COMPUTATIONAL LINGUISTICS: Lecture 9: Semantics and Inference 2

Stephen.Pulman@comlab.ox.ac.uk

A Semantics for a Fragment of English

1.	S	$\rightarrow$ NP VP	T iff I(NP) $\in$ I(VP)
2.	NP	$\rightarrow$ Name	I(NP)=I(Name)
3.	VP	$\rightarrow$ V_{intr}	I(VP)=I(V)
4.	VP	$\rightarrow$ V_{trans} NP	I(VP)=$\{$X $\mid$ $\langle$X,I(NP)$\}$ $\in$ I(V)$\rangle$
5.	V_{intr}	$\rightarrow$ snores	I(Vi)=$\{$X $\mid$ X snores$\}$ etc.
6.	Name	$\rightarrow$ John	I(John) etc.
7.	V_{tran}	$\rightarrow$ likes	I(Vt)=$\{\langle$X,Y$\rangle$ $\mid$ X likes Y$\}$
8.	S	$\rightarrow$ S and S	T iff both daughter S are true
9.	VP	$\rightarrow$ VP and VP	I(VP0)= $\{$X $\mid$ X $\in$ I(VP1) and $\in$ I(VP2)$\}$
10.	VP	$\rightarrow$ doesn't VP	I(VP0)=$\{$X $\mid$ X is not in I(VP1)$\}$

Domain = $\{$john,bill,mary,sue$\}$

I(likes) = $\{\langle$john,mary$\rangle$,$\langle$sue,mary$\rangle\}$; I(snores) = $\{$john,sue$\}$; I(John) = john, etc.

Logical Form

But giving the semantics this way is very complicated. In practice it is easier to give the semantics indirectly via translation into logical form:

Sentence $\rightarrow$ Parse Tree $\rightarrow$ Logical Form $\rightarrow$ Truth Conditions/Entailments

Arguably we need such a level of represention anyway:

> A dog barked. It was hungry.
> Not every dog didn't bark. *It was hungry.

But 'A dog barked' and 'Not every dog didn't bark' are logically equivalent and therefore have the same truth conditions. So truth conditions alone cannot account for this difference.

In fact, while a level of logical form (LF) is crucial if we are going to use proof-theoretic methods to derive inferences, in terms of truth conditions it is theoretically dispensable (although very convenient). If we can describe the process of going from a syntax tree to an LF as a function, then since the process of going from an LF to set theoretic truth conditions is also a function, we could just compose the two functions to go directly from syntax to truth conditions.

But if we are translating parse trees into LFs compositionally - i.e. in such a way that the meaning of every constituent is a function of the meanings of its parts - we need every constituent to have a well formed (if incomplete) interpretation. We require a bit more machinery to achieve this as the LF level, although our direct denotation grammar above had this property.

Types: an assignment of logical expressions (and syntactic categories, roughly) into different semantic categories.

Our base types: ‘e’ (entity) - the type of individuals (objects) within our domain; and ‘t’ (truth value: i.e. true or false) - the type of closed wffs. (Note that in computer science, ‘i’ and ‘o’ are usually used with the same meaning.)

Rule for complex (functional) types:

if α is a type and β is a type, then $\alpha \rightarrow \beta$ is a type (read it ‘from α to β’). An alternative notation is $\langle\alpha,\beta\rangle$.

type of **john**	$=$	e
type of **snores(john)**	$=$	t
type of **snores**	$=$	e $\rightarrow$ t
type of $\neg$	$=$	t $\rightarrow$ t

From now on, we will require our logical formulae to be 'well-typed', i.e. if the formula contains an application of a function to an argument, then the formula must be of type $\alpha \rightarrow \beta$, the type of the argument must be α, and the type of the whole thing will be β:

$[\alpha \rightarrow \beta](\alpha) = \beta$

n.b. informal notational convention: I will where possible put expressions with functional types in [], and their arguments in ().

What type should we give to things like ‘hit’ or ‘and’? These seem to need two arguments, but our notation will only allow one.

Solution: we ‘curry’ them, making them all functions of one argument. So ‘hit’ will be of type $e \rightarrow (e \rightarrow t)$, and ‘and’ will be of type $t \rightarrow (t \rightarrow t)$. (We will temporarily ignore the fact that ‘and’ is an infix operator). This means that these functions will find first one argument, and then produce a function looking for the remaining argument.

We will also introduce one final concept: the ‘lambda’ operator, λ.

This is a way of forming new functions from any term of our logic. We do this by ‘abstracting’ over some position(s) in an expression: e.g.

λ**x.hit(x,mary)** (‘the property of being an x such that x hit Mary’)

λ**x.hit(mary,x)** (‘the property of being an x such that Mary hit x’)

λ**P.P(john)** $\wedge$ **P(mary)** (‘a property which is true of both John and Mary’)

λ**x.**λ**y.hit(john,x)** $\wedge$ **hit(y, mary)** (‘the property of being an x that john hit, and that gives a property of being a y that hit Mary’).

Types: type(λvariable.body) = type of variable $\rightarrow$ type of body

Given an expression of the form [λv.b](a) the rules for typing imply that v and a have the same type, say α, and that if b has type β, then

type of [λv.b] = $\alpha \rightarrow \beta$, and

type of [λv.b](a) = β.

Lambda-reduction*: in our logic, an expression of the form **[λ variable.body](argument)** means exactly the same as the expression you get by taking just the body of the λ-expression and replacing every occurrence of the variable in it by an occurrence of the argument:

[λx.snores(x)](john) = snores(john)

[λx.hit(mary,x)](john) = hit(mary,john)

[λx.hit(x,x)](john) = hit(john,john) (2 occurrences of x)

[λx.snores(mary)](john) = snores(mary) (0 occurrences of x)

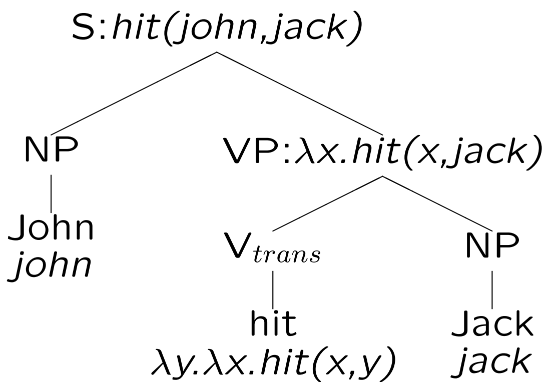
*Strictly, β-reduction

Giving the semantics via logical form

A simple example grammar. The part after the colon describes how the meaning of the mother constituent is built up as a function of the meanings of the daughters.

1.	S	$\rightarrow$ NP VP	: VP(NP)
2.	NP	$\rightarrow$ Name	: Name
3.	VP	$\rightarrow$ V_{intr}	: V_{intr}
4.	VP	$\rightarrow$ V_{trans} NP	: V_{trans}(NP)
5.	V_{intr}	$\rightarrow$ snores	: λx.snore(x), etc
6.	Name	$\rightarrow$ John	: john, etc
7.	V_{trans}	$\rightarrow$ hits	: λy. λx. hit(x,y)
8.	S	$\rightarrow$ S and S	: S $\wedge$ S
9.	VP	$\rightarrow$ VP and VP	: λx.VP_1(x) $\wedge$ VP_2(x)
10.	VP	$\rightarrow$ doesn't VP	: λx.$\neg$(VP(x))

SEMANTICS IS COMPOSITIONAL, DRIVEN BY SYNTAX

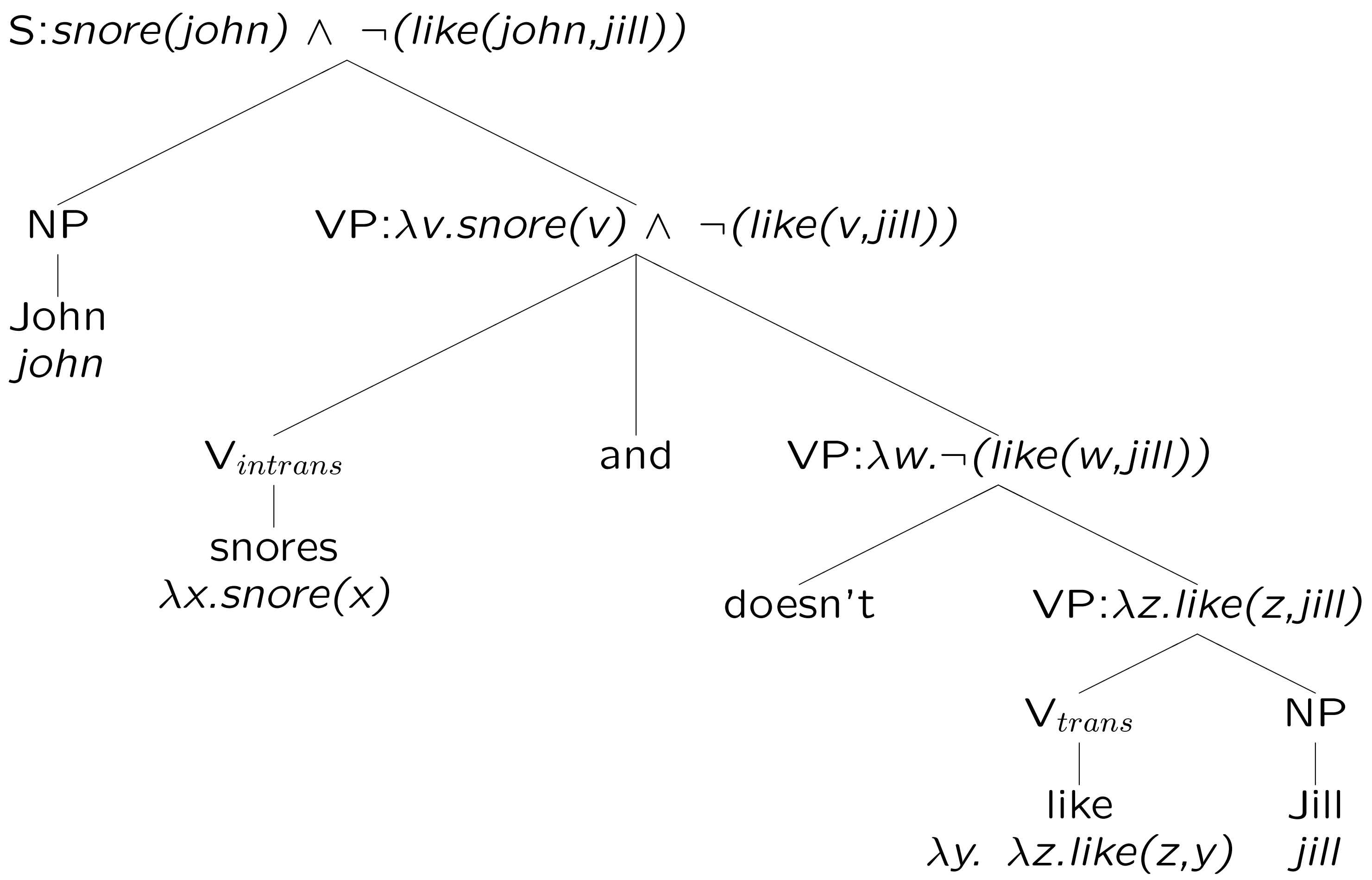

S:*hit(john,jack)*
NP
VP:λ*x.hit(x,jack)*
John
john
V_{trans}
NP
hit
λ*y.*λ*x.hit(x,y)*
Jack
jack

Although there are some purely semantic ambiguities: e.g. every student went to a lecture

$\forall$x.student(x) $\rightarrow$ $\exists$y.lecture(y) $\wedge$ go(x,y)
$\exists$y.lecture(y) $\wedge$ $\forall$x.student(x) $\rightarrow$ go(x,y)

John wants to marry a Norwegian girl.

We don't want to say that these sentences are syntactically ambiguous.


S:$snore(john) \wedge \neg(like(john,jill))$
NP
John
$john$
VP:$\lambda v.snore(v) \wedge \neg(like(v,jill))$
V$_{intrans}$
snores
$\lambda x.snore(x)$
and
VP:$\lambda w.\neg(like(w,jill))$
doesn't
VP:$\lambda z.like(z,jill)$
V$_{trans}$
like
$\lambda y.\ \lambda z.like(z,y)$
NP
Jill
$jill$

References

Please read the 'Higher Order Logic in Semantics' handout on the Computational Linguistics Course Materials page.

D. Dowty, R. Wall, S. Peters 1981 Introduction to Montague Semantics, Dordrecht: D. Reidel

Bob Carpenter 1998 Type Logical Semantics, MIT Press. (first couple of chapters)

James Allen, 1996, Natural Language Understanding, Chap 8 and 9.

D Jurafsky and J Martin, 2008, Speech and Language Processing (Second Edition), Chapters 17 and 18.

There are many good textbooks on first order logic, and most textbooks on Artificial Intelligence contain a chapter on it.