EBERHARD KARLS
UNIVERSITÄT
TÜBINGEN

Interpreting grammars

Or: How to model compositional meaning

- *The Language of Thought: computational cognitive science approaches to category learning*
- Who: Fausto Carcassi
- When: Sommer semester 2022

EBERHARD KARLS
UNIVERSITÄT
TÜBINGEN

Last time

- Last time we have seen how to define formal grammars
- Basically, now you can forget everything except for context-free grammars and probabilistic context-free grammars, cause that is what we'll use in the rest of the course.
- The basic idea is that we can use formal grammars to model parts of the LoT.
- However, remember that LoT expressions also have semantic properties, i.e. they have a meaning.
- And recall that it is compositional!
- This week, we'll look at the meaning of sentences generated by a context-free grammar.
- In particular, how can we formally specify a compositional interpretation.
- But before that, let's introduce some formalism…

EBERHARD KARLS
UNIVERSITÄT
TÜBINGEN

Writing down functions

- Whenever we write an expression, we can distinguish between ‘symbols’…
  - That refer to something specific
    - Things like 1, 2, 3, …, +, %, ‘every’, ‘some’, etc.
  - That do not refer to any specific thing
    - These are the variables. Usually represented by letters x, y, z, etc.
- For instance, the expression ‘x + 1’ contains a mix of both
  - Conceptually, there’s something ‘unfinished’ or ‘unsaturated’ about the expression
- An unsaturated expression can be used to define a *function*
- For instance, f(x) = x+1
- The big drawback of defining a function this way is that it only makes sense *in the context* of a definition
- However, we will want a way to write down functions without context
- e.g. ‘the function that takes an integer and returns that integer plus one’
- Without necessarily giving it a name like ‘f’

EBERHARD KARLS
UNIVERSITÄT
TÜBINGEN

The notation of λ-calculus

- λ-calculus is a notation to go from a (possibly unsaturated) expression to a *function*.
- Think about it in these terms: we need to specify….
  - Which variable is the input to the function
  - What is the output of the function given the variable
- Basically, the idea of λ-calculus is the following:
  - We start with an expression with some free variable
    - e.g. $x + 2$
  - Then, at the beginning of the expression we put λ followed by the variable we want to bind
    - e.g. $\lambda x. x + 2$
  - The resulting expression is a *function* from the variable we bound to the evaluated expression
- Basically, λ-calculus is a way of going from an expression with free variables to a function!

The notation of λ-calculus

- If we have more than one free variable in our expression, and we only bind one with a lambda, the result is going to still be unsaturated.
- E.g. x + y
- $\lambda x.\, x + y$ still has unsaturated symbol y.
- However, we can nest lambda expressions!
- For instance:
- $\lambda y.\, \lambda x.\, x + y$

The notation of λ-calculus

- More formally, here's a recursive definition of the set of well-formed formulas (wff) we care about.
- We start with
  - A set of (possibly unsaturated) expressions X
  - A set of variables V that can appear in X
- If $x \in X$, then x is a wff
- If α is a wff and $v \in V$, $\lambda v.\alpha$ is a wff (this is called **lambda abstraction**)
- NOTE: This is not quite usual lambda calculus. I am giving you a non-default version because we are not going to use the 'full power' of lambda calculus.
- If you are interested in more, see e.g. the SEP entry on lambda calculus.

λ-calculus

- This approach for defining functions did not specify what shape the expressions inside the lambda expressions can be.
- Basically in what 'language' we are writing the unsaturated expressions.
- For instance, we could use English to define the function from an individual to True if the individual is a bird and False otherwise:
  - $\lambda x.\, x$ is a bird
- In practice, we will use slightly different languages depending on the fragment of the LoT we are looking at.
- They will be mostly variations of predicate logic.
- The important thing about the language we'll see is that we can give an explicit semantics for them.

EBERHARD KARLS
UNIVERSITÄT
TÜBINGEN

The notation of λ-calculus

- We have discussed lambda abstraction as a way of notating functions.
- Lambda calculus also has 'operations' to deal with applying an argument to a function.
- Consider for instance $\lambda x.P(x)$
- We can write $\big(\lambda x.P(x)\big)N$, and this means that we are applying argument N to function $\lambda x.P(x)$.
- Then we can use so-called β-reduction: we go from an expression $\big(\lambda x.P(x)\big)N$ to the result of substituting N in every occurrence of x, and removing the lambda. In this case this would result in: $P(N)$
- Basically, it represents the fact of calling function α with argument β.

EBERHARD KARLS
UNIVERSITÄT
TÜBINGEN

Example of β reduction

Suppose we have the following expression:

- $\Big(\big((\lambda x.\lambda y.x(y))\lambda z.P(z)\big)\,a\Big)\,b$

We substitute the lambda term $\lambda z.P(z)$ into $\lambda x.\lambda y.x(y)$:

- $\Big(\big(\lambda y.\lambda z.P(z)(y)\big)a\Big)\,b$

We substitute a into $\lambda z.P(z)(a)$

- $(\lambda z.P(z)(a))b$

Finally, we substitute b into $\lambda z.P(z)(a)$:

- $P(b)(a)$

Type theory

- For reasons that will become clear soon, we associate each of our expressions with a *type*.
- Let's define the set of types:
  - e and t are types
  - If σ and τ are types, then $< \sigma, \tau >$ is a type
  - Nothing else is a type
- And how to interpret them:
  - *e* refers to the set of individuals
  - *t* refers to the set of truth values
  - $< \sigma, \tau >$ refers to the set of functions from objects of type σ to objects of type τ

Type theory

Let's consider some expressions and what type they are:

- $\lambda x.P(x)$, where P is a predicate and x an individual.
  - $< e, t >$
- $\lambda x.\lambda y.Q(x, y)$, where Q is a predicate with two arguments and x and y individuals
  - $< e, < e, t \gg$
- $\lambda X.X(a)$, where a is an individual and X is a predicate
  - $\ll e, t >, t >$
- $\lambda X.\lambda Y.X(a) \wedge Y(a)$, where a is an individual and X and Y predicates
  - $\ll e, t >, \ll e, t >, t \gg$
- Basically, it can get as complicated as you want!

Type theory

In order to keep things tidy, we can put domain restrictions after a colon.

Therefore, we can write the type of each argument of a lambda function as follows:

- $\lambda x{:}\, x \in e.\, P(x)$, where P is a predicate
- $\lambda x{:}\, x \in e.\, \lambda y{:}\, y \in e.\, Q(x, y)$, where Q is a predicate with two arguments
- $\lambda X{:}\, X \in < e, t >.\, X(a)$, where a is an individual
- $\lambda X{:}\, X \in < e, t >.\, \lambda Y{:}\, Y \in < e, t >.\, X(a) \wedge Y(a)$

Sometimes, you'll also see the type written as a subfix:

- $\lambda x e.\, P(x)$

EBERHARD KARLS
UNIVERSITÄT
TÜBINGEN

Semantics

- All this stuff with lambda calculus and types is all well and good, but how does it help us?
- The point here is that we can use it to build a compositional interpretation function!
- This is a function that for each symbol in our grammar gives us the ‘meaning’ of that symbol.
- That meaning is an object of one of our semantic types.
  - For instance, it can give us an individual (e.g. Mary)
  - Or it can give us a function from individuals to truth values
  - Or a function from functions to functions
  - Etc.

An example of interpretation

- As a simple example, consider propositional logic
- This is the grammar for a simple version of propositional logic (just giving you the rules):
  - S → p | q | (S ∧ S) | (S ∨ S) | ¬S
- Note that while you are familiar with the meaning of these symbols, they are still not specified in the grammar!
- Now we can define an interpretation function that gives us the reference of each terminal symbol in the grammar in the actual world @:
  - $I_{@}(p)$ = True
  - $I_{@}(q)$ = False
  - $I_{@}(\wedge) = \lambda x.\lambda y.x\ is\ True\ and\ y\ is\ True$
  - $I_{@}(\vee) = \lambda x.\lambda y.x\ is\ True\ or\ y\ is\ True$
  - $I_{@}(\neg) = \lambda x.x\ is\ False$

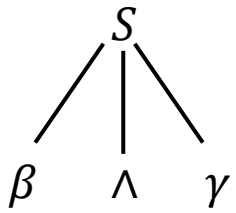
EBERHARD KARLS
UNIVERSITÄT
TÜBINGEN

Interpretation function

- This is very nice, but we're not quite there yet.
- The problem is that we want our interpretation function to be *compositional*.
- Therefore, it should:
  - Give us the meaning of complex expressions
  - Consider how the constituents are put together
- Basically, we still need to say how it should interpret *parse trees*, not just symbols.
- Let's see how we can do this.
- (I'll ignore brackets from now on because if we work with parse trees we don't need them)

EBERHARD KARLS
UNIVERSITÄT
TÜBINGEN

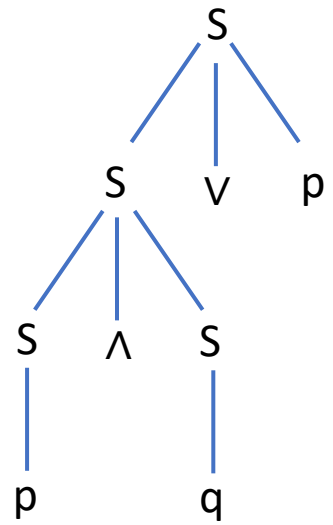
Compositional interpretation

- Now we have an interpretation function for the basic symbols
- But how do we extend it to complex structured sentences?
- The idea is that our interpretation function doesn't take symbols in, but rather parse trees or subtrees.
- We can introduce a rule as follows:
- If α has the form 

then $\mathrm{I}(\alpha) = (\mathrm{I}(\wedge)(\mathrm{I}(\beta))\ \mathrm{I}(\gamma))$
- Can you tell what the rules are for the other entries?
- Note that since our interpretation returns lambda expressions, we can β reduce!

Example of interpretation

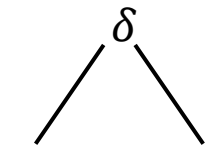
- For example, consider the following tree:



- Can you do a step-by-step derivation of the denotation of the whole tree?

Minimize number of rules

- We'll see in the second part of the course that the general tendency is only have a single rule for combining meanings, namely functional application:



- If α has the form then $I(\alpha) = I(\beta)(I(\gamma))$ or $I(\alpha) = I(\gamma)(I(\beta))$, whichever makes sense with the types.
- Rather than having complicated rules that depend on specific symbols appearing in the string, we just have few really general rules and we make sure to give interpretations to the single entries that work with those rules.

EBERHARD KARLS
UNIVERSITÄT
TÜBINGEN

Denotation vs meaning

- Note that this interpretation function in a sense does not give us the *meaning* of an expression, but it gives us the *denotation* at a world: what the expression refers to at a possible world.
- However, the general premise of formal semantics is that meaning of an expression is its *intension*: the function from possible worlds to denotation in that world.
- For example, the meaning of the word ‘duck’ is the function from each possible world to the set of ducks in that world.
- Therefore, we can go from our interpretation function to the meaning of an expression by abstracting over the world.
- This way of modeling meaning might seem completely bonkers, but there are very good reasons for doing it this way.
- However, you don’t need to worry about it now!

Note: sets vs functions

- You might have noticed now that there is a correspondence between sets and functions.
- Namely, the characteristic function of a set is the function from objects to True if the object belongs to the set and False otherwise.
- For instance, what is the characteristic function of the set {x s.t. x is a person | x is tall} ?
  - It's the function from persons to True if the person is tall and False otherwise.
  - In lambda notation: $\lambda x\text{: } x \textit{ is a person}.\, x \textit{ is tall}$
  - (Note that we haven't defined a type of 'persons', but we could have!)

EBERHARD KARLS
UNIVERSITÄT
TÜBINGEN

Conclusions

- This week, we have seen (very roughly and condensed-ly) how to build a compositional interpretation function for context-free grammars
- We have used several new tools to do this:
  - Lambda calculus
  - Type theory
  - Recursive interpretation rules
- Now there's only one piece left before we can look at the probabilistic LoT models. Namely, Bayesian inference.
- That's what we are going to do for the next two weeks!