Implementação de um Analisador Léxico Utilizando Máquina de Moore

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

1 Introdução

Segundo LOUDEN (2004) Compiladores são programas de computador que traduzem de uma linguagem para outra. Um compilador recebe como entrada um programa escrito na **linguagem-fonte** e produz um programa equivalente na **linguagem-alvo**.

Na maior parte dos casos, o programa escrito na linguagem-fonte, como por exemplo C, será transformado em um código objeto, ou mesmo código de máquina. Em alguns casos, podemos ter também o problema de como tornar um programa executável em diversas arquiteturas diferentes, sendo assim necessário o uso de um compilador para adaptar as do programa inicial para a arquitetura do computador em questão.

O objetivo deste trabalho é apresentar a primeira etapa com respeito à criação de um compilador, a análise léxica, também chamada de scanner ou lexer. Esta etapa tem como objetivo criar uma lista de tokens a partir do código de entrada, onde essa lista será utilizada posteriormente por um analisador sintático, também chamado de *parser*. Neste trabalho, estarei apresentando uma implementação em máquina de Moore, um autômato finito determinístico (AFD) que possui saída em cada um dos estados do AFD.

1.1 Máquina de Moore

Como descrito em Paulo Blauth Menezes (2011), a máquina de Moore é um AFD com saídas associadas a estados. Uma forma simples de olhar para a máquina é pensar nela como um AFD que irá imprimir uma saída sempre que chegar a um determinado estado. A máquina de Moore, assim como um AFD, pode ser definida por uma N-Tupla :

$$M(\Sigma, Q, \delta, q_0, F, \Delta, \delta_S)$$

onde:

- Σ: Alfabeto de entrada
- Q: Conjunto de estados
- δ: Função de transição
- q_0: Estado inicial
- F: Estados que possuem saída
- Δ: Alfabeto de saída
- δ_S: Função associada às saídas

Corresponding author(s): Vinícius Kurek, Universidade Tecnológica do Paraná, viniciuskurek@alunos.utfpr.edu.br

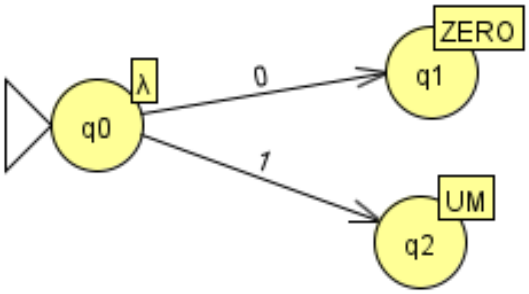

Figura 1: Exemplo de uma Máquina de Moore simples

Um exemplo simples de uma máquina de Moore seria um reconhecedor de números binários que irá imprimir o número em si escrito por extenso. A máquina citada está descrita na Figura 1

Observe que podemos extrair as informações para a tupla M a partir da Figura 1:

- Σ: { 0, 1 }
- Q: { q0, q1, q2 }
- δ: { ($\delta(q0, 0) = q1$), ($\delta(q0, 1) = q2$) }
- q_0: q0
- F: { q1, q2 }
- Δ: { ZERO , UM }
- δ_S: { ($\delta_s(q1) = ZERO$), ($\delta_s(q2) = UM$) }

2 Análise Léxica

A análise léxica, varredura, ou *scanner*, tem como objetivo principal transformar o programa de entrada em *tokens*, ou marcas. Cada um desses *tokens* representa uma instrução ou unidade de informação do programa a ser "varrido". Como já esclarecido anteriormente, estaremos utilizando máquina de Moore. Todo token terá um lexema o qual será a palavra que está representando e uma classe do token em si que será definido pela linguagem utilizada.

Para a implementação do projeto estaremos utilizando a linguagem de programação Python junto de uma biblioteca chamada automatha_python a qual receberá algumas mudanças para satisfazer o meu uso.

3 A Linguagem C-

Antes de começarmos a construir o *lexer* há a necessidade de definir para qual linguagem iremos fazê-lo, em nosso caso utilizaremos uma simplificação da linguagem C, chamada C-. As convenções léxicas para a linguagem foram extraídas de LOUDEN (2004).

Algumas coisas que podemos definir previamente são que a parte léxica da linguagem necessita de suportar o alfabeto de entrada como sendo o alfabeto romano com letras maiúscula e minúsculas, números, e alguns símbolos, além de fazer o reconhecimento de alguns conjuntos, sendo eles: números, identificadores, palavras reservadas e símbolos.

3.1 Alfabeto de Entrada

Uma parte importante de construir um *lexer* é a necessidade de delimitar o conjunto de caracteres os quais serão analisados, ou seja, quais caracteres são aceitos pela linguagem. Os caracteres os quais a C- aceita são:

$letters = \{\{[A-Z]\} \cup \{[a-z]\}\}$
$numbers = \{[0-9]\}$
$simbols$ = { +, -, *, /, =, !, <, >, (,), {, }, [,], ;, ,}

$$\Sigma = letters \cup numbers \cup simbols$$

Perceba que, por exemplo, ao colocar [A-Z], estou considerando todos os caracteres entre A e Z , também precisarei de mais alguns conjuntos auxiliares durante os meus códigos para explicar a lógica de meu programa:

$reserved\_letters = \{i, e, f, r, v, w\}$
$difficult\_treatment\_symbols = \{!, <, >, =, /\}$
$easy\_treatment\_symbols = \{simbols - hard\_treatment\_symbols\}$

3.2 Palavras Reservadas

Toda linguagem de programação possui ao menos algumas palavras reservadas, a C- não é diferente. Palavras reservadas são aquelas as quais o compilador as reconhecerá e aplicará uma lógica dentro do que ela significa. Por exemplo, em C, a palavra ”for”significa que ocorrerá um *loop* que se esperam três parâmetros dentro dele, uma variável que será inicializada dentro do laço, uma condição de parada para o laço e por fim algo que acontecerá ao final do laço, o exemplo está no Código 1.

```
1 for(int i = 0; i < 10; i++) {
2     // Corpo do loop
3 }
```

Código 1: Exemplo de um loop `for` em C.

Observe que no Código 1, temos duas palavras reservadas, o próprio lexema ”for”e o lexema ”int”, eles não podem ser reconhecidos como identificadores, pois fazem parte da lógica interna da linguagem, eles têm que ser reconhecidos como ”FOR”e ”INT”, respectivamente. Assim como em C, no C- as palavras reservadas são em inglês e *token* do lexema é uma capitalização da própria palavra. As palavras reservadas que consideraremos do C- e os respectivos *tokens* gerados por seus lexemas estão mostradas na Tabela 1.

Tabela 1: Palavras Reservadas

Palavras Reservadas (Lexemas)	Tokens Correspondentes
if	IF
else	ELSE
int	INT
float	FLOAT
return	RETURN
void	VOID
while	WHILE

3.3 Símbolos

Como já dito, toda palavra válida no código fonte gerará um *token*, ou marca, de identificação na saída da varredura. A Tabela 2 apresenta a lista de tokens que serão gerados pelos símbolos, os quais, assim como as palavras reservadas, são necessários para a lógica do programa.

Tabela 2: Operadores e Delimitadores

Lexemas dos Símbolos	Tokens Correspondentes
+	PLUS
-	MINUS
*	TIMES
/	DIVIDE
<	LESS
<=	LESS_EQUAL
>	GREATER
>=	GREATER_EQUAL
==	EQUALS
!=	DIFFERENT
(	LPAREN
)	RPAREN
[	LBRACKETS
]	RBRACKETS
{	LBRACES
}	RBRACES
;	SEMICOLON
,	COMMA

3.4 Números e Identificadores

Todo o restante dos caracteres da linguagem são os números e identificadores, isso significa que, caso a máquina de Moore receba um número, ela irá imprimir NUMBER, caso receba uma sequencia de caracteres que comece com uma letra, e também onde essa letra não é a palavra reservada, imprimirá ID. Há algumas outras coisas a serem consideradas, como por exemplo, um identificador que começa com uma letra de uma palavra reservada, porém, tais exceções serão mencionadas quando necessário.

4 Implementação do Analex

Para implementar o projeto estarei utilizando a linguagem de programação Python e fazendo o uso da biblioteca automata_python que fornece o suporte para a máquina de Moore. Essa biblioteca precisa que você passe as informações da N-Tupla citada na seção 1.1, os comentários inclusos no código foram feitos pelo autor da biblioteca. Há algumas mudanças feitas por mim no código apresentado no Código 2. Na função `get_output_from_string` eu tive que criar uma lista temporária que garante que a string que termine com um caractere vazio, fiz isso para que eu nunca ignore o último caractere na máquina. Além disso também criei a variável `tokens` e fiz com que ela recebesse todos os *tokens* encontrados pelo laço da função, e fiz a função retornar um ”\n”entre cada um dos *tokens*.

```
1 class Moore(object):
2     """Moore Machine : Finite Automata with Output"""
```

```
3
4       def __init__(self, states, input_alphabet, output_alphabet,
    transitions, initial_state, output_table ):
5           """
6           states: Finite set of states
7           input_alphabet: Alphabet of letters for forming input string
8           output_alphabet: Alphabet of letters for forming output
    characters
9           transitions: Transition Table
10          output_table: Output Table to show what character from
    output_alphabet is printed when a state from 'states'
11          is reached
12          """
13
14          self.states = states
15          self.input_alphabet = input_alphabet
16          self.output_alphabet = output_alphabet
17          self.transitions = transitions
18          self.output_table = output_table
19          self.initial_state = initial_state
20
21      def get_output_from_string(self, string):
22          """Return Moore Machine's output when a given string is given as
    input"""
23          temp_list = list(string) + ['']
24          tokens = []
25          current_state = self.initial_state
26
27          initial_token = self.output_table.get(current_state, '')
28          if initial_token:
29              tokens.append(initial_token)
30
31          for x in temp_list:
32              current_state = self.transitions[current_state][x]
33              token = self.output_table.get(current_state, '')
34              if token:
35                  tokens.append(token)
36
37          return "\n".join(tokens)
```

Código 2: Classe que Define a Máquina de Moore

4.1 Analex

No analex, no código principal, há a necessidade de criar um objeto Moore, com os atributos já mostrados no Código 2, para isso eu criei uma instanciação de cada um deles, observe no Código 3. Nesse código eu consigo chamar a função da classe Moore para receber todos os *tokens* e os imprimi-los no terminal.

```
1 def main():
2
3     moore = Moore(
4                     states,
5                     input_alphabet,
6                     output_alphabet,
7                     transition_table,
8                     initial_state,
9                     output_table
10                )
```

```
11
12      output = (moore.get_output_from_string(source_file))
13      output = output.replace("\r\n", "\n")
14      print(output)
```

Código 3: Classe que Define a Máquina de Moore

```
1  def create_state(name):
2      transition_table[name] = {}
3
4  def create_output(name, output):
5      output_table[name] = output
6
7  def join_output(name, first_output, second_output):
8      output_table[name] = f'{output_table[first_output]}\n{output_table[
       second_output]}'
9
10 def copy_transition(to_state, from_state='start', exception=None):
11     if exception is None:
12         exception = []
13     for input, output in transition_table[from_state].items():
14         if input not in exception:
15             transition_table[to_state][input] = output
```

Código 4: Funções Auxiliares no Analex

4.1.1 *Start*

Como já foi mostrado na seção 3.1, esses são os todos os caracteres aceitos pela linguagem, e como a máquina de Moore é um AFD, isso significa que todo estado precisa de uma transição para cada um daqueles caracteres, optei por fazer o estado inicial ter a inclusão todas as transições por força bruta. Observe como fiz a transição *start* no Código 5.

Peço que olhe as funções descritas no Código 4, pois acabarei usando elas com frequência ao longo dos códigos, como por exemplo, na transição *start* fiz o uso do `create_state` para inserir o estado *start* no dicionário chamado `transition_table`.

Sabendo que a transição inicial precisará lidar com todos os os caracteres, começo definindo que para cada letra presente no conjunto letters, caso a letra não esteja no conjunto reservado eu crio a transição de *start* com essa letra para o tratamento de id, caso a letra esteja no conjunto de letras reservadas cada uma delas vai para um novo estado que terá o nome da concatenação da letra com uma string `_treatment`.

O tratamento de números é mais simples, caso seja reconhecido um número na transição inicial, então esse número levará a transição `number_treatment`.

Para o tratamento de símbolos, para cada símbolo que possui um tratamento fácil será criado um estado com a concatenação desse símbolo com `_accepted` e então o símbolo terá a transição para esse novo estado. Caso o símbolo esteja no conjunto de símbolos de tratamento difícil irá criar um estado que será a concatenação de tal símbolo com `_treatment` e então esse símbolo receberá a transição até tal estado.

E por fim os caracteres delimitadores, que não necessariamente "existem", já que são aqueles caracteres que dizem se há um espaço em branco, uma quebra de linha, ou uma tabulação, serão apenas consumidos na transição inicial.

O motivo pelo qual comecei com o estado inicial, foi pois usei ele como estado padrão para a função auxiliar `copy_transition`, já que constantemente tenho que copiar as transições do

estado inicial para estados que possuem saída na `output_table`, criando assim um laço na minha máquina, para que ela não morra enquanto ela não consuma toda a cadeia de entrada.

```
# start state
create_state('start')

# start letters
for letter in letters:
    if letter not in reserved_letters: # Se a letra não for uma letra
    inicial de palavra reservada
        transition_table['start'][letter] = 'id_treatment'
    else:
        transition_table['start'][letter] = f'{letter}_treatment'
#
# start numbers
for number in numbers:
    transition_table['start'][number] = 'number_treatment'
#
# start symbols
for symbol in symbols:
    if symbol not in difficult_treatment_symbols:
        transition_table['start'][symbol] = f'{symbol}_accepted'
    else:
        transition_table['start'][symbol] = f'{symbol}_treatment'
#
# start delimiter_characters
for char in delimiter_characters:
    transition_table['start'][char] = 'start'
#
# start state finished
```

Código 5: Definição da Transição Inicial

4.2 Tratamento de Identificadores

Observe na Figura 2 uma abstração de como foi feita a implementação para o tratamento de identificadores, há uma transição para o estado `id_treatment` para cada letra no alfabeto de entrada que não está inclusa no conjunto de `reserved_letters`.

A partir do `id_treatment`, qualquer letra ou número que chegar ali será considerado parte do nome do identificador, sendo assim, irá consumir todas letras e números até encontrar ou um caractere delimitador ou um símbolo. No caso de símbolos, se o símbolo não estiver no conjunto de `hard_streatment_symbols` será criado um novo estado que concatena o nome e saída do `id_accept` e nome e saída do estado para o qual o símbolo leva, além de concatenar a saída de ambos os estados `id_accepted` e `{symbol}_accepted`. Caso seja um símbolo que é difícil de tratar é copiada o estado de tratamento desse `symbol`, e essa cópia terá o nome de `id_accepted_{symbol}_treatment`, observe que o novo estado contém `id_accepted` no nome, isso faz com que ele receba a mesma saída de `id_accepted` na `output_table`. Além disso tudo também temos o caso base, onde caso encontre um caractere delimitador o idendtificador terá sido reconhecido. Vale lembrar que `id_accepted` e `id_accepted_{symbol}_accepted` receberam uma cópia de todas as transições da transição inicial.

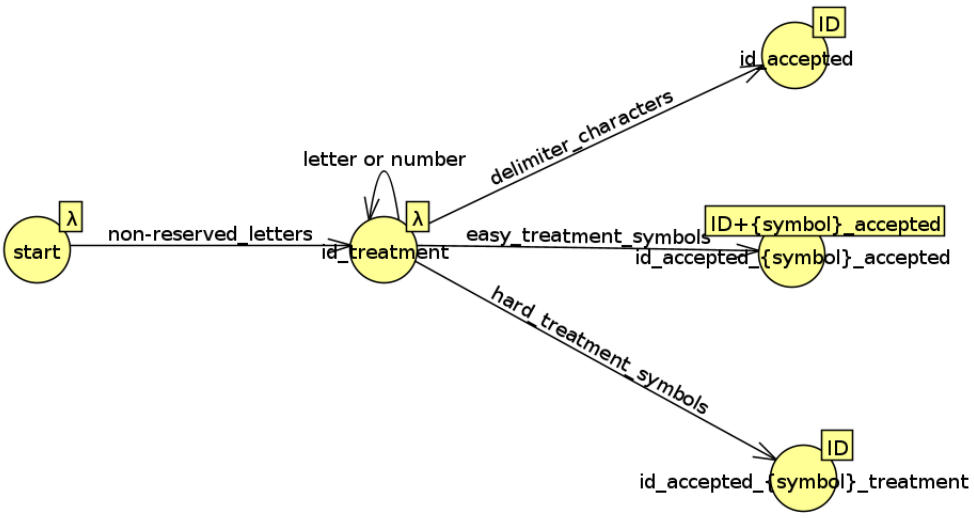

Figura 2: Representação de Tratamento de Identificadores

4.3 Representação de Tratamento de Números

A máquina que abstrai o tratamento de números está apresentada na Figura 3, elá é muito similar a máquina que representa o tratamento de identificadores, porém há uma questão a ser levantada: E se tiver uma letra no meio do número? Resolvi esse problema fazendo com que caso encontre uma letra, haverá a criação de um novo estado e esse novo estado terá as mesmas transições de `id_treatment` e receberá um output `NUMBER` na `output_table`.

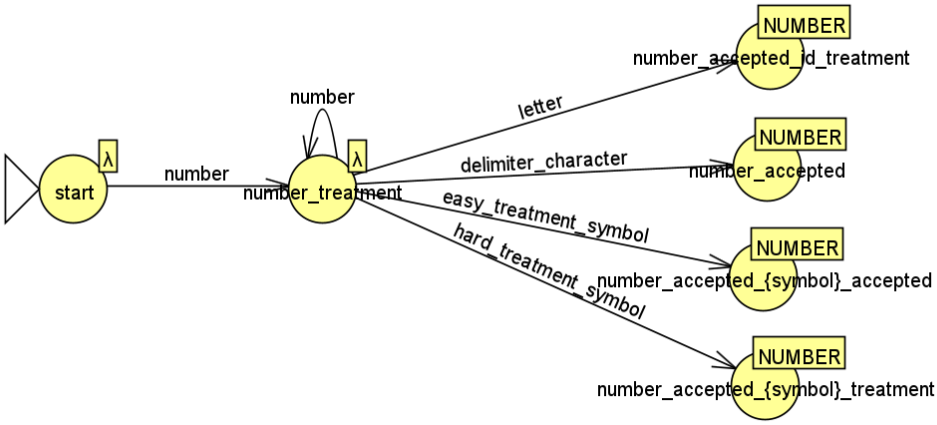

Figura 3: Representação de Tratamento de Números

4.4 Tratamento de Símbolos

Para o tratamento de símbolos, houve a necessidade de eu separar os símbolos em duas categorias, os símbolos de tratamento difícil, apresentados no conjunto `hard_treatment_symbols` na seção 3.1 e `easy_treatment_symbols`.

O motivo da necessidade de separação entre os símbolos seria pela forma de verificar se um símbolo é aceito, ou pela necessidade de verificar a composição desse símbolo em outro estado. A diferenciação entre eles é mostrada na Figura 4, o ponto e vírgula seria um símbolo de tratamento fácil, pois há a necessidade de apenas um estado para tratá-lo enquanto para o ponto de interrogação, considerado símbolo de tratamento difícil, há a necessidade de um estado intermediário

chamado `!_treatment` para tratá-lo.

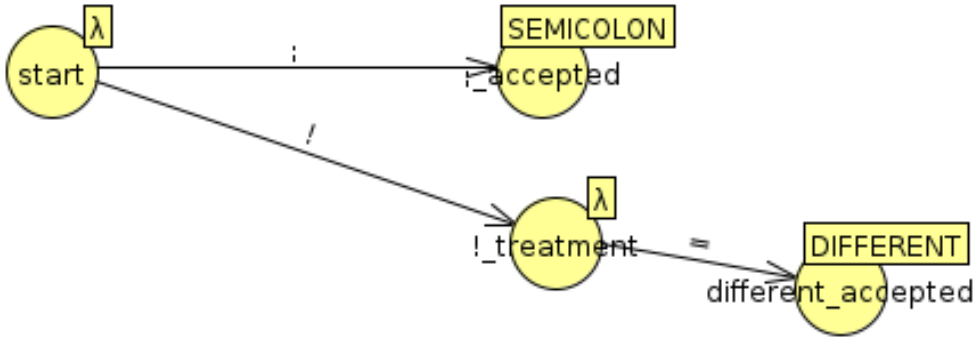

Figura 4: Exemplos de Símbolos Fáceis e Difíceis

4.4.1 Símbolos de Tratamento Fácil

Como já citado, os símbolos de tratamento fácil são aqueles que não precisam verificar qual o próximo caractere, sendo assim, ao verifica-los basta enviá-los para o estado que imprime seu respectivo símbolo. Observe na Figura 5 a generalização do tratamento de tal símbolos. Lembre-se que todos os esses estados de `{symbol}_accepted` receberão uma cópia das transições do `start`.

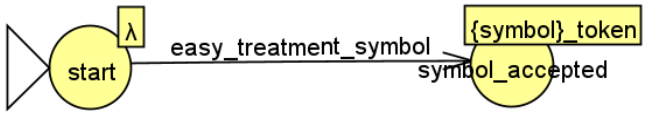

Figura 5: Generalização para Tratamento de Símbolos Fáceis

4.4.2 Símbolos de Tratamento Difícil

O problema com essas símbolos que caracterizei como símbolos de tratamento difícil é como todos eles dependem de saber qual o próximo do próximo símbolo da cadeia para determinar o quê imprimir na saída. Sendo assim, cada um deles precisa de um tratamento especial.

Observe na Figura 6 o tratamento para o símbolo de barra, ao mesmo tempo que tenho que verificar se há um comentário, há a necessidade de verificar se tenho que apenas aceitar o símbolo, e para isso tenho que fazer criar todos esses novos estados e concatenar o estado e o output de `/_accepted` para cada um dos casos. `comment_accepted` e `/_accepted` recebem uma cópia das transições de `start`.

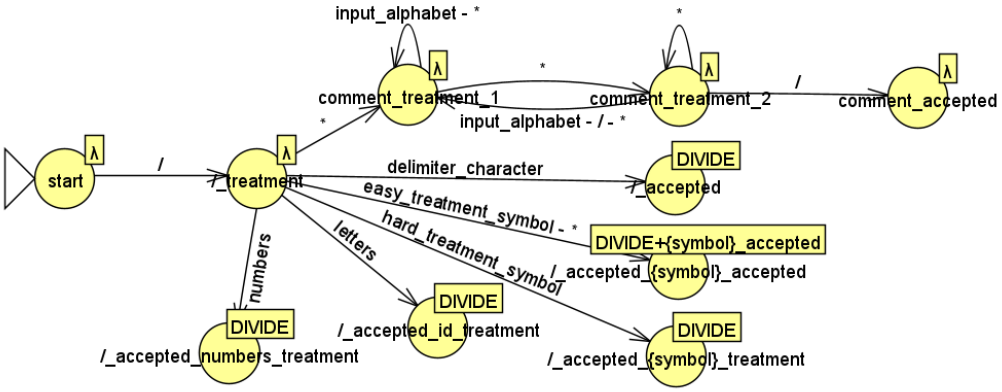

Figura 6: Tratamento Especial para o Símbolo /

O tratamento para o símbolo =, apresentado na Figura 7, acabará por ser muito similar ao tratamento do símbolo /, porém sem a verificação para comentário, e ao invés disso terá a verificação de símbolo de comparação para a linguagem C-, representado por ==. No estado `=_accepted_xxx`, comparando com o tratamento de /, imagine que todos estados que usam `/_accepted` devem estar ali também, só que com o símbolo de atribuição, claro temos de excluir o símbolo = de `hard_treatment_symbols` nesse caso.

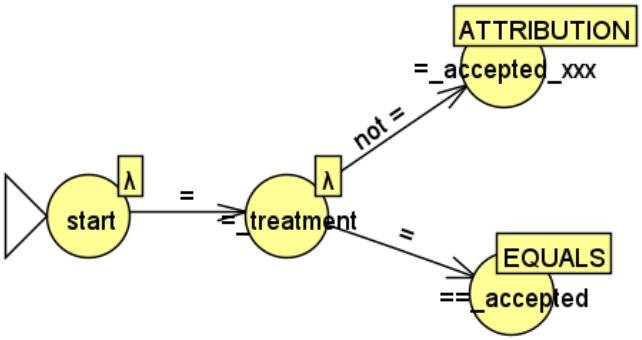

Figura 7: Tratamento Especial para o Símbolo =

Para o < e o >, o tratamento deles são muito semelhantes entre si, observe na imagem 8. Ambos estados `<_accepted_xxx` e `>_accepted_xxx` possuem o mesmo tratamento já citado para o símbolo =.

Já para o símbolo !, já mostrei na Figura 4 como ele deve ser tratado, apenas gostaria de adicionar que há a necessidade de uma cópia das transições do `start` exceto a transição do símbolo =.

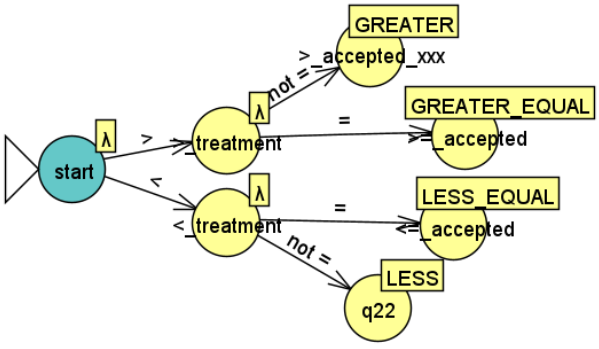

Figura 8: Tratamento Especial para os Símbolos < e >

4.5 Tratamento de Palavras Reservadas

Na Seção 4.2 eu citei que para tratar identificadores eu utilizei apenas as letras que não estavam no conjunto de `reserved_letters` mas estavam no conjunto `letters`, então nos resta tratar as letras iniciais de palavras reservadas. Observe na Figura 9 o tratamento para a palavra reservada `else`, queria ressaltar aqui a quantidade absurda de transições e estados necessários para fazer o tratamento de uma letra que é o início de uma palavra reservada.

Todos os estados entre o reconhecimento da letra no `start` e até o estado final, que no nosso exemplo é o `else_else`, necessitam do mesmo tratamento apresentado na imagem. Já no estado final, eu tenho que fazer algo similar à quando aceito um símbolo de tratamento difícil:

1. Um estado para caracteres delimitadores, que será chamado da concatenação da palavra reservada mais `_accepted`, que também receberá o output como a capitalização da palavra.
2. Um estado para símbolos de tratamento fácil, que receberá no nome da concatenação o nome do estado de aceitação mais `_symbol_accepted`. Além disso tal estado também recebe a concatenação do output do estado de aceitação mais o output de `_symbol_accepted`.
3. Um estado para símbolos de tratamento difícil, que receberá no nome da concatenação o nome do estado de aceitação mais `_symbol_treatment`. Além disso tal estado também recebe o output do estado de aceitação da palavra.
4. O estado que trata a recorrência de números no fim da palavra reservada é um caso especial, eu poderia considerar ou desconsiderar o número como algo que é conjunto da palavra reservada. No meu caso eu optei por desconsiderar, o quê irá gerar casos onde a palavra reservada terminou de ser gerada e já será impresso o *token* referente à ela e irá para o caso de tratamento de números, caso eu não tivesse feito isso, a palavra reservada não seria impressa e iria diretamente para o tratamento de identificadores.
5. Para tratar letras após o estado final, estou considerando que não será impresso o *token* da palavra, e que haverá um novo estado `id_treatment` e que haverá uma transição para esse estado, e esse estado receberá uma cópia de todas as transições do `id_treatment` original.

O exemplo que usei aqui foi para a palavra reservada `else`, porém, essas ”regras”foram aplicadas para todas as palavras reservadas que descrevi na Tabela 1.

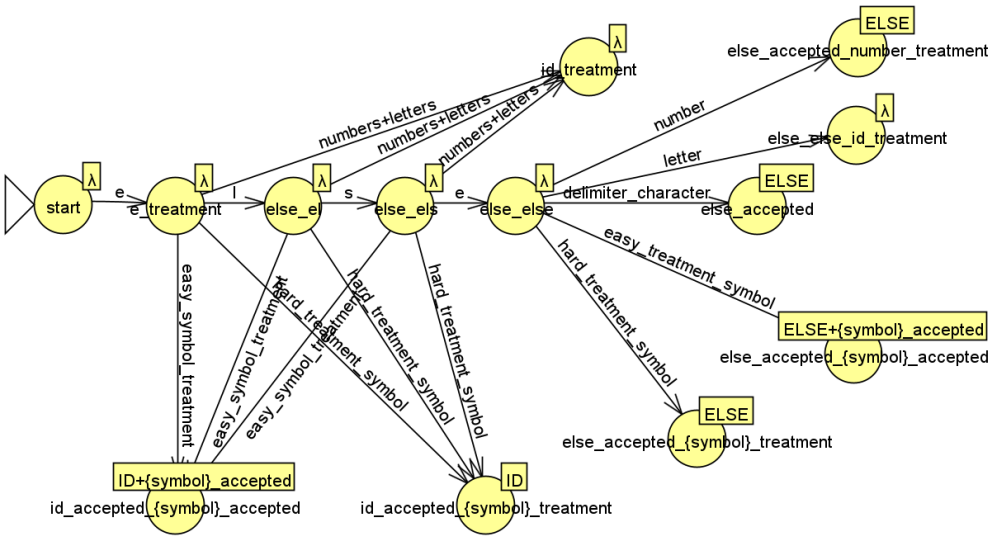

Figura 9: Exemplo de Tratamento de Palavras Reservadas

5 Considerações finais

Fazer esse analisador léxico ao mesmo tempo que me proporcionou obtenção de muito conhecimento, também me deu muita dor de cabeça, são tantas transições e estado que é impraticável fazer a máquina de Moore por inteiro num simulador tipo o que usei (JFLAP), devido à essa dificuldade eu diria que é melhor buscar fazer um *lexer* de outra forma.

Escrevendo esse trabalho, optei por não incluir muitas coisas sobre o código, pois, isso não deixaria ele mais fácil de ler, além de também aumentar significativamente o tamanho do trabalho. Optei por tentar mostrar a minha linha de raciocínio ao resolver esse problema que é fazer um *lexer* com máquina de Moore. A forma como combati esse problema foi lidando com as coisas como se fossem conjuntos matemáticos, as quais também poderiam ser vistas como Diagramas de Venn para facilitar a visualização mental.

Outro motivo que eu poderia dizer ser um motivo de eu ter um pouco de receio de mostrar os meus códigos, era a minha falta de experiência com a linguagem Python. Isso acabou por causar múltiplas refatorações de código, e que eu ainda acredito que, depois de escrever esse trabalho, eu conseguiria fazer um código ainda melhor.

Referências

LOUDEN, Kenneth C. 2004. *Compiladores: Princípios e práticas*. São Paulo, SP: Thomson 1st edn.

Paulo Blauth Menezes. 2011. *Linguagens formais e autômatos - v3*. Porto Alegre, RS: UFRGS 6th edn.