DANILO CHAGAS CLEMENTE, LUCAS ALVARENGA LOPES, LARA RAMOS LINHARES, MARCO TÚLIO AMARAL

Relatório técnico

Analisador Léxico, Sintático e Semântico

Lavras – MG

04/08/2024

Visão geral da linguagem: Nome e descrição das características gerais da linguagem.

A linguagem ZooLogic adota características inspiradas na linguagem C/C++ em seus quesitos sintáticos e semânticos. Já sobre sua composição lexical, é utilizado termos relacionados ao mundo animal, incluindo habitat, nome de animais e outras características que compõem a fauna e o ambiente.

GitHub do Projeto:

https://github.com/Am4ral/ZooLogic/tree/main

Analisador Léxico

Análise Léxica

lexemas aceitos, classes de lexemas e seus padrões de identificação.

Padrão	Tipo de Lexema	Sigla
Função main()	selva	MAIN
Função	arvore	FUNC
Saída (cout)	lhama	ESC
Controle (return)	desmatamento	RET
Entrada (cin)	porco	LER
Controle (if)	cobra	IF
Controle (else)	cauda	ELSE
Controle (elif)	caudaCobra	ELIF
Tipo de Dado (bool)	boi	TIPO
Tipo de Dado (float)	pato	TIPO

Padrão	Tipo de Lexema	Sigla
Tipo de Dado (int)	indio	TIPO
Tipo de Dado (string)	centopeia	TIPO
Tipo de Dado (char)	pe	TIPO
Estrutura de Repetição (loop)	formiga	FOR
Estrutura de Repetição	baleia	WHILE
Operadores Aritméticos	*,/,+,-, %	OP_ARIT
Operadores Relacionais	<,<=,>=,>,=,<>	OP_REL
Operador de Atribuição	=	OP_ATR
Operadores Condicionais	&&, ||, ==	OP_COND
Sequência de letras	[a-zA-Z]	fragment LETRA
Sequências de dígitos (sem vírgula)	NÚMERO INTEIRO	fragment DIGITO
Sequências de dígitos (com vírgula)	NÚMERO REAL	NUM
Sequências de letras e números que começam com uma letra	VARIÁVEL	VAR
Delimitador	'('	AP
Delimitador	')'	FP
Delimitador	'{'	AC
Delimitador	'}'	FC
Delimitador	‘;’	PV
Delimitador	‘ “ ‘	AASP
Comentário	‘/ /’	COMEN

Exemplos de uso da linguagem

Implementação dos algoritmos de Fatorial, Fibonacci e Soma dos N Termos da Sequência de Fibonacci.

Fatorial:

```C/C++
indio selva() {
  indio n

  indio fatorial;

  lhama << "Digite um numero (0 <= n <= 12) " << endl;
  porco << n << endl;

  fatorial = 1;
  formiga (indio i = 1; i <= n; i++) {
    fatorial *= i;
  }

    vai dar erro aqui ::::

  cout << n << "!=" << fatorial << endl;

  desmatamento 0;
}
```

Fibonacci:

```C/C++
arvore indio fibonacci(indio n) {
    cobra (n == 0 || n == 1) {
        desmatamento n;
    }
    cauda{
        desmatamento fibonacci(n - 1) + fibonacci(n - 2);
    }
}

arvore indio somaFibonacci(indio n) {
  indio soma = 0;
  formiga (indio i = 0; i < n; i++) {
```

```
    soma += fibonacci(i);
  }
  desmatamento soma;
}

selva() {
indio n;

lhama << "Digite um numero ";
porco >> n;

lhama << "O " << n << "numero da sequencia de Fibonacci eh " << fibonacci(n)
<< endl;

desmatamento 0;
}
```

Soma de N termos:

```
C/C++
arvore indio somaFibonacci(indio n) {
    indio soma = 0;

    formiga (indio i = 0; i < n; i++) {
        soma += fibonacci(i);
    }

    desmatamento soma;
}


arvore indio fibonacci(indio n) {
    cobra (n == 0 || n == 1) {
        desmatamento n;
    }

    cauda {
        desmatamento fibonacci(n - 1) + fibonacci(n - 2);
    }
}


selva() {
    indio n;

    lhama << "Digite um numero ";
```

```
    porco >> n;

      lhama << "A soma dos " << n << " primeiros termos da sequencia de
Fibonacci eh " << somaFibonacci(n) << endl;

    desmatamento 0;
}
```

Implementação do Analisador Léxico:

Descrição de cada etapa da implementação do analisador léxico, incluindo a criação da gramática e descrição de arquivos no caso de uso de geradores de analisadores, descrição do programa desenvolvido para execução do analisador léxico. São esperadas descrições dos artefatos e capturas de tela.

1. Definição Léxica da Gramática:

Primeiramente, definimos a gramática da linguagem Zoologic, identificando os tokens (lexemas) e suas regras de formação. Isso inclui palavras-chave, símbolos especiais, identificadores, números, etc. Essa definição foi feita em um arquivo *GrammarZooLogic.g4* em classes de tokens e os identificadores foram criados por fragmentos de regra.

```
Unset
MAIN: 'selva';
FUNC: 'arvore';
IF: 'cobra';
ELSE: 'cauda';
ELIF: 'caudaCobra';
RET: 'desmatamento';
FOR: 'formiga';
WHILE: 'baleia';
TIPO: 'indio' | 'pato' | 'boi' | 'pe' | 'centopeia';
AP: '(';
FP: ')';
```

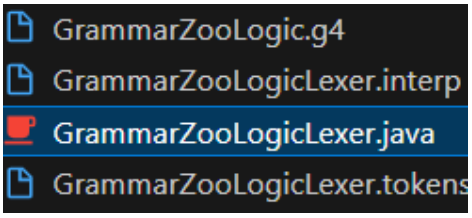
```
AC: '{';
FC: '}';
ASP: '"';
PV: ';';
COMEN: '//';
ESC: 'lhama';
LER: 'porco';
STRING: ASP (~["\r\n] | '\\' .)* ASP;
VAR: LETRA(DIGITO|LETRA)*;
NUM: DIGITO+('.'DIGITO+)?;
fragment DIGITO: [0-9];
fragment LETRA: [a-zA-Z];
OP_ARIT: '+'|'-'|'*'|'/'|'%';
OP_REL: '<'| '>'| '>='| '<='| '==' | '!=';
OP_COND: '&&' | '||';
OP_ATR: '=';
OP_CONCAT: '++';
WS: [ \r\t\n]+ ->skip;
ErrorChar: . ;
```

2. Escolha da Ferramenta:

Optamos por utilizar o ANTLR (ANother Tool for Language Recognition) devido à sua eficiência e flexibilidade na geração de analisadores léxicos e sintáticos.

3. Geração do Analisador Léxico:

Utilizamos o ANTLR em Java para gerar o analisador léxico a partir da gramática definida. Passamos o arquivo *GrammarZooLogic.g4* que possui as definições da gramática para o ANTLR e foi gerado um analisador léxico a partir do arquivo *.g4*.



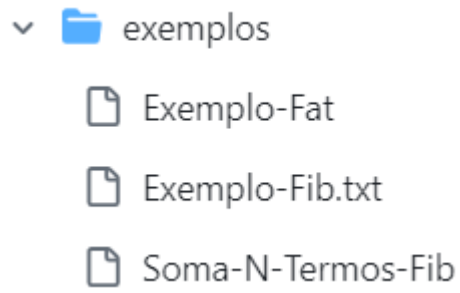
4. Desenvolvimento do Programa de Execução:

Desenvolvemos um programa em Java que utiliza o analisador léxico gerado pelo ANTLR para analisar o código fonte em Zoologic. Este programa pode ler o código fonte, tokeniza-lo usando o analisador léxico e fornecer os tokens reconhecidos como saída.

```Java
import org.antlr.v4.runtime.CharStream;
import org.antlr.v4.runtime.CharStreams;
import org.antlr.v4.runtime.Token;

import java.io.IOException;

public class ExemploLexer {

    public static void  main (String[] args){
                                String    filename    =    "F:\\Faculdade\\5
periodo\\Compiladores\\ZooLogic\\exemplos\\Exemplo-Fat";
        try{
            CharStream input = CharStreams.fromFileName(filename);
            GrammarZooLogicLexer lexer = new GrammarZooLogicLexer(input);
            Token token;
            while (!lexer._hitEOF){
                token = lexer.nextToken();
                                        System.out.println("Token:  <Classe:
"+lexer.getVocabulary().getSymbolicName(token.getType())   +"   ,Lexema:   "+
token.getText() +">");
                }

        } catch (IOException e){
            e.printStackTrace();
        }
    }
}
```

5. Teste e Depuração

Testamos o analisador léxico com três arquivos teste de código em Zoologic para garantir que os lexemas sejam corretamente reconhecidos e classificados. Realizamos depuração conforme necessário para corrigir quaisquer erros ou falhas de reconhecimento.



Casos de Teste

Apresentação do resultado da execução dos algoritmos de exemplo com e sem erros léxicos.

1. Exemplo Com Erro | Exemplo-Fat

```
C/C++
indio selva() {
  indio n

  indio fatorial;

  lhama << "Digite um numero (0 <= n <= 12) " << endl;
  porco << n << endl;

  fatorial = 1;
  formiga (indio i = 1; i <= n; i++) {
    fatorial *= i;
  }
```

```
    vai dar erro aqui ::::

  cout << n << "!=" << fatoral << endl;

  desmatamento 0;
}
```

Unset

```
.
.
.
Token: <Classe: VAR ,Lexema: vai>
Token: <Classe: VAR ,Lexema: dar>
Token: <Classe: VAR ,Lexema: erro>
Token: <Classe: VAR ,Lexema: aqui>
Token: <Classe: ErrorChar ,Lexema: :>
Token: <Classe: ErrorChar ,Lexema: :>
Token: <Classe: ErrorChar ,Lexema: :>
Token: <Classe: ErrorChar ,Lexema: :>
Token: <Classe: VAR ,Lexema: cout>
.
.
.
```

2. Caso Sem Erro | Exemplo-Fat

Sem a linha sublinhada em vermelho, temos o mesmo arquivo sem erro

Unset

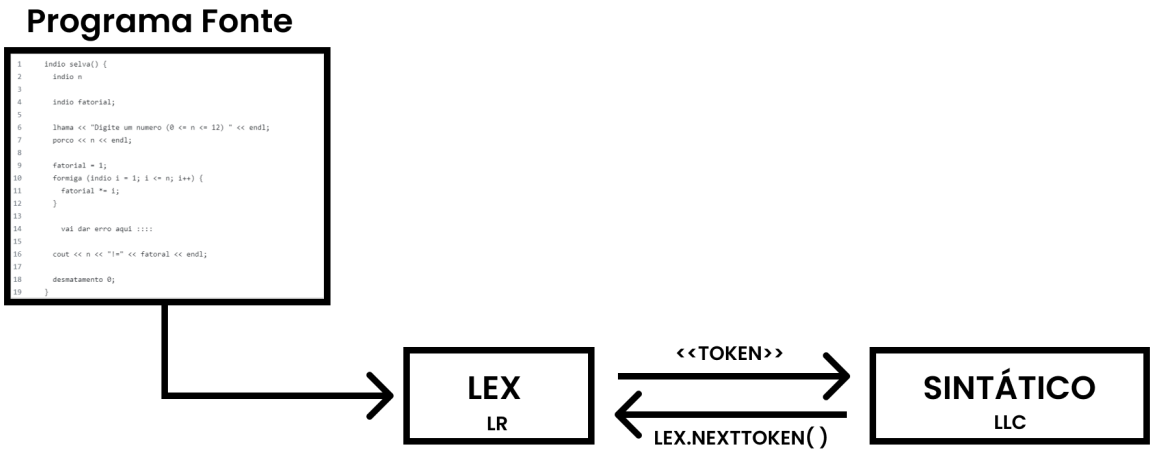
```
Token: <Classe: TIPO ,Lexema: indio>
Token: <Classe: MAIN ,Lexema: selva>
Token: <Classe: AP ,Lexema: (>
Token: <Classe: FP ,Lexema: )>
Token: <Classe: AC ,Lexema: {>
Token: <Classe: TIPO ,Lexema: indio>
Token: <Classe: VAR ,Lexema: n>
Token: <Classe: TIPO ,Lexema: indio>
Token: <Classe: VAR ,Lexema: fatorial>
Token: <Classe: PV ,Lexema: ;>
Token: <Classe: ESC ,Lexema: lhama>
Token: <Classe: OP_REL ,Lexema: <>
Token: <Classe: OP_REL ,Lexema: <>
Token: <Classe: CENTOPEIA ,Lexema: "Digite um numero (0 <= n <= 12) ">
Token: <Classe: OP_REL ,Lexema: <>
Token: <Classe: OP_REL ,Lexema: <>
```

```
Token: <Classe: VAR ,Lexema: endl>
Token: <Classe: PV ,Lexema: ;>
Token: <Classe: LER ,Lexema: porco>
Token: <Classe: OP_REL ,Lexema: <>
Token: <Classe: OP_REL ,Lexema: <>
Token: <Classe: VAR ,Lexema: n>
Token: <Classe: OP_REL ,Lexema: <>
Token: <Classe: OP_REL ,Lexema: <>
Token: <Classe: VAR ,Lexema: endl>
Token: <Classe: PV ,Lexema: ;>
Token: <Classe: VAR ,Lexema: fatorial>
Token: <Classe: OP_ATR ,Lexema: =>
Token: <Classe: NUM ,Lexema: 1>
Token: <Classe: PV ,Lexema: ;>
Token: <Classe: FOR ,Lexema: formiga>
Token: <Classe: AP ,Lexema: (>
Token: <Classe: TIPO ,Lexema: indio>
Token: <Classe: VAR ,Lexema: i>
Token: <Classe: OP_ATR ,Lexema: =>
Token: <Classe: NUM ,Lexema: 1>
Token: <Classe: PV ,Lexema: ;>
Token: <Classe: VAR ,Lexema: i>
Token: <Classe: OP_REL ,Lexema: <=>
Token: <Classe: VAR ,Lexema: n>
Token: <Classe: PV ,Lexema: ;>
Token: <Classe: VAR ,Lexema: i>
Token: <Classe: OP_ARIT ,Lexema: +>
Token: <Classe: OP_ARIT ,Lexema: +>
Token: <Classe: FP ,Lexema: )>
Token: <Classe: AC ,Lexema: {>
Token: <Classe: VAR ,Lexema: fatorial>
Token: <Classe: OP_ARIT ,Lexema: *>
Token: <Classe: OP_ATR ,Lexema: =>
Token: <Classe: VAR ,Lexema: i>
Token: <Classe: PV ,Lexema: ;>
Token: <Classe: FC ,Lexema: }>
Token: <Classe: VAR ,Lexema: cout>
Token: <Classe: OP_REL ,Lexema: <>
Token: <Classe: OP_REL ,Lexema: <>
Token: <Classe: VAR ,Lexema: n>
Token: <Classe: OP_REL ,Lexema: <>
Token: <Classe: OP_REL ,Lexema: <>
Token: <Classe: CENTOPEIA ,Lexema: "!=">
Token: <Classe: OP_REL ,Lexema: <>
Token: <Classe: OP_REL ,Lexema: <>
Token: <Classe: VAR ,Lexema: fatoral>
Token: <Classe: OP_REL ,Lexema: <>
Token: <Classe: OP_REL ,Lexema: <>
Token: <Classe: VAR ,Lexema: endl>
Token: <Classe: PV ,Lexema: ;>
Token: <Classe: RET ,Lexema: desmatamento>
Token: <Classe: NUM ,Lexema: 0>
Token: <Classe: PV ,Lexema: ;>
Token: <Classe: FC ,Lexema: }>
```

Analisador Sintático

Análise Sintática

Conceitos utilizados para a definição do analisador sintático: estratégia para gerar uma árvore de derivação válida a partir de tokens;



O exemplo utilizado para a construção da ideia foi o *Exemplo-fat,* no qual, a principal ideia é dividi-lo em tokens, como abaixo:

```
Unset
<indio, TIPO> <selva, MAIN> <(), MAIN> <{, AC> <indio, TIPO>
<n, VAR> <indio, TIPO> <fatorial, VAR> <;, PV> <lhama, ESC>
<<, ESC> <"Digite um numero (0 <= n <= 12) ", ESC> <<, ESC>
<endl, ESC> <;, PV> <porco, LER> <<, LER> <n, VAR> <<, LER>
<endl, LER> <;, PV> <fatorial, VAR> <=, OP_ATR> <1, DIGITO>
<;, PV> <formiga, FOR> <(, AP> <indio, TIPO> <i, VAR>
<=,OP_ATR> <1,DIGITO> <;, PV> <i, VAR> <=, OP_REL> <n, VAR>
<;, PV> <i++, OP_ARIT> <), FP> <{, AC> <fatorial, VAR> <*=>
<i, VAR> <;, PV> <}, FC> <vai, ERRO> <dar, ERRO> <erro, ERRO>
<aqui, ERRO> <::::, ERRO> <cout, ESC> <<, ESC> <n, VAR> <<,
ESC> <"!="> <<, ESC> <fatorial, VAR> <<, ESC> <endl, ESC> <;,
PV> <desmatamento, RET> <0, DIGITO> <;, PV> <}, FC>
```

O parser tentará construir uma árvore de sintaxe a partir dos tokens. Quando encontrar tokens marcados como `ERRO`, ou uma sequência de tokens que não corresponde a nenhuma regra de produção, ele gerará um erro sintático.
O parser encontra os tokens `ERRO` ou uma sequência de tokens que não corresponde a nenhuma regra válida da gramática.

Gera um erro sintático e possivelmente fornece uma mensagem de erro indicando a linha e a posição onde o erro ocorreu.
Exemplo *Exemplo-Fib*:

```
arvore indio fibonacci(indio n) {
    cobra (n == 0 || n == 1) {
        desmatamento n;
    }
    cauda{
        desmatamento fibonacci(n - 1) + fibonacci(n - 2);
    }
}

arvore indio somaFibonacci(indio n) {
  indio soma = 0;
  formiga (indio i = 0; i < n; i++) {
    soma += fibonacci(i);
  }
  desmatamento soma;
}

selva() {
indio n;

lhama << "Digite um numero ";
porco >> n;

lhama << "O " << n << "numero da sequencia de Fibonacci eh " << fibonacci(n) << endl;

desmatamento 0;
}
```

Os tokens seriam formados dessa maneira:

Unset

```
<arvore, FUNC> <indio, TIPO> <fibonacci, VAR> <(, AP> <indio, TIPO> <n, VAR> <), FP> <{, AC> <cobra, IF> <(, AP> <n, VAR> <==, OP_COND> <0, DIGITO> <||, OP_COND> <n, VAR> <==, OP_COND> <1, DIGITO> <), FP> <{, AC> <desmatamento, RET> <n, VAR> <;, PV> <}, FC> <cauda, ELSE> <{, AC> <desmatamento, RET> <fibonacci, VAR> <(, AP> <n, VAR> <-, OP_ARIT> <1, DIGITO> <), FP> <+, OP_ARIT> <fibonacci, VAR> <(, AP> <n, VAR> <-, OP_ARIT> <2, DIGITO> <), FP> <;, PV> <}, FC> <}, FC> <arvore, FUNC> <indio, TIPO> <somaFibonacci, VAR> <(, AP> <indio, TIPO> <n, VAR> <), FP> <{, AC> <indio, TIPO> <soma, VAR> <=, OP_ATR> <0, DIGITO> <;, PV> <formiga, FOR> <(, AP> <indio, TIPO> <i, VAR> <=, OP_ATR> <0, DIGITO> <;, PV> <i, VAR> <<, OP_REL> <n, VAR> <;, PV> <i++, OP_ARIT> <), FP> <{, AC> <soma, VAR> <+=, OP_ARIT> <fibonacci, VAR> <(, AP> <i, VAR> <), FP> <;, PV> <}, FC> <desmatamento, RET> <soma, VAR> <;, PV> <}, FC> <selva, MAIN> <(, AP> <), FP> <{, AC> <indio, TIPO> <n, VAR> <;, PV> <lhama, ESC> <<, ESC> <"Digite um numero ", ESC> <;, PV> <porco, LER> <>>, LER> <n, VAR> <;, PV> <lhama, ESC> <<, ESC> <"O ", ESC> <<, ESC> <n, VAR> <<, ESC> <"numero da sequencia de Fibonacci eh ", ESC> <<, ESC> <fibonacci, VAR> <(, AP> <n,
```

```
VAR> <), FP> <<, ESC> <endl, ESC> <;, PV> <desmatamento, RET>
<0, DIGITO> <;, PV> <}, FC>
```

As regras de produção podem ser listadas dessa maneira para a linguagem acima:

```
Unset
Programa -> FuncDecl FuncDecl MainDecl
FuncDecl -> Tipo FuncName <AP> ParamList <FP> <AC> Corpo <FC>
MainDecl -> <selva, MAIN> <AP> <FP> <AC> Corpo <FC>
Tipo -> <indio, TIPO>
FuncName -> <fibonacci, VAR> | <somaFibonacci, VAR>
ParamList -> Param | ParamList <,> Param
Param -> Tipo VarNome
Corpo -> DeclVar* Comando*
DeclVar -> Tipo VarNome <PV>
VarNome -> <LETRA><VAR>
Comando -> Controle | Loop | Entrada | Saida | Retorno |
Atribuicao
Controle -> IfElse
IfElse -> <cobra, IF> <AP> Cond <FP> <AC> Corpo <FC> ElsePart
ElsePart -> <cauda, ELSE> <AC> Corpo <FC> | ε
Cond -> VarNome <OP_COND> VarNome
Loop -> <formiga, FOR> <AP> Init Cond <PV> Incremento <FP>
<AC> Corpo <FC>
Init -> Tipo VarNome <OP_ATR> Num <PV>
Incremento -> VarNome <++> | VarNome <->
Entrada -> <porco, LER> <>> VarNome <PV>
Saida -> <lhama, ESC> << Texto << VarNome << <endl, ESC> <PV>
Retorno -> <desmatamento, RET> VarNome <PV>
Atribuicao -> VarNome <OP_ATR> Expr <PV>
Expr -> Termo | Termo <OP_ARIT> Expr
Termo -> VarNome | Num
Num -> <DIGITO>
```

Formando a partir disso uma árvore de derivação. E a partir desses exemplos e derivações, foi criado uma Gramática Livre de Contexto que satisfaça totalmente as questões que envolvem a gramática ZooLogic, que será explicado logo abaixo.

Implementação do Analisador Sintático

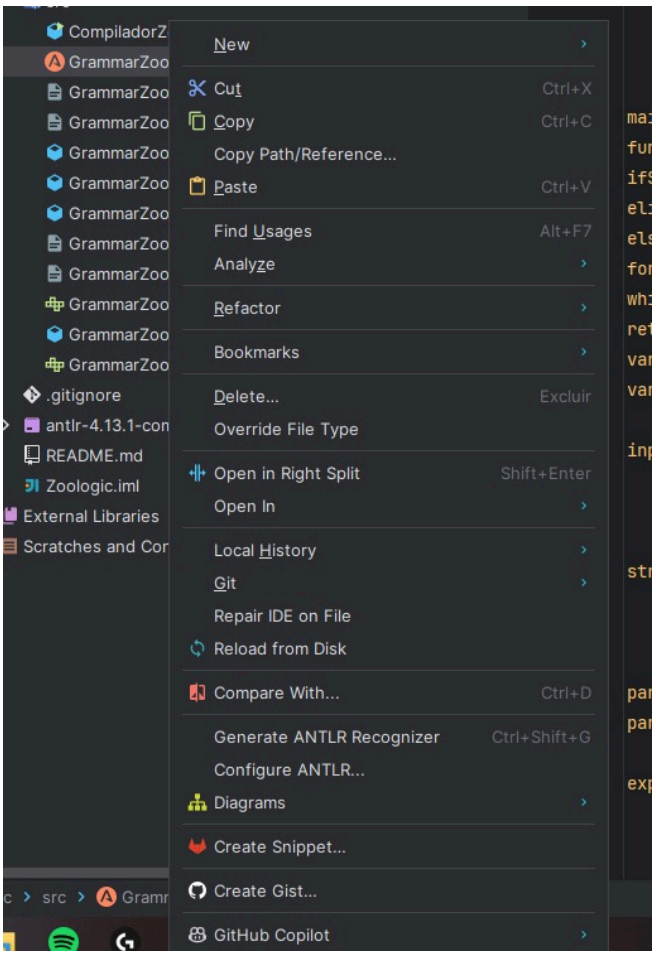
1. Definição Sintática da Linguagem

Gramática Livre de Contexto que define a linguagem.

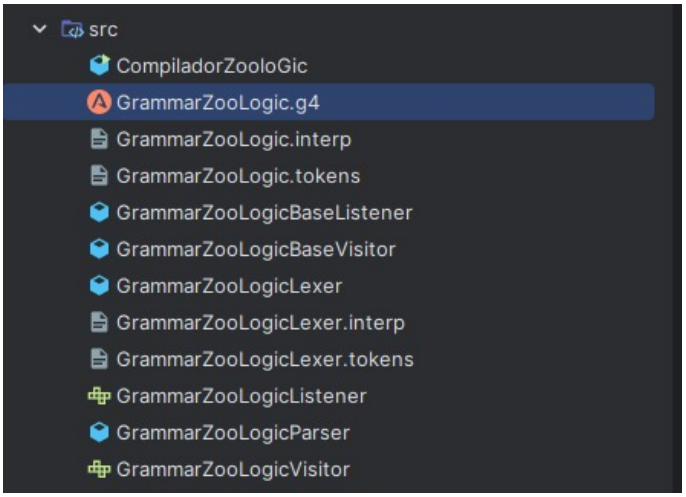
```
Java
prog:   (stmt)*;
stmt:   mainStmt
    |   funcDef
    |   ifStmt
    |   forStmt
    |   whileStmt
    |   retStmt
    |   varDecl
    |   inputOutput
    |   varAssign
    ;
mainStmt:   'selva''()' '{' stmt* '}' ;
funcDef:    'arvore' TIPO '(' paramList? ')' '{' stmt* '}' ;
ifStmt:     'cobra' '(' expr ')' '{' stmt* '}' (elifStmt)* (elseStmt)? ;
elifStmt:   'caudaCobra' '(' expr ')' '{' stmt* '}' ;
elseStmt:   'cauda' '{' stmt* '}' ;
forStmt:    'formiga' '(' varDecl expr ';' expr ')' '{' stmt* '}' ;
whileStmt:  'baleia' '(' expr ')' '{' stmt* '}' ;
retStmt:    'desmatamento' expr ';' ;
varDecl:    TIPO VAR ('=' expr)? ';' ;
varAssign:  VAR '=' expr ';' ;
inputOutput: 'lhama' '(' (stringExpr | VAR | NUM) ')' ';'
           | 'porco' '(' VAR ')' ';'
           ;
stringExpr:  STRING
           | STRING OP_CONCAT (VAR | NUM | STRING | '(' expr ')')
           ;
paramList:  param (',' param)* ;
param:      TIPO VAR ;
expr:       expr OP_ARIT expr
    |       expr OP_REL expr
    |       expr OP_COND expr
    |       '(' expr ')'
    |       NUM
    |       STRING
    |       VAR
    |       'lhama' expr
    |       'porco' expr;
```

2. Geração do Analisador Sintático

Após definir a Gramática Livre de Contexto da linguagem dentro dos padrões esperados pelo ANTLR foi executado o comando *Generate ANTLR Recognizer*



E o arquivo *GrammarZooLogicParser* foi criado, no qual será feito o desenvolvimento do programa.



3. Desenvolvimento do Programa de Execução

Depois de gerar o analisador sintático, foi necessária a construção da parte do código responsável por executar e reconhecer erros sintáticos com base na árvore de tokens gerada. Para isso, foi realizada a seguinte implementação:

```java
public static void  main (String[] args){
                        String   filename   =   "F:\\Faculdade\\Matérias\\5 periodo\\Compiladores\\ZooLogic\\exemplos\\Exemplo-Fat";
        try{
            CharStream input = CharStreams.fromFileName(filename);
            GrammarZooLogicLexer lexer = new GrammarZooLogicLexer(input);
            CommonTokenStream tokens = new CommonTokenStream(lexer);
                                    GrammarZooLogicParser   parser   =   new GrammarZooLogicParser(tokens);

            GrammarZooLogicParser.ProgContext ast = parser.prog();

            System.out.println(ast.toStringTree(parser));
        } catch (IOException e){
            e.printStackTrace();
        }
    }
}
```

Onde, aqui, especificamos o arquivo do código realizado, geramos uma “CommonTokenStream” que é gerada a partir do resultado do lexer e com ela obtemos a árvore de parser(ou ParseTree) responsável por mostrar os tokens em formato de árvore.

4. Teste e Depuração

Caso de Teste

Apresentação do resultado da execução dos algoritmos de exemplo com e sem erros léxicos. Durante o desenvolvimento do analisador sintático, o ZooLogic passou por atualizações de gramática e definição. Dessa forma, os códigos apresentados para teste anteriormente são diferentes dos códigos a seguir.

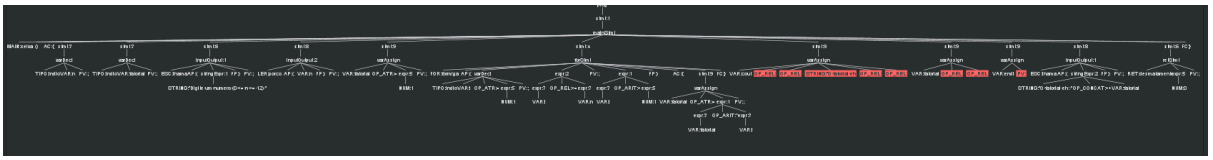
1. Exemplo Com Erro | Exemplo-Fat

```c
selva() {
```

```
  indio n;

  indio fatorial;

  lhama("Digite um numero (0 <= n <= 12) ");
  porco(n);

  fatorial = 1;
  formiga (indio i = 1; i <= n; i+1) {
    fatorial = fatorial * i;
  }

  cout << "O fatorial eh: " << fatorial << endl;

  lhama("O fatorial eh: " ++ fatorial);

  desmatamento 0;
}
```

A palavra “cout”, “<<” e “endl” não fazem parte do alfabeto do ZooLogic. Logo, geram erros de derivação, que não eram gerados antes pelo analisador léxico.

1. Árvore de Derivação com Erro (clique aqui):



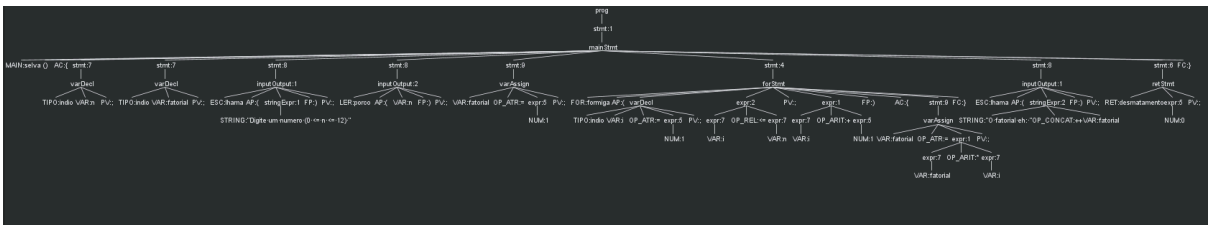
2. Caso Sem Erro | Exemplo-Fat

```C/C++
selva() {
  indio n;

  indio fatorial;

  lhama("Digite um numero (0 <= n <= 12) ");
  porco(n);

  fatorial = 1;
  formiga (indio i = 1; i <= n; i+1) {
    fatorial = fatorial * i;
  }
```

```
  lhama("O fatorial eh: " ++ fatorial);

  desmatamento 0;
}
```

Nesse código, removemos a linha que continha palavras não deriváveis. Dessa forma, o analisador não apresenta erros, e a árvore é gerada sem problemas

2. Árvore de Derivação sem Erro (clique aqui):



Implementação do Analisador Semântico

1. Definição Semântica de Linguagem

Na parte de análise semântica da linguagem utilizamos a implementação de um listener para criar as seguintes checagens:

- **Checagem de tipo de variável:** um valor não pode ser atribuído a uma variável de um tipo incompatível.
- **Checagem de variáveis não declaradas/escopo:** variáveis que não foram declaradas no escopo, ou variáveis que foram declaradas num escopo diferente não podem ser utilizadas no programa.
- **Checagem de variáveis duplicas:** duas variáveis não podem ter o mesmo nome.

2. Implementação do Analisador Semântico

O listener da linguagem utiliza a ideia de ter uma pilha de escopos, e ela é utilizada para fazer as checagens.

Checagem de Tipo de Variável

A checagem de tipo de variável é feito tanto na entrada da regra de declaração quanto na entrada da regra de atribuição. De acordo com a gramática definida na análise sintática, esperamos um regra do tipo expr para realizar a atribuição de valor, tanto para atribuição isolada ou atribuição junto com a declaração. Dessa forma, verificamos se a regra expr não é nula e fazemos a verificação. Segue o código:

Para atribuições em conjunto com a declaração:

```java
 if (ctx.expr() != null) {
            String exprType = getType(ctx.expr());
            if (!varType.equals(exprType)) {
                        throw new RuntimeException("Erro de tipo: Tentando inicializar a variável '" + varName + "' do tipo " + varType + " com um valor do tipo " + exprType);
            }
        }
```

Para atribuições isoladas:

```java
       String varType = getVariableType(varName);
       String exprType = getType(ctx.expr());


       if (!varType.equals(exprType)) {
              throw new RuntimeException("Erro de tipo: Tentando atribuir um valor do tipo " + exprType + " à variável " + varName + " do tipo " + varType);
       }
```

A função *getType()* utilizada nos dois casos foi uma função criada para extrair o tipo de uma regra do tipo *expr*. Ela verifica a expressão para cada tipo possível e retorna qual o tipo encontrado. Segue o código:

```java
private String getType(GrammarZooLogicParser.ExprContext ctx) {
        if (ctx.NUM() != null) {
            return "indio";
        } else if (ctx.STRING() != null) {
            return "centopeia";
        } else if (ctx.VAR() != null) {
            return getVariableType(ctx.VAR().getText());
        } else if (ctx.expr().size() > 1) {
            return getType(ctx.expr(0));
             } else if (ctx.OP_ARIT() != null || ctx.OP_REL() != null || ctx.OP_COND() != null || ctx.OP_ATR() != null) {
            return "indio";
        }
        return "desconhecido";
```

Checagem de Variáveis Não Declaradas/Escopo

A verificação de escopo e variáveis não declaradas, utiliza da estratégia de pilha de escpos descrita mais cedo. Quando entramos na regra de atribuição de variáveis, essa checagem:

```java
 if (!isVariableDeclared(varName)) {
            throw new RuntimeException("Erro: Variável '" + varName + "' não foi declarada.");
        }
```

A função *isVariableDeclared()* percorre a pilha de escopos verificando se aquela váriavel está declarada dentro do escopo local, retornando verdadeiro ou falso.

```java
 private boolean isVariableDeclared(String varName) {
        for (Map<String, String> scope : scopeStack) {
            if (scope.containsKey(varName)) {
                return true;
            }
        }
        return false;
    }
```

Checagem de Variáveis Duplicas

Para a checagem do tipo de variável, seguimos o vídeo tutorial disponibilizado no Campus virutal. Foram realizados adaptações para a nossa linguagem. Segue o código:

```java
@Override
       public  void  enterNDeclaracao(GrammarZooLogicParser.NDeclaracaoContext ctx) {
        String varName = ctx.VAR().getText();
        String varType = ctx.TIPO().getText();


        if (currentScope().containsKey(varName)) {
             throw new RuntimeException("Erro: Variável '" + varName + "' já foi declarada.");
        }
```

```
        currentScope().put(varName, varType);
```

Nesse código, sempre que entramos na regra de declaração de variável, verificamos se ela já existe no escopo local, se sim disparamos um erro senão adicionamos ela ao escopo local.

3. Casos de Teste

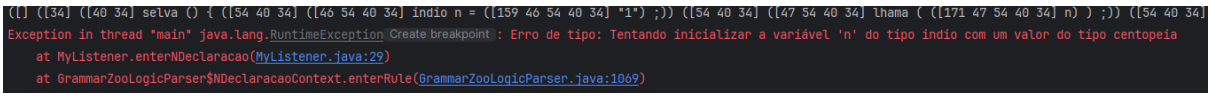
Variável Duplicada

```
Unset
selva() {
    indio n = 1;
    indio n = 2;

    lhama(n);

    desmatamento 0;
}
```

```
([] ([34] ([40 34] selva () { ([54 40 34] ([46 54 40 34] indio n = ([159 46 54 40 34] 1) ;)) ([54 40 34] ([46 54
Exception in thread "main" java.lang.RuntimeException Create breakpoint : Erro: Variável 'n' já foi declarada.
    at MyListener.enterNDeclaracao(MyListener.java:19)
```

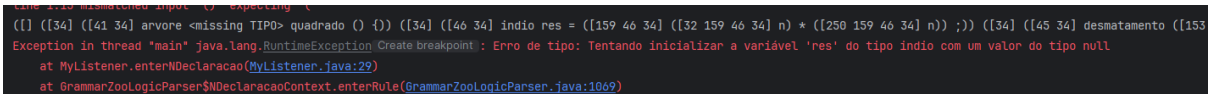
Tipo Incompatível

```
Unset
selva() {
    indio n = "1";

    lhama(n);

    desmatamento 0;
}
```



Variável Fora do Escopo

```
arvore quadrado(){
	indio res = n*n;
	desmatamento res;
}

selva(){
	indio n = 2;
	indio res;
	res = quadrado();
	desmatamento 0;
}
```



Variável Não Declarada

```
selva(){
	indio n = 2;
	soma = n + n;
	desmatamento soma;
}
```