Assignment 3: Simple Interpreter in Prolog

The Assignment

This assignment is about implementing an interpreter in Prolog for assignment statements in some 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..z)+` and `int` is defined as `(0..9)+`.

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..z)+` and `int` is defined as `(0..9)+`.

The interpreter should consist of the following parts:
a) parser,
b) evaluator.

If you would like to read your input from file there is a scanner-tokenizer already implemented for you (in SICStus Prolog using SICStus Prolog specific predicates). Otherwise the input could be given as a list of tokens directly in the query.

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 the top-most predicate 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 Program.

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 Grammar 1 with input according to program1.txt should output a parse-tree according to parsetree1.txt, and a parser for Grammar2 with input according to program2.txt should output a parse-tree 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 Grammar 1 with input according to program1.txt should output a program-state according to parsetree1.txt, and an evaluator for

Grammar2 with input according to program2.txt should output a program-state according to parsetree2.txt.

Note that the grammar rules are right recursive (tail recursive) but the arithmetic operators are left associative. This complexity needs to be handled in a correct way to get a grade of A or B.

Note also 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.

Grading

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

The following table will be used for grading:

Grade	**Parser**	**Evaluator**
A	Good	Good
B	Good	Ok
C	Good	Ok
D	Ok	Ok
E	Ok	NotOk

Recall that to get a grade A or B you must implement an interpreter for Grammar2, and an evaluator implementing the left associativity of the arithmetic operators.

Good luck!