UNIVERSIDADE FEDERAL DE SANTA CATARINA
DEPARTAMENTO DE INFORMÁTICA E ESTATÍSTICA
CONSTRUÇÃO DE COMPILADORES

Analisador Léxico

Francisco Luiz Vicenzi
Gustavo Emanuel Kundlatsch
Thiago Sant Helena da Silva

PROFESSOR
Alvaro Junio Pereira Franco

Florianópolis
Fevereiro de 2021

Sumário

1 Introdução

Este relatório apresenta a descrição do trabalho realizado para a construção de um Analisador Léxico para a gramática **CC-2020-2**, descrita em 1.1.

A análise léxica é a primeira etapa do processo de compilação. Nela, o código fonte de entrada é percorrido com o intuito de agrupar os caracteres em *lexemas*. Desse modo, são gerados *tokens* para cada lexema obtido, na forma de $< nome\_token, valor >$. A identificação dos tokens da gramática de estudo está descrita na seção 2, assim como as expressões regulares utilizadas para definí-los.

Diagramas de transição consistem em representações gráficas de como é feita a varredura dos caracteres pelo analisador léxico. Para os lexamas da gramática de estudo, os diagramas estão descritos na seção 3.

Tabelas de símbolos representam esturtura de dados responsáveis por armazenar informações a cerca do código fonte a ser analisado. Em geral, cada entrada possúi seu lexema associado a outras informações que possam ser relevantes, tal como tipo, posição, etc. A descrição da tabela de símbolos implementada é apresentada na seção 4.

A implementação do exercício-programa é apresentado na seção 5. Nela, a ferramenta utilizada é descrita, enfatizando entrada esperada e saídas geradas. Além disso, são apontados e expostos alguns trechos de códigos julgados pertinentes para este relatório. Por fim, apresentamos exemplos de saída do exercício-programa, assim como a descrição dos três programas solicitados.

As referências bibliográficas utilizadas para este relatório foram (AHO et al., 2006), (DELAMARO, 2004) e as vídeo aulas disponibilizadas até então.

1.1 Gramática de estudo: CC-2020-2

$PROGRAM$ → (STATEMENT | FUNCLIST)?
$FUNCLIST$ → FUNCDEF FUNCLIST | FUNCDEF
$FUNCDEF$ → def ident(PARAMLIST) {STATELIST}
$PARAMLIST$ → ((int | float | string) ident, PARAMLIST |
(int | float | string) ident)?
$STATEMENT$ → (VARDECL; |
ATRIBSTAT; |
PRINTSTAT; |
READSTAT; |
RETURNSTAT; |
IFSTAT |
FORSTAT |
STATELIST |
break; |
;)
$VARDECL$ → (int | float | string) ident ([int constant])*
$ATRIBSTAT$ → LVALUE= (EXPRESSION |
ALLOCEXPRESSION | FUNCCALL)
$FUNCCALL$ → ident(PARAMLISTCALL)
$PARAMLISTCALL$ → (ident, PARAMLISTCALL | ident)?
$PRINTSTAT$ → print EXPRESSION
$READSTAT$ → read LVALUE
$RETURNSTAT$ → return
$IFSTAT$ → if(EXPRESSION) STATEMENT
(else STATEMENT)?
$FORSTAT$ → for(ATRIBSTAT; EXPRESSION; ATRIBSTAT)
STATEMENT
$STATELIST$ → STATEMENT(STATELIST)?
$ALLOCEXPRESSION$ → new(int | float | string) ([NUMEXPRESSION])$^+$
$EXPRESSION$ → NUMEXPRESSION((< | > | <= | >= | == |
!=) NUMEXPRESSION)?
$NUMEXPRESSION$ → TERM((+ | -) TERM)*
$TERM$ → UNARYEXPR((*| \|%) UNARYEXPR)*
$UNARYEXPR$ → ((+ | -))? FACTOR
$FACTOR$ → (int constant | float constant | string constant |
null | LVALUE | (NUMEXPRESSION))
$LVALUE$ → ident([NUMEXPRESSION])*

2 Identificação e definição regular dos tokens

A tabela 1 apresenta a identificação e definição dos tokens. Suas declarações, no código, estão contidas na classe *CC20202Lexer*, no arquivo *compiler/lexer/CC20202_lexer.py*.

Note que os primeiros tokens, de def até $float$, estão definidos com a mesma expressão regular na tabela. Definimos assim para refletir a implementação realizada para eles, em função do token $ident$. Maiores detalhes desta implementação e sua motivação estão descritos na seção 5.2, explicando a função t_IDENT.

token	**identificador**	**expressão regular**
DEF	def	
IF	if	
FOR	for	
ELSE	else	
NEW	new	
STRING	string	
BREAK	break	[A-Za-z][A-Za-z0-9_]*
READ	read	
PRINT	print	
RETURN	return	
IDENT	ident	
INT	int	
FLOAT	float	
LBRACKETS	{	{
RBRACKETS	}	}
LPAREN	(	(
RPAREN	)	)
LSQBRACKETS	[	[
RSQBRACKETS	]	]
GREATER_THAN	>	>
LOWER_THAN	<	<
GREATER_OR_EQUALS_THAN	>=	>=
LOWER_OR_EQUALS_THAN	<=	<=
EQ_COMPARISON	==	==
NEQ_COMPARISON	!=	!=
PLUS	+	+
MINUS	-	-
TIMES	*	*
DIVIDE	/	/
MODULE	%	%
SEMICOLON	;	;
COMMA	,	,
NULL	null	null
ATTRIBUTION	=	=
FLOAT_CONSTANT		\d+\.\d+
INT_CONSTANT		\d+
STRING_CONSTANT		".*"

Tabela 1 – Identificação e definição dos tokens

3 Digramas de transição

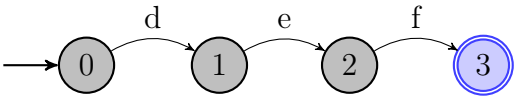

Figura 1 – Token def

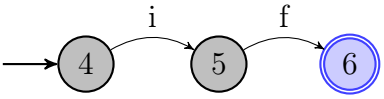

Figura 2 – Token if

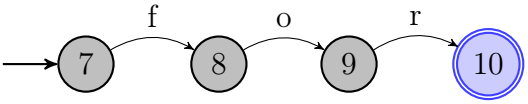

Figura 3 – Token for

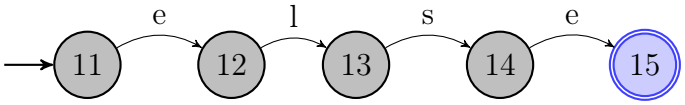

Figura 4 – Token else

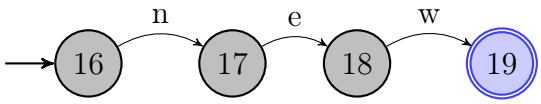

Figura 5 – Token new

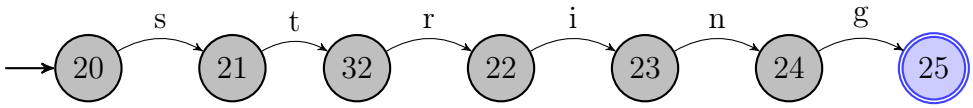

Figura 6 – Token string

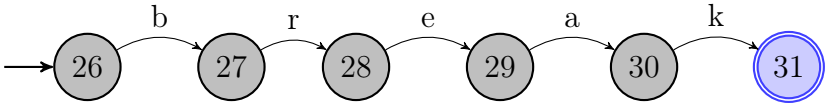

Figura 7 – Token break

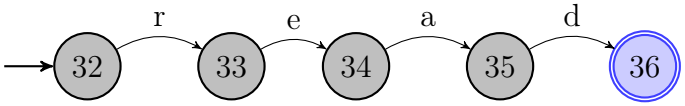

Figura 8 – Token read

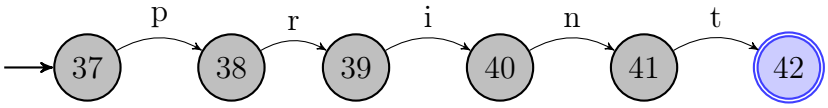

Figura 9 – Token print

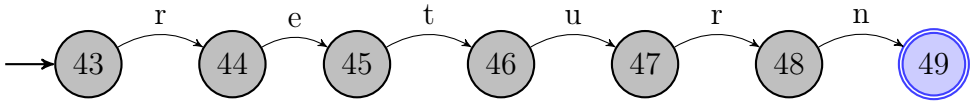

Figura 10 – Token return

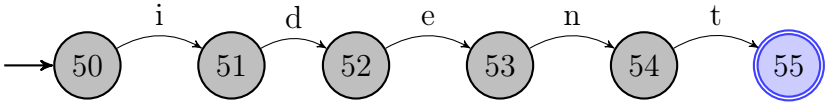

Figura 11 – Token ident

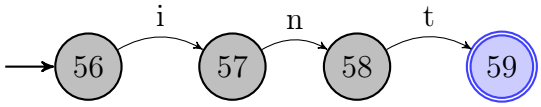

Figura 12 – Token int

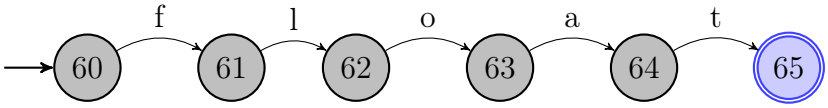

Figura 13 – Token float

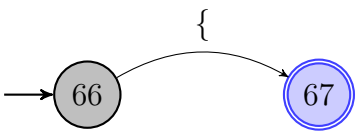

Figura 14 – Token lbrackets

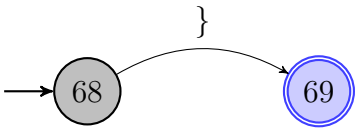

Figura 15 – Token rbrackets

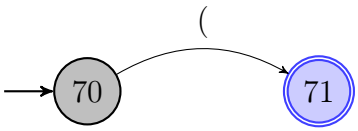

Figura 16 – Token lparen

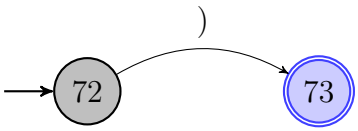

Figura 17 – Token rparen

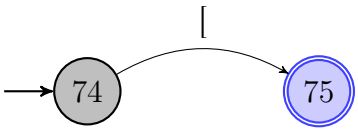

Figura 18 – Token lsqbrackets

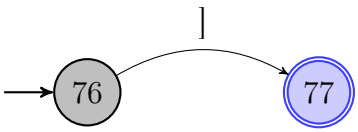

Figura 19 – Token rsqbrackets

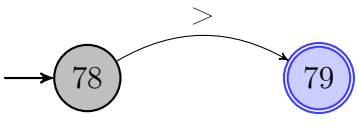

Figura 20 – Token greater_than

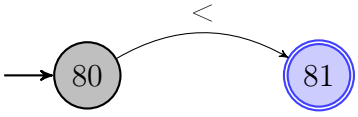

Figura 21 – Token lower_than

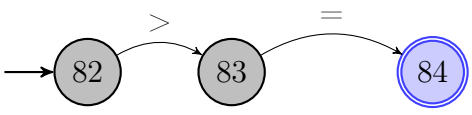

Figura 22 – Token grater_or_equals_than

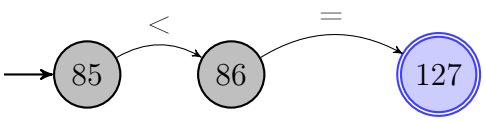

Figura 23 – Token lower_or_equals_than

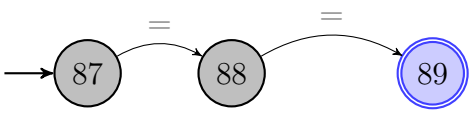

Figura 24 – Token eq_comparision

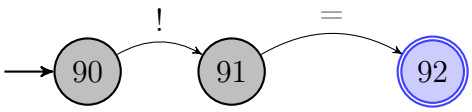

Figura 25 – Token neq_comparision

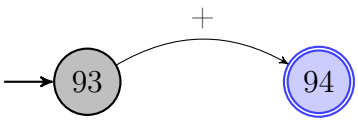

Figura 26 – Token plus

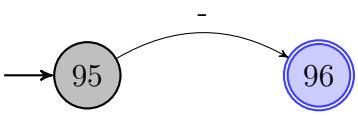

Figura 27 – Token minus

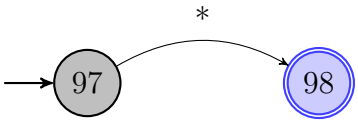

Figura 28 – Token times

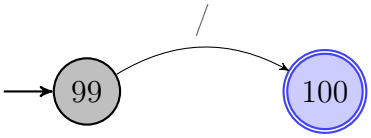

Figura 29 – Token divide

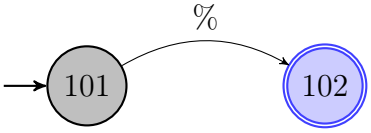

Figura 30 – Token module

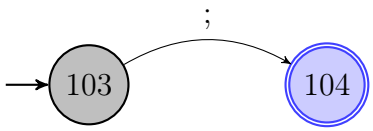

Figura 31 – Token semicolon

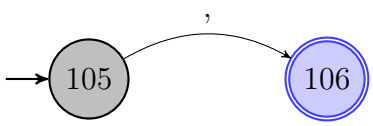

Figura 32 – Token comma

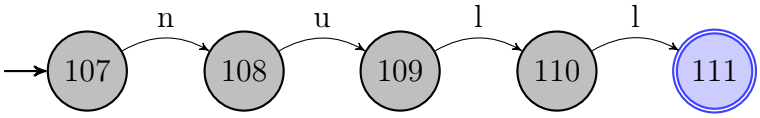

Figura 33 – Token null

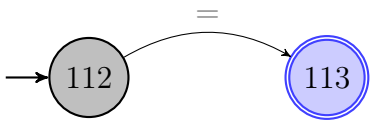

Figura 34 – Token attribution

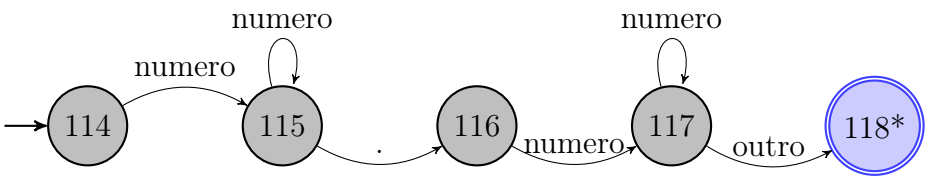

Figura 35 – Token float_const

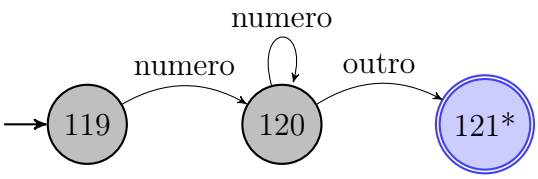

Figura 36 – Token int_const

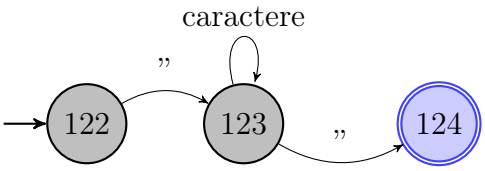

Figura 37 – Token string_const

4 Descrição da tabela de símbolos

A tabela de símbolos é montada a partir da lista de tokens extraídos do código fonte passado pelo analisador léxico. Nela, estão listadas apenas identificadores. Por exemplo, a Tabela 2 exemplifica a tabela de símbolos gerada para o código abaixo. Na Tabela 3, estão especificadas as descrições de cada coluna da tabela de símbolos construída.

```
def func1(int A, int B)
{
    int SM[2];
    SM[0] = A + B;
    SM[1] = B * C;
    return;
}
```

Para este mesmo trecho de código, a lista de tokens gerada é:

```
<DEF, def>, <IDENT, func1>, <LPAREN, (>, <INT_KEYWORD, int>,
<IDENT, A>, <COMMA, ,>, <INT_KEYWORD, int>, <IDENT, B>, <RPAREN, )>,
<LBRACKETS, {>, <INT_KEYWORD, int>, <IDENT, SM>, <LSQBRACKETS, [>,
<INT_CONSTANT, 2>, <RSQBRACKETS, ]>, <SEMICOLON, ;>, <IDENT, SM>,
<LSQBRACKETS, [>, <INT_CONSTANT, 0>, <RSQBRACKETS, ]>, <ATTRIBUTION, =>,
<IDENT, A>, <PLUS, +>, <IDENT, B>, <SEMICOLON, ;>,
<IDENT, SM>, <LSQBRACKETS, [>, <INT_CONSTANT, 1>, <RSQBRACKETS, ]>,
<ATTRIBUTION, =>, <IDENT, B>, <TIMES, *>, <IDENT, C>,
<SEMICOLON, ;>, <RETURN_ST_AT, return>, <SEMICOLON, ;>, <RBRACKETS, }>
```

Indice	**Linha**	**Tipo**	**Lexema**
2	1	IDENT	func1
5	1	IDENT	A
8	1	IDENT	B
12	3	IDENT	SM
14	3	INT_CONSTANT	2
17	4	IDENT	SM
19	4	INT_CONSTANT	0
22	4	IDENT	A
24	4	IDENT	B
26	5	IDENT	SM
28	5	INT_CONSTANT	1
31	5	IDENT	B
33	5	IDENT	C

Tabela 2 – Exemplo de tabela de símbolos

Nome da coluna	**Descrição**
Indice	Posição do lexema na lista de tokens gerada pelo analisador
Linha	Linha onde o lexema foi identificado
Tipo	Tipo do lexema, correspondendo a um símbolo final da gramática
Lexema	O lexema proprieamente dito, extraído do código fonte

Tabela 3 – Metadados da tabela de símbolo

5 Implementação

5.1 Descrição da ferramenta utilizada: Python Lex-Yacc (ply)

Para a criação do analisador, foi utilizada a biblioteca PLY. O *framework* oferecido por esta define que o usuário deve criar uma lista chamada *tokens* no escopo que será analisado inicializado o analisador. Cada item dessa lista, será utilizado pelo analisador como um grupo de tokens a ser identificado.

A seguir, para cada *token*, o usuário deve especificar expressões regulares, por meio de *strings* ou de funções, seguindo o formato `"t_<nome_do_token>"`. A especificação de uma expressão regular por meio de uma função é necessária quando algum tratamento adicional precisa ser feito sobre um *token* específico, nesses casos, a função deve esperar receber um objeto instância da classe `LexToken` e deve ter em sua *docstring* (artifício da linguagem Python utilizado normalmente para documentação de código) a expressão regular que reconhece o token.

Além das expressões para cada *token*, o usuário também deve especificar uma funções para lidar com erros de análise (chamando-a de `t_error`), que recebe um token representando o segmento de texto não identificado, uma função para lidar com caracteres de nova linha (`t_newline`, recebendo também um *token*) e uma string com a lista de caracteres que deve ser ignorado durante a análise, chamando a variável de `t_ignore`.

Uma observação importante sobre o uso da biblioteca, é a atenção quanto a precedência da aplicação das expressões regulares. O analisador gerado aplica primeiro as expressões definidas diretamente como *strings*, ordenando-as pelo comprimento da expressão. Dessa maneira, a expressão para o símbolo `'=='` é aplicada antes da expressão para `'='`, garantindo que não haverão enganos. Depois, o analisador aplica as expressões definidas em funções, utilizando a ordem de definição das funções. Dessa forma, é importante colocar a definição da função `t_FLOAT_CONSTANT` antes de `t_INT_CONSTANT`, evitando que constantes de ponto flutuante sejam identificadas como duas constantes de número inteiro, separados por um ponto, o que provocaria uma falha de análise.

Feitas as configurações, o usuário só precisa instanciar o analisador utilizando a biblioteca, e este acessará o escopo do arquivo (ou o escopo especificado pelo usuário), procurará as variáveis e funções especificadas e criará o analisador léxico propriamente dito.

Na Seção 5.2, demonstrações de como utilizar a biblioteca utilizando como exemplo a implementação feita pelo grupo.

5.2 Estrutura do Código

A estrutura do código como um todo está organizado em uma hierarquia de pastas demonstrada abaixo, com a omissão de arquivos de organização de pastas da linguagem utilizada:

```
compiler/
    lexer/
        CC20202_lexer.py
    exceptions.py
    symbol_table.py
examples/
    exemplo1.ccc
    ...outros exemplos de código válido...
run.py
pyporject.toml
poetry.lock
requirements.txt
README.md
LICENSE
```

O analisador está definido no arquivo `compiler/lexer/CC20202_lexer.py`, o processo de criação da tabela de símbolos a partir da lista de *tokens* está em `compiler/symbol_table.py`. O arquivo `run.py` é responsável por receber o caminho para o arquivo a ser analisado passado pelo usuário através do `Makefile` e executar a análise do código fonte, utilizando as funções e objetos definidos nos demais arquivos. O analisador executa apenas até o primeiro erro léxico, ou até o final do arquivo fonte, caso não hajam erros.

O analisador léxico gerador pela biblioteca PLY foi encapsulada em uma classe, visando melhor organização do código. A lista de tokens foi especificada como demosntrado abaixo:

```
reserved_keywords = {
    'def': 'DEF',
    'if': 'IF',
    'for': 'FOR',
    'else': 'ELSE',
    'new': 'NEW',
    'int': 'INT_KEYWORD',
    'float': 'FLOAT_KEYWORD',
    'string': 'STRING_KEYWORD',
    'break': 'BREAK',
```

```
    'read': 'READ_AT',
    'print': 'PRINT_AT',
    'return': 'RETURN_ST_AT',
}

tokens = [
    *reserved_keywords.values(),
    'LBRACKETS', 'RBRACKETS', 'LPAREN', 'RPAREN',
    'LSQBRACKETS', 'RSQBRACKETS', 'GREATER_THAN', 'LOWER_THAN',
    'GREATER_OR_EQUALS_THAN', 'LOWER_OR_EQUALS_THAN', 'EQ_COMPARISON',
    'NEQ_COMPARISON', 'PLUS', 'MINUS', 'TIMES', 'DIVIDE', 'MODULE',
    'SEMICOLON', 'COMMA', 'NULL', 'ATTRIBUTION', 'IDENT', 'FLOAT_CONSTANT',
    'INT_CONSTANT','STRING_CONSTAT']
```

Para a definição das expressões regulares, foram utilizadas *strings* para os *tokens* mais simples, como demonstrado abaixo:

```
t_LBRACKETS = r'{'
t_RBRACKETS = r'}'
t_GREATER_OR_EQUALS_THAN = r'>='
t_LOWER_OR_EQUALS_THAN = r'<='
t_EQ_COMPARISON = r'=='
t_NEQ_COMPARISON = r'!='
t_NULL = r'null'
t_STRING_CONSTAT = r'".*"'
```

Foi utilizada uma função para identificar o *token* `IDENT`, por conta de palavras reservadas poderem ser classificadas como identificadores. Assim, como pode ser observado abaixo, a função `t_IDENT` primeiro verifica se o token identificado está como uma chave no dicionário de palavras reservadas, e se estiver, torna o tipo do `token` identificado para o tipo correspondente. Caso contrário, mantém o tipo inicial, configurado como valor padrão do acesso ao dicionário. No mesmo trecho de código, estão as funções para identificação dos *tokens* `INT_CONSTANT` e `FLOAT_CONSTANT`, que converte do valor do token para os tipos primitivos `int` e `float`, respectivamente, da linguagem Python.

```
def t_IDENT(self, t):
    r'[A-Za-z][A-Za-z0-9_]*'
    # Check identifier word is not a reserved keyword
    t.type = self.reserved_keywords.get(t.value, 'IDENT')
    return t
```

```
def t_FLOAT_CONSTANT(self, t):
    r'\d+\.\d+'
    t.value = float(t.value)
    return t

def t_INT_CONSTANT(self, t):
    r'\d+'
    t.value = int(t.value)
    return t
```

As funções abaixo demonstram a forma de se lidar com caracteres de quebra de linha, que incrementa a linha atual do analisador e a função que lidar com erros encontrados. A manipulação de erros provoca uma exceção no código, que é tratada pelo arquivo `run.py`. A variável *t_ ignore* especifica que os caracteres de espaço e tabulação como caracteres ignorados pelo analisador.

```
def t_newline(self, t):
    r'\n+'
    t.lexer.lineno += len(t.value)

def t_error(self, t):
    raise InvalidTokenError("Illegal character '%s' at line %s" %
                            (t.value[0], t.lexer.lineno))

t_ignore = ' \t'
```

5.3 Exemplo de saída

Após o setup da aplicação com `make setup`, a execução do comando `make run` vai executar o analisador para o arquivo examples/exemplo1.ccc, obtendo a saída abaixo. Partes do texto foi omitido para melhor organização deste documento.

```
Executing...
2021-02-23 13:49:28.743 | INFO     | __main__:main:30 - Total tokens: 160
2021-02-23 13:49:28.744 | INFO     | __main__:main:31 - Lista de tokens:
<LBRACKETS, {>
<LBRACKETS, {>
<FLOAT_KEYWORD, float>
<IDENT, x>
<SEMICOLON, ;>
```

```
<FLOAT_KEYWORD, float>
[...]
<RBRACKETS, }>
2021-02-23 13:49:28.745 | INFO     | __main__:main:36 -
Imprimindo tabela de símbolos...
Indice          Linha           Tipo            Lexema
4               3               IDENT           x
7               4               IDENT           z
10              5               IDENT           i
13              6               IDENT           max
15              7               IDENT           x
[...]
41              10              IDENT           x
43              11              IDENT           x
45              11              IDENT           x
47              11              FLOAT_CONSTANT  0.001
[...]
152             36              IDENT           y
154             36              IDENT           i
```

5.4 Descrição dos programas escritos

Os três programas seguindo os padrões solicitados estão localizados na pasta *examples*, junto com os exemplos disponibilizados pelo professor.

O programa `example/bhaskara.ccc` apresenta duas funções, além da `main`: `bhaskara` e `calculate_delta`. A função `calculate_delta` recebe como parâmetros três variáveis de ponto flutuante e calcula o delta, mostrando na tela o seu resultado, também. A função `bhaskara` resolve a equação quadrática, apresentando erro se o primeiro parâmetro for 0 e apresentando na tela os resultados. A função `main` chama as funções repetidamente, mas com argumentos diferentes em todas elas, com intuito de testar os casos possíveis.

O programa `example/math.ccc` visa representar o que seria a implementação de uma biblioteca de ferramentas matemáticas, dadas as restrições da linguagem. A função `gcd` é o cálculo do maior divisor comum entre dois números iterativamente, usando o algoritmo de Euclides, `is_prime` aplica um algoritmo de identificação de números primos, `pow` calcula o primeiro argumento elevado a potência do segundo, `factorial` calcula o fatorial de um número. A função `main` executa algumas chamadas sobre as funções previamente definidas, para fins de demonstração.

O programa `example/geometry.ccc` apresenta algumas funções geométricas. Para triângulos, a função `triangle_area` que calcula sua área dada a base e a altura e a função

`form_triangle` que, dado o comprimento de três segmentos de reta, calcula se é possível formar um triângulo com eles. Para círculos, a função `calc_circle_circumference` calcula a circunferência, e a função `calc_circle_area` a área. Ambas recebem como entrada o raio do círculo em ponto flutuante. Por fim, a função `calc_square_area` calcula a área de um quadrado dado o comprimento do lado. A função `main` chama cada uma das funções geométricas para demonstrar seu funcionamento.

Referências

AHO, A. V. et al. *Compilers: Principles, Techniques, and Tools (2nd Edition)*. USA: Addison-Wesley Longman Publishing Co., Inc., 2006. ISBN 0321486811.

DELAMARO, M. *Como Construir um Compilador Utilizando Ferramentas Java*. Novatec, 2004. ISBN 9788575220559. Disponível em: <https://books.google.com.br/books?id=_MpSXwAACAAJ>.