Análise Sintática Ascendente

João Marcelo Uchôa de Alencar
joao.marcelo@ufc.br
UFC-Quixadá

Introdução

- **Análise Sintática LR(1)**: entrada processada da esquerda para a direita, mas a derivação produzida é uma derivação à direita, analisando um símbolo da entrada;
- **Análise Sintática LR(0)**: não analisa o símbolo na entrada, apenas quando ele já aparece na pilha;
- **Análise Sintática SLR(1)**: versão melhorada da LR(0), com verificação da entrada;
- **Análise Sintática LALR(1)**: mais poderoso que a SLR(1).

Ordem de complexidade e poder de expressão:

$$LR(0) < SLR(1) < LALR(1) < LR(1)$$

Recursão à esquerda não é um problema para a análise ascendente. Por que?

Visão Geral

A **pilha** conterá tanto marcas como não terminais, e também informações adicionais de estados.

```
$       ...          CadeiaEntrada $
        ...               ...      $
        ...               ...      $
$ SímboloInicial                   $ aceita
```

Duas operações possíveis:

1. **Carrega** (*shift*) um terminal do topo da entrada para o topo da pilha;
2. **Reduz** (*reduce*) uma cadeia α do topo da pilha para um não terminal A, dada a escolha BNF $A \rightarrow \alpha$.

As gramáticas são aumentadas com um novo **símbolo inicial**, com uma única produção unitária para o símbolo inicial anterior.

Visão Geral - Exemplo

$S' \rightarrow S$
$S \rightarrow (S)S|\varepsilon$

Considerando a cadeia () temos:

	Pilha de Análise Sintática	Entrada	Ação
1	\$	()\$	carrega
2	\$(	)\$	reduz $S \rightarrow \varepsilon$
3	\$(S	)\$	carrega
4	\$(S)	\$	reduz $S \rightarrow \varepsilon$
5	\$(S)S	\$	reduz $S \rightarrow (S)S$
6	\$S	\$	reduz $S' \rightarrow S$
7	\$$S'$	\$	aceita

Visão Geral - Exemplo

$E' \rightarrow E$
$E \rightarrow E + n | n$

Considerando a cadeia $n + n$ temos:

	Pilha de Análise Sintática	Entrada	Ação
1	\$	n+n\$	carrega
2	\$n	+n\$	reduz $E \rightarrow n$
3	\$E	+n\$	carrega
4	\$E+	n\$	carrega
5	\$E+n	\$	reduz $E \rightarrow E + n$
6	\$E	\$	reduz $E' \rightarrow E$
7	\$$E'$	\$	aceita

Visão Geral

- Um analisador ascendente pode carregar os símbolos de entrada para a pilha até determinar que ação deve executar;
- ao mesmo tempo, ele pode precisar de outros elementos da pilha, além do topo, para determinar a ação a ser executada;
- **verificações à frente na pilha**: autômato finito determinístico de *itens*;
- as verificações na pilha não eliminam a necessidade de analisar a entrada;
- a forma em que a verificação é feita é o que difencia o poder e a complexidade dos algoritmos ascendentes.

Visão Geral - Conceitos

Um analisador *shift-reduce* acompanha uma derivação à direita da cadeia, mas os passos ocorrem em ordem inversa:

- **Forma sentencial à direita**: é uma cadeia intermediária e terminais e não terminais na derivação;
- **prefixo viável**: porção da forma sentencial à direita que está na pilha em um dado momento da derivação;
- **gancho**: cadeia de símbolos no topo da pilha que casa com o lado direito de uma produção + posição na forma sentencial à direita onde ocorre + produção para a redução.

A tarefa principal de um analisador carrega-reduz é determinar o gancho seguinte em uma análise sintática.

Análise Sintática LR(0)

- Um **item LR(0)** de uma gramática livre de contexto é uma escolha de produção com uma posição identificada em seu lado direito;
- se $A \to \alpha$ e $\beta\gamma = \alpha$, $A \to \beta.\gamma$ é um item LR(0).

Para os exemplos:

$S' \to .S$
$S' \to S.$
$S \to .(S)S$
$S \to (.S)S$
$S \to (S.)S$
$S \to (S).S$
$S \to (S)S.$
$S \to .$

$E' \to .E$
$E' \to E.$
$E \to .E + n$
$E \to E. + n$
$E \to E + .n$
$E \to E + n.$
$E \to .n$
$E \to n.$

Um item registra um passo intermediário no reconhecimento do lado direito de uma escolha.

Autômatos Finitos para Itens

Controlar as Opções de Redução

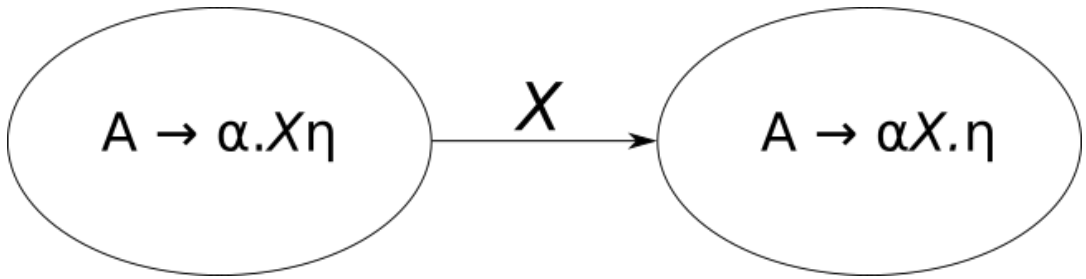
Os itens LR(0) podem ser utilizados como os estados de um autômato finito que mantém as informações sobre a pilha de análise sintática e o **progresso** de uma análise *shift-reduce*.

Etapas:

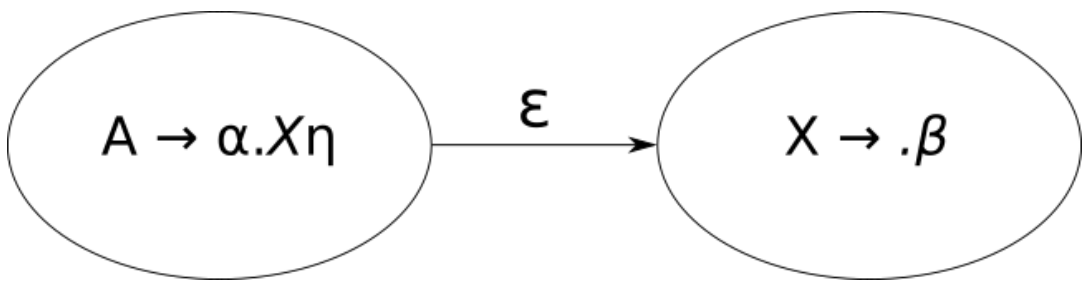
1. Construção de um NFA a partir dos itens;
2. utilizar a construção de subconjuntos para definir um DFA.

Construção de um NFA dos Itens

Considere $A \to \alpha\gamma$ e γ inicia com X (terminal ou não), tal que $A \to \alpha.X\eta$. Então existe a transição para o item $A \to \alpha X.\eta$:



Se X for marca, carregar X da entrada para o topo da pilha. Se X for não terminal, ele só pode aparecer por redução. Logo, para $X \to \beta$, β deve ser reconhecido *simultaneamente* com a transição acima.



Para todas as possibilidades de β, $X \to \beta$.

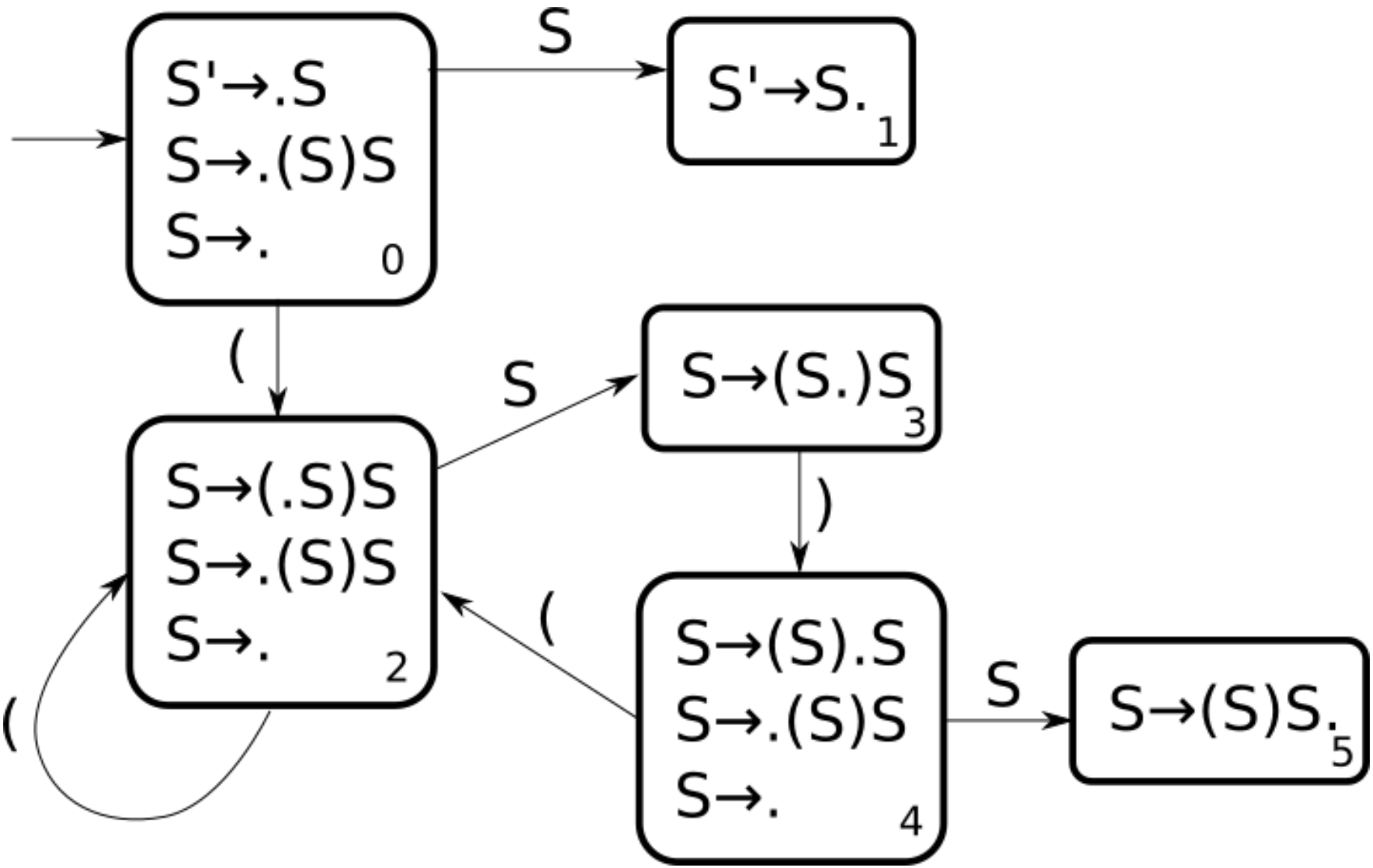
Exemplos

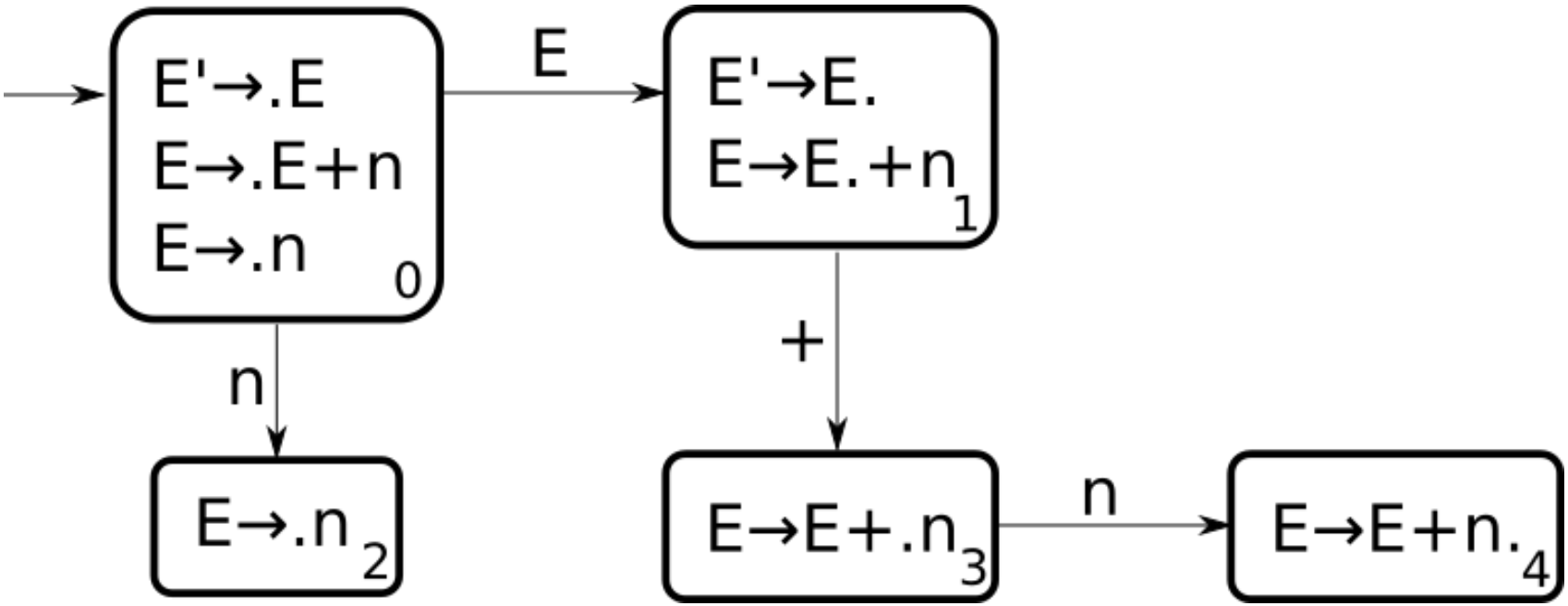
Vamos construir os NFAs para os exemplos abaixo.

$$S' \to .S$$
$$S' \to S.$$
$$S \to .(S)S$$
$$S \to (.S)S$$
$$S \to (S.)S$$
$$S \to (S).S$$
$$S \to (S)S.$$
$$S \to .$$

$$E' \to .E$$
$$E' \to E.$$
$$E \to .E + n$$
$$E \to E. + n$$
$$E \to E + .n$$
$$E \to E + n.$$
$$E \to .n$$
$$E \to n.$$

Depois, usar a **construção de subconjuntos** para transformá-los em DFAs.


S'→.S
S→.(S)S
S→.
0
S
S'→S.
1
(
S
S→(S.)S
3
S→(.S)S
S→.(S)S
S→.
2
)
(
(
S→(S).S
S→.(S)S
S→.
4
S
S→(S)S.
5


E'→.E
E→.E+n
E→.n
0
E
E'→E.
E→E.+n
1
n
+
E→.n
2
E→E+.n
3
n
E→E+n.
4

Definições Adicionais sobre os Autômatos LR(0)

- **Itens de fecho**: itens que são adicionados a um estado durante um ε-fecho;
- **itens de núcleo**: itens que geram estados alvos de transições que não são ε-transições.

Os itens de núcleo determinam de forma única o estado e suas transições.

O Algoritmo de Análise Sintática LR(0)

A pilha passa a conter não apenas símbolos, mas também número de estados. Início:

Pilha de Análise Sintática	Entrada
\$ 0	*CadeiaEntrada* \$

Carregar a marca *n* para a pilha e ir para o estado 2:

Pilha de Análise Sintática	Entrada
\$ 0*n*2	*Restante da CadeiaEntrada* \$

O algoritmo escolhe uma ação com base no estado corrente do DFA, que está no topo da pilha.

O Algoritmo de Análise Sintática LR(0)

Seja s o estado corrente (topo da pilha), ações:

1. Se o estado s contiver um item da forma $A \rightarrow \alpha.X\beta$, X terminal, então a **ação** é carregar a marca da entrada para a pilha. Se a marca for X e s contiver o item $A \rightarrow \alpha.X\beta$, então o novo estado a ser empilhado é o que contiver $A \rightarrow \alpha X.\beta$. Caso contrário, erro.
2. Se o estado s contiver um item completo $A \rightarrow \gamma.$ então a **ação** é reduzir por $A \rightarrow \gamma$. Uma redução por $S' \rightarrow S$ equivale a aceitação, se a entrada estiver vazia. Caso contrário:
   - Remover γ e os estados correspondentes;
   - retorne o DFA para o estado no qual iniciou a construção de γ;
   - o estado atual deve conter item da forma $B \rightarrow \alpha.A\beta$;
   - coloque A na pilha e atualize o estado para o que contiver $B \rightarrow \alpha A.\beta$.

Conflitos da Análise Sintática LR(O)

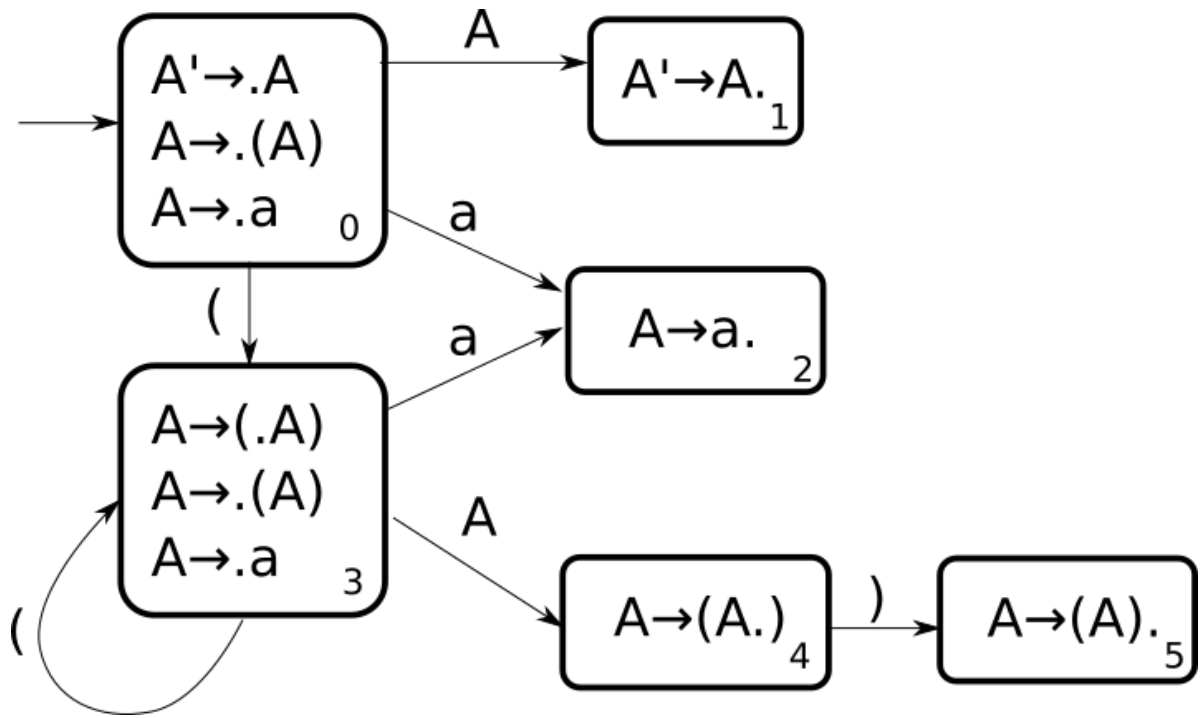
Uma gramática é dita LR(0) se as regras anteriores não forem ambíguas. Tipos de conflitoss:

- **Conflito carrega-reduz**: um estado contém tanto um item $A \rightarrow \alpha.$ quanto $A \rightarrow \alpha.X\beta$;
- **conflito reduz-reduz**: um estado contém tanto um item $A \rightarrow \alpha.$ quanto $B \rightarrow \beta.$;

Nenhuma das gramáticas apresentadas como exemplo até agora são LR(0). Por quê?

Exemplo de Gramática LR(0)

Para a gramática:
$A \rightarrow (A)|a$
Temos o seguinte autômatos de itens:



Vamos fazer o reconhecimento de ((a))?

Análise Sintática SLR(1)

Vamos aumentar o poder da análise LR(0):

1. Consultar a marca de entrada *antes* de carregar, para garantir que exista uma transição apropriada no DFA;
2. utilizar o conjunto Sequência de um não-terminal para decidir se uma redução deve ser efetuada.

Algoritmo de Análise Sintática SLR(1)

Seja s o estado corrente (topo da pilha), ações:

1. Se o estado s contiver um item da forma $A \rightarrow \alpha.X\beta$, X terminal, **e X for a marca seguinte na cadeia de entrada**, então a **ação** é carregar a marca da entrada para a pilha. O novo estado a ser empilhado é o que contiver $A \rightarrow \alpha X.\beta$.
2. Se o estado s contiver um item completo $A \rightarrow \gamma.$, **e a marca seguinte na cadeia de entrada estiver em Sequência(A)**, então a **ação** é reduzir por $A \rightarrow \gamma$. Uma redução por $S' \rightarrow S$ equivale a aceitação, se a entrada for \$. Caso contrário:
   - Remover γ e os estados correspondentes;
   - retorne o DFA para o estado no qual iniciou a construção de γ;
   - o estado atual deve conter item da forma $B \rightarrow \alpha.A\beta$;
   - coloque A na pilha e atualize o estado para o que contiver $B \rightarrow \alpha A.\beta$.
3. Se a marca de entrada for tal que nenhum caso acima se aplique, erro!

Conflitos da Análise Sintática SLR(1)

Dizemos que uma gramática é SLR(1) se a aplicação das regras não resultarem em ambiguidade. As duas condições devem ser satisfeitas:

1. Para qualquer item $A \to \alpha.X\beta$ em s em que X for um não terminal, não existe um item completo $B \to \gamma.$ em s com X em Sequência(B);
2. Para quaisquer dois itens completos $A \to \alpha.$ e $B \to \beta.$ em s, Sequência(A) $\cap$ Sequência(B) é vazio.

A violação da primeira condição representa um **conflito carrega-reduz**. A violação da segunda condição representa um **conflito reduz-reduz**.

Construção da Tabela SLR(1)

Percorrendo o autômato gerado, considerando cada terminal e não-terminal, podemos construir a tabela SLR(1). Dada a gramática

$E' \rightarrow E$

$E \rightarrow E + n | n$

e o autômato gerado para ela e o fato de que Sequência(E)= $\{\$\}$ e Sequência(E')= $\{\$, +\}$, temos a tabela:

Estado	**Entrada**			**Ir-para**
	n	$+$	\$	E
0	s2			1
1		s3	aceita	
2		r($E \rightarrow n$)	r($E \rightarrow n$)	
3	s4			
4		r($E \rightarrow E + n$)	r($E \rightarrow E + n$)	

Vamos fazer a análise da cadeia $n + n + n$.

Regras para Eliminar Ambiguidades

No caso da SLR(1), podemos tomar algumas atitudes para eliminar ambiguidades:

- No caso dos conflitos carrega-reduz, existe uma regra natural, que é sempre preferir carregar em vez de reduzir;
- o caso dos conflitos reduz-reduz, não há solução geral, geralmente é indício de erro no projeto.

A solução de carregar no lugar de reduzir tem algum impacto no problema do *else* aninhado?

Exemplo de Eliminação de Ambiguidade

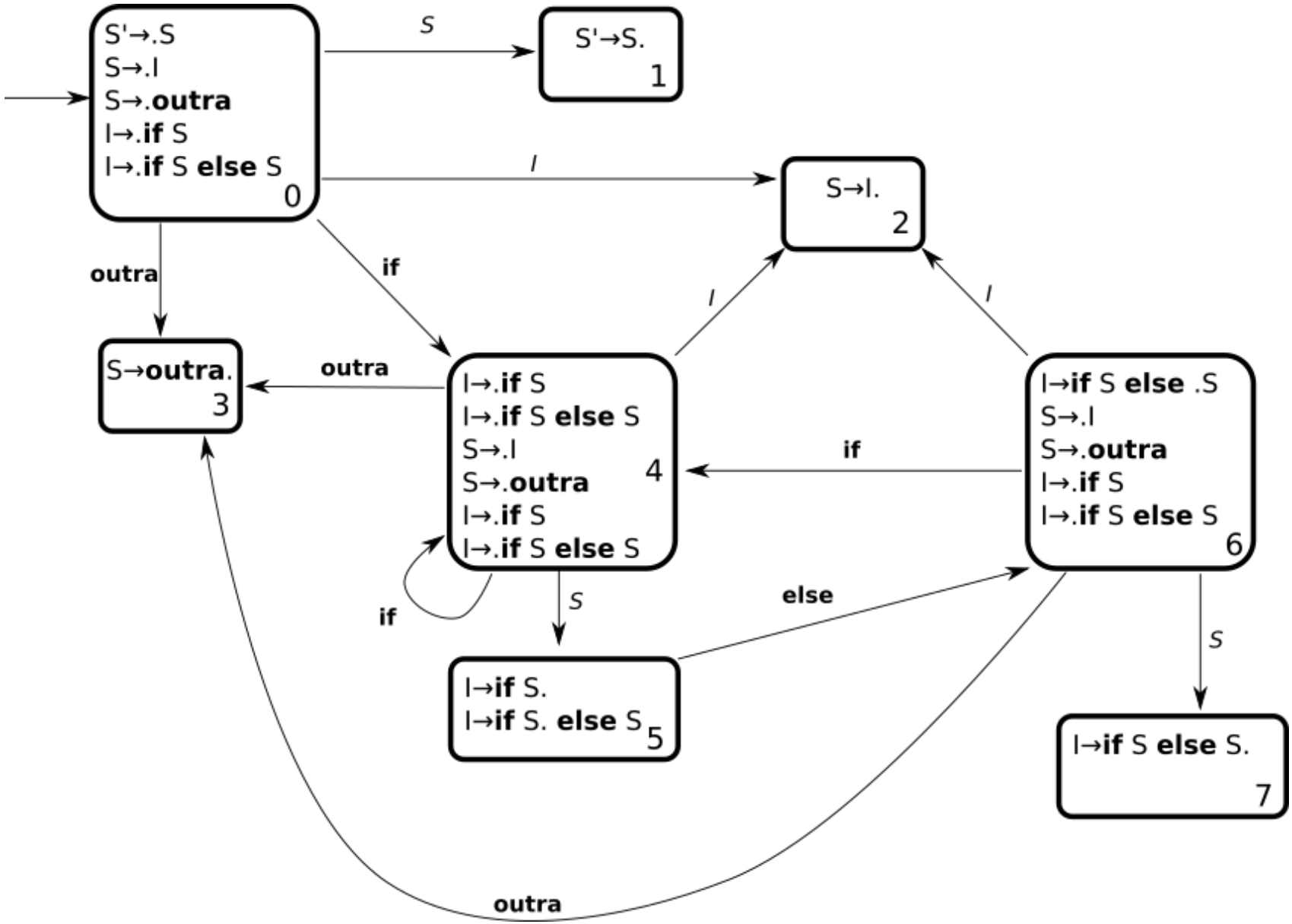
Considere a gramática:

declaração $\rightarrow$ *if-decl*|**outra**
if-decl $\rightarrow$ **if**(*exp*)*declaração*
|**if**(*exp*)*declaração* **else** *declaração*
exp $\rightarrow$ **0**|**1**

É notoriamente ambígua. Vamos simplificá-la para poder criar o autômato:

$S \rightarrow I|$**outra**
I $\rightarrow$ **if** S|**if** S **else** S

Temos a seguinte propriedade:
Sequência(S) = Sequência(I) = $\{\$, else\}$


S'→.S
S→.I
S→.outra
I→.if S
I→.if S else S
0
S
S'→S.
1
I
S→I.
2
outra
if
S→outra.
3
outra
I→.if S
I→.if S else S
S→.I
S→.outra
I→.if S
I→.if S else S
4
I
I
if
I→if S else .S
S→.I
S→.outra
I→.if S
I→.if S else S
6
if
S
else
I→if S.
I→if S. else S
5
S
I→if S else S.
7
outra

Exemplo de Eliminação de Ambiguidade

O estado 5 tem $\{I \rightarrow \textbf{if}\ S.; I \rightarrow \textbf{if}\ S.\ \textbf{else}\ S\}$.

- Devemos reduzir nas entradas **else** e \$ na primeira derivação?
- ou devemos apenas carregar o **else** na segunda derivação?

Estado	**Entrada**				**Ir-para**	
	if	else	outra	\$	S	I
0	s4		s3		1	2
1				aceita		
2		r1		r1		
3		r3		r2		
4	s4		s3		5	2
5		**s6**		r3		
6	s4		s3		7	2
7		r4		r4		

Limitações da Análise SLR(1)

decl $\rightarrow$ *ativação-decl*|*atribuição-decl*
ativação-decl $\rightarrow$ **identificador**
atribuição-decl $\rightarrow$ *var* := *exp*
var $\rightarrow$ *var*[*exp*]|**identificador**
exp $\rightarrow$ *var*|**número**

Tanto as atribuições quanto declarações começam com **identificador**.

Versão simplificada:
$S \rightarrow$ **id**$|V := E$
$V \rightarrow$ **id**
$E \rightarrow V|$**n**
Considere o estado inicial com os itens:
$S' \rightarrow .S$
$S \rightarrow .$**id**
$S \rightarrow .V := E$
$V \rightarrow .$**id**

Há conflito para os terminais **id**.

Algoritmos SLR(k)

Considerar algoritmos SLR(k) aumenta a expressividade para esses casos, mas a **complexidade** se torna intratável.

Análise Sintática Geral LR(1) e LALR(1)

- Podemos resolver o problema anterior com a análise LR(1) **canônica**;
- essa análise torna o processo complexo devido ao tamanho do autômato gerado;
- uma modificação, LALR(1), permite condensar alguns estados em um autômato menor;
- entretanto, para compreender a LALR(1), precisamos entender a LR(1) antes.

Autômatos Finitos de Itens LR(1)

O poder do método LR(1) geral é ele utilizar um DFA novo com as verificações à frente construídas em sua concepção.

- **Item LR(1)**: par composto por um item LR(0) e uma marca de *verificação à frente*;
- $[A \to \alpha.\beta, a]$, onde $A \to \alpha.\beta$ é um item LR(0) e a é uma marca. Não é a marca à frente na entrada, mas sim uma informação sobre o que se espera na entrada após uma redução.

A maior diferença entre os autômatos LR(0) e LR(1) aparece na definição das ε-transições.

Transições dos Autômatos LR(1)

Transições Determinísticas

Dado um item LR(1) $[A \to \alpha.X\gamma, a]$, onde X é qualquer símbolo (terminal ou não), existe uma transição em X para o item LR(1) $[A \to \alpha X.\gamma, a]$. **Não há mudança no símbolo de verificação**.

Transições ε

Dado um item LR(1) $[A \to \alpha.B\gamma, a]$, onde **B é um não terminal**, existem ε-transições $[B \to .\beta, b]$ para cada produção $B \to \beta$ e cada **marca** b pertencente a Primeiro(γa).

- $[A \to \alpha.B\gamma, a]$ indica que reconhecemos B se após ocorrer uma cadeia *derivável* de γa, começando por marcas presentes em Primeiro(γa);
- restringimos as capturas à conjuntos *Primeiro*, não *Sequencia*;
- a verificação original a só é propagada se $\gamma \Rightarrow \varepsilon$.

Exemplo de Autômato LR(1)

Considere a gramática:
$A \rightarrow (A)|a$
Vamos construir o autômato LR(1)?

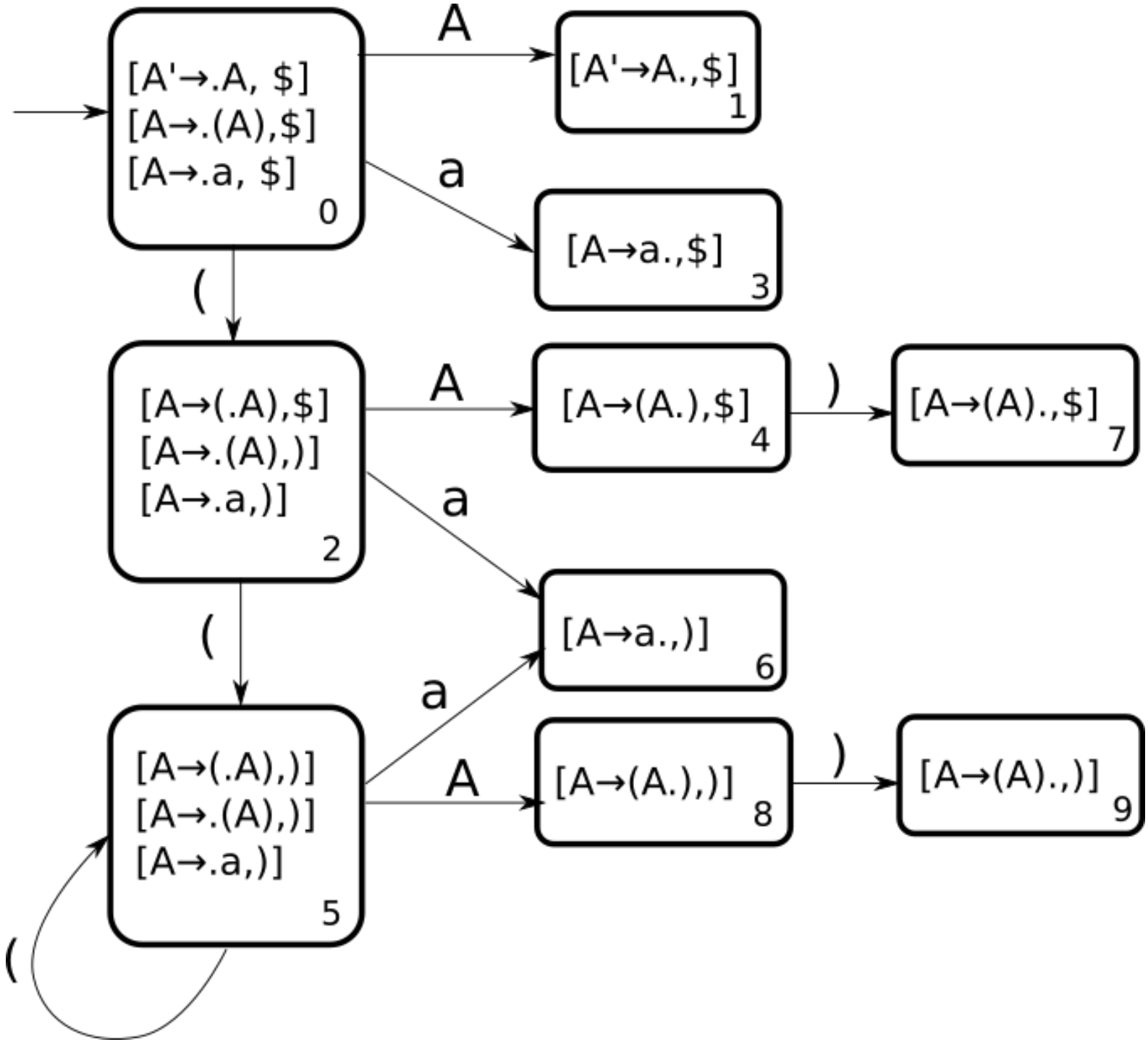

[A'→.A, $]
[A→.(A),$]
[A→.a, $]
0
A
[A'→A.,$]
1
a
[A→a.,$]
3
(
[A→(.A),$]
[A→.(A),)]
[A→.a,)]
2
A
[A→(A.),$]
4
)
[A→(A).,$]
7
a
[A→a.,)]
6
(
[A→(.A),)]
[A→.(A),)]
[A→.a,)]
5
a
A
[A→(A.),)]
8
)
[A→(A).,)]
9
(

Tabela LR(1)

Enumeramos as produções da gramática:

1. $A \to (A)$
2. $A \to a$

Estado	**Entrada**				**Ir-para**
	(	a	)	$	A
0	s2	s3			1
1				aceita	
2	s5	s6			4
3				r2	
4			s7		
5	s5	s6			8
6			r2		
7				r1	
8			s9		
9			r1		

Algoritmo de Análise Sintática LR(1) Geral

Seja s o estado corrente (topo da pilha), ações:

1. Se o estado s contiver um item LR(1) da forma $[A \rightarrow \alpha.X\beta, a]$, X terminal, **e X for a marca seguinte na cadeia de entrada**, então a **ação** é carregar a marca da entrada para a pilha. O novo estado a ser empilhado é o que contiver $[A \rightarrow \alpha X.\beta, a]$.
2. Se o estado s contiver um item completo LR(1) $[A \rightarrow \alpha., a]$, **e a marca seguinte na entrada for** a, então a **ação** é reduzir por $A \rightarrow \alpha$. Uma redução por $S' \rightarrow S$ equivale a aceitação, se a entrada for \$. Caso contrário:
   - Remover α e os estados correspondentes;
   - retorne o DFA para o estado no qual iniciou a construção de α;
   - o estado atual deve conter item da forma $[B \rightarrow \alpha.A\beta, b]$;
   - coloque A na pilha e atualize o estado para o que contiver $[B \rightarrow \alpha A.\beta, b]$.
3. Se a marca de entrada for tal que nenhum caso acima se aplique, erro!

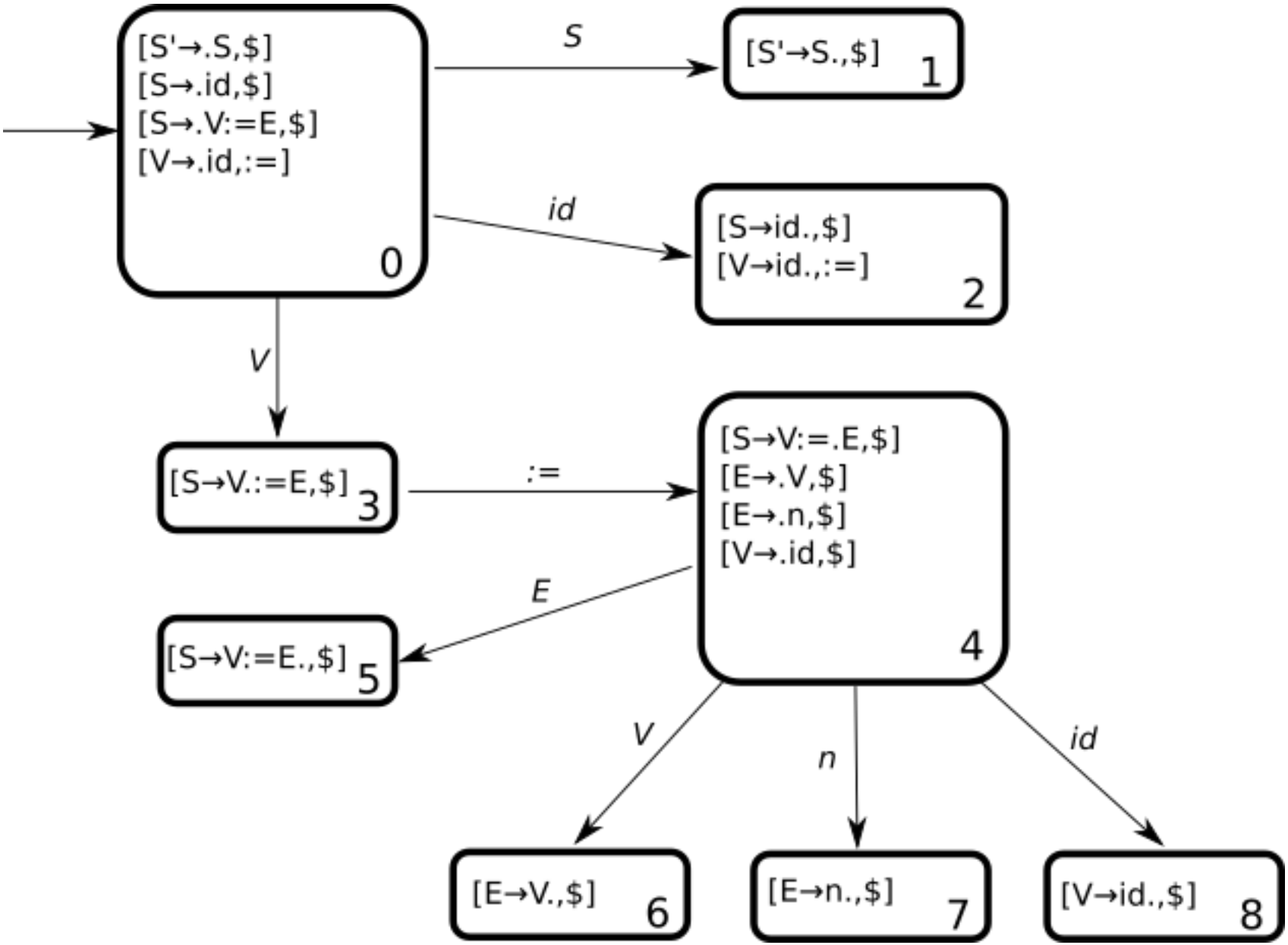
Requisitos para uma Gramática LR(1)

Uma gramática é LR(1) se e somente se, para qualquer estado s, as seguintes duas condições forem satisfeitas:

1. Para qualquer item $[A \to \alpha.X\beta, a]$ pertencente a s em que X é um terminal, não existe um item pertencente a s da forma $[B \to \gamma., X]$ (carrega-reduz);
2. não há dois itens em s da forma $[A \to \alpha., a]$ e $[B \to \beta., a]$ (reduz-reduz).

Exemplo de Gramática LR(1)

Considere o exemplo problemático para SLR(1):
$S \rightarrow \mathbf{id} | V := E$
$V \rightarrow \mathbf{id}$
$E \rightarrow V | \mathbf{n}$
Vamos construir o autômato LR(1)?


[S'→.S,$]
[S→.id,$]
[S→.V:=E,$]
[V→.id,:=]
0
S
[S'→S.,$]
1
id
[S→id.,$]
[V→id.,:=]
2
V
[S→V.:=E,$]
3
:=
[S→V:=.E,$]
[E→.V,$]
[E→.n,$]
[V→.id,$]
4
E
[S→V:=E.,$]
5
V
n
id
[E→V.,$]
6
[E→n.,$]
7
[V→id.,$]
8

Análise Sintática LALR(1)

- O tamanho do DFA LR(1) se deve à existência de estados com o mesmos itens LR(0), mas com símbolos de verificação diferentes;
- a análise LALR(1) identifica tais estados e os combina;
- agora, o estado LALR(1) não tem um símbolo de verificação, mas sim um conjunto deles;
- **núcleo** de um estado LR(1): conjunto de itens LR(0) composto pelos primeiros componentes de todos os itens LR(1) no estado.

Princípios da Análise Sintática LALR(1)

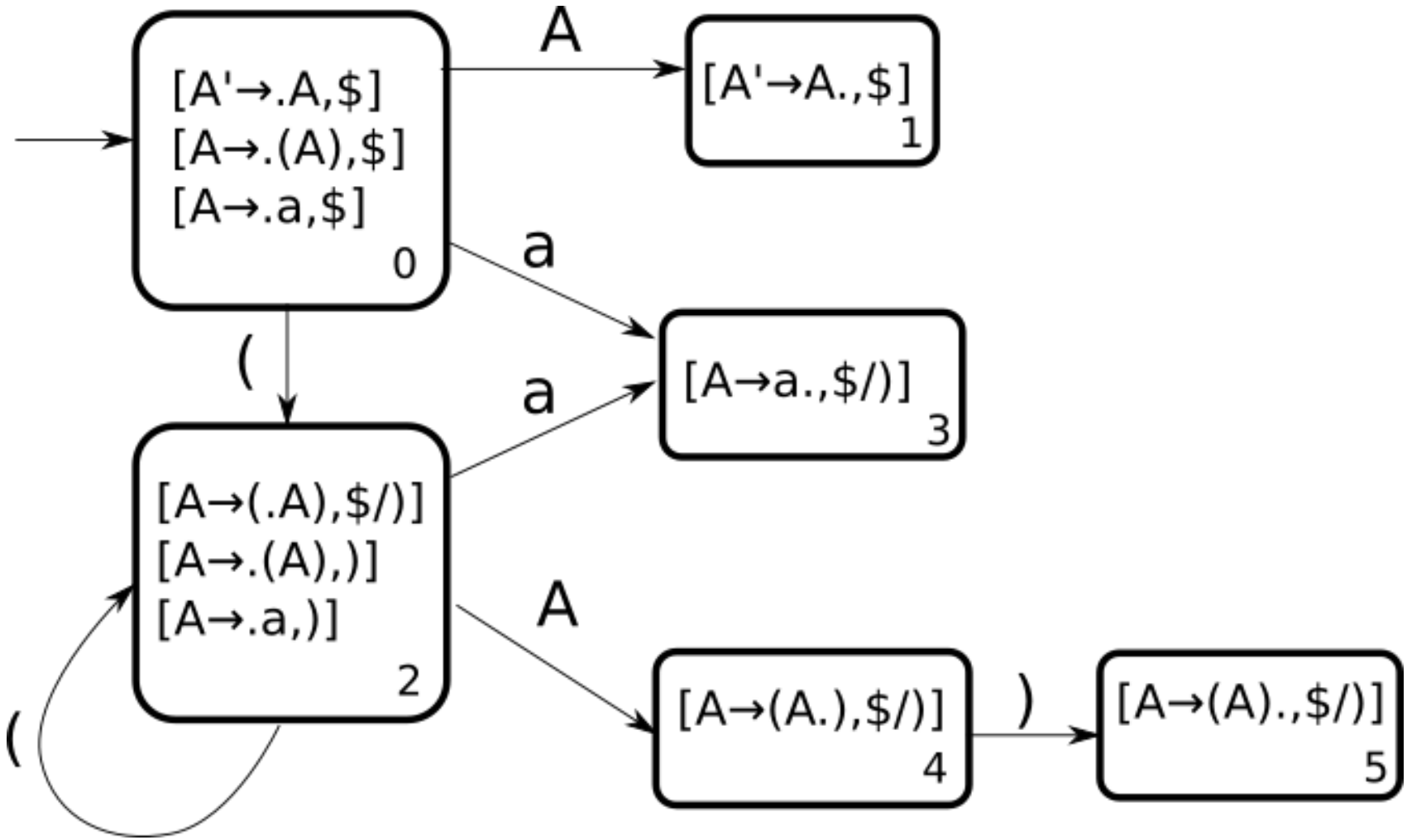
Primeiro Princípio da Análise Sintática LALR(1)

O núcleo de um estado do DFA de itens LR(1) é um estado DFA de itens LR(0).

Segundo Princípio da Análise Sintática LALR(1)

Dados dois estados s_1 e s_2 do DFA de itens LR(1) com o mesmo núcleo, suponha que exista uma transição em X de s_1 para um estado t_1. Portanto, existe também uma transição em X do estado s_2 para o estado t_2, e os estados t_1 e t_2 têm o mesmo núcleo.

DFA de itens LALR(1): a partir do autômato LR(1), identificar todos os estados com o mesmo núcleo e unir todos os símbolos de verificação correspondentes a um item LR(0) em um conjunto.


[A'→.A,$]
[A→.(A),$]
[A→.a,$]
0
A
[A'→A.,$]
1
a
[A→a.,$/)]
3
(
a
[A→(.A),$/)]
[A→.(A),)]
[A→.a,)]
2
A
[A→(A.),$/)]
4
)
[A→(A).,$/)]
5
(

Consequências da Análise LALR(1)

- O algoritmo para análise LALR(1) é o mesmo usado na LR(1);
- é possível que a transformação do autômato em LALR(1) cause conflitos inexistentes no autômato LR(1);
- na maioria das construções de linguagens de programação, esses conflitos não ocorrem;
- a **propagação de verificações à frente** permite computar o autômato LALR(1) diretamente a partir do autômato LR(0).

YACC - *Yet Another Compiler Compiler*

Gerador de Analisadores Sintáticos

Programa que recebe como entrada uma especificação da sintaxe de uma linguagem e produz como saída um procedimento de análise sintática para aquela linguagem.

- **Compilador de compiladores**;
- algoritmo LALR(1);
- Várias versões;
- funciona com C/C++.

YACC - *Yet Another Compiler Compiler*

Instalar a versão Ubuntu:

$ sudo apt install bison

Arquivos de especificação **.y** com a gramática e ações:

```
{definições}
%%
{regras}
%%
{rotinas auxiliares}
```

São gerados os arquivos *y.tab.c* e *y.tab.h* para arquivo de entrada da gramática.

Definições

- Marcas e tipos de dados necessários para a geração do analisador sintático;
- as marcas são constantes inteiras definidas que devem corresponder às marcas geradas pelo analisador léxico;
- você pode recortar e colar essas marcas nos dois cabeçalhos, ou incluir o arquivo de cabeçalho gerado pelo *yacc* nos arquivos gerados pelo *lex*.
- também nesta seção está o código em C que deve ser inserido diretamente no analisador sintático:
  - funções auxiliares;
  - diretivas #include.
- toda a seção de Definições pode estar vazia se o objetivo for apenas estudar a geração da tabela.

Regras

- São as regras da gramática em BNF adaptada;
- para cada regra pode existir uma **ação** associada:
  - ações são código em C que é executado toda vez que o analisador LALR(1) realiza uma **redução** na regra associada;
  - em teoria esse código pode ser qualquer coisa, inclusive a construção de uma árvore sintática.
- como não há o símbolo $\rightarrow$ na maioria dos teclados, usamos : em seu lugar;
- a regra gramatical termina sempre com ponto e vírgula.

Rotinas Auxiliares

- Rotinas auxiliares e declarações de funções;
- código que o desenvolvedor optou por não incluir em um cabeçalho (no caso, seria incluso na seção de Definições);
- pode ser vazia, eliminando a necessidade do segundo símbolo de separação %%;
- os comentários em estilo C são permitidos aqui, assim como nas Regras e Definições.

Exemplo Básico - Gramática de Parênteses

Considere a gramática mais simples que usamos nos exemplos:
$A \rightarrow (A)$
$A \rightarrow a$
Vamos criar um arquivo de entrada para o *yacc* reconhecer cadeias desta gramática:

- Só tem três *tokens*: (, a e);
- portanto, vamos fazer a análise léxica manualmente.

Arquivo *src/yacc/parenteses/parenteses.y*

Seções **Definições** e **Regras**:

```
%{
#include <stdio.h>
#include <ctype.h>
int yylex(void);
int yyerror(char *);
%}

%token LPARAM RPARAM LITTLEA

%%

A : LPARAM A RPARAM { printf("Fazendo redução por A -> (A).\n"); }
   | LITTLEA        { printf("Fazendo redução por A -> a.\n"); }
   ;

%%
```

Arquivo *src/yacc/parenteses/parenteses.y*

Seção **Rotinas Auxiliares**:

```
int main(int argc, char *argv[]) {
   return yyparse();
}

int yylex(void) {
   int c;

   while ((c = getchar()) == ' ');

   if (c == '(' ) return LPARAM;
   else if (c == ')') return RPARAM;
   else if (c == 'a') return LITTLEA;
   else if (c == '\n') return 0;
   return c;
}

int yyerror(char *s) {
   fprintf(stderr, "%s\n", s);
   return 0;
}
```

Geração da Tabela e Compilação do Analisador

```
$ yacc parenteses.y
$ gcc yy.tab.c -o parenteses
$ ./parenteses
((a))
Fazendo redução por A -> a.
Fazendo redução por A -> (A).
Fazendo redução por A -> (A).
```

Criamos os *tokens* LPARAM, RPARAM e LITTLEA. Mas para símbolos de um único caracteres, podemos inseri-los entre aspas:

```
A : '(' A ')'
  | 'a'
  ;
```

Para cenários mais complexos (identificadores, números, etc), precisamos criar os *tokens*.

Opções *yacc*

Se quisermos que as funções geradas pelo *yacc* e outras definições (como as marcas para o *lex*) estejam disponíveis em um cabeçalho:

- $ yacc -d parenteses.y
- arquivo *y.tab.h* é criado.

É possível gerar uma descrição textual do autômato LALR(1):

- $ yacc -v parenteses.y
- o arquivo *y.output* é criado;
- os estados contém apenas os itens de **núcleo**.
- as transições indicam as ações do analisador e o contéudo da tabela IR-PARA.

Compare o conteúdo do arquivo com Figura 5.9 do livro.

Gramática de Expressões Ariméticas para o YACC

Vamos estudar um exemplo um pouco mais robusto:
$exp \rightarrow exp\ soma\ termo | termo$
$soma \rightarrow + | -$
$termo \rightarrow termo\ mult\ fator | fator$
$mult \rightarrow *$
$fator \rightarrow (exp) | \textbf{número}$
Esta gramática foi usada como exemplo em vários capítulos.

Definições - arquivo *src/yacc/aritmetica/aritmetica.y*

```
%{
#include <stdio.h>
#include <ctype.h>

int yylex(void);
int yyerror(char *);
%}

%token NUMBER
```

Regras - arquivo *src/yacc/aritmetica/aritmetica.y*

```
%%
command : exp { printf("%d\n", $1); }
        ; /* permite imprimir o resultado */

exp : exp '+' term { $$ = $1 + $3; }
    | exp '-' term { $$ = $1 + $3; }
    | term { $$ = $1; }
    ;

term : term '*' factor { $$ = $1 * $3; }
     | factor { $$ = $1 ;}
     ;

factor : NUMBER { $$ = $1; }
       | '('exp')' { $$ = $2; }
       ;
```

Rotinas Auxiliares - *src/yacc/aritmetica/aritmetica.y*

```
%%

int main() {
   return yyparse();
}

int yylex(void) {
   int c;
   while ((c=getchar()) == ' '); /* elimina espaços em branco */
   if (isdigit(c)) {
      ungetc(c,stdin);
      scanf("%d", &yylval);
      return(NUMBER);
   }
   if (c == '\n') return 0; /* interrompe a análise sintática */
   return c;
}

/* imprimir a mensagem erro */
int yyerror(char *s) {
   fprintf(stderr, "%s\n", s);
   return 0;
}
```

Gramática de Expressões Ariméticas para o YACC

- O único *token* (NUMBER) é apresentado nas definições;
- os outros terminais são inclusos diretamente na gramática, entre aspas simples (os operadores);
- temos um novo símbolo inicial, command, para deixar a definição do autômato LALR(1) mais organizada;
- as ações fazem uso de **pseudovariáveis**:
  - quando ocorre uma redução por uma regra, o *yacc* atribui um **valor** a cada não-terminal da regra;
  - por padrão, esse **valor** é inteiro, mas pode ser qualquer tipo em C/C++, inclusive ponteiros, estruturas, etc;
  - o *yacc* mantém uma **pilha de valores**, em paralelo com a pilha de análise sintática.

Pseudovariáveis

Considere a regra e ação associada:

```
exp : exp '+' termo {$$ = $1 + $3;}
```

- O símbolo $ indica um valor da pilha de valores;
- o símbolo $$ representa o símbolo à esquerda na regra gramatical;
- As variáveis $1, $2, $3 indicam os valores dos símbolos na primeira, segunda e terceira posição, do lado direito;
- no caso, não existe valor para o segundo símbolo, '+, uma marca terminal;

Neste exemplo, estamos afirmando que o valor da expressão que estará pilha após a redução é igual a soma do valor da expressão do lado direito e termo.

Pseudovariáveis

Considere a regra e ação associada:

```
factor : NUMBER {$$ = $1;}
```

- Neste caso, NUMBER é um terminal, com valor ($1);
- o analisador léxico deve retornar o valor de NUMBER preenchendo a variável *yylval* quando a marca é reconhecida;
- a *yylval* deve ser ma variável global compartilhada entre o análisado léxico e o sintático.

Você pode considerar que *yylval* seria o lexema ou um outro atributo da marca.

Rotinas Auxiliares

- A função *main* ativa *yyparse()*, que retorna 0 em sucesso da análise sintática, 1 caso contrário;
- implicitamente, *yyparse()* invoca uma função *yylex()* que pode ter sido criada pelo *lex*;
- no exemplo, *yylex()* foi feita manualmente, vale observar como *yylval* é preenchida;
- temos também a função *yyerror()*, apenas imprimindo uma mensagem de erro.

Gerando o Autômato LALR(1) para Expressões Aritméticas

Arquivo simplicado, sem as ações e rotinas auxiliares:

```
%token NUMBER
%%
command : exp
          ;
exp : exp '+' term
          | exp '-' term
          | term
          ;
term : term '*' factor
          | factor
          ;
factor : NUMBER
          | '('exp')'
          ;
```

$ yacc -v aritmetica_simplicada.y

Teremos o arquivo *y.output* com o autômato LALR(1). Sem as ações e rotinas, fica mais fácil visualizar as transições dos estados.

Depurando a Execução do Analisador Sintático

Adicionar:

```
#define YYDEBUG 1
```

na seção de Definições, e:

```
extern int yydebug;
yydebug = 1;
```

no começo da função *main*. Agora basta gerar o executável novamente, informando entrada.

Conflitos e Eliminação de Ambiguidade

- O *yacc* tem regras internas para eliminar ambiguidade;
- o arquivo *y.output* permite verificar se essas regras surtiram efeito;
- conflitos **carrega-reduz** são resolvidos para opção carregar;
- conflitos **reduz-reduz** tem a ambiguidade eliminada pela precedência da redução da regra que aparece primeiro no arquivo de especificação.

Considere a gramática visivelmente ambígua:
$A \rightarrow A \mid B$
$A \rightarrow a$
$B \rightarrow a$
Gere o arquivo *y.output* dela.

Precedência de Operadores e Associatividade

```
%{
#include <stdio.h>
#include <ctype.h>
%}

%token NUMBER

%left '+' '-'
%left '*'

%%
command : exp { printf("%d\n", $1); }
        ;
exp     : NUMBER { $$ = $1 ;}
        | exp '+' exp { $$ = $1 + $3 ;}
        | exp '-' exp { $$ = $1 - $3 ;}
        | exp '*' exp { $$ = $1 * $3 ;}
        | '(' exp ')' { $$ = $2;}
        ;
%%
/* Mesmas funções auxiliares do outro exemplo. */
```

Precedência de Operadores e Associatividade

As linhas abaixo:

```
%left '+' '-'
%left '*'
```

Indicam que os operadores + e - têm a mesma precedência e são associativos à esquerda. O operador * é associativo à esquerda e tem precedência maior que os outros. Os operadores %right e %nonassoc também estão disponíveis.

Tipos de Valores Arbitrários

- No exemplo da gramática de expressões aritmeticas, as pseudovariáveis tem valor inteiro (o padrão);
- mas se quisessemos calcular operações de ponto flutuante?
- ou melhor, se quisessemos construir a árvore sintática durante a derivação?
- podemos redefinir o tipo das pseudovariáveis usando o símbolo YYSTYPE:

```
#define YYSTYPE double
```

Tipos de Valores Arbitrários

Considere a gramática:
exp $\rightarrow$ exp soma termo | termo
soma $\rightarrow$ + | -
Podemos ter tipos diferentes para cada não-terminal:

```
%token NUMBER
%union { double val; char op;}
%type <val> exp term factor NUMBER
%type <op> addop mulop
%%
command : exp { printf("%d\n", $1); }
        ;

exp     : exp op termo { switch ($2) {
                           case '+' : $$ = $1 + $3; break;
                           case '-' : $$ = $1 - $3; break;
                           }
                       }
        | term { $$ = $1; }

op      : '+' { $$ = '+'; }
        | '-' { $$ = '-'; }
        ;
```

Ações Embutidas

Veja a gramática:
decl $\rightarrow$ *tipo var-lista*
tipo $\rightarrow$ **int** | **float**
var-lista $\rightarrow$ *var-lista*, **id** | **id**

- Pela ordem da análise, a redução a *tipo* ocorrerá antes do processamento de *var-lista*;
- gostariamos que o valor da pseudovariável de *tipo* (**int** ou **float**) pudesse ser propagado para *var-list*;
- \$2 = \$1 é proibido!!!
- vamos colocar uma **ação embutida** para adicionar o valor de *tipo* a uma variável global.

Ações Embutidas

```
decl : tipo { tipo_corrente = $1; } var-lista
     ;
tipo : INT { $$ = TIPO_INTEIRO; }
     | FLOAT { $$ = TIPO_FLOAT; }
     ;
var-lista : var-lista ',' ID
                {configuraTipo(tokenString, tipo_corrente);}
          | ID {configuraTipo(tokenString, tipo_corrente);}
          ;
```

O *yacc* interpreta uma ação embutida:

```
A : B { /* ação embutida */ } C;
```

como:

```
A : B E C;
E: {/* ação embutida */}
```

Resumo YACC

Nome Interno	Significado/Uso
y.tab.c	Nome do arquivo de saída
y.tab.h	Arquivo de cabeçalho com definição de marcas
yyparse	Rotina do analisador sintático
yylval	Valor da marca corrente na pilha
yyerror	Função padrão de erro
error	Pseudomarca de erro
yyerrork	Procedimento que reinicia a análise após erro
yychar	Marca à frente que causou erro
YYSTYPE	Tipo do valor na pilha
yydebug	Variável que ativa a depuração
Mecanismo	**Significado/Uso**
%token	Define símbolos para marcas
%start	Define símbolo inicial
%union	União YYSTYPE
%type	Define o tipo diferenciado da união para não-terminal
%left %right %nonassoc	Associatividade e precedência

Analisador Sintático para TINY em YACC

- No apêndice B, há a listagem do código do analisador sintático para TINY em *yacc* (tiny.y);
- também há uma versão alterada do arquivo globals.h, já que as marcas agora são definidas pelo *yacc*;
- iremos construir a árvore sintática, portanto YYSTYPE passa a ser do tipo *TreeNode **;
- *savedName* e *savedLineNo* servem para anotar informações sobre variáveis ao serem atribuídas.

O comportamento geral de cada ação é construir o nó correspondente da árvore.

```
write_stmt : WRITE ID {
               $$ = newStmtNode(WriteK);
               $$->child[0] = $2;
             }
             ;
```

Analisador Sintático para TINY em YACC

No caso da atribuição, precisamos de uma ação embutida para salvar o nome e a linha da variável.

```
assign_stmt : ID {
  savedName = copyString(tokenString);
  savedLineNo = lineno;
} ASSIGN exp {
  $$ = newStmtNode(AssignK);
  $$->child[0] = $4;
  $$->attr.name = savedName;
  $$->lineno = savedLineNo;
}
;
```

Importante lembrar que a função *newStmtNode* está definida em *globals.h*. Este arquivo também precisa ser adaptado para funcionar com o Yacc.

Recuperação de Erros

- Como vimos na discussão dos algoritmos LR(0), SLR(1), LR(1) e LALR(1), um analisador sintático ascendente detecta erro quando a análise leva uma entrada na tabela vazia, sem ação;
- entretanto, um objetivo importante é reduzir o tamanho da tabela;
- rapidez na detecção de erros:
  1. LR(1)
  2. LALR(1)
  3. SLR(1)
  4. LR(0)

Recuperação de Erros

Estado	Entrada				Ir-para
	(	a	)	$	A
0	s2	s3			1
1				aceita	
2	s5	s6			4
3				r2	
4			s7		
5	s5	s6			8
6			r2		
7				r1	
8			s9		
9			r1		

Tabela: LR(1)

Estado	Ação	Regra	Entrada			Ir-para
			(	a	)	A
0	carrega		3	2		1
1	reduz	$A' \rightarrow$A				
2	reduz	A$\rightarrow$a				
3	carrega		3	2		4
4	carrega				5	
5	reduz	A$\rightarrow$(A)				

Tabela: LR(0)

Comportamento para as entradas:

1. (a$
2. a)$

Quem retorna erro mais rápido?

Recuperação de Erros

Continuamos com três opções:

1. Retirar um estado da pilha;
2. retirar sucessivamente marcas da entrada até encontrar uma que possa reiniciar a análise;
3. colocar um novo estado na pilha.

Método geral:

1. Retire estados da pilha até encontrar um estado com entradas Ir-Para não vazias;
2. se existir uma ação na marca da entrada, reinicie a análise. Dê preferência a carregar em vez de reduzir, e entre as reduções, escolha àquelas com não terminais menos gerais.
3. não havendo ação legal para a entrada, avance a entrada até uma ação legal ou \$.

Recuperação de Erros

- Assim como no caso dos analisadores descendentes, não há técnica perfeita;
- novamente, uma abordagem **modo pânico**;
- como existe um autômato, o analisado sabe por qual caminho retornar na presença de erros;
- o YACC permite ao desenvolvedor definir funções para o tratamento de erro de acordo com a gramática.

Fim

Dúvidas?