UNIVERSIDADE FEDERAL DE SANTA CATARINA
DEPARTAMENTO DE INFORMÁTICA E ESTATÍSTICA
CURSO DE CIÊNCIAS DA COMPUTAÇÃO

Trabalho 1 - Analisador Léxico

Letícia do Nascimento (16104595)

Ramna Sidharta (16100742)

Matheus Schaly (18200436)

Este trabalho tem como objetivo implementar e descrever os passos da criação de um analisador léxico para uma linguagem AL.

Análise Léxica

A análise léxica, ou linear, é a primeira etapa de um processo de compilação. Podemos defini-la como sendo um processo de conversão de uma entrada, sendo ela uma sequência de caracteres, em uma sequência de *tokens*, que são conjuntos de caracteres (strings) com significados. Em outras palavras, nesta etapa do processo de compilação, o código fonte (entrada para o programa compilador) é agrupado em elementos específicos (token). Cada elemento possui um lexema (a sequência literal lida do código fonte) e um tipo. Além destes dois dados, um token pode conter mais informações que facilitem a etapa seguinte, como sua posição no arquivo de entrada. Nessa etapa também é iniciada a construção de uma tabela de símbolos - uma estrutura de dados utilizada para fazer o armazenamento de informações dos identificadores.

Processo de Implementação

Ferramentas Utilizadas

O analisador léxico, o qual chamamos de lexer, foi implementado na linguagem Python, fazendo uso da biblioteca PLY, que é uma implementação das ferramentas lex e yacc, originárias do Unix. Nesta etapa utilizou-se apenas o componente lex, que simplifica o trabalho de implementação de um analisador léxico.

Lex

Identificação de Tokens

O Lex nos permite focar nas **definições regulares** dos tokens, que são determinadas através de expressões regulares, eliminando a necessidade de escrever lógica para de fato percorrer o programa de entrada. Com o lex há duas formas de definir um token:

1. por uma declaração simples de variável, a qual deve ter seu nome iniciando em `t_`, seguido pelo tipo do token, e ter um valor de expressão regular compatível com a biblioteca "re" nativa do Python. Por exemplo, `t_ASSIGN = r'='` define que o token do tipo ASSIGN (que significa atribuição de um valor à uma variável) é representado pelo lexema "=".
2. através de uma função, a qual deve ter seu nome iniciado em `t_`, seguido pelo tipo do token, e receber um parâmetro, do tipo LexToken, que é a representação de um token na biblioteca lex. Esta forma de definir tokens é preferível, sobre a anterior, quando deseja-se executar alguma operação além de simplesmente definir uma expressão regular. Por exemplo, para definir um literal inteiro, pode-se utilizar a seguinte função:

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

Neste caso, a "operação extra" é converter o lexema (t.value), ou seja, a cadeia de caracteres que define o token, de string para o tipo int.

Definição de Tokens

O conjunto de todos os tipos de tokens definidos pela linguagem é especificado por uma lista contendo strings, cada string determina um tipo. Além desta lista, que deve ter o nome `tokens`, é preciso definir as palavras reservadas da linguagem e seus respectivos tipos de token. Para isso, é utilizado uma estrutura do dicionário, onde cada chave é uma palavra reservada e seu respectivo valor é seu tipo. A lista `tokens` inclui todos os tipos definidos nos valores deste dicionário, que deve ser nomeado `reserved`. Veja a seguir:

```
import ply.lex as lex

reserved = {
   'int': 'INT',
   'float': 'FLOAT',
   'string': 'STRING',

   'if': 'IF',
   'then': 'THEN',
   'else': 'ELSE',
   'for': 'FOR',
   'break': 'BREAK',
   'print': 'PRINT',
   'return': 'RETURN',
   'def': 'DEF',
   'new': 'NEW',
   'null': 'NULL',
   'read': 'READ'
}

tokens = [
   'FLOAT_CONSTANT',
   'INT_CONSTANT',
   'STRING_CONSTANT',
   'IDENT',
   'PLUS',
   'MINUS',
   'MULTIPLY',
   'DIVIDE',
```

```
    'MODULE',
    'EQUALS',
    'ASSIGN',
    'NOT_EQUAL',
    'LPAREN',  # 'left parenthesis'
    'RPAREN',
    'LCBRACKET',  # 'left curly bracket'
    'RCBRACKET',
    'LBRACKET',
    'RBRACKET',
    'LTE',  # 'less than or equal'
    'LT',
    'GTE',
    'GT',
    'SEMICOLON',
    'COMMA'
] + list(reserved.values())
```

Definição Regular dos Tokens

A sintaxe de expressão regular utilizada segue a definição de expressões regulares da linguagem de Python (lib re), iniciada por um `r` e segue entre `‘`.

```
INT -> int
FLOAT -> float
STRING -> string
IF -> if
THEN -> then
ELSE -> else
FOR -> for
BREAK -> break
PRINT -> print
RETURN -> return
DEF -> def
NEW -> new
NULL -> null
READ -> read
PLUS -> r'\+'
MINUS -> r'-'
```

```
MULTIPLY -> r'\*'
DIVIDE -> r'\\'
MODULE -> r'%'
EQUALS -> r'=='
ASSIGN -> r'='
NOT_EQUAL -> r'!='
LPAREN -> r'\('
RPAREN -> r'\)'
LCBRACKET -> r'{'
RCBRACKET -> r'}'
LBRACKET -> r'\['
RBRACKET -> r']'
LTE -> r'<='
LT -> r'<'
GTE -> r'>='
GT -> r'>'
SEMICOLON ->  r'\;'
COMMA -> r','
COMMENT-> r'\#.*'
IDENT -> r'[a-zA-Z_]+[a-zA-Z0-9_]*'
STRING_CONSTANT -> r'".*"'
FLOAT_CONSTANT -> r'\d+\.\d+'
INT_CONSTANT -> r'\d+'
```

Erros

Exemplo de entrada e saída do programa

Entrada:

```
def max(int a, int b) {
  if (a >= b) {
      return a;
  }
  return b;
}
```

Saída:

```
(DEF,'def',1,0)
(IDENT,'max',1,4)
(LPAREN,'(',1,7)
```

```
(INT,'int',1,8)
(IDENT,'a',1,12)
(COMMA,',',1,13)
(INT,'int',1,15)
(IDENT,'b',1,19)
(RPAREN,')',1,20)
(LCBRACKET,'{',1,22)
(IF,'if',2,26)
(LPAREN,'(',2,29)
(IDENT,'a',2,30)
(GTE,'>=',2,32)
(IDENT,'b',2,35)
(RPAREN,')',2,36)
(LCBRACKET,'{',2,38)
(RETURN,'return',3,44)
(IDENT,'a',3,51)
(SEMICOLON,';',3,52)
(RCBRACKET,'}',4,56)
(RETURN,'return',5,60)
(IDENT,'b',5,67)
(SEMICOLON,';',5,68)
(RCBRACKET,'}',6,70)
```

Onde a saída representa: (token, valor, linha, coluna)

Tabela de símbolos

A implementação da tabela de símbolos é relativamente simples, utilizando o PLY. Na definição de cada token "complexo" (aqueles que são definidos por uma função, e não apenas por uma expressão regular), é possível executar operações extras. Então para armazenar informações sobre os tokens (construir a tabela de símbolos), definimos a variável do tipo `dict` (dicionário do Python), vazia, no objeto `Lexer`. Veja a linha indicada na inicialização do parsing:

```
if __name__ == '__main__':
        lexer = lex.lex()
    lexer.symbol_table = {}  # Inicialização da tabela de símbolos
    lex.runmain(lexer=lexer)
```

E como mencionado acima, para preenchê-la utilizamos uma operação extra na função de definição dos tokens. Tal operação de inserção na tabela será executada no parsing de cada símbolo:

```
def t_INT_CONSTANT(t):
   r'\d+'  # Any numeric character whose length is more than 0
   t.value = int(t.value)
   t.lexer.symbol_table[t.value] = (t.type, t.lineno, t.lexpos)  # Inserção
   return t
```

A linha indicada com o comentário "Inserção" no trecho de código acima mostra as informações adicionadas na tabela nesta fase do projeto: o tipo do símbolo (`t.type`), a número da linha em que ele se encontra (`t.lineno`) e seu índice relativo ao início do arquivo de entrada (`t.lexpos`).

Nas fases posteriores do projeto podemos adaptar aquela operação para modificar o estado armazenado na tabela, de acordo com o que for conveniente para, por exemplo, a análise léxica.

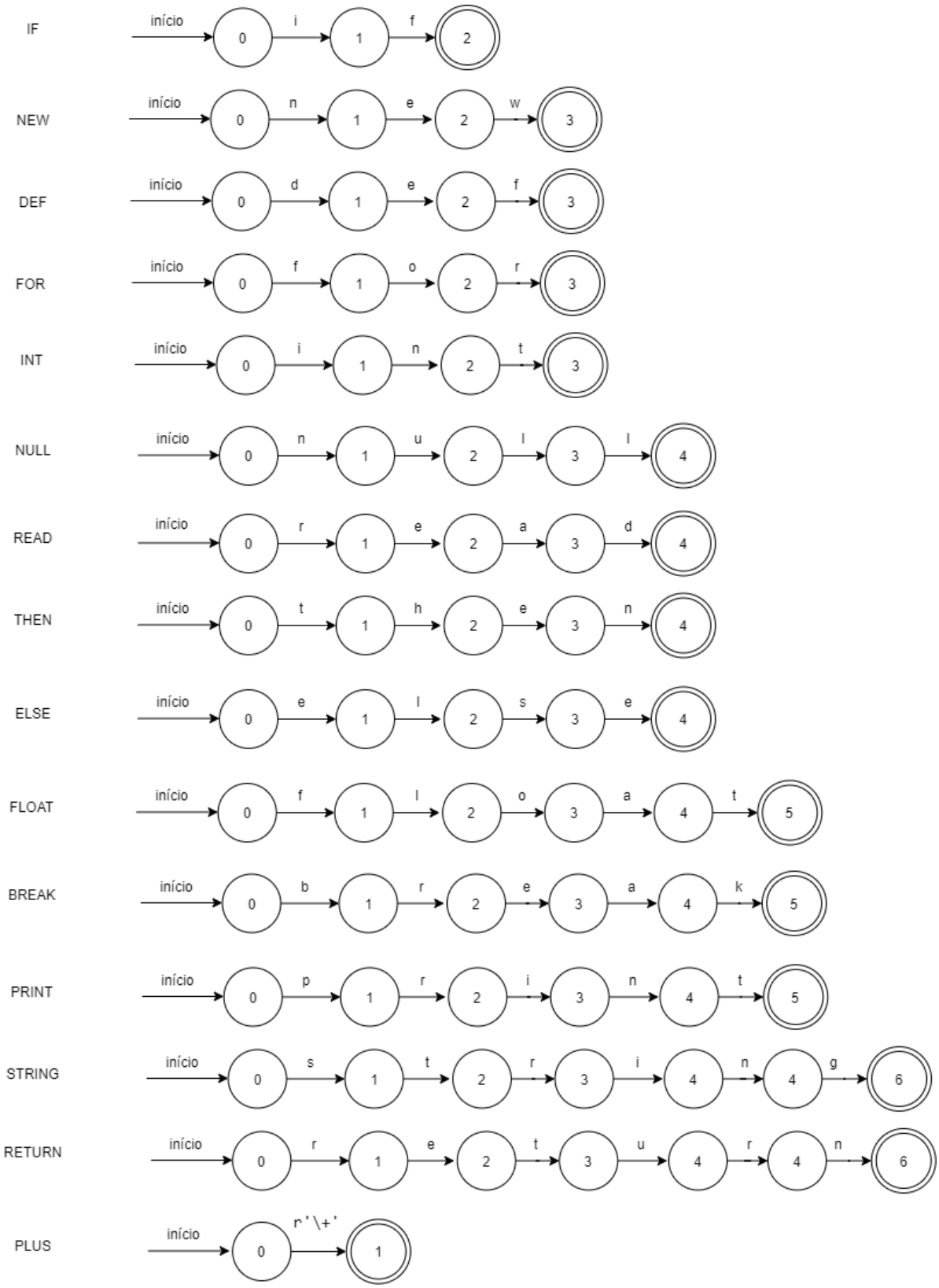
Vejamos um exemplo da tabela de símbolos para a função `max(int, int)`, definida anteriormente neste documento:

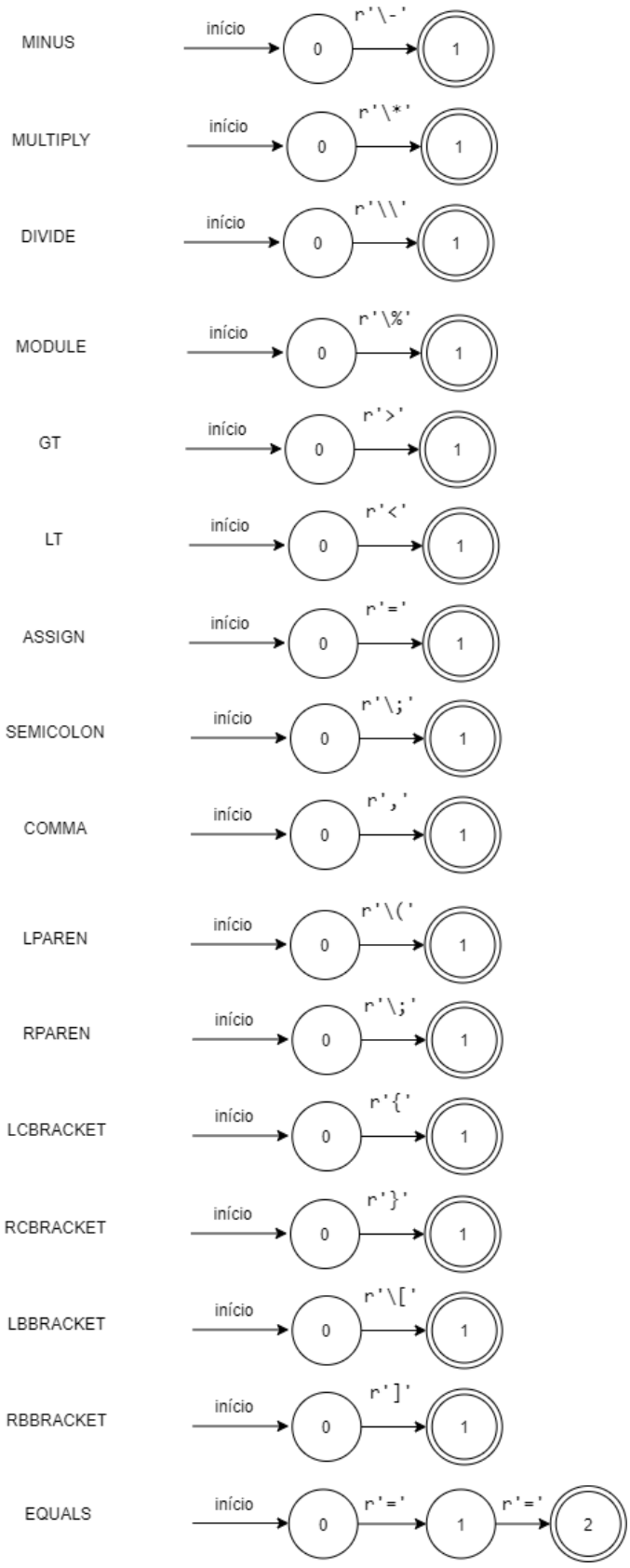
```
Symbol table:
{
 "def": ["DEF", 1, 0],
 "max": ["IDENT", 1, 4],
 "int": ["INT", 1,15],
 "a": ["IDENT", 3, 51],
 "b": ["IDENT", 5, 67],
 "if": ["IF", 2, 26],
 "return": ["RETURN", 5, 60]
}
```

O código fonte produzirá a tabela de símbolos para o programa de entrada neste formato (JSON). Note que o valor de uma chave é definido no código como uma tupla, mas nesta saída ela aparece como uma lista. Todo o estado armazenado na tabela poderá ser acessado desde que se tenha acesso à instância do Lexer. Por exemplo, a saída acima é produzida com `json.dumps(lexer.symbol_table)`, logo após o comando de execução do parsing. Ou seja:

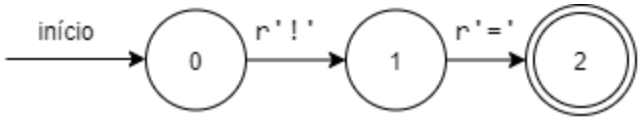
```
lex.runmain(lexer=lexer)

print('\n######################')
print('\nSymbol table: ')
print(json.dumps(lexer.symbol_table, indent=2))
```

Diagramas de Transição

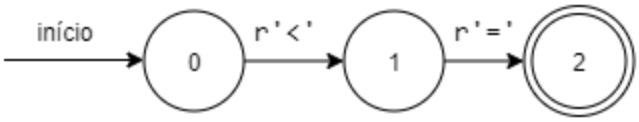



MINUS
início
0
r'\-'
1
MULTIPLY
início
0
r'*'
1
DIVIDE
início
0
r'\\'
1
MODULE
início
0
r'\%'
1
GT
início
0
r'>'
1
LT
início
0
r'<'
1
ASSIGN
início
0
r'='
1
SEMICOLON
início
0
r'\;'
1
COMMA
início
0
r','
1
LPAREN
início
0
r'\('
1
RPAREN
início
0
r'\;'
1
LCBRACKET
início
0
r'{'
1
RCBRACKET
início
0
r'}'
1
LBBRACKET
início
0
r'\['
1
RBBRACKET
início
0
r']'
1
EQUALS
início
0
r'='
1
r'='
2

NOT_EQUAL

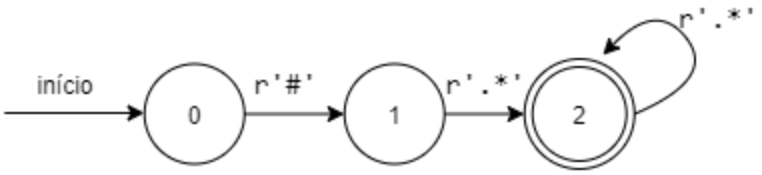


LTE

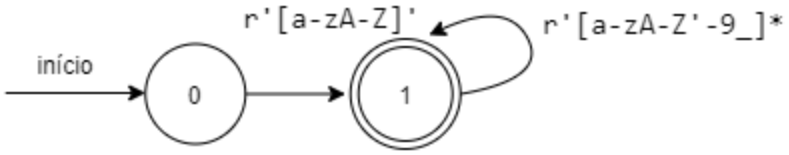


NOT_EQUAL

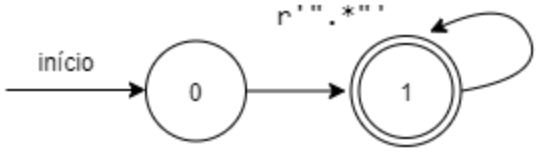
COMMENT



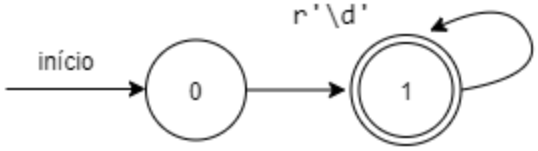
IDENT



STRING_CONSTANT



INT_CONSTANT



FLOAT_CONSTANT