

Universidade Federal de Santa Catarina

Departamento de Informática e Estatística
INE5426: Construção de Compiladores

Relatório do Exercício Programa 1: Analisador Léxico

Grupo:

Artur Barichello (16200636)

Lucas Verdade (17104409)

Lucas Zacchi (16104597)

julho
2021

Sumário

1	Introdução	1
2	Gramática CC-2021-1	2
3	Implementação	3
4	Identificação de Tokens	5
5	Diagramas de Transição	6
6	Entradas e saídas	13

1 Introdução

A atividade deste Exercício Programa consistiu em implementar um analisador léxico para a linguagem CC-2021-1 definida no enunciado e descrita em 2. Para desenvolver essa atividade foi utilizada a linguagem de programação *Python* e também a biblioteca *ply*[3].

O capítulo 2 apresenta a linguagem proposta pelo enunciado da atividade com suas descrições no formato BNF disponível no livro de *Delamaro*[1]. Alguns trechos de código estão descritos no capítulo 3 onde é mostrada a estrutura da biblioteca *ply* e como ela é utilizada para reconhecer os tokens e palavras reservadas. Os capítulos 4 e 5 mostram respectivamente a tabela de identificação dos tokens e os diagramas de transição que foram criados para representar as expressões regulares dos lexemas. O último capítulo mostra um exemplo de entradas e saídas e a resposta do programa conforme requisitados pelo enunciado.

A gramática fornecida foi levemente modificada para se assemelhar com a linguagem *Lua*[2] criada por Roberto Ierusalimschy, Luiz Henrique de Figueiredo, e Waldemar Celes através do grupo Tecgraf da PUC-Rio. Para auxiliar na programação e diferenciar dos arquivos da linguagem original que foram cedidas as extensões dos arquivos de exemplo também foram modificadas para *.lua*.

2 Gramática CC-2021-1

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

3 Implementação

A implementação do analisador léxico foi feita na linguagem *Python* versão 3.6.9 e utilizou como ferramenta auxiliar a biblioteca *ply*[3], especificamente o módulo *ply.lex* que permite separar e classificar arquivos de entrada em uma coleção de tokens, com base em regras descritas em expressões regulares.

Inicialmente foi definido um conjunto de palavras reservadas da gramática, a serem usadas pelos programas escritos, e a partir desse conjunto foi definida uma lista de tokens, que é utilizada para identificação pelo analisador.

As duas implementações estão descritas a seguir:

```
reserved = {
    "def": "DEF",
    "int": "INTEGER",
    "float": "FLOATING_POINT",
    "string": "STRING",
    "break": "BREAK",
    "print": "PRINT",
    "read": "READ",
    "return": "RETURN",
    "if": "IF",
    "else": "ELSE",
    "for": "FOR",
    "new": "NEW",
}

tokens = list(reserved.values()) + [
    "GREATER_OR_EQUAL_THAN",
    "EQUAL",
    "NOT_EQUAL",
    "PLUS",
    "MINUS",
    "DIVISION",
    "COMMA",
    "SEMICOLON",
    "LEFT_BRACKET",
    "LEFT_PARENTHESIS",
    "LEFT_SQUARE_BRACKET",
    "NULL",
    "ATtribution",
    "STRING_CONSTANT",
    "FLOATING_POINT_CONSTANT",
    "INTEGER_CONSTANT",
]
```

Código 1: Dicionário de palavras reservadas

Código 2: Trecho da lista de tokens

Os operadores que foram modificados para se inspirar na linguagem *Lua*[2] estão identificados com um comentário '*# updated*', o símbolo de desigualdade por exemplo é *~=* ao invés do mais comum '*!=*', e o símbolo para *NULL* foi alterado para *nil*.

Expressões regulares simples são definidas através de uma variável prefixada com `t_` conforme especificado nas documentações da biblioteca utilizada. Regras mais complexas como por exemplo os identificadores que não podem incluir palavras reservadas são definidas através de uma função com o mesmo prefixo, porém neste caso a expressão regular é incluída na docstring na primeira linha da função.

```
def t_LABEL(self, t: LexToken) -> LexToken:
    r"[a-zA-Z][A-Za-z0-9_]*"
    t.type = self.reserved.get(t.value, "LABEL")
    return t
```

Código 3: Trecho da lista de tokens

Outras funções especiais da ply também foram utilizadas para casos extras de tokenização, regras prefixadas por `t_ignore` são utilizadas para descartar tokens inúteis para o próximo passo de análise semântica (trabalho 2). Tais tokens incluem o caractere de tabulação e caracteres de comentários que possuem apenas a função de informar o programador.

4 Identificação de Tokens

Token	identificador	Expressão Regular
DEF	def	"[A-Za-z][A-Za-z0-9_]*"
IF	if	
FOR	for	
ELSE	else	
NEW	new	
STRING	string	
BREAK	break	
READ	read	
PRINT	print	
RETURN	return	
LABEL	ident	
INT	int	
FLOAT	float	
GREATER_THAN	>	">"
LESSER_THAN	<	"<"
GREATER_OR_EQUAL_THAN	>=	">="
LESSER_OR_EQUAL_THAN	<=	"<="
EQUAL	==	"=="
NOT_EQUAL	~=	"~="
PLUS	+	"+"
MINUS	-	"-"
TIMES	*	"*"
DIVISION	/	"/"
MODULO	%	"%"
COMMA	,	","
SEMICOLON	;	";"
LEFT_BRACKET	{	"{"
RIGHT_BRACKET	}	"}"
LEFT_PARENTHESIS	("("
RIGHT_PARENTHESIS)	")"
LEFT_SQUARE_BRACKET	["["
RIGHT_SQUARE_BRACKET]	"]"
NULL	nil	"nil"
COMMENT	-	--.*"
IGNORE		"\t"
STRING_CONSTANT		".*"
ATtribution	=	"="
FLOAT_CONSTANT		"\d+\.\d+"
INT_CONSTANT		"\d+"

5 Diagramas de Transição

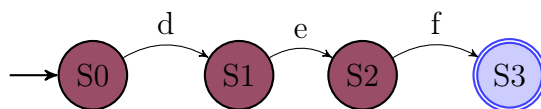


Figura 1: Token def

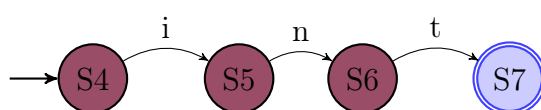


Figura 2: Token int

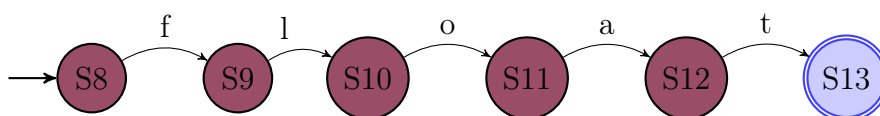


Figura 3: Token float

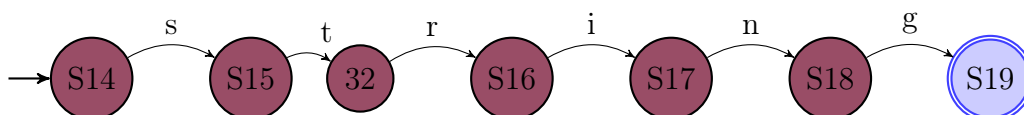


Figura 4: Token string

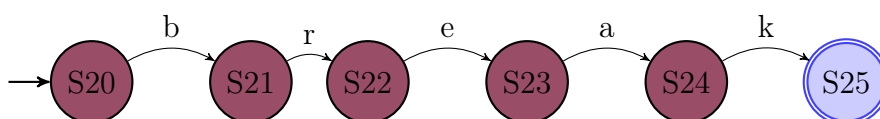


Figura 5: Token break

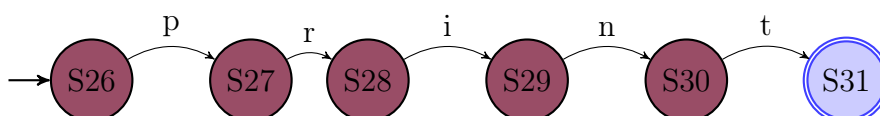


Figura 6: Token print

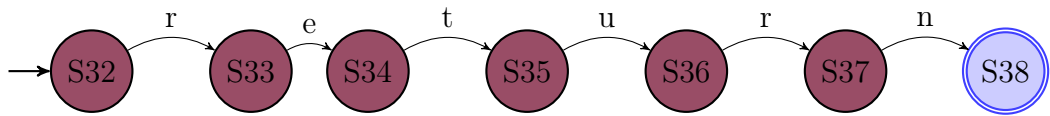


Figura 7: Token return

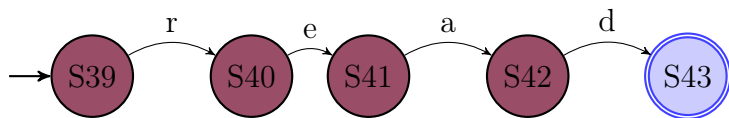


Figura 8: Token read

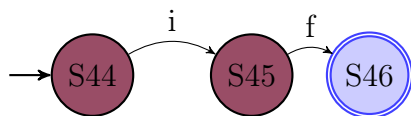


Figura 9: Token if

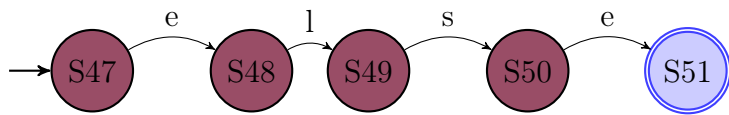


Figura 10: Token else

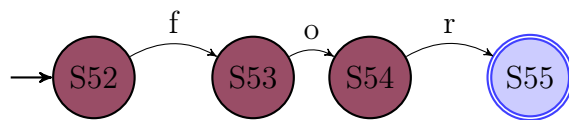


Figura 11: Token for

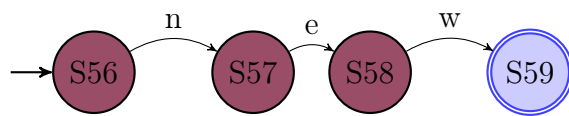


Figura 12: Token new

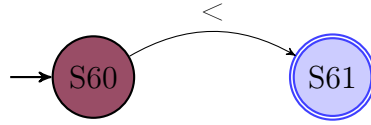


Figura 13: Token lesser_than

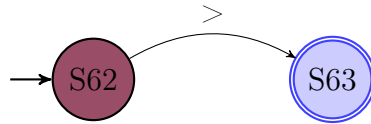


Figura 14: Token greater_than

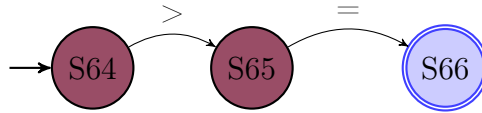


Figura 15: Token grater_or_equal_than

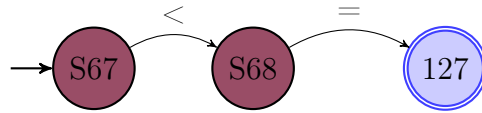


Figura 16: Token lesser_or_equal_than

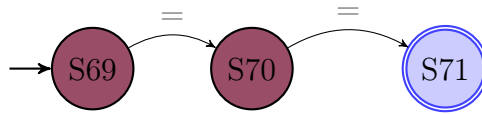


Figura 17: Token equal

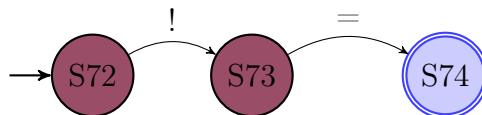


Figura 18: Token not_equal

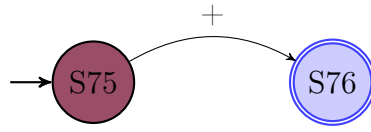


Figura 19: Token plus

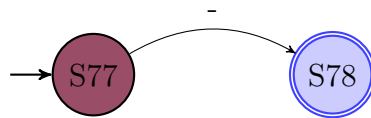


Figura 20: Token minus

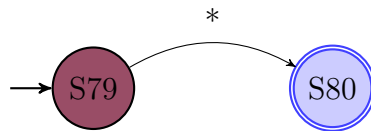


Figura 21: Token times

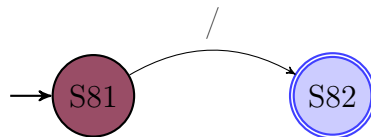


Figura 22: Token division

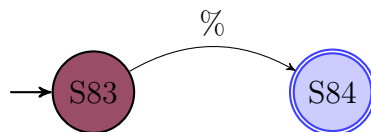


Figura 23: Token modulo

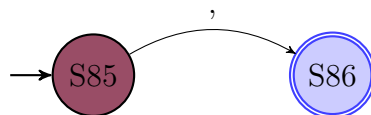


Figura 24: Token comma

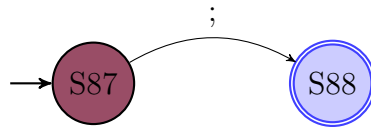


Figura 25: Token semicolon

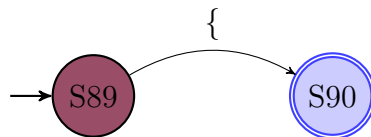


Figura 26: Token left_bracket

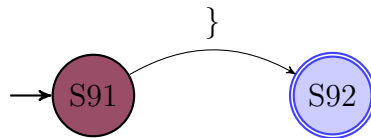


Figura 27: Token right_bracket

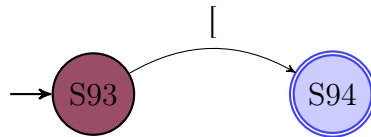


Figura 28: Token left_square_bracket

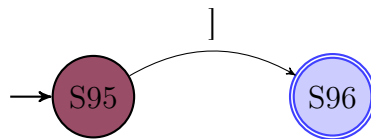


Figura 29: Token right_square_bracket

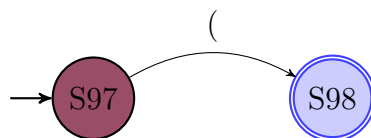


Figura 30: Token left_parenthesis

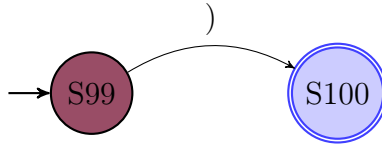


Figura 31: Token right_parenthesis

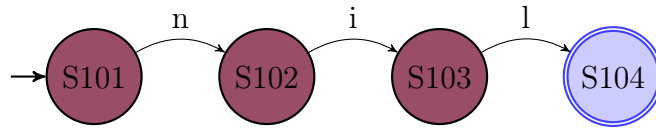


Figura 32: Token nil

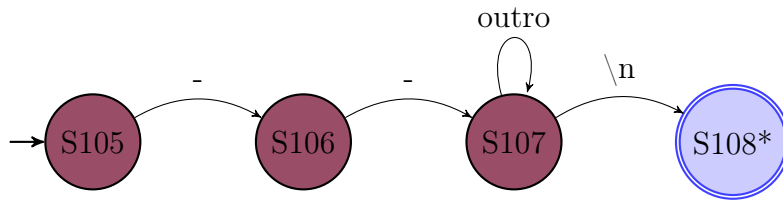


Figura 33: Token comment

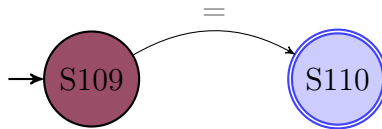


Figura 34: Token attribution

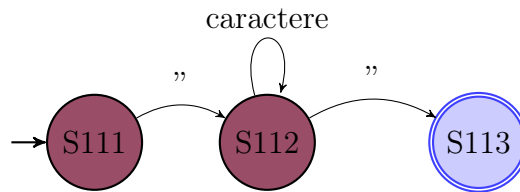


Figura 35: Token string_constant

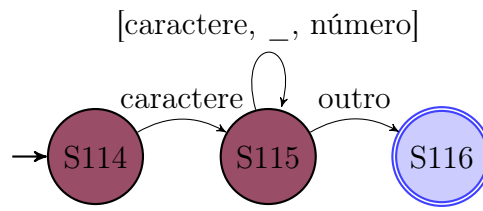


Figura 36: Token label

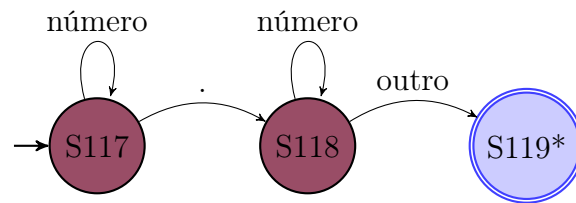


Figura 37: Token floating_point_constant

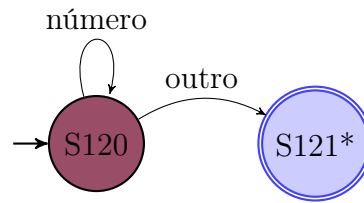


Figura 38: Token integer_constant

6 Entradas e saídas

```
Printing token list: ('Token enumerator', 'Token value'):
[('LABEL', 'max'), ('ATtribution', '='), ('INTEGER_CONSTANT', 10),
 ('SEMICOLON', ';'), ('LABEL', 'i'), ('ATtribution', '='),
 ('LABEL', 'random_int'), ('LEFT_PARENTHESIS', '('), ('RIGHT_PARENTHESIS', ')'),
 ('SEMICOLON', ';'), ('IF', 'if'), ('LEFT_PARENTHESIS', '('),
 ('LABEL', 'i'), ('NOT_EQUAL', '~='), ('INTEGER_CONSTANT', 2), ('EQUAL', '=='),
 ('INTEGER_CONSTANT', 0), ('RIGHT_PARENTHESIS', ')'), ('LEFT_BRACKET', '{'),
 ('LABEL', 'max'), ('ATtribution', '='), ('INTEGER_CONSTANT', 25),
 ('SEMICOLON', ';'), ('RIGHT_BRACKET', '}')]
```

Printing symbol table:

Index	Line	Type	Value
16	2	LABEL	max
20	2	ATtribution	=
22	2	INTEGER_CONSTANT	10
24	2	SEMICOLON	;
26	3	LABEL	i
28	3	ATtribution	=
30	3	LABEL	random_int
40	3	LEFT_PARENTHESIS	(
41	3	RIGHT_PARENTHESIS)
42	3	SEMICOLON	;
44	4	IF	if
47	4	LEFT_PARENTHESIS	(
48	4	LABEL	i
50	4	NOT_EQUAL	~=
53	4	INTEGER_CONSTANT	2
55	4	EQUAL	==
58	4	INTEGER_CONSTANT	0
59	4	RIGHT_PARENTHESIS)
61	4	LEFT_BRACKET	{
67	5	LABEL	max
71	5	ATtribution	=
73	5	INTEGER_CONSTANT	25
75	5	SEMICOLON	;
77	6	RIGHT_BRACKET	}

A lista de tokens e tabela de símbolos acima foram geradas a partir da seguinte entrada:

```
-- Success case
max = 10;
i = random_int();
if (i ~= 2 == 0) {
    max = 25;
}
```

O trecho acima foi retirado de um dos exemplos incluídos no trabalho através do arquivo `./source_code/success.lua`. Estão presentes na saída as colunas representando o índice do token, a linha em que ele foi encontrado, seu tipo segundo a gramática e o valor reconhecido.

No caso da entrada possuir um símbolo que não está definido na gramática o programa pula este símbolo e aborta a sua execução com uma mensagem de erro indicando o caracter desconhecido e a sua posição no formato [linha:coluna] de maneira similar como é feito em Lua.

Tal comportamento pode ser observado ao rodar o arquivo `./source_code/error.lua` que computa o seguinte resultado:

```
def invalid_char_function(int A, int B) {
    int invalidchar[2];
    return;
}
```

Input file(2:18) '!' is an invalid character

Também foram implementados 3 arquivos (*program1.lua*, *program2.lua*, *program3.lua*) conforme especificado pelo enunciado. O primeiro algoritmo simula um sistema de notas de uma turma de alunos com as notas das provas geradas de acordo com a sua posição. O programa calcula as médias de cada aluno e salva numa lista de notas finais. Também é executada uma função que imprime o nome de cada aluno, suas notas, sua média final e diz se o aluno foi aprovado ou não.

O segundo programa (*program2.lua*) simula um sistema de controle de estoque de produtos genéricos. Cada produto tem nome, descrição e preço. É possível adicionar novos produtos, atualizar os dados de um produto e deletar um produto. Também trata erros para não permitir a adição de um produto

quando o estoque estiver cheio e avisa quando o produto solicitado não existe. Ao final do programa a função *main()* é chamada e nela são feitas algumas chamadas das funções, definidas anteriormente. São adicionados 5 produtos diferentes seguido de uma tentativa de adicionar outro produto com o estoque cheio, logo após um produto é apagado e outro é atualizado.

O terceiro programa (*program3.lua*) simula um sistema de reserva de assentos de um cinema/teatro. Nele o usuário pode definir o nome do seu estabelecimento, que será exibido na tela de inicialização, e também o tamanho máximo de filas de assentos. Também pode consultar o status de quantos assentos estão ocupados, quantos estão livres e qual o total de assentos. Foram definidas funções para lidar com a reserva de assentos e também para liberar uma posição.

A função de reservar verifica se a posição escolhida é válida e se está livre, caso esteja então coloca o id do usuário na posição do assento. Caso não esteja livre o programa informa que está ocupado e então faz uma sugestão de assentos vizinhos livres. A função *main* inicializa um cinema novo com 25 assentos (5 filas e 5 assentos) com o nome de: 'Praia de Belas Cinema'. Então, imprime o status do cinema e tenta fazer duas reservas, a segunda reserva é apenas para demonstrar que não é possível reservar um lugar já ocupado.

Referências Bibliográficas

Referências

- [1] *Como Construir um Compilador Utilizando Ferramentas Java*. URL: <https://sites.icmc.usp.br/delamaro/SlidesCompiladores/CompiladoresFinal.pdf>. Márcio Delamaro.
- [2] *Lua language*. URL: <https://www.lua.org/>. PUC-Rio.
- [3] *Ply (Python Lex-Yacc)*. URL: <https://ply.readthedocs.io/en/latest/ply.html>. (Acessado em: 03/07/2021).