JJ Javier and Victor Mullison, HW14 Group17

HW13—Prolog project: natural language and PL parser
Finally, we are ready to see the power of Prolog!

Prolog is a "specialist": it does only a few things but does them very well.

SUBMIT:

Group work (1-3 ppl). Only 1 person in 1 group should submit + add group members names in the comment section.

Please add your finished code, answers to the questions, and required screenshots into this file and submit this file (in pdf). The chunk of file that contains your code should stay in the Prolog format so that the grader can copy/paste it to the online compiler and run the program.

Grading:

10pt.

Natural language parser 4pt.

Submitted some code – 1pt

Answered all the questions – 1pt

Has parse tree screenshot –1pt

Code is working+ submitted the output screenshot showing correct output --1pt

PL parser 6pt.

Submitted some code – 1pt

Answered all the questions – 1pt

Has parse tree screenshot –1pt

Code is working+ submitted the output screenshot showing correct output --3pt

Questions:

The questions below are formatted with Prolog comments so that you can directly copy paste the content into the online compiler & complete the work.

```
/*#################

 * Natural Language Processing Example -- Parser
```

* If a computer wants to understand a line of code, it will check if this line of code fits into BNF grammar.

* Similarly, if a computer wants to understand an English sentence, it would want to first check if this line is

* grammatically correct.

*

*

* Let us see a short version of English grammar that would define sentences like: a human sees a dog.

* This is the "BNF grammar" for this sentence:

* RULE1: Sentence is defined as -> noun_phrase   verb_phrase

* RULE2: noun_phrase->determiner   noun   (E.g. an apple)

* RULE3: verb_phrase->verb   noun_phrase

* RULE4: ver_phrase -> verb (E.g. looks an apple, or looks)

* */

/*The following is how we write the grammar into Prolog database:*/

/*The first line translates the BNF RULE1 into Prolog. It means:

* To generate a sentence S, the prolog solver will

*   1.find a noun_phrase and save it in the variable Np

*   2.find a verb_phrase and save it in the variable Vp

*   3.call append function by merge Np and Vp into the same list and save the merged list in S (which is the variable used to save a sentence)

* Put it into a function style that is more familiar to you, it looks like:

```
*    string [] Find_Sentence ()
*    {
*       string [] Np=Find_Noun_Phrase();
*       string [] Vp=Find_Verb_Phrase();
*       string [] S= merge(Np, Vp)
*       return S
```

```
*/

/* Follow the example and translation RULE2 and RULE3 below into Prolog in a similar manner.
* Make sure you have variables for each item and use append to "return" the merged list*/

sentence(S):- noun_phrase(Np),verb_phrase(Vp),append(Np,Vp,S). % RULE1
noun_phrase(Np):-determiner(D),noun(N),append(D,N,Np). % RULE2
verb_phrase(Vp):-verb(V),noun_phrase(Np),append(V,Np,Vp). % RULE3

/*Rule 4 is slightly different since it does not have 2 lists to merge together. We can use
* empty list [] to occupy the space. Vp=V+[]. */
verb_phrase(Vp):-verb(V),append(V,[],Vp).% RULE4

%define words/token below
determiner([the]). % define determiner the
determiner([a]). % define determiner a

noun([human]). % define noun human
noun([cat]). % define noun cat

verb([sees]). %define verb sees

/*Run query: ?-sentence(NewSen)
 * What will this query give you?*/
NewSen = [the, human, sees, the, human]
NewSen = [the, human, sees, the, cat]
NewSen = [the, human, sees, a, human]
NewSen = [the, human, sees, a, cat]
NewSen = [the, human, sees]

NewSen = [the, cat, sees, the, human]

NewSen = [the, cat, sees, the, cat]

NewSen = [the, cat, sees, a, human]

NewSen = [the, cat, sees, a, cat]

NewSen = [the, cat, sees]

NewSen = [a, human, sees, the, human]

NewSen = [a, human, sees, the, cat]

NewSen = [a, human, sees, a, human]

NewSen = [a, human, sees, a, cat]

NewSen = [a, human, sees]

NewSen = [a, cat, sees, the, human]

NewSen = [a, cat, sees, the, cat]

NewSen = [a, cat, sees, a, human]

NewSen = [a, cat, sees, a, cat]

NewSen = [a, cat, sees]
```

/*Set breakpoint at the first line of the code, run the same query step by step and observe closely.

 * How did this program work? Can you explain the process?*/

% The program constructs all valid sentences by combining the determiners, verbs, and nouns according to the given BNF grammar rules. Starting from RULE 1, it goes to check the rule for noun_phrase(Np), from there it goes to RULE 2. RULE 2 gets the determiner we declare along with the noun and appends it into the list. After RULE 2 is finished it returns to RULE 1 and continues to the next rule, verb_phrase(Vp) RULE 3. RULE 3 gets the verb we added, the noun phrase from RULE 2 appended and then appends it back together. RULE 4 takes place when it doesn't find the other noun and determiner so it returns just a 3 word sentence. But when detected it returns a 5 word sentence grammatically correct.

/*The above query shows you how we can generate grammatically correct sentences.

 Are all the generated sentences semantically correct?/

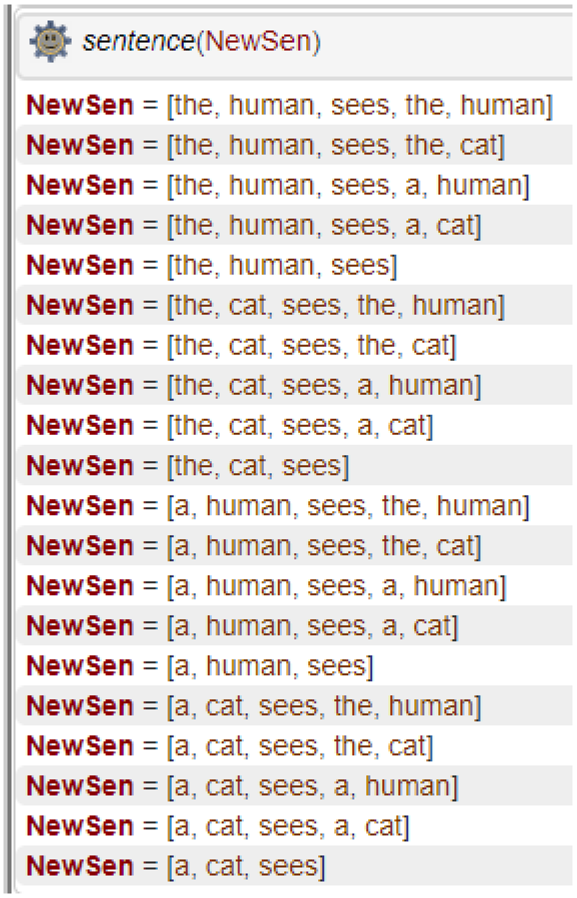
% Yes.

/*Why do you need to append in every line?*/

% The appends combine sections of the sentence together to form phrases, and then the final append combines the phrases to form the sentence.

/*Please take a screenshot of your program's output and add it here*/

%



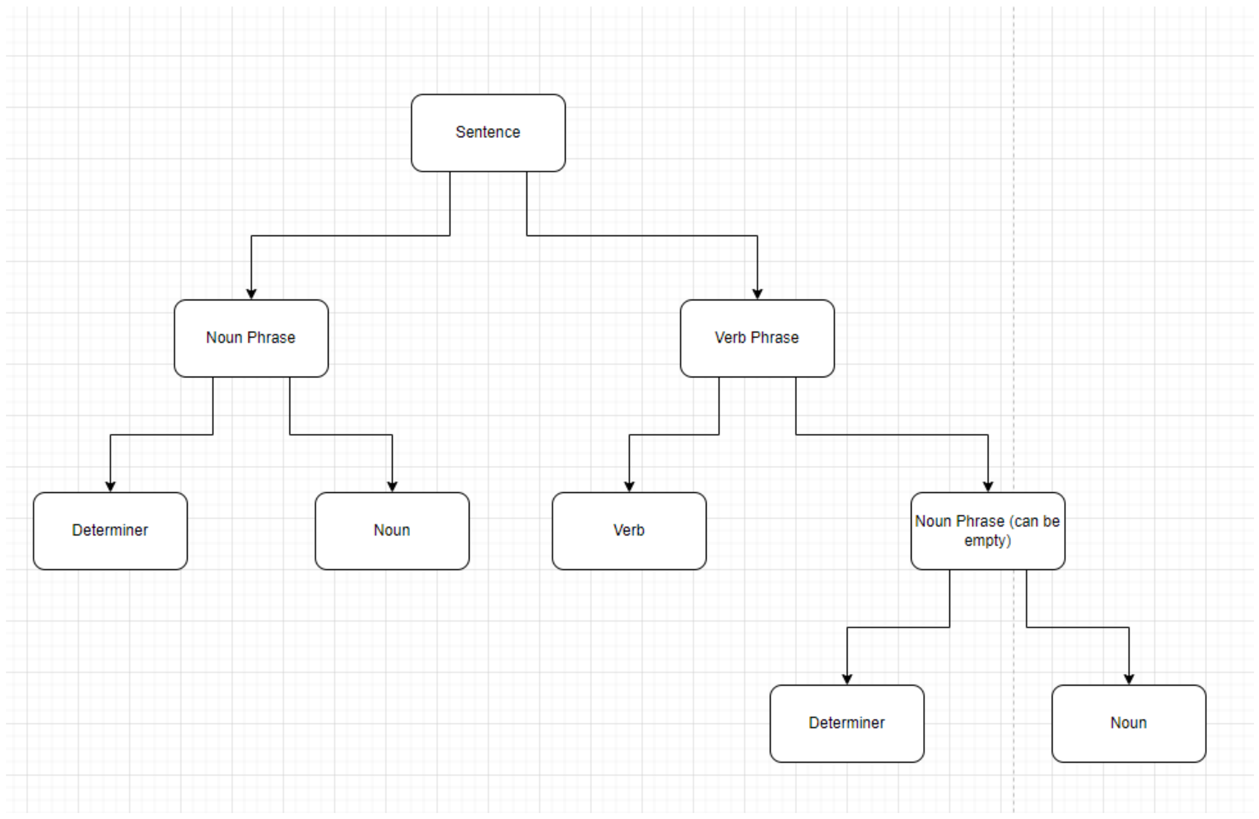
```
sentence(NewSen)

NewSen = [the, human, sees, the, human]
NewSen = [the, human, sees, the, cat]
NewSen = [the, human, sees, a, human]
NewSen = [the, human, sees, a, cat]
NewSen = [the, human, sees]
NewSen = [the, cat, sees, the, human]
NewSen = [the, cat, sees, the, cat]
NewSen = [the, cat, sees, a, human]
NewSen = [the, cat, sees, a, cat]
NewSen = [the, cat, sees]
NewSen = [a, human, sees, the, human]
NewSen = [a, human, sees, the, cat]
NewSen = [a, human, sees, a, human]
NewSen = [a, human, sees, a, cat]
NewSen = [a, human, sees]
NewSen = [a, cat, sees, the, human]
NewSen = [a, cat, sees, the, cat]
NewSen = [a, cat, sees, a, human]
NewSen = [a, cat, sees, a, cat]
NewSen = [a, cat, sees]
```

/*Now let us see how a parser can check whether the input line is a grammatically correct sentence.

* Run query: ?-sentence([a, human, sees, the, cat]).

* What is the output?

* Set a breakpoint and go step by step. Can you draw a parse tree following the steps? Please copy/paste the screenshot into this file. */



/*We say programming languages are just like human languages all the time without really thinking this through.

 * After the above exercise, can you see the deeper link between natural vs. computer languages?

 * What are they?

 * Can you link the compiler process to the process that your brain *might* use to understand human languages?

 * For those who have a psychology background, this might be an interesting topic to go deeper.*/

Indeed, natural programming languages resemble speech, whereas computer languages tend to be more technical. A compiler checks to see whether or not the code makes sense, similar to how our brains might screen a sentence as gibberish or not.

/*###################

 * PL Parser

 * We spent quite some time on parsing this semester.

```
 * The parsing process (those LL,LR algorithms) seems super hard to write in C++.
 * But Prolog is born for this. See below:*/

/*Let us use the BNF grammar for code like: varA=1+5
 * BNF:
 * Code->identifier equalop math
 * math->int mathop int
 *
 * Since the append func can only take 3 parameters, let us modify the BNF:
 *
 * Code->identifier remaining1
 * remaining1->equalop math
 * math->int remaining2
 * remaining2->mathop int
 *
 * modify the above grammar to Prolog
 * */
%prolog code
math_code(Code) :-identifier(Identifier),remaining1(Remaining1),Code = [Identifier | Remaining1].
remaining1(Remaining1) :-equalop(EqualOp),math(Math),Remaining1 = [EqualOp | Math].
math(Math) :-int(Int1),remaining2(Remaining2),Math = [ Int1 | Remaining2].
remaining2(Remaining2) :-mathop(MathOp),int2(Int2),Remaining2 = [MathOp, Int2].
%tokens
identifier('varA').
equalop('=').
mathop('+').
int(1).
int2(5).
```

/*Run query: ?-math_code(NewCode).

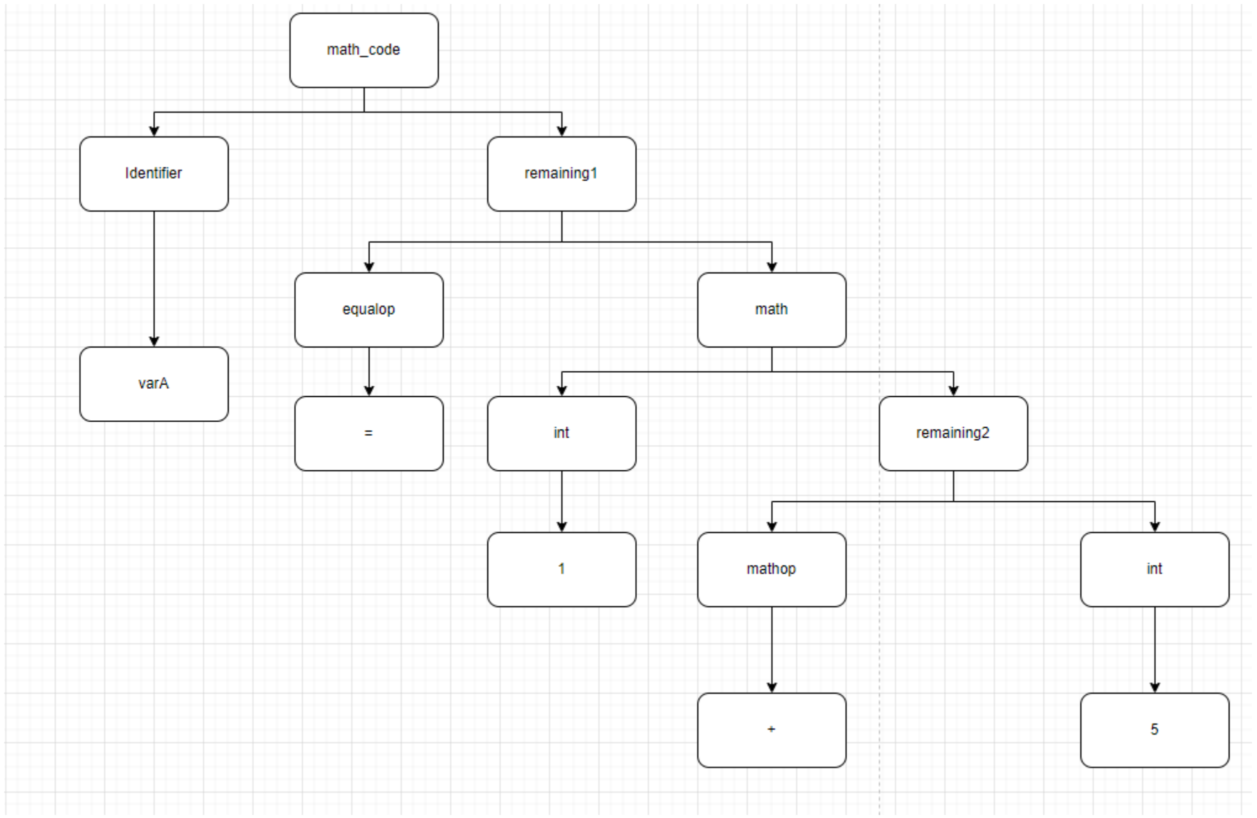
 * What will this query do?*/

%It goes through the BNF rules made through prolog. Starting from the first rule of Code->identifier remaining1 it checks the identifier and calls the remaining rule RULE2. RULE 2 reamining1->equalop math, then using the identifier and code gets the equalop declared and calls the next rule. RULE 3 math->int remaining2, then gets the first int declared and then calls the fourth rule remaining2->mathop int which gets the mathop declared, the second int declared and then finally appends it altogether with the identifier and 2 nums in the list.

/*What query should you write if you want to check whether VarA=1+5 is a legal line of math_code?*/

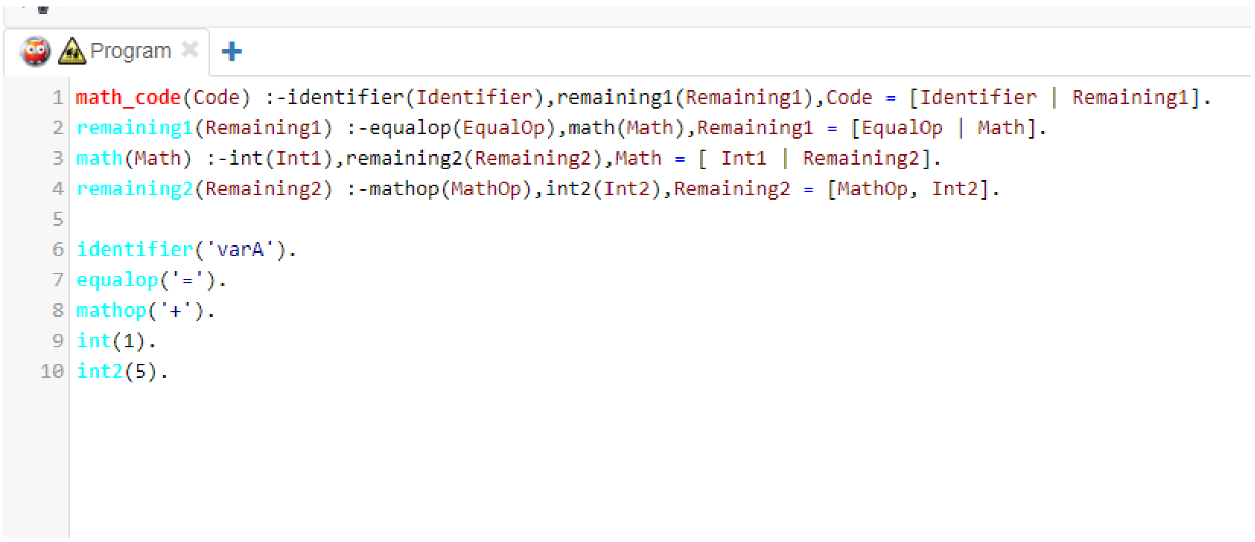
query: ?-math_code([varA, =, 1, +, 5]).

/*Set a breakpoint and go step by step. How does the code work and generate new lines of code? Draw *a parse tree of the given example. */

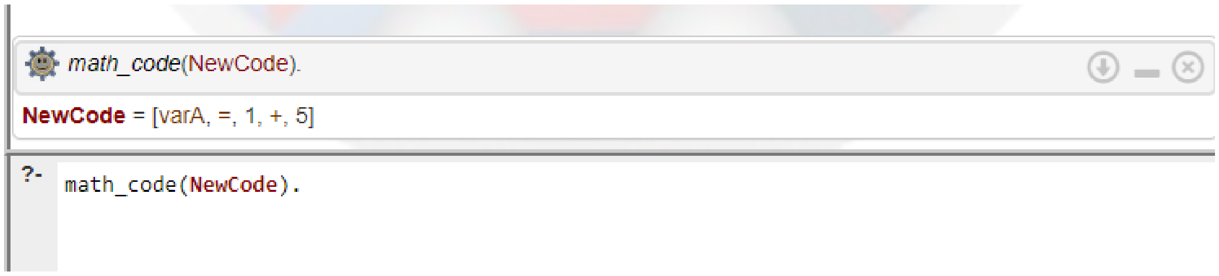


/*Please take a screenshot of your program's output and add it here*/

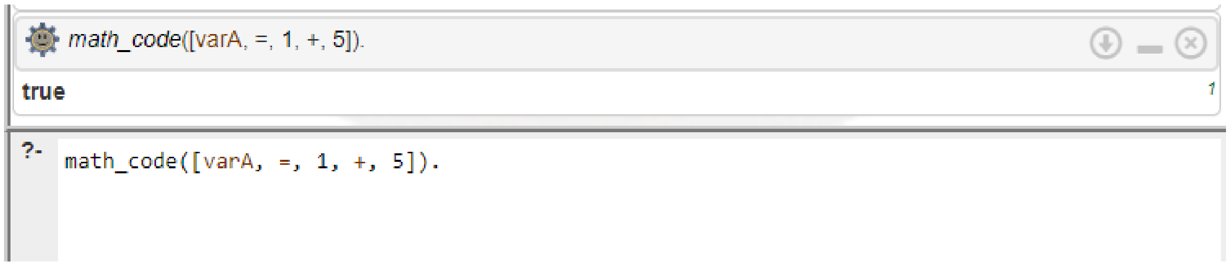
%____Add your code output screenshot here____



```
math_code(Code) :-identifier(Identifier),remaining1(Remaining1),Code = [Identifier | Remaining1].
remaining1(Remaining1) :-equalop(EqualOp),math(Math),Remaining1 = [EqualOp | Math].
math(Math) :-int(Int1),remaining2(Remaining2),Math = [ Int1 | Remaining2].
remaining2(Remaining2) :-mathop(MathOp),int2(Int2),Remaining2 = [MathOp, Int2].

identifier('varA').
equalop('=').
mathop('+').
int(1).
int2(5).
```



```
math_code(NewCode).
NewCode = [varA, =, 1, +, 5]

?- math_code(NewCode).
```



```
math_code([varA, =, 1, +, 5]).
true

?- math_code([varA, =, 1, +, 5]).
```