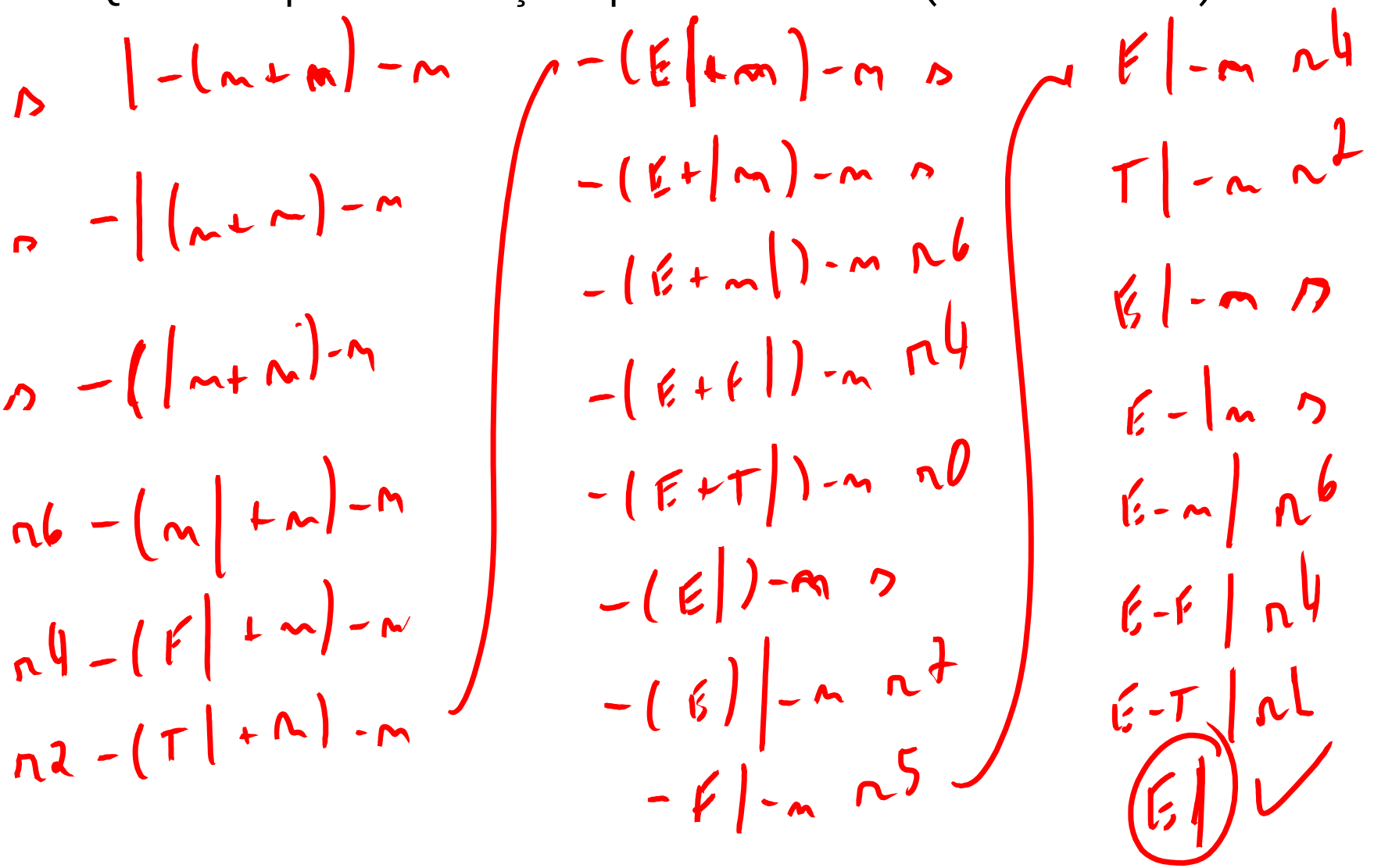
Shift e reduce

- Shift: move o foco uma posição à direita
  - $A\ B\ C \mid x\ y\ z \Rightarrow A\ B\ C\ x \mid y\ z$ é uma ação *shift*
- Reduce: reduz o que está imediatamente à esquerda do foco usando uma produção
  - Se $A \rightarrow x\ y$ é uma produção, então $C\ b\ x\ y \mid i\ j\ k \Rightarrow C\ b\ A \mid i\ j\ k$ é uma ação *reduce* $A \rightarrow x\ y$
- Acontece um erro sintático quando não se pode tomar nenhuma das duas ações, e reconhecemos a entrada quando o chegamos a $S \mid$, onde S é o símbolo inicial

Exercício

- Qual a sequência de ações para a cadeia - (num + num) - num



```
0 E -> E + T
1 E -> E - T
2 E -> T
3 T -> T * F
4 T -> F
5 F -> - F
6 F -> num
7 F -> ( E )
```

Implementação

- O que está à esquerda do foco pode ser implementado usando uma pilha
- O foco é o topo da pilha mais uma posição na entrada
- A ação de *shift* empilha o próximo token e incrementa a posição
- A ação de *reduce* $A \rightarrow w$:
  - Desempilha |w| símbolos (que devem formar w, ou a redução estaria errada)
  - Empilha A

Escolhas e conflitos

- As técnicas de análise ascendente usam a análise shift-reduce como base, a diferença é a estratégia que elas usam para escolher entre ações de shift e reduce

- Problemas na gramática (como ambiguidade), ou limitações da técnica específica adotada, pode levar a *conflitos*

  - Um conflito *shift-reduce* é quando o analisador não tem como decidir entre uma (ou mais) ações de shift e uma ação reduce, o que normalmente acontece por limitações da técnica escolhida

  - Um conflito *reduce-reduce* é quando o analisador não tem como decidir entre duas ou mais ações de reduce, o que normalmente é um *bug* na gramática

Handles

- Mas como o analisador escolhe qual ação deve tomar?
- Uma escolha errada pode levar a um beco sem saída, mesmo para uma entrada válida
- Intuição: devemos reduzir se a redução vai nos levar um passo para trás em uma derivação mais à direita, e “shiftar” caso contrário
- Em um passo de uma derivação mais à direita $uAw \longrightarrow uvw$, a cadeia uv é um *handle* de uvw
- Queremos reduzir quando o conteúdo da pilha for um handle

Reconhecendo um handle

- Não existe um algoritmo infalível e eficiente para reconhecer um handle no topo da pilha shift-reduce

- Mas existem boas heurísticas, que sempre encontram os handles corretamente para certas classes de gramáticas

  - LR(0), SLR, LALR, LR(1), LR(k), etc...

- A maioria reconhece (ou não) um handle examinando a pilha e o próximo token da entrada (o *lookahead*)

Prefixos viáveis

- Embora não exista um algoritmo exato e eficiente para reconhecer handles, existe um para *prefixos* de handles
- Um prefixo de um handle é um *prefixo viável*
- Enquanto o analisador tem um prefixo viável na pilha, ainda não foi detectado um erro sintático
- O conjunto de prefixos viáveis de uma gramática é uma *linguagem regular*!
- Isso quer dizer que podemos construir um autômato finito para dizer se o que está na pilha é um prefixo viável ou não

Itens LR(0)

- Para construir um autômato que reconhece prefixos viáveis usamos o conceito de *itens LR(0)*

- Um item LR(0) de uma gramática é uma produção da gramática com uma marca no seu lado direito

- Por exemplo, os itens para a produção F → (E) são:

```
F -> . ( E )
F -> ( . E )
F -> ( E . )
F -> ( E ) .
```

- Uma produção vazia tem um único item LR(0), e itens com a marca no final são *itens de redução*

Construindo o autômato não-determinístico

- Para construir o autômato, primeiro adicionamos um novo símbolo inicial S’ e uma produção S’ → S

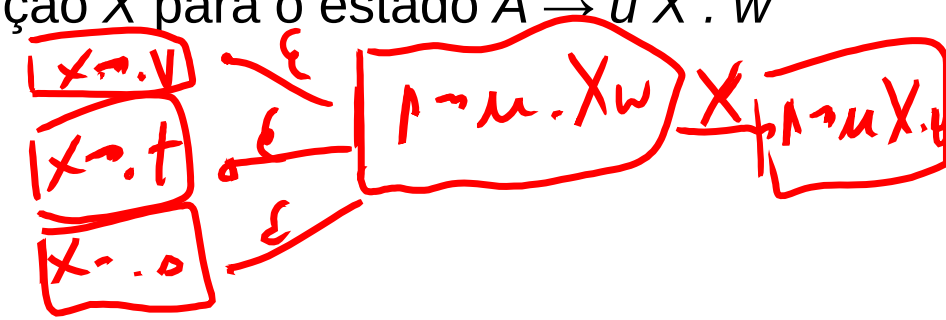
- O item S’ → . S é o *item inicial*, e ele dá o estado inicial do autômato

- Cada um dos itens da gramática é um estado do autômato

- Um item $A \rightarrow u \,.\, x\, w$, onde x é um terminal, tem uma transição x para o estado $A \rightarrow u\, x \,.\, w$, lembre que tanto u quanto w podem ser vazios!

- Um item $A \rightarrow u \,.\, X\, w$, onde X é um não-terminal, tem transições ε para todos os itens iniciais do não-terminal X, e uma transição X para o estado $A \rightarrow u\, X \,.\, w$

- Todos os estados do autômato são finais!



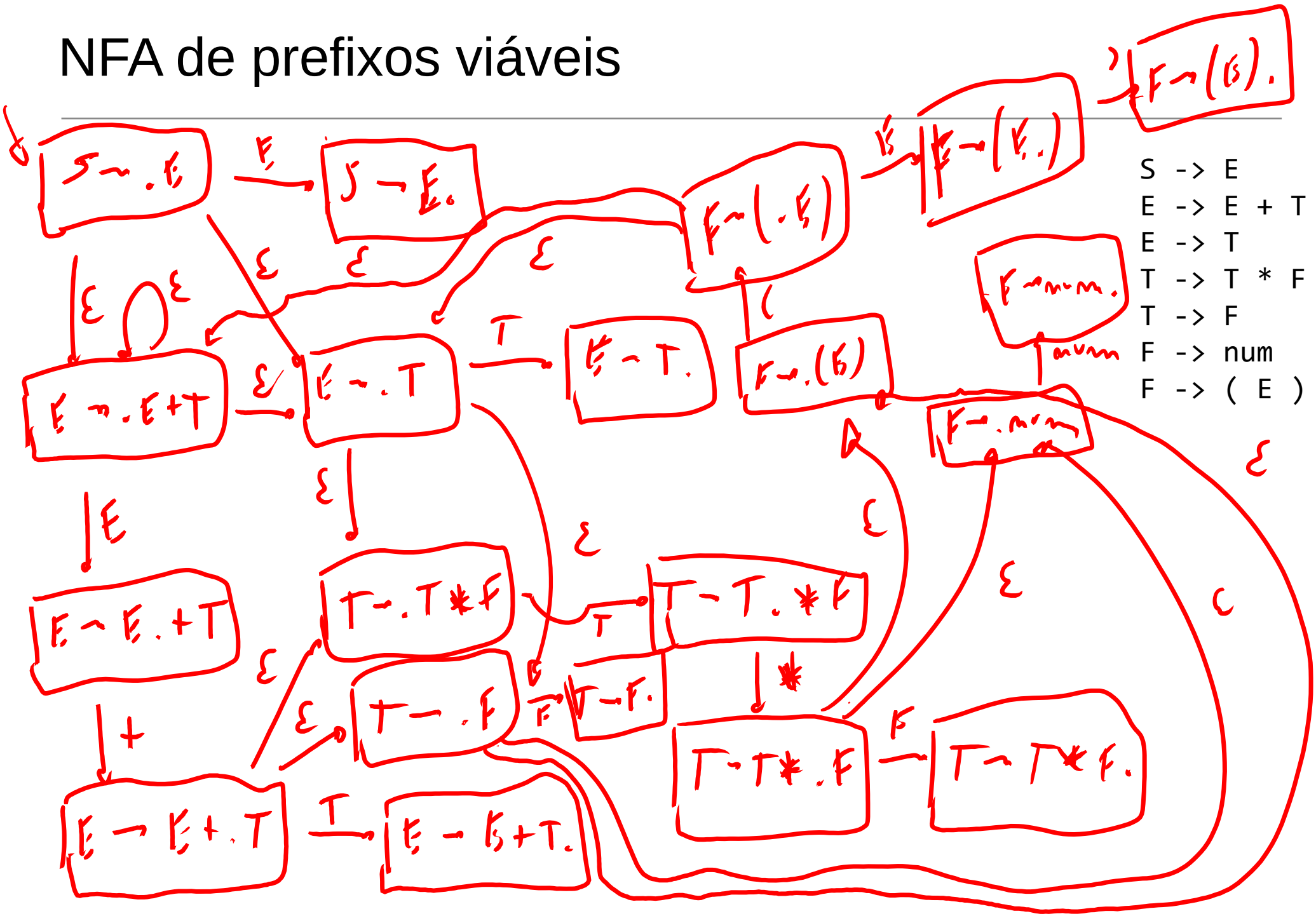
Gramática de Expressões

- Vamos usar como exemplo uma gramática de expressões simplificada:

```
S -> E
E -> E + T
E -> T
T -> T * F
T -> F
F -> num
F -> ( E )
```

- Vamos construir o NFA de prefixos viáveis dessa gramática

NFA de prefixos viáveis



De NFA para DFA

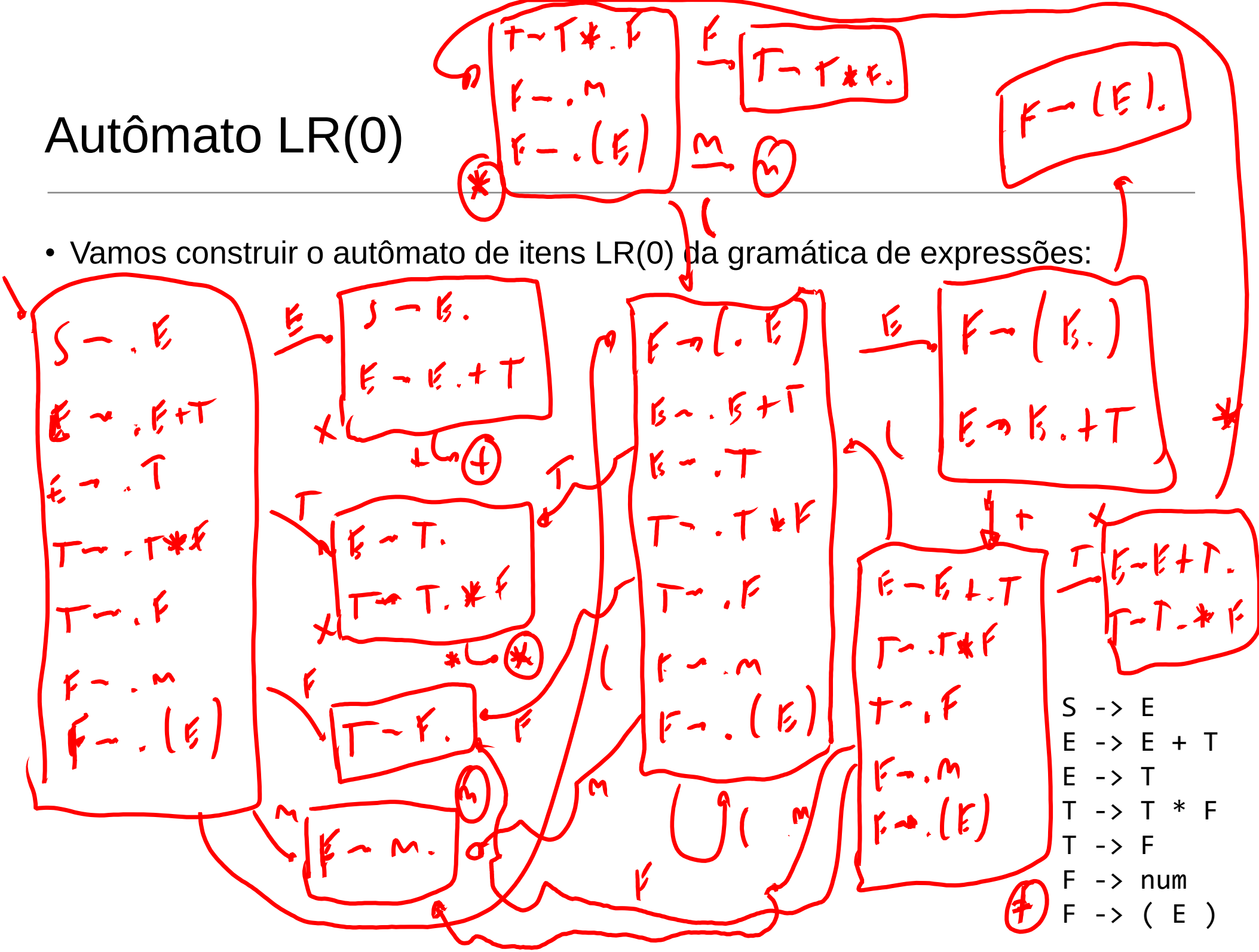
- Podemos converter o NFA para um DFA usando o algoritmo usual
- Isso nos dá um autômato reconhecedor de prefixos viáveis
- Esse autômato é a base da técnica de análise LR(0):
  - Se o autômato leva a um estado com um único item de redução, reduza por aquela produção
  - Senão faça um *shift* e tente de novo
  - Se o autômato chegou em $S' \rightarrow S\ .$ e chegamos no final da entrada a entrada foi aceita

Construção direta do DFA

- Na prática construimos diretamente o DFA de itens LR(0) para prefixos viáveis
- Aplicamos a um estado uma operação de *fecho*, que é equivalente ao fecho-ε do NFA
  - Se o estado tem um item $A \rightarrow u \; . \; X \; w$, onde X é um não-terminal, então inclua todos os itens iniciais de X
  - Faça isso até nenhum outro item ser incluído
- Sobrarão apenas as transições em terminais e não-terminais, com no máximo uma para cada terminal ou não-terminal saindo de cada estado

Autômato LR(0)

- Vamos construir o autômato de itens LR(0) da gramática de expressões:



```
S -> E
E -> E + T
E -> T
T -> T * F
T -> F
F -> num
F -> ( E )
```

Um exemplo que funciona

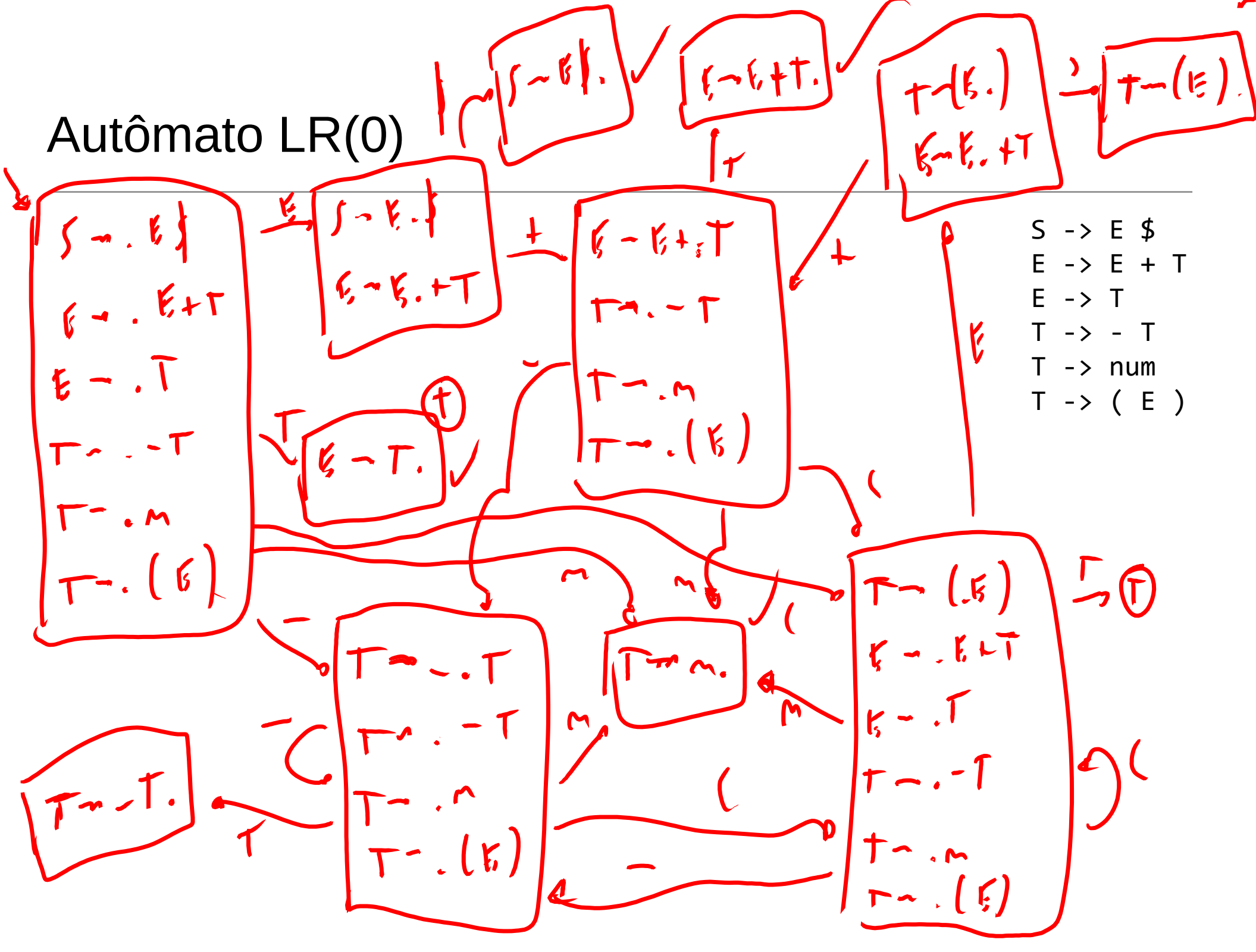
- Todo estado com um item de redução e algum outro item causa conflito LR(0)!

- A técnica LR(0) é bem fraca, mas ainda assim existem gramáticas que ela consegue analisar mas que as técnicas de análise descendente não:

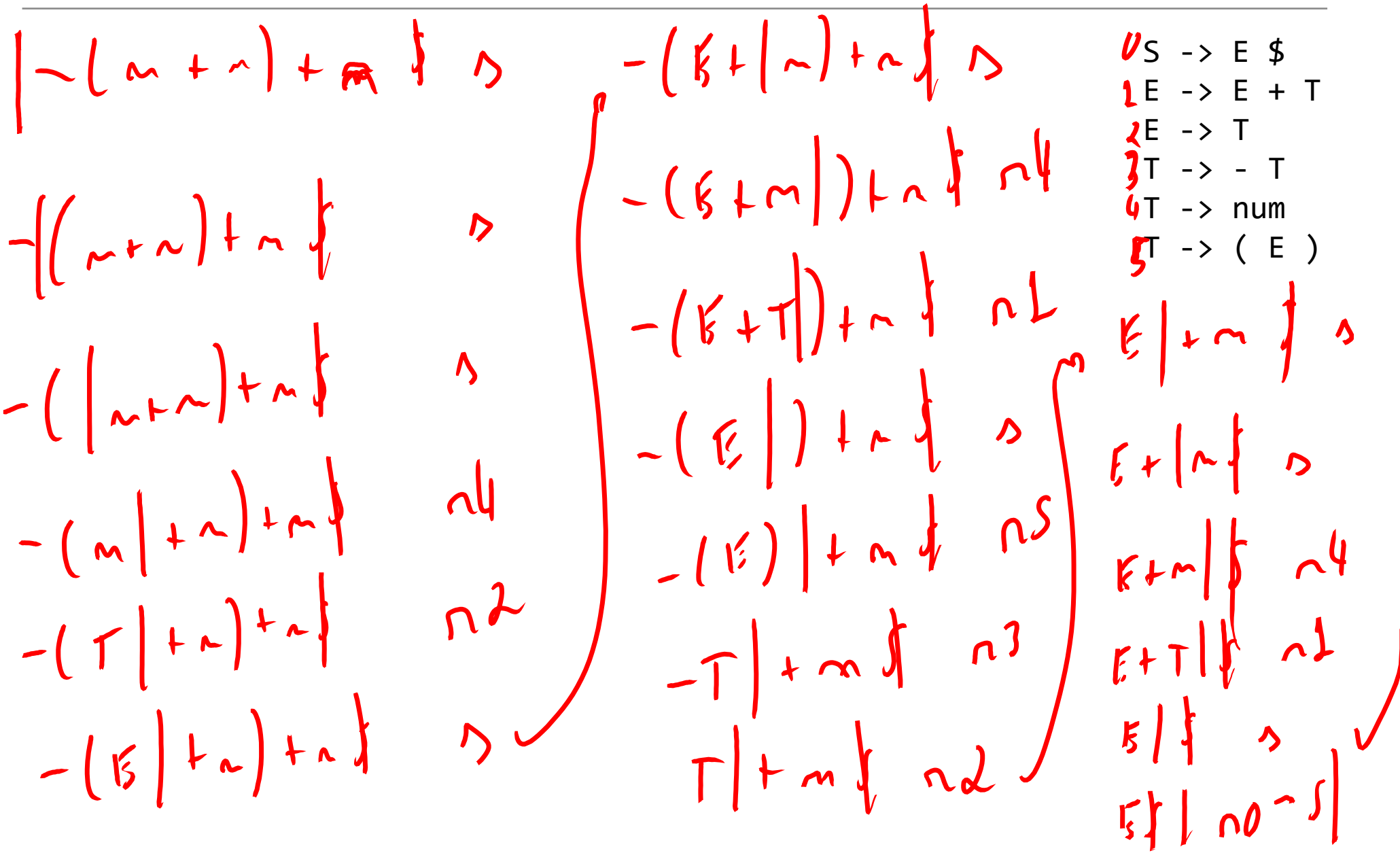
```
S -> E $
E -> E + T
E -> T
T -> - T
T -> num
T -> ( E )
```

- Vamos construir o autômato de itens LR(0) dessa gramática e usá-lo para analisar `- ( num + num ) + num $`

Autômato LR(0)



Analisando uma entrada



Analisando uma entrada