Analisador LL(1) de tabela

- No analisador LL(1) recursivo, o contexto de análise (onde estamos na árvore sintática) é mantido pela pilha de chamadas da linguagem

- Mas podemos escrever um analisador LL(1) genérico (que funciona para qualquer gramática LL(1)), mantendo esse contexto em uma pilha explícita

- O analisador funciona a partir de uma *tabela LL(1)*

  - As linhas da tabela são os não-terminais, as colunas são terminais

  - As células são a regra escolhida para aquele não-terminal, dado o terminal como *lookahead*

Exemplo

- Uma gramática LL(1) simples:

```
PROG -> CMD ; PROG
PROG ->
CMD  -> id = EXP
CMD  -> print EXP
EXP  -> id
EXP  -> num
EXP  -> ( EXP + EXP )
```

- Vamos construir a tabela LL(1)

Conjuntos FIRST+

- Calculados a partir dos conjuntos FIRST e FOLLOW das regras

```
PROG -> CMD ; PROG          -> [id, print]
PROG ->                     -> [<<EOF>>]
CMD  -> id = EXP            -> [id]
CMD  -> print EXP           -> [print]
EXP  -> id                  -> [id]
EXP  -> num                 -> [num]
EXP  -> ( EXP + EXP )       -> [(]
```

Tabela LL(1)

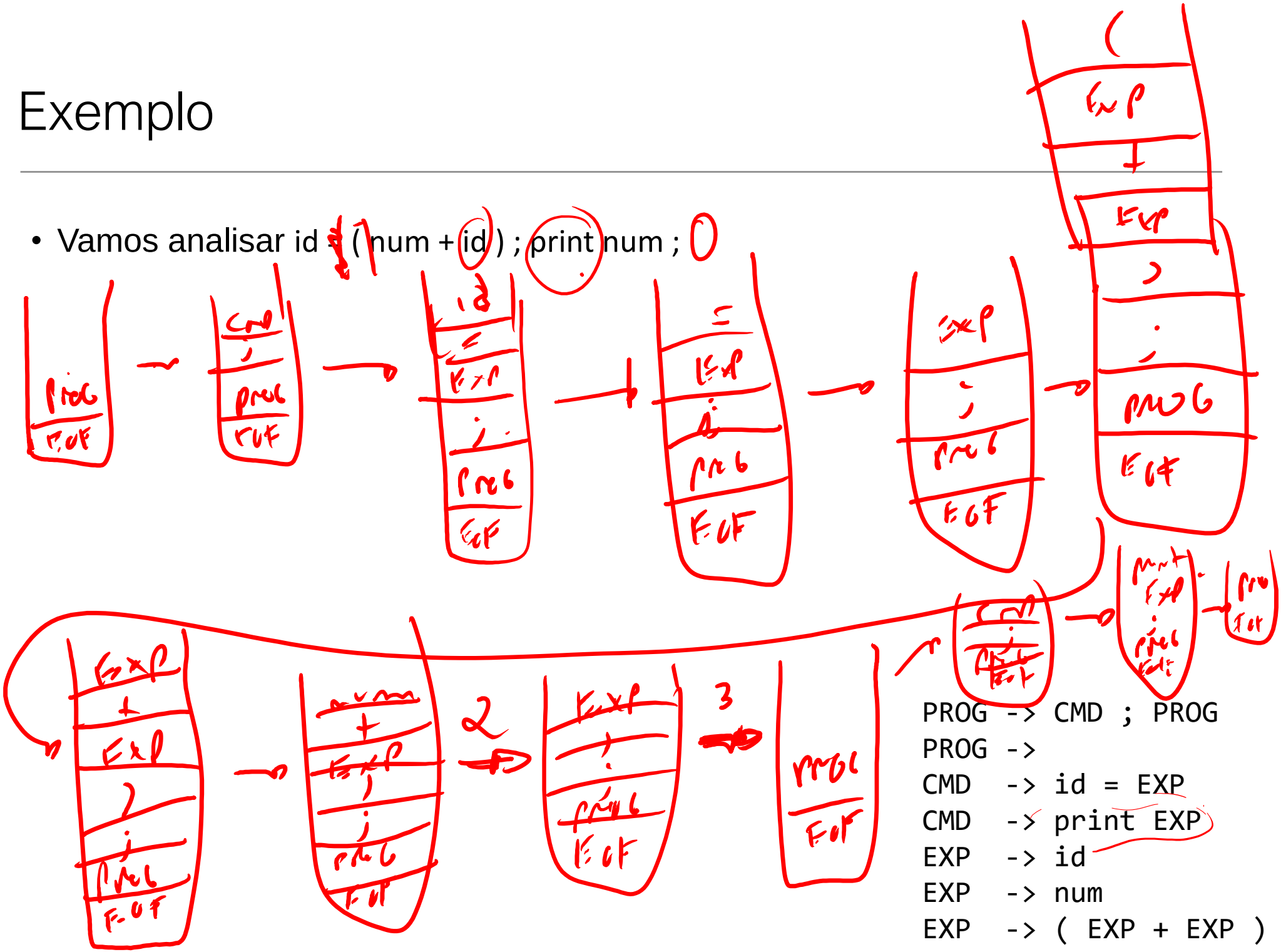
	id	num	;	+	(	)	print	=	EOF
PROG	PROG -> CMD ; PROG						PROG -> CMD ; PROG		PROG ->
CMD	CMD -> id = EXP						CMD -> print EXP		
EXP	EXP -> id	EXP -> num			EXP -> (EXP + EXP)				

Algoritmo

- Pilha começa com <<EOF>> e o símbolo inicial
- Enquanto a pilha não está vazia retiramos o topo da pilha e:
  - Se for um terminal: se casa com o lookahead, avançamos o lookahead, senão dá erro
  - Se for um não-terminal: consultamos a tabela LL(1) e empilhamos o lado direito da produção correspondente, *na ordem reversa*
- Para o algoritmo construir uma árvore, é só empilhar nós ao invés de termos, e acrescentar os filhos ao nó que saiu da pilha

Exemplo

- Vamos analisar id = (num + id) ; print num ;



```
PROG -> CMD ; PROG
PROG ->
CMD  -> id = EXP
CMD  -> print EXP
EXP  -> id
EXP  -> num
EXP  -> ( EXP + EXP )
```