An Introduction to Computational Semantics

by **Patrick Blackburn** & **Johan Bos**

September 17, 2001

Foundational Course, ESSLLI XIII, Helsinki, 13–24 August, 2001

Table of Contents

An Introduction to Computational Semantics

Foundational Course, ESSLLI XIII, Helsinki, 13-24 August, 2001

Patrick Blackburn
INRIA, Lorraine
patrick@aplog.org

Johan Bos
University of Edinburgh
jbos@cogsci.ed.ac.uk

Sponsored by EACL, the European Association for Computational Linguistics.

Endorsed by SIGSEM, the ACL Special Interest Group in Computational Semantics.

About this course ...

- This is a first course in computational semantics.
- Each session will contain (varying mixtures of) theoretical discussion, discussion of implementation, and demos.
- The website `http://www.comsem.org` is an integral part of this course. There's lots of relevant stuff there, and more will be added.
- The implementation language is PROLOG. You can find an introduction to PROLOG at `http://www.comsem.org`.
- We hope you enjoy the course, and welcome feedback!

What is computational semantics?

Computational semantics revolves around two fundamental questions:

- How can we automate the process of associating semantic representations with expressions of natural language?
- How can we use semantic representations of natural language expressions to automate the process of drawing inferences?

This course introduces some fundamental techniques for tackling these questions. The key idea is to use lambda calculus to build logical semantic representations.

Why is this a good strategy?

- The best known logics have a precise semantics in terms of models — so if we can translate a natural language sentence S into a logical formula ϕ, then we have a precise grasp on at least part of the meaning of S.
- Important inference problems have been studied for the best known logics, and often good computational implementations exist. So translating into logic gives us a handle on inference, and this is *vital* for natural language.

Moreover...

- Lambda calculus itself has a logical interpretation, which we don't discuss in this course, but which is important.
- Further, lambda calculus is a great glue language. For a start, it's very well understood.
- And it's flexible. For example, in Lecture 8 we will stop working with first-order logic, and start working with Discourse Representation Theory (DRT) instead.
- But we won't have to redo all our work. Simple adjustments to our existing programs make it possible to use lambdas with DRT very straightforwardly.

Where we're going...

Week 1	Week 2
Lecture 1: First-Order Logic	Lecture 6: First-Order Inference
Lecture 2: Lambda Calculus (I)	Lecture 7: Putting it all Together
Lecture 3: Lambda Calculus (II)	Lecture 8: DRT
Lecture 4: Scope Ambiguities	Lecture 9: Presupposition
Lecture 5: Propositional Inference	Lecture 10: Putting it all Together again

Today Lecture 1: First-Order Logic

- we review the syntax and semantics of first order logic, the representation formalism we will use in the first seven lectures
- we define the three inference tasks we are interested in: querying, consistency checking, and informativity checking.
- we present a first-order model checker, a tool for carrying out querying, the simplest of these three inference tasks
- Demos:
  - `modelChecker1.pl`
  - `modelChecker2.pl`

First Order Logic

The three fundamental concepts are

- Vocabularies
- Models
- First Order Languages

Basically, first order languages are relatively simple, precisely defined languages for talking about simple pictures of the world (models). Vocabularies basically 'set the stage'.

Vocabularies

Here is an example:

{ (LOVE,2),
(CUSTOMER,1), (ROBBER,1),
(MIA,0), (VINCENT,0), (HONEY-BUNNY,0), (YOLANDA,0) }

- Vocabularies tell us 'the topic of conversation'. That is, they specify which language we are going to use.
- Items in a vocabulary associated with the number 0 are called constants. For example, YOLANDA is a constant.
- Items in a vocabulary associated with a number greater than 0 are called relation symbols, (or, sometimes, predicate symbols).
- For example, LOVE is a relation symbol. In fact, it is a 2-place relation symbol (or to use more technical terminology, a relation symbol of arity 2).

First-Order Models

A model is a pair (D, F)

- D contains the set of entities we want to talk about. D is called the domain, or universe of discourse, and must be non-empty.
- F is the interpretation function. It specifies what each symbol in the vocabulary stands for. It does so by associating each symbol in the vocabulary with an appropriate entity built from items in D.

Simple Model Example

- Let domain D be $\{d_1, d_2, d_3, d_4\}$
- Specify an interpretation function F

$F(\text{MIA}) = d_1$
$F(\text{HONEY-BUNNY}) = d_2$
$F(\text{VINCENT}) = d_3$
$F(\text{YOLANDA}) = d_4$
$F(\text{CUSTOMER}) = \{d_1, d_3\}$
$F(\text{ROBBER}) = \{d_2, d_4\}$
$F(\text{LOVE}) = \{(d_4, d_2), (d_3, d_1)\}$

Note that every entity in this model has a name.

Another Model

- We 'll use the same domain $(D = \{d_1, d_2, d_3, d_4\})$
- but change the interpretation function

$F_2(\text{MIA}) = d_2$
$F_2(\text{HONEY-BUNNY}) = d_1$
$F_2(\text{VINCENT}) = d_4$
$F_2(\text{YOLANDA}) = d_3$
$F_2(\text{CUSTOMER}) = \{d_1, d_2, d_4\}$
$F_2(\text{ROBBER}) = \{d_3\}$
$F_2(\text{LOVE}) = \{\}$

In this model, three of the individuals are customers, and only one is a robber. Again, every entity in the model has a name. But note: the love relation is empty.

Yet Another Model

- We 'll use the domain $D = \{d_1, d_2, d_3, d_4, d_5\}$
- and the following interpretation function

$F_2(\text{MIA}) = d_2$
$F_2(\text{HONEY-BUNNY}) = d_1$
$F_2(\text{VINCENT}) = d_4$
$F_2(\text{YOLANDA}) = d_1$
$F_2(\text{CUSTOMER}) = \{d_1, d_2, d_4\}$
$F_2(\text{ROBBER}) = \{d_3, d_5\}$
$F_2(\text{LOVE}) = \{(d_3, d_4)\}$

In this model, not every entity has a name.
Moreover, d_1 has two names.

First-Order Languages: the ingredients

1. All the symbols in the vocabulary. We call these symbols the non-logical symbols of the language.
2. An infinite collection of variables x, y, z, w, ..., and so on.
3. The Boolean operators $\neg$ (negation), $\rightarrow$ (implication), $\vee$ (disjunction), and $\wedge$ (conjunction).
4. The quantifiers $\forall$ (the universal quantifier) and $\exists$ (the existential quantifier).
5. The round brackets) and (and the comma.

Terms

Terms are the 'noun phrases' of first order languages.

1. All constants and variables are terms.
2. Nothing else is a term.

(We'll see shortly that there are richer first order languages that allow us to build more interesting terms than this.)

Atomic Formulas

We combine our 'noun phrases' with our 'predicates' to form *atomic formulas*:

If R is a relation symbol of arity n, and $\tau_1, \ldots, \tau_n$ are terms, then $r(\tau_1, \cdots, \tau_n)$ is an atomic (or basic) formula.

Well Formed Formulas (wffs)

1. All atomic formulas are wffs
2. If ϕ and ψ are wffs then so are $\neg\phi$, $(\phi \to \psi)$, $(\phi \vee \psi)$, and $(\phi \wedge \psi)$
3. If ϕ is a wff, and x is a variable, then both $\exists x\phi$ and $\forall x\phi$ are wffs (We call ϕ the matrix of such wffs)
4. Nothing else is a wff

But what do all these formulas mean?

Roughly speaking

1. Atomic formulas correspond to simple natural language sentences.
2. $\neg$ is 'not', $\to$ is 'if ... then ...', $\vee$ is 'or', $\wedge$ is 'and'.
3. $\exists$x is 'there is at least one' and $\forall$x is 'for all' or 'every'.

But we can't stick with this informal explanation — we need something precise that we can work with on a computer.

And making this precise requires a little thought ...

Free and Bound Variables

Example:

$\neg\, (\textsc{customer}(x) \vee \forall x(\textsc{robber}(x) \wedge \forall y\ \textsc{person}(y)))$

- the first occurrence of x is free
- the second and third occurrences of x are bound
- the first and second occurrences of the variable y are also bound

It may be useful to think of a free variable as pronoun.

Sentences

A sentence is a formula that contains no free variables. (Warning: the word 'sentence' is being used here as a technical term.)

- What does it mean for a first-order sentence to be true in a model?
  We would like an inductive definition.
- Problem: we cannot give a *direct* inductive definition of the truth relation, because the matrix of a quantified sentence need not be a sentence.

For example, $\forall x\ \textsc{robber}(x)$ *is a sentence, but* $\textsc{robber}(x)$ *is not*.

- Solution: We will work *indirectly*, via the notion of satisfaction.

Variable Assignment Functions

- Suppose we are working with a model $\mathbf{M} = (D, F)$
- Then an assignment g of values to variables in $\mathbf{M}$ is a function from the set of variables to D
- Think of g as a 'context', which specifies values for our 'pronouns' (free variables)

Assignment functions allows us to define satisfaction for any formula.

Interpretation of Terms

Let

- $\mathbf{M} = (D, F)$ be a model
- g be an assignment of values to variables in $\mathbf{M}$
- τ be a term

Then by the interpretation of τ with respect to $\mathbf{M}$ and g is meant:

- $F(\tau)$ if τ is a constant
- $g(\tau)$ if τ is a variable

We denote the interpretation of τ by $I_F^g(\tau)$.

Varying assignments

We need one more technical notion before we can define satisfaction: the notion of a variant of an assignment.

- Let g be an assignment of values to variables (in some model $\mathbf{M}$) , and let x be a variable.
- Then an assignment g' (in the same model $\mathbf{M}$) is an x-variant of g if and only if $g(\mathrm{v}) = g'(\mathrm{v})$ for all variables v that are distinct from x.
- Intuitively, an x-variant g' of g is a 'context' that agrees with g about the values of all the 'pronouns' — except possibly the pronoun x.
- On the next slide, we'll see why this concept is important ...

The Satisfaction Definition

$$
\begin{array}{lll}
\mathbf{M}, g \models R(\tau_1, \cdots, \tau_n) & \textit{iff} & (I_F^g(\tau_1), \cdots, I_F^g(\tau_n)) \in F(R) \\
\mathbf{M}, g \models \neg\phi & \textit{iff} & \text{not } \mathbf{M}, g \models \phi \\
\mathbf{M}, g \models \phi \wedge \psi & \textit{iff} & \mathbf{M}, g \models \phi \text{ and } \mathbf{M}, g \models \psi \\
\mathbf{M}, g \models \phi \vee \psi & \textit{iff} & \mathbf{M}, g \models \phi \text{ or } \mathbf{M}, g \models \psi \\
\mathbf{M}, g \models \phi \to \psi & \textit{iff} & \text{not } \mathbf{M}, g \models \phi \text{ or } \mathbf{M}, g \models \psi \\
\mathbf{M}, g \models \exists x\phi & \textit{iff} & \mathbf{M}, g' \models \phi, \\
 & & \text{for some x-variant } g' \text{ of } g. \\
\mathbf{M}, g \models \forall x\phi & \textit{iff} & \mathbf{M}, g' \models \phi, \\
 & & \text{for all x-variants } g' \text{ of } g.
\end{array}
$$

Truth

We can now define what it means for a *sentence* to be true in a model:

> A sentence is true in a model $\mathbf{M}$ if and only if for *any* assignment g of values to variables in $\mathbf{M}$, we have that $\mathbf{M}, g \models \phi$.
>
> Slightly tricky! Make sure you understand this definition!

Well, that's the basics ...

But there's two other things we should know about before moving on, namely,

- first order languages with function symbols (we'll use these in lecture six),
- and first order languages with equality (very important when working with natural language).

Function symbols

We can add add collection of functions symbols f to first order languages, each of some fixed arity. This lets us build richer terms (that is, more interesting noun phrases):

1. All constants and variables are terms.
2. If f is a function symbols of arity n, and $\tau_1, \ldots, \tau_n$ are terms, then $f(\tau_1, \ldots, \tau_n)$ is a also a term.
3. Nothing else is a term.

A term is said to be closed if and only if it contains no variables.

Example

Let 'father' be a one place function symbol, and let 'butch' be a constant. Then:

father(father(butch))

is a term that denotes Butch's grandfather.

From this example, it is pretty clear how function symbols should be interpreted...

Interpreting function symbols

We simply extend our definition of I_F^g in the obvious way. Given a model $\mathbf{M}$ and an assignment g in $\mathbf{M}$, we define:

- $I_F^g(\tau)$ is $F(\tau)$ if τ is a constant (just as before)
- $I_F^g(\tau)$ is $g(\tau)$ if τ is a variable (just as before)
- If τ is a term of the form $f(\tau_1, \ldots, \tau_n)$, then $I_F^g(\tau)$ is $F(f)(I_F^g(\tau_1), \ldots, I_F^g(\tau_n))$.

That is, we apply the n-place function $F(f)$ — the function interpreting f — to the interpretation of the n argument terms.

Equality

- It is common to add a special two place relation symbol $=$ to first order languages.
- This symbol is always interpreted as follows. In any model $\mathbf{M}$, given any assignment g in $\mathbf{M}$, and any terms τ_1 and τ_2

$$\mathbf{M}, g \models \tau_1 = \tau_2 \text{ iff } I_F^g(\tau_1) = I_F^g(\tau_2).$$

- That is, this symbol really means *equality*.
- Although syntactically just a relation symbol, $=$ is really a *logical* symbol like $\neg$ or $\forall$.

Example

yolanda = honey-bunny

is an atomic formula asserting that Yolanda and Honey Bunny are the same person.

Is first-order logic a good choice?

- We now know a lot of what we need to know about first-order logic. Time for a fundamental question: is it a good choice for semantic representation?
- Basically yes. In spite of what you may have read elsewhere, first-order logic can represent just about anything, including temporal information and modal information. It can even simulate higher-order logic. About its only obvious expressive weakness is that it does not handle quantifiers like 'most'.
- As far as we're concerned, it's real failings are representational rather than expressive. We shall be forced to add lambdas, and later to move to DRT.
- And yet, under the surface, first-order logic will still be there, ready and willing to do lots of work for us, as we shall see...

Inference

In this course we are interested in the following three inference tasks:

- Querying
- Consistency checking
- Informativity checking

Querying

Given a model $\mathbf{M}$ and a formula ϕ, is ϕ true in model $\mathbf{M}$ or not?

- Think of the model $\mathbf{M}$ as a picture of (a little part of) the world
- Then querying ϕ is asking whether or not the information ϕ is true in our little picture of the world
- A tool that carries out this inference task is called a model checker.
- This is a relatively simple form of inference to implement for finite models (as we shall later today) and will turn out to be useful later for question answering. (It is also useful for tasks such as natural language generation.)

Consistency checking (I)

- A formula is consistent if it is satisfied in at least one model. So consistent formulas describe 'conceivable' or 'possible' states of affairs. For example, $\text{ROBBER}(\text{MIA})$ is consistent.
- A formula that is not consistent is called inconsistent. So inconsistent describe 'inconceivable' or 'impossible' states of affairs. For example, $\text{ROBBER}(\text{MIA}) \wedge \neg\text{ROBBER}(\text{MIA})$ is inconsistent.
- A finite set of formulas $\{\phi_1, \ldots, \phi_n\}$ is consistent if $\phi_1 \wedge \cdots \wedge \phi_n$ is consistent. A finite set of formulas that is not consistent is called inconsistent.

Consistency checking (II)

- We would like to check whether the information given to us in natural language discourses is consistent or not. For if we are given inconsistent information, something may well be going wrong with communicative process ...
- However consistency checking is a *much* harder inference task than querying. In fact, for first order logic, consistency checking is undecidable
- However there are two computational tools that can help us carry it out: theorem provers and model builders. We'll learn more about them later in the course.

Informativity checking (I)

- A valid sentence is a sentence that is true in *all* models (for example: $\text{ROBBER}(\text{MIA}) \vee \neg\text{ROBBER}(\text{MIA})$). A sentence that is not valid is called invalid.
- Suppose ϕ_1, ..., ϕ_n, and ψ are a finite collection of first-order sentences. Then we say that the argument with premises ϕ_1, ..., ϕ_n and conclusion ψ is a valid argument if whenever all the premises are true in some model, the conclusion is true in that model also.
- The notation $\phi_1, \ldots, \phi_n \models \psi$ indicates a valid argument.
- The notation $\phi_1, \ldots, \phi_n \not\models \psi$ indicates a invalid argument.

Informativity checking (II)

So what does validity have to do with informativity? This:

- We often call valid sentences uninformative sentences. (After all, if they're true in all models, they don't give us any specific information!) We often call a sentence that is not valid informative.
- If $\phi_1, \ldots, \phi_n \models \psi$, then we say that ψ is uninformative with respect to $\phi_1, \ldots, \phi_n$.
- If $\phi_1, \ldots, \phi_n \not\models \psi$, then we say that ψ is informative with respect to $\phi_1, \ldots, \phi_n$.

Informativity checking (III)

- We would like to check whether the information given to us in natural language discourses is informative with respect to the information we already have. For if it is not, something may well be going wrong with communicative process ...
- Informativity checking is a *much* harder inference task than querying. In fact, for first order logic, consistency checking is undecidable.
- However there are two computational tools that can help us carry it out: theorem provers and model builders.

Relationships between Consistency and Informativity

- ϕ is informative (that is, not valid) if and only if $\neg\phi$ is consistent. That is, informativity means the opposite really was an option.
- $\phi_1, \ldots, \phi_n \not\models \psi$ (that is, ψ is informative with respect to ϕ_1, ..., ϕ_n) iff $\{\phi_1, \ldots, \phi_n, \neg\psi\}$ is consistent.
- ϕ is uninformative (that is, valid) if and only if $\neg\phi$ is inconsistent. That is, uninformativity means that the opposite simply was not an option.
- $\phi_1, \ldots, \phi_n \models \psi$ (that is, ψ is uninformative with respect to ϕ_1, ..., ϕ_n) iff $\{\phi_1, \ldots, \phi_n, \neg\psi\}$ is inconsistent.

Something to bear in mind...

- We said that consistency checking and informativity checking are much harder tasks than querying. Why is this?
- Because both tasks are defined in terms of all models, and there are lots of models, and most are infinite. In short, both task are defined semantically and in a very abstract way. (By way of contrast, querying is defined semantically, but in a very concrete way.)
- To get to grips with consistency checking and informativity checking computationally we shall need to gain a syntactic handle on them (that is, we will need to look at the branch of logic called proof theory) and that will be our chief task in Lectures 5 and 6.

A simple model checker

We are now going to discuss a simple first-order model checker written in PROLOG. (Actually, we're going to discuss two versions of this program.) Before the demo, we need some background:

- How we define PROLOG representation for models
- How we define PROLOG representation for formulas

Representing Models

Here's a typical example of the way we represent models in Prolog:

```
model([d1,d2,d3,d4],
      [f(0,mia,d1),
       f(0,vincent,d2),
       f(0,pumpkin,d3),
       f(0,honey_bunny,d4),
       f(1,customer,[d1,d2]),
       f(1,robber,[d3,d4]),
       f(2,love,[(d3,d4)])]).
```

We have chosen to give the vocabulary explicitly as part of the model representation.

Representing Formulas

- First-order variables are represented by PROLOG variables
- A first-order constant c is represented by the PROLOG atom `c`
- A first-order relation symbol R is represented by the PROLOG atom `r`

The Booleans

The symbols

`&`   `v`   `>`   `~`

will be used to represent the connectives $\wedge$, $\vee$, $\rightarrow$, and $\neg$ respectively. The following PROLOG code ensure that these operators have their usual precedences:

```
:- op(900,yfx,>).        % implication
:- op(850,yfx,v).        % disjunction
:- op(800,yfx,&).        % conjunction
:- op(750, fy,~).        % negation
```


The Quantifiers

Suppose *formula* is a first-order formula, and `Formula` is its representation as a PROLOG term.

Then $\forall$x *formula* will be represented as

`forall(X,Formula)`

and $\exists$x *formula* will be represented as

`exists(X,Formula)`

And now we're ready to demo:

- modelChecker1.pl
- modelChecker2.pl

Both programs are available at `http://www.comsem.org`

Today Lecture 2: Lambda Calculus (I)

- We begin studying the first of the fundamental questions we began the course with: How can we automate the process of associating semantic representations with expressions of natural language?
- We discuss the idea of compositionality, experiment with ways of implementing compositional semantic construction, and are gradually lead to the idea of lambda calculus.
- We implement α-conversion, an important concept in lambda calculus. The implementation of β-conversion, the other key concept, will be discussed tomorrow.
- Demos:
  - `experiment2.pl`
  - `experiment3.pl`
  - `alphaConversion.pl`

Semantic construction

- Given a sentence of English, is there a systematic way of constructing its semantic representation?
- This question is too ambitious for now. Let's begin with more specific one.
- Is there a systematic way of translating simple sentences such as:
  - Vincent likes Mia
  - Every woman snorts
  - Every boxer loves a woman

  into first-order logic?

Meaning flows from the lexicon

- The natural language sentence Vincent likes Mia should be represented by the first-order sentence like(vincent,mia).
- Now, pretty obviously, the proper name Vincent is what gives rise to the constant vincent, the proper name Mia is what gives rise to the constant mia, and the verb likes contributes the 2-place relation symbol like.
- A simple observation, but it leads to an important generalization: meaning ultimately flows from the lexicon.

What do other word contribute?

- This is a simple insight, but it forces us to face up to some nontrivial questions.
- For example, Every woman snorts has ∀x(woman(x)→snort(x)) as its representation.
- What exactly does the word Every contribute to this representation? The ∀? The →? Both together, in some sort of pattern (incidentally: this is the right answer)? And how can we be precise about what its contribution is?

Syntax also plays a role

- Why did we get like(vincent,mia) as the representation of Vincent likes Mia, and not (say) like(mia,vincent)?
- That is, how do all the bits and pieces that the lexicon provides come to be 'glued together' in the right way?
- The basic principle here is that the syntactic structure of the natural language sentence should guide the process of semantic construction.
- Briefly: syntactic structure is the guide to gluing.
- Let's look at an example ...

Syntactic structure guiding gluing

Vincent likes Mia (S)
LIKE(VINCENT,MIA)

Vincent (NP)
VINCENT

likes Mia (VP)
LIKE(?,MIA)

likes (TV)
LIKE(?,?)

Mia (NP)
MIA

Compositionality

Our discussion has isolated the idea of compositionality, or compositional semantic construction:

- Meaning (representation) ultimately flows from the lexicon
- Meanings (representations) are combined by making use of syntactic information
- More precisely, the meaning of the whole is a function of the meaning of its parts, where 'parts' refers to the substructure given to us by the syntax.

This gives us a guide to how to proceed in the days that follow. We need to...

Task 1 Specify a reasonable syntax for the fragment of natural language of interest.

Task 2 Specify semantic representations for the lexical items.

Task 3 Specify the translation compositionally. That is, we should specify the translation of all expressions in terms of the translation of their parts.

Moreover, all three tasks need to be carried out in a way that leads naturally to computational implementation.

We use DCGs for task 1

```
s --> np, vp.                  noun --> [woman].
np --> pn.                     noun --> [foot,massage].
np --> det, noun.              vp --> iv.
pn --> [vincent].              vp --> tv, np.
pn --> [mia].                  iv --> [walks].
det --> [a].                   tv --> [loves].
det --> [every].               tv --> [likes].
```

This grammar accepts sentence like Vincent walks and Mia likes a foot massage.

The Joy of DCGs

- They are not only a natural notation for specifying grammars they are part and parcel of PROLOG. So they automatically have a computational interpretation.
- For example, by posing the query

  ```
  ?- s([mia,likes,a,foot,massage],[])
  ```

  we can test whether this sentence belongs to the fragment the grammar describes.
- Moreover, by using extra arguments, the PROLOG unification mechanism gives us the tools needed to combine and pass up semantic representations.
- For more on DCGs, look at Learn Prolog Now! which is available at `http://www.comsem.org`.

An Experiment in Semantic Construction

To build representations we need to

- Specify the meanings of the lexical entries (words). These representations are typically 'incomplete' formulas (remember the Vincent likes mia tree).
- Indicate where the information they lack must come from (with the help of the syntactic information the DCG grammar offers us, and other Prolog tricks).
- Key idea: let's try associating missing information with an explicit PROLOG variable.

DCG with Semantics: Lexical Entries

```
noun(X,woman(X)) --> [woman].

pn(jules) --> [jules].

iv(Y,snort(Y)) --> [snorts].

tv(Y,Z,love(Y,Z)) --> [loves].

det(X,Rest,Scope,exists(X,Rest & Scope)) --> [a].

det(X,Rest,Scope,forall(X,Rest > Scope)) --> [every].
```


DCG with Semantics: Production Rules

```
s(Sem) --> np(X,SemVP,Sem), vp(X,SemVP).

vp(X,Sem) --> tv(X,Y,SemTV), np(Y,SemTV,Sem).
vp(X,Sem) --> iv(X,Sem).

np(X,Scope,Sm) --> det(X,Rest,Scope,Sm), noun(X,Rest).
np(SemPN,Sem,Sem) --> pn(SemPN).
```


How did we do?

- It works, and the underlying intuition is pretty clear. Explicitly marking missing information is clearly a good idea; it gives us a good level of control.
- However, much of the work is done by the rules. Sometimes this requires clever programming tricks. This makes it hard to think about the grammar in a modular way.
- This suggests we are missing something. Maybe a more disciplined approach to missing information would reduce — or even eliminate — the need for rule specific combination methods?
- This is exactly happens if we make use of the discipline provided by the lambda calculus.

The Lambda Calculus

- In this course, lambda calculus will be regarded as a small notational extension of first order logic that allows us to bind variables using an operator λ. Variables bound by λ are 'placeholders' for missing information.
- When we place a lambda expression next to another expression (the argument expression) it is an instruction to substitute the argument expression into the placeholders. This is called functional application
- An operation called β-conversion does the actual work of carrying out the substitutions. Another operation called α-conversion performs small (but crucial) bookkeeping function.
- We want you to think of the lambda calculus as glue language, a 'programming language' dedicated to a single task: gluing together the items needed to build semantic representations

The Lambda Operator

The lambda operator marks missing information by binding variables.

- Here is a simple lambda expression:

  $\lambda x.\textsc{man}(x)$

  The prefix λx binds the occurrence of x in $\textsc{man}(x)$.
- $\lambda x.\textsc{man}(x)$ can be read as: 'I am the 1-place predicate MAN, and I am looking for a term to fill my argument slot'.

Functional application

The @ operator is used to indicate 'functional application'. That is, it indicates that we wish to perform a substitution.

Example:

$\lambda x.\textsc{man}(x)@\textsc{vincent}$

- The expression $\lambda x.\textsc{man}(x)$ is called the functor. The expression VINCENT is called the argument.
- Intuitively this says: fill all the placeholders in the functor by occurrences of the argument VINCENT.
- But how do we actually do this...?

β-conversion

The required substitution is performed by β-conversion:

From

$\lambda x.\textsc{man}(x)@\textsc{vincent}$

β-conversion produces

$\textsc{man}(\textsc{vincent})$

Basically, we throw away the $\lambda x.$ at the start of the expression, and substitute the argument for all occurrences of x that were bound by λx. That's β-conversion.

Marking more complex kinds of information

Our semantic representation of "a woman" will be:

$\lambda Q.\exists x(\textsc{woman}(x) \wedge Q@x)$

We use the variable Q to indicate that:

- some information is missing
- and where this information has to be plugged in

In essence, we can use lambda notation to build up 'patterns of meaning' or 'patterns of representation', explicitly indicating where the various bit and pieces have to be glued into place.

An example: Every boxer growls

Step 1: assign lambda expressions to the basic syntactic categories

boxer: $\lambda y.\textsc{boxer}(y)$

growls: $\lambda x.\textsc{growl}(x)$

every: $\lambda P.\lambda Q.\forall x(P@x \rightarrow Q@x)$

Reminder

In our experiment we gave the determiner "every" the representation

```
det(X,Rest,Scope,forall(X,Rest > Scope))
```

If we use the PROLOG variable `P` instead of `Rest`, and `Q` instead of `Scope` this becomes

```
det(X,P,Q,forall(X,P > Q))
```

which is clearly analogous to

$\lambda P.\lambda Q.\forall x(P@x \rightarrow Q@x)$

So what's the big difference...?

The big difference

- We are no longer considering the process of combining expressions simply as a programming exercise.
- We have isolated a representational format (lambda notation) that let's us deal with 'missing information' once and for all. That is, we have performed an important data abstraction.
- Moreover we have isolated the key ideas we need to work with these representations:
  - functional application,
  - β conversion, and (later)
  - α-conversion.
- In a nutshell: we have worked out ahead of time exactly the tools we need.

Every boxer growls (continued)

Step 2: associate the NP node with the application that has the DET representation as functor and the NOUN representation as argument.

every boxer (NP)
$\lambda P.\lambda Q.\forall x(P@x \rightarrow Q@x)@\lambda y.\textsc{boxer}(y)$

every (DET)
$\lambda P.\lambda Q.\forall x(P@x \rightarrow Q@x)$

boxer (NOUN)
$\lambda y.\textsc{boxer}(y)$

β-Conversion

Applications are instructions to carry out β-conversion

Performing the demanded substitution yields:

every boxer: $\lambda Q.\forall x(\lambda y.\textsc{boxer}(y)@x \rightarrow Q@x)$

This expression contains a subexpression of the form $\lambda y.\textsc{boxer}(y)@x$.

This is another instruction to perform β-conversion:

every boxer: $\lambda Q.\forall x(\textsc{boxer}(x) \rightarrow Q@x)$

We can't perform any more β-conversions, so let's carry on with the analysis of the sentence.

The final representation

every boxer growls (S)
$\forall x(\textsc{boxer}(x) \rightarrow \textsc{growl}(x))$

every boxer (NP)
$\lambda Q.\forall x(\textsc{boxer}(x) \rightarrow Q@x)$

growls (VP)
$\lambda x.\textsc{growl}(x)$

every (DET)
$\lambda P.\lambda Q.\forall x(P@x \rightarrow Q@x)$

boxer (NOUN)
$\lambda y.\textsc{boxer}(y)$

A Moment of Reflection

In two important respects, our approach to semantic construction is getting simpler:

1. the process of combining two representations is now uniform
2. most of the real work of semantic analysis is done in the lexicon
3. This is a good sign that we are doing something right, but let's check this out a bit...

Proper Names: Clouds on the Horizon?

- Quantifying noun phrases can clearly be used as functors
- But what about NPs like "Vincent"?

In fact, there's no problem at all!

"Mia": λP.P@MIA

"Vincent": λP.P@VINCENT

These representations *can* be used as functors.

Proper Names in Action

Vincent loves Mia (S)
LOVE(VINCENT,MIA)

Vincent (NP)
λP.P@VINCENT

loves Mia (VP)
λz.LOVE(z,MIA)

loves (TV)
λX.λz.X@λx.LOVE(z,x)

Mia (NP)
λP.P@MIA

Is β-conversion always safe?

- The representations

  λx.λy.BLOOGLE(x,y)

  and

  λz.λw.BLOOGLE(z,w)

  are intended to have the same effect. The x, y, z, and w, simply mark placeholders. They have no intrinsic meaning.
- Mostly things work out fine. For example, if we apply either of the above expressions first to FEE and then to BOO, and carry out the β-conversions we get the same thing, namely BLOOGLE(FEE,BOO).

A problem

- But things can go wrong if we apply a lambda expression to a variable that occurs bound in the functor.
- For example, if we apply λx.λy.BLOOGLE(x,y) to the variable w we obtain λy.BLOOGLE(w,y) which is what we want.
- On the other hand, if we apply λz.λw.BLOOGLE(z,w) to w we obtain λw.BLOOGLE(w,w).
- This is not what we want. The variable w has become accidentally bound.

α-conversion

- α-conversion is the process of replacing (renaming) bound variables.
- For example, we obtain

  λx.λy.BLOOGLE(x,y)

  from

  λz.λw.BLOOGLE(z,w)

  by α-conversion by replacing z by x and w by y.
- When working with lambda calculus, we always α-convert before carrying out β-conversion. In particular, we always rename all the bound variables in the functor so they are distinct from all the variables in the argument.
- This prevents accidental binding.
- So our fundamental combination method is really α-conversion (for safety) followed by β-conversion.

The Three Tasks Revisited

Task 1 Specify a reasonable syntax for the fragment of natural language of interest. We can do this using DCGs.

Task 2 Specify semantic representations for the lexical items with the help of the lambda calculus. We now know what this involves.

Task 3 Specify the translation of an item $\mathcal{R}$ whose parts are $\mathcal{F}$ and $\mathcal{A}$ with the help of functional application. That is, specify which part is to be thought of as functor (here it's $\mathcal{F}$), which as argument (here it's $\mathcal{A}$) and then let the resultant translation $\mathcal{R}'$ be $\mathcal{F}'@\mathcal{A}'$. We now know that β-conversion (with the help of α-conversion), give us the tools needed to actually construct the representations built by this process.

We must now show that tasks 2 and 3 lend themselves naturally to computational implementation.

The Lambda Calculus in PROLOG

We have to do three things:

Step 1: Decide how to represent lambda expressions in Prolog. Easy — let's just do it this way:

```
lambda(X,F)
```

Step 2: Decide how to represent functional application in Prolog. Again, easy: we may as well simply use `@` (see `comsemOperators.pl`).

Step 3: Implement alpha conversion (we'll do this next).

Step 4: Implement beta conversion (we'll do this tomorrow).

Alpha Conversion in Prolog (I)

Here's the top level predicate:

```
alphaConvert(F1,F2):-
        alphaConvert(F1,[],F2).
```

The middle argument, the list, is a stack of substitutions, written `sub(X,Y)`, meaning substitute variable `X` for variable `Y`.

The idea is to work through F1, and every time we find a bound variable, record the required substitution on the stack. When we get right down to the level of variables, we carry out the required substitutions.

At the start, the stack (as shown above) is empty.

Alpha Conversion in Prolog (II)

```
alphaConvert(X,Subs,Y):-
        var(X),
        (
         memberList(sub(Z,Y),Subs),
         X==Z, !
        ;
         Y=X
        ).
```

This says: if we are dealing with a variable, and there is an appropriate substitution, perform it and never undo this choice (`X==Z, !`); otherwise return the same variable(`X=Y`).

Alpha Conversion in Prolog (III)

```
alphaConvert(Expression,Subs,exists(Y,F2)):-
        nonvar(Expression),
        Expression = exists(X,F1),
        alphaConvert(F1,[sub(X,Y)|Subs],F2).
```

This says: if we find an existentially quantified expression, make a record of the required substitution (that is, push it onto the stack).

Alpha Conversion in Prolog (IV)

And we handle universal quantifiers and lambdas analogously:

```
alphaConvert(Expression,Subs,forall(Y,F2)):-
        nonvar(Expression),
        Expression = forall(X,F1),
        alphaConvert(F1,[sub(X,Y)|Subs],F2).

alphaConvert(Expression,Subs,lambda(Y,F2)):-
        nonvar(Expression),
        Expression = lambda(X,F1),
        alphaConvert(F1,[sub(X,Y)|Subs],F2).
```


Alpha Conversion in Prolog (IV)

```
alphaConvert(F1,Subs,F2):-
        nonvar(F1),
        \+ F1 = exists(_,_),
        \+ F1 = forall(_,_),
        \+ F1 = lambda(_,_),
        compose(F1,Symbol,Args1),
        alphaConvertList(Args1,Subs,Args2),
        compose(F2,Symbol,Args2).
```

This says: turn any other kind of term into a list, and recursively alpha convert all the way down it, and then turn the alpha converted list back into a term.

Alpha Conversion in Prolog (V)

The required recursion down the list is the obvious one:

```
alphaConvertList([],_,[]).

alphaConvertList([X|L1],Subs,[Y|L2]):-
        alphaConvert(X,Subs,Y),
        alphaConvertList(L1,Subs,L2).
```


Today Lecture 3: Lambda Calculus (II)

- Carrying on from yesterday, we discuss the implementation of β-conversion in Prolog.
- We also discuss grammar engineering, and the way the programs associated with this course are organized.
- Demos:
  - `betaConversion.pl`
  - `lambda.pl`

β-Conversion in Prolog

- Because of its fundamental importance, we are going to examine the implementation of β-Conversion in Prolog in quite some detail.
- We'll start by defining the basic program. However one crucial step of this program is implemented by making direct use of Prolog unification, instead of really sitting down and thinking about unification.
- As we'll see, this approach breaks down. Worse, it breaks down in ways that are linguistically important (it does not allow us to handle coordination correctly).
- So we turn to a genuine substitution based approach. This solves the problem. Moreover, we don't have to change out earlier work drastically — in fact, we simply replace the unification component by a substitution component.
- So let's get to work ...

β-Conversion (Basic Ideas)

Basic Ideas:

- We use a stack to keep track of applications
- Applications push expressions on the stack
- Abstractions pop expressions from the stack

Clause introducing empty stack:

```
betaConvert(X,Y):-
   betaConvert(X,Y,[]).
```


β-Conversion (Variable)

A variable requires no changes:

```
betaConvert(X,Y,[]):-
   var(X),
   Y=X.
```


β-Conversion (Application)

An application is an instruction to push the argument on the stack, and then further reduce the functor (note that we α-convert the functor first):

```
betaConvert(Expression,Result,Stack):-
   nonvar(Expression),
   Expression = (X@Y),
   nonvar(X),
   alphaConvert(X,C),
   betaConvert(C,Result,[Y|Stack]).
```


β-Conversion (Unification-based)

A lambda-abstraction is an instruction to pop the stack and unify the lambda-abstracted variable with the stack member:

```
betaConvert(Expression,Result,[X|Stack]):-
   nonvar(Expression),
   Expression = lambda(X,Formula),
   betaConvert(Formula,Result,Stack).
```


β-Conversion (other cases)

All other cases are covered by breaking down the expression, β-converting it parts, and then building a new expression from its converted parts:

```
betaConvert(Formula,Result,[]):-
   nonvar(Formula),
   \+ (Formula = (X@_), nonvar(X)),
   compose(Formula,Functor,Formulas),
   betaConvertList(Formulas,ResultFormulas),
   compose(Result,Functor,ResultFormulas).
```


β-Conversion (List-wise)

Substituting a list of expressions using Prolog recursion

```
betaConvertList([],[]).
betaConvertList([Formula|Others],[Result|ResultOthers]):-
   betaConvert(Formula,Result),
   betaConvertList(Others,ResultOthers).
```


So — are we finished now?

- Well, this works up to a point.
- But there is a problem, and it's linguistically relevant.
- To see the problem, we need to think about coordination.

Example: Vincent and Mia dance

- Unsurprisingly, and has the following semantic representation
  $\lambda X.\lambda Y.\lambda P.(X@P \wedge Y@P)$.
- If we apply it to the representations for Mia and Vincent we get the following result:
  $\lambda P.(\lambda Q.Q@\textsc{vincent}@P \wedge \lambda R.R@\textsc{mia}@P)$.
- This is exactly what we need. In particular, applying this expression to $\lambda x.\textsc{dance}$ yields the formula
  $\lambda x.\textsc{dance}(x)@\textsc{vincent} \wedge \lambda x.\textsc{dance}(x)@\textsc{mia}$
- This can be reduced to the representation we would like to have:
  $\textsc{dance}(\textsc{vincent}) \wedge \textsc{dance}(\textsc{mia})$.

But now look at the Prolog...

- The lexical entry for the coordinator and will be:

```
coord(lambda(X,lambda(Y,lambda(P,(X@P)&(Y@P)))))--> [and].
```

- When we analyze Vincent and Mia dance, in the process of combining the lexical entries `X` unifies with the semantic representation of Vincent, `Y` with the semantic representation of Mia. Further, `P` gets instantiated with `lambda(X,dance(X))`.
- Then `betaConvert/3`, which uses two copies of `P`, tries to force the occurrence of `X` to unify with both `mia` *and* `vincent`. As distinct atoms do not unify, this fails.
- Oops!

So what do we do?

- The problem is, we tried to be too clever.
- As should be clear by now, β-conversion is really about substitution. We tried to make life easy for ourselves by using Prolog unification as a replacement for real substitution.
- We need to sit down, write a predicate that really carries out substitution, and use this instead.
- So let's get to work...

Substitution (I)

Glossed as: substitute `Term` for free occurrences of `Var` in `Exp` giving `Result`

```
substitute(Term,Var,Exp,Result):-
   Exp==Var, !, Result=Term.

substitute(_Term,_Var,Exp,Result):-
   \+ compound(Exp), !, Result=Exp.
```


Substitution (II)

Prevent substituting bound variables:

```
substitute(Term,Var,Formula,Result):-
   compose(Formula,Functor,[Exp,F]),
   memberList(Functor,[lambda,forall,exists]), !,
   (
    Exp==Var, !,
    Result=Formula
   ;
    substitute(Term,Var,F,R),
    compose(Result,Functor,[Exp,R])
   ).
```


Substitution(III)

All other cases:

```
substitute(Term,Var,Formula,Result):-
   compose(Formula,Functor,ArgList),
   substituteList(Term,Var,ArgList,ResultList),
   compose(Result,Functor,ResultList).
```

List-wise:

```
substituteList(_Term,_Var,[],[]).
substituteList(Term,Var,[Exp|Others],[Result|ResultOthers]):-
   substitute(Term,Var,Exp,Result),
   substituteList(Term,Var,Others,ResultOthers).
```


Only Free Variables

Important: Note that only free variables will be substituted:

```
?- substitute(a,X,man(X)&exists(X,woman(X))&p(X),R).

R = man(a)&exists(X,woman(X))&p(a)
```


β-Conversion (Substitution-based)

And that solves the problem...

A lambda-abstraction is an instruction to pop the stack and substitute the stack member for all free occurrences of `X` in `Formula`:

```
betaConvert(Expression,Result,[A|Stack]):-
   nonvar(Expression),
   Expression = lambda(X,Formula),
   substitute(A,X,Formula,New),
   betaConvert(New,Result,Stack).
```


Grammar Engineering

- The explicit notation for functional application and the implementation of β-conversion are the basic tools we shall work with in this course
- So it is time to define a bigger grammar and start exploring computational semantics
- But let's try to observe some basic principles of grammar engineering as we do so!
- we should strive for a grammar that is:
  - *modular*
  - *extendible*
  - *reusable*

Four-Level Grammar Architecture

- The Syntax Rules (Prolog DCGs)
- The Semantic Rules (mirroring the syntax rules)
- The Lexicon (a dictionary of words)
- The Semantic Macros (semantic representations for the lexical items)

The Syntax Rules and Lexicon stay fixed in the course. The Semantic Rules and Semantic Macros are the levels that involve modifications as we do most of our semantic work here.

The Syntax Rules (that we would like to use)

```
s--> [if], s, [then], s.                 s--> np, vp.

np--> np, coord, np.                     np--> det, noun.
np--> pn.                                np--> pro.
np--> whnp.                              np--> whdet, noun.

noun--> noun, coord, noun.
noun--> adj, noun.                       noun--> noun, pp.
noun--> noun, rc.                        noun--> noun, coord, noun.

vp--> vp, coord, vp.                     vp--> v(fin).
vp--> v(fin).                            vp--> mod, v(inf).
v(I)--> v(I), coord, v(I).               v(I)--> tv(I), np.
v(I)--> tv(I), np.                       v(I)--> iv(I).
v(fin)--> cop, np.                       v(fin)--> cop, neg, np.

pp--> prep, np.                          rc--> relpro, vp.
```


The Syntax Rules (that we actually use)

- Some of the rules are left-recursive, hence the standard Prolog DCG interpreter will loop when given this grammar
- as we *don't* want to implement a parser that deals with left-recursive rules, we're simply going to adopt the following fix

```
s2--> s1.                           s2--> [if], s1, [then], s1.
s1--> np2, vp2.                     np2--> np1.
np2--> np1, coord, np1.             np1--> det, n2.
np1--> pn.                          noun2--> noun1.
noun2--> noun1, coord, noun1.       noun1--> noun.
noun1--> noun, pp.                  noun1--> noun, rc.
```

This is a brutal way of dealing with the problem, but it enables us to generate the examples we want without having to worry about left-recursion.

The Semantic Annotations

The Syntactic Rules have placeholders for semantic information:

```
s1(S1)--> np2(NP2), vp2(VP2),
          {combine(s1:S1,[np2:NP2,vp2:VP2])}.

np1(NP1)--> det(Det), noun2(N2),
            {combine(np1:NP1,[det:Det,n2:N2])}.

np1(NP1)--> pn(PN),
            {combine(np1:NP1,[pn:PN])}.
```

How the semantic information is passed upwards the tree is specified by the semantic rules (instances of `combine/2`)

The Semantic Rules

The required semantic annotations for our implementation of the lambda calculus are utterly straightforward; they are simply the obvious "apply the function to the argument statements" expressed with the help of `@`:

```
combine(np1:(A@B),[det:A,n2:B]).
combine(n2:((B@A)@C),[n1:A,coord:B,n1:C]).
combine(np1:A,[pn:A]).
```

Of course, because `combine/3` offers us extreme flexibility in implementing semantic construction, we can also choose to apply β-conversion directly:

```
combine(s1:Converted,[np2:A,vp2:B]):-
   betaConvert(A@B,Converted).
```


Lexical Rules

Lexical rules apply to terminal symbols (the actual strings in the input of the parser) and need to call the lexicon to check if a string belongs to the syntactic category searched for.

```
noun(Noun)-->
   {
    lexicon(noun,Sym,Phrase,_),
    nounSem(Sym,Noun)
   },
   Phrase.
```

Each lexical category is associated with such a macro, and this enables us to abstract away from specific types of structures. So we have set up the syntax rules in such a way that we are independent from the semantic theory we want to work with!

The Lexicon

The general format of a lexical entry is:

```
lexicon(Cat,Sem,Phrase,Misc)
```

where

- `Cat` is the syntactic category
- `Sem` is the semantic information introduced by the phrase
- `Phrase` is the string of words that span the phrase
- `Misc` miscellaneous information depending on the type of entry.

For example, the entries for intransitive verbs are:

```
lexicon(iv,collapse,[collapses],fin).
lexicon(iv,collapse,[collapse],inf).
lexicon(iv,dance,[dances],fin).
lexicon(iv,dance,[dance],inf).
```


The Semantic Macros specify the lexical semantics

```
nounSem(Sym,lambda(X,F)):-
   compose(F,Sym,[X]).

pnSem(Sym,lambda(P,P@Sym)).

tvSem(Sym,lambda(K,lambda(Y,K@lambda(X,F)))):-
   compose(F,Sym,[Y,X]).

detSem(uni,lambda(P1,lambda(P2,forall(X,(P1@X)>(P2@X))))).

detSem(indef,lambda(P1,lambda(P2,exists(X,(P1@X)&(P2@X))))).
```

Today Lecture 4: Scope Ambiguities

- We discuss the problem of scope ambiguities (concentrating on quantifier scope ambiguities) and look for ways of dealing with them computationally.
- We first discuss *Montague's Trick* (*quantifier raising*).
- This leads us on naturally to *Cooper Storage* and *Keller storage*. We implement both approaches in Prolog.
- Finally, we say a little bit about modern *underspecification* approaches to such problems.
- Demos:
  - `cooperStorage.pl`
  - `kellerStorage.pl`

Quantifier Scope Ambiguities

Sentences with quantifiers trigger multiple interpretations ('readings'), e.g.:

Mia told a joke in every episode (2 readings)

Most actors tell a joke in every film (5 readings)

Context disambiguates:

A cyclist is hit by a bus every day in Helsinki.
He doesn't appreciate it very much.

Consider our methods for semantic construction developed so far work with

Every boxer loves a woman (2 readings)

Every boxer loves a woman (S)
$\forall x(\text{BOXER}(x)\rightarrow \exists y(\text{WOMAN}(y)\wedge\text{LOVE}(x,y)))$

Every boxer (NP)
$\lambda P.\forall x(\text{BOXER}(x)\rightarrow P@x)$

loves a woman (VP)
$\lambda z.\exists y(\text{WOMAN}(y)\wedge\text{LOVE}(z,y))$

loves (TV)
$\lambda X.\lambda z.X@\lambda x.\text{LOVE}(z,x)$

a woman (NP)
$\lambda Q.\exists y(\text{WOMAN}(y)\wedge Q@y)$

However, Every boxer loves a woman has a second meaning:

$$\exists y(\text{WOMAN}(y)\wedge\forall x(\text{BOXER}(x)\rightarrow\text{LOVE}(x,y)))$$

Is this a genuine Problem?

Consider again **Every boxer loves a woman**.

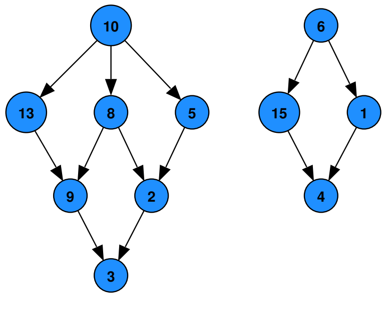
- In a sense, the first representation is sufficient to cover both readings of our example, since it is entailed by the other reading.
- Arguably, that makes the stronger representation superfluous: perhaps it is pragmatically inferred from the weaker one with the help of contextual knowledge.

So do we really need techniques for coping with quantifier ambiguity?

We do:

- direct construction doesn't always lead to the weakest reading
- quantifier scope ambiguities do not always give rise to logically *independent* readings

Every owner of a hash bar gives every criminal a big kahuna burger
(15 initial readings, 11 groups of logically equivalent readings)



Gabsdil & Striegnitz (ICoS-1)

Node 1: (reading 1)
1:∀A(criminal(A)&male(A)→∃B(bkburger(B)&(neuter(B)&∀C(∀D(hashbar(D)&(neuter(C)&(neuter(D)&(owner(C)&of(C,D))))→give(C,A,B))))))

Node 2: (reading 2)
2:∀A(criminal(A)&male(A)→∃B(∃C(bkburger(C)&(hashbar(B)&(neuter(B)&(neuter(C)&∀D(neuter(D)&(owner(D)&of(D,B))→give(D,A,C))))))))

Node 3: (reading 3)
3:∀A(criminal(A)&male(A)→∃B(hashbar(B)&(neuter(B)&∀C(neuter(C)&(owner(C)&of(C,B))→∃D(bkburger(D)&(neuter(D)&give(C,A,D)))))))

Node 4: (reading 4 and 14)
4:∀A(criminal(A)&male(A)→∀B(∀C(hashbar(C)&(neuter(B)&(neuter(C)&(owner(B)&of(B,C))))→∃D(bkburger(D)&(neuter(D)&give(B,A,D))))))
14:∀A(∀B(hashbar(B)&(neuter(A)&(neuter(B)&(owner(A)&of(A,B))))→∀C(criminal(C)&male(C)→∃D(bkburger(D)&(neuter(D)&giveA(C,D))))))

Node 5: (reading 5)
5:∃A(bkburger(A)&(neuter(A)&∀B(criminal(B)&male(B)→∃C(hashbar(C)&(neuter(C)&∀D(neuter(D)&(owner(D)&of(D,C))→give(D,B,A)))))))

Node 6: (reading 6 and 7)
6:∃A(bkburger(A)&(neuter(A)&∀B(criminal(B)&male(B)→∀C(∀D(hashbar(D)&(neuter(C)&(neuter(D)&(owner(C)&of(C,D))))→give(C,B,A))))))
7:∃A(bkburger(A)&(neuter(A)&∀B(∀C(hashbar(C)&(neuter(B)&(neuter(C)&(owner(B)&of(B,C))))→∀D(criminal(D)&male(D)→give(B,D,A))))))

Node 8: (reading 8)
8:∃A(hashbar(A)&(neuter(A)&∀B(criminal(B)&male(B)→∃C(bkburger(C)&(neuter(C)&∀D(neuter(D)&(owner(D)&of(D,A))→give(D,B,C)))))))

Node 9: (reading 9 and 12)
9:∃A(hashbar(A)&(neuter(A)&∀B(criminal(B)&male(B)→∀C(neuter(C)&(owner(C)&of(C,A))→∃D(bkburger(D)&(neuter(D)&give(C,B,D)))))))
12:∃A(hashbar(A)&(neuter(A)&∀B(neuter(B)&(owner(B)&of(B,A))→∀C(criminal(C)&male(C)→∃D(bkburger(D)&(neuter(D)&give(B,C,D)))))))

Node 10: (reading (10 and 11)
10:∃A(∃B(bkburger(B)&(hashbar(A)&(neuter(A)&(neuter(B)&∀C(criminal(C)&male(C)→∀D(neuter(D)&(owner(D)&of(D,A))→give(D,C,B))))))))
11:∃A(∃B(bkburger(B)&(hashbar(A)&(neuter(A)&(neuter(B)&∀C(neuter(C)&(owner(C)&of(C,A))→∀D(criminal(D)&male(D)→give(C,D,B))))))))

Node 13: (reading 13)
13:∃A(hashbar(A)&(neuter(A)&∀B(neuter(B)&(owner(B)&of(B,A))→∃C(bkburger(C)&(neuter(C)&∀D(criminal(D)&male(D)→give(B,D,C)))))))

Node 15: (reading 15)
15:∀A(∀B(hashbar(B)&(neuter(A)&(neuter(B)&(owner(A)&of(A,B))))→∃C(bkburger(C)&(neuter(C)&∀D(criminal(D)&male(D)→give(A,D,C))))))

Montague's Approach

- Rule of Quantification ("Quantifier Raising")
- Combine with 'indexed pronoun' instead of the quantifying noun phrase
- Indexed pronouns are placeholders for the quantifying noun phrase
- When the placeholder has moved high enough in the tree to give us the scoping we're interested in, we are permitted to replace it by the quantifying NP

Montague's Approach ("Quantifier Raising")

Every boxer loves a woman (S)
$\exists y(\text{WOMAN}(y)\ \&\ \forall x(\text{BOXER}(x)\rightarrow\text{LOVE}(x,y)))$

a woman (NP)
$\lambda P.\exists y(\text{WOMAN}(y)\ \&\ P@y)$

Every boxer loves her-3 (S)
$\lambda z_3.\forall x(\text{BOXER}(x)\rightarrow\text{LOVE}(x,z_3))$

Every boxer (NP)
$\lambda P.\forall x(\text{BOXER}(x)\rightarrow P@x)$

loves her-3 (VP)
$\lambda y.\text{LOVE}(y,z_3))$

loves (TV)
$\lambda X.\lambda y.X@\lambda x.\text{LOVE}(y,x)$

her-3 (NP)
$\lambda P.P@z_3$

What can we say about Montague's Approach?

- The basic idea is good. In particular, the idea that we have a dummy semantic representation which we pull out when we need it, is good. In fact, we'll be seeing a lot more of this idea today.
- But the way Montague makes use of this idea is bad. It places heavy burden on the grammar rules. We don't want to play with the grammar — we want to 'bolt on' semantic construction to existing grammars.
- Storage methods, which we shall now discuss in detail, in essence move the key component of Montague's approach from the syntax, and into the semantic construction component — where it belongs.

Cooper storage

Key ideas:

- associate each node of a parse tree with a *store*
- a store contains a 'core' semantic representation together with the quantifiers associated with nodes lower in the tree
- after the sentence is parsed, the store is used to generate scoped representations
- the order in which the store is retrieved determines the different scopings

Stores more formally

- a store is an n-place sequence
- stores are represented by the angle brackets $\langle$ and $\rangle$
- the first item of the sequence is the core semantic representation
- subsequent elements are pairs (β, i) where β is the semantic representation of an NP (that is, another lambda expression) and i is an index.
- an index is a label which picks out a free variable in the core semantic representation

The Storage Rule

- How do we use stores for semantic construction?
- Quantified noun phrases are free to 're-package' the information that the store contains

More precisely:

Storage (Cooper)
If the store $\langle \phi, (\beta, j), \ldots, (\beta', k)\rangle$ is a semantic representation for a quantified NP, then the store $\langle \lambda\text{P.P@z}_i, (\phi, i), (\beta, j), \ldots, (\beta', k)\rangle$, where i is some unique index, is also a representation for that NP.

Every boxer loves a woman (S)
$\langle \text{LOVE}(z_6,z_7),\ (\lambda P.\forall x(\text{BOXER}(x)\rightarrow P@x),6),\ (\lambda P.\exists y(\text{WOMAN}(y)\wedge P@y),7)\rangle$

Every boxer (NP)
$\langle \lambda Q.Q@z_6,\ (\lambda P.\forall x(\text{BOXER}(x)\rightarrow P@x),6)\rangle$

loves a woman (VP)
$\langle \lambda u.\text{LOVE}(u,z_7),\ (\lambda P.\exists y(\text{WOMAN}(y)\wedge P@y),7)\rangle$

loves (TV)
$\langle \lambda X.\lambda u.X@\lambda v.\text{LOVE}(u,v)\rangle$

a woman (NP)
$\langle \lambda Q.Q@z_7,\ (\lambda P.\exists y(\text{WOMAN}(y)\wedge P@y),7)\rangle$

Retrieval (I)

We really want ordinary scoped first-order representations!

How do we do get these?

- remove one of the indexed binding operators from the store
- combine it with the core representation
- the result is a new core representation
- continue this procedure until the store contains one element

Retrieval (II)

More precisely:

Retrieval (Cooper)
Let σ_1 and σ_2 be (possibly empty) sequences of binding operators. If the store $\langle \phi, \sigma_1, (\beta, i), \sigma_2\rangle$ is associated with an expression of category S, then the store $\langle \beta@\lambda z_i.\phi, \sigma_1, \sigma_2\rangle$ is also associated with this expression.

Back to our example:

$$\langle \text{LOVE}(z_6,z_7),\ (\lambda P.\forall x(\text{BOXER}(x)\rightarrow P@x),6),\ (\lambda P.\exists y(\text{WOMAN}(y)\wedge P@y),7)\rangle$$

Apply the retrieval rule to the store associated with the S node:

$$\langle \lambda P.\forall x(\text{BOXER}(x)\rightarrow P@x)@\lambda z_6.\text{LOVE}(z_6,z_7),\ (\lambda P.\exists y(\text{WOMAN}(y)\wedge P@y),7)\rangle$$

Using β-conversion, this simplifies to:

$$\langle \forall x(\text{BOXER}(x)\rightarrow \text{LOVE}(x,z_7)),\ (\lambda P.\exists y(\text{WOMAN}(y)\wedge P@y),7)\rangle$$

Retrieving the quantifier left in store:

$$\langle \lambda P.\exists y(\text{WOMAN}(y)\wedge P@y)@\lambda z_7.\forall x(\text{BOXER}(x)\rightarrow \text{LOVE}(x,z_7))\rangle$$

Final result:

$$\langle \exists y(\text{WOMAN}(y)\wedge \forall x(\text{BOXER}(x)\rightarrow \text{LOVE}(x,y)))\rangle$$

Question: How do we get the other reading?

Nested Noun Phrases

- Cooper storage allows us a great deal of freedom
- We are allowed to retrieve quantifiers in any order we like
- The only safety net provided is the use of co-indexed variables

Is this really safe?

Every piercing that is done with a gun goes against the entire idea behind it.

Mia knows every owner of a hash bar.

Nested Noun Phrases: a problem...

Store:

$$\langle \textsc{know}(\textsc{mia},z_2),\ (\lambda P.\forall y(\textsc{owner}(y)\wedge\textsc{of}(y,z_1)\rightarrow P@y),2),\ (\lambda Q.\exists x(\textsc{hashbar}(x)\wedge Q@x),1)\rangle$$

Option 1:

$$\langle\forall y(\textsc{owner}(y)\wedge\textsc{of}(y,z_1)\rightarrow\textsc{know}(\textsc{mia},y)),(\lambda Q.\exists x(\textsc{hashbar}(x)\wedge Q@x),1)\rangle$$

$$\downarrow$$

$$\langle\exists x(\textsc{hashbar}(x)\wedge\forall y(\textsc{owner}(y)\wedge\textsc{of}(y,x)\rightarrow\textsc{know}(\textsc{mia},y)))\rangle$$

Option 2:

$$\langle\exists x(\textsc{hashbar}(x)\wedge\textsc{know}(\textsc{mia},z_2)),\ (\lambda P.\forall y(\textsc{owner}(y)\wedge\textsc{of}(y,z_1)\rightarrow P@y),2)\rangle$$

$$\downarrow$$

$$\langle\forall y(\textsc{owner}(y)\wedge\textsc{of}(y,z_1)\rightarrow\exists x(\textsc{hashbar}(x)\wedge\textsc{know}(\textsc{mia},y)))\rangle$$

An elegant solution: nested stores (Bill Keller)

- allow stores to contain other stores
- the nesting structure of the stores should automatically track the nesting structure of NPs
- nesting is easier to implement than a free variable check

Here's the new storage rule:

Storage (Keller)
If the (nested) store $\langle\phi,\sigma\rangle$ is an interpretation for an NP, then the (nested) store $\langle\lambda P.P@z_i,(\langle\phi,\sigma\rangle,i)\rangle$, for some unique index i, is also an interpretation for this NP.

every owner of a hash bar (NP)

$\lambda P.P@z_2,(\langle\lambda P.\forall y(\textsc{owner}(y)\wedge\textsc{of}(y,z_1)\rightarrow P@y),(\langle\lambda Q.\exists x(\textsc{hashbar}(x)\wedge Q@x)\rangle,1)\rangle,2)\rangle$

every (DET)

$\lambda Q.\lambda P.\forall y(Q@y\rightarrow P@y)\rangle$

owner of a hash bar (NBAR)

$\langle\lambda u.\textsc{owner}(u)\wedge\textsc{of}(u,z_1),\ (\langle\lambda Q.\exists x(\textsc{hashbar}(x)\wedge Q@x)\rangle,1)\rangle$

owner (N)

$\langle\lambda x.\textsc{owner}(x)\rangle$

of a hash bar (PP)

$\langle\lambda P.\lambda u.P@u\wedge\textsc{of}(u,z_1),(\langle\lambda Q.\exists x(\textsc{hashbar}(x)\wedge Q@x)\rangle,1)\rangle$

Retrieval for Nested Storage

The new retrieval rule:

Retrieval (Keller)
Let σ, σ_1 and σ_2 be (possibly empty) sequences of binding operators.
If the (nested) store $\langle\phi,\sigma_1,(\langle\beta,\sigma\rangle,i),\sigma_2\rangle$ is an interpretation for an expression of category S, then $\langle\beta@\lambda z_i.\phi,\sigma_1,\sigma,\sigma_2\rangle$ is too.

The rule ensures that any operators stored while processing β become accessible for retrieval only after β itself has been retrieved.

Nesting automatically overcomes the problem of generating readings with free variables.

Mia knows every owner of a hash bar (Reading 1)

Nested Store:

$\langle$
  $\textsc{know}(\textsc{mia},z_2)$,
  (
    $\langle$
      $\lambda P.\forall y(\textsc{owner}(y)\wedge\textsc{of}(y,z_1)\rightarrow P@y)$,
      (
        $\langle\lambda Q.\exists x(\textsc{hashbar}(x)\wedge Q@x)\rangle$,1
)
    $\rangle$,2)
$\rangle$

There is only one way to perform retrieval:
$\langle\exists x(\textsc{hashbar}(x)\wedge\forall y(\textsc{owner}(y)\wedge\textsc{of}(y,x)\rightarrow\textsc{know}(\textsc{mia},y)))\rangle$

Mia knows every owner of a hash bar (Reading 2)

every owner of a hash bar (NP)

$\langle\lambda P.P@z_2,\ (\langle\lambda P.\forall y(\textsc{owner}(y)\wedge\exists x(\textsc{hashbar}(x)\wedge\textsc{of}(z,x))\rightarrow P@y)\rangle,2)\rangle$

every (DET)

$\langle\lambda Q.\lambda P.\forall x(Q@x\rightarrow P@x)\rangle$

owner of a hash bar (NBAR)

$\langle\lambda z.\textsc{owner}(z)\wedge\exists x(\textsc{hashbar}(x)\wedge\textsc{of}(z,x))\rangle$

owner (N)

$\langle\lambda x.\textsc{owner}(x)\rangle$

of a hash bar (PP)

$\langle\lambda P.\lambda z.P@z\wedge\exists x(\textsc{hashbar}(x)\wedge\textsc{of}(z,x))\rangle$

(avoid storing the nested NP a hashbar)

Mia knows every owner of a hash bar (Reading 2) II

This leads to the following analysis for:

$$\langle\textsc{know}(\textsc{mia},z_2),(\langle\lambda P.\forall y(\textsc{owner}(y)\wedge\exists x(\textsc{hashbar}(x)\wedge\textsc{of}(y,x))\rightarrow P@y)\rangle,2)\rangle$$

There is only one operator the store. Retrieving it yields the reading we want.

$$\langle\forall y(\textsc{owner}(y)\wedge\exists x(\textsc{hashbar}(x)\wedge\textsc{of}(y,x))\rightarrow\textsc{know}(\textsc{mia},y))\rangle$$

Basic Messages:
(1) Pushing a quantifier on the store is a non-deterministic choice
(2) Use nested stores to deal with complex NPs

Quantifier Storage: Conclusion

- The original version of Cooper storage doesn't handle storage and retrieval in a sufficiently disciplined way
- This causes it to generate spurious readings (in fact, nonsensical readings) when faced with nested NPs
- The problem can be elegantly cured by making use of nested stores
- Implementing this idea requires only trivial changes to our earlier PROLOG code.
- A little care is required in implementing optionality if we want to avoid generating one reading several times.

Limitations of Storage Methods

- they are not as expressive as we might wish
- storage is essentially a technique which enables us to represent all possible meanings compactly
- it doesn't allow us to express additional *constraints* on possible readings
- this is precisely what most modern underspecified representations let us do

Reflection

- Storage vs. Constraint-based Underspecification
  - Expressive Power
  - Computational Properties
- We discussed how to cover all readings.
- But not how to choose a preferred reading.

Today Lecture 5: Propositional Inference

- We begin studying the second of the fundamental questions we began the course with: How can we use semantic representations of natural language expressions to automate the process of drawing inferences?
- Today we discuss inference for the quantifier free fragment of first-order logic, or propositional logic, as it is usually called.
- First we discuss the tableaux method.
- We then discuss the resolution method.
- Demos:
  - `propTableaux.pl`
  - `propResolution.pl`

Propositional logic

- First-order inference is a computationally complex task (in fact, an undecidable one) so we're going to have to get to grips with it gradually.
- Today we will look at the quantifier free fragment of first-order logic, or propositional logic as it's usually called. That is, today we will be learning about inferences involving $\neg$, $\rightarrow$, $\vee$ and $\wedge$. We'll discuss the quantifiers in Lecture 6.
- Instead of writing (say)

  $$(\textsc{dead}(\textsc{vincent})\rightarrow\textsc{happy}(\textsc{butch}))\wedge(\neg\textsc{dead}(\textsc{vincent})\rightarrow\textsc{happy}(\textsc{mia}))$$

  we will simply write

  $(p \rightarrow q) \wedge (\neg p \rightarrow r)$.

  The internal makeup of the atomic symbols isn't important in propositional logic. We call symbols like p, q, and r sentence symbols.

Which inference task?

- We discussed querying in Lecture 1. So today we want to talk about the other two tasks, namely consistency checking and informativity checking.
- But we can kill two birds with one stone. Remember from Lecture 1 that
  - ϕ is informative (that is, not valid) if and only if $\neg\phi$ is consistent.
  - ϕ is uninformative (that is, valid) if and only if $\neg\phi$ is inconsistent.
- The practical consequence of this is that once we have computational tool for solving one problem, we automatically have a tool for solving the other, which is nice.

Which inference tool?

- We mentioned in Lecture 1 that there are two tools that can help us with these tasks: theorem provers and model builders.
- We are going to concentrate on learning about theorem provers. These are computational tools for telling us whether a formula is valid (uninformative) or not.
- We will say a little bit about model building today and in Lecture 6, but we won't study it in any depth.

How do we approach the task?

- We shall study two techniques from the branch of logic called proof theory.
- Proof theory is the study of syntactic techniques for determining whether a formula is valid (uninformative) or not.
- We shall look at two proof theoretic techniques that have turned out to be computationally important, namely tableaux and resolution.
- Although they both carry out the same task, they work very differently. It is important (and interesting) to know a bit about both.

The Tableaux Method

- Syntactic, but based on clear semantic intuitions.
- Finding a tableaux proof does not *depend* on human insight.
- Can be adapted to many different logics.
- Tableaux systems are more than just theorem provers: they can also be regarded as *model building tools*.

The basic idea

> *Suppose we are given a formula, and one of the truth values* True *or* False*. Is it possible to find a model in which the given formula has the given truth value?*

The tableaux method is essentially a *syntactic* way of *systematically* checking whether or not this is possible.

Gives us a systematic test for validity. The valid formulas are those that the tableaux method *cannot* falsify.

So the tableaux method is a refutation method. That is, to prove ϕ, we input $\neg\phi$ and show that this leads to problems.

Example 1

The formula $p \vee \neg p$ is valid. What would a systematic search for a falsification looks like?

One possible answer: fill out its truth table. But this won't generalize to first-order logic, and it's only practical for small formulas.

Instead we'll develop *tableaux expansion rules*.

Example 1 continued...

$$F(p \vee \neg p).$$

- This is our first tableau!
- The 'F' prefix means that we want to *falsify* $p \vee \neg p$
- When writing tableaux by hand, it's handy to include extra book-keeping information, like line numbers:

$$1 \quad F(p \vee \neg p)$$

Example 1 continued...

$$\begin{array}{lcl} 1 & F(p \vee \neg p) & \surd \\ 2 & Fp & 1, F_{\vee} \\ 3 & F\neg p & 1, F_{\vee} \end{array}$$

- Our second tableau!
- Uses the *tableaux expansion rule* called $F_{\vee}$ (*falsify a disjunction*) to break up the formula in line 1 into two pieces.
- The $\surd$ symbol in line 1 shows that we've applied the appropriate rule to line 1. (We *never* need to apply a rule to the same line twice, which is nice.)

Example 1 continued...

$$\begin{array}{lcl} 1 & F(p \vee \neg p) & \surd \\ 2 & Fp & 1, F_{\vee} \\ 3 & F\neg p & 1, F_{\vee}, \surd \\ 4 & Tp & 3, F_{\neg}. \end{array}$$

- In line 3 we used expansion rule $F_{\neg}$ (*falsify a negation*)
- We're finished! The tableau is *rule saturated* (that is, we can't apply any more rules).
- The tableau is also *closed* (because of the conflict between lines 2 and 4).

Thus $p \vee \neg p$ is a validity.

Example 2

Is $(q \wedge r) \rightarrow (\neg q \vee \neg r)$ valid? Let's see ...

$$1 \quad F\neg(q \wedge r) \rightarrow (\neg q \vee \neg r)$$

We use rule $F_{\rightarrow}$, which tells us how to falsify an implication.

$$\begin{array}{lcl} 1 & F\neg(q \wedge r) \rightarrow (\neg q \vee \neg r) & \surd \\ 2 & T\neg(q \wedge r) & 1, F_{\rightarrow} \\ 3 & F(\neg q \vee \neg r) & 1, F_{\rightarrow} \end{array}$$

Line 3 wants us to *falsify a disjunction*. We use rule $F_{\vee}$...

Example 2 continued...

$$\begin{array}{lcl} 1 & F\neg(q \wedge r) \rightarrow (\neg q \vee \neg r) & \surd \\ 2 & T\neg(q \wedge r) & 1, F_{\rightarrow} \\ 3 & F(\neg q \vee \neg r) & 1, F_{\rightarrow}, \surd \\ 4 & F\neg q & 3, F_{\vee} \\ 5 & F\neg r & 3, F_{\vee} \end{array}$$

Why are we free to work on line 3? We haven't dealt with line 2!

No problem! We're free to choose. (One of the nice things about tableaux.)

Example 2 continued...

1	$F\neg(q \wedge r) \rightarrow (\neg q \vee \neg r)$	$\surd$
2	$T\neg(q \wedge r)$	$1, F_{\rightarrow}$
3	$F(\neg q \vee \neg r)$	$1, F_{\rightarrow}, \surd$
4	$F\neg q$	$3, F_{\vee}, \surd$
5	$F\neg r$	$3, F_{\vee}, \surd$
6	Tq	$4, F_{\neg}$
7	Tr	$5, F_{\neg}$

Example 2 continued...

1	$F\neg(q \wedge r) \rightarrow (\neg q \vee \neg r)$	$\surd$
2	$T\neg(q \wedge r)$	$1, F_{\rightarrow}, \surd$
3	$F(\neg q \vee \neg r)$	$1, F_{\rightarrow}, \surd$
4	$F\neg q$	$3, F_{\vee}, \surd$
5	$F\neg r$	$3, F_{\vee}, \surd$
6	Tq	$4, F_{\neg}$
7	Tr	$5, F_{\neg}$
8	$F(q \wedge r)$	$2, T_{\neg}$

Next, we must deal with line 8. Here things get interesting...

Example 2 continued...

1	$F\neg(q \wedge r) \rightarrow (\neg q \wedge \neg r)$	$\surd$
2	$T\neg(q \wedge r)$	$1, F_{\rightarrow}, \surd$
3	$F(\neg q \wedge \neg r)$	$1, F_{\rightarrow}, \surd$
4	$F\neg q$	$3, F_{\vee}, \surd$
5	$F\neg r$	$3, F_{\vee}, \surd$
6	Tq	$4, F_{\neg}$
7	Tr	$5, F_{\neg}$
8	$F(q \wedge r)$	$2, T_{\neg}, \surd$

9	Fq	$8, F_{\wedge}$	10	Fr	$8, F_{\wedge}$

Example 3

Let's consider what happens if the formula we are working with is *not* a validity, for example, $(p \wedge q) \rightarrow (r \vee s)$.

$$1 \quad F(p \wedge q) \rightarrow (r \vee s)$$

We need to falsify an implication, so we use rule $F_{\rightarrow}$ ...

Example 3 continued...

1	$F(p \wedge q) \rightarrow (r \vee s)$	$\surd$
2	$T(p \wedge q)$	$1, F_{\rightarrow}$
3	$F(r \vee q)$	$1, F_{\rightarrow}$

Line 2 demands that we make a conjunction true. We use expansion rule $T_{\wedge}$.

1	$F(p \wedge q) \rightarrow (r \vee s)$	$\surd$
2	$T(p \wedge q)$	$1, F_{\rightarrow}, \surd$
3	$F(r \vee q)$	$1, F_{\rightarrow}$
4	Tp	$2, T_{\wedge}$
5	Tq	$2, T_{\wedge}$

Example 3 continued...

Now let's deal with line 3. The relevant rule is $F_{\vee}$.

1	$F(p \wedge q) \rightarrow (r \vee s)$	$\surd$
2	$T(p \wedge q)$	$1, F_{\rightarrow}, \surd$
3	$F(r \vee s)$	$1, F_{\rightarrow}, \surd$
4	Tp	$2, T_{\wedge}$
5	Tq	$2, T_{\wedge}$
6	Fr	$3, F_{\vee}$
7	Fs	$3, F_{\vee}$

We're finished — but the tableau is not closed, it is *open*. Thus the input formula was *not* valid. And we can read off a (propositional) model from the tableaux.

Expansion rules for negation

$$\frac{T\neg\phi}{F\phi} \qquad \frac{F\neg\phi}{T\phi}$$

- Read these rules from top to bottom. The signed formula above the horizontal line is the *input* to the rule, and the signed formula below it is the *output*.
- We call these two rules *unary rules*, as both return a single formula as output.

Expansion rules for binary connectives

$$\frac{T(\phi \wedge \psi)}{\begin{array}{c}T\phi\\ T\psi\end{array}} \qquad \frac{F(\phi \wedge \psi)}{F\phi \mid F\psi}$$

$$\frac{F(\phi \vee \psi)}{\begin{array}{c}F\phi\\ F\psi\end{array}} \qquad \frac{T(\phi \vee \psi)}{T\phi \mid T\psi}$$

$$\frac{F(\phi \rightarrow \psi)}{\begin{array}{c}T\phi\\ F\psi\end{array}} \qquad \frac{T(\phi \rightarrow \psi)}{F\phi \mid T\psi}$$

Propositional tableaux

- A (propositional) *tableau* is a tree, each of whose nodes is a signed (propositional) formula. A *branch* of a tableau is a branch of such a tree.
- We start with an *initial tableau*. This can be *any* tableau that has only has a single branch.
- Tableaux expansion works as follows. Given a tableau, try to find a node that (a) isn't a signed atomic formula, and (b) to which we haven't already applied an expansion rule (we call such nodes *unexpanded nodes*).

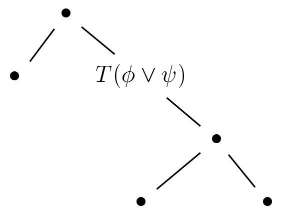
Propositional tableaux continued

- If there are no unexpanded nodes, **STOP**!
  The tableau is rule saturated.
- If there are unexpanded nodes, choose one and apply the relevant expansion rule. That is, extend the tableau by adding on new nodes in the way demanded by the rule.

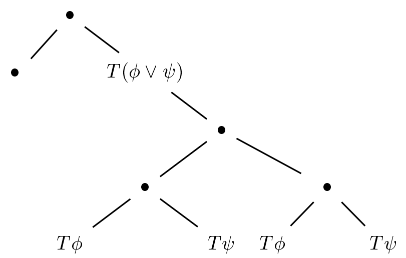
One point of clarification ...

A signed formula may belong to several branches. When we perform an expansion, we have to extend every *branch on which the input formula lies.*

For example, consider this tableau:



If we expand the indicated node $T(\phi \vee \psi)$, we have to do so as follows:



Closed and Open Tableaux

- A branch of a tableau is *closed* if it contains both $T\phi$ and $F\phi$, where ϕ is some formula.
- A branch that is not closed is called *open*.
- A tableau is closed if *every* branch it contains is closed, and open if it contains *at least one* open branch.

The key definition

A formula ϕ is tableaux-provable *(or, more simply:* provable*) if and only if it is possible to expand the initial tableau consisting of a single node $F\phi$ to a closed tableau. The notation $\vdash \phi$ means that ϕ is provable.*

Testing for uninformativity

Simply use the following initial tableau:

$$T\phi_1$$
$$.$$
$$.$$
$$.$$
$$T\phi_n$$
$$F\psi$$

That is, try to make all the premisses true and the conclusion false. If we *can't* do this — that is, if we obtain a closed tableau — the argument is valid. Or to put it another way: ψ is uniformative with respect to $\phi_1 \ldots \phi_n$.

Soundness and Completeness

- The tableaux system is *sound*. That is, for any propositional formula ϕ,

  if $\vdash \phi$ then $\models \phi$.

  Tableaux proofs will never lead us astray.
- The tableaux system is *complete*. That is, for any propositional formula ϕ,

  If $\models \phi$ then $\vdash \phi$.

  No validity lies beyond the reach of the tableaux proof method: if a formula ϕ is valid, then it is possible to expand the initial tableau $F\phi$ to a closed tableau.

Tableaux in Prolog

Tableaux in Prolog are lists containing lists of signed formulas.

Signed formulas are represented as normal Prolog terms with functors `f` and `t` of arity 1.

For example:

```
[[f(p > q)]].
```

This represents the tableau with just one branch, namely a branch containing the single signed formula $F(p \rightarrow q)$.

A second example

```
[[t(p & q), f(p & r)]].
```

This represents a tableau with just one branch. This time, however, the branch contains *two* signed formulas, namely $T(p \wedge q)$ and $F(p \wedge r)$.

The Prolog implementation of propositional tableaux consists of a small collection of predicates which manipulate such lists in accordance with the tableaux expansion rules.

For example, it is easy to define a predicate which when given

```
[[t(p & q), f(p & r)]].
```

as input, returns the following one as output:

```
[[t(p), t(q), f(p & r)]].
```

Note that the output does *not* contain `t(p & q)`. Why is this?

We can further process

```
[[t(p), t(q), f(p & r)]].
```

We need to apply $F_{\wedge}$, a *disjunctive* rule. Here's what we get back in our Prolog implementation:

```
[[t(p), t(q), f(p)], [t(p), t(q), f(r)]]
```

Note that this list contains *two* lists.
(Our Prolog implementation copies whole branches.)

The Prolog Implementation

The main predicate is `closed tableaux`, which recursively attempts to build a closed tableau.

```
closedTableau([]).

closedTableau(OldTableau):-
   expand(OldTableau,NewTableau),
   closedTableau(NewTableau).
```


An organisational predicate

```
expand([Branch|Tableau],NewTableau):-
   unaryExpansion(Branch,NewBranch),!,
   insert(NewBranch,Tableau,NewTableau).

expand([Branch|Tableau],NewTableau):-
   conjunctiveExpansion(Branch,NewBranch),!,
   insert(NewBranch,Tableau,NewTableau).

expand([Branch|Tableau],NewTableau):-
   disjunctiveExpansion(Branch,NewBranch1,NewBranch2),!,
   insert(NewBranch1,Tableau,TempTableau),
   insert(NewBranch2,TempTableau,NewTableau).

expand([Branch|Rest],[Branch|Newrest]):-
   expand(Rest,Newrest).
```


Inserting branches

```
insert(Branch,Tableau,NewTableau) :-
   (
      closedBranch(Branch),
      NewTableau = Tableau, !
    ;
      NewTableau = [Branch|Tableau]
   ).

closedBranch(Branch):-
    memberList(t(X),Branch),
    memberList(f(X),Branch).
```


The real work

```
unaryExpansion(Branch,[Component|Temp]) :-
   unary(SignedFormula,Component),
   removeFirst(SignedFormula,Branch,Temp).

conjunctiveExpansion(Branch,[Comp1,Comp2|Temp]):-
   conjunctive(SignedFormula,Comp1,Comp2),
   removeFirst(SignedFormula,Branch,Temp).

disjunctiveExpansion(Branch,[Comp1|Temp],[Comp2|Temp]):-
   disjunctive(SignedFormula,Comp1,Comp2),
   removeFirst(SignedFormula,Branch,Temp).
```

Final tasks

```
conjunctive(t(X & Y),t(X),t(Y)).
conjunctive(f(X v Y),f(X),f(Y)).
conjunctive(f(X > Y),t(X),f(Y)).

disjunctive(f(X & Y),f(X),f(Y)).
disjunctive(t(X v Y),t(X),t(Y)).
disjunctive(t(X > Y),f(X),t(Y)).

unary(t(~X),f(X)).
unary(f(~X),t(X)).
```


A simple driver

This driver tests formulas for validity:

```
tprove(F):-
   (
    closedTableau([[f(F)]]), !,
    write('Theorem.'), nl
   ;
    write('Not a theorem.'), nl
   ).
```


Resolution

- A machine oriented method. Unlike the tableaux method, resolution is based on repeated application of a single rule only, the resolution rule.
- Moreover, resolution typically does not work directly on the input formula. Rather, the input formula is first pre-processed into an equivalent formula in conjunctive normal form (CNF). The resolution rule then gets to work on the CNF formula.
- In one respect the resolution mehtod is similar to the tableaux method. It too works by refutation. That is, to prove ϕ, we input $\neg\phi$ and show that this leads to problems.

Conjunction Normal Form (CNF)

- A (propositional) literal is either a sentence symbol (e.g., p, q, r,...), or the negation of a sentence symbol (e.g., $\neg p$, $\neg q$, $\neg r$,...)
- A clause is a disjunction of literals. For example $(p \vee q \vee \neg r \vee s \vee \neg t)$ is a clause. Note that to make a clause true we simply have to make one of the literals it contains true. There is a special clause, the empty clause, sometimes written [ ] which can never be made true.
- A formula is CNF if it is a conjunction of clauses. For example, $(p \vee q) \wedge (r \vee \neg p \vee s) \wedge (q \vee \neg s)$ is in CNF. Note that for a CNF formula to be true, all the clauses it contains must be true.
- CNF formulas have a very simple structure, and we often take advantage of this by writing them in list notation. For example, the previous example could be written as $[[p, q], [r, \neg p, s], [q, \neg s]]$.

Transforming formulas into CNF

- Fact: Every propositional formula is equivalent to a formula in CNF.
- To transform a formula into CNF, we first drive negations down to the sentence symbols using the following rules, and simultaneously eliminate all occurrences of $\rightarrow$
  - Rewrite $\neg(\phi \wedge \psi)$ as $\neg\phi \vee \neg\psi$
  - Rewrite $\neg(\phi \vee \psi)$ as $\neg\phi \wedge \neg\psi$
  - Rewrite $\neg(\phi \rightarrow \psi)$ as $\phi \wedge \neg\psi$
  - Rewrite $(\phi \rightarrow \psi)$ as $\neg\phi \vee \psi$
  - Rewrite $\neg\neg\phi$ as ϕ

Transforming formulas into CNF (continued)

- Once the negations are against the sentence letters, and all occurrences of $\rightarrow$ have vanished, we carry on rewriting using the following rules:
  - Rewrite $\theta \vee (\phi \wedge \psi)$ as $(\theta \vee \phi) \wedge (\theta \vee \psi)$
  - Rewrite $(\phi \wedge \psi) \vee \theta$ as $(\phi \vee \theta) \wedge (\psi \vee \theta)$
  - Rewrite $(\phi \wedge \psi) \wedge \theta$ as $\theta \wedge (\phi \wedge \psi)$
  - Rewrite $(\phi \vee \psi) \vee \theta$ as $\theta \vee (\phi \vee \psi)$
- Note the effect of the first two rewrite rules: they drive occurrences of $\vee$ deeper into the formula, and 'lift' occurrences of $\wedge$. That's what we want for CNF.
- That's all!

Example: Putting $(\neg p \rightarrow q) \rightarrow (\neg r \rightarrow s)$ in CNF

1. $(\neg p \rightarrow q) \rightarrow (\neg r \rightarrow s)$
2. $\neg(\neg p \rightarrow q) \vee (\neg r \rightarrow s)$
3. $(\neg p \wedge \neg q) \vee (\neg\neg r \vee s)$
4. $(\neg p \wedge \neg q) \vee (r \vee s)$
5. $(\neg p \vee r \vee s) \wedge (\neg q \vee r \vee s)$
6. $[[\neg p, r, s], [\neg q, r, s]]$

The Resolution Rule

From:

$[p_1, \ldots, p_n, R, p_{n+1}, \ldots, p_m]$ and $[q_1, \ldots, q_j, \neg R, q_{j+1}, \ldots, p_k]$

deduce

$$[p_1, \ldots, p_n, p_{n+1}, \ldots, p_m, q_1, \ldots, q_j, q_{j+1}, \ldots, p_k].$$

- For example, from $[p, s, \neg t, r]$ and $[q, v, \neg s, r]$ we can deduce $[p, \neg t, r, q, v, r]$.

Using resolution

- We negate the formula we are trying to prove, and put it in CNF.
- We then just keep applying the resolution rule to the clauses in the CNF formula until we can't apply them anymore.
- If we produce an empty clause during this process, this shows that $\neg\phi$ is not consistent (why?), and hence that ϕ is valid.

A simple example: Proving $p \vee \neg p$

- We first negate $p \vee \neg p$, obtaining $\neg(p \vee \neg p)$
- We then put $\neg(p \vee \neg p)$ in CNF, obtaining $[[\neg p],[p]]$.
- One application of the resolution rule immediately yields $[[]]$.
- Hence $\neg(p \vee \neg p)$ is not consistent.
- Hence $p \vee \neg p$ is valid.

Today Lecture 6: First-Order Inference

- Today we discuss first-order inference (for languages without equality).
- We first show how to extend our propositional tableaux system to first-order languages. (We say a little about resolution also, but don't give details.)
- Much of our discussion revolves around the crucial concept of unification.
- We will learn that implementing first-order inference is a subtle and difficult task, and that it is probably advisable to use off-the-shelf theorem provers and model builders rather than trying to build tools ourselves from scratch.
- Demos:
  - `freeVarTabl.pl`
  - `callInference.pl` (featuring MACE and OTTER)

The Key Lesson

- It's easy to extend out propositional tableaux system to a (sound and complete) first-order tableaux system.
- It's a lot harder to do so in a way that lends itself to computational implementation. In fact, we will end up by defining a composite tableaux + unification inference engine.

First-order tableaux for humans

The basic idea: add tableaux rules that eliminate quantifiers, thus turning first-order formulas in propositional ones. Here's the first two universal rules we need:

$$\frac{T\forall x\phi}{T\phi(\tau)} \qquad \frac{F\exists x\phi}{F\phi(\tau)}$$

Here $\phi(t)$ denotes the result of replacing the variable bound by the quantifier by some closed term t.

So, from $T\forall x\textsc{killer}(x)$ we can deduce $T\textsc{killer}(\textsc{jules})$, $T\textsc{killer}(\textsc{butch})$,and so on.

An example

We show the validity of $\forall x(\textsc{die}(x) \rightarrow \textsc{die}(\textsc{mia}) \wedge \textsc{die}(\textsc{zed}))$

1	$F(\forall x\textsc{d}(x) \rightarrow \textsc{d}(\textsc{m}) \wedge \textsc{d}(\textsc{z}))$	
2	$T\forall x\textsc{d}(x)$	$1, F_{\rightarrow}$
3	$F(\textsc{d}(\textsc{m}) \wedge \textsc{d}(\textsc{z}))$	$1, F_{\rightarrow}$
4	$T\textsc{d}(\textsc{m})$	$2, T_{\forall}$
5	$T\textsc{d}(\textsc{z})$	$2, T_{\forall}$
6	$F\textsc{d}(\textsc{m})$ $3, F_{\wedge}$	7 $F\textsc{d}(\textsc{z})$ $3, F_{\wedge}$

Note that we had to apply $T_{\forall}$ twice to line 2.

What sort of tableaux rules are needed to deal with signed formulas of the form $T\exists x\phi$ or $F\forall x\phi$? This is a more subtle matter.

- Suppose a tableau contains the signed formula $T\exists x\textsc{killer}(x)$.
- It is *not* legitimate on the basis of this information to deduce that $T\textsc{killer}(\textsc{jules})$, or that $T\textsc{killer}(\textsc{butch})$, or indeed that $T\textsc{killer}(\textsc{closed-term})$ for *any* closed term of the language we are working with.
- We will introduce a brand new name (we call such names *parameters*) and eliminate the quantifier by substituting this instead. (So we'll now be working in a richer language.)

The Existential Rules

Here are $F_{\forall}$ and $T_{\exists}$:

$$\frac{F\forall x\phi}{F\phi(c)} \qquad \frac{T\exists x\phi}{T\phi(c)}$$

Here $\phi(c)$ denotes the result of substituting a parameter c that we haven't used so far in the tableau proof, for the newly freed variable in the matrix.

- It is crucial that we use new parameters
- Note: the universal rules must be able to use parameters too! But this means the universal rules offer us infinitely many choices!

An example

We show that $\exists x \forall y \text{SHOOTS}(x,y) \rightarrow \forall y \exists x \text{SHOOTS}(x,y)$ is valid. (This pretty example puts all four quantifier rules to work.)

1	$F(\exists x \forall y \text{SHOOTS}(x,y) \rightarrow \forall y \exists x \text{SHOOTS}(x,y))$	
2	$T\exists x \forall y \text{SHOOTS}(x,y)$	1, $F_{\rightarrow}$
3	$F\forall y \exists x \text{SHOOTS}(x,y)$	1, $F_{\rightarrow}$
4	$T\forall y \text{SHOOTS}(c_1, y)$	2, $T_{\exists}$
5	$F\exists x \text{SHOOTS}(x, c_2)$	3, $F_{\forall}$
6	$T\text{SHOOTS}(c_1, c_2)$	4, $T_{\forall}$
7	$F\text{SHOOTS}(c_1, c_2)$	5, $F_{\exists}$

Note the interaction of the existential and universal rules.

An efficiency problem

How efficient is this tableaux system? This we must answer in two ways.

- The first-order tableaux system does not give rise to an algorithm for determining which first-order formulas are valid. There is no such algorithm. So we can't speak of efficiency in an absolute sense.
- So how good is the tableaux system as a practical basis for automated first-order theorem proving?

It's terrible!

The Problem

The universal rules offer us an infinite menu of substitutable terms. We can make intelligent choices from it, but machines can't.

The Solution

Change the universal rules so that we *always* substitute free variables. We are not going to make a real choice of substitutions at all—we are going to use free variables as 'dummies' that will enable us to delay making this decision.

We will gradually build up a whole system of constraints and use the famous unification algorithm to look for solutions to the constraints that lead to branch closure.

Unification

- Unification is the process of carrying out substitutions on two terms so that they become identical.
- PROLOG makes use of a version of unification — but not the one we need.
- We shall discuss unification, see how it differs from the PROLOG version, and then modify the in-built PROLOG unification algorithm accordingly.

Substitutions

- A substitution is a function that maps variables to the terms. We write $x\sigma$ to denote the value of x under the substitution σ.
- We are most interested in finite substitutions. These are substitutions which only assign new terms to a finite number of variables; the rest they map to themselves.
- The simplest finite substitution is the identity substitution, written $\{\}$. This maps every variable to itself.
- There is a special notation for other finite substitutions: $\{x_1/\tau_1, \ldots, x_n/\tau_n\}$.

Substitutions on terms

Let σ be a substitution and τ a term. Then:

- If τ is a variable x, then $\tau\sigma = x\sigma$;
- If τ is a constant, then $\tau\sigma = \tau$;
- If τ has the form $f(\tau_1, \ldots, \tau_n)$, then $[f(\tau_1, \ldots, \tau_n)]\sigma = f(\tau_1\sigma, \ldots, \tau_n\sigma)$.

Substitutions on formulas

First α-convert to make sure no variable occurs both free and bound in the formula we are working with. Then:

- If $R(\tau_1, \ldots, \tau_n)$ is an atomic formula, then $[R(\tau_1, \ldots, \tau_n)]\sigma$ is $R(\tau_1\sigma, \ldots, \tau_n\sigma)$;
- $[\neg\phi]\sigma$ is $\neg[\phi\sigma]$;
- $[\phi \wedge \psi]\sigma$, $[\phi \vee \psi]\sigma$, and $[\phi \rightarrow \psi]\sigma$, are $[\phi\sigma] \wedge [\psi\sigma]$, $[\phi\sigma] \vee [\psi\sigma]$, and $[\phi\sigma] \rightarrow [\psi\sigma]$ respectively;
- $[\forall x\phi]\sigma$ is $\forall x[\phi\sigma_x]$, and $[\exists x\phi]\sigma$ is $\exists x[\phi\sigma_x]$. (Here σ_x is the substitution that is exactly like σ except that it maps the variable x to itself.)

Substitutions on signed tableaux

If σ is a substitution and $\mathcal{T}$ is a signed tableau, then $\mathcal{T}\sigma$ is the signed tableau obtained by replacing every signed formula of the form $T\phi$ in $\mathcal{T}$ by $T[\phi\sigma]$, and every signed formula of the form $F\phi$ in T by $F[\phi\sigma]$.

Composing Substitutions

- Because substitutions are functions we can compose them in the usual way.
- That is, if σ_1 and σ_2 are substitutions, then $\sigma_1\sigma_2$, the composition of σ_1 and σ_2 is defined as follows:
  for every variable x, $\mathrm{x}(\sigma_1\sigma_2)$ is $(\mathrm{x}\sigma_1)\sigma_2$.
- That is, $\sigma_1\sigma_2$ first carries out the substitution σ_1 and then carries out the substitution σ_2.

An important example

- Suppose we want to make the terms f(c,y,w) and f(x,y,g(z)) identical (where c is a constant, w, x, y, and z are variables, f is a 3-place function symbol, and g is a 1-place function symbol).
- Let σ_1 be the substitution $\{\mathrm{x/c, w/g(z)}\}$. Applying σ_1 to these terms has the desired result, for f(c,y,w))σ_1 = f(x,y,g(z))σ_1 = f(c,y,g(z)).
- But other substitutions also work, for example σ_2, the finite substitution $\{\mathrm{x/c, w/g(z), y/h(u,x)}\}$.

Why is σ_1 a better choice than σ_2?

- A substitution σ_1 is more general than a substitution σ_2 if and only if there is some substitution θ such that $\sigma_2 = \sigma_1\theta$.
- That is, σ_1 is more general than σ_2 if we can get the effect of σ_2 by first carrying out σ_1 and then making a further substitution θ.
- Thus $\{\mathrm{x/c, w/g(z)}\}$ is more general than $\{\mathrm{x/c, w/g(z), y/h(u,x)}\}$ because there is a substitution θ (namely $\{\mathrm{y/h(u,x)}\}$) such that $\{\mathrm{x/c, w/g(z), y/h(u,x)}\} = \{\mathrm{x/c, w/g(z)}\}\theta$.
- So the answer to the question on the previous slide is: $\{\mathrm{x/c, w/g(z)}\}$ is a better way to make the two terms equal than $\{\mathrm{x/c, w/g(z), y/h(u,x)}\}$, because it does less work, is less committal.

The key definition

Let τ_1 and τ_2 be terms. A substitution is a unifier for τ_1 and τ_2 if and only if $\tau_1\sigma = \tau_2\sigma$. Terms τ_1 and τ_2 are said to be unifiable if and only they have a unifier. A substitution σ is a most general unifier (or mgu) for two terms if and only if it is unifier for these terms, and is more general than any other unifier.

This definition immediately leads to a question: how do we compute unifiers?

Terms as trees

The term f(h(x),g(y,x,w)) has the following parse tree:

```
      f
     / \
    h   g
    |  /|\
    x y x w
```

Thinking of terms as trees is a valuable way of thinking about the differences between terms. For example, why are the terms f(h(x),g(y,x,w)) and f(h(x),g(y,x,k(u,v))) different ...?

Disagreement pairs

```
      f                 f
     / \               / \
    h   g             h   g
    |  /|\            |  /|\
    x y x w           x y x k
                           / \
                          u   v
```

Disagreement Pair

More precisely: a disagreement pair for terms t_1 and t_2 is subterm s_1 (of t_1) and s_2 (of t_2) such that: s_1 and s_2 have distinct labels at their root, but the paths in t_1 and t_2 that lead down to the two roots is the same. (So difference pairs are the highest places in the tree where there is disagreement.)

Computing unifiers

- Disagreement pairs make terms different, thus unification algorithms must eliminate them. How can this be done?
- Suppose terms τ_1 and τ_2 are different because there a disagreement pair (d_1, d_2) such that neither d_1 nor d_2 is a variable. Then there is nothing we can do; τ_1 and τ_2 are not unifiable.

The occurs check

Now for the tricky question. Suppose that d_1, say, is a variable. Are the two terms unifiable?

'Yes', *provided that the variable d_1 does not occur in d_2.*

Suppose d_1 is a variable, say x, and that x doesn't occur in d_2. Then we can eliminate this disagreement pair by replacing x by d_2.

On the other hand, if d_1 is a variable (say x) and x does occur in d_2, then unification is impossible.

A disagreement pair (d_1, d_2) is called simple if and only if at least one of the terms d_1 or d_2 is a variable that does not occur in the other. Simple disagreement pairs are the ones we can repair.

We do so by carrying out the relevant repair

- If d_1 is a variable not occurring in d_2, the relevant repair is the substitution $\{d_1/d_2\}$.
- If d_2 is a variable not occurring in d_1, the relevant repair is the substitution $\{d_2/d_1\}$.
- If both d_1 and d_2 are variables, we'll stipulate that the relevant repair is $\{d_1/d_2\}$.

input terms τ_1 and τ_2
let $\sigma := \{\}$
while $\tau_1\sigma \neq \tau_2\sigma$
  choose a disagreement pair (d_1, d_2) for $\tau_1\sigma$, $\tau_2\sigma$
  if (d_1, d_2) is not simple **then**
    write τ_1 and τ_2 are not unifiable **and** HALT
  else
    let $\sigma := \sigma \cup \rho$, where ρ is the relevant repair
  endif
endwhile

Remarks on the algorithm

- This is a genuine computational solution of the unification problem.
- No matter which two input terms it given, it halts after finitely many steps.
- When it halts, it will either have told us that the terms are not unifiable (and if it says this, it's right!) or it will have found out how to build the mgu of the two terms.
- Moreover, the mgus it produces are idempotent. That is, if σ is an mgu produced by this algorithm, then $\sigma\sigma = \sigma$.

Why is such an abstract looking property interesting?

Simultaneous unification (I)

- We will be using unification to try and close tableaux branches.
- We will look for branches containing pairs of atomic formula $T(R(\tau_1, \ldots, \tau_n))$ and $F(R(\tau_1, \ldots, \tau_n))$.
- If we can find a substitution σ that makes τ_1 identical to τ'_1, and …, τ_n identical to τ'_n, then by applying σ we obtain a branch containing contradictory formula, that is, a closed branch.
- Thus we need to find a single substitution that *simultaneously* makes n pairs of terms identical.

This looks harder than ordinary unification, but it's not.

Simultaneous unification (II)

To simultaneously unify $\tau_1, \ldots \tau_n$ and $\tau'_1, \ldots \tau'_n$:

- Find an *idempotent* mgu σ_1 for τ_1 and τ'_1.
- Find an *idempotent* mgu σ_2 for $\tau_1\sigma_1$ and $\tau'_1\sigma_1$.
- Find an *idempotent* mgu σ_3 for $\tau_1\sigma_1\sigma_2$ and $\tau'_1\sigma_1\sigma_2$.
- In short, keep 'chaining together' the solutions to each individual pairs.

The substitution $\sigma = \sigma_1\sigma_2 \ldots \sigma_{n-1}\sigma_n$ obtained in this way is a simultaneous mgu for the n pairs of terms. For this method to work, the substitutions constructed at each step must be idempotent.

PROLOG unification

PROLOG's in-built unification algorithm is reasonably close to the above algorithm. There are two main differences.

- The algorithm given above is non-deterministic, whereas PROLOG's in-built mechanism for unifying terms is deterministic.
- In the interests of efficiency, PROLOG does not bother making the 'occurs check'.

Given a disagreement pair (d_1, d_2), one of which is a variable, PROLOG does not check whether this variable occurs in the other term, but will attempt to carry out what it thinks the required repair is. This behavior is unacceptable for our purposes.

Building in an occurs check

```
unify(X,Y):-
   var(X), var(Y), X=Y.

unify(X,Y):-
   var(X), nonvar(Y), not_occurs_in(X,Y), X=Y.

unify(X,Y):-
   var(Y), nonvar(X), not_occurs_in(Y,X), X=Y.

unify(X,Y):-
   nonvar(X), nonvar(Y), atomic(X), atomic(Y), X=Y.

unify(X,Y):-
   nonvar(X), nonvar(Y), compound(X),
   compound(Y), term_unify(X,Y).
```


The real work

We define `not_occurs_in/2` with the help of the in-built PROLOG metalogical predicate `\==`. (Recall that query `X \== Y` succeeds if and only if `X` and `Y` are not identical.)

```
not_occurs_in(X,Term):-
   var(Term), X \== Term.

not_occurs_in(_,Term):-
   nonvar(Term),atomic(Term).

not_occurs_in(X,Term):-
   nonvar(Term),compound(Term),
   functor(Term,_,Arity),
   not_occurs_in_complex_term(Arity,X,Term).
```

The not_occurs_in_complex_term/3 predicate recursively checks whether the variable X does not occur in the arguments of the complex term Term, making use of the not_occurs_in/2 predicate.

```
not_occurs_in_complex_term(N,X,Term):-
   N > 0, arg(N,Term,Arg),
   not_occurs_in(X,Arg),
   M is N - 1,
   not_occurs_in_complex_term(M,X,Term).

not_occurs_in_complex_term(0,_,_).
```


Remaining details

```
term_unify(X,Y):-
   functor(X,Functor,Arity),
   functor(Y,Functor,Arity),
   unify_args(Arity,X,Y).

unify_args(N,X,Y):-
   N > 0,
   M is N - 1,
   arg(N,X,ArgX),
   arg(N,Y,ArgY),
   unify(ArgX,ArgY),
   unify_args(M,X,Y).
unify_args(0,_,_).
```


Some versions of PROLOG already support compound/1. If yours doesn't, it's easy to define it.

```
compound(Term):-
   functor(Term,_,Arity), Arity > 0.
```


First-order tableaux for machines

- Let's use unification to define a more realistic tableaux proof system.
- We plan to substitute free variables in the universal rules, thus gradually building up a system of constraints. (Hence the new method is called **Free variable tableaux**)
- We hope unification will solve these constraints, thus bypassing the need for human insight.

The new universal rules

$$\frac{T\forall \mathrm{x}\phi}{T\phi(\mathrm{v})} \qquad \frac{F\exists \mathrm{x}\phi}{F\phi(\mathrm{v})}$$

Here $\phi(\mathrm{v})$ denotes the result of replacing all instances of the variable that the quantifier bound by a new variable v that does not occur bound anywhere in the tableau.

A problem

- But now we have a problem with the existential rules!
- Existential rules introduce brand-new names called parameters — but the use of unification threatens to identify them, destroying the newness idea!
- A simple idea overcomes this problem. We will substitute *structured* terms when we use the existential rules. The term structure will ensure that unification cannot spoil anything.
- More precisely, we will use Skolem terms.

The new existential rules

$$\frac{F\forall \mathrm{x}\phi}{F\phi(s(\mathrm{x}_1,\ldots,\mathrm{x}_n))} \qquad \frac{T\exists \mathrm{x}\phi}{T\phi(s(\mathrm{x}_1,\ldots,\mathrm{x}_n))}$$

Here s is a new Skolem function symbol, and x_1, ..., x_n are all the free variables in the tableau. (If there are no free variables, s is a new Skolem constant.)

Why it works

With Skolem function, we can do something we couldn't do with parameters: we can build in newness in a way that will survive the unification operation!

The existential rules demands we substitute: $s(\mathrm{x}_1,\ldots,\mathrm{x}_n)$, where x_1, ..., x_n are all the free variables in the tableaux. Now recall our discussion of the occurs check. Quite simply, $s(\mathrm{x}_1,\ldots,\mathrm{x}_n)$ *cannot* unify with any of x_1, ..., x_n!

Our new 'Skolem structured' term really will be new.

An example

We prove that ∃x∀ySHOOTS(x,y) → ∀y∃xSHOOTS(x,y) using our new quantifier rules and unification. Here are the first 7 steps:

1	$F(\exists x \forall y \text{SHOOTS}(x,y) \rightarrow \forall y \exists x \text{SHOOTS}(x,y))$	
2	$T\exists x \forall y \text{SHOOTS}(x,y)$	$1,\ F_{\rightarrow}$
3	$F\forall y \exists x \text{SHOOTS}(x,y)$	$1,\ F_{\rightarrow}$
4	$T\forall y \text{SHOOTS}(s_1, y)$	$2,\ T_{\exists}$
5	$F\exists x \text{SHOOTS}(x, s_2)$	$3,\ F_{\forall}$
6	$T\text{SHOOTS}(s_1, v_1)$	$4,\ T_{\forall}$
7	$F\text{SHOOTS}(v_2, s_2)$	$5,\ F_{\exists}$

The example continued

Now apply the substitution $\{v_1/s_2, v_2/s_1\}$:

1	$F(\exists x \forall y \text{SHOOTS}(x,y) \rightarrow \forall y \exists x \text{SHOOTS}(x,y))$	
2	$T\exists x \forall y \text{SHOOTS}(x,y)$	$1,\ F_{\rightarrow}$
3	$F\forall y \exists x \text{SHOOTS}(x,y)$	$1,\ F_{\rightarrow}$
4	$T\forall y \text{SHOOTS}(s_1, y)$	$2,\ T_{\exists}$
5	$F\exists x \text{SHOOTS}(x, s_2)$	$3,\ F_{\forall}$
6	$T\text{SHOOTS}(s_1, s_2)$	$4,\ T_{\forall}$
7	$F\text{SHOOTS}(s_1, s_2)$	$5,\ F_{\exists}$

This tableau *is* closed.

MGU Closure Rule

Suppose $\mathcal{T}$ is a tableau formed from some initial tableau $\mathcal{I}$, and that some branch of $\mathcal{T}$ contains a pair of signed formulas of the form $T(R(\tau_1, \ldots, \tau_n))$ and $F(R(\tau_1, \ldots, \tau_n))$.

Then $\mathcal{T}\sigma$ is also a tableau formed from the initial tableau $\mathcal{I}$, where σ is a simultaneous mgu of τ_1 and τ_1', ..., and τ_n and τ_n'.

This rule not an extension rule, it's a *transformation* rule.

Some comments

- This free variable tableaux is not as nice as our previous system if one wants to prove things by hand. The need to compute unifiers and transform an entire tableaux destroy the nice step-by-step character nature of tableaux proof.
- The 'global transformation' way that free variable tableaux work has another undesirable effect — it makes them hard to use as model builders. This possibility was one of the more interesting points about the tableaux methods, and now we've pretty much lost it.
- That said, recent work replaces use of the free variables by building up a set of constraints. This approach is elegant, and it seems that this it may allow tableaux model building to be retained

Some further comments

- So there are drawbacks. Nonetheless, by making use of other ideas (such as converting the input formula into normal form) and various theoretical and computational optimizations, tableaux methods have shown to give rise to reasonably efficient first-order theorem provers.
- Tableaux methods have also been extraordinarily successful with other logics, notably modal and description logics.
- By and large, however, so far they don't seem to handle equality particularly well. (This may change very soon.)

First-order resolution: basic ideas

- As in the propositional case, we transform the input formula into a normal form — basically CNF in which all the existential quantifiers have been replaced by Skolem functions.
- In more detail, we proceed as follows:
  - Given an input formula ϕ, pull the quantifiers to the front. That is, rewrite it in prenex form. This is the form $Q_1 \cdots Q_n\psi$, where each Q_i is $\exists$ or $\forall$, and ψ is quantifier free.
  - Put ψ (which is quantifier free) into CNF. This step is just like what we did last lecture.
  - Get rid of existential quantifiers by using Skolem functions.

Example: $\forall x \exists y \exists z((\neg P(x,y) \wedge Q(x,z)) \vee R(x,y,z))$

1. $\forall x \exists y \exists z((\neg P(x,y) \wedge Q(x,z)) \vee R(x,y,z))$ (already in Prenex form).
2. $\forall x \exists y \exists z((\neg P(x,y) \vee R(x,y,z)) \wedge (Q(x,z) \vee R(x,y,z))$ (matrix now in CNF).
3. $\forall x((\neg P(x,f(x)) \vee R(x,f(x),g(x))) \wedge (Q(x,g(x)) \vee R(x,f(x),g(x)))$ (existential quantifiers eliminated with help of Skolem functions).
4. $(\neg P(x,f(x)) \vee R(x,f(x),g(x))) \wedge (Q(x,g(x)) \vee R(x,f(x),g(x))$ (drop universal quantifiers — simply take all variables as universally quantified).
5. $[[\neg P(x,f(x)), R(x,f(x),g(x))], [(Q(x,g(x)), R(x,f(x),g(x))]]$ (put in list notation).

First order resolution

The basic idea is that unification and resolution are used together. We also need to take care to relabel variables when appropriate. Here's an example:

$[P(x), Q(x)]$ and $[\neg P(a), R(x)]$

If we unified a with x we could apply the propositional resolution rule. Note, however, that this unification would affect the x in $R(X)$, which we don't want. So we first relabel this variable obtaining

$[P(x), Q(x)]$ and $[\neg P(a), R(y)]$

We now unify a with x, obtaining:

$[P(a), Q(a)]$ and $[\neg P(a), R(y)]$

Resolution yields:

$[Q(a), R(y)]$

Some remarks

- First-order resolution has been an extraordinarily successful method. Historically, it was the technique that opened the door to realistic theorem proving, and recent years have seen quite astounding improvements in performance.
- The reason for it success boils down to its simplicity. This makes it worthwhile to think hard about good strategies for choosing where to carry out resolution steps next.
- Moreover, recent work has begun to deliver realistic theorem provers for handling equality (a notoriously difficult problem).

Some further remarks

- On the other hand, resolution doesn't easily help us with model building.
- It also doesn't adapt straightforwardly to other logics, such as modal and description logics.
- Moreover, lot's of the structure of the original problem is lost in this method — and this may not always be a good thing.
- Nonetheless, resolution is clearly a method that's here to stay.

Theorem Proving

Fact: First-order logic is not decidable!

Now—Suppose we give a first-order problem ϕ to a theorem prover.

- This means that there is a chance that, in theory, our theorem prover never comes back with an answer...
- Practical consequence: if no proof for ϕ is found, this does—strictly speaking—**not** mean that ϕ is not a theorem.
  (In practice, you will have to work with time-outs.)
- On the other hand, if your theorem prover finds a proof, you can be pretty sure ϕ is a theorem (is valid).

Model Building

Theorem provers check whether a formula (or a set of formulas) is valid (true in all possible models)

Model Builders attempt to construct a model for a formula (or a set of formulas) and thereby show that this formula is satisfiable
(true in at least one model)

There are some 'unpleasant' restrictions for model building:

- You need to specify a domain size. This means that if your model builder cannot find a model with domain size n for input ϕ, it does not mean that ϕ is not satisfiable. Perhaps there is a model with domain size $n+1$! On the other hand, if it finds a model, you can be pretty sure that ϕ is satisfiable.
- Restricted to finite models. (Everybody has a mother)

Theorem Proving vs. Model Building (I)

Consistency

Suppose you want to check whether ϕ is consistent:

- Give $\neg\phi$ to a theorem prover.
  If it succeeds in finding a proof, ϕ is not consistent
- Give ϕ to a model builder.
  If it finds a model, ϕ is consistent.

Theorem Proving vs. Model Building (II)

Informativity

Suppose you want to check whether ϕ is informative with respect to ψ:

- Give $\psi \rightarrow \phi$ to a theorem prover.
  If it succeeds in finding a proof, ϕ is not informative with respect to ψ.
- Give $\psi \wedge \phi$ and $\psi \wedge \neg\phi$ to a model builder.
  If it finds a model in both cases, ϕ is informative with respect to ψ.

Consistency checking for theorem provers and model builders

$\Phi \setminus \neg\Phi$	valid	?	invalid
satisfiable	–	consistent	consistent
?	inconsistent	?	consistent
not satisfiable	inconsistent	inconsistent	–

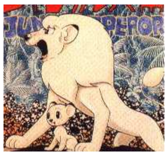
Off-the-shelf Theorem Provers

Provers that accept "first-order formula" syntax
(as opposed to special CNF notation):

- OTTER `http://www-unix.mcs.anl.gov/AR/otter/`
- BLIKSEM `http://www.mpi-sb.mpg.de/~bliksem/`
- FDPLL `http://www.uni-koblenz.de/~peter/FDPLL/`
- SPASS `http://spass.mpi-sb.mpg.de/`

There are many more:
GANDALF, LEANTAP, E-SETHEO, SCOTT, MUSCADET, VAMPIRE, ...

Off-the-shelf Model Builders



Accepting full first-order formula syntax
(as opposed to special CNF notation):

- MACE `http://www-unix.mcs.anl.gov/AR/mace/`
- KIMBA `http://www.ags.uni-sb.de/~konrad/kimba.html`

There are some more: FINDER, SATO, SEM, ...

Today: Lecture 7: Putting It All Together

We now have some answers to the two fundamental questions with which this course began.

- We know how to build first-order representations for natural language expressions with the help of the lambda calculus, and have a number of tools for handling scope ambiguities.
- We have learned a lot about the process of performing inference with first-order representations. In particular, we have learned something about theorem provers, model builders and (from Lecture 1) model checking.
- Now it's time to have a little fun, bring the various bits and pieces together, and see what they can do for us ...
- Demos: various versions of Curt

Chatting with CURT

- We will plug together our lambda calculus, quantifier storage, model checking, theorem proving and model building programs together in a system that allows simple interaction with the user.
- This system is called CURT.
- The user can extend CURT's knowledge by entering English sentences and then query CURT about its acquired knowledge.
- We will present different versions of CURT, that represent the different stages of development in CURT's life.

Baby Curt

- This is the backbone of the CURT system, with a minimum of inference tools included (namely none).
- The user can enter English sentence and check the knowledge state of CURT
- The dialogue control structure is implemented by `curtTalk/2`
- The predicate `curtUpdate/4` updates the previous knowledge base with newly entered English sentences (it also takes care of meta-talk)

Prolog source for Baby Curt: `www.comsem.org/source/babyCurt.pl`

But Baby Curt is having some problems!

- As Baby Curt doesn't impose on any inference tools, it accepts discourses like:

  USER: "Mia smokes and does not smoke."
  CURT: "OK"

  USER: "Vincent is a man."
  CURT: "OK"
  USER: "Mia likes every man."
  CURT: "OK"
  USER: "Mia does not like Vincent."
  CURT: "OK"

- So we need to do something here to make CURT deal with inconsistent information.

Clever Curt

- Clever CURT only accepts consistent English sentences.
- The user is also able to view the models generated for the English sentences given to CURT.
- Ambiguous input might eliminate some of the inconsistent interpretations
- Clever Curt uses the model builder **MACE** to check consistency, and the theorem prover **OTTER** to check inconsistency of interpretations.

Prolog source for Clever Curt: `www.comsem.org/source/cleverCurt.pl`

But Clever Curt has its problems too!

- Although Clever Curt rejects inconsistent English sentences, it has no clue how to distinguish old from new information:

  USER: "Mia smokes."
  CURT: "OK"
  USER: "Mia smokes."
  CURT: "OK"
  USER: "Mia smokes."
  CURT: "OK"
  USER: "Vincent knows every boxer."
  CURT: "OK"
  USER: "Butch is a boxer."
  CURT: "OK"
  USER: "Vincent knows Butch."
  CURT: "OK"

- So we extend CURT to deal with uninformative information.

Sensitive Curt

- Sensitive Curt is clever! (it knows what consistency means)
- Sensitive Curt will also note uninformative contributions of the user.
- It does so by using **MACE** and **OTTER** to check whether new contributions are informative with respect to to the readings CURT has in its memory.

Prolog source for Sensitive Curt:
`www.comsem.org/source/sensitiveCurt.pl`

Sensitive (and Clever) Curt however are quite sloppy with respect to memory management!

- Quantifying noun phrases might induce different interpretations (Storage)
- But some of these readings can be logically equivalent, but neither clever not sensitive CURT notice:

  USER: A boxer kills a man
  CURT: OK.
  USER: readings

  ```
  1 exists(A,boxer(A)&exists(B,man(B)&kill(A,B)))
  2 exists(A,man(A)&exists(B,boxer(B)&kill(B,A)))
  ```
- Using theorem proving techniques once again, we make CURT filter out logical equivalent interpretations

Laconic Curt

- Laconic Curt eliminates equivalent readings using the Theorem Prover **OTTER**.
- Given R1 and R2 from a set of readings, R1 and R2 will be replaced by R1 if **OTTER** finds a proof for

  ((R1 → R2) & (R2 → R1))
- Note that this is not necessarily efficient.

Prolog source for Laconic Curt: `www.comsem.org/source/laconicCurt.pl`

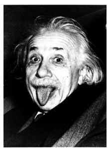
Still, Curt is quite dumb!

- The CURTs we developed so far possess no world knowledge at all!
- For instance, Sensitive Curt does not know that Mia is woman, nor does it know that men and women denote different kinds of entities.

  USER: Mia is a woman.
  CURT: OK.
  USER: Mia is a man.
  CURT: OK.
- So we should supply additional information about the logical symbols we use. But how?

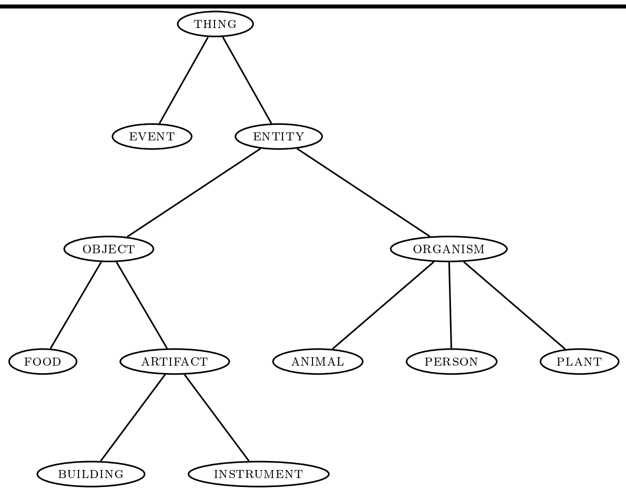
Knowledgeable Curt

- Adding Lexical Knowledge: Let's focus on nouns and adjectives
- Hypernyms (example: vehicle is a hypernym of car)
  In first-order logic: $\forall x(\textsc{car}(x) \rightarrow \textsc{vehicle}(x))$
- Antonyms (example: males are disjoint from females)
  In first-order logic: $\forall x(\textsc{male}(x) \rightarrow \neg\textsc{female}(x))$



- Put this information in the lexicon (what would be a better place?)
- the required background knowledge is computed from the lexicon on the basis of CURT's input (so not *all* available knowledge is used all of the time)

Source for Knowledgeable Curt: `www.comsem.org/source/knowledgeableCurt.pl`



Now Curt is knowledgeable, but not able to share its knowledge...

- We not only want to assert information to CURT's database.
- We also want to query information!
- For instance, by posing "Who likes Mia" CURT should return all named entities that like Mia (according to its current model).
- We will use our Model Checker for first-order models to accomplish this task.

Super Curt

- Questions introduce lambda-expressions
- Model Checkers can be used to extract information from the model
- For instance, "Who walks" gives us `lambda(X,walk(X))`
- Then we try to evaluate this expression (using `modelChecker2.pl`) in the current model wrt assignment `g(X,Answer)`
- If we are successful `Answer` will unify with an element of the domain
- Having a member of the domain at our disposal, we realize CURT's answer as a:
  - proper name if it is named in the model
  - indefinite noun phrase if there is a one-place noun relation

Source for Super Curt: `www.comsem.org/source/superCurt.pl`

Improving Super Curt

- Super Curt combines several inference tools (Theorem Prover, Model Checker, Model Builder) for making responses
- The current architecture implements a purely sequential approach
- Especially for consistency/informativity checking, we might want to run theorem provers and model builders in parallel:

  While the theorem prover tries to prove $\neg\phi$, the model builder attempts to construct a model for ϕ
- One might also consider spreading the computational power by running computationally demanding tasks on different machines
- There are various directions one might go: LINDA (Blackboard Architecture), MathWeb (theorem proving agents), OAA (Open Agent Architecture), CORBA, ...

Today Lecture 8: DRT

- We introduce the basic ideas Discourse Representation Theory (DRT), a semantic framework which uses a novel logical languages (the language of Discourse Representation Structures (DRSs)) to represent the meaning of sentences (and indeed, entire discourses).
- We also show that it is possible to translate DRSs into first-order logic. This gives us a semantics for the language of DRSs. It also opens the way to using off the shelf inference tools.
- We show that DRT is compatible with a bottom up, λ-driven, approach to semantic construction, thus making it possible to use the semantic construction tools we have developed so far.
- Demos:
  - `lambdaDRT.pl`

Discourse Representation Theory

We shall

- Give a swift overview of DRT,
- emphasizing constructing a picture of the changing context,
- informally sketch its semantics,
- introduce the concept of accessibility
- describe how to compile DRSs to first-order logic,
- and how to build DRSs with Lambdas.

Aims:

- To make clear that DRT offers an entire architecture for thinking about semantics, an architecture that blends formal, empirical, and computational ideas in a flexible and suggestive way.
- To make clear that in spite of many novel features, pretty much everything in DRT is compatible with our earlier work. In particular, we can still use lambda calculus as a glue language, and first-order inference tools.

An overview of DRT

- DRT employs a language based on box-like structures called DRSs
- We will be making heavy use of DRSs for many different purposes
- The present discussion is largely based around the following intuition:

  DRSs are Pictures.
- We shall present DRT from a *representational* perspective
- There is also a natural *dynamic* perspective which insists that

  DRSs are Programs.

  and we discuss this too.

Representing the Meaning of Discourse

- Discourses will simply be sequences of natural language sentences.
- How can we represent the meaning of a discourse?
- Examples:

  "Mia is a woman. She loves Vincent."

  $\textsc{woman}(\textsc{mia}) \land \textsc{love}(x, \textsc{vincent})$

  "A woman snorts. She collapses."

  $\exists z(\textsc{woman}(z) \land \textsc{snort}(z)) \land \textsc{collapse}(x)$.

Two problems: complex post-processing & counter-intuitive

Context Change Potential

When we utter a "A woman snorts" we don't simply make a claim about the world, we also change the context in which subsequent utterances will be interpreted.

- Start a new discourse with the empty box:

- expand this empty box with information from the entire discourse:

x
$\textsc{woman}(x)$ $\textsc{snort}(x)$

Think of the discourse referents as pegs, and the conditions as coats we hang on these pegs as best we can.

Dealing with Pronouns

Analyzing: "She snorts"

1. adding a new discourse referent (e.g., "y")
2. adding the condition "$\textsc{collapse}(y)$"
3. adding an equation "y=x"

A woman snorts.
She collapses.

x y
$\textsc{woman}(x)$ $\textsc{snort}(x)$ $\textsc{collapse}(y)$ $y=x$

The discourse referent introduced by the pronoun must be identified with an accessible discourse referent.

Another example: Mia snorts. She collapses.

x
$x=\textsc{mia}$ $\textsc{snort}(x)$

1. The first sentence introduces a new discourse referent (namely x) and simultaneously records that this new 'peg' has to be identified with the entity named Mia.
2. So DRT handles proper names and existentially quantified NPs in essentially the same way: both introduce a discourse referent which is then constrained by adding conditions.
3. Moreover, there is an even closer parallelism between the treatment of anaphoric pronouns and proper names: both introduce new discourse referents, and both constrain them equationally (that is, using the equality symbol).

The example continued

Mia snorts. She collapses.

$$\begin{array}{|l|}\hline \mathsf{x\ y} \\ \hline \mathsf{x}{=}\text{MIA} \\ \text{SNORT}(\mathsf{x}) \\ \mathsf{collapse(y)} \\ \mathsf{y{=}x} \\ \hline \end{array}$$

DRT Fundamentals

- building pictures of the changing context
- by introducing discourse referents and stating constraints
- proper names and existentially quantified NPs handled in essentially the same way
- closer parallelism between anaphoric pronouns and proper names

DRS Languages

- How do we deal with universal quantification and negation?
- DRSs can be nested, and combined using various connectives
- these more complex pictures offer us all representational power we would expect

DRS languages share many of the ingredients of first-order languages.

- they contain the symbols $\neg$, $\vee$, $\rightarrow$ (though they normally don't contain $\wedge$), (and like first-order languages with equality, they contain the symbol $=$)
- In addition, DRS languages contain the symbols x, y, z, and so on, though these are called discourse referents, not variables.

Differences:

- DRS languages *don't* contain $\forall$ or $\exists$ (quantification via boxes ($\forall$) or implicit ($\exists$)).

Discourse Representation Structures

1. If $\mathsf{x}_1, \ldots, \mathsf{x}_n$ are discourse referents $(n \geq 0)$ and $\gamma_1, \ldots, \gamma_m$ $(m > 0)$ are conditions then $\begin{array}{|l|}\hline \mathsf{x}_1 \ldots \mathsf{x}_n \\ \hline \gamma_1 \ldots \gamma_m \\ \hline \end{array}$ is a DRS.
2. If R is a relation symbol of arity n, and $\mathsf{x}_1,\ldots,\mathsf{x}_n$ are some discourse referents, then $\mathsf{R}(\mathsf{x}_1,\ldots,\mathsf{x}_n)$ is a condition.
3. If t_1 and t_2 are discourse referent or constants, then $t_1 = t_2$ is a condition
4. If K_1 and K_2 are DRSs, then $\mathsf{K}_1 \Rightarrow \mathsf{K}_2$ is a condition.
5. If K_1 and K_2 are DRSs, then $\mathsf{K}_1 \vee \mathsf{K}_2$ is a condition.
6. If K is a DRS, then $\neg$ K is a condition.
7. Nothing is a condition or a DRS unless it can be shown to be so using clauses 1–6.

Indefinite Noun Phrases

The DRS for "a woman snorts" is:

$$\begin{array}{|l|}\hline \mathsf{x} \\ \hline \text{WOMAN}(\mathsf{x}) \\ \text{SNORT}(\mathsf{x}) \\ \hline \end{array}$$

The box and the discourse referent x are contributed by the NP "a woman".

The VP "snorts" then fills in the box with the condition SNORT(x).

Proper Names

Proper names also introduce discourse referents.
For example, the DRS for "Vincent does not die" is:

$$\begin{array}{|l|}\hline \mathsf{x} \\ \hline \mathsf{x}{=}\text{VINCENT} \\ \neg \begin{array}{|l|}\hline \\ \hline \text{DIE}(\mathsf{x}) \\ \hline \end{array} \\ \hline \end{array}$$

Universal Quantifiers

The DRS for "Every gimp shrieks" is:

$$\begin{array}{|c|}\hline \\ \hline \begin{array}{|l|}\hline \mathsf{x} \\ \hline \text{GIMP}(\mathsf{x}) \\ \hline \end{array} \Rightarrow \begin{array}{|l|}\hline \\ \hline \text{SHRIEK}(\mathsf{x}) \\ \hline \end{array} \\ \hline \end{array}$$

Informal Semntics: Boxes as Pictures

When is a DRSs satisfied in a model?

> If and only if it is an accurate image of the information recorded inside the model.

$$\begin{array}{|l|}\hline \mathsf{x\ y} \\ \hline \text{WOMAN}(\mathsf{x}) \\ \text{BOXER}(\mathsf{y}) \\ \text{ADMIRE}(\mathsf{x,y}) \\ \hline \end{array}$$

Should be satisfied iff it is possible to associate the discourse referents x and y with entities in the model such that:

1. *the first entity is a woman*
2. *the second is a boxer*
3. *the first stands in the admires relation to the second*

Complex Conditions

Negated DRS: satisfied if it is *not* possible to find the picture inside the model

Disjunctive DRSs: satisfied if *at least one* of these pictures can be embedded

But what about implicational conditions?

y	⇒	x
WOMAN(y)		BOXER(x) LOVE(y,x)

No matter which entities we use to embed the antecedent picture, we will be able to embed the consequent picture too.

Accessibility

- is a 'geometrical' concept; it has to do with the way DRSs are stacked one inside the other.
- Discourse referents of DRS K_1 are accessible from DRS K_2 when K_1 *subordinates* K_2, or when K_1 *equals* K_2.

Subordination

Let K_1 and K_2 be DRSs. K_1 subordinates K_2 if and only if

1. K_1 contains a condition of the form $\neg K_2$.
2. K_1 contains a condition of the form $K_2 \Rightarrow K$, where K is some DRS.
3. $K_1 \Rightarrow K_2$ is a condition in some DRS K.
4. K_1 contains a condition of the form $K_2 \vee K$ or $K \vee K_2$, for some DRS K.
5. Some DRS K subordinates K_2, and K_1 subordinates K.

In short: look up, and (with ⇒) look left.

A woman snorts. She collapses.

x y
WOMAN(x) SNORTS(x) COLLAPSE(y) y=x

Here x is accessible to y (they are part of the same DRS)

? Every woman snorts. She collapses.

y

x	⇒	
WOMAN(x)		SNORTS(x)

COLLAPSE(y)

y=?

There is no accessible discourse referent for y

Donkey Sentences

For example: "if a man eats a big kahuna burger, he enjoys it"

Correct:

$\forall x \forall y[\text{MAN}(x) \wedge \text{BIG\_KAHUNA\_BURGER}(y) \wedge \text{EAT}(x,y) \rightarrow \text{ENJOY}(x,y)]$

But (using our translations for the quantifiers):

$\exists x[\text{MAN}(x) \wedge \exists y[\text{BIG\_KAHUNA\_BURGER}(y) \wedge \text{EAT}(x,y)]] \rightarrow \text{ENJOY}(x,y)$

Unfortunately, this 'representation' does not capture the meaning of the sentence.

Donkey Sentences in DRT

In DRT the meaning of the sentence would be represented as follows:

x y	⇒	v w
MAN(x) BIG_KAHUNA_BURGER (y) EAT(x,y)		ENJOY(v,w) v=x w=y

This representation is natural in several ways.

Questions

- Does this representation really capture the meaning of the donkey sentence?
- Can such representations really be built systematically?

Answers

- There is a systematic translation of the DRS language into first-order logic with equality. If we carry out this translation we get:
  $\forall x \forall y[\text{MAN}(x) \wedge \text{BIG\_K\_BURGER}(y) \wedge \text{EAT}(x,y) \rightarrow$
  $\exists v \exists w(\text{ENJOY}(x,y) \wedge v{=}x \wedge w{=}y)]$
- DRS can be systematically built either top down *or* bottom up. We concentrate on the latter.

From Discourse Representations to First-Order Logic

- translate DRSs into formulas of first-order logic with equality
- translation is performed by a translation function t
- we will write $(arg)^t$ to denote the application of function t to argument arg. Here arg will either be a DRS or a condition.

Translating Boxes

DRS with a non-empty universe

$$\left(\begin{array}{|l|}\hline \mathrm{x}_1,\ \ldots,\ \mathrm{x}_n \\ \hline \gamma_1 \\ \cdot \\ \cdot \\ \cdot \\ \gamma_m \\ \hline \end{array}\right)^t = \exists \mathrm{x}_1 \cdots \exists \mathrm{x}_n((\gamma_1)^t \wedge \cdots \wedge (\gamma_m)^t)$$

Translating Boxes (II)

DRSs with an empty universe

$$\left(\begin{array}{|l|}\hline \ \\ \hline \gamma_1 \\ \cdot \\ \cdot \\ \cdot \\ \gamma_m \\ \hline \end{array}\right)^t = (\gamma_1)^t \wedge \cdots \wedge (\gamma_m)^t$$

Translating basic conditions

basic conditions and atomic formulas are identical:

$$\begin{array}{rcl} (R(\mathrm{x}_1,\ldots,\mathrm{x}_n))^t & = & R(\mathrm{x}_1,\ldots,\mathrm{x}_n) \\ (\mathrm{x}_1 = \mathrm{x}_2)^t & = & \mathrm{x}_1 = \mathrm{x}_2 \\ (\mathrm{x} = \mathrm{c})^t & = & \mathrm{x} = \mathrm{c} \\ (\mathrm{c} = \mathrm{x})^t & = & \mathrm{c} = \mathrm{x} \end{array}$$

Translating complex conditions

Negations and disjunctions are straightforward to translate:

$$\begin{array}{rcl} (\neg \mathrm{K})^t & = & \neg(\mathrm{K})^t \\ (\mathrm{K}_1 \vee \mathrm{K}_2)^t & = & (\mathrm{K}_1)^t \vee (\mathrm{K}_2)^t \end{array}$$

Translating the $\Rightarrow$

antecedent DRS has a non-empty universe:

$$\left(\begin{array}{|l|}\hline \mathrm{x}_1,\ \ldots,\ \mathrm{x}_n \\ \hline \gamma_1 \\ \cdot \\ \cdot \\ \cdot \\ \gamma_m \\ \hline \end{array} \Rightarrow \mathrm{K}\right)^t = \forall \mathrm{x}_1 \cdots \forall \mathrm{x}_n(((\gamma_1)^t \wedge \cdots \wedge (\gamma_m)^t) \rightarrow (\mathrm{K})^t)$$

Translating the $\Rightarrow$ (II)

antecedent DRS has an empty universe:

$$\left(\begin{array}{|l|}\hline \ \\ \hline \gamma_1 \\ \cdot \\ \cdot \\ \cdot \\ \gamma_m \\ \hline \end{array} \Rightarrow \mathrm{K}\right)^t = ((\gamma_1)^t \wedge \cdots \wedge (\gamma_m)^t) \rightarrow \mathrm{K}^t$$

Important

- This translation gives us a semantics for our DRS language. Why?
- DRSs built using a vocabulary V talk about models of that vocabulary.
- So what exactly does a DRS B say about such models? This is simple: B says exactly what B^t says!
- Moreover, the way is now open for performing inference on DRSs. In effect we now know how to compile DRSs down into first-order formulas.

Building DRSs with Lambdas

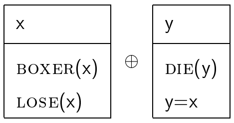
We will:

- take our DRS language, add the λ operator, the application operator @ and a new operator $\oplus$ (the *merge*).
- build representations bottom up with the help of λ, @, and $\oplus$.

Result: a system called λ-DRT which can give natural treatments of a wide range of phenomena.

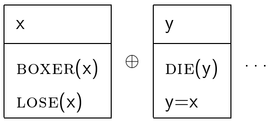
Merging

An additional ingredient: the *merge* operator: $\oplus$ allows to build a new DRS from two existing DRSs:

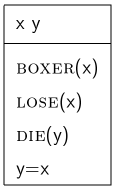


Merge-Reduction

The complex DRS:

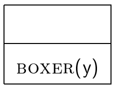


...can be reduced to the DRS:

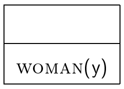


Lexical Items in λ-DRT: Nouns

boxer: λy.

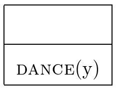


woman: λy.



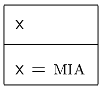
Lexical Items in λ-DRT: Intransitive Verbs

dances: λy.



Lexical Items in λ-DRT: Proper Names

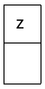
Mia: λP.



$\oplus$ P@x

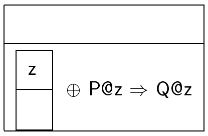
Lexical Items in λ-DRT: Determiners

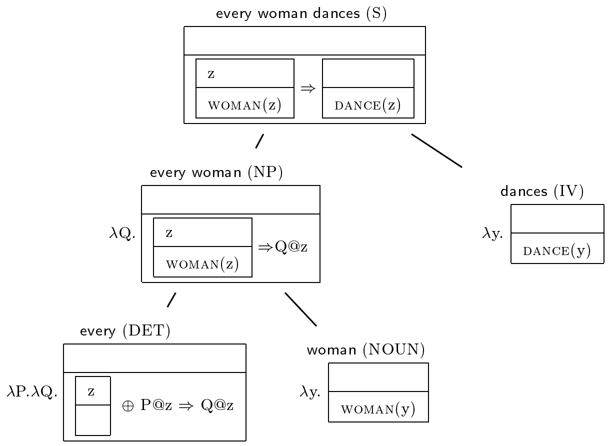
a: λP.λQ.



$\oplus$ P@z $\oplus$ Q@z

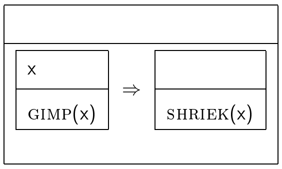
every: λP.λQ.





Representing DRSs in PROLOG

DRS in box notation



DRS in PROLOG:

```
drs([],[drs([X],[gimp(X)]) > drs([],[shriek(X)])]).
```


Implementing λ-DRT in PROLOG (I)

- We need to supply α-conversion for DRSs (why?)
- Therefore, we also need to revise β-conversion
- Finally, we need to implement merge-reduction

These operations are defined in `drsPredicates.pl`

Implementing λ-DRT in Prolog (II)

- Semantic Rules:
  essentially the same as for the lambda calculus
- Semantic Macros:

```
nounSem(Relation,lambda(X,drs([],[BasicCondition]))):-
   compose(BasicCondition,Relation,[X]).

ivSem(Relation,lambda(X,drs([],[BasicCondition]))):-
   compose(BasicCondition,Relation,[X]).

detSem(indef,
      lambda(P,lambda(Q,merge(merge(drs([X],[]),P@X),Q@X))))
```


Today Lecture 9: Presupposition

Anaphora & Presupposition

- Presupposition, presuppositions triggers, binding problem, projection problem, accommodation
- Presupposition as anaphora in DRT (Van der Sandt/Geurts)
- Implementing Presupposition Resolution
- Demos:
  - `presupDRT.pl`
  - `presupScoreDRT.pl`

What are Presuppositions?

These examples force us to take something for granted:

The couple that won the dance contest was pleased.
Jody loves her husband.
Vincent regrets that Mia is married.

The following sequences are unacceptable:

Jody has no husband. ? Jody loves her husband.
Mia is not married. ? Vincent regrets that Mia is married.

Presupposition is not entailment

Whatever we are dealing with here, it behaves completely differently from ordinary entailment!

Not only does

"Jody loves her husband"

take it for granted that Jody has a husband, but so does its negation:

"Jody does not love her husband"

Presupposition Triggers

Question: How do presuppositions come to life?
Answer: They are lexically triggered (at least for most cases in English)

- the definite article "the"
- the possessive "her"
- the verb "regret"

Such lexical items are called presupposition triggers: once they are used, they introduce a presupposition thereby putting extra constraints on the context.

Notorious Problems with Presupposition

- The Binding Problem
- The Projection Problem
- Accommodation

The Binding Problem

An example like

"A boxer nearly escaped from his apartment"

induces the presupposition that "someone has an apartment".

But it does *not* presuppose that just anyone has an apartment, nor that some boxer or other owns an apartment.

- It is *the boxer we are actually talking about* who has an apartment.
- The existentially quantified NP ties together two types of information — ordinary factual information, and presuppositional information.
- As we shall soon see, Van der Sandt's DRT-based account handles it rather elegantly.

The Projection Problem

The following sentence presupposes that Mia has a husband.

"Mia's husband is out of town."

Now consider the following complex sentences:

"If Mia has a husband, then her husband is out of town."

"If Mia is married, then her husband is out of town."

These do *not* presuppose that Mia has a husband. Although:

"If Mia dates Vincent, then her husband is out of town."

clearly does.

The moral is clear: we need to be careful when dealing with presupposition triggers in complex sentences!

Accommodation

Analyzing

"Vincent informed his boss."

yields the elementary presupposition that Vincent has a boss.

What if we don't have a clue whether Vincent has a boss or not?

Two solutions:

1. the simplistic way is simply to refuse to accept the utterance
2. the robust way is try to add the presupposition to the context

The addition process is called *accommodation*, and it is best viewed as a repair strategy.

Dealing with Presupposition in DRT

"Presupposition as Anaphora" (Van der Sandt)

- Presuppositions are essentially extremely *rich* pronouns.
- Like ordinary pronouns, they make use of the notion of accessibility in the way we are familiar with.
- But presuppositions are rich in a way that pronouns aren't: they have descriptive content.
- Thus: *presuppositions introduce new DRSs*.

This turns out to be a good way of coping with the binding, projection, and accommodation problems.

Let's turn to details!

We have to carry out three tasks:

1. select presupposition triggers in the lexicon; and
2. indicate what they presuppose.
3. implement a resolution algorithm for presuppositions

Task 1 appeals to our empirical knowledge of presupposition. We classify as presupposition triggers:

- **definite article**,
- **possessive constructions**, and
- **proper names**

For task 2 we extend the core DRS language with the α operator.

Marking Presuppositions in DRSs — Presupposition triggers introduce the α-operator

Example: "A patron calls the waitress"

DRS:

y
PATRON(y)

$\oplus$ (

x
WAITRESS(x)

α

CALL(y,x)

)

But how do we *resolve* these DRSs containing presuppositions?

The Presupposition Resolution Algorithm

1. Generate a DRS for the input sentence with with all elementary presuppositions marked by the α-operator. Merge this DRS with the DRS of the discourse so far processed. Go to Step 2.
2. Traverse the DRS, and on encountering an α-marked DRS try to
   (a) link the presupposed information to an accessible antecedent (partial match). Go to Step 2.
   (b) alternatively, accommodate the information to a superordinated level of discourse. Go to Step 2.

   otherwise go to Step 3.
3. Reduce any merges appearing in the DRS obtained from Step 3. Go to Step 4.
4. Remove those DRS from the set of potential readings that violate the acceptability constraints.

Resolution Example

A waitress serves Vincent a coffee. Vincent lights up a cigarette. The waitress leaves.

Step 1:

u v y c
WAITRESS(y) COFFEE(u) VINCENT(v) SERVE(y,v,u) CIGARETTE(c) LIGHT-UP(v,c)

$\oplus$ (

x
WAITRESS(x)

α

LEAVE(x)

)

Step 2:

u v y c
WAITRESS(y) COFFEE(u) VINCENT(v) SERVE(y,v,u) CIGARETTE(c) LIGHT-UP(v,c)

$\oplus$

LEAVE(y)

Step 3:

u v y c
WAITRESS(y) COFFEE(u) VINCENT(v) SERVE(y,v,u) CIGARETTE(c) LIGHT-UP(v,c) LEAVE(y)

Note: this algorithm is non-deterministic

In other words, Step 3 will results in a set of potential readings.
So it will also produce the following (some being rather obscure) DRSs:

u v y c	u v y c	u v y c	u v y c w
WAITRESS(y) COFFEE(u) VINCENT(v) SERVE(y,v,u) CIGARETTE(c) LIGHT-UP(v,c) WAITRESS(u) LEAVE(u)	WAITRESS(y) COFFEE(u) VINCENT(v) SERVE(y,v,u) CIGARETTE(c) LIGHT-UP(v,c) WAITRESS(v) LEAVE(v)	WAITRESS(y) COFFEE(u) VINCENT(v) SERVE(y,v,u) CIGARETTE(c) LIGHT-UP(v,c) WAITRESS(c) LEAVE(c)	WAITRESS(y) COFFEE(u) VINCENT(v) SERVE(y,v,u) CIGARETTE(c) LIGHT-UP(v,c) WAITRESS(w) LEAVE(w)

(Step 4 will help us out here. More about this tomorrow.)

Partial Match Example

Vincent talked to Marvin. The bastard shot him in the face.

Step 1:

x y
VINCENT(x) MARVIN(x) TALKED-TO(x,y)

$\oplus$ (

z
BASTARD(z)

α

SHINTF(z)

)

Step 2:

x y
VINCENT(x) MARVIN(y) TALKED-TO(x,y) BASTARD(x)

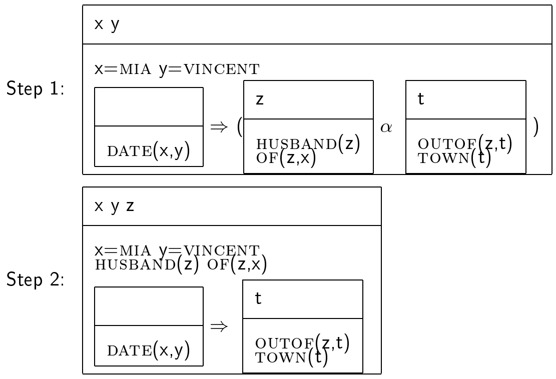
$\oplus$

SHINTF(x)

Note: more solutions

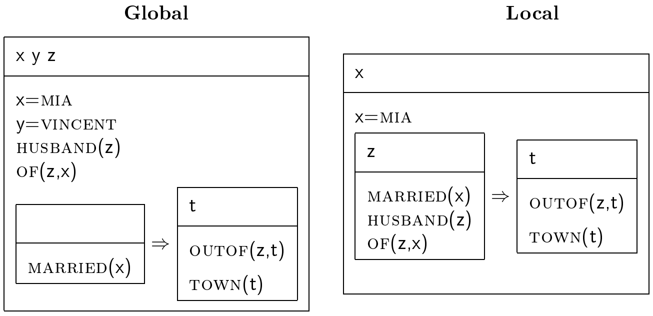
Another Example (Accommodation)

If Mia dates Vincent, then her husband is out of town.

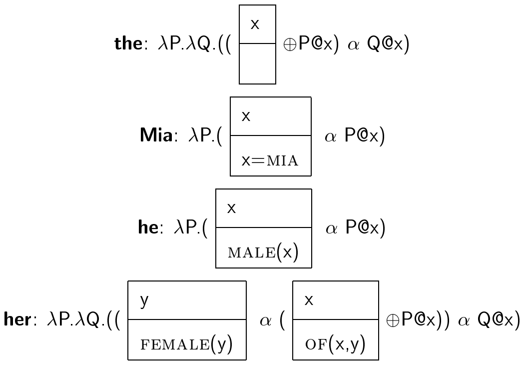


Local Accommodation

If Mia is married, then her husband is out of town.



Representing Presupposition Triggers



Presupposition Projection in Prolog

`presupDRT.pl`

- Rendering the *Lexical Macros* for presupposition triggers
- Semantic Rules are essentially the same as for `lambdaDRT.pl`
- Implementation of the Resolution Algorithm (`resolvePresup.pl`)

`presupScoreDRT.pl`

- Gives "scores" for potential readings, based on the trigger and how it is resolved.
- For instance, proper names that are accommodated at the global DRSs get a high score. Pronouns that are accommodated get a low score.

Today Lecture 10: Putting it all together again

- Imposing Van der Sandt's Acceptability Constraints for the Presupposition Resolution algorithm
- Implementing these—using theorem provers and model builders
- Putting It All Together (Curt/Doris)
- Demos:
  - `acDRT.pl`
  - `curtDRT.pl`

The Presupposition Resolution Algorithm

1. Generate a DRS for the input sentence with with all elementary presuppositions marked by the α-operator. Merge this DRS with the DRS of the discourse so far processed. Go to Step 2.
2. Traverse the DRS, and on encountering an α-marked DRS try to
   (a) link the presupposed information to an accessible antecedent (partial match). Go to Step 2.
   (b) alternatively, accommodate the information to a superordinated level of discourse. Go to Step 2.
   otherwise go to Step 3.
3. Reduce any merges appearing in the DRS obtained from Step 3. Go to Step 4.
4. **Remove those DRS from the set of potential readings that violate the acceptability constraints**.

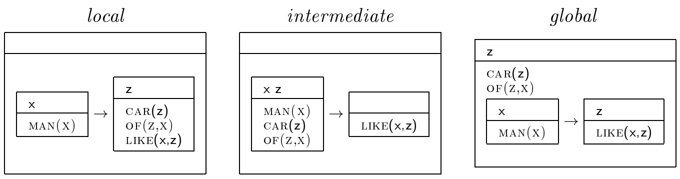
The Acceptability Constraints

1. Solution should not contain free variables (Free Variable Check)
2. We need to obey the consistency constraint;
3. We should obey the informativity constraint;
4. We also should obey local informativity and consistency.

Free Variable Check

Example: **Every man likes his car**

Readings:



The Other Acceptability Constraints

Be consistent:

Jody is married. Her husband is a dealer.
Jody is not married. Her husband is a dealer.

The consistency constraint clearly helps to filter out non-sensical readings.

Be informative:

Jody is a boxer. Jody is a boxer.
Mia is married. She has a husband.

This might not be directly clear why the informativeness constraint is useful for presupposition resolution (in fact, it gives non-intuitive predictions sometimes), but it will be more clear when we discuss the local constraints.

Checking the Constraints

Assume: KB is a formula with relevant background information

Consistency:

1) Φ is the translation of the DRS to first-order logic

2) use first-order tools to check whether $KB \wedge \Phi$ is consistent.

(i.e., if $\neg(KB \wedge \Phi)$ is valid then Φ is inconsistent)

Informativity

1) Let Ψ be the first-order formula $(KB \wedge OLD) \rightarrow NEW$, where OLD is the translation of the old DRS and NEW is the translation of the new DRS.

2) Then the new utterance is informative if and only if Ψ is *not* valid.

The Local Constraints

Van der Sandt's locality constraints are as follows:

superordinated DRSs should neither imply a subordinated DRS, nor a negated subordinated DRS

Examples:

If Marsellus has a car, then he has a car.
If Marsellus has no car, then he has a car.
Vincent eats a burger. If he eats a burger, he enjoys it.
If Vincent eats a burger, he enjoys it. Vincent eats no burger.

Effect: filter out certain accommodation possibilities.

If Mia is married, then her husband is out of town.
Either Mia is a spinster or her husband is out of town.

global accommodation:

x z
x=MIA HUSBAND(z,x) [| MARRIED(x)] ⇒ [| OUT-OF-TOWN(z)]

intermediate accommodation:

x
x=MIA [z | MARRIED(x), HUSBAND(z,x)] ⇒ [| OUT-OF-TOWN(z)]

local accommodation:

x
x=MIA [| MARRIED(x)] ⇒ [z | HUSBAND(z,x), OUT-OF-TOWN(z)]

Defining the Local Constraints

Let <U,C> be a DRS with U a set of discourse referents and C a set of conditions. Then the function **SuperSub** returns a set of ordered pairs of DRSs that are computed for <U,C> according to the following clauses only:

1. ⟨<U,C-c>, B⟩ if c∈C and c is of the form B⇒B′, B∨B′, B′∨B or ¬B.
2. ⟨<U,C-c>⊕B, B′⟩ if c∈C and c is of the form B⇒B′.
3. ⟨<U,C-c>⊕B_1, B_2⟩ if c∈C and c is of the form B⇒B′, B∨B′, B′∨B or ¬B, and ⟨B_1,B_2⟩∈**SuperSub**(B).
4. ⟨<U,C-c>⊕B⊕B_1, B_2⟩ if c∈C and c is of the form B⇒B′, and ⟨B_1,B_2⟩∈**SuperSub**(B′).

SuperSub Example

SuperSub([$\vec{x}$ | $\vec{c_1}$, ¬ B_1, $\vec{c_2}$, B_2 ⇒ B_3, $\vec{c_3}$]) =

{⟨ [$\vec{x}$ | $\vec{c_1}$, $\vec{c_2}$, B_2 ⇒ B_3, $\vec{c_3}$], B_1⟩, ⟨ [$\vec{x}$ | $\vec{c_1}$, ¬ B_1, $\vec{c_2}$, $\vec{c_3}$], B_2⟩, ⟨ [$\vec{x}$ | $\vec{c_2}$, ¬ B_1, $\vec{c_2}$, $\vec{c_3}$] ⊕B_2, B_3⟩}

More on Van der Sandt's Local Constraints

- The local informativity constraint is useful: It allows blocking of global accommodation (dealing with the Projection Problem)
- Sometimes local informativity seems too strong.

Mia loves a man. Every man is a boxer.

x y
MAN(y) LOVE(X,Y) x = MIA [z | MAN(Z)] ⇒ [| BOXER(z)]

Solution: allow "levels of consistency/informativity"

Implementing the Acceptability Constraints

This is how consistency is implemented in Prolog:

```
consistent(Drs):-
        drs2fol(Drs,Phi),
        backgroundKnowledge(Phi,Chi),
        callTheoremProver(~(Chi & Phi),Proof),
        (
            Proof=proof, !, fail
        ;
            DomainSize=15,
            callModelBuilder(Chi & Phi,DomainSize,Model),
            Model=model(_,_)
        ).
```

Informativity, and the local constraints are implemented similarly. Use acDRT to test the definitions.

Finally: Putting it all together again

- CURT is back, but now with a DRT backbone
- Functionality of `curtDRT.pl` is similar to that of `superCurt.pl`, but now of course it deals with pronouns and presuppositions.
- there is a similar system named DORIS. An internet interface is available on `http://www.coli.uni-sb.de/~bos/doris/`