

Relatório Trabalho de Compiladores

Nome dos alunos:

Pedro Henrique Cabral Moreira

Paulo Eduardo Pereira Carvalho

Link github: <https://github.com/PauloEduCarvalho/trabalhoCompiladores>

Visão Geral da Linguagem

Nome da Linguagem: CPP - (Camilly Pedro Paulo)

Descrição Geral:

A linguagem CPP foi criada com inspiração em 3 grandes estudantes de computação: Camilly, Pedro e Paulo. Conta com uma sintaxe mais simples e possui as palavras reservadas em português.

A linguagem possui os seguintes elementos:

- Tipos de dados básicos: `int`, `real`, `bool`, `letra`, `palavra`.
- Comandos básicos: Atribuição(“=”), leitura de entradas(“`entrada`”), e escrita de saídas(“`saída`”).
- Operações aritméticas: Soma(“+”), subtração(“-”), multiplicação(“*”) e divisão(“/”).
- Operações relacionais: Comparações de igualdade(“==”), diferença(“!=”), maior(“>”) e menor(“<”).
- Estruturas de controle: Condicional (“`se`”, “`senão se`” e “`senão`”) e laços de repetição (“`enquanto`”).
- Suporte a funções com passagem de parâmetros e retorno ou sem retorno, com a palavra reservada “`retornar`”.

Características principais:

- É uma linguagem fortemente tipada, exigindo que as variáveis sejam declaradas com tipos explícitos.
- Suporta controle de fluxo através de condicionais e repetições.
- Tem suporte a funções com recursividade, como no exemplo clássico de fatorial.
- Sua sintaxe foi criada para ser simples, buscando reduzir a quantidade de operadores complexos.

Definição Léxica:

Lexemas aceitos pela linguagem conforme a tabela

Categoria	Lexema	Padrão
Palavras-chave	<code>int</code> , <code>real</code> , <code>bool</code> , <code>letra</code> , <code>palavra</code> , <code>entrada</code> , <code>saída</code> , <code>se</code> , <code>senão</code> , <code>enquanto</code> , <code>faça</code> , <code>retornar</code> , <code>ou</code> ,	Palavras reservadas fixas

	e	
Operadores	=, +, -, *, /, >, <, ==, !=	Operadores aritméticos e relacionais
Delimitadores	`	“ ”, “()” “ ” “{}”
Números inteiros	Ex: 123	[0-9]+
Números reais	Ex: 12.34	[0-9]+\.[0-9]+
Booleanos	true, false	Valores booleanos pré-definidos
Caracteres	Ex: 'a'	[a-zA-z]
Strings	Ex: "texto"	"\".*\""
Identificadores	Ex: nome_var	[a-zA-Z_][a-zA-z0-9]
Comentários	Ex: // comentário	Comentário de uma linha

Análise Léxica - Gramatica

/*

Regras Léxicas

*/

DEC : 'DECLARACOES';

ALG : 'ALGORITMO';

// Palavras-chave

TIPO : 'int' | 'real' | 'bool' | 'letra' | 'palavra';

ENTRADA : 'entrada';

SAIDA : 'saida';

SE : 'se';

SENAO : 'senão';

ENQUANTO : 'enquanto';

FAZER : 'faça';

RETORNAR : 'retornar';

OU : 'ou';

E : 'e';

// Operadores

OP_ARIT : '+' | '-' | '*' | '/';

OP_COND : '>' | '<' | '==' | '!=';

OP_ATR : '=';

// Outros símbolos

COLON : ':';

PIPE : '|';

VIRGULA : ',';

```

LPAREN    : '(';
RPAREN    : ')';
LCHAVE    : '{';
RCHAVE    : '}';
PONTO_VIRGULA : ';';

// Literais
INT       : DIGITO+ ;
REAL      : DIGITO+ '.' DIGITO+ ;
BOOL      : 'verdadeiro' | 'falso' ;
PALAVRA   : '"' (~'"')* '"';

// Identificadores
ID        : LETRA (DIGITO | LETRA)* ;

// Fragmentos
fragment DIGITO : [0-9];
fragment LETRA  : [a-zA-Z];

// Ignorar espaços em branco e novas linhas
WS        : [ \t\r\n]+ -> skip;

```

Exemplos de uso da linguagem

fatorial:

```

fatorial | int n | {
    se | n == 0 ou n == 1 | faça {
        retornar 1;
    } senão faça {
        retornar n * fatorial| n-1|;
    }
}

```

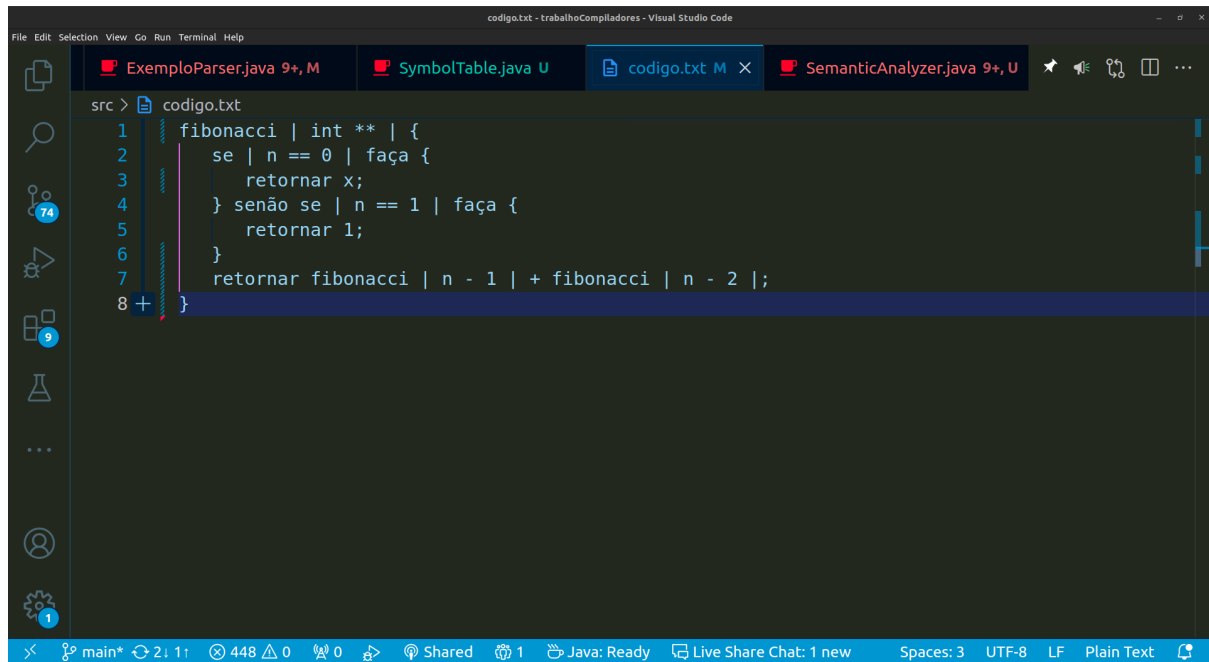
fibonacci:

```

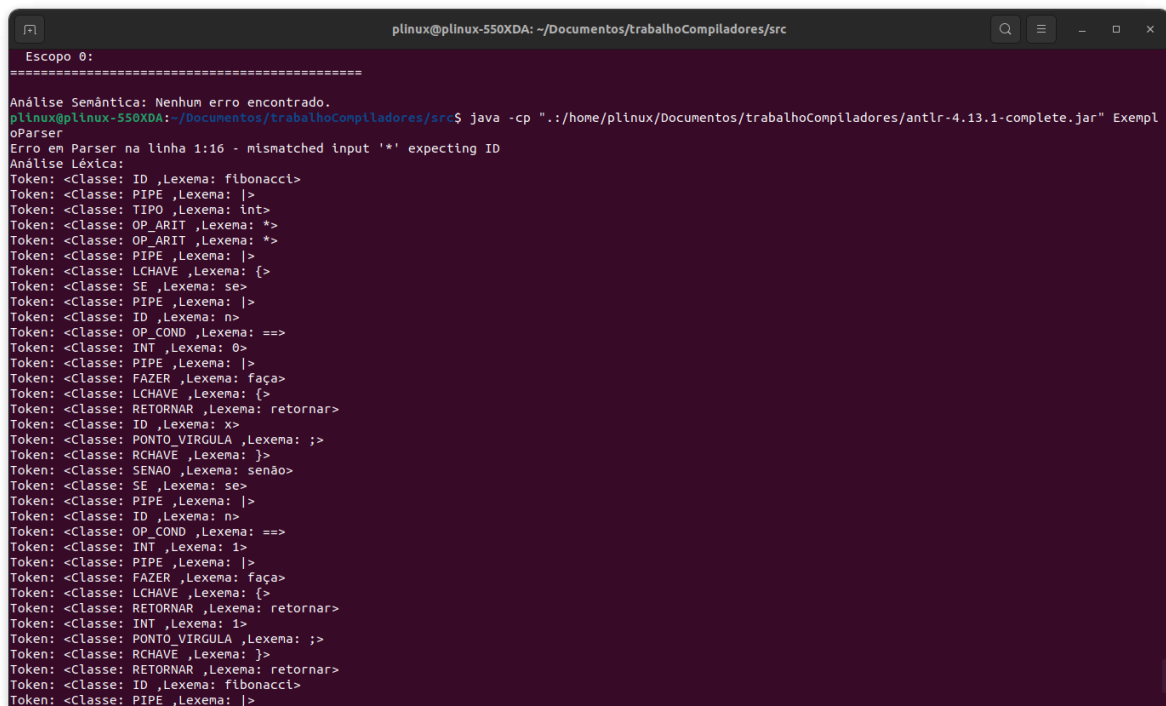
fibonacci | int n | {
    se | n == 0 | faça {
        retornar 0;
    } senão se | n == 1 | faça {
        retornar 1 ;
    }senão faça{
        retornar fibonacci | n - 1 | + fibonacci | n - 2 |;
    }
}

```

Print de tela de código com erro léxico e a saída gerada pelo analisador léxico



```
src > codigo.txt
1 fibonacci | int ** | {
2     se | n == 0 | faça {
3         retornar x;
4     } senão se | n == 1 | faça {
5         retornar 1;
6     }
7     retornar fibonacci | n - 1 | + fibonacci | n - 2 |;
8 }
```



```
plinux@plinux-550XDA: ~/Documentos/trabalhoCompiladores/src
Escopo 0:
=====
Análise Semântica: Nenhum erro encontrado.
plinux@plinux-550XDA:~/Documentos/trabalhoCompiladores/src$ java -cp ".:~/home/plinux/Documentos/trabalhoCompiladores/antlr-4.13.1-complete.jar" ExemploParser
Erro em Parser na linha 1:16 - mismatched input '**' expecting ID
Análise Léxica:
Token: <Classe: ID ,Lexema: fibonacci>
Token: <Classe: PIPE ,Lexema: |>
Token: <Classe: TIPO ,Lexema: int>
Token: <Classe: OP_ARIT ,Lexema: ==>
Token: <Classe: OP_ARIT ,Lexema: ==>
Token: <Classe: PIPE ,Lexema: |>
Token: <Classe: LCHAVE ,Lexema: {>
Token: <Classe: SE ,Lexema: se>
Token: <Classe: PIPE ,Lexema: |>
Token: <Classe: ID ,Lexema: n>
Token: <Classe: OP_COND ,Lexema: ==>
Token: <Classe: INT ,Lexema: 0>
Token: <Classe: PIPE ,Lexema: |>
Token: <Classe: FAZER ,Lexema: faça>
Token: <Classe: LCHAVE ,Lexema: {>
Token: <Classe: RETORNAR ,Lexema: retornar>
Token: <Classe: ID ,Lexema: x>
Token: <Classe: PONTO_VIRGULA ,Lexema: ;>
Token: <Classe: RCHAVE ,Lexema: }>
Token: <Classe: SENAO ,Lexema: senão>
Token: <Classe: SE ,Lexema: se>
Token: <Classe: PIPE ,Lexema: |>
Token: <Classe: ID ,Lexema: n>
Token: <Classe: OP_COND ,Lexema: ==>
Token: <Classe: INT ,Lexema: 1>
Token: <Classe: PIPE ,Lexema: |>
Token: <Classe: FAZER ,Lexema: faça>
Token: <Classe: LCHAVE ,Lexema: {>
Token: <Classe: RETORNAR ,Lexema: retornar>
Token: <Classe: INT ,Lexema: 1>
Token: <Classe: PONTO_VIRGULA ,Lexema: ;>
Token: <Classe: RCHAVE ,Lexema: }>
Token: <Classe: RETORNAR ,Lexema: retornar>
Token: <Classe: ID ,Lexema: fibonacci>
Token: <Classe: PIPE ,Lexema: |>
```

2 - Análise Sintática

Nessa parte do trabalho desenvolvemos a gramática livre de contexto da linguagem CPP, nessa gramática formalizamos descrição da repetição, da condicional, definição de funções, chamadas de função, parâmetros de função, etc.

Definição sintática da linguagem CPP

grammar CPP;

```

// Programa principal
programa
    : funcao+ ;

// Definição de função
funcao
    : ID PIPE parametros PIPE LCHAVE comando* RCHAVE ;

// Parâmetros de função
parametros
    : (tipo ID (VIRGULA tipo ID)*)? ;

// Comandos gerais
comando
    : atribuicao
    | condicional
    | enquanto
    | retorno
    | chamadaFuncao PONTO_VIRGULA
    | expressao PONTO_VIRGULA ;

// Comando de atribuição
atribuicao
    : ID OP_ATR expressao PONTO_VIRGULA ;

// Comando de retorno
retorno
    : RETORNAR expressao PONTO_VIRGULA ;

// Estruturas de controle
condicional
    : SE PIPE condicao PIPE FAZER LCHAVE comando* RCHAVE SENAO?
      (SENAO FAZER LCHAVE comando* RCHAVE
      | SENAO SE PIPE condicao PIPE FAZER LCHAVE comando* RCHAVE)? ;

// Estrutura de repetição
enquanto
    : ENQUANTO PIPE condicao PIPE FAZER LCHAVE comando* RCHAVE ;

// Operadores lógicos
operadorLogico
    : OU
    | E ;

// Condições e expressões
condicao
    : expressao (operadorComparacao expressao | operadorLogico expressao)* ;

```

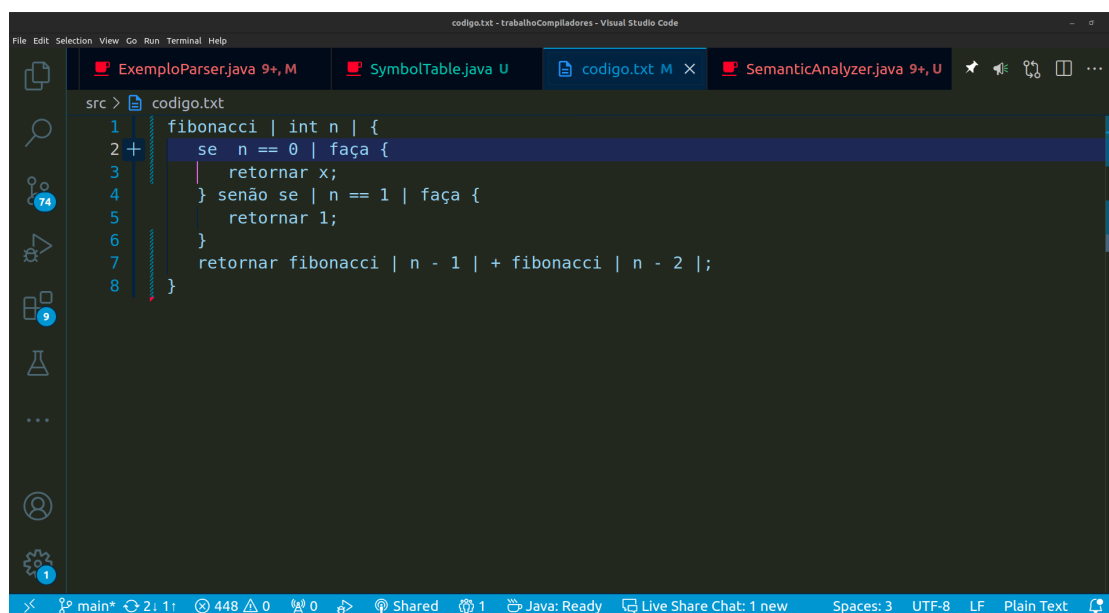
```

// Expressões e termos
expressao
    : termo ((OP_ARIT) termo)* ;

termo
    : fator ((OP_ARIT) fator)* ;
fator
    : INT
    | REAL
    | BOOL
    | PALAVRA
    | ID
    | LPAREN expressao RPAREN
    | chamadaFuncao ;
// Chamadas de função
chamadaFuncao
    : ID PIPE argumentos PIPE ;
// Argumentos de funções
argumentos
    : expressao (VIRGULA expressao)* | ;
// Tipos e operadores
tipo
    : TIPO ;
operadorComparacao
    : OP_COND ;

```

Print de tela de código com erro sintático e a saída gerada pelo analisador sintático



The screenshot shows the Visual Studio Code editor with a file named 'codigo.txt' open. The code is a Java-like pseudocode for a Fibonacci function. A syntax error is highlighted on line 2, where the code is 'se n == 0 | faça {'. The error message 'SyntaxError: Unexpected token' is visible in the output panel. The code is as follows:

```

1 fibonacci | int n | {
2 +   se n == 0 | faça {
3     |   retornar x;
4     } senão se | n == 1 | faça {
5       |   retornar 1;
6     }
7   retornar fibonacci | n - 1 | + fibonacci | n - 2 |;
8 }

```

The output panel shows the following error message:

```

SyntaxError: Unexpected token

```

```
plinux@plinux-550XDA: ~/Documentos/trabalhoCompiladores/src
Escopos ativos (1):
Escopo 0:
=====
Análise Semântica: Nenhum erro encontrado.
plinux@plinux-550XDA:~/Documentos/trabalhoCompiladores/src$ java -cp " ../home/plinux/Documentos/trabalhoCompiladores/antlr-4.13.1-complete.jar" Exenpl
oParser
Erro em Parser na linha 2:7 - missing '|' at 'n'
Análise Léxica:
Token: <Classe: ID ,Lexema: fibonacci>
Token: <Classe: PIPE ,Lexema: |>
Token: <Classe: TIPO ,Lexema: int>
Token: <Classe: ID ,Lexema: n>
Token: <Classe: PIPE ,Lexema: |>
Token: <Classe: LCHAVE ,Lexema: {>
Token: <Classe: SE ,Lexema: se>
Token: <Classe: ID ,Lexema: n>
Token: <Classe: OP_COND ,Lexema: ==>
Token: <Classe: INT ,Lexema: 0>
Token: <Classe: PIPE ,Lexema: |>
Token: <Classe: FAZER ,Lexema: faça>
Token: <Classe: LCHAVE ,Lexema: {>
Token: <Classe: RETORNAR ,Lexema: retornar>
Token: <Classe: ID ,Lexema: x>
Token: <Classe: PONTO_VIRGULA ,Lexema: ;>
Token: <Classe: RCHAVE ,Lexema: }>
Token: <Classe: SENAO ,Lexema: senão>
Token: <Classe: SE ,Lexema: se>
Token: <Classe: PIPE ,Lexema: |>
Token: <Classe: ID ,Lexema: n>
Token: <Classe: OP_COND ,Lexema: ==>
Token: <Classe: INT ,Lexema: 1>
Token: <Classe: PIPE ,Lexema: |>
Token: <Classe: FAZER ,Lexema: faça>
Token: <Classe: LCHAVE ,Lexema: {>
Token: <Classe: RETORNAR ,Lexema: retornar>
Token: <Classe: INT ,Lexema: 1>
Token: <Classe: PONTO_VIRGULA ,Lexema: ;>
Token: <Classe: RCHAVE ,Lexema: }>
Token: <Classe: RETORNAR ,Lexema: retornar>
Token: <Classe: ID ,Lexema: fibonacci>
Token: <Classe: PIPE ,Lexema: |>
Token: <Classe: ID ,Lexema: n>
```

3 - Análise Semântica

Nessa parte do trabalho desenvolvemos a análise semântica da linguagem CPP, desenvolvemos a tabela de símbolos, checagem de tipo, checagem de variáveis não declaradas, checagem de declarações duplicadas de variáveis, checagem de escopo de variáveis

Implementação do analisador semântico

Nosso analisador semantico fez com que tivéssemos que alterar e incrementar nossa gramatica com novas regras para se adequar ao analisador.

Algumas das regras adicionadas/alteradas foram:

programa:

funcao+ EOF **#NInicio;**

// Definição de função

funcao : ID PIPE parametros PIPE LCHAVE comando RCHAVE

#NPrincipal;

// Comando de atribuição

atribuicao :

ID OP_ATR expressao PONTO_VIRGULA **#NAtribuicao;**

criacao :

TIPO ID OP_ATR fator PONTO_VIRGULA **#NCriacao;**

// Expressões e termos

expressao :

termo ((OP_ARIT) termo)* **#NExpressao;**

// Argumentos de funções

argumentos :

expressao (VIRGULA expressao)* **#NArgumentos;**

...

Para não ficar ambíguo, adicionamos uma parte da gramática, para destacar os rótulos e porque eles são importantes. Uma explicação detalhada dele seria utilizar um exemplo para o melhor entendimento: ao sobrescrever um método, podemos alterar por exemplo, o comportamento de uma condicional, se uma comparação entre tipos diferentes deve retornar um erro ou não, no caso, temos duas variáveis ao lado de um operador de comparação, nesse pequeno trecho, podemos verificar que, caso as variáveis sejam de tipos diferentes, real e palavra por exemplo, um operador '>=' não deveria fazer sentido, portanto deveria retornar um erro de tipos diferentes de variáveis.

Os passos principais que utilizamos, foram: Criar o arquivo `MyListener.java` e `AnalizadorSemantico.java`. No arquivo `AnalizadorSemantico` nós lemos o arquivo `codigo.txt`, nele rodamos a `ParseTree` declarada como `ast` e iniciamos o `Parser`. Com o `ParseTreeWalker` percorremos toda a árvore gerada. Então usamos `walker.walk(listener,ast)` para fazer o listener analisar toda a árvore.

No `MyListener` todos os artefatos adicionados que são considerados importantes são sobrescritos utilizando um rótulo, nele, adicionamos as regras semânticas necessárias para garantir as propriedades da Definição Semântica da Linguagem: Especificação das ações semânticas (Checagem de tipo, Checagem de variáveis não declaradas, checagem de declarações duplicadas de variáveis, checagem de escopo de variáveis).

Dificuldades encontradas:

Tentamos utilizar o `Visitor`, mas percorrer a árvore era um desafio e os dados não eram passados adequadamente para a estrutura criada.

Tivemos muita dificuldade em incorporar o `Antlr` como biblioteca externa no projeto. O `VS Code` não estava permitindo que o endereço absoluto do `antlr` fosse definido no `json`, sendo incorporado incorretamente.

Parte desses “erros” se deve ao fato de que os métodos de sobrescrita não estão completos, faltando alguns como `EnterCondicional` etc.

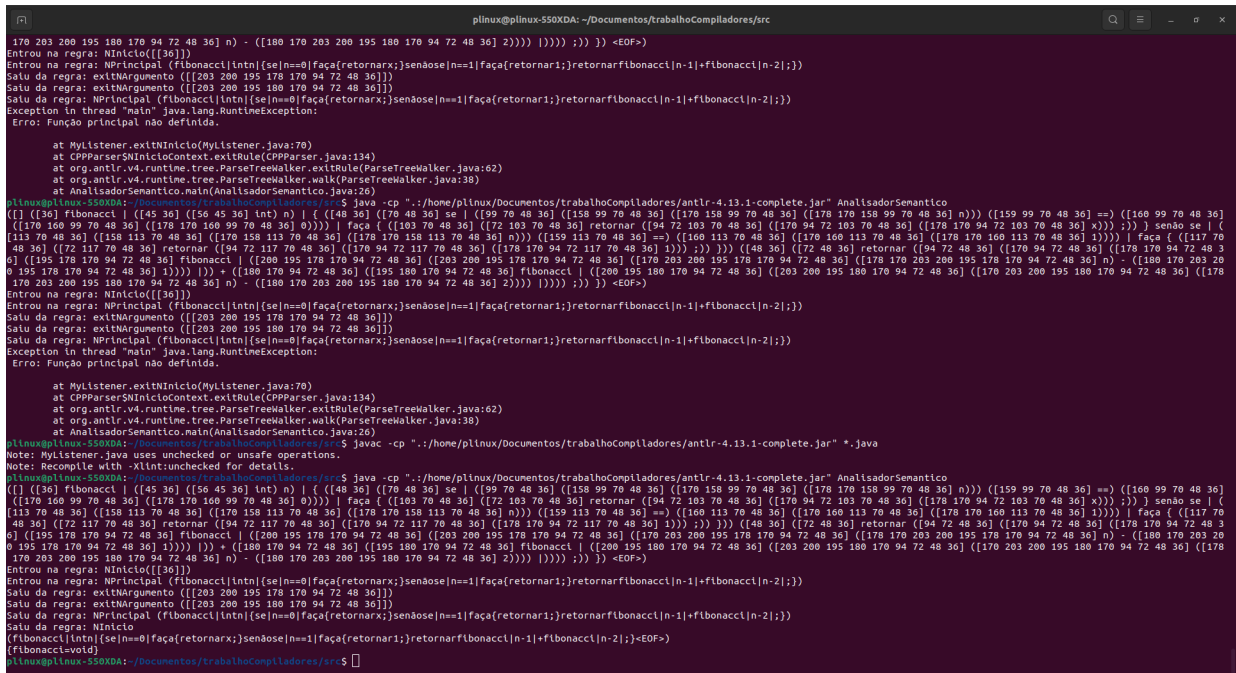
Nesse caso, a comparação de tipos que foi declarada como um método privado não é chamado, portanto, não existe o tratamento da comparação e não é exibido ao rodar o código da `AnáliseSemantica`.

Prints referentes a parte 3, contém o código de teste e a saída do analisador.



The screenshot shows the Visual Studio Code editor with the file 'codigo.txt' open. The code is a Java-like function for calculating Fibonacci numbers. The Explorer sidebar on the left shows a project named 'TRABALHOCompiladores' with a directory structure including 'src' and 'output'. The 'src' directory contains several files, including 'CodigoLexer.class', 'CodigoLexer.interp', 'CodigoLexer.java', and 'CodigoLexer.tokens'. The 'output' directory contains 'AnalizadorSemantico.class', 'AnalizadorSemantico.java', 'antlr-4.13.1-complete.jar', and 'MyListener.java'. The 'codigo.txt' file contains the following code:

```
1 fibonacci | int n | {
2     se | n == 0 | faça {
3         retornar x;
4     } senão se | n == 1 | faça {
5         retornar 1;
6     }
7     retornar fibonacci | n - 1 | + fibonacci | n - 2 |;
8 }
```



The screenshot shows a terminal window with the output of the semantic analyzer. The output is a detailed log of the analysis process, including the input code, the rules of the grammar, and the results of the semantic checks. The output is as follows:

```
170 203 200 195 180 170 94 72 48 36] n - ([180 170 203 200 195 180 170 94 72 48 36] 2))))) )))) ;)) <EOF>
Entrou na regra: NInteiro([36])
Entrou na regra: NPrincipal (fibonacci|intn|se|n==0|faça{retornarx;};senãose|n==1|faça{retornar1;};retornarfibonacci|n-1|+fibonacci|n-2;))
Saiu da regra: extNArgumento ([[203 200 195 178 170 94 72 48 36]])
Saiu da regra: extNArgumento ([[203 200 195 180 170 94 72 48 36]])
Saiu da regra: NPrincipal (fibonacci|intn|se|n==0|faça{retornarx;};senãose|n==1|faça{retornar1;};retornarfibonacci|n-1|+fibonacci|n-2;))
Exception in thread "main" java.lang.RuntimeException:
Erro: Função principal não definida.

at MyListener.extNInteiro(MyListener.java:70)
at CPPParserSNIntContext.extRule(CPPParser.java:134)
at org.antlr.v4.runtime.tree.ParseTreeWalker.walk(ParseTreeWalker.java:62)
at org.antlr.v4.runtime.tree.ParseTreeWalker.walk(ParseTreeWalker.java:38)
at AnalizadorSemantico.main(AnalizadorSemantico.java:26)

plinux@plinux-SS0XDA: ~/Documentos/trabalhoCompiladores/src$ java -cp ".:~/home/plinux/Documentos/trabalhoCompiladores/antlr-4.13.1-complete.jar" AnalizadorSemantico
[[[ [36] fibonacci | [[45 36] [[56 45 36] int] n] | ([48 36] [[70 48 36] se | ([99 70 48 36] [[158 99 70 48 36] [[170 158 99 70 48 36] [[178 170 158 99 70 48 36] n))) [[159 99 70 48 36] ==] [[160 99 70 48 36]
[[170 158 99 70 48 36] [[178 170 158 99 70 48 36] 0))] | faça { ([103 70 48 36] [[12 103 70 48 36] retornar [[94 72 103 70 48 36] [[170 94 72 103 70 48 36] x))] ;)) } senão se | ([
[[13 70 48 36] [[158 113 70 48 36] [[170 158 113 70 48 36] n)) [[159 113 70 48 36] ==] [[160 113 70 48 36] [[170 160 113 70 48 36] [[178 170 160 113 70 48 36] 1))] | faça { ([117 70
48 36] [[172 117 70 48 36] retornar [[94 72 117 70 48 36] [[170 94 72 117 70 48 36] [[178 170 94 72 117 70 48 36] i))] ;)) ;)) [[48 36] [[72 48 36] retornar [[94 72 48 36] [[170 94 72 48 36] [[178 170 94 72 48 3
6] [[195 178 170 94 72 48 36] fibonacci | [[200 195 178 170 94 72 48 36] [[203 200 195 178 170 94 72 48 36] [[178 170 203 200 195 178 170 94 72 48 36] n] - [[180 170 203 20
0 195 178 170 94 72 48 36] i))] ;)) ;)) [[180 170 94 72 48 36] [[195 180 170 94 72 48 36] fibonacci | [[200 195 180 170 94 72 48 36] [[203 200 195 180 170 94 72 48 36] [[178 170 203 200 195 180 170 94 72 48 36] n] - [[180 170 203 20
0 195 178 170 94 72 48 36] n] - ([180 170 203 200 195 180 170 94 72 48 36] 2))))) )))) ;)) <EOF>
Entrou na regra: NInteiro([36])
Entrou na regra: NPrincipal (fibonacci|intn|se|n==0|faça{retornarx;};senãose|n==1|faça{retornar1;};retornarfibonacci|n-1|+fibonacci|n-2;))
Saiu da regra: extNArgumento ([[203 200 195 178 170 94 72 48 36]])
Saiu da regra: extNArgumento ([[203 200 195 180 170 94 72 48 36]])
Saiu da regra: NPrincipal (fibonacci|intn|se|n==0|faça{retornarx;};senãose|n==1|faça{retornar1;};retornarfibonacci|n-1|+fibonacci|n-2;))
Exception in thread "main" java.lang.RuntimeException:
Erro: Função principal não definida.

at MyListener.extNInteiro(MyListener.java:70)
at CPPParserSNIntContext.extRule(CPPParser.java:134)
at org.antlr.v4.runtime.tree.ParseTreeWalker.walk(ParseTreeWalker.java:62)
at org.antlr.v4.runtime.tree.ParseTreeWalker.walk(ParseTreeWalker.java:38)
at AnalizadorSemantico.main(AnalizadorSemantico.java:26)

plinux@plinux-SS0XDA: ~/Documentos/trabalhoCompiladores/src$ javac -cp ".:~/home/plinux/Documentos/trabalhoCompiladores/antlr-4.13.1-complete.jar" *.java
Note: MyListener.java uses unchecked or unsafe operations.
Note: Recompile with -Xlint:unchecked for details.

plinux@plinux-SS0XDA: ~/Documentos/trabalhoCompiladores/src$ java -cp ".:~/home/plinux/Documentos/trabalhoCompiladores/antlr-4.13.1-complete.jar" AnalizadorSemantico
[[[ [36] fibonacci | [[45 36] [[56 45 36] int] n] | ([48 36] [[70 48 36] se | ([99 70 48 36] [[158 99 70 48 36] [[170 158 99 70 48 36] [[178 170 158 99 70 48 36] n))) [[159 99 70 48 36] ==] [[160 99 70 48 36]
[[170 158 99 70 48 36] [[178 170 158 99 70 48 36] 0))] | faça { ([103 70 48 36] [[12 103 70 48 36] retornar [[94 72 103 70 48 36] [[170 94 72 103 70 48 36] x))] ;)) } senão se | ([
[[13 70 48 36] [[158 113 70 48 36] [[170 158 113 70 48 36] n)) [[159 113 70 48 36] ==] [[160 113 70 48 36] [[170 160 113 70 48 36] [[178 170 160 113 70 48 36] 1))] | faça { ([117 70
48 36] [[172 117 70 48 36] retornar [[94 72 117 70 48 36] [[170 94 72 117 70 48 36] [[178 170 94 72 117 70 48 36] i))] ;)) ;)) [[48 36] [[72 48 36] retornar [[94 72 48 36] [[170 94 72 48 36] [[178 170 94 72 48 3
6] [[195 178 170 94 72 48 36] fibonacci | [[200 195 178 170 94 72 48 36] [[203 200 195 178 170 94 72 48 36] [[178 170 203 200 195 178 170 94 72 48 36] n] - [[180 170 203 20
0 195 178 170 94 72 48 36] i))] ;)) ;)) [[180 170 94 72 48 36] [[195 180 170 94 72 48 36] fibonacci | [[200 195 180 170 94 72 48 36] [[203 200 195 180 170 94 72 48 36] [[178 170 203 200 195 180 170 94 72 48 36] n] - [[180 170 203 20
0 195 178 170 94 72 48 36] n] - ([180 170 203 200 195 180 170 94 72 48 36] 2))))) )))) ;)) <EOF>
Entrou na regra: NInteiro([36])
Entrou na regra: NPrincipal (fibonacci|intn|se|n==0|faça{retornarx;};senãose|n==1|faça{retornar1;};retornarfibonacci|n-1|+fibonacci|n-2;))
Saiu da regra: extNArgumento ([[203 200 195 178 170 94 72 48 36]])
Saiu da regra: extNArgumento ([[203 200 195 180 170 94 72 48 36]])
Saiu da regra: NPrincipal (fibonacci|intn|se|n==0|faça{retornarx;};senãose|n==1|faça{retornar1;};retornarfibonacci|n-1|+fibonacci|n-2;))
Exception in thread "main" java.lang.RuntimeException:
Erro: Função principal não definida.

at MyListener.extNInteiro(MyListener.java:70)
at CPPParserSNIntContext.extRule(CPPParser.java:134)
at org.antlr.v4.runtime.tree.ParseTreeWalker.walk(ParseTreeWalker.java:62)
at org.antlr.v4.runtime.tree.ParseTreeWalker.walk(ParseTreeWalker.java:38)
at AnalizadorSemantico.main(AnalizadorSemantico.java:26)

plinux@plinux-SS0XDA: ~/Documentos/trabalhoCompiladores/src$
```