# INF441 - Roteiro para Aula Prática
# Listener com ANTLR

Faça download do arquivo usando o link a seguir, que contém um projeto Eclipse compactado:
https://drive.google.com/file/d/1iomn2KCZEX8LdzOrBEw7MuXbwAnYbgZW/view?usp=sharing

Em seguida, descompacte o conteúdo em uma pasta qualquer, em um computador que tenha Eclipse + ANTLR4 instalado.

Abra o IDE Eclipse e importe o projeto descompactado. Antes de iniciar este roteiro, verifique se o IDE Eclipse está devidamente configurado, seguindo as instruções em:

https://drive.google.com/file/d/1TWBsuBXlzZmSvHniZQ80POzoCVSaPbhP/view?usp=sharing

Em seguida, siga as instruções do roteiro para atividades relacionadas à construção de um *Listener* usando ANTLR.

## Diretivas Iniciais

Na ferramenta ANTLR, o produto de uma análise sintática bem-sucedida é uma árvore de sintaxe representando o programa analisado. Para realizar operações sobre essa árvore, ANTLR oferece facilidades para se trabalhar com os padrões de projeto *Visitor* e *Listener*. *Visitor* (https://pt.wikipedia.org/wiki/Visitor_Pattern) é um padrão de projetos clássico. *Listener* é uma variação utilizada na ferramenta ANTLR. ANTLR permite usar as duas abordagens (Visitor e Listener) para percorrer uma árvore de sintaxe e executar operações desejadas.

A principal diferença entre esses padrões é que, no Visitor, um programador deve definir explicitamente como realizar o caminhamento na árvore junto com o código para implementar as operações desejadas. O Listener realiza um caminhamento automaticamente, e as operações são associadas a funções chamadas na entrada e saída de nós da árvore, enquanto o caminhamento é executado. O Visitor oferece mais flexibilidade, enquanto o Listener requer menos trabalho do programador. Neste roteiro, iremos trabalhar com o padrão Listener na ferramenta ANTLR.

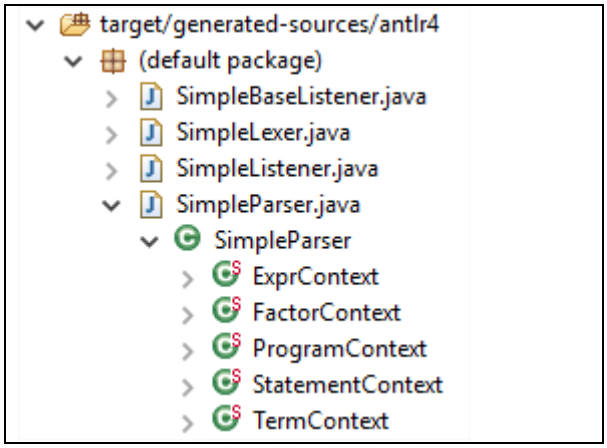
Abra o projeto Eclipse mencionado no início deste roteiro.
Abra o arquivo *grammars/Simple.g4* e veja a definição de uma linguagem *Simple*, com comandos de atribuição e impressão.

```
program: statement*;

statement:
    IDENT '=' expr  ';'
  | 'print' expr';'
  ;

expr:
    '-'? term
  | expr ('+'|'-') term
  ;

term:
    factor
  | term ('*'|'/') factor
  ;

factor:
    REAL_NUMBER
  | IDENT
  | '(' expr ')'
  ;
```

Exemplo de programa válido:

```
y = 10;
x = -1 + y + 2 * 3;
print x;
```

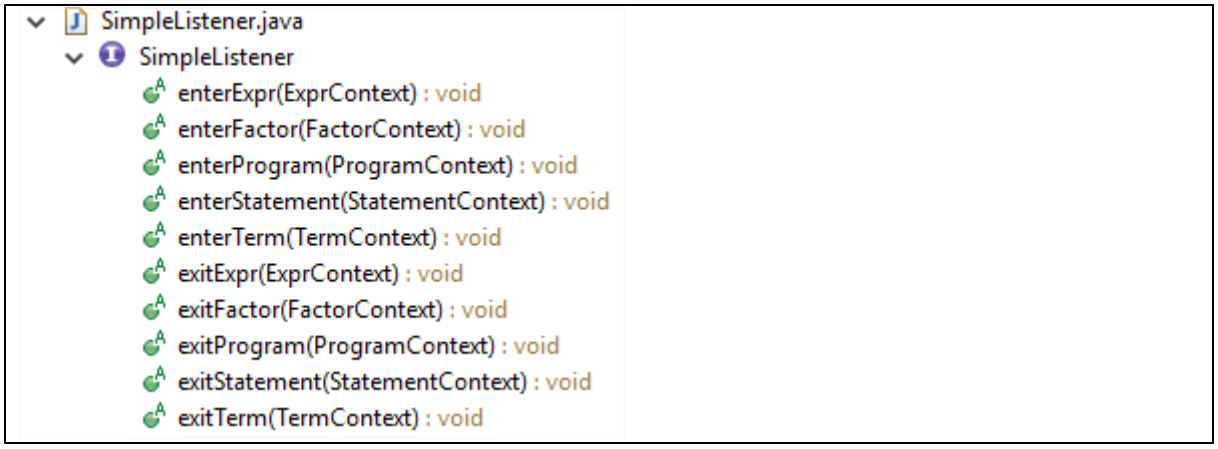
Na pasta de arquivos gerados automaticamente pelo ANTLR, a classe *SimpleParser* representa um analisador sintático gerado automaticamente, a partir da gramática acima. Essa classe contém também outras classes internas, que são usadas para representar a árvore de sintaxe resultante de uma análise sintática.




Na figura ao lado, pode-se ver as classes internas a *SimpleParser*, usadas para representar os nós da árvore de sintaxe. Essas classes são geradas automaticamente pelo ANTLR, baseando nas regras da gramática: *program*, *statement*, *expr*, *term* e *factor*.

Essas classes associadas à árvore de sintaxe contêm métodos para obter os componentes de cada nó da árvore. Por exemplo, a classe *StatementContext*, cujos métodos são apresentados abaixo, é usada para representar nós da regra *statement*. O método *expr* serve para obter a expressão que pode aparecer em um comando de atribuição ou de impressão; o método *IDENT* serve para obter no nome do identificador usado em um comando de atribuição.


Além da classe *SimpleParser*, ANTLR gera automaticamente definições para se trabalhar com o padrão Listener. A interface *SimpleListener* é uma delas, contendo métodos que serão chamados à medida que um Listener executar um caminhamento sobre uma árvore de sintaxe. Há dois métodos para cada nó da árvore, sendo um método chamado quando um nó é encontrado (por exemplo, *enterStatement*) e outro método chamado quando toda a subárvore daquele nó já foi visitada (por exemplo, *exitStatement*).




## Atividade 1

Para facilitar a identificação de cada alternativa de uma mesma regra, ANTLR permite criar métodos de Listener diferentes para cada alternativa. Para isso, basta inserir comentários na frente de cada alternativa, na definição da gramática.

Modifique a gramática inserindo os elementos em destaque, como definido abaixo.

```
program: statement*;

statement:
    IDENT '=' expr  ';'    # Assign
  | 'print' expr';'        # Print
  ;

expr:
    op='-'? term               # OneTerm
  | expr op=('+'|'-') term     # AddSub
  ;

term:
    factor                       # OneFactor
  | term op=('*'|'/') factor     # MulDiv
  ;

factor:
    REAL_NUMBER              # Real
  | IDENT                    # Id
  | '(' expr ')'             # Parens
  ;
```

Os elementos após # são simplesmente comentários associados a cada alternativa das regras. O elemento op= significa que uma variável *op* poderá ser acessada durante o caminhamento na árvore, permitindo que se identifique qual operador foi utilizado.

Em seguida, grave o arquivo *Simple.g4* e observe as alterações na interface *SimpleListener*. Veja que foram criados métodos com nome *enter_* e *exit_* para cada alternativa das regras, usando os nomes fornecidos nos comentários. Esses métodos vão facilitar nossa tarefa de implementar um Listener mais adiante, pois saberemos exatamente que tipo de nó da árvore de sintaxe estaremos visitando, na entrada e na saída do caminhamento.

## Atividade 2

Vamos começar a definir uma operação sobre os programas analisados. Essa operação deverá simular o comportamento do programa, avaliando expressões, armazenando valores de variáveis e imprimindo valores quando o comando de impressão for utilizado.

Crie um arquivo *EvalListener* na pasta *src* do projeto, copiando o código abaixo.

```
import java.util.Stack;

public class EvalListener extends SimpleBaseListener {

	private Stack<Double> stack;

	public EvalListener() {
		stack = new Stack<Double>();
	}

	public void exitReal(SimpleParser.RealContext ctx) {
		stack.push(Double.parseDouble(ctx.REAL_NUMBER().getText()));
	}

	public void exitOneTerm(SimpleParser.OneTermContext ctx) {
		if (ctx.op != null) {
			stack.push(-stack.pop());
		}
	}

	public void exitAddSub(SimpleParser.AddSubContext ctx) {
		Double d2 = stack.pop();
		Double d1 = stack.pop();
		if (ctx.op.getType() == SimpleParser.ADD) {
			stack.push(d1 + d2);
		} else {
			stack.push(d1 - d2);
		}
	}

	public void exitPrint(SimpleParser.PrintContext ctx) {
		System.out.println(stack.pop());
	}

}
```

No código acima, *EvalListener* utiliza uma pilha de valores Double para simular a execução de um programa da linguagem Simple.

O método *exitReal* é associado à alternativa # Real da gramática, isto é, a constantes reais que aparecem em expressões. O código desse método consiste em armazenar o valor da constante na pilha. Observe como o valor da constante é obtido, usando métodos da classe *RealContext* recebida.

Os outros métodos da classe simulam as alternativas # OneTerm, #AddSub e # Print da gramática. Procure entender o funcionamento do código, para poder criar novos códigos que serão requisitados nas outras atividades deste roteiro.

Para executar as operações definidas em EvalListener, é preciso dispará-lo sobre uma árvore de sintaxe construída. Para isso, altere o programa piloto, acrescentando os comandos destacados no código a seguir.

```
public class Piloto {

      public static void testeParser(String fileName) throws Exception {
            ...

            ParseTree tree = parser.program();
            System.out.println(tree.toStringTree(parser));

            ParseTreeWalker walker = new ParseTreeWalker();
            EvalListener eval = new EvalListener();
            walker.walk(eval, tree);

      }
...
```

Execute o programa *Piloto.java* e observe o resultado do processamento do arquivo de entrada *input/teste01.txt*. Altere o arquivo de entrada para ver outros resultados. Lembre que, por enquanto, o Listener simula apenas comandos de impressão (não atribuição) e que as expressões ainda não podem ter operações de multiplicação e divisão, nem variáveis.

## Atividade 3

Acrescente definições de métodos em EvalListener para processar operações de multiplicação e divisão em expressões. Na gramática, essas operações são definidas pelas alternativas # OneFactor e # MulDiv.

O método que deve ser definido é:

```
      void exitMulDiv(SimpleParser.MulDivContext ctx) { ... }
```

Não é necessário acrescentar definição para o método

```
        void exitOneFactor(SimpleParser.OneFactorContext ctx) { ... }
```

pois a alternativa # OneFactor não realiza nenhuma operação sobre o valor associado.
Sempre que ficar em dúvida sobre o formato do cabeçalho dos métodos a serem acrescentados, basta copiá-los da interface *SimpleListener*.

Escreva código para o método *exitMulDiv*. Como base para o seu código, observe como foram implementadas as operações de soma e subtração, na Atividade 1. Em seguida, modifique o arquivo de entrada de modo a conter impressão de expressões envolvendo também multiplicação e divisão. Verifique se os resultados funcionam como esperado.

## Atividade 4

Acrescente definições de métodos em *EvalListener* para processar comandos de atribuição e expressões que contenham variáveis. Se uma variável for usada antes de ser definida, deve-se assumir que ela tem valor 0 (zero).

Sugestão: acrescente em *EvalListener* um mapeamento de identificadores para valores reais, como no código em destaque abaixo; esse mapeamento deve associar o nome de uma variável ao seu valor. A interface *Map* e a classe *HashMap* devem ser importadas da biblioteca padrão *java.util*.

```
public class EvalListener extends SimpleBaseListener {

        private Map<String, Double> map;
        private Stack<Double> stack;

        public EvalListener() {
                map = new HashMap<String, Double>();
                stack = new Stack<Double>();
        }
...
```

Dois serviços da interface *Map* podem ser particularmente úteis no código desta tarefa*:*

- *Double get(String)*: dado o nome de uma variável, consulta o mapeamento e retorna o valor atual dessa variável; caso a variável ainda não tenha sido instanciada, irá retornar null (neste caso, seu programa deve considerar que a variável tem valor 0).
- *put(String, Double)*: dado o nome de uma variável e um valor, registra essa associação no mapeamento; caso já tenha sido registrado, altera o valor do registro.

Os serviços acima poderão ser usados para implementar a tarefa solicitada. Nos comandos de atribuição, os valores do mapeamento devem ser alterados. Quando variáveis forem usadas em expressões, o mapeamento deve ser consultado.

Na gramática, as alternativas # Assign e # Id são as envolvidas. Os métodos que devem ser definidos são:

```
public void exitId(SimpleParser.IdContext ctx) { ... }

public void exitAssign(SimpleParser.AssignContext ctx) { ... }
```

Implemente os métodos acima. Em seguida, modifique o arquivo de entrada de modo a conter programa com atribuições e uso de variáveis em expressões. Verifique se os resultados funcionam como esperado.

## Instruções para entrega

Entregue apenas o arquivo **EvalListener.java** modificado.