American University of Sharjah
School of Engineering
Computer Science & Engineering Dpt.
P. O. Box 26666
Sharjah, UAE

Instructor: Dr. Michel Pasquier
Office: ESB 2057
Phone: 971-6-515-2883
E-mail: mpasquier@aus.edu
Semester: Fall 2022

CMP 321 PROGRAMMING LANGUAGES

PARSER PROJECT

Due date/time: Wednesday 7 December 2022, at 11 pm

Instructions: Submit via the assignment box on *iLearn,* before the deadline, your *project deliverables* as elaborated hereafter, without this document. Make sure to include in all your files the *name and ID of all team members.* No printed/hard copy is required. Follow any further instructions as may be posted on *iLearn.* Note that due to the belated deadline, late submissions will *not* be accepted.

For this assignment, you must send your complete *source code* (one plain *text* file) as well as the *program output,* and a *contribution brief,* as explained later in this document. You should bundle all files in a *single* Zip/Rar *archive* named after all team members e.g., FAloul-GBarlas-MPasquier.zip. One should be able to extract your program from the archive and run it "as is". All these are *strict requirements;* note that non-compliant submissions will be treated as "not submitted."

You are to complete this project assignment *as a team of three or four students.* Furthermore, each team must *work independently* and hand in their own original answers. You are *not* allowed to discuss or *share* any solution or to *copy* from others or from any sourced material. Plagiarism and cheating will be severely penalized, starting with a zero grade for this assignment. Recall you are bound by the AUS Academic Integrity Code which you signed when joining AUS. Lastly, note that *any team member may be called upon,* as may be needed, *to explain* any part of the project code or design, also their contribution, etc.

Project Overview: In this project, you are to implement a *Recursive Descent Parser* that determines whether a given *Prolog program* is correct, or whether it contains syntax errors, according to the *simplified BNF grammar* given hereafter. Note that it is not necessary to know the language to implement a parser – only the grammar!

You are strongly advised to follow the approach described in class/the textbook when implementing both *lexical analyzer* (cf. section 4.2) and *syntax analyzer* (cf. section 4.4). Coding a parser is essentially about implementing sequential steps, strictly as defined by the rules of the language's grammar. In this assignment, you *must* use Python to implement the parser. Using regular expressions is optional.

Prolog Grammar: Below is a small grammar in BNF (Backus-Naur Form) that defines a much *simplified* version of the *Prolog* programming language.

```
<program>        ->  <clause-list> <query> | <query>
<clause-list>    ->  <clause> | <clause> <clause-list>
<clause>         ->  <predicate> . | <predicate> :- <predicate-list> .
<query>          ->  ?- <predicate-list> .
<predicate-list> ->  <predicate> | <predicate> , <predicate-list>
<predicate>      ->  <atom> | <atom> ( <term-list> )
<term-list>      ->  <term> | <term> , <term-list>
<term>           ->  <atom> | <variable> | <structure> | <numeral>
<structure>      ->  <atom> ( <term-list> )
<atom>           ->  <small-atom> | ' <string> '
<small-atom>     ->  <lowercase-char> | <lowercase-char> <character-list>
<variable>       ->  <uppercase-char> | <uppercase-char> <character-list>
<character-list> ->  <alphanumeric> | <alphanumeric> <character-list>
<alphanumeric>   ->  <lowercase-char> | <uppercase-char> | <digit>
<lowercase-char> ->  a | b | c | ... | x | y | z
<uppercase-char> ->  A | B | C | ... | X | Y | Z | _
<numeral>        ->  <digit> | <digit> <numeral>
<digit>          ->  0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9
<string>         ->  <character> | <character> <string>
<character>      ->  <alphanumeric> | <special>
<special>        ->  + | - | * | / | \ | ^ | ~ | : | . | ? |   | # | $ | &
```

Note that this simplified grammar does *not* include arithmetic, lists, input/output, special operators, and many other Prolog features. Yet it contains all the core features needed for logic programming – in only half a page!

Project Requirements: The correctness of your parser will be tested on the six *sample programs* provided to you as well as twenty or more *other programs* as devised by your instructor. The output of your parser should be, for each test program, either a statement that it is *syntactically correct,* or else a list of statements indicating *which syntax errors* were found, and *where.* The parser should not stop at the first error but continue analyzing the input program, as far as possible. While the *accuracy* and *usefulness* of the parser are the most important criteria by far, note that the *quality of your code* will also be examined. Of course, any code that is not your own creation will be disqualified, resulting in a zero grade for the project.

Note that you need to submit the *complete source code* of your parser, in a *single* Python file. So, if you use multiple source files during development, you simply need to merge them all into a single file prior to submission (and make sure the resulting file runs 'as is').

To facilitate testing, your program should *automatically read and parse* in order *all test code files* found in the *same folder/directory* as your parser. The code files shall be named sequentially as "1.txt", "2.txt", … "10.txt", … "24.txt.", and so on. Do not assume any maximum number; just use a counter and keep parsing code files until there are no more. In the above example, your program should parse the 24 code files then stop, since "25.txt" is not found. Note that if your program does not run at all, if it cannot find the files, or if it requires manual user input, it will be treated as “not submitted”. So make sure you test thoroughly.

In addition, your program should save *the output of the parser* to a file called *parser_output.txt.* You should *submit* this file as well, that will include your parser’s output for the six *sample programs* provided hereafter. Lastly, you must include a *brief* explaining the exact *contribution* of each team member (i.e., which part of the parser code he/she is responsible for…)

Note that, in case clarifications are required, any team member may be called upon to explain any part of the project and/or his/her contribution. Each team member should thus be familiar with every part of the parser program (which is the learning objective). Using a “pair/peer programming” approach is strongly recommended.

Correct Programs: Below are examples of *valid* programs i.e., *syntactically correct* according to the *simplified Prolog BNF.* Your parser should therefore recognize them as such. (Note that these programs work fine in any Prolog interpreter.)

Program “1.txt” – Crime puzzle

```
criminal(X) :- american(X), weapon(Y), nation(Z),
               hostile(Z), sells(X,Z,Y).
owns(nono,msl(nono)).   missile(msl(nono)).
sells(west,nono,M) :- owns(nono,M), missile(M).
weapon(W) :- missile(W).
hostile(H) :- enemy(H,america).
american(west).
nation(nono).   enemy(nono,america).
nation(america).
?- criminal(Who).
```

Program “2.txt” – Object database

```
object(candle,red,small,1).
object(apple,red,small,1).
object(apple,green,small,1).
object(table,blue,big,50).
location(object(candle,red,small,1), kitchen).
location(object(apple,red,small,1), kitchen).
location(object(apple,green,small,1), kitchen).
location(object(table,blue,big,50), kitchen).
?- location(object(_name,red,_,_), kitchen).
```

Program “3.txt” – Natural numbers

```
num(0).
num(s(N)) :- num(N).
num_plus(0,N,N).
num_plus(s(M),N,P) :- num_plus(M,s(N),P).
?- num( s(s(s(0))) ).
```

<u>Incorrect Programs:</u> Below are examples of *invalid* programs i.e., that are somehow *syntactically incorrect* according to the *simplified Prolog BNF* given to you. Your parser should be able to identify the error(s) therein – the more the better. (Note that all these programs actually work fine in any Prolog interpreter.)

<u>Program "4.txt" – Hanoi towers</u>

```
say(N, From, To) :- write('move disc '), write(N), write(' from '),
                    write(From), write(' to '), write(To), nl.
hanoi(N) :- move(N, left, center, right).
move(0, _, _, _).
move(N, From, To, Using) :- is(M, N-1), move(M, From, Using, To),
                            say(N, From, To), move(M, Using, To, From).
?- hanoi(3).
```

<u>Program "5.txt" – Factorial</u>

```
factorial(N,F) :- factorial(N,1,F).
factorial(0,F,F).
factorial(N,A,F) :- A1 is N*A, N1 is N-1, factorial(N1,A1,F).
?- factorial(100,Result).
```

<u>Program "6.txt" – Merge sort</u>

```
mergeSort([], []).
mergeSort([A], [A]).
mergeSort([A, B | Rest], S) :- divide([A, B | Rest], L1, L2),
                               mergeSort(L1, S1),
                               mergeSort(L2, S2),
                               merge(S1, S2, S).
divide([], [], []).   divide([A], [A], []).
divide([A, B | R], [A | Ra], [B | Rb]) :- divide(R, Ra, Rb).
merge(A, [], A).   merge([], B, B).
merge([A | Ra], [B | Rb], [A | M]) :- A =< B, merge(Ra, [B | Rb], M).
merge([A | Ra], [B | Rb], [B | M]) :-  A > B, merge([A | Ra], Rb, M).
?- mergeSort([3, 4, 8, 0, 1, 9, 5, 6], Sorted).
```

<u>Test Programs:</u> As mentioned, your parser will be tested on several *code files / programs* devised by your instructor. Some will be *syntactically correct* and some will *not:* a number of them will feature basic syntax errors such as a missing or mismatched parenthesis, a missing period at the end of a clause, an invalid operator, etc. All of these your parser should detect if it abides by the given BNF grammar. Sometimes the test code file will feature a single error, sometimes there will be two or more…

When starting, your program should *automatically read and parse* in sequence *all the code files* that are found in the *same folder/directory,* as explained earlier. Initially, that means parsing the six given files "1.txt" to "6.txt"; later, more test files will be added. Again, you should assume that numbers are in sequence and stop your program at the first missing number/file. You can (and should) test your parser by creating more files: just duplicate any of the given examples then edit the source code to create some errors and check if they are properly recognized, or not.

For *each input file,* your parser will do a lexical and syntax analysis of the Prolog code contained in that file and *print out each error* (“missing parenthesis”, “invalid symbol”, etc.) and *its location* (line number, etc.) If there is no error, it should of course *report that the program is correct.* To catch multiple errors, the parser will need to recover from a given error and carry on the analysis as if the code was correct. For instance, if the input program is missing a closing parenthesis, the parser should report that error then continue parsing as if it was actually there.

Caution: You should *not* use tracing or include any extraneous print statement in the final version of your parser! Only the required information should be output, as elaborated above – otherwise meaningful messages will be buried in the mass… By all means use tracing during development, but turn it off before submitting your code. (Best would be to have an obvious Boolean flag that can easily be switched.)

To get full marks, your parser should analyze correctly all the Prolog test files, provide useful syntax error messages, and be reasonably well programmed as well. Conversely, the more mistakes the parser makes, the less informative the messages, the less readable your program is, etc. the lower the grade. Again, if the program does not even run, no credit will be given – so please double-check. Refer to the instructions on the first page and make sure you follow them. Then of course, feel free to ask of you have any question (well before the deadline!)

☺

Ok, here is a joke about parsing, scope, language ambiguity… and programmers:

A computer programmer is going to the grocery store to buy some supplies. On the way out, the spouse yells down the stairs, “Hey, when you are at the store, could you grab a gallon of milk? And if they have any eggs, get a dozen!”

So later the programmer comes home with… 12 gallons of milk.