Representation and Inference for Natural Language

A First Course in
Computational Semantics

By Patrick Blackburn & Johan Bos

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?*

Thus course introduces some fundamental techniques for tackling these questions.

Outline

Monday: First-order logic + model checking

Tuesday: The lambda calculus + quantifier raising

Wednesday: Storage methods + propositional inference

Thursday: First-order inference + putting it all together

Friday: Discourse Representation Theory

The lectures follow the course notes closely: it's important that you read them.

Please feel free to approach us at any time with questions!

Today:

- we review the syntax and semantics of first order logic
- we write a simple first-order model checker in

PROLOG

Vocabularies

Here is an example:

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

This vocabulary tells us two important things:

- the topic of conversation
- the language the conversation is going to be conducted in

First-Order Models

A model is a pair (D, F)

- it contains the set of entities we want to talk about (the domain D)
- for each symbol in the vocabulary, it gives (via the *interpretation function* F), an appropriate entity built from items in D

Simple Model Example

- Let D be $\{d_1, d_2, d_3, d_4\}$ ('universe of discourse')
- Specify an interpretation function F

$$F(\textsc{mia}) = d_1$$
$$F(\textsc{honey-bunny}) = d_2$$
$$F(\textsc{vincent}) = d_3$$
$$F(\textsc{pumpkin}) = d_4$$
$$F(\textsc{customer}) = \{d_1, d_3\}$$
$$F(\textsc{robber}) = \{d_2, d_4\}$$
$$F(\textsc{love}) = \{(d_4, d_2), (d_3, d_1)\}$$

This is a *well-named* model.

Another Model

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

$F_2(\textsc{mia}) = d_2$
$F_2(\textsc{honey-bunny}) = d_1$
$F_2(\textsc{vincent}) = d_4$
$F_2(\textsc{pumpkin}) = d_3$
$F_2(\textsc{customer}) = \{d_1, d_2, d_4\}$
$F_2(\textsc{robber}) = \{d_3\}$
$F_2(\textsc{love}) = \{(d_3, d_4)\}$

In this model, three of the individuals are customers, and only one is a robber. Once again, this is a well-named model.

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.

Atomic Formulas

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

> If R is a relation symbol of arity n, and τ_1, ..., τ_n are terms, then $r(\tau_1, \cdots, \tau_n)$ is an atomic (or basic) formula.

Well Formed Formulas

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

Free and Bound Variables

Example:

$$\neg\ (\textsc{customer}(x) \lor \forall x(\textsc{robber}(x) \land \forall y\ \textsc{person}(y)))$$

- 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

Sentences

- 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, ∀x ROBBER(x) *is a sentence, but* ROBBER(x) *is not*.

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

Variable Assignment Functions

- Model $\mathbf{M} = (D, F)$
- g is an assignment of values to variables in $\mathbf{M}$
- g is a function from the set of variables to D

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 $I^g(\tau)$.

The Satisfaction Definition

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

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 $\mathbf{M}, g \models \phi$.

Valid Sentences and Valid Arguments

- A *valid sentence* is a sentence that is true in *all* models (for example: $\textsc{robber}(\textsc{mia}) \vee \neg\textsc{robber}(\textsc{mia})$)
- Suppose $\phi_1, \ldots, \phi_n$, and ψ are a finite collection of first-order sentences. Then we say that the argument with *premisses* $\phi_1, \ldots, \phi_n$ and *conclusion* ψ is a *valid argument* if and only if whenever all the premisses are true in some model, the conclusion is true in that model also.
- Note that both definitions are *semantic*.

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:

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.

Interpreting function symbols

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

- $I^g(\tau)$ is $F(\tau)$ if τ is a constant (just as before)
- $I^g(\tau)$ 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 $I^g(\tau)$ is $F(f)(I^g(\tau_1), \ldots, I^g(\tau_n))$.

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

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

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

What is PROLOG?

The two key features of PROLOG

- its *declarative* nature
- its extensive use of *unification*

In PROLOG, we simply state the nature of the problem, and let the PROLOG unification-driven inference engine search for a solution.

Basics of Prolog

Three basic constructs:

- *facts*
- *rules*
- *queries*

A set of facts and rules is all a Prolog program consists of!

Facts

Facts are used to state things that are unconditionally true of the domain of interest.

```
woman(mia).
woman(jody).
```

Rules

Rules relate facts by logical implications.

```
plays_air_guitar(mia):-
   listens_to_music(mia).
```

Prolog Terminology

- the *head* of a rule is the part on the left of the `:-`
- the *body* of a rule is the part on the right of the `:-`
- facts and rules contained in a knowledge base are called *clauses*
- the collection of all clauses with the same head is called a *predicate*

Queries

Posing queries makes the Prolog inference engine try to deduce a positive answer from the information contained in the knowledge base.

There are two basic circumstances under which Prolog can return a positive answer:

- the question posed is a fact
- Prolog can deduce the positive answer by using the deduction rule called modus ponens

That is, if `head :- body` and `body` are both in the knowledge base, then PROLOG can deduce that `head` is true.

An Example Query

We can ask PROLOG whether Mia is a woman:

```
?- woman(mia).
```

We can ask whether Mia plays air-guitar:

```
?- plays_air_guitar(mia).
```

Conjunction in Prolog

The comma , expresses logical conjunction in PROLOG

```
plays_air_guitar(mia):-
   listens_to_music(mia),
   happy(mia).
```

Mia plays air-guitar if she listens to music, *and* if she is happy.

Disjunction in Prolog

The semi-colon ; expresses disjunction in PROLOG

```
plays_air_guitar(mia):-
   listens_to_music(mia);
   happy(mia).
```

Mia plays air-guitar if she listens to music, *or* if she is happy.

Prolog Terms

There are three kinds of terms in PROLOG:

- atoms
- variables
- complex terms

An atom is a sequence of characters starting with a *lowercase* character.

A variable is a sequence of characters starting with an *uppercase* or an underscore.

Complex Terms

- Complex terms are build out of a *functor* and a sequence of *arguments*.
- The arguments are put in ordinary brackets, separated by commas, and placed after the functor.
- The functor *must* be an atom
  (variables *cannot* be used as functors)
- Arguments can be any kind of term

The number of arguments that a complex term has is called its *arity*.

Variables

"every woman likes a foot massage"

```
like_foot_massage(X):-
    woman(X).
```

This can be read as: `X` likes a foot massage, if `X` is a woman.

- Variables have no fixed values
- Variables can be instantiated with a value by *unification*

Unification

Pose the query: `?- like_foot_massage(mia).`

Variable `X` is *unified* with the atom `mia`.

Two terms are unifiable if:

- they are the same atoms
- if one of them is a variable, or
- if they are both complex terms with the same functor name and arity, and all corresponding arguments unify

Unification makes terms identical.

It is the heart of the engine that drives Prolog.

Recursion

Predicates can be defined recursively

```
more_expensive(X,Y):-
    costs_a_little_more(X,Y).

more_expensive(X,Y):-
    costs_a_little_more(X,Z),
    more_expensive(Z,Y).
```

A recursive predicate should always have at least two clauses:

- a non-recursive one
- and one that contains the recursion

Lists

Lists are recursive data structures:

1. the empty list is a list.
2. a complex term is a list if it consists of two items: the first a term and the second a list

Examples:

```
[butch|[pumpkin|[marsellus|[]]]]

[butch,pumpkin,marsellus]

[vincent,[honey_bunny,pumpkin],[marsellus,mia]]
```

A simple model checker

We will implement a simple first-order model checker in PROLOG

- define PROLOG representation for Vocabularies
- define PROLOG representation for Models
- define PROLOG representation for Formulas
- evaluating (representations of) Formulas in (representations of) Models

Representing Vocabularies

Vocabularies will be represented as PROLOG databases

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

This would be represented by the following database:

```
relation(love,2).              constant(mia).
relation(hate,2).              constant(vincent).
relation(customer,1).          constant(honey_bunny).
relation(robber,1).            constant(pumpkin).
```

Representing Models

Suppose: we have fixed our vocabulary.

How shall we represent models of this vocabulary in PROLOG?

```
[customer(vincent),customer(mia),
 robber(honey_bunny),robber(pumpkin),
 loves(pumpkin,honey_bunny)].
```

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(40,yfx,>).        % implication
:- op(30,yfx,v).        % disjunction
:- op(30,yfx,&).        % conjunction
:- op(20, 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)
```

Semantic Evaluation

- Final task: evaluating formulas in models.
- The predicate which carries out the task is `sem/2`.

The clause which evaluates (representations of) atomic formulas is:

```
sem(Formula,Model):-
    member(Formula,Model).
```

Now for the clauses for Boolean combinations of formulas.

```
sem(Formula1 & Formula2,Model):-
   sem(Formula1,Model),
   sem(Formula2,Model).

sem(Formula1 v Formula2,Model):-
   sem(Formula1,Model);
   sem(Formula2,Model).

sem(Formula1 > Formula2,Model):-
   sem(Formula2,Model);
   \+ sem(Formula1,Model).

sem(~ Formula,Model) :-
   \+ sem(Formula,Model).
```

These mirror the first-order satisfaction definition in an obvious way.

And finally, the clauses for quantifiers.

The existential quantifier clause:

```
sem(exists(X,Formula),Model):-
   constant(X),
   sem(Formula,Model).
```

The universal quantifier clause:

```
sem(forall(X,Formula),Model):-
   sem(~ exists(X,~ Formula),Model).
```

Checking Models

Some example models:

```
example(1,[customer(mia),customer(vincent),
           robber(pumpkin),robber(honey_bunny),
           love(pumpkin,honey_bunny)]).


example(2,[customer(mia),
           robber(pumpkin),robber(honey_bunny),
           love(pumpkin,honey_bunny)]).
```

A driver predicate:

```
evaluate(Formula,Example):-
   example(Example,Model),sem(Formula,Model).
```

First Problem

Suppose we query:

```
?- evaluate(customer(X),1).
```

(we're asking whether it is possible to assign a value to the free variable that satisfies the formula)

Now suppose we ask:

```
?- evaluate(~ customer(X),1).
```

In example 1, there are two non-customers, namely Honey Bunny and Pumpkin.

However, PROLOG wrongly answers `no`!

Second problem

Suppose we try evaluating the atomic formula

```
tasty(royale_with_cheese)
```

in any of example models. Then our model checker will say `no`.

Worse is in store:

```
~ tasty(royale_with_cheese)
```

in any of example models our model checker will return the answer `yes`!

The `wff/1` Predicate

```
wff(forall(X,Formula)):-
   var(X),wff(Formula).

wff(exists(X,Formula)):-
   var(X),wff(Formula).

wff(Formula1 > Formula2):-
   wff(Formula1),wff(Formula2).

wff(Formula1 & Formula2):-
   wff(Formula1),wff(Formula2).

wff(~ Formula):- wff(Formula).
```

The real work (I)

```
wff(Formula):-
   functor(Formula,Symbol,Arity),
   relation(Symbol,Arity),
   good_arguments(Arity,Formula).
```

The real work (II)

```
good_arguments(0,_).
good_arguments(N,Formula):-
   arg(N,Formula,Arg),
   first_order_term(Arg),
   M is N - 1,
   good_arguments(M,Formula).

first_order_term(X):- var(X).
first_order_term(X):- constant(X).
```

A New Driver

```
evaluate(Formula,Example):-
   wff(Formula),
   example(Example,Model),
   sem(Formula,Model).

evaluate(Formula,_):-
    \+ wff(Formula),
    nl,write('Not a wff over the given vocabulary.'),
    nl,write('Cannot be evaluated.'),nl.
```

A Third Problem!

In first-order logic the following two formulas

$$\exists x \, (\textsc{customer}(x) \wedge \exists y \, \textsc{robbr}(y))$$

$$\exists x \, (\textsc{customer}(x) \wedge \exists x \, \textsc{robbr}(x))$$

are equivalent.

However

```
?- evaluate(exists(X,custmr(X) & exists(Y,robber(Y))),1).
```

succeeds while

```
?- evaluate(exists(X,custmr(X) & exists(X,robber(X))),1).
```

returns `no`.

Conclusion

We have learned two principal lessons:

- it is very easy to implement a model checker by making use of Prolog variables to represent first-order variables!
- this simple implementation method also has its dangers!
(our 'obvious' implementation is not completely correct)

Today:

We begin the study of computational semantics proper:

How can we automate the process of associating semantic representations with expressions of natural language?

- Today's work centers around the *lambda calculus* and its implementation in Prolog. The lambda calculus is the fundamental tool for semantic construction used in this course.
- We also examine the problems raised by quantifier scope ambiguities. Today's approach proves too naive, but we isolate important ideas which will be used when we return to the topic tomorrow.

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

Vincent likes Mia (S)

LIKE(VINCENT,MIA)

Vincent (NP)

VINCENT

likes Mia (VP)

LIKE(?,MIA)

likes (TV)

LIKE(?,?)

Mia (NP)

MIA

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.
np --> pn.
np --> det, noun.
pn --> [vincent].
pn --> [mia].
det --> [a].
det --> [every].
```

```
noun --> [woman].
noun --> [foot,massage].
vp --> iv.
vp --> tv, np.
iv --> [snorts].
tv --> [loves].
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*

An Experiment

To build representations we need to

- work with 'incomplete' formulas
- indicate where the information they lack must go

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,Sem) --> det(X,Rest,Scope,Sem), 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

- 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:

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

The prefix λx binds the occurrence of x in $\textsc{man}(\mathrm{x})$.

Functional application

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

Example:

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

Lambda conversion

The required substitution is performed by lambda conversion:

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

Marking more complex kinds of information

Our semantic representation of a woman will be:

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

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

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

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

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)

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

every (DET)

$$\lambda P.\lambda Q.\forall x(P(x){\rightarrow}Q(x))$$

boxer (NOUN)

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

Lambda Conversion

Applications are instructions to carry out lambda 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 lambda conversion:

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

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)

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

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

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: $\lambda P.P(\textsc{mia})$

vincent: $\lambda P.P(\textsc{vincent})$

These representations *can* be used as functors.

Proper Names in Action

Vincent loves Mia (S)

$\textsc{love}(\textsc{vincent},\textsc{mia})$

Vincent (NP)

$\lambda P.P(\textsc{vincent})$

loves Mia (VP)

$\lambda z.(\textsc{love}(z,\textsc{mia}))$

loves (TV)

$\lambda X.\lambda z.(X(\lambda x.\textsc{love}(z,x)))$

Mia (NP)

$\lambda P.P(\textsc{mia})$

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.

The Lambda Calculus in Prolog

Step 1: *Represent lambda expressions*

```
lambda(L,F)
```

Step 2: *Define lambda conversion*

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

This can be simplified to:

```
fa(lambda(Argument,Result),Argument,Result).
```

Does this Work?

Suppose we wanted to apply $\lambda x.\textsc{growl}(x)$ to BUTCH.

We would do this by means of the following query:

```
?- fa(lambda(X,growl(X)),butch,Result).

Result = growl(butch)
```

The required substitutions are automatically carried out by PROLOG'S *inbuilt unification mechanism!*

DCGs and λ

Note: `lambda` and `fa` are all we need to build semantic representations!

Add the `fa/3` goals to the original DCG rules:

```
s(S)   --> np(NP), vp(VP), {fa(NP,VP,S)}.
np(NP) --> pn(NP).
np(NP) --> det(DET), noun(NOUN), {fa(DET,NOUN,NP)}.
vp(VP) --> iv(VP).
vp(VP) --> tv(TV), np(NP), {fa(TV,NP,VP)}.
```

- unary branching rules percolate up their semantic representation
- binary branching rules use `fa/3` 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,PN)) --> [vincent],{fa(P,vincent,PN)}.
pn(lambda(P,PN)) --> [mia],{fa(P,mia,PN)}.
```

Determiners

a: λP.λQ.$\exists$x.(P(x)$\wedge$Q(x))

```
det(lambda(P,lambda(Q,exists(X,R & S)))) -->
                [a], {fa(P,X,R),fa(Q,X,S)}.
```

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

```
det(lambda(P,lambda(Q,forall(X,R > S)))) -->
                [every], {fa(P,X,R),fa(Q,X,S)}.
```

Eliminating `fa/3` Goals

Old rule for NPs:

```
s(S) --> np(NP), vp(VP), {fa(NP,VP,S)}.
```

Rule without `fa/3`:

```
s(S) --> np(lambda(VP,S)), vp(VP).
```

Quantifying Noun Phrases

We can assign semantic representations to sentences with transitive verbs and two quantifying noun phrases.

every boxer loves a woman (S)

$$\forall x(\textsc{boxer}(x) \rightarrow \exists y(\textsc{woman}(y) \wedge \textsc{love}(x,y)))$$

every boxer (NP)

$$\lambda P.\forall x(\textsc{boxer}(x) \rightarrow P(x))$$

loves a woman (VP)

$$\lambda z.\exists y(\textsc{woman}(y) \wedge \textsc{love}(z,y))$$

loves (TV)

$$\lambda X.\lambda z.X(\lambda x.\textsc{love}(z,x))$$

a woman (NP)

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

Scope Ambiguities

- The first-order formula constructed for this parse tree states that for each boxer there is a woman that he loves.
- However, every boxer loves a woman has a second meaning

Second reading:

$$\exists y(\textsc{woman}(y) \wedge \forall x(\textsc{boxer}(x) \rightarrow \textsc{love}(x,y)))$$

(There is *one* woman who is loved by *all* boxers)

Montague's Approach (I)

every boxer loves her-3 (S)

$\forall x(\textsc{boxer}(x) \rightarrow \textsc{love}(x, z_3))$

every boxer (NP)

$\lambda P.\forall x(\textsc{boxer}(x) \rightarrow P(x))$

loves her-3 (VP)

$\lambda y.\textsc{love}(y, z_3)$

loves (TV)

$\lambda X.\lambda y.X(\lambda x.\textsc{love}(y, x))$

her-3 (NP)

$\lambda P.P(z_3)$

Montague's Approach (II)

every boxer loves a woman (S)

$$\exists y(\textsc{woman}(y) \wedge \forall x(\textsc{boxer}(x) \rightarrow \textsc{love}(x,y)))$$

a woman (NP)

$$\lambda P.\exists y(\textsc{woman}(y) \wedge P(y))$$

every boxer loves her-3 (S, 3)

$$\lambda z_3.\forall x(\textsc{boxer}(x) \rightarrow \textsc{love}(x,z_3))$$

Montague's Trick in DCGs (I)

```
np([bo(Sem,I)],lambda(P,R)) -->
                det(SemDet), noun(SemNoun),
                {fa(SemDet,SemNoun,Sem),fa(P,I,R)}.
```

- `Sem` is the semantics of the quantifying noun phrase, `I` is the Index, and *indexed binding operators* are represented as `bo(Sem,Index)`.
- `lambda(P,R)` and `fa(P,I,R)` mimic the pronoun semantics.
- the first argument of `np` holds the list of raised quantifiers, together with their associated indexes.

Montague's Trick in DCGs (II)

Extending the rules for Noun Phrases

```
np([],Sem) --> det(SemDet), noun(SemNoun),
               {fa(SemDet,SemNoun,Sem)}.

np([],Sem) --> pn(Sem).
```

Extending the rules for Verb Phrases

```
vp([],Sem) --> iv(Sem).
vp(Q,SemVP) --> tv(SemTV), np(Q,SemNP),
                     {fa(SemTV,SemNP,SemVP)}.
```

Montague's Trick in DCGs (III)

Rule 1: Replace the placeholder pronouns by the real quantifiers

```
s([],Sem) --> s([(SemNP,I)],SemS),
              {fa(SemNP,lambda(I,SemS),Sem)}.
```

Rule 2: The standard rule for forming sentences

```
s(Q,Sem) --> np([],SemNp), vp(Q,SemVp),
             {fa(SemNp,SemVp,Sem)}.
```

Time to test our program

Prolog Query:

```
?- s([],Sem,[every,boxer,loves,a,woman],[]).

Sem = exists(X,woman(X)&forall(Y,boxer(Y)>love(Y,X))) ;

Sem = forall(Y,boxer(Y)>exists(X,woman(X)&love(Y,X)))
```

It gives us exactly the two readings that we want!

What can we say about Montague's Approach?

- places heavy burden on the grammar rules!
- the obvious implementation is not very attractive

New strategy:

- abandon quantifier raising
- try storage!

Today:

- We introduce *Cooper storage* and *nested Cooper storage*, which are more sophisticated approaches to quantifier scope ambiguities than Montague's, and show how to implement them in PROLOG.
- We then start discussing how to automate *inferencing*. Today we'll discuss *propositional tableaux* and show how to implement them in PROLOG

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 the store contains

More precisely:

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

Time for an example...

every boxer loves a woman (S)

$\langle \textsc{love}(z_6,z_7),\ (\lambda P.\forall x(\textsc{boxer}(x)\rightarrow P(x),6),\ (\lambda P.\exists y(\textsc{woman}(y)\wedge P(y)),7)\rangle$

every boxer (NP)

$\langle \lambda Q.Q(z_6),\ (\lambda P.\forall x(\textsc{boxer}(x)\rightarrow P(x)),6)\rangle$

loves a woman (VP)

$\langle \lambda u.\textsc{love}(u,z_7),\ (\lambda P.\exists y(\textsc{woman}(y)\wedge P(y)),7)\rangle$

loves (TV)

$\langle \lambda X.\lambda u.X(\lambda v.\textsc{love}(u,v))\rangle$

a woman (NP)

$\langle \lambda Q.Q(z_7),\ (\lambda P.\exists y(\textsc{woman}(y)\wedge P(y)),7)\rangle$

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**
> 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 $\langle \beta(\lambda \mathrm{z}_i.\phi), \sigma_1\sigma_2 \rangle$ is also associated with this expression.

This is Montague's trick with the aid of stores!

Back to our example...

$$\langle \textsc{love}(z_6,z_7),\ (\lambda P.\forall x(\textsc{boxer}(x)\rightarrow P(x),6),(\lambda P.\exists y(\textsc{woman}(y)\wedge P(y)),7)\rangle$$

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

$$\langle \lambda P.\forall x(\textsc{boxer}(x)\rightarrow P(x)(\lambda z_6.\textsc{love}(z_6,z_7)),\ (\lambda P.\exists y(\textsc{woman}(y)\wedge P(y)),7)\rangle$$

Using lambda conversion, this simplifies to:

$$\langle \forall x(\textsc{boxer}(x)\rightarrow \textsc{love}(x,z_7),\ (\lambda P.\exists y(\textsc{woman}(y)\wedge P(y)),7)\rangle$$

Retrieving the quantifier left in store:

$$\langle \lambda P.\exists y(\textsc{woman}(y)\wedge P(y))(\lambda z_7.\forall x(\textsc{boxer}(x)\rightarrow \textsc{love}(x,z_7)))\rangle$$

Final result:

$$\langle \exists y(\textsc{woman}(y)\wedge \forall x(\textsc{boxer}(x)\rightarrow \textsc{love}(x,y))\rangle$$

Cooper storage in Prolog

The following Prolog list represents a store:

```
[walk(X),bo(lambda(lambda(Y,F),forall(Y,boxer(Y) > F)),X)]
```

Re-implement functional application:

```
fa([lambda(Arg,Result)|Store1],[Arg|Store2],[Result|Store]):-
     append(Store1,Store2,Store).
```

What about the storage rule?

```
np_storage([lambda(Arg,Res)],[Arg|S],[lambda(P,R),bo(Res,X)|S]):-
    fa([P],[X],[R]).
```

DCG Rules for Storage

Quantified noun phrases:

```
np(Sem) --> det(SemDet), noun(SemNoun),
            {np_storage(SemDet,SemNoun,Sem)}.
```

Warning: this DCG rule makes storage *compulsory*!

Alternative:

```
np(Sem) --> det(SemDet), noun(SemNoun),
              {
                np_storage(SemDet,SemNoun,Sem)
                ;
                fa(SemDet,SemNoun,Sem)
              }.
```

A Sample Query

Our little program does what we want:

```
?- s(Sem,[every,boxer,loves,a,woman],[]).

Sem = [love(Y,X),
       (lambda(lambda(A,P),forall(A,boxer(A)>P)),Y),
       (lambda(lambda(B,Q),exists(B,woman(B)&Q)),X)
      ] ;

no
```

All that remains is to implement the retrieval rule.

Implementing Retrieval

- Recursive predicate `s_retrieval/2`
- First argument is the store
- Second argument is the derived scoped representation.

```
s_retrieval([S],S).
s_retrieval([Sem|Store],S):-
   remove_from_store(bo(Q,X),Store,NewStore),
   fa([Q],[lambda(X,Sem)],[Result]),
   s_retrieval([Result|NewStore],S).
```

Putting Retrieval in the DCG

We make use of the retrieval in the DCG rule for forming sentences:

```
s(Sem) --> np(SemNp), vp(SemVp),
           {fa(SemNp,SemVp,SemS),s_retrieval(SemS,Sem)}.
```

Time to test our program:

```
?- s([Sem,[every,boxer,loves,a,woman],[]).

Sem = exists(X,woman(X)&forall(Y,boxer(Y)>love(Y,X))) ;

Sem = forall(Y,boxer(Y)>exists(X,woman(X)&love(Y,X))) ;

no
```

Summing up: Cooper storage

- Cooper Storage is a generalization of Montague's trick
- Advantage: we don't need to to make use of additional grammar rules
- Define two additional predicates: `np_storage` and `s_retrieval`

The Cooper storage technique is indeed a useful one!

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 and Montague's trick.

Is this really safe?

Every piercing that is done with a gun goes behind the entire idea of piercing

Mia told a joke in every episode

Nested Noun Phrases: a problem...

Store:

$\langle \textsc{told}(\textsc{mia},z_2),(\lambda P.\exists y(\textsc{joke}(y)\wedge\textsc{in}(y,z_1)\wedge P(y)),2),(\lambda Q.\forall x(\textsc{episode}(x)\rightarrow Q(x)),1)\rangle$

Option 1:

$$\langle \exists y(\textsc{joke}(y)\wedge\textsc{in}(y,z_1)\wedge\textsc{told}(\textsc{mia},y)),\ (\lambda Q.\forall x(\textsc{episode}(x)\rightarrow Q(x)),1)\rangle$$

$$\downarrow$$

$$\langle \forall x(\textsc{episode}(x)\rightarrow \exists y(\textsc{joke}(y)\wedge\textsc{in}(y,x)\wedge\textsc{told}(\textsc{mia},y)))\rangle$$

Option 2:

$$\langle \forall x(\textsc{episode}(x)\rightarrow\textsc{told}(\textsc{mia},z_2)),\ (\lambda P.\exists y(\textsc{joke}(y)\wedge\textsc{in}(y,z_1)\wedge P(y)),2)\rangle$$

$$\downarrow$$

$$\langle \exists y(\textsc{joke}(y)\wedge\textsc{of}(y,z_1)\wedge \forall x(\textsc{episode}(x)\rightarrow\textsc{told}(\textsc{mia},y)))\rangle$$

An elegant solution: nested stores

- 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
If the (nested) store $\langle\phi, \sigma\rangle$ is an interpretation for an NP, then the (nested) store $\langle\lambda\mathrm{P}.\mathrm{P}(\mathrm{z}_i),(\langle\phi,\sigma\rangle,i)\rangle$, for some unique index i, is also an interpretation for this NP.

a joke in every episode (NP)

$\langle \lambda P.P(z_2), (\langle \lambda P.\exists y(\textsc{joke}(y) \wedge \textsc{in}(y,z_1) \wedge P(y)), (\langle \lambda Q.\forall x(\textsc{epi}(x) \rightarrow Q(x))\rangle,1)\rangle,2)\rangle$

a (DET)

$\langle \lambda Q.\lambda P.\exists y(Q(y) \wedge P(y))\rangle$

joke in every episode (CN)

$\langle \lambda u.\textsc{joke}(u) \wedge \textsc{in}(u,z_1), (\langle \lambda Q.\forall x(\textsc{epi}(x) \rightarrow Q(x))\rangle,1)\rangle$

joke (CN)

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

in every episode (PP)

$\langle \lambda P.\lambda u.P(u) \wedge \textsc{in}(u,z_1), (\langle \lambda Q.\forall x(\textsc{epi}(x) \rightarrow Q(x))\rangle,1)\rangle$

Retrieval for Nested Storage

The new retrieval rule:

> **Retrieval**
> 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 $\langle \beta(\lambda \mathrm{z}_i . \phi), \sigma_1 \sigma \sigma_2 \rangle$ 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 told a joke in every episode (Reading 1)

Nested Store:

$\langle \textsc{told}(\textsc{mia},z_2),\ (\langle\lambda P.\exists y(\textsc{joke}(y)\wedge\textsc{in}(y,z_1)\wedge P(y)),\ (\langle\lambda Q.\forall x(\textsc{epi}(x)\rightarrow Q(x))\rangle,1)\rangle,2)\rangle$

There is only one way to perform retrieval:

$$\langle\forall x(\textsc{episode}(x)\rightarrow \exists y(\textsc{joke}(y)\wedge\textsc{in}(y,x)\wedge\textsc{told}(\textsc{mia},y)))\rangle$$

Mia told a joke in every episode (Reading 2)

Avoid storing the sub-NP every episode:

a joke in every episode (NP)

$\langle \lambda P.P(z_2), (\langle \lambda P.\exists y(\textsc{joke}(y) \wedge \forall x(\textsc{episode}(x) \rightarrow \textsc{in}(z,x)) \wedge P(y))\rangle, 2)\rangle$

a (DET)

$\langle \lambda Q.\lambda P.\exists x(Q(x) \wedge P(x))\rangle$

joke in every episode (CN)

$\langle \lambda z.\textsc{joke}(z) \wedge \forall x(\textsc{episode}(x) \rightarrow \textsc{in}(z,x))\rangle$

joke (CN)

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

in every episode (PP)

$\langle \lambda P.\lambda z.P(z) \wedge \forall x(\textsc{episode}(x) \rightarrow \textsc{in}(z,x))\rangle$

This leads to the following analysis for mia told a joke in every episode:

$$\langle \textsc{told}(\textsc{mia},z_2),(\langle\lambda P.\exists y(\textsc{joke}(y)\wedge\forall x(\textsc{episode}(x)\rightarrow\textsc{in}(y,x))\wedge P(y))\rangle,2)\rangle$$

There is only one operator the store.

Retrieving it yields:

$$\langle\exists y(\textsc{joke}(y)\wedge\forall x(\textsc{episode}(x)\rightarrow\textsc{in}(y,x))\wedge\textsc{told}(\textsc{mia},y))\rangle$$

This is the reading we want!

Nested Stores in Prolog

Nested Storage:

```
np_storage([lambda(Argument,Result)|Store1],
           [Argument|Store2],
           [lambda(P,R),bo([Result|Store],X)]):-
    append(Store1,Store2,Store),
    fa([P],[X],[R]).
```

The only other change is in the first clause of `remove_from_store/3`, to enable it to deal with lists of lists:

```
remove_from_store(bo(O,I),[bo([O|T1],I)|T2],T):-
    append(T1,T2,T).
```

A small piece of computational unpleasantness...

```
?- s(Sem,[mia,told,a,joke,in,every,episode],[]).

Sem = forall(Y,episode(Y)>exists(X,(joke(X)&in(X,Y))&told(mia,X))) ;

Sem = forall(Y,episode(Y)>exists(X,(joke(X)&in(X,Y))&told(mia,X))) ;

Sem = exists(X,(joke(X)&forall(Y,episode(Y)>in(X,Y)))&told(mia,X)) ;

Sem = exists(X,(joke(X)&forall(Y,episode(Y)>in(X,Y)))&told(mia,X)) ;

no
```

Here is a little trick to overcome this problem:

```
mysetof(X,Goal,Solutions):-
   bagof(X,Goal,Temp),
   remove_duplicates(Temp,Solutions).
```

Having added this code, we can pose a query using `mysetof/3` and obtain the list of results that we wish to have:

```
?- mysetof(Sem,s(Sem,[mia,told,a,joke,in,every,episode],[]),L).

L = [forall(Y,episode(Y)>exists(X,joke(X)&of(X,Y)&told(mia,X))),
     exists(U,joke(U)&forall(V,episode(V)>of(U,V))&told(mia,U))]
```

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.

Inference

How can we use the logical representations of natural language expressions to automate the process of drawing inferences?

- We need a *syntactic* handle on validity. We'll look to *proof theory* for some answers.
- Today we'll examine the *propositional* case. Tomorrow we'll cover full first-order languages.

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.

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 generalise 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
- 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...

1	$F(p \vee \neg p)$	$\surd$
2	Fp	$1, F_\vee$
3	$F\neg p$	$1, F_\vee$

- 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...

1	$F(p \vee \neg p)$	$\surd$
2	Fp	$1, F_\vee$
3	$F\neg p$	$1, F_\vee, \surd$
4	Tp	$3, F_\neg.$

- In line 3 we used expansion rule $F_\neg$ (*falsify a negation*)
- We're finished! The tableau is *rule saturated*.
- The tableau is also *closed*.

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...

1	$F\neg(q \wedge r) \to (\neg q \vee \neg r)$	$\surd$
2	$T\neg(q \wedge r)$	$1, F_{\to}$
3	$F(\neg q \vee \neg r)$	$1, F_{\to}, \surd$
4	$F\neg q$	$3, F_{\vee}$
5	$F\neg r$	$3, F_{\vee}$

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 $\quad Fq \quad 8, F_{\wedge}$ $\qquad\qquad$ 10 $\quad Fr \quad 8, F_{\wedge}$

Example 3

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

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

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

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	$F(r \vee q)$	$1, F_{\rightarrow}, \surd$
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.

Expansion rules for negation

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

- 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

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

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

$$\frac{F(\phi \to \psi)}{\begin{array}{c} T\phi \\ F\psi \end{array}} \qquad \frac{T(\phi \to \psi)}{F\phi \;\Big|\; T\psi}$$

Propositional tableaux

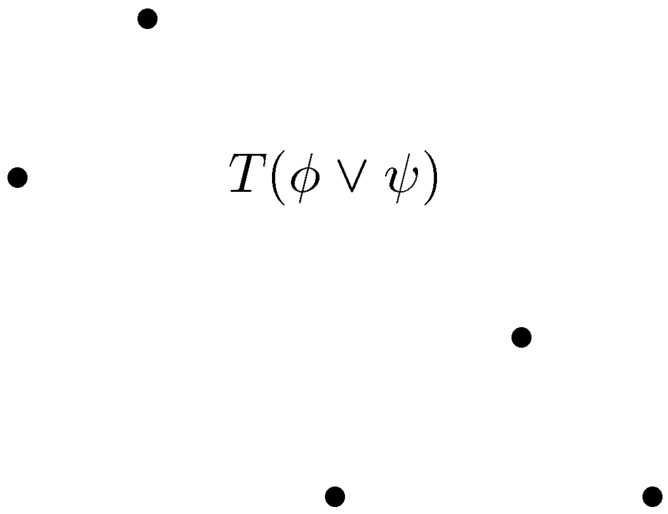
- A (propositional) *tableau* is a tree, each of whose nodes is a signed (propositional) formula. A *branch* of a tableau is a branch of such a tree.
- We start with an *initial tableau*. This can be *any* tableau that has only has a single branch.
- Tableaux expansion works as follows. Given a tableau, 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:

$T(\phi \vee \psi)$

$T\phi \qquad T\psi \qquad T\phi \qquad T\psi$

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 Arguments for Validity

Simply use the following initial tableau:

$$T\phi_1$$

$$\vdots$$

$$T\phi_n$$

$$F\psi$$

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.

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 $F(p \to q)$.

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 `saturate`, which recursively attempts to rule saturate an initial tableau.

```
saturate(Tableau):-
   closed(Tableau).

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

Testing for closure

```
closed([]).

closed([Branch|Rest]):-
    member(t(X),Branch),
    member(f(X),Branch),
    closed(Rest).
```

An organisational predicate

```
expand([Branch|Tableau],[NewBranch|Tableau]):-
   unary_expansion(Branch,NewBranch).

expand([Branch|Tableau],[NewBranch|Tableau]):-
   conjunctive_expansion(Branch,NewBranch).

expand([Branch|Tabl],[NewBranch1,NewBranch2|Tabl]):-
   disjunctive_expansion(Branch,NewBranch1,NewBranch2).

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

The real work

```
unary_expansion(Branch,[Component|Temp]):-
   unary(SignedFormula,Component),
   remove_first(SignedFormula,Branch,Temp).

conjunctive_expansion(Branch,[Comp1,Comp2|Temp]):-
   conjunctive(SignedFormula,Comp1,Comp2),
   remove_first(SignedFormula,Branch,Temp).

disjunctive_expansion(Branch,[Comp1|Tmp],[Comp2|Tmp]):-
   disjunctive(SignedFormula,Comp1,Comp2),
   remove_first(SignedFormula,Branch,Tmp).
```

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:

```
valid(F):-
    saturate([[f(F)]]).
```

How would you write a driver that tests *arguments* for validity?

Soundness and Completeness

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

  $$\text{if} \quad \vdash \phi \text{ then } \models \phi.$$

  Tableaux proofs will never lead us astray.

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

  $$\text{If} \quad \models \phi \quad \text{then} \quad \vdash \phi.$$

  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.

Termination and Efficiency

- The expansion process is guaranteed to halt on all possible input. Why is this?
- But is it truly efficient? The honest answer is: *nobody knows*. Nobody knows if there is *any* proof method which can prove all valid formulas efficiently.

Today:

- We discuss how to extend our propositional tableaux system to the full first-order language. Much of our discussion revolves around the concept of *unification*.
- We then 'glue together' the programs we have developed so far in these lectures: the model checker, the lambda implementation, the storage predicates, and the first-order theorem prover.

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:

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

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

So, from $T\forall\mathrm{x}\textsc{killer}(\mathrm{x})$ 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	$T\textsc{d}(\textsc{z})$	$2, 2_{\forall}$

6 $\quad F\textsc{d}(\textsc{m}) \quad 3, F_{\wedge}$ $\qquad$ 7 $\quad F\textsc{d}(\textsc{z}) \quad 3, F_{\wedge}$

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 $T\exists\mathrm{x}\phi$ or $F\forall\mathrm{x}\phi$? This is a more subtle matter.

- Suppose a tableau contains the signed formula $T\exists\mathrm{x}\textsc{killer}(\mathrm{x})$.
- 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 $T_\exists$:

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

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 $\exists x \forall y \textsc{shoots}(x,y) \rightarrow \forall y \exists x \textsc{shoots}(x,y)$ is valid. (This pretty example puts all four quantifier rules to work.)

1	$F(\exists x \forall y \textsc{shoots}(x,y) \rightarrow \forall y \exists x \textsc{shoots}(x,y))$	
2	$T\exists x \forall y \textsc{shoots}(x,y)$	1, $F_{\rightarrow}$
3	$F\forall y \exists x \textsc{shoots}(x,y)$	1, $F_{\rightarrow}$
4	$T\forall y \textsc{shoots}(c_1, y)$	2, $T_{\exists}$
5	$F\exists x \textsc{shoots}(x, c_2)$	3, $F_{\forall}$
6	$T\textsc{shoots}(c_1, c_2)$	4, $T_{\forall}$
7	$F\textsc{shoots}(c_1, c_2)$	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 a 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 $\mathrm{x}\sigma$ 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: $\{\mathrm{x}_1/\tau_1, \ldots, \mathrm{x}_n/\tau_n\}$.

Substitutions on terms

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

- If τ is a variable x, then $\tau\sigma = \mathrm{x}\sigma$;
- 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

- If $\mathrm{R}(\tau_1, \ldots, \tau_n)$ is an atomic formula, then $[\mathrm{R}(\tau_1, \ldots, \tau_n)]\sigma$ is $\mathrm{R}(\tau_1\sigma, \ldots, \tau_n\sigma)$;
- $[\neg\phi]\sigma$ is $\neg[\phi\sigma]$;
- $[\phi \wedge \psi]\sigma$, $[\phi \vee \psi]\sigma$, and $[\phi \to \psi]\sigma$, are $[\phi\sigma] \wedge [\psi\sigma]$, $[\phi\sigma] \vee [\psi\sigma]$, and $[\phi\sigma] \to [\psi\sigma]$ respectively;
- $[\forall \mathrm{x}\phi]\sigma$ is $\forall \mathrm{x}[\phi\sigma_x]$, and $[\exists \mathrm{x}\phi]\sigma$ is $\exists \mathrm{x}[\phi\sigma_x]$.

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 $\{\mathrm{x/c, w/g(z), y/h(u, x)}\} = \{\mathrm{x/c, w/g(z)}\}\theta$.

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.

But 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

Disagreement pairs

f

h g

x y x w

f

h g

x y x k

u v

Disagreement Pair

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? T

'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 not occurring in the other, 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 behaviour 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 either of the arguments of the complex term `Term`, making use of the `not_occurs_in/2` predicate.

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

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

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 (Version I)

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

Here s is a new *Skolem function symbol*, and x_1, $\ldots$, 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.

The new existential rules (Version II)

A little black magic: this version of the existential rules is sound.

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

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

An example

We prove that $\exists x \forall y \textsc{shoots}(x,y) \rightarrow \forall y \exists x \textsc{shoots}(x,y)$ using our new quantifier rules and unification. Here are the first 7 steps:

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

The example continued

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

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

This tableau *is* closed.

Atomic 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 atomic 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.
- But it wasn't designed for humans, it was designed for computers. And for computers, it's pretty good.

Remarks on the PROLOG implementation

- The implementation given in the notes is extension of the propositional tableaux implementation — but not a straightforward extension.
- The basic ideas underlying signed free variable tableaux make it possible to devise practical implementations, there is a lot more detail to take care of than in the propositional case (substitution, Skolem functions, . . . , and so on).
- Moreover, the fact remains that first-order logic is undecidable, so the implementation has to defuse the (very real) threat of non-terminating tableaux constructions

Ensuring termination

- The outermost predicate is called `saturate` (as in the propositional case).
- However the first-order version `saturate` has a second argument, a number called `Qdepth` This is number of times that we are allowed to apply universal rules in the course of constructing a tableau.

```
saturate(Tableau,_):-
   closed(Tableau).

saturate(OldTableau,Qdepth):-
   expand(OldTableau,Qdepth,NewTableau,NewQdepth),!,
   saturate(NewTableau,NewQdepth).
```

Fun and games with Qdepth

```
expand([Brch|Tabl],QD,[NewBrch|Tabl],QD):-
   existential_expansion(Brch,NewBrch).

expand([Brch|Tabl],OldQD,NewTabl,NewQD):-
   universal_expansion(Brch,OldQD,NewBrch,NewQD),
   append(Tabl,[NewBrch],NewTabl).

expand([Brch|Tabl],QD,[NewBrch1,NewBrch2|Tabl],QD):-
   disjunctive_expansion(Brch,NewBrch1,NewBrch2).
```

More fun and games with Qdepth

```
universal_expansion(Branch,OldQD,NewBranch,NewQD):-
    member(NotatedFormula,Branch),
    notated_formula(NotatedFormula,Free,SignedFormula),
    universal(SignedFormula),
    remove_first(NotatedFormula,Branch,Temp),
    OldQD > 0, NewQD is OldQD - 1,
    instance(SignedFormula,V,Inst),
    notated_formula(NotatedInst,[V|Free],Inst),
    append([NotatedInst|Temp],[NotatedFormula],NewBranch).
```

A simple driver

```
valid(X,Qdepth):-
   notated_formula(NotatedFormula,[],f(X)),
   saturate([[NotatedFormula]],Qdepth).
```

Note the use of Qdepth!

A warning

- If `valid/2` fails, this does *not* mean that the input formula was not a theorem! (Perhaps we just set `Qdepth` too small.)
- On the other hand, if the program tells us that ϕ *is* a first-order tableaux theorem, this is not open to question. (If a closed tableau has been formed from the initial tableau $F\phi$, then ϕ is provable, and that's that.)

Putting It All Together

So far, we have developed a number of useful tools.

1. We can build first-order representations in a compositional way for simple natural language expressions.
2. We have automated the process of performing reasoning with (equality free) first-order representations.

Time to have a little fun!

Adding Questions

1. *We know how to compositionally build semantic representations*
2. *We know how to evaluate first-order formulas in well-named models*

Let's extend our grammar coverage with questions:

```
q(Sem) --> wh(SemWH), vp(SemVP), {fa(SemWH,SemVP,Sem)}.

wh(lambda(P,lambda(X,R))) --> [who], {fa(P,X,R)}.

wh(lambda(P,lambda(X,S & R))) --> [which], noun(SemNoun),
                                  {fa(SemNoun,X,S),fa(P,X,R)}.
```

With these entries we can handle questions like Who is a customer?

Adding Questions (continued)

The semantic representations that are derived for questions are lambda expressions.

```
?- q(Sem,[who,is,a,customer],[]).
Sem = lambda(X,customer(X))
```

Let's write a driver that translates a question, feeds the result to an example model, and returns the answer.

```
query(Question,Example,Answer):-
    q(lambda(Answer,Sem),Question,[]),
    example(Example,Model), sem(Sem,Model).
```

Here is an example query, evaluated to example model 1:

```
?- query([which,robber,loves,honey,bunny],1,Answer).
Answer = pumpkin
```

Basic Argumentation

We might want to let the computer decide whether a natural language argument is valid.

Vincent gives every woman a foot massage.
Mia is a woman.

Vincent gives Mia a foot massage.

This argument is *valid*. If the premisses are true, then the conclusion is true too.

- We already have the basic tool needed to test arguments for validity
- We need to add a driver which hands the theorem prover a list of formulas instead of just a single formula
- The theorem prover will then attempt to make the premisses true and the conclusion false

Implementing Basic Argumentation

The driver for the theorem prover:

```
valid_argument([Conclusion|Premisses],Qdepth):-
   notated_formula(NotatedConclusion,[],f(Conclusion)),
   premisses(Premisses,[],NotatedPremisses),
   saturate([[NotatedConclusion|NotatedPremisses]],Qdepth).
```

A predicate that turns a list of formulas in a list of signed, notated formulas:

```
premisses([],N,N).
premisses([P|Premisses],SoFar,NotatedPremisses):-
   notated_formula(NotatedP,[],t(P)),
   premisses(Premisses,[NotatedP|SoFar],NotatedPremisses).
```

Implementing Basic Argumentation (continued)

A driver that takes care of the interface between the natural language argument and `valid_argument` predicate:

```
argument(P1,P2,C):-
   s(SemP1,P1,[]),
   s(SemP2,P2,[]),
   s(SemC,C,[]),
   valid_argument([SemC,SemP1,SemP2],25).
```

A test session:

```
?- argument([vincent,gives,every,woman,a,foot,massage],
            [mia,is,a,woman],
            [vincent,gives,mia,a,foot,massage]).
yes
```

Implementing Basic Argumentation (continued)

We can try and see if the whole thing works for arguments that are *not* valid.

Mia loves a man.
Every boxer is a man.

Mia loves a boxer.

So let's try:

```
?- argument([mia,loves,a,man],
            [every,boxer,is,a,man],
            [mia,loves,a,boxer]).
no
```

Argumentation with Scope Ambiguities

Let's have a look at the following argument:

Every man is loved by a woman.
Every boxer is a man.

A woman loves every boxer.

Is this a valid argument?

The answer is: *sometime yes, sometimes no.*

It *is* a valid argument if the first premise is understood as:

> there is a woman such that she loves every man

Implementing Argumentation with Scope Ambiguities

With our Cooper Storage implementation we get both readings for the first premise:

```
?- s(Sem,[every,man,is,loved,by,a,woman],[]).

Sem = exists(Y,woman(Y)&forall(X,man(X)>love(Y,X))) ;
Sem = forall(X,man(X)>exists(Y,woman(Y)&love(Y,X)))
```

Now we are able to do test arguments with scoping ambiguities:

```
?- argument([every,man,is,loved,by,a,woman],
            [every,boxer,is,a,man],
            [a,woman,loves,every,boxer]).
yes
```

Today:

- We introduce *Discourse Representation Theory* (DRT), and show that it is compatible with a bottom up, λ-driven, approach to semantic construction.
- We then implement this bottom up approach in PROLOG, and treat a number of other issues, including anaphora resolution.

Discourse Representations

In this lecture we change our underlying logical representation formalism: instead of using first-order representations we will use *Discourse Representation Structures* (DRSs).

Discourse Representation Theory

DRSs consist of a list of *discourse referents* and a list of *conditions*.

1. If $x_1, \ldots, x_n$ are discourse referents ($n \geq 0$) and $\gamma_1, \ldots, \gamma_m$ ($m > 0$) are conditions then

   $$\begin{array}{|l|}\hline x_1, \ldots, x_n \\ \hline \gamma_1 \\ . \\ . \\ . \\ \gamma_m \\ \hline \end{array}$$

   is a DRS.

2. If R is a relation symbol of arity n, and $x_1,\ldots,x_n$ are some discourse referents, then $R(x_1,\ldots,x_n)$ is a condition.
3. If x is a discourse referent, and c is a constant, then x=c and c=x are conditions.
4. If x and y are discourse referents, then x=y is a condition.
5. If K_1 and K_2 are DRSs, then $K_1 \rightarrow K_2$ is a condition.
6. If K_1 and K_2 are DRSs, then $K_1 \vee K_2$ is a condition.
7. If K is a DRS, then $\neg$ K is a condition.
8. Nothing is a condition or a DRS unless it can be shown to be so using clauses 1–7.

Indefinites

The DRS for a woman snorts is:

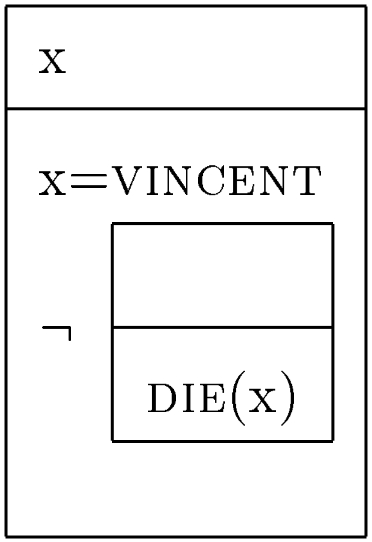
$$
\begin{array}{|l|}
\hline
x \\
\hline
\textsc{woman}(x) \\
\textsc{snort}(x) \\
\hline
\end{array}
$$

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 $\textsc{snort}(x)$.

Proper Names

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



Universal Quantifiers

The DRS for every gimp shrieks is:

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

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.

A woman snorts. She collapses.

$$
\begin{array}{|l|}
\hline
x\ y \\
\hline
\text{WOMAN}(x) \\
\text{SNORTS}(x) \\
\text{COLLAPSE}(y) \\
y=x \\
\hline
\end{array}
$$

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

Every woman snorts. ? She collapses.

$$\begin{array}{|l|}\hline \mathrm{y} \\ \hline \begin{array}{|l|}\hline \mathrm{x} \\ \hline \textsc{woman}(\mathrm{x}) \\ \hline \end{array} \rightarrow \begin{array}{|l|}\hline \\ \hline \textsc{snorts}(\mathrm{x}) \\ \hline \end{array} \\ \textsc{collapse}(\mathrm{y}) \\ \mathrm{y}=? \\ \hline \end{array}$$

There is no accessible discourse referent for y

Donkey Sentences

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

Correct:

$$\forall x \forall y[\textsc{man}(x) \wedge \textsc{big\_kahuna\_burger}(y) \wedge \textsc{eat}(x,y) \rightarrow \textsc{enjoy}(x,y)]$$

But (using our translations for the quantifiers):

$$\exists x[\textsc{man}(x) \wedge \exists y[\textsc{big\_kahuna\_burger}(y) \wedge \textsc{eat}(x,y)]] \rightarrow \textsc{enjoy}(x,y)$$

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)	$\rightarrow$	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
  $\forall x \forall y[\textsc{man}(x) \wedge \textsc{big\_k\_burger}(y) \wedge \textsc{eat}(x,y) \rightarrow \exists v \exists w(\textsc{enjoy}(x,y) \wedge v{=}x \wedge w{=}y)]$
- Top Down *or* Bottom Up. (We choose the latter.)

λ-DRT

We will:

- take our nice new box language, add the now familiar λ operator, and a new operator $\oplus$ (the *merge*).
- build up 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.

Merge

Additional ingredient: the *merge* operator: $\oplus$

$$\begin{array}{|l|}\hline \text{x} \\ \hline \textsc{boxer}(\text{x}) \\ \textsc{lose}(\text{x}) \\ \hline \end{array} \oplus \begin{array}{|l|}\hline \text{y} \\ \hline \textsc{die}(\text{y}) \\ \text{y}=\text{x} \\ \hline \end{array} = \begin{array}{|l|}\hline \text{x y} \\ \hline \textsc{boxer}(\text{x}) \\ \textsc{lose}(\text{x}) \\ \textsc{die}(\text{y}) \\ \text{y}=\text{x} \\ \hline \end{array}$$

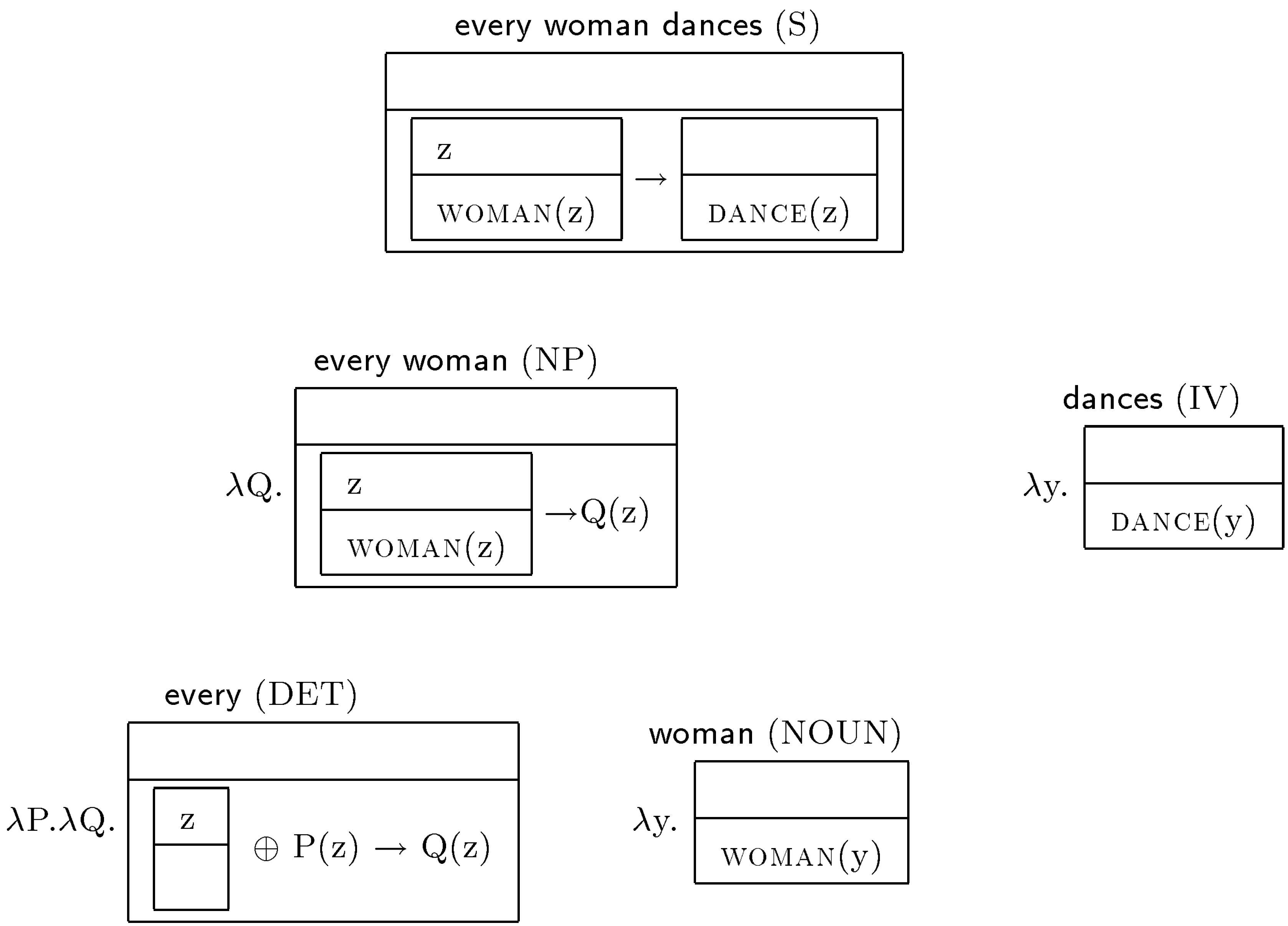
Lexical Items in λ-DRT

boxer: $\lambda y.\begin{array}{|l|}\hline \\ \hline \textsc{boxer}(y) \\ \hline\end{array}$ woman: $\lambda y.\begin{array}{|l|}\hline \\ \hline \textsc{woman}(y) \\ \hline\end{array}$ dances: $\lambda y.\begin{array}{|l|}\hline \\ \hline \textsc{dance}(y) \\ \hline\end{array}$

a: $\lambda P.\lambda Q.\begin{array}{|l|}\hline z \\ \hline \\ \hline\end{array} \oplus P(z) \oplus Q(z)$

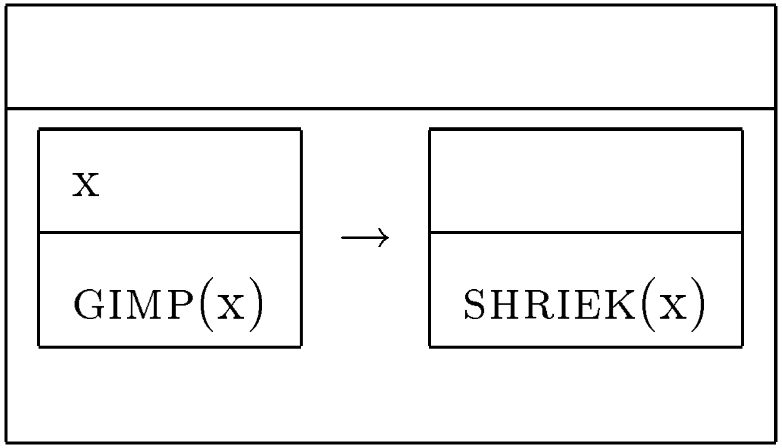
mia: $\lambda P.\begin{array}{|l|}\hline x \\ \hline x = \textsc{mia} \\ \hline\end{array} \oplus P(x)$

every: $\lambda P.\lambda Q.\begin{array}{|l|}\hline \\ \hline \left(\begin{array}{|l|}\hline z \\ \hline \\ \hline\end{array} \oplus P(z)\right) \rightarrow Q(z) \\ \hline\end{array}$


every woman dances (S)
z
WOMAN(z)
→
DANCE(z)
every woman (NP)
λQ.
z
WOMAN(z)
→Q(z)
dances (IV)
λy.
DANCE(y)
every (DET)
λP.λQ.
z
⊕ P(z) → Q(z)
woman (NOUN)
λy.
WOMAN(y)

Representing DRSs in PROLOG

DRS in box notation



DRS in PROLOG:

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

Implementing λ-DRT in Prolog

- we know how to represent DRSs in Prolog
- we already have the predicates needed for the lambda calculus
- and we have DCGs

Lexical entries:

```
noun(lambda(X,drs([],[boxer(X)]))) --> [boxer].

iv(lambda(X,drs([],[dance(X)]))) --> [dances].

pn(lambda(P,drs([X|D],[X=mia|C])))
                        --> [mia],{fa(P,X,drs(D,C))}.
```

Problematic Merge Operations

The determiner `a`:

```
det(lambda(P,lambda(Q,drs(D3,C3))))  --> [a],
    {fa(P,X,drs(D1,C1)),fa(Q,X,drs(D2,C2)),
     append([X|D1],D2,D3),append(C1,C2,C3)}.
```

→ Inefficient and unwanted backtracking!

Postponing Merge

Represent the merge of two DRSs as: `merge(B1,B2)`

Revised entry `a`:

```
det(lambda(P,lambda(Q,merge(drs([X|D],C),B)))) --> [a],
    {fa(P,X,drs(D,C)),fa(Q,X,B)}.
```

First Example Output

What happens when we query a DCG containing such entries?

```
?- s(B,[a,boxer,eats,a,big,kahuna,burger],[]).

B = merge(drs([X],[boxer(X)]),
          merge(drs([Y],[big_kahuna_burger(Y)]),
                drs([],[eat(X,Y)])
               )
         )
```

Carry out the stipulated merges

```
mergeDrs(drs(D,C1),drs(D,C2)):-
   mergeDrs(C1,C2).
mergeDrs(merge(B1,B2),drs(D3,C3)):-
   mergeDrs(B1,drs(D1,C1)),mergeDrs(B2,drs(D2,C2)),
   append(D1,D2,D3),append(C1,C2,C3).

mergeDrs([B1 > B2|C1],[B3 > B4|C2]):-!,
   mergeDrs(B1,B3),mergeDrs(B2,B4),mergeDrs(C1,C2).
mergeDrs([~ B1|C1],[~ B2|C2]):-!,
   mergeDrs(B1,B2),mergeDrs(C1,C2).
mergeDrs([C|C1],[C|C2]):-
   mergeDrs(C1,C2).
mergeDrs([],[]).
```

Merging in the grammar rules

Add the following rule to the DCG:

```
m(Drs) --> s(D), {mergeDrs(D,Drs)}.
```

Now we get 'real' DRSs from the parser. For example:

```
?- m(K,[a,boxer,dies],[]).

K = drs([X],[boxer(X),die(X)])
```

Anaphora Resolution

- restricted by the geometrical structure of DRSs
- a discourse referent can be an antecedent for an anaphor only if it is accessible.
- we extend our implementation for handling anaphora resolution.

DCG Rules for Discourse

Two new DCG rules we need:

```
d(drs([],[])) --> [].
d(merge(Drs1,Drs2)) --> s(Drs1), d(Drs2).
```

Clause for conditionals:

```
d(drs([],[Drs1 > Drs2])) --> [if], s(Drs1), s(Drs2).
```

Handling the pronouns

Practical problem: *antecedents are not in the lexicon!*

Solution: *postpone resolution!*

Sample Representation of an α-DRS:

```
alfa(Y,drs([X],[X=Y,snorts(X)])).
```

Lexical entry for a pronoun:

```
pro(lambda(P,merge(alfa(Y,drs([X],[X=Y])),B))) -->
          [she], {fa(P,X,B)}.
```

A First Try

Before implementing anaphora resolution, let's try out the parser:

```
?- d(B,[a,woman,snorts,she,collapses],[]).

B = merge(merge(drs([X],[woman(X)]),drs([],[snort(X)])),
          merge(merge(alfa(Z,drs([Y],[Y=Z])),
                      drs([],[collapse(Y)])),
                drs([],[])))
```

Defining α–Resolution: DRSs (I)

```
resolveDrs(Drs,Resolved):-
   resolveDrs(Drs,[]-_Acc,Resolved).

resolveDrs(drs(D,C1),Acc1-Acc3,drs(D,C2)):-
   append(D,Acc1,Acc2),
   resolveDrs(C1,Acc2-Acc3,C2).
```

Defining α–Resolution: DRSs (II)

```
resolveDrs(merge(B1,B2),Acc1-Acc3,merge(B3,B4)):-
   resolveDrs(B1,Acc1-Acc2,B3),
   resolveDrs(B2,Acc2-Acc3,B4).

resolveDrs(alfa(Pron,B1),Acc1-Acc2,B2):-
   member(Pron,Acc1),
   resolveDrs(B1,Acc1-Acc2,B2).
```

Defining α–Resolution: Conditions (I)

```
resolveDrs([B1 > B2|C1],Acc1-Acc3,[B3 > B4|C2]):-!,
   resolveDrs(B1,Acc1-Acc2,B3),
   resolveDrs(B2,Acc2-_,B4),
   resolveDrs(C1,Acc1-Acc3,C2).
resolveDrs([B1 v B2|C1],Acc1-Acc2,[B3 v B4|C2]):-!,
   resolveDrs(B1,Acc1-_,B3),
   resolveDrs(B2,Acc1-_,B4),
   resolveDrs(C1,Acc1-Acc2,C2).
```

Defining α–Resolution: Conditions (II)

```
resolveDrs([~ B1|C1],Acc1-Acc2,[~ B2|C2]):-!,
   resolveDrs(B1,Acc1-_,B2),
   resolveDrs(C1,Acc1-Acc2,C2).
resolveDrs([C|C1],Acc1-Acc2,[C|C2]):-
   resolveDrs(C1,Acc1-Acc2,C2).
resolveDrs([],Acc-Acc,[]).
```

Integrating α-Resolution in the DCG

Add the following DCG-rule

```
r(Drs) --> d(D1), {resolveDrs(D1,D2),mergeDrs(D2,Drs)}.
```

Now for a test. Let's parse: A boxer collapses. He dies.

```
?- r(K,[a,boxer,collapses,he,dies],[]).

K = drs([X,Y],[boxer(X),collapse(X),Y=X,die(Y)])
```

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.

Translating Boxes

DRS with a non-empty universe

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

Translating Boxes (II)

DRSs with an empty universe

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

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

straightforward to translate:

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

Translating the $\rightarrow$

antecedent DRS has a non-empty universe:

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

Translating the $\rightarrow$ (II)

antecedent DRS has an empty universe:

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

Summing up

- This translation gives us a semantics for our DRS language
- DRSs built using a vocabulary V talk about models of that vocabulary
- What exactly do they say about such models?
- Define an interpretation for the DRS language over models of this vocabulary
- Another way is simply to associate each DRS with a formula of first-order logic with equality