Introduction to Glue Semantics: class 1

Basic theory

Jamie Y. Findlay
University of Oslo
jamie.findlay@iln.uio.no

04/08/25

1 Motivations

- Many approaches to the syntax–semantics interface rely on a syntactic tree (at some level of derivation) to provide the input to the semantics.[1]

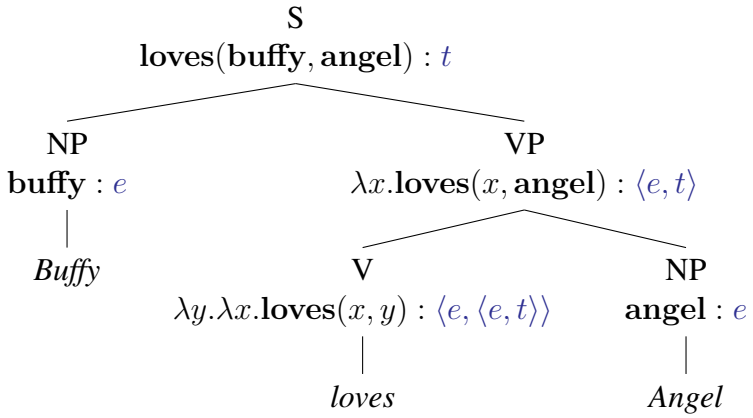


Figure 1: A Logical Form (LF)-style derivation

- But this has some undesirable consequences.
- Firstly, when sentences get more complex, it forces us to posit a number of syntactically-unmotivated objects and processes in the syntax, e.g.
  - The presence of traces/variables in the syntactic tree, or semantically-defined phrase-types like the 'lambda phrase' (LP) of Coppock & Champollion (2024: 298).
  - 'Covert' movement in the form of quantifier raising, reconstruction, etc.
- Such artefacts are undesirable from the point of view of the autonomy of syntax, and are wholly incompatible with a surface-oriented theory of syntax (such as Construction Grammar, Lexical Functional Grammar, Dependency Grammar, etc.).

[1] The (literal) textbook example of this approach is Heim & Kratzer (1998).

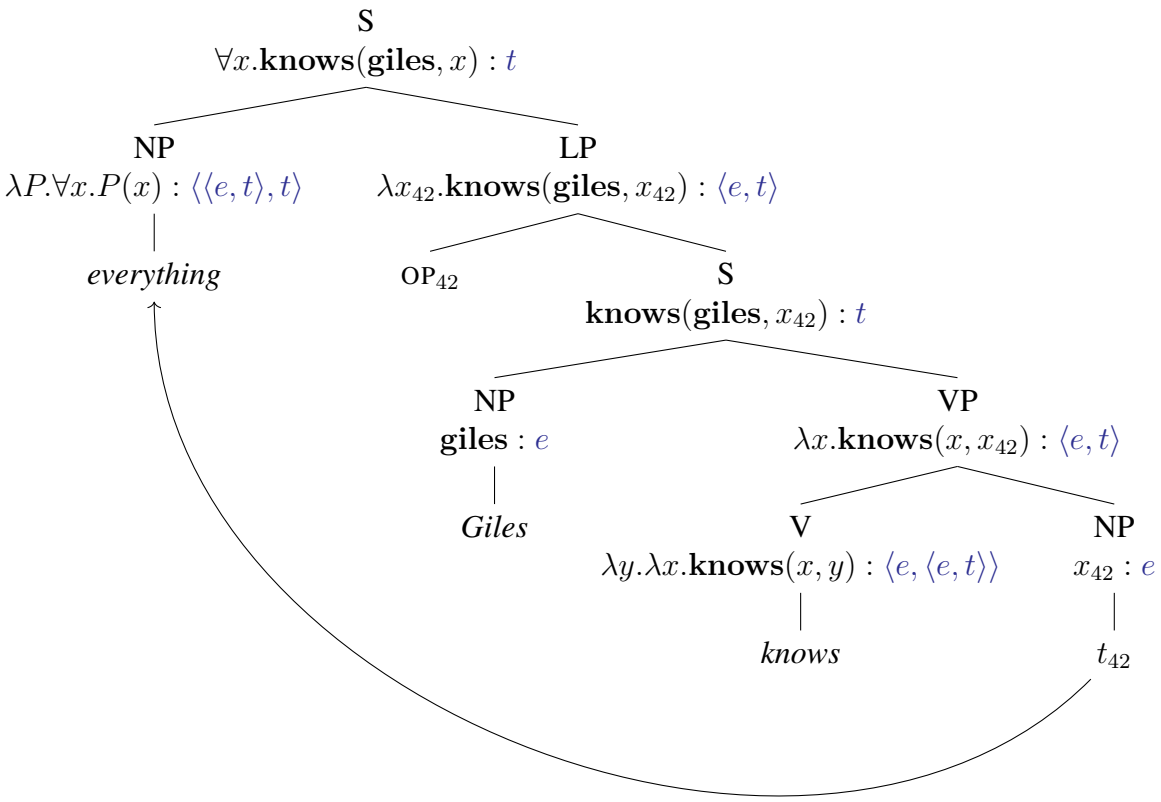


Figure 2: Quantifier raising and lambda abstraction in the tree

- Secondly, since we rely on the geometry of the tree to tell us the order of composition, this approach requires a view of syntax that assumes all languages have the same phrase structure underlyingly, even when there is no language-internal evidence for this.
- Most obviously, we require that all languages have a VP, even though the syntactic evidence is weak or non-existent for many languages (e.g. Warlpiri, Latin, German, . . .).
- Glue Semantics is an approach to the syntax–semantics interface that tries to avoid these potentially undesirable commitments and to remain agnostic as to the particular syntactic framework it is paired with.
- The 'LF' approach to semantic interpretation illustrated above essentially overloads the tree-based representation: not only does the tree represent constituency and linear order, but also abstract grammatical relations like subject and object, along with purely semantic information like where lambda abstraction takes place.
- For example, we want an expression like $\lambda y.\lambda x.\mathbf{love}(x, y)$ to combine with the loved thing first, and the lover second. In an active-voice English sentence, that means the object before the subject.
- In an LF approach, 'object' has no independent status, and is simply defined as the complement of V in the VP (or whatever more complex configuration is assumed in modern Minimalism).
- But this means that when the object *doesn't* appear in that position (or when there is no syntactic evidence for that position in a particular language, or when the thing appearing

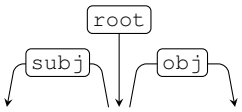
in that position has an incompatible type, or . . .), we need to do some fancy footwork to keep the semantic composition going smoothly.

- Glue takes a different approach: we equip meanings with explicit instructions about how their combinatorics works, and how they connect to the syntax – a connection that also does not have to map directly onto constituent structure.
- As long as we have some means of saying which syntactic element each semantic argument corresponds to, this leaves us free to pick whichever syntactic framework we prefer on syntactic grounds.[2]
- Glue is closely related to Categorial Grammar, but where CG treats the syntax of semantic composition as isomorphic to syntax proper, Glue is combined with a separate, independent theory of syntax, and therefore allows for mismatches between the two.
- This flexibility brings a number of advantages, which we will explore in Section 5.

2 A note on syntax

- For the sake of simplicity, we will assume a very rudimentary dependency grammar in this course.[3]
- Nothing about Glue relies on this choice, but it avoids us having having to spend the first half of the week teaching a new theoretical framework or reaching a consensus on what flavour of Minimalism we should be using(!)
- Dependency grammars take abstract grammatical relations to be the only relevant syntactic relations, and take such dependencies to hold between words.
  - Such theories therefore reject the existence of empty nodes or hidden/unpronounced elements.[4]
- The labels used for these relations vary, but they are often taken to constitute a superset of traditional grammatical terms like subject, object, oblique, etc.
- An example dependency parse for the sentence in Figure 1 is given in (1):

(1)



[2]Glue has most commonly been paired with Lexical Functional Grammar (see e.g. Asudeh 2023), but it has also been integrated with LTAG (Frank & van Genabith 2001), Categorial Grammar and Context-Free Grammar (Asudeh & Crouch 2001), HPSG (Asudeh & Crouch 2002), Minimalism (Gotham 2018), Universal Dependencies (Gotham & Haug 2018), and Dependency Grammar more broadly (Haug & Findlay 2023).

[3]Dependency grammar has a long tradition: see Tesnière (1959), Mel'čuk (1988), and Hudson (1984, 2007) for some representative examples. The version assumed here is based loosely on the Universal Dependencies framework (Nivre et al. 2020; de Marneffe et al. 2021), but with several divergences.

[4]This is one reason why dependency grammars have been particularly popular in computational linguistics and NLP: if we further require that each word can only bear one dependency, then there are clearly a finite number of ways to connect up all the words in a sentence, which makes the task computationally tractable. By contrast, if we are allowed to posit empty nodes, then there are, in principle, an infinite number of possible parses for any given sentence.

- The head word of the main clause is given the special label `root`, and other dependencies are represented as relations from one word, the HEAD, to another, the DEPENDENT.
- Each dependency relation from a given word is unique: a single word cannot have two `subj` dependents, for example.

Exercise 1: Thinking with dependencies

Give a dependency parse for the following sentences (use whatever labels seem sensible; the most important thing is to identify the head–dependent relations). Note: whatever is the main predicate of a clause will generally be considered the head, and in the main clause will therefore receive the `root` relation.

1. *Glory smirked.*
2. *Buffy is eating.*
3. *Giles is reading a book.*
4. *Xander spoke to Willow.*
5. *Spike is a vampire.*

3 Glue Semantics: the basic idea

- The basic building blocks of Glue Semantics are complex expressions called MEANING CONSTRUCTORS, which are associated with words in the lexicon or with syntactic constructions in the grammar.
- A meaning constructor consists of some meaning representation paired with an expression in a special kind of logic called LINEAR LOGIC.
- This logical side does two things:
  1. It controls/guides composition, analogously to the type system on the meaning side.
  2. It hooks up, or 'glues', the semantics to the syntax by using elements of the syntactic representation as terms in the logic.
- For the meaning side, we will assume (fairly standardly) a simply typed lambda calculus over a higher-order predicate logic.
- Here is an example meaning constructor for *loves*:

(2) $\lambda y.\lambda x.\mathbf{loves}(x, y) : E(\bullet\ \texttt{obj}) \multimap [E(\bullet\ \texttt{subj}) \multimap T(\bullet)]$

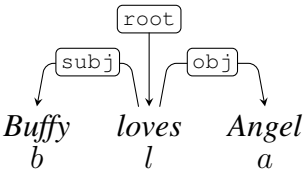
- We use the symbol '•' as a meta-variable that is resolved to whichever node the meaning constructor appears on.

- Intuitively speaking, we can think of the right-hand side of this expression as saying the following:

  > "If you give me a type e meaning corresponding to my object, then you give me a type e meaning corresponding to my subject, I will give you a type t meaning for myself."

- Let us look at each of the pieces in turn and see how this is achieved.
- On the left-hand side of the colon we have a standard translation of *loves* into a lambda-equipped predicate logic.
- On the right-hand side we have our linear logic expression. In examining this, it will be useful to be able to refer to an INSTANTIATED parse, so that we can replace the $\bullet$ symbol with the name of a specific node in the dependency tree.
- For example, we can label the nodes mnemonically:

(3)



- Now, since (2) represents the meaning contribution of *loves*, $\bullet$ in (2) refers to the node that *loves* occupies, namely l, and we can therefore rewrite (2) as (4):

(4) $\lambda y.\lambda x.\mathbf{loves}(x, y) : E(l\ \texttt{obj}) \multimap [E(l\ \texttt{subj}) \multimap T(l)]$

- Now let us examine the linear logic expression in more detail.
- 'E' and 'T' are predicates in this logic that are mnemonic for the types on the meaning side.
- 'l obj', 'l subj', and 'l' are terms which refer to nodes in the syntactic structure. They are PATHS which can be of arbitrary length, and describe a route through the dependency structure – for example, 'l obj' refers to the node that one reaches by starting at l and following the edge labelled 'obj'; in this case, that takes you to a. So we can further simplify the expression in (4) by resolving these paths fully:

(5) $\lambda y.\lambda x.\mathbf{loves}(x, y) : E(a) \multimap [E(b) \multimap T(l)]$

This shows the compositional instructions even more clearly: give me the meaning of *Angel*, then the meaning of *Buffy*, and I will give you the meaning of *Buffy loves Angel*.

- '$\multimap$' is the symbol for LINEAR IMPLICATION, the equivalent of classical logic's '$\rightarrow$' in linear logic; for now we can simply think of this as equivalent to the comma in type descriptions; this expression is analogous to the type of transitive verbs: $\langle e, \langle e, t\rangle\rangle$.
- Assuming the following meaning constructors for *Buffy* and *Angel*, this correctly achieves our goal of telling the function **loves** to first apply to **angel**, then to **buffy**:

(6) a. $\mathbf{buffy} : E(\bullet)$
$(\equiv \mathbf{buffy} : E(b))$

b. $\mathbf{angel} : E(\bullet)$
$(\equiv \mathbf{angel} : E(a))$

- We can show this in the following derivation, where each step corresponds to applying the function to one of its arguments, and then reducing the linear logic expression on the right-hand side of the meaning constructor accordingly:

$$\dfrac{\dfrac{\textbf{[}\textit{loves}\textbf{]}\quad \lambda y.\lambda x.\mathbf{loves}(x,y) : E(a) \multimap [E(b) \multimap T(l)] \qquad \textbf{[}\textit{Angel}\textbf{]}\quad \mathbf{angel} : E(a)}{\lambda x.\mathbf{loves}(x, \mathbf{angel}) : E(b) \multimap T(l)} \qquad \textbf{[}\textit{Buffy}\textbf{]}\quad \mathbf{buffy} : E(b)}{\mathbf{loves}(\mathbf{buffy}, \mathbf{angel}) : T(l)}$$

Figure 3: Derivation for the meaning of *Buffy loves Angel*

- One way of thinking about the linear logic component of meaning constructors is as a more fine-grained type system. Now the function **loves** not only says that it requires two type e arguments, but in fact that it requires two *specific* type e arguments, namely those corresponding to its object and subject.

Exercise 2: Meaning constructors

1. Thinking about the types of their translations into the lambda calculus, give meaning constructors for the following words:
   (a) *hates*
   (b) *sleeps*
   (c) *Giles*
   (d) *book*
   (e) *sister*
   (f) *everyone*
2. Give a derivation showing the composition of the following sentence:
   Spike hates Angel.

4 Linear logic

- We said that the right-hand side of a meaning constructor is an expression in a special kind of logic called linear logic.
- What is special about linear logic, and why couldn't we just use the classical predicate logic we're familiar with?
- Linear logic (Girard 1987) is special because it is RESOURCE SENSITIVE.

> "[Premises in linear logic] are not context-independent assertions that may be used or not[, but rather] *occurences* of information which are generated and used exactly once".
>
> (Dalrymple et al. 1999: 15)

- In classical logics, if something is true w.r.t. a model, it remains true no matter what deductions we might use it in.
- That is, from $p \rightarrow q$ and p, we can conclude not only q (via *modus ponens*) but also $q \wedge p$ (p remains true), or $q \wedge [p \rightarrow q]$, or just $p \rightarrow q$, or, indeed, any other of an infinite number of possible combinations.
- This is *not* true in linear logic. From $p \multimap q$ and p we can still conclude q, but in so doing we 'use up' the two premises, and they cannot then be used again.
- Intuitively, we can think of this like a vending machine. If a coin will buy us a chocolate bar, and we have a coin, then we can get a chocolate bar:

(7) 🪙 ⊸ 🍫
🪙

∴ 🍫

- We cannot, though, get a chocolate bar and still keep our coin:

(8) 🪙 ⊸ 🍫
🪙

∴ 🍫, 🪙

- Formally, linear logic lacks the structural rules of WEAKENING and CONTRACTION (Restall 2000).

(9) WEAKENING:

$$\frac{\Gamma \vdash B}{\Gamma, A \vdash B}$$

(premises can be freely added)

(10) CONTRACTION:

$$\frac{\Gamma, A, A \vdash B}{\Gamma, A \vdash B}$$

(premises can be freely discarded)

- To see why this is appropriate for our purposes, we must look a little more closely at how composition works in Glue Semantics.
- Rather than proceeding according to the syntactic tree and applying functions to their argument as we move up the tree, in Glue Semantics composition is a kind of logical deduction.
- We collect up all the meaning constructors contributed by the sentence, and then, looking purely at the linear logic sides, try to prove $T(\texttt{root})$ from those premises.
- As we proceed through our logical deduction, we also perform operations on the meaning side.
- It turns out that there is an exact correspondence between proofs in a logic and derivations in an algebraic system like the lambda calculus: the CURRY-HOWARD CORRESPONDENCE (Curry & Feys 1958; Howard 1980).
- For our purposes, only two specific correspondences are relevant:
  1. Implication elimination (i.e. *modus ponens*) in the logic corresponds to function application in the lambda calculus.

     I.e., for any function β with Glue type $A \multimap B$, and any expression α with Glue type A:

     $$\frac{\beta : A \multimap B \quad \alpha : A}{\beta(\alpha) : B} \multimap_{\mathcal{E}}$$

     This is what we have witnessed so far, in Figure 3, which we can see is actually a proof involving two instances of implication elimination, as spelled out in Figure 4.[5]

$$\dfrac{\dfrac{\dfrac{\lambda y.\lambda x.\mathbf{loves}(x,y) : E(a) \multimap [E(b) \multimap T(l)] \quad \textbf{[}\textit{\textbf{loves}}\textbf{]} \qquad \mathbf{angel} : E(a) \quad \textbf{[}\textit{\textbf{Angel}}\textbf{]}}{[\lambda y.\lambda x.\mathbf{loves}(x,y)](\mathbf{angel}) : E(b) \multimap T(l)} \multimap_{\mathcal{E}} \qquad \mathbf{buffy} : E(b) \quad \textbf{[}\textit{\textbf{Buffy}}\textbf{]}}{[\lambda y.\lambda x.\mathbf{loves}(x,y)](\mathbf{angel})(\mathbf{buffy}) : T(l)} \multimap_{\mathcal{E}}}{\mathbf{loves}(\mathbf{buffy},\mathbf{angel}) : T(l)} \Rightarrow_{\beta} \times 2$$

Figure 4: Glue proof for *Buffy loves Angel*

[5] Going forward, we will adopt the convention that unannotated proof steps are understood as instances of implication elimination/function application, and in general we will silently apply β-reduction.

2. Implication introduction (i.e. hypothetical reasoning) in the logic corresponds to lambda abstraction in the lambda calculus.

   I.e., for any expression ϕ with Glue type B, and any variable u with Glue type A,

$$\dfrac{\begin{array}{c}[u : A]^1 \\ \vdots \\ \phi : B\end{array}}{\lambda u.\phi : A \multimap B} \multimap_{\mathcal{I},1}$$

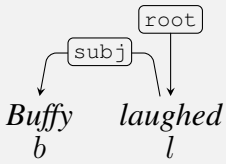
- This second correspondence mentions HYPOTHETICAL REASONING; this works as follows:
  - We introduce a hypothesis, indicated by square brackets in the proof and flagged with a unique identifier.
  - On the logic side, if we can then prove some conclusion B using the hypothesised A, we can 'discharge' the hypothesis by introducing an implication with the conclusion as its consequent and the hypothesis as its antecedent.
  - On the meaning side, this corresponds to lambda abstraction over the variable that was introduced as the left-hand side of the hypothesis.
  - All hypotheses must be discharged by the end of a proof.
- While its purpose might not be obvious now, we will see several uses for this rule below.
- Since semantic composition is driven by logical deduction, we need a logic that doesn't allow us to simply discard or re-use meanings, since this would incorrectly predict that certain meanings are optional or can be repeated.
- This is most significant when it comes to modifiers (which we will look at in more detail tomorrow): for instance, *mouse* has the same Glue type as *big mouse*, just $E(m) \multimap T(m)$, so in terms of its role in deriving the meaning for the whole sentence, nothing goes wrong if we ignore the modification altogether. Similarly for negation, which also has a modifier type.
- On the flip side, a *very, very big mouse* is bigger than a *very big mouse*, so we cannot allow modifiers to be repeated, either.
- If we are not allowed to have any resources left over, however, because we cannot discard them, and we also cannot re-use resources, then we correctly predict that meanings have to be used exactly once. This is why we need to use a resource-sensitive logic like linear logic for our composition logic.

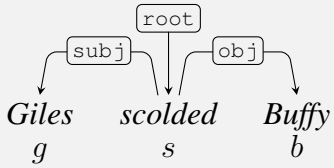
Exercise 3: Practising with Glue proofs

Give a Glue proof for each of the following sentences, showing the derivation of a meaning for the whole sentence. In each case, you are given a dependency parse, but must provide meaning constructors yourself.

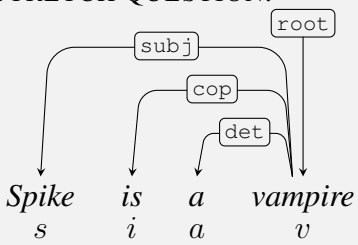
You may give instantiated meaning constructors without worrying about what the abstract lexical entries for the words would look like.

Do not necessarily assume that every word should contribute a meaning constructor.

1. 

2. 

3. STRETCH QUESTION:



5 Perks of the logic-based approach

5.1 Quantifier scope

- One of the major advantages of Glue as an approach to the syntax–semantics interface is its handling of quantifier scope ambiguity.
- As an example/reminder of quantifier scope ambiguity, consider the sentence in (11):

  (11) *Every student saw a vampire*.

- Figure 5 illustrates two possible states of affairs which are both compatible with the truth of sentence (11).
- From a single sentence we must therefore produce more than one meaning.

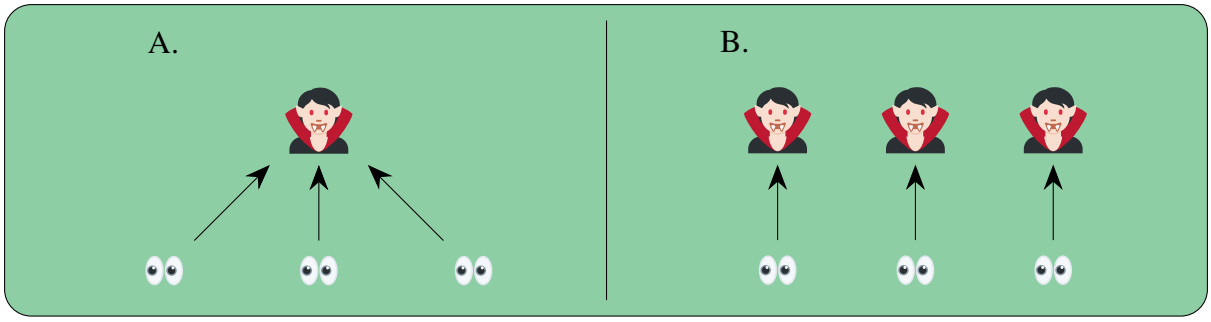


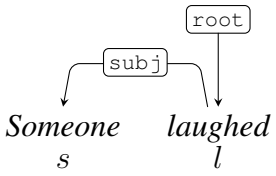
Figure 5: *Every student saw a vampire*: is there one vampire or more than one?

- Classically, ambiguities are thought to arise either from the lexicon or the syntax: either there are two senses associated with the same string of phonemes, or two syntactic structures associated with the same string of words.
- But (quantifier) scope ambiguities aren't obviously like either of these: the meanings of the words are the same in both cases, and there is no *syntactic* evidence for a different constituency or assignment of grammatical functions.
- In LF-based approaches (inspired by Montague), the solution is to claim that there actually *is* a syntactic ambiguity here, by invoking a rule of Quantifier Raising. But it is rather controversial whether there is any evidence for this structural difference beyond the interpretation facts it was introduced to explain.
- With Glue, quantifier scope ambiguities 'come for free': they are expected and natural, and do not require us to invent any new rules.
- To see how this works, we first need to see what a quantifier looks like in Glue Semantics.
- The meaning constructor for *someone* is given in (12):

(12) $\lambda P.\exists x.\mathbf{person}(x) \wedge P(x) : \forall\alpha.[E(\bullet) \multimap T(\alpha)] \multimap T(\alpha)$

- The Glue side mirrors the familiar quantifier type $\langle\langle e,t\rangle,t\rangle$; it looks for a dependency on its node and returns the output of that dependency.
- There is quantification over the argument of the T predicate since the quantifier does not know *what* is looking for its resource, but whatever would be produced by consuming it, the quantifier will provide.
- In a simple sentence with no quantifier scope ambiguities, this works as expected:[6]

(13)



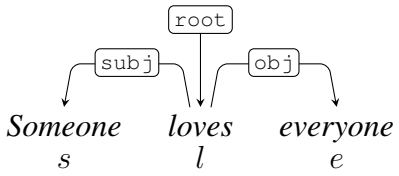
[6] I include the universal instantiation step explicitly in Figure 6, but going forward this will be carried out silently.

$$\dfrac{\dfrac{\textbf{[}\textbf{\textit{someone}}\textbf{]}\quad \lambda P.\exists x.\mathbf{person}(x) \wedge P(x) : \forall\alpha.[E(s) \multimap T(\alpha)] \multimap T(\alpha)}{\lambda P.\exists x.\mathbf{person}(x) \wedge P(x) : [E(s) \multimap T(l)] \multimap T(l)}\,\forall_{\mathcal{E}} \qquad \textbf{[}\textbf{\textit{laughed}}\textbf{]}\quad \lambda x.\mathbf{laughed}(x) : E(s) \multimap T(l)}{\exists x.\mathbf{person}(x) \wedge \mathbf{laughed}(x) : T(l)}$$

Figure 6: Glue proof for *Someone laughed*

- Here is the parse for a more complex sentence, *Someone loves everyone*:

(14)



- To generate the scope ambiguity, we do not need to posit any syntactic ambiguity. Rather, the ambiguity arises within the proofs themselves: from the same premises we can arrive at the conclusion in different ways, corresponding to different readings.
- This happens essentially because we can manipulate the order in which predicates combine with their arguments in the logic (see below for more on this), allowing quantifiers to scope in in different orders.
- Figures 7 and 8 illustrate this, using the instantiated meaning constructors below:

(15)
$$\begin{aligned} \textit{someone} &\rightsquigarrow \lambda P.\exists x.\mathbf{person}(x) \wedge P(x) : \forall\alpha.[E(s) \multimap T(\alpha)] \multimap T(\alpha) \\ \textit{loves} &\rightsquigarrow \lambda y.\lambda x.\mathbf{loves}(x, y) : E(e) \multimap [E(s) \multimap T(l)] \\ \textit{everyone} &\rightsquigarrow \lambda P.\forall y.\mathbf{person}(y) \rightarrow P(y) : \forall\alpha.[E(e) \multimap T(\alpha)] \multimap T(\alpha) \end{aligned}$$

- Whenever there is more than one scope-taking element present, the Glue approach, simply by the nature of the logic involved, predicts that they should interact, and all possible scope permutations should be permitted.

$$
\dfrac{\dfrac{\textbf{[}\textbf{\textit{loves}}\textbf{]}\quad \lambda y.\lambda x.\mathbf{loves}(x,y) : E(e) \multimap [E(s) \multimap T(l)] \qquad \left[\begin{array}{l} y : \\ E(e) \end{array}\right]^1}{\lambda x.\mathbf{loves}(x,y) : E(s) \multimap T(l)} \qquad \left[\begin{array}{l} x : \\ E(s) \end{array}\right]^2}{\mathbf{loves}(x,y) : T(l)}
$$

$$
\dfrac{\mathbf{loves}(x,y) : T(l)}{\lambda y.\mathbf{loves}(x,y) : E(e) \multimap T(l)} \multimap_{\mathcal{I},1}
$$

$$
\dfrac{\textbf{[}\textbf{\textit{everyone}}\textbf{]}\quad \lambda P.\forall y.\mathbf{person}(y) \to P(y) : [E(e) \multimap T(l)] \multimap T(l) \qquad \lambda y.\mathbf{loves}(x,y) : E(e) \multimap T(l)}{\forall y.\mathbf{person}(y) \to \mathbf{loves}(x,y) : T(l)}
$$

$$
\dfrac{\textbf{[}\textbf{\textit{someone}}\textbf{]}\quad \lambda P.\exists x.\mathbf{person}(x) \wedge P(x) : [E(s) \multimap T(l)] \multimap T(l) \qquad \lambda x.\forall y.\mathbf{person}(y) \to \mathbf{loves}(x,y) : E(s) \multimap T(l)}{\exists x.\mathbf{person}(x) \wedge \forall y.\mathbf{person}(y) \to \mathbf{loves}(x,y) : T(l)} \multimap_{\mathcal{I},2}
$$

Figure 7: Derivation of the surface scope reading of *Someone loves everyone*

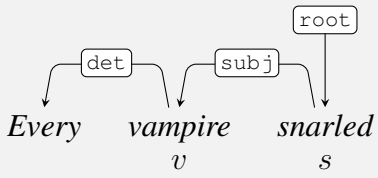
$$\dfrac{\dfrac{\textbf{[}\textit{loves}\textbf{]}}{\begin{array}{l}\lambda y.\lambda x.\mathbf{loves}(x,y):\\ E(e)\multimap[E(s)\multimap T(l)]\end{array}} \quad \left[\begin{array}{l} y:\\ E(e)\end{array}\right]^1}{\begin{array}{l}\lambda x.\mathbf{loves}(x,y):\\ E(s)\multimap T(l)\end{array}} \quad \textbf{[}\textit{someone}\textbf{]}\ \begin{array}{l}\lambda P.\exists x.\mathbf{person}(x)\wedge P(x):\\ [E(s)\multimap T(l)]\multimap T(l)\end{array}}{\dfrac{\exists x.\mathbf{person}(x)\wedge\mathbf{loves}(x,y):\ T(l)}{\begin{array}{l}\lambda y.\exists x.\mathbf{person}(x)\wedge\mathbf{loves}(x,y):\\ E(e)\multimap T(l)\end{array}}\ \multimap_{Z,1} \quad \textbf{[}\textit{everyone}\textbf{]}\ \begin{array}{l}\lambda P.\forall y.\mathbf{person}(y)\to P(y):\\ [E(e)\multimap T(l)]\multimap T(l)\end{array}}$$

$$\forall y.\mathbf{person}(y)\to\exists x.\mathbf{person}(x)\wedge\mathbf{loves}(x,y):\ T(l)$$

Figure 8: Derivation of the inverse scope reading of *Someone loves everyone*

Exercise 4: Quantificational determiners

Consider the example below:



We have seen the meaning constructor for a simple quantifier like *everyone*. A more complex quantifier like *every vampire* will have the same logical structure; here is its meaning constructor:

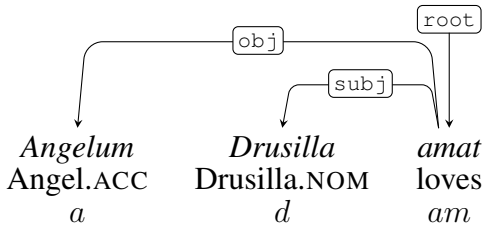
$$\lambda P.\forall x.\mathbf{vampire}(x) \rightarrow P(x) : \forall\alpha.[E(v) \multimap T(\alpha)] \multimap T(\alpha)$$

1. Given this, what should the meaning constructor for the determiner *every* be? What do we need to add to our description language to capture this?
2. Give a Glue proof showing the meaning of *Every vampire snarled* using the meaning constructor you came up with in part 1.
3. Give two Glue proofs showing both readings of *Every vampire killed a werewolf*.
4. STRETCH QUESTION:
   How many Glue proofs do we get for *Giles gave everyone a book*? How many should we get? (Note: You will have to come up with a sensible meaning constructor for ditransitive *give*. Use the relation `obj2` for its second object.)

5.2 Word order flexibility

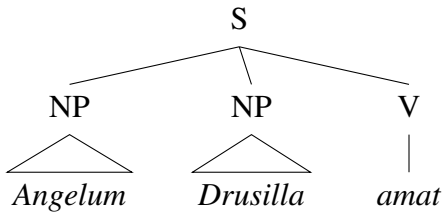
- Since we can 'glue' our meanings to the syntax by reference to things other than linear order or syntactic constituency, flexible word orders are no challenge.
- For example, consider the Latin example in (16):

(16)



- Here we have an OSV word order so that the verb and object don't even form a constituent, something which would be taken at face value in many surface-oriented frameworks, so that the constituent structure is flat:

(17)



- For Glue, no problem of composition even arises, assuming we have some way of identifying which NP is the subject and which the object of *amat*.
- That is, the expected meaning constructors for each word, shown in (18), will give us the correct meaning for the sentence.

(18) a. $\textit{amat} \rightsquigarrow \lambda y.\lambda x.\mathbf{loves}(x, y) : E(\bullet\ \texttt{obj}) \multimap [E(\bullet\ \texttt{subj}) \multimap T(\bullet)]$
b. $\textit{Angelum} \rightsquigarrow \mathbf{angel} : E(\bullet)$
c. $\textit{Drusilla} \rightsquigarrow \mathbf{drusilla} : E(\bullet)$

Exercise 5: Latin

1. Give the *instantiated* versions of the meaning constructors in (18), assuming the dependency parse in (16).
2. Give the Glue proof corresponding to the derivation of the meaning for the sentence as a whole.

5.3 Reordering arguments

- For students first studying formal semantics, it can seem fairly obtuse that the meaning of a transitive verb like *loves* is (19a) and not (19b); why should the lambda-bound arguments come in the opposite order from the list of arguments following the predicate?

(19) a. $\lambda y.\lambda x.\mathbf{loves}(x, y)$
b. $\lambda x.\lambda y.\mathbf{loves}(x, y)$

- Of course, writing $\mathbf{loves}(x, y)$ is already an abuse of notation, since what we really mean is that **loves** is a function that takes two arguments; if we wrote it in this functional format, the order of the arguments would match: $\lambda y.\lambda x.\mathbf{loves}(y)(x)$.
- But in a concession to readability, we often prefer to write the arguments in the order they occur in English, as if $\mathbf{loves}$ were a relation rather than a function.
- However, this flexibility only goes so far: when it comes to the order the arguments are presented in, an LF-based approach is constrained by the fact that, in English at least, the verb combines with its object before its subject.
- And this means the order of composition is the opposite of the preferred order of interpretation.

- This is an unfortunate consequence of relying too heavily on the surface syntactic structure to determine the order of composition, and can be avoided in a theory with a looser connection like Glue.
- Owing to the flexibility of the logical component of Glue, arguments can be freely reordered, and so we can write the meanings of predicates with their arguments in whichever order is preferred for other reasons:

$$\dfrac{\dfrac{\dfrac{\lambda y.\lambda x.f(x,y) : A \multimap B \multimap C \qquad [y : A]^1}{\lambda x.f(x,y) : B \multimap C} \qquad [x : B]^2}{f(x,y) : C}}{\dfrac{\lambda y.f(x,y) : A \multimap C}{\lambda x.\lambda y.f(x,y) : B \multimap A \multimap C}\ \multimap_{\mathcal{I},2}}\ \multimap_{\mathcal{I},1}$$

Figure 9: Lambda-bound arguments can be reordered freely

- Going forward, I will present the arguments of a predicate in whichever order makes the most sense given the context, without explicitly showing the reordering steps in the proof.

> **Exercise 6: Reordering arguments**
>
> Given that we can silently reorder the arguments of a predicate, there is a shorter way of writing the proof in Figure 7. What is it?

5.4 Type raising

- It is often convenient/necessary to manipulate the types of certain expressions.
- For example, we sometimes want names to have the type of quantifiers, $\langle\langle e, t\rangle, t\rangle$, rather than the type of individuals, e; for example, when they are coordinated with quantifiers:

  (20) [[*Angel*] *and* [*two other vampires*]] *appeared.*

- But we don't *always* want them to have this type (*pace* Montague 1973), since they sometimes behave differently from quantifiers, e.g. with respect to introducing new discourse referents:

  (21) a. *Willow passed the exam. She was very pleased.*
  b. *Every girl passed the exam. #She was very pleased.*

- So we need a process that 'raises' a type e thing into a type $\langle\langle e, t\rangle, t\rangle$ thing (see e.g. Partee & Rooth 1983; Partee 1987).

- In many approaches, this must be explicitly introduced as a new rule in the compositional system, but in Glue it once again 'comes for free', since type raising is a theorem in the logic:

$$\dfrac{\dfrac{\dfrac{[x : E(\alpha)]^1 \qquad [P : E(\alpha) \multimap T(\beta)]^2}{P(x) : T(\beta)}}{\lambda P.P(x) : [E(\alpha) \multimap T(\beta)] \multimap T(\beta)} \multimap_{\mathcal{I},2}}{\lambda x.\lambda P.P(x) : E(\alpha) \multimap [[E(\alpha) \multimap T(\beta)] \multimap T(\beta)]} \multimap_{\mathcal{I},1}$$
$$\dfrac{}{\lambda x.\lambda P.P(x) : \forall\alpha.\forall\beta.E(\alpha) \multimap [[E(\alpha) \multimap T(\beta)] \multimap T(\beta)]} \forall_{\mathcal{I}} \times 2$$

Figure 10: Type raising is a theorem in Glue Semantics

- Figure 11 shows an example of type raising the proper name *Buffy* (using b to stand for whatever node *Buffy* occupies in the parse).

$$\dfrac{\begin{array}{l}\textbf{[}\textbf{\textit{Buffy}}\textbf{]} \\ \mathbf{buffy} : E(b)\end{array} \qquad \dfrac{\begin{array}{c}\vdots \\ \lambda x.\lambda P.P(x) : \forall\alpha.\forall\beta.E(\alpha) \multimap [[E(\alpha) \multimap T(\beta)] \multimap T(\beta)]\end{array}}{\lambda x.\lambda P.P(x) : \forall\beta.E(b) \multimap [[E(b) \multimap T(\beta)] \multimap T(\beta)]} \forall_{\mathcal{E}}}{\lambda P.P(\mathbf{buffy}) : \forall\beta.[E(b) \multimap T(\beta)] \multimap T(\beta)}$$

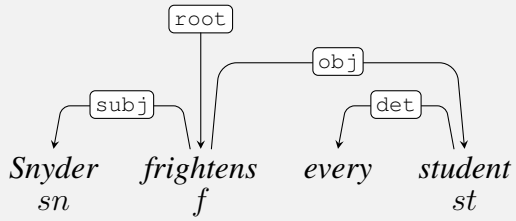
Figure 11: An example of type raising *Buffy*

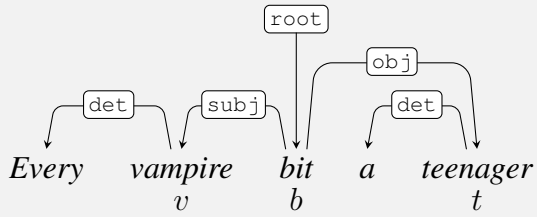
6 Further reading

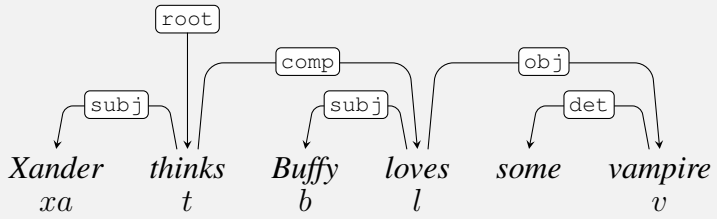
- Asudeh (2022) and Asudeh (2023) are good, recent introductions to Glue Semantics.
- Dalrymple et al. (2019: ch. 8) offers a different style of introduction, but one that relies on knowledge of Lexical Functional Grammar (LFG).
- If you're interested in learning more about LFG, Falk (2001) is an introductory textbook aimed at those coming from a GB background; Bresnan et al. (2016) and Börjars et al. (2019) are more recent textbook presentations; and Asudeh & Toivonen (2015) and Dalrymple & Findlay (2019) are shorter, handbook-style introductions.
- Dalrymple et al. (2019) is the most comprehensive modern exposition of LFG, and includes a number of Glue analyses of various phenomena.
- Dalrymple (2023) is a handbook which contains more in-depth explorations of various topics in LFG, including Glue Semantics *per se* as well as analyses that make use of Glue.

Exercise 7: Further practice

Give Glue proofs for the following sentences. You will need to provide appropriate meaning constructors.

1. 

2. 

3. 

References

Asudeh, Ash. 2022. Glue Semantics. *Annual Review of Linguistics* 8. 321–341. https://doi.org/10.1146/annurev-linguistics-032521-053835.

Asudeh, Ash. 2023. Glue semantics. In Mary Dalrymple (ed.), *The handbook of Lexical Functional Grammar* (Empirically Oriented Theoretical Morphology and Syntax 13), 651–697. Language Science Press. https://doi.org/10.5281/zenodo.10185964.

Asudeh, Ash & Richard Crouch. 2001. Glue semantics: a general theory of meaning composition. Talk given at Stanford Semantics Fest 2, March 16 2001.

Asudeh, Ash & Richard Crouch. 2002. Glue semantics for HPSG. In Frank Van Eynde, Lars Hellan & Dorothee Beermann (eds.), *Proceedings of the 8th International Conference on Head-Driven Phrase Structure Grammar*, 1–19. Stanford, CA: CSLI Publications. http://web.stanford.edu/group/cslipublications/cslipublications/HPSG/2001/Ash-Crouch-pn.pdf.

Asudeh, Ash & Ida Toivonen. 2015. Lexical-Functional Grammar. In Bernd Heine & Heiko Narrog (eds.), *The Oxford handbook of linguistic analysis* (2nd edn.), 373–406. Oxford: Oxford University Press. https://doi.org/10.1093/oxfordhb/9780199677078.013.0017.

Börjars, Kersti, Rachel Nordlinger & Louisa Sadler. 2019. *Lexical-Functional Grammar: an introduction*. Cambridge: Cambridge University Press. https://doi.org/10.1017/9781316756584.

Bresnan, Joan, Ash Asudeh, Ida Toivonen & Stephen Wechsler. 2016. *Lexical-functional syntax* (2nd edn.). Oxford: Wiley-Blackwell. https://doi.org/10.1002/9781119105664.

Coppock, Elizabeth & Lucas Champollion. 2024. *Invitation to formal semantics*. https://eecoppock.info/bootcamp/semantics-boot-camp.pdf.

Curry, Haskell B. & Robert Feys. 1958. *Combinatory logic: volume I*. Amsterdam: North Holland.

Dalrymple, Mary (ed.). 2023. *Handbook of Lexical Functional Grammar* (Empirically Oriented Theoretical Morphology and Syntax 13). Berlin: Language Science Press. https://doi.org/10.5281/zenodo.10037797.

Dalrymple, Mary & Jamie Y. Findlay. 2019. Lexical Functional Grammar. In András Kertész, Edith Moravcsik & Csilla Rákosi (eds.), *Current approaches to syntax: a comparative handbook*, 123–154. Berlin: De Gruyter Mouton. https://doi.org/10.1515/9783110540253-005.

Dalrymple, Mary, John Lamping, Fernando Pereira & Vijay Saraswat. 1999. Overview and introduction. In Mary Dalrymple (ed.), *Semantics and syntax in Lexical Functional Grammar: the resource logic approach*, Cambridge, MA: MIT Press.

Dalrymple, Mary, John J. Lowe & Louise Mycock. 2019. *The Oxford reference guide to Lexical Functional Grammar*. Oxford: Oxford University Press. https://doi.org/10.1093/oso/9780198733300.001.0001.

Falk, Yehuda N. 2001. *Lexical-Functional Grammar: an introduction to parallel constraint-based syntax*. Stanford, CA: CSLI Publications.

Frank, Anette & Josef van Genabith. 2001. GlueTag: linear logic based semantics for LTAG—and what it teaches us about LFG and LTAG. In Miriam Butt & Tracy Holloway King (eds.), *Proceedings of the LFG01 Conference*, 104–126. Stanford, CA: CSLI Publications. http://web.stanford.edu/group/cslipublications/cslipublications/LFG/6/pdfs/lfg01frankgenabith.pdf.

Girard, Jean-Yves. 1987. Linear logic. *Theoretical Computer Science* 50(1). 1–102. https://doi.org/10.1016/0304-3975(87)90045-4.

Gotham, Matthew. 2018. Making Logical Form type-logical: Glue semantics for Minimalist syntax. *Linguistics and Philosophy* 41(5). 511–556. https://doi.org/10.1007/s10988-018-9229-z.

Gotham, Matthew & Dag T. T. Haug. 2018. Glue semantics for Universal Dependencies. In Miriam Butt & Tracy Holloway King (eds.), *Proceedings of the LFG18 Conference*, 208–226. Stanford, CA: CSLI Publications. https://web.stanford.edu/group/cslipublications/cslipublications/LFG/LFG-2018/lfg2018-gotham-haug.pdf.

Haug, Dag T. T. & Jamie Y. Findlay. 2023. Formal semantics for Dependency Grammar. In *Proceedings of the Seventh International Conference on Dependency Linguistics (Depling 2023)*, 22–31. Association for Computational Linguistics. https://aclanthology.org/2023.depling-1.3/.

Heim, Irene & Angelika Kratzer. 1998. *Semantics in generative grammar* (Blackwell Textbooks in Linguistics 13). Oxford: Blackwell.

Howard, William A. 1980. The formulae-as-types notion of construction. In *To H. B. Curry: essays on combinatory logic, lambda calculus, and formalism*, 479–490. London: Academic Press. Circulated in unpublished form from 1969.

Hudson, Richard. 1984. *Word Grammar*. Oxford: Blackwell.

Hudson, Richard. 2007. Word Grammar. In Dirk Geeraerts & Hubert Cuyckens (eds.), *The Oxford handbook of cognitive linguistics*, 509–540. Oxford: Oxford University Press. https://doi.org/10.1093/oxfordhb/9780199738632.013.0019.

de Marneffe, Marie-Catherine, Christopher D. Manning, Joakim Nivre & Daniel Zeman. 2021. Universal Dependencies. *Computational Linguistics* 47(2). 255–308. https://doi.org/10.1162/coli_a_00402.

Mel'čuk, Igor A. 1988. *Dependency syntax: theory and practice*. Albany, NY: State University of New York Press.

Montague, Richard. 1973. The proper treatment of quantification in ordinary English. In K. Jaakko, J. Hintikka, Julius M. E. Moravcsik & Patrick Suppes (eds.), *Approaches to natural language: proceedings of the 1970 Stanford workshop on grammar and semantics* (Synthese Library 49), 221–243. Dordrecht: Reidel.

Nivre, Joakim, Marie-Catherine de Marneffe, Filip Ginter, Jan Hajic, Christopher Manning, Sampo Pyysalo, Sebastian Schuster, Francis Tyers & Daniel Zeman. 2020. Universal Dependencies v2: an evergrowing multilingual treebank collection. In *Proceedings of the 12th International Conference on Language Resources and Evaluation (LREC 2020)*, 4034–4043. Marseille: European Language Resources Association. https://aclanthology.org/2020.lrec-1.497.

Partee, Barbara. 1987. Noun phrase interpretation and type-shifting principles. In Jeroen Groenendijk, Dick de Jongh & Martin Stokhof (eds.), *Studies in Discourse Representation Theory and the theory of generalized quantifiers*, 115–143. Dordrecht: Foris.

Partee, Barbara & Mats Rooth. 1983. Generalized conjunction and type ambiguity. In Rainer Bäuerle, Christoph Schwarze, & Arnim von Stechow (eds.), *Meaning, use, and interpretation of language*, 361–383. Berlin: Walter de Gruyter.

Restall, Greg. 2000. *An introduction to substructural logics*. London: Routledge.

Tesnière, Lucien. 1959. *Éléments de syntaxe structurale*. Paris: Klincksieck.