Computational Semantics
Day 3: Lambda calculus
and the composition of meanings

Jan van Eijck[1] & Christina Unger[2]

[1]CWI, Amsterdam, and UiL-OTS, Utrecht, The Netherlands
[2]CITEC, Bielefeld University, Germany

ESSLLI 2011, Ljubljana

Outline

Lambda calculus

Lambdas changed my life.

(Barbara H. Partee)

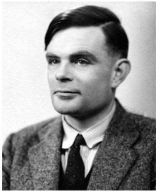
All you need is lambda.

(Simon Peyton-Jones)

History

In 1936, Turing and Church independently introduced two equivalent models of computation:

- **Alan Turing: Turing Machine**
  A function is computable if a sequence of instructions can be specified and then carried out by a simple abstract computational device.



- **Alonzo Church: Lambda Calculus**
  Every computable function is a function that is definable in the lambda calculus.

Connection to programming languages

- **Imperative programming languages** are based on the way a Turing machine is instructed.
- **Functional programming languages** are based on the lambda calculus.

In fact, the lambda calculus is the smallest universal programming language of the world (universal, because any computable function can be expressed and evaluated).

- Expressions correspond to programs.
- The reduction of an expression corresponds to program execution.

In fact, it is the core of functional programming languages, which are basically executable (typed) lambda calculi extended with constants, datatypes, input/output, etc.

Lambda calculus

The lambda calculus is a formal system for defining and investigating functions.

Two basic concept:

- function abstraction for representing functions, using a variable-binding operator λ
- function application, corresponding to substitution of bound variables

Formal definition and properties

Lambda calculus: Formal definition

Variables v and expressions E are defined as follows:

$$v ::= x \mid v'$$
$$E ::= v \mid \lambda v.E \mid (E\ E)$$

Variables

$$v ::= x \mid v'$$
$$E ::= v \mid \lambda v.E \mid (E\ E)$$

For our purposes, we write **variables** as lower case letters $x, y, z, \ldots$, possibly with indices.

Haskell: Variables (including function names) begin with a lower case letter.

- `x`, `x'`, `x1`
- `variable`, `newVAR`, `my_variable`
- ...

Function abstraction

$$v ::= x \mid v'$$
$$E ::= v \mid \lambda v.E \mid (E\ E)$$

$\lambda v.E$ represents a function, where v is the variable abstracted over (bound by the operator λ), and E is the body of the function.

Examples: $\lambda x.x$, $\lambda x.\lambda y.x$

Haskell: Function abstraction is written as `\ v -> E`.

- `\ x -> x`
- `\ x -> (\ y -> x)`
  or shorter: `\ x y -> x`

Function application

$$v ::= x \mid v'$$
$$E ::= v \mid \lambda v.E \mid (E\ E)$$

Function application represents applying an expression to another expression, e.g. a function to an argument.

Example: $(\lambda x.x\ \ y)$

Haskell: Function application is written as E E.

- `(\ x -> x) y`
- `(\ x y -> x) z`

Reducing expressions

Function application expressions can be reduced to simpler expressions. This corresponds to substitution of bound variables.

Reduction rule (called *beta reduction*):

$$(\lambda v.E_1 \quad E_2) \;\rhd\; E_1\,[v := E_2]$$

Where $E_1\,[v := E_2]$ denotes the substitution of E_2 for all free occurrences of v in E_1.

Example:

- $(\lambda x.(x\ y) \quad \lambda z.z) \;\rhd$

Reducing expressions

Function application expressions can be reduced to simpler expressions. This corresponds to substitution of bound variables.

Reduction rule (called *beta reduction*):

$$(\lambda v.E_1 \;\; E_2) \;\rhd\; E_1\,[v := E_2]$$

Where $E_1\,[v := E_2]$ denotes the substitution of E_2 for all free occurrences of v in E_1.

Example:

- $(\lambda x.(x \;\, y) \;\; \lambda z.z) \;\rhd\; (\lambda z.z \;\;\, y) \;\rhd$

Reducing expressions

Function application expressions can be reduced to simpler expressions.
This corresponds to substitution of bound variables.

Reduction rule (called *beta reduction*):

$$(\lambda v.E_1 \;\; E_2) \;\triangleright\; E_1\,[v := E_2]$$

Where $E_1\,[v := E_2]$ denotes the substitution of E_2 for all free occurrences of v in E_1.

Example:

- $(\lambda x.(x\;y) \;\; \lambda z.z) \;\triangleright\; (\lambda z.z \;\; y) \;\triangleright\; y$

Free and bound variables

An occurrence of the variable v in the expression E is bound if it is in the scope of a lambda prefix λv.

Example: $\lambda y.((\lambda x.x\ y)\ x)$

Free and bound variables

An occurrence of the variable v in the expression E is bound if it is in the scope of a lambda prefix λv.

Example: $\lambda y.((\lambda x.x\ y)\ x)$

Note: When substituting expressions, we have to make sure that no variables get accidentally captured.

- $(\lambda x \lambda y.(y\ x)\ \ y)$

This can be ensured by variable renaming.

Observation

Reductions need not come to an end.

- $(\lambda x.(x\ x)\ \ \lambda x.(x\ x))$
- $(\lambda x.((x\ x)\ x)\ \ \lambda x.((x\ x)\ x))$

Confluence

The result of beta reduction is independent from the order of reduction, i.e. if an expression can be evaluated in two different ways and both ways terminate, then both ways will yield the same result (*Church-Rosser theorem*).

- $(\lambda y.(y\ x)\ \ (\lambda x.x\ \ z))$

Note: The reduction order does, however, play a role for efficiency and can influence whether a reduction terminates or not.

- $(\lambda z.y\ \ (\lambda x.(x\ x)\ \ \lambda x.(x\ x)))$

Conventions

- Applications associate to the left; thus, when applying a function to a number of arguments, we can write $f\,x\,y\,z$ instead of $(((f\ x)\ y)\ z)$.
- The body of a lambda abstraction (the part after the dot) extends as far to the right as possible. I.e., $\lambda x.E_1\ E_2$ means $\lambda x.(E_1\ E_2)$, and not $(\lambda x.E_1)\ E_2$.

Adding function constants

Lambda calculus as we saw it is already enough to define natural numbers and arithmetic operations. We can abbreviate the corresponding expressions by adding constants to the language:

- 1,2,3. . . for natural numbers
- $+$ and * for addition and multiplication

Analogously, we can add constants a, b, c for entities, *wizard* for unary functions, *admire* for binary functions, and so on.

Observation

We can build expressions that do not make much sense.

- $(+\ x\ \lambda y.(1\ 2))$

Typed lambda calculus

Types

Types are sets of expressions, classifying expressions according to their combinatorial behavior.

Types

$$\tau ::= e \mid t \mid (\tau \to \tau)$$

Where e (for entities) and t (for truth values) are basic types and $\tau \to \tau$ are functional types.

Typed lambda calculus

Each lambda expression is assigned a type, specified as follows:

- **Variables**:
  For each type τ we have variables for that type.
- **Abstraction**:
  If $v :: \delta$ and $E :: \tau$, then $\lambda v.E :: \delta \to \tau$.
- **Application**:
  If $E_1 :: \delta \to \tau$ and $E_2 :: \delta$, then $(E_1\ E_2) :: \tau$.

Examples

Of which types are the following expressions? (Assuming that numbers are of type `Int`, + and * are of type `Int` → `Int` → `Int`.)

- $\lambda x.(+\ 1\ x)$
- $(\lambda x.(x\ 2)\ \ \lambda y.(*\ y\ y))$
- $(\lambda x.(y\ x)\ \ z)$
- $\lambda z.(z\ z)$

Typed meanings for natural language

Lambda calculus with constants

We extend lambda calculus with logical and non-logical constants.

$$\mathbf{E} ::= \mathbf{c} \mid \mathbf{v} \mid (\mathbf{E}\ \mathbf{E}) \mid (\lambda \mathbf{v}.\mathbf{E})$$

$$\mathbf{v} ::= x \mid \mathbf{v}'$$

$$\mathbf{c} ::= a \mid b \mid c \mid d \mid e \mid f$$
$$\mid giant \mid princess \mid wizard \mid happy \mid laugh \mid admire \mid \dots$$
$$\mid \ \wedge \ \mid \ \vee \ \mid \rightarrow \mid \ \neg \ \mid \ \forall \ \mid \ \exists$$

Lambda calculus with constants

Example expressions:

- $((\wedge\ (evil\ x))\ (wizard\ x))$
- $(\forall\ \lambda x.((admire\ x)\ c))$

Types of logical expressions:

- $\wedge :: t \to (t \to t)$
- $\vee :: t \to (t \to t)$
- $\to :: t \to (t \to t)$
- $\neg :: t \to t$
- $\forall :: (e \to t) \to t$
- $\exists :: (e \to t) \to t$

Types of non-logical constants

Individual constants are of type e.

- $a :: e$

Types of non-logical constants

Individual constants are of type e.

- $a :: e$

One-place predicate constants are of type $e \to t$.

- $wizard :: e \to t$
- $laugh :: e \to t$
- $evil :: e \to t$

Types of non-logical constants

Individual constants are of type e.

- $a :: e$

One-place predicate constants are of type $e \rightarrow t$.

- $\mathit{wizard} :: e \rightarrow t$
- $\mathit{laugh} :: e \rightarrow t$
- $\mathit{evil} :: e \rightarrow t$

Two-place predicate constants are of type $e \rightarrow (e \rightarrow t)$.

- $\mathit{admire} :: e \rightarrow (e \rightarrow t)$

Types of non-logical constants

Individual constants are of type e.

- $a :: e$

One-place predicate constants are of type $e \rightarrow t$.

- $wizard :: e \rightarrow t$
- $laugh :: e \rightarrow t$
- $evil :: e \rightarrow t$

Two-place predicate constants are of type $e \rightarrow (e \rightarrow t)$.

- $admire :: e \rightarrow (e \rightarrow t)$

Three-place predicate constants are of type $e \rightarrow (e \rightarrow (e \rightarrow t))$.

- $give :: e \rightarrow (e \rightarrow (e \rightarrow t))$

Lambda calculus with constants

Lambda calculus with constants subsumes predicate logic.

Lambda calculus	Predicate logic
$((\mathit{admire}\ x)\ y)$	$\mathit{admire}(y, x)$
$((\wedge\ (\mathit{evil}\ x))\ (\mathit{wizard}\ y))$	$\mathit{evil}(x) \wedge \mathit{wizard}(y)$
$(\forall\ \lambda x.(\mathit{happy}\ x))$	$\forall x.\mathit{happy}(x)$

Lambda calculus with constants

Lambda calculus with constants subsumes predicate logic.

Lambda calculus	Predicate logic
$((admire\ x)\ y)$	$admire(y,x)$
$((\wedge\ (evil\ x))\ (wizard\ y))$	$evil(x) \wedge wizard(y)$
$(\forall\ \lambda x.(happy\ x))$	$\forall x.happy(x)$

For better readability, we abbreviate

- $(\forall\ \lambda x.(P\ x))$ as $\forall x.(P\ x)$
- $((\wedge\ E_1)\ E_2)$ as $E_1 \wedge E_2$
- $(\neg\ E)$ as $\neg E$

Example translation

Lexical item	Constant	
Atreyu	*a*	(individual constant)
princess	*princess*	(unary function constant)
cheered	*cheer*	(unary function constant)
drunken	*drunken*	(unary function constant)
admired	*admire*	(binary function constant)
gave	*give*	(ternary function constant)

Eta-reduction

Eta-reduction eliminates redundant lambda abstractions, i.e. lambda abstractions that only have the purpose of passing its argument to another function.

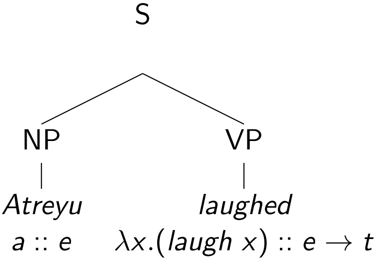
$$\lambda x.(E\ x) \rhd E$$
if x does not occur free in E

For example, *happy* and $\lambda x.(\mathit{happy}\ x)$ are equivalent.

Composing meanings

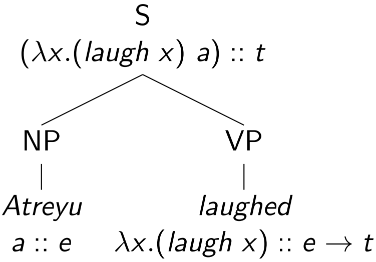
Example

Atreyu laughed.



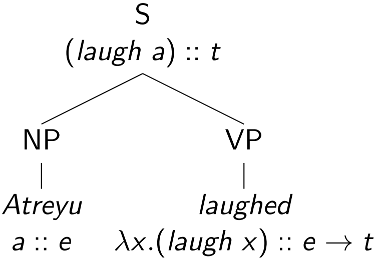
Example

Atreyu laughed.



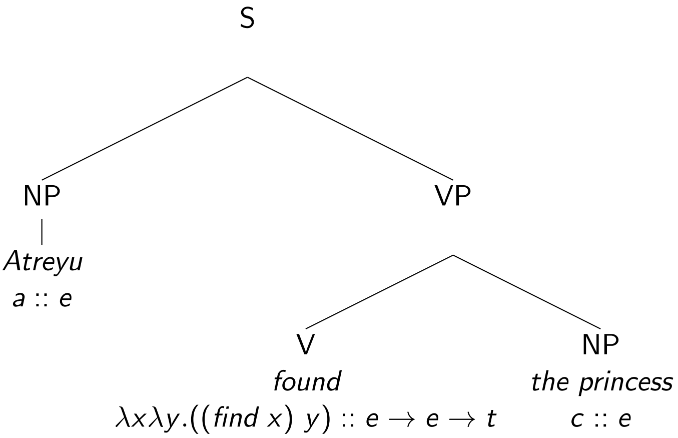
Example

Atreyu laughed.



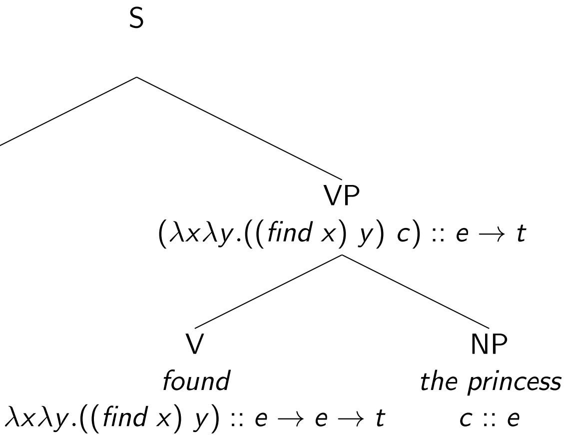
Example

Atreyu found the princess.



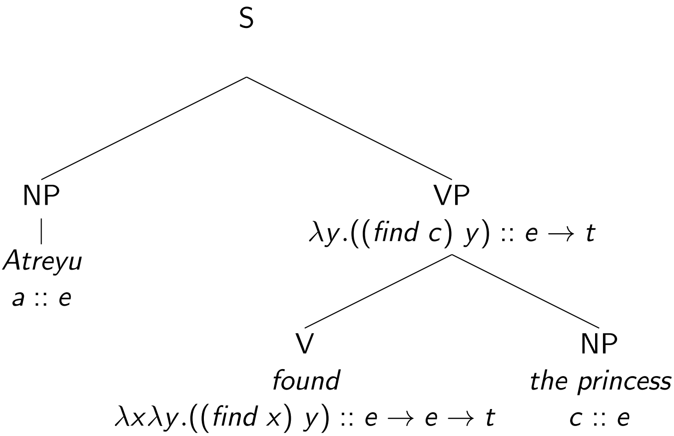
Example

Atreyu found the princess.



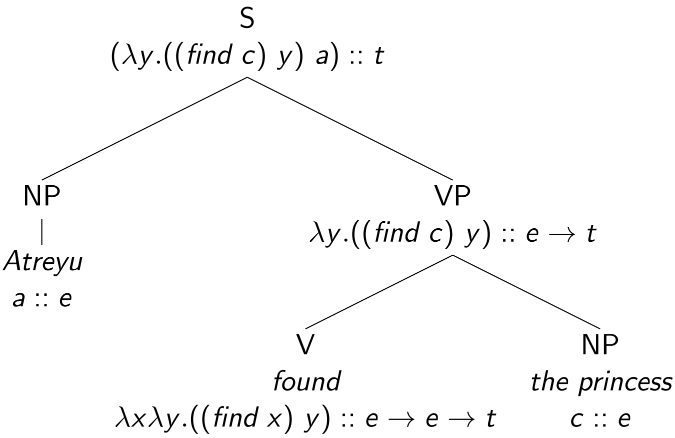
Example

Atreyu found the princess.



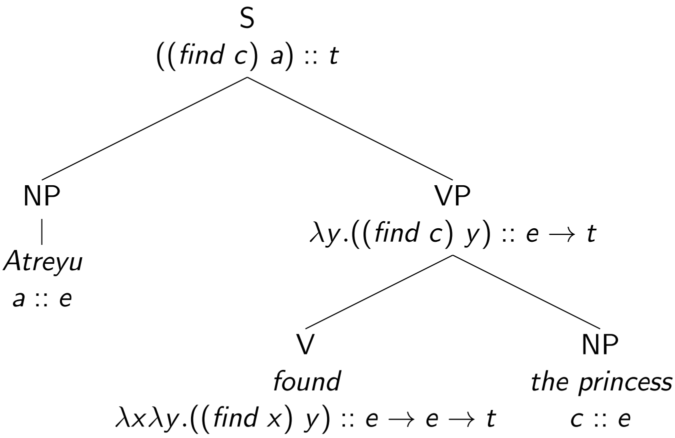
Example

Atreyu found the princess.



Example

Atreyu found the princess.



The meaning of sentences and their parts

- **The meaning of a sentence** is an expression of the typed lambda calculus corresponding to a formula of first-order predicate logic.
- **The meaning of its parts** are functional expressions of the typed lambda calculus.
- **The semantic rules for combining the parts** are function application (as interpretation of subcategorization rules) and predicate modification (as interpretation of adjunction rules).

Rule-to-rule correspondence:
Match every grammar rule with a rule for semantic interpretation.

Getting started

NAME → *Atreyu*	⟦**NAME**⟧ →
NP → **NAME**	⟦**NP**⟧ →
N → *wizard*	⟦**N**⟧ →
ADJ → *evil*	⟦**ADJ**⟧ →
IV → *laughed*	⟦**IV**⟧ →
VP → **IV**	⟦**VP**⟧ →
S → **NP VP**	⟦**S**⟧ →
TV → *admired*	⟦**TV**⟧ →
VP → **TV NP**	⟦**VP**⟧ →
N → **ADJ N**	⟦**N**⟧ →
RN → **N REL VP**	⟦**RN**⟧ →
RN → **N REL NP TV**	⟦**RN**⟧ →

Getting started

NAME $\rightarrow$ *Atreyu*	$[\![$**NAME**$]\!] \rightarrow a :: e$
NP $\rightarrow$ **NAME**	$[\![$**NP**$]\!] \rightarrow$
N $\rightarrow$ *wizard*	$[\![$**N**$]\!] \rightarrow$
ADJ $\rightarrow$ *evil*	$[\![$**ADJ**$]\!] \rightarrow$
IV $\rightarrow$ *laughed*	$[\![$**IV**$]\!] \rightarrow$
VP $\rightarrow$ **IV**	$[\![$**VP**$]\!] \rightarrow$
S $\rightarrow$ **NP VP**	$[\![$**S**$]\!] \rightarrow$
TV $\rightarrow$ *admired*	$[\![$**TV**$]\!] \rightarrow$
VP $\rightarrow$ **TV NP**	$[\![$**VP**$]\!] \rightarrow$
N $\rightarrow$ **ADJ N**	$[\![$**N**$]\!] \rightarrow$
RN $\rightarrow$ **N REL VP**	$[\![$**RN**$]\!] \rightarrow$
RN $\rightarrow$ **N REL NP TV**	$[\![$**RN**$]\!] \rightarrow$

Getting started

NAME → *Atreyu*	⟦**NAME**⟧ → $a :: e$
NP → **NAME**	⟦**NP**⟧ → ⟦**NAME**⟧ $:: e$
N → *wizard*	⟦**N**⟧ →
ADJ → *evil*	⟦**ADJ**⟧ →
IV → *laughed*	⟦**IV**⟧ →
VP → **IV**	⟦**VP**⟧ →
S → **NP VP**	⟦**S**⟧ →
TV → *admired*	⟦**TV**⟧ →
VP → **TV NP**	⟦**VP**⟧ →
N → **ADJ N**	⟦**N**⟧ →
RN → **N REL VP**	⟦**RN**⟧ →
RN → **N REL NP TV**	⟦**RN**⟧ →

Getting started

NAME $\rightarrow$ *Atreyu*	$[\![\text{NAME}]\!] \rightarrow a :: e$
NP $\rightarrow$ NAME	$[\![\text{NP}]\!] \rightarrow [\![\text{NAME}]\!] :: e$
N $\rightarrow$ *wizard*	$[\![\text{N}]\!] \rightarrow \lambda x.(\mathit{wizard}\ x) :: e \rightarrow t$
ADJ $\rightarrow$ *evil*	$[\![\text{ADJ}]\!] \rightarrow$
IV $\rightarrow$ *laughed*	$[\![\text{IV}]\!] \rightarrow$
VP $\rightarrow$ IV	$[\![\text{VP}]\!] \rightarrow$
S $\rightarrow$ NP VP	$[\![\text{S}]\!] \rightarrow$
TV $\rightarrow$ *admired*	$[\![\text{TV}]\!] \rightarrow$
VP $\rightarrow$ TV NP	$[\![\text{VP}]\!] \rightarrow$
N $\rightarrow$ ADJ N	$[\![\text{N}]\!] \rightarrow$
RN $\rightarrow$ N REL VP	$[\![\text{RN}]\!] \rightarrow$
RN $\rightarrow$ N REL NP TV	$[\![\text{RN}]\!] \rightarrow$

Getting started

NAME $\rightarrow$ *Atreyu*	$[\![\textbf{NAME}]\!] \rightarrow a :: e$
NP $\rightarrow$ **NAME**	$[\![\textbf{NP}]\!] \rightarrow [\![\textbf{NAME}]\!] :: e$
N $\rightarrow$ *wizard*	$[\![\textbf{N}]\!] \rightarrow \lambda x.(wizard\ x) :: e \rightarrow t$
ADJ $\rightarrow$ *evil*	$[\![\textbf{ADJ}]\!] \rightarrow \lambda x.(evil\ x) :: e \rightarrow t$
IV $\rightarrow$ *laughed*	$[\![\textbf{IV}]\!] \rightarrow$
VP $\rightarrow$ **IV**	$[\![\textbf{VP}]\!] \rightarrow$
S $\rightarrow$ **NP VP**	$[\![\textbf{S}]\!] \rightarrow$
TV $\rightarrow$ *admired*	$[\![\textbf{TV}]\!] \rightarrow$
VP $\rightarrow$ **TV NP**	$[\![\textbf{VP}]\!] \rightarrow$
N $\rightarrow$ **ADJ N**	$[\![\textbf{N}]\!] \rightarrow$
RN $\rightarrow$ **N REL VP**	$[\![\textbf{RN}]\!] \rightarrow$
RN $\rightarrow$ **N REL NP TV**	$[\![\textbf{RN}]\!] \rightarrow$

Getting started

NAME $\rightarrow$	*Atreyu*	$[\![\textbf{NAME}]\!] \rightarrow$	$a :: e$
NP $\rightarrow$	**NAME**	$[\![\textbf{NP}]\!] \rightarrow$	$[\![\textbf{NAME}]\!] :: e$
N $\rightarrow$	*wizard*	$[\![\textbf{N}]\!] \rightarrow$	$\lambda x.(wizard\ x) :: e \rightarrow t$
ADJ $\rightarrow$	*evil*	$[\![\textbf{ADJ}]\!] \rightarrow$	$\lambda x.(evil\ x) :: e \rightarrow t$
IV $\rightarrow$	*laughed*	$[\![\textbf{IV}]\!] \rightarrow$	$\lambda x.(laugh\ x) :: e \rightarrow t$
VP $\rightarrow$	**IV**	$[\![\textbf{VP}]\!] \rightarrow$	
S $\rightarrow$	**NP VP**	$[\![\textbf{S}]\!] \rightarrow$	
TV $\rightarrow$	*admired*	$[\![\textbf{TV}]\!] \rightarrow$	
VP $\rightarrow$	**TV NP**	$[\![\textbf{VP}]\!] \rightarrow$	
N $\rightarrow$	**ADJ N**	$[\![\textbf{N}]\!] \rightarrow$	
RN $\rightarrow$	**N REL VP**	$[\![\textbf{RN}]\!] \rightarrow$	
RN $\rightarrow$	**N REL NP TV**	$[\![\textbf{RN}]\!] \rightarrow$	

Getting started

NAME $\rightarrow$ *Atreyu*	$[\![\textbf{NAME}]\!] \rightarrow a :: e$
NP $\rightarrow$ **NAME**	$[\![\textbf{NP}]\!] \rightarrow [\![\textbf{NAME}]\!] :: e$
N $\rightarrow$ *wizard*	$[\![\textbf{N}]\!] \rightarrow \lambda x.(\mathit{wizard}\ x) :: e \rightarrow t$
ADJ $\rightarrow$ *evil*	$[\![\textbf{ADJ}]\!] \rightarrow \lambda x.(\mathit{evil}\ x) :: e \rightarrow t$
IV $\rightarrow$ *laughed*	$[\![\textbf{IV}]\!] \rightarrow \lambda x.(\mathit{laugh}\ x) :: e \rightarrow t$
VP $\rightarrow$ **IV**	$[\![\textbf{VP}]\!] \rightarrow [\![\textbf{IV}]\!] :: e \rightarrow t$
S $\rightarrow$ **NP VP**	$[\![\textbf{S}]\!] \rightarrow$
TV $\rightarrow$ *admired*	$[\![\textbf{TV}]\!] \rightarrow$
VP $\rightarrow$ **TV NP**	$[\![\textbf{VP}]\!] \rightarrow$
N $\rightarrow$ **ADJ N**	$[\![\textbf{N}]\!] \rightarrow$
RN $\rightarrow$ **N REL VP**	$[\![\textbf{RN}]\!] \rightarrow$
RN $\rightarrow$ **N REL NP TV**	$[\![\textbf{RN}]\!] \rightarrow$

Getting started

$$
\begin{array}{rll}
\textbf{NAME} \rightarrow \textit{Atreyu} & [\![\textbf{NAME}]\!] \rightarrow a :: e \\
\textbf{NP} \rightarrow \textbf{NAME} & [\![\textbf{NP}]\!] \rightarrow [\![\textbf{NAME}]\!] :: e \\
\textbf{N} \rightarrow \textit{wizard} & [\![\textbf{N}]\!] \rightarrow \lambda x.(\textit{wizard}\ x) :: e \rightarrow t \\
\textbf{ADJ} \rightarrow \textit{evil} & [\![\textbf{ADJ}]\!] \rightarrow \lambda x.(\textit{evil}\ x) :: e \rightarrow t \\
\textbf{IV} \rightarrow \textit{laughed} & [\![\textbf{IV}]\!] \rightarrow \lambda x.(\textit{laugh}\ x) :: e \rightarrow t \\
\textbf{VP} \rightarrow \textbf{IV} & [\![\textbf{VP}]\!] \rightarrow [\![\textbf{IV}]\!] :: e \rightarrow t \\
\textbf{S} \rightarrow \textbf{NP VP} & [\![\textbf{S}]\!] \rightarrow ([\![\textbf{VP}]\!]\ [\![\textbf{NP}]\!]) :: t \\
\textbf{TV} \rightarrow \textit{admired} & [\![\textbf{TV}]\!] \rightarrow \\
\textbf{VP} \rightarrow \textbf{TV NP} & [\![\textbf{VP}]\!] \rightarrow \\
\textbf{N} \rightarrow \textbf{ADJ N} & [\![\textbf{N}]\!] \rightarrow \\
\textbf{RN} \rightarrow \textbf{N REL VP} & [\![\textbf{RN}]\!] \rightarrow \\
\textbf{RN} \rightarrow \textbf{N REL NP TV} & [\![\textbf{RN}]\!] \rightarrow
\end{array}
$$

Getting started

NAME $\rightarrow$ *Atreyu*	$[\![\mathbf{NAME}]\!] \rightarrow a :: e$
NP $\rightarrow$ **NAME**	$[\![\mathbf{NP}]\!] \rightarrow [\![\mathbf{NAME}]\!] :: e$
N $\rightarrow$ *wizard*	$[\![\mathbf{N}]\!] \rightarrow \lambda x.(wizard\ x) :: e \rightarrow t$
ADJ $\rightarrow$ *evil*	$[\![\mathbf{ADJ}]\!] \rightarrow \lambda x.(evil\ x) :: e \rightarrow t$
IV $\rightarrow$ *laughed*	$[\![\mathbf{IV}]\!] \rightarrow \lambda x.(laugh\ x) :: e \rightarrow t$
VP $\rightarrow$ **IV**	$[\![\mathbf{VP}]\!] \rightarrow [\![\mathbf{IV}]\!] :: e \rightarrow t$
S $\rightarrow$ **NP VP**	$[\![\mathbf{S}]\!] \rightarrow ([\![\mathbf{VP}]\!]\ [\![\mathbf{NP}]\!]) :: t$
TV $\rightarrow$ *admired*	$[\![\mathbf{TV}]\!] \rightarrow \lambda x \lambda y.((admire\ x)\ y) :: e \rightarrow e \rightarrow t$
VP $\rightarrow$ **TV NP**	$[\![\mathbf{VP}]\!] \rightarrow$
N $\rightarrow$ **ADJ N**	$[\![\mathbf{N}]\!] \rightarrow$
RN $\rightarrow$ **N REL VP**	$[\![\mathbf{RN}]\!] \rightarrow$
RN $\rightarrow$ **N REL NP TV**	$[\![\mathbf{RN}]\!] \rightarrow$

Getting started

NAME $\rightarrow$ *Atreyu*	$[\![\text{NAME}]\!] \rightarrow a :: e$
NP $\rightarrow$ **NAME**	$[\![\text{NP}]\!] \rightarrow [\![\text{NAME}]\!] :: e$
N $\rightarrow$ *wizard*	$[\![\text{N}]\!] \rightarrow \lambda x.(\mathit{wizard}\ x) :: e \rightarrow t$
ADJ $\rightarrow$ *evil*	$[\![\text{ADJ}]\!] \rightarrow \lambda x.(\mathit{evil}\ x) :: e \rightarrow t$
IV $\rightarrow$ *laughed*	$[\![\text{IV}]\!] \rightarrow \lambda x.(\mathit{laugh}\ x) :: e \rightarrow t$
VP $\rightarrow$ **IV**	$[\![\text{VP}]\!] \rightarrow [\![\text{IV}]\!] :: e \rightarrow t$
S $\rightarrow$ **NP VP**	$[\![\text{S}]\!] \rightarrow ([\![\text{VP}]\!]\ [\![\text{NP}]\!]) :: t$
TV $\rightarrow$ *admired*	$[\![\text{TV}]\!] \rightarrow \lambda x \lambda y.((\mathit{admire}\ x)\ y) :: e \rightarrow e \rightarrow t$
VP $\rightarrow$ **TV NP**	$[\![\text{VP}]\!] \rightarrow ([\![\text{TV}]\!]\ [\![\text{NP}]\!]) :: e \rightarrow t$
N $\rightarrow$ **ADJ N**	$[\![\text{N}]\!] \rightarrow$
RN $\rightarrow$ **N REL VP**	$[\![\text{RN}]\!] \rightarrow$
RN $\rightarrow$ **N REL NP TV**	$[\![\text{RN}]\!] \rightarrow$

Getting started

NAME $\rightarrow$ *Atreyu*	$[\![\textbf{NAME}]\!] \rightarrow a :: e$
NP $\rightarrow$ **NAME**	$[\![\textbf{NP}]\!] \rightarrow [\![\textbf{NAME}]\!] :: e$
N $\rightarrow$ *wizard*	$[\![\textbf{N}]\!] \rightarrow \lambda x.(\mathit{wizard}\ x) :: e \rightarrow t$
ADJ $\rightarrow$ *evil*	$[\![\textbf{ADJ}]\!] \rightarrow \lambda x.(\mathit{evil}\ x) :: e \rightarrow t$
IV $\rightarrow$ *laughed*	$[\![\textbf{IV}]\!] \rightarrow \lambda x.(\mathit{laugh}\ x) :: e \rightarrow t$
VP $\rightarrow$ **IV**	$[\![\textbf{VP}]\!] \rightarrow [\![\textbf{IV}]\!] :: e \rightarrow t$
S $\rightarrow$ **NP VP**	$[\![\textbf{S}]\!] \rightarrow ([\![\textbf{VP}]\!]\ [\![\textbf{NP}]\!]) :: t$
TV $\rightarrow$ *admired*	$[\![\textbf{TV}]\!] \rightarrow \lambda x \lambda y.((\mathit{admire}\ x)\ y) :: e \rightarrow e \rightarrow t$
VP $\rightarrow$ **TV NP**	$[\![\textbf{VP}]\!] \rightarrow ([\![\textbf{TV}]\!]\ [\![\textbf{NP}]\!]) :: e \rightarrow t$
N $\rightarrow$ **ADJ N**	$[\![\textbf{N}]\!] \rightarrow \lambda x.(([\![\textbf{ADJ}]\!]\ x) \wedge ([\![\textbf{N}]\!]\ x)) :: e \rightarrow t$
RN $\rightarrow$ **N REL VP**	$[\![\textbf{RN}]\!] \rightarrow$
RN $\rightarrow$ **N REL NP TV**	$[\![\textbf{RN}]\!] \rightarrow$

Getting started

NAME $\rightarrow$ *Atreyu*	$[\![\mathbf{NAME}]\!] \rightarrow a :: e$
NP $\rightarrow$ **NAME**	$[\![\mathbf{NP}]\!] \rightarrow [\![\mathbf{NAME}]\!] :: e$
N $\rightarrow$ *wizard*	$[\![\mathbf{N}]\!] \rightarrow \lambda x.(\mathit{wizard}\ x) :: e \rightarrow t$
ADJ $\rightarrow$ *evil*	$[\![\mathbf{ADJ}]\!] \rightarrow \lambda x.(\mathit{evil}\ x) :: e \rightarrow t$
IV $\rightarrow$ *laughed*	$[\![\mathbf{IV}]\!] \rightarrow \lambda x.(\mathit{laugh}\ x) :: e \rightarrow t$
VP $\rightarrow$ **IV**	$[\![\mathbf{VP}]\!] \rightarrow [\![\mathbf{IV}]\!] :: e \rightarrow t$
S $\rightarrow$ **NP VP**	$[\![\mathbf{S}]\!] \rightarrow ([\![\mathbf{VP}]\!]\ [\![\mathbf{NP}]\!]) :: t$
TV $\rightarrow$ *admired*	$[\![\mathbf{TV}]\!] \rightarrow \lambda x \lambda y.((\mathit{admire}\ x)\ y) :: e \rightarrow e \rightarrow t$
VP $\rightarrow$ **TV NP**	$[\![\mathbf{VP}]\!] \rightarrow ([\![\mathbf{TV}]\!]\ [\![\mathbf{NP}]\!]) :: e \rightarrow t$
N $\rightarrow$ **ADJ N**	$[\![\mathbf{N}]\!] \rightarrow \lambda x.(([\![\mathbf{ADJ}]\!]\ x) \wedge ([\![\mathbf{N}]\!]\ x)) :: e \rightarrow t$
RN $\rightarrow$ **N REL VP**	$[\![\mathbf{RN}]\!] \rightarrow \lambda x.([\![\mathbf{N}]\!]\ x) \wedge ([\![\mathbf{VP}]\!]\ x) :: e \rightarrow t$
RN $\rightarrow$ **N REL NP TV**	$[\![\mathbf{RN}]\!] \rightarrow$

Getting started

NAME $\rightarrow$ *Atreyu*	$[\![\textbf{NAME}]\!] \rightarrow a :: e$
NP $\rightarrow$ **NAME**	$[\![\textbf{NP}]\!] \rightarrow [\![\textbf{NAME}]\!] :: e$
N $\rightarrow$ *wizard*	$[\![\textbf{N}]\!] \rightarrow \lambda x.(\mathit{wizard}\ x) :: e \rightarrow t$
ADJ $\rightarrow$ *evil*	$[\![\textbf{ADJ}]\!] \rightarrow \lambda x.(\mathit{evil}\ x) :: e \rightarrow t$
IV $\rightarrow$ *laughed*	$[\![\textbf{IV}]\!] \rightarrow \lambda x.(\mathit{laugh}\ x) :: e \rightarrow t$
VP $\rightarrow$ **IV**	$[\![\textbf{VP}]\!] \rightarrow [\![\textbf{IV}]\!] :: e \rightarrow t$
S $\rightarrow$ **NP VP**	$[\![\textbf{S}]\!] \rightarrow ([\![\textbf{VP}]\!]\ [\![\textbf{NP}]\!]) :: t$
TV $\rightarrow$ *admired*	$[\![\textbf{TV}]\!] \rightarrow \lambda x \lambda y.((\mathit{admire}\ x)\ y) :: e \rightarrow e \rightarrow t$
VP $\rightarrow$ **TV NP**	$[\![\textbf{VP}]\!] \rightarrow ([\![\textbf{TV}]\!]\ [\![\textbf{NP}]\!]) :: e \rightarrow t$
N $\rightarrow$ **ADJ N**	$[\![\textbf{N}]\!] \rightarrow \lambda x.(([\![\textbf{ADJ}]\!]\ x) \wedge ([\![\textbf{N}]\!]\ x)) :: e \rightarrow t$
RN $\rightarrow$ **N REL VP**	$[\![\textbf{RN}]\!] \rightarrow \lambda x.([\![\textbf{N}]\!]\ x) \wedge ([\![\textbf{VP}]\!]\ x) :: e \rightarrow t$
RN $\rightarrow$ **N REL NP TV**	$[\![\textbf{RN}]\!] \rightarrow \lambda x.([\![\textbf{N}]\!]\ x) \wedge (([\![\textbf{TV}]\!]\ x)\ [\![\textbf{NP}]\!])$

Quantifier denotations

Observation

Quantificational NPs do not refer to particular individuals.

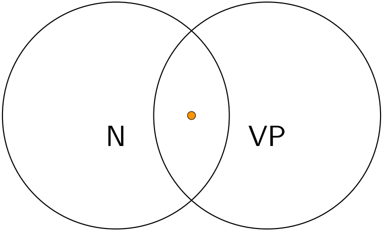
- *Every zombie bites someone.*
- *Nobody has seen a unicorn, because there aren't any!*

Maybe quantifiers indicate the quantity of something (all zombies, the empty set, and so on). But that's not exactly right, as it's not quantities that get predicated over (it's not the empty set that has seen a unicorn).

Rather, quantifiers relate sets.

Examples

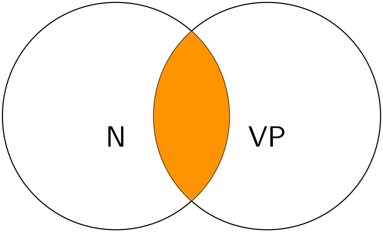
[$_{NP}$ *Some* [$_{N}$ *robot*]] [$_{VP}$ *failed the Turing Test*].



$$N \cap VP \neq \emptyset$$

Examples

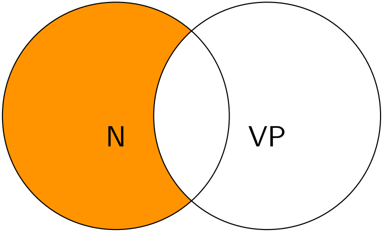
[$_{NP}$ *Every* [$_{N}$ *robot*]] [$_{VP}$ *failed the Turing Test*].



$$\mathrm{N} - \mathrm{VP} = \emptyset$$

Examples

$[_{NP}$ *No* $[_{N}$ *robot*$]]$ $[_{VP}$ *failed the Turing Test*$]$.



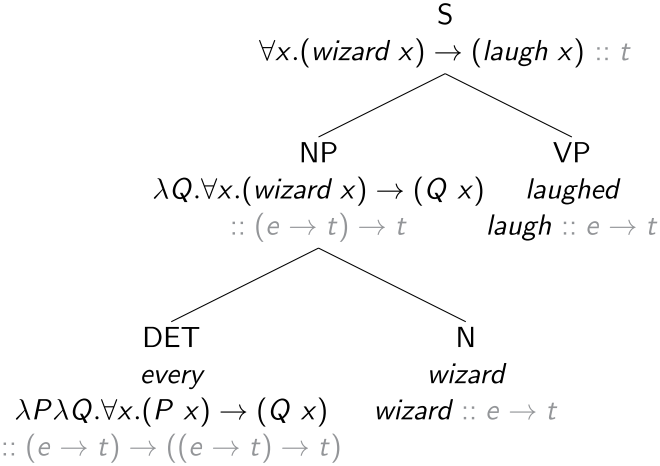
$$N \cap VP = \emptyset$$

Quantifiers as second-order predicates

Quantifiers can be expressed as second-order predicates of type $(e \rightarrow t) \rightarrow (e \rightarrow t) \rightarrow t$.

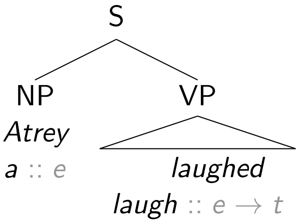
$$\begin{aligned} [\![some]\!] &= \lambda P\, \lambda Q.\, \exists x.(P\ x) \wedge (Q\ x) \\ [\![every]\!] &= \lambda P\, \lambda Q.\, \forall x.(P\ x) \rightarrow (Q\ x) \\ [\![no]\!] &= \lambda P\, \lambda Q.\, \forall x.(P\ x) \rightarrow \neg(Q\ x) \\ & \lambda P\, \lambda Q.\, \neg\exists x.(P\ x) \wedge (Q\ x) \end{aligned}$$

Example (the easy case)



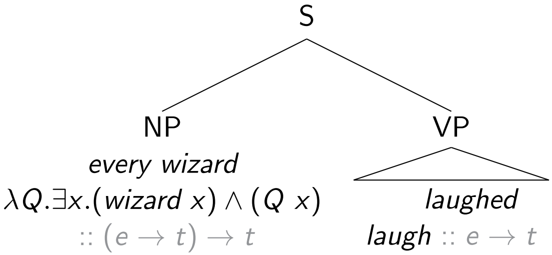
Problem 1:
Uniformity of NP denotations

Uniformity of NP denotations



Semantic rule: $[\![\mathbf{S}]\!] \rightarrow ([\![\mathbf{VP}]\!]\ [\![\mathbf{NP}]\!])$

Uniformity of NP denotations



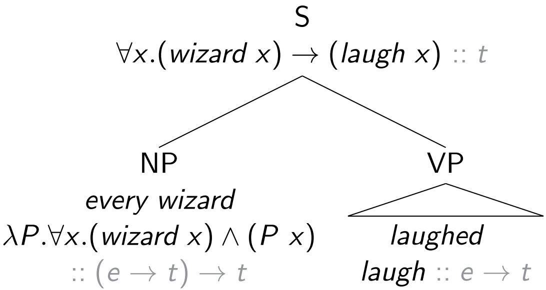
Semantic rule: $[\![\mathbf{S}]\!] \rightarrow ([\![\mathbf{NP}]\!]\ [\![\mathbf{VP}]\!])$

Solution: 'Generalization to the worst case'

All NPs denote expressions of type $(e \to t) \to t$.

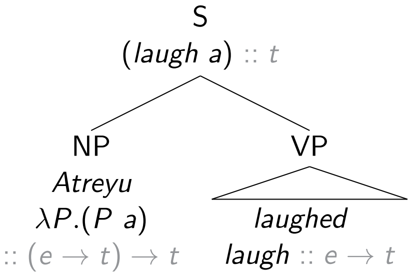
$$[\![\textit{some princess}]\!] = \lambda P.\exists x.(\textit{princess}\ x) \wedge (P\ x)$$
$$[\![\textit{every wizard}]\!] = \lambda P.\forall x.(\textit{wizard}\ x) \to (P\ x)$$
$$[\![\textit{Atreyu}]\!] = \lambda P.(P\ a)$$
$$[\![\textit{Dorothy}]\!] = \lambda P.(P\ d)$$

Solution: ‘Generalization to the worst case’



Semantic rule: $[\![\textbf{S}]\!] \rightarrow ([\![\textbf{NP}]\!]\ [\![\textbf{VP}]\!])$

Solution: ‘Generalization to the worst case’



Semantic rule: $[\![\mathbf{S}]\!] \rightarrow ([\![\mathbf{NP}]\!]\ [\![\mathbf{VP}]\!])$

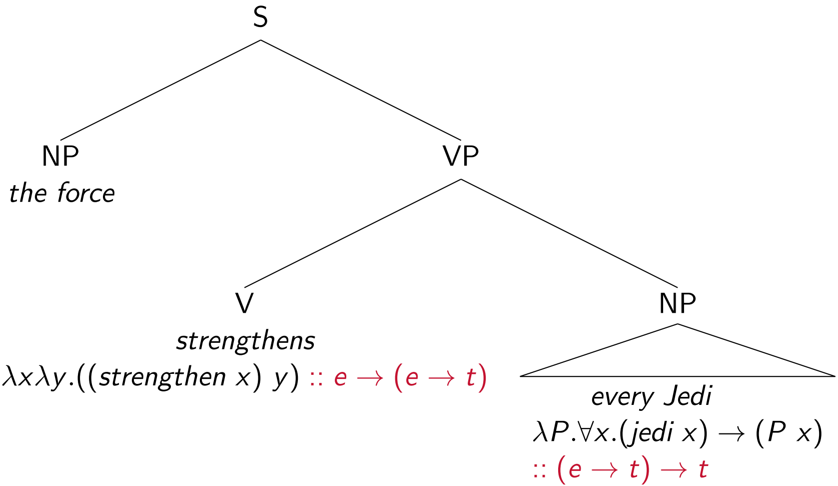
Individuals as generalized quantifiers

$$[\![Atreyu]\!] = \lambda P.(P\ a)$$

- *Atreyu* denotes a function that takes a predicate and hands it the argument *a*.
- So it tells us, which properties are true of Atreyu.
- Technically, *Atreyu* denotes the characteristic function of the set of all sets that contain the individual Atreyu.
  In other words: *Atreyu* denotes the set of all properties of Atreyu.

Problem 2: Quantifiers in object position

Quantifiers in object position

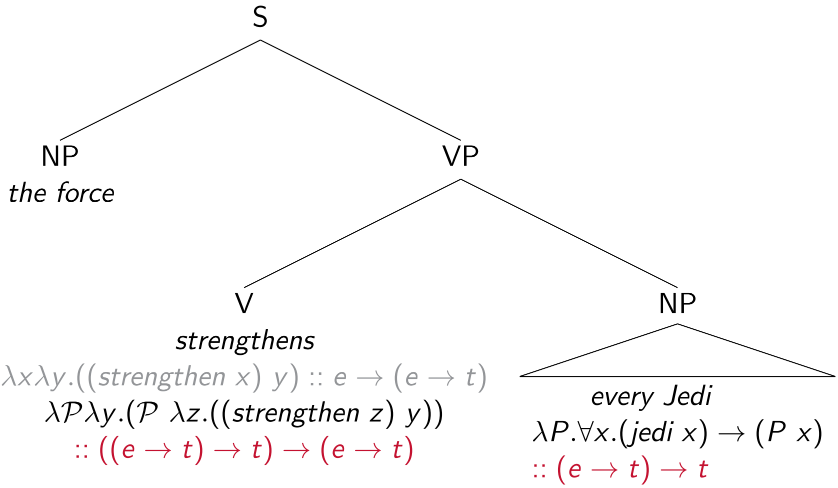


Solution 1: Type raising

Herman Hendriks' Flexible Types approach:

- Lexical expressions are assigned a minimal type (e.g. e for denotations of proper names).
- Translations of higher types are derived by type-lifting rules:
    - **Value raising:** Any expression of type a can be lifted to type $(a \to b) \to b$.
    - **Argument raising:** Any expression of type $a \to c$ can be lifted to type $((a \to b) \to b) \to c$.

Example


S
NP
the force
VP
V
strengthens
$\lambda x \lambda y.((strengthen\ x)\ y) :: e \to (e \to t)$
$\lambda \mathcal{P} \lambda y.(\mathcal{P}\ \lambda z.((strengthen\ z)\ y))$
$:: ((e \to t) \to t) \to (e \to t)$
NP
every Jedi
$\lambda P.\forall x.(jedi\ x) \to (P\ x)$
$:: (e \to t) \to t$

Solution 2: Extracting quantifiers

- **Quantifier raising**
- The same effect can be achieved by a semantic rule, e.g. Montague's **Quantifying in** rule. Here is our version of it:

$$\textbf{VP} \rightarrow \textbf{TV} \quad \textbf{NP}$$
$$[\![\textbf{VP}]\!] = \lambda y.([\![\textbf{NP}]\!]\ \lambda x.(([\![\textbf{TV}]\!]\ x)\ y))$$

Interpreting our grammar and implementation

The picture

string $\longrightarrow$ tree structure $\longrightarrow$ meaning representation

The picture

string $\longrightarrow$ tree structure $\longrightarrow$ meaning representation

We will now consider tree structures and how to map them to expressions of typed lambda calculus.

```
module Day3 where

import Day2 hiding (Tree(..))
```

Our grammar

$$
\begin{aligned}
\textbf{S} &::= \textbf{NP} \quad \textbf{VP} \\
\textbf{NP} &::= \textbf{NAME} \mid \textbf{DET} \quad \textbf{N} \mid \textbf{DET} \quad \textbf{RN} \\
\textbf{ADJ} &::= \textit{happy} \mid \textit{drunken} \mid \textit{evil} \\
\textbf{NAME} &::= \textit{Atreyu} \mid \textit{Dorothy} \mid \textit{Goldilocks} \mid \textit{Snow White} \\
\textbf{N} &::= \textit{boy} \mid \textit{princess} \mid \textit{dwarf} \mid \textit{wizard} \mid \textbf{ADJ} \quad \textbf{N} \\
\textbf{RN} &::= \textbf{N} \quad \textbf{REL} \quad \textbf{VP} \mid \textbf{N} \quad \textbf{REL} \quad \textbf{NP} \quad \textbf{TV} \\
\textbf{REL} &::= \textit{that} \\
\textbf{DET} &::= \textit{some} \mid \textit{every} \mid \textit{no} \\
\textbf{VP} &::= \textbf{IV} \mid \textbf{TV} \quad \textbf{NP} \mid \textbf{DV} \quad \textbf{NP} \quad \textbf{NP} \\
\textbf{IV} &::= \textit{cheered} \mid \textit{laughed} \mid \textit{shuddered} \\
\textbf{TV} &::= \textit{admired} \mid \textit{helped} \mid \textit{defeated} \mid \textit{found} \\
\textbf{DV} &::= \textit{gave}
\end{aligned}
$$

Tree structures

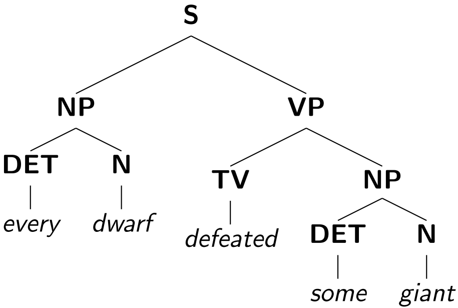
A **parse tree** for a string generated by a grammar G is a tree where:

- The root is the start symbol for G.
- The interior nodes are nonterminals of G and the children of a node N correspond to the symbols on the right hand side of some production rule for T in G.
- The leaf nodes are terminal symbols of G.

Every string generated by a grammar has a corresponding parse tree that illustrates a derivation for that string.

Example

Every dwarf defeated some giant.



Parse trees

A parse tree is either a leaf with information, or a branch with information dominating a list of trees.

```
data Tree a b = Leaf a | Branch b [Tree a b]   deriving Show
```

Parse trees

A parse tree is either a leaf with information, or a branch with information dominating a list of trees.

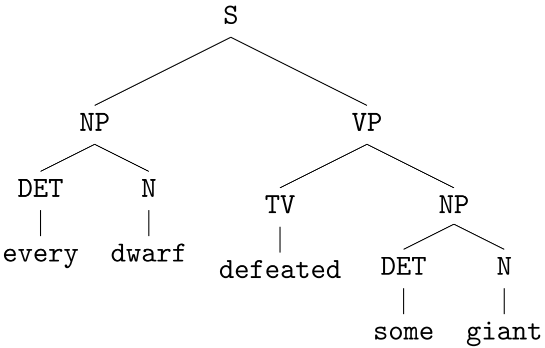
```
data Tree a b = Leaf a | Branch b [Tree a b]   deriving Show
```

Example:

```
tree :: Tree String String
tree = Branch "S" [Branch "NP" [Branch "DET" [Leaf "every"],
                                Branch "N"   [Leaf "dwarf"]],
                   Branch "VP" [Branch "TV"
                                          [Leaf "defeated"],
                                Branch "NP"
                                 [Branch "DET" [Leaf "some"],
                                  Branch "N"   [Leaf "giant"]]]]
```

Example

```
tree :: Tree String String
tree = Branch "S" [Branch "NP" [Branch "DET" [Leaf "every"],
                                Branch "N"   [Leaf "dwarf"]],
                   Branch "VP" [Branch "TV"  [Leaf "defeated"],
                                Branch "NP" [Branch "DET" [Leaf "some"],
                                             Branch "N"   [Leaf "giant"]]]]
```



Implementation

We define a mapping from parse trees to lambda expressions by recursion over the structure of a parse tree. For a sentence tree it will return the analogue of the predicate logical formula representing the meaning of this sentence.

General type: `Tree String String ->` $f(\tau)$, where

- $f(\tau_1 \rightarrow \tau_2) = f(\tau_1)$ `->` $map(\tau_2)$
- $f(e) =$ `Term`
- $f(t) =$ `Formula`

Interpretation and implementation

$$[\![\mathrm{S}]\!] :: t$$

$$\mathbf{S} \rightarrow \mathbf{NP} \quad \mathbf{VP} \qquad [\![\mathrm{S}]\!] = ([\![\mathrm{NP}]\!] \quad [\![\mathrm{VP}]\!])$$

```
transS :: Tree String String -> Formula
transS (Branch "S" [np,vp]) = (transNP np) (transVP vp)
```

Interpretation and implementation

$$[\![\text{NP}]\!] :: (e \to t) \to t$$

$$\textbf{NP} \to \textbf{NAME} \quad [\![\text{NP}]\!] = [\![\text{NAME}]\!]$$

$$\textbf{NP} \to \textbf{DET} \ \ \textbf{N} \quad [\![\text{NP}]\!] = ([\![\text{DET}]\!] \ \ [\![\text{N}]\!])$$

$$\textbf{NP} \to \textbf{DET} \ \ \textbf{RN} \quad [\![\text{NP}]\!] = ([\![\text{DET}]\!] \ \ [\![\text{RN}]\!])$$

```
transNP :: Tree String String -> (Term -> Formula) -> Formula
transNP (Branch "NP" [name])   = transNAME name
transNP (Branch "NP" [det,n@(Branch "N" _)])   =
                               (transDET det) (transN n)
transNP (Branch "NP" [det,rn@(Branch "RN" _)]) =
                               (transDET det) (transRN rn)
```

Interpretation and implementation

$$
\begin{array}{rl}
 & [\![\text{NAME}]\!] :: (e \to t) \to t \\
\textbf{NAME} \to \textit{Atreyu} & [\![\text{NAME}]\!] = \lambda P.(P\ a) \\
\textbf{NAME} \to \textit{Goldilocks} & [\![\text{NAME}]\!] = \lambda P.(P\ b) \\
\textbf{NAME} \to \textit{Dorothy} & [\![\text{NAME}]\!] = \lambda P.(P\ d)
\end{array}
$$

```
transNAME :: Tree String String -> (Term -> Formula)
                                -> Formula
transNAME (Branch "NAME" [Leaf "Atreyu"]) =
                                        \ p -> p (Const "a")
transNAME (Branch "NAME" [Leaf "Goldilocks"]) =
                                        \ p -> p (Const "b")
transNAME (Branch "NAME" [Leaf "Dorothy"]) =
                                        \ p -> p (Const "d")
```

Interpretation and implementation

$$[\![\mathrm{N}]\!] :: e \rightarrow t$$

$$\mathbf{N} \rightarrow \textit{wizard} \quad [\![\mathrm{N}]\!] = \lambda x.(\textit{wizard}\ x)$$

$$\ldots$$

$$\mathbf{N} \rightarrow \mathbf{ADJ}\ \ \mathbf{N} \quad [\![\mathrm{N}]\!] = \lambda x.(([\![\mathrm{ADJ}]\!]\ x) \wedge ([\![\mathrm{N}]\!]\ x))$$

```
transN :: Tree String String -> Term -> Formula
transN (Branch "N" [Leaf "wizard"]) =
                                \ x -> (Atom "wizard" [x])
transN (Branch "N" [Leaf "giant"]) =
                                \ x -> (Atom "giant" [x])
transN (Branch "N" [Leaf "princess"]) =
                                \ x -> (Atom "princess" [x])
transN (Branch "N" [Leaf "dwarf"]) =
                                \ x -> (Atom "dwarf" [x])
transN (Branch "N" [adj,n]) =
                      \ x -> Conj [transADJ adj x,transN n x]
```

Interpretation and implementation

$$[\![\text{ADJ}]\!] :: e \rightarrow t$$

$$\textbf{ADJ} \rightarrow \textit{happy} \quad [\![\text{ADJ}]\!] = \lambda x.(\textit{happy } x)$$

$$\ldots$$

```
transADJ :: Tree String String -> Term -> Formula
transADJ (Branch "ADJ" [Leaf "happy"]) =
                                \ x -> Atom "happy" [x]
transADJ (Branch "ADJ" [Leaf "drunken"]) =
                                \ x -> Atom "drunken" [x]
transADJ (Branch "ADJ" [Leaf "evil"])  =
                                \ x -> Atom "evil" [x]
```

Interpretation and implementation

$$[\![\mathrm{RN}]\!] :: e \to t$$

$$\textbf{RN} \to \textbf{N REL VP} \quad [\![\mathrm{RN}]\!] = \lambda x.([\![\mathrm{N}]\!]\ x) \wedge ([\![\mathrm{VP}]\!]\ x)$$

$$\textbf{RN} \to \textbf{N REL NP TV} \quad [\![\mathrm{RN}]\!] = \lambda x.([\![\mathrm{N}]\!]\ x) \wedge ([\![\mathrm{NP}]\!]\ \lambda y.(([\![\mathrm{TV}]\!]\ y)\ x))$$

```
transRN :: Tree String String -> Term -> Formula
transRN (Branch "RN" [n,rel,vp])    =
          \ x -> Conj [(transN n x),(transVP vp x)]
transRN (Branch "RN" [n,rel,np,tv]) =
          \ x -> Conj [(transN n x),(transNP np (\ y -> (t
```

Interpretation and implementation

$$[\![\text{VP}]\!] :: e \rightarrow t$$

$$\textbf{VP} \rightarrow \textbf{IV} \quad [\![\text{VP}]\!] = [\![\text{IV}]\!]$$

$$\textbf{VP} \rightarrow \textbf{TV} \;\; \textbf{NP} \quad [\![\text{VP}]\!] = \lambda y.([\![\text{NP}]\!] \; \lambda x.(([\![\text{TV}]\!] \; x) \; y))$$

$$\textbf{VP} \rightarrow \textbf{DV} \;\; \textbf{NP} \;\; \textbf{NP} \quad [\![\text{VP}]\!] = \lambda z.([\![\text{NP}]\!] \; \lambda y.([\![\text{NP}]\!] \; \lambda x.((([\![\text{TV}]\!] \; x) \; y) \; z)))$$

```
transVP :: Tree String String -> Term -> Formula
transVP (Branch "VP" [iv]) = transIV iv
transVP (Branch "VP" [tv,np]) =
  \ y -> (transNP np)  (\ x -> (transTV tv) x y)
transVP (Branch "VP" [dv,np1,np2]) =
  \ z -> (transNP np1) (\ y -> (transNP np2)
                                 (\ x -> (transDV dv) x y z))
```

Interpretation and implementation

$$[\![\mathrm{IV}]\!] :: e \rightarrow t$$

$$\mathbf{IV} \rightarrow \textit{cheered} \quad [\![\mathrm{IV}]\!] = \lambda x.(\textit{cheered}\ x)$$

$$\ldots$$

```
transIV :: Tree String String -> Term -> Formula
transIV (Branch "IV" [Leaf "cheered"]) =
                                   \ x -> Atom "cheer" [x]
transIV (Branch "IV" [Leaf "laughed"]) =
                                   \ x -> Atom "laugh" [x]
transIV (Branch "IV" [Leaf "shuddered"]) =
                                   \ x -> Atom "shudder" [x]
```

Interpretation and implementation

$$[\![\text{TV}]\!] :: e \to (e \to t)$$

$$\textbf{TV} \to \textit{admired} \quad [\![\text{TV}]\!] = \lambda x \lambda y.((\textit{admire}\ x)\ y)$$

$$\ldots$$

```
transTV :: Tree String String -> Term -> Term -> Formula
transTV (Branch "TV" [Leaf "admired"])  =
                        \ x y -> Atom "admire" [y,x]
transTV (Branch "TV" [Leaf "helped"])   =
                        \ x y -> Atom "help"   [y,x]
transTV (Branch "TV" [Leaf "defeated"]) =
                        \ x y -> Atom "defeat" [y,x]
transTV (Branch "TV" [Leaf "found"])    =
                        \ x y -> Atom "find"   [y,x]
```

Interpretation and implementation

$$[\![\text{DV}]\!] :: e \to (e \to (e \to t))$$

$$\textbf{DV} \to \textit{gave} \quad [\![\text{TV}]\!] = \lambda x \lambda y \lambda y.(((\textit{give}\ x)\ y)\ z)$$

```
transDV :: Tree String String -> Term -> Term -> Term
                                        -> Formula
transDV (Branch "DV" [Leaf "gave"]) =
                \ x y z -> Atom "give" [z,y,x]
```

Interpretation and implementation

$$[\![\text{DET}]\!] :: (e \rightarrow t) \rightarrow (e \rightarrow t) \rightarrow t$$

$$\textbf{DET} \rightarrow \textit{every} \quad [\![\text{DET}]\!] = \lambda P \lambda Q.\forall x.((P\ x) \rightarrow (Q\ x))$$

$$\textbf{DET} \rightarrow \textit{some} \quad [\![\text{DET}]\!] = \lambda P \lambda Q.\exists x.((P\ x) \rightarrow (Q\ x))$$

$$\textbf{DET} \rightarrow \textit{no} \quad [\![\text{DET}]\!] = \lambda P \lambda Q.\neg\exists x.((P\ x) \rightarrow (Q\ x))$$

```
transDET :: Tree String String -> (Term -> Formula)
                                -> (Term -> Formula)
                                -> Formula
transDET (Branch "DET" [Leaf "every"]) p q =
         Forall i (Impl (p (Var i)) (q (Var i)))
                                where i = fresh [p,q]
transDET (Branch "DET" [Leaf "some"])  p q  =
         Exists i (Conj [p (Var i),q (Var i)])
                                where i = fresh [p,q]
transDET (Branch "DET" [Leaf "no"])    p q =
         Neg (Exists i (Conj [p (Var i),q (Var i)]))
                                where i = fresh [p,q]
```

Fresh variables

...`where i = fresh [p,q]`

```
fresh :: [Term -> Formula] -> Int
fresh xs | vars == [] = 1
         | otherwise  = 1 + maximum vars
    where
    vars = concat $ map (\ f -> getVars (f (Const "*"))) xs
```

Where `getVars :: Formula -> [Int]` collects all variables occurring in a formula.

Collecting variables

```
getVars :: Formula -> [Int]
getVars (Atom _ ts)   = concat $ map getVar ts
getVars (Neg  f)      = getVars f
getVars (Conj fs)     = concat $ map getVars fs
getVars (Disj fs)     = concat $ map getVars fs
getVars (Impl f1 f2) = getVars f1 ++ getVars f2
getVars (Forall n f) = n : getVars f
getVars (Exists n f) = n : getVars f

getVar :: Term -> [Int]
getVar (Const _) = []
getVar (Var   n) = [n]
```

Course overview

Day 2:
Meaning representations and (predicate) logic

Day 3:
Lambda calculus and the composition of meanings

- **Day 4:**
  Extensionality and intensionality
- **Day 5:**
  From strings to truth conditions and beyond