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.
Provide Prolog clauses for → and ∀ 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:**

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

**Alternative clauses without reusing other connectives:**

```prolog
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).
```
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 \( \exists \):**

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

\[
satisfy(some(X,Formula),model(D,F),G,neg):-
    setof(V,(memberList(V,D),satisfy(Formula,model(D,F),[g(X,V)|G],neg)),Dom),
    setof(V,memberList(V,D),Dom).
\]
Clauses used in modelChecker1.pl for ∀:

\[
\text{satisfy(all}(X,\text{Formula}),\text{Model},G,\text{Pol}):\neg \\
\text{satisfy(not(some}(X,\text{not(Formula)})),\text{Model},G,\text{Pol}).
\]

Clauses for existential quantification ∃:

\[
\text{satisfy(some}(X,\text{Formula}),\text{model}(D,F),G,\text{pos}):\neg \\
\text{memberList}(V,D), \\
\text{satisfy(Formula},\text{model}(D,F),[g(X,V)|G],\text{pos}).
\]

\[
\text{satisfy(some}(X,\text{Formula}),\text{model}(D,F),G,\text{neg}):\neg \\
\text{setof}(V,(\text{memberList}(V,D),\text{satisfy(Formula},\text{model}(D,F),[g(X,V)|G],\text{neg})),\text{Dom}), \\
\text{setof}(V,\text{memberList}(V,D),\text{Dom}).
\]

Alternative clauses for ∀ without reusing other connectives:

\[
\text{satisfy(all}(X,\text{Formula}),\text{model}(D,F),G,\text{pos}):\neg \\
\text{setof}(V,(\text{memberList}(V,D),\text{satisfy(Formula},\text{model}(D,F),[g(X,V)|G],\text{pos})),\text{Dom}), \\
\text{setof}(V,\text{memberList}(V,D),\text{Dom}).
\]

\[
\text{satisfy(all}(X,\text{Formula}),\text{model}(D,F),G,\text{neg}):\neg \\
\text{memberList}(V,D), \\
\text{satisfy(Formula},\text{model}(D,F),[g(X,V)|G],\text{neg}).
\]
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?

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

| Vincent loves Mia | \( \rightarrow \) | \( \text{LOVE(VINCENT, MIA)} \) |
| Everyone hates Butch | \( \rightarrow \) | \( \forall x. \text{HATE}(x, \text{BUTCH}) \) |

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:

```
S  Vincent loves Mia
   NP  VP
      Vincent loves Mia
         V  NP
              loves Mia
```

- 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.
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

- 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.

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

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

?- np(X,[]).
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...

Fortunately, we can use a notational extension of FOL that will make these tasks easy: the $\textit{lambda calculus}$. 
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 $\lambda$:

$\lambda x. \text{WOMAN}(x)$

- the prefix $\lambda x.$ binds the occurrence of $x$ in $\text{WOMAN}(x)$
- 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:

$\lambda x. \lambda y. \text{LOVE}(x, y)$
We can think of the lambda calculus as a tool dedicated to gluing together the items needed to build semantic representations.

\[ \lambda x.\text{WOMAN}(x) \]

- 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 \( \text{WOMAN}(x) \) indicates that \( \text{WOMAN} \) 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]

\[ \lambda x.\text{WOMAN}(x)@\text{MIA} \]

- a compound expression of this sort refers to the application of the functor \( \lambda x.\text{WOMAN}(x) \) to the argument MIA.
β-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:

$$
\begin{align*}
\lambda x.\text{WOMAN}(x)@\text{MIA} & \leadsto \text{WOMAN}(\text{MIA}) \\
\lambda y.\lambda x.\text{HATE}(x, y)@\text{BUTCH} & \leadsto \lambda x.\text{HATE}(x, \text{BUTCH})
\end{align*}
$$

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

$$
\begin{align*}
\lambda v.\exists x.\left(\text{BOXER}(x) \land v@x\right) & @ \lambda x.\text{DANCE}(x) \leadsto \exists x.\left(\text{BOXER}(x) \land \lambda x.\text{DANCE}(x)@x\right) \\
& \leadsto \exists x.\left(\text{BOXER}(x) \land \text{DANCE}(x)\right)
\end{align*}
$$
Lambda Calculus for Semantic Construction

Lambda abstraction, functional application, and β-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 β-conversion to combine semantic representations compositionally.

  * Given a syntactic constituent $R$ with subparts $Ra$ and $Rb$, we need to specify which subpart is to be thought as the functor $F$ and which as the argument $A$.
  * We then construct the semantic representation of $R$ by functional application $F@A$

\[
\begin{array}{c}
  S \\
  \lambda x.\text{WALK}(x)@\text{VINCENT} \leadsto \text{WALK(\text{VINCENT})} \\
  \text{NP} & \text{VP} \\
  \text{VINCENT} & \lambda x.\text{WALK}(x) \\
  \text{Vincent} & \text{walks} \\
  F & A \\
  Ra & Rb \\
  \mathcal{R} \\
  \mathcal{F}@\mathcal{A}
\end{array}
\]
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

\[ \exists x.\text{BOXER}(x) \land \text{WALK}(x) \]

- What does each word contribute to this formula? And what is the contribution of the determiner?
A boxer walks: $\exists x. \text{BOXER}(x) \land \text{WALK}(x)$

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

For the determiner we then need three arguments:

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

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:

$$a, \text{some}: \lambda u. \lambda v. \exists x. (u@x \land v@x)$$
A boxer walks

\[ S \]
\[ \lambda v. \exists x. (BOXER(x) \land v@x) \ @ \ \lambda x. WALK(x) \]
\[ \leadsto \ \exists x. (BOXER(x) \land WALK(x)) \]

\[ NP \]
\[ \lambda u. \lambda v. \exists x. (u@x \land v@x) \ @ \ \lambda x. BOXER(x) \]
\[ \leadsto \ \lambda v. \exists x. (BOXER(x) \land v@x) \]

\[ Det \]
\[ \lambda u. \lambda v. \exists x. (u@x \land v@x) \]
\[ a \]

\[ N \]
\[ \lambda x. BOXER(x) \]
\[ boxer \]

\[ VP \]
\[ \lambda x. WALK(x) \]
\[ \text{walks} \]
We have represented quantified NPs as functors:

\[
\text{a boxer: } \lambda v. \exists x. (\text{BOXER}(x) \land v@x)
\]

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:

\[
\text{Mia: } \lambda u. (u@\text{MIA})
\]

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

\[
\text{loves: } \lambda v. \lambda x. (v@\lambda y. \text{LOVE}(x, y))
\]

Let us see why...
Vincent loves Mia

\[ S \]
\[ \lambda u. (u@\text{VINCENT})@\lambda x.\text{LOVE}(x, \text{MIA}) \]
\[ \leadsto \lambda x.\text{LOVE}(x, \text{MIA})@\text{VINCENT} \]
\[ \leadsto \text{LOVE(\text{VINCENT, MIA})} \]

NP
\[ \lambda u. (u@\text{VINCENT}) \]
\text{Vincent}

VP
\[ \lambda v. \lambda x. (v@\lambda y.\text{LOVE}(x, y))@\lambda u. (u@\text{MIA}) \]
\[ \leadsto \lambda x. (\lambda u. (u@\text{MIA})@\lambda y.\text{LOVE}(x, y)) \]
\[ \leadsto \lambda x.\text{LOVE}(x, \text{MIA}) \]

V
\[ \lambda v. \lambda x. (v@\lambda y.\text{LOVE}(x, y)) \]
\text{loves}

N
\[ \lambda u. (u@\text{MIA}) \]
\text{Mia}
Implementing Lambda Calculus
DCGs for Semantic Construction

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

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

Here are the main syntactic rules decorated with semantic representations:

\[
\begin{align*}
\text{s(app(NP,VP))} & \rightarrow \text{np}(\text{NP}), \text{vp}(\text{VP}). \\
\text{np(app(Det,Noun))} & \rightarrow \text{det}(\text{Det}), \text{noun}(\text{Noun}). \\
\text{np(PN)} & \rightarrow \text{pn}(\text{PN}). \\
\text{vp(IV)} & \rightarrow \text{iv}(\text{IV}). \\
\text{vp(app(TV,NP))} & \rightarrow \text{tv}(\text{TV}), \text{np}(\text{NP}).
\end{align*}
\]

And here are some lexical entries:

\[
\begin{align*}
\text{noun(\text{lam}(X,\text{woman}(X)))} & \rightarrow [\text{woman}]. \\
\text{iv(\text{lam}(Y,\text{walk}(Y)))} & \rightarrow [\text{walks}]. \\
\text{tv(\text{lam}(X,\text{lam}(Y,\text{app}(X,\text{lam}(Z,\text{like}(Y,Z))))))} & \rightarrow [\text{likes}]. \\
\text{pn(\text{lam}(P,\text{app}(P,\text{mia})))} & \rightarrow [\text{mia}]. \\
\text{det(\text{lam}(P,\text{lam}(Q,\text{all}(X,\text{imp}(\text{app}(P,X),\text{app}(Q,X))))))} & \rightarrow [\text{every}]. \\
\text{det(\text{lam}(P,\text{lam}(Q,\text{some}(X,\text{and}(\text{app}(P,X),\text{app}(Q,X))))))} & \rightarrow [\text{a}].
\end{align*}
\]

This code can be found in the file $\text{experiment3.pl}$
Implementing $\beta$-conversion

We can use the above DCG for semantic construction:

```prolog
?- 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:

```prolog
?- 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:

\[
\lambda u. \exists x. (\text{WOMAN}(x) \land u@x) \\
\lambda v. \exists z. (\text{WOMAN}(z) \land v@z)
\]

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 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:

<table>
<thead>
<tr>
<th>lambda.pl</th>
<th>SYNTAX</th>
<th>SEMANTICS</th>
</tr>
</thead>
<tbody>
<tr>
<td>LEXICON</td>
<td>the Lexicon</td>
<td>the Semantic Lexicon</td>
</tr>
<tr>
<td></td>
<td>englishLexicon.pl</td>
<td>semLexLambda.pl</td>
</tr>
<tr>
<td>GRAMMAR</td>
<td>the Syntax Rules</td>
<td>the Semantic Rules</td>
</tr>
<tr>
<td></td>
<td>englishGrammar.pl</td>
<td>semRulesLambda.pl</td>
</tr>
</tbody>
</table>
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:

\[
s([\text{sem:Sem}]) \rightarrow \\
\quad \text{np}([\text{num:Num,sem:NP}]), \\
\quad \text{vp}([\text{num:Num,sem:VP}]), \\
\quad \{\text{combine}(s:Sem,[\text{np:NP,vp:VP}])\}.
\]

The Lexicon

- The general format of a lexical entry is \text{lexEntry}(\text{Cat}, \text{Features}), where \text{Cat} is the syntactic category and \text{Features} is a list of features.
- For example, the entries for the intransitive verb ‘to walk’ are:

\[
\text{lexEntry}(iv,[\text{symbol:walk,syntax:[walk],inf:inf,num:sg}]). \\
\text{lexEntry}(iv,[\text{symbol:walk,syntax:[walks],inf:fin,num:sg}]). \\
\text{lexEntry}(iv,[\text{symbol:walk,syntax:[walk],inf:fin,num:pl}]).
\]
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:

```prolog
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

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

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

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

This is the main level program:

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

```prolog
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.

```prolog
?- 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.