

- [JOLANG](#)
 - [Estrutura](#)
 - [Dependências](#)
 - [Windows](#)
 - [MacOS](#)
 - [Linux](#)
 - [Executando](#)
 - [Explicação](#)
 - [Análise Léxica](#)
 - [Análise Sintática](#)
 - [Análise Semântica](#)
 - [Objetivo e Demo](#)
 - [Equipe](#)

JOLANG

Simple e minimalista linguagem de programação e compilador feito em **Java** com as bibliotecas **JFLEX** e **JCUP**. Projeto final A3 da UC Teoria da Computação e Compiladores da Unisociesc.

Estrutura

Segue abaixo a estrutura de pastas e arquivos do projeto:

- **build**: contém as classes java compiladas;
- **lib**: contém as bibliotecas jflex e jcup .jar;
- **scripts**: contém os scripts .bat para a construção, compilação e execução;
- **src**: contém o código-fonte do compilador;
- **.gitignore**: arquivo que contém o caminho de arquivos e pastas a serem ignorados pelo GitHub;
- **main.jo**: arquivo que contém o código da linguagem de programação a ser compilada;
- **Makefile**: arquivo make para a execução prática dos scripts;
- **README.md**: arquivo contendo informações do projeto.

Dependências

- Java JRE e JDK Oracle ou OpenJDK (**Obrigatório**);
- Make (Opcional)

[Oracle JRE](#)

[Oracle JDK](#)

[OpenJDK](#)

Windows

```
choco install make
```

MacOS

```
brew install make
```

Linux

```
sudo apt install make
```

Executando

Para executar basta executar os scripts da pasta scripts na seguinte ordem:

1. build.bat
2. compile.bat
3. run.bat

Ou utilizando a ferramenta make:

```
make build && make run
```

Explicação

Análise Léxica

O analisador léxico é o responsável por ler o código fonte e retornar os tokens encontrados, é no léxico que se determina quais caracteres, símbolos, letras e números além de suas combinações serão suportadas pela linguagem. Neste projeto a definição de símbolos e tokens é feita no arquivo **lexer.flex**.

Na imagem abaixo é definido os conjuntos a serem utilizados como: letras, números, símbolos especiais, etc. Mais abaixo é definido algumas regras utilizando expressões regulares, nesse caso são definidas as regras para identificação de strings, inteiros e identificadores e na sequência a criação de símbolos depois dos %% como: () {} + / print void etc.

```

1  import java_cup.runtime.*;
2
3  %%
4  %cup
5  %unicode
6  %line
7  %column
8  %public
9  %class Lexer
10
11  %{
12      private ErrorStack errorStack;
13
14      public Lexer(java.io.FileReader in, ErrorStack errorStack) {
15          this(in);
16          this.errorStack = errorStack;
17      }
18
19      public ErrorStack getErrorStack() {
20          return errorStack;
21      }
22
23      public void newError(int line, int column, String text) {
24          this.errorStack.wrap(line, column, text);
25      }
26
27      public void newError(int line, int column) {
28          this.errorStack.wrap(line, column);
29      }
30
31      public void newError(String text) {
32          this.errorStack.wrap(text);
33      }
34
35      private Symbol createSym(int code, Object value) {
36          return new Symbol(code, yyline, yycolumn, value);
37      }
38
39      private Symbol createSym(int code) {
40          return new Symbol(code, yyline, yycolumn);
41      }
42  %}

```

```

44  // sets
45  numbers = [0-9]
46  letters = [A-Za-z]
47  symbols = ["/^$. *+?()[]{}|!#@&_- = ~ ` ' < > " ]
48
49  // rules
50  newLine = \r | \n | \r\n
51  comment = "//".* | "/*"[^*]*["/*"]
52  blanks = [ \t\f ] | {newLine} | {comment}
53
54  integer = ({numbers})+
55  string = \"({letters}|{integer}|{symbols}|{blanks})*\"
56  identifier = {letters} ({letters}|{integer})*
57

```

Conjuntos e Regras

```

59  %%
60  ";" {return createSym(Sym.SEMICOLON);}
61  // " , " {return createSym(Sym.COLON);}
62
63  "+" {return createSym(Sym.PLUS);}
64  "-" {return createSym(Sym.MINUS);}
65  "*" {return createSym(Sym.MULTIPLY);}
66  "/" {return createSym(Sym.DIVIDE);}

```

```

65  "*"          {return createSym(Sym.TIMES);}
66  "/"          {return createSym(Sym.SLASH);}
67
68  "="          {return createSym(Sym.EQUAL);}
69  // "=="      {return createSym(Sym.EQUALTO);}
70  // "<"       {return createSym(Sym.LESS);}
71  // "<="      {return createSym(Sym.LESSEQUAL);}
72  // ">"       {return createSym(Sym.GREATER);}
73  // ">="      {return createSym(Sym.GREATEREQUAL);}
74  // "!="      {return createSym(Sym.NOTEQUAL);}
75
76  "("          {return createSym(Sym.LEFTPAR);}
77  ")"          {return createSym(Sym.RIGHTPAR);}
78  "{"          {return createSym(Sym.LEFTBRACE);}
79  "}"          {return createSym(Sym.RIGHTBRACE);}
80  // "["       {return createSym(Sym.LEFTBRACKET);}
81  // "]"       {return createSym(Sym.RIGHTBRACKET);}
82
83  "string"     {return createSym(Sym.STRVAR);}
84  "int"        {return createSym(Sym.INTVAR);}
85  "void"       {return createSym(Sym.VOID);}
86  "return"     {return createSym(Sym.RETURN);}
87  "print"      {return createSym(Sym.PRINT);}
88
89  {integer}    {
90                int value = Integer.parseInt(yytext());
91                return createSym(Sym.INTEGER, value);
92            }
93
94  {string}     {return createSym(Sym.STRING, yytext());}
95  {identifier} {return createSym(Sym.IDENTIFIER);}
96
97  {blanks}     {}
98  <<EOF>>      {return createSym(Sym.EOF, yytext());}
99  .|\n         {newError(yyline, yycolumn, "Símbolo desconhecido: " + yytext());}
100

```

Tokens/Símbolos

Análise Sintática

O analisador sintático é responsável por determinar quais regras, estruturas e combinações dos tokens gerados na etapa anterior são válidas. Nessa etapa são definidas as regras da linguagem e sua estrutura "gramatical" definindo quais são os termos e "frases", assim como suas combinações, a serem utilizados no código-fonte. Nesse projeto a definição da sintaxe foi feita no arquivo **parser.cup** onde foram definidas estruturas para variáveis, funções, expressões matemáticas e o print como mostrado na imagem abaixo:

```

1  import java_cup.runtime.*;
2  import java.util.Stack;
3
4  parser code
5  {:
6      IFactory factory;
7      Stack<Object> stack = new Stack<Object>();
8
9      public Parser(Scanner scanner, IFactory factory) {
10         this(scanner);
11         this.factory = factory;
12     }
13
14     public void newError(int line, int column, String text) {
15         Lexer lexer = (Lexer) this.getScanner();
16         lexer.newError(line, column, text);
17     }
18
19     public void newError(int line, int column) {
20         Lexer lexer = (Lexer) this.getScanner();
21         lexer.newError(line, column);
22     }
23
24     public void newError(String text) {
25         Lexer lexer = (Lexer) this.getScanner();
26         lexer.newError(text);
27     }
28
29     public void syntax_error(Symbol sym) {
30         this.newError(sym.left, sym.right);
31     }
32 :};
33
34 terminal SEMICOLON;
35 terminal PLUS, MINUS, TIMES, SLASH;
36 terminal EQUAL;
37 terminal LEFTPAR, RIGHTPAR, LEFTBRACE, RIGHTBRACE;
38 terminal STRVAR, INTVAR, VOID, RETURN, PRINT;
39
40 terminal Integer INTEGER;
41 terminal String STRING, IDENTIFIER;
42
43 non terminal Integer expr, term, factor;
44 non terminal Object expr_list, expr_semicolon;
45 non terminal Object var, return, args, func, print;
46
47 expr_list ::= expr_list expr_semicolon
48             | expr_semicolon {:
49                 if (!stack.empty()) {
50                     Object result = stack.pop();
51                     RESULT = result;
52                 }
53             :};
54
55 expr_semicolon ::= var SEMICOLON
56                  | func SEMICOLON
57                  | print SEMICOLON

```

```

58         | error:e {: newError("Sintaxe incorreta!"); :};
59
60     expr ::= expr PLUS term {:
61         IExpression e1 = (IExpression) stack.pop();
62         IExpression e2 = (IExpression) stack.pop();
63         IOperation op = factory.createBinaryOp(Sym.PLUS, e1, e2);
64
65         stack.push(op);
66     :}
67     | expr MINUS term {:
68         IExpression e1 = (IExpression) stack.pop();
69         IExpression e2 = (IExpression) stack.pop();
70         IOperation op = factory.createBinaryOp(Sym.MINUS, e1, e2);
71
72         stack.push(op);
73     :}
74     | term:t;
75
76     term ::= factor TIMES term {:
77         IExpression e1 = (IExpression) stack.pop();
78         IExpression e2 = (IExpression) stack.pop();
79         IOperation op = factory.createBinaryOp(Sym.TIMES, e1, e2);
80
81         stack.push(op);
82     :}
83     | factor SLASH term {:
84         IExpression e1 = (IExpression) stack.pop();
85         IExpression e2 = (IExpression) stack.pop();
86         IOperation op = factory.createBinaryOp(Sym.SLASH, e1, e2);
87
88         stack.push(op);
89     :}
90     | factor;
91
92     factor ::= INTEGER:i {:
93         IExpression value = factory.createInteger(i);
94         stack.push(value);
95     :}
96     | LEFTPAR expr RIGHTPAR;
97
98     var ::= STRVAR IDENTIFIER EQUAL STRING
99         | INTVAR IDENTIFIER EQUAL expr
100         | STRVAR IDENTIFIER
101         | INTVAR IDENTIFIER
102         | error {: newError("Sintaxe da variável incorreta!"); :};
103
104     return ::= RETURN IDENTIFIER SEMICOLON
105             | RETURN STRING SEMICOLON
106             | RETURN INTEGER SEMICOLON;
107
108     args ::= LEFTPAR var RIGHTPAR;
109     func ::= IDENTIFIER args | IDENTIFIER LEFTPAR RIGHTPAR;
110     func ::= VOID func LEFTBRACE expr_list RIGHTBRACE
111         | STRVAR func LEFTBRACE expr_list return RIGHTBRACE
112         | INTVAR func LEFTBRACE expr_list return RIGHTBRACE;
113
114     print ::= PRINT LEFTPAR IDENTIFIER RIGHTPAR
115            | PRINT LEFTPAR STRING:s RIGHTPAR {:
116                IPrint print = factory.createPrint(s, null, null);
117                stack.push(print);
118            :}
119            | PRINT LEFTPAR expr RIGHTPAR {:

```

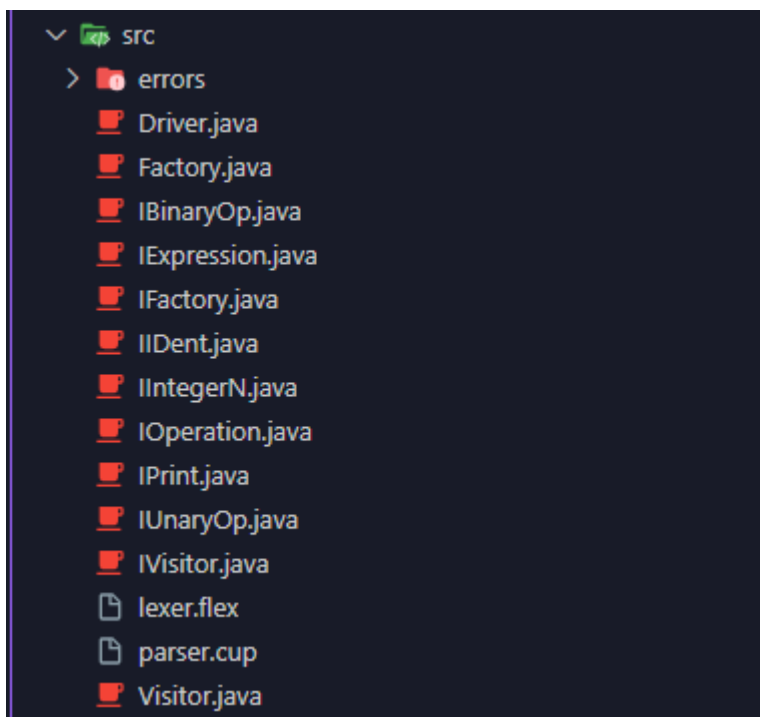
```
120         IExpression e = (IExpression) stack.pop();
121         IPrint print = factory.createPrint(null, e, null);
122         stack.push(print);
123     :};
```

É nessa etapa também que são definidos os erros e uma ponte com a parte semântica onde se dá o sentido a essas expressões e estruturas sintáticas.

Análise Semântica

A análise semântica está intimamente ligada a etapa anterior pois é com base nas estruturas sintáticas definidas e analisadas como válidas é que definimos qual será o "comportamento"/função de cada estrutura gramatical. Nesse projeto a parte semântica foi dividida em classes onde as classes principais são a Factory e a Visitor, ambas trabalham em conjunto para criar instâncias das outras classes e permitir que essas instâncias sejam visitadas e executem funções conforme o necessário.


Na etapa anterior utilizamos a classe factory para criar as instâncias adequadas para as estruturas gramaticais, como por exemplo, as definições expr, term e factor se válidas instanciam as classes responsáveis por realizar os cálculos matemáticos. Segue na imagem abaixo algumas das classes:



Objetivo e Demo

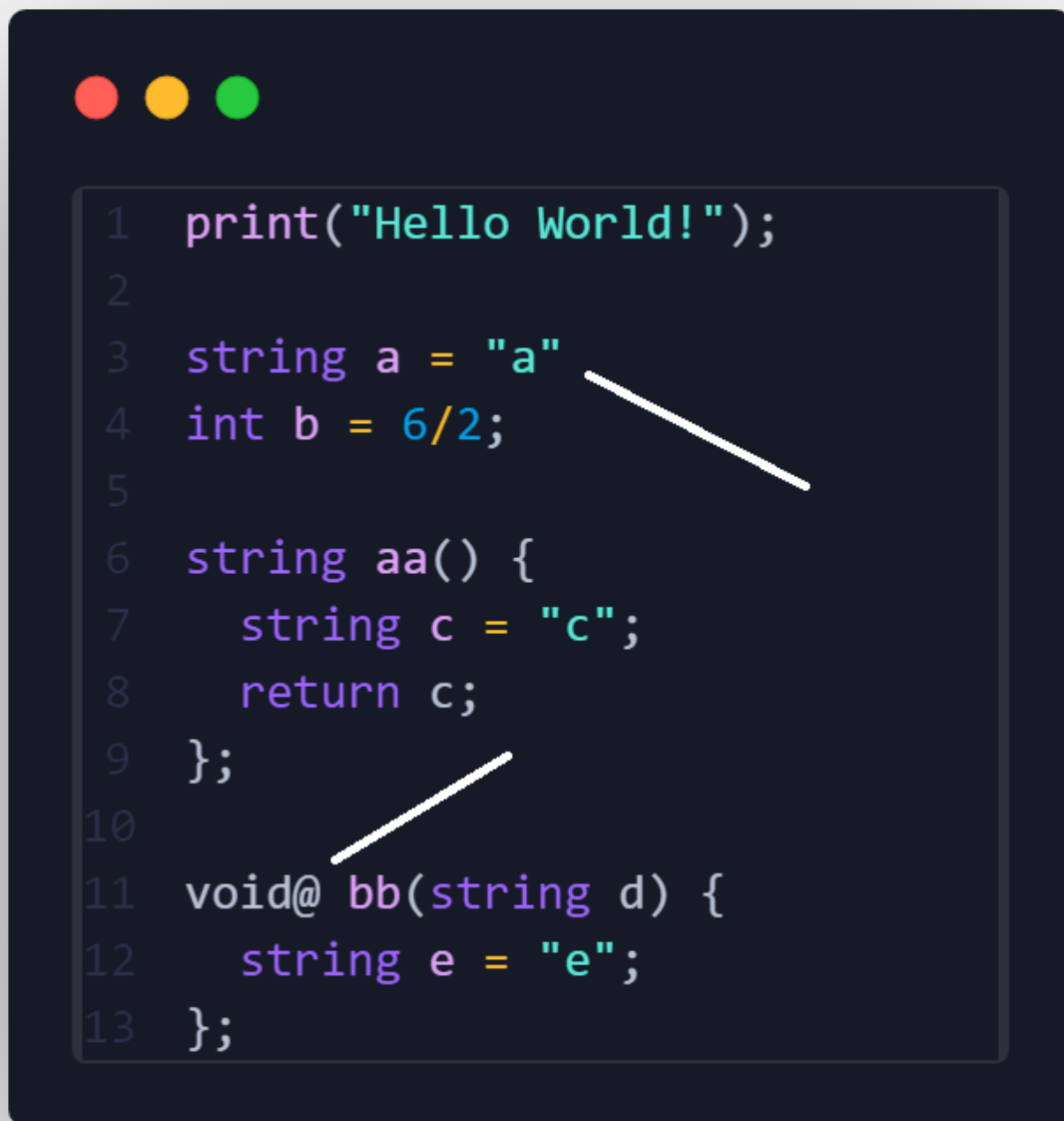
O objetivo principal é validar o código-fonte inserido no arquivo **main.jo** verificando se o mesmo contém os tokens e a sintaxe correta além de executar cálculos matemáticos e exibi-los na tela através da estrutura sintática **print**. Esse projeto possui validação para variáveis, funções e a estrutura print exibindo uma pilha de erros caso os mesmos não sejam válidos. Segue abaixo imagens referentes a estrutura correta do código-fonte e os erros gerados caso alguma linha não corresponda com as regras definidas nas etapas anteriores.

Código-fonte correto:



```
1  print("Hello World!");
2
3  string a = "a";
4  int b = 6/2;
5
6  string aa() {
7      string c = "c";
8      return c;
9  };
10
11 void bb(string d) {
12     string e = "e";
13 };
```

Código-fonte errado:



```
1  print("Hello World!");
2
3  string a = "a"
4  int b = 6/2;
5
6  string aa() {
7      string c = "c";
8      return c;
9  };
10
11 void@ bb(string d) {
12     string e = "e";
13 };
```

Como mostra a imagem acima, esse código-fonte possui dois erros: uma falta de **ponto e vírgula** na linha 03 e um **símbolo especial** na palavra reservada **void** na linha 11. Como resultado temos uma pilha com 02 erros como mostra a imagem abaixo:

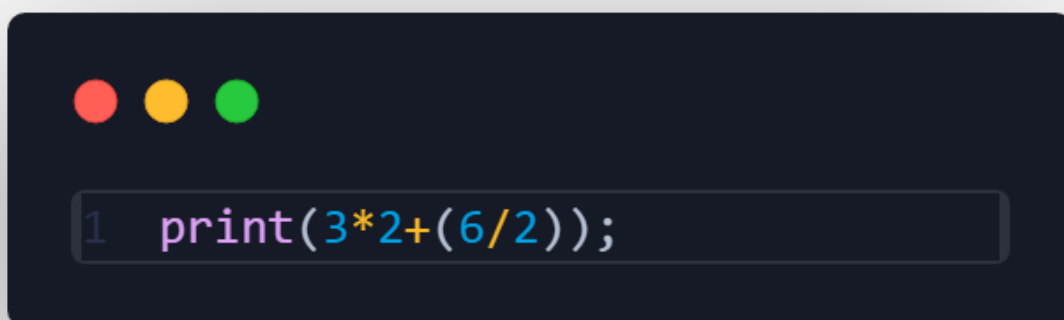
```
→ a3_compiladores git:(jolang) x make run
Running the application

C:\Users\joaol\Desktop\Java\a3_compiladores>java -cp ./build;./lib/java-cup-11b-runtime.jar Driver

Pilha de erros
2 erros encontrados:
-> Linha: 4, Coluna: 0 | Sintaxe incorreta!
-> Linha: 11, Coluna: 4 | SÃ-mbolo desconhecido: @
→ a3_compiladores git:(jolang) x
```

Os erros acima contém linha e coluna aproximada de onde o compilador achou a excessão. Os erros são definidos na Análise Sintática e Semântica caso o código-fonte contenha sintaxes inválidas.

Segue abaixo agra um exemplo do código-fonte rodando com a sintaxe válida:



```
1 print(3*2+(6/2));
```

```
→ a3_compiladores git:(jolang) x make run
Running the application

C:\Users\joaol\Desktop\Java\a3_compiladores>java -cp ./build;./lib/java-cup-11b-runtime.jar Driver
9
→ a3_compiladores git:(jolang) x
```

Equipe

João Lucas Oliveira Sereia - RA122112337

Guilherme Eduardo Corso Maffei - RA122125528

Ciência da Computação, UC Teoria da computação e compiladores - Unisociesc/Anita Garibaldi