Assignment 3: Simple Interpreter in Prolog

The Assignment

This assignment is about implementing an interpreter in Prolog for assignment statements in a programming language.

If you are aiming for grade C, D or E you should implement an interpreter for the following grammar in EBNF (ISO standard) Grammar1:

```
assign = id , ‘=’ , expr , ‘;’ ;
expr = term , [ ( ‘+’ | ‘-’ ) , expr ] ;
term = factor , [ ( ‘*’ | ‘/’) , term] ;
factor = int | ‘(’ , expr , ‘)’ ;
```

where `id` is defined as a Prolog atom and `int` is defined as an integer.

If you are aiming for grade A or B you should implement an interpreter for the following grammar in EBNF (ISO standard) Grammar2:

```
block = ‘{’ , stmts , ‘}’ ;
stmts = [ assign , stmts ] ;
assign = id , ‘=’ , expr , ‘;’ ;
expr = term , [ ( ‘+’ | ‘-’ ) , expr ] ;
term = factor , [ ( ‘*’ | ‘/’ ) , term ] ;
factor = int | id | ‘(’ , expr , ‘)’ ;
```

where `id` is defined as a Prolog atom and `int` is defined as an integer.

Note that in Grammar2 the assignment statement may include identifiers (variables) in the expression.

The interpreter should consist of two separate parts:
a) a parser,
b) an evaluator.

You should read your input from file using the scanner-tokenizer already implemented for you in SICStus Prolog (using SICStus Prolog specific predicates). If you would like to use another Prolog system you need to port this scanner-tokenizer to that Prolog system yourself.

Given Code Files

There are some code files for you in the zip-file Assignment3-Code.zip:
1) “tokenizer.pl”, which includes the already implemented scanner-tokenizer.
2) “interpreter.pl”, which includes a skeleton of the interpreter you should implement.
2) “program1.txt”, “program2.txt”, “parsetree1.txt” and “parsetree2.txt”, which are examples of input and output to the interpreter.

Parser

The parser should take a list of lexemes/tokens as input, and from that list of lexemes/tokens create a parse tree as output. The output of your interpreter should **exactly** follow the given examples. A parser for Grammar1 with input according to “program1.txt” should output a parse-tree **exactly** according to “parsetree1.txt”, and a parser for Grammar2 with input according to “program2.txt” should output a parse-tree **exactly** according to “parsetree2.txt”.

Evaluator

The evaluator should take a parse tree as input, and return the program-state after execution as output. The program-state is represented by a list of all variables and their values. The output of your evaluator should **exactly** follow the given example. An evaluator for Grammar1 with input according to "program1.txt" should output a program-state **exactly** according to "parsetree1.txt", and an evaluator for Grammar2 with input according to "program2".txt should output a program-state **exactly** according to "parsetree2.txt".

Note that the grammar rules are right recursive but the arithmetic operators are left associative. This complexity needs to be handled in a correct way to get the valuation "good" for the implementation of the evaluator (grade A or C).

Remember that in Grammar2 the assignment statement may include identifiers (variables) in the expression. To be able to evaluate expressions including variables, the evaluator needs to maintain a data structure including the current value of all found variables. Assume the default value of an unassigned variable is 0.

Other requirements

You are not allowed to use any non-ISO-standard built-in predicates in your implementation of parse/1 and evaluate/3.

Submission

Your assignment should be submitted as a single Prolog file called "interpreter#.pl" where "#" should be replace by your group number, for example "interpreter7.pl". The submission should be made to the appropriate place in iLearn, including a comment in the beginning of the file with the name of all authors. If the submission is not in the correct format the grade will be FX. (However, in this case an FX could be upgraded to a higher grade than E.)

Grading

Each part, parser and evaluator, will be given a valuation with respect to the quality of the implementation:
a) Missing means it is missing or it is not a serious attempt;
b) NotOk means it is not functioning;
c) Ok means it is functioning;
d) Good means it is well functioning and the code is well written and well structured.

The following table will be used for grading:

Grade	**Correct format**	**Grammar**	**Parser**	**Evaluator**
A	yes	Grammar2	Good	Good
B	yes	Grammar2	Ok	Ok
C	yes	Grammar1	Good	Good
D	yes	Grammar1	Ok	Ok
E	yes	Grammar1	Ok	NotOk/Missing
FX	yes		NotOk	
FX	no		Not Missing	
F			Missing	

Good luck!