Natural Language Processing and FOL Construction in Prolog

CSC 173, Prolog: Weeks #3/4

Author: Amsal Karic

10/30/2010

Index:

Goal: The assignment was clear and the assignment document was followed closely to get the expected results. Nothing was interpreted differently from the document, apart a slight modification to the grammar which will be elaborated on below.

Methods: The first part of the assignment required the provided grammar and lexicons to be implemented with a parsing routine. This was initially accomplished by designing the following grammar:

% S --> NP VP	% sentence --> noun_phrase, verb_phrase
% NP --> Det Noun	% noun_phrase --> determiner, noun
% --> Det Noun RELCL	% --> determiner, noun, relative_clause
% --> Noun	% --> noun
% --> Det Adj Noun	**% --> determiner, adjective, noun**
% VP --> Verb	% verb_phrase --> verb
% --> Verb NP	% --> verb, noun_phrase
% --> BeVerb Adj	% --> verb2, adjective
% RELCL --> Rel VP	% relative_clause --> rel, verb_phrase
% --> Rel NP Verb	% --> rel, noun_phrase, verb

This grammar allowed many of the sentences to be parsed correctly the issue here was that it lacked the addition of the highlighted part above and was not able to parse some sentences correctly, such as “The brown cow runs.” Of course this grammar is by no means complete for the entire English language but agreement with test cases was necessary and prompted this grammar addition.

These are some of the lexicons which were used in the final implementation:

```
Noun = {apple, boy, girl, government, watermelon}
Det = {a, the}
Verb = {conscript, like, run}
BeVerb = {is, are}
Adj = {evil}
Rel = {that, whom, who, which}.
```

A lot of lexicons have been added later to meet the need of the later input cases and far exceed the initial recommended lexicons given by the assignment document.

After all that was taken care of the actual parser and parse tree creator had to be coded (A lot of intermediate code manipulations have been skipped but can be seen in the appendices section.) It was decided that the more elegant --> design would be used to save space and time and was coded with number agreement consideration.

```
sentence6(X, sentence(NP, VP)) --> noun_phrase6(X, NP), verb_phrase6(X, VP).
```

We can see the first rule above which has the X variable consider number agreement throughout the parse. The parts of sentences (such as nouns, verbs etc.) were coded by taking number agreement into consideration also and had the following recommended form:

```
noun6(X, noun(A)) --> [A], {is_noun(A, X)}.
```

Where “noun” can be replaced with any part of the sentence, X designates the number agreement variable, and A is any word, such as “boy.”

The last part required adding number agreement to the individual words and was done in the similar fashion (once again taking number agreement into consideration):

```
is_noun(A, B).
```

Where “A” is any word such as “boy,” and “B” is the word “singular,” “plural,” or “_.” The last one is used in cases where the word can be considered to be singular or plural, such as “the.”

Now that the parser and parse tree generator was finished, the last stretch to the FOL translator began. First a “translate” procedure was created which feeds input from the parser directly into the translator and gives the output.

The code initially given in the assignment document was indeed used as a “springboard” to get the finalized FOL translator.

```
s2(sentence(NP, VP), F) :-
    gensym(x, X),
    np2(X, NP, T),
    vp2(X, T, VP, F).
```

We can see the first rule above for the translator which generates a variable and feeds it into the np2 and vp2 subroutines.

Notice that the subroutine np2 gets the results from vp2 and compiles them to obtain the final result.

Np2 in this particular case consisted of a noun phrase with a determiner, adjective, and noun. It takes the three parts and runs subroutines on them also with the determiner subroutine compiling all of the results from the other routines here.

```
vp2(Var, verb_phrase(V, NP), P3) :-
    vernp(Var, V, P1, P3),
    gensym(x,X),
    np2(X, NP, P1).
```

This is where things get interesting with the np2 subroutine. It uses the vernp subroutine to combine the verb and noun phrase (hence vernp) and gives the output. An important thing to consider here is the fact that np2 uses its own gensym to accurately reflect the fact that a different item is being considered with another noun phrase, or np2 call.

All of the noun phrase, verb phrase, and relative clause rules were created in a similar fashion (and can be seen in the appendices.)

Note: an instrumental tool for working on this component was the additional “little tutorial” link on the assignment page right above the extra credit section.

```
det2(Var, determinerX(A), P1, exists(Var, P1)).

adj2(Var, DESC, P) :-
    DESC =.. [adjective, ADJ],
    P =.. [ADJ, Var].

n2(Var, DESC, P) :-
    DESC =.. [noun, N],
    P =.. [N, Var].
```

Now we can see the determiner, adjective, and noun word rules above. The determiner rule adds “exists” or “all” information to the output and the adjective and noun rules simply link the adjective or nouns to their representation. For example “x7,” “apple” would return “apple(x7)” as the output. Number agreement didn’t matter here because it was dealt with in the parser routine. The last thing that mattered then was being able to tell when to use “all” and when to use “exists.” This was done by looking out for words such as “all” and “some” which indicate what it should be and keeping all of the procedures true to their function. For this one procedure was created for “exists” and one was created for “all” and these identifiers from the output of another procedure would tell the program which procedure to go into.

After this was done the code was polished with inclusion of I/O which used a scanner which was included in the assignment document. The code was slightly modified to make use of the list output it gave. Since the ending period or exclamation marks were included in the list, the old friend “reverse2” from another Prolog assignment was used to reverse the list, remove the head, and reverse the list again. This produced the desired list to be fed in and got rid of the ending marks which would have messed up the parsing and translation procedures. This is a bit different from the way it was handled in the assignment document where use of a sentence finisher was made.

Results: We will now go over all of the inputs and results we should have obtained in this project from the assignment document and naturally add some commentary.

The first part of the assignment was just implementation of the grammar, lexicons, and parser which parsed sentences and returned if they were true according to the grammar or false. We have run these through again with our finalized procedure and will show those results along with their parse trees/FOL translations if applicable.

```
273 ?- read_in(X,Y).
|: The boy runs..

X = sentence(noun_phrase(determiner(the), noun(boy)), verb_phrase(verb(run))),
Y = all(x125, boy(x125)=>run(x125)) .
```

Here we can see that the sentence was deemed correct by the grammar and produced a parse tree representation which is consistent with the grammar. The FOL translation here happened using "the" as "all" but doesn't matter since we did not have to worry about this distinction issue in the project according to the assignment document.

```
274 ?- read_in(X,Y).
|: Girls like an apple..

false.
```

The sentence above was not deemed correct by the parser on the grounds of number agreement. While girls and like are plural, an and apple being singular makes an exception occur. This was attempted to be fixed numerous times and if it is fixed in the code with modification of the "X" number agreement variable, another sentence would not function at all.

```
275 ?- read_in(X,Y).
|: A government that conscripts people is evil..

X = sentence(noun_phrase(determiner(a), noun(government),
relative_clause(relative_pronoun(that), verb_phrase(verb(conscript),
noun_phrase(noun(people))))), verb_phrase(verb(is), adjective(evil))),
Y = all(x126, government(x126)&conscript(x126, people(x127))=>evil(x126)) .
```

```
276 ?- read_in(X,Y).
|: The boy whom the girl likes likes a watermelon..
```

X = sentence(noun_phrase(determiner(the), noun(boy),
relative_clause(relative_pronoun(whom), noun_phrase(determiner(the), noun(girl)),
verb(like))), verb_phrase(verb(like), noun_phrase(determiner(a), noun(watermelon)))) .

The next sentences show that they parsed correctly and produced the desired parse tree. Once again the "a" was interpreted as "all" and this was not required to be addressed in this assignment.

277 ?- read_in(X,Y).
|: The boy run.
false.

278 ?- read_in(X,Y).
|: Girls likes an apple..

false.

279 ?- read_in(X,Y).
|: A government that conscripts people are evil..

false.

280 ?- read_in(X,Y).
|: The boy who the girl likes likes a watermelon..

false.

281 ?- read_in(X,Y).
|: The boy which the girl likes likes a watermelon..

false.

This is the part of the assignment which shows if the number agreement code works. We can see above that all of the sentences which should have failed due to number agreement issues have indeed failed and produced the desired output. It highlights the number agreement implementation working in all of these cases.

282 ?- read_in(X,Y).
|: All boys run..

X = sentence(noun_phrase(determiner(all), noun(boy)), verb_phrase(verb(run))),
Y = all(x129, boy(x129)=>run(x129)) .

283 ?- read_in(X,Y).

|: All boys like all watermelons that contain some divine flavors..

X = sentence(noun_phrase(determiner(all), noun(boy)), verb_phrase(verb(like),
noun_phrase(determiner(all), noun(watermelon), relative_clause(relative_pronoun(that),
verb_phrase(verb(contain), noun_phrase(determinerX(some), adjective(divine),
noun(flavor))))))),

Y = all(x140, boy(x140)=>like(x140, all(x141, watermelon(x141)&contain(
x141, exists(x142, flavor(x142)&divine(x142)))))) .

284 ?- read_in(X,Y).
|: Some boy eats some apple..

X = sentence(noun_phrase(determinerX(some), noun(boy)), verb_phrase(verb(eat),
noun_phrase(determinerX(some), noun(apple)))),
Y = exists(x133, boy(x133), eat(x133, exists(x134, apple(x134)))) .

285 ?- read_in(X,Y).
|: Some governments conscript some pacifist people..

X = sentence(noun_phrase(determinerX(some), noun(government)),
verb_phrase(verb(conscript), noun_phrase(determinerX(some), adjective(pacifist),
noun(people)))),
Y = exists(x135, government(x135), conscript(x135, exists(x136, people(x136)&pacifist(x136)))) .

286 ?- read_in(X,Y).
|: All governments that conscript some pacifist people are evil..

X = sentence(noun_phrase(determiner(all), noun(government),
relative_clause(relative_pronoun(that), verb_phrase(verb(conscript),
noun_phrase(determinerX(some), adjective(pacifist), noun(people))))), verb_phrase(verb(are),
adjective(evil))),
Y = all(x137, government(x137)&conscript(x137, exists(x138,
people(x138)&pacifist(x138)))=>evil(x137)) .

287 ?- read_in(X,Y).
|: All big apples are delicious..

X = sentence(noun_phrase(determiner(all), adjective(big), noun(apple)), verb_phrase(verb(are),
adjective(delicious))),
Y = all(x139, apple(x139)&big(x139)=>delicious(x139)) .

We can see if we go back to the assignment document and compare this output to that of it that we get the identical output based on the given sentences. This shows that scanning,

parsing and parse tree generation, number agreement, and FOL translation all work successfully in this assignment.

Since this was all the output that was given by the assignment document that could be compared to, once the correct results were obtained to all of the provided cases, some additional cases were run through (not shown here) and produced a similar desired output. One could therefore consider all of the implementations in the assignment to be successful.

Discussion:

This project has demonstrated the power that the prolog programming language has. The actual implementation of the parser and FOL translator was an enjoyable experience compared to the nitty-gritty madness which constituted the work on the C parser and scanner programming project. The beauty here is that with a few hundred lines of rather elegant code, a parser, scanner, and FOL translator can be implemented with not too much frustration. The parser and translators however had a rather limited grammar which may make certain more complex English sentences fail to parse. Naturally in order to make a larger grammar, one runs the issue of taking considerable more code to implement, and still has the possibility of not having all possible wacky sentences be able to be evaluated. Nevertheless, this seems like a good place where this project can expand.

In AI there needs to be an evaluation of input, understanding by machine, and output command formulation based on the understanding. This project involved the evaluation of the input by creation of parse trees. The understanding of the input only has been partially worked on here with the FOL translation which lessens some of the sentence ambiguity. Therefore it is possible in the future to expand on this project and make the computer able to somehow use a database of commands and compare to the FOL translation and give the proper input based on those commands.

References:

The only two resources used for this project were the assignment description on the CSC 173 course site, and the book “Programming in Prolog using the ISO Standard” Fifth Edition by William F. Clocksin, and Christopher S. Mellish (primarily chapters 9 and 10.)

Appendices:

We will cover some code pieces here which were not included anywhere else in this report. This code includes intermediate steps which were taken but were modified greatly to get to the finalized result.