UNIVERSIDADE FEDERAL DE RORAIMA
DISCIPLINA CONSTRUÇÃO DE COMPILADORES
PROF.: DR. LUCIANO FERREIRA
ALUNO: FELIPE DERKIAN DE SOUSA FREITAS

TRABALHO
SLIDE 10

BOA VISTA, 31 DE OUTUBRO DE 2020

GRAMÁTICAS LL(1)

1) Apresente sua definição e a explique.

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). O terminal entre || é o lookahead, usado para escolher qual regra usar.

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 }.

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): ...At... -*-> ...t... ou ...A... -*-> ...tw…; No primeiro caso isso quer dizer que t está no FOLLOW(A). No segundo caso, t está no FIRST 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!

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.

Um analisador sintático LL é um algoritmo de análise sintática para um subconjunto de gramáticas livre de contexto. Ele é dito um analisador sintático descendente (top-down) pois tenta deduzir as produções da gramática a partir do nó raiz. Ele lê a entrada de texto da esquerda para a direita, e produz uma derivação mais à esquerda (por isso LL, do termo em inglês left-left, diferente do analisador sintático LR). As gramáticas que podem ser analisadas sintaticamente usando esse tipo de analisador são chamadas gramáticas LL.

Outro termo usado é analisador sintático LL(x), no qual x refere-se à quantidade de tokens posteriores ao símbolo atual (ainda não consumidos da entrada) que são usados para tomar decisões na análise. Se tal analisador sintático existe para uma dada gramática, e ele pode analisar sentenças nessa gramática sem o uso de backtracking, então essa gramática é chamada gramática LL(x). Dessas gramáticas com análise posterior, as gramáticas LL(1) são as mais populares, pela necessidade de verificar

somente o token posterior para a análise. Linguagens mal desenvolvidas tipicamente possuem gramáticas com um valor alto de x, e necessitam de bastante esforço computacional para serem analisadas.

ALGORITMO

• Pilha começa com <<EOF>> ou $ e o símbolo inicial.

• Enquanto a pilha não está vazia retiramos o topo da pilha e segue as operações:

    • 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 em vez de termos, e acrescentar os filhos ao nó que saiu da pilha.

2) Apresente 03 (três) exemplos de gramáticas que são LL(1) e 03 (três) exemplos de gramáticas que não são LL(1), explique em cada um dos casos o porquê da classificação.

EX.: 1)

PROG -> CMD ; PROG

PROG -> £

CMD -> id = EXP

CMD -> print EXP

EXP -> id

EXP -> num

EXP -> (EXP + EXP)

Vamos analisar o seguinte comando:

id = (num + id); print num;

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

PROG

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

PROG -> CMD ; PROG

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

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

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

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

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

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

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

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

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

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 ;

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| ;

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 ; ||

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 ; ||

TABELA LL(1) DA GRAMÁTICA

	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)				

EX.: 2)

A gramática abaixo será usada para o exemplo a seguir. Ela trata expressões matemáticas, no qual são aceitas somas entre uns:

(1) S → F
(2) S → (S + F)
(3) F → 1

Vamos analisar a seguinte expressão matemática:

(1 + 1)

PROCEDIMENTO DE ANÁLISE

O analisador sintático primeiro lê o terminal **(** da entrada de texto, e o **S** da pilha. Da tabela é indicado que deve-se aplicar a regra 2, isto é, reescrever **S** para **(S + F)** na pilha e escrever o número dessa regra na saída de dados. No próximo passo é removido o **(** da entrada de dados e da pilha. Agora o analisador verifica o terminal **1** na entrada de texto então aplica a regra 1 e então a regra 3 da gramática, e escreve seus números na saída de dados.

Nos próximos dois passos o analisador sintático lê o **1** e o **+** da entrada de dados e os compara aos valores da pilha. Como são iguais, eles são removidos da pilha. Nos próximos três passos o **F** será substituído da pilha por **1**, e a regra 3 será escrita na saída de dados. Então o **1** e o **)** são removidos da pilha e da entrada de dados. Por fim, o analisador termina com **$** tanto na pilha quanto na entrada de dados. Nesse caso será

retornado que a cadeia de caracteres de entrada foi aceita, e na saída de dados está a lista de regras usadas na análise.

Como pode ser visto no exemplo, o analisador sintático LL realiza três tipos de passos dependendo do conteúdo do topo da pilha, seja não-terminal, terminal ou **$**:

* Se o topo é não terminal então ele verifica a tabela de análise, com base do valor não terminal e o símbolo na entrada de dados, qual regra da gramática deve ser usada. O número da regra é escrito na saída de dados. Se a tabela de análise indica que não há regra programada, é retornado um erro.

* Se o topo é terminal então ele compara o símbolo na entrada com o símbolo do topo da pilha, e se são iguais ambos são removidos. Se eles não são iguais é retornado um erro de sintaxe.

* Se o topo é **$** e na entrada de dados também existe um **$** então ele retorna sucesso de análise, senão erro de sintaxe. Parecido com o tratamento para um topo terminal, note que nesse caso o algoritmo é terminado em ambos os casos.

Esses três passos são repetidos até o algoritmo parar, seja com sucesso ou com erro.

TABELA LL(1) DA GRAMÁTICA

	(	)	1	+	$
S	2	-	1	-	-
F	-	-	3	-	-

EX.: 3)

Gramática que gera cadeias de parênteses balanceados:

(1) S → (S) S

(2) S → £

Vamos analisar a seguinte expressão de parênteses:

( )

Sequência para validação do balanceamento de parênteses:

	Pilha de análise sintática	Entrada	Ação
1	$S	( ) $	$S \rightarrow (S) S$
2	$ S) S (	( ) $	casamento
3	$ S) S	) $	$S \rightarrow \varepsilon$
4	$ S)	) $	casamento
5	$ S	$	$S \rightarrow \varepsilon$
6	$	$	aceita

TABELA LL(1) DA GRAMÁTICA

	(	**)**	**$**
S	**1**	**2**	**-**

REFERÊNCIAS

https://dcc.ufrj.br/~fabiom/comp20131/08LL1.pdf, acessado em: 21/10/2020

https://pt.wikipedia.org/wiki/Analisador_sint%C3%A1tico_LL#Exemplo, acessado em: 21/10/2020

https://erinaldosn.files.wordpress.com/2012/05/anc3a1lise-sintc3a1tica-ll1.pdf, acessado em: 21/10/2020