Relative clauses

Ling 331 / 731
Spring 2016

In this lecturelet, we will :

- discuss relative clauses
- introduce variable binding,
- develop a semantics for relative clauses
- introduce predicate abstraction to compose binders

1 Relative clause syntax

A relative clause is an embedded clause that modifies an entity. In English, they are marked in three ways.

1. A *wh-* pronoun at the relative clause's left edge.

   (1) *I met a doctor [who you know].*

2. A complementizer at the relative clause's left edge.

   (2) *A doctor [that you know] is outside.*

3. No apparent marking.

   (3) *A doctor [you know] is outside.*

Relative clauses are linked to a **head** noun, which is outside the clause.

(4) *I met a doctor [who you know].*

At least in English…Many languages leave the head noun inside the clause. These clauses are known as **internally-headed relative clauses**. Kiowa is one such language (although the head *can* be extraposed).

Kiowa

(5) *[Áugàu chégùn bấò ∅− ál−hêl] =dè gà bṓ.*
SUB dog cat 3S<3S− chase.PFV−EVID =REL 1S>3S− see.PFV
'I saw the dog that (I heard) chased the cat'
'I saw the cat that (I heard) chased the dog'

Some languages (like Korean) allow both kinds of relative clauses. We will focus on the English type, though, which are called **externally-headed relative clauses**.

Many languages allow right-edge extraposition of a relative clause. We will ignore this.

(6) *I gave her [[some professor]$_{DP}$'s card]$_{DP}$ [who works with me].*

(7) *Susan brought up [an issue]$_{DP}$ at the meeting yesterday [that nobody had thought of before].*

German (Pochmann & Wagner 2015)

(8) *[$_{DP}$ Jeder Wanderer] hat das Riemannhaus erreicht, [$_{CP}$ der Schneeschuhe trug.]*
every hiker has the Riemannhaus reached who snow shoes wore
'Every hiker who was wearing snow shoes has reached the Riemannhaus'

Another distinction often made concerns the role of the head in the embedded clause.

Subject relatives have a head that is the subject of the relative clause.

(9) *There's [a doctor [who knows you]] waiting over there.*

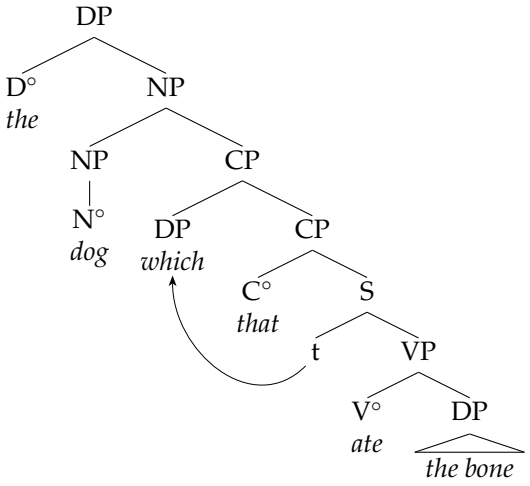
Object relatives have a head that is the object of the relative clause.

(10) *There's [a doctor [who(m) you know]] waiting over there.*

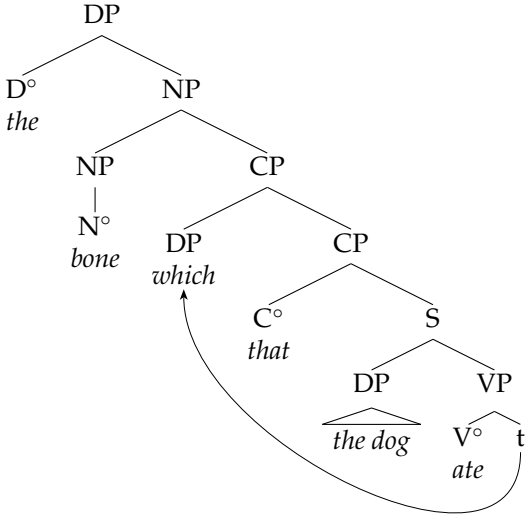
Externally-headed relative clauses are built through *wh-* movement to a CP projection (or $\overline{\text{S}}$ in older varieties). The CP adjoins to the head NP. (see Adger & Ramchand 2006 for a discussion of Irish relative clauses that can use base-generation instead of movement).

Inside the CP, we will include both the relative pronoun (*which, who*) and the complementizer (*that*), even though one or both will not be pronounced. That non-pronunciation is a problem for the phonological spell-out.

(11) The dog which/that ate the bone



(12) The bone the dog ate



2 Semantics of relative clauses

The semantics of an externally-headed relative clause is fairly simple. It acts like a modifying property.

(13) a. *The crepe which had Nutella was delicious.*
The crepe, which had the property of having Nutella, was delicious
b. *The crepe with Nutella was delicious.*

(14) a. *The delicious crepe had Nutella.* (Not the foul-tasting one)
b. *The crepe that was delicious had Nutella*.
The crepe, which had the property of being delicious, had Nutella

Relative clauses can be recursively stacked:

(15) *The crepe which had Nutella that I made yesterday was delicious.*

A relative clause can also be either restrictive or non-restrictive (see Potts (2005) for a finer-grained distinction):

(16) *The man who heads the Catholic Church is from Argentina.*

(17) *Pope Francis, who heads the Catholic Church, is from Argentina.*

Prescriptivists will claim that *which* is used only with non-restrictive relatives, and *that* is used only with restrictive ones.

We will again focus on restrictive relative clauses.

If a relative clause denotes a property, then it takes an argument and gives you 1 if the argument has the property. That is, if it's in the right set of things.

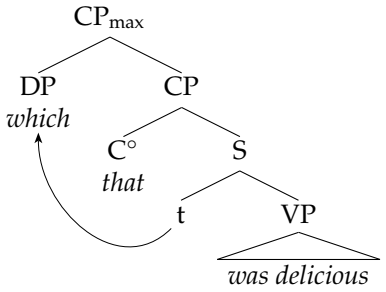
(18) $[\![\textit{which had Nutella}]\!] = [\![\textit{with Nutella}]\!] = \lambda x \in D_e.\ \text{had(Nutella)(x)} = \{ x \in D_e \mid x \text{ had Nutella} \}$

(19) $[\![\textit{which was delicious}]\!] = [\![\textit{delicious}]\!] = \lambda x \in D_e.\ \text{tastiness(x)} \leq d = \{ x \in D_e \mid x \text{ was delicious} \}$

So now to our question: If a relative clause (a CP) and an adjective (an AP) mean the same thing…how does that work in the composition?

Let's look at the syntax of a simpler clause. Remember that this is the LF, so the deletion of *which* or *that* is not an issue. They are both present in the semantics.

(20) *The [$_{NP}$ crepe [$_{CP}$ which was delicious]]*



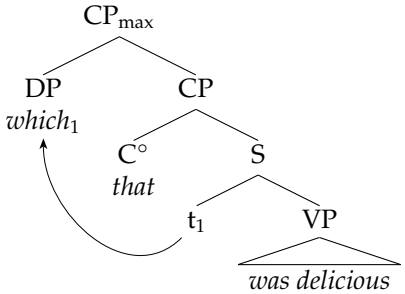
The CP_{max} and the VP mean the same thing.

(21) $[\![CP_{max}]\!] = \lambda x \in D_e.\ \text{delicious(x)} : \langle e, t \rangle$

(22) $[\![VP]\!] = \lambda x \in D_e.\ \text{delicious(x)} : \langle e, t \rangle$

What does the trace mean? It's a terminal node, but it is not lexically defined. Its meaning is tied to the meaning of what left it behind. How do we tie link syntactic objects together? Co-indexation, which also resolves a number of syntactic facts. Traces are co-indexed to a binding DP.

(23) *The [$_{NP}$ crepe [$_{CP}$ which$_1$ was t_1 delicious]]*

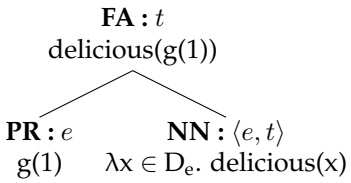


Well, we know how to interpret terminal nodes with indices: Via the Pronoun Rule. So we'll need an assignment. Let's pick ... g. The Pronoun Rule gets a meaning of g(1) for the trace t_1, and that's of type e.[1]

(**??**) $[\![\text{VP}]\!]^g = \lambda x \in D_e.\ \text{delicious}(x) : \langle e, t \rangle$

(24) $[\![t_1]\!]^g = g(1) : e$

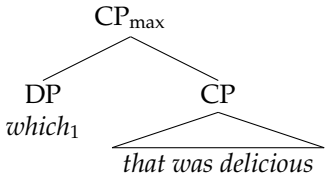
We can plug the latter into the former:

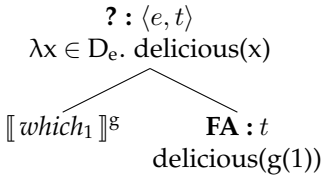


This leaves us with a saturated proposition. Uh oh. The complementizer, for now, does nothing. We can leave it 'vacuous' (without any meaning), or give it an identity function meaning:

(25) $[\![\textit{that}_C]\!] = \lambda p \in D_t.\ p : \langle t, t \rangle$

In either case, it just gives you what you had, so now we have the following structure.





How do we get from g(1) is delicious (of type t), compose it with *which*, and end up with $\lambda x \in D_e$.delicious(x)? We have to basically unsaturate a saturated expression. No rule we have can do this: FA saturates, PM and NN do not saturate, while LT and PR introduce new expressions.

$[\![\textit{which}_1]\!]^g$ cannot be introduced with LT, since its meaning is not lexically determined. We can't use PR, either, because that would get us (g(1)) of type e, and that cannot compose with delicious(g(1)) of type t.

Instead, we rely on the concept of **predicate abstraction**, by which you take an expression, and make a predicate out of it.

Abstraction takes an expression like $delicious(x)$ and gives you $\lambda x[delicious(x)]$. In a sense, we just tack on a λ-operator that binds the variable.

But wait!!! Our variable has an index: g(1). How do we turn g(1) into x?

To figure that out, let's start with what g(1) is. It's a pronoun, interpreted via PR, so its denotation is the output of plugging 1 into an assignment, g.

If g maps 1 to Tom, then g(1) = Tom... and *vice versa*: If g(1) = Tom, then g maps 1 to Tom. Likewise, if you find out that g(2) = Yolanda, then you can be sure that g maps 2 to Yolanda. And in the other direction, if you find out that g maps 3 to Renée, then you can be sure that g(3) = Renée.

So if we want g(1) to turn into x, we just make sure that g(1) *equals* x. If g(1) = x, then we can be sure that g maps 1 to x. So do we **make sure** that g(1) = x? We use assignment modification (see the lecturelet if you haven't).

Modfying an assignment directs the assignment to map a certain input to a certain output. So we can simply modify our assignment g to make sure that it maps 1 to x.

[1] Recall that the assignment function maps onto D_e.

(26) $[\![\textit{which}_1]\!]^{g^{1 \to x}} = g^{1 \to x}(1) = x$

Since $g^{1 \to x}(1)$ *equals* x, we can replace one with the other.

So now, we tack on a λx, and we've changed the variable to x:

(27) $\lambda x \in D_e$. delicious($g^{1 \to x}(1)$) =
$\lambda x \in D_e$. delicious(x)

Wizardry: We've added a λ-argument.

However, we now need a composition rule that can do all this.

Rule 6:

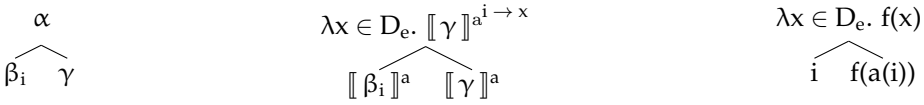
> **Predicate Abstraction (PA):**
> If α is a branching node whose daughters are { β_i, γ },
> such that β_i only dominates an index i, and i ∈ ℕ, then for any variable assignment a,
>
> $$[\![\alpha]\!]^a = \lambda x \in D_e. [\![\gamma]\!]^{a^{i \to x}}$$

READ: *. . . then for any variable assignment, a, the meaning of α under a equals lambda x in D-sub-e, the meaning of γ under assignment a, modified so that i maps to x.*

This rule applies no matter what 'x' is, or no matter what a or i are.

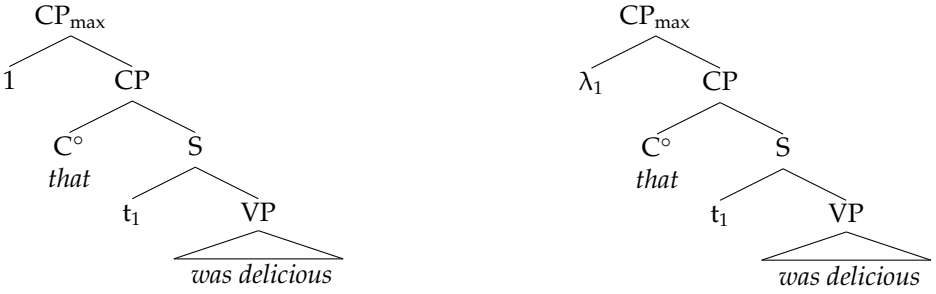
Schematically:

(28)



How does this work? At LF, we assume that the pronoun *which* is not a variable with an index, but rather nothing but the index[2]. Since the index will bind the trace, we say that it is a **binder index**. We often write it with a λ-operator to symbolize this binding.

(29)



So let's zoom in to the important part.

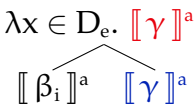
(30)

[2]This assumption creates problems that we'll deal with later

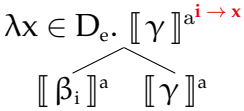
The PA rule operates in three steps.

1. **Abstraction** adds an argument by tacking on the λx expression.

2. **Value Copy** copies the sister of the binder as the value condition

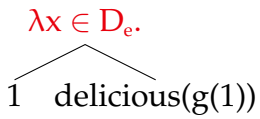


3. **Modify assignment** to map the index on the binder to the variable introduced by Abstraction. In this case, the index is i and the variable is x, so the modification is $i \rightarrow x$.

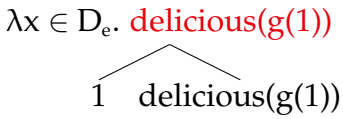


Let's try this with our example, which has index 1 and assignment g.

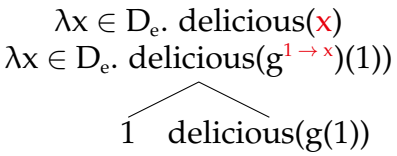
Abstraction



Value Copy



Modify Assignment



Look! We've wound up with our property.