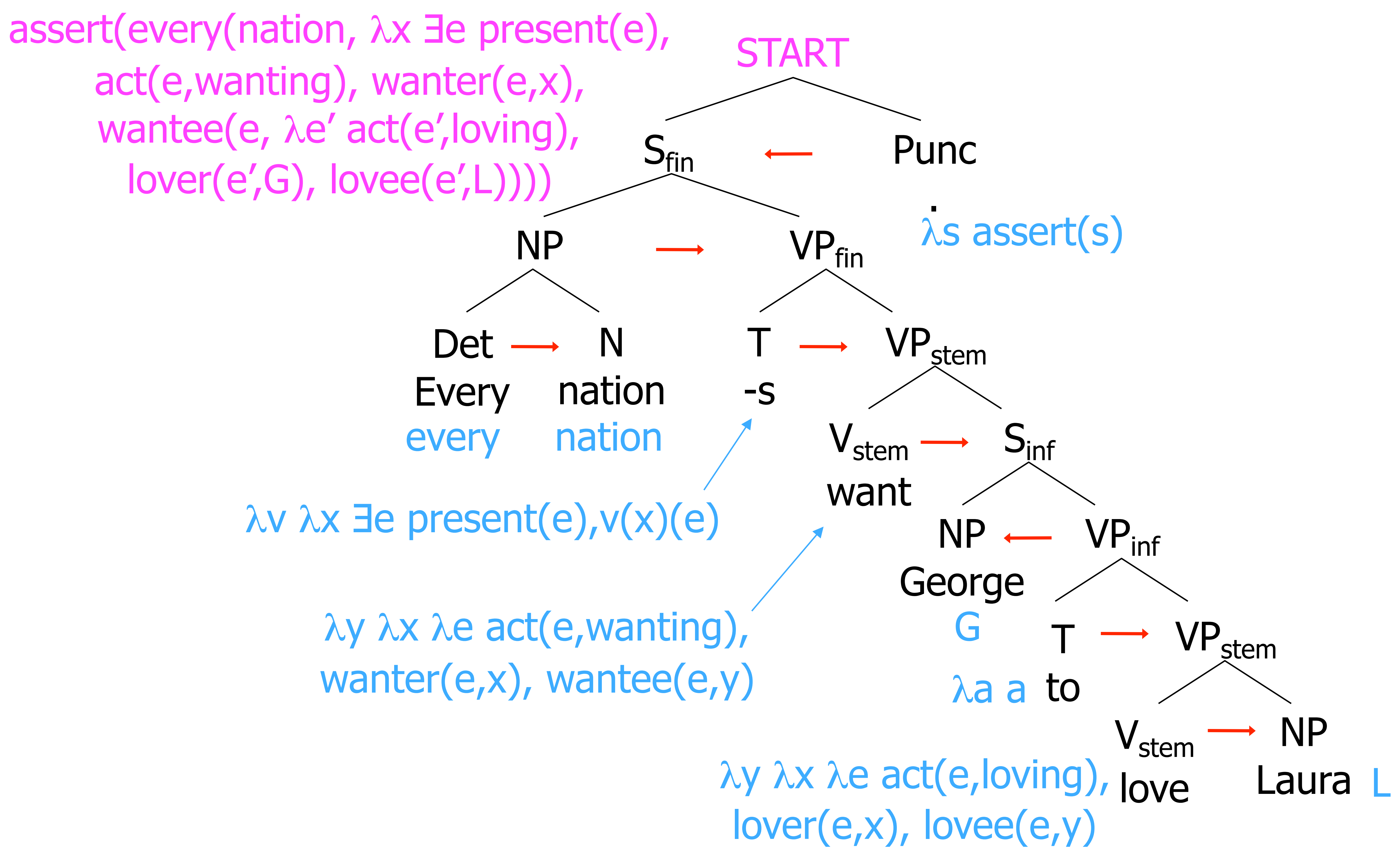
Compositional Semantics



Compositional Semantics



Compositional Semantics


assert(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
N
Every
nation
every
nation
T
-s
VP stem
λv λx ∃e present(e),v(x)(e)
V stem
want
S inf
λy λx λe act(e,wanting), wanter(e,x), wantee(e,y)
NP
George
G
VP inf
T
to
λa a
VP stem
V stem
love
NP
Laura
L
λy λx λe act(e,loving), lover(e,x), lovee(e,y)

Compositional Semantics

Compositional Semantics

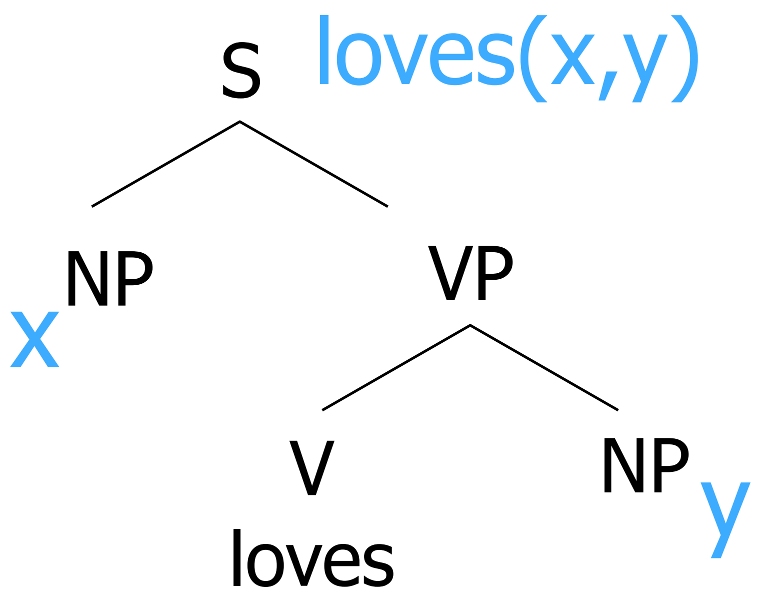
- Add a "sem" feature to each context-free rule
  - S → NP loves NP
  - S[sem=loves(x,y)] → NP[sem=x] loves NP[sem=y]
  - Meaning of S depends on meaning of NPs

Compositional Semantics

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

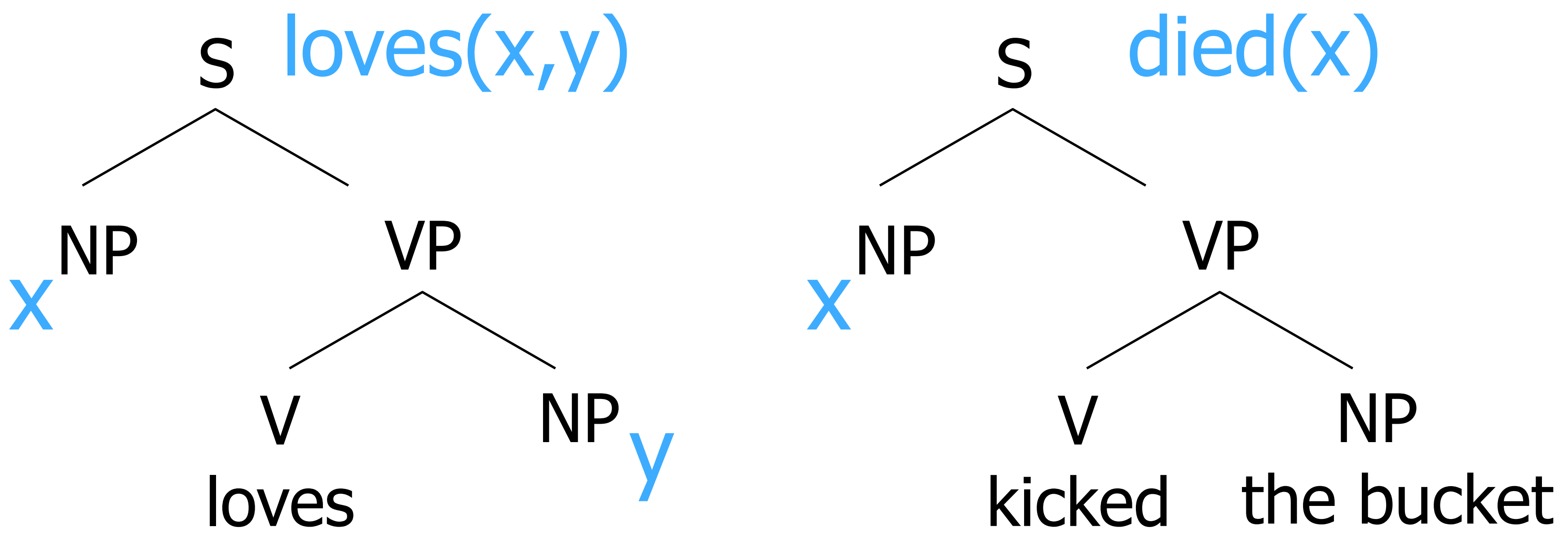
Compositional Semantics

- 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:



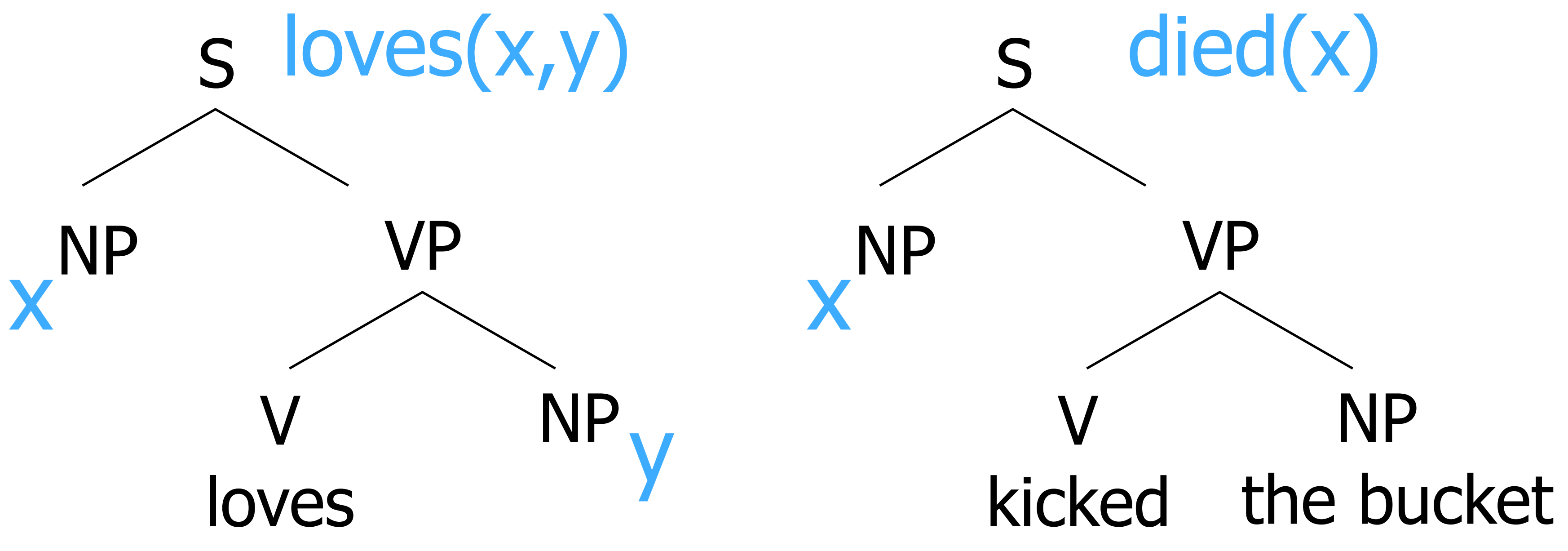
Compositional Semantics

- Add a "sem" feature to each context-free rule
  - S $\rightarrow$ NP loves NP
  - S[sem=loves(x,y)] $\rightarrow$ 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]

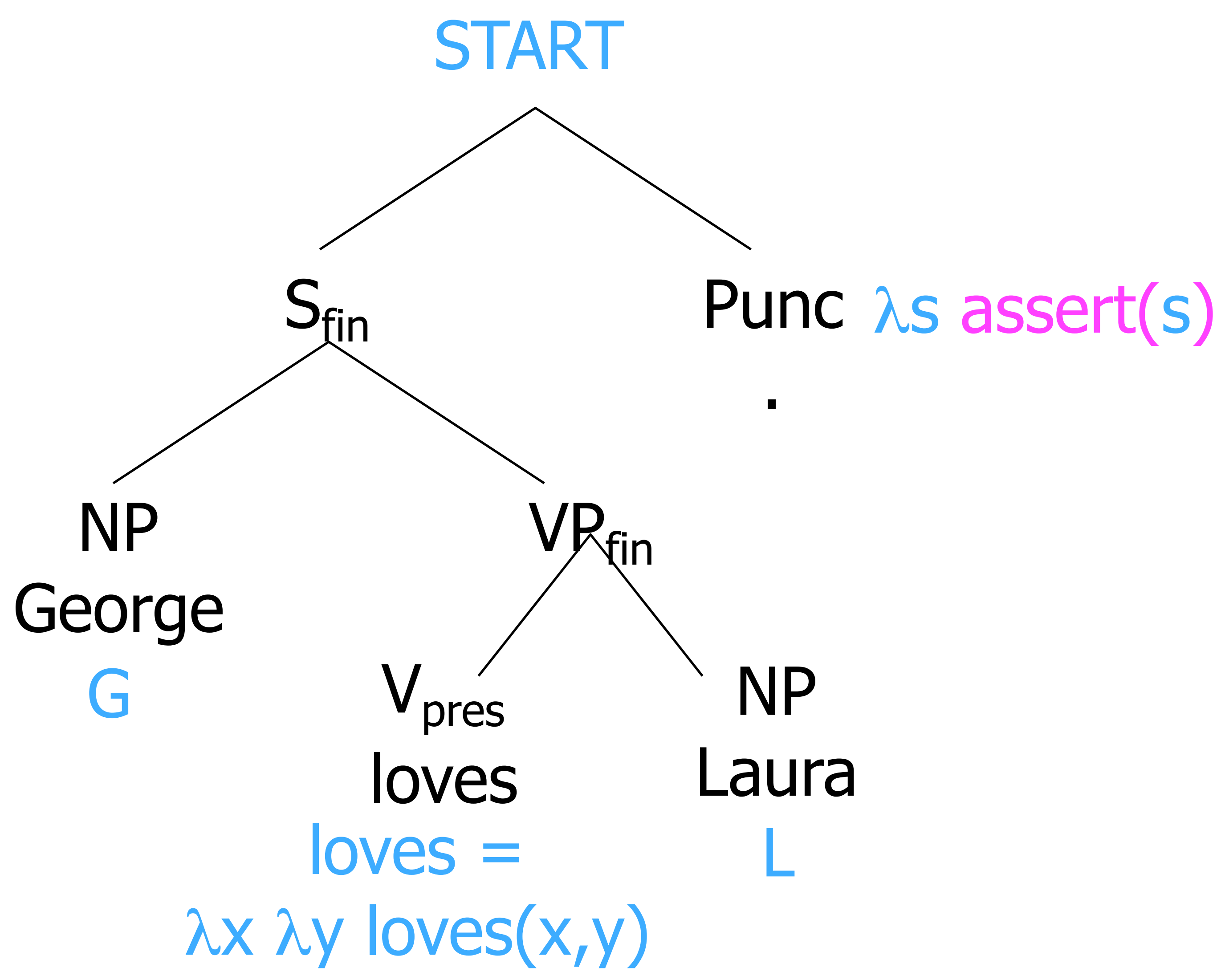
Compositional Semantics

- 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)

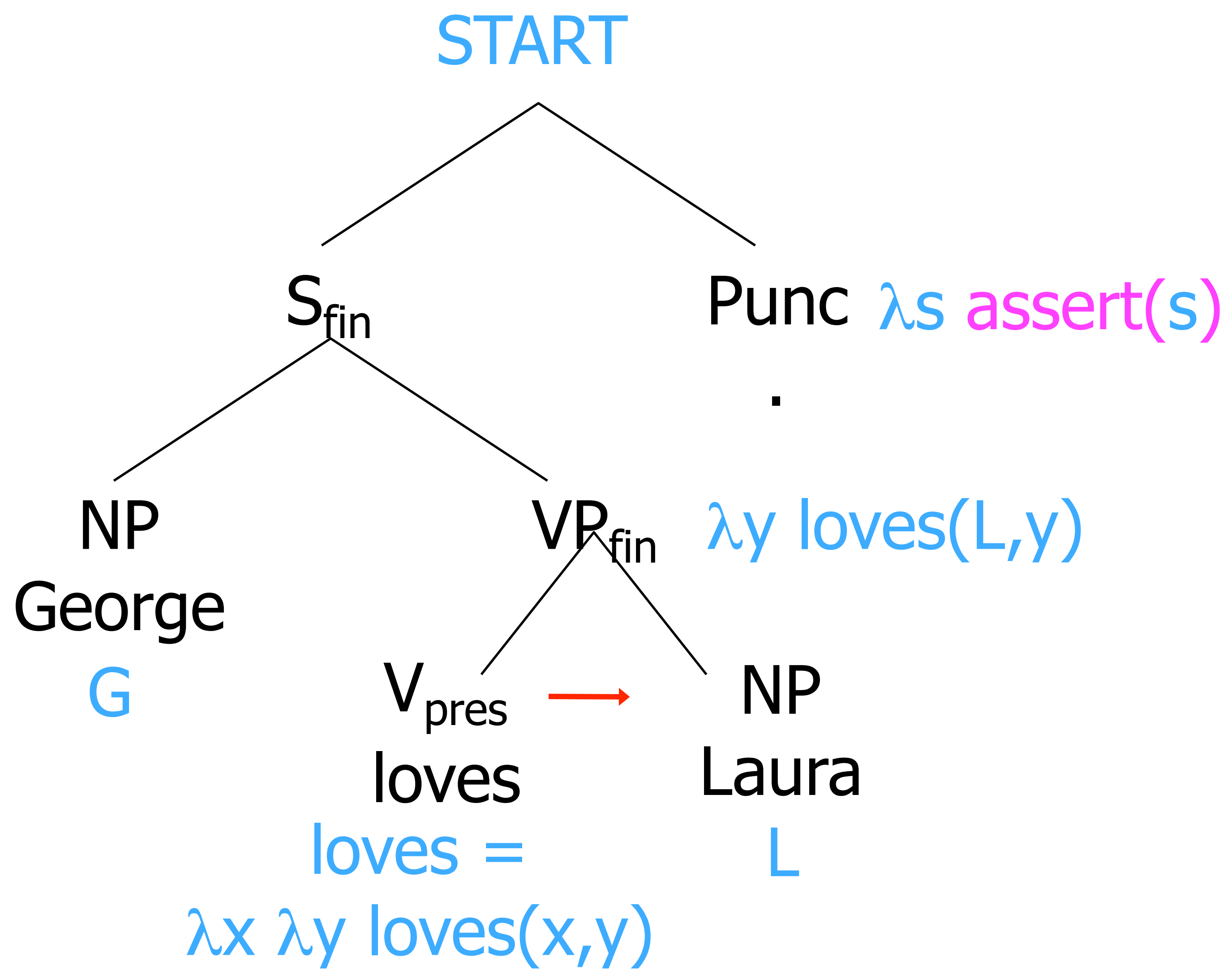
Compositional Semantics

- 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!

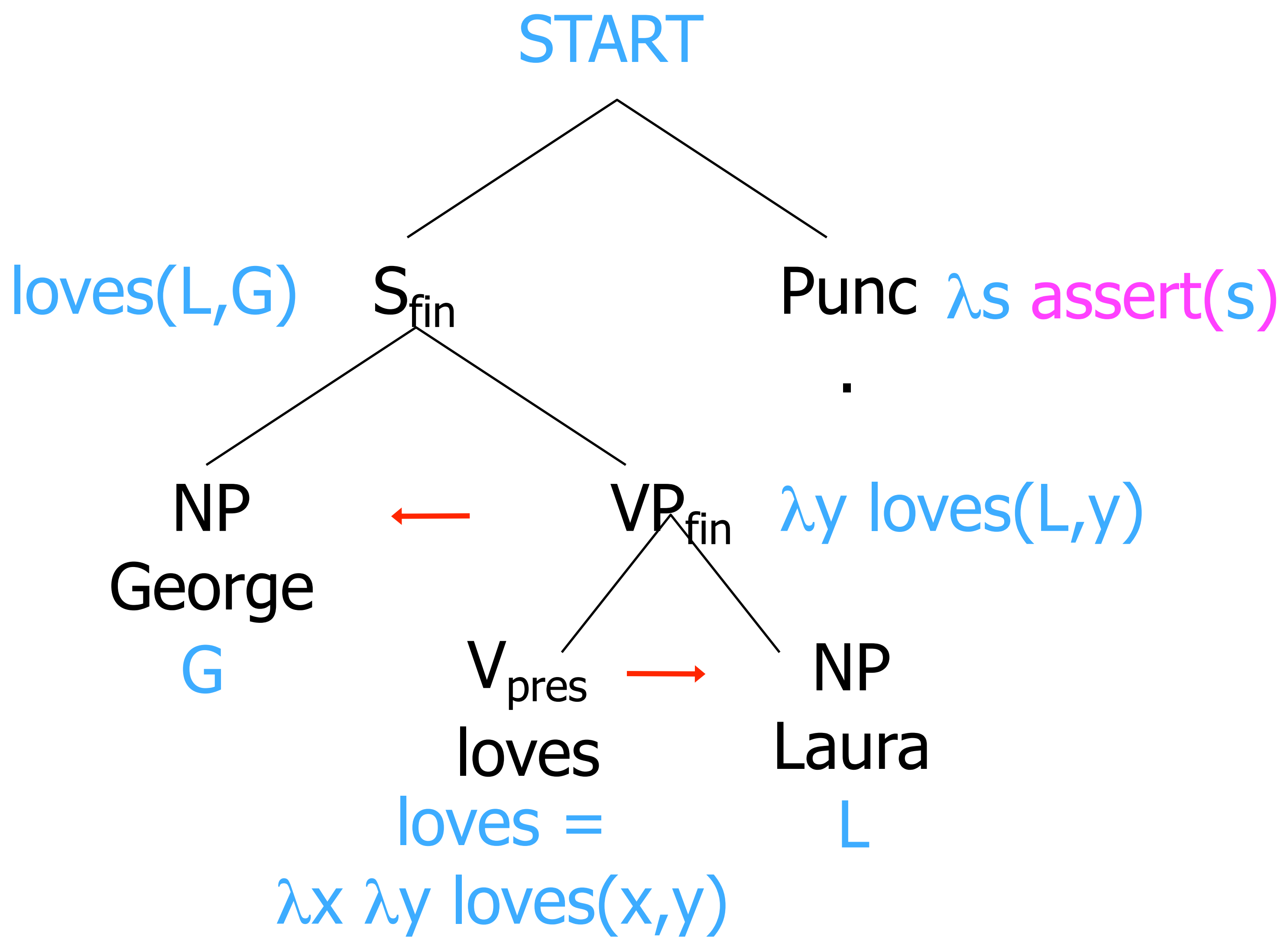
Compositional Semantics



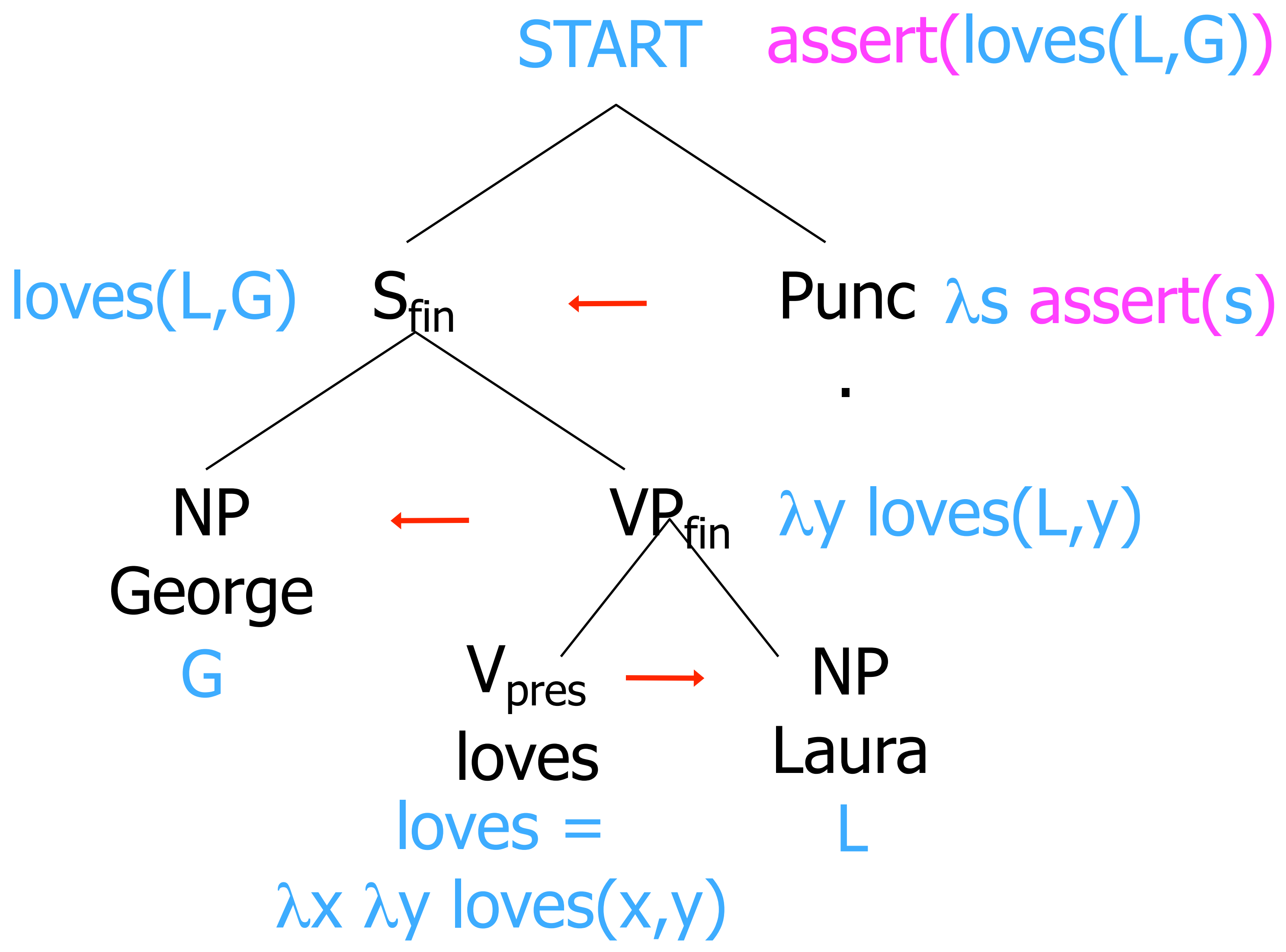
Compositional Semantics



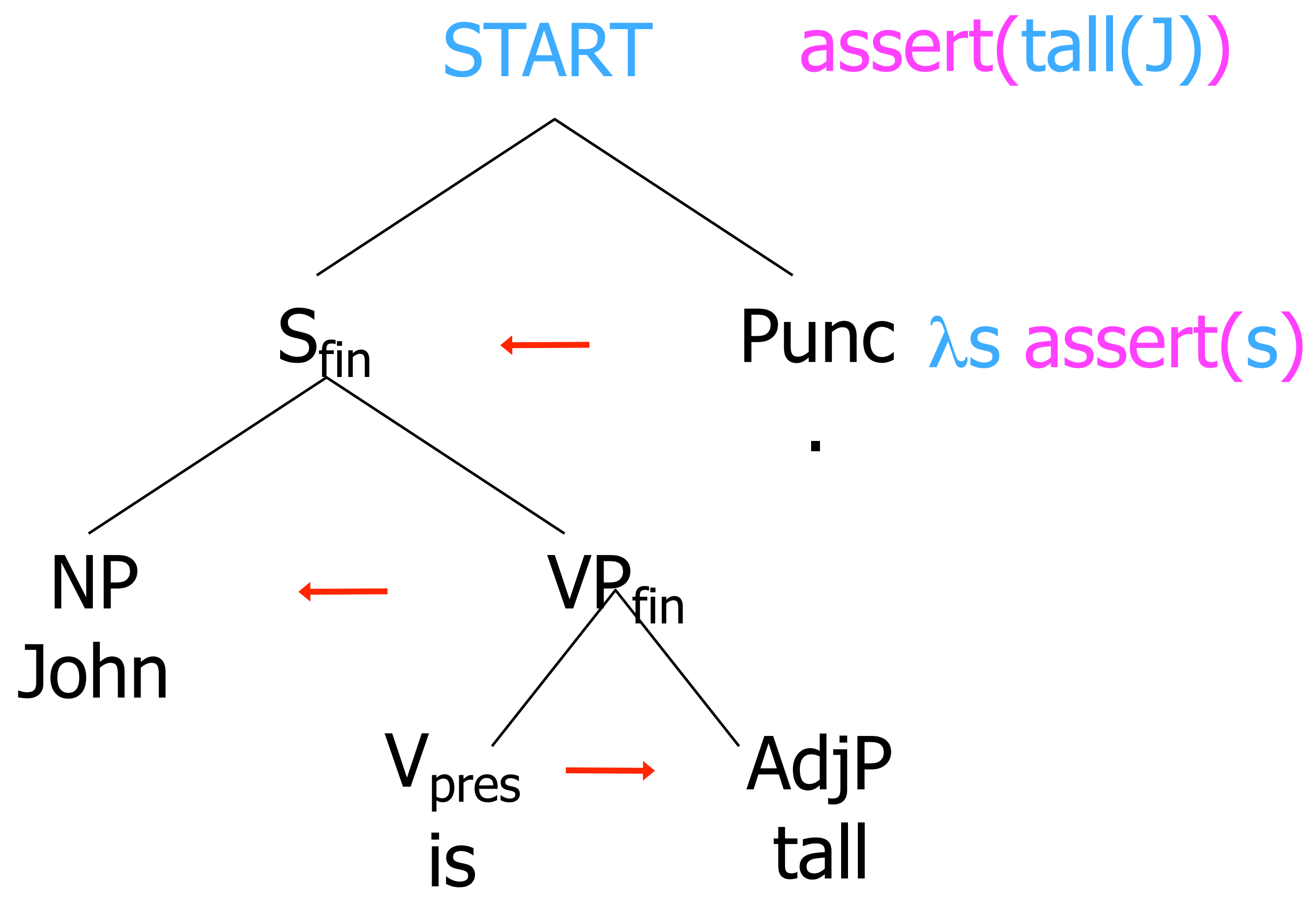
Compositional Semantics



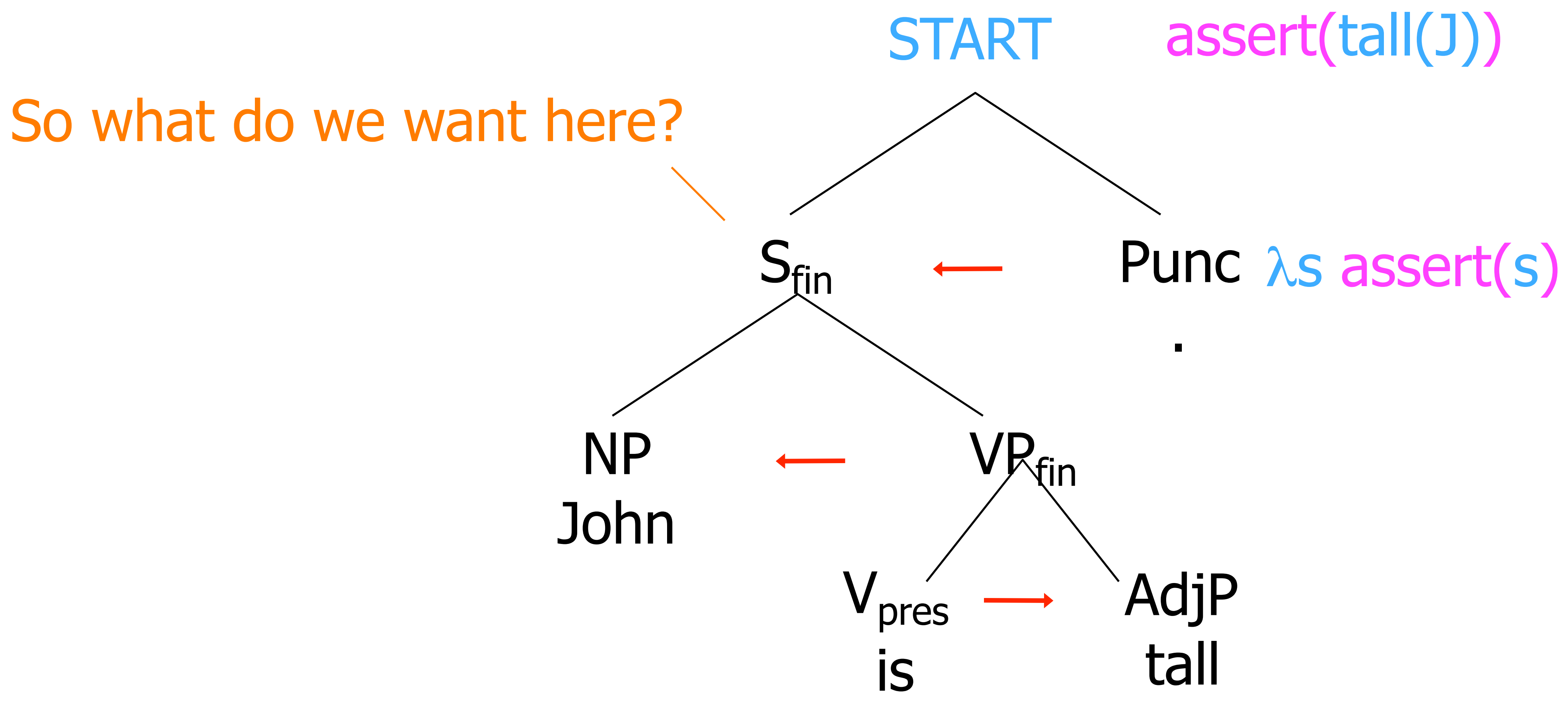
Compositional Semantics



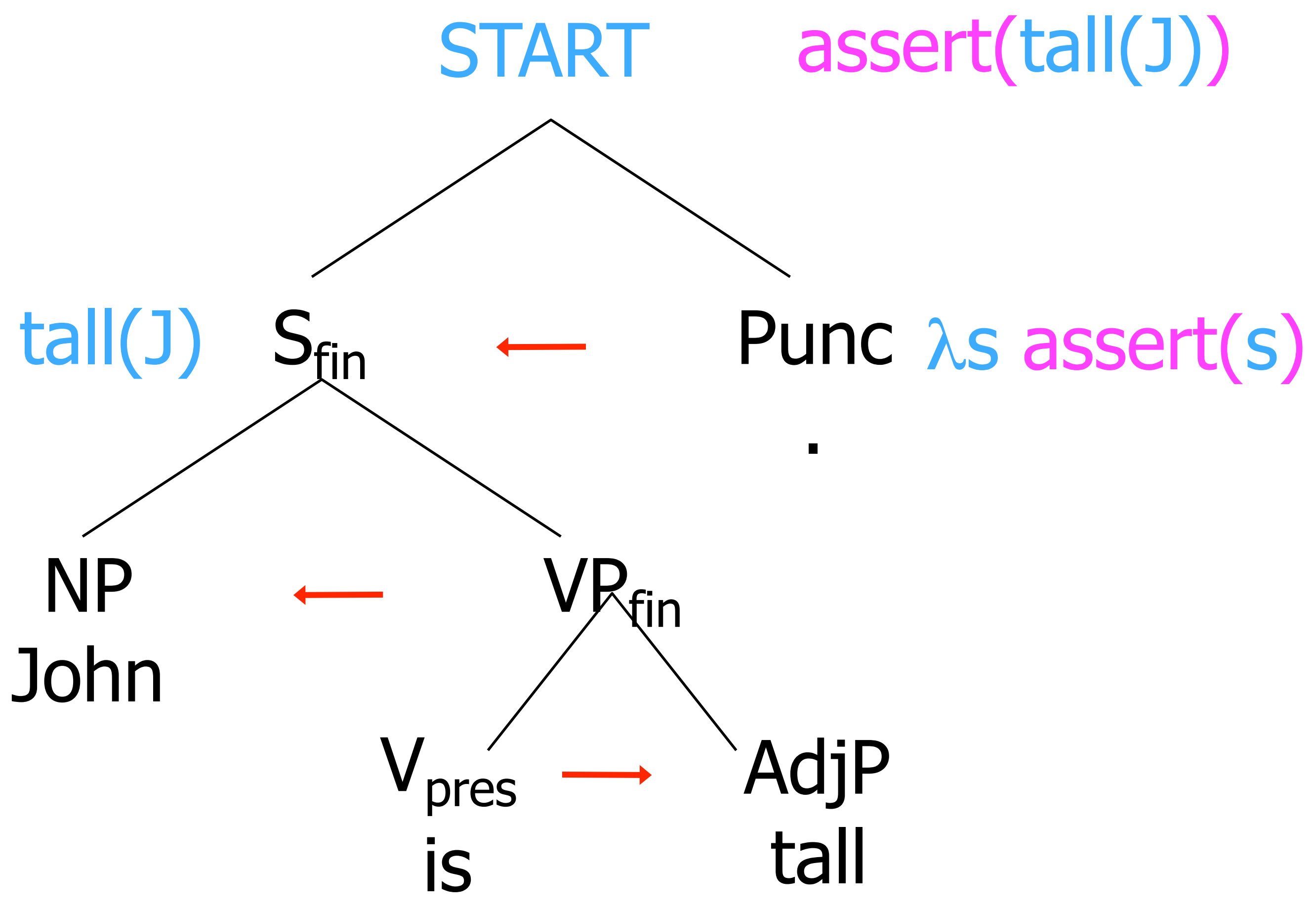
Compositional Semantics



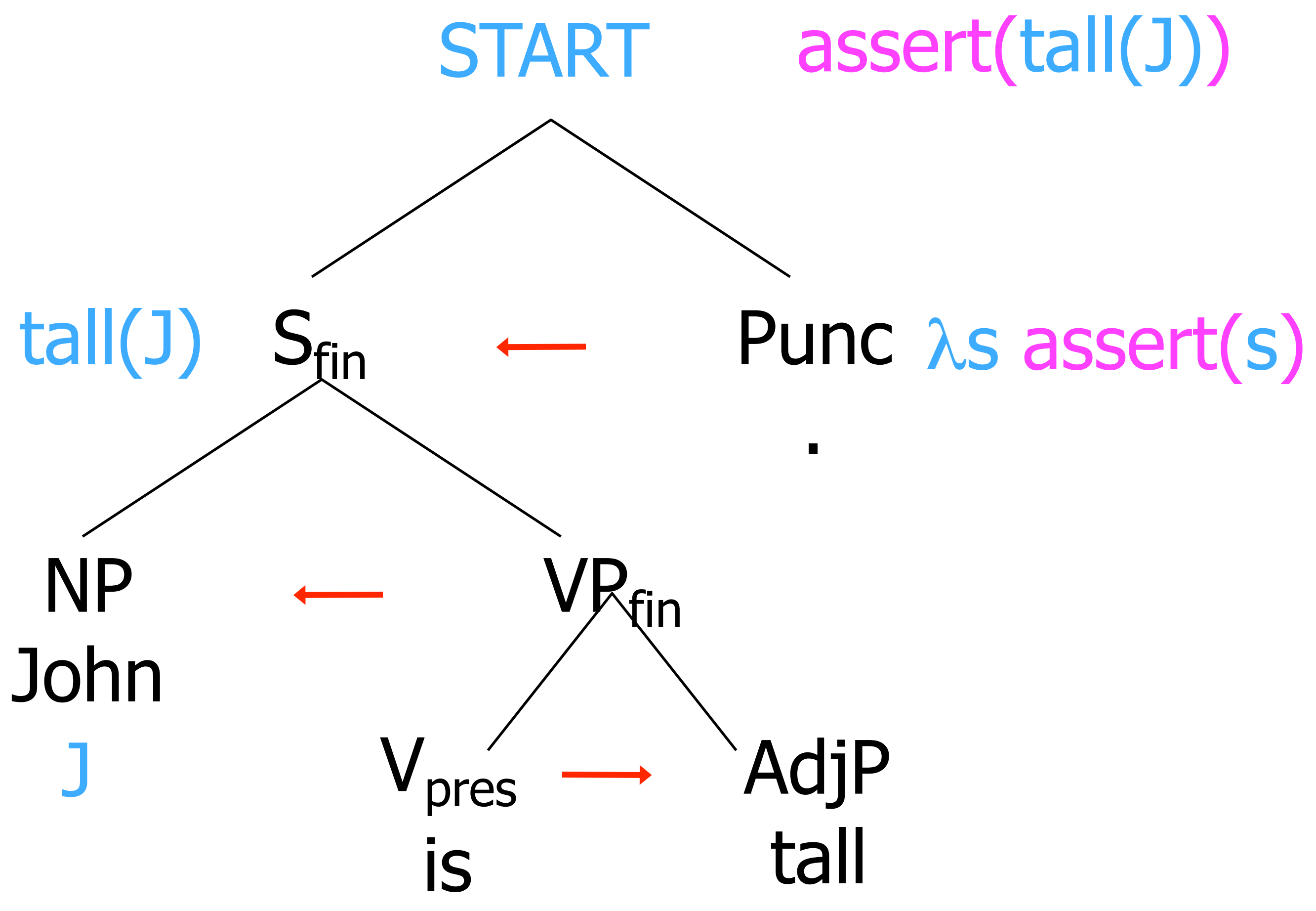
Compositional Semantics



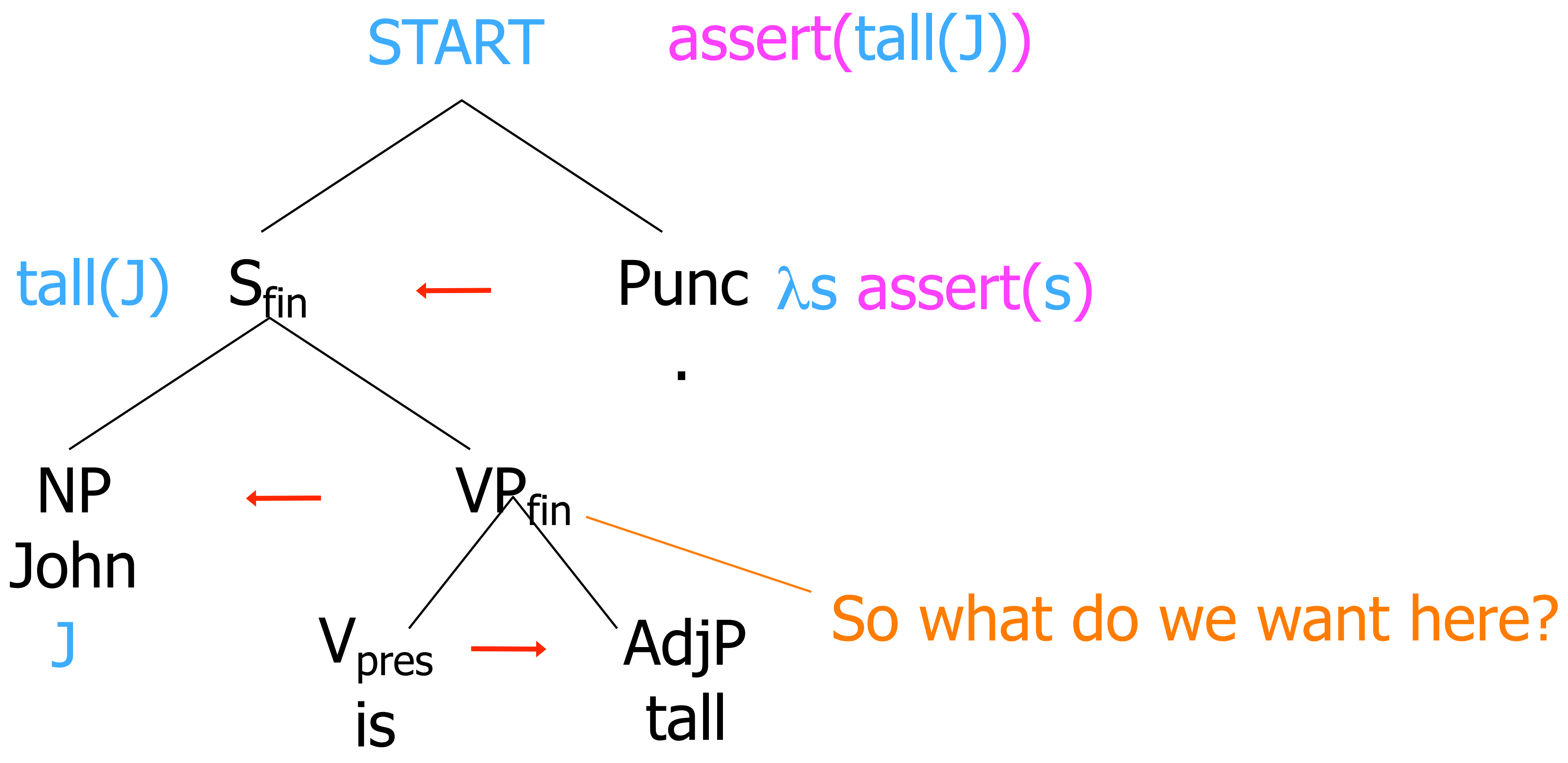
Compositional Semantics



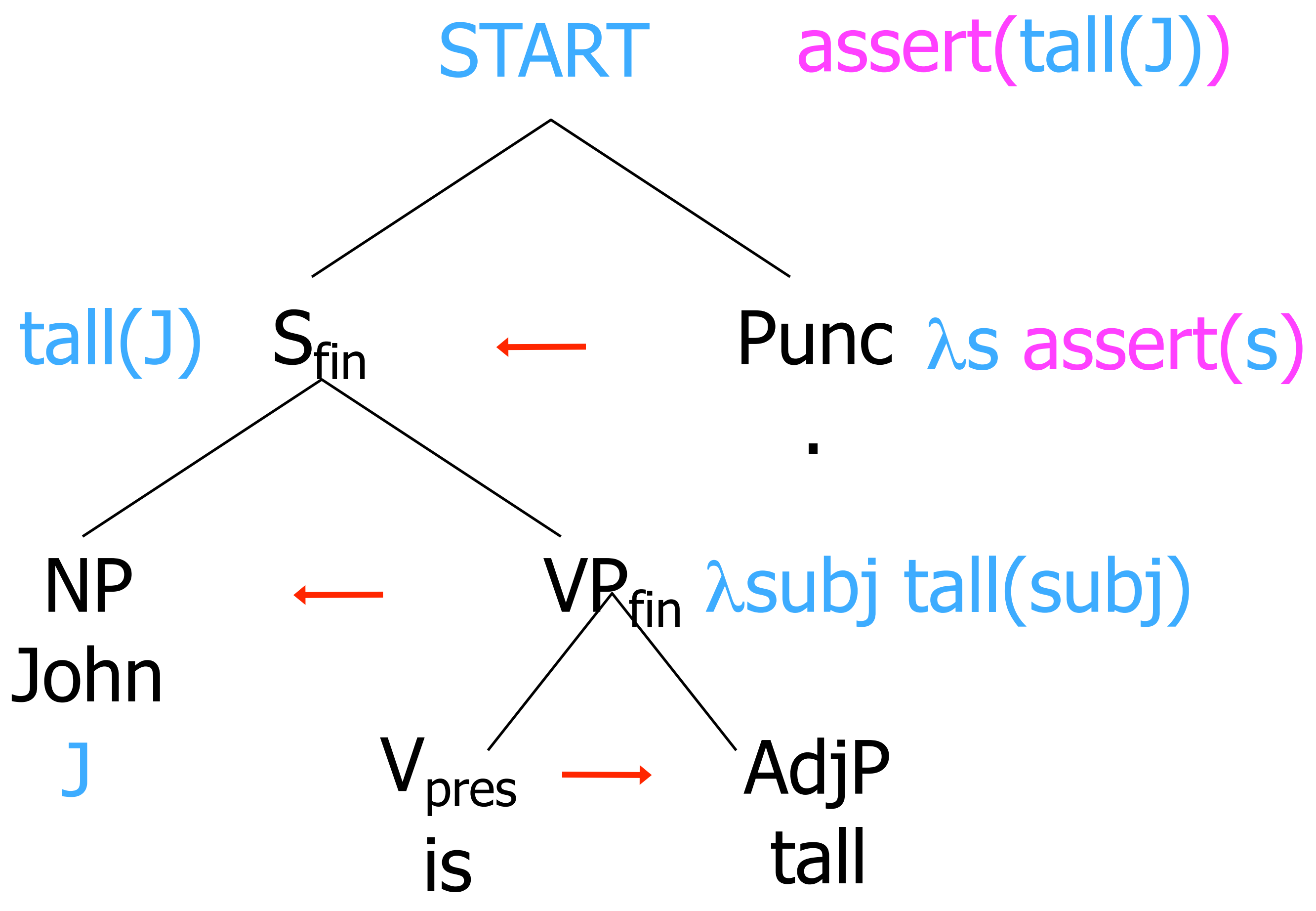
Compositional Semantics



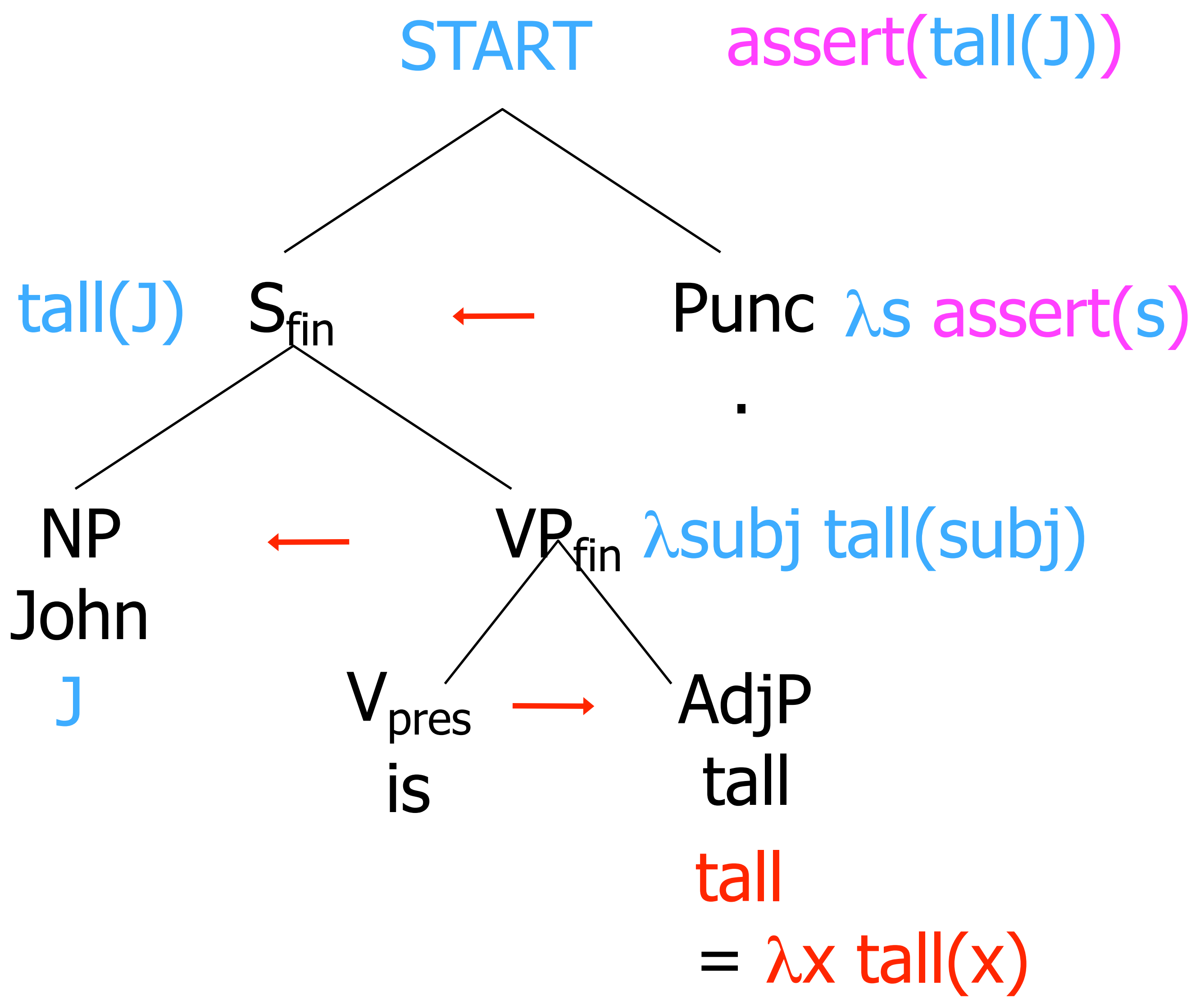
Compositional Semantics



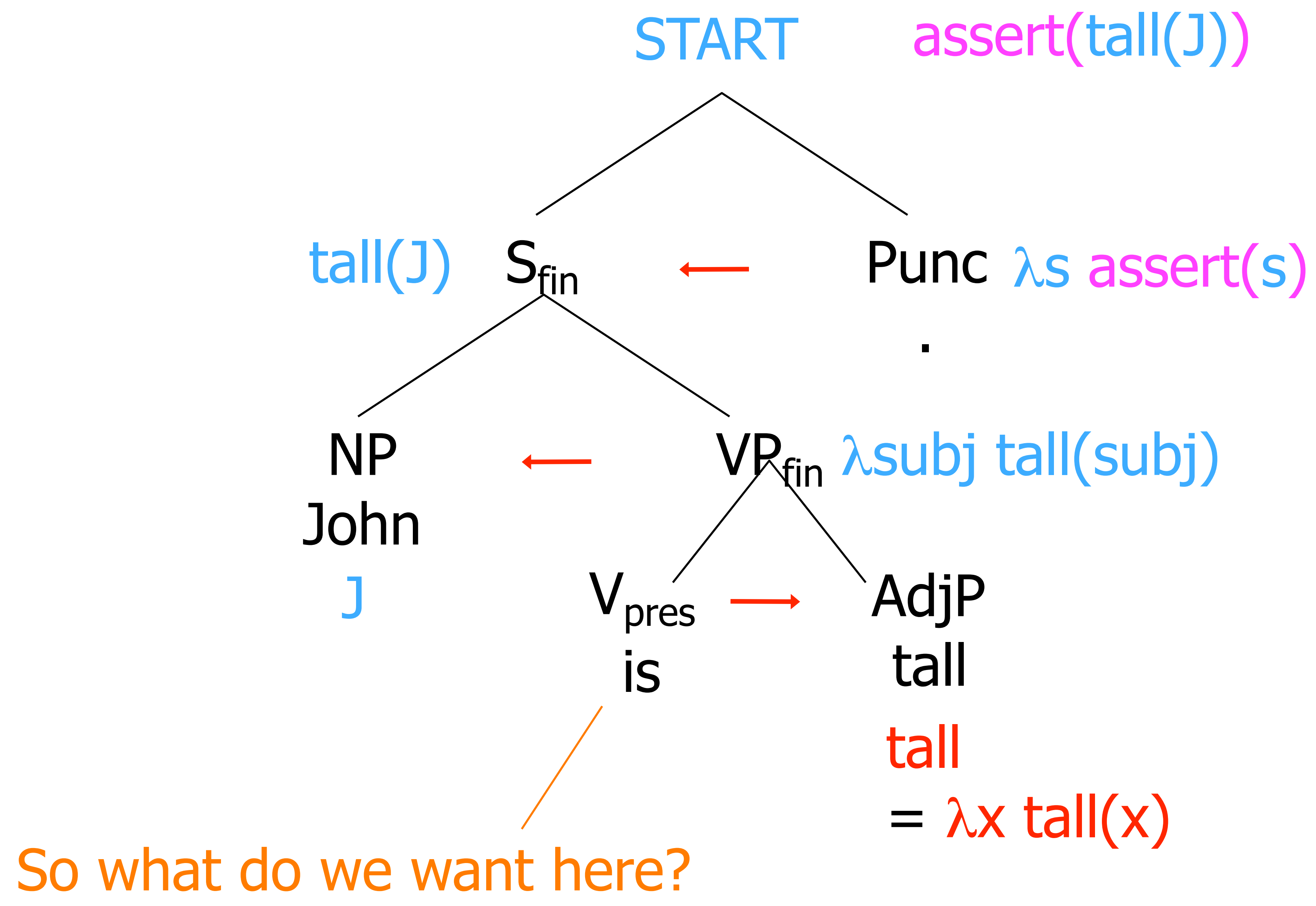
Compositional Semantics



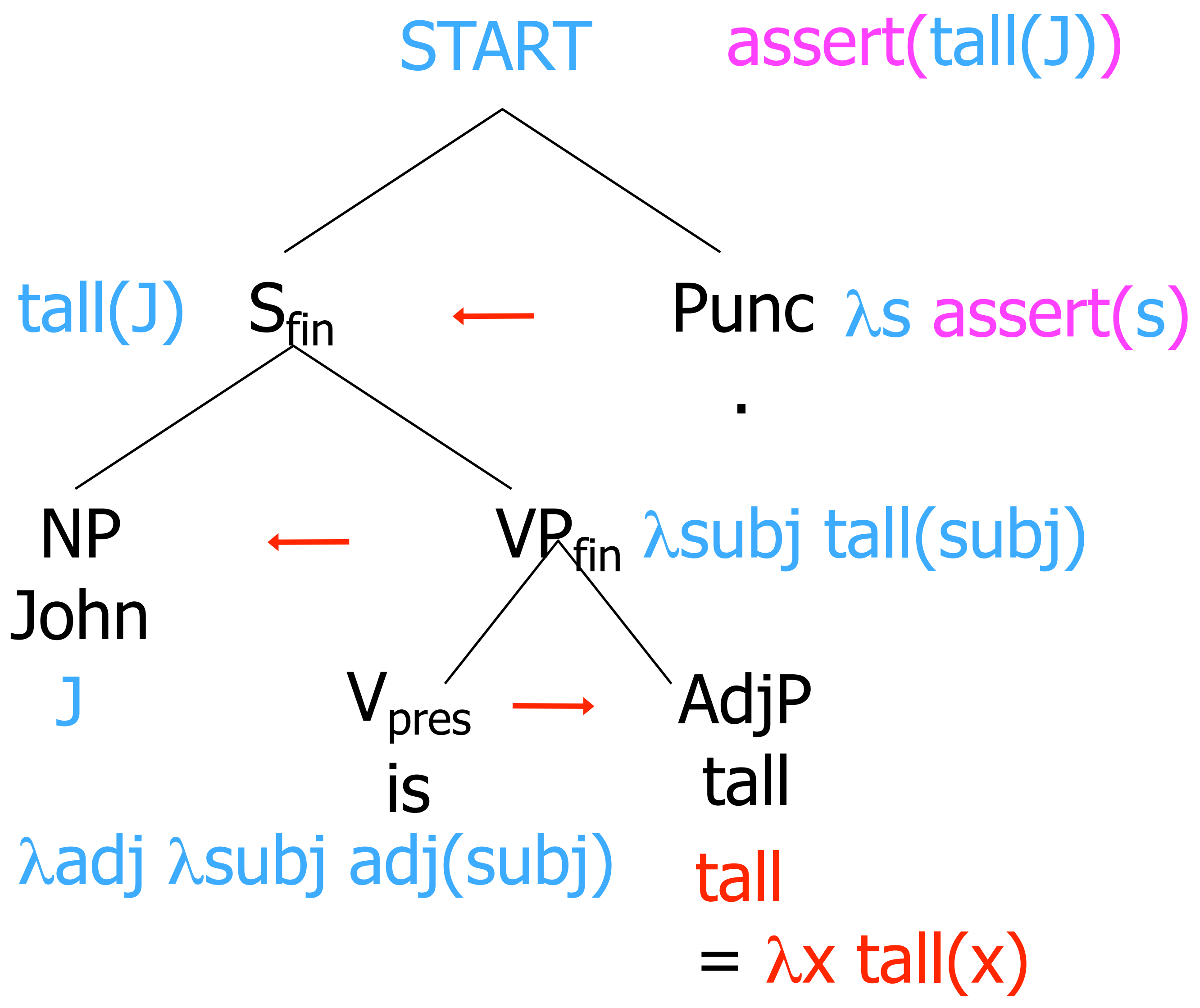
Compositional Semantics



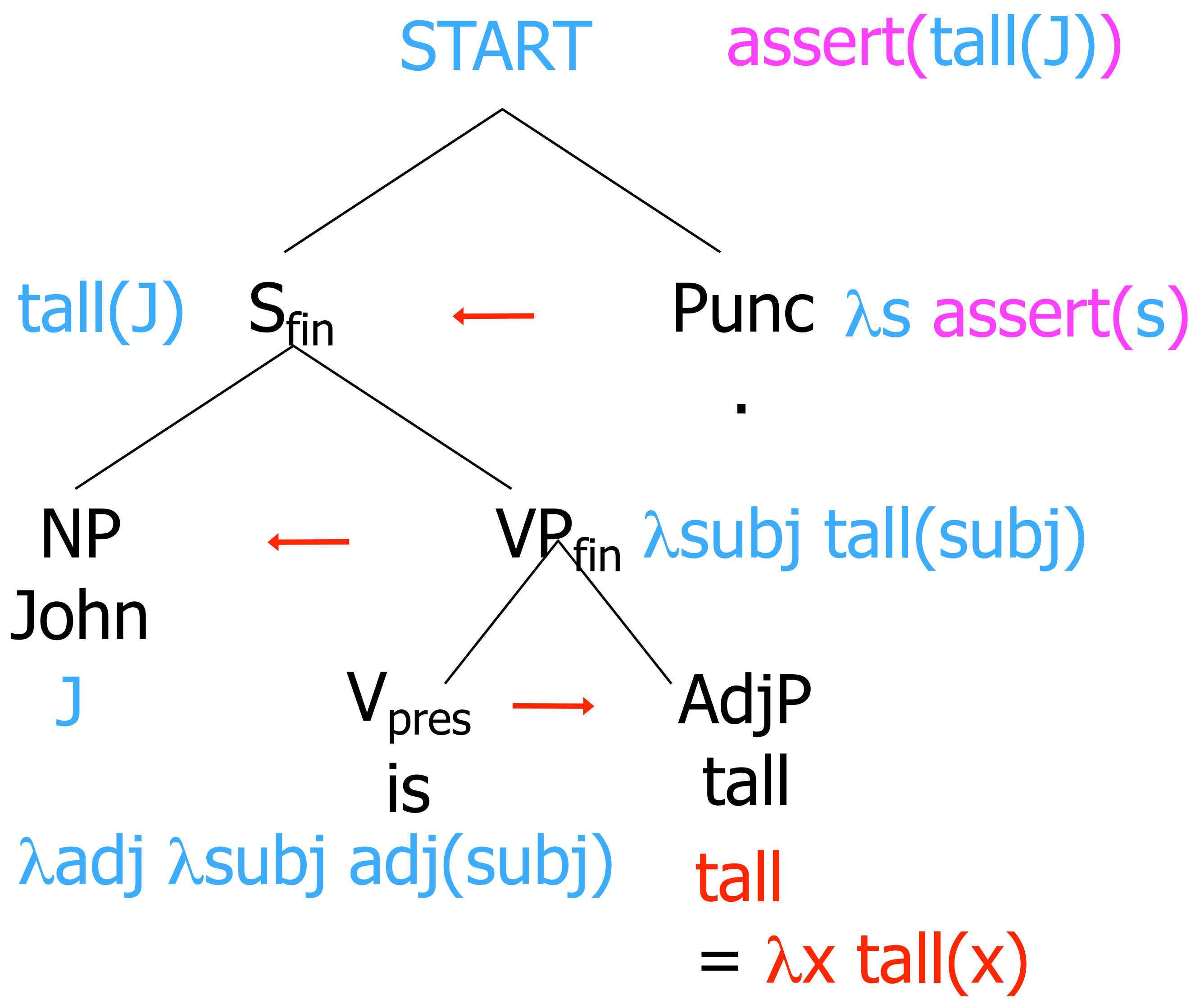
Compositional Semantics



Compositional Semantics



Compositional Semantics

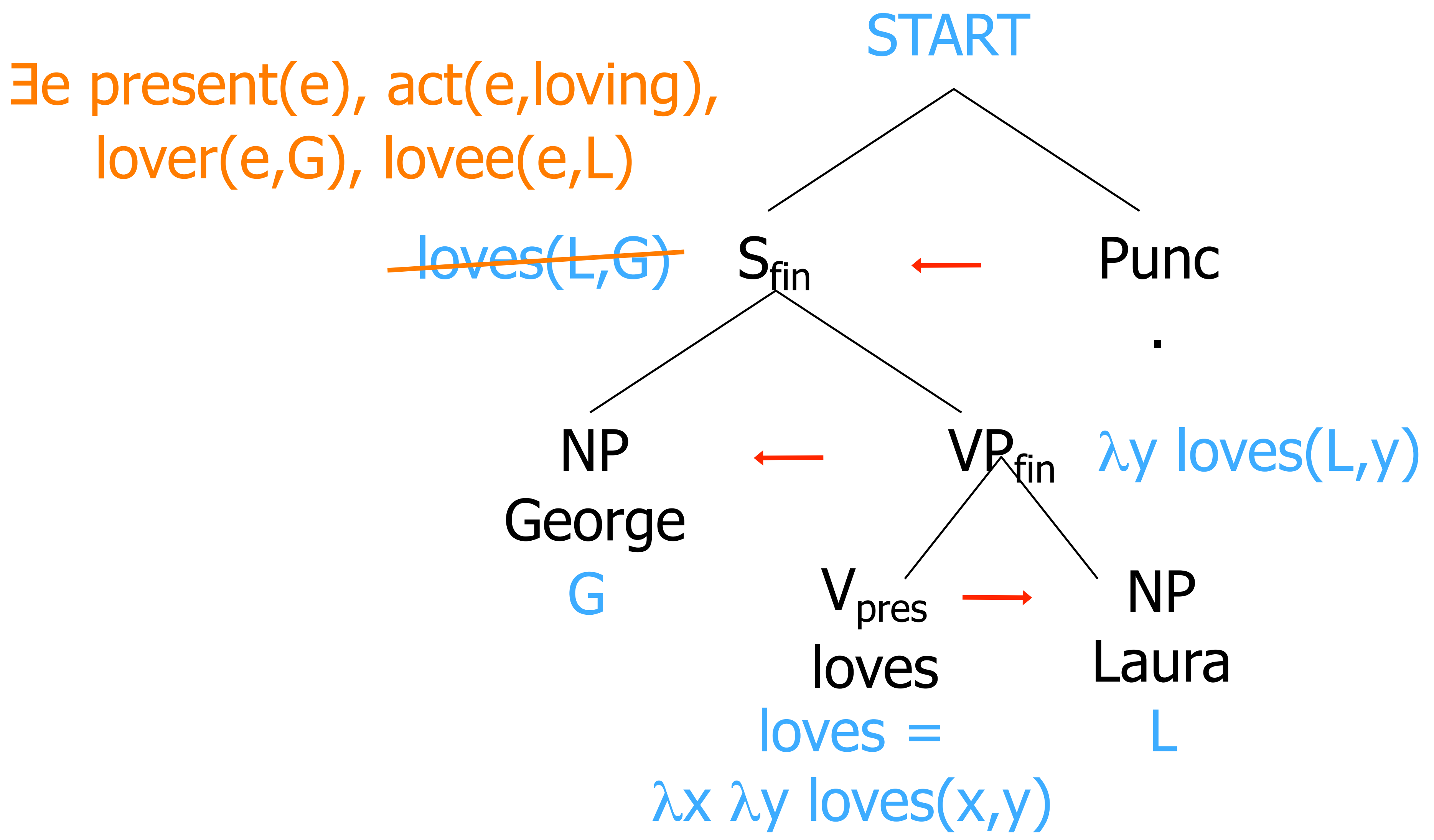


$(\lambda adj\ \lambda subj\ adj(subj))(\lambda x\ tall(x))$

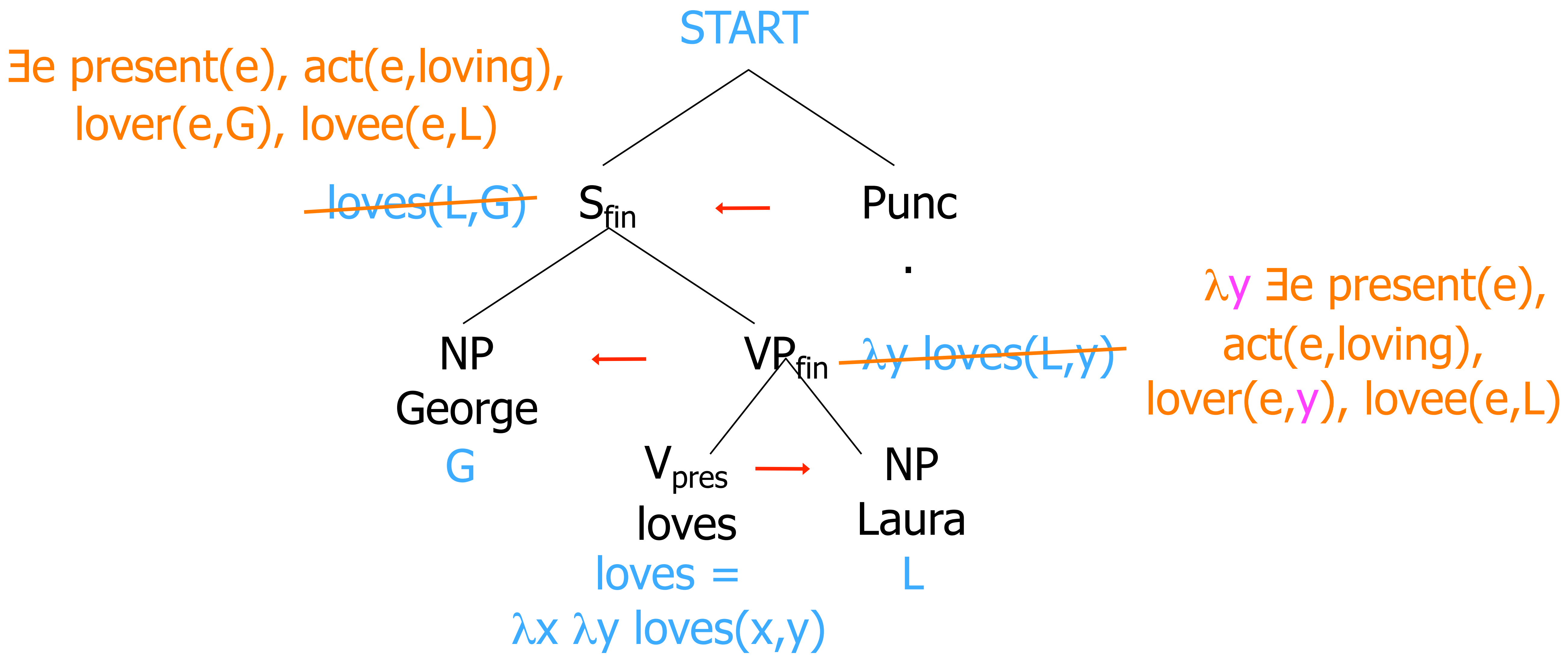
$= \quad \lambda subj\ (\lambda x\ tall(x))(subj)$

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

Compositional Semantics

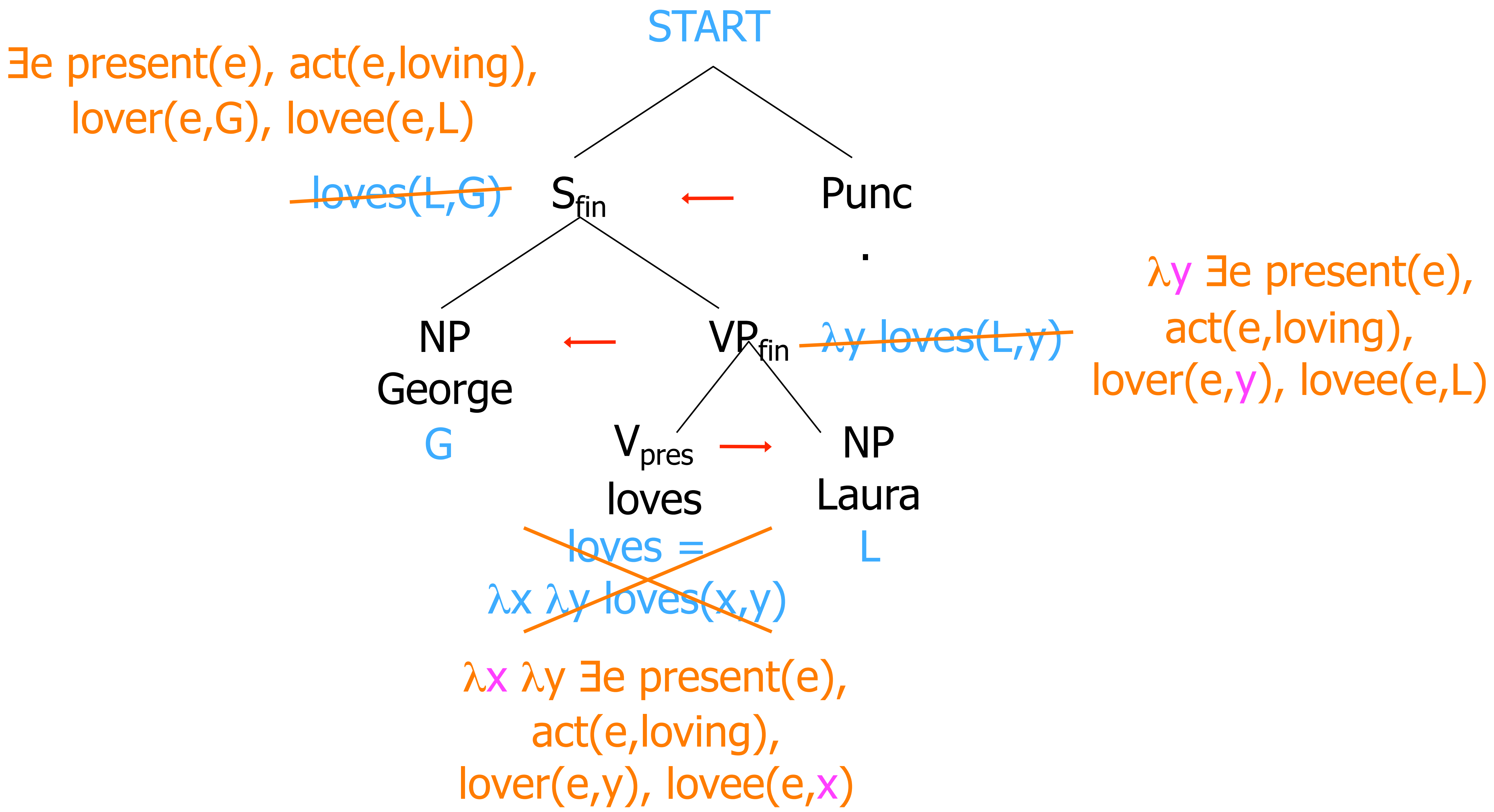


Compositional Semantics

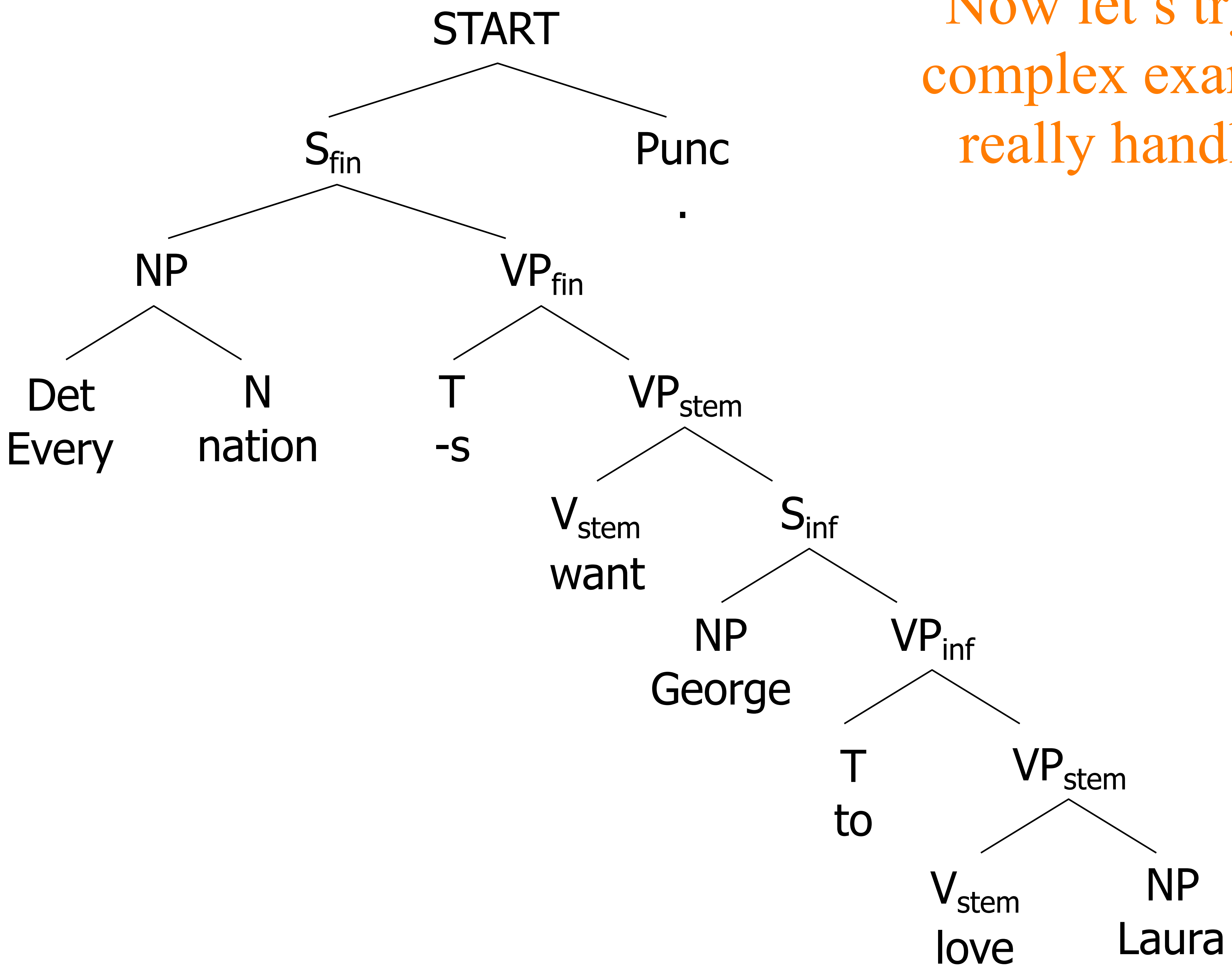


START
∃e present(e), act(e,loving), lover(e,G), lovee(e,L)
~~loves(L,G)~~
S_fin
Punc
.
λy ∃e present(e), act(e,loving), lover(e,y), lovee(e,L)
NP
George
G
VP_fin
~~λy loves(L,y)~~
V_pres
loves
loves = λx λy loves(x,y)
NP
Laura
L

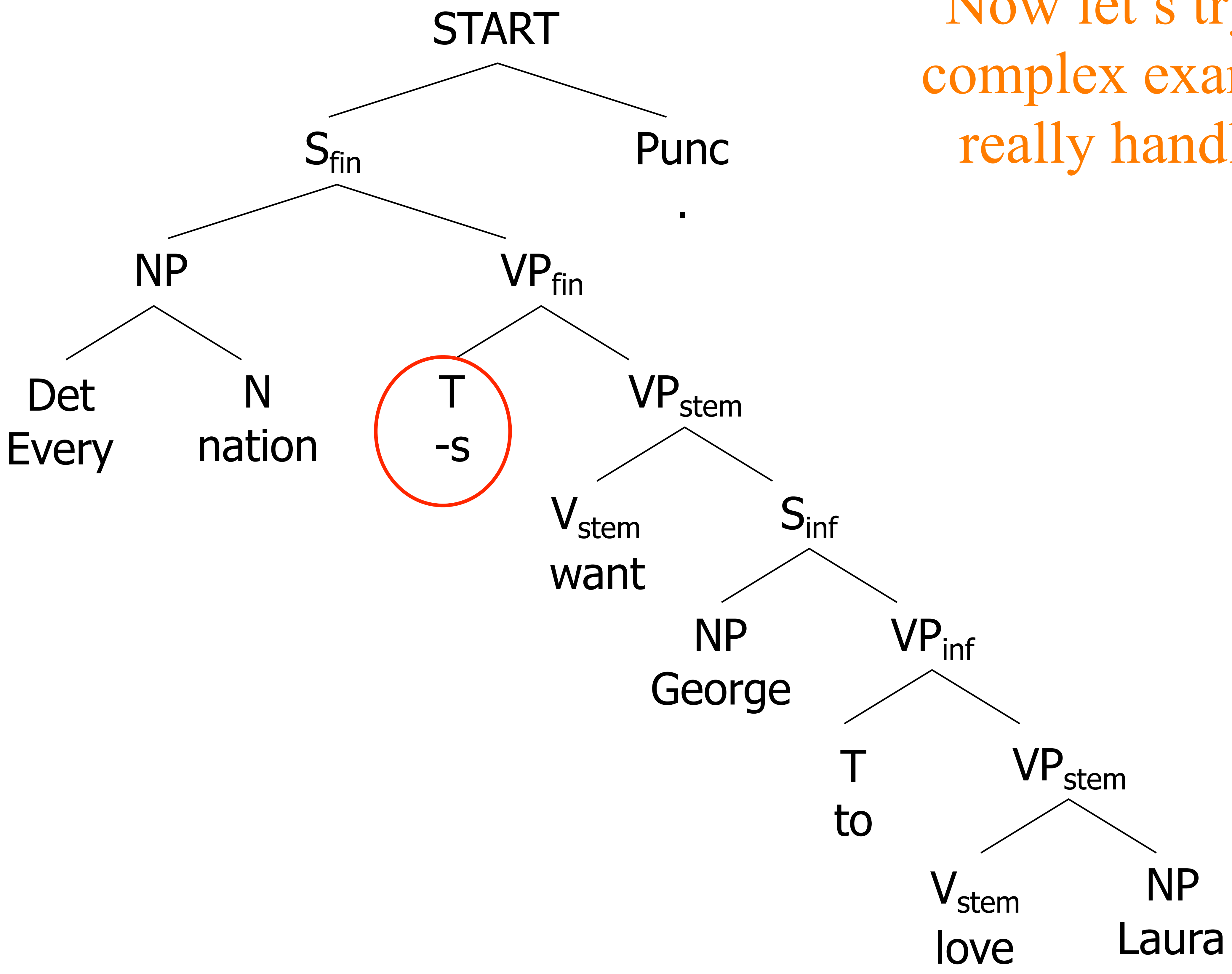
Compositional Semantics


START
∃e present(e), act(e,loving), lover(e,G), lovee(e,L)
~~loves(L,G)~~
S fin
Punc
.
λy ∃e present(e), act(e,loving), lover(e,y), lovee(e,L)
NP
George
G
VP fin
~~λy loves(L,y)~~
V pres
loves
~~loves = λx λy loves(x,y)~~
NP
Laura
L
λx λy ∃e present(e), act(e,loving), lover(e,y), lovee(e,x)

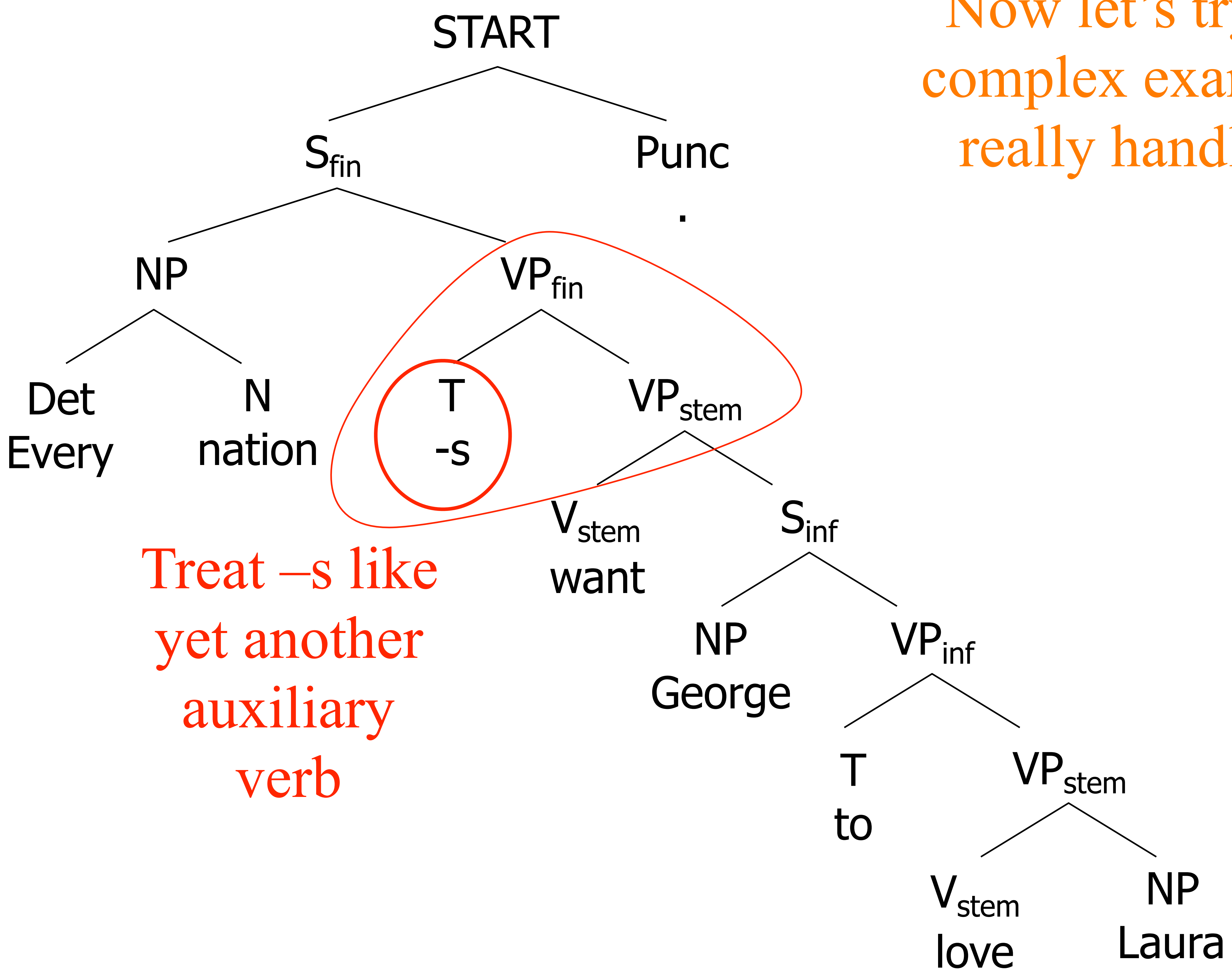
Now let's try a more complex example, and really handle tense.



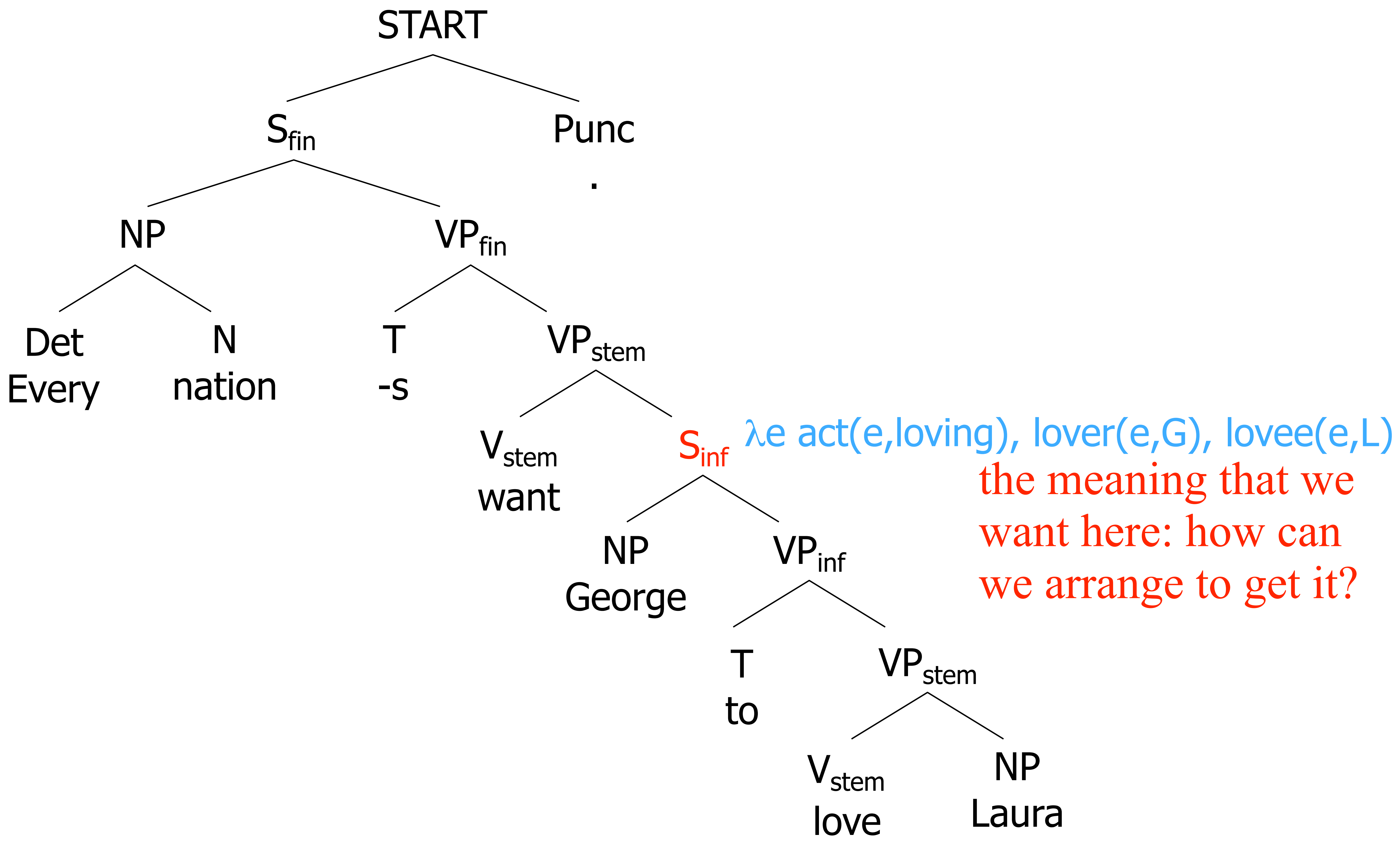
Now let's try a more complex example, and really handle tense.



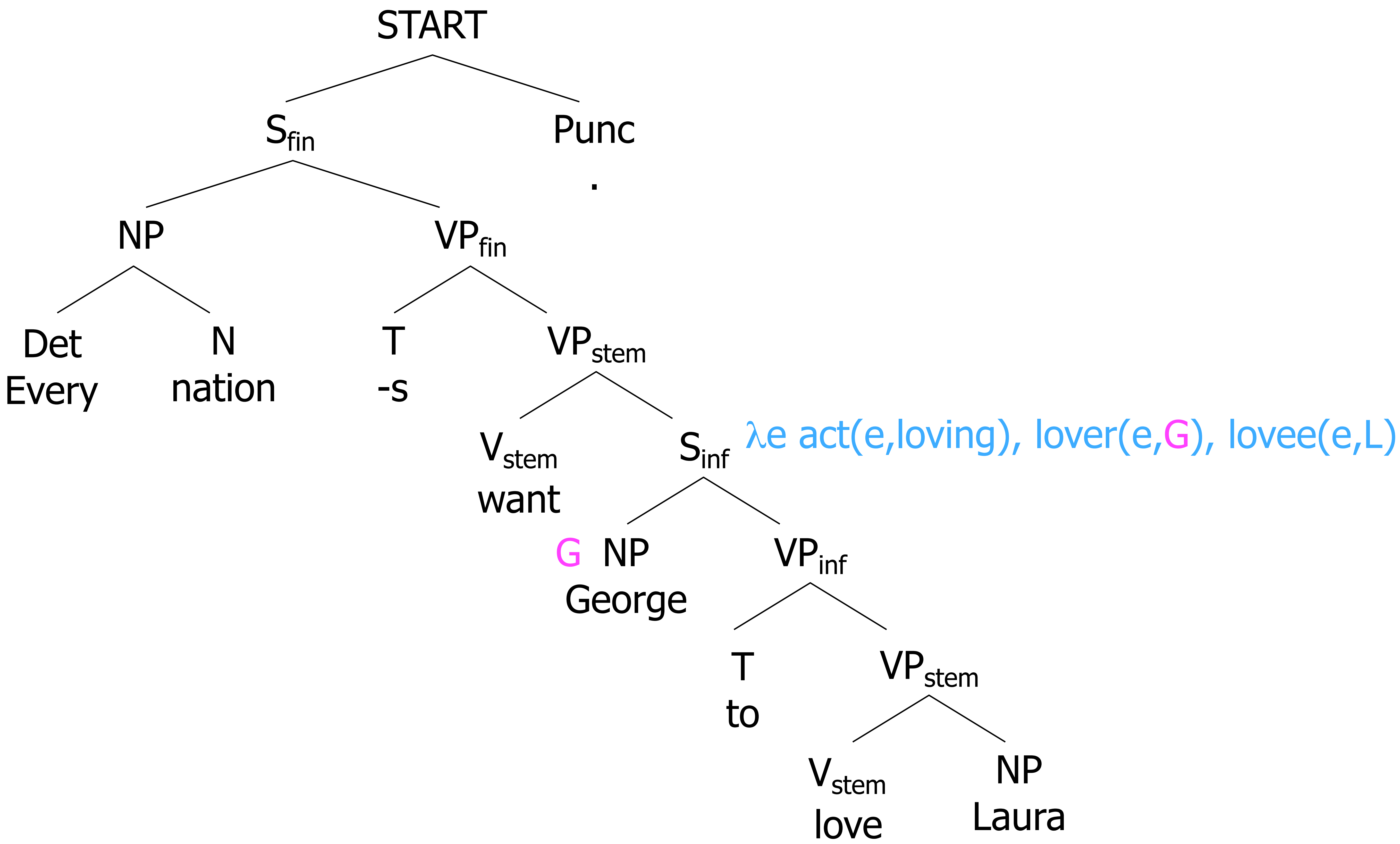
Now let's try a more complex example, and really handle tense.

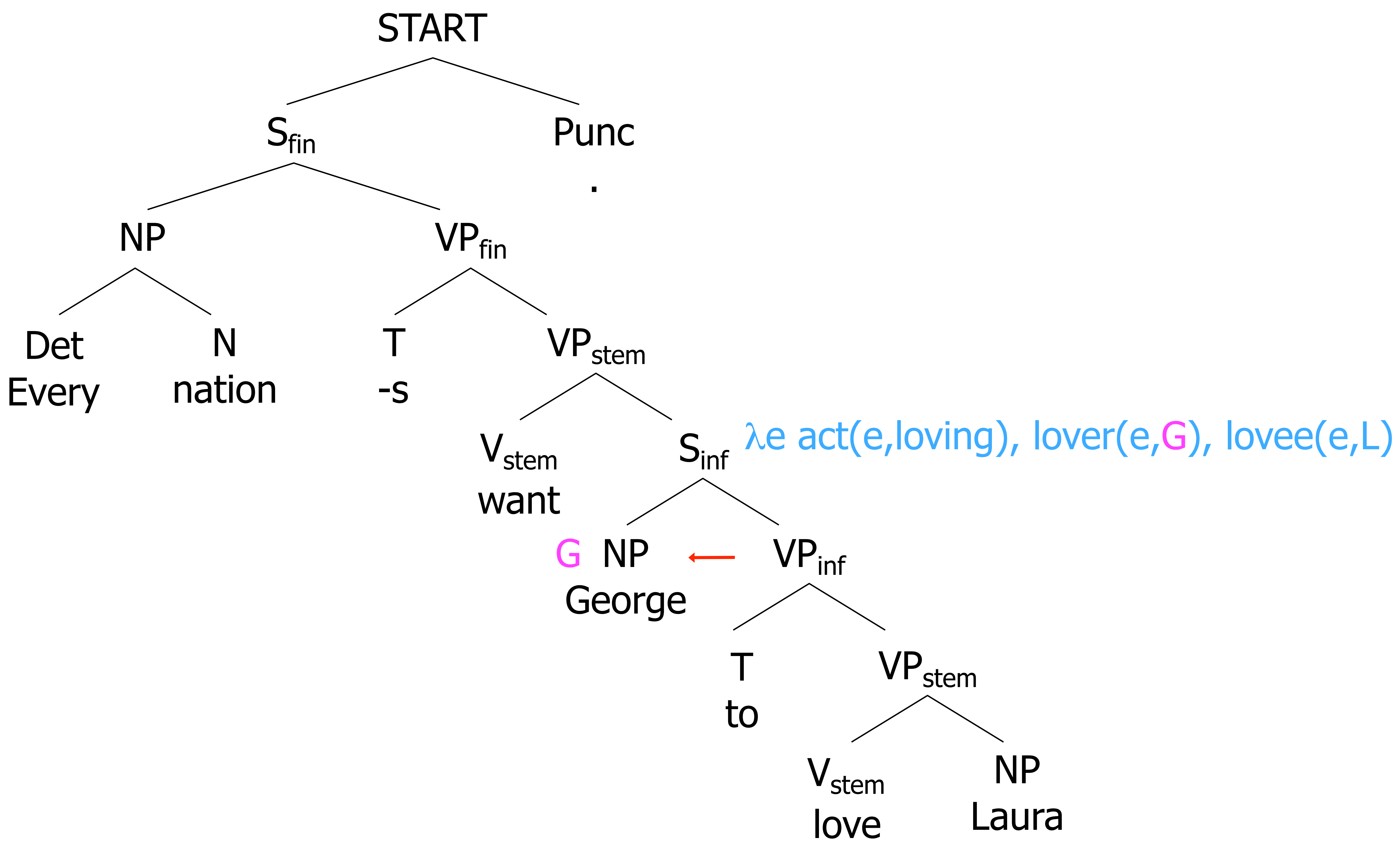


Treat –s like yet another auxiliary verb

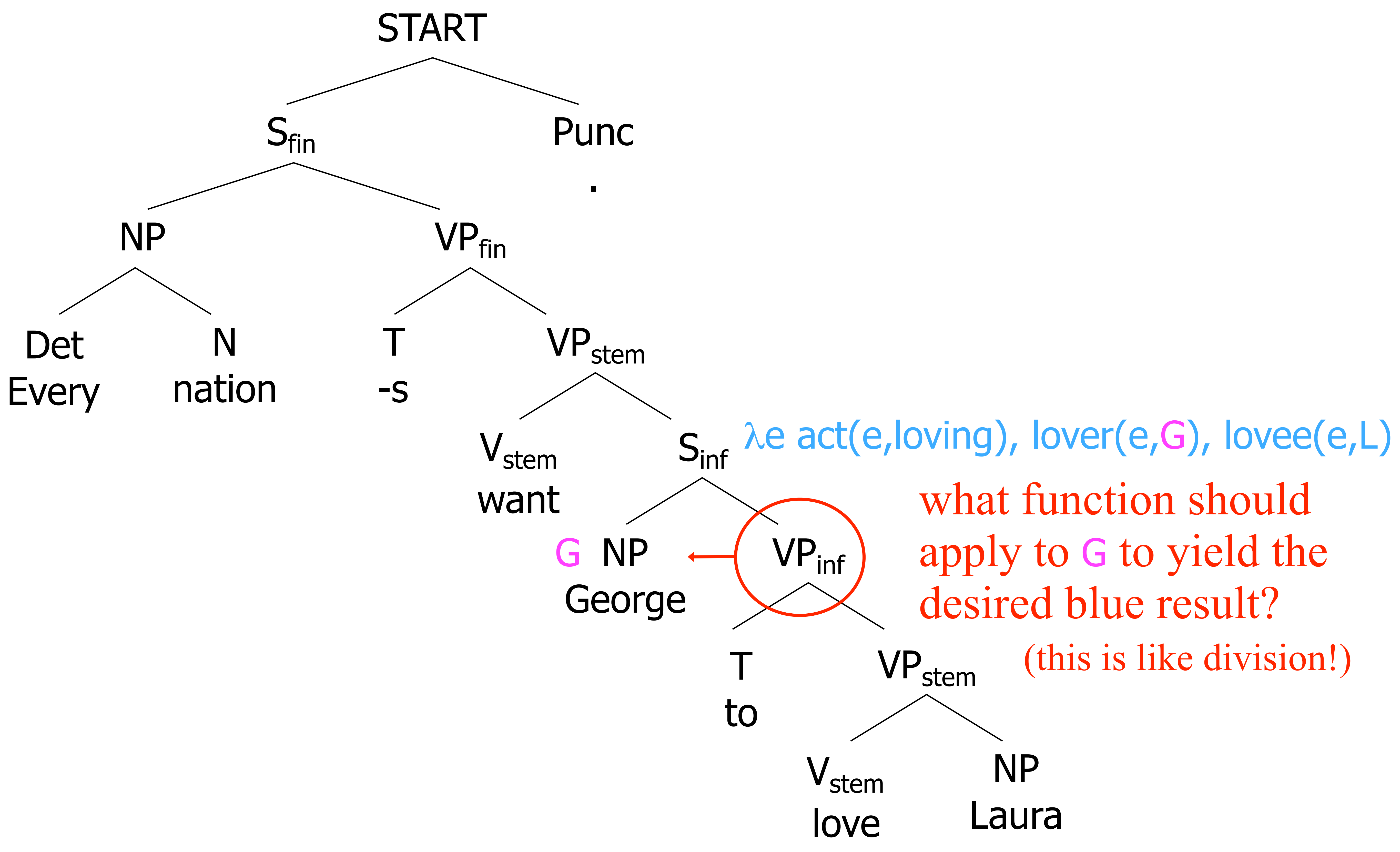


the meaning that we want here: how can we arrange to get it?


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

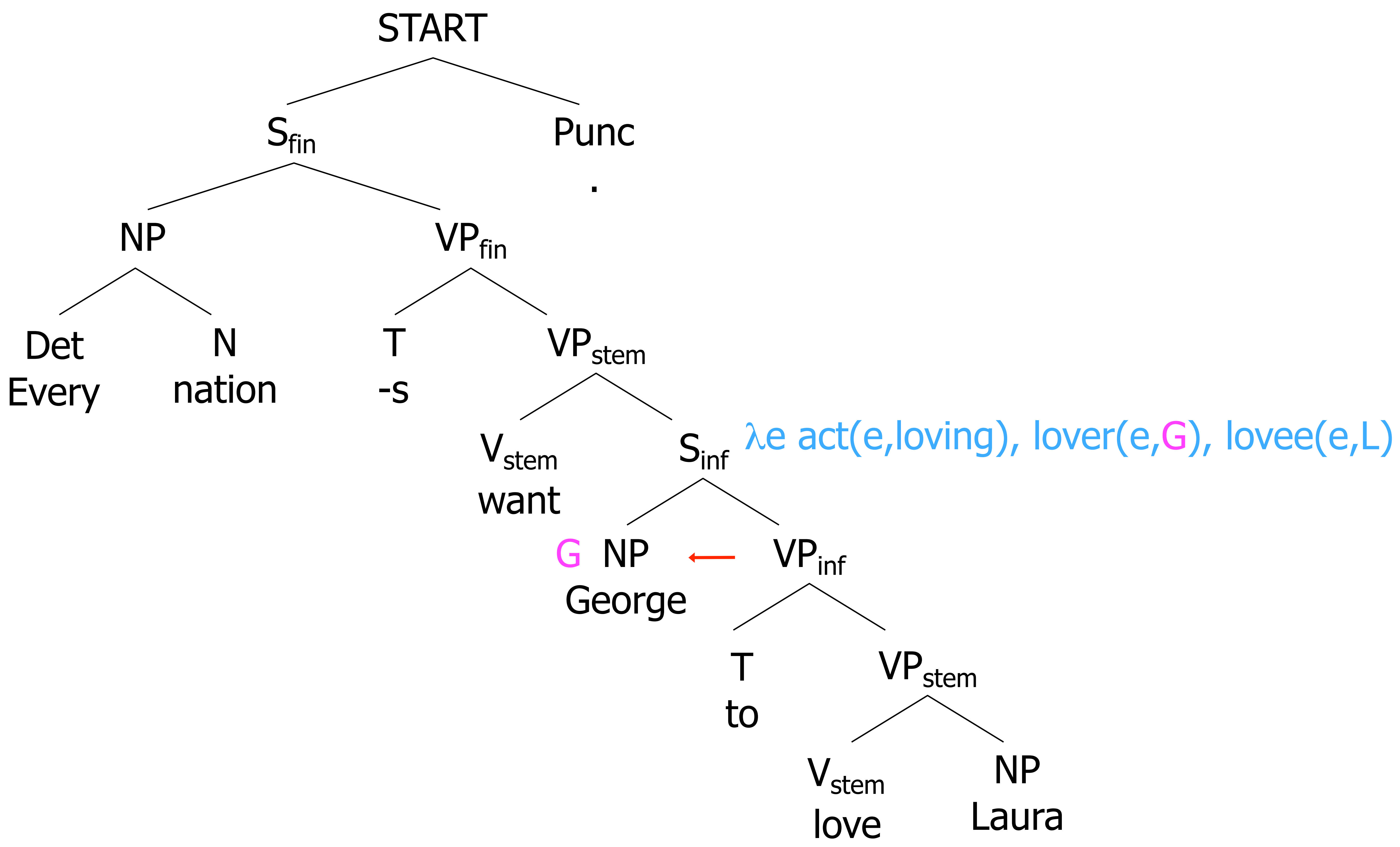


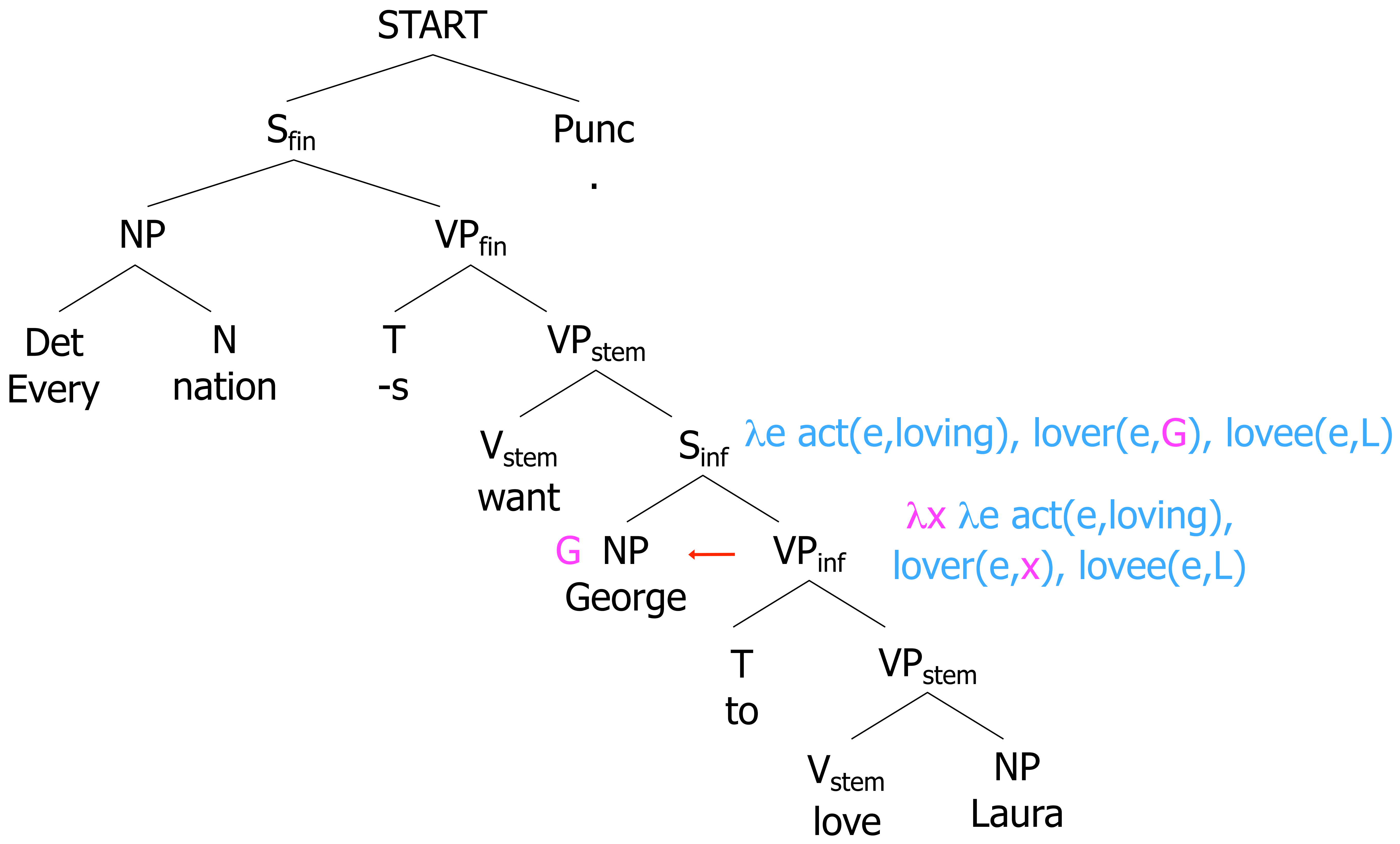
START
S_{fin}
Punc
.
NP
VP_{fin}
Det
Every
N
nation
T
-s
VP_{stem}
V_{stem}
want
S_{inf}
λe act(e,loving), lover(e,G), lovee(e,L)
G
NP
George
VP_{inf}
T
to
VP_{stem}
V_{stem}
love
NP
Laura

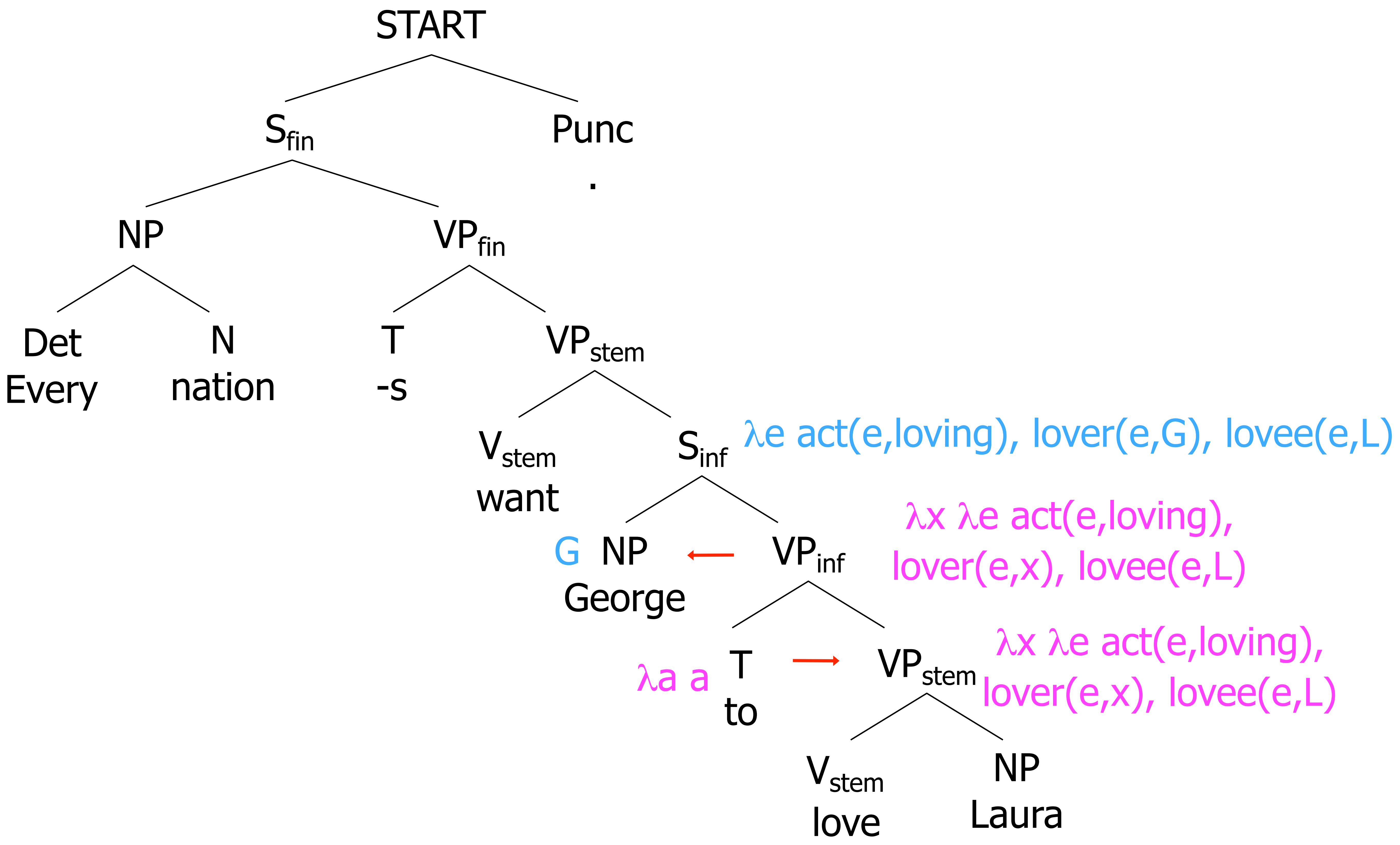


what function should apply to G to yield the desired blue result?

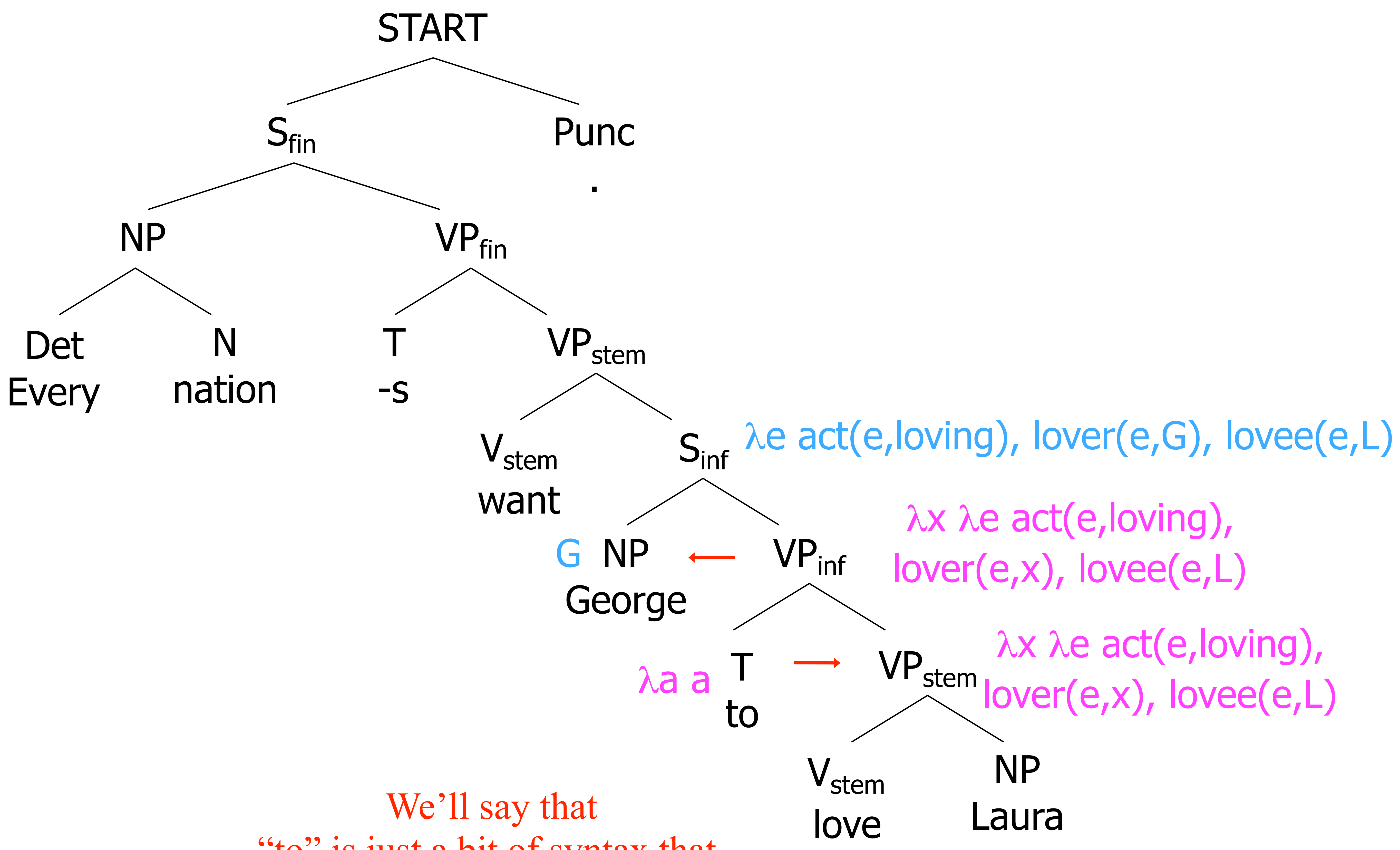
(this is like division!)


START
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

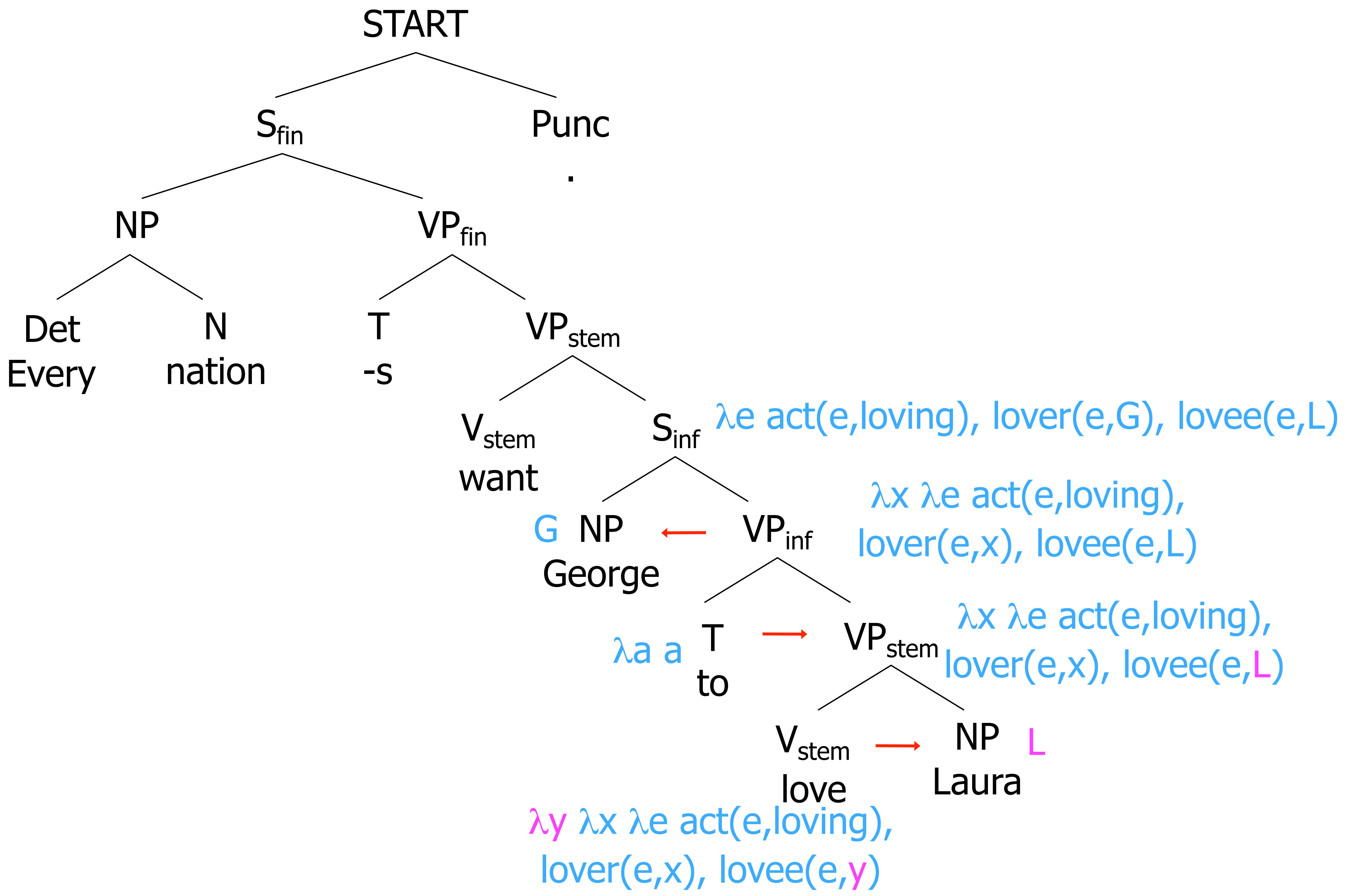

START
S_fin
Punc
.
NP
VP_fin
Det
Every
N
nation
T
-s
VP_stem
V_stem
want
S_inf
λe act(e,loving), lover(e,G), lovee(e,L)
G NP
George
VP_inf
λx λe act(e,loving), lover(e,x), lovee(e,L)
T
to
VP_stem
V_stem
love
NP
Laura

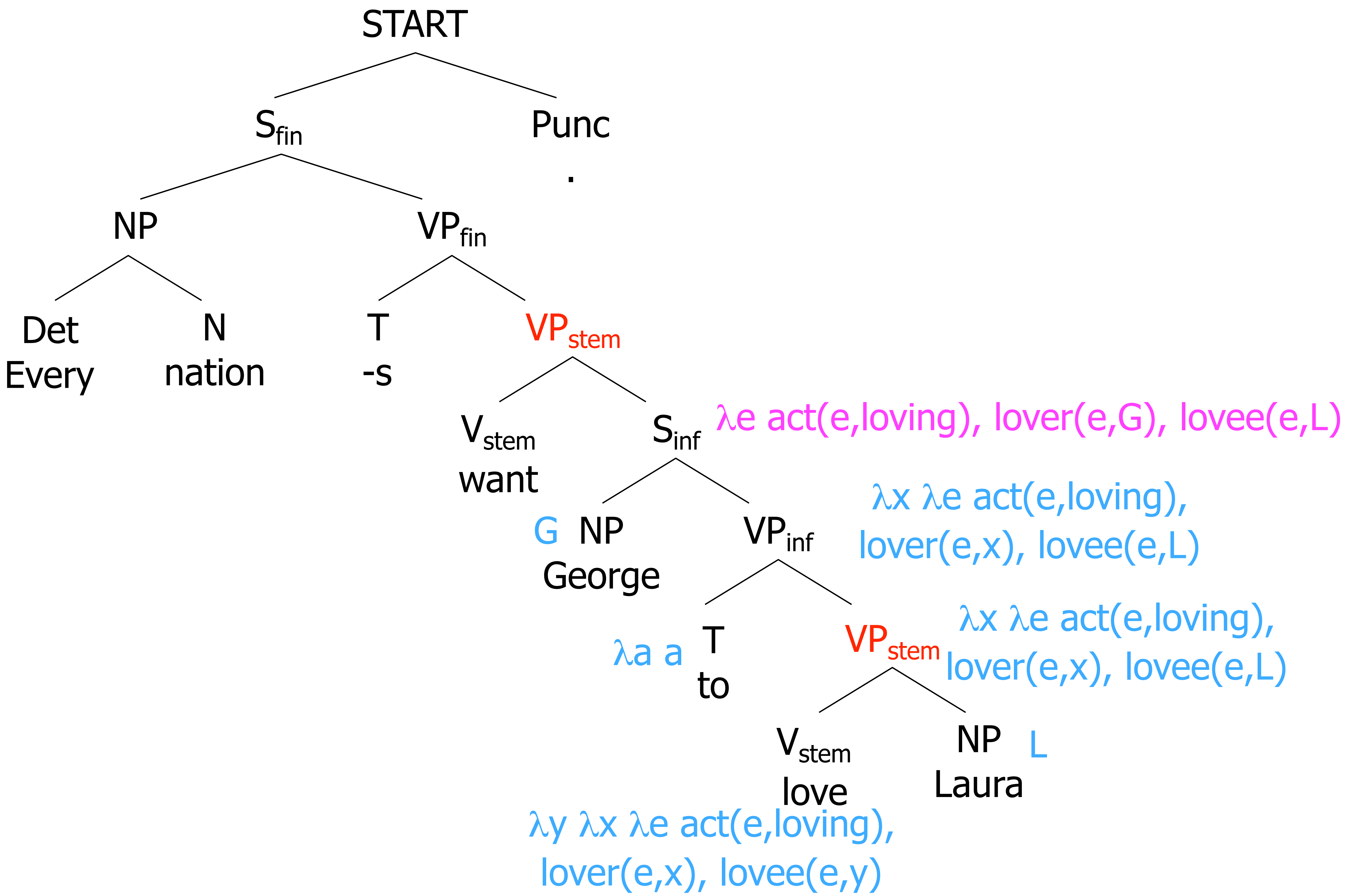


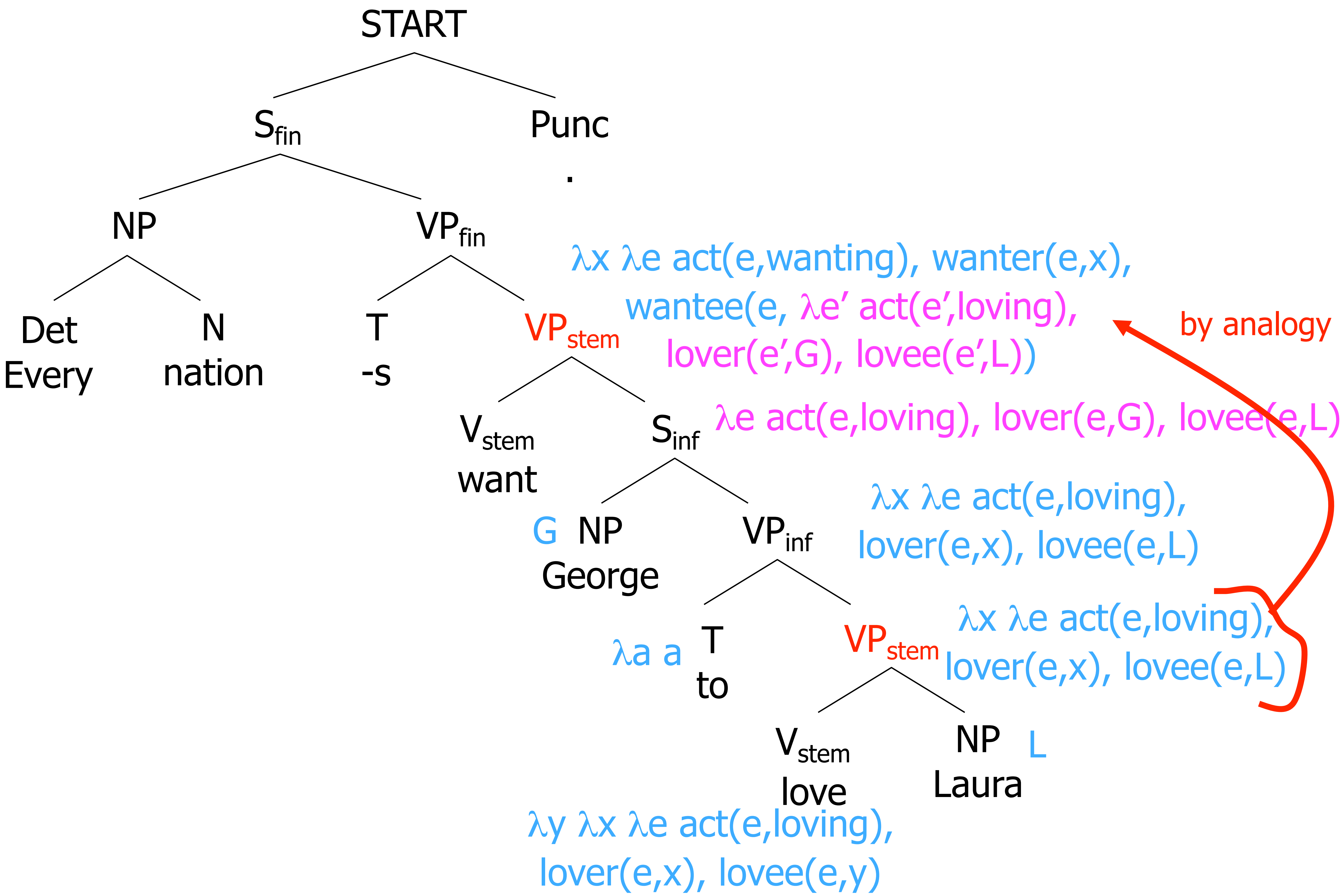
START
S_fin
Punc
.
NP
VP_fin
Det
Every
N
nation
T
-s
VP_stem
V_stem
want
S_inf
λe act(e,loving), lover(e,G), lovee(e,L)
G
NP
George
VP_inf
λx λe act(e,loving), lover(e,x), lovee(e,L)
λa a
T
to
VP_stem
λx λe act(e,loving), lover(e,x), lovee(e,L)
V_stem
love
NP
Laura

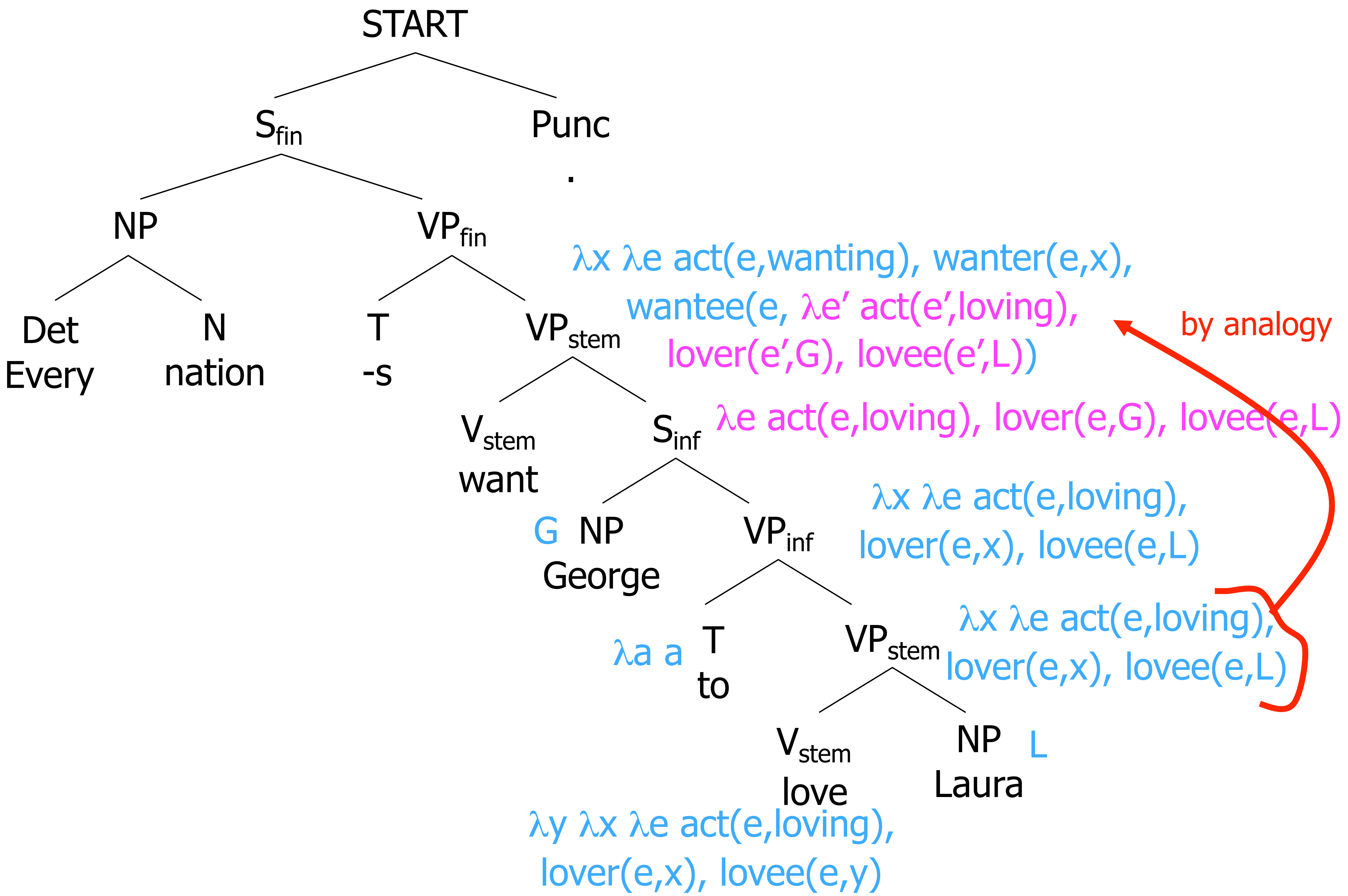

START
S_fin
Punc
.
NP
VP_fin
Det
Every
N
nation
T
-s
VP_stem
V_stem
want
S_inf
λe act(e,loving), lover(e,G), lovee(e,L)
G
NP
George
VP_inf
λx λe act(e,loving), lover(e,x), lovee(e,L)
λa a
T
to
VP_stem
λx λe act(e,loving), lover(e,x), lovee(e,L)
V_stem
love
NP
Laura

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

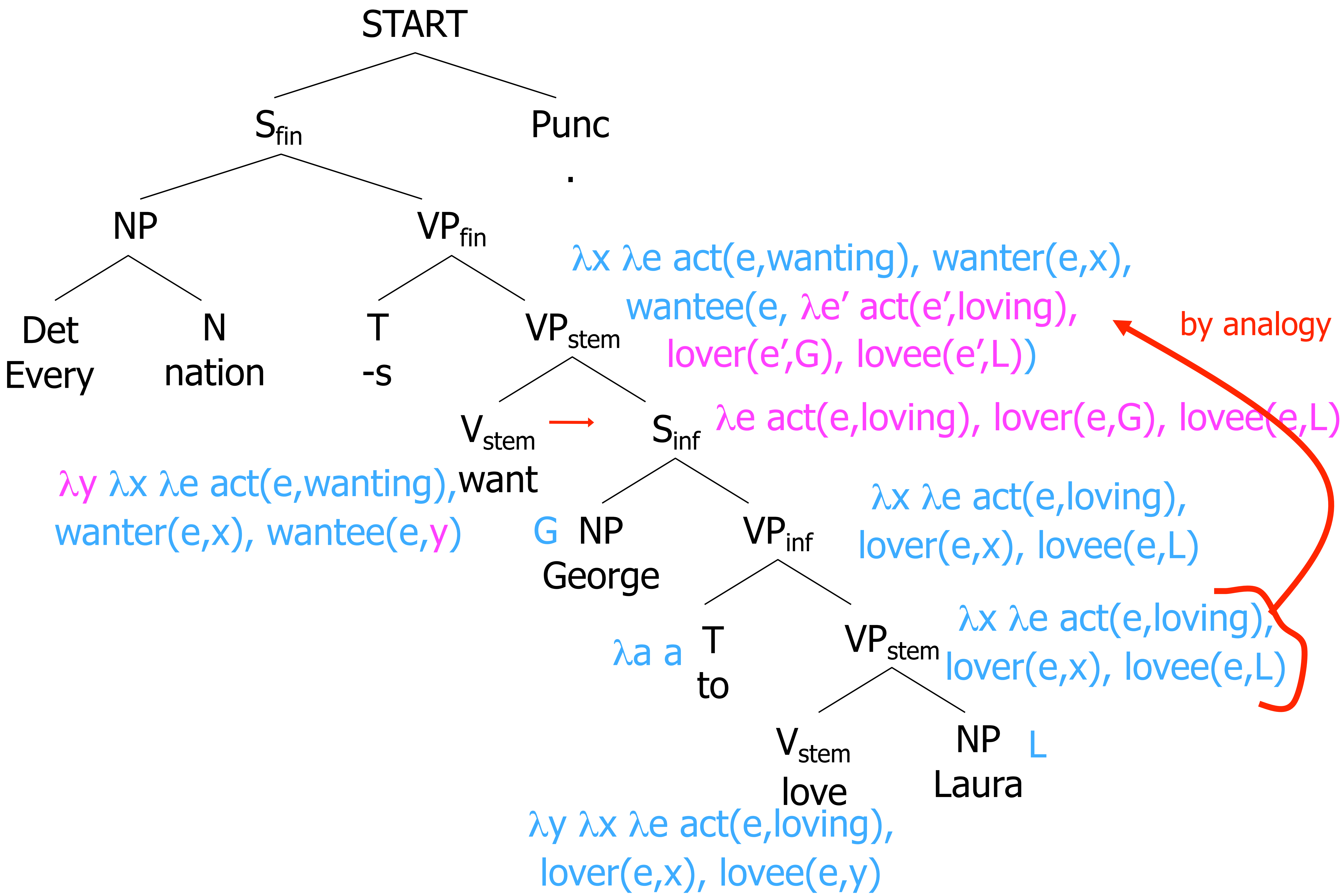

START
S_fin
Punc
.
NP
VP_fin
Det
Every
N
nation
T
-s
VP_stem
V_stem
want
S_inf
λe act(e,loving), lover(e,G), lovee(e,L)
G NP
George
VP_inf
λx λe act(e,loving), lover(e,x), lovee(e,L)
λa a T
to
VP_stem
λx λe act(e,loving), lover(e,x), lovee(e,L)
V_stem
love
NP L
Laura
λy λx λe act(e,loving), lover(e,x),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

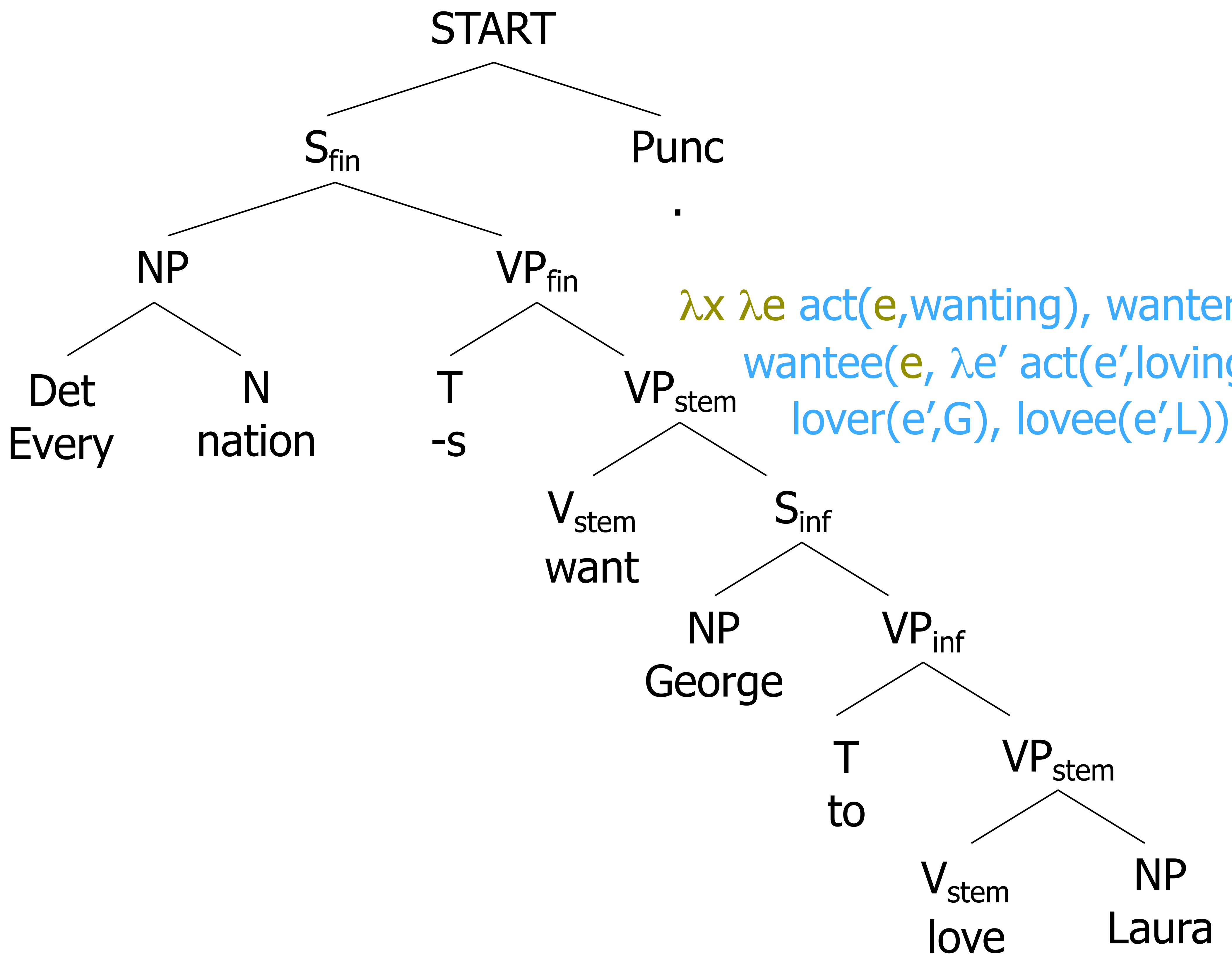

START
S_{fin}
Punc
.
NP
VP_{fin}
Det
Every
N
nation
T
-s
VP_{stem}
V_{stem}
want
S_{inf}
λe act(e,loving), lover(e,G), lovee(e,L)
G
NP
George
VP_{inf}
λx λe act(e,loving), lover(e,x), lovee(e,L)
λa a
T
to
VP_{stem}
λx λe act(e,loving), lover(e,x), lovee(e,L)
V_{stem}
love
NP
Laura
L
λy λx λe act(e,loving),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


START
Sfin
Punc
.
NP
VPfin
Det
Every
N
nation
T
-s
VPstem
Vstem
want
Sinf
G NP
George
VPinf
λa a T
to
VPstem
Vstem
love
NP L
Laura
λx λe act(e,wanting), wanter(e,x), wantee(e, λe′ act(e′,loving), lover(e′,G), lovee(e′,L))
by analogy
λe act(e,loving), lover(e,G), lovee(e,L)
λx λe act(e,loving), lover(e,x), lovee(e,L)
λx λe act(e,loving), lover(e,x), lovee(e,L)
λy λx λe act(e,loving), lover(e,x), lovee(e,y)

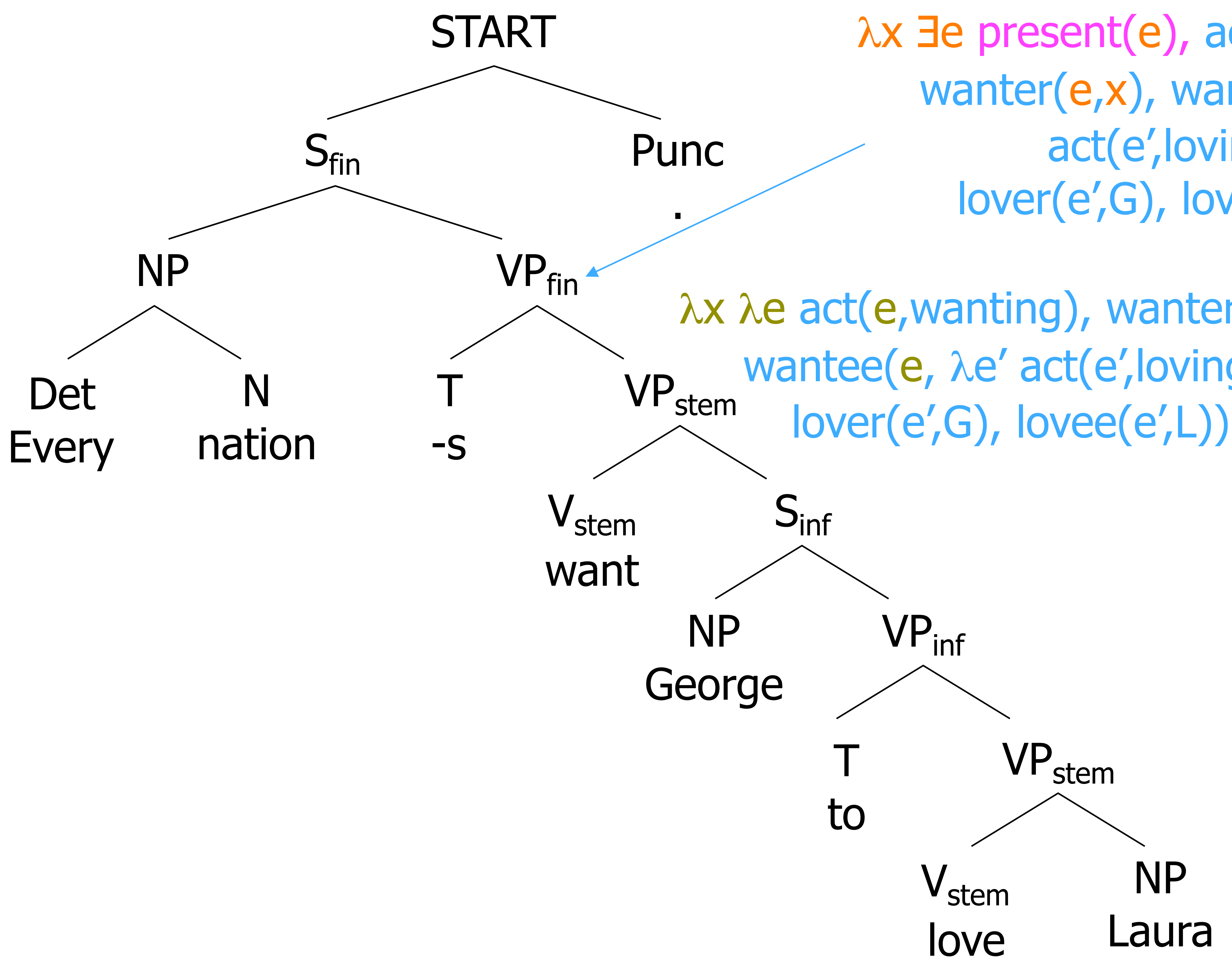


START
S fin
Punc
.
NP
VP fin
Det
Every
N
nation
T
-s
VP stem
λx λe act(e,wanting), wanter(e,x),
wantee(e, λe′ act(e′,loving),
lover(e′,G), lovee(e′,L))
by analogy
V stem
want
S inf
λe act(e,loving), lover(e,G), lovee(e,L)
G
NP
George
VP inf
λx λe act(e,loving),
lover(e,x), lovee(e,L)
λa a
T
to
VP stem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

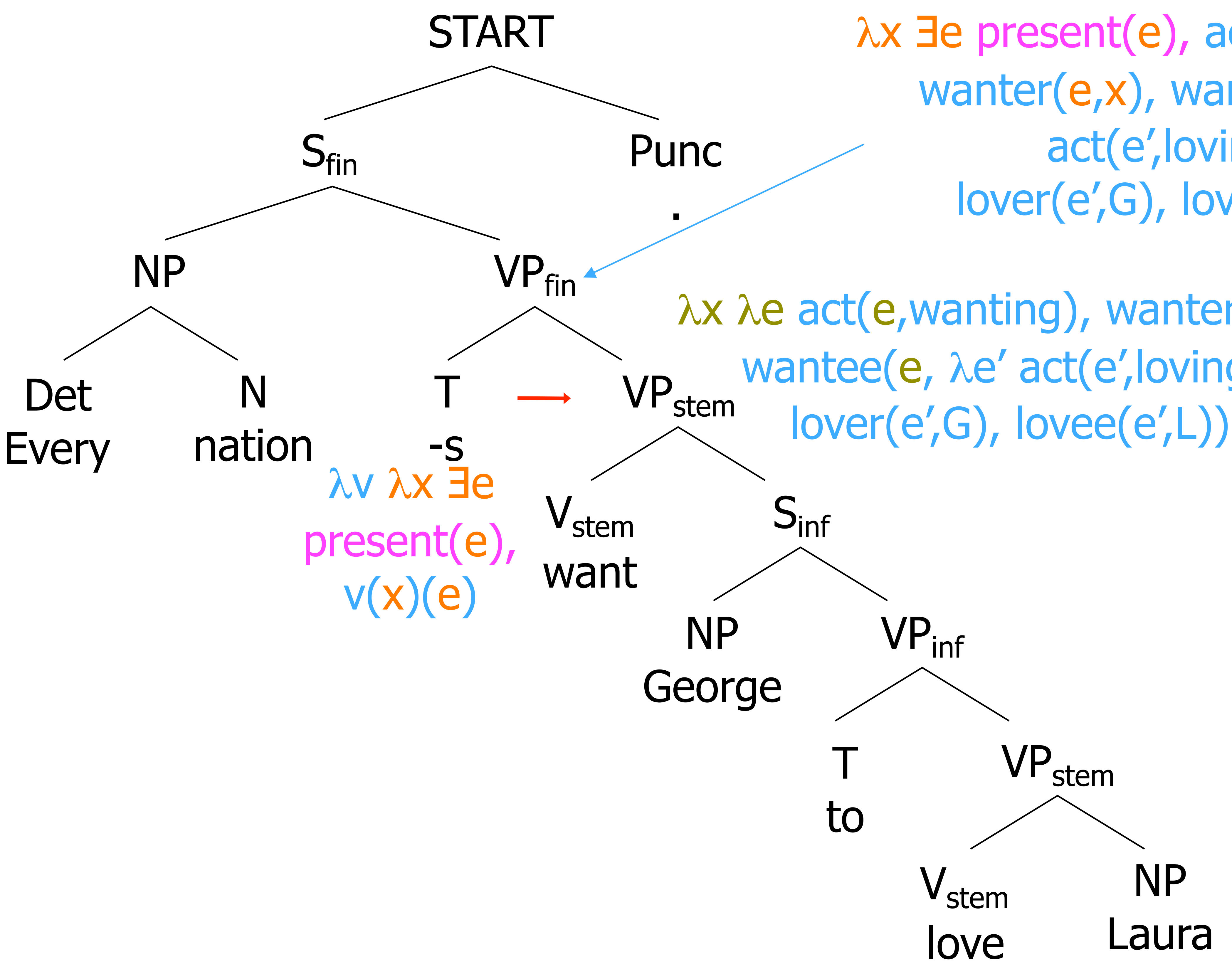


λx λe act(e,wanting), wanter(e,x),
wantee(e, λe′ act(e′,loving),
lover(e′,G), lovee(e′,L))



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

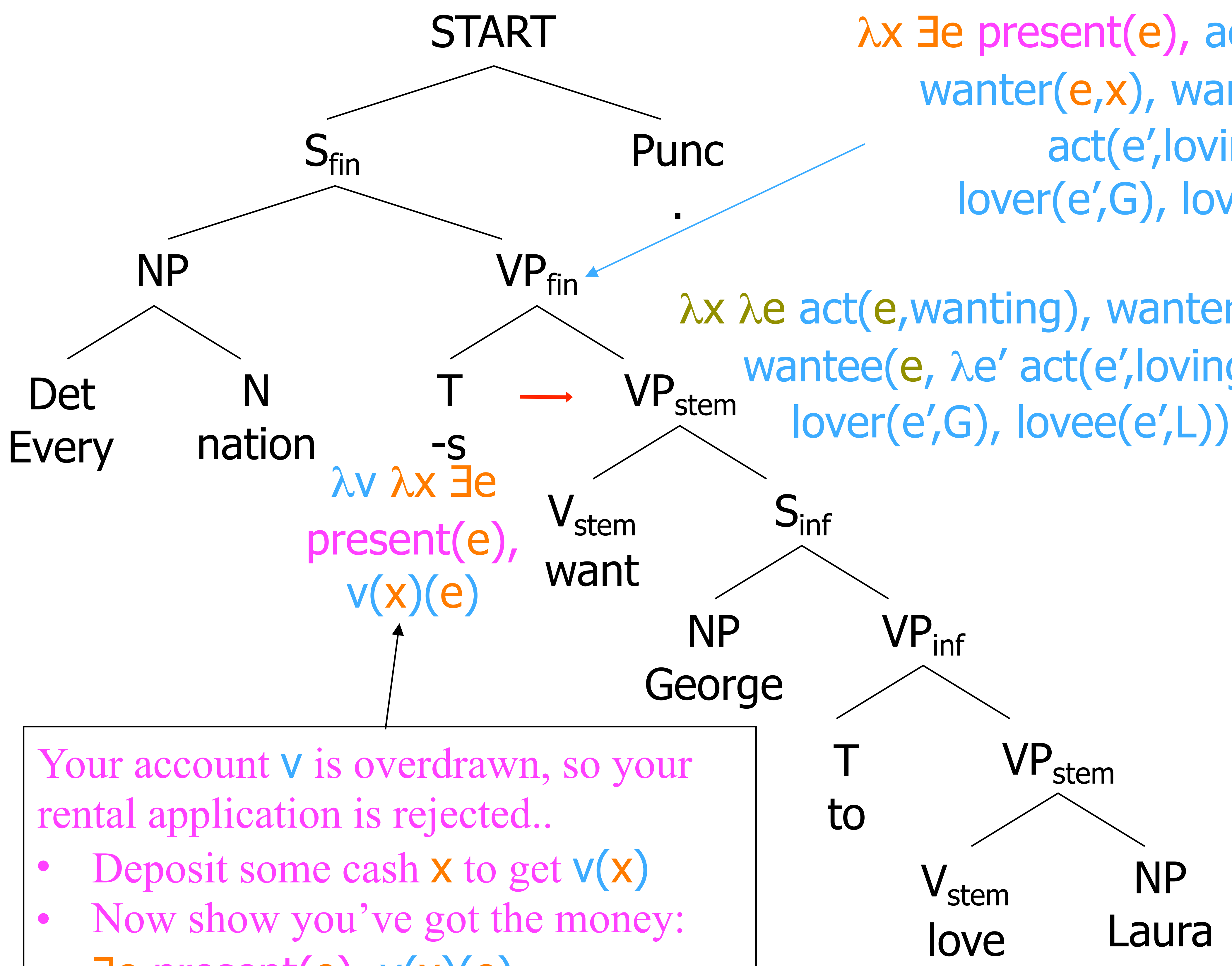
λx λe act(e,wanting), wanter(e,x), wantee(e, λe′ act(e′,loving), lover(e′,G), lovee(e′,L))



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

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

$\lambda v\ \lambda x\ \exists e\ present(e), v(x)(e)$



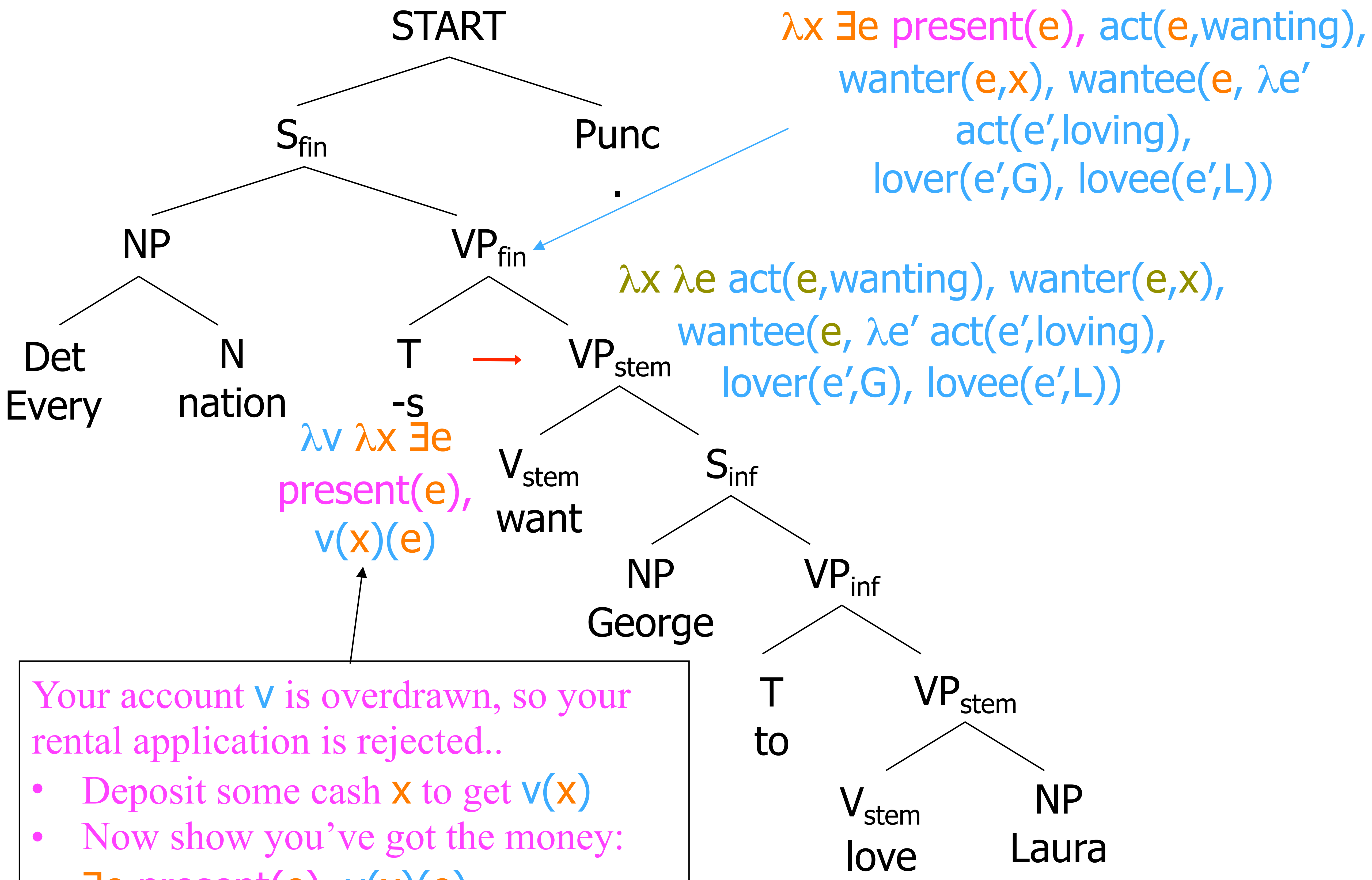
$\lambda x\ \exists e$ present(e), act(e,wanting), wanter(e,x), wantee(e, $\lambda e'$ act(e',loving), lover(e',G), lovee(e',L))

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

$\lambda v\ \lambda x\ \exists e$ 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: $\exists e$ present(e), v(x)(e)
- Now you can withdraw x again: $\lambda x\ \exists e$ present(e), v(x)(e)



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

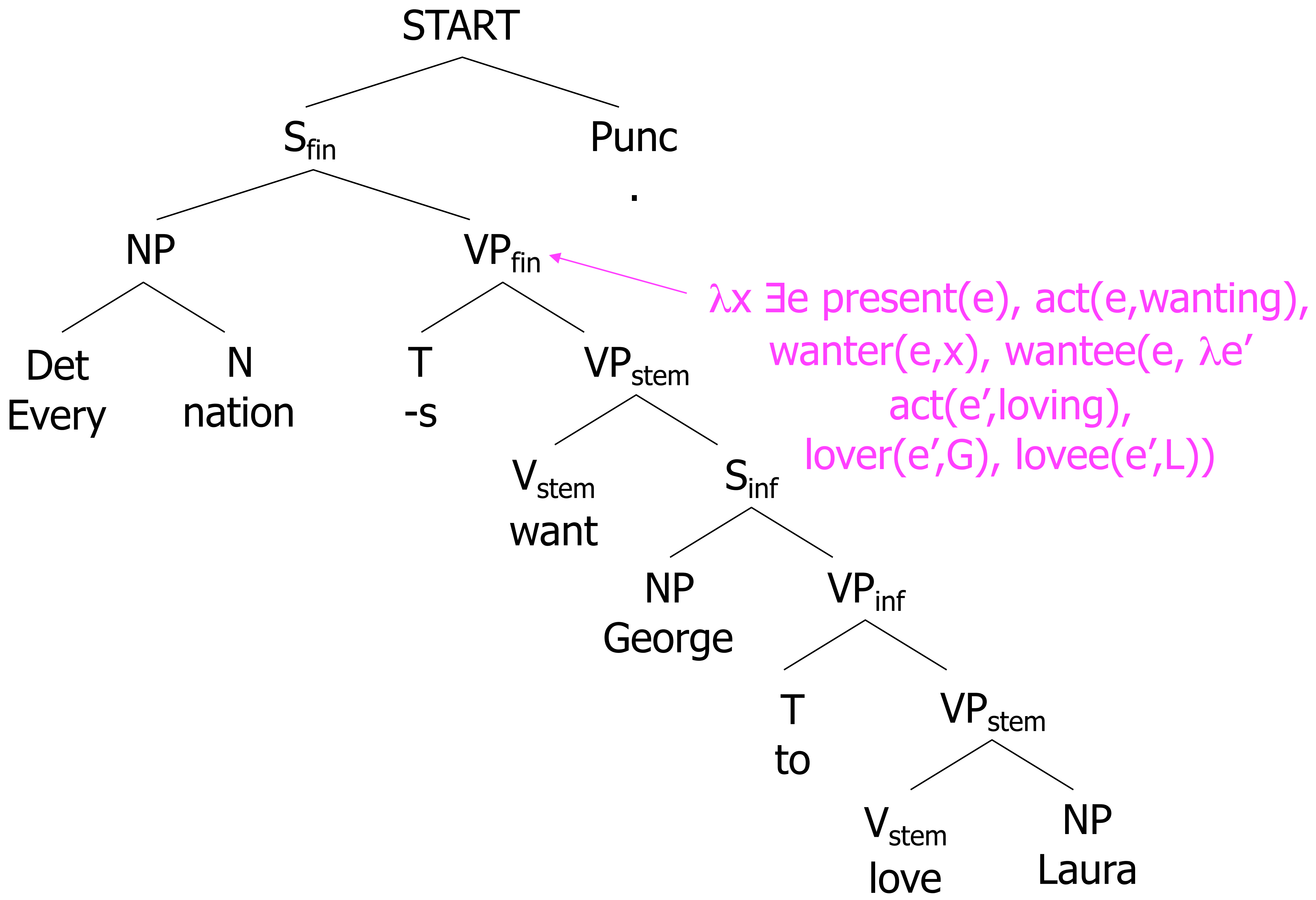
$\lambda x\ \lambda e\ \mathrm{act}(e,\mathrm{wanting}), \mathrm{wanter}(e,x), \mathrm{wantee}(e, \lambda e'\ \mathrm{act}(e',\mathrm{loving}), \mathrm{lover}(e',G), \mathrm{lovee}(e',L))$

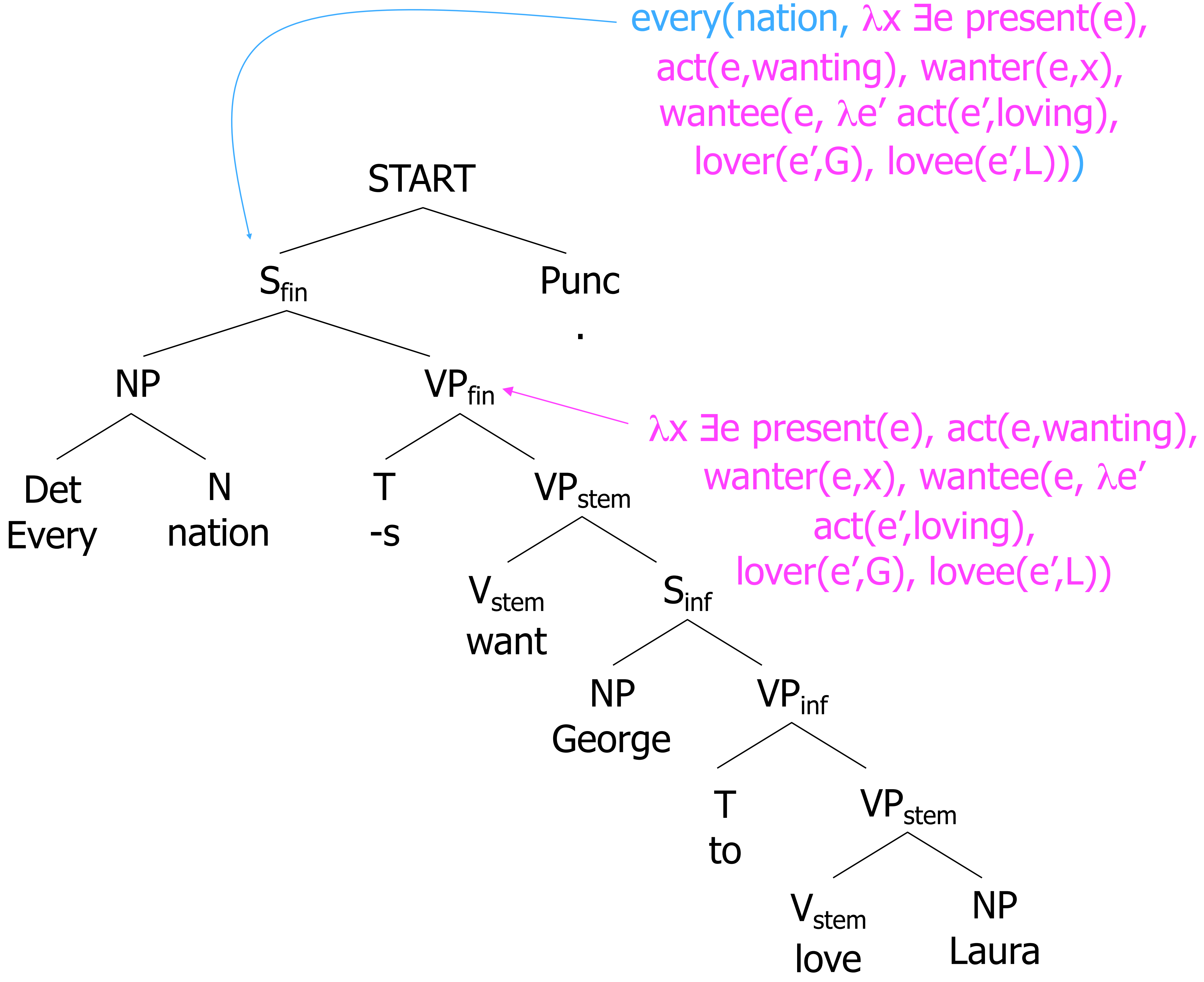
$\lambda v\ \lambda x\ \exists e\ \mathrm{present}(e), v(x)(e)$

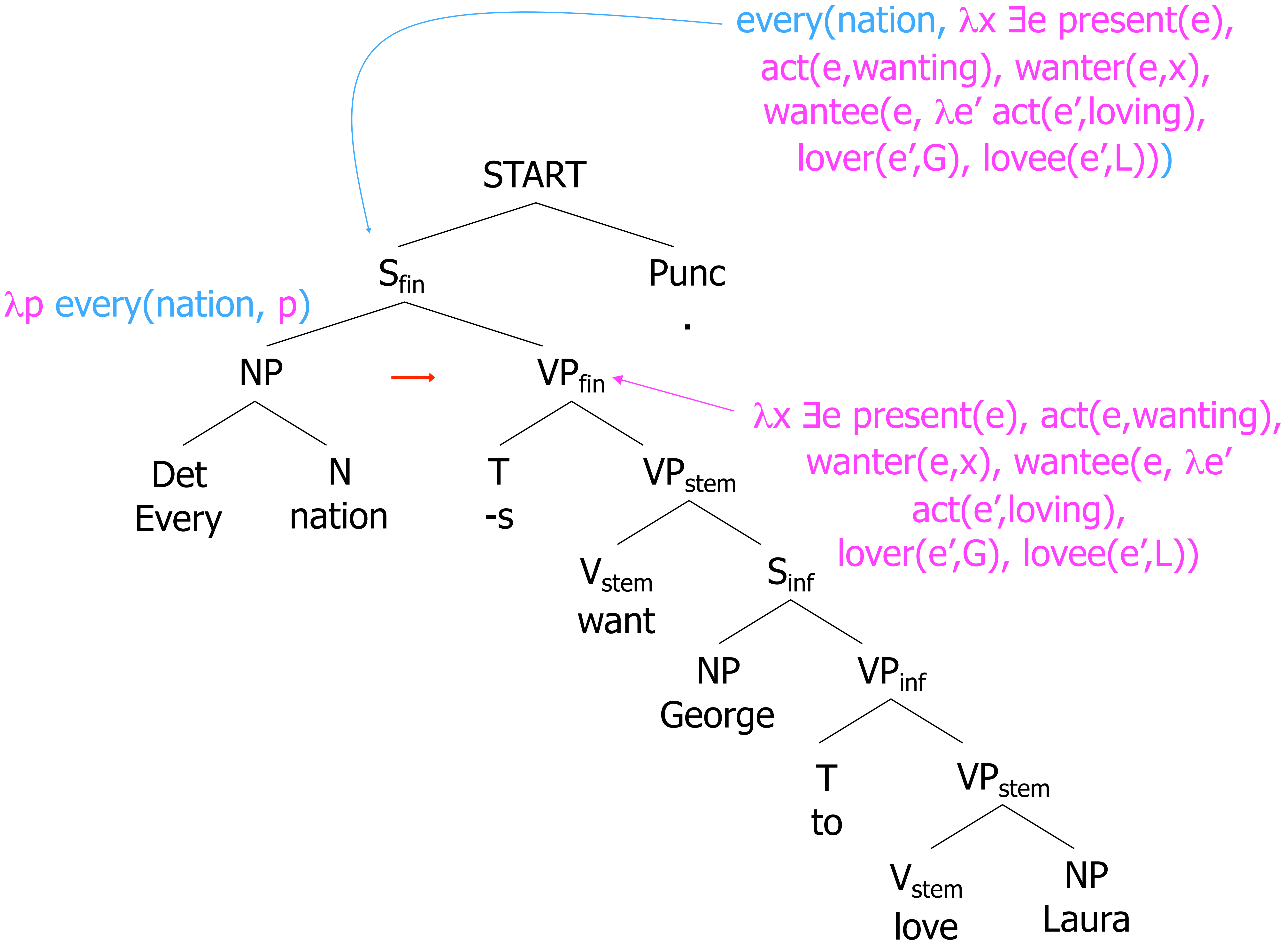
Your account v is overdrawn, so your rental application is rejected..

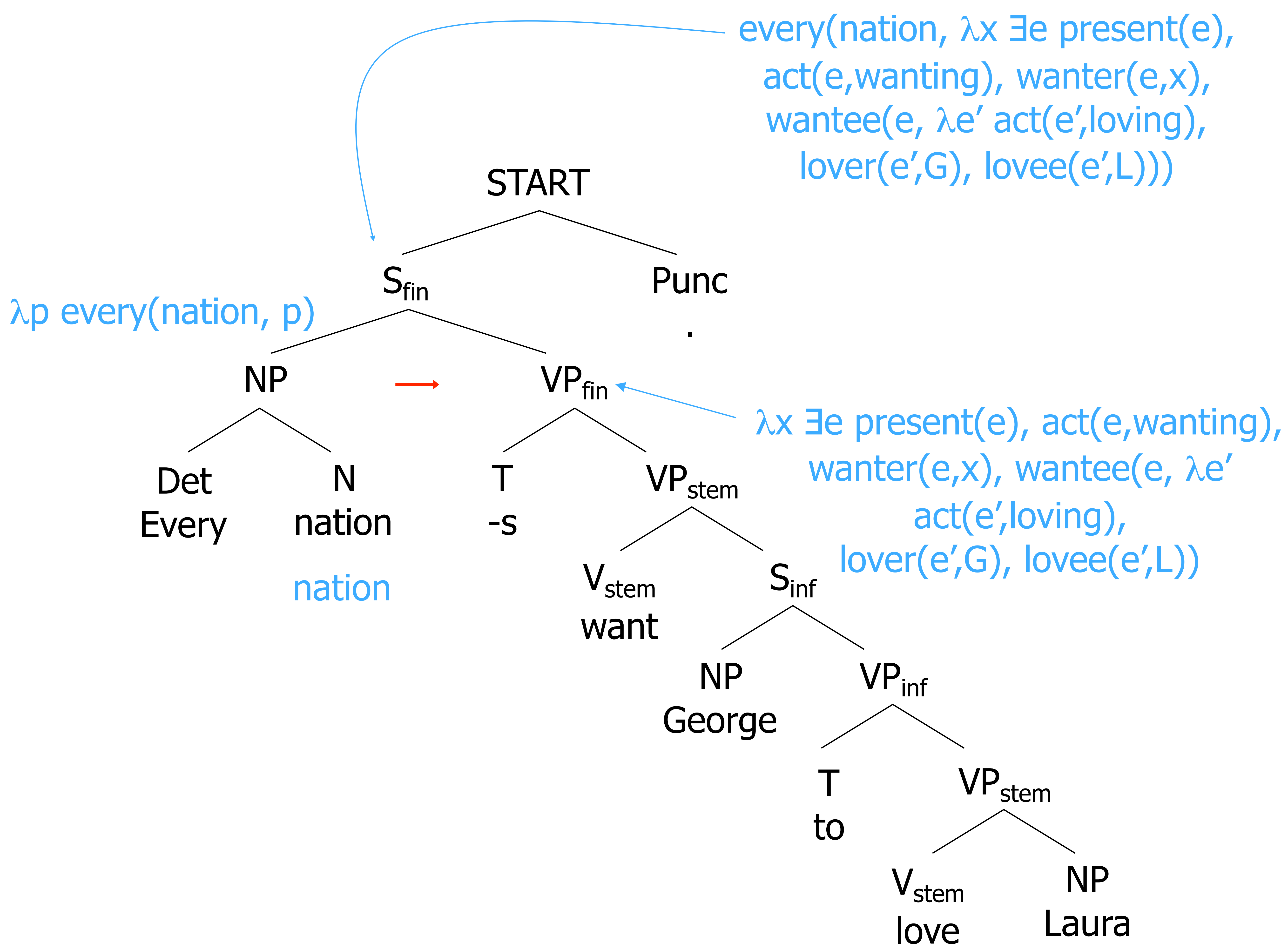
- Deposit some cash x to get v(x)
- Now show you've got the money:
  $\exists e\ \mathrm{present}(e), v(x)(e)$
- Now you can withdraw x again:
  $\lambda x\ \exists e\ \mathrm{present}(e), v(x)(e)$

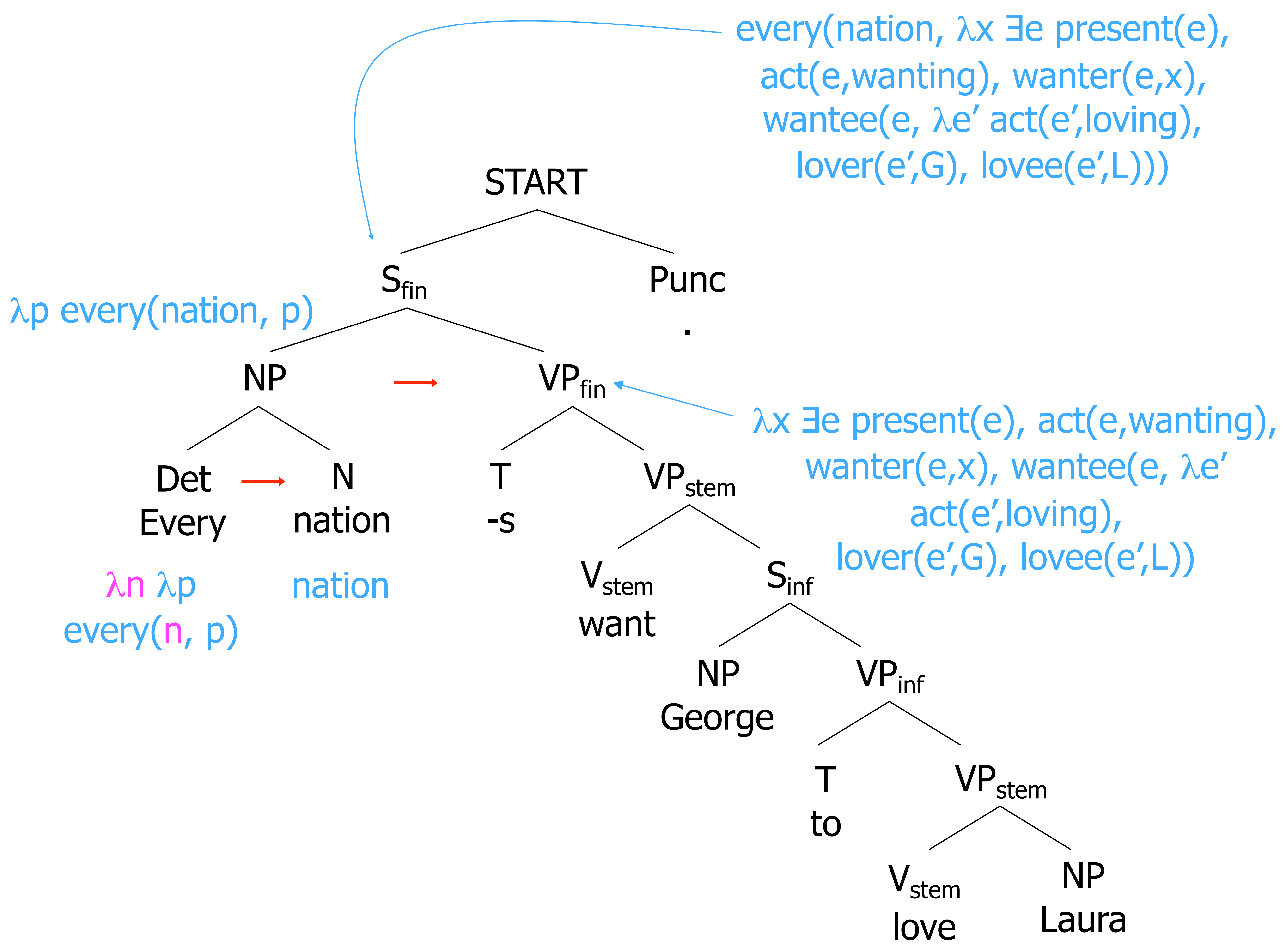
Better analogy: How would you modify the <u>second</u> object on a stack (λx,λe,act…)?

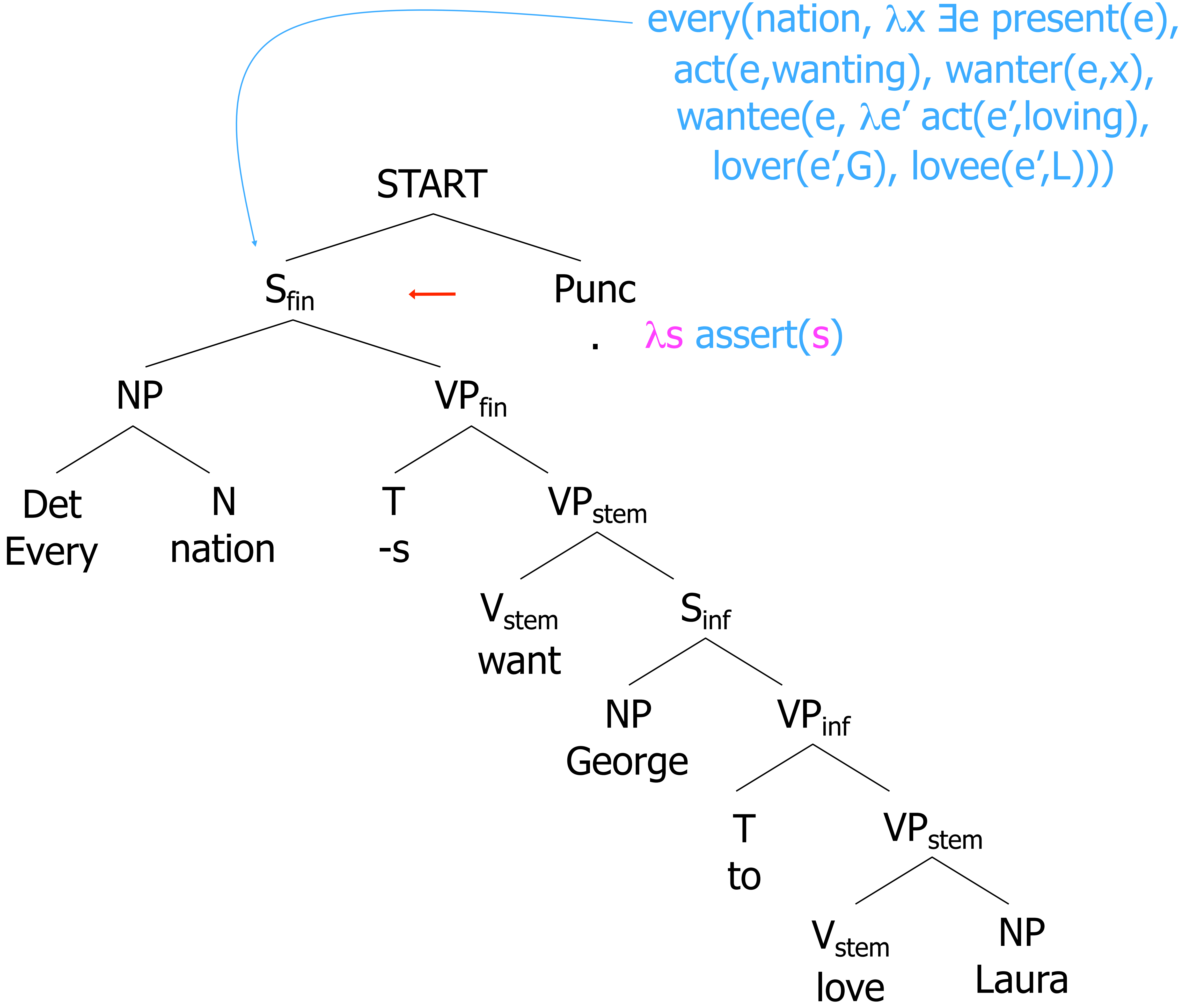

START
S_fin
Punc
.
NP
VP_fin
Det
Every
N
nation
T
-s
VP_stem
V_stem
want
S_inf
NP
George
VP_inf
T
to
VP_stem
V_stem
love
NP
Laura
λx ∃e present(e), act(e,wanting), wanter(e,x), wantee(e, λe′ act(e′,loving), lover(e′,G), lovee(e′,L))


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
.
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

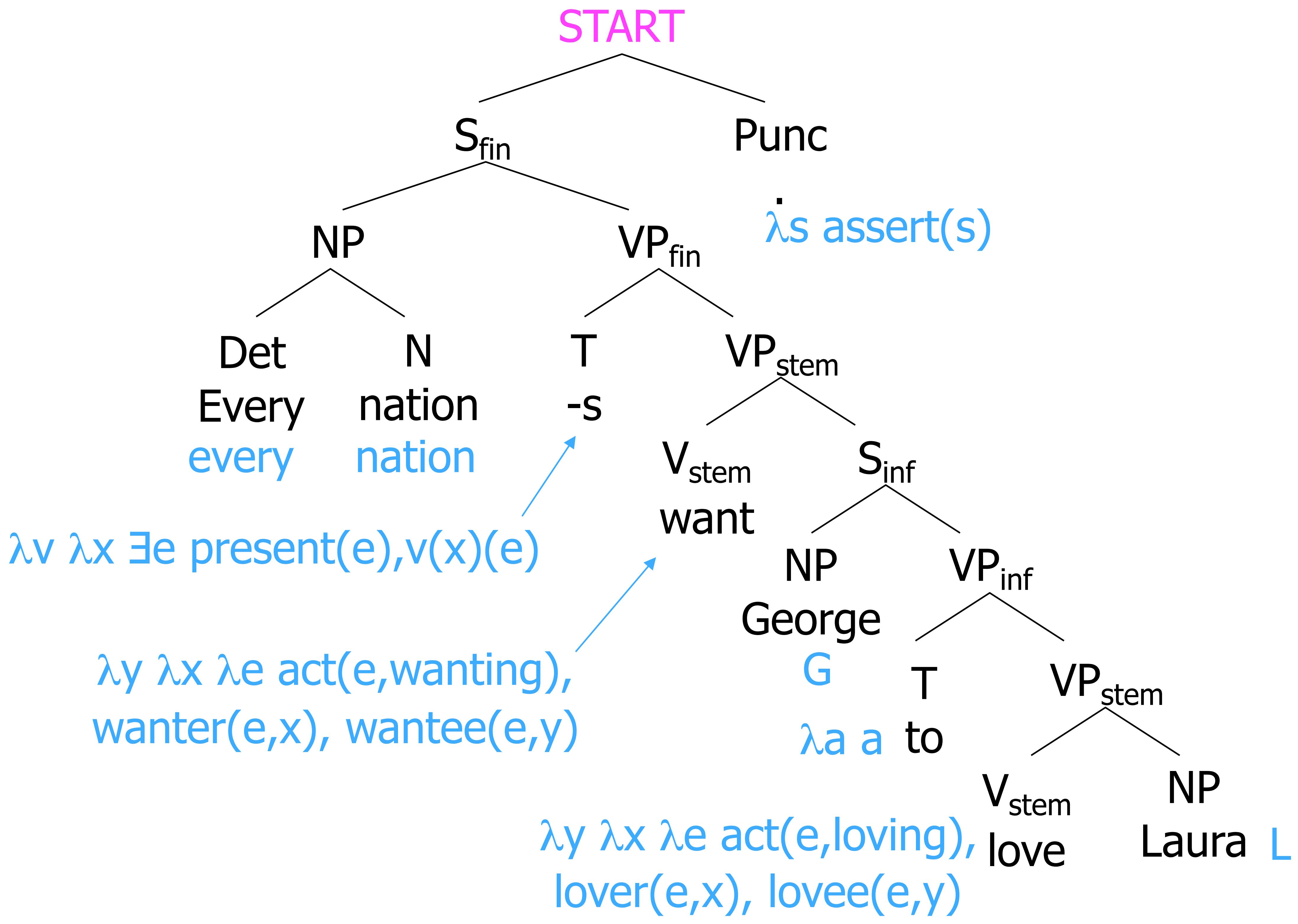

every(nation, λx ∃e present(e), act(e,wanting), wanter(e,x), wantee(e, λe′ act(e′,loving), lover(e′,G), lovee(e′,L)))
START
S fin
Punc
.
λp every(nation, p)
NP
VP fin
λx ∃e present(e), act(e,wanting), wanter(e,x), wantee(e, λe′ act(e′,loving), lover(e′,G), lovee(e′,L))
Det
Every
N
nation
T
-s
VP stem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

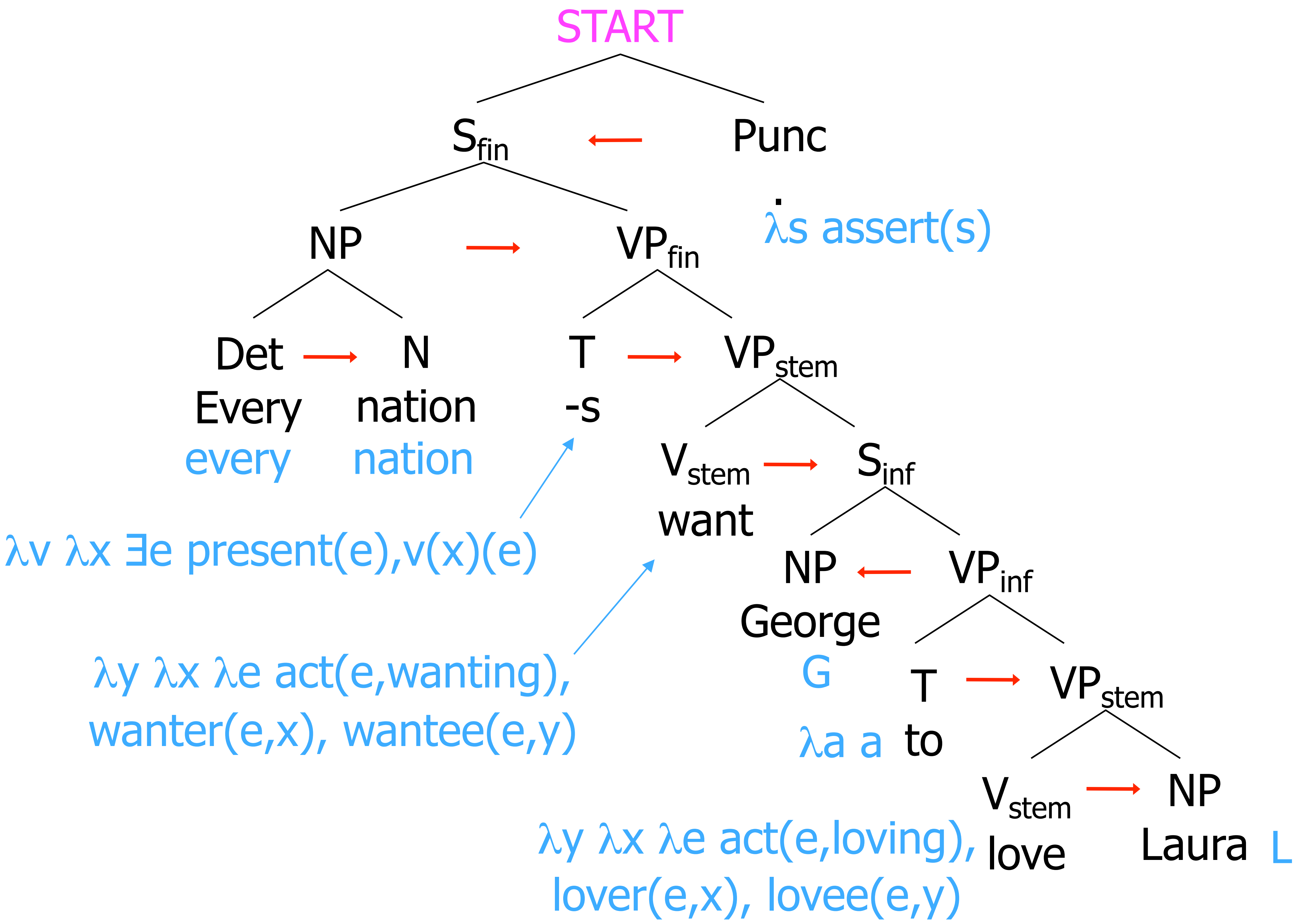

every(nation, λx ∃e present(e), act(e,wanting), wanter(e,x), wantee(e, λe′ act(e′,loving), lover(e′,G), lovee(e′,L)))
START
S_fin
Punc
.
λp every(nation, p)
NP
VP_fin
λx ∃e present(e), act(e,wanting), wanter(e,x), wantee(e, λe′ act(e′,loving), lover(e′,G), lovee(e′,L))
Det
Every
N
nation
T
-s
VP_stem
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

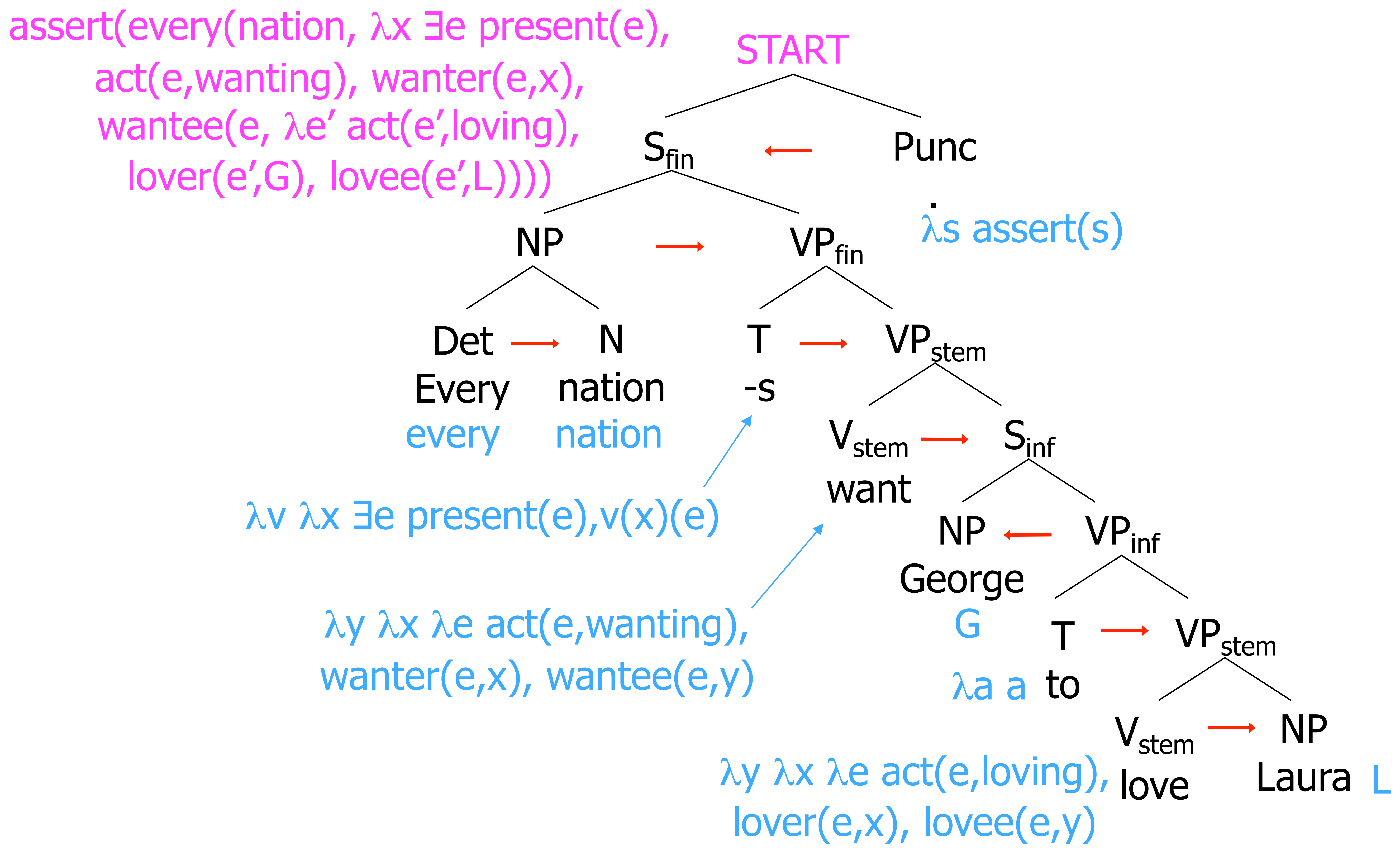
In Summary: From the Words



In Summary: From the Words



In Summary: From the Words



assert(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
N
Every
nation
every
nation
T
-s
VP_{stem}
λv λx ∃e present(e),v(x)(e)
V_{stem}
want
S_{inf}
λy λx λe act(e,wanting), wanter(e,x), wantee(e,y)
NP
George
G
VP_{inf}
T
to
λa a
VP_{stem}
V_{stem}
love
λy λx λe act(e,loving), lover(e,x), lovee(e,y)
NP
Laura
L

Other Fun Semantic Stuff: A Few Much-Studied Miscellany

- Temporal logic
  - `Gilly had swallowed eight goldfish`
    `before Milly reached the bowl`
  - `Billy said Jilly was pregnant`
  - `Billy said, "Jilly is pregnant."`
- 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.`
  - `If you have a dime, put it in the meter.`
  - `The woman who every Englishman loves is his mother.`
  - `I love my mother and so does Billy.`

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: 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, but cf. TAG)
    - Lexical semantics
        - What are the ground terms?
        - And can lexical semantics help us learn better compositional semantics?
        - Excursus on Machine Translation

Semantic Role Labeling

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

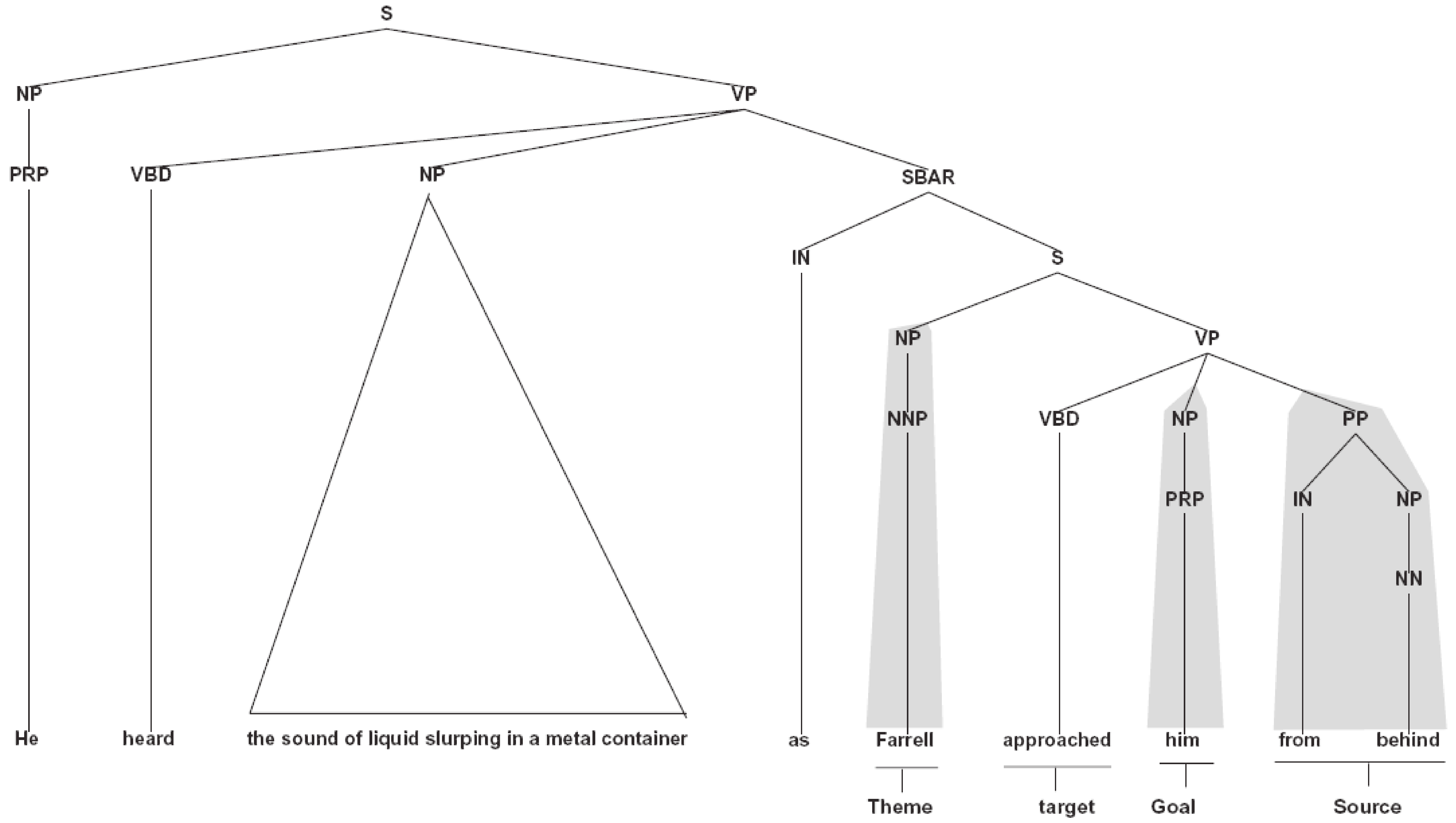
  [$_{\textit{Judge}}$ She] **blames** [$_{\textit{Evaluee}}$ the Government] [$_{\textit{Reason}}$ for failing to do enough to help] .

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

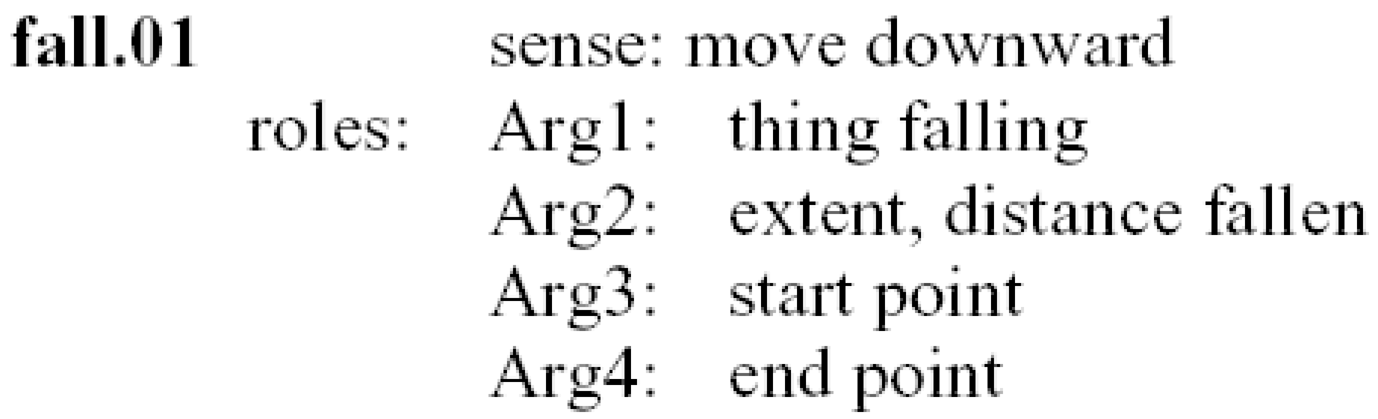
  The letter quotes Black as saying that [$_{\textit{Judge}}$ white and Navajo ranchers] misrepresent their livestock losses and **blame** [$_{\textit{Reason}}$ everything] [$_{\textit{Evaluee}}$ on coyotes] .

- 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.

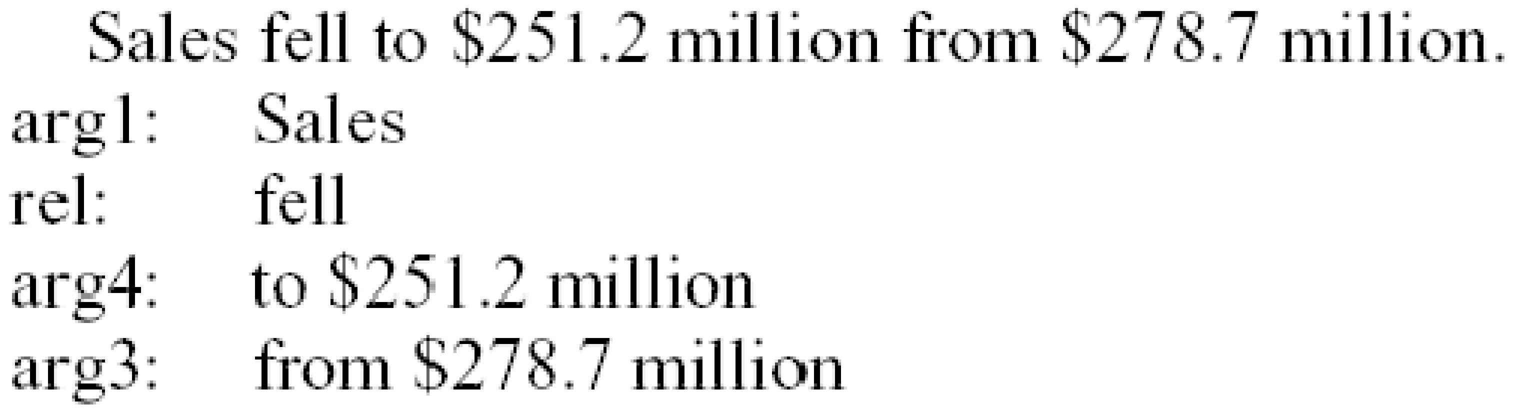
SRL Example


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



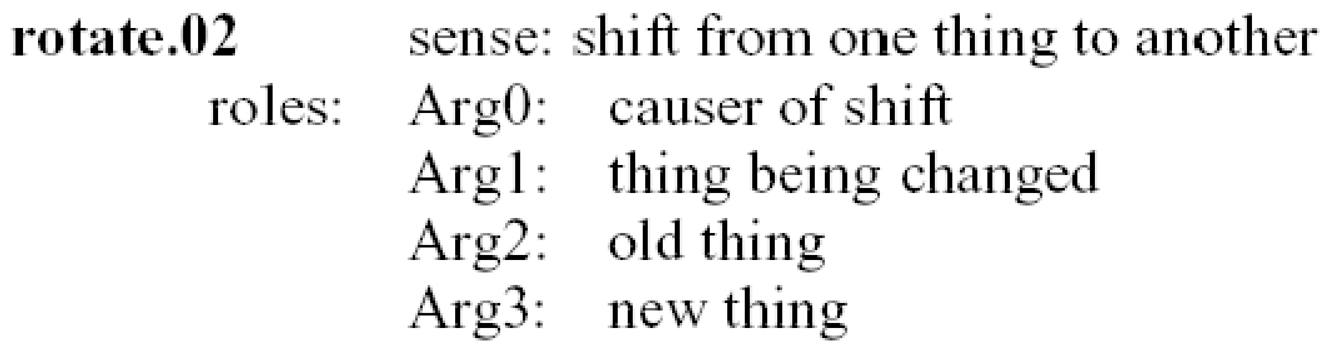
fall.01		sense: move downward	
	roles:	Arg1:	thing falling
		Arg2:	extent, distance fallen
		Arg3:	start point
		Arg4:	end point



Sales fell to $251.2 million from $278.7 million.

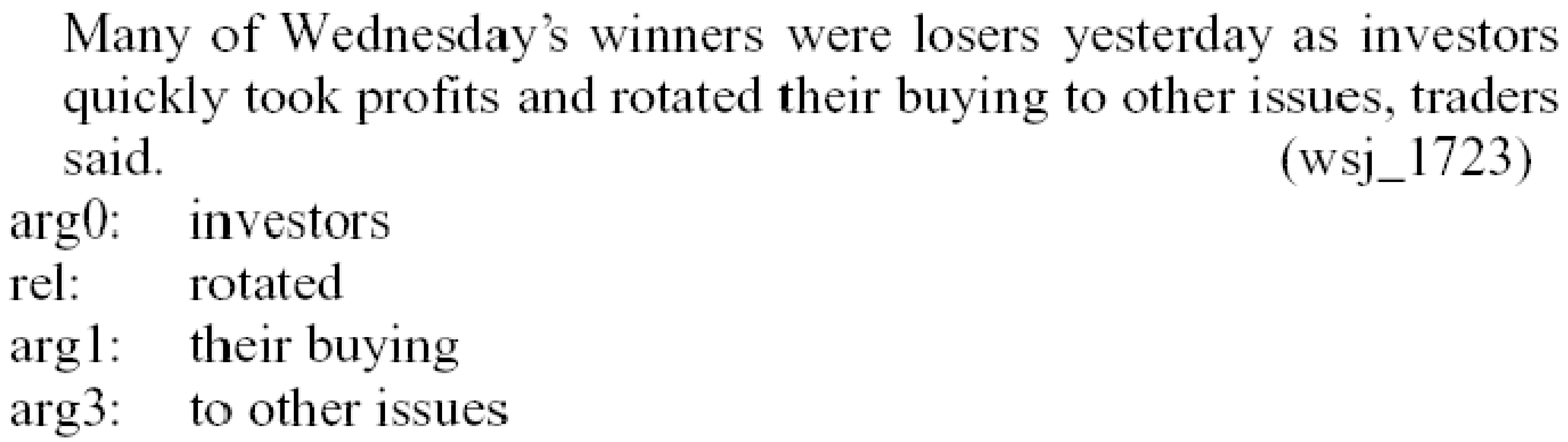
arg1:	Sales
rel:	fell
arg4:	to $251.2 million
arg3:	from $278.7 million

PropBank Example



rotate.02 sense: shift from one thing to another

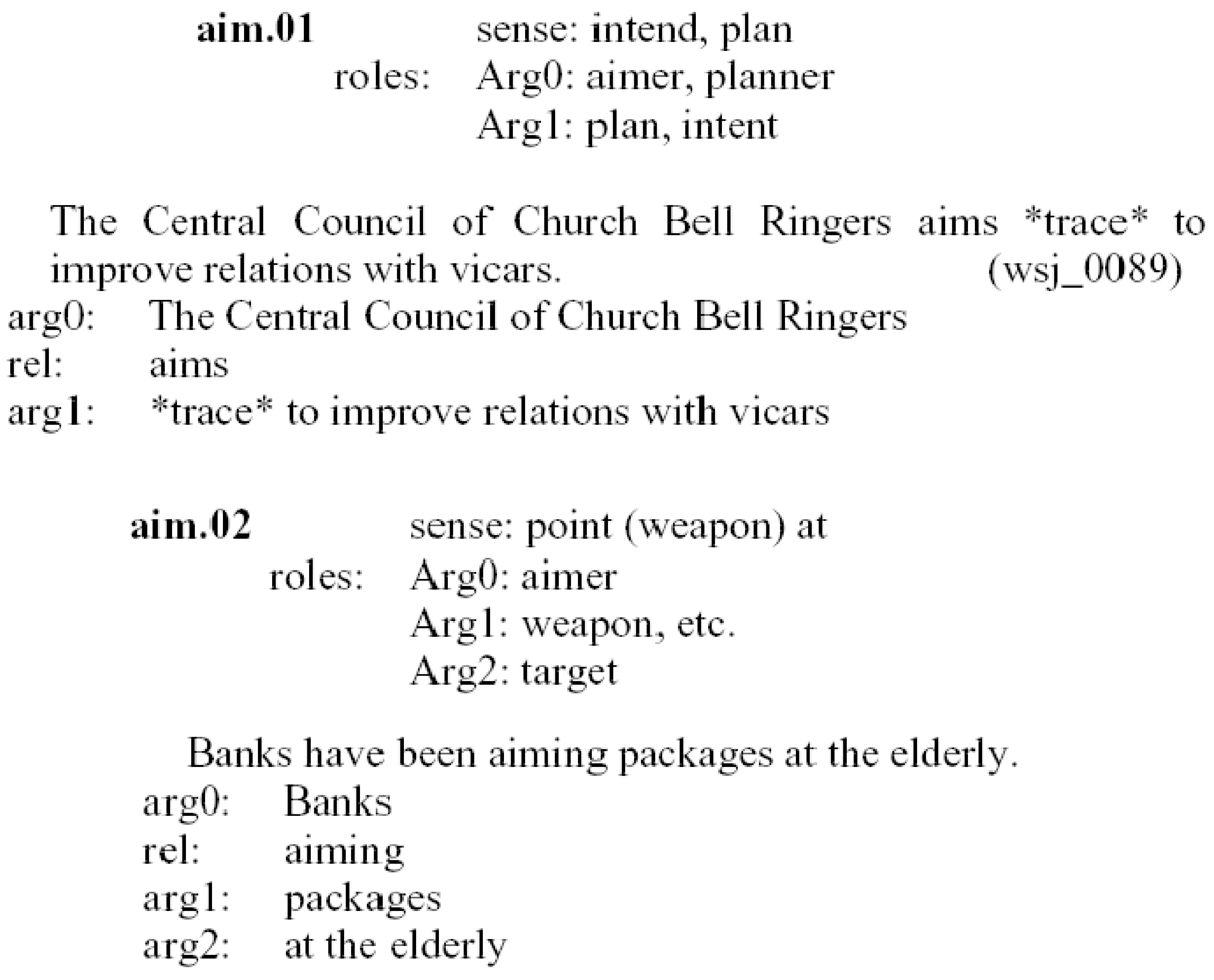
roles:	Arg0:	causer of shift
	Arg1:	thing being changed
	Arg2:	old thing
	Arg3:	new thing



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

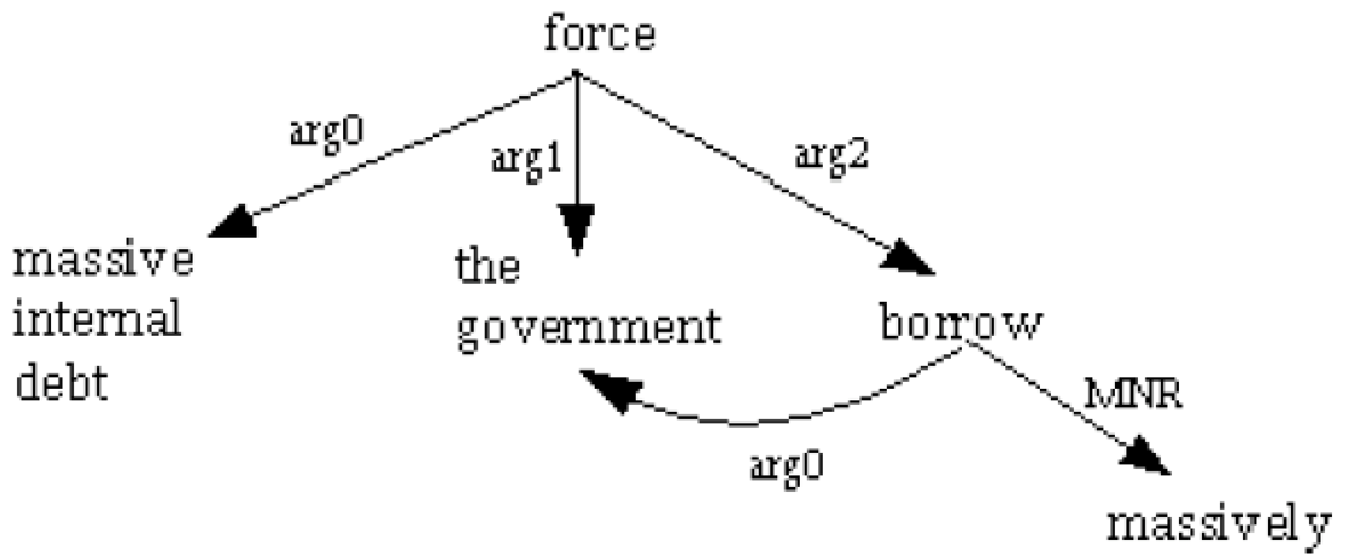
aim.02		sense: point (weapon) at
	roles:	Arg0: aimer
		Arg1: weapon, etc.
		Arg2: target

Banks have been aiming packages at the elderly.

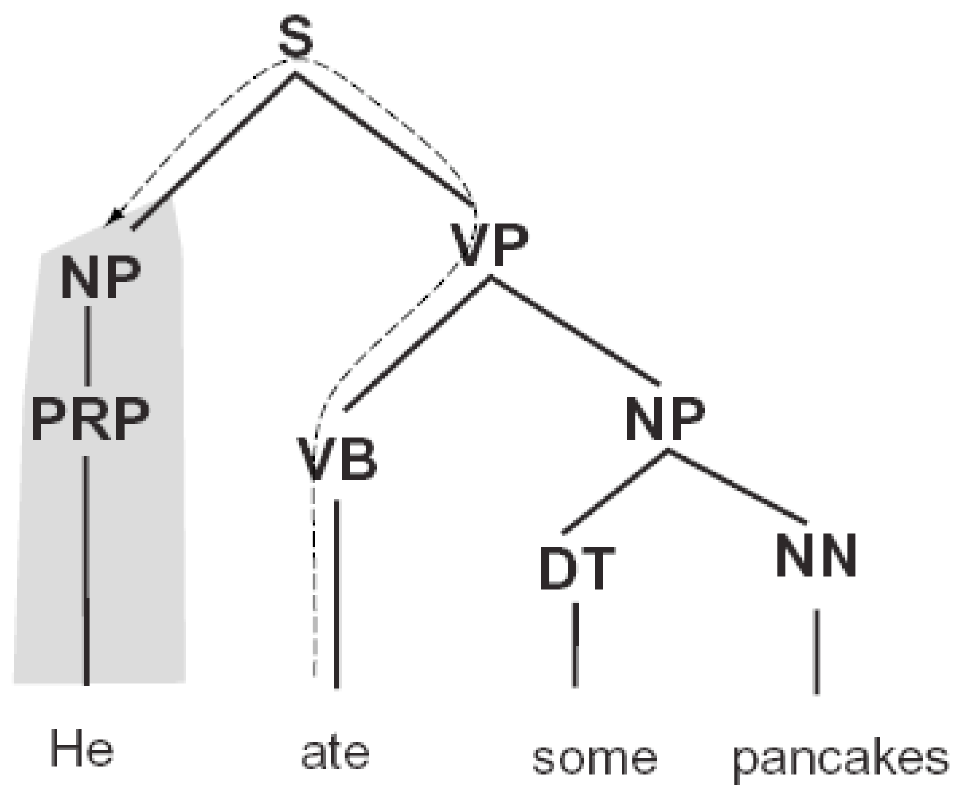
arg0:	Banks
rel:	aiming
arg1:	packages
arg2:	at the elderly

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) )...
```



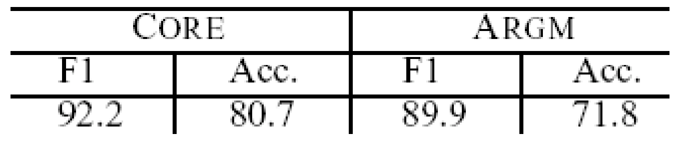
Path Features



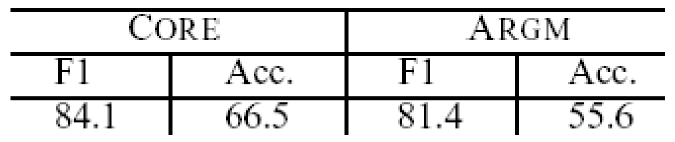
Path	*Description*
VB↑VP↓PP	PP argument/adjunct
VB↑VP↑S↓NP	subject
VB↑VP↓NP	object
VB↑VP↑VP↑S↓NP	subject (embedded VP)
VB↑VP↓ADVP	adverbial adjunct
NN↑NP↑NP↓PP	prepositional complement of noun

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

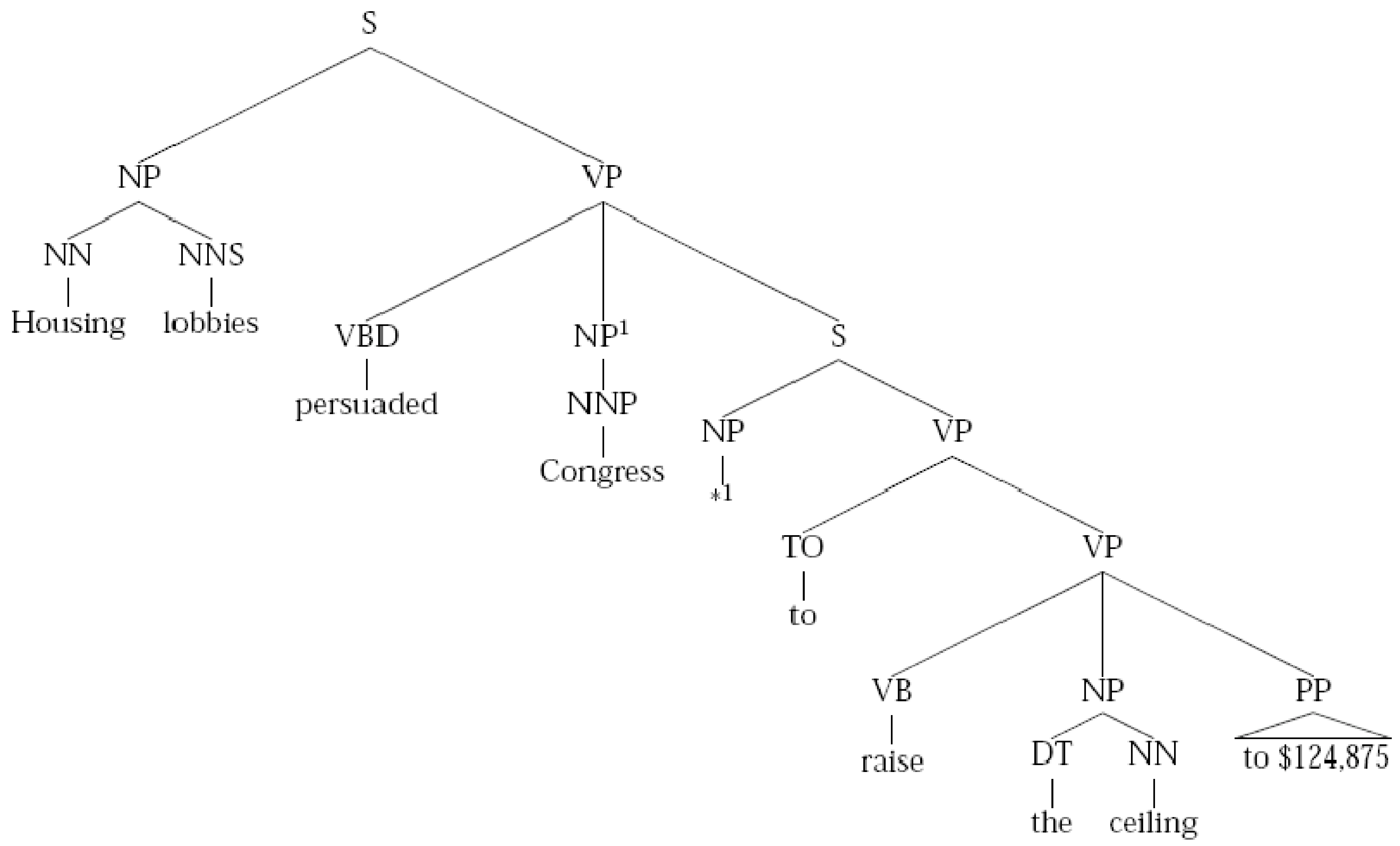


CORE		ARGM	
F1	Acc.	F1	Acc.
92.2	80.7	89.9	71.8



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

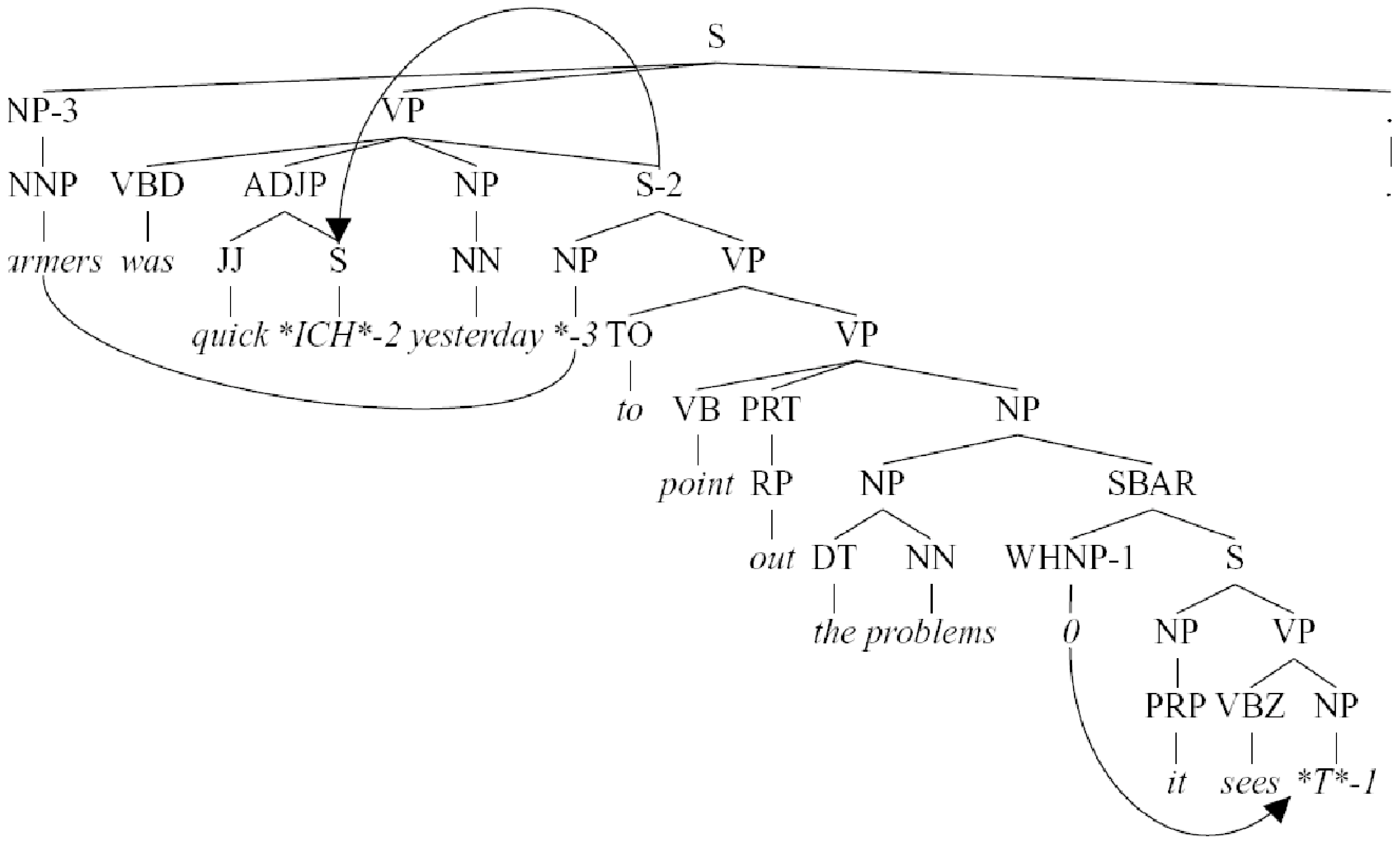
Interaction with Empty Elements



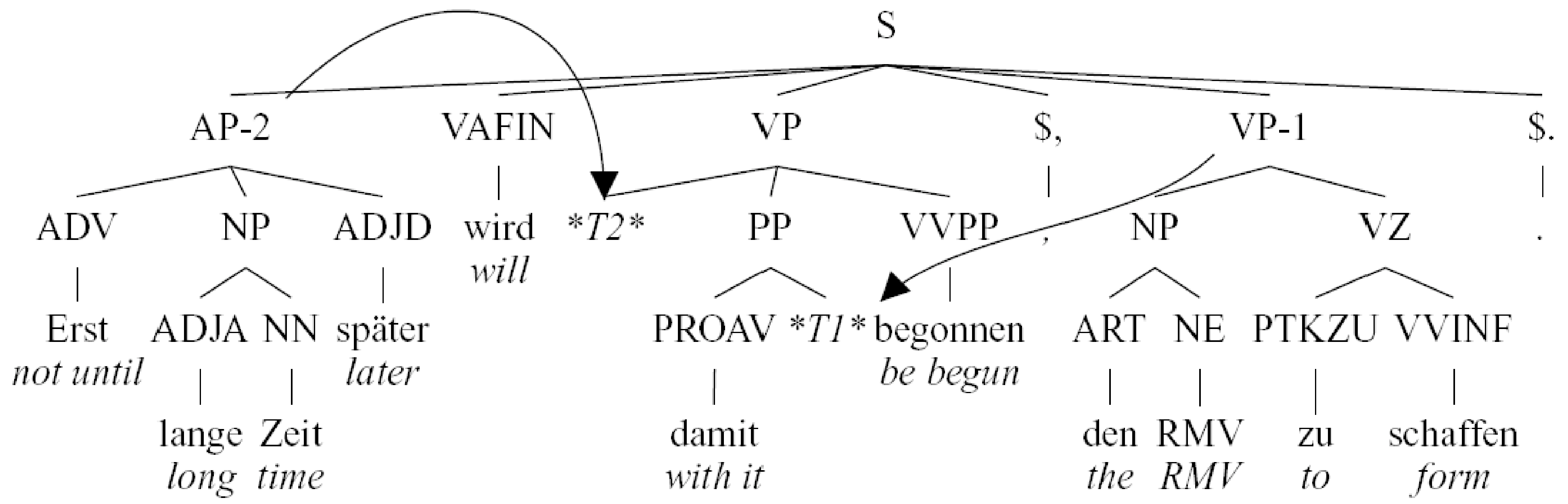
Empty Elements

- 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; prosody 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 subcategorization 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

$$\begin{array}{ccc}
interleukin-10 & inhibits & production \\
\hline
NP & (S\backslash NP)/NP & NP \\
\cline{2-3}
 & \multicolumn{2}{c}{S\backslash NP} \quad > \\
\hline
\multicolumn{3}{c}{S} \quad <
\end{array}$$

$>$	forward application
$<$	backward application

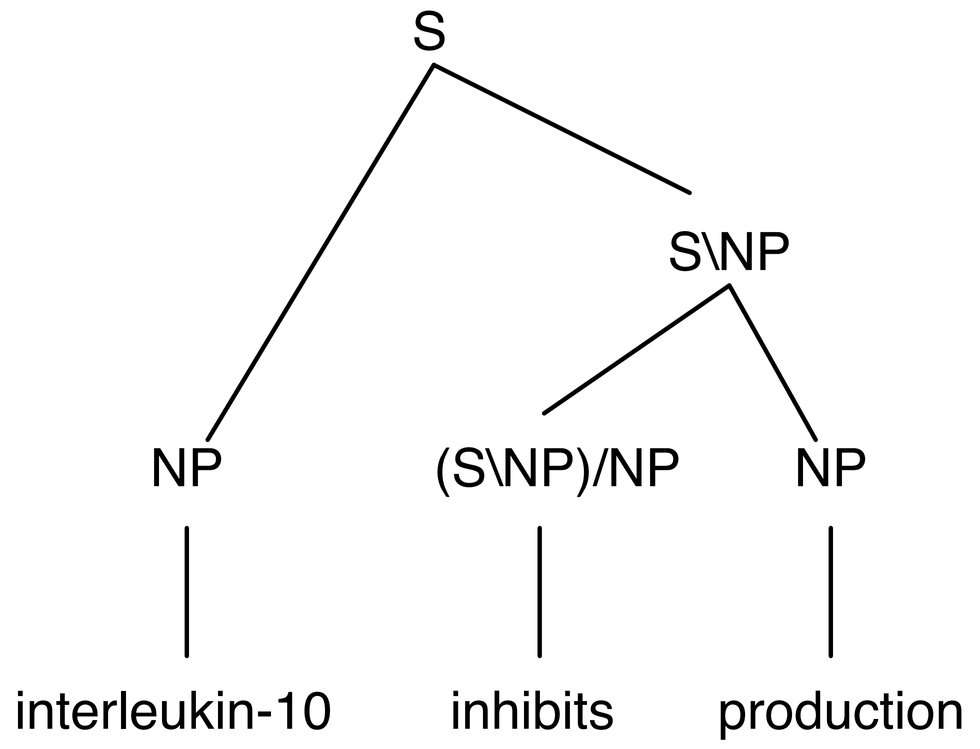
Function Application Schemata

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

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

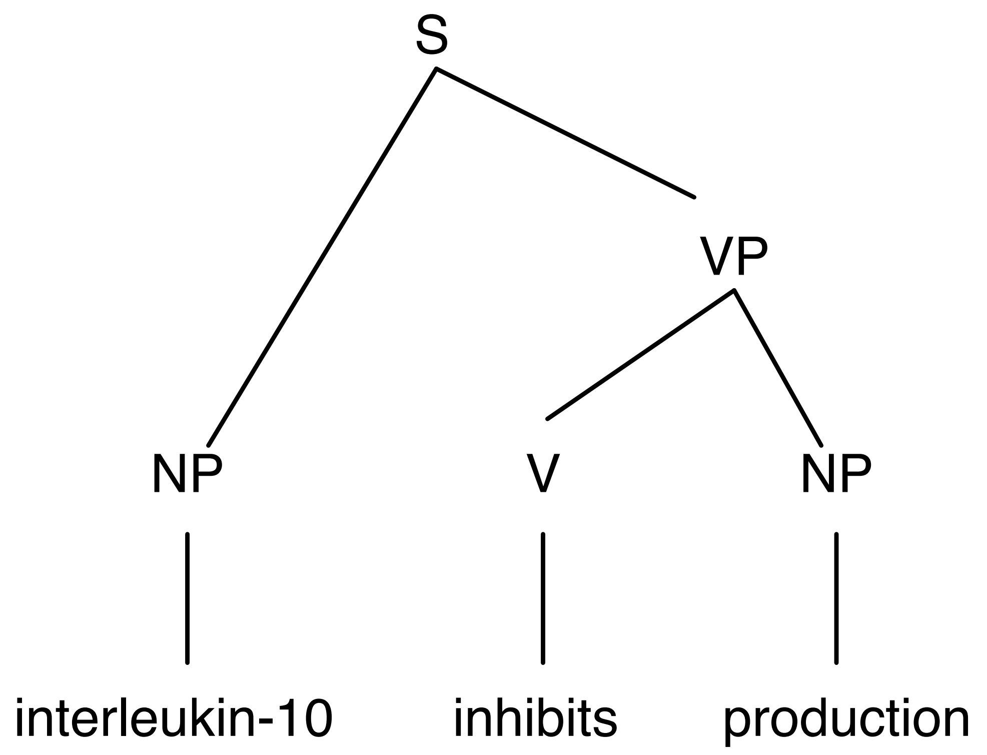
Classical Categorial Grammar

- ‘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 from a Relative Clause

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

Extraction from a Relative Clause

$$\begin{array}{ccccc}
\mathit{The} & \mathit{company} & \mathit{which} & \mathit{Microsoft} & \mathit{bought} \\
\hline NP/N & \overline{N} & \overline{(NP\backslash NP)/(S/NP)} & \overline{NP} & \overline{(S\backslash NP)/NP} \\
 & & & \overline{S/(S\backslash NP)}{>}\mathbf{T} &
\end{array}$$

$>\mathbf{T}$ type-raising

Extraction from a Relative Clause

$$\begin{array}{ccccc}
\textit{The} & \textit{company} & \textit{which} & \textit{Microsoft} & \textit{bought} \\
\hline
NP/N & N & (NP\backslash NP)/(S/NP) & NP & (S\backslash NP)/NP \\
\cline{4-4}
 & & & S/(S\backslash NP) \quad {>}\mathbf{T} & \\
\cline{4-5}
 & & & \multicolumn{2}{c}{S/NP \quad {>}\mathbf{B}}
\end{array}$$

$>$ **T** type-raising
$>$ **B** forward composition

Extraction from a Relative Clause

$$
\begin{array}{cccccl}
\textit{The} & \textit{company} & \textit{which} & \textit{Microsoft} & \textit{bought} & \\
\cline{1-1}\cline{2-2}\cline{3-3}\cline{4-4}\cline{5-5}
NP/N & N & (NP\backslash NP)/(S/NP) & NP & (S\backslash NP)/NP & \\
\cline{4-4}
 & & & S/(S\backslash NP) & & >\mathbf{T} \\
\cline{4-5}
 & & & \multicolumn{2}{c}{S/NP} & >\mathbf{B} \\
\cline{3-5}
 & & \multicolumn{3}{c}{NP\backslash NP} & >
\end{array}
$$

Extraction from a Relative Clause

$$
\begin{array}{ccccc}
\mathit{The} & \mathit{company} & \mathit{which} & \mathit{Microsoft} & \mathit{bought} \\
\hline
NP/N & N & (NP\backslash NP)/(S/NP) & NP & (S\backslash NP)/NP \\
\cline{1-2} \cline{4-4}
\multicolumn{2}{r}{>} & & \multicolumn{1}{r}{>\mathbf{T}} & \\
\multicolumn{2}{c}{\color{blue}{NP}} & & S/(S\backslash NP) & \\
\cline{4-5}
& & & \multicolumn{2}{r}{>\mathbf{B}} \\
& & & \multicolumn{2}{c}{S/NP} \\
\cline{3-5}
& & \multicolumn{3}{r}{>} \\
& & \multicolumn{3}{c}{NP\backslash \color{blue}{NP}} \\
\cline{1-5}
\multicolumn{5}{r}{<} \\
\multicolumn{5}{c}{NP}
\end{array}
$$

Forward Composition & Type Raising

- Forward composition (>**B**)

$$X/Y\ Y/Z \implies X/Z\ (>_{\mathbf{B}})$$

- Type raising (**T**)

$$X \implies T/(T \setminus X)\ (>_{\mathbf{T}})$$
$$X \implies T \setminus (T/X)\ (<_{\mathbf{T}})$$

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

Non-constituents & Coordination

$$\begin{array}{cccccc}
\underline{\mathit{Google}} & \underline{\mathit{sells}} & \underline{\mathit{but}} & \underline{\mathit{Microsoft}} & \underline{\mathit{buys}} & \underline{\mathit{shares}} \\
NP & (S\backslash NP)/NP & conj & NP & (S\backslash NP)/NP & NP \\
\overline{}{}^{>\mathbf{T}} & & & \overline{}{}^{>\mathbf{T}} & & \\
S/(S\backslash NP) & & & S/(S\backslash NP) & &
\end{array}$$

$> \mathbf{T}$ type-raising

Non-constituents & Coordination

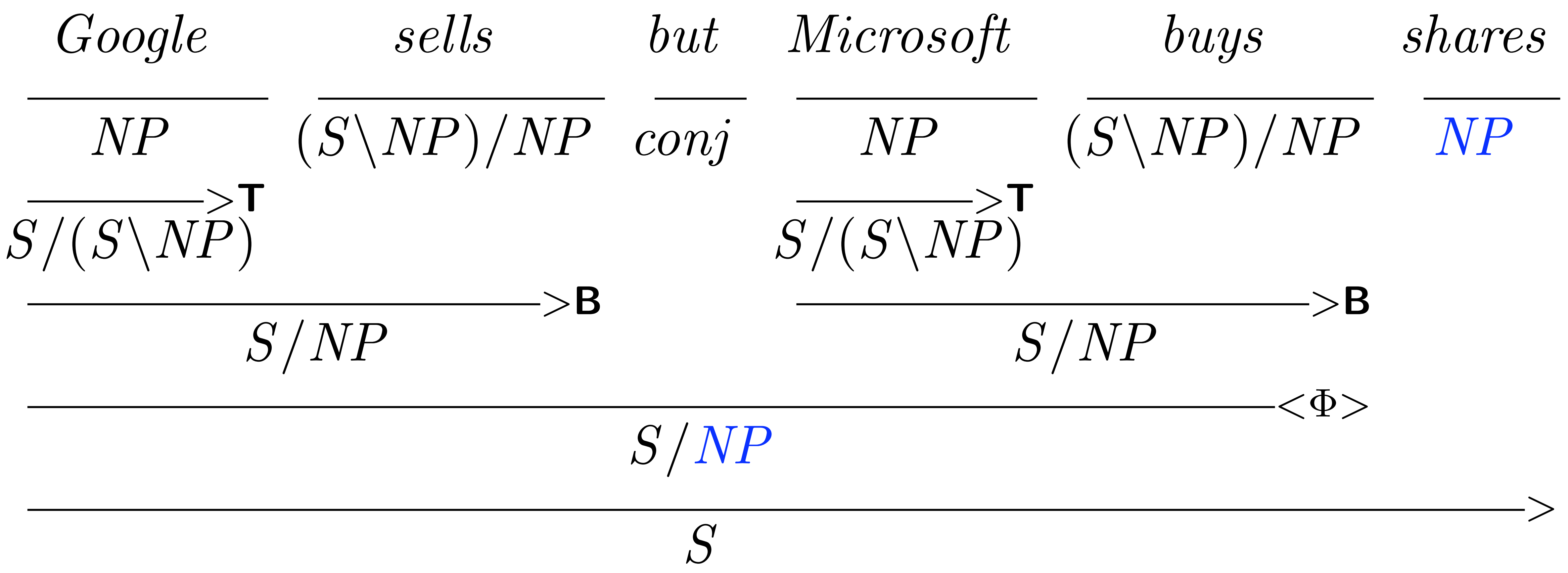
$$\begin{array}{cccccc}
\textit{Google} & \textit{sells} & \textit{but} & \textit{Microsoft} & \textit{buys} & \textit{shares}\\
\hline
NP & (S\backslash NP)/NP & conj & NP & (S\backslash NP)/NP & NP\\
\overline{\hspace{3em}}{>}\mathbf{T} & & & \overline{\hspace{3em}}{>}\mathbf{T} & & \\
S/(S\backslash NP) & & & S/(S\backslash NP) & & \\
\multicolumn{2}{c}{\overline{\hspace{12em}}{>}\mathbf{B}} & & \multicolumn{2}{c}{\overline{\hspace{12em}}{>}\mathbf{B}} & \\
\multicolumn{2}{c}{S/NP} & & \multicolumn{2}{c}{S/NP} &
\end{array}$$

$>$ **T**	type-raising
$>$ **B**	forward composition

Non-constituents & Coordination

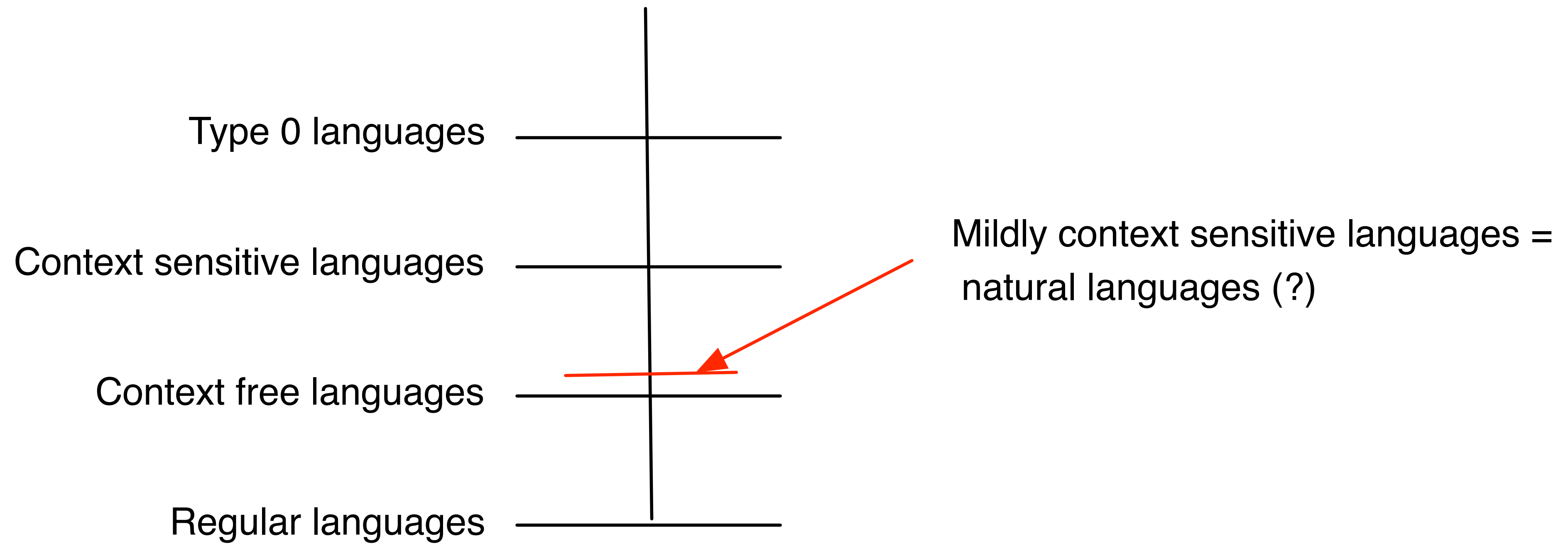
$$\begin{array}{cccccc}
\textit{Google} & \textit{sells} & \textit{but} & \textit{Microsoft} & \textit{buys} & \textit{shares}\\
\overline{NP} & \overline{(S\backslash NP)/NP} & \overline{conj} & \overline{NP} & \overline{(S\backslash NP)/NP} & \overline{NP}\\
\overline{}^{>\mathbf{T}} & & & \overline{}^{>\mathbf{T}} & & \\
S/(S\backslash NP) & & & S/(S\backslash NP) & & \\
\multicolumn{2}{c}{\overline{}^{>\mathbf{B}}} & & \multicolumn{2}{c}{\overline{}^{>\mathbf{B}}} & \\
\multicolumn{2}{c}{S/NP} & & \multicolumn{2}{c}{S/NP} & \\
\multicolumn{5}{c}{\overline{}^{<\Phi>}} & \\
\multicolumn{5}{c}{S/NP} &
\end{array}$$

Non-constituents & Coordination



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
    - Due to *crossing dependencies* (which CCG can handle)


Type 0 languages
Context sensitive languages
Context free languages
Regular languages
Mildly context sensitive languages = natural languages (?)

CCG Semantics

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

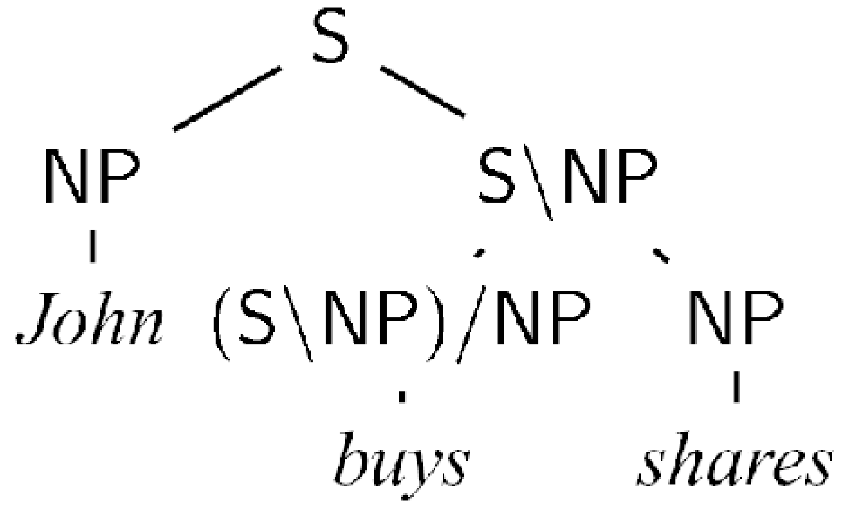
John ⊢ NP : *john′*

shares ⊢ NP : *shares′*

buys ⊢ (S\NP)/NP : $\lambda x.\lambda y.buys'xy$

sleeps ⊢ S\NP : $\lambda x.sleeps'x$

well ⊢ (S\NP)\(S\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.

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

Reading

- Jurafsky & Martin, chapter 17–20
- NLTK book, chapter 10
- McDonald et al., Non-projective Dependency Parsing using Spanning Tree Algorithms, *EMNLP* 2005. http://www.aclweb.org/anthology/H/H05/H05-1066.pdf
- Bansal et al., Structured Learning for Taxonomy Induction with Belief Propagation, *ACL* 2014. http://aclweb.org/anthology/P/P14/P14-1098.pdf