

Natural Language Processing



Parsing (Syntax and Semantics)

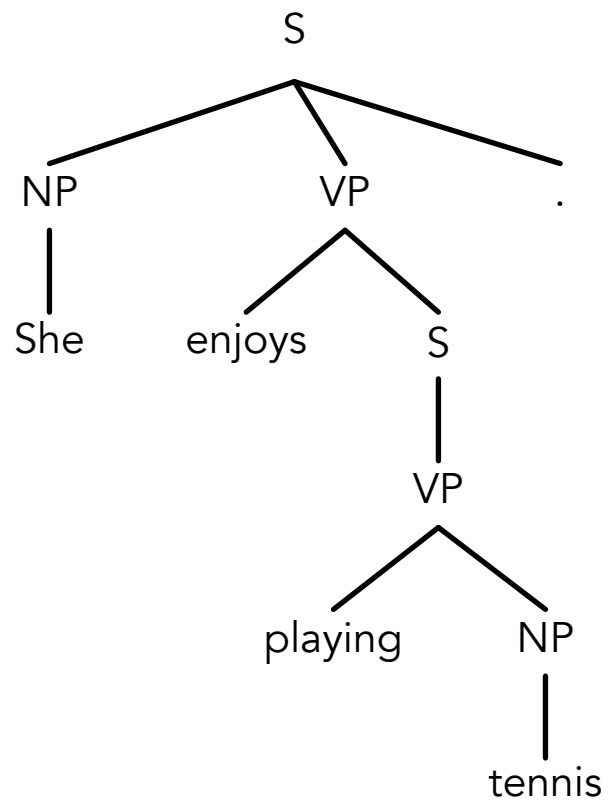


Syntactic Parsing

She enjoys playing tennis.

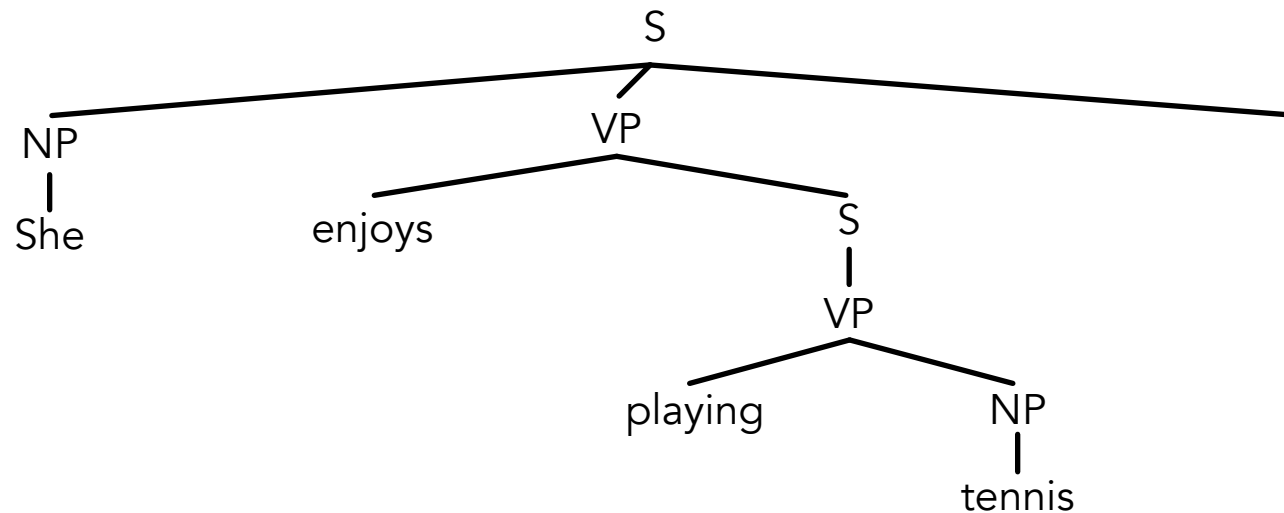


Syntactic Parsing





Span-Based Parsing



She enjoys playing tennis .



Parsing as Span Classification

VP



She

enjoys

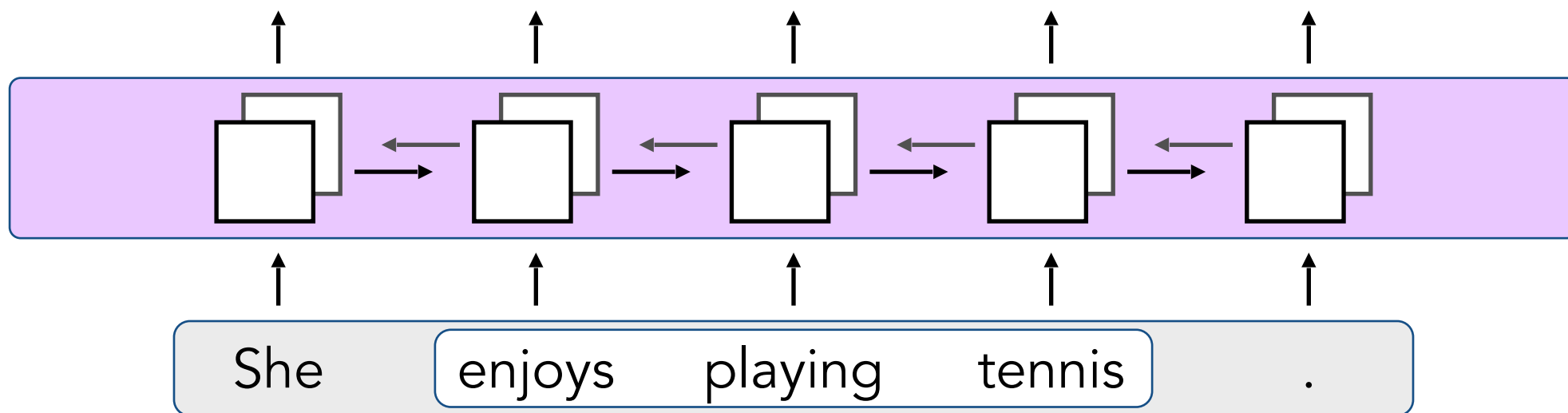
playing

tennis

.

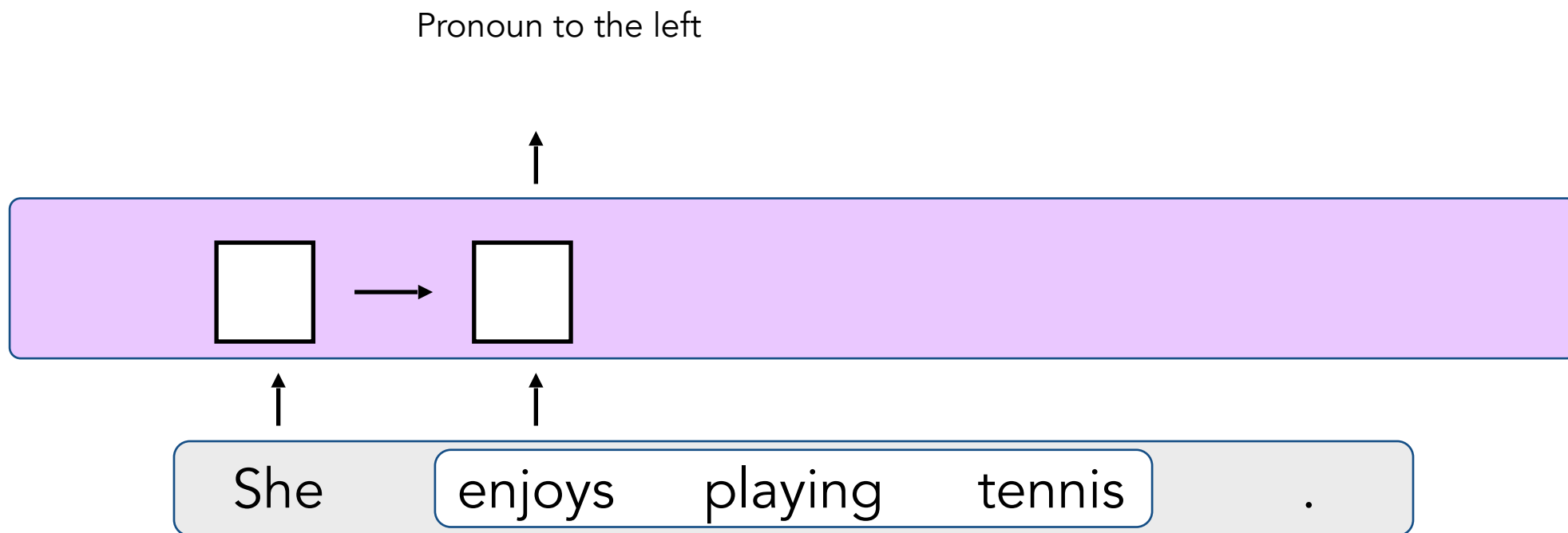


Routing with LSTMs



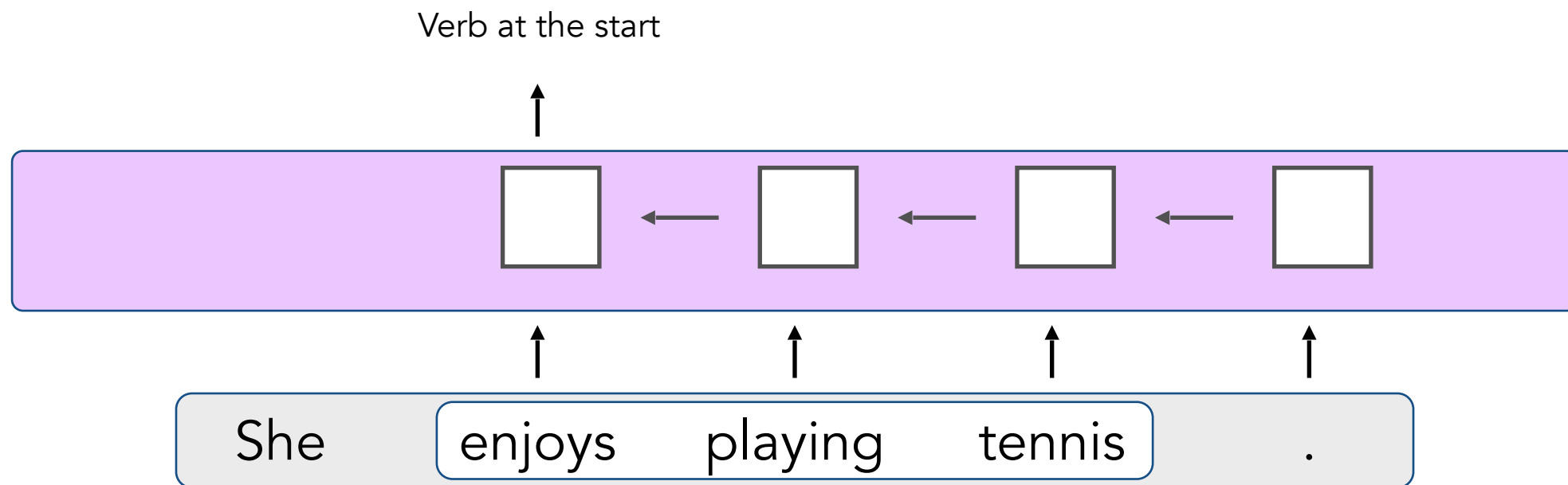


Routing with LSTMs



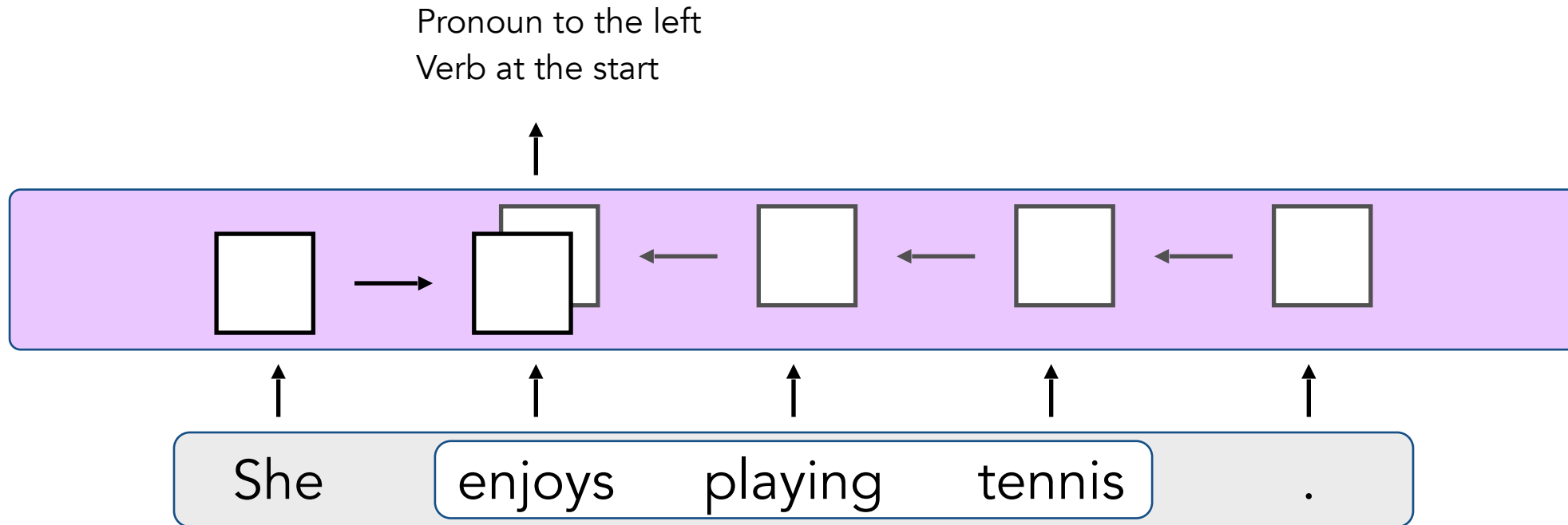


Routing with LSTMs



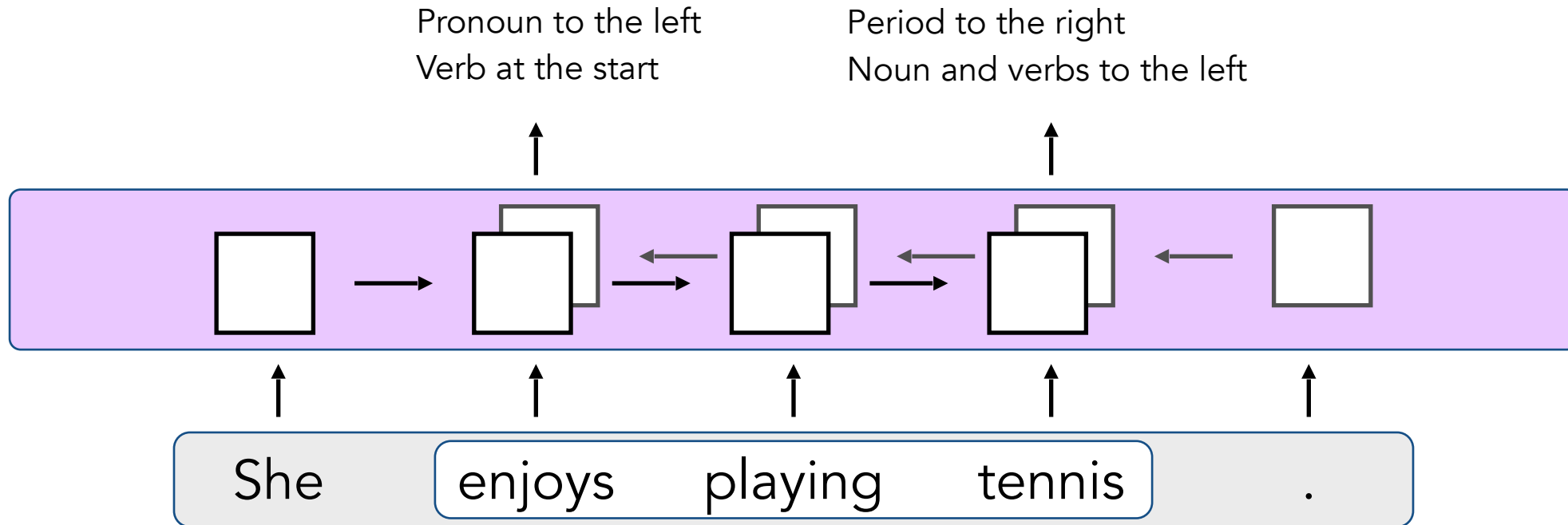


Routing with LSTMs



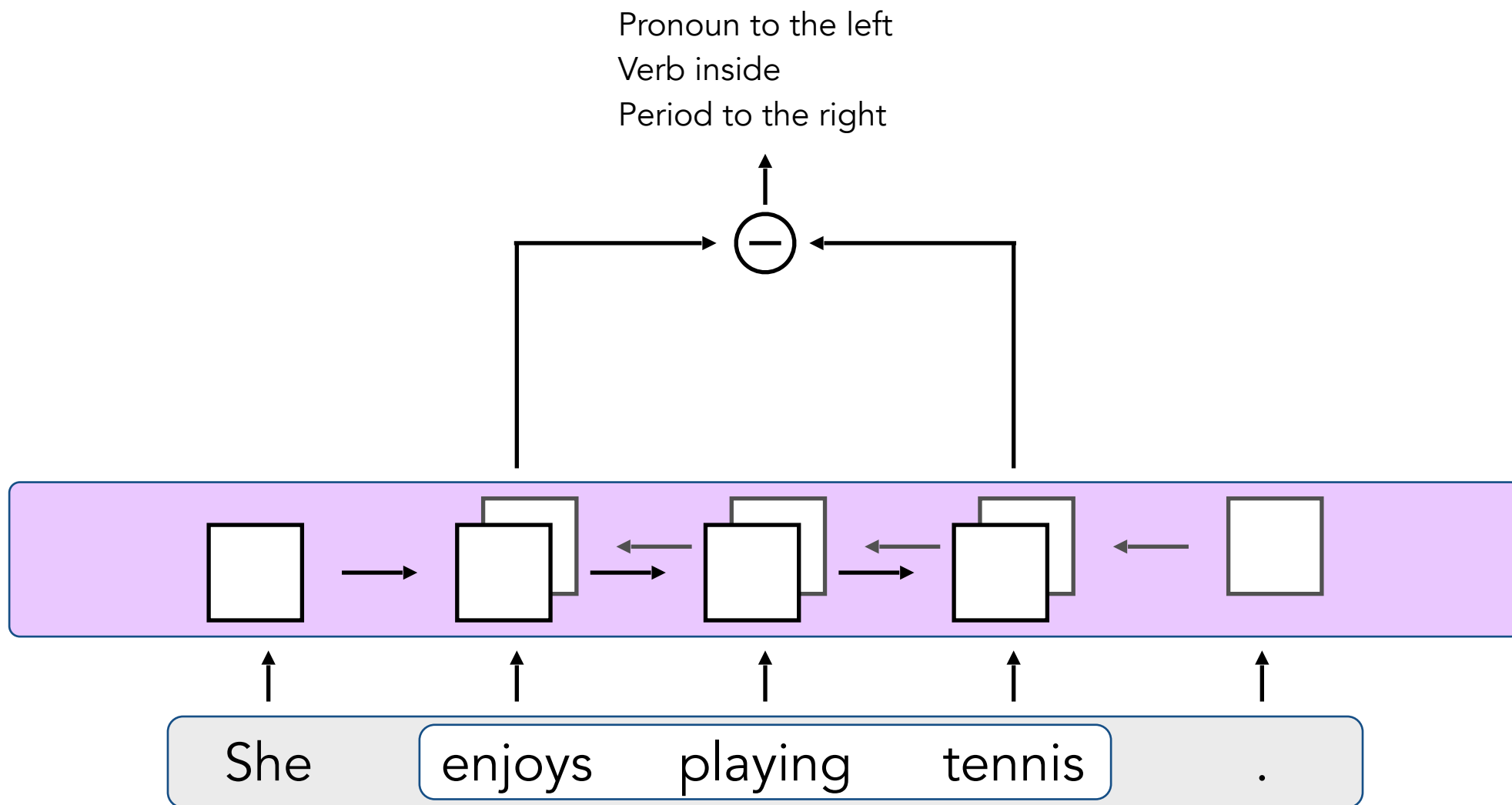


Routing with LSTMs



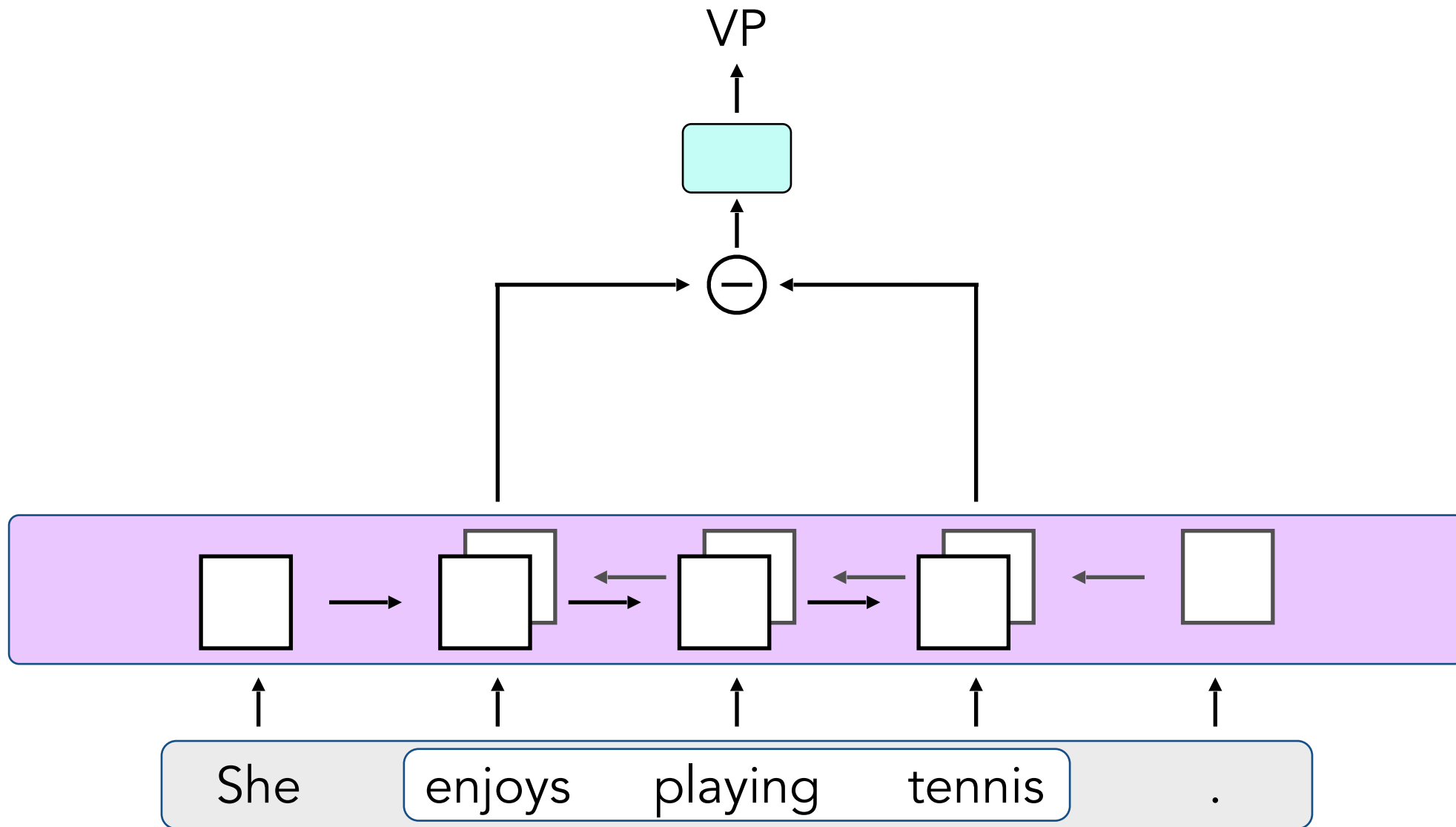


Span Classification

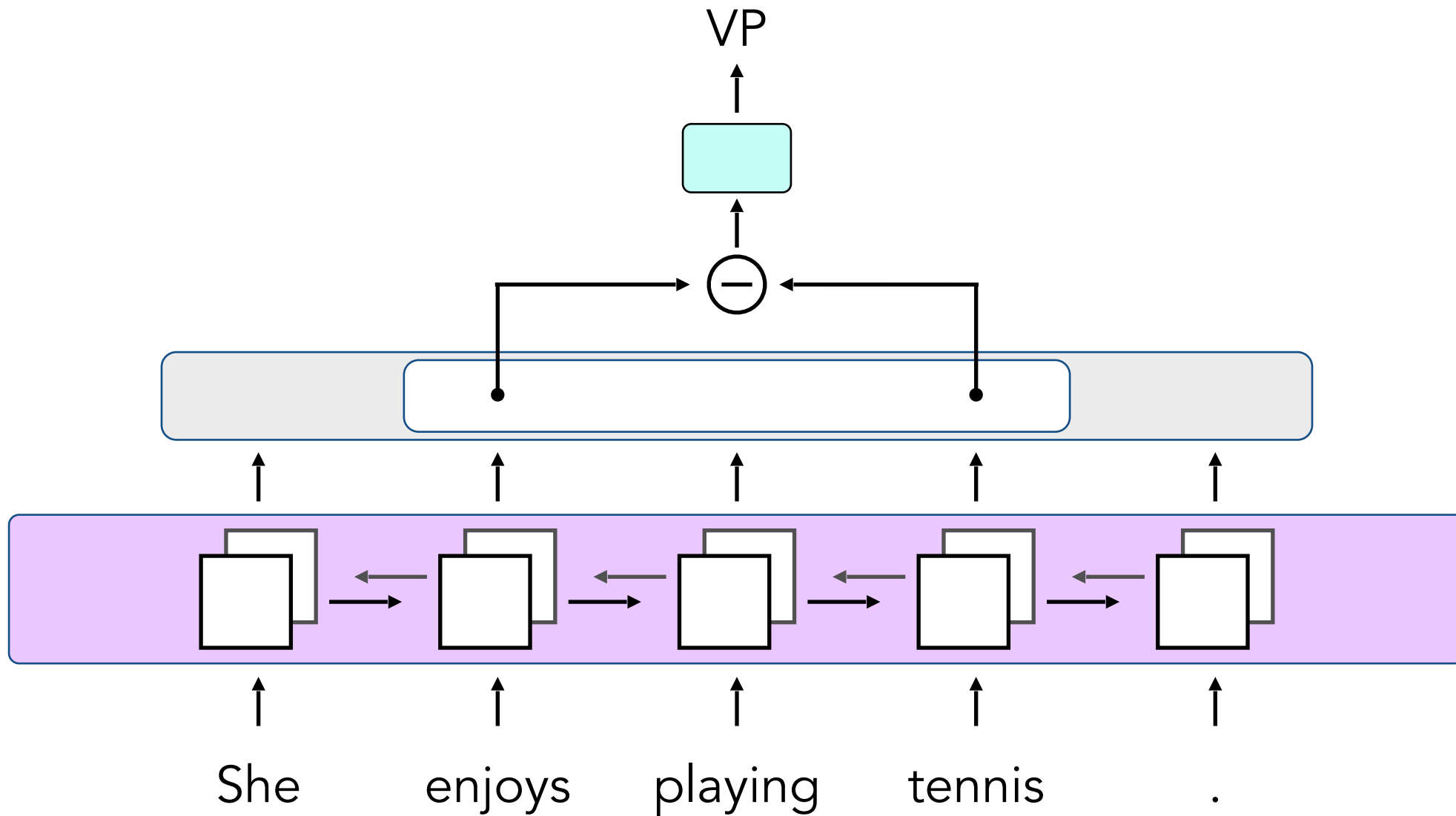




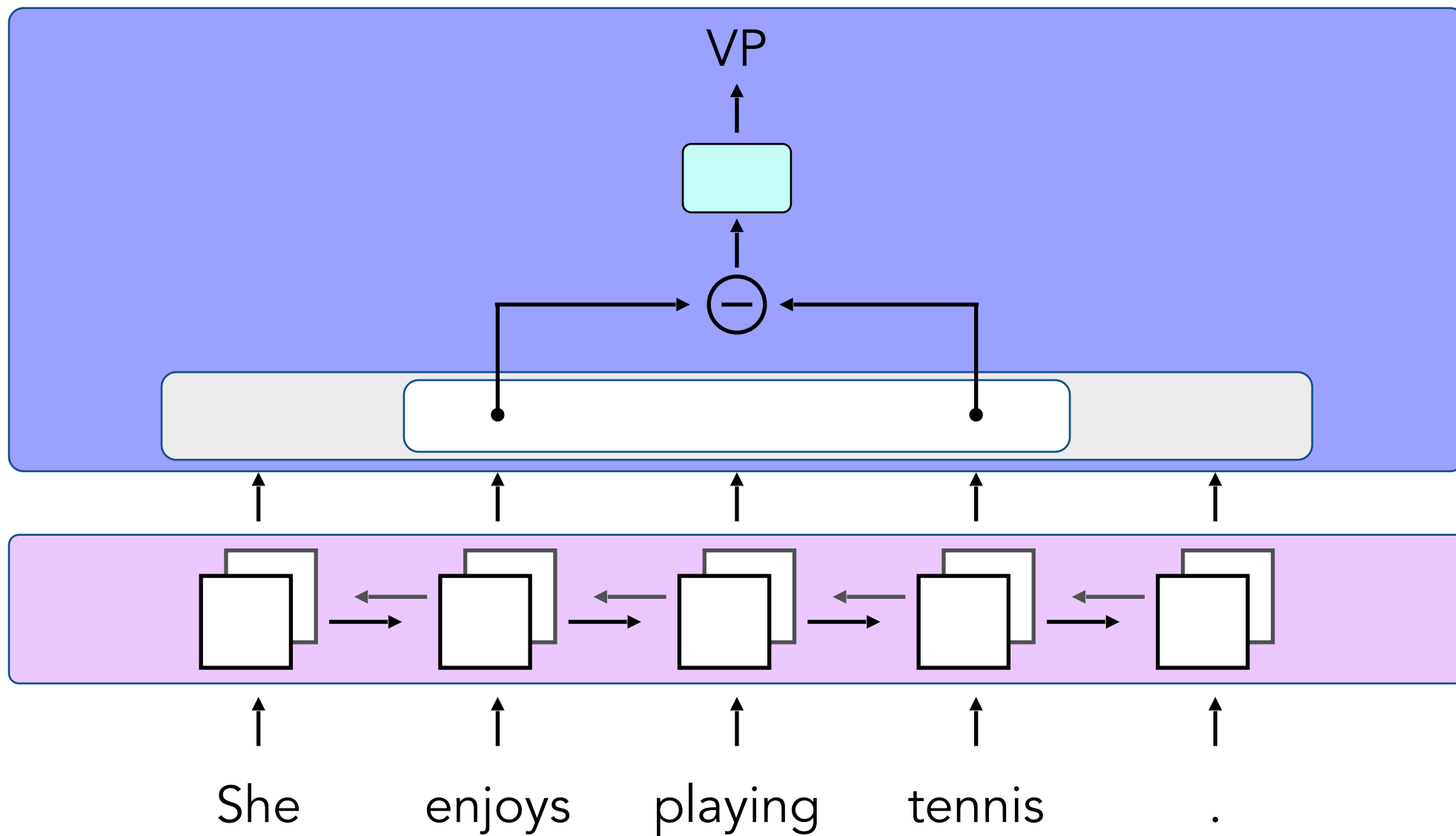
Span Classification



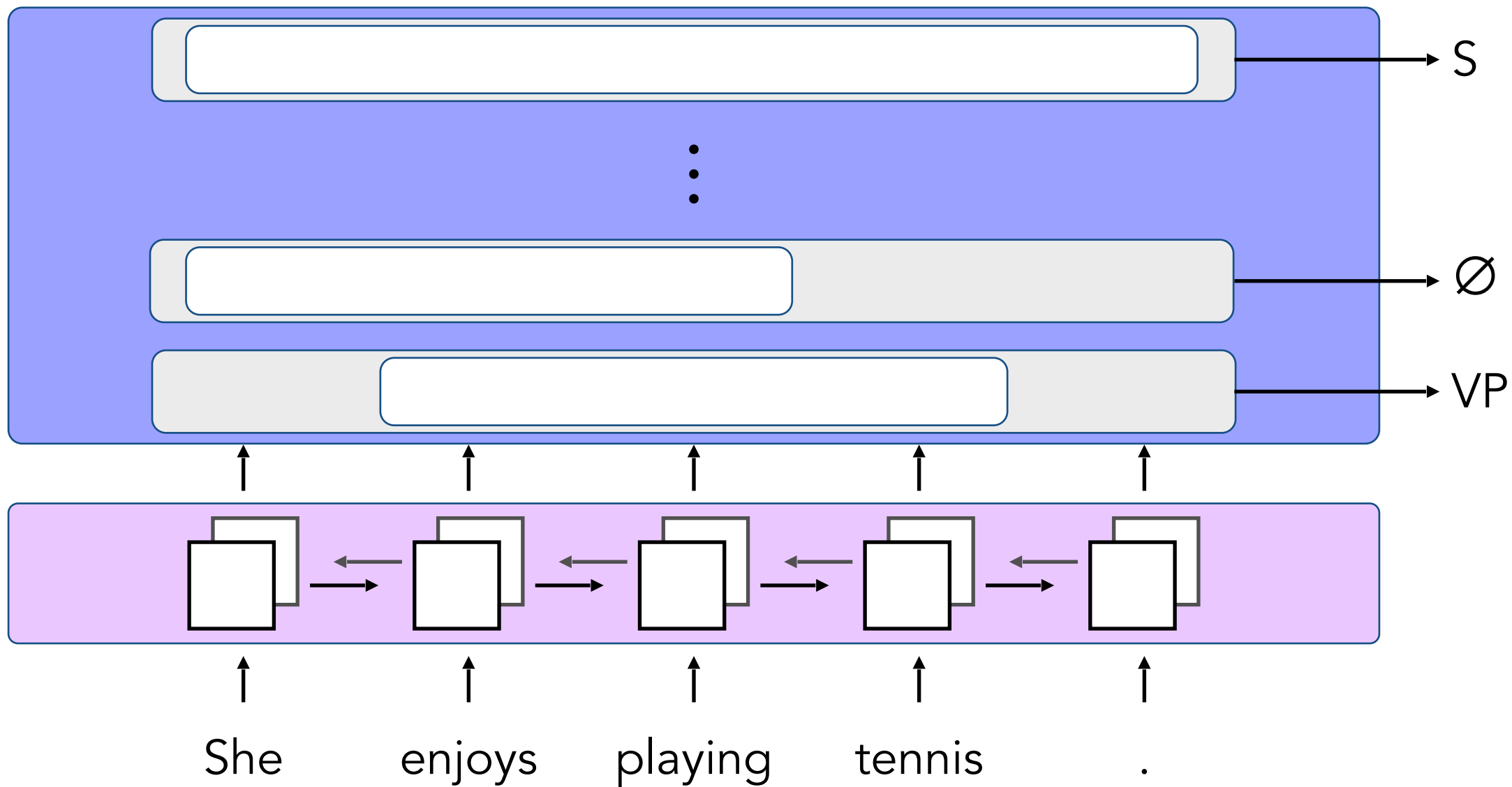
Span Classification



Span Classification

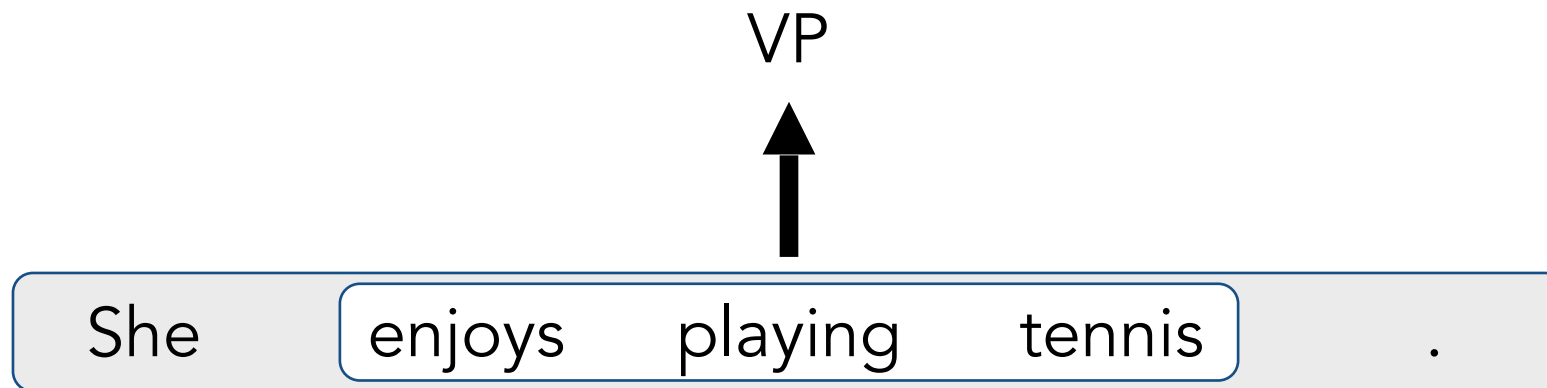
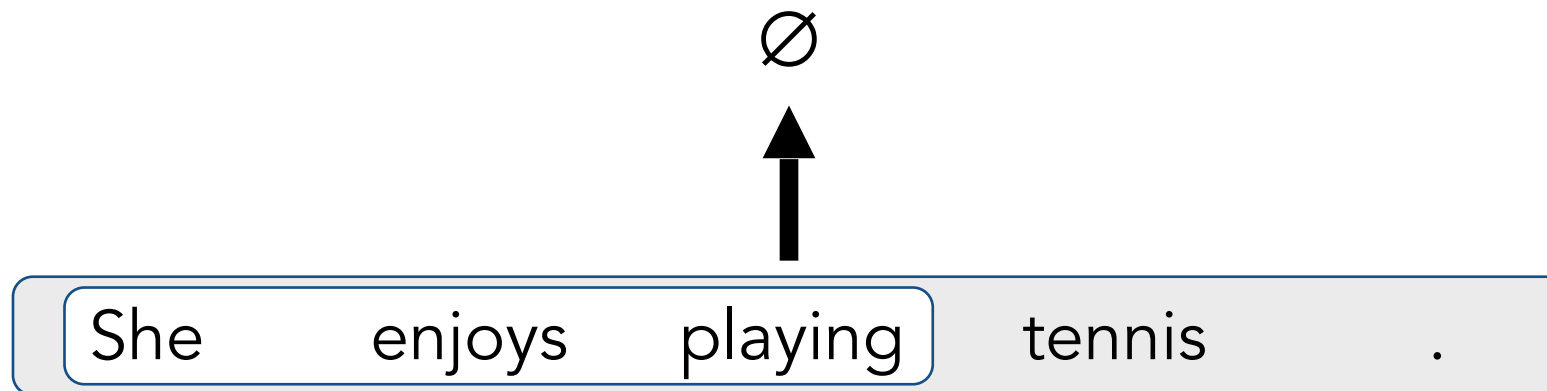


Span Classification



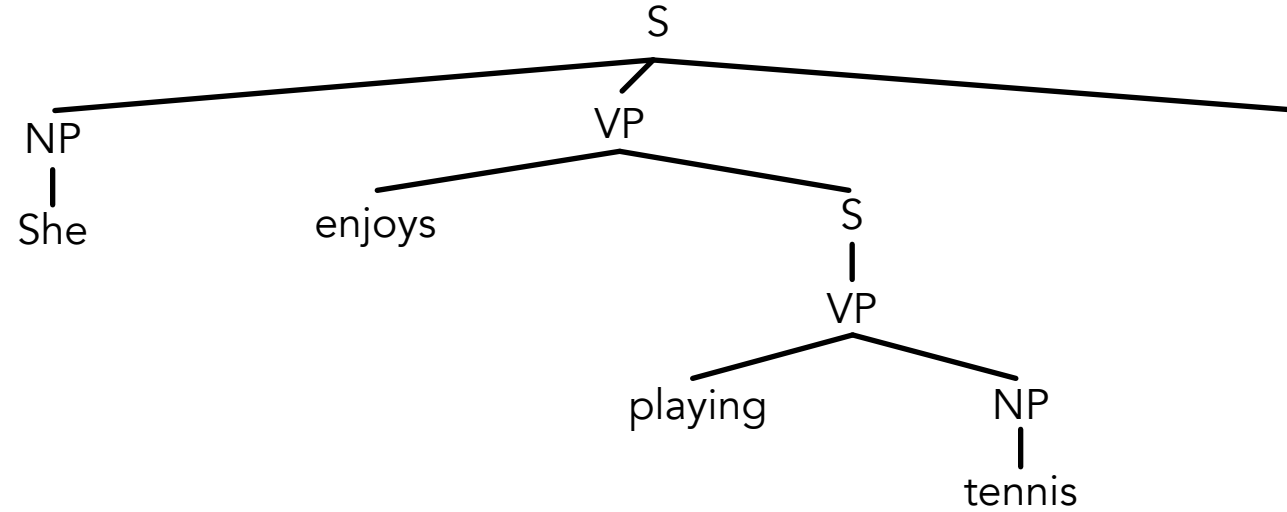


Non-Constituents





... But Will We Get a Tree Out?



S

NP

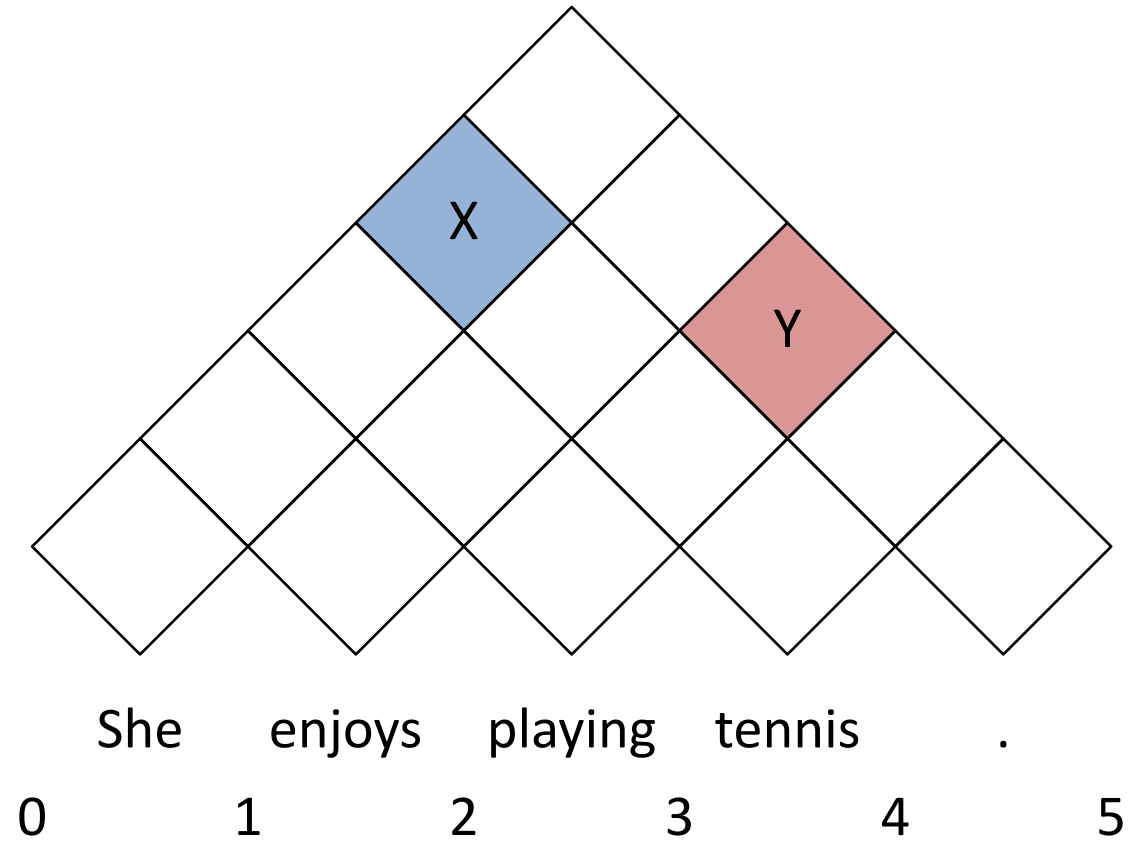
VP

S-VP

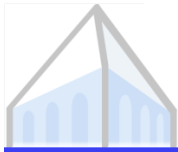
NP

She enjoys playing tennis .

Reconciliation

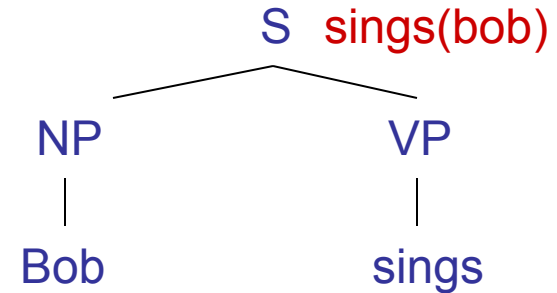


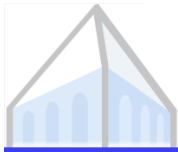
Truth-Conditional Semantics



Truth-Conditional Semantics

- Linguistic expressions:
 - “Bob sings”
- Logical translations:
 - $\text{sings}(\text{bob})$
 - Could be $p_{1218}(e_{397})$
- Denotation:
 - $[[\text{bob}]]$ = some specific person (in some context)
 - $[[\text{sings}(\text{bob})]]$ = ???
- Types on translations:
 - $\text{bob} : e$ (for entity)
 - $\text{sings}(\text{bob}) : t$ (for truth-value)





Truth-Conditional Semantics

- Proper names:

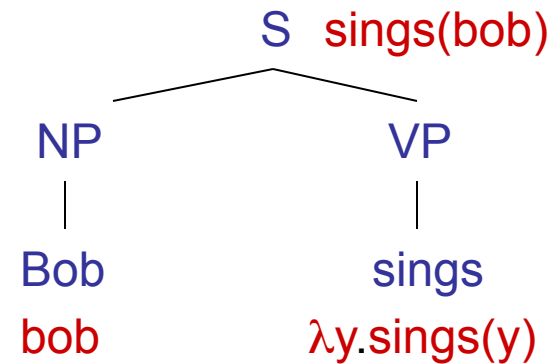
- Refer directly to some entity in the world
- Bob : bob $[[\text{bob}]]^w \rightarrow ???$

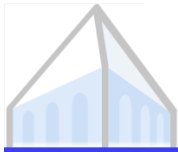
- Sentences:

- Are either true or false (given how the world actually is)
- Bob sings : sings(bob)

- So what about verbs (and verb phrases)?

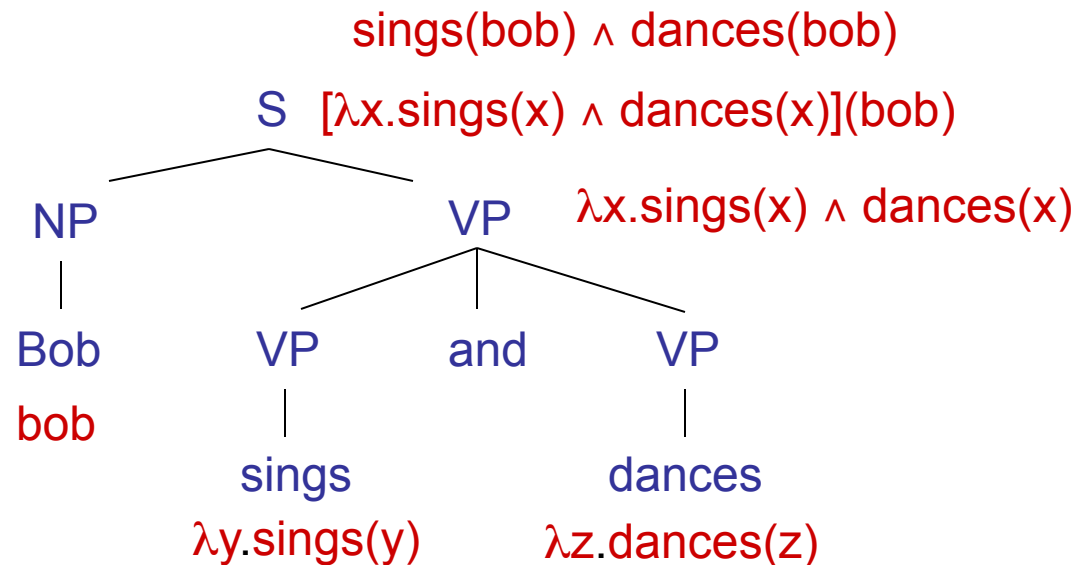
- sings must combine with bob to produce sings(bob)
- The λ -calculus is a notation for functions whose arguments are not yet filled.
- sings : $\lambda x.\text{sings}(x)$
- This is a *predicate* – a function which takes an entity (type e) and produces a truth value (type t). We can write its type as $e \rightarrow t$.
- Adjectives?





Compositional Semantics

- So now we have meanings for the words
- How do we know how to combine words?
- Associate a combination rule with each grammar rule:
 - $S : \beta(\alpha) \rightarrow NP : \alpha \quad VP : \beta$ (function application)
 - $VP : \lambda x . \alpha(x) \wedge \beta(x) \rightarrow VP : \alpha \quad \text{and} : \emptyset \quad VP : \beta$ (intersection)
- Example:





Denotation

- What do we do with logical translations?
 - Translation language (logical form) has fewer ambiguities
 - Can check truth value against a database
 - Denotation (“evaluation”) calculated using the database
 - Or the opposite: assert truth and modify a database, either explicitly or implicitly
eg prove a consequence from asserted axioms
 - Questions: check whether a statement in a corpus entails the (question, answer) pair:
 - “Bob sings and dances” → “Who sings?” + “Bob”
 - Chain together facts and use them for comprehension

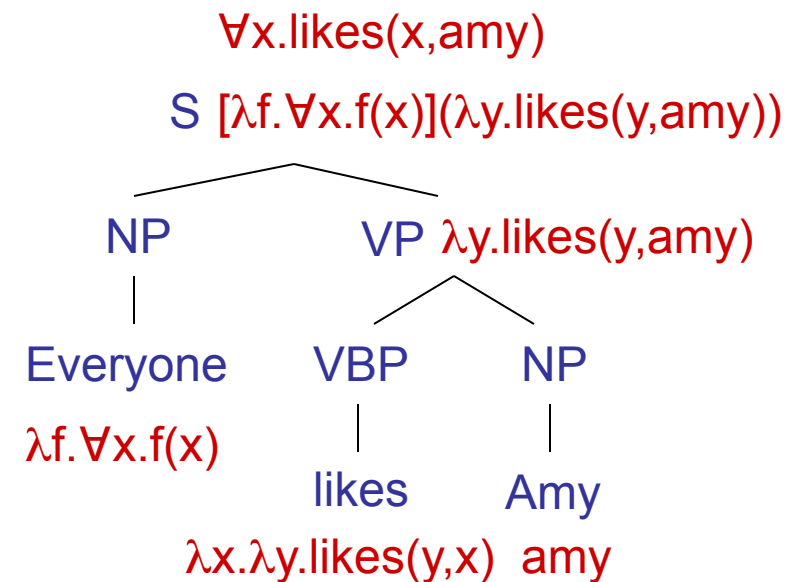
Other Cases

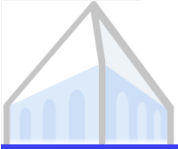
- Transitive verbs:

- likes : $\lambda x.\lambda y.likes(y,x)$
- Two-place predicates of type $e \rightarrow (e \rightarrow t)$.
- likes Amy : $\lambda y.likes(y,Amy)$ is just like a one-place predicate.

- Quantifiers:

- What does “Everyone” mean here?
- Everyone : $\lambda f.\forall x.f(x)$
- Mostly works, but some problems
 - Have to change our NP/VP rule.
 - Won't work for “Amy likes everyone.”
- “Everyone likes someone.”
- This gets tricky quickly!

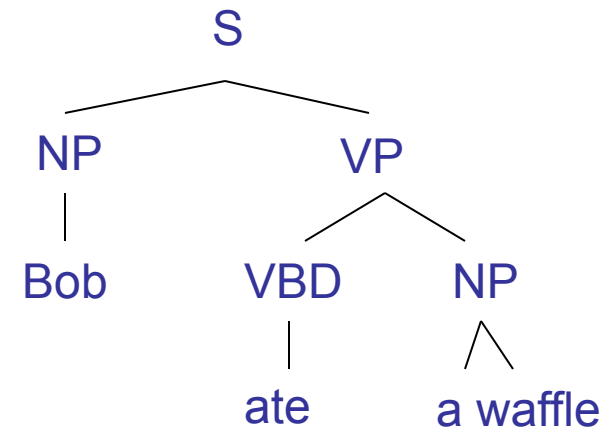


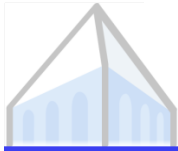


Indefinites

- First try
 - “Bob ate a waffle” : $\text{ate}(\text{bob}, \text{waffle})$
 - “Amy ate a waffle” : $\text{ate}(\text{amy}, \text{waffle})$

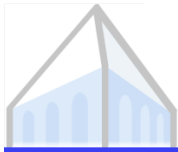
- Can't be right!
 - $\exists x : \text{waffle}(x) \wedge \text{ate}(\text{bob}, x)$
 - What does the translation of “a” have to be?
 - What about “the”?
 - What about “every”?





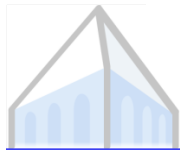
Grounding

- Grounding
 - So why does the translation `likes` : $\lambda x.\lambda y.likes(y,x)$ have anything to do with actual liking?
 - It doesn't (unless the denotation model says so)
 - Sometimes that's enough: wire up `bought` to the appropriate entry in a database
- Meaning postulates
 - Insist, e.g $\forall x,y.likes(y,x) \rightarrow knows(y,x)$
 - This gets into lexical semantics issues
- Statistical / neural version?



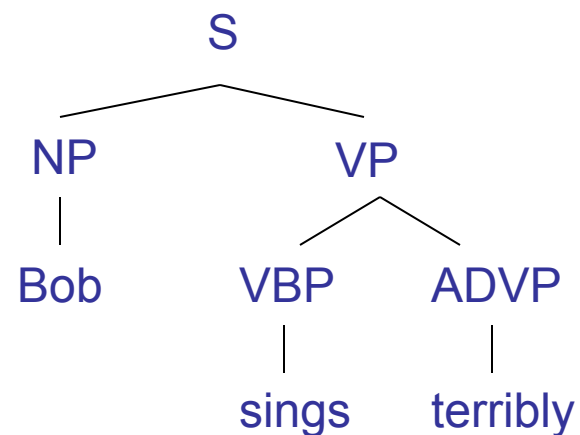
Tense and Events

- In general, you don't get far with verbs as predicates
- Better to have event variables e
 - “Alice danced” : $\text{danced}(\text{alice})$
 - $\exists e : \text{dance}(e) \wedge \text{agent}(e, \text{alice}) \wedge (\text{time}(e) < \text{now})$
- Event variables let you talk about non-trivial tense / aspect structures
 - “Alice had been dancing when Bob sneezed”
 - $\exists e, e' : \text{dance}(e) \wedge \text{agent}(e, \text{alice}) \wedge$
 $\text{sneeze}(e') \wedge \text{agent}(e', \text{bob}) \wedge$
 $(\text{start}(e) < \text{start}(e') \wedge \text{end}(e) = \text{end}(e')) \wedge$
 $(\text{time}(e') < \text{now})$
- Minimal recursion semantics, cf “object oriented” thinking



Adverbs

- What about adverbs?
 - “Bob sings terribly”
 - $\text{terribly}(\text{sings}(\text{bob}))?$
 - $(\text{terribly}(\text{sings}))(\text{bob})?$
 - $\exists e \text{ present}(e) \wedge \text{type}(e, \text{singing}) \wedge \text{agent}(e, \text{bob}) \wedge \text{manner}(e, \text{terrible})?$
 - Gets complex quickly...





Propositional Attitudes

- “Bob thinks that I am a gummi bear”
 - $\text{thinks}(\text{bob}, \text{gummi}(\text{me}))$?
 - $\text{thinks}(\text{bob}, \text{“I am a gummi bear”})$?
 - $\text{thinks}(\text{bob}, \wedge \text{gummi}(\text{me}))$?
- Usual solution involves intensions ($\wedge X$) which are, roughly, the set of possible worlds (or conditions) in which X is true
- Hard to deal with computationally
 - Modeling other agents’ models, etc
 - Can come up in even simple dialog scenarios, e.g., if you want to talk about what your bill claims you bought vs. what you actually bought



Trickier Stuff

- Non-Intersective Adjectives
 - green ball : $\lambda x. [\text{green}(x) \wedge \text{ball}(x)]$
 - fake diamond : $\lambda x. [\text{fake}(x) \wedge \text{diamond}(x)]$? $\longrightarrow \lambda x. [\text{fake}(\text{diamond}(x))]$
- Generalized Quantifiers
 - the : $\lambda f. [\text{unique-member}(f)]$
 - all : $\lambda f. \lambda g [\forall x. f(x) \rightarrow g(x)]$
 - most?
 - Could do with more general second order predicates, too (why worse?)
 - $\text{the}(\text{cat}, \text{meows}), \text{all}(\text{cat}, \text{meows})$
- Generics
 - “Cats like naps”
 - “The players scored a goal”
- Pronouns (and bound anaphora)
 - “If you have a dime, put it in the meter.”
- ... the list goes on and on!



Scope Ambiguities

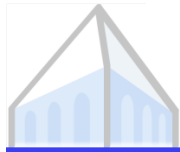
- Quantifier scope

- “All majors take a data science class”
- “Someone took each of the electives”
- “Everyone didn’t hand in their exam”

- Deciding between readings

- Multiple ways to work this out
 - Make it syntactic (movement)
 - Make it lexical (type-shifting)

Logical Form Translation



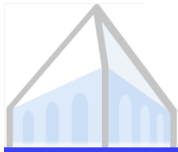
Supervision: Logical Forms

- Data: input sentences paired with annotated LFs

Show me flights to Prague

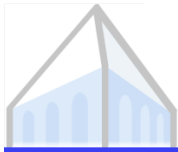
$\lambda \mathbf{x}. \mathit{flight}(\mathbf{x}) \wedge \mathit{to}(\mathbf{x}, \mathit{PRG})$

- Problem: no supervision on how to get from sentence to LF
- But we can assume our LF has been generated from some formal grammar
- Combinatory Categorical Grammar (CCG)



CCG: Lexicon

Words	Category
<code>flights</code>	<code>N : $\lambda x. flight(x)$</code>
<code>to</code>	<code>(N\N) / NP : $\lambda x. \lambda f. \lambda y. f(x) \wedge to(y, x)$</code>
<code>Prague</code>	<code>NP : PRG</code>
<code>New York city</code>	<code>NP : NYC</code>
<code>...</code>	<code>...</code>



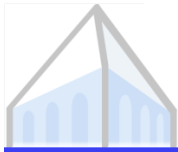
CCG: Combinators

Application

- $X/Y : f \quad Y : a \Rightarrow X : f(a)$
- $Y : a \quad X \backslash Y : f \Rightarrow X : f(a)$

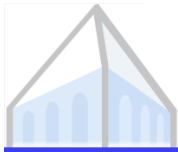
Composition

- $X/Y : f \quad Y/Z : g \Rightarrow X/Z : \lambda x. f(g(x))$
- $Y \backslash Z : f \quad X \backslash Y : g \Rightarrow X \backslash Z : \lambda x. f(g(x))$



CCG: Parsing

Show me	flights	to	Prague
S/N	N	(N\N) /NP	NP
$\lambda f.f$	$\lambda x.flight(x)$	$\lambda y.\lambda f.\lambda x.f(y) \wedge to(x,y)$	PRG
		N\N	
		$\lambda f.\lambda x.f(x) \wedge to(x,PRG)$	
		N	
		$\lambda x.flight(x) \wedge to(x,PRG)$	
		S	
		$\lambda x.flight(x) \wedge to(x,PRG)$	



Weighted CCG

- Lexicon Λ
- GEN: all possible parses y for sentence x given the lexicon

- Feature function

$$f : X \times Y \rightarrow \mathbb{R}^m$$

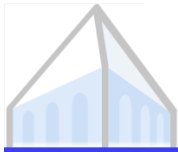
- (Learned) weights

$$w \in \mathbb{R}^m$$

- Best parse:

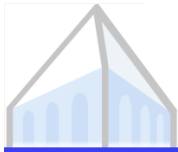
$$y^* = \arg \max_{y \in GEN(x, \Lambda)} w \cdot f(x, y)$$

Words	Category
flights	$N : \lambda x. flight(x)$
to	$(N \setminus N) / NP : \lambda x. \lambda f. \lambda y. f(x) \wedge to(y, x)$
Prague	$NP : PRG$
New York city	$NP : NYC$
...	...



Training (ZC05/07)

- Start with (x, z) sentence-LF pairs and a small seed lexicon
- Iterate T times:
 - Propose new lexical entries from each example (x, z) :
 - Generate all possible lexical entries pairing words/phrases in x with predicates in z
 - Use GEN to get all possible parses of x given the existing and new lexicon
 - Find the best parse y among these and add its lexical entries to the existing lexicon



GENLEX: Substrings X Categories

Input Training Example

Sentence: Show me flights to Prague.
Logic Form: $\lambda x. flight(x) \wedge to(x, PRG)$

Output Lexicon

All possible substrings:

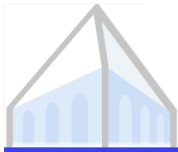
Show
me
flights
Show me
Show me flights
Show me flights to
...

X

Categories created by rules that trigger on the logical form:

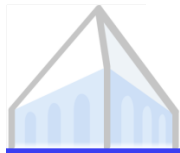
NP : PRG
N : $\lambda x. flight(x)$
(S\NP)/NP : $\lambda x. \lambda y. to(y, x)$
(N\N)/NP : $\lambda y. \lambda f. \lambda x. \dots$

...



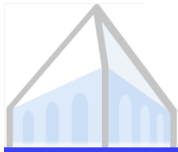
Training (ZC05/07)

- Start with (x, z) sentence-LF pairs and a small seed lexicon
- Iterate T times:
 - Propose new lexical entries from each example (x, z)
 - Update weights:
 - Re-parse all examples using newest lexicon and GEN
 - Sort parses into “good” and “bad” according to whether they are valid or invalid
 - Update weights to upweight “good” parses and downweight “bad” parses



Training (ZC05/07)

- Start with (x, z) sentence-LF pairs and a small seed lexicon
- Iterate T times:
 - Propose new lexical entries from each example (x, z)
 - Update weights
- Return full lexicon and weights



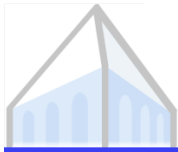
Supervision: Denotations Only

- Data: input sentences paired with denotations only (no LFs)

`Show me flights to Prague`

`Flight #s: 123, 456, 78, 342`

- Problem: no LF supervision at all!
 - Even worse problem of spuriousness
 - Complicates lexicon building
- Can still take advantage of knowing there's a (latent) structured representation



Learning from Denotations

- Example applications:
 - Grounded QA
 - Instruction following
 - Truth-conditional semantics
- Modification of ZC05/07 approach
 - New validation function: does proposed parse+LF yield expected denotation?
 - New method for generating lexical entries: place constraints (e.g., type constraints) on possible new entries

Year	City	Country	Nations
1896	Athens	Greece	14
1900	Paris	France	24
1904	St. Louis	USA	12
...
2004	Athens	Greece	201
2008	Beijing	China	204
2012	London	UK	204

$x =$ Greece held its last Summer Olympics in which year?

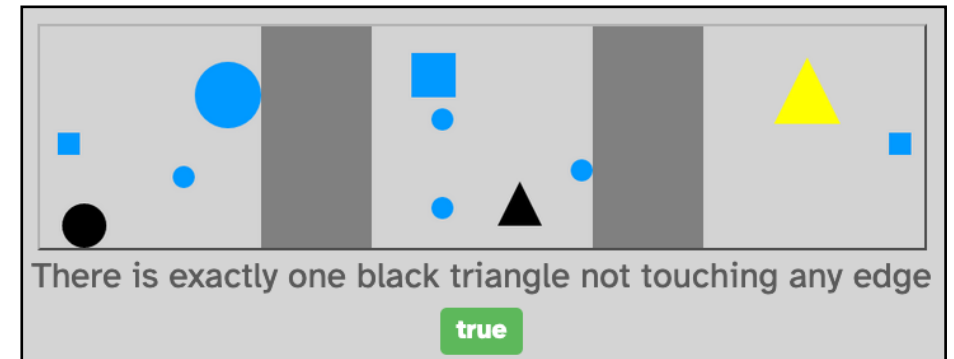
$y = 2004$

WikiTableQuestions, Pasupat and Liang 2015, ACL

at the chair, move forward three steps past the sofa



Artzi and Zettlemoyer 2013, TACL



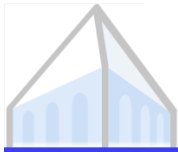
NLVR, Suhr et al. 2017, ACL

Neural Approaches

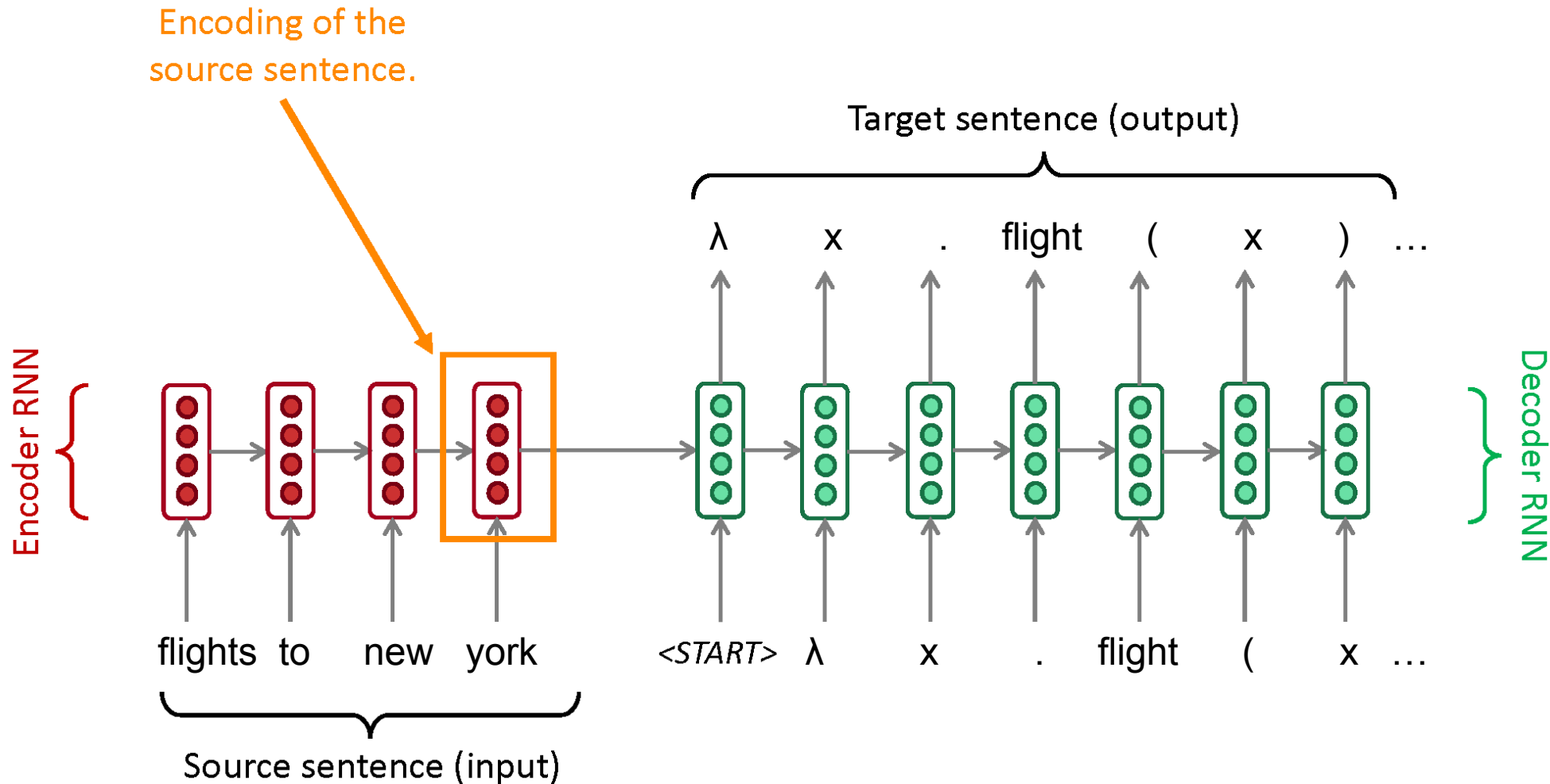


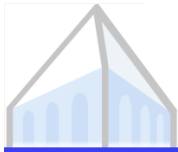
Sequence-to-Sequence Models

- Same methods from NMT! Encode input with an RNN, decode LF token-by-token
- Training: maximize log likelihood of gold LF conditioned on input utterance
- Can apply techniques like attention, beam search, etc.
- Problems:
 - Out-of-vocabulary terms, e.g., proper names (also a problem in MT)
 - No longer a clear divide between lexical and compositional semantics
 - No guarantee of syntactic validity or executability

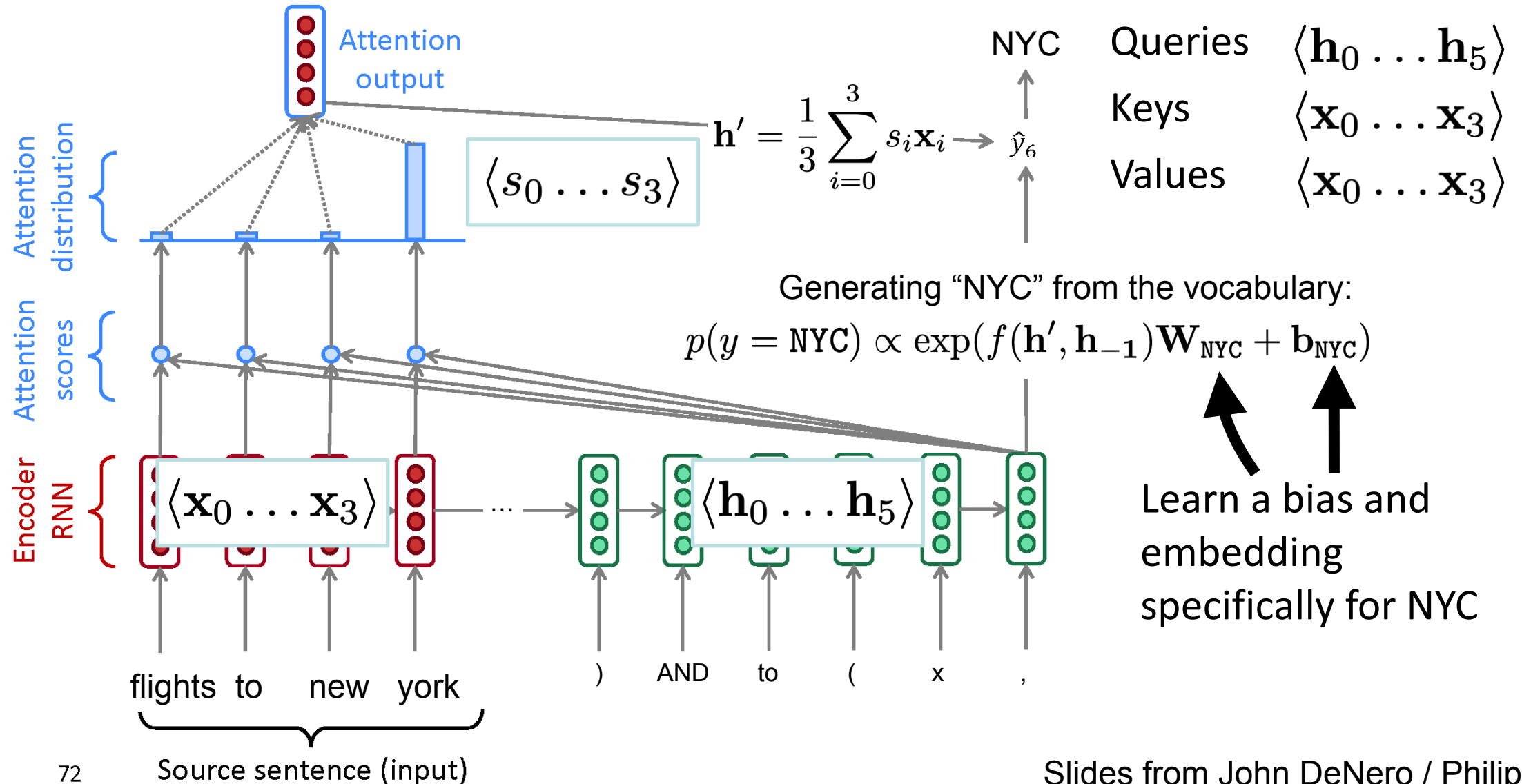


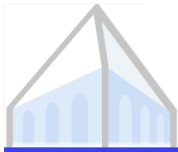
Attending, Pointing, and Copying



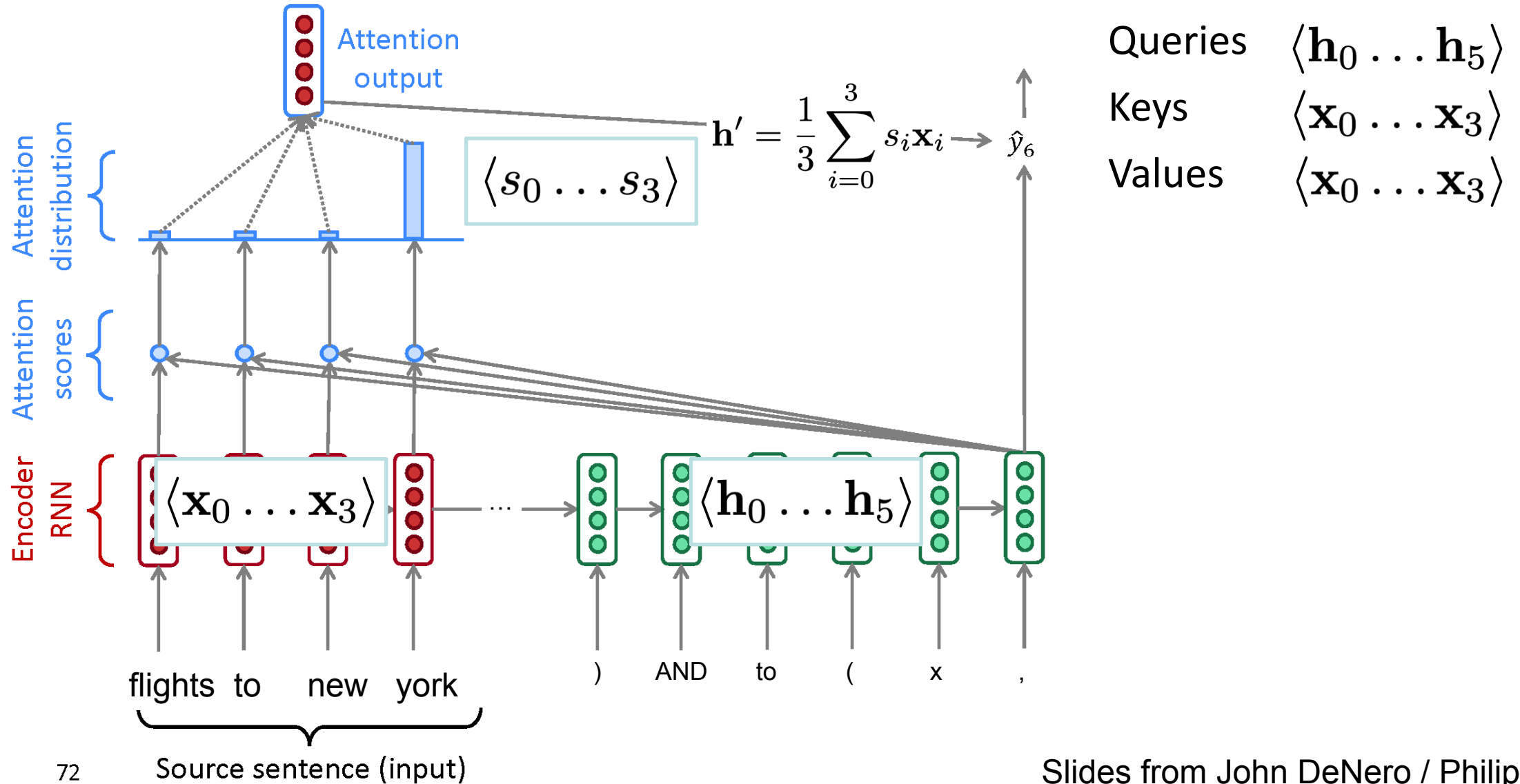


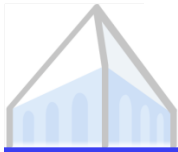
Attending, Pointing, and Copying



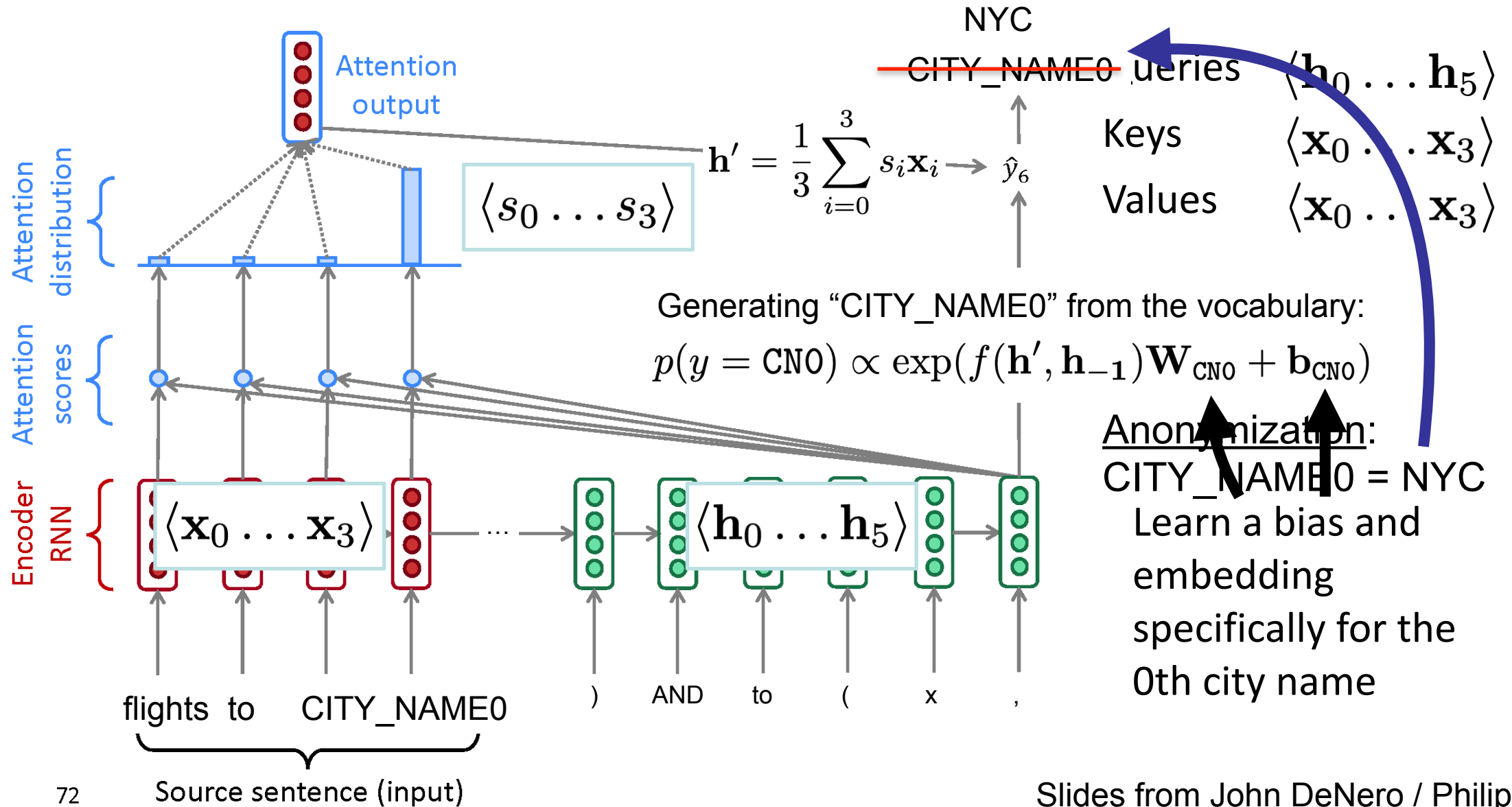


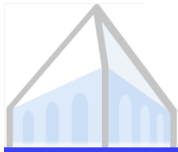
Attending, Pointing, and Copying



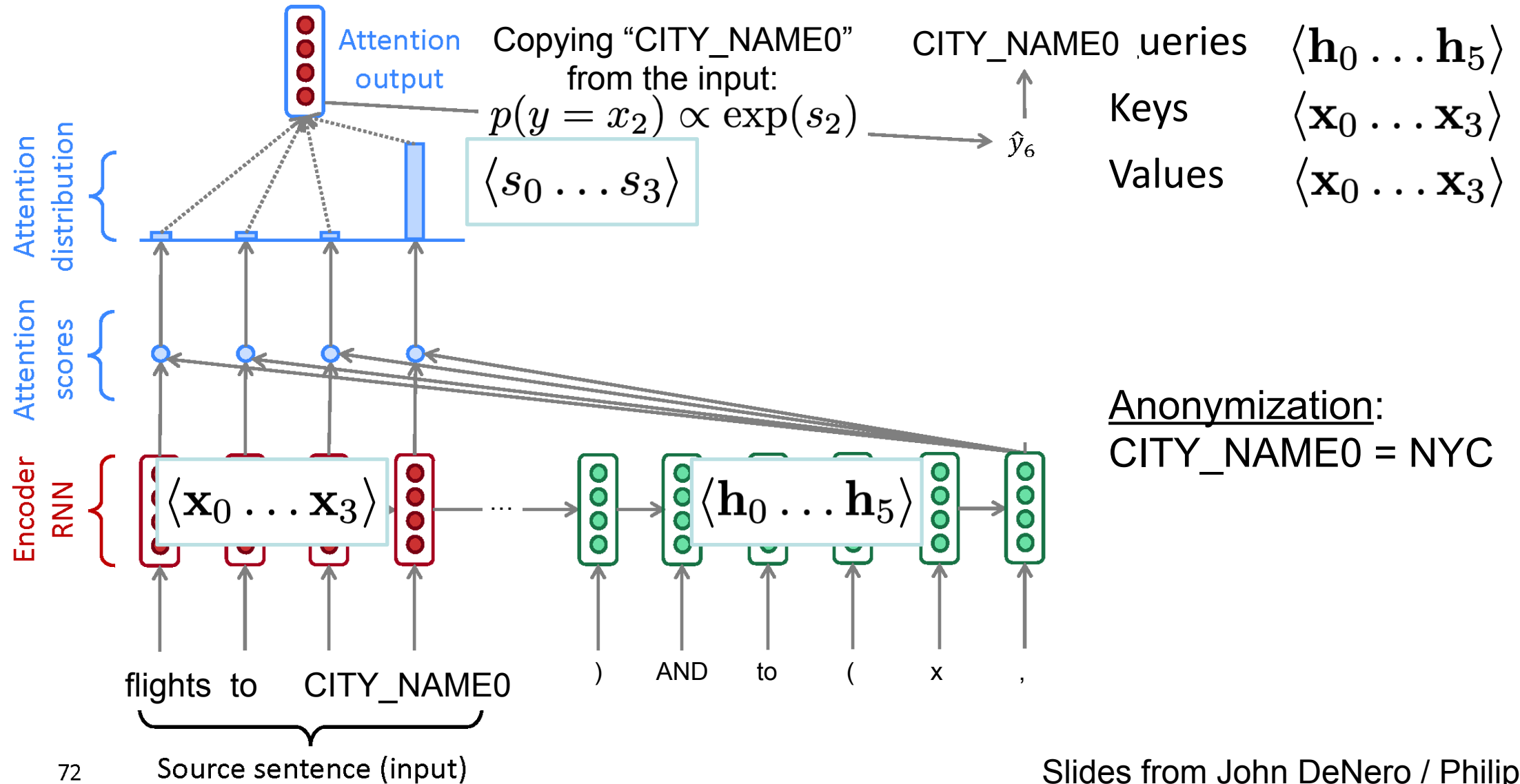


Attending, Pointing, and Copying





Attending, Pointing, and Copying



Intrinsic Structure



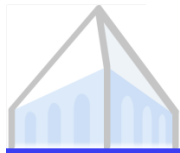
Constraints

- With token-by-token decoding, we lose the benefit of generating from a grammar
 - Our network now needs to (implicitly) learn the grammar from data
 - No guarantees that it will generate executable code
 - Syntax
 - Semantics
- How can we take advantage of this underlying structure?



Rejection Sampling

- Generate a number of candidate(s) (e.g., via beam search)
- Execute candidates, ensuring it compiles and runs without an error
- Return the highest-probability candidate that executes
- Could be very inefficient, especially because it requires running code at inference time



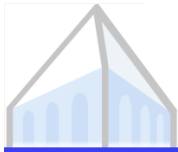
Intermediate Logical Forms

- Design an intermediate representation that implicitly captures structural dependencies in the code
- Generation in this output space reduces the need for the network to learn particular dependencies

SQL: SELECT people.name FROM people JOIN films ON people.id = film.person_id
 WHERE films.id = 5

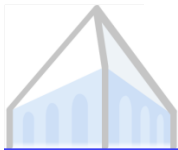
SQL^{UF}: SELECT people.name **UF** WHERE films.id = 5

- However:
 - Cannot capture full expressivity of target language
 - Requires manual engineering of intermediate language, and deterministic mapping to / from the target language



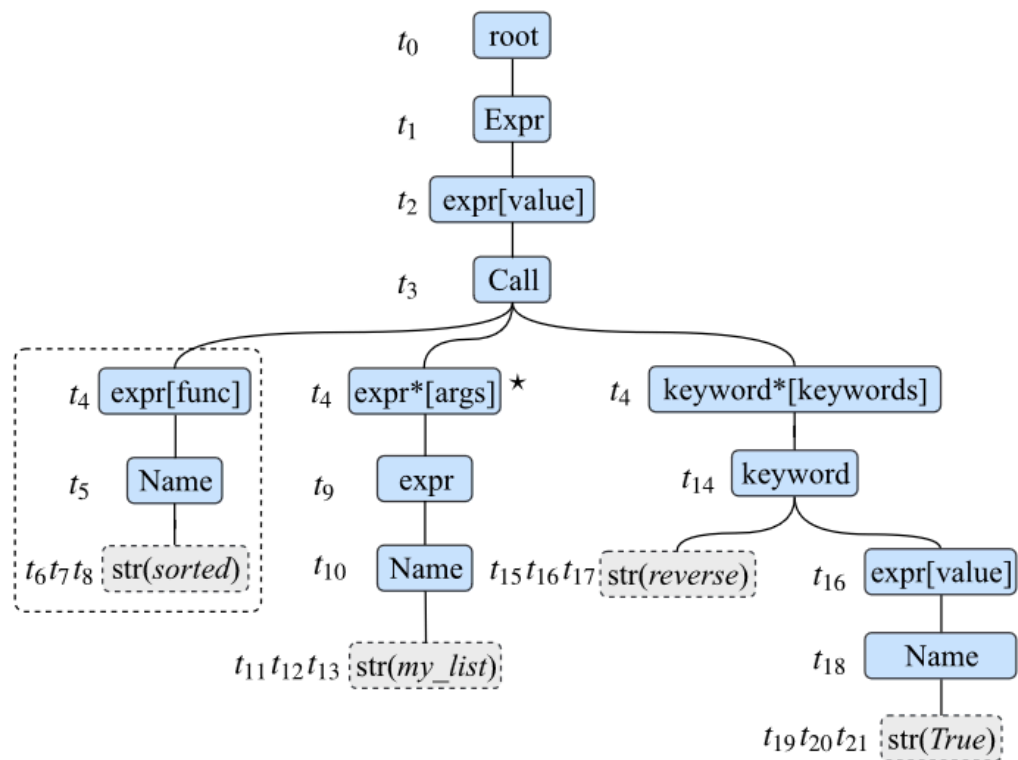
Constrained Decoding

- Generate actions that construct the AST that underlies the target code rather than the code itself
- Output space includes two types of actions:
 - ApplyRule r — apply production rule r to the current derivation tree
 - GenerateToken t — generate a variable terminal t
- Tokens t in sequence comprise the surface form of the code
- The current derivation tree constrains the set of rules r that can be applied and tokens t that can be generated
- At decoding time, simply mask out rules and tokens that cannot be generated



Constrained Decoding

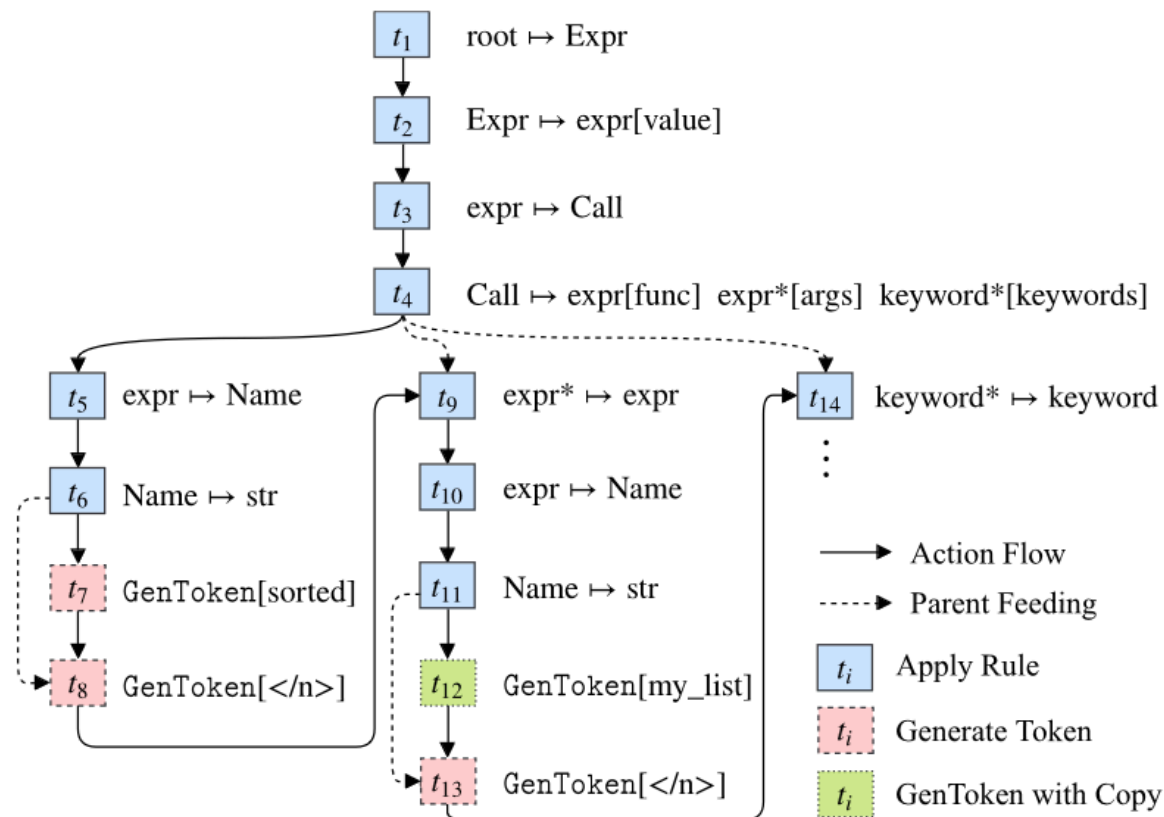
Generated AST



(a)

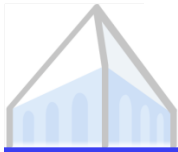
Input: sort my_list in descending order

Production Rule Actions



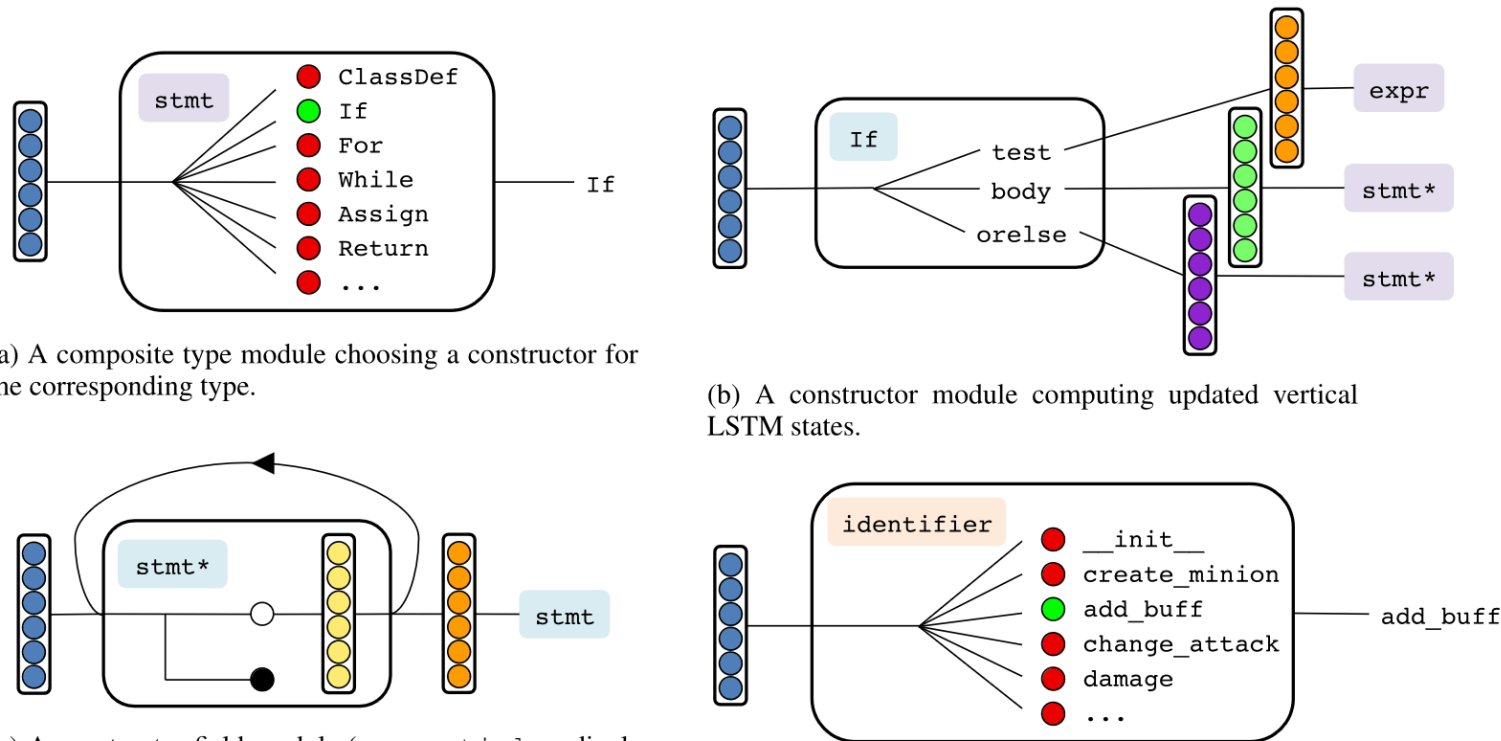
(b)

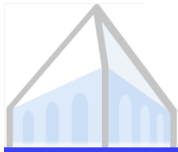
Code: sorted(my_list, reverse=True)



Abstract Syntax Networks

- Generate AST, but learn and use custom decoders (“modules”) for different parts of the grammar

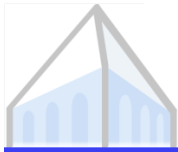




Training at Scale

- With enough training data, modern neural architectures can capture underlying code structure without requiring injection of inductive biases
- It's also easy to generate arbitrary amounts of code for training
- However, provides no guarantees
 - Without explicit copying mechanisms:
 - Possible for the model to learn biases in its vocabulary
 - No guarantees it will properly use new variables and functions
 - Ability to generalize to completely new programming languages and new structures?

General-Purpose Code Generation



Code Generation

- Before: tasks with clear denotations
- What about general-purpose code generation?

Year	City	Country	Nations
1896	Athens	Greece	14
1900	Paris	France	24
1904	St. Louis	USA	12
...
2004	Athens	Greece	201
2008	Beijing	China	204
2012	London	UK	204

$x =$ Greece held its last Summer Olympics in which year?
 $y = 2004$

WikiTableQuestions, Pasupat and Liang 2015, ACL

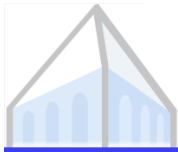
at the chair, move forward three steps past the sofa



Artzi and Zettlemoyer 2013, TACL

There is exactly one black triangle not touching any edge
true

NLVR, Suhr et al. 2017, ACL



Code Generation

- Before: tasks with clear denotations
- What about general-purpose code generation?

```
"""Compute dates for today
and 1 month ago."""
import datetime

today =
datetime.date.today()
one_month_ago = today -
datetime.timedelta(days=30
)

print(today)
print(one_month_ago)
```

OpenAI Codex, 2021



Code Generation

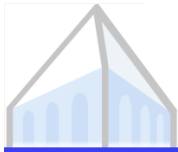
- Before: tasks with clear denotations
- What about general-purpose code generation?
 - Denotation: program output?

```
"""Compute dates for today
and 1 month ago."""
import datetime

today =
datetime.date.today()
one_month_ago = today -
datetime.timedelta(days=30
)

print(today)
print(one_month_ago)
```

OpenAI Codex, 2021



Code Generation

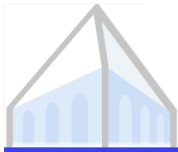
- Before: tasks with clear denotations
- What about general-purpose code generation?
 - Denotation: program output?
 - Less alignment between NL and LF

```
"""Compute dates for today
and 1 month ago."""
import datetime

today =
datetime.date.today()
one_month_ago = today -
datetime.timedelta(days=30
)

print(today)
print(one_month_ago)
```

OpenAI Codex, 2021

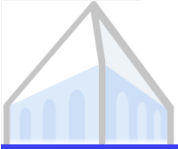


Code Generation

- Before: tasks with clear denotations
- What about general-purpose code generation?
 - Denotation: program output?
 - Less alignment between NL and LF
- What is a “denotation” isn’t always clear...

```
/* Increment the score by 1
point, every 500ms. */
var scoreIncrement =
setInterval(function() {
    score++;
    scoreDisplay.innerHTML =
'Score: ' + score;
}, 500);
```

OpenAI Codex, 2021



Evaluation

- Code doesn't always produce a single, evaluable output
- Instead: write test cases, report pass@k
- Labor-intensive: requires programming expertise for annotation (HumanEval only contains 164 problems)

```
def solution(lst):  
    """Given a non-empty list of integers, return the sum of all of the odd elements  
    that are in even positions.
```

Examples

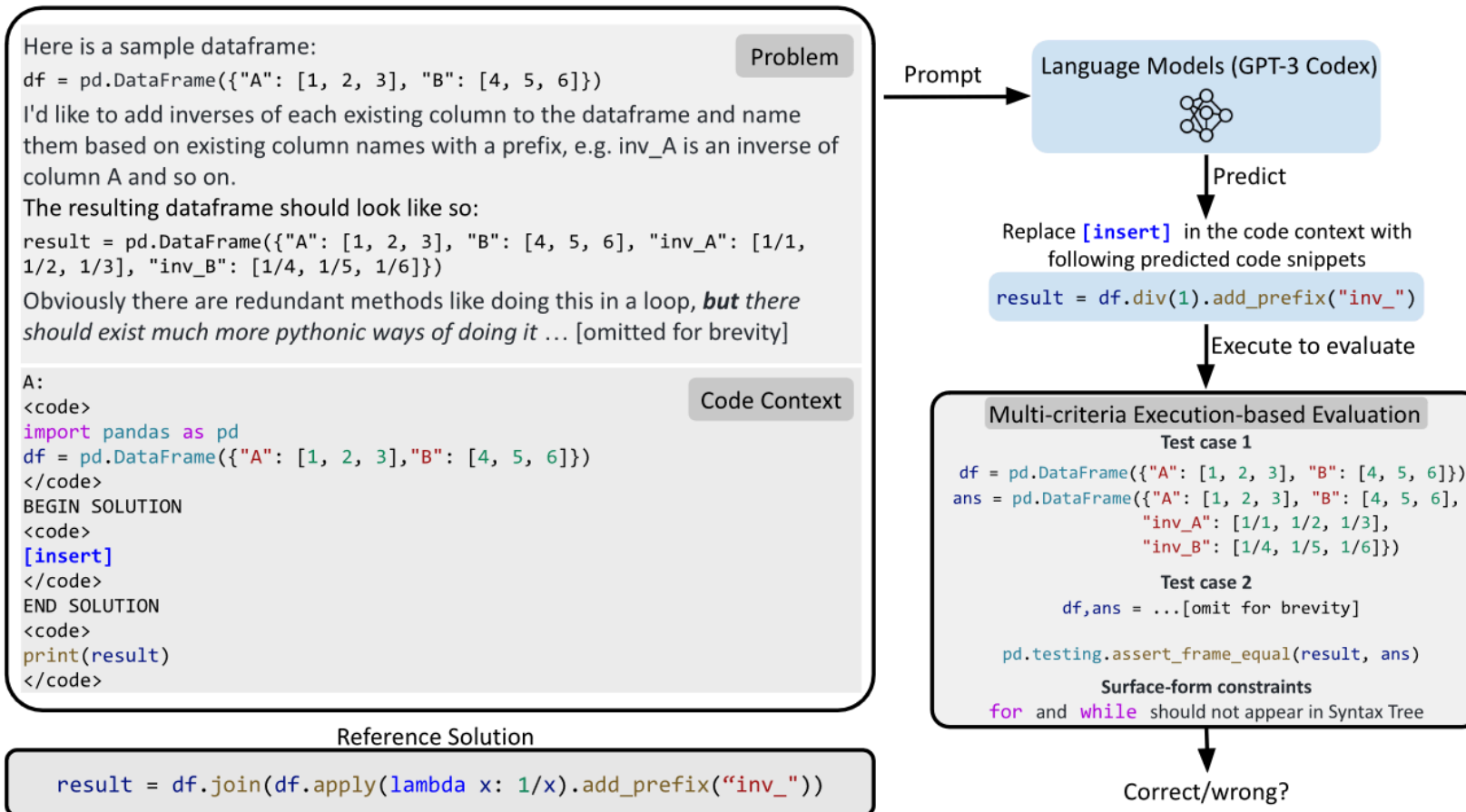
```
solution([5, 8, 7, 1]) ==>12  
solution([3, 3, 3, 3, 3]) ==>9  
solution([30, 13, 24, 321]) ==>0  
"""
```

```
return sum(lst[i] for i in range(0, len(lst)) if i % 2 == 0 and lst[i] % 2 == 1)
```

HumanEval, Chen et al. 2021

Evaluation

- Any automated benchmark has to focus on a subset of problems

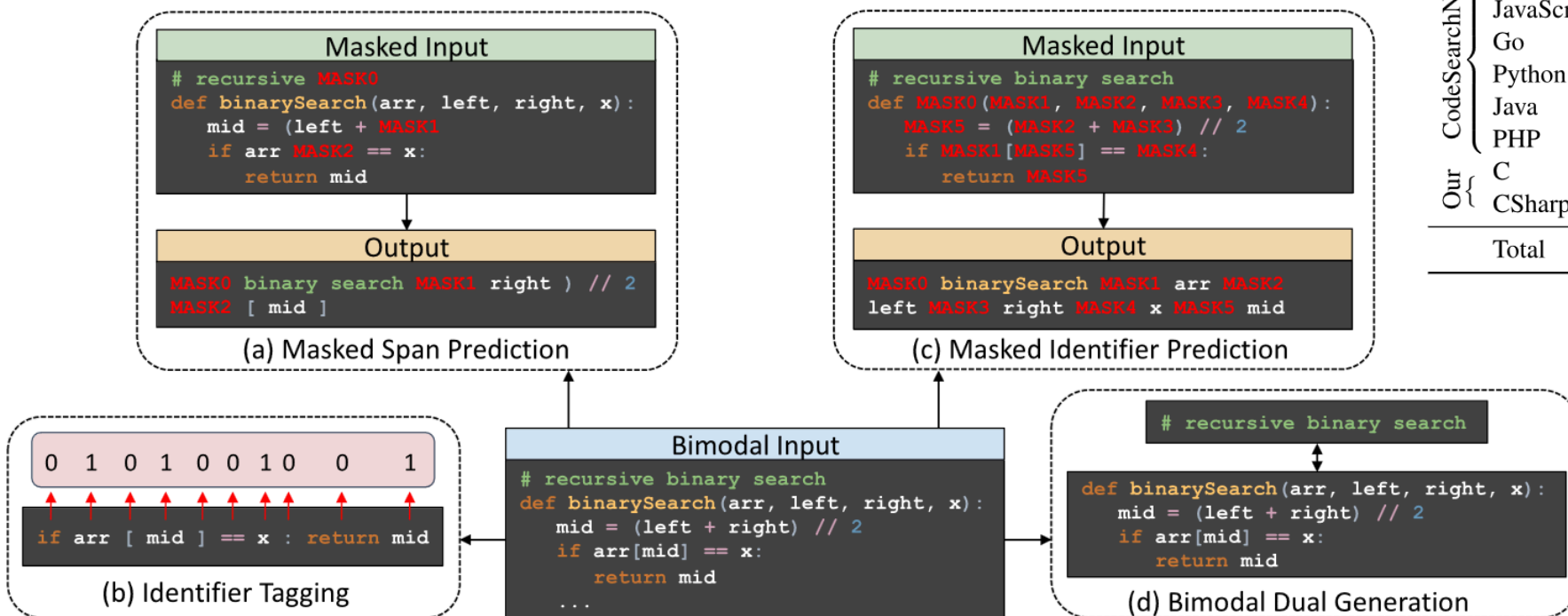


[DS-1000, Lai et al. 2022](#)

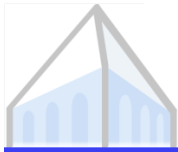
- Sample real problems from StackOverflow
- Collect reference solutions and setting up environment for testing
- Expert-written test cases
- Evaluate adherence to surface form constraints (e.g., that a library must be used)

Approaches

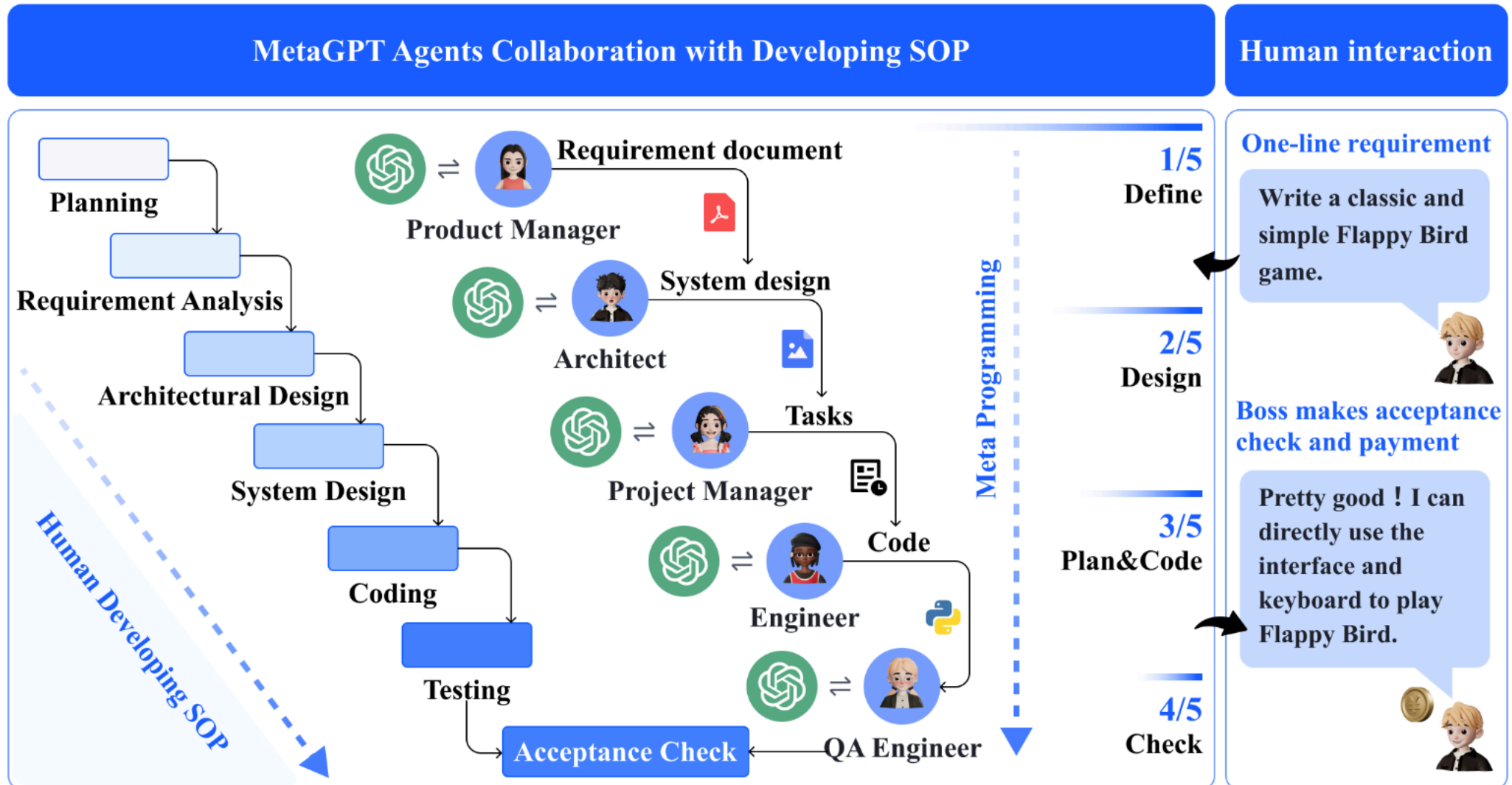
- Multi-task learning: masking, tagging, generation
- Train on a large amount of code, some annotated with natural language



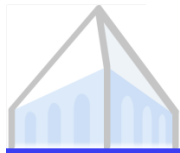
PLs	W/ NL	W/o NL	Identifier
CodeSearchNet { Ruby	49,009	110,551	32.08%
JavaScript	125,166	1,717,933	19.82%
Go	319,132	379,103	19.32%
Python	453,772	657,030	30.02%
Java	457,381	1,070,271	25.76%
PHP	525,357	398,058	23.44%
Our { C	1M	-	24.94%
CSharp	228,496	856,375	27.85%
Total	3,158,313	5,189,321	8,347,634



Automated Software Development?



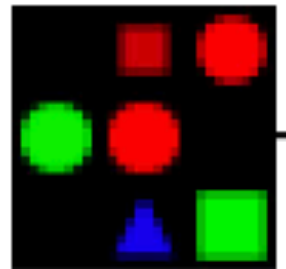
Modularity

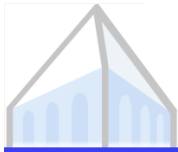


Neural Module Networks

- Task: visual question answering
 - Cast it as a semantic parsing task
 - What is a denotation?
- Tie predicates in LF to composable neural modules


*“Is there a red shape
above a circle?”*





Neural Module Networks

- Determine layout from sentence
 - Option 1: deterministic layouts — requires gold annotation
 - Get dependency parse for input question
 - Construct layout of modules given this parse
 - Option 2: latent layouts — requires RL
- Compose modules and run inference / training (end-to-end)



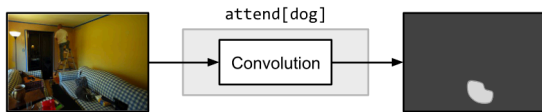
how many different lights in various different shapes and sizes?

```
measure[count](
  attend[light])
```

four (four)

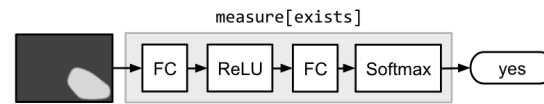
Attention

$attend : Image \rightarrow Attention$



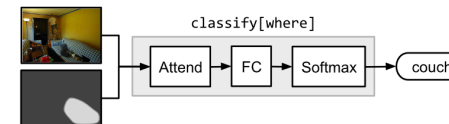
Measurement

$measure : Attention \rightarrow Label$



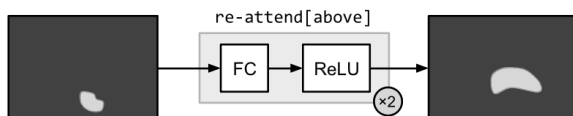
Classification

$classify : Image \times Attention \rightarrow Label$



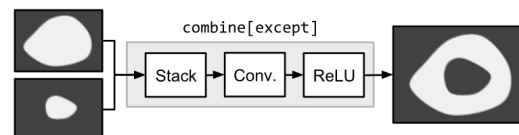
Re-attention

$re-attend : Attention \rightarrow Attention$



Combination

$combine : Attention \times Attention \rightarrow Attention$





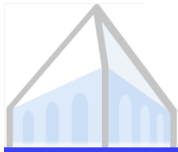
Neural Module Networks

- Benefits: interpretability and controllability
 - You know what modules are being used
 - You know how they are composed
 - You know the intermediate outputs of each module
- Problems
 - Requires formalizing the set of modules
 - Doesn't work very well, empirically

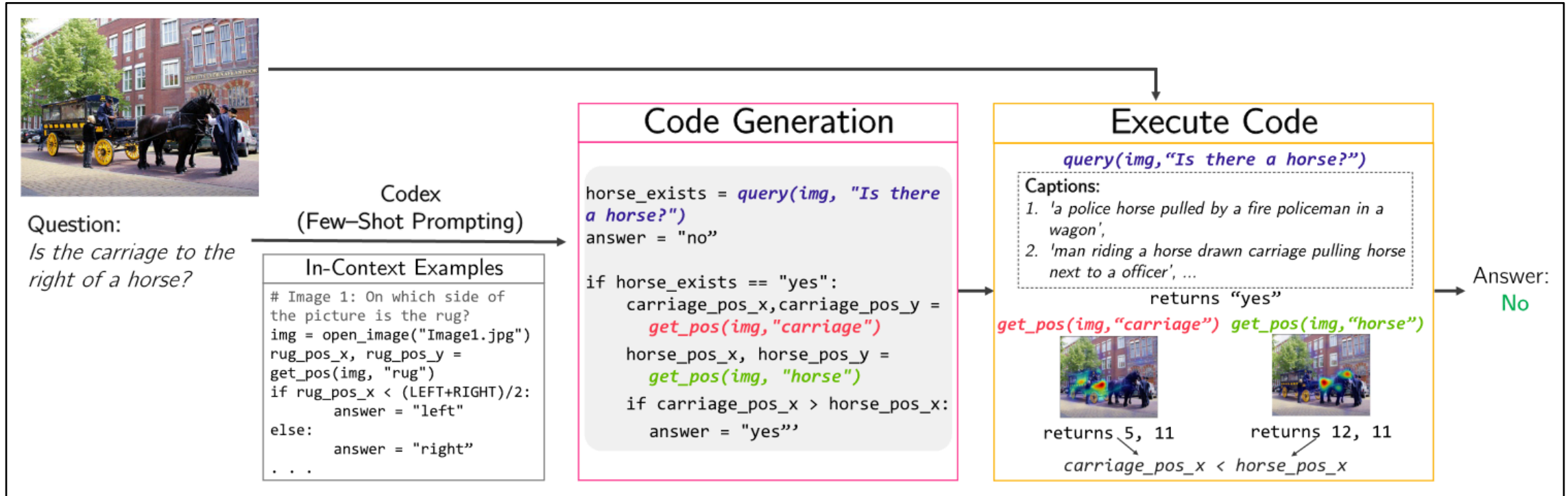


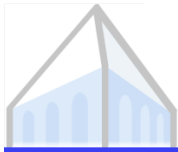
Code as a Reasoning Bottleneck

- Taking advantage of general-purpose code models
- Formal representation is given to us!
- Need very little paired data (use in-context learning)
- Still interpretable and controllable
- Some drawbacks:
 - Still requires choosing a few modules
 - Particular choice of in-context examples and modules can limit reasoning




Code as a Reasoning Bottleneck






Code as a Reasoning Bottleneck

IMAGE:



Prediction: IMAGE0



Instruction: Replace desert with lush green grass

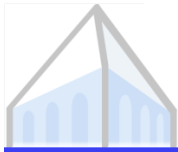
Program:

```
OBJ0=Seg(image=IMAGE)
OBJ1=Select(image=IMAGE, object=OBJ0, query='desert', category=None)
IMAGE0=Replace(image=IMAGE, object=OBJ1, prompt='lush green grass')
RESULT=IMAGE0
```

VisProg, Gupta and Kembhavi 2023, CVPR

Subramanian et al. 2023, ACL

ViperGPT, Surís et al. 2023



Code as a Reasoning Bottleneck

Query: Return the two kids that are furthest from the woman right before she hugs the girl



```
def execute_command(video):
    video_segment = VideoSegment(video)
    hug_detected = False
    for i, frame in enumerate(video_segment.frame_iterator()):
        if frame.exists("woman") and frame.exists("girl") and \
            frame.simple_query("Is the woman hugging the girl?") == "yes":
            hug_detected = True
            break
    if hug_detected:
        index_frame = i - 1
        frame_of_interest = ImagePatch(video_segment, index_frame)
        woman_patches = frame_of_interest.find("woman")
        woman_patch = woman_patches[0]
        kid_patches = frame_of_interest.find("kid")
        kid_patches.sort(key=lambda kid: distance(kid, woman_patch))
        kid_patch_1 = kid_patches[-1]
        kid_patch_2 = kid_patches[-2]
    return [kid_patch_1, kid_patch_2]
```

