Parser para Expressões Aritméticas utilizando Autômato com Pilha

Rubens Antonio da Silva Filho[1]

[1]Universidade Tecnológica Federal do Paraná (UTFPR)

[2]R. Rosalina Maria Ferreira, 1233

1. Introdução

Um autômato é um modelo matemático composto por máquinas de estados finitas. Dado uma alfabeto de entrada, estados e uma entrada, ele é capaz de julgar a entrada como aceita ou não.

Podemos ter um autômato finito determinístico, tendo como principal característica a obrigatoriedade em haver uma, e apenas uma, transição para cada símbolo do alfabeto em cada estado existente. Ou seja, dado o alfabeto de entrada como A = {a, b} e os estados Q = {q0, q1}, é necessário que no estado q0, tenha uma transição quando for a, e uma transição quando for b; o mesmo ocorre em q1. Na figura 1 temos um exemplo de autômato finito determinístico para o alfabeto de entrada e estados citados.

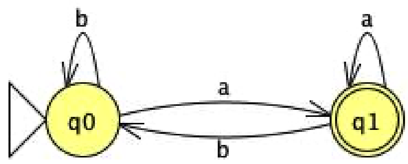


Figura 1. Exemplo de autômato finito determinístico. Fonte: Autoria própria.

Já um autômato finito não determinístico não há obrigatoriedade de transições. Nele é podemos ter um estado que não contenha uma transição para um símbolo do alfabeto, e também pode conter mais de uma transição para a mesmo símbolo. Outra característica, é que nele podemos ter transições vazias, ou seja, transições que não irão 'gastar' um símbolo da entrada. Com isso, temos que o autômato finito não determinístico possui características de paralelismo, ou seja, uma execução com uma cadeia de entrada pode estar em diversos estados ao mesmo tempo. Na figura 2 temos um exemplo de autômato finito não determinístico.

Até então os autômatos possuíam memória limitada ao caminho que era percorrido entre os estados. Agora, com o autômato com pilha é possível tem uma memória auxiliar. Nele é adicionado uma nova condição, que é o símbolo que está no topo da pilha. Por exemplo, se estiver no estado *q0* com símbolo de entrada *a* e com o símbolo no topo da pilha *Z* vai para o estado *q2*, já se for com o símbolo no topo da pilha *V*, vai para o estado *qF*. Veja o exemplo na figura 3.

Uma das principais utilidades de autômatos a pilha é reconhecer linguagens livre de contexto.

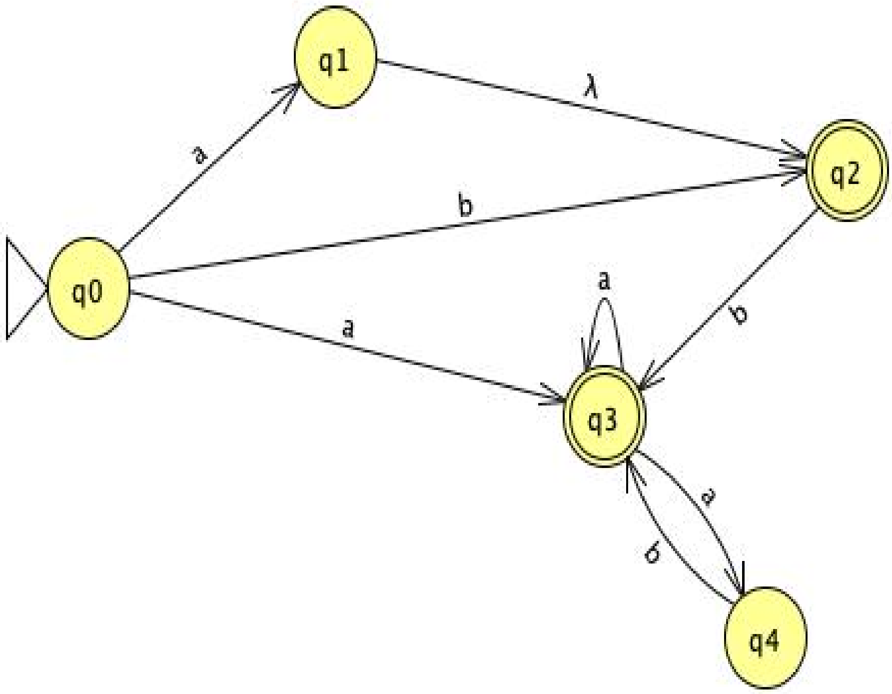


Figura 2. Exemplo de autômato finito não determinístico. Fonte: Autoria própria.

2. Objetivo

O objetivo deste trabalho é implementar um Autômato com Pilha para realizar o *parsing* da gramática vista na figura 4.

Já no *Grammophone*, utilizando o algoritmo SLR(1) obtemos a seguinte *Parsing Table*, como visto na figura 5.

Agora, no *JFlap* obtemos a árvore gerada com a gramática para a entrada de exemplo*id=num*num*. Veja na figura 6.

3. Implementação

A primeira parte da implementação foi transformar a gramática em um autômato com pilha no *JFlap*, como visto na figura 7

A segunda parte da implementação, foi desenvolver um código em *Python* que implementa um autômato com pilha. Para o auxiliar e simplificar o desenvolvimento, foi implementado 4 classes, *StackTransition*, *Transition*, *State* e *Automata*.

3.1. *StackTransition*

A classe *StackTransition* é responsável pela transição entre os estados do autômato de acordo com símbolo no topo da pilha. Ela é composta de uma *stackCondition*, símbolo no topo da pilha que é necessário para que a transição ocorra; *destinationState*, estado de destino da transição; e *insertStack*, símbolos que serão inseridos na pilha quando ocorrer a transição.

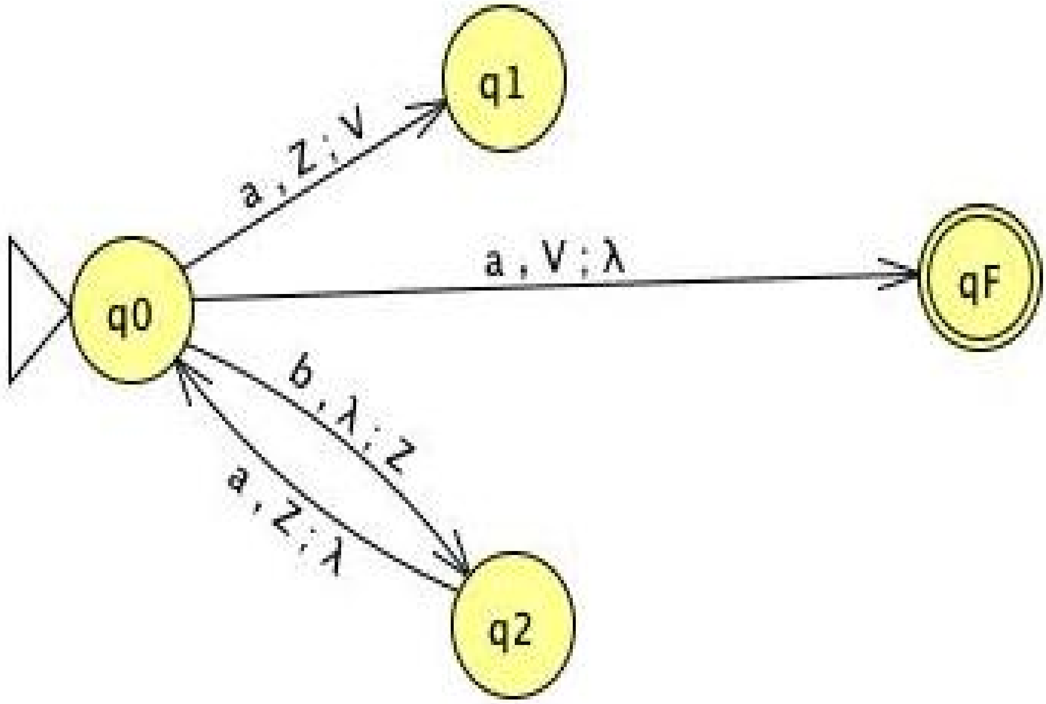


Figura 3. Exemplo de autômato com pilha. Fonte: Autoria própria.

3.2. *Transition*

A classe *Transition* é responsável pela transição entre os estados do autômato de acordo com o símbolo da entrada. Ela é composta de uma *condition*, símbolo do alfabeto que é necessário para que a transição ocorra; e *stackTransitions*, transições de acordo com o símbolo no topo da pilha.

3.3. *State*

A classe *State* é responsável pelos estados do autômato. Ela é composta de um *name*, nome do estado; e *transitions*, transições que ocorrem quando o autômato se encontra naquele estado.

3.4. *Automata*

Por fim, temos a classe *Automata*, ela é responsável por criar o autômato que iremos utilizar. Ela é composta de um *inputAlphabet*, alfabeto de entrada; *stackAlphabet*, alfabeto da pilha; *initialState*, estado inicial do autômato; e *endsStates*, estados de aceitação do autômato, quando chegar nele, significa que a entrada é aceita pela gramática.

Dentro desta classe temos a função *getNPDAAutomata()*, é nela aonde o autômato é em si criado. Mais detalhes da criação do autômato, será dado na seção 3.5.

3.5. Biblioteca

Para a criação do autômato foi utilizado a biblioteca *automata-lib*, disponível em https://github.com/caleb531/automata. Nela criamos um objeto *NPDA*, contendo um vetor com o nome dos estados, um vetor com o alfabeto de entrada, um vetor com o alfabeto

```
S -> E
S -> V = E
E -> E + T
T -> T * F
F -> ( E )
E -> E - T
E -> T
T -> T / F
T -> F
F -> V
V -> id
F -> num
```

Figura 4. Gramática utilizada para o *parsing*. Fonte: Autoria própria.

da pilha, um dicionário com as transições, o nome do estado inicial, o símbolo inicial da pilha, e os estados finais (aceitação).

Como já citado na seção 3, foi criado 4 classes para auxiliar no desenvolvimento, o objetivo final delas, é criar o objeto *NPDA*. Isso ocorre dentro da função *getNPDAAutomata()* da classe *Automata*.

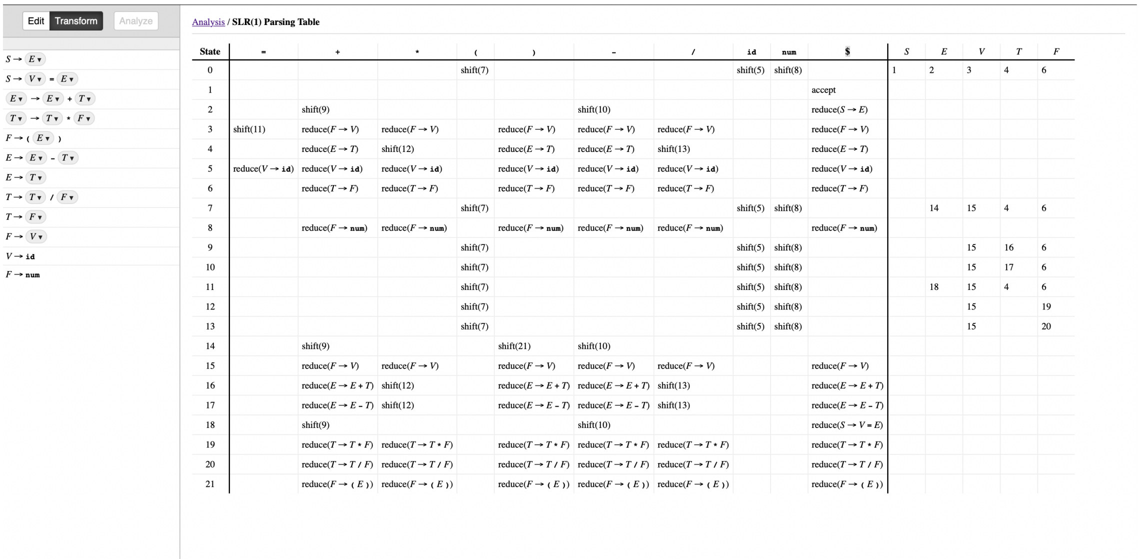
Nesta função, um *for* percorre todos os estados obtendo seu nome e adicionado em um vetor de nomes.

```
statesName = []
for state in self.states:
    statesName.append(state.name)
```

Depois a função *getDict()* irá criar o dicionário com as transições. O primeiro *for* percorre todos os estados do autômato, já o segundo *for* percorre todas as transições do autômato. Por fim, o terceiro *for* percorre as transições aonde elas realmente acontecem, que é de acordo com o símbolo que esta no topo da pilha.

```
for state in self.states:
    transitionsState = {}
    for transition in state.transitions:
        stackTransitionsState = {}
        stackTransitionsStateAtual = set()

        stackCondition = transition.stackTransitions[0].stackCondition
        for stackTransition in transition.stackTransitions:
            if stackCondition != stackTransition.stackCondition:
                stackTransitionsState[stackCondition] =
                ↪  stackTransitionsStateAtual
                stackCondition = stackTransition.stackCondition
                stackTransitionsStateAtual = set()
```



State	=	+	*	(	)	-	/	id	num	$	S	E	V	T	F
0				shift(7)				shift(5)	shift(8)		1	2	3	4	6
1										accept					
2		shift(9)				shift(10)				reduce(S → E)					
3	shift(11)	reduce(F → V)	reduce(F → V)		reduce(F → V)	reduce(F → V)	reduce(F → V)			reduce(F → V)					
4		reduce(E → T)	shift(12)		reduce(E → T)	reduce(E → T)	shift(13)			reduce(E → T)					
5	reduce(V → id)	reduce(V → id)	reduce(V → id)		reduce(V → id)	reduce(V → id)	reduce(V → id)			reduce(V → id)					
6		reduce(T → F)	reduce(T → F)		reduce(T → F)	reduce(T → F)	reduce(T → F)			reduce(T → F)					
7				shift(7)				shift(5)	shift(8)			14	15	4	6
8		reduce(F → num)	reduce(F → num)		reduce(F → num)	reduce(F → num)	reduce(F → num)			reduce(F → num)					
9				shift(7)				shift(5)	shift(8)				15	16	6
10				shift(7)				shift(5)	shift(8)				15	17	6
11				shift(7)				shift(5)	shift(8)			18	15	4	6
12				shift(7)				shift(5)	shift(8)				15		19
13				shift(7)				shift(5)	shift(8)				15		20
14		shift(9)			shift(21)	shift(10)									
15		reduce(F → V)	reduce(F → V)		reduce(F → V)	reduce(F → V)	reduce(F → V)			reduce(F → V)					
16		reduce(E → E + T)	shift(12)		reduce(E → E + T)	reduce(E → E + T)	shift(13)			reduce(E → E + T)					
17		reduce(E → E - T)	shift(12)		reduce(E → E - T)	reduce(E → E - T)	shift(13)			reduce(E → E - T)					
18		shift(9)				shift(10)				reduce(S → V = E)					
19		reduce(T → T * F)	reduce(T → T * F)		reduce(T → T * F)	reduce(T → T * F)	reduce(T → T * F)			reduce(T → T * F)					
20		reduce(T → T / F)	reduce(T → T / F)		reduce(T → T / F)	reduce(T → T / F)	reduce(T → T / F)			reduce(T → T / F)					
21		reduce(F → (E))	reduce(F → (E))		reduce(F → (E))	reduce(F → (E))	reduce(F → (E))			reduce(F → (E))					

Figura 5. ***Parsing Table* obtido para a gramática através do *Grammophone*. Fonte: Autoria própria.**

```
            ↪  stackTransitionsStateAtual.add((stackTransition.destinationState.name,
            ↪  stackTransition.insertStack))

        stackTransitionsState[stackTransition.stackCondition] =
         ↪  stackTransitionsStateAtual
        transitionsState[transition.condition] = stackTransitionsState

    dic[state.name] = transitionsState
```

Como podemos ter mais de uma transição quando tem o mesmo símbolo de entrada e mesmo símbolo no topo da pilha, foi necessário utilizar um *set()* dentro do terceiro *for*. Dentro do *if* mais interno é verificado se já terminou de adicionar as transições com a mesma condição de topo da pilha, quando termina, adiciona esse *set()* a transição de acordo com o símbolo de entrada.

Ao fim, teremos um dicionário similar à {estado: {simbolo_entrada: {simbolo_pilha: (proximo_estado, simbolos_adicionar_pilha) }}}.

3.6. Estados

Nesse autômato temos apenas 4 estados.

Os estados *q0* e *q1* irão iniciar a pilha com *S* e *$*, respectivamente. Depois irá par ao estado *qM*, aonde ficará em *loop* adicionando e removendo os símbolos na pilha. Quando a pilha estiver vazia, ou seja, encontrar o primeiro elemento *$*, irá para o estado de aceitação *qF* e a entrada será aceita pela gramática.

4. Exemplos

Como primeiro exemplos, temos a entrada como **id=id*num+id**, sendo ela **aceita** pela gramática. Outra entrada possível é **numm**, sendo ela **rejeitada** pela gramática.

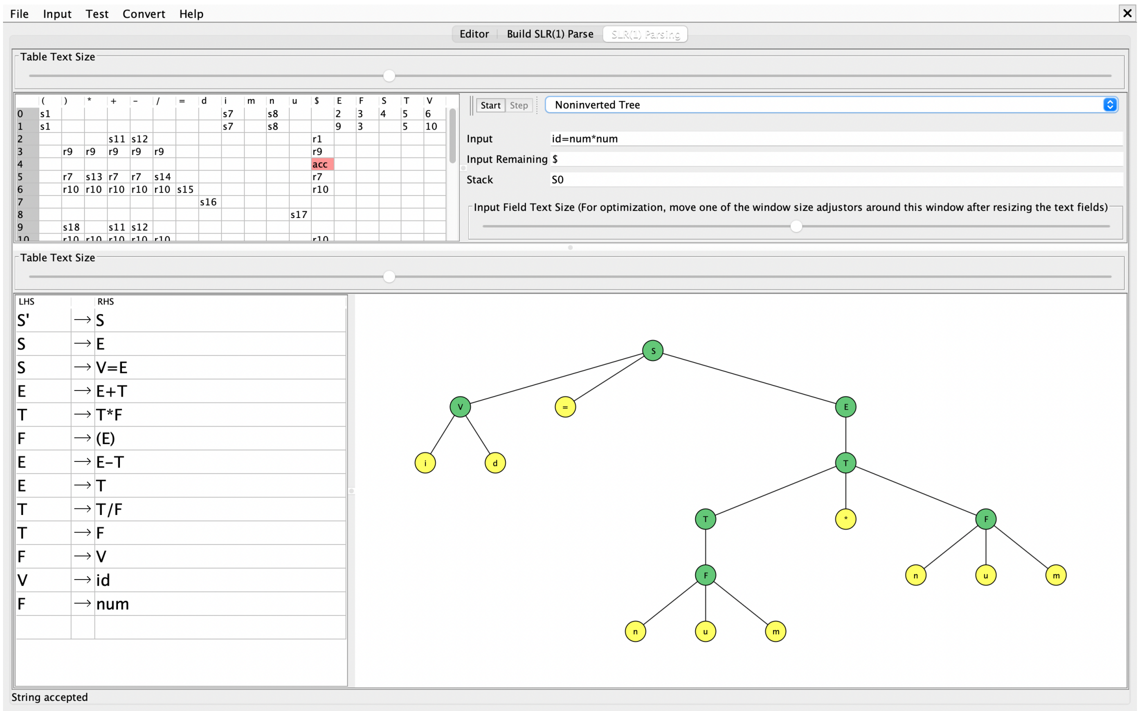


Figura 6. Exemplo de árvore obtida com a gramática através do *JFlap*.

5. Conclusão

Ao fim do desenvolvimento temos um autômato com pilha que é capaz de dizer se uma entrada é aceita ou não pela gramática dada na figura 4.

Por fim, o código fonte e como utilizar, está disponível no GitHub em https://github.com/RubinhoSilva/parser_expressoes_aritmeticas.

Referências

Menezes, P. (2009). *Linguagens Formais e Autômatos: Volume 3 da Série Livros Didáticos Informática UFRGS*. Bookman Editora.

Rodger, S. H. (2006). *JFLAP: An Interactive Formal Languages and Automata Package*. Jones and Bartlett Publishers, Inc., USA.

Sipser, M. (2007). *Introdução à Teoria da Computação: Tradução da 2ª edição norte-americana (trad. Ruy José Guerra Barreto de Queiroz)*. Thomson Learning, São Paulo.

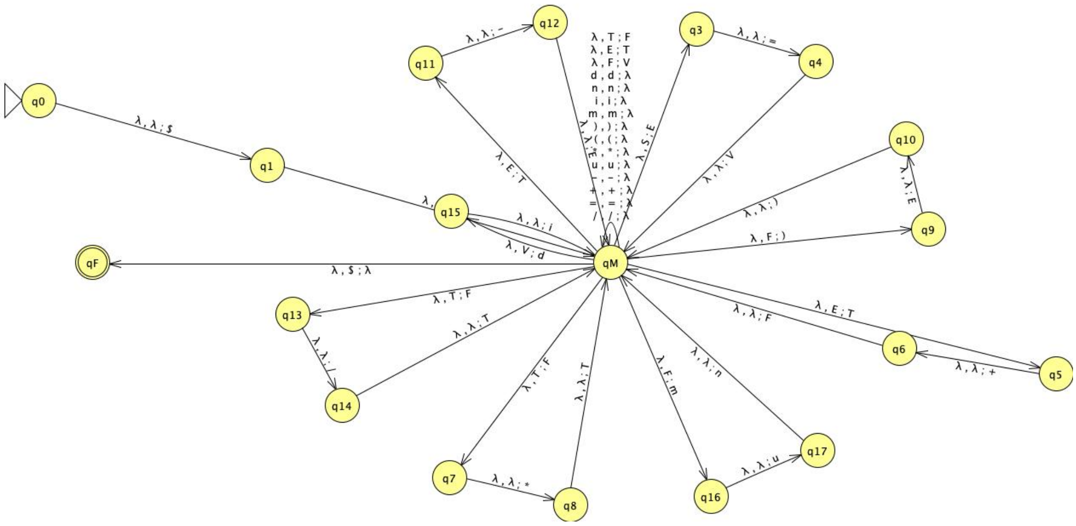


Figura 7. Implementação do Autômato com Pilha no *JFlap*. Fonte: Autoria própria.