



# Antlr

Analizador sintático



# Antlr

Para fazer esse lab vocês vão utilizar o Antlr, que é uma ferramenta em java que recebe a descrição de uma gramática e transforma ela em um código que faz a análise sintática de uma linguagem. Ela pode gerar código em várias linguagens diferentes, mas nós vamos trabalhar com java também.

Para instalar o antlr baixe do link: <https://www.antlr.org/download/antlr-4.7.2-complete.jar>



# Antlr

Assim como o Flex, a execução do lab vai se passar em várias etapas:

- Criar a gramática .g4
- Compilar a gramática em código Java
- Implementar uma herança de uma das classes geradas.
- Implementar um programa que chama o parser gerado pelo antlr e a herança criada
- “Compilar” o código java.
- Executar o parser passando códigos exemplos por parâmetro.



# Antlr

Para compilar a gramática .g4 que vocês vão fazer em código java se usa o comando:

- `java -jar "antlr-4.7.2-complete.jar" -no-listener -visitor grammar.g4`

Os próximos comandos dependem que seja criado uma variável de ambiente que contenha o caminho para o .jar do antlr:

- `export CLASSPATH=".:[seu_caminho]/antlr-4.7.2-complete.jar:$CLASSPATH"`

Não esqueça do “.” no início, ele é necessário para que o java saiba que o diretório corrente deve ser usado para buscar outras classes.



# Antlr

Os comandos a seguir dependem da variável CLASSPATH, se ela não estiver preenchida eles não vão funcionar!

- `javac *.java`

Compila todos os arquivos gerados em um executável java.

- `Java exec`

Executa o programa. Nós vamos fazer ele ler o stdin, logo os arquivos de entradas devem ser passados com “< input.file”. Para funcionar precisamos ter implementado `exec.java` com o metodo `main()`.



# Antlr

Os comandos a seguir dependem da variável CLASSPATH, se ela não estiver preenchida eles não vão funcionar!

- `java org.antlr.v4.runtime.misc.TestRig GRAMMAR PRODUCTION -tokens < input`

GRAMMAR é a gramática definida por GRAMMAR.g4, PRODUCTION é a produção raiz. Esse comando imprime todas as tokens extraídas da entrada (input). Se trocar -tokens por -tree será impresso a saída com parênteses separando cada produção. Se trocar -tree por -gui ele gera uma imagem com a árvore gerada por consequência do parsing da entrada.



# Antlr - gramática .g4

```
grammar Sum;
sum: '('expr '+' expr ')';
expr: expr '+' expr # ExprSum
     | NUM          # ExprNum
     ;
NUM : [0-9]+;
ADD: '+'
```



# Antlr - gramática .g4

- `grammar Sum;`

Define a gramática “Sum”.

- `sum: '(' expr '+' expr ')';`

Produções são definidas por uma palavra de letras minúsculas seguida de `':'`. Dentro da produção podemos referenciar outras produções ou terminais, mesmo que eles ainda não tenham sido definidos. A inserção de strings como `(''`, `+` e `')` são automaticamente definidas como terminais (tokens).





## Antlr - gramática .g4

- ```
expr: expr '+' expr # ExprSum
    | NUM          # ExprNum
    ;
```

Uma expressão pode representar um OU de várias outras. A separação é feita pelo carácter '|'. Nesse caso temos duas opções, “expr '+' expr” ou “NUM”.

A cerquilha NÃO é um comentário, ela é um “label” para a opção que vem antes dela. Isso será útil no futuro. Lembre-se que a label não pode ser igual a uma produção.



# Antlr - gramática .g4

- NUM: [0-9]+;

Se um identificador estiver em maiúsculo ele é um terminal.

- ADD: '+'

Em outras partes da gramática nós também usamos o token '+' de forma explícita. Criar um terminal para ele faz com que todos os outros tokens '+' também sejam reconhecidos como ADD pelo programa.



## Antlr - SumVisitor.java

Dentre os arquivos criados depois de compilar o .g4 vai existir uma classe SumBaseVisitor (Sum pois é derivado do nome da gramática). Essa classe recebe uma árvore resultado de um “parser” e itera sobre ela.

Essa classe possui um método para cada produção, e no caso onde nós damos uma “label” para uma opção de produção ela também possui um método para cada opção.

Nós vamos herdar dessa classe e sobrescrever alguns métodos.



## Antlr - SumVisitor.java

Dentre os arquivos criados depois de compilar o .g4 vai existir uma classe SumBaseVisitor (Sum pois é derivado do nome da gramática). Essa classe recebe uma árvore resultado de um “parser” e itera sobre ela.

Essa classe possui um método para cada produção, e no caso onde nós damos uma “label” para uma opção de produção ela também possui um método para cada opção.

Nós vamos herdar dessa classe e sobrescrever alguns métodos.



## Antlr - SumVisitor.java

```
public class SumVisitor extends SumBaseVisitor<Integer> {  
    @Override  
    public Integer visitSum(SumParser.SumContext ctx) {  
        Integer val = visit(ctx.expr(0)) + visit(ctx.expr(1));  
        System.out.println(ctx.ADD().getText() + val);  
        return 0;  
    }  
    @Override  
    public Integer visitExprSum(SumParser.ExprSumContext ctx)  
        { return visit(ctx.expr(0)) + visit(ctx.expr(1)); }  
    @Override  
    public Integer VisitExprNum(SumParser.ExprNumContext ctx)  
        { return Integer.valueOf(ctx.NUM().getText()); }  
}
```



# Antlr - SumVisitor.java

Detalhes a se observar:

- O nome dos métodos é uma variação do nome da produção, assim como o nome da variável de contexto (como eles seguem camel case, a primeira letra sempre é maiúscula, mesmo que a produção seja minúscula)
- A partir das variáveis de contexto é possível se chamar um método com o nome das produções que estão dentro da atual para acessar o contexto delas. Se tiverem mais que uma igual, acessamos elas passando a sua posição por parâmetro.
- Podemos chamar métodos para acessar os “tokens” que fazem parte da produção, e a partir deles podemos chamar o método `getText()` para pegar a string que levou ao token.



## Antlr - MyParser.java

- Esse arquivo é o que vai ser responsável por juntar a parte léxica, sintática e o “visitor”.

O nome não precisa ser MyParser.java, o importante é que ele implemente a função main.

Ele vai usar os métodos gerados pelo antlr: SumLexer(), CommonTokenStream() e SummerParser(),

E o método que nós devemos criar indiretamente na classe SumVisitor, visit().

Repare no código fornecido como exemplo o comando `parser.root()`, o método chamado do parser deve ter o mesmo nome que a regra gramatical que deve ser executada como raiz.



## Antlr - Summer.g4

- No zip do lab existe a pasta summer, dentro dela estão exemplos muito semelhantes ao dado nos slides e um script mostrando como compilar e executar.
- A pasta não vem com o .jar do antlr, então vocês tem que baixar e colocar dentro dela.





## Lab3 - SimpleMath.g4

Essa laboratório consiste em vocês implementarem a gramática de uma linguagem e um visitor que vai fazer uma verificação de consistência.

Vocês devem inferir a gramática a partir dos arquivos de teste e devem usar implementar um “visitor” que gere os mesmos resultados que os apresentados.

Tentem primeiro gerar uma árvore bem estruturada, depois se preocupem em fazer as verificações.



## Lab3 - Entrega

As duplas devem entregar:

- O arquivo implementando a gramática .g4
- O arquivo implementando o visitor especial que vai encontrar as irregularidades .java
- O arquivo que implementa a main()
- Um script que compila os arquivos assumindo o .jar do antlr e os testes estão na pasta raiz, executa a main no arquivo colocando os resultados em resultX.txt (mesmo que testX.sm) e termina abrindo com a opção -gui o resultado do test7.sm

DICA: usem o HashMap do java para guardar os identificadores do programa.

DICA: CUIDADO COM O NOME DOS SEUS ARQUIVOS. Nomes simples como Parser e Lexer podem gerar conflitos difíceis de descobrir.