CS 345 - Programming Languages
Assignment 2

1 Programming in Prolog (30 Points)

The task is to implement a parser and evaluator for mySIMPL (my Simple IMperative Programming Language). The core concept of the language is arithmetic expressions over floating points numbers and variables. It has additional imperative constructs: declarations, assignments, and a return statement which terminates the program and returns a number.

The syntax of mySIMPL is:

\[
\begin{align*}
\text{prog} &::= \text{retStatement } \cdot \mid \text{declaration } \cdot \mid \text{assignment } \cdot \mid \text{prog} \\
\text{declaration} &::= \text{var } < \text{id} > \\
\text{assignment} &::= < \text{id} > := \text{base} \\
\text{retStatement} &::= \text{return } \text{base} \\
\text{base} &::= < \text{id} > \mid < \text{number} > \mid (\text{expression } \cdot) \\
\text{expression} &::= [\text{expression addOp}] \text{term} \\
\text{term} &::= [\text{term mulOp}] \text{factor} \\
\text{factor} &::= \text{base} \\
\text{addOp} &::= '+' \mid '-' \\
\text{mulOp} &::= '*' \mid '/' 
\end{align*}
\]

For simplicity, we assume that mySIMPL programs are given as token (prolog) lists. An example of a program is:

\begin{verbatim}
var x;
x := (5 * 2);
return (x + 1).
\end{verbatim}

which is represented by the token list:

\[
[\text{'var'}, \text{'x'}, \cdot, \cdot, \text{'x'}, :=', '(' , 5 , '*' , 2 , ')', \cdot, \cdot, \text{'return'}, '(' , 'x' , '+' , 1 , ')', \cdot, \cdot]
\]
Implement a predicate
\[ \text{parse}(+\text{TokenList}, -\text{AST}) \]
which parses the token list and returns an abstract syntax tree (no constraints on the format) for syntactically correct programs and fail for incorrect programs.

Then implement a predicate
\[ \text{evaluate}(+\text{AST}, -\text{Number}) \]
which evaluates the program’s abstract syntax tree and returns a numeric value for semantically correct program and fails for incorrect programs.

A program is syntactically correct if it can be expressed through the given grammar. A program is semantically correct if furthermore no undeclared or uninitialized variables are used in any statement.

The language is very simple and lacks fundamental concepts like control flow, types, or functions but we will extend it in future assignments.

**Additional Information and Hints**

- Prolog features DCGs for building parsers in an intuitive way.
- The grammar has left-recursive elements in it. For better parsing, a format like
  \[
  \text{expr}(E) \rightarrow \text{term}(T), [+], \left_assoc(E,T,\text{plus}).
  \left_assoc(\text{expression}(\text{plus},T,T1),T,\text{plus}) \rightarrow \text{term}(T1).
  \ldots
  \left_assoc(E,T,\text{minus}) \rightarrow \text{term}(T1), [-], \left_assoc(E,\text{expression}(\text{minus},T,T1),\text{minus}).
  \ldots
  \]
  is preferable. This code produces an AST node called 'expression' with arguments 'plus' and the two terms it has parsed around the addOp.
- The predicate \[ \text{phrase}(:\text{DCGBody}, ?\text{List}) \] can be used to parse a list of tokens against a DCG grammar.
- A crucial task of the evaluator is keeping track of variable assignments. Prolog’s associative lists can be of great help. See \text{put_assoc/4, get_assoc/3, and empty_assoc/1}.
- If you use associative lists, the different evaluation steps must keep track of the state transformations and therefore carry an input state list and an output state list which is the input list after the transformations have been applied (recall that Prolog variables cannot be redefined within an execution).
- Make sure that your parser accepts numbers in both float and integer format (i.e., 10 and 10.0).
- The return value of a successful evaluation can either be in integer or float format.
- The parser should not allow the keywords of the language (e.g., the arithmetic operators, \text{=:, var, return}) as variable identifiers.
- If you want to use any other Prolog dialect, make sure that it runs on SWIProlog. This is the platform on which we will test and grade the solutions.