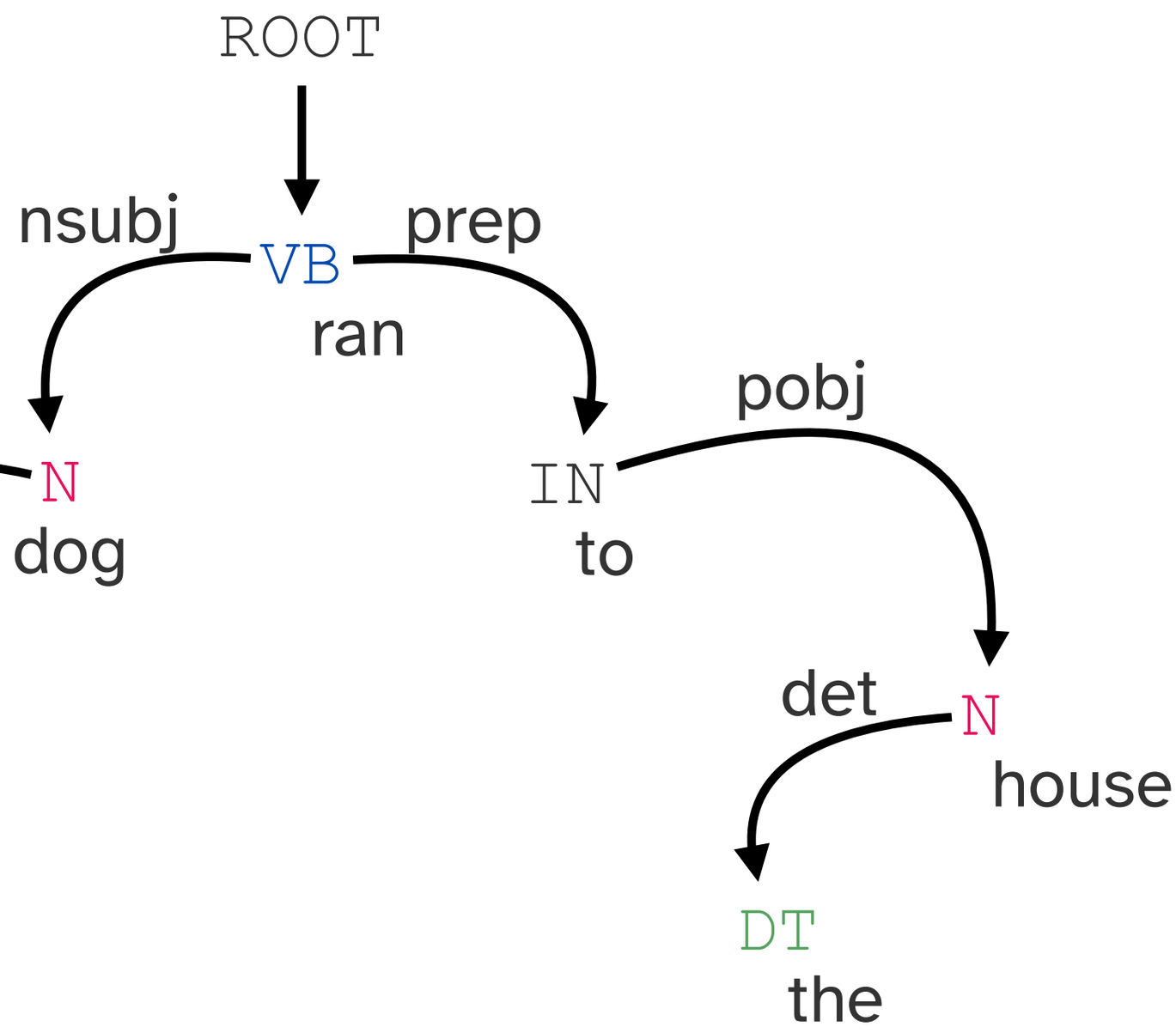
Linguistics: Compositional Semantics

EECS 183/283a: Natural Language Processing

Recap: Dependency Parsing

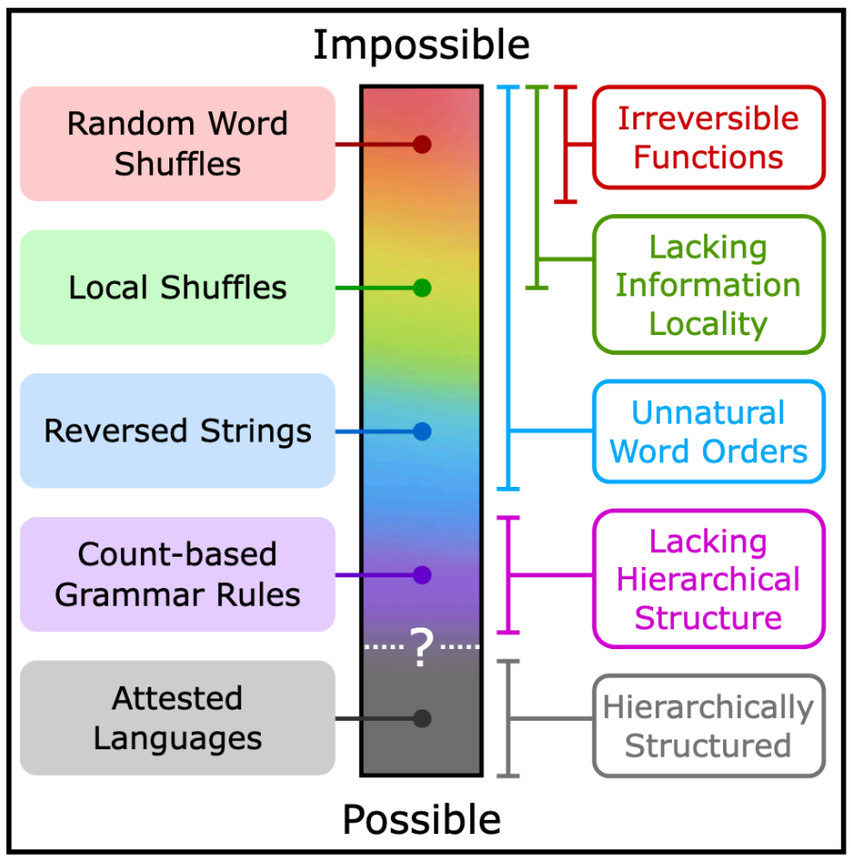
the dog ran to the house

Recap: Dependency Parsing



Why is Syntax Still Relevant?

Learnability of humanlike syntactic structure
(formal expressivity of architectures, or learnability from data)



What Formal Languages Can Transformers Express? A Survey

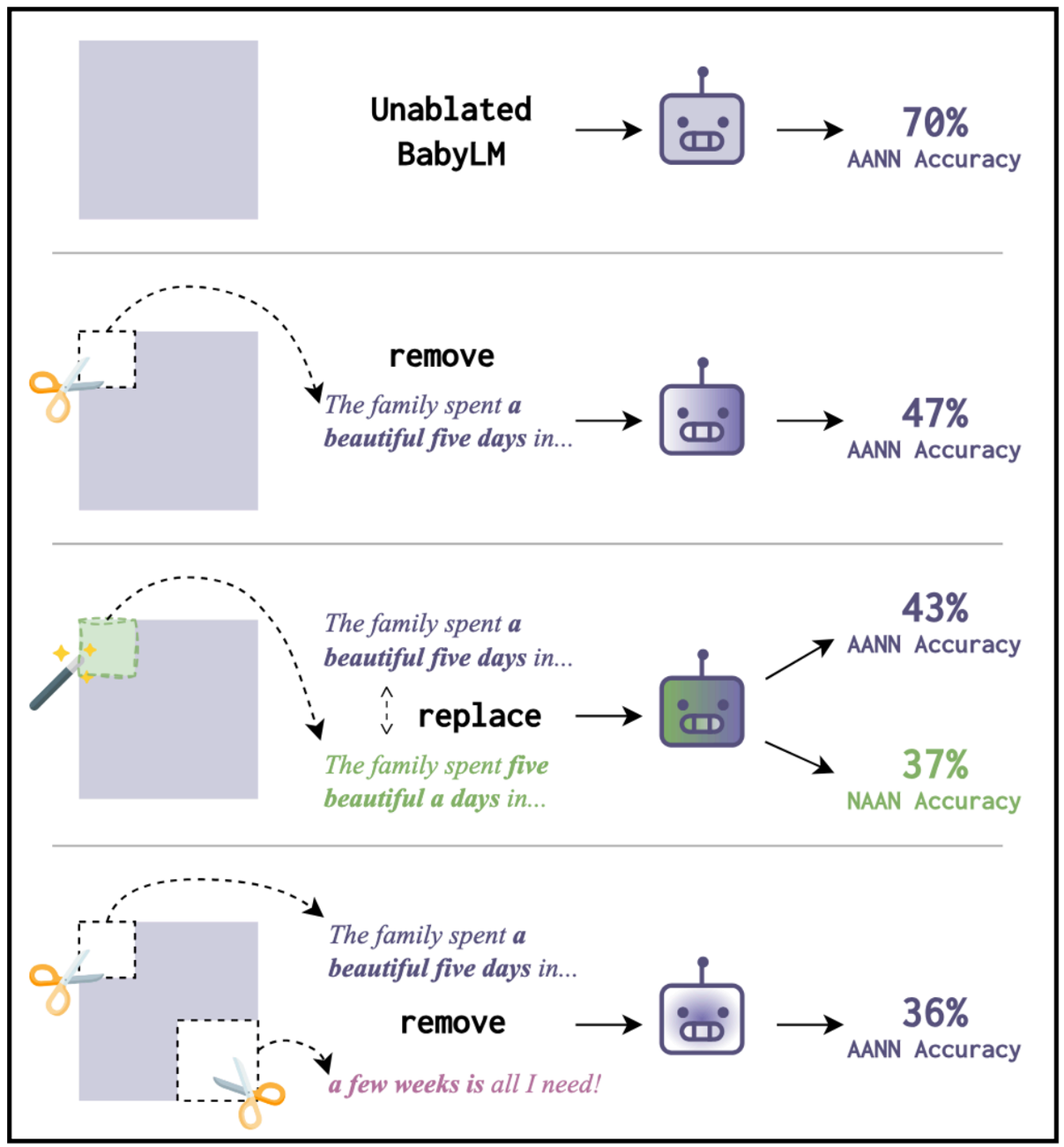
Lena Strobl
Umeå University, Sweden
lena.strobl@umu.se

William Merrill
New York University, USA
willm@nyu.edu

Gail Weiss
EPFL, Switzerland
gail.weiss@epfl.ch

David Chiang
University of Notre Dame, USA
dchiang@nd.edu

Dana Angluin
Yale University, USA
dana.angluin@yale.edu

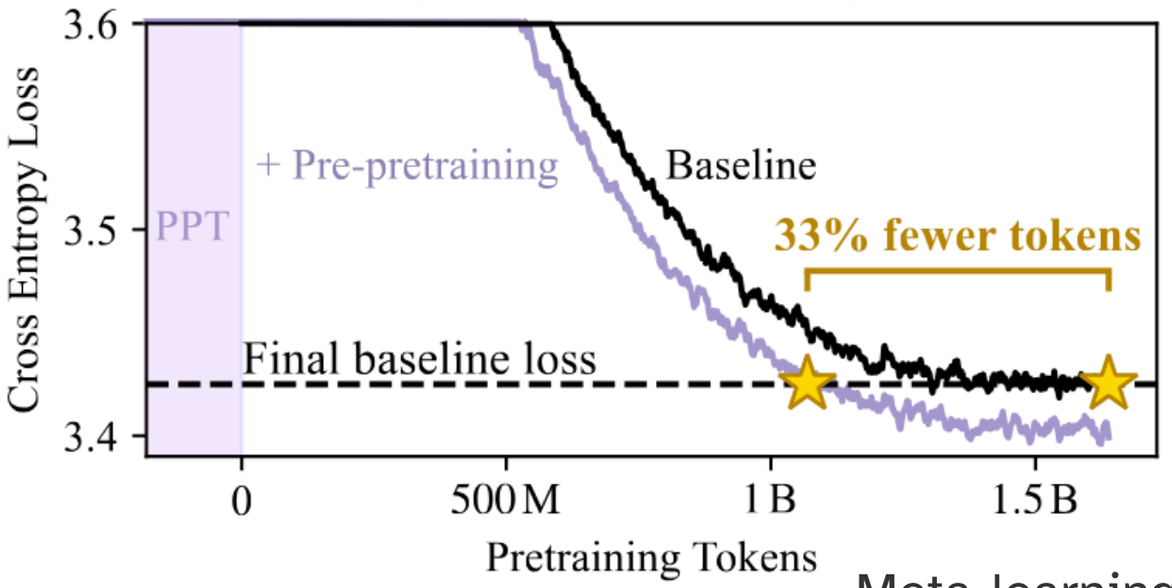


Kallini et al. 2024, Strobl et al. 2024, Hu et al. 2024, Misra and Mahowald 2025

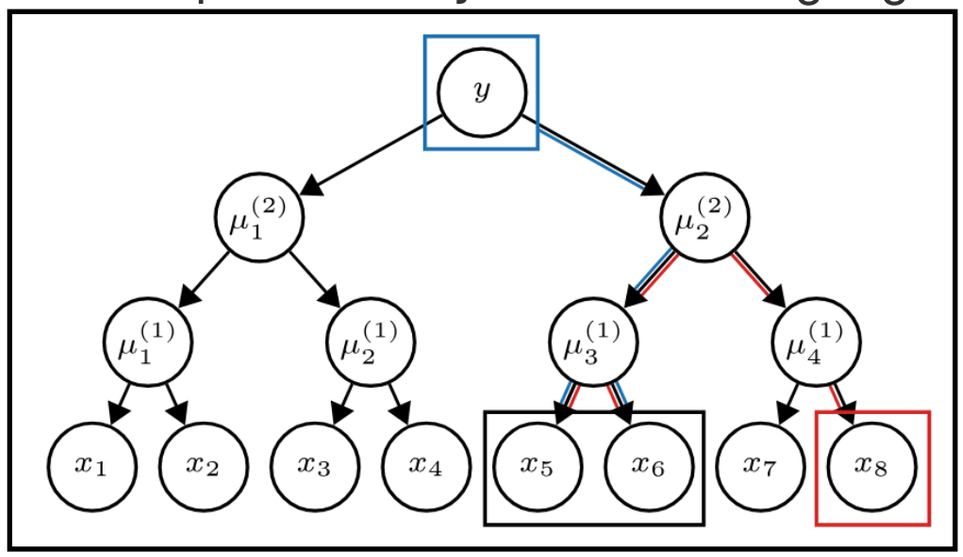
Why is Syntax Still Relevant?

Serves as a good inductive bias in low-data settings

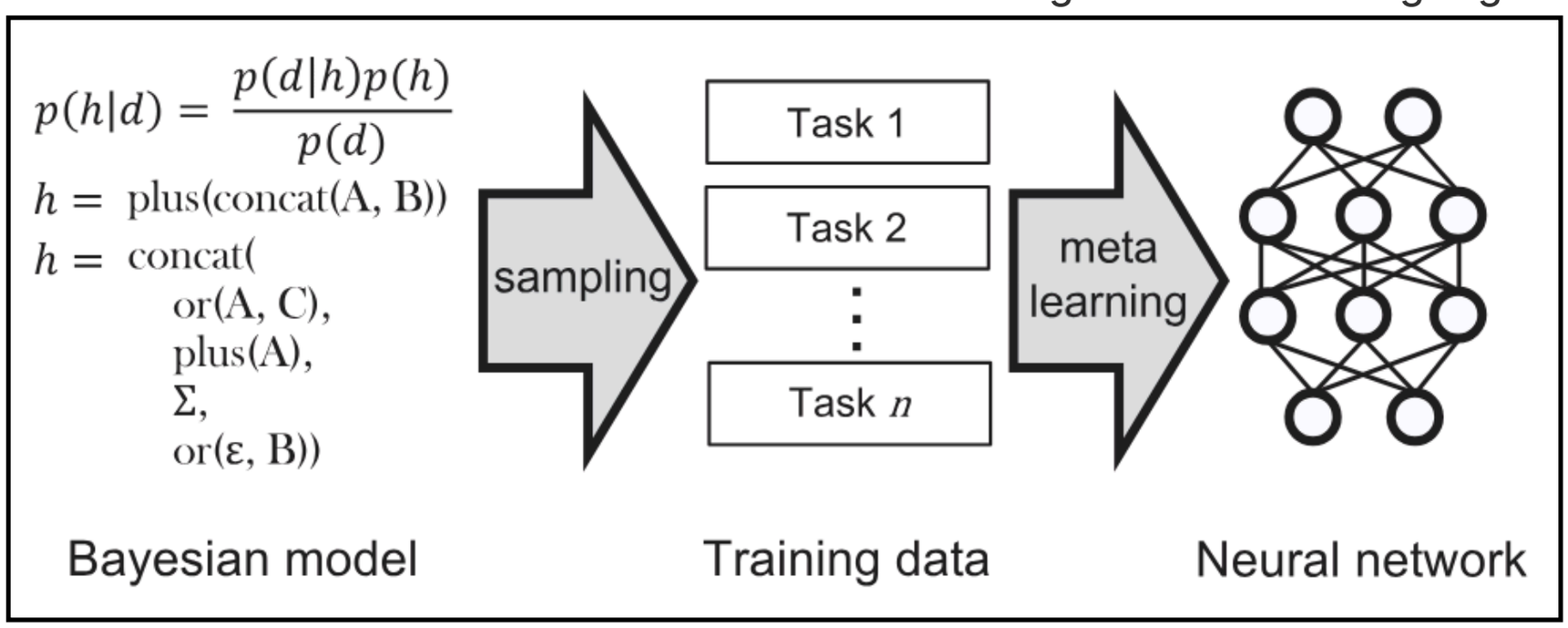
Pre-training on formal languages



Relationship between scaling laws and hierarchical compositionality of human language



Meta-learning with formal languages



Hu et al. 2025, McCoy and Griffiths 2025, Cagnetta et al. 2025

Compositional Semantics

- How do we represent sentence meaning?
- How can we get from word meaning to sentence meaning?
- What are some applications of formal semantics in NLP?

Compositional Semantics

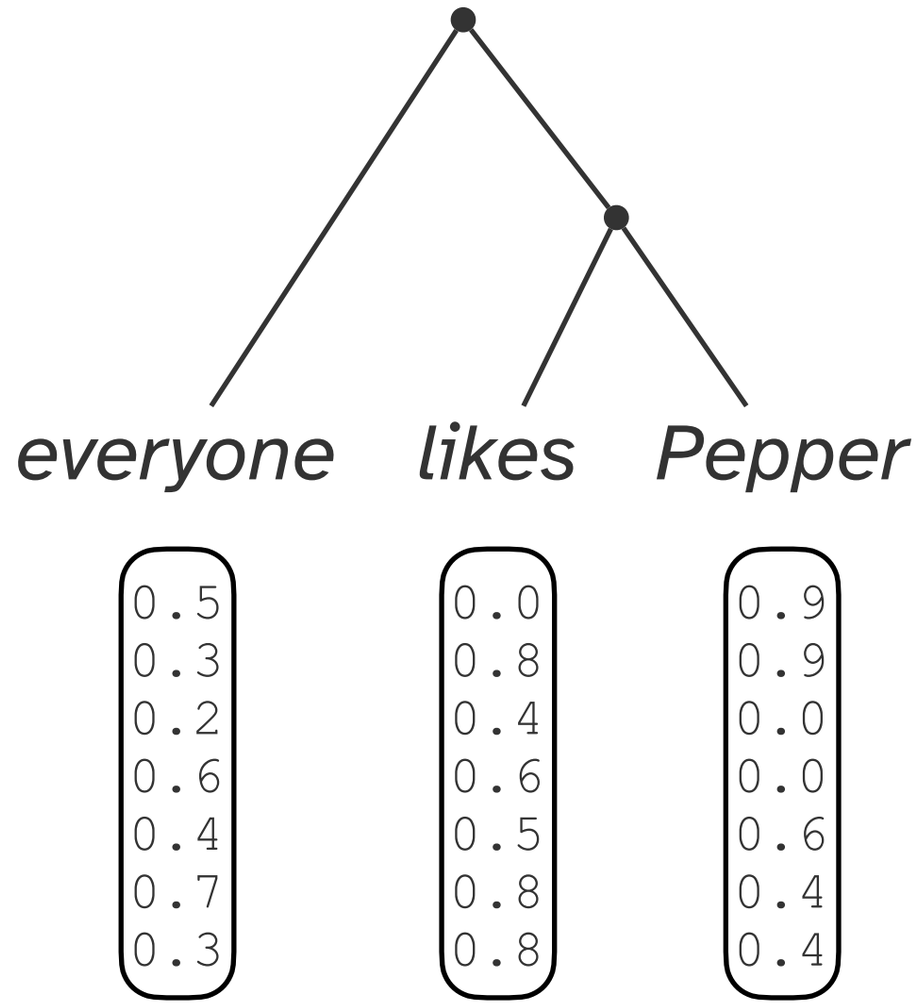
- **Lexical semantics:** we can get word meanings
  - Denotational semantics: tokens are references to things in the real world
  - Ontologies: tokens are references to nodes in some knowledge graph
  - Word embeddings: tokens are represented by continuous vectors

everyone	*likes*	*Pepper*
0.5	0.0	0.9
0.3	0.8	0.9
0.2	0.4	0.0
0.6	0.6	0.0
0.4	0.5	0.6
0.7	0.8	0.4
0.3	0.8	0.4

Compositional Semantics

- **Lexical semantics:** we can get word meanings
  - Denotational semantics: tokens are references to things in the real world
  - Ontologies: tokens are references to nodes in some knowledge graph
  - Word embeddings: tokens are represented by continuous vectors
- **Syntax:** we can determine what sequences of word types are possible or not possible in a language by modeling latent structure
  - Constituency grammar aka phrase structure grammar aka context-free grammar
  - Dependency grammar

Main challenge of <u>semantic</u> parsing: how do we get a single representation of the <u>entire</u> sentence's meaning from (a) the meanings of its words, and (b) their order and latent structure?



Recap: Combinatory Categorial Grammar (CCG)

- Another way of representing a constituency grammar: bottom-up

- Elements of a CCG:
  - Lexical items (wordtypes)
  - Each paired with a syntactic type (≈ nonterminal or composition thereof)

the : NP/N	dog : N	John : NP	bit : $(S\backslash NP)/NP$

the	dog	bit	John

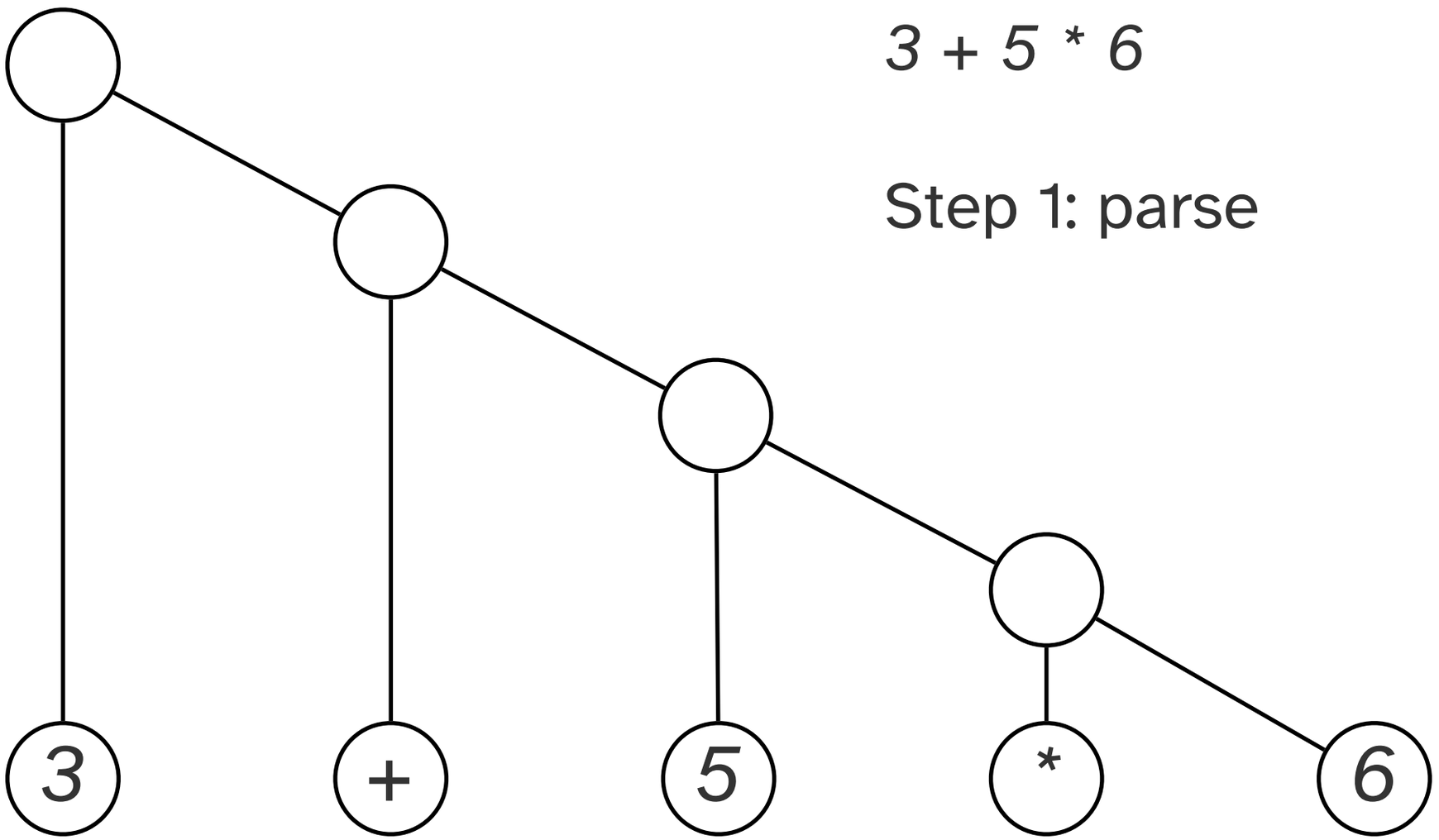
Example from Wikipedia

An Analogy: Code Interpreters

Task: evaluate the expression:

*3 + 5 * 6*

An Analogy: Code Interpreters

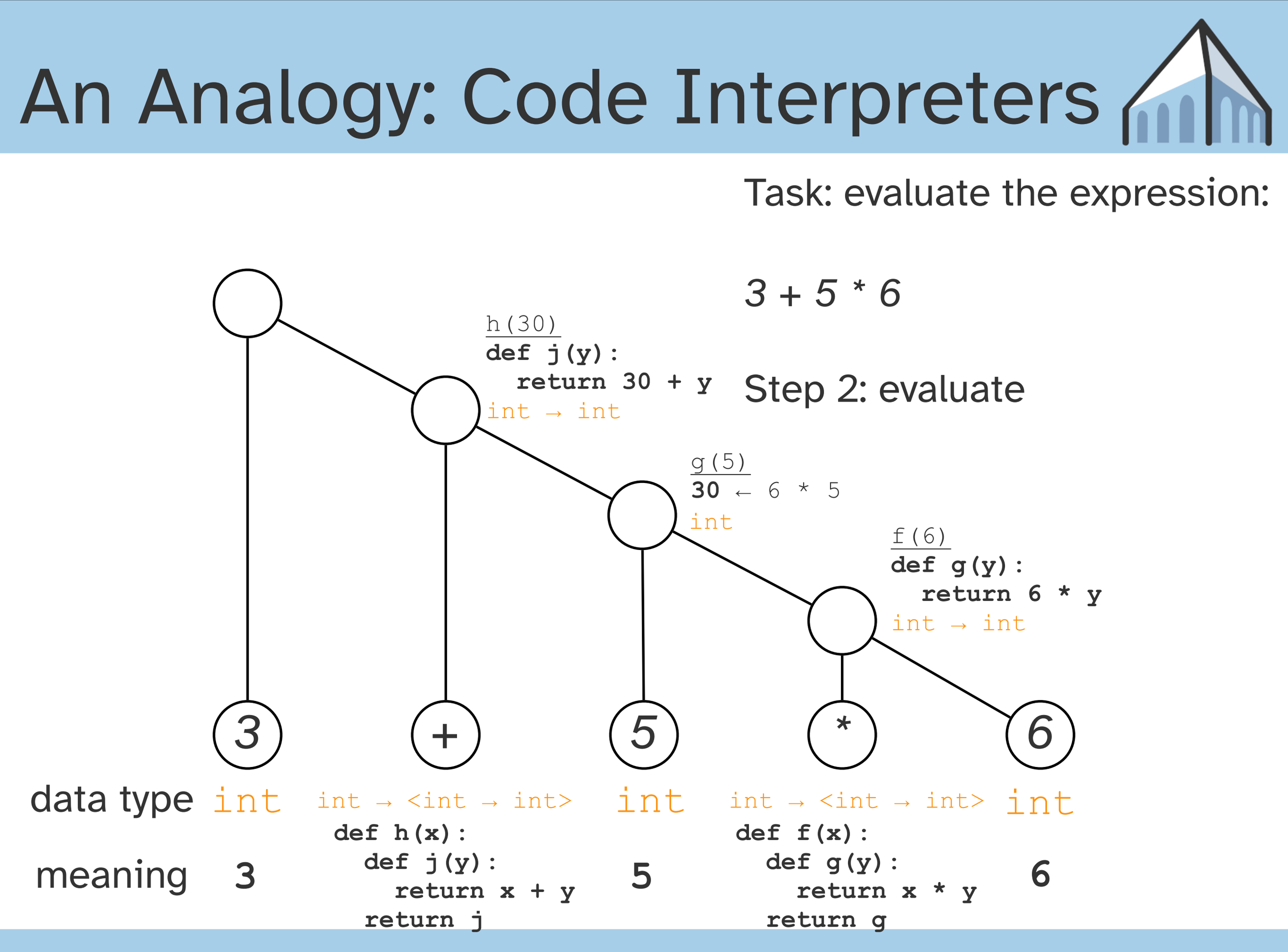


Task: evaluate the expression:

*3 + 5 * 6*

Step 1: parse

An Analogy: Code Interpreters



Task: evaluate the expression:

*3 + 5 * 6*

Step 2: evaluate

```
h(30)
def j(y):
  return 30 + y
int → int
```

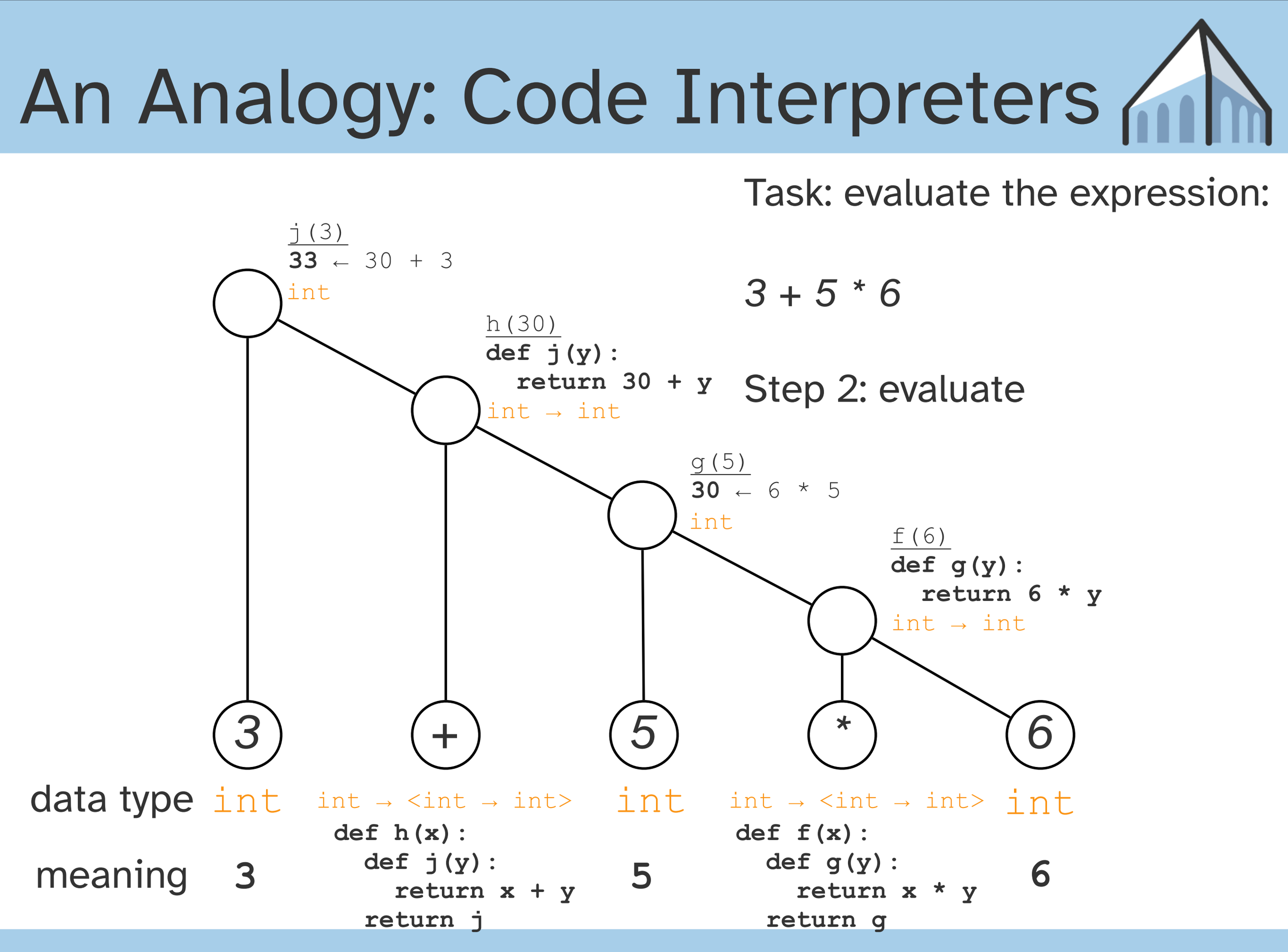
```
g(5)
30 ← 6 * 5
int
```

```
f(6)
def g(y):
  return 6 * y
int → int
```

	3	+	5	*	6
data type	int	int → <int → int>	int	int → <int → int>	int

meaning	3	def h(x):
		  def j(y):
		  return x + y
		  return j
			5	def f(x):
				  def g(y):
				  return x * y
				  return g
					6

An Analogy: Code Interpreters



Task: evaluate the expression:

*3 + 5 * 6*

Step 2: evaluate

```
j(3)
33 ← 30 + 3
int
```

```
h(30)
def j(y):
  return 30 + y
int → int
```

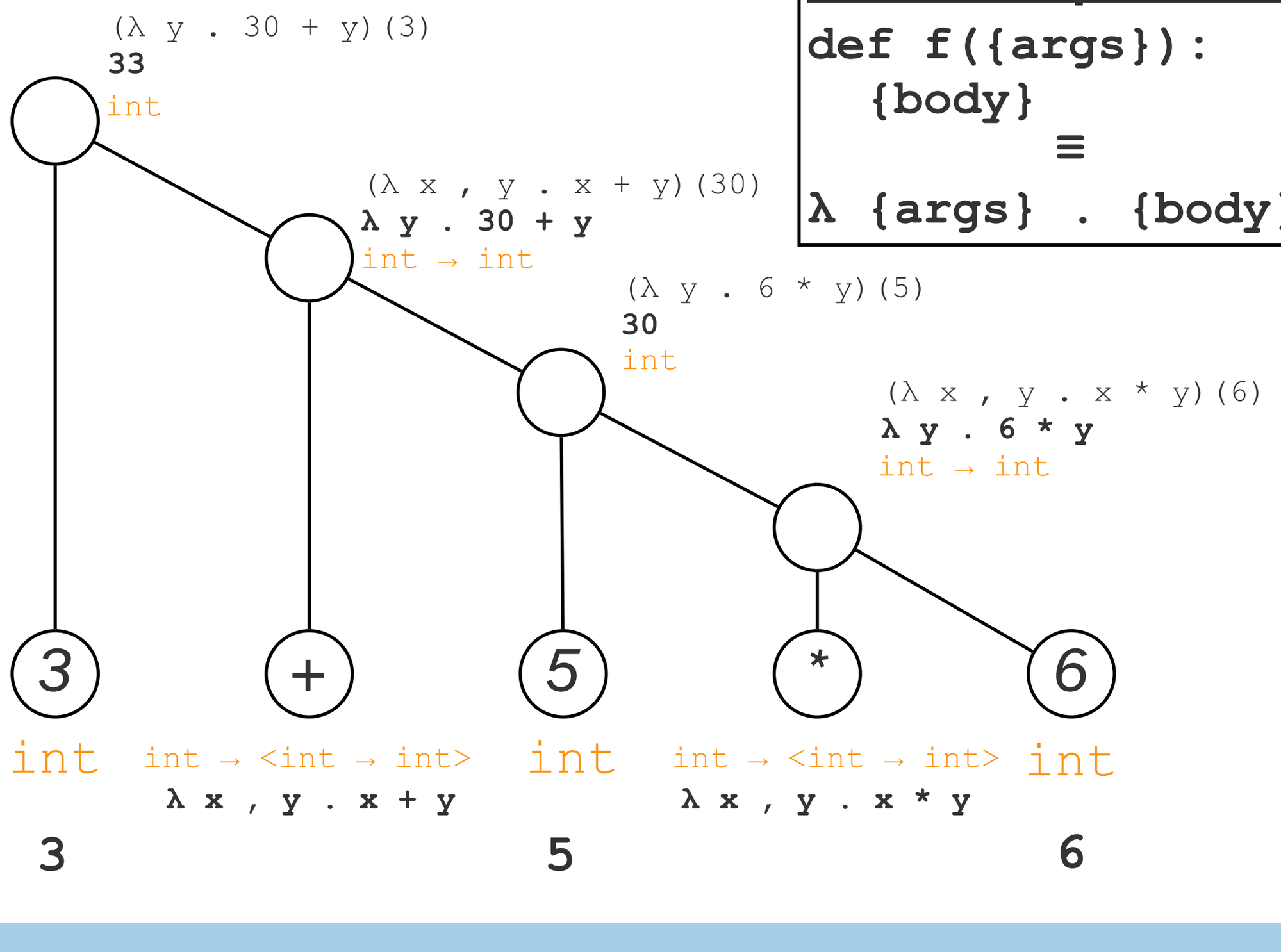
```
g(5)
30 ← 6 * 5
int
```

```
f(6)
def g(y):
  return 6 * y
int → int
```

data type	`int`	`int → <int → int>`	`int`	`int → <int → int>`	`int`

meaning	3	`def h(x):`
		`  def j(y):`
		`  return x + y`
		`  return j`
			5	`def f(x):`
				`  def g(y):`
				`  return x * y`
				`  return g`
					6

CCG and Lambda Calculus



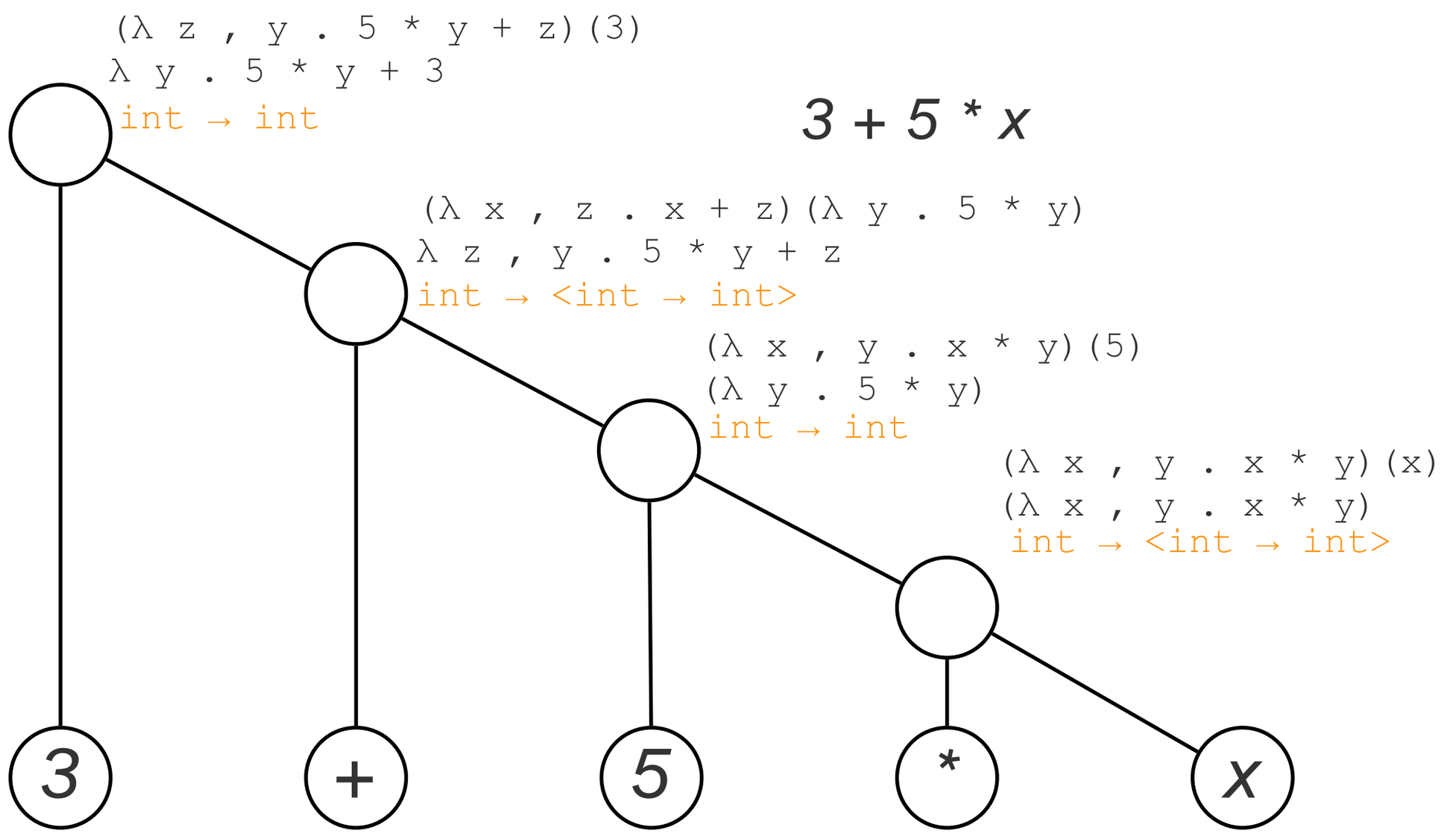
Lambda expressions:

```
def f({args}):
   {body}
          ≡
λ {args} . {body}
```

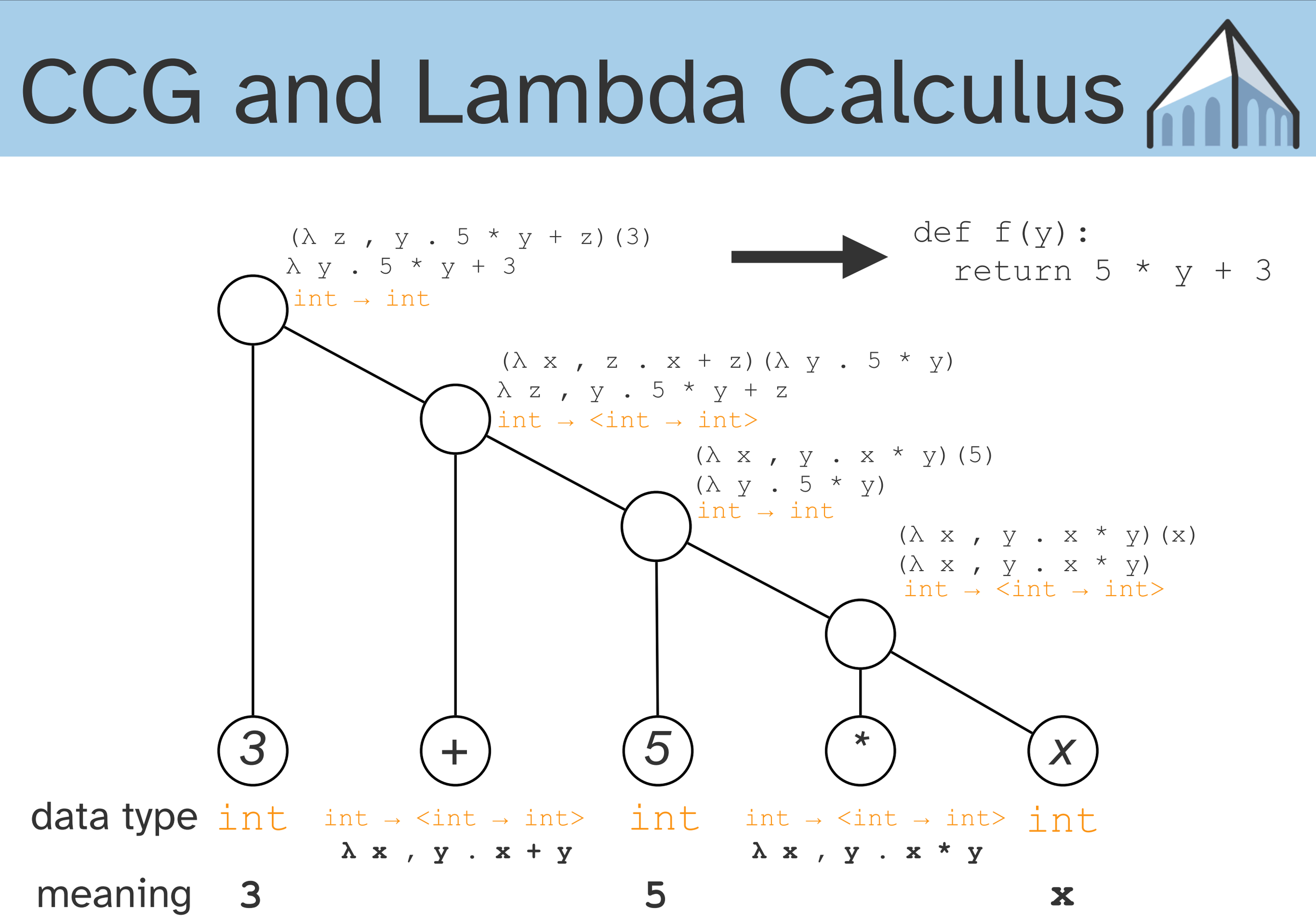
CCG and Lambda Calculus

Task: evaluate the expression:

$$3 + 5 * x$$



CCG and Lambda Calculus



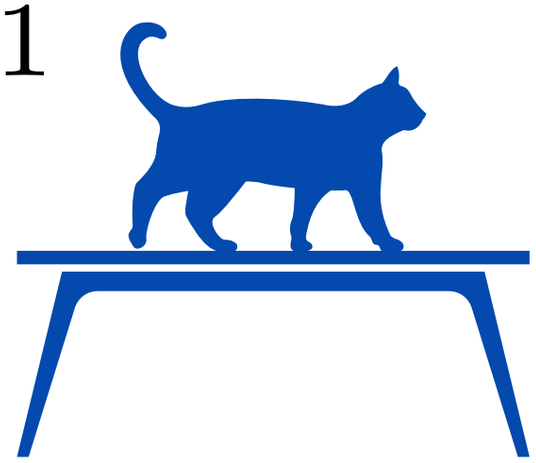
Truth-Conditional Semantics

- In the context of their use, statements are either true or false, i.e., they have the type *t* (aka, `bool` in python)

- Let's call this context a world *w*

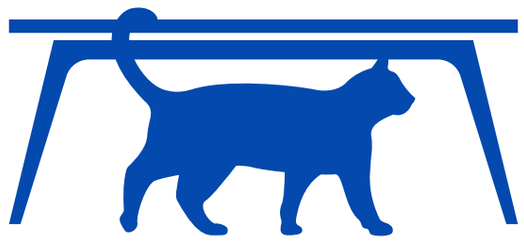
- We'd like the outcome of our semantic parsing to be a some that can evaluate to true or false (i.e., $\mathcal{W} \rightarrow [0, 1]$)

w_1



$[\![\text{the cat is on the table}]\!]^{w_1}$

w_2

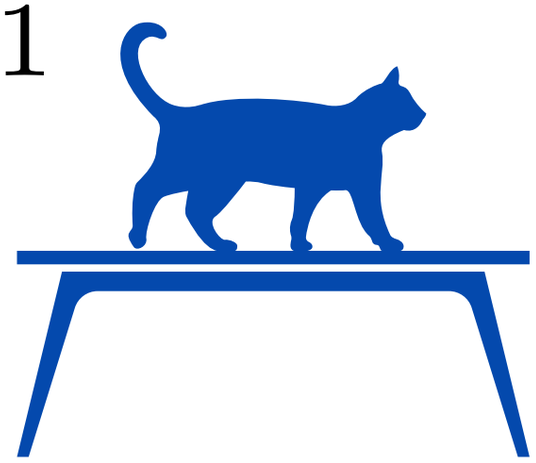


$[\![\text{the cat is on the table}]\!]^{w_2}$

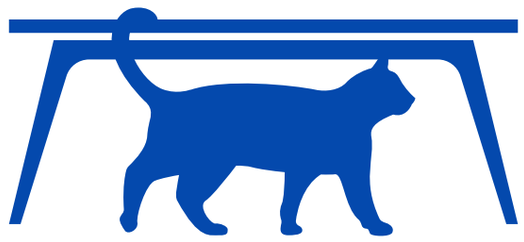
Truth-Conditional Semantics

- In the context of their use, statements are either true or false, i.e., they have the type *t* (aka, `bool` in python)
- Let's call this context a world *w*
- We'd like the outcome of our semantic parsing to be a some that can evaluate to true or false (i.e., $\mathcal{W} \rightarrow [0, 1]$)

w_1



w_2



the cat is on the table —CCG→ some function that can be evaluated to give us the denotation in an arbitrary world

Lambda Calculus for Natural Language

CCG for semantic parsing:

- Lexical items (wordtypes)
- Each paired with a syntactic type
- **And** paired with a lambda expression and its semantic type

	everyone	***likes***	***Pepper***
Syntactic type			
Semantic type			
λ-expression			

Example Parse

everyone likes Pepper

Example Parse

`Pepper`

Syntactic type: noun phrase

Semantic type: entity

λ-expression: refers to the unique `Pepper` entity

	everyone	*likes*	*Pepper*
Syntactic type			`NP`
Semantic type			`e`
λ-expression			`Pepper`

Example Parse

everyone likes Pepper

NP

e

Pepper

	everyone	***likes***	***Pepper***
Syntactic type			NP
Semantic type			e
λ-expression			Pepper

Example Parse

`λ x, y . likes (y, x)`

Syntactic type: expects a noun phrase to follow, and a noun phrase to precede

Semantic type: expects an entity as its first argument, produces a new function

λ-expression: calls the primitive `likes` function that checks if *y* likes *x*

	everyone	***likes***	***Pepper***
Syntactic type		(S \ NP) / NP	NP
Semantic type		e → <e → t>	e
λ-expression		λ x, y . likes (y, x)	Pepper

Example Parse

everyone

likes

(S \ NP) / NP

e → < e → t >

λ x, y . likes (y, x)

Pepper

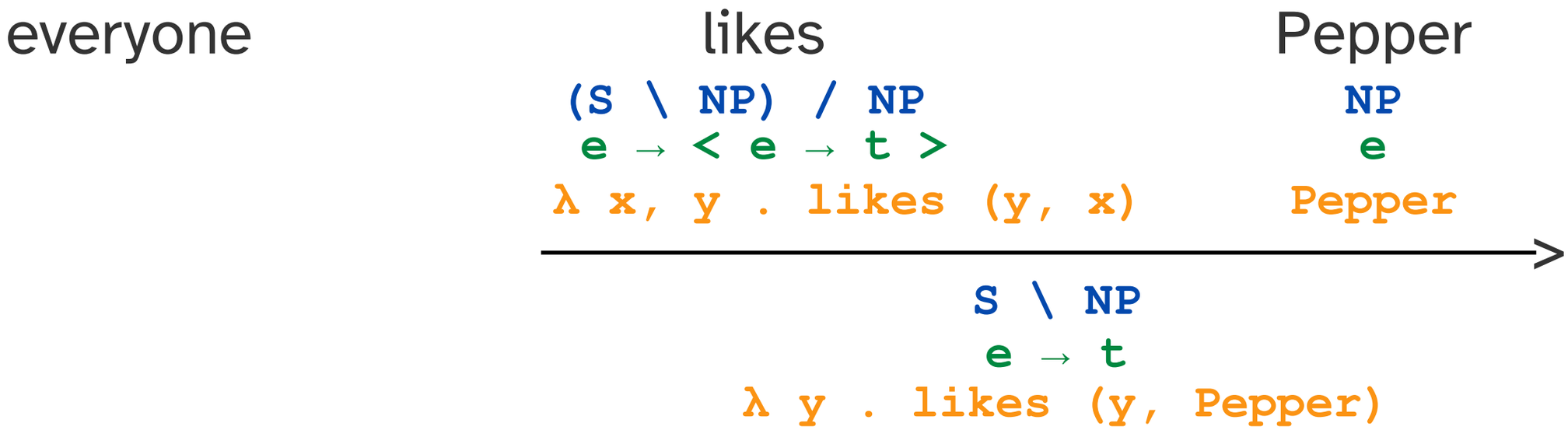
NP

e

Pepper

	everyone	*likes*	*Pepper*
Syntactic type		(S \ NP) / NP	NP
Semantic type		e → <e → t>	e
λ-expression		λ x, y . likes (y, x)	Pepper

Example Parse



Example Parse

λ f . ∀ x (person(x) → f(x))

λ-expression: checks whether, for all people, the function *f* is true

	everyone	***likes***	***Pepper***
Syntactic type	S / (S \ NP)	(S \ NP) / NP	NP
Semantic type	<e → t> → t	e → <e → t>	e
λ-expression	λ f . ∀ x (person(x) → f(x))	λ x, y . likes (y, x)	Pepper

Example Parse

everyone
S / (S \ NP)
< e → t > → t
λ f . ∀ x
(person(x) → f(x))

likes
(S \ NP) / NP
e → < e → t >
λ x, y . likes (y, x)

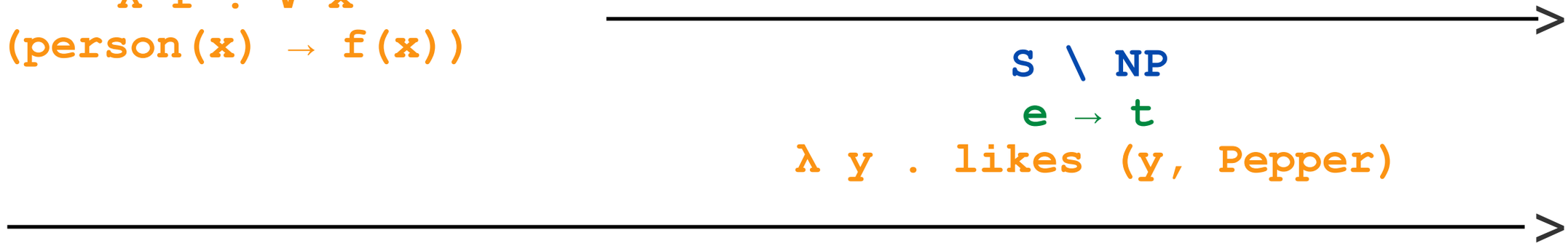
Pepper
NP
e
Pepper

S \ NP
e → t
λ y . likes (y, Pepper)

	everyone	*likes*	*Pepper*
Syntactic type	S / (S \ NP)	(S \ NP) / NP	NP
Semantic type	<e → t> → t	e → <e → t>	e
λ-expression	λ f . ∀ x (person(x) → f(x))	λ x, y . likes (y, x)	Pepper

Example Parse

everyone	likes	Pepper
S / (S \ NP)	(S \ NP) / NP	NP
< e → t > → t	e → < e → t >	e
λ f . ∀ x (person(x) → f(x))	λ x, y . likes (y, x)	Pepper



S \ NP
e → t
λ y . likes (y, Pepper)

S
t

λ f . ∀ x (person(x) → f(x)) (λ y . likes (y, Pepper))

∀ x (person(x) → (λ y . likes (y, Pepper))(x)

∀ x (person(x) → likes (x, Pepper))

Sentence Meaning

$$\forall\ x\ (person(x) \rightarrow likes\ (x, Pepper))$$

- What can we do with our sentence now that it's a function?
- We can check its meaning against some world!

entities

ID	Name	Person?
1	Alane	yes
2	Gopala	yes
…	…	…
N	Pepper	no

likes

ID 1	ID 2
1	N
2	N
…	…
N	1

Sentence Meaning

$$\forall x\ (person(x) \rightarrow likes\ (x, Pepper))$$

- What can we do with our sentence now that it's a function?
- We can check its meaning against some world!
- We can check use it to make inferences!

$$person(Alane)$$
$$\wedge$$
$$\forall x\ (person(x) \rightarrow likes\ (x, Pepper))$$

$$\Rightarrow likes(Alane, Pepper)$$

Formal Semantics

- Logical operators, like ∨, ∧, and ¬
  *Pepper is clever **and** curious*
- Quantifiers like ∀ and ∃
  ***Some** cats like water*
- Relationships between functions ⇒ and ⇔

  Squares are rectangles (∀`x(square(x) ⇒ rectangle(x))`)
- Verbs can have tenses, and can be modified with adverbs
- We can talk about beliefs others have
- Some combinations of meanings are nonsensical (unevaluable)
  green ideas
- Sentences aren't just statements — sometimes they are commands, questions, etc.
- Sentences exist in the context of previous sentences and their meanings

Modern Formal Semantics

- In NLP, nobody is really mapping from sentences to lambda calculus representations anymore

- However, many of our problems still take the form of mapping from language to some meaningful structured representation

Modern Formal Semantics

Planning (e.g., task specification to PDDL)

```
[DOMAIN]
(define (domain blocksworld-4ops)
  (:requirements :strips)
(:predicates (clear ?x)
             (ontable ?x)
             (handempty)
             (holding ?x)
             (on ?x ?y))

(:action pick-up
  :parameters (?ob)
  :precondition (and (clear ?ob) (ontable ?ob) (handempty))
  :effect (and (holding ?ob) (not (clear ?ob)) (not (ontable ?ob))
               (not (handempty))))

(:action put-down
  :parameters (?ob)
  :precondition (holding ?ob)
  :effect (and (clear ?ob) (handempty) (ontable ?ob)
               (not (holding ?ob))))

(:action stack
  :parameters (?ob ?underob)
  :precondition (and (clear ?underob) (holding ?ob))
  :effect (and (handempty) (clear ?ob) (on ?ob ?underob)
               (not (clear ?underob)) (not (holding ?ob))))

(:action unstack
  :parameters (?ob ?underob)
  :precondition (and (on ?ob ?underob) (clear ?ob) (handempty))
  :effect (and (holding ?ob) (clear ?underob)
               (not (on ?ob ?underob)) (not (clear ?ob)) (not (handempty)))))
```

the table has six blocks on it, arranged into a tower that, from bottom to top, has the following blocks: a, c, e, f, b, and d. rearrange the tower so that their order is, from bottom to top, e, f, a, c, b, d.

```
[QUERY PROBLEM]
(define(problem BW-rand-6)
(:domain blocksworld-4ops)
(:objects a b c d e f )
(:init
(handempty)
(ontable a)
(on b f)
(on c a)
(on d b)
(on e c)
(on f e)
(clear d)
)
(:goal
(and
(on a f)
(on b c)
(on c a)
(on d b)
(on f e))
)
)
)
```

Example from Valmeekam et al. 2025

Modern Formal Semantics

Coding

```python
def incr_list(l: list):
    """Return list with elements incremented by 1.
    >>> incr_list([1, 2, 3])
    [2, 3, 4]
    >>> incr_list([5, 3, 5, 2, 3, 3, 9, 0, 123])
    [6, 4, 6, 3, 4, 4, 10, 1, 124]
    """
    return [i + 1 for i in l]
```

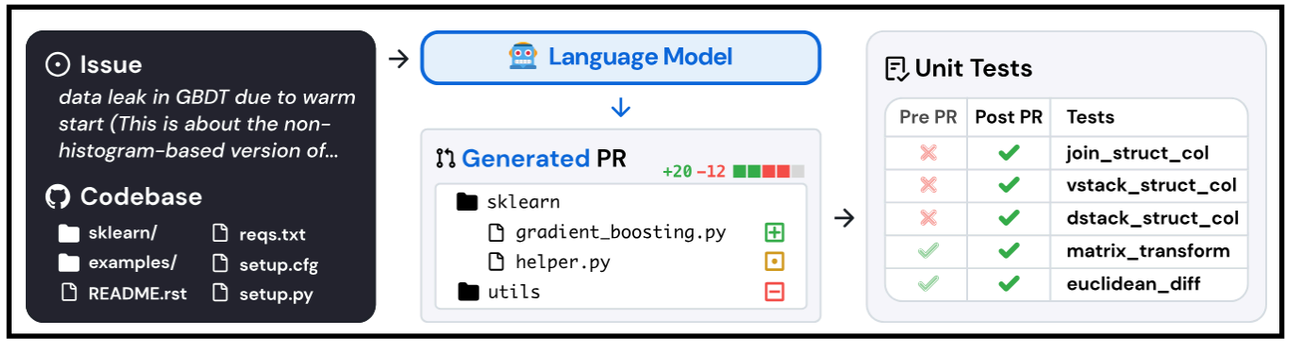
```python
def solution(lst):
    """Given a non-empty list of integers, return the sum of all of the odd elements
    that are in even positions.

    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)
```

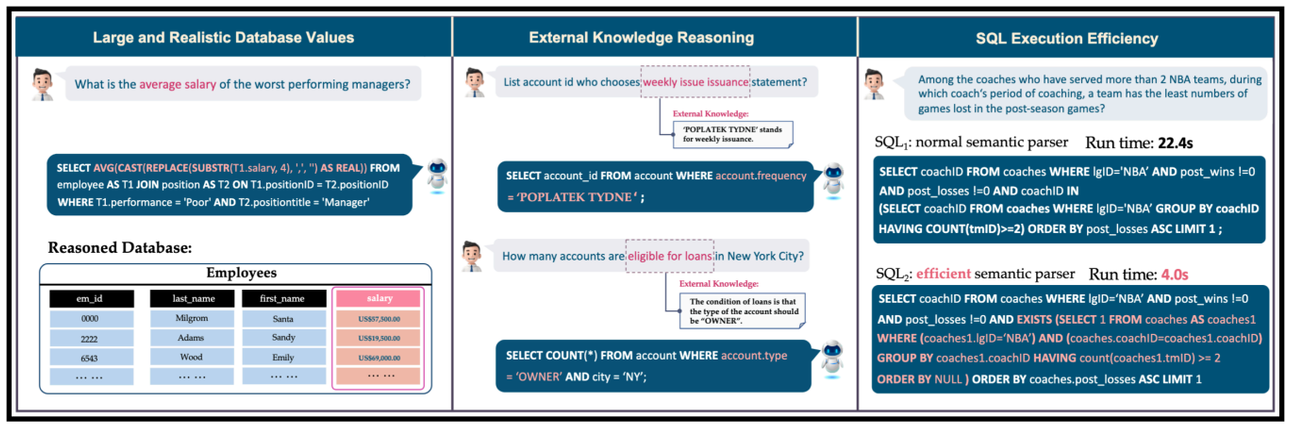
```python
def encode_cyclic(s: str):
    """
    returns encoded string by cycling groups of three characters.
    """
    # split string to groups. Each of length 3.
    groups = [s[(3 * i):min((3 * i + 3), len(s))] for i in range((len(s) + 2) // 3)]
    # cycle elements in each group. Unless group has fewer elements than 3.
    groups = [(group[1:] + group[0]) if len(group) == 3 else group for group in groups]
    return "".join(groups)

def decode_cyclic(s: str):
    """
    takes as input string encoded with encode_cyclic function. Returns decoded string.
    """
    # split string to groups. Each of length 3.
    groups = [s[(3 * i):min((3 * i + 3), len(s))] for i in range((len(s) + 2) // 3)]
    # cycle elements in each group.
    groups = [(group[-1] + group[:-1]) if len(group) == 3 else group for group in groups]
    return "".join(groups)
```

HumanEval, Chen et al. 2021



SWE-Bench, Jimenez et al. 2023



BIRD, Li et al. 2023

Modern Formal Semantics

Reasoning

Theorem For every integer n such that $n > 1$, n can be expressed as the product of one or more primes, uniquely up to the order in which they appear.

Proof In Integer is Expressible as Product of Primes, it is proved that every integer n such that $n > 1$, n can be expressed as the product of one or more primes.
In Prime Decomposition of Integer is Unique, it is proved that this prime decomposition is unique up to the order of the factors.

Premises

```
theorem perm_of_prod_eq_prod : ∀ {l₁ l₂ :
List M}, l₁.prod = l₂.prod → (∀ p ∈ l₁,
Prime p) → (∀ p ∈ l₂, Prime p) → Perm l₁ l₂
                    ⋮
theorem prime_of_mem_factors {n : ℕ} : ∀ {p
: ℕ}, (h : p ∈ factors n) → Prime p
```

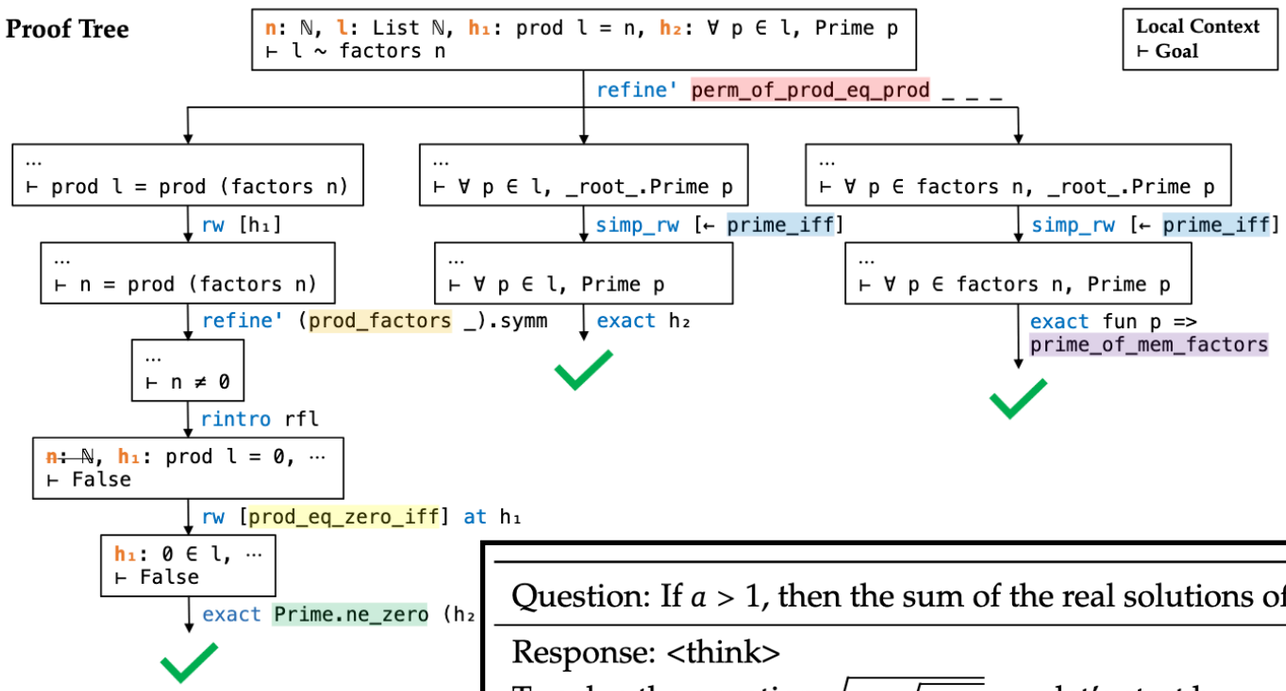
Theorem

```
theorem factors_unique {n : ℕ} {l : List ℕ}
(h₁ : prod l = n) (h₂ : ∀ p ∈ l, Prime p) :
l ~ factors n := by
```

Proof

```
  refine' perm_of_prod_eq_prod _ _ _
  · rw [h₁]
  refine' (prod_factors _).symm
  rintro rfl
  rw [prod_eq_zero_iff] at h₁
  exact Prime.ne_zero (h₂ 0 h₁) rfl
  · simp_rw [← prime_iff]
  exact h₂
  · simp_rw [← prime_iff]
  exact fun p => prime_of_mem_factors
```

Proof Tree



Li et al. 2024

Question: If $a > 1$, then the sum of the real solutions of $\sqrt{a - \sqrt{a + x}} = x$ is equal to

Response: <think>
To solve the equation $\sqrt{a - \sqrt{a + x}} = x$, let's start by squaring both $\cdots$
$\left(\sqrt{a - \sqrt{a + x}}\right)^2 = x^2 \implies a - \sqrt{a + x} = x^2$.
Rearrange to isolate the inner square root term:
$(a - x^2)^2 = a + x \implies a^2 - 2ax^2 + (x^2)^2 = a + x \implies x^4 - 2ax^2 - x + (a^2 - a) = 0$
$\cdots$
Wait, wait. Wait. That's an aha moment I can flag here.
Let's reevaluate this step-by-step to identify if the correct sum can be $\cdots$
We started with the equation:
$\sqrt{a - \sqrt{a + x}} = x$
First, let's square both sides:
$a - \sqrt{a + x} = x^2 \implies \sqrt{a + x} = a - x^2$
Next, I could square both sides again, treating the equation: $\cdots$
$\cdots$

DeepSeek-AI