Compiladores - Análise LL(1)

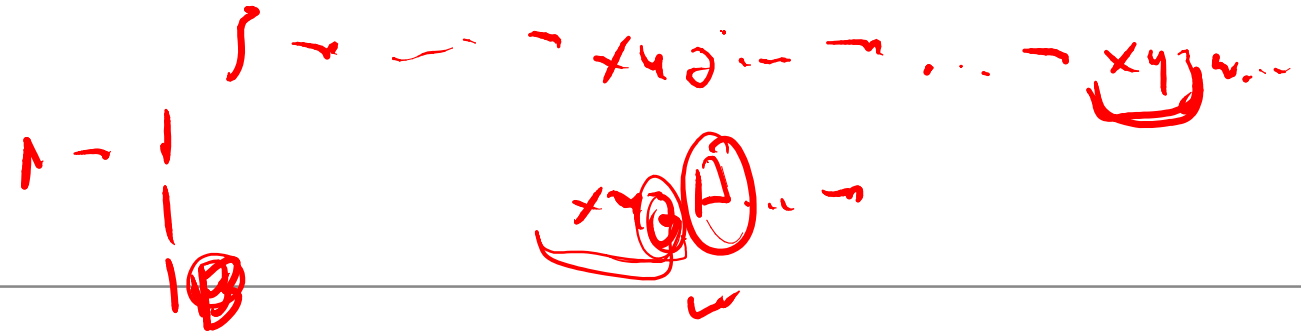
Fabio Mascarenhas – 2015.1

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

Gramáticas LL(1)

- Uma gramática é LL(1) se toda predição pode ser feita examinando um único token à frente
- Muitas construções em gramáticas de linguagens de programação são LL(1), ou podem ser tornadas LL(1) com alguns “jeitinhos”
- A vantagem é que um analisador LL(1) é bastante fácil de construir, e muito eficiente
- Como a análise LL(1) funciona?

Análise LL(1)



- Conceitualmente, o analisador LL(1) constrói uma *derivação mais à esquerda* para o programa, partindo do símbolo inicial

- A cada passo da derivação, o prefixo de terminais da forma sentencial tem que casar com um prefixo da entrada

- Caso exista mais de uma regra para o não-terminal que vai gerar o próximo passo da derivação, o analisador usa o primeiro token após esse prefixo para escolher qual regra usar

- Esse processo continua até todo o programa ser derivado ou acontecer um erro (o prefixo de terminais da forma sentencial não casa com um prefixo do programa)

Exemplo

- Uma gramática LL(1) simples:

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

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

Exemplo

- O terminal entre || é o *lookahead*, usado para escolher qual regra usar

```
PROG

|id| = ( num + id ) ; print num ;
```

Exemplo

- O terminal entre || é o *lookahead*, usado para escolher qual regra usar

```
PROG -> CMD ; PROG

|id| = ( num + id ) ; print num ;
```

Exemplo

- Um prefixo de terminais na forma sentencial desloca o lookahead

```
PROG -> CMD ; PROG -> id = EXP ; PROG

id = |(| num + id ) ; print num ;
```

Exemplo

- Um prefixo de terminais na forma sentencial desloca o lookahead

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

```
id = ( |num| + id ) ; print num ;
```

Exemplo

- Um prefixo de terminais na forma sentencial desloca o lookahead

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

```
id = ( num + |id| ) ; print num ;
```

Exemplo

- Um prefixo de terminais na forma sentencial desloca o lookahead

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

```
id = ( num + id ) ; |print| num ;
```

Exemplo

- Um prefixo de terminais na forma sentencial desloca o lookahead

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

```
id = ( num + id ) ; |print| num ;
```

Exemplo

- Um prefixo de terminais na forma sentencial desloca o lookahead

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

```
id = ( num + id ) ; print |num| ;
```

Exemplo

- O lookahead agora está no final da entrada (EOF)

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

```
id = ( num + id ) ; print num ; ||
```

Exemplo

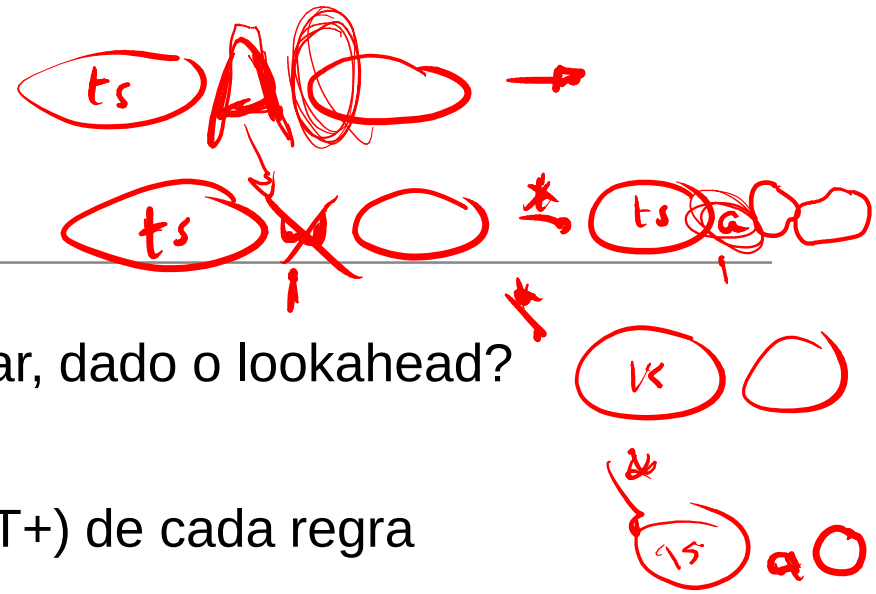
- Chegamos em uma derivação para o programa, sucesso!

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

```
id = ( num + id ) ; print num ; ||
```

+ 12

FIRST, FOLLOW e FIRST+



- Como o analisador LL(1) sabe qual regra aplicar, dado o lookahead?

- Examinando os *conjuntos de lookahead* (FIRST+) de cada regra

```
FIRST+(A -> w) = | FIRST(w) U FOLLOW(A) - { '' }, se '' em FIRST(w)
                 | FIRST(w), caso contrário
```

- E quem são os conjuntos FIRST e FOLLOW? Revisão de linguagens formais!

```
FIRST(w) = { x é terminal | w -*-> xv, para alguma string v } U
   { '' | w -*-> '' }
FOLLOW(A) = { x é terminal ou EOF | S -*-> wAxv para algum w e v }
```

A condição LL(1)

- Uma gramática é LL(1) se os conjuntos FIRST+ das regras de cada não-terminal são *disjuntos*

- Por que isso faz a análise LL(1) funcionar? Vejamos um passo LL(1):

```
...A... -*-> ...t...  ou
...A... -*-> ...tw...
```

- No primeiro caso isso quer dizer que `t` está no FOLLOW(A)

- No segundo caso, `t` está no FIRST do lado direito da regra de A que foi usada

- A derivação é mais à esquerda, então o primeiro ... é um prefixo de terminais, logo `t` é o lookahead!

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+

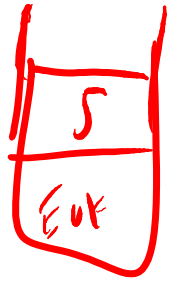
- Calculados a partir dos conjuntos FIRST e FOLLOW das regras

```
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)				

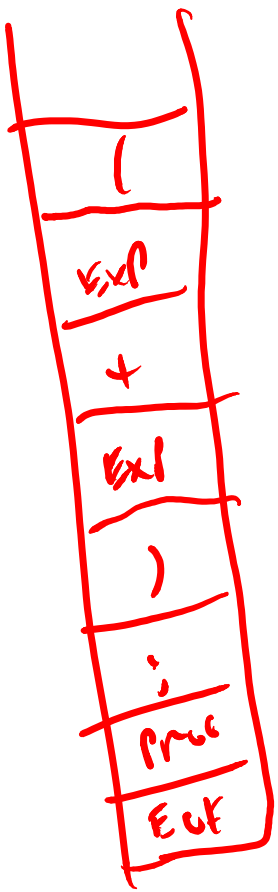
Algoritmo

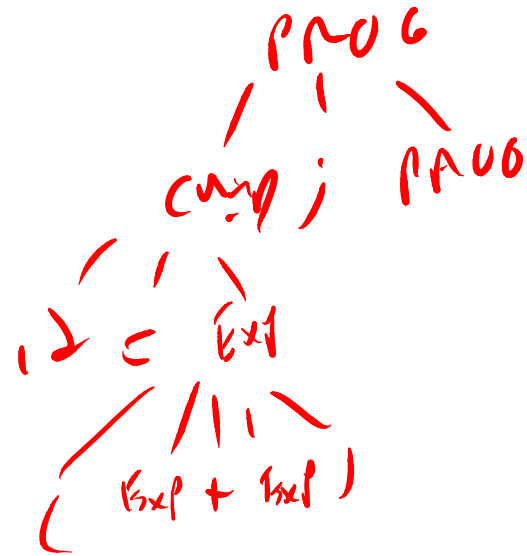


- Pilha começa com <<EOF>> e o símbolo inicial
- Enquanto a pilha não está vazia retiramos o topo da pilha e:
  - Se for um terminal: se casa com o lookahead, avançamos o lookahead, senão dá erro
  - Se for um não-terminal: consultamos a tabela LL(1) e empilhamos o lado direito da produção correspondente, *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 )
```