# Discourse

## BSc Artificial Intelligence, Spring 2011

Raquel Fernández

Institute for Logic, Language & Computation
University of Amsterdam

# Summary from Last Week

- The querying task: given a first-order formula $\varphi$ and a first-order model $M$, is $\varphi$ satisfied in $M$ or not?
  - recall that this corresponds to the process of evaluating natural language descriptions in world situations while making use of contextual information.
- We addressed this task computationally by implementing a first-order model checker.

- Homework #1 asked you to play around with and test several aspects of the model checker. We will briefly discuss Exercise 3.

# HW1 – Exercise 3

Provide Prolog clauses for $\rightarrow$ and $\forall$ according to the satisfaction definition of these connectives, without reusing clauses for other connectives.

3.a) Implication:

**Satisfaction definition:** $M, g \models \varphi \rightarrow \psi$ iff not $M, g \models \varphi$ or $M, g \models \psi$

**Clauses used in `modelChecker1.pl`:**

```
satisfy(imp(Formula1,Formula2),Model,G,Pol):-
  satisfy(or(not(Formula1),Formula2),Model,G,Pol).
```

**Alternative clauses without reusing other connectives:**

```
satisfy(imp(Formula1,Formula2),Model,G,pos):-
   satisfy(Formula1,Model,G,pos);
   satisfy(Formula2,Model,G,neg).

satisfy(imp(Formula1,Formula2),Model,G,neg):-
   satisfy(Formula1,Model,G,pos),
   satisfy(Formula2,Model,G,neg).
```

# HW1 – Exercise 3

## 3.b) Universal Quantification:

**Satisfaction definition:** $M, g \models \forall x \varphi$ iff $M, g' \models \varphi$ for all $x$-variants $g'$ of $g$

**Clauses used in `modelChecker1.pl` for $\forall$:**

```
satisfy(all(X,Formula),Model,G,Pol):-
  satisfy(not(some(X,not(Formula))),Model,G,Pol).
```

**Clauses for existential quantification $\exists$:**

```
satisfy(some(X,Formula),model(D,F),G,pos):-
  memberList(V,D),
  satisfy(Formula,model(D,F),[g(X,V)|G],pos).

satisfy(some(X,Formula),model(D,F),G,neg):-
 setof(V,(memberList(V,D),satisfy(Formula,model(D,F),[g(X,V)|G],neg)),Dom),
 setof(V,memberList(V,D),Dom).
```

# HW1 – Exercise 3

**Clauses used in `modelChecker1.pl` for ∀:**

```
satisfy(all(X,Formula),Model,G,Pol):-
  satisfy(not(some(X,not(Formula))),Model,G,Pol).
```

**Clauses for existential quantification ∃:**

```
satisfy(some(X,Formula),model(D,F),G,pos):-
  memberList(V,D),
  satisfy(Formula,model(D,F),[g(X,V)|G],pos).

satisfy(some(X,Formula),model(D,F),G,neg):-
 setof(V,(memberList(V,D),satisfy(Formula,model(D,F),[g(X,V)|G],neg)),Dom),
 setof(V,memberList(V,D),Dom).
```

**Alternative clauses for ∀ without reusing other connectives:**

```
satisfy(all(X,Formula),model(D,F),G,pos):-
 setof(V,(memberList(V,D),satisfy(Formula,model(D,F),[g(X,V)|G],pos)),Dom),
 setof(V,memberList(V,D),Dom).

satisfy(all(X,Formula),model(D,F),G,neg):-
  memberList(V,D),
  satisfy(Formula,model(D,F),[g(X,V)|G],neg).
```

# Plan for Today

We now know how to deal with the querying task, which is the core of mode-theoretic semantics (meaning $\approx$ truth in a model).

The querying task manipulates logical formulas, which we take to be our semantic representations.

But how do we associate natural language sentences with logical formulas?

$\Rightarrow$ This is the task we'll address today, namely **semantic construction**: how to systematically associate logical semantic representations with natural language sentences.

Once we have this essential piece as part of our system, we'll be able to move on to discourse-related issues involving more than single sentences.

# Compositionality

We want to be able to establish a systematic (non-arbitrary) relation between sentences and formulas.

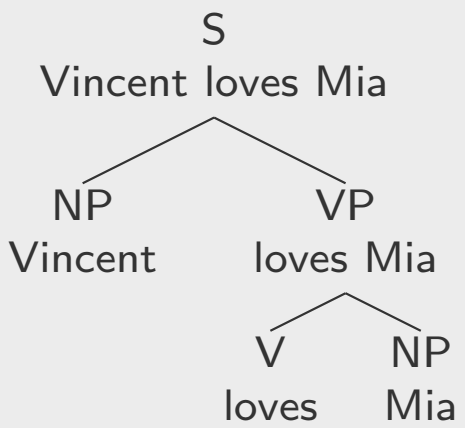
| | | |
|---|---|---|
| Vincent loves Mia | $?\rightsquigarrow$ | $\textsc{love}(\textsc{vincent}, \textsc{mia})$ |
| Everyone hates Butch | $?\rightsquigarrow$ | $\forall x.\textsc{hate}(x, \textsc{butch})$ |

Intuitively, we know that the meaning of a sentence is based on the meaning of its bits and pieces (*compositionality*):

- we may be able to associate a representation with each lexical item, but how is this information combined?
- the meaning of a sentence is not only based on the words that make it up, but also on the ordering, grouping, and relations among such words
- the missing ingredient is a notion of **syntactic structure**.

# Syntax and Compositional Semantics

As you know, syntax tells us how to hierarchically decompose a sentence into sub-parts that ultimately lead to the lexical items:




- If we associate a semantic representation with each lexical item, and...
- describe how the semantic representation of a syntactic constituent is to be built up from the representation of its sub-parts, then...
- we have at our disposal a compositional semantics: a systematic way of constructing semantic representations for sentences.

# Semantic Construction

Now we have a plausible strategy for finding a way to systematically associate first-order semantic representations with sentences.

We need to:

1. Specify a reasonable syntax for the fragment of natural language of interest.
2. Specify semantic representations for the lexical items.
3. Specify how the semantic representation of a syntactic constituent is constructed in terms of the representations of its subparts.

Since we are interested in semantics, task 1 and 2 are where our real interests lie.

To handle task 1, we'll adopt a very simple solution: we'll use Definite Clause Grammars (DCGs), the built-in Prolog mechanism for grammar specification and parsing.

# Syntax with DCGs

```
s --> np, vp.
np --> pn.
vp --> iv.
vp --> tv, np.
np --> det, noun.
det --> [a].
det --> [every].

pn --> [vincent].
pn --> [mia].
noun --> [woman].
noun --> [foot,massage].
iv --> [snorts].
iv --> [walks].
tv --> [likes].
```

- complex syntactic categories: `s`, `np`, `vp`
- simple syntactic categories: `det`, `noun`, `iv`, `tv`
- lexical items: `a`, `every`, `vincent`, `mia`, `woman`, `footmassage`, `snorts`, `walks`, `likes`.

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

?- s([every,walks],[]).
fail.
```

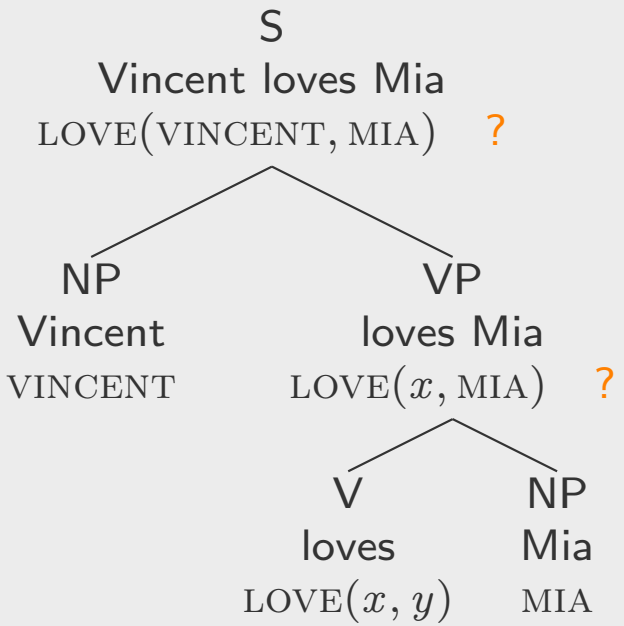
```
?- np(X,[]).
X = [vincent] ;
X = [mia] ;
X = [a, woman] ;
X = [a, foot, massage] ;
X = [every, woman] ;
X = [every, foot, massage].
```

# Semantic Construction

How shall we deal with tasks 2 and 3?

2. Specify semantic representations for the lexical items.
3. Specify how the semantic representation of a syntactic constituent is constructed in terms of the representations of its subparts.

Using plain FOL does not seem very handy...




we could represent lexical items with first-order terms and formulas, but how do we combine them?

we'd like to replace variables with terms, but how should we do that?

Fortunately, we can use a notational extension of FOL that will make these tasks easy: the *lambda calculus*.

# The Lambda Calculus

# Lambda Abstraction

We shall view the lambda calculus as a notational extension of FOL that allows us to bind variables with a new operator $\lambda$:

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

- the prefix $\lambda x.$ binds the occurrence of $x$ in $\textsc{woman}(x)$
- we often say the that prefix $\lambda x.$ *abstracts over* $x$, and call expressions with such prefixes *lambda expressions* or *lambda abstractions*
- we can use one lambda expression as the body of another one:

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

# Functional Application

We can think of the lambda calculus as a tool dedicated to gluing together the items needed to build semantic representations.

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

- the purpose of abstracting over variables is to mark the slots where we want substitutions to be made
  - the binding of the free variable $x$ in $\textsc{woman}(x)$ indicates that WOMAN has an argument slot where we may perform substitutions
- lambda abstractions can be seen as *functors* that can be applied to *arguments* [we shall use the symbol @ for functional application]

$$\lambda x.\textsc{woman}(x)@\textsc{mia}$$

- a compound expression of this sort refers to the application of the functor $\lambda x.\textsc{woman}(x)$ to the argument MIA.

# $\beta$-conversion

Compound expressions $\mathcal{F}@\mathcal{A}$ can be seen as instructions to

- throw away the $\lambda x.$ prefix of the functor $\mathcal{F}$, and
- replace any occurrence of $x$ bound by the $\lambda$-operator with the argument $\mathcal{A}$

This replacement or substitution process is called $\beta$-conversion:

$$\begin{aligned} \lambda x.\textsc{woman}(x)@\textsc{mia} &\rightsquigarrow \textsc{woman}(\textsc{mia}) \\ \lambda y.\lambda x.\textsc{hate}(x,y)@\textsc{butch} &\rightsquigarrow \lambda x.\textsc{hate}(x,\textsc{butch}) \end{aligned}$$

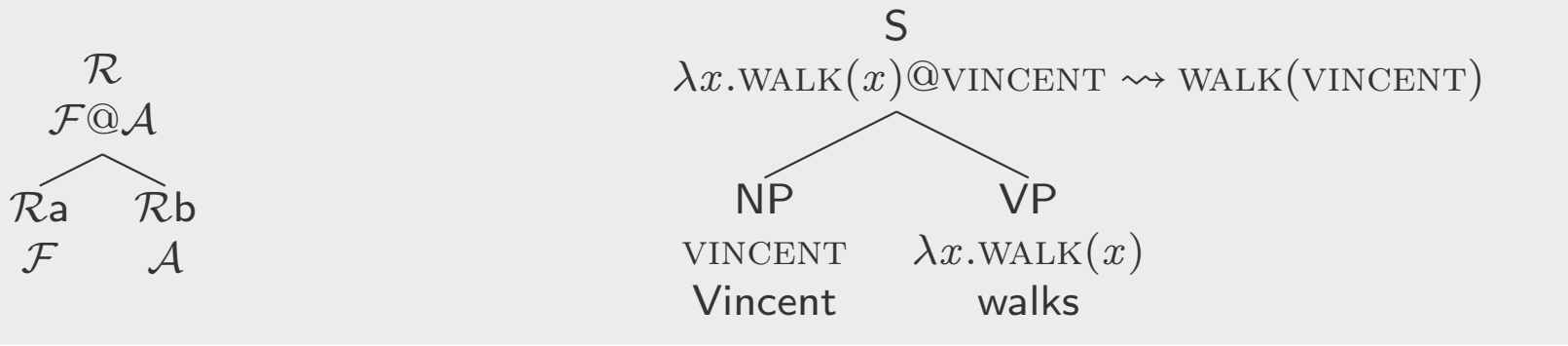
Note that the $\lambda$-operator can bind variables ranging over complex expressions: lambda abstractions can also act as arguments

$$\begin{aligned} \lambda v.\exists x.(\textsc{boxer}(x) \wedge v@x) \ @ \ \lambda x.\textsc{dance}(x) &\rightsquigarrow \exists x.(\textsc{boxer}(x) \wedge \lambda x.\textsc{dance}(x)@x) \\ &\rightsquigarrow \exists x.(\textsc{boxer}(x) \wedge \textsc{dance}(x)) \end{aligned}$$

# Lambda Calculus for Semantic Construction

Lambda abstraction, functional application, and $\beta$-conversion are the main ingredients we need to deal with semantic construction:

- Once we have devised lambda abstractions to represent lexical items, we only need to use functional application and $\beta$-conversion to combine semantic representations compositionally.
  - Given a syntactic constituent $\mathcal{R}$ with subparts $\mathcal{R}$a and $\mathcal{R}$b, we need to specify which subpart is to be thought as the functor $\mathcal{F}$ and which as the argument $\mathcal{A}$.
  - We then construct the semantic representation of $\mathcal{R}$ by functional application $\mathcal{F}@\mathcal{A}$



# $\lambda$-abstractions for Lexical Items (1)

How shall we represent the different basic syntactic categories?

We have been representing intransitive verbs and nouns as 1-place relations which are missing their argument:

walk: $\lambda x.\text{WALK}(x)$
boxer: $\lambda x.\text{BOXER}(x)$

What about determiners such as 'a' and 'every' in NPs like 'a boxer'?

- For instance, we'd like to represent the meaning of 'a boxer walks' as

$$\exists x.\text{BOXER}(x) \wedge \text{WALK}(x)$$

- What does each word contribute to this formula? And what is the contribution of the determiner?

# $\lambda$-abstractions for Lexical Items (2)

A boxer walks: $\exists x.\text{BOXER}(x) \wedge \text{WALK}(x)$

If 'boxer' contributes $\text{BOXER}(x)$ and 'walks' contributes $\text{WALK}(x)$, then the determiner 'a' must contribute something like this $\exists x \ldots \wedge \ldots$

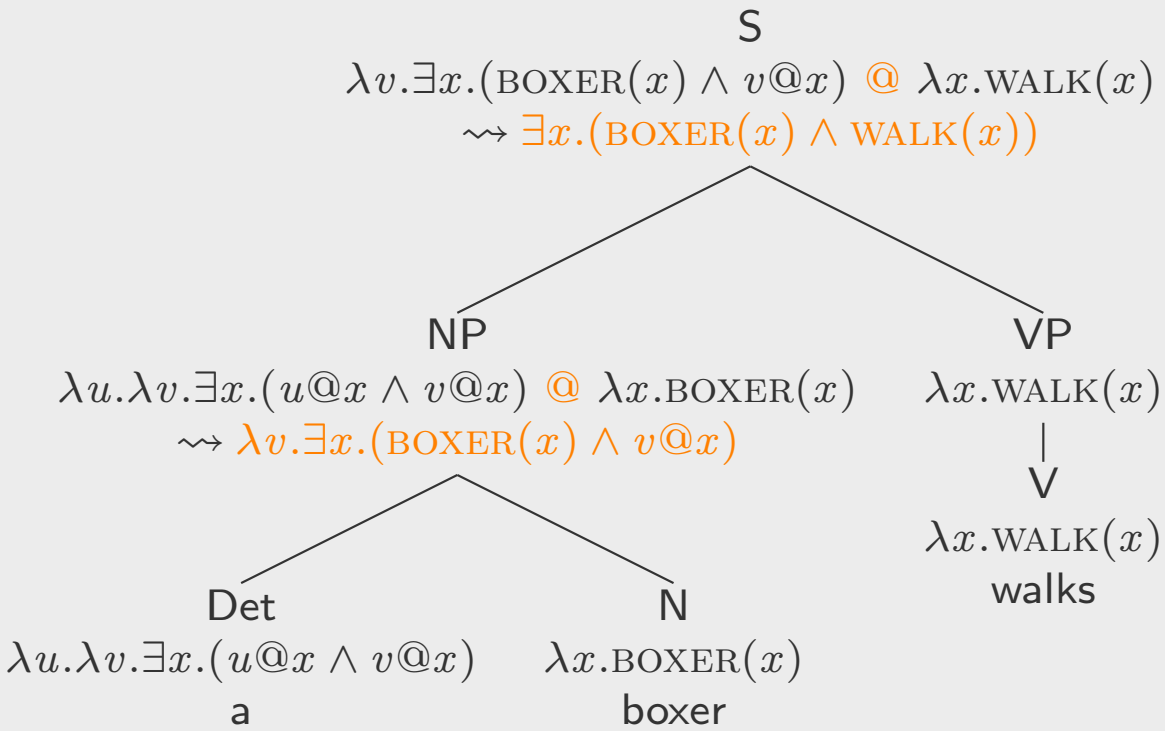
For the determiner we then need three arguments:

- one for the existentially bound variable $\exists x \ldots \wedge \ldots$
- one for the contribution of the NP (the *restriction*) $\exists x \ldots \wedge \ldots$
- one for the contribution of the VP (the *scope*) $\exists x \ldots \wedge \ldots$

We can use lambda abstraction to mark the missing arguments that will be filled in during semantic construction. This is the representation for existential determiners:

a, some: $\lambda u.\lambda v.\exists x.(u@x \wedge v@x)$

# A boxer walks



# $\lambda$-abstractions for Lexical Items (3)

We have represented quantified NPs as functors:

a boxer: $\lambda v.\exists x.(\textsc{boxer}(x) \wedge v@x)$

In order to have a uniform representation of all NPs as functors, we can use the following representation for proper nouns (instead of simply using constants) — a functor that applies its own argument to itself:

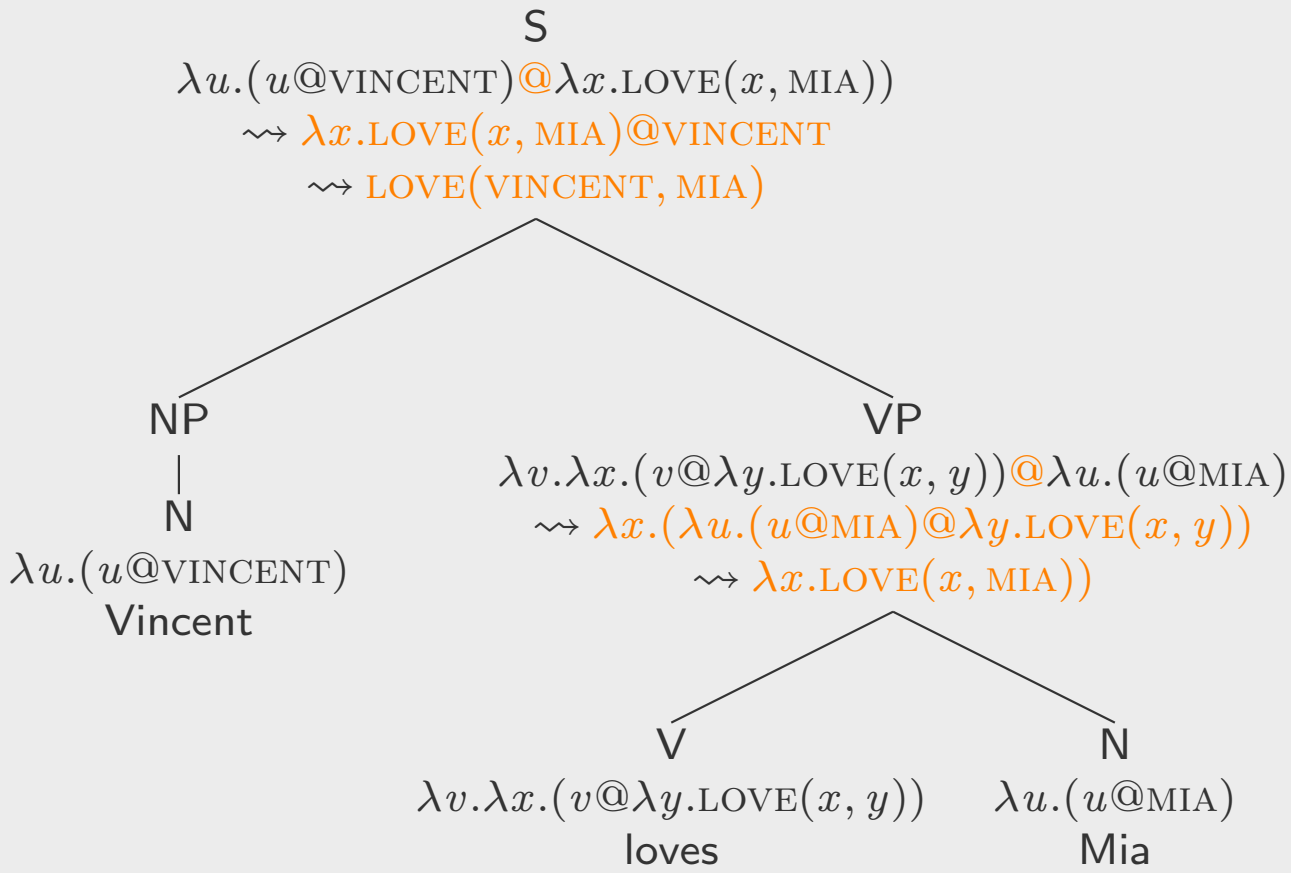
Mia: $\lambda u.(u@\textsc{mia})$

This complicates a little bit the representation of transitive verbs; instead of $\lambda y.\lambda x.\textsc{love}(x, y)$, we need the following representation:

loves: $\lambda v.\lambda x.(v@\lambda y.\textsc{love}(x, y))$

Let us see why...

# Vincent loves Mia



# Implementing Lambda Calculus

# DCGs for Semantic Construction

We shall use the following Prolog terms to implement the lambda calculus:

- lambda abstraction ($\lambda x.E$): `lam(X,E)`
- functional application ($\mathcal{F}@\mathcal{A}$): `app(F,A)`

Here are the main syntactic rules decorated with semantic representations:

```
s(app(NP,VP))--> np(NP), vp(VP).
np(app(Det,Noun))--> det(Det), noun(Noun).
np(PN)--> pn(PN).
vp(IV)--> iv(IV).
vp(app(TV,NP))--> tv(TV), np(NP).
```

And here are some lexical entries:

```
noun(lam(X,woman(X)))--> [woman].
iv(lam(Y,walk(Y)))--> [walks].
tv(lam(X,lam(Y,app(X,lam(Z,like(Y,Z))))))--> [likes].
pn(lam(P,app(P,mia)))--> [mia].
det(lam(P,lam(Q,all(X,imp(app(P,X),app(Q,X))))))--> [every].
det(lam(P,lam(Q,some(X,and(app(P,X),app(Q,X))))))--> [a].
```

This code can be found in the file `experiment3.pl`

# Implementing $\beta$-conversion

We can used the above DCG for semantic construction:

```
?- s(Sem,[mia,walks],[]).
Sem = app(lam(_G262, app(_G262, mia)), lam(_G268, walk(_G268)))
```

The output is correct, but what we want instead is `walk(mia)`. To get genuine first-order formulas we need $\beta$-conversion.

$\Rightarrow$ We will not discuss the code for $\beta$-conversion. You can find it in the file `betaConversion.pl`. It makes use of a stack to keep track of the expressions that need to be used as arguments.

The output of the DCG can be fed into the $\beta$-conversion predicate to obtain the first-order semantic representation for the sentence:

```
?- s(Sem,[mia,walks],[]), betaConvert(Sem,Reduced).
Sem = app(lam(_G295, app(_G295, mia)), lam(_G301, walk(_G301))),
Reduced = walk(mia) .
```

# Addendum: $\alpha$-conversion

There is one more ingredient in the lambda calculus we have not yet mentioned:

- In order to avoid accidental bindings during $\beta$-conversion, we should first change all the bound variables (bound by lambdas or quantifiers) in the functor to variables not used in the argument.
- The process of relabeling bound variables is called $\alpha$-conversion.

The following expressions are $\alpha$-equivalent:
$\lambda u.\exists x.(\text{WOMAN}(x) \wedge u@x)$
$\lambda v.\exists z.(\text{WOMAN}(z) \wedge v@z)$

Bound variables are *dummies* – it doesn't matter which particular variable we use.

- We will not discuss the code for implementing $\alpha$-conversion. You can find it in the file `alphaConversion.pl`. The $\beta$-conversion predicate uses `alphaConvert/2` to relabel all bound variables in the functor to fresh new symbols.

# Summary of Programs for the Lambda Calculus

- experiment3.pl (DCG with lambda calculus for a small fragment of English )
- betaConversion.pl
- alphaConversion.pl
- betaConversionTestSuite.pl
- comsemPredicates.pl (auxiliary predicates)

If you load the file `betaConversion.pl` and issue the command

```
?- betaConvertTestSuite.
```

the examples in the test suite will be evaluated; they have this form:

```
expression(app(lam(A,sleep(A)),mia),
           sleep(mia)).
```

the first argument of `expression/2` is the lambda expression to be $\beta$-converted and the second one is the result.

The output will be a series of entries of the following form:

```
Expression: app(lam(_G227, sleep(_G227)), mia)
Expected: sleep(mia)
Converted: sleep(mia)
Result: ok
```

The test suite file betaConversionTestSuite.pl contains many interesting and instructive examples with comments on many of them.

# Grammar Engineering

We have all the ingredients we need for semantic construction, but the DCG we have been playing with is very simple. It's time to move to a more interesting grammar fragment.

- Blackburn and Bos strive to develop a grammar that is:
  - modular
  - extendible
  - reusable
- This diagram illustrates the two-dimensional architecture of their grammar:

| `lambda.pl` | **SYNTAX** | **SEMANTICS** |
|---|---|---|
| **LEXICON** | the Lexicon<br>`englishLexicon.pl` | the Semantic Lexicon<br>`semLexLambda.pl` |
| **GRAMMAR** | the Syntax Rules<br>`englishGrammar.pl` | the Semantic Rules<br>`semRulesLambda.pl` |

# Syntax

## The Syntax Rules

- DCG rules annotated with additional grammatical information (s.a. agreement, morphology, etc).
- license several types of constructions: relative clauses, coordination,...
- they have a placeholder for semantic information:

```
s([sem:Sem])-->
   np([num:Num,sem:NP]),
   vp([num:Num,sem:VP]),
   {combine(s:Sem,[np:NP,vp:VP])}.
```

## The Lexicon

- The general format of a lexical entry is `lexEntry(Cat,Features)`, where `Cat` is the syntactic category and `Features` is a list of features.
- For example, the entries for the intransitive verb *'to walk'* are:

```
   lexEntry(iv,[symbol:walk,syntax:[walk],inf:inf,num:sg]).
   lexEntry(iv,[symbol:walk,syntax:[walks],inf:fin,num:sg]).
   lexEntry(iv,[symbol:walk,syntax:[walk],inf:fin,num:pl]).
```

# Semantics

## The Semantic Rules

- they implement the lambda calculus as we have seen earlier, with the help of app/2.
- here is where the predicate `combine/2` is defined. For instance:

```
combine(s:app(A,B),[np:A,vp:B]).
```

## The Semantic Lexicon

- lexical semantics, set of semantic macros
- the most important part of the grammar: the semantic definition of the lexical items determines the result of semantic construction

```
semLex(noun,M):-
   M = [symbol:Sym,
        sem:lam(X,Formula)],
   compose(Formula,Sym,[X]).

semLex(det,M):-
   M = [type:indef,
        sem:lam(P,lam(Q,some(X,and(app(P,X),app(Q,X)))))].
```

N.B: `compose/3` coerces a symbol and a variable into a $\lambda$-abstracted formula, e.g. `lam(X,boxer(X))`

# Wrapping Everything Together

This is the main level program:

```
lambda:-
readLine(Sentence),
        lambda(Sentence,Sems),
        printRepresentations(Sems).

lambda(Sentence,Sems):-
        setof(Sem,t([sem:Sem],Sentence,[]),Sems).
```

It uses `readLine/1` to read in a sentence, computes all semantic representations with `t/3` (defined in `englishGrammar.pl`), and prints them out.

```
?- lambda.
> Mia knows a boxer.
1 some(A, and(boxer(A), know(mia, A)))
```

Summary of programs for the full grammar fragment:

- `lambda.pl` main file for lambda calculus using the extended grammar
- `readLine.pl`
- `sentenceTestSuite.pl`
- `semLexLambda.pl`
- `semRulesLambda.pl`
- `englishLexicon.pl`
- `englishGrammar.pl`

# What's Next?

We now have a basic architecture for translating natural language sentences into a formal meaning representation (FOL) and for checking whether they are valid in a given situation.

We are ready to move on to discourse – to dealing with more than single sentences. We'll start by addressing these two tasks:

- Consistency Checking Task: given the logical representation of a discourse, is it consistent or inconsistent?
- Informativity Checking Task: given the logical representation of a discourse, is it informative or uninformative?

These tasks are much more difficult than the querying task: they are *undecidable* for FOL as we shall see. To deal with them efficiently, we'll use automated reasoning tools for theorem proving and model building.