

Compiladores - Análise Ascendente

Fabio Mascarenhas - 2013.2

<http://www.dcc.ufrj.br/~fabiom/comp>

Análise Descendente vs. Ascendente




- As técnicas de análise que vimos até agora (recursiva com retrocesso, recursiva preditiva, LL(1) de tabela) usam a mesma estratégia de análise: a *análise descendente*, ou *top-down*
- Vamos ver agora uma outra estratégia de análise, a *ascendente*, ou *bottom-up*, e as técnicas que a utilizam
- A diferença mais visível entre as duas é a forma de construção da árvore: na análise descendente construímos a árvore de cima para baixo, começando pela raiz, e na ascendente de baixo para cima, começando pelas folhas

Análise Ascendente (Bottom-up)

- A análise ascendente é mais complicada de implementar, tanto para um analisador escrito à mão (o que é muito raro) quanto para geradores
- Mas é mais geral, o que quer dizer que impõe menos restrições à gramática
- Por exemplo, recursão à esquerda e prefixos em comum não são problemas para as técnicas de análise ascendente
- Vamos usar um exemplo que deixa essas vantagens bem claras

Gramática de Expressões

- Vamos usar como exemplo uma gramática de expressões simplificada:

 $E \rightarrow E + T$
 $E \rightarrow E - T$
 $E \rightarrow T$
 $T \rightarrow T * F$
 $T \rightarrow F$
 $F \rightarrow - F$
 $F \rightarrow \text{num}$
 $F \rightarrow (E)$

- Vamos analisar a cadeia $\text{num} * \text{num} + \text{num}$

Reduções

- A análise ascendente analisa uma cadeia através de *reduções*, aplicando as regras da gramática ao contrário:

	num	*	num	+	num	
·	F	*	num	+	num	
·	T	*	num	+	num	
·	T	*	F	+	num	
·	T			+	num	
·	E			+	num	
·	E			+	F	
·	E			+	T	
·	E					

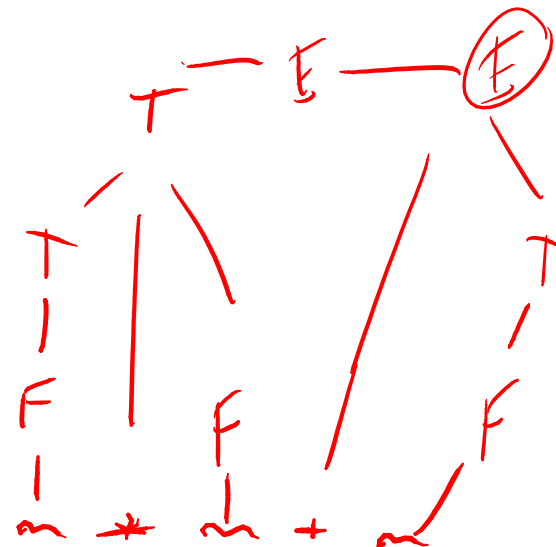
Handwritten notes on the right side of the table:

- $F \rightarrow num$
- $T \rightarrow F$
- $F \rightarrow num$
- $T \rightarrow T * F$
- $F \rightarrow T$
- $F \rightarrow num$
- $T \rightarrow F$
- $E \rightarrow E + T$

- Vamos ler a sequência de reduções de trás para a frente: $E \rightarrow E + T \rightarrow E + F \rightarrow E + num \rightarrow T + num \rightarrow T * F + num \rightarrow T * num + num \rightarrow F * num + num \rightarrow num * num + num$
- Handwritten note: \Rightarrow derivação à direita

Reduções vs derivações

- A sequência de reduções da análise ascendente equivale a uma *derivação mais à direita*, lida de trás pra frente
- Lembre-se que, para uma gramática não ambígua, cada entrada só pode ter uma única derivação mais à direita
- Isso quer dizer que a sequência de reduções também é única! O trabalho do analisador é então achar qual a próxima redução que tem que ser feita a cada passo



Exercício

$$\begin{aligned}
 E &\rightarrow E - (T) \rightarrow E - (F) \rightarrow E - n \rightarrow E - n - F - n \\
 &\rightarrow -(E) - n \rightarrow -(E + T) - n \rightarrow \\
 &\rightarrow -(E + F) - n \rightarrow -(E + n) - n \rightarrow \\
 &\rightarrow -(T + n) - n \\
 &\rightarrow -(F + n) - n \\
 &\rightarrow -(n + n) - n
 \end{aligned}$$

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

$$\begin{aligned}
 &-(n + n) - n \\
 &-(F + n) - n \\
 &-(T + n) - n \\
 &-(E + n) - n \\
 &-(E + F) - n \\
 &-(E + T) - n \\
 &-(E) - n \\
 &-F - n \\
 &F - n \\
 &T - n
 \end{aligned}$$



$$\begin{aligned}
 &F \rightarrow n \\
 &t \rightarrow F \\
 &E \rightarrow T \\
 &F \rightarrow n \\
 &t \rightarrow F \\
 &E \rightarrow E + T \\
 &F \rightarrow (F) \\
 &F \rightarrow -F \\
 &T \rightarrow F \\
 &E \rightarrow T
 \end{aligned}$$

$$\begin{aligned}
 &E \rightarrow n \quad F \rightarrow n \\
 &E \rightarrow F \quad t \rightarrow F \\
 &E \rightarrow T \quad E \rightarrow E + t \\
 &E
 \end{aligned}$$

Análise shift-reduce

- As reduções da análise ascendente formam uma derivação mais à direita de trás para frente
- Tomemos o passo da análise ascendente que leva a string uvw para uXw pela redução usando uma regra $X \rightarrow v$
- O pedaço w da string só tem terminais, pois essa redução corresponde ao passo $uXw \rightarrow uvw$ de uma derivação mais à direita \Rightarrow não existem nt em w
- Isso implica que a cada passo da análise temos um *sufixo* que corresponde ao resto da entrada que ainda não foi reduzido

Análise shift-reduce

- Vamos marcar o foco atual da análise com uma | 
 - À direita desse foco temos apenas terminais ainda não reduzidos
 - À esquerda temos uma mistura de terminais e não-terminais
 - Imediatamente à esquerda do foco temos um potencial candidato à redução
 - O foco começa no início da entrada
-  • A *análise shift-reduce* funciona tomando uma de duas ações a cada passo: *shift*, que desloca o foco para à direita, e *reduce*, que faz uma redução

Shift e reduce

$|w \xrightarrow{*} s|$

- Shift: move o foco uma posição à direita
 - $A B C | x y z \Rightarrow A B C x | y z$ 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 $C b \underline{x y} | i j k \Rightarrow C b A | i j k$ 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 \textcircled{S} , onde S é o símbolo inicial

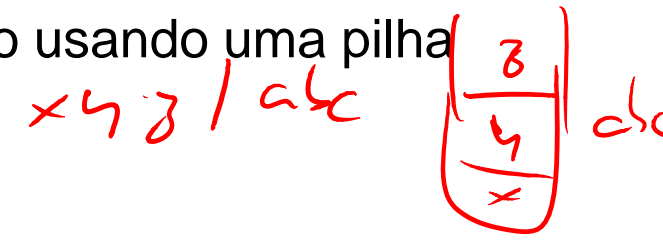
Exercício

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

- (+ ~) - ~ S	- (E + ~) - ~ S	T - ~ R
- (~ + ~) - ~ S	- (E + ~) - ~ R	E - ~ S
- (~ + ~) - ~ S	- (E + F) - ~ R	E - ~ S
- (~ + ~) - ~ R	- (E + T) - ~ R	E - ~ R
- (E + ~) - ~ R	- (E) - ~ S	E - F R
- (T + ~) - ~ R	- (E) - ~ R	E - T R
- (E + ~) - ~ S	- E - ~ R	E ✓
	F - ~ R	

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 $uAw \rightarrow uvw$, a cadeia v é um *handle* de uvw
- Queremos reduzir quando o topo 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 *→ autômato LR(0)*

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 $F \rightarrow (E)$ são:

$F \rightarrow \cdot (E)$
 $F \rightarrow (\cdot E)$
 $F \rightarrow (E \cdot)$
 $F \rightarrow (E) \cdot$

item de redução

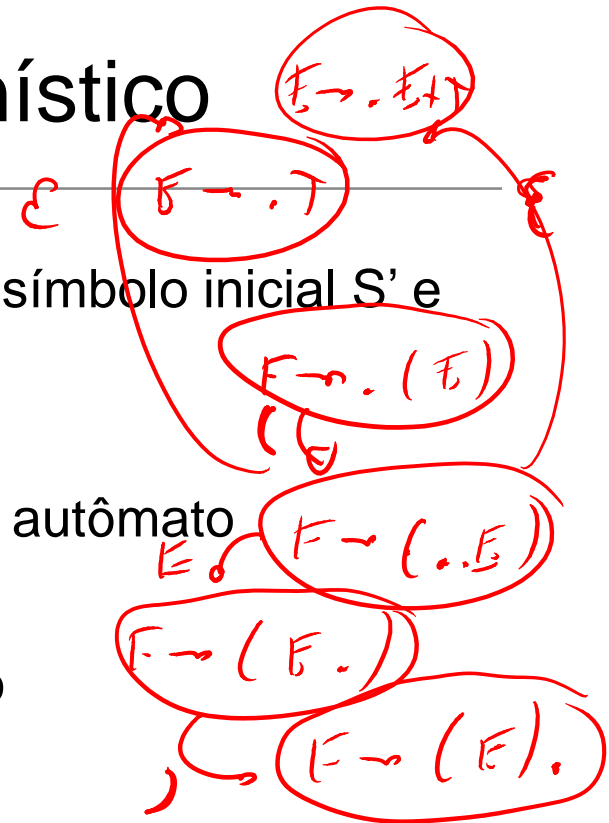
$(F \rightarrow (E), 2)$

$\rho \rightarrow \cdot$

- 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 $S' \rightarrow S$
- O item $S' \rightarrow . S$ 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

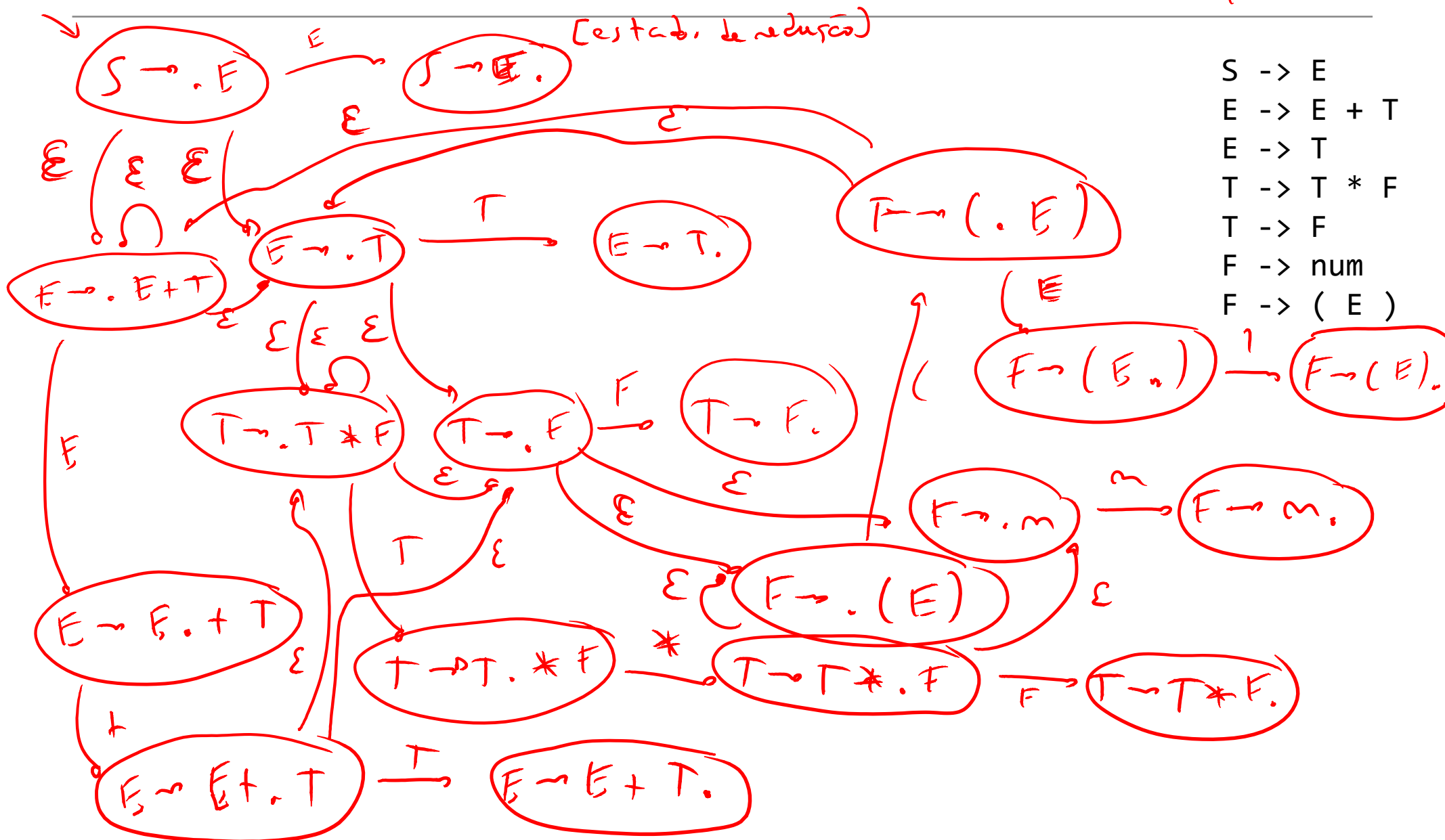
- Vamos usar como exemplo uma gramática de expressões simplificada:

$S \rightarrow E$
 $E \rightarrow E + T$
 $E \rightarrow T$
 $T \rightarrow T * F$
 $T \rightarrow F$
 $F \rightarrow \text{num}$
 $F \rightarrow (E)$

- Vamos construir o NFA de prefixos viáveis dessa gramática

NFA de prefixos viáveis

x42 / abc
~~x47~~ / bc



$S \rightarrow E$
 $E \rightarrow E + T$
 $E \rightarrow T$
 $T \rightarrow T * F$
 $T \rightarrow F$
 $F \rightarrow \text{num}$
 $F \rightarrow (E)$

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 *e o item*
 - Senão faça um *shift* e tente de novo *(não há itens de redução no estado!)*
 - Se o autômato chegou em $S' \rightarrow S$, e chegamos no final da entrada a entrada foi aceita

Construção direta do DFA

- Na prática construímos 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, com no máximo uma para cada terminal saindo de cada estado

$+ | * | n + n$

Autômato LR(0)

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



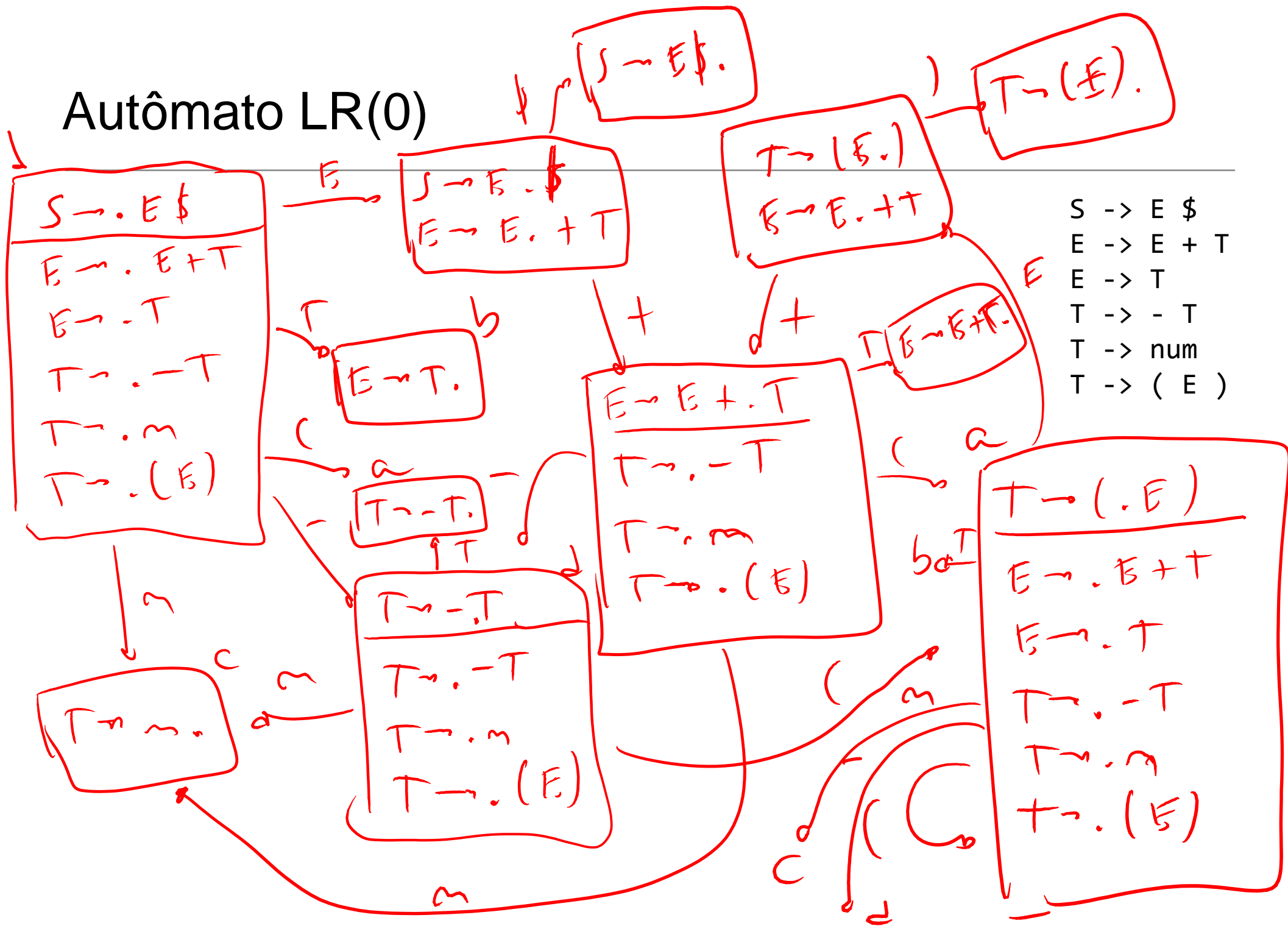
Um exemplo que funciona

- 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:

$S \rightarrow E \$$
 $E \rightarrow E + T$
 $E \rightarrow T$
 $T \rightarrow - T$
 $T \rightarrow \text{num}$
 $T \rightarrow (E)$

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

Autômato LR(0)



$$-(E+m|f \rightarrow -(E+T|f \rightarrow -(E|f$$

Analisando uma entrada

$$-(m+m)+m|f \quad -(E|f$$

$$1-(m+m)+m|f \quad S$$

$$-1-(m+m)+m|f \quad S$$

$$-(1+m+m)+m|f \quad S$$

$$-(m|+m)+m|f \quad R$$

$$-(+|+m)+m|f \quad R$$

$$-(E|+m)+m|f \quad S$$

$$-(E+|m)+m|f \quad S$$

$$-(E+m|)+m|f \quad R$$

$$-(E+T|)+m|f \quad R$$

$$-(E|)+m|f \quad S$$

$$-(E)|+m|f \quad R$$

$$-T|+m|f \quad R \quad S|$$

$$T|+m|f \quad R \quad S$$

$$E|+m|f \quad S \quad E|f$$

$$E+|m|f \quad S \quad S$$

$$E+m|f \quad R \quad E|f \quad S$$

$$E+T|f \quad R \rightarrow$$