What is computational semantics?

1. *How can we automate the process of associating semantic representations with expressions of natural language?*
2. *How can we use logical representations of natural language expressions to automate the process of drawing inferences?*
3. *What is the role of lexical semantics in 1 and 2*

This course introduces some fundamental techniques for tackling these questions.

Outline

Day 1: Lambdas, Grammar Engineering, Ontologies

Day 2: Discourse Representations, Pronouns, Focusing

Day 3: Presupposition, Acceptability Constraints, Curt

Lecture Notes

Representation & Inference for Natural Language
Patrick Blackburn & Johan Bos
(Chapter 2, 6.3, 7, 8.2, 9, 10, 11)
`http://www.coli.uni-sb.de/~bos/comsem`

Prolog Implementation

`http://www.coli.uni-sb.de/~bos/comsem/prolog`

Semantic construction

Given a sentence of English, is there a **systematic** *way of constructing its semantic representation?*

This question is far too general. Let's ask a more specific one:

Is there a **systematic** *way of translating simple sentences such as*

Vincent likes Mia

A woman snorts.

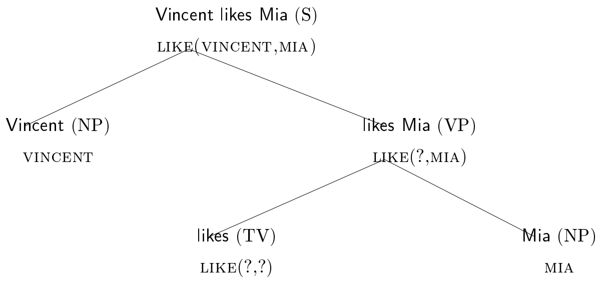
Every boxer loves a woman.

into first-order logic?

Compositionality

- 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.

Syntactic Structure



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 --> [snorts].
det --> [a].               tv --> [loves].
det --> [every].           tv --> [likes].
```

This grammar accepts sentence like Vincent snorts 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!

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, the PROLOG *unification mechanism gives us the tools needed to combine and pass up semantic representations*

The Lambda Calculus

- A 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
- and an operation called *lambda conversion* performs the required substitutions.

Think of the lambda calculus as 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:

λx.MAN(x)

The prefix λx binds the occurrence of x in MAN(x).

Functional application

the symbol @ *is used to indicate 'functional application'* (it indicates that we wish to perform a substitution.)

Example:

λx.MAN(x)@VINCENT

β-conversion

The required substitution is performed by β-conversion:

MAN(VINCENT)

Marking more complex kinds of information

Our semantic representation of a woman will be:

λQ.$\exists$x(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

Every boxer growls

Step 1: assign lambda expressions to the basic syntactic categories

boxer: λy.BOXER(y)

growls: λx.GROWL(x)

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

Every boxer growls (II)

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)
λP.λQ.$\forall$x(P@x$\rightarrow$Q@x)@λy.BOXER(y)

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

boxer (NOUN)
λy.BOXER(y)

β-Conversion

Applications (@) are instructions to carry out β-conversion

Performing the demanded substitution yields:

every boxer (1): λP.λQ.$\forall$x(P@x$\rightarrow$Q@x)@λy.BOXER(y)

every boxer (2): λQ.$\forall$x(λy.BOXER(y)@x$\rightarrow$Q@x)

every boxer (3): λQ.$\forall$x(BOXER(x)$\rightarrow$Q@x)

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

The final representation

every boxer growls (S)
$\forall$x(BOXER(x)$\rightarrow$GROWL(x))

every boxer (NP)
λQ.$\forall$x(BOXER(x)$\rightarrow$Q@x)

growls (VP)
λx.GROWL(x)

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

boxer (NOUN)
λy.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

Proper Names: Clouds on the Horizon?

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

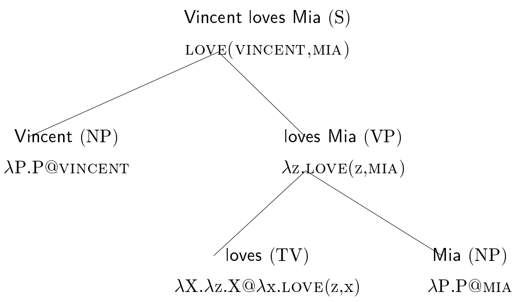
In fact, there's no problem at all!

Mia: λP.P@MIA

Vincent: λP.P@VINCENT

These representations *can* be used as functors.

Proper Names in Action



The Three Tasks Revisited

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

Task 2 Specify semantic representations for the lexical items with the help of the lambda calculus.

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 must now show that these tasks lend themselves naturally to computational implementation.

Adding λ and @ to the DCGs

Note: `lambda` and `@` are (almost) all we need to build semantic representations!

Add the `@` instructions to the original DCG rules:

```
s(NP@VP)   --> np(NP), vp(VP).
np(PN) --> pn(NP).
np(DET@NOUN) --> det(DET), noun(NOUN).
vp(IV) --> iv(IV).
vp(TV@NP) --> tv(TV), np(NP).
```

- unary branching rules percolate up their semantic representation
- binary branching rules use `@` to build a semantic representations out of their component representations

The real work is done at the lexical level

Some very simple lexical entries:

```
noun(lambda(X,footmassage(X))) --> [foot,massage].
noun(lambda(X,woman(X))) --> [woman].
iv(lambda(X,growl(X))) --> [growls].
```

Abstractions formed using complex variables need a different treatment.

```
pn(lambda(P,P@vincent)) --> [vincent].
pn(lambda(P,P@mia)) --> [mia].
```


Determiners

a: $\lambda P.\lambda Q.\exists x.(P@x \wedge Q@x)$

```
det(lambda(P,lambda(Q,exists(X,(P@X)&(Q@X))))) --> [a].
```

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

```
det(lambda(P,lambda(Q,forall(X,(P@X)>(Q@X))))) --> [every].
```


Let's try our DCG with lambdas!

- semantic construction during parsing is extremely easy
- we simply use `@` to record the required function/argument structure.

Here is an example query:

```
?- s(Sem,[mia,snorts],[]).

Sem = lambda(P,P@mia)@lambda(X,snort(X))
```

But of course, we need to do more work *after* parsing!

Carrying out β-conversion.

```
betaConvert(Var,Result):-
   var(Var), !, Result=Var.

betaConvert(Functor @ Arg,Result):-
   compound(Functor),
   betaConvert(Functor,ConvertedFunctor),
   apply(ConvertedFunctor,Arg,BetaConverted), !,
   betaConvert(BetaConverted,Result).

betaConvert(Formula,Result):-
   compose(Formula,Functor,Formulas),
   betaConvertList(Formulas,ResultFormulas),
   compose(Result,Functor,ResultFormulas).
```


Define the apply/3 predicate

By using Prolog unification we can do so with a single clause:

```
apply(Functor,Argument,Result):-
      Functor=lambda(Argument,Result).
```

Suppose we wanted to apply λx.WALK(x) to VINCENT. We would do this in Prolog by means of the following query:

```
?- apply(lambda(X,walk(X)),vincent,Result).

X = vincent
Result = walk(vincent)

yes
```


Grammar Engineering

- The lambda-based construction methods make sense from a grammar engineering perspective (representation independent)
- we should strive for a grammar that is MODULAR, EXTENDIBLE, and REUSABLE.
- We adopt a simple three-level grammar architecture:
  1. DCG *rules* (including a slot for the semantics)
  2. a *lexicon* (syntactic and semantic information about words)
  3. a set of *semantic macros*

How do these three levels interact?

The lexical rules include a lexical look-up and a call to a semantic macro.

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


The LEXICON

Intransitive verbs:

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

Nouns:

```
lexicon(noun,boxer,[boxer],human).
lexicon(noun,bkburger,[big,kahuna,burger],nonhuman).
```

Determiners:

```
lexicon(det,_,[every],uni).
lexicon(det,_,[a],indef).
```

Pronouns:

```
lexicon(pro,_,[he],male).
lexicon(pro,_,[she],female).
```


The Semantic Macros

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

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

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

A Driver Predicate

```
:- module(mainLambda,[parse/0]).

:- use_module(readLine,[readLine/1]),
   use_module(comsemOperators),
   use_module(comsemPredicates,[printRepresentation/1,compose/3]),
   use_module(betaConversion,[betaConvert/2]).

:- [englishGrammar], [englishLexicon], [semMacrosLambda].

parse:-
   readLine(Sentence),
   s(Formula,Sentence,[]),
   betaConvert(Formula,Converted),
   printRepresentation(Converted).
```


First Results:

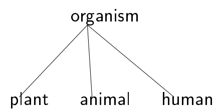
```
?- parse.
> Vincent snorts
snort(vincent)
yes

?- parse.
> Vincent knows every boxer
forall(A,boxer(A)>know(vincent,A))
yes

?- parse.
> Vincent knows every boxer that likes Mia.
forall(A,like(A,mia)&boxer(A)>know(vincent,A))
yes
```

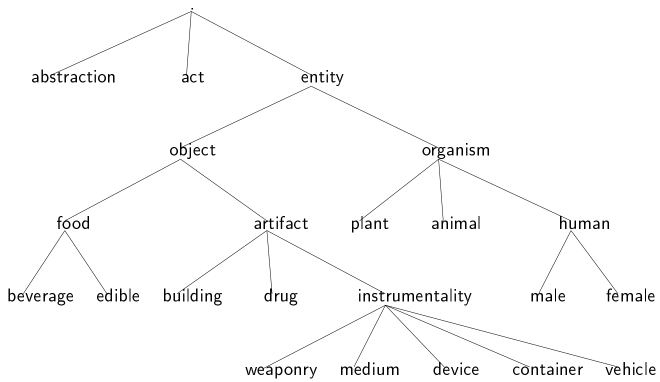

Question:
What is the meaning of BOXER? Or FIVE_DOLLAR_SHAKE?

Add lexical knowledge:

- divide the objects in the (our) world into classes.
- some of these classes are *disjoint*
- some classes *inherit* properties of other classes.

These relations are interconnected.

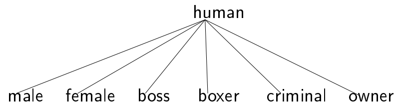




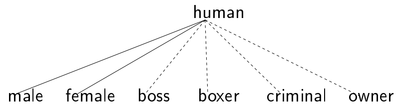
Refinement

What about nouns like 'boss', 'boxer', 'owner', or 'criminal'?

These should be definitely classified as human. But they aren't either male or female, they can be both.



But this gives us incorrect information! New version:



Purpose of the Ontology

Now we have a nice hierarchy of objects of our domain, but:

- how do we represent it?
- what can we do with it?

In this course, we will use it for two purposes:

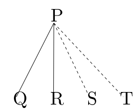
- as background knowledge for automated reasoning tasks
- and as the basic source for determining sortal conflicts

Integrating the Ontology in the Lexicon

```
lexicon(noun,car,[car],[vehicle]).
lexicon(noun,chainsaw,[chainsaw],[device]).
lexicon(noun,criminal,[criminal],human).
lexicon(noun,customer,[customer],human).
lexicon(noun,drug,[drug],[artifact]).
lexicon(noun,entity,[entity],[top]).
lexicon(noun,episode,[episode],abstraction).
lexicon(noun,female,[female],[human]).
```


Translate to first-order

For the following scheme



we can derive the formulas:

$\forall x(Q(x) \rightarrow P(x))$,
$\forall x(R(x) \rightarrow P(x))$,
$\forall x(S(x) \rightarrow P(x))$,
$\forall x(T(x) \rightarrow P(x))$, and also $\forall x(Q(x) \rightarrow \neg R(x))$.

Question: do we also need $\forall x(R(x) \rightarrow \neg Q(x))$?

In Prolog:

```
generateOntology(Formulas):-
   generateIsa(I0),
   generateDisjoint(I0-I1,I2),
   isa2fol(I1,[]-F),
   isa2fol(I2,F-Formulas).
```

Running this predicate gives us a list of terms that look like:

```
?- generateOntology(O).

O = [forall(A,abstraction(A)>top(A)),
     forall(B,act(B)>top(B)),
     forall(C,animal(C)>organism(C)),
     forall(D,artifact(D)>object(D)),
     forall(E,beverage(E)>food(E)),
     forall(F,bkburger(F)>edible(F)),
     forall(G,boss(G)>human(G)),.....]
```


Detecting Sortal Conflicts

Two noun objects from our ontology are consistent if we can't proof that they are inconsistent.

```
consistent(X,Y):-
     generateIsa(I), generateDisjoint(I-Isa,Disjoint),
     \+ inconsistent(X,Y,Isa,Disjoint).
```

Two objects are inconsistent if they are disjoint:

```
inconsistent(X,Y,_,Disjoint):-
     member(disjoint(X,Y),Disjoint).
```

Or if for one of them there exists a hypernym inconsistent with the other noun object.

```
inconsistent(X,Y,Isa,Disjoint):-
     member(isa(X,Z),Isa),
     inconsistent(Z,Y,Isa,Disjoint).
```


DAY 2
Discourse Representations
Pronouns
Focusing

Discourse Representation Theory

- a swift overview of DRT
- constructing a picture of the changing context
- informally sketch the *semantics*
- introduce the concept of *accessibility*
- the *standard construction algorithm*
- a DRS to first-order logic compiler

Aim: making 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.

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

Representing the Meaning of a 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'.

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

'A woman snorts. She collapses'.

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

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
WOMAN(x) 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 "COLLAPSE(y)"
3. adding an equation "y=x"

A woman snorts.
She collapses.

x y
WOMAN(x) SNORT(x) COLLAPSE(y) y=x

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

Accessibility

is a 'geometrical' concept; it has to do with the way DRSs are stacked one inside the other.

Discourse referents of DRS B_1 are accessible from DRS B_2 when B_1 *subordinates* B_2, or when B_1 *equals* B_2.

Informally: *B_1 subordinates B_2 if B_2 is embedded in B_1.*

Every woman snorts. ? She collapses.

y		
[x | WOMAN(x)]	⇒	[| SNORTS(x)]
COLLAPSE(y)		
y=?		

There is no accessible discourse referent for y

Every gangster who eats a big kahuna burger enjoys it.

$$\begin{array}{|l|}\hline \\ \hline \begin{array}{|l|}\hline x\ y \\ \hline \textsc{gangster}(x) \\ \textsc{big-kahuna-burger}(y) \\ \textsc{eat}(x,y) \\ \hline \end{array} \Rightarrow \begin{array}{|l|}\hline z \\ \hline \textsc{enjoy}(x,z) \\ z=? \\ \hline \end{array} \\ \\ \hline \end{array}$$

There *is* an accessible discourse referent for z: y (and x)

DRS Languages

- How to deal with universal quantification and negation.
- DRSs can be combined using various connectives
- these nested pictures offer us all representational power needed

DRS languages share many ingredients of first-order languages.

- they contain the symbols $\neg$, $\vee$, $\rightarrow$
- they normally don't contain $\wedge$
- they contain the symbol $=$
- the contain 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)

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.

$$\begin{array}{|l|}\hline x\ y \\ \hline \textsc{woman}(x) \\ \textsc{boxer}(y) \\ \textsc{admire}(x,y) \\ \hline \end{array}$$

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?

$$\begin{array}{|l|}\hline \\ \hline \begin{array}{|l|}\hline y \\ \hline \textsc{woman}(y) \\ \hline \end{array} \Rightarrow \begin{array}{|l|}\hline x \\ \hline \textsc{boxer}(x) \\ \textsc{love}(y,x) \\ \hline \end{array} \\ \\ \hline \end{array}$$

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

DRT and First-Order Logic

- In spite of their differences, DRT and first-order logic are obviously related.
- Given any vocabulary without function symbols, we can use it to build either a DRS language or a first-order language.
- Moreover, both languages will interpreted on exactly the same models.
- Thus it is natural to try to relate DRS and first-order languages via satisfaction preserving translations.
- any DRS language and its corresponding first-order language are intertranslatable: either can be compiled down into the other, and the translations involved are straightforward and efficient.

Translate DRSs to first-order formulas

$$\left(\begin{array}{|l|}\hline x_1,\ \ldots,\ x_n \\ \hline \gamma_1,\ \ldots,\ \gamma_m \\ \hline \end{array}\right)^{fo} = \exists x_1 \cdots \exists x_n((\gamma_1)^{fo} \wedge \cdots \wedge (\gamma_m)^{fo})$$

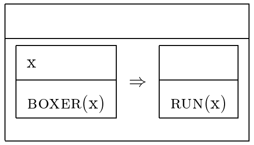
$$\begin{aligned} (R(x_1,\ldots,x_n))^{fo} &= R(x_1,\ldots,x_n) \\ (\tau_1 = \tau_2)^{fo} &= \tau_1 = \tau_2 \end{aligned}$$

$$\begin{aligned} (\neg B)^{fo} &= \neg(B)^{fo} \\ (B_1 \vee B_2)^{fo} &= (B_1)^{fo} \vee (B_2)^{fo} \end{aligned}$$

$$\left(\begin{array}{|l|}\hline x_1,\ \ldots,\ x_n \\ \hline \gamma_1,\ \ldots,\ \gamma_m \\ \hline \end{array} \Rightarrow B\right)^{fo} = \forall x_1 \cdots \forall x_n(((\gamma_1)^{fo} \wedge \cdots \wedge (\gamma_m)^{fo}) \rightarrow (B)^{fo})$$

DRT in Prolog

'Every boxer runs':



In Prolog we represent this DRS as:

```
drs([],[drs([X],[boxer(X)]) > drs([],[run(X)])]).
```

- use operator definitions for $\rightarrow$, $\neg$, and $\vee$: `>` `~` `v`
- discourse referents are represented as Prolog variables

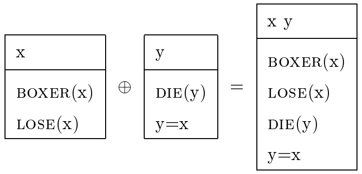
Building Discourse Representations

DRT is a highly flexible theory, compatible with a wide range of semantic construction strategies.

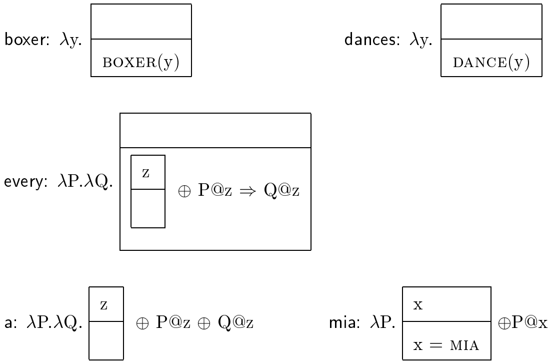
Alternative approaches:

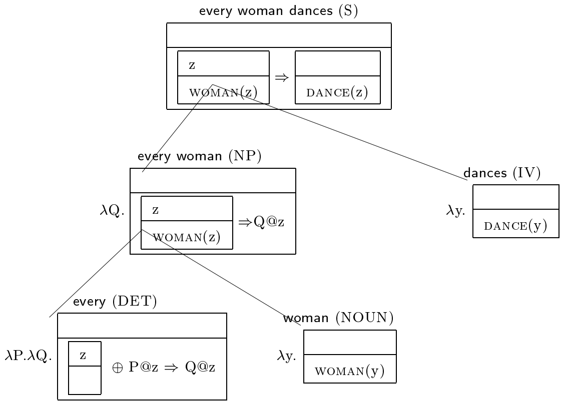
1. DRS-threading;
2. λ-DRT;
3. Underspecified DRSs

The Merge operation: $\oplus$



Lexical Items in λ-DRT





Define Semantic Macros

Abstract away from specific lexical entries by using semantic macros:

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

ivSem(Sym,lambda(X,drs([],[Cond]))):-
   compose(Cond,Sym,[X]).

modSem(neg,lambda(P,lambda(X,drs([],[~(P@X)])))).

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

Define a Driver

```
parse:-
   readLine(Discourse),
   d(MergeDrs,Discourse,[]),
   betaConvert(MergeDrs,ReducedDrs),
   mergeDrs(ReducedDrs,Drs),
   printRepresentation(Drs).
```

Example session:

```
?- parse.

> Every boxer growls.

drs([],[drs([A],[boxer(A)])>drs([],[growl(A)])])
```


Implementing Pronoun Resolution in DRT

- take into account the geometrical structure of DRSs.
- we will use *gender* as sortal information to rule out some anaphoric links.

the grammar rule for handling pronouns:

```
pro(Pro)-->
  {
   lexicon(pro,Gender,Phrase,Type),
   proSem(Gender,Type,Pro)
  },
  Phrase.
```


The LEXICON provides gender information:

```
lexicon(pro,female,[she],nonrefl).
lexicon(pro,male,[he],nonrefl).
lexicon(pro,nonhuman,[it],nonrefl).
```

Extend our Prolog DRS language with notation

```
alfa(X,Type,Gender,Drs)
```

The SEMANTIC MACRO introduces a new marker α

```
proSem(Gender,Type,lambda(P,alfa(X,Type,Cond,P@X))):-
   compose(Cond,Gender,[X]).
```


Example Prolog query:

```
?- d(B,[a,woman,snorts,she,collapses],[]).
B = merge(
         merge(
              merge(
                   drs([A],[]),
                   drs([],[woman(A),female(A)])
                  ),
              drs([],[snort(A)])
             ),
         alfa(B,nonrefl,female(B),drs([],[collapse(B)]))
        )
```

In box-notation this Prolog-expression is translated as:

$$((\begin{array}{|c|}\hline \text{a} \\ \hline \\ \hline \end{array} \oplus \begin{array}{|c|}\hline \\ \hline \textsc{woman}(\text{a}) \\ \hline \end{array}) \oplus \begin{array}{|c|}\hline \\ \hline \textsc{snort}(\text{a}) \\ \hline \end{array}) \oplus \alpha_{\text{b}} \begin{array}{|c|}\hline \\ \hline \textsc{collapse}(\text{b}) \\ \hline \end{array}$$

Implementing Pronoun Resolution

```
resolveDrs([merge(B1,B2)|A1]-[drs(D,C)|A3]):-
       resolveDrs([B1|A1]-A2),
       resolveDrs([B2|A2]-[drs(D2,C2),drs(D1,C1)|A3]),
       append(D1,D2,D),
       append(C1,C2,C).

resolveDrs([alfa(Referent,Type,Gender,B1)|A1]-A2):-
       potentialAntecedent(A1,Referent,Gender),
       resolveDrs([B1|A1]-A2).

resolveDrs([drs(D1,C1)|A1]-A2):-
       resolveConds(C1,[drs(D1,[])|A1]-A2).
```


Selecting a potential antecedent

```
potentialAntecedent(A,X,Gender):-
   member(drs(Dom,Conds),A),
   member(X,Dom),
   compose(Gender,Symbol1,_),
   \+ (
         member(Cond,Conds),
         compose(Cond,Symbol2,[Y]),
         Y==X,
         \+ consistent(Symbol1,Symbol2)
      ).
```

...define a driver predicate...

```
parse:-
   readLine(Discourse),
   d(MergeDrs,Discourse,[]),
   betaConvert(MergeDrs,Drs),
   resolveDrs([Drs]-[ResolvedDrs]),
   printRepresentation(ResolvedDrs).
```

...and test it as follows:

```
?- parse.
> A woman snorts. She collapses.

drs([A],[collapse(A),snort(A),female(A),woman(A)])
```


Focusing, the Basic Ideas

- Add preferences to the readings that we obtain by
- using the focusing algorithm (due to Candy Sidner)
- The focusing algorithm predicts an antecedent for a pronoun
- Constraints we already have confirm or reject this choice

Focus: the element or object of discourse on which the attention of the speakers is centered

Focusing: the process of keeping track (and updating) the focus of the discourse.

Speakers use PRONOUNS to refer to the Focus!

Discourse Focus (The Gold Watch)

The Captain pulls a **gold wrist watch** out of his pocket.

This watch was first purchased by Butch's great-granddaddy.

It was bought during the First World War in a little general store in Knoxville, Tennessee.

Actor Focus

Motivation:

- the focus of the discourse is often distinguished from the actor
- both can be referred to with pronouns

Jules grabs the burger. He takes a bite of it.

discourse focus: burger
actor focus: Jules

Focus Shifts

Butch picks up a big destructive-looking hammer,
(discourse focus: hammer)

then discards it: not destructive enough.
(discourse focus: hammer)

He picks up a chainsaw,
(discourse focus: chainsaw)

thinks about it for a moment,
(discourse focus: chainsaw)

then puts it back.
(discourse focus: chainsaw)

Ordinary DRSs: `drs(Dom,Conds)`
Focus DRSs: `drs(Dom,ActorFocus,DiscourseFocus,Conds)`

x y
x
y
X=BUTCH HAMMER(y) PICK-UP(x,y) DISCARD(x,y)

Butch picks up a hammer.
He discards it.

x y z
x
z y
X=BUTCH HAMMER(y) PICK-UP(x,y) DISCARD(x,y) PICK-UP(x,z) CHAINSAW(z)

He picks up a chainsaw

The Focusing Algorithm

1. choose foci based on the first sentence of the discourse
2. use foci to resolve pronouns
3. update the foci on the basis of the current DRS

Focus Update

Actor focus: updated by taking the expected actor focus:

```
updateActorFocus([drs(D,AF,DF,C)|A],Updated):-
   expectedActorFocus(drs(D,C),Focus), !,
   addActorFocus(Focus,[drs(D,AF,DF,C)|A],Updated).
```

Discourse focus: remains if bound by a pronoun

```
updateDiscourseFocus(Anaphora,DRSs,Updated):-
   currentDiscourseFocus(DRSs,Current),
   member(_:A,Anaphora), A==Current, !, Updated=DRSs.

updateDiscourseFocus(_Anaphora,[drs(D,AF,DF,C)|A],Updated):-
   expectedDiscourseFocus(drs(D,C),Focus), !,
   addDiscourseFocus(Focus,[drs(D,AF,DF,C)|A],Updated).
```


Driver Predicate for Focusing

```
parse:-
   parse(drs([],[],[],[])).

parse(Old):-
   readLine(Discourse),
   d(SDrs,Discourse,[]),
   betaConvert(SDrs,Drs),
   resolveDrs([merge(Old,Drs)]-A,[]-Anaphora),
   updateFocus(Anaphora,A,[New]),
   printRepresentation(New), !,
   parse(New).
```


Focus in Action

```
?- parse.

> Butch picks up a hammer
drs([A,B],[B],[A],[pickup(B,A),hammer(A),B=butch,male(B)])

> He discards it.
drs([A,B],[B],[A],[discard(B,A),male(B),B=butch,hammer(A),pickup(B,A)]

> He picks up a chainsaw.
drs([A,B,C],[C],[A,B],[pickup(C,A),chainsaw(A),pickup(C,B),hammer(B),(

> He likes it.
drs([A,B,C],[C],[A,B],[like(C,A),discard(C,B),male(C),C=butch,hammer(I
```


DAY 3
Presupposition
Acceptability Constraints
Curt

Presupposition Projection

- Presupposition Triggers
- The Projection Problem
- Presupposition as Anaphora
- Presuppositional Accommodation
- Integration of Presupposition in DRT

Introducing Presuppositions

These examples force us to take something for granted:

(1) The couple that won the dance contest was pleased.

(2) Jody loves her husband.

(3) Vincent regrets that Mia is married.

The following sequences are unacceptable:

(4) Jody has no husband. ? Jody loves her husband.

(5) 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'.

Note that:

(6) Jody has no husband. ? Jody does not love her husband.

is just as bad as:

(7) Jody has no husband. ? Jody loves her husband.

Presupposition Triggers

Question: How do presuppositions come to life?
Answer: They are lexically triggered

- 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.

The Projection Problem

Mia's husband is out of town.	+
If Mia has a husband, then her husband is out of town.	–
If Mia is married, then her husband is out of town.	–
If Mia dates Vincent, then her husband is out of town.	+

\+ presupposes that Mia has a husband.
– does *not* presuppose that Mia has a husband.

- 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 two tasks:

1. select presupposition triggers in the lexicon; and
2. indicate what they presuppose.

Task 1 appeals to our empirical knowledge of presupposition.

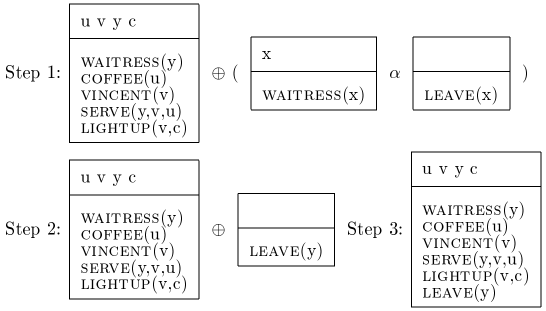
We classify as presupposition triggers:

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

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

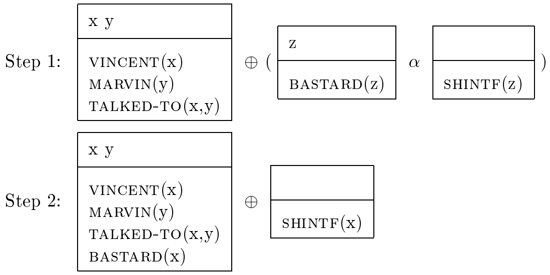
Resolution Example

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



Partial Match Example

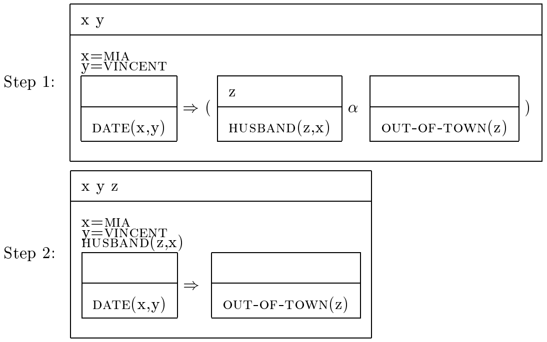
Vincent talked to Marvin. The bastard shot him in the face.



Note: more solutions

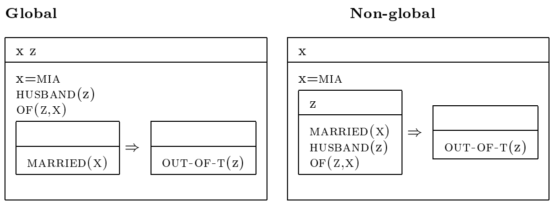
Accommodation Example

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



Non-global Accommodation

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



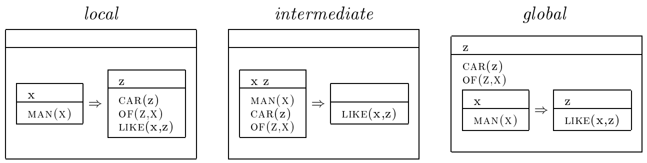
The Resolution Algorithm

1. Generate a DRS for the input sentence with all elementary presuppositions given as an α-DRS.
2. Merge this DRS with the DRS of the discourse so far processed.
3. Traverse the DRS, and on encountering an α-DRS try to
   (a) link the presupposed information to an accessible antecedent;
   (b) or accommodate the information to a superordinated DRS.
4. **Remove those DRSs from the set of potential readings that violate the acceptability constraints.**

Free Variable Check

Example: 'Every man likes his car'

Readings:



Other Acceptability Constraints

Here matters get a lot more interesting (and a lot more difficult):

1. We need to obey the *consistency* constraint;
2. We should obey the *informativity* constraint;
3. We also should obey *local* informativity and consistency.

⇒ We have to do some real work! (requires *inference*)

Presupposition Projection in Prolog

Extending the *Lexical Macros*: determiners

'the':

```
detSem(def,
       lambda(P,
          lambda(Q,
             alfa(X,def,merge(drs([X],[]),P@X),Q@X)))).
```

'a':

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


Other Lexical Macros

Proper names:

```
pnSem(Sym,Gender,
      lambda(P,alfa(X,name,drs([X],[X=Sym,Cond]),P@X))):-
             compose(Cond,Gender,[X]).
```

Pronouns:

```
proSem(Gender,Type,
       lambda(P,alfa(X,Type,drs([X],[Cond]),P@X))):-
              compose(Cond,Gender,[X]).
```


Implementing Linking and Accommodation: DRS

```
projectDrs([merge(B1,B2)|A1]-[drs(D3,C3)|A3]):-
        projectDrs([B1|A1]-A2),
        projectDrs([B2|A2]-[drs(D1,C1),drs(D2,C2)|A3]),
        append(D1,D2,D3),
        append(C1,C2,C3).

projectDrs([alfa(X,Type,B1,B3)|A1]-A5):-
        projectDrs([B1|A1]-[B2|A2]),
        resolveAlfa(X,B2,A2-A3),
        projectDrs([B3|A3]-A4),
        addBindingInfo(X,Type,A4-A5).

projectDrs([drs(D1,C1)|A1]-A2):-
        projectConds(C1,[drs(D1,[])|A1]-A2).
```


Implementing Linking and Accommodation: DRS-conditions

```
projectConds([~B1|Conds],A1-A3):-
   projectDrs([B1,pre([])|A1]-[B2,pre(Pres),drs(D,C)|A2]),
   accommodate(Pres,B2-B3),
   projectConds(Conds,[drs(D,[~B3|C])|A2]-A3).

projectConds([B1 > B2|Conds],A1-A4):-
   projectDrs([B1,pre([])|A1]-A2),
   projectDrs([B2,pre([])|A2]-[B4,pre(P2),B3,pre(P1),drs(D,C)|A3]),
   accommodate(P1,B3-B5),
   accommodate(P2,B4-B6),
   projectConds(Conds,[drs(D,[B5 > B6|C])|A3]-A4).

projectConds([],A-A).
```

Implementing Accommodation

```
accommodate([],B-B).

accommodate([drs(D1,C1)|Presups],drs(D2,C2)-B):-
        append(D1,D2,D3),
        append(C1,C2,C3),
        accommodate(Presups,drs(D3,C3)-B).
```


Implementing Resolution

```
resolveAlfa(X,AlfaDrs,[drs(D,C)|Others]-[New|Others]):-
        matchDrs(X,AlfaDrs,drs(D,C),New).

resolveAlfa(X,AlfaDrs,[AnteDrs|Others]-[AnteDrs|NewOthers]):-
        resolveAlfa(X,AlfaDrs,Others-NewOthers).

resolveAlfa(_,AlfaDrs,[pre(A)|Others]-[pre([AlfaDrs|A])|Others]).

resolveAlfa(X,Alfa,[pre(A1)|Others]-[pre([New|A2])|Others]):-
        select(Drs,A1,A2),
        matchDrs(X,Alfa,Drs,New).
```


DRS Matching

```
matchDrs(X,drs(D1,C1),drs(D2,C2),drs(D3,C3)):-
   member(X,D2),
   mergeLists(D1,D2,D3),
   mergeLists(C1,C2,C3),
   consistentConditions(X,Conds).

consistentConditions(X,Conds):-
   \+ (
          member(Cond1,Conds),
          member(Cond2,Conds), \+ Cond1=Cond2,
          compose(Cond1,Symbol1,[Y]), Y==X,
          compose(Cond2,Symbol2,[Z]), Z==X,
          \+ consistent(Symbol1,Symbol2)
      ).
```


Driver

```
parse:-
   readLine(Discourse),
   d(Drs1,Discourse,[]),
   betaConvert(Drs1,Drs2),
   findall(Drs4,(
                 projectDrs([Drs2,pre([])]-[Drs3,pre(P)]),
                 accommodate(P,Drs3-Drs4),
                 checkBinding(Drs4)
                ),
           Readings),
   printReadings(Readings).
```


Example

```
?- parse.

> A woman smokes. The woman collapses.

Readings:
1 drs([A],[collapse(A),smoke(A),woman(A)])
2 drs([A,B],[woman(A),collapse(A),smoke(B),woman(B)])
```


Putting it all together

- *Implementing the acceptability constraints imposed by Van der Sandt on presupposition accommodation.*
- *Describe and implement the local versions of these constraints that Van der Sandt also imposes.*
- *In essence:*
  Integrating semantic construction and inference!

The Acceptability Constraints

Be consistent:
Mia is a boxer. Vincent knows all boxers. He doesn't know Mia.

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

Be informative:
Jody is a boxer. Jody is a boxer.
Mia is married. She has a husband.

...this requires inference — what are we to do?

Exploiting First-Order Inference Tools

1. show *how* inference problems in DRT can be translated into inference problems in first-order logic
2. be precise about *what* the inference problems in DRT that van der Sandt's appeals to actually are

The Discourse Manager (Curt)

- we are going to build a small interactive discourse system
- user can build up a discourse by inputting a sentence to the system
- Use the DCG-parser and the DRS-construction program
- Integrate the result with the DRS from the previous discourse
- Presuppositions will be handled as discussed
- Use a theorem prover to check the maxims.
- Allow the user to ask what the DRS under consideration is
- The system will tell the user if one of the maxims is violated

Which theorem prover are we going to use?

1. theorem provers are not able to process DRSs directly
2. so we need to translate DRSs to first-order logic
3. DRS languages translate to first-order logic *with equality*
4. there are a number of freely available provers that handle equality.
5. Otter, Bliksem, etc.

The Prolog interface to the Theorem Prover Otter

```
callTheoremProver(Axioms,Formula,Proof):-
      fol2otter(Axioms,Formula),
      shell('./otter < temp.in > temp.out',X),
      (
       X=26368,
       Proof=yes
      ;
       X=26624,
       Proof=no
      ).
```


Implementing: `curtInput` and `curtOutput`

```
curtOutput(Input,Readings):-
   d(MergeDrs,Input,[]),!,betaConvert(MergeDrs,ReducedDrs),
   getReadings(ReducedDrs,Readings,PotentialReadings),
   consistentReadings(PotentialReadings,ConsistentReadings),
   (
       ConsistentReadings=[],
       curtSays('No! I do not believe that!'),!,
       curtInput(Readings)
   ;
       informativeReadings(ConsistentReadings,InformativeReadings),
       (
           InformativeReadings=[],
           curtSays('Yes, I knew that!'),!,
           curtInput(Readings)
       ;
           curtSays('Ok.'),
           curtInput(InformativeReadings)
       )
   ).
```

Checking for *consistency*

1. Translate the DRS to first-order logic: call this formula Φ
2. if $\neg\Phi$ is a theorem, then Φ is inconsistent
3. Most provers are refutation based:
   we have to give it the *negation* of $\neg\Phi$
4. Incorporate Background Knowledge

```
consistent(NewDrs):-
   backgroundKnowledge(Chi),
   drs2fol(NewDrs,Phi),
   callTheoremProver(Chi,Phi,Proof),
   (Proof=yes, !, fail; true).
```


Checking for *informativity*

1. Translate the old DRS to Φ, and the new DRS to Ψ
2. If $\Phi \rightarrow \Psi$ is a theorem then we have no new information
3. Refutation based proof: give $\neg(\Phi \rightarrow \Psi)$ to prover
4. Incorporate Background Knowledge

```
informative(drs([],[]),_):-!.
informative(OldDrs,NewDrs):-
   backgroundKnowledge(Chi),
   drs2fol(OldDrs,Phi),
   drs2fol(NewDrs,Psi),
   callTheoremProver(Chi,~(Phi > Psi),Proof),
   (Proof=yes, !, fail; true).
```


Background Knowledge

```
backgroundKnowledge(Formulas):-
   knowledge(Formulas1),
   generateOntology(Formulas2),
   append(Formulas1,Formulas2,Formulas).

knowledge(
  [
   forall(X,forall(Y,have(X,Y)>of(Y,X))),
   forall(X,forall(Y,of(Y,X)>have(X,Y))),
   forall(X,female(X)&married(X)>exists(Y,husband(Y)&have(X,Y))),
   forall(X,forall(Y,husband(Y)&have(X,Y)>married(X)&female(X)))
  ]).
```


A chat with Curt: `?- curt.`

```
> Marsellus loves Butch and Mia.
Ok.

> Mia is a boxer.
Ok.

> Butch is a boxer.
Ok.

> Marsellus loves one boxer.
No! I do not believe that!

> bye
Bye bye!
```


Local Informativity

Van der Sandt (1992):

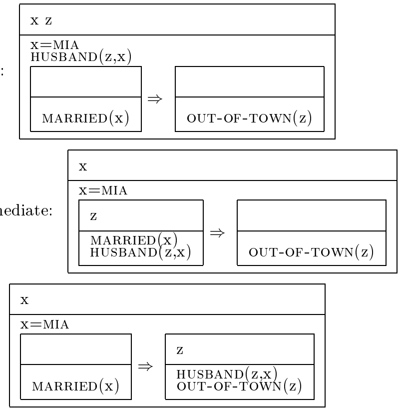
superordinated DRSs should neither imply a subordinated DRS, nor a negated subordinated DRS

This constraint rules global accommodation of Mia's husband in:

> If Mia is married then her husband is out of town.
> Either Mia is not married, or her husband is out of town.

! Action: Extend our discourse system to deal with this.

global:
intermediate:
local:



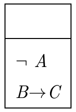
Redefining the Local Constraints

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 (- denotes set subtraction):

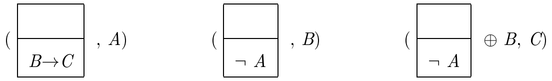
1. ⟨<U,C-c>, B⟩ if c∈C and c is of the form B⇒B′, B∨B′, B′∨B or ¬B.
2. ⟨<U,C-c>⊕B, B′⟩ if c∈C and c is of the form B⇒B′.
3. ⟨<U,C-c>⊕B_1, B_2⟩ if c∈C and c is of the form B⇒B′, B∨B′, B′∨B or ¬B, and ⟨B_1,B_2⟩∈**SuperSub**(B).
4. ⟨<U,C-c>⊕B⊕B_1, B_2⟩ if c∈C and c is of the form B⇒B′, and ⟨B_1,B_2⟩∈**SuperSub**(B′).

Calculating pairs of super-sub DRSs

For a DRS



the pairs of super-sub DRSs are:



Implementing the Local Constraints

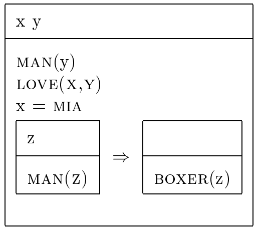
```
localInformative(Drs):-
   findall((Super,Sub),superSubDrs(Drs,drs([],[])-Super,Sub),List),
   allLocalInformative(List).

allLocalInformative([]).
allLocalInformative([(Super,Sub)|Others]):-
   backgroundKnowledge(Chi),
   drs2fol(drs([],[Super>Sub]),Phi),
   callTheoremProver(Chi,~Phi,Proof),
   (Proof=yes, !, fail; allLocalInformative(Others)).
```

Take care of binding!

Are the local constraints too strong?

- The local informativity constraint is useful: It allows blocking of global accommodation
- Sometimes local informativity seems too strong.

Mia loves a man. Every man is a boxer.



Consistency checking for theorem provers and model builders

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

The MathWeb System

- competitive ATP: calls sets of ATP concurrently as competing services
- ATP services: Bliksem, EQP, Otter, Protein, Spass, Waldmeister, Satchmo.
- calls are distributed over a network of workstations
- at present: 15-20 machines on four local networks

The Doris System

- Hooked up to MathWeb, using Bliksem, Spass, and Otter.
- `http://www.coli.uni-sb.de/~bos/atp/doris.html`