Projeto e Implementação de um analisador léxico para a linguagem C-

Eduardo Kurek[1]
[1]Departamento Acadêmico de Computação (DACOM)
Universidade Tecnológica Federal do Paraná (UTFPR)

Abstract

Lexical analyzers are the first step in creating a compiler. They split the code into tokens, which are later used in subsequent compilation phases. This work aims to implement a lexical analyzer for the C- language using automata. For this implementation, a Mealy machine was employed. As it is equivalent to a Moore machine, it can also work similarly. After the lexical analysis phase, the output will be a list of tokens that describe the code structure.

Resumo

Analisadores léxicos são a primeira etapa na criação de um compilador. Eles separam o código em marcas, usadas posteriormente nas próximas fases de compilação. Este trabalho tem por finalidade a implementação de um analisador léxico para a linguagem **C-** usando autômatos. Neste trabalho usei a máquina de *Mealy* para a implentação, como ela é equivalente à máquina de *Moore*, também funcionará. Após a fase de análise léxica, teremos como saída uma lista de marcas, que descreve o código.

1 Introdução

Analisadores léxicos são programas capazes de identificar uma sequência de caracteres (lexemas) e converter em um *token* (marca). Os tokens que serão identificados pela linguagem C- estão descritos na Tabela 1. Além de identificar números positivos e negativos (NUMBER), variáveis que possuem letras e números (ID), e ignorar comentários entre barra e asterisco (/* Exemplo de comentário */).

1. Exemplo de código na linguagem C-

```
int gcd (int u, int v){
  if (v == 0) return u;
  else return gcd(v,u-u/v*v);
  /* u-u/v*v == u mod v */
}

void main(void){
  int x; int y;
  x = input();
  y = input();
  output(gcd(x,y));
}
```

Corresponding author(s): Eduardo Kurek, Universidade Tecnológica Federal do Paraná - Campo Mourão, eduardokurek@alunos.utfpr.edu.br

Lexema	Token
if	IF
else	ELSE
int	INT
float	FLOAT
return	RETURN
void	VOID
while	WHILE
+	PLUS
-	MINUS
*	TIMES
/	DIVIDE
<	LESS
<=	LESS_EQUAL
>	GREATER
>=	GREATER_EQUAL
==	EQUALS
!=	DIFFERENT
(	LPAREN
)	RPAREN
[	LBRACKETS
]	RBRACKETS
{	LBRACES
}	RBRACES
=	ATTRIBUTION
;	SEMICOLON
,	COMMA

Tabela 1: Tokens da linguagem C-.

Neste trabalho, será utilizado o aplicativo JFLAP (Rodger & Finley) para a visualização do autômato, e a biblioteca automata_python na linguagem python para a implementação do analisador léxico.

A máquina de Moore é um modelo de computação em que se transita de um estado para outro, e quando um estado é atingido, uma determinada sentença é escrita na saída, no nosso caso, a saída padrão do sistema operacional. Já a máquina de Mealy, é semelhante à de Moore, porém a sua saída é na transição, e não no estado.

1.1 Comparação entre máquina de Moore e Mealy

Para a nossa linguagem, devemos identificar que o lexema '<=' resulta no token LESS_EQUAL. Caso haja qualquer caractere diferente de '=' após o '<', então será impresso o token LESS, e também token do próximo caractere. Devido a esses casos, a máquina se torna mais complexa.

Considerando as letras do alfabeto (+,<,=) e que não há o lexema '==' na linguagem, veja na Figura 1 a solução para esse problema usando a máquina de Moore, e na Figura 2 usando uma máquina de Mealy.

Podemos notar a diferença de complexidade entre as duas máquinas. Isso acontece pois na máquina de Moore, é necessário criar um estado para cada saída diferente, e consequentemente, cada estado deve ter transição para todas as letras do alfabeto. Considerando um alfabeto de

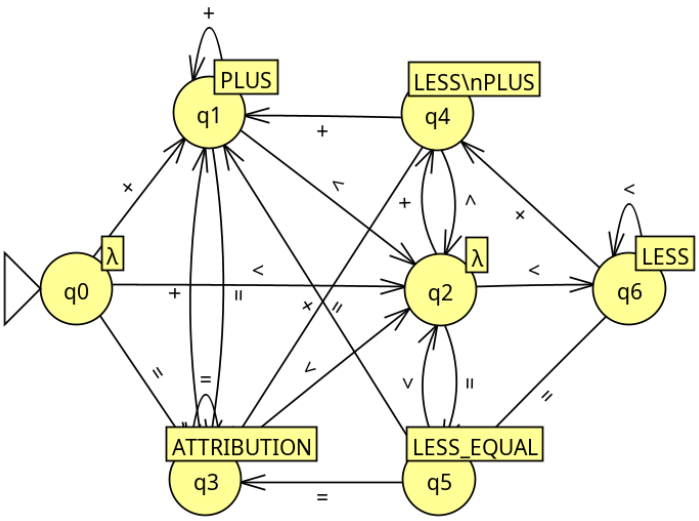

Figura 1: Exemplo da máquina de Moore

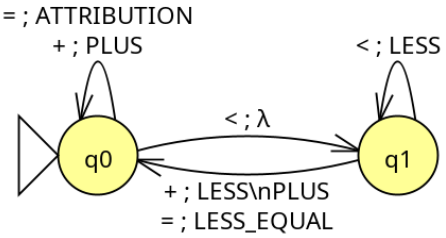

Figura 2: Exemplo da máquina de Mealy

mais de 30 letras, ficaria inviável compreender a máquina de Moore. Quando a saída está na transição, esse processo é simplificado. Logo, ambas as máquinas são equivalentes, porém, é muito mais complexo utilizar a máquina de Moore para o caso em questão. Por esse motivo, estarei utilizando a máquina de Mealy neste projeto, mas ainda assim, seria possível converte-la para a máquina de Moore.

2 Autômato no JFLAP

Para facilitar a visualização, não criei explicitamente todas as transições. Todas as transições não listadas em um estado, ficam listadas na *label* do estado.

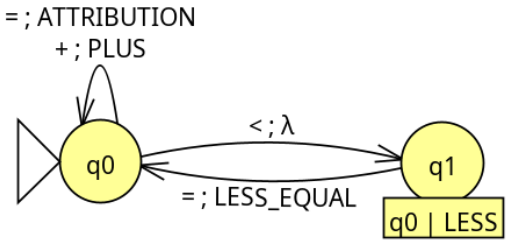

Figura 3: Exemplo de transição implícita em q1: cópia de q0 com o prefixo LESS e sobreposição da transição =

Na figura 3 podemos ver no estado `q1` o seguinte texto em sua label: **q0 | LESS**. Posso explicar essa sintaxe em 3 passos:

Passo 1, `q1` recebe uma cópia das transições de `q0`:

- `<;λ -> q1`

- `=;ATTRIBUTION -> q0`
- `+;PLUS -> q0`

Passo 2, Todas as saídas das transições de `q1` recebem o prefixo `LESS`:

- `<;LESS -> q1`
- `=;LESS ATTRIBUTION -> q0`
- `+;LESS PLUS -> q0`

Depois, a transição (`=;LESS ATTRIBUTION -> q0`) é sobreposta pela (`=;LESS_EQUAL -> q0`). Assim, as transições do estado `q1` são:

- `<;LESS -> q1`
- `=;LESS_EQUAL -> q0`
- `+;LESS PLUS -> q0`

Ou seja, as figuras 2, 1 e 3 são equivalentes. Apenas utilizarei dessa sintaxe para facilitar a visualização e compreensão do autômato.

2.1 Letras únicas

Para identificar os lexemas que eram apenas uma única letra, basta fazer uma transição para o mesmo estado, e que imprime o token. Veja a Figura 4.

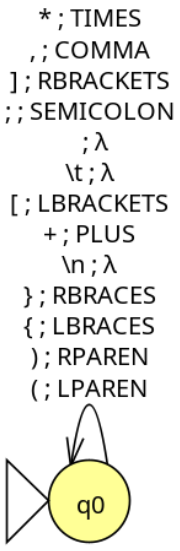

Figura 4: Transições dos lexemas de um único caractere

2.2 Comentários

Para ignorar comentários, é preciso fazer um estado em que o autômato fica preso ao reconhecer '/*', e só é liberado depois de reconhecer '*/'.

Na Figura 5 podemos ver que caso o estado `q0` identifique o caractere '/' ele passa ao estado `q1`. O estado `q1` possui uma definição implícita de todas as transições que não sejam '*', isso permite que sejam identificadas sequências como '/+' corretamente. Em `q14` o autômato fica preso, e só

passa para o estado q15 se reconhecer '*', e posteriormente retorna ao estado inicial se reconhecer '/' novamente.

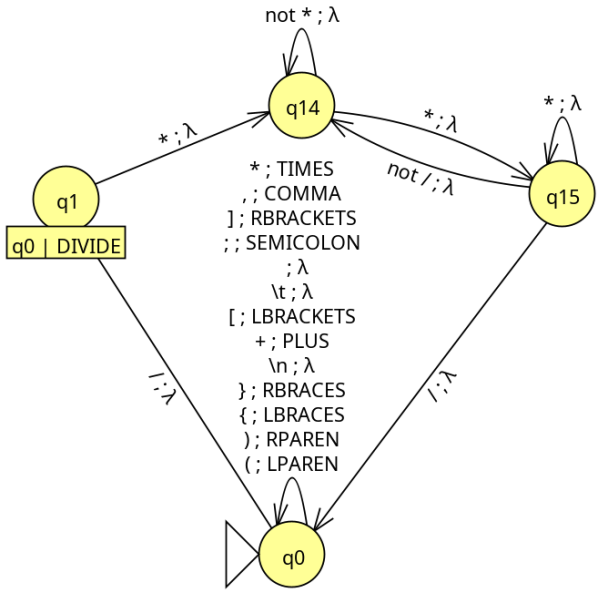

Figura 5: Autômato para ignorar comentários

2.3 Lexemas duplos

Lexemas duplos são aqueles que precisam de dois caracteres para serem reconhecidos, todos seguem o mesmo princípio já visto na Figura 3.

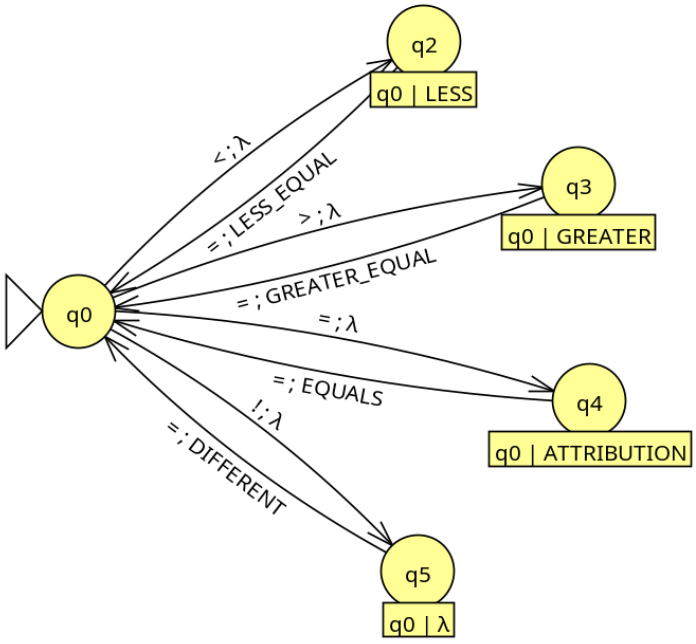

Figura 6: Autômato para identificar os lexemas duplos

2.4 Identificadores

Identificadores são palavras iniciadas por letra, seguidas de repetições de letras ou números. Um simples autômato para reconhecer identificadores está descrito na Figura 7. A abreviação *an* significa todos os alfa-numéricos.

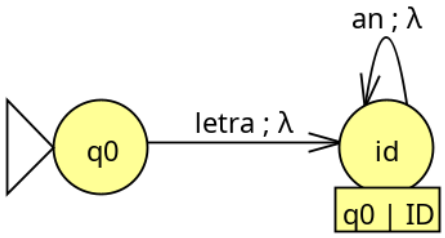

Figura 7: Autômato para reconhecer IDs

O autômato da Figura 7 não está incluso no autômato final dessa forma, há mais restrições. Essa é apenas uma forma genérica para reconhecer identificadores.

2.5 Palavras reservadas

A linguagem C- deve reconhecer as seguintes palavras reservadas: **if**, **int**, **else**, **float**, **return**, **void** e **while**. A Figura 8 exibe o reconhecimento da palavra 'if'.

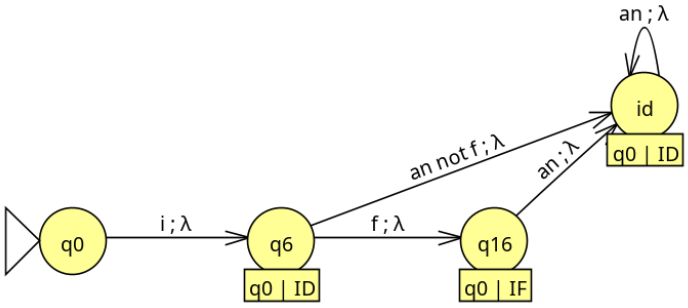

Figura 8: Autômato para reconhecer a palavra reservada `if`

Ainda na Figura 8, pode-se notar que no estado `q6` há uma transição com qualquer alfa-numérico (an) que não seja 'f' para `id`. Essa transição existe para que cadeias como **ia** ou **i3a** sejam consideradas `ID`. Assim também, a transição de `q16` para `id` identifica cadeias como **ifa**, **if1**, etc.

As transições implícitas nos estados permitem identificar o encerramento das palavras. Por exemplo, com a transição implícita no estado `q6`, é possível reconhecer a palavra **i+** como: `ID PLUS`. Pois através dos passos da Seção 2 o estado `q6` terá a transição `+;ID PLUS -> q0`.

As palavras `if` e `int` começam pela mesma letra, logo, deve haver um tratamento especial para esses dois casos. Veja na Figura 9.

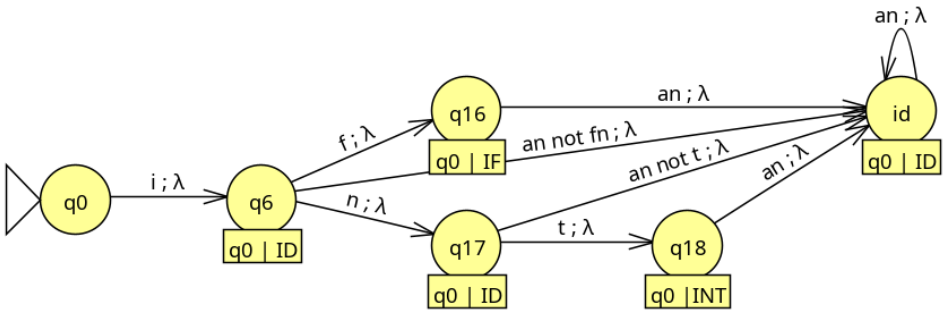

Figura 9: Autômato para reconhecer as palavras reservadas `if` e `int`

Todo estado intermediário de uma palavra reservada, deve ter uma transição para `id` que abrange todos os alfa-numéricos restantes daquele estado. `q6` tem uma transição para `id` com todos os alfa-numéricos diferentes de 'f' e 'n', pois estes caracteres já possuem transições para os estados `q16` e `q17` respectivamente.

Todo estado final de uma palavra reservada deve ter uma transição para o estado `id` com todos os alfa-numéricos. Como exemplo, temos os estados `q16` e `q18`.

Esse processo de construção do autômato se repete para todas as palavras reservadas restantes. Para melhor visualização, irei omitir todas as transições para `id`. As Figuras 10 e 11 destacam essas diferenças para a palavra reservada `else`.

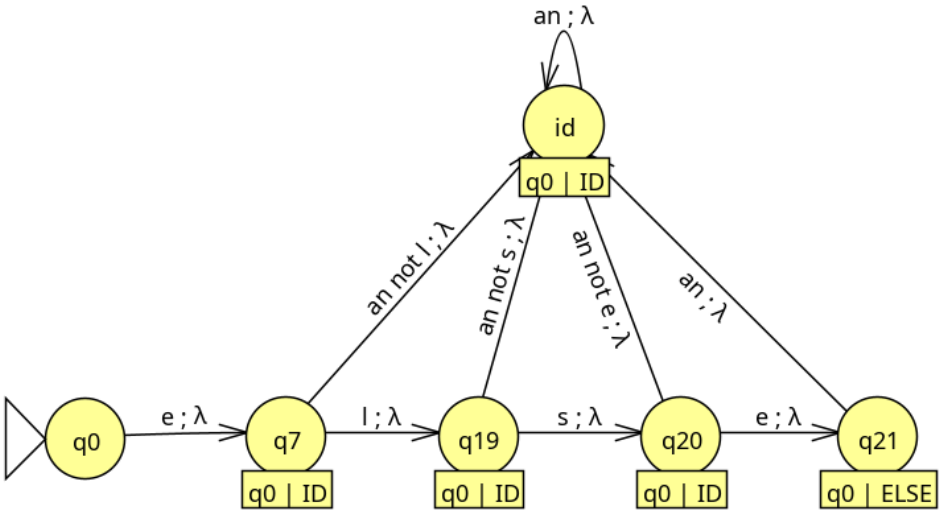

Figura 10: Autômato para reconhecer a palavra reservada `else` com todas as transições para `id`

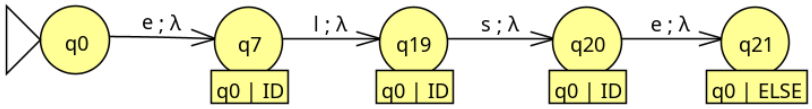

Figura 11: Autômato para reconhecer a palavra reservada `else` sem as transições para `id`

2.6 Números

Os números podem ser negativos, por isso, o símbolo '–' pode estar antes de uma sequência numérica e deve ser reconhecido corretamente. Caso haja qualquer caractere diferente de número apos o sinal de menos, será reconhecido o token `MINUS`.

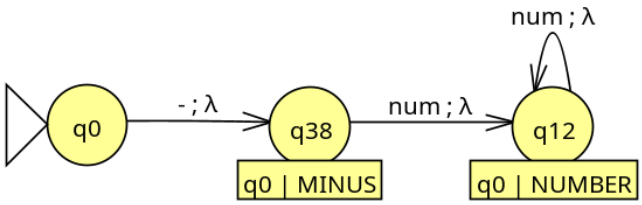

Figura 12: Autômato para reconhecer números

2.7 Resultado do autômato

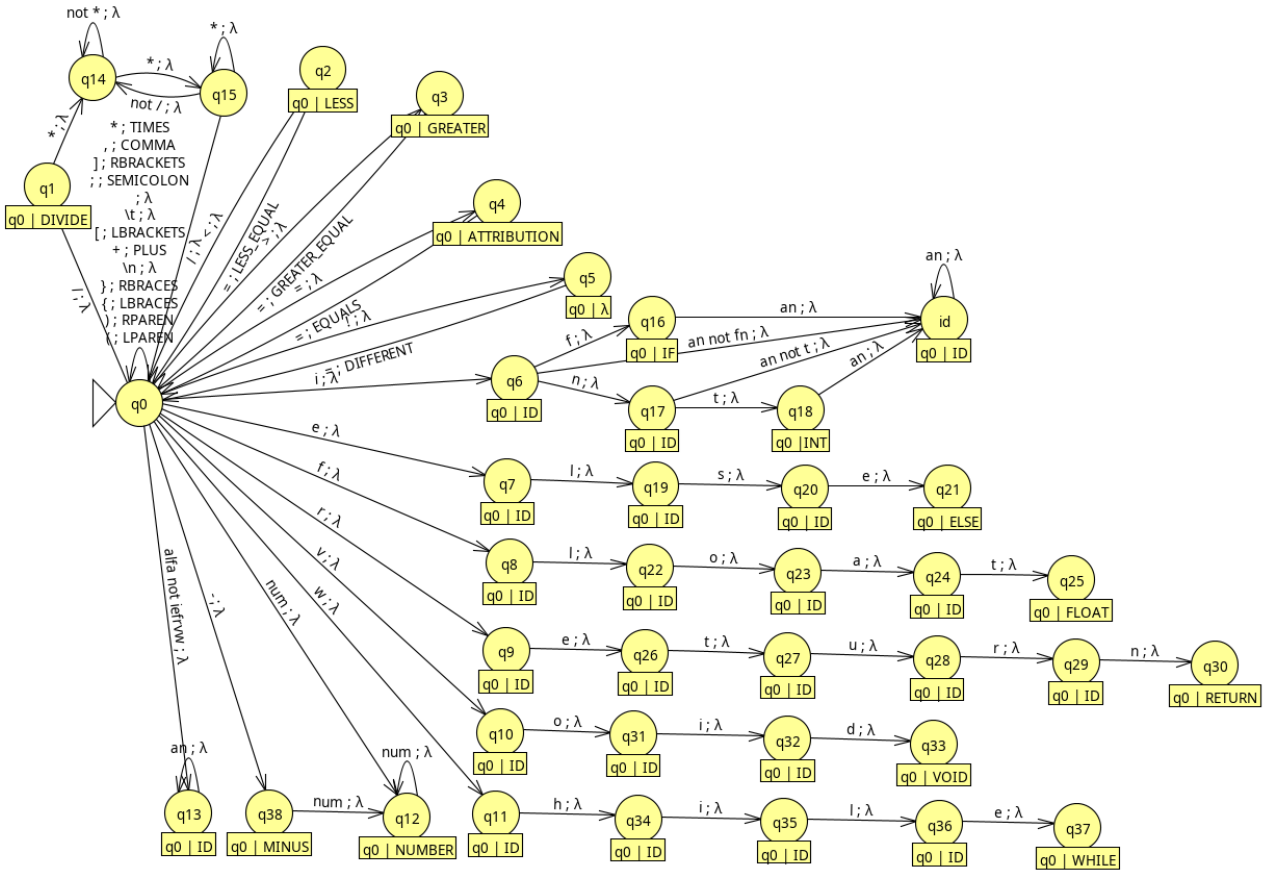

Figura 13: Resultado final do autômato para o analisador léxico

Na Figura 13, o estado `q13` não é necessário, pois poderia ser substituido pelo estado `id`. No entanto, criei outro para melhorar a visualização.

3 Implementação em Python

Para a implementação do autômato na linguagem de programação Python, foi escolhida a biblioteca automata_python. Para criar uma máquina de Mealy com essa biblioteca é necessário informar os seguintes dados:

2. Interface para criar uma máquina Mealy

```
mealy = Mealy(
    states,
    input_alphabet,
    output_alphabet,
    transitions,
    initial_state
)
```

Como as transições do estado inicial seriam copiadas para todos os outros estados, declarei uma variável global que continha essas transições.

3. Transições do estado inicial (q0)

```
initial = {
    # Transições para o estado q0
    ' ': ('q0', ''),
    '\t': ('q0', ''),
```

```
    '\n': ('q0', ''),
    '+': ('q0', 'PLUS\n'),
    '*': ('q0', 'TIMES\n'),
    '(': ('q0', 'LPAREN\n'),
    ')': ('q0', 'RPAREN\n'),
    '[': ('q0', 'LBRACKETS\n'),
    ']': ('q0', 'RBRACKETS\n'),
    '{': ('q0', 'LBRACES\n'),
    '}': ('q0', 'RBRACES\n'),
    ';': ('q0', 'SEMICOLON\n'),
    ',': ('q0', 'COMMA\n'),

    # Transições para outros estados
    '-': ('q38', ''),
    '/': ('q1', ''),
    '<': ('q2', ''),
    '>': ('q3', ''),
    '=': ('q4', ''),
    '!': ('q5', ''),
    'i': ('q6', ''),
    'e': ('q7', ''),
    'f': ('q8', ''),
    'r': ('q9', ''),
    'v': ('q10', ''),
    'w': ('q11', ''),

    # Números
    '0': ('q12', ''), '1': ('q12', ''),
    '2': ('q12', ''), '3': ('q12', ''),
    '4': ('q12', ''), '5': ('q12', ''),
    '6': ('q12', ''), '7': ('q12', ''),
    '8': ('q12', ''), '9': ('q12', ''),

    # Restante das letras
    'a': ('id', ''), 'b': ('id', ''), 'c': ('id', ''), 'd': ('id', ''), 'g': ('id', ''),
    'h': ('id', ''), 'j': ('id', ''), 'k': ('id', ''), 'l': ('id', ''), 'm': ('id', ''),
    'n': ('id', ''), 'o': ('id', ''), 'p': ('id', ''), 'q': ('id', ''), 's': ('id', ''),
    't': ('id', ''), 'u': ('id', ''), 'x': ('id', ''), 'y': ('id', ''), 'z': ('id', ''),
}
```

Criei também uma classe para facilitar a manipulação das transições. Essa classe inicia com as transições contidas na variável `initial`, e possui métodos para adicionar prefixo, sobrescrever transições, e `forEach`s que servem como funções auxiliares.

4. Classe Transition

```
class Transition:
    def __init__(self):
        self.transition = initial.copy()

    def prefix(self, pre):
        for i, j in self.transition.items():
            self.transition[i] = (j[0], pre + j[1])
        return self

    def overwrite(self, values):
        self.transition.update(values)
        return self

    def forEach(self, func):
        for i, j in self.transition.items():
```

```
            self.transition[i] = func(i, j)
        return self

    def forEachAlfaNum(self, func):
        for i, j in self.transition.items():
            if i in alfanum:
                self.transition[i] = func(i, j)
        return self

    def forEachAlfa(self, func):
        for i, j in self.transition.items():
            if i in alfa:
                self.transition[i] = func(i, j)
        return self

    def forEachNum(self, func):
        for i, j in self.transition.items():
            if i in num:
                self.transition[i] = func(i, j)
        return self

    def __getitem__(self, key):
        return self.transition[key]
```

Para auxiliar a criação dos estados das palavras reservadas, defini duas funções que realizam os processos vistos na seção 2.5.

5. Funções auxiliares para criar estados das palavras reservadas

```
def LastReservedWordState(word):
    transitions = Transition()
    transitions.prefix(word + '\n')
    transitions.forEachAlfaNum(lambda _, __: ('id', ''))
    return transitions

def IntermediateReservedWordState(input, nextState):
    transitions = Transition()
    transitions.prefix('ID\n')
    transitions.forEachAlfaNum(lambda _, __: ('id', ''))
    for i in range(len(input)):
        transitions.overwrite({ input[i]: (nextState[i], '')})
    return transitions
```

Por fim, criei a transição de cada estado da máquina.

6. Transições do analisador léxico

```
transitions = {
    'id': Transition()
        .prefix("ID\n")
        .forEachAlfaNum(lambda _, __: ('id', '')),

    'q0': Transition(),

    'q1': Transition()
        .prefix("DIVIDE\n")
        .overwrite({ '*': ('q14', '') }),

    'q14': Transition()
        .forEach(lambda _, __: ('q14', ''))
        .overwrite({ '*': ('q15', '') }),

```

```
    'q15': Transition()
        .forEach(lambda _, __: ('q14', ''))
        .overwrite({ '/': ('q0', '') }),

    'q2': Transition()
        .prefix("LESS\n")
        .overwrite({ '=': ('q0', 'LESS_EQUAL\n') }),

    'q3': Transition()
        .prefix("GREATER\n")
        .overwrite({ '=': ('q0', 'GREATER_EQUAL\n') }),

    'q4': Transition()
        .prefix("ATTRIBUTION\n")
        .overwrite({ '=': ('q0', 'EQUALS\n') }),

    'q5': Transition()
        .overwrite({ '=': ('q0', 'DIFFERENT\n') }),


    # IF e INT
    'q6': IntermediateReservedWordState(['f', 'n'], ['q16', 'q17']),
    'q16': LastReservedWordState("IF"),
    'q17': IntermediateReservedWordState(['t'], ['q18']),
    'q18': LastReservedWordState("INT"),

    # ELSE
    'q7': IntermediateReservedWordState(['l'], ['q19']),
    'q19': IntermediateReservedWordState(['s'], ['q20']),
    'q20': IntermediateReservedWordState(['e'], ['q21']),
    'q21': LastReservedWordState("ELSE"),

    # FLOAT
    'q8': IntermediateReservedWordState(['l'], ['q22']),
    'q22': IntermediateReservedWordState(['o'], ['q23']),
    'q23': IntermediateReservedWordState(['a'], ['q24']),
    'q24': IntermediateReservedWordState(['t'], ['q25']),
    'q25': LastReservedWordState("FLOAT"),

    # RETURN
    'q9': IntermediateReservedWordState(['e'], ['q26']),
    'q26': IntermediateReservedWordState(['t'], ['q27']),
    'q27': IntermediateReservedWordState(['u'], ['q28']),
    'q28': IntermediateReservedWordState(['r'], ['q29']),
    'q29': IntermediateReservedWordState(['n'], ['q30']),
    'q30': LastReservedWordState("RETURN"),

    # VOID
    'q10': IntermediateReservedWordState(['o'], ['q31']),
    'q31': IntermediateReservedWordState(['i'], ['q32']),
    'q32': IntermediateReservedWordState(['d'], ['q33']),
    'q33': LastReservedWordState("VOID"),

    # WHILE
    'q11': IntermediateReservedWordState(['h'], ['q34']),
    'q34': IntermediateReservedWordState(['i'], ['q35']),
    'q35': IntermediateReservedWordState(['l'], ['q36']),
    'q36': IntermediateReservedWordState(['e'], ['q37']),
    'q37': LastReservedWordState("WHILE"),

```

```
    # Números
    'q12': Transition()
        .prefix("NUMBER\n")
        .forEachNum(lambda _, __: ('q12', '')),

    'q38': Transition()
        .prefix("MINUS\n")
        .forEachNum(lambda _, __: ('q12', '')),
}
```

4 Resultados

Para executar o analisador léxico, podemos informar o arquivo a ser analisado como parâmetro, e então a saída será os tokens daquele código. É possível adicionar a *flag* -k para mostrar apenas as chaves dos tokens e erros, desse jeito, não serão mostradas informações adicionais.

7. Código para teste do analisador léxico

```
int a;
float b;

int main(void){
  int x;
  float y;

  return(0);
}
```

8. Resultado do analisador léxico

```
INT
ID
SEMICOLON
FLOAT
ID
SEMICOLON
INT
ID
LPAREN
VOID
RPAREN
LBRACES
INT
ID
SEMICOLON
FLOAT
ID
SEMICOLON
RETURN
LPAREN
NUMBER
RPAREN
SEMICOLON
RBRACES
```

Referências

Rodger, Susan H. & Thomas W. Finley. 2006. *Jflap: An interactive formal languages and automata package*. Sudbury, MA: Jones & Bartlett Learning.