Compiladores - Prova 2

André L. Mendes Fakhoury
Gustavo V. V. Silva Soares
Eduardo Dias Pennone
Matheus S. Populim
Thiago Preischadt

2021

1 Considere a gramática simples abaixo:

```
<A> ::= ( <B> ) | x
<B> ::= <A> , <B> | <A>
```

1.1 (a) Considerando a análise ascendente de precedência de operadores, mostre passo a passo a construção da tabela sintática pelo método mecânico.(Calcule primeiros e últimos, relações <, > e = de operadores e construa a tabela sintática).

A linguagem é de precedência de operadores e não ambígua.
Os primeiros terminais de <A> são $\{(, x\}$ e os últimos são $\{), x\}$.
Os primeiros e os últimos terminais de <B> serão os mesmos de <A>.
A partir dos pares do tipo aX, que são $(< B >$ e $, < B >$, podemos extrair as seguintes relações: $(<$ (, $(< x$, , $<$ (e , $< x$.
A partir dos pares do tipo Xb, que são $< B >)$ e $< A >$,, podemos extrair as seguintes relações: $) >)$, $x >)$, $) >$, e $x >$,.
A partir da sequência do tipo $a\beta b$, que é $(< B >)$, podemos extrair a seguintes relação: $(=)$.
Também temos que $\$ < (, \$ < x,) > \$$ e $x > \$$.
A tabela sintática é:

	(	)	x	,	$
(	<	=	<		
)		>		>	>
x		>		>	>
,	<		<		
$	<		<		

1.2 (b) Considerando a análise ascendente SLR, construa passo a passo a tabela sintática.

Utilizaremos a gramática equivalente:

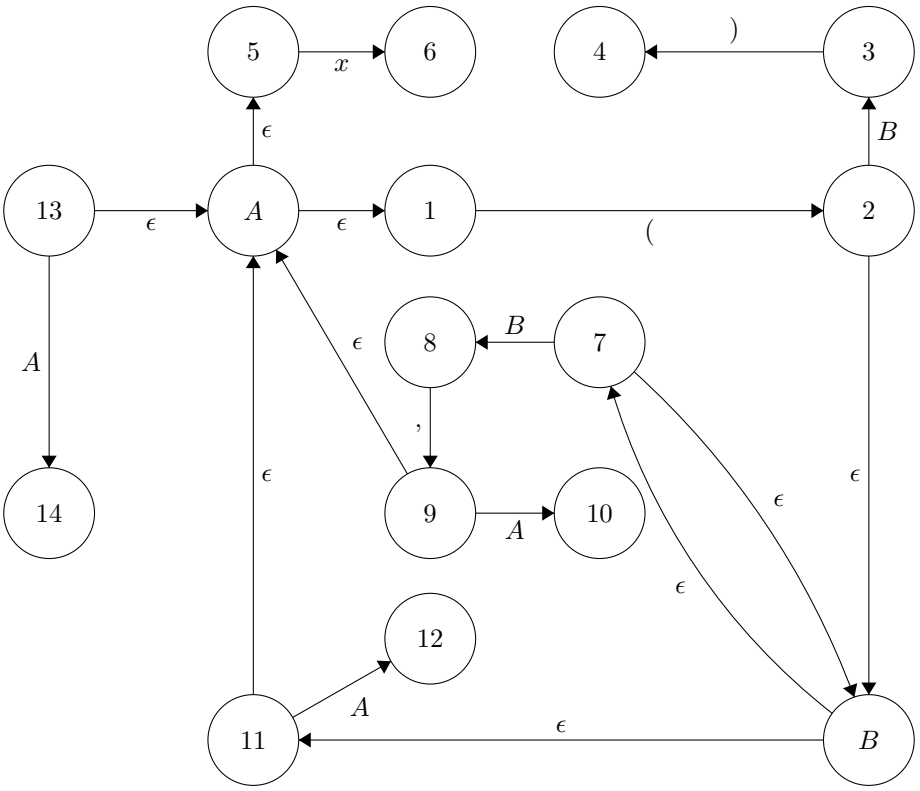
```
(01)  <S> ::= <A>
(02)  <A> ::= ( <B> )
(03)  <A> ::= x
(04)  <B> ::= <B> , <A>
(05)  <B> ::= <A>
```

O conjunto canônico de itens LR(0) da linguagem é:

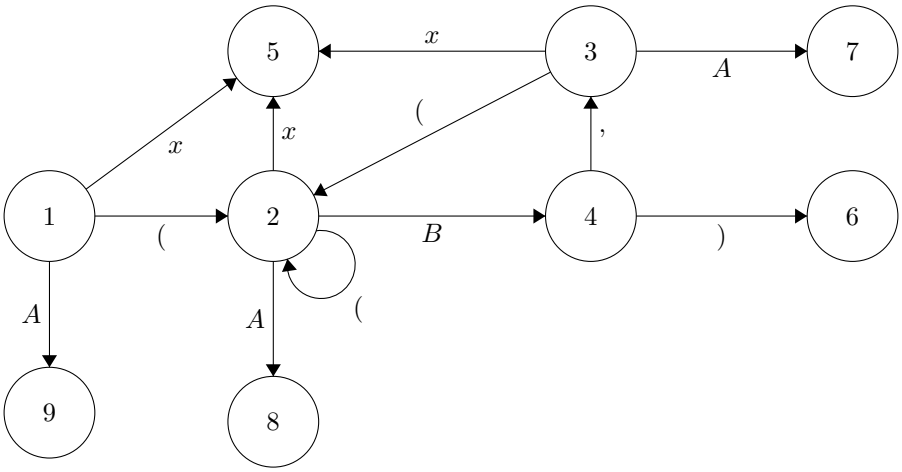
```
(01)  <A> ::= .( <B> )
(02)  <A> ::= (. <B> )
(03)  <A> ::= ( <B>. )
```

```
(04)   <A> ::= ( <B> ).
(05)   <A> ::= .x
(06)   <A> ::= x.
(07)   <B> ::= .<B> , <A>
(08)   <B> ::= <B>. , <A>
(09)   <B> ::= <B> ,. <A>
(10)   <B> ::= <B> , <A>.
(11)   <B> ::= .<A>
(12)   <B> ::= <A>.
(13)   <S> ::= .<A>
(14)   <S> ::= <A>.
```

Com isso podemos construir o seguinte AFND.



Transformando o AFND em um AFD temos:



Com isso podemos construir a tabela sintática:

estado	(	)	x	,	<A>	<B>
1	s2		s5		s9	
2	s2		s5		s8	s4
3	s2		s5		s7	
4		s6		s3		
5	r3	r3	r3	r3	r3	r3
6	r2	r2	r2	r2	r2	r2
7	r4	r4	r4	r4	r4	r4
8	r5	r5	r5	r5	r5	r5
9	r1	r1	r1	r1	r1	r1

2 Construir a tabela sintática de uma análise SLR para a gramática abaixo:

```
S → if E then C | C
E → a
C → b
```

Como o símbolo S pode gerar duas transições, devemos aplicar uma ampliação na gramática com `S'` → `S`. Após isso, geramos o seguinte conjunto canônico de itens `LR(0)`:

```
S' → .S
S' → S.
S → .if E then C
S → if .E then C
S → if E. then C
S → if E then .C
S → if E then C.
S → .C
S → C.
E → .a
E → a.
C → .b
C → b.
```

Que, utilizados como estados, geram o AFND visto na figura 1.

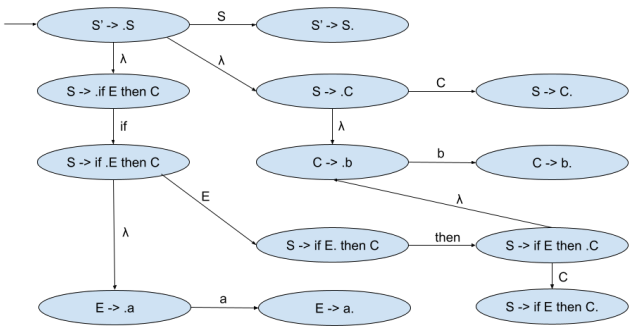


Figura 1: AFND da gramática do Ex. 2

Este autômato pode ser transformado no seguinte AF determinístico, visto na figura 2.

Com as informações das regras da gramática, e analisando o autômato, podemos desenvolver a seguinte tabela sintática de análise SLR, descrita na figura 3.

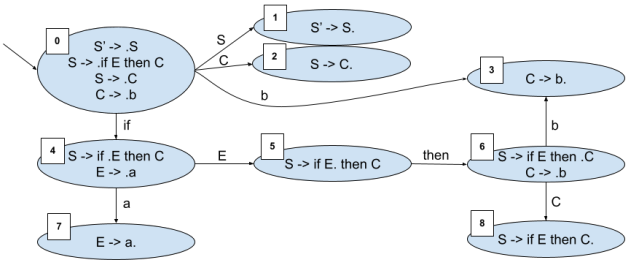


Figura 2: AFD da gramática do Ex. 2

Estado	Operação	Regra	Input				Transição		
			if	a	then	b	S	E	C
0	empilha		4			3	1		2
1	redução	S' -> S							
2	redução	S -> C							
3	redução	C -> b							
4	empilha			7				5	
5	empilha				6				
6	empilha					3			8
7	redução	E -> a							
8	redução	S -> if E then C							

Figura 3: Tabela sintática SLR da gramática do Ex. 2

3 Uma gramática de atributos pode ser utilizada também na fase de geração de código (ver exemplo na questão 5). Neste contexto:

3.1 (a) Cite uma desvantagem do uso de gramáticas de atributos para gerar código.

Uma desvantagem do uso de gramáticas de atributos na geração de código é a possibilidade do alto consumo de memória, visto que a passagem de informações dos nós-folha aos seus ancestrais até a chegada na raiz acaba gerando muitas cópias da cadeia.

3.2 (b) Cite outro tipo de implementação de geração de código que não utilize a tabela sintática.

Um tipo de implementação de geração de código que não utiliza a tabela sintática é a solução *ad-hoc*, isto é, ao invés de se utilizar a tabela sintática, podemos fazer a geração de código em conjunto com a análise sintática da cadeia. Ao se amarrar esse processo aos próprios procedimentos sintáticos, evitamos problemas de consumo de memória (citado acima) e complexidade adicional intrínseca as gramáticas de atributos.

4 Considere a seguinte gramática de atributos para geração de código de 3 endereços:

```
exp → id = exp | aexp
aexp → aexp + fator | fator
fator → (exp) | num | id
```

Regra Gramatical	Regras Semânticas
$exp_1 \rightarrow$ `id` = exp_2	exp_1 .name = exp_2 .name exp_1.tacode = exp_2 .tacode ++ `id`.strval || "=" || exp_2 .name
$exp \rightarrow aexp$	exp .name = aexp .name exp .tacode = aexp .tacode
$aexp_1 \rightarrow aexp_2$ + fator	$aexp_1$ .name = newtemp( ) $aexp_1$ .tacode = $aexp_2$ .tacode ++ fator .tacode ++ $aexp_1$ .name || "=" || $aexp_2$ .name || "+" || fator .name
$aexp \rightarrow fator$	aexp .name = fator .name aexp .tacode = fator .tacode
$fator \rightarrow$ (exp)	fator .name = exp .name fator .tacode = exp .tacode
$fator \rightarrow$ `num`	fator .name = `num`.strval fator .tacode = ""
$fator \rightarrow$ `id`	fator .name = `id`.strval fator .tacode = ""

4.1 Considere que o símbolo || representa concatenação sem pular linha, ++ representa concatenação que pula linha após a operação e strval representa o valor numérico da string. Considere também que newtemp() gera novo nome temporário, que é guardado no atributo "name". Mostre a árvore sintática da cadeia (x=x+1)+2+y e mostre o código de 3 endereços final gerado. Mostre também os códigos gerados nos vértices intermediários da árvore.

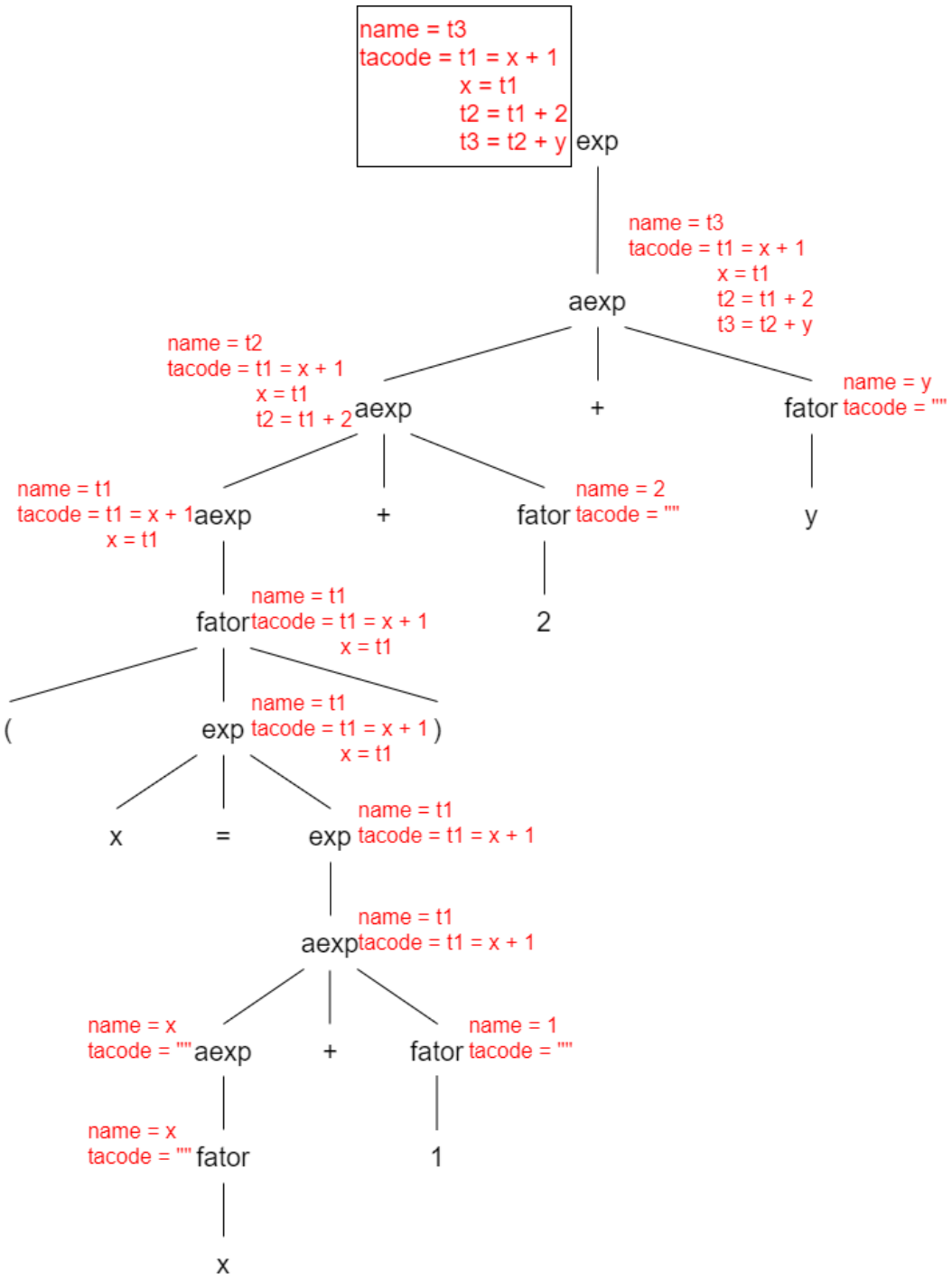


Figura 4: Árvore sintática e código de 3 endereços gerado do Ex. 4