Interpretador de expressões com memória

Desenvolva um interpretador para a seguinte gramática:

$$
\begin{aligned}
\mathit{Prog} &::= \{\mathit{Dec}\}\ \mathit{Exp} \\
\mathit{Dec} &::= \mathit{Var}\ \texttt{`='}\ \mathit{Exp} \\
\mathit{Var} &::= [\texttt{a-z}] \\
\mathit{Exp} &::= \mathit{Exp}\ \mathit{BinOp}\ \mathit{Exp} \\
&\quad | \ \mathit{UnOp}\ \mathit{Exp} \\
&\quad | \ \mathit{Num} \\
&\quad | \ \texttt{`@'}\ \mathit{Var} \\
&\quad | \ \texttt{`('}\ \mathit{Exp}\ \texttt{`)'} \\
\mathit{BinOp} &::= \texttt{`+'} \mid \texttt{`-'} \mid \texttt{`*'} \mid \texttt{`/'} \mid \texttt{`\^{}'} \\
\mathit{UnOp} &::= \texttt{`-'} \\
\mathit{Num} &::= [\texttt{0-9}] + ([\texttt{.}][\texttt{0-9}]+)?
\end{aligned}
$$

Note que a gramática possui ambiguidades que devem ser resolvidas pela sua implementação. As ambiguidades têm relação com a precedência dos operadores (menor prioridade para maior prioridade):

```
+ e -
* e /
- (unário)
^
```

A exponenciação tem associatividade à direita, enquanto todos os outros operadores binários têm associatividade à esquerda.

O seu interpretador também deverá tratar os seguintes erros semânticos:

- divisão por zero;
- declaração de variável duplicada;
- uso de variável não declarada.

A entrada do seu interpretador é um programa válido na linguagem da gramática da calculadora simples, o qual deverá ser informado pela linha de comando. A saída do seu interpretador é a impressão na tela do resultado da interpretação do programa ou uma mensagem de erro. Por exemplo, se o programa válido, conforme o exemplo abaixo

```
1 - 2 * 3 + 4
```

a saída deve ser o resultado da interpretação da expressão:

```
-1.00
```

Contudo, se o programa contiver erros, como no exemplo a seguir

```
2 / (1 - 1)
```

o saída deve ser uma mensagem de erro, por exemplo:

```
stdin:1:8: divisão por zero!
```

Fases do projeto:

1. **Análise léxica**
   Nesta fase você precisa apenas criar uma varredura do programa, isto é, basta classificar cada token e imprimir na tela dentro de um laço de repetição.

2. **Análise sintática**
   Nesta fase você deve integrar a análise léxica com a sintática, isto é, agora o seu analisador léxico deve fornecer uma função `get_token()` que o analisador sintático pode usar para validar a gramática, além de gerar uma árvore de sintaxe abstrata para o programa. Nesta fase é interessante criar uma função que imprime a árvore, pois pode ajudar com debug.

3. **Interpretação com análise semântica**
   Nesta fase você deve integrar a análise sintática com a interpretação do programa. É nesta fase que você também deve fazer a análise semântica das expressões e declarações de variáveis, o que pode ocorrer durante a interpretação, parecido com o que ocorre em linguagens dinamicamente tipadas. Em resumo, você deve percorrer a árvore gerada pela análise sintática e imprimir o resultado da interpretação do programa. Se existir algum problema semântico a interpretação é abortada e uma mensagem de erro deve ser exibida.

O exemplo disponibilizado pelo professor mostra como integrar as fases do projeto.