Abstract

In this work, I implemented a program which could answer particular questions based on input sentences stated according to the grammar rules defined. Input sentences were chosen to imply basic arithmetic operations, hence questions were answered by performing basic arithmetic operations by the program. The application program is based on semantic representations of the sentences and implemented in Prolog by using DCG (Definite Clause Grammars). In the process of constructing semantic representations, Lambda Calculus is used.
1. INTRODUCTION

Natural Language Processing (NLP) is the use of computers to understand human (natural) languages such as English. “to understand” means that the computer can recognize and use information expressed in a human language [5].

The structure of a human language contains five levels: Phonology (sound), Morphology (word formation), Syntax (sentence structure), Semantics (meaning) and Pragmatics (use of language in context) [5]. In this work, the levels “Syntax and Semantics” are considered.

This paper includes an introduction about the notation called Definite Clause Grammars which is used in the program, explanations about how the meanings of the sentences are represented in the program, explanations about an approach based on using Lambda Calculus to combine semantic representations and the methods used in the program to provide answers to particular questions.
2. DEFINITE CLAUSE GRAMMARS

The following explanations are about a notation which is used in the program [1,2].

2.1. The notation

DCG (Definite Clause Grammar) is a notation to write grammars. Consider the example

\[
\begin{align*}
s & \rightarrow np, vp. \\
np & \rightarrow det, n. \\
v & \rightarrow \text{v}. \\
vp & \rightarrow v, np. \\
vp & \rightarrow v. \\
det & \rightarrow [\text{the}]. \\
det & \rightarrow [\text{a}]. \\
n & \rightarrow [\text{woman}]. \\
n & \rightarrow [\text{man}]. \\
v & \rightarrow [\text{shoots}].
\end{align*}
\]

The query “\(s([a, \text{woman}, \text{shoots}, a, \text{man}], [])\)” can be posed to find out whether “a woman shoots a man” is a sentence or not. The query means to ask whether we can get a sentence by consuming the symbols in the first list leaving the symbols in the second list behind. Hence, we ask whether the symbols in the first list make a sentence or not (leaving nothing behind, that is using all symbols in the list). It is also possible to pose the query \(s(X, [])\) to generate all sentences. The queries can be posed for other grammatical categories as well (np(X, [])).

Prolog translates this notation into difference lists. The respond to the query listing(s) explains what Prolog translates \(s \rightarrow np, vp\) into:

\[
s(A, B) :- \\
np(A, C), \\
vp(C, B).
\]

This rule says that “use some of the first argument to make a noun phrase, then use the rest to make a verb phrase. The rule det \(\rightarrow [\text{the}]\) may either be translated into det([$\text{the}$|W], W) or det(A,B) :- 'C'(A,$\text{the}$,B).

depending on the Prolog implementation. (In the second case the idea is basically the same which says “get a B from a A by consuming a the”.)

As seen, DCG notation hides this difference list variables and provides a simple way to write grammars.
2.2. Obtaining meaning with grammar rules

There is no fixed way about how to represent the meaning of natural language. But, according to the application, a good choice is found.

Consider an example sentence “Jack walks”. In logic, it is natural to express the meaning of this sentence as a Prolog term “walks(jack)”. Since “walks” is an intransitive verb the predicate “walks” has one argument. So, another sentence “Jack likes Mary” can be represented as “likes(jack,mary)”. The predicate has two arguments since the corresponding verb is transitive. Below is the grammar which is created by modifying the first example grammar rules slightly to cover the two example sentence.

```
s --> np,vp.
np --> pn.
vp --> iv.
vp --> tv,np.
iv--> [walks].
tv --> [likes].
pn --> [jack].
pn --> [mary].
```

The grammar rules should provide a way to build the representations mentioned above. These grammar rules can only say which sentences are acceptable by the rules defined. The grammar can be modified to provide a feature to build representations of the meanings of sentences. The way of doing it is adding extra arguments to the rules where needed. For example, the meaning of the proper noun “Jack” is “jack”. So, to extract the meaning “jack” from the proper noun “Jack”, an argument is added to the rule “pn” and the new rule becomes

```
 pn(jack) --> [jack].
```

The meaning of the intransitive verb “walks” can be defined as “walks(X)” where X is a variable and its value is to contain a noun phrase which will be available from the context. So, the DCG rule for “walks” is

```
iv(walks(X))--> [walks].
```

The meaning of the sentence “walks(jack)” must to be constructed by these two meanings “jack and walks(X)” in a way that X must be equal to “jack”. The syntactic structure of the sentence specifies how the meanings of phrases are incorporated into the meaning of the sentence. The meaning from the proper noun is taken to the noun phrase since noun phrase consists of proper noun.
Similarly, the meaning from the intransitive verb is taken to the verb phrase. Finally, since the sentence consists of a noun phrase and a verb phrase, the meanings from noun phrase and verb phrase should be combined. So, in the rule “s” we can combine the meanings. If we make the argument X in term “walks(X)” accessible for instantiation by making it visible from the outside of the term, we can easily combine the meanings. To achieve this, the rule for the intransitive verb is modified like in the following.

\[
\text{iv}(X, \text{walks}(X)) \rightarrow \text{[walks]}. \\
\]

The verb phrase takes the meaning from the intransitive verb with the rule

\[
\text{vp}(X, \text{VP}) \rightarrow \text{iv}(X, \text{VP}). \\
\]

And the rule for the sentence becomes

\[
\text{s} \ (\text{VP}) \rightarrow \text{np}(X), \text{vp}(X, \text{VP}). \\
\]

The whole grammar to cover the sentence “jack walks” then becomes

\[
\text{s}(\text{VP}) \rightarrow \text{np}(X), \text{vp}(X, \text{VP}). \\
\text{np}(X) \rightarrow \text{pn}(X). \\
\text{vp}(X, \text{VP}) \rightarrow \text{iv}(X, \text{VP}). \\
\text{iv}(X, \text{walks}(X)) \rightarrow \text{[walks]}. \\
\text{pn}(\text{jack}) \rightarrow \text{[jack]}. \\
\]

If we pose the query \(s(A,[\text{jack, walks}],[]))\), we get the respond \( A=\text{walks(jack)} \) as intended meaning carried in the argument in the rule for the sentence.

With the same idea, we can get the meaning representation of the second sentence by adding the following rules to the grammar.

\[
\text{vp}(P1, TV) \rightarrow \text{tv}(P1, P2, TV), \text{np}(P2). \\
\text{tv}(X, Y, \text{likes}(X, Y)) \rightarrow \text{[likes]}. \\
\text{pn}(\text{mary}) \rightarrow \text{[mary]}. \\
\]

And the respond to the query \(s(A,[\text{jack, likes, mary}],[]))\) becomes

\( A = \text{likes(jack, mary)} \)

as intended.
3. GRAMMAR RULES

The following explanations are about what kind of semantic representations are built based on the input and how the grammar rules are implemented.

The grammar rules shown in Appendix 3, are designed based on the lambda calculus to handle four kinds of sentences. The representations built for each kind of sentences can be seen from the examples below:

- **jack finds 3 apples**  →  add(jack, obj(3, apples))
- **jack eats 3 apples**  →  sub(jack, obj(3, apples))
- **jack gives 3 apples to bill**  →  sub(jack, obj(3, apples))
  add(bill, obj(3, apples))
- **jack takes 3 apples from bill**  →  add(jack, obj(3, apples))
  sub(bill, obj(3, apples))
- **jack bill bob share 12 apples**  →  add(jack, obj(4, apples))
  add(bill, obj(4, apples))
  add(bob, obj(4, apples))
- **jack takes 3 apples from each of the 4 people**  →  add(jack, obj(12, apples))

As seen for the third and the fourth sentences two constructs are built and for the fifth sentence, the number of constructs are the same as the number of subjects in the sentence. All mathematical operations implied by the input sentences are passed to the program as addition or subtraction.

Grammar rules are defined in a way such that verbs “find, take, share” imply the mathematical operation addition, “eat, give” imply subtraction, the word group “from each of the 4 people” implies multiplication, “share” implies division also, the preposition “to” implies addition for the noun phrase (a proper noun) following it and the preposition “from” implies subtraction for the noun phrase following it. What is implied is looked up from “lexicon” which contains the words in sentences and some information regarding the words in the program. For example: lexicon(transitiveverb, add, [finds]).
The grammar should be easy to upgrade, so to have that property, the grammar can be divided into parts where different tasks are performed [3,4]. The grammar has three parts in the program: a lexicon, DCG-rules, semantic predicates. Consider the example below to see the tasks of these parts. The figure shows how to analyze the sentence “Mary walks”:

The left-hand side shows the syntactic part of the grammar. The syntactic analysis includes two steps. First, a lexicon look-up tells that ‘Mary’ is a proper name (PN) and that ‘walks’ is an intransitive verb (IV). Second, the two DCG-rules NP→PN and VP→IV say that ‘Mary’ is also a noun phrase (NP) and that ‘walks’ is also a verb phrase (VP). The rule S→NP VP says that ‘Mary walks’ is a sentence.
The right-hand side shows the semantic part of the grammar. The lambda expressions come from the semantic predicates which are called in the lexical level in the grammar. The lambda expressions provide the semantic representations of the lexical items. And the combine-rules which are defined as “applications” in the arguments in DCG-rules specify how to obtain the semantic representations from the representations of the children nodes. To have the semantic representation of the sentence, the combine rule is to apply the representation of the left child to the representation of the right child and as a result we get “(\(\lambda u.\text{u@mary}\))(\(\lambda x.\text{walk( }x\text{)}\))”.

Finally, another step, namely doing beta-conversion is needed to get the representation of the example sentence which is “walk(mary)”.

The division of the grammar into syntactic and semantic parts provides an efficient way to build representations of meaning, because after building DCG-rules, it only remains to define semantic predicates to give meanings to the lexical entries and to combine representations to form a semantic construct.

As explained previously, the root node (the sentence rule) in the syntax tree has the representation of the sentence which is formed by applying the representation of the noun phrase to the representation of the verb phrase. Lambda expressions are incorporated in the program in the lexical level. The semantic predicates provide lambda expressions which are appropriate for each word according to the syntactic structure of the grammar. In the dcg-rules representations are combined by performing applications. So, from the lexicon up to the sentence rule, the representation of the sentence is obtained as lambda expression (or lambda expressions). Then, beta-conversion is performed.

The grammar rules in the program contain the following rules (they are available in Appendix 3)

\[
\text{s(B)} \rightarrow \text{np(NP), vp1(VP), } \{\text{betaConvert(app(NP,VP),B),!}\}.
\]

\[
\text{s(A)} \rightarrow \text{np(NP), vp1(VP,Op,N), } \{\text{betaConvert(app(NP,VP),B),compose(Z,Op,[B,N,A]),Z,!}\}.
\]

\[
\text{s(R)} \rightarrow \text{np(NP), vp2(VP,NP2), } \{\text{betaConvert(app(NP,VP),B),betaConvert(NP2,C),R=[B,C],!}\}.
\]

\[
\text{s(W)} \rightarrow \text{nps(X,Y), vp1(VP,Op), } \{\text{all(X,VP,A),appBetaConvertAll(A,C),compose(Z,Op,[C,Y,W]),Z,!}\}.
\]

In the second and the fourth sentence rules, the verbs imply multiplication and division respectively. By the lexicon lookup, the operation is specified, passed up to the sentence rule with the extra argument “Op” in vp and by the predicate “compose”, the predicate defined to perform the arithmetic operation with the proper arguments is called. As seen from the following
lexicon(transitiveverb,add,divide,[share]).
lexicon(pphrase,multiply,N,[from,each,of,the,N,people]).

the lexicon lookup for the verb “share” and the word group “from each of the N people” find out that an arithmetic operation is to be done on the semantic representation for the sentence. In the fourth rule, more than one subject appear in a sentence. So, more than one semantic representations are constructed. And beta-conversion is performed for all of them.

It is possible to relate verbs with particular sentences by adding an extra argument to verb phrases to indicate the sentence number like in the following rules.

s1(B)--> np(NP), vp1(VP,1),{betaConvert(app(NP,VP),B),!}.
vp1(app(TV,NP),X)-->tv(TV,X),np(NP).
tv(TV,X)-->{lexicon(transitiveverb,Sym,Word,X),tvSem(Sym,TV)}, Word.
lexicon(transitiveverb,sub,[eats],1).
lexicon(transitiveverb,add,[finds],1).

(All changes to the grammar can be seen from the Appendix 1)

As seen from the following query examples, undesired sentences can be eliminated in this way.
?- s2(A,[jack,share,3,apples,to,bill],[]).
no
?- s4(A,[jack,gives,3,apples,from,each,of,the,4,people],[]).
no

It is also possible to make the number of subjects variable for each kind of sentences like in the third kind of sentence in the first grammar in Appendix 3. In that case it is needed to use verbs like “find, eat, take...” instead of “finds, eats, takes” with the plural subjects. To achieve this distinction, an extra argument is added to verb phrases which has the value “plural” or “singular” according to the number of subjects. The grammar rules modified to have semantic representations like the following are in Appendix 2. (The grammar rules for these sentences cover the sentences with one subject as well)
For the sentence “jack bill bob find 3 apples” representations are:

\[
\begin{align*}
\text{add}(\text{jack}, \text{obj}(3, \text{apples})) \\
\text{add}(\text{bill}, \text{obj}(3, \text{apples})) \\
\text{add}(\text{bob}, \text{obj}(3, \text{apples}))
\end{align*}
\]

For the sentence “jack gives 3 apples to bill 4 bananas to bob” representations are:

\[
\begin{align*}
\text{add}(\text{bill}, \text{obj}(3, \text{apples})) \\
\text{add}(\text{bob}, \text{obj}(4, \text{bananas})) \\
\text{sub}(\text{jack}, \text{obj}(3, \text{apples})) \\
\text{sub}(\text{jack}, \text{obj}(4, \text{bananas}))
\end{align*}
\]

For the sentence “jack bill bob take 4 apples from anne” representations are:

\[
\begin{align*}
\text{sub}(\text{anne}, \text{obj}(12, \text{apples})) \\
\text{add}(\text{jack}, \text{obj}(4, \text{apples})) \\
\text{add}(\text{bill}, \text{obj}(4, \text{apples})) \\
\text{add}(\text{bob}, \text{obj}(4, \text{apples}))
\end{align*}
\]

For the sentence “jack bill bob give 3 apples to anne 4 apples to mary” representations are:

\[
\begin{align*}
\text{add}(\text{anne}, \text{obj}(9, \text{apples})) \\
\text{add}(\text{mary}, \text{obj}(12, \text{apples})) \\
\text{sub}(\text{jack}, \text{obj}(3, \text{apples})) \\
\text{sub}(\text{jack}, \text{obj}(4, \text{apples})) \\
\text{sub}(\text{bill}, \text{obj}(3, \text{apples})) \\
\text{sub}(\text{bill}, \text{obj}(4, \text{apples})) \\
\text{sub}(\text{bob}, \text{obj}(3, \text{apples})) \\
\text{sub}(\text{bob}, \text{obj}(4, \text{apples}))
\end{align*}
\]

For the sentence “jack bill bob anne take 4 apples from each of the 5 people” representations are:

\[
\begin{align*}
\text{add}(\text{bill}, \text{obj}(20, \text{apples})) \\
\text{add}(\text{bob}, \text{obj}(20, \text{apples})) \\
\text{add}(\text{anne}, \text{obj}(20, \text{apples})) \\
\text{add}(\text{jack}, \text{obj}(20, \text{apples}))
\end{align*}
\]

The grammar rules used to generate the semantic representations above contain these sentence rules:

\[
\begin{align*}
s1(C) & \rightarrow \text{nps}(X,Y), \text{vp1}(V, \text{V}, 1), !\{\text{all}(X, \text{VP}, A), \text{appBetaConvertAll}(A, C), (Y > 1 \rightarrow V = \text{plural}; V = \text{singular}), !\}. \\
s2(R) & \rightarrow \text{nps}(X,Y), \text{vp2}(\text{VP}, \text{NP2}, \text{V}, 2), !\{\text{allsubj}(X, \text{VP}, A), \text{appBetaConvertAll}(A, C), \text{appBetaConvertAll}(\text{NP2}, D), \text{multiply}(D, Y, W), (Y > 1 \rightarrow V = \text{plural}; V = \text{singular}), \text{append}(W, C, R), !\}. \\
s3(W) & \rightarrow \text{nps}(X,Y), \text{vp1}(\text{VP}, \text{Op}, \text{plural}, 3), !\{\text{all}(X, \text{VP}, A), \text{appBetaConvertAll}(A, C), \text{compose}(Z, \text{Op}, [C, Y, W]), Z, Y > 1, !\}.
\end{align*}
\]
s4(A)→nps(X,Y),vp1(VP,Op,N,V,4),!(all(X,VP,D),appBetaConvertAll(D,C), compose(Z,Op,[C,N,A]),Z,(Y>1→V=plural;V=singular!),).  

The variable Y in the predicate nps contains the number of subjects. If it is greater than 1, from the lexicon verbs defined as plural can be included in the sentence. There are two examples from the lexicon below.

lexicon(transitiveverb,singular,add,[takes],4).
lexicon(transitiveverb,plural,sub,[eat],1).
4. LAMBDA CALCULUS

The following explanations are about how to represent the meaning of a sentence and how to use Lambda Calculus in the process of constructing semantic representations [4]. An application is given to illustrate the notion explained. And an approach to implement beta conversion is explained as well. The approach is adapted from [3].

4.1. The Semantic representation of a sentence

Given a sentence in English, a systematic way of constructing its semantic representation could be followed. First of all, let’s inspect how to translate the example sentence "Jack loves Mary" into a semantic representation. The semantic content of the sentence can be denoted by the relation love(jack,mary). The constant jack is added for the proper name "Jack", the relation symbol love( , ) is added for the transitive verb "loves" and the constant mary is added for "Mary". As seen, words in the sentence contribute everything to build the semantic representations of the sentence. (Namely, meaning representations are built on the words in the lexicon.) Additionally, the formula love(jack,mary) is chosen not love(mary,jack). The reason why love(jack,mary) is the appropriate formula is that it indicates correctly the syntactic and semantic structure of the sentence. The sentence consists of the noun phrase "Jack" and the verb phrase "loves Mary". The verb phrase consists of the transitive verb "loves" and the object "Mary" as seen below.
“jack” needs to be put into the first argument in the relation love( , ), because the first argument is reserved for the semantic representations of noun phrases to be combined with verb phrases to construct a sentence. Consequently, to construct meaning representations, information needed are obtained from the words (lexical items) in a sentence and the syntactic structure. The words are the basic ingredients in the representation and the syntactic structure specifies how to combine what is taken from the substructures of a sentence as semantic entity such as “jack,mary,love( , )”.

It is possible to automate the process of associating semantic representations with natural language expressions by means of lambda calculus. By means of lambda calculus, to combine semantic representations could be easy. The lambda calculus is a tool for controlling the process of making substitutions. For example, the lambda expression $\lambda x.\text{man}(x)$ can be interpreted in a way that there is a missing information in the formula $\text{man}(x)$ and the name of the missing part is $x$. Hence, lambda expressions can be used to mark where to make substitutions while constructing the semantic representations. In other words, the variable $x$ in the lambda expression can be seen as a mark in the place where information is missing. When
the missing information is available, it is substituted for the variable. The substitution process is called beta conversion. "@" is used to indicate the substitution to be performed. The expression \((\lambda x. \text{man}(x))@\text{jack}\) is called application. The left-hand expression is the function and the right-hand expression is the argument and the whole expression indicates that the argument jack will be substituted for \(x\) in \(\text{man}(x)\). By performing beta conversion we get “\(\text{man}(@\text{jack})\)”. "jack" which represents the proper name Jack is used as argument. But, according to the syntactic structure, when it is a noun phrase we can use it as function. To see the reason why it should be used as function in an application, consider the first example sentence “Jack loves Mary”. The tree structure which shows how the semantic representation for the example sentence is built is in Figure 2. Every parent node in the tree is associated with application that has the representation in the left child node as function and the representation in the right child node as argument. Hence, to combine semantic representations which are obtained from substructures of the sentence becomes very easy. The combination is performed by applying function to argument and beta converting (substituting) at the end. Appropriate lambda expressions are defined in the lexical level. The expression \(\lambda x. x@\text{jack}\) is chosen to represent proper name Jack in the noun phrase. The expression states that the argument jack will be placed in a formula. That expression is preferred, since in the parent node it will be applied to the semantic construct coming from the verb phrase node. The semantic representation for the transitive verb “loves” is applied to the semantic representation for the (object) noun phrase and the semantic representation for the (subject) noun phrase is applied to the result. So, the expression \(\lambda x. \lambda y. \text{love}(x,y)\) is not suitable for those applications. Instead, the expression \(\lambda w. \lambda z. (w@\lambda x. \text{love}(z,x))\) is used to perform applications as mentioned.
Figure 2: how the semantic representation for the example sentence is built

As seen from the figure above, the representation in the left child node is applied to the representation in the right child node in the parent node in every level in the tree, which is combining representations in a simple way. The relation to denote the semantic content of the sentence is constructed easily when the appropriate lambda expressions are provided. So, most of the work is done before the combination, that is, constructing appropriate lambda expressions.
4.2. Application:

I constructed grammar rules to get semantic representations for four kinds of sentences based on the notion described above.

For sentences like “Jack finds 3 apples”, the semantic representation “add(jack, obj(3, apples))” is constructed as shown in the following figure.
For sentences like “Jack gives 3 apples to Bill”, the semantic representation “sub(jack, obj(3, apples)), add(bill, obj(3, apples))” is constructed as shown in the following figure.

```
s(NP@VP.add(bill, obj(3, apples)))
NP@VP = (λu.u@jack)@(λz.sub(z, obj(3, apples))) = (λz.sub(z, obj(3, apples))}@jack = sub(jack, obj(3, apples))
```

```
np(λu.u@jack) np2(VP,NP2)
VP=λw. λz.(w@ λx.sub(z,x))@ λa.a@obj(3,apples)
=λz.(λa.a@obj(3,apples))@ λx.sub(z,x)
=λz.(λx.sub(z,x))@obj(3,apples)
VP=λz.(sub(z, obj(3, apples)), NP2= add(bill, obj(3, apples))
```

```
tv(λw. λz.(w@ λx.sub(z,x)))
np3(NP,NP2)
NP2= (λa.a@obj(3,apples))@ (λs.(add(bill, s))
= (λs.(add(bill, s))@obj(3,apples)
= add(bill, obj(3, apples))
NP= λa.a@obj(3,apples)
```

```
np(Det@Cn)= np(λa.a@obj(3,apples))
```

```
np2(P@NP)
(λy.(λs.(y@λi.add(i,s))))@
(λu.u@bill)=
(λs.(λu.u@bill)@λi.add(i,s)))=
(λs.(λi.add(i,s))@bill)=
(λs.(add(bill, s))= the argument
```

```
det(λv.v@(λx.(λa.a@obj(3,x)))) cn(λu.u@apples) prep(λy.(λs.(y@λi.add(i,s))))) np(λu.u@bill)
```
For sentences like “Jack, Bob, Bill share 6 apples”, the semantic representation “add(jack, obj(2, apples)), add(bill, obj(2, apples)), add(bob, obj(2, apples))” is constructed as shown in the following figure.
For sentences like “Jack takes 3 apples from each of the 5 people”, the semantic representation “add(jack, obj(15, apples))” is constructed as shown in the following figure.

The grammar rules above can be seen in Appendix 3.
4.3. Implementing lambda calculus:

In the implementation, lam(x,F) is used to represent \( \lambda x. F \) and app(F,A) is used to represent \( F@A \). An example query with the betaConvert predicate which is defined to implement beta-conversion (substitution) is seen below.

\[
\text{betaConvert(app(lam(a,app(a,mary)),lam(b,walk(b))),Q).}
\]
\[
Q=\text{walk(mary)}
\]

which is equivalent to \((\lambda a.a@mary)@\lambda b.\text{walk(b)} = (\lambda b.\text{walk(b)})@\text{mary} = \text{walk(mary)}\)

Beta-conversion can be implemented by making use of a stack that keeps track of all expressions that need to be used as arguments at some point. As in a stack, the information is stored in a last-in first-out way, a list in Prolog can be used as a stack, then the head of the list is the top of the stack and the last element put in the list will be the first one to be taken. Initially, the stack is empty. When the formula in the expression is an application to an argument, the argument is pushed onto the stack. If it is not an application, the top element is popped and substituted for the variable. If the expression is neither an application nor starting with lambda then the beta-conversion is finished. Below, it is seen how the approach which is based on using a stack to implement beta conversion is applied for the example:

<table>
<thead>
<tr>
<th>Expression</th>
<th>Stack</th>
</tr>
</thead>
<tbody>
<tr>
<td>app(lam(a,app(a,mary)),lam(b,walk(b)))</td>
<td>[]</td>
</tr>
<tr>
<td>lam(a,app(a,mary))</td>
<td>[lam(b,walk(b))]</td>
</tr>
<tr>
<td>app(lam(b,walk(b)),mary)</td>
<td>[]</td>
</tr>
<tr>
<td>lam(b,walk(b))</td>
<td>[mary]</td>
</tr>
<tr>
<td>walk(mary)</td>
<td>[]</td>
</tr>
</tbody>
</table>

As seen, when the expression is an application, the argument is pushed onto the stack. When it is not, the top element in the stack is substituted for the variable in the function.
The substitution in betaConvert predicate is performed by a predicate called substitute. An example call to substitute predicate:

\[
\text{substitute(new,old,p(old,p(p(old,p(old))))),Result).}
\]
\[
\text{Result} = p(new,p(p(new,p(new))))
\]

All variables in lambda expressions can be replaced by a constant like “v1” to ensure that all substitutions are performed correctly without implementing alpha-conversion. I placed constants in lambda expressions in the examples by hand. But instead, with the predicate “var” which builds constants like “v1,v2..”, new constants can be generated. (the definition of the predicate “var” and “substitute” can be seen in the module usepredicates in Appendix 3)

An example query:

\[
\text{var(K,N),assertion(N).}
\]
\[
K = v1,
\]
\[
N = 1
\]

The definition of the betaConvert predicate:

\[
\text{betaConvert(X,Y):-}
\]
\[
\text{betaConvert(X,Y,[]).}
\]
\[
\text{betaConvert(app(Functor,Argument),Result,Stack):-}
\]
\[
\text{betaConvert(Functor,Result,[Argument|Stack]).}
\]
\[
\text{betaConvert(lam(X,Formula),Result,[A|Stack]):-}
\]
\[
\text{substitute(A,X,Formula,New),}
\]
\[
\text{betaConvert(New,Result,Stack).}
\]
\[
\text{betaConvert(X,X,__).}
\]

The predicate is called with the expression in its first argument, so as seen if it is an application, the argument in the application is pushed onto the stack. If it is not, substitution is performed. Each occurrences of the variable in the formula is replaced by the top element in the stack. And when the expression is neither an application nor starting with lambda then the beta-conversion is finished.
5. ANSWER GENERATION

In this section, how particular questions are answered in the application is explained.

Questions such as

"how many apples does jack have,
who has apples,
what does jack have,
who has apples more than (less than or equal to) bananas,
who has the most (the least) apples"

could be answered based on the representations derived from the input sentences during parsing. Enough information to be able to find the answer to the question is extracted from the question sentence.

During parsing, the input sentences are represented as

```
add(subject, obj(number, object)),
sub(subject, obj(number, object))```

constructs according to what is implied in the sentences. From the question sentences, information regarding object or subject which is asked in the question is extracted and that information is used to generate an appropriate answer based on “add( ), sub( )” constructs. In the “basedOnQuery” module I tried to answer questions like mentioned above in this way.

Although it is natural to answer questions based on only the constructs which carry related information to the question (by looking up “add( ), sub( )” constructs), I also tried to answer questions by creating new constructs which are of the form

```
has(subject, obj(number, object))```

The “has” constructs are available for all combinations of subjects and objects in the lexicon. Before looking at the question, the “has” constructs are created and they show “which subject has how many object” according to the input sentences. As mentioned, this is not a very efficient way of answering, since redundant “has” constructs are created as well. In the module “basedOnHasConstructs” I tried to answer questions such as

```
what does jack have,
who has apples,
how many apples does jack have```

by making use of “has” constructs.
5.1. The part of the program to answer questions

Below is the explanation of how the following result is obtained by the program. (The code to be explained can be seen in Appendix 3)

Jack takes 3 apples from each of the 5 people.
Jack finds 3 apples.
Jack finds 5 apples.
What does Jack have?
#Jack has 23 apples!
Jack eats 2 apples.
Jack gives 3 apples to Bill.
Anne, Jack, Bill, Bob share 24 apples.
What does Bill have?
#Bill has 9 apples!
Anne, Jack, Bill share 12 apples.
Jack takes 4 apples from Bill.
Bill takes 2 apples from Bob.
Anne takes 4 apples from each of the 5 people.
What does Anne have?
#Anne has 30 apples!
Bill finds 5 oranges.
What does Jack have?
#Jack has 32 apples!
Who has the most apples?
#Jack do have!
Bill gives 2 oranges to Bob.
What does Bill have?
#Bill has 11 apples 3 oranges!
Anne finds 6 bananas.
Jack takes 2 bananas from Anne.
Anne, Bob share 28 apples.
Who has oranges?
#Bob, Bill do have oranges!
How many apples does Jack have?
#Jack has 32 apples!
Who has apples?
#Jack, Bill, Anne, Bob do have apples!
How many oranges does Jack have?
#Jack does not have oranges!

Who has bananas?
#Anne, Jack do have bananas!
How many bananas does Anne have?
#Anne has 4 bananas!
What does Jack have?
#Jack has 32 apples 2 bananas!
What does Anne have?
#Anne has 4 bananas!
What does Bill have?
#Bill has 11 apples 3 oranges!
What does Bob have?
#Bob has 2 oranges 18 apples!
Who has more apples than bananas?
#Jack, Anne do have!
Bob takes 4 oranges from each of the 4 people.
Who has apples equal to oranges?
#Bob do have!
Who has apples less than bananas?
#Noone has!
Who has oranges less than apples?
#Bill do have!
Who has the most apples?
#Jack, Anne do have!
Who has the most bananas?
#Anne do have!
Who has the most oranges?
#Bob do have!
Who has the least apples?
#Bill do have!
Who has the least bananas?
#Jack do have!
Who has the least oranges?
#Bill do have!
The main module in the program is the module called “generateAnswers”. The call below shows how to use it.

    answerAll( ' input.txt ',' output.txt ' ) .

The definition of the “answerAll” predicate summarizes what is being done by the module.

    answerAll( F, O ) :-
        tell( O ),
        readInput( F, E ),
        parse( E, [ ] ),
        told.

Firstly, the input file is read by the “readInput” predicate and all the sentences in the input is taken into a list. The elements of the list are like the following.

[ jack, takes, 3, apples, from, each, of, the, 5, people, ' . ' , jack, finds, 3, apples, ' . ' , jack, finds, 5, apples, ' . ' , what, does, jack, have, ? |... ]

As seen from the result above, answers are written between ‘#’ and ‘!’ . That result can be input to the program, since while reading from the input, what is between ‘#’ and ‘!’ are skipped. The “skip” predicate in the “input” module is called when ‘#’ is encountered to get the next sentence. (The predicates used to read from the file are in the module “input” in Appendix 3.)

The other predicate “parse” is called with the list which contains every sentences in the input and this predicate does parsing of sentences. The “parse” predicate distinguishes question sentences from the others by looking at the ‘.’ and ‘?’ at the end of the sentences. If it is parsing a sentence, it writes the sentence to the output file and inserts the result of parsing which is the semantic representation of the sentence such as

    add( jack, obj( 4, apples ) )

into the list in its second argument. Otherwise, the sentence to be parsed is a question sentence, it writes the question sentence to the output file and calls a predicate to provide an answer. The predicate which is to find an answer and write it to the output file between ‘#’ and ‘!’ is “answer_basedOnQuery”. The predicate is called with the information extracted from the question sentence by parsing it and the list of constructs denoting the meaning of the
sentences before the question sentence. For example, the predicate has a list like

\[ \text{[add( jack, obj( 3, apples ), sub( bill, obj( 4, bananas ) ), |...]} \]

in its first argument and a construct like

\text{quest1(Object,Subject)}

in its second argument. While parsing a question sentence, information needed to answer the question is extracted from it. Below is what is extracted from questions mentioned above.

\text{how many apples does jack have,} \rightarrow \text{quest1(Object,Subject),}
\text{who has apples,} \rightarrow \text{quest2(Object),}
\text{what does jack have,} \rightarrow \text{quest3(Subject),}
\text{who has apples more than (less than or equal to) bananas,} \rightarrow \text{quest4(Object1,C,Object2),}
\text{who has the most (the least) apples} \rightarrow \text{quest5(C,Object)}

( \text{C may have the value of “more, less, equal, most, least”.} )

The arguments of the “quest( )” constructs which represent question sentences are supplied by the grammar rules for question sentences during parsing of them as seen from the module “questions” in Appendix 3.

“answer_basedOnQuery” predicate calls appropriate predicates according to the type of the question which is indicated by the “quest( )” construct in its second argument.

To answer the first kind of question, semantic constructs ( “add( ), sub( )” ) representing the sentences before the question are scanned and the ones containing the information from the question (Object, Subject) are taken into account. For example, if the question is “how many apples does jack have”, the semantic constructs containing “apples and jack” are considered and the numbers in them are collected. The numbers coming from “sub( )” constructs are multiply by -1 and added to the ones coming from add( ) constructs. The answer is formed by using “Object, Subject” information from the question and the number calculated, then printed. If the number calculated is ‘0’ then the answer “Subject does not have Object” is printed. The number calculated can have the value ‘0’, if the sum of the numbers is ‘0’ or there is no construct containing the “Object, Subject” from the question as
seen from the following example results (in second case, the predicate “padd” which finds the numbers in the constructs related to the question, has an empty list for numbers in its argument and the “sum” predicate gives the result ‘0’ for the empty list as seen from the module “usepredicates” in Appendix 3).

To answer the second kind of question, firstly the semantic constructs for the sentences before the question sentence are scanned to find the subjects. The subjects are found in the semantic constructs having the object in the “quest( )” construct for the question and kept in a list. Each element of the list is processed in the same way as explained previously to find the number of object in the question for each of the subjects, if the number is greater than zero, the corresponding subject is included in the answer. If the list of subjects is empty or the result of the calculations for each subject in the list is ‘0’, the message “noone has Object” is printed as illustrated in the following.

To answer the third kind of question, similarly to what is done for the second kind of question, firstly a list is used to keep all objects which occur with the subject from the quest3(Subject) in the semantic constructs for the sentences before the question sentence. Secondly, again in the same way as mentioned above, the number of each object in the list for the subject is calculated by scanning the semantic constructs with the object and subject. If the number of an object in the list is greater than ‘0’, the object is included in the answer. When the list of the objects is empty or the result of the calculations for each object in the list
is ‘0’, the message “Subject does not have anything” is printed as illustrated in the following.

\[
\begin{align*}
&\text{anne bob bill share 9 apples .} \\
&\text{bill takes 2 apples from bob .} \\
&\text{what does jack have ?} \\
&\text{#jack does not have anything!}
\end{align*}
\]

To answer the fourth kind of question, the way to answer the second kind of question is used, since this kind of question contains two subquestions in the second kind and new constructs based on the semantic constructs for the sentences before the question sentence are formed. The new constructs are of the form “has(subject, obj(number, object))”. A construct like that denotes the answer to the question in the second kind. Namely, it is built in the way to answer the second kind of questions. The construct to represent the question quest4(Object1,C,Object2) contains the objects mentioned in the question and the comparison criteria. For each of the two objects, a list is stored to keep “has” constructs which are built based on the semantic constructs (add( ), sub( )) and represent “which subject have how many object”. After that, it is only needed to make the comparison specified in the question between the two lists which have the same subject. For each elements in the first list, the other list is scanned and the matches satisfying the comparison, having the same subjects are looked for. If such a match is not found, the message “noone has” is printed as illustrated in the following.

\[
\begin{align*}
&\text{anne bob bill share 9 oranges .} \\
&\text{who has bananas more than oranges ?} \\
&\text{#noone has!} \\
&\text{who has oranges less than apples ?} \\
&\text{#noone has!} \\
&\text{who has apples equal to bananas ?} \\
&\text{#noone has!}
\end{align*}
\]

To answer the fifth kind of question, again firstly the way to answer the second kind of question is followed then the comparison criteria is taken into account. The construct to represent the question quest5(C, Object) contains the object mentioned in the question and the

\[
\begin{align*}
&\text{anne bob bill share 9 oranges .} \\
&\text{anne finds 2 bananas .} \\
&\text{who has bananas more than oranges ?} \\
&\text{#noone has!}
\end{align*}
\]
comparison criteria. “has( )” constructs are built based on the semantic constructs which represent the sentences before the question sentence and have the object. Then, a list containing the has constructs is scanned by looking for the biggest or the smallest number greater than ‘0’ (according to the comparison criteria-most, least) and comparing elements with each other (starting from the first one). The subject(s) in the element(s) of the list which satisfies the criteria, is included in the answer. If there is no such “has( )” construct, a message like “noone has apples” is printed as illustrated in the following.

```
anne bob bill share 9 oranges.
  bob finds 4 bananas.
  who has the most apples?
  #noone has apples!
  who has the least apples?
  #noone has apples!
```

In the predicate “parse”, if the call to the predicate “answer_basedOnQuery” is replaced by the predicate “x”, the module “basedOnHasConstructs” can be used. As mentioned previously, only the first three kind of questions are evaluated in this module and a result like the following could be obtained.

```
jack takes 3 apples from each of the 5 people.
jack finds 3 apples.
jack finds 5 apples.
what does jack have?
#jack has 23 apples!
jack eats 2 apples.
jack gives 3 apples to bill.
anne jack bill bob share 24 apples.
what does bill have?
#bill has 9 apples!
anne jack bill share 12 apples.
jack takes 4 apples from bill.
bill takes 2 apples from bill.
anne takes 4 apples from each of the 5 people.
what does anne have?
#anne has 30 apples!
bill finds 5 oranges.
what does jack have?
#jack has 32 apples!
bill gives 2 oranges to bob.

what does bill have?
#bill has 3 oranges 11 apples!
anne finds 6 bananas.
jack takes 2 bananas from anne.
anne bob share 28 apples.
who has oranges?
#bob bill do have oranges!
how many apples does jack have?
#jack has 32 apples!
who has apples?
#anne bob bill jack do have apples!
how many oranges does jack have?
#jack does not have oranges!
who has bananas?
#anne jack do have bananas!
how many bananas does anne have?
#anne has 4 bananas!
what does jack have?
#jack has 2 bananas 32 apples!
```

The predicate “x” has the same arguments as the predicate “answer_basedOnQuery” which are the semantic constructs of all sentences in the input and the construct to represent the question like “quest1(Obj,Subj)”. The predicate “x” calls the appropriate predicates to answer
the corresponding question. In all predicates to answer the question “has( )” constructs are built first based on the semantic constructs for the input sentences for all combinations of the proper nouns and common nouns (subjects and objects) by lookups from the lexicon. Namely, they are “has(jack, obj(N1, apples)), has(jack, obj(N1, bananas)), has(jack, obj(N1, oranges)), has(bill, obj(N4, apples))...”. Having all these constructs, it is very easy to answer the questions. According to the constructs to represent the question, the “has()” constructs are scanned to give the answer. For the first kind of question, the “has” construct containing the object and the subject extracted from the question; for the second kind of question, the “has” constructs containing the object extracted from the question; for the third kind of question, the ones containing the subject extracted from the question are taken into account to form the answer. (The code for the module “basedOnHasConstructs” is in Appendix 4)
Conclusions

The program developed is able to answer particular questions related to the input. During syntactic analysis of the input sentences, enough information to answer possible questions is extracted from the input sentences as meaning. Since the meanings of the sentences are represented in such a way that all information implied by the sentences are included, it is easy to answer questions. The answer generation part of the program uses the semantic representations of the input sentences according to the information extracted from the question sentences.

The kinds (or forms) of the sentences evaluated by the program could be increased by extending the syntactic analysis and improving the method to construct semantic representations. For example, morphological parse of the sentences could be included in the program which provides a comprehensive way to analyse the sentence.

The program gets what is meant by the particular words by using previously defined information. If the program is supplied with more information about what may be implied by some particular words, it is possible to evaluate more kinds of sentences.
APPENDIX

Appendix 1:

\[
\begin{align*}
s1(B) & \rightarrow np(NP), vp1(VP, 1), \{\betaConvert(app(NP, VP), B), \}. \\
s2(R) & \rightarrow np(NP), vp2(VP, NP2, 2), \{\betaConvert(app(NP, VP), B), \betaConvert(NP2, C), R = \{B, C\}, \}. \\
s3(W) & \rightarrow nps(X, Y), vp1(VP, Op, 3), \{\alpha(X, VP, A), appBetaConvertAll(A, C), compose(Z, Op, [C, Y, W]), Z, \}. \\
s4(A) & \rightarrow np(NP), vp1(VP, Op, N, 4), \{\betaConvert(app(NP, VP), B), compose(Z, Op, [B, N, A]), Z. \}
\end{align*}
\]

\[
\begin{align*}
vp1(app(TV, NP), X) & \rightarrow tv(TV, X), np(NP). \\
v1(app(TV, NP), Op, X) & \rightarrow tv(TV, Op, X), np(NP). \\
v1(app(TV, NP), Op, N, X) & \rightarrow tv(TV, X), np(NP, Op, N). \\
v2(app(TV, NP), NP2, X) & \rightarrow tv(TV, X), np3(NP, NP2).
\end{align*}
\]

\[
\begin{align*}
tv(TV, X) & \rightarrow \{\text{lexicon(transitiveverb, Sym, Word, X)}, tvSem(Sym, TV)\}, \text{Word}. \\
tv(TV, Operation, X) & \rightarrow \{\text{lexicon(transitiveverb, Sym, Operation, Word, X)}, tvSem(Sym, TV)\}, \text{Word}.
\end{align*}
\]

\[
\begin{align*}
\text{lexicon(transitiveverb, sub, [eats], 1).} \\
\text{lexicon(transitiveverb, add, [finds], 1).} \\
\text{lexicon(transitiveverb, sub, [gives], 2).} \\
\text{lexicon(transitiveverb, add, [takes], 2).} \\
\text{lexicon(transitiveverb, add, [takes], 4).} \\
\text{lexicon(transitiveverb, add, divide, [share], 3).}
\end{align*}
\]

Appendix 2:

\[
\begin{align*}
s1(C) & \rightarrow nps(X, Y), vp1(VP, V, 1), \{\alpha(X, VP, A), appBetaConvertAll(A, C), (Y > 1 \rightarrow V = plural; V = singular), \}. \\
s2(R) & \rightarrow nps(X, Y), vp2(VP, NP2, V, 2), \{\alpha(subj(X, VP, A), appBetaConvertAll(A, C), appBetaConvertAll(NP2, D), multiply(D, Y, W), (Y > 1 \rightarrow V = plural; V = singular), append(W, C, R), \}. \\
s3(W) & \rightarrow nps(X, Y), vp1(VP, Op, plural, 3), \{\alpha(X, VP, A), appBetaConvertAll(A, C), compose(Z, Op, [C, Y, W]), Z, Y > 1, \}. \\
s4(A) & \rightarrow nps(X, Y), vp1(VP, Op, N, V, 4), \{\alpha(X, VP, A), appBetaConvertAll(A, C), compose(Z, Op, [C, N, A]), Z, (Y > 1 \rightarrow V = plural; V = singular), \}. \\
nps([X], Y) & \rightarrow np(X, Y). \\
nps(Zs, Z) & \rightarrow a(Xs, D), nps(Ys, E), [Z is D+E, append(Xs, Ys, Zs)]. \\
a(X, Y) & \rightarrow np3s(X, Y). \\
np(app(DET, CN)) & \rightarrow det(DET), cn(CN). \\
np(app(DET, CN), Op, N) & \rightarrow det(DET), cn(CN), pp(Op, N). \\
np3(NP, app(NP, NP2)) & \rightarrow np(NP), np2(NP2). \\
np2(app(P, NP)) & \rightarrow prep(P), np(NP). \\
np3s([NP], [NP2]) & \rightarrow np3s(NP, NP2). \\
np3s(NP, NP2) & \rightarrow a3(X, Y), b3(Z, D), [append(X, Z, NP), append(Y, D, NP2)]. \\
a3(X, Y) & \rightarrow np3s(X, Y). \\
b3(Z, D) & \rightarrow np3s(Z, D).
\end{align*}
\]
vp1(app(TV,NP),V,X)-->tv(TV,V,X),np(NP).
vp1(app(TV,NP),Op,V,X)-->tv(TV,Op,V,X),np(NP).
vp1(app(TV,NP),Op,N,V,X)-->tv(TV,V,X),np(NP,Op,N).
vp2(NP1,NP2,V,X)-->tv(TV,V,X),np3s(NP,NP2),{appAll(TV,NP,NP1)}.

np(NP)-->{lexicon(propernoun,Sym,Word),npSem(Sym,NP)},Word.
np(NP,1)-->{lexicon(propernoun,Sym,Word),npSem(Sym,NP)},Word.
cn(CN)-->{lexicon(commonnoun,Sym,Word),cnSem(Sym,CN)},Word.

tv(TV,V,X)-->{lexicon(transitiveverb,V,Sym,Word,X),tvSem(Sym,TV)},Word.
tv(TV,Operation,V,X)-->{lexicon(transitiveverb,V,Operation,Word,X),tvSem(Sym,TV)},Word.
det(Det)-->{detSem(X,Det)},[X],[integer(X)].
prep(P)-->{lexicon(preposition,Sym,Word),prepSem(Sym,P)},Word.
pp(Op,N)-->{lexicon(pphrase,Op,N,Words)},Words,[integer(N)].

compose(Term,Symbol,ArgList):- Term =.. [Symbol|ArgList].
all([],_,[]):-
all(Xs,VP,[app(X,VP)|As]):-
alI(Xs,VP,As).
appAll(_,[],[]):-
appAll(TV,[NP|Ns],[app(TV,NP)|NPs]):-
appAll(TV,Ns,NPs).

allsubj([],_,[]):-
allsubj([X|Xs],VP,As):-
appAll(X,VP,C),append(C,R,As),
allsubj(Xs,VP,R).
divide([], _, []).
divide([add(W, obj(D, F)) | As], Y, [add(W, obj(Z, F)) | Xs]) :-
    Z is D / Y, divide(As, Y, Xs).

multiply([], _, []).
multiply([add(W, obj(D, F)) | As], Y, [add(W, obj(Z, F)) | Xs]) :-
    Z is D * Y, multiply(As, Y, Xs).

appendBetaConvertAll([], []).
appendBetaConvertAll([A | As], [B | Bs]) :-
    betaConvert(A, B), appendBetaConvertAll(As, Bs).

Appendix 3:

The module “grammar2”

:- use_module(sem).
:- use_module(lex).
:- use_module(mybeta).
:- use_module(usepredicates).

% sentences
s([B]) --> np(NP), vp1(VP), [betaConvert(app(NP, VP), B), !].
s([A]) --> np(NP), vp1(VP, Op, N), [betaConvert(app(NP, VP), B), compose(Z, Op, [B, N, A]), Z, !].
s(R) --> np(NP), vp2(VP, NP2), [betaConvert(app(NP, VP), B), betaConvert(NP2, C), R=[B, C], !].
s(W) --> nps(X, Y), vp1(VP, Op), [all(X, VP, A), appendBetaConvertAll(A, C), compose(Z, Op, [C, Y, W]), Z, !].

% noun phrases
nps([X | Xs], Z) --> a(Xs, D), np(X, _), [Z is D + 1].
a([X], Y) --> np(X, Y).
a([X, Y]) --> nps(X, Y).

np(app(DET, CN)) --> det(DET), cn(CN).
np(app(DET, CN), Op, N) --> det(DET), cn(CN), pp(Op, N).
np3(NP, app(NP, NP2)) --> np(NP), np2(NP2).
np2(app(P, NP)) --> prep(P), np(NP).

% verb phrases
vp1(app(TV, NP)) --> tv(TV), np(NP).
v1(app(TV, NP), Op) --> tv(TV, Op), np(NP).
v1(app(TV, NP), Op, N) --> tv(TV), np(NP, Op, N).
v2(app(TV, NP), NP2) --> tv(TV), np3(NP, NP2).
% lexicon

np(NP)-->\{lexicon(propernoun,Sym,Word),npSem(Sym,NP)\},Word.
np(NP,1)-->{lexicon(propernoun,Sym,Word),npSem(Sym,NP)},Word.
cn(CN)-->\{lexicon(commonnoun,Sym,Word),cnSem(Sym,CN)\},Word.
tv(TV)-->\{lexicon(transitiveverb,Sym,Word),tvSem(Sym,TV)\},Word.
tv(TV,Operation)-->{lexicon(transitiveverb,Sym,Operation,Word),tvSem(Sym,TV)},Word.
det(Det)-->\{detSem(X,Det)\},[X],[integer(X)].
prep(P)-->\{lexicon(preposition,Sym,Word),prepSem(Sym,P)\},Word.
pp(Op,N)-->{lexicon(pphrase,Op,N,Words)},Words,[integer(N)].

The module “questions” :

qw-->[how],[many].
qw-->[who].
qw-->[what].
object(Object)--->[Object].
q-->[does].
subject(Subject)--->[Subject].
opw-->[have].
opw-->[has].
comp(more)-->[more],[than].
comp(less)-->[less],[than].
comp(equal)-->[equal],[to].
comp(most)-->[the],[most].
comp(least)-->[the],[least].
query1(Object,Subject)-->qw,object(Object),q,subject(Subject),opw.
query2(Object)-->qw,opw,object(Object).
query3(Subject)-->qw,q,subject(Subject),opw.
query4(Obj1,C,Obj2)-->qw,opw,subject(Obj1),comp(C),subject(Obj2).
query5(C,Obj)-->qw,opw,comp(C),subject(Obj).

The module “input” :

readInput(F,E):-
  see(F),
  get0(Ch),
  create(Ch,E),
  seen.
create(C,[]):-
  C<0.
create(35,E):-
  skip,
  get0(Ch),
  create(Ch,E).
create(10,E):-
  get0(Ch),
  create(Ch,E).
create(Ch,E):-
  makeword(Ch,E).
makeword(Ch, [A|As]):-
    make(Ch, Ls),
    name(A, Ls),
    get0(Ch1),
    create(Ch1, As).
make(63, [63]).
make(46, [46]).
make(32, []).
make(L, [L|Ls]):-
    get0(Char),
    make(Char, Ls).

skip:-
get0(Ch), (Ch\=33 -> skip; true).

The module “generateAnswers”:

:-use_module(input).
:-use_module(basedOnQuery).
:-use_module(grammar2).
:-use_module(questions).

answerAll(F, O):-
    tell(O),
    readInput(F, E),
    parse(E, []),
    told.
parse([], A).
parse([E|Es], S):-
    sparse(Es, R, T, W),
    W==s,
    phrase(s(B), R),
    append(B, S, A),
    writeAnswer(R),
    write(.), nl,
    parse(T, A).
parse([E|Es], S):-
    sparse(Es, R, T, W),
    W==q,
    phrase(query1(Object, Subject), R),
    answer_basedOnQuery(S, quest1(Object, Subject)),
    write(?)
    nl,
    parse(T, S).
parse([E|Es], S):-
    sparse(Es, R, T, W),
    W==q,
    phrase(query2(Obj), R),
    answer_basedOnQuery(S, quest2(Obj)),
    parse(T, S).
parse(Es,S):-
    sparse(Es,R,T,W),
    W=q,
    phrase(query3(Subj),R),
    writeAnswer(R),
    write(?),nl,
    answer_basedOnQuery(S,quest3(Subj)),
    parse(T,S).
parse(Es,S):-
    sparse(Es,R,T,W),
    W=q,
    phrase(query4(Obj1,C,Obj2),R),
    writeAnswer(R),
    write(?),nl,
    answer_basedOnQuery(S,quest4(Obj1,C,Obj2)),
    parse(T,S).
parse(Es,S):-
    sparse(Es,R,T,W),
    W=q,
    phrase(query5(C,Obj),R),
    writeAnswer(R),
    write(?),nl,
    answer_basedOnQuery(S,quest5(C,Obj)),
    parse(T,S).

sparse([[E,'?']|Es],[E],Es,q):-!.
sparse([E,'.'|Es],Es,s):-!.
sparse([E|Es],X,S,W):-
    sparse(Es,X,S,W).

The module “usepredicates”:

compose(Term,Symbol,ArgList):-
    Term =.. [Symbol|ArgList].

all([],_,-[]).
all([X|Xs],VP,[app(X,VP)|As]):-
    all(Xs,VP,As).

divide([],_,-[]).

divide([add(W,obj(D,F))|As],Y,[add(W,obj(Z,F))|Xs]):-
    Z is D//Y,divide(As,Y,Xs).

multiply(add(W,obj(D,F)),Y,[add(W,obj(Z,F))|Xs]):-
    Z is D*Y.
appBetaConvertAll([], []).

appBetaConvertAll([A|As], [B|Bs]) :-
  betaConvert(A, B),
  appBetaConvertAll(As, Bs).

writeAnswer(Subject, X, Object) :-
  (X == 0 -> write(Subject),
   write(' '),
   write(does),
   write(' '),
   write(not),
   write(' '),
   write(have),
   write(' '),
   write(Object);
   write(Subject),
   write(' '),
   write(has),
   write(' '),
   write(X),
   write(' '),
   write(Object)).

writeAnswer([], Obj) :-
  write(' '),
  write(do),
  write(' '),
  write(have),
  write(' '),
  write(Obj).

writeAnswerSubjs(As, Obj) :-
  (As == [] -> write(noone),
   write(' '),
   write(has),
   write(' '),
   write(Obj); writeAnswer(As, Obj)).

writeAnswerSubjs(As, A, Obj) :-
  (As == [] -> write(noone),
   write(' '),
   write(has),
   write(' '),
   write(Obj); writeAnswer(A),
   write(do),
   write(' '),
   write(have)).

writeAnswer(has(_, obj(X, O))|Hs) :-
  write(X),
  write(' '),
  write(O), write(' '),
  writeAnswer(Hs).
writeAnswer([has(Subj,_)|Hs],Obj):-
  write(Subj),
  write(" "),
  writeAnswer(Hs,Obj).
writeAnswer([S|Ss],Obj):-
  write(S),
  write(" "),
  writeAnswer(Ss,Obj).
writeAnswerObjs(As,Subj):-
  (As==[]->write(Subj),
   write(" "),write(does),
   write(" "),write(not),
   write(" "),write(have),
   write(" "),write(anything);
   write(Subj),write(" "),
   write(has),write(" "),
   writeAnswer(As)).
writeAnswer([obj(N,Obj)|Objs]):-
  write(N),
  write(" "),
  write(Obj),
  write(" "),
  writeAnswer(Objs).
writeAnswer([]).
writeAnswer([A|As]):-
  write(A),
  write(" "),
  writeAnswer(As).
writeAnswerSubjs(A):-
  (A==[]->write(noone),
   write(" "),
   write(has);
   writeAnswer(A),
   write(do),
   write(" "),
   write(have)).
append([],L,L).
append([H|T],L2,[H|L3]):-
  append(T,L2,L3).
accRev([H|T],A,R) :-
  accRev(T,[H|A],R).
accRev([],A,A).
rev(L,R) :-
  accRev(L,[],R).
:- dynamic(counter/1).

var(Sym,N) :-
    counter(O),
    N is O+1,
    number_chars(N,L),
    atom_chars(A,L),
    atom_concat(v,A,Sym).

assertion(N):-
    retractall(counter(_)),
    New =.. [counter,N],assert(New),!.

counter(0).

reset :-
    retractall(counter(_)),
    assert(counter(0)).

padd(_,_,[],[]).

padd(Subject, Object, [add(Subject, obj(N, Object))|Ks], [N|Ns]):-
    padd(Subject, Object, Ks, Ns).

padd(Subject, Object, [sub(Subject, obj(N, Object))|Ks], [R|Ns]):-
    padd(Subject, Object, Ks, Ns), R is -1*N.

padd(Subject, Object, [_|Ys], Ns):-
    padd(Subject, Object, Ys, Ns).

sum([],0).

sum([H|T],S):-
    sum(T,TS), S is H+TS.

member(X, [X|_]).

member(X, [_|L]) :- member(X, L).

The module “basedOnQuery” :

answer_basedOnQuery(S, quest2(Obj)):-
    answer_who_has_object(Obj, S, As),
    write(#),
    writeAnswerSubjs(As, Obj),
    write(!), nl.

answer_basedOnQuery(S, quest3(Subj)):-
    answer_what_does_subject_have(Subj, S, As),
    write(#),
    writeAnswerObjs(As, Subj),
    write(!), nl.
answer_basedOnQuery(S,quest1(Obj,Subj)):—
  subject_has_Object2(Subj,Obj,X,S),
  write(#),
  writeAnswer(Subj,X,Obj),
  write(!),nl.

answer_basedOnQuery(S,quest4(Obj1,C,Obj2)):—
  answer_who_has_object(Obj1,S,As1),
  answer_who_has_object(Obj2,S,As2),
  compare(As1,C,As2,A),
  write(#),
  writeAnswerSubjs(A),
  write(!),nl.

answer_basedOnQuery(S,quest5(C,Obj)):—
  answer_who_has_object(Obj,S,As),
  (C==least->H=1000;
   C==most->H=1;true),
  findC(H,C,As,_,A),
  write(#),
  writeAnswerSubjs(As,A,Obj),
  write(!),nl.

subject_has_Object2(Subject,Object,A,S):—
  padd(Subject,Object,S,Xs),sum(Xs,A),!.

answer_what_does_subject_have(Subj,S,As):—
  what_does_subject_have(Subj,S,[],Os),!,
  find_answers(S,Subj,Os,As),!.

find_answers(_,_,[],[]).

find_answers(S,Subj,[Obj|Os],[obj(A,Obj)|As]):—
  subject_has_Object2(Subj,Obj,A,S),
  A>0,
  find_answers(S,Subj,Os,As).

find_answers(S,Subj,[],[As]):—
  find_answers(S,Subj,Os,As).

what_does_subject_have(_,[],A,A).

what_does_subject_have(Subj,[add(Subj,Obj[])|Ss],Os,A):—
  \+ member(Obj,Os),
  what_does_subject_have(Subj,Ss,[Obj|Os],A).

what_does_subject_have(Subj,[sub(Subj,Obj[])|Ss],Os,A):—
  \+ member(Obj,Os),
  what_does_subject_have(Subj,Ss,[Obj|Os],A).
what_does_subject_have(Subj,[|Ss],Os,A):-
    what_does_subject_have(Subj,Ss,Os,A).

answer_who_has_object(Obj,S,As):-
    who_has_Object(Obj,S,[|],Subjs),!
    find_answers2(S,Obj,Subjs,As),!.

find_answers2(_,_,[|],[]).

find_answers2(S,Obj,[|Subjs],[|Ss],[|As]):-
    subject_has_Object2(Subj,Obj,A,S),
    A>0,
    find_answers2(S,Obj,Subjs,As).

find_answers2(S,Obj,[|Subjs],As):-
    find_answers2(S,Obj,Subjs,As).

who_has_Object(_,[],A,0).

who_has_Object(Obj,[add(Subj,obj(_,Obj))|Ss],Subjs,A):-
    \+ member(Subj,Subjs),
    who_has_Object(Obj,Ss,[|Subjs],A).

who_has_Object(Obj,[sub(Subj,obj(_,Obj))|Ss],Subjs,A):-
    \+ member(Subj,Subjs),
    who_has_Object(Obj,Ss,[|Subjs],A).

who_has_Object(Obj,[|Ss],Subjs,A):-
    who_has_Object(Obj,Ss,Subjs,A).

compare([],_,_,[]).

compare([has(S,obj(N1,_))|H1],W,As2,[|S|A]):-
    finds(S,N1,W,As2),
    compare(H1,W,As2,A).

compare([|H1],W,As2,A):-
    compare(H1,W,As2,A).

finds(_,_,_,[]):fail.

finds(S,N1,more,[has(S,obj(N2,_))|_]),N1>N2.

finds(S,N1,less,[has(S,obj(N2,_))|_]),N1<N2.

finds(S,N1,equal,[has(S,obj(N2,_))|_]),N1==N2.

finds(S,N1,W,[|H2]),finds(S,N1,W,H2).
findC(_,_,[],A,A).

findC(Ni,least,[has(S,obj(N,_))|Hs],_,X):-
    N>0,N<Ni,findC(Ni,least,Hs,[S],X).

findC(Ni,least,[has(S,obj(N,_))|Hs],A,X):-
    N>0,N==Ni,findC(Ni,least,Hs,[S|A],X).

findC(Ni,most,[has(S,obj(N,_))|Hs],_,X):-
    N>0,N>Ni,findC(Ni,most,Hs,[S],X).

findC(Ni,most,[has(S,obj(N,_))|Hs],A,X):-
    N>0,N==Ni,findC(Ni,most,Hs,[S|A],X).

findC(Ni,W,[_|Hs],A,X):-
    findC(Ni,W,Hs,A,X).

The module sem:

cnSem(Sym,lam(a,app(a,Sym))).

tvSem(Sym,lam(w,lam(z,app(w,lam(x,Formula))))):-
    compose(Formula,Sym,[z,x]).

npSem(Sym,lam(b,app(b,Sym))).

detSem(Sym,lam(c,lam(d,lam(g,app(g,obj(Sym,d)))))).

prepSem(Sym,lam(v,lam(s,app(v,lam(i,Formula))))):-
    compose(Formula,Sym,[i,s]).

The module lex:

lexicon(determiner,X,[X]).

lexicon(commonnoun,apples,[apples]).

lexicon(commonnoun,bananas,[bananas]).

lexicon(commonnoun,oranges,[oranges]).

lexicon(propernoun,jack,[jack]).

lexicon(propernoun,bill,[bill]).

lexicon(propernoun,bob,[bob]).

lexicon(propernoun,anne,[anne]).

lexicon(transitiveverb,sub,[eats]).

lexicon(transitiveverb,add,[finds]).

lexicon(transitiveverb,sub,[gives]).

lexicon(transitiveverb,add,[takes]).

lexicon(transitiveverb,add,divide,[share]).

lexicon(pphrase,multiply,N,[from,each,of,the,N,people]).

lexicon(preposition,add,[to]).

lexicon(preposition,sub,[from]).
The module myBeta:

betaConvert(X,Y):-
    betaConvert(X,Y,[]).
betaConvert(app(Functor,Argument),Result,Stack):-
    betaConvert(Functor,Result,[Argument|Stack]).
betaConvert(lam(X,Formula),Result,[A|Stack]):-
    substitute(A,X,Formula,New),
    betaConvert(New,Result,Stack).
betaConvert(X,X,___).

substitute(T,V,Exp,Res):-
    Exp==V,!,
    Res=T.
substitute(_,_,Exp,Res):-
    
    substitute(T,V,Formula,Res):-
    compose(Formula,Functor,ArgList),
    substituteList(T,V,ArgList,ResList),
    compose(Res,Functor,ResList).
substituteList(_,[],[]).
substituteList(T,V,[Exp|O],[Res|R]):-
    substitute(T,V,Exp,Res),
    substituteList(T,V,O,R).

Appendix 4:

The module “generateAnswers”

:-use_module(input).
:-use_module(basedOnHasConstructs).
:-use_module(grammar2).
:-use_module(questions).
:-use_module(usepredicates).

answerAll(F,O):-
    tell(O),
    readInput(F,E),
    parse(E,[]),!,
    told.

parse([],A).

parse(Es,S):-
    sparse(Es,R,T,W),
    W==s,
    phrase(s(B),R),!,
    append(B,S,A),
    writeAnswer(R),
    write(.,nl),
    parse(T,A).
parse(Es,S):-
    sparse(Es,R,T,W),
    W==q,
    phrase(query1(Object,Subject),R),!,
    writeAnswer(R),
    write(?),nl,
    x(S,quest1(Object,Subject)),
    parse(T,S).

parse(Es,S):-
    sparse(Es,R,T,W),
    W==q,
    phrase(query2(Obj),R),!,
    writeAnswer(R),
    write(?),nl,
    x(S,quest2(Obj)),
    parse(T,S).

parse(Es,S):-
    sparse(Es,R,T,W),
    W==q,
    phrase(query3(Subj),R),!,
    writeAnswer(R),
    write(?),nl,
    x(S,quest3(Subj)),
    parse(T,S).

sparse([E,'?'|Es],Def,Es,q):-!.
sparse([E,'.'|Es],Def,Es,s):-!.
sparse([E|Es],[E|X],S,W):-sparse(Es,X,S,W).

The module “basedOnHasConstructs”

x(S,quest1(Obj,Subj)):-
    answer1(S,quest1(Obj,Subj)).

x(S,quest2(Obj)):-
    answer2(S,quest2(Obj)).

x(S,quest3(Subj)):-
    answer3(S,quest3(Subj)).

answer1(S,quest1(Obj,Subj)):-
    subjecthasObject([],R,S),
    find(Subj,Obj,R,Z),!,
    write(#),
    writeAnswer(Subj,Z,Obj),
    write(!),nl.
answer2(S,quest2(Obj)):-
subjecthasObject([],R,S),
find2(Obj,Subjs,Obj,R),!
write('#'),
writeAnswerSubjs(Subjs,Obj),
write(!),nl.

answer3(S,quest3(Obj)):-
sujecthasObject([],R,S),
find3(As,Subj,R),!
write('#'),
writeAnswerObjs(As,Subj),
write(!),nl.

subjecthasObject(A,R6,S):-
all(CN,PN),
searchAll(CN,PN,S,A,R6),!.

find_the_sentence(Subject,Obj,R,A,[has(Subject,object|Y,Subject)])|A):-
padd(Subject,Obj,R,Xs),sum(Xs,Y).

all(CN,PN):-
findall(X,lexicon(propernoun,X,-),PN),
findall(Y,lexicon(commonnoun,Y,-),CN).

searchAll(_,[],A,A).

searchAll([],_,[],S,A,R):-
all(CN,A),
searchAll(CN,PN,S,A,R).

searchAll([C|CN],_,[P|PN],S,A,R):-
find_the_sentence(P,C,S,A,R1),
searchAll(CN,[P|PN],S,R1,R).

find(Subject,Obj,[has(Subject,object|Z,Subject)])|Z).

find(Subject,Obj,_,[Ys],Z):-
find(Subject,Obj,Ys,Z).

find2([],__).

find2([Subject|A],Obj,[has(Subject,object|Z,Subject)])|Hs):-
Z>0,find2(A,Obj,Hs).

find2(A,Obj,[|Hs]):-
find2(A,Obj,Hs).

find3([],__).

find3([has(Subject,object|Z,Subject)])|A,Subject,[has(Subject,object|Z,Subject)])|Hs):-
Z>0,find3(A,Subject,Hs).

find3(A,Subject,[|Hs]):-
find3(A,Subject,Hs).
References

1. Ivan Bratko, Prolog Programming for Artificial Intelligence, 2001, Addison-Wesley.
2. Patrick Blackburn, Johan Bos, Kristina Striegnitz, Learn Prolog Now, 
3. Patrick Blackburn, Johan Bos, Computational Semantics for Natural Language, 
4. Aljoscha Burchardt, Alexander Koller, Stephan Walter, Computational Semantics, 
5. Michael A. Covington, Natural Language Processing For Prolog Programmers, 1994, 
   Prentice Hall.