The semantics of modality
Day 1: Introduction to truth-conditional semantics

Andrés Pablo Salanova
kaitire@uottawa.ca

I Escuela de Lingüística de Buenos Aires

15-19 February 2016

What does a sentence mean?

What is it to know the meaning of sentences such as the following?

(1) Snow is white.

(2) The acceleration of gravity at the surface of the Earth is 9,81 km/s².

(3) Ana looks happy.

One possibility, due to polish logician Alfred Tarski (1901-1983), is that knowing the meaning of these is equivalent to knowing under what conditions they are true.

Meaning as truth conditions

In Tarski's formulation:

(4) The sentence *Snow is white* is true iff snow is white.

In the notation that we'll be using here:

(5) $[\![\text{Snow is white}]\!] = 1$ iff snow is white

I may or may not know whether a particular statement is true or false. But if I know the meaning of a sentence I know what it takes for it to be true.

Compositionality

We'll start from this meaning for sentences, and work our way down to words in a compositional way. That is to say:

(6) Principle of compositionality: the meaning of a sentence is derived in a regular way from the meaning of its parts.

(7) Semantics tracks syntax: the hierarchical structure produced by syntax is the input to semantic interpretation.

Predication

A fundamental insight regarding the intepretation of simple sentences such as *Caesar conquered Gaul* is that they may be immediately split into two parts. One, complete unto itself, refers to an entity in the world. The other, "unsaturated", refers to a property.
This insight, formulated in this way by Frege but traceable to Aristotle, divides linguistic expressions into at least two types: *arguments* and *functions*. Functions are inherently unsaturated, and there will be one type of argument that serves to saturate them.

The meaning of a predicate

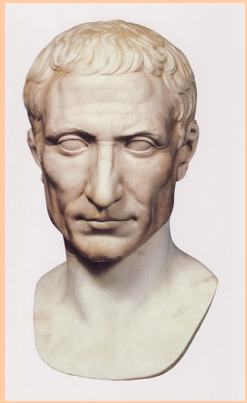
In other words:

(8) $[\![$conquered Gaul$]\!]$ = a function of x such that $[\![x$ conquered Gaul$]\!] = 1$ iff x conquered Gaul.

The lack of saturation can be "read off" from the fact that there is an open variable x in this expression's denotation.
Before we go any further, we need to decide what type of entity goes in x.

Entities

Following Richard Montague (1930-71), we admit one basic semantic type in addition to the semantic type that characterizes the meaning of sentences: entities. Entities are objects or individuals in the world. The simplest entity-denoting expression is a proper noun:



(9) $[\![\text{Caesar}]\!] = \text{Caesar}$

Caesar here is the individual called Caesar, i.e., him:

Basic semantic types

We thus have two basic semantic types in our ontology, entities and truth values. Until we begin discussing tense and modality, these two types are all we'll use to represent the meaning of linguistic expressions.

(10) Basic semantic types:

Entities, represented as e

Truth values, represented as t

Complex semantic types

How about the predicate "conquered Gaul"?
As we saw earlier, unsaturated expressions are treated as functions. This might come to mind:

(11) $$f(x) = \begin{cases} 1 \text{ iff } x \text{ conquered Gaul} \\ 0 \text{ otherwise} \end{cases}$$

The domain of this function is the set of entities, and the range is the set of truth values. In fact, one way to represent a function is as a set of ordered pairs representing mappings between the domain and the range, $< D, R >$.
So our function exemplifies our first complex semantic type:

(12) Complex semantic types:

Functions mapping individuals to truth values, $< e, t >$

An example

Suppose we live in a world where only three individuals exist: Caesar, Augustus, Nero.
We can fully specify our function f with respect to the individuals in this world. It will be like this, in a slightly different notation:

(13) $$f(x) = \begin{cases} \text{Caesar} & \rightarrow & 1 \\ \text{Augustus} & \rightarrow & 0 \\ \text{Nero} & \rightarrow & 0 \end{cases}$$

In our ordered pair notation, the function is a set:

(14) $$f = \{< \mathit{Caesar}, 1 >, < \mathit{Augustus}, 0 >, < \mathit{Nero}, 0 >\}$$

Functions and sets

Any function mapping entities to a truth value has a special relation to sets. Take for example:

(15) $$f(x) = \begin{cases} 1 \text{ iff } x \text{ is a woman} \\ 0 \text{ otherwise} \end{cases}$$

Another way to say this is that the function returns 1 if x belongs to the set of women, i.e.:

(16) $$f(x) = \begin{cases} 1 \text{ iff } x \in W \\ 0 \text{ otherwise} \end{cases}$$

where W is the set containing all women in the world

Characteristic function

We call a function such as (16) the *characteristic function* of the set W.
Our predicates so far are characteristic functions, and therefore they can also be thought of as sets.
Consider the following sentences:

(17) Chantal is a woman.

(18) Some candidates are women.

(19) Every guest is a woman.

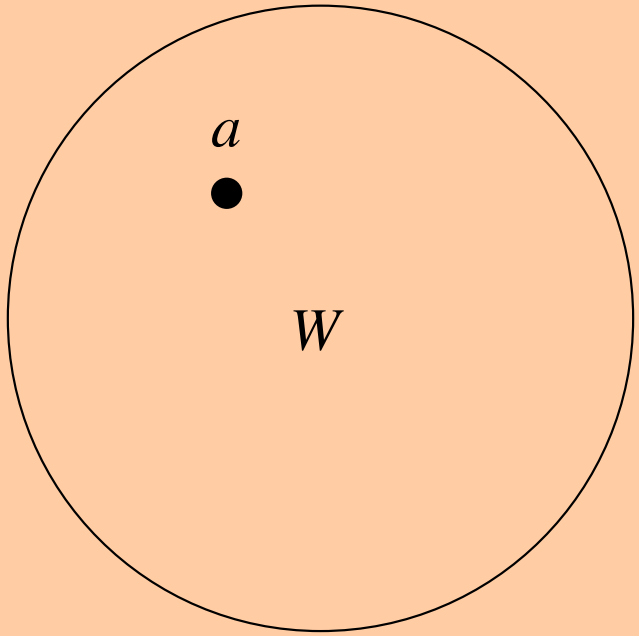
We will not deal with quantified noun phrases in this course, but I want to appeal to your intuition that these expressions all relate individuals or sets to other sets, in the following way:

Set membership

Chantal is a woman.

$a =$ Chantal

$W = \{x : x \text{ is a woman}\}$



$a \in W$

or

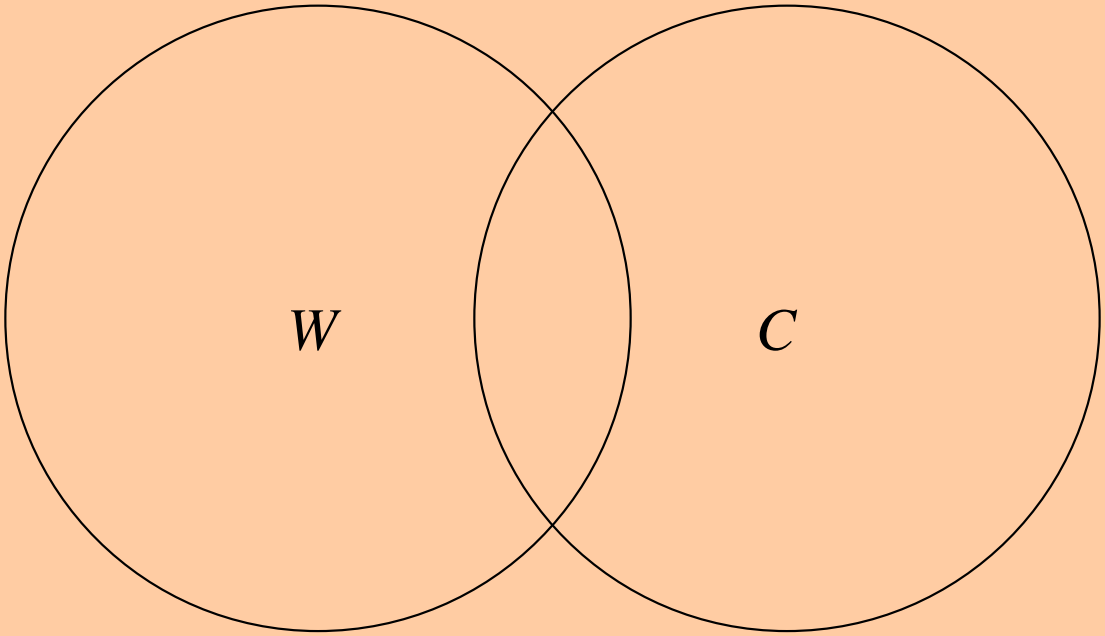
$W(a) = 1$

Non-empty intersection

Some candidates are women.

$C = \{x : x \text{ is a candidate}\}$

$W = \{x : x \text{ is a woman}\}$



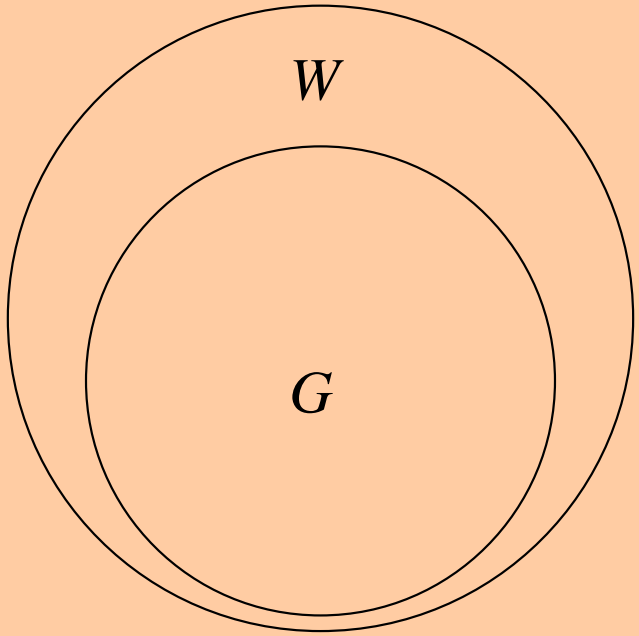
$$C \cap W \neq \emptyset$$

or

$$\exists x : C(x) = 1 \wedge W(x) = 1$$

Subset relation

Every guest is a woman.
$G = \{x : x \text{ is a guest}\}$
$W = \{x : x \text{ is a woman}\}$



$$G \subseteq W$$

or

$$\forall x : G(x) = 1 \rightarrow W(x) = 1$$

Functional application

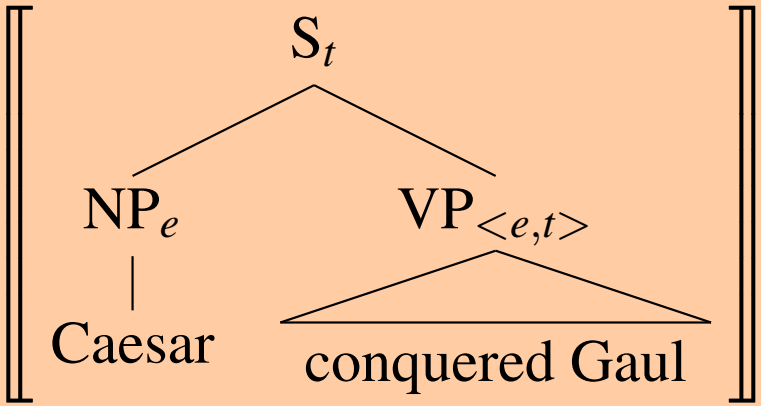
So, going back to the idea that predicates are functions of type $< e, t >$, sentences are of type t, and proper nouns are of type e, to work out the semantic composition of the sentence *Caesar conquered Gaul*, we need to define a rule of composition:

(20) Functional application: if a node α has daughters β of type $< D, R >$ and γ of type D, $[\![\alpha]\!]$ will be of type R, and have the meaning $[\![\beta]\!]([\![\gamma]\!])$.

What is essential here is that the types *fit*, as an argument-function pair: one type is the type of the other.

Functional application

So:

(21) 

$= [\![\text{conquered Gaul}]\!]([\![\text{Caesar}]\!])$

In terms of our set notation, in fact, we can define functional application as follows:

(22) Functional application: if a node α has daughters β of type $< D, R >$ and γ of type D, $[\![\alpha]\!]$ will be of type R, and have the meaning $[\![\gamma]\!] \in [\![\beta]\!]$.

Lambda notation for functions

Instead of the f or set notation that we introduced informally above, we will represent functions using the λ-notation developed by Alonzo Church:

(23) $[\![\text{conquered Gaul}]\!] = \lambda x.1 \leftrightarrow x \text{ conquered Gaul}$

That is, the function that returns 1 iff it is true that x conquered Gaul. For short:

(24) $[\![\text{conquered Gaul}]\!] = \lambda x.x \text{ conquered Gaul}$

λx is a *binder* in this formula: all open instances of x are bound by it and will be replaced by a constant when the function takes that constant as an argument. That operation is called *lambda conversion*, and you can practice it extensively with the Lambda calculator.

Lambda conversion

How we derive the meaning of the following?

(25) Caesar conquered Gaul.

Lambda conversion

How we derive the meaning of the following?

(26) Caesar conquered Gaul.

Functional application:

$$[\![\text{Caesar conquered Gaul}]\!] = [\![\text{conquered Gaul}]\!]([\![\text{Caesar}]\!])$$

Lambda conversion

How we derive the meaning of the following?

(27) Caesar conquered Gaul.

Functional application:

$$[\![\text{Caesar conquered Gaul}]\!] = [\![\text{conquered Gaul}]\!]([\![\text{Caesar}]\!])$$

Denotation of *Caesar*: $[\![\text{Caesar}]\!] =$ Caesar

Lambda conversion

How we derive the meaning of the following?

(28) Caesar conquered Gaul.

Functional application:

$$[\![\text{Caesar conquered Gaul}]\!] = [\![\text{conquered Gaul}]\!]([\![\text{Caesar}]\!])$$

Denotation of *Caesar*: $[\![\text{Caesar}]\!] = \text{Caesar}$

Denotation of *conquered Gaul*:

$$[\![\text{conquered Gaul}]\!] = \lambda x.1 \leftrightarrow x \text{ conquered Gaul}$$

Lambda conversion

How we derive the meaning of the following?

(29) Caesar conquered Gaul.

Functional application:

$[\![\text{Caesar conquered Gaul}]\!] = [\![\text{conquered Gaul}]\!]([\![\text{Caesar}]\!])$

Denotation of *Caesar*: $[\![\text{Caesar}]\!] = \text{Caesar}$

Denotation of *conquered Gaul*:

$[\![\text{conquered Gaul}]\!] = \lambda x.1 \leftrightarrow x \text{ conquered Gaul}$

Substitution of denotations

$[\![\text{conquered Gaul}]\!]([\![\text{Caesar}]\!]) = [\lambda x.x \text{ conquered Gaul}](\text{Caesar})$

Lambda conversion

How we derive the meaning of the following?

(30) Caesar conquered Gaul.

Functional application:

$[\![\text{Caesar conquered Gaul}]\!] = [\![\text{conquered Gaul}]\!]([\![\text{Caesar}]\!])$

Denotation of *Caesar*: $[\![\text{Caesar}]\!] = \text{Caesar}$

Denotation of *conquered Gaul*:

$[\![\text{conquered Gaul}]\!] = \lambda x.1 \leftrightarrow x \text{ conquered Gaul}$

Substitution of denotations

$[\![\text{conquered Gaul}]\!]([\![\text{Caesar}]\!]) = [\lambda x.x \text{ conquered Gaul}](\text{Caesar})$

Lambda-conversion:

$[\lambda x.x \text{ conquered Gaul}](\text{Caesar}) = 1 \leftrightarrow \text{Caesar conquered Gaul}$

Two-place predicates

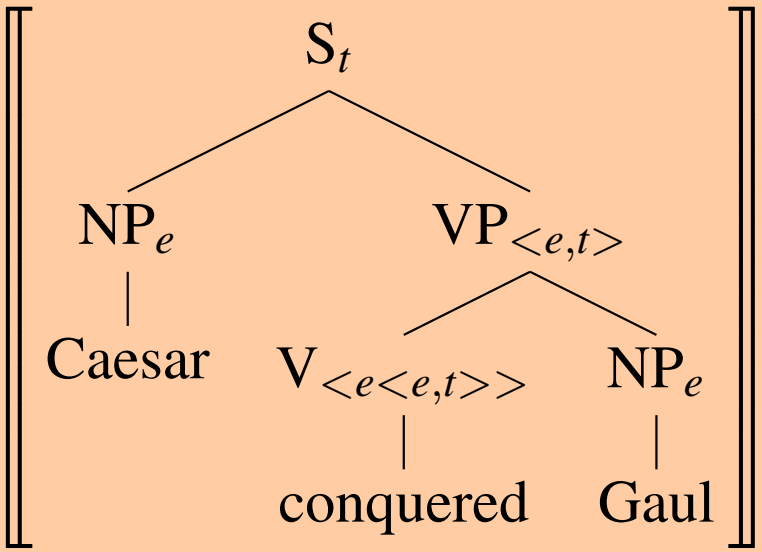
Caesar conquered Gaul is in fact a transitive verb, but we've been treating it as a simple predicate for simplicity. But of course we want to derive the semantics of *conquered Gaul* from *conquered* and *Gaul*
The type of a two-place verb has to contain two *e* arguments. Our syntax is binary branching, so:

The type of *see* is $< e, < e, t >>$.

The conversion of "flat argument structures" such as $<< e_1, ..., e_n >, t >$ to "layered" structures such as $< e_1, < e_2, ... < e_n, t > ... >>$ is called *Schönfinkelization*.

Two-place predicates

We can assume that *Gaul* is, like *Caesar* an entity. The only truly new thing in the following tree is the semantic type of the V:



$= [[\text{conquered}([\![\text{Gaul}]\!])]]\,([\![\text{Caesar}]\!])$

The reference of pronouns

Now take the following:

(31) He conquered Gaul.

The truth conditions of this sentence clearly depend on the reference of *he*.

(32) $[\![\text{He conquered Gaul}]\!] = 1 \leftrightarrow$ whoever *he* refers to conquered Gaul

Words such as *he* take their reference from the linguistic or extralinguistic context.
We'll deal with this by relativizing our denotation function to context:

(33) $[\![X]\!]^c =$ the denotation of X in context c.

For example, in a particular context:

(34) $[\![he]\!]^c =$ Caesar

Assignment functions

We'll be more precise about this. Since many things in a single utterance may depend on context, we'll assign a different index to each of these:

(35) He_1 thinks she_2 conquered Gaul.

Our *context* will be reduced to a contextual *assignment function*, g_c, which takes each index to its referent:

(36) $$g_c = \begin{cases} 1 & \rightarrow & \text{John} \\ 2 & \rightarrow & \text{Mary} \\ \ldots & & \end{cases}$$

So, given this particular definition of g_c:

(37) $[\![\text{He}_1]\!]^{g_c} = \text{John}$

Interpretation of bound pronouns

There are cases when these variable-reference elements are *bound* and their reference becomes fixed intra-sententially. Resumptive pronouns are one such case (but examples are complicated). PRO and traces are better examples:

(38) Caesar$_1$ wants PRO$_1$ to conquer Gaul.

(39) the Roman dictator$_1$ who t_1 conquered Gaul

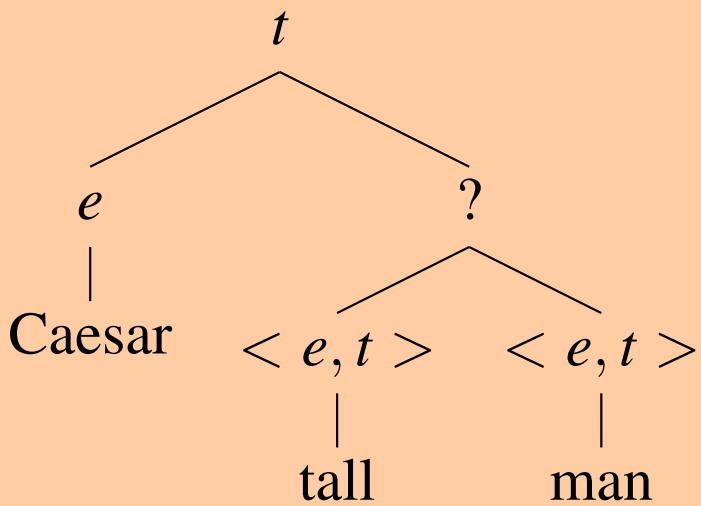
We'll fully work out the relative clause example. But first we need to understand adjectival modification and lambda abstraction.

Predicate modification

Take the following:

(40) Caesar was a short man.

Informally, this has the meaning that *Caesar* is both *tall* and *a man*. Both *tall* and *a man* are predicates of type $< e, t >$. So how can we put this together?



(Yes, I'm assuming both *was* and *a* to be semantically vacuous.)

Predicate modification

There is no context for functional application.
We need a new rule:

(41) Predicate modification: if a node α has daughters β and γ both of type $< e, t >$, $[\![\alpha]\!]$ will be of type $< e, t >$, and have the meaning $[\![\beta]\!] \wedge [\![\gamma]\!]$.

The meaning is just the same as that of coordinated predicates.

Relative clause meaning

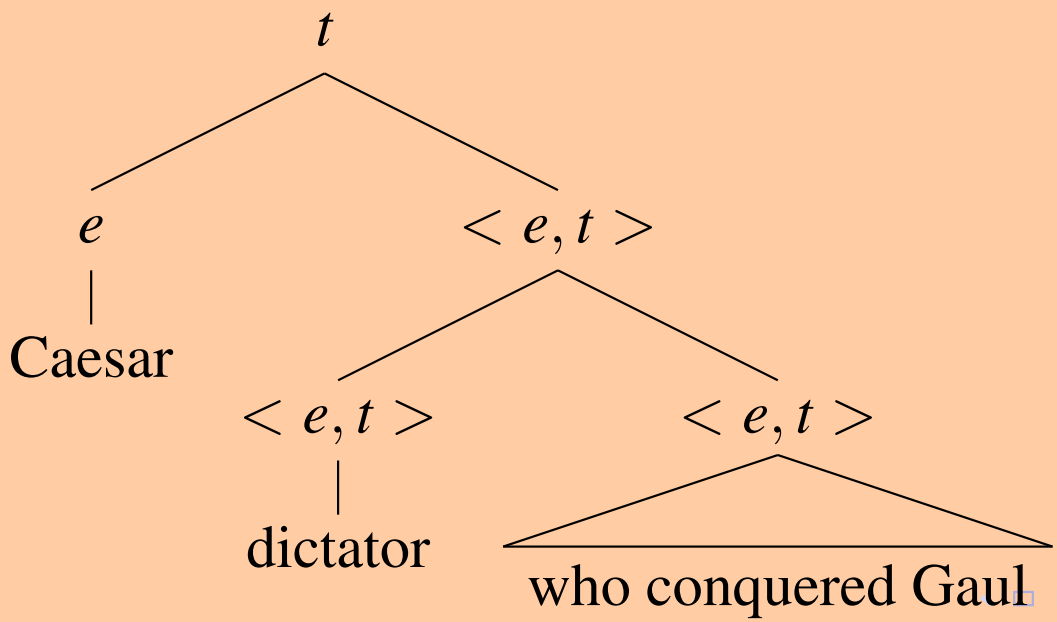
Take the following:

(42) Caesar was a dictator who conquered Gaul.

Like with adjectival modification, we'll assume that the meaning of this is:

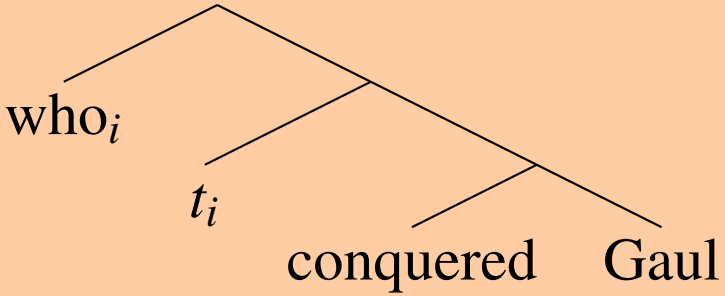
(43) $[\![$Caesar was a dictator who conquered Gaul$]\!] =$
$[\![$Caesar was a dictator$]\!] \wedge [\![$Caesar conquered Gaul$]\!]$

So:



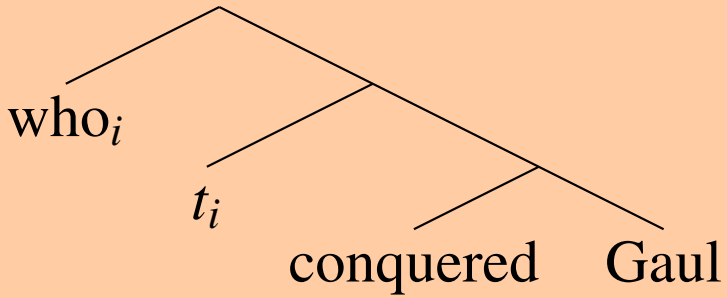
Relative clause meaning

A relative clause creates a function out of any complete sentence.
Syntactically, in better-known languages this is usually done by movement of a relative pronoun:



Relative clause meaning

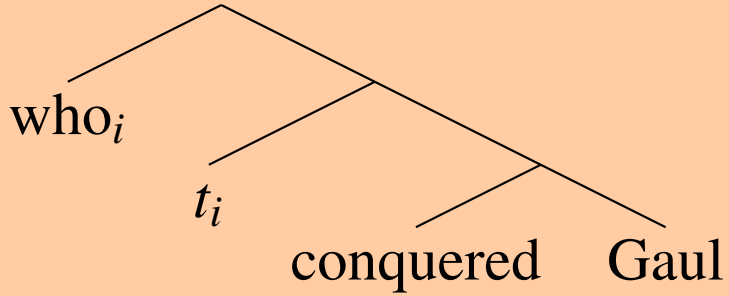
A relative clause creates a function out of any complete sentence.
Syntactically, in better-known languages this is usually done by movement of a relative pronoun:



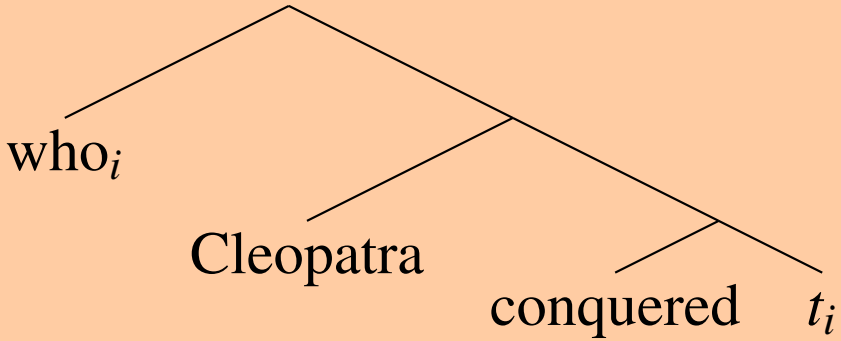
But also:

Relative clause meaning

A relative clause creates a function out of any complete sentence.
Syntactically, in better-known languages this is usually done by movement of a relative pronoun:

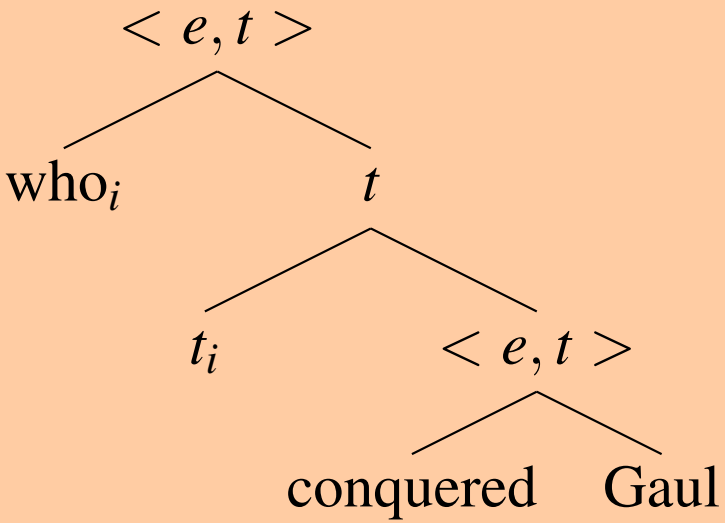


But also:



Relative clause meaning

So we want a way to turn a sentence (of type t) again into a predicate:



Lambda abstraction

We will deal with this by treating the trace as a pronoun, whose denotation is given by an assignment function:

(44) $[\![t_1]\!]^{g_c} = g_c(1)$

But how do we get this trace to act as a variable?

Lambda abstraction

We will deal with this by treating the trace as a pronoun, whose denotation is given by an assignment function:

(46) $[\![t_1]\!]^{g_c} = g_c(1)$

But how do we get this trace to act as a variable?
This is achieved by means of the syncategorematic rule of lambda abstraction.

(47) Lambda abstraction
Given two sister nodes $[\![\alpha]\!]^g$ and an index a, the denotation of the node dominating those two sisters is

$$\lambda x.[\![\alpha]\!]^{g/a \to x}$$

where $g/a \to x$ is the assignment function identical to g except for taking index a to x.