

Compiladores: Cheat Sheet

Eduardo Morgado

fevereiro, 2020

Conteúdo:

1	Introdução	2
2	Top-Down	2
2.1	Nullable	3
2.2	First	3
2.3	Follow	4
2.3.1	Resolver restrições	4
2.3.2	Calcular Follow	4
2.4	Left-To-Right LL(1)	5
2.5	Aplicação de LL(1)	6
3	Bottom-Up	7
3.1	LR(0) - Um exemplo	7
3.2	LR(1) - Um exemplo	7
4	Alocação de Registos	8
5	Optimizações	8

Figuras:

1	Fases de Compilação	9
2	Gramática 1	10
3	Gramática 2	10
4	Gramática 3	10
5	Autômato LR(0) da gramática 3	10
6	Autômato LR(1) da gramática 3	10

Tabelas:

1	Restrições a gramática 2	11
2	Tabela de correspondências de gramática 2	11
3	Tabela de análise LL(1) para gramática, com entrada ()\$ e tabela de correspondências 2	11
4	Tabela de <i>parsing</i> LR(0)	12
5	<i>Parsing</i> LR(0) a <i>input</i> ((a))\$	12
6	Tabela de <i>parsing</i> LR(1)	12
7	<i>Parsing</i> LR(1) a <i>input</i> ((a))\$	13

1 Introdução

O objectivo deste documento é apresentar, de forma resumida, os temas referentes à disciplina de Compiladores, no entanto, não irão ser abordados os seguintes temas:

- Utilização de expressões regulares
- Definição de Gramáticas Independentes de Contexto
- Remoção de Ambiguidade em CFG (*context free grammars*)
- Geração de autómatos a partir de expressões regulares
- Minimização de autómatos
- Programação avançada em C¹

Será também importante referir que os temas apresentados neste documentos iram ser abordados de uma forma mais superficial, uma vez que, para uma abordagem mais pormenorizada deverá ser utilizado o outro documento "Compiladores: Um Guia para uma compilação simples de Rust"².

2 Top-Down

Durante a compilação existem várias fases que podem ser observadas na figura de [fase de compilação](#), a análise **top-down** está inserida na fase de análise sintáctica, nesta fase iremos abordar dois tipos de análise e *parsing*:

- Análise Top-Down - LL
- Analise Bottom-Up - LR

¹Deve eixtir uma familiaridade e conforto com utilização de **structs**, **eums** e **unions**.²

²É de referir que este documento pode não esta completo ou finalizado dependendo da data de consulta, uma vez que, se tratam de apontamentos retirados num contexto de aula e com recurso ao livro "Introduction to Compiler Design" de Torben Ægidius Mogense, pelo que, é recomendado a consulta desse livro.

É importante referir que, ambos estes métodos utilizam a gramática da linguagem gerada na fase de análise sintáctica (esta gramática geralmente é criada com recurso a código **flex** e **bison**).

Estes dois tipos de análise diferem na forma como percorrem o *input* para gerar a **árvore sintáctica abstracta** (AST) da linguagem que reconhece o *input*. Ambos utilizam a gramática gerada de forma diferente, no Top-Down, a partir da produção inicial iremos percorrer as produções, implementado mecanismos que facilitam a escolha de produções baseadas no símbolo de *input* a ser analisado, caso seja possível, através das produções da gramática, consumir todos os símbolos de entrada, então esta gramática e por consequência a sua linguagem reconhecem a palavra de entrada, no Top-Down começamos a partir da raiz da AST numa tentativa de chegar a folhas. Já no Bottom-Up, a partir de folhas na AST tentamos chegar à raiz da AST, caso isso aconteça, a gramática reconhece a palavra.

Para poderem ser aplicados algoritmos Top-Down é necessário primeiro introduzir conceitos de **nullable**, **first** e **follow**.

2.1 Nullable

Considerando α como uma sequência de símbolos na gramática,

$$Nullable(\alpha) \quad sse \quad \alpha \Rightarrow \varepsilon$$

ou seja, α é *nullable* se derivar a palavra vazia, por exemplo:

- $Nullable(\varepsilon) = \text{true}$
- $Nullable(a) = \text{false}$ (a é símbolo terminal)
- $Nullable(\alpha\beta) = Nullable(\alpha) \wedge Nullable(\beta)$
- $Nullable(N) = Nullable(\alpha_1) \vee \dots \vee Nullable(\alpha_n)$
onde $N \longrightarrow \alpha_1 \mid \dots \mid \alpha_n$

2.2 First

First retorna o conjunto de símbolos **iniciais** das *strings* derivadas por cada símbolo de cada produção, ou seja, $\gamma \in \text{First}(\alpha)$ sse $\alpha \Rightarrow * \gamma\beta$, por exemplo:

- $\text{First}(\varepsilon) = \emptyset$
- $\text{First}(a) = \{a\}$
- $\text{First}(\alpha\beta) = \begin{cases} \text{First}(\alpha) \cup \text{First}(\beta) & \text{se } Nullable(\alpha) \\ \text{First}(\alpha) & \text{se } \alpha \notin Nullable(\alpha) \end{cases}$
- $\text{First}(N) = \text{First}(\alpha_1) \cup \dots \cup \text{First}(\alpha_n)$
onde $N \longrightarrow \alpha \mid \dots \mid n$

α e β são sequências de símbolos, a é símbolo terminal e N símbolo não terminal

2.3 Follow

$\text{Follow}(N)$ é o conjunto das produções que podem directa ou indirectamente utilizar/seguir a regra N , dessa forma $\alpha \in \text{Follow}(N)$ sse $S \Rightarrow^* \alpha N a \beta$, ou seja, α está no conjunto $\text{Follow}(N)$ se numa altura da derivação, α segue a regra N .

A forma mais fácil de calcular o conjunto Follow é gerar o **conjunto e tabela de restrições**.

2.3.1 Resolver restrições

Para resolver as restrições de uma gramática devem ser seguidos estes passos:

1. Considerar conjuntos Follow vazios para todos os símbolos não terminais.
2. Resolver primeiro restrições da forma $\text{First}(\beta) \subseteq \text{Follow}(N)$ ³:
 - 2.1. Calcular $\text{First}(\beta)$
 - 2.2. Adicionar $\text{First}(\beta)$ a $\text{Follow}(N)$
3. Depois de todas as restrições 2 serem resolvidas, avançar para passo 4
4. Resolver restrições da forma $\text{Follow}(M) \subseteq \text{Follow}(N)$ ⁴:
 - 4.1. Adicionar Todos os elementos de $\text{Follow}(M)$ a $\text{Follow}(N)$
 - 4.2. Repetir até resolver todos os casos

2.3.2 Calcular Follow

Estes são os passos a seguir para calcular Follow:

1. Adicionar $S' \rightarrow S\$$ à gramática como nova produção inicial
2. Para cada símbolo **não terminal** N , localizar todas as suas ocorrências na gramática e para cada uma realizar os seguintes passos:
 - 2.1. Para produções no formato $M \rightarrow \alpha N \beta$, onde β pode ser vazio e M pode ser igual a N :
 - 2.1.1. Se $\beta \notin \text{Nullable}(\beta)$, $\mathbf{m} = \mathbf{First}(\beta)$ e é adicionada a restrição $\mathbf{m} \subseteq \mathbf{Follow}(N)$
 - 2.1.2. Se $\beta \in \text{Nullable}(\beta)$ e se o símbolo M for **diferente** de N adicionar a restrição $\mathbf{Follow}(M) \subseteq \mathbf{Follow}(N)$
3. Resolver as restrições segundo os passos apresentados anteriormente.

³Neste caso β é uma sequência de símbolos e N um símbolo não terminal.

⁴Neste caso, tanto M como N são símbolos não terminais.

2.4 Left-To-Right LL(1)

Existem vários algoritmos Top-Down, iremos analisar o Left-to-Right, com um *lookahead*⁵. Este algoritmo escolhe as produções da seguinte forma, seja $N \rightarrow \alpha$, onde α é uma sequência de símbolos, esta produção é escolhida para o input c se:

- $c \in \text{First}(\alpha)$
ou
- $\text{Nullable}(\alpha) \wedge c \in \text{Follow}(N)$

Este algoritmo pode ser implementado de duas formas, no entanto, iremos implementar o algoritmo LL(1) com recurso a uma **tabela** de correspondências de símbolos terminais a símbolos não terminais, este método não é recursivo e utiliza uma **pilha**.

O algoritmo segue os seguintes passos:

- Se o topo da pilha contiver um símbolo **terminal**, e se for igual ao símbolo de entrada atual, este é satisfeito e é removido da pilha (*pop*)
- Se o topo da pilha contiver um símbolo **não terminal**, deve ser consultada a tabela LL(1) com base no *lookahead*, caso:
 - a Não exista uma entrada na tabela para esse símbolo então a análise gera um erro, a produção caso seja escolhida irá criar um erro na derivação
 - b Caso exista uma entrada na tabela, o símbolo terminal é removido e a sua produção é inserida na pilha de forma a que o primeiro símbolo da produção seja o topo da pilha

A construção de *parsers* LL(1) procede-se da seguinte forma:

1. Eliminar ambiguidade na gramática
2. Eliminar recursividade à esquerda na gramática
3. Caso seja necessário, factorizar a gramática à esquerda
4. Adicionar nova regra inicial $S' \rightarrow S\$$ à gramática
5. Calcular First para todas as produções
6. Calcular Follow para todos os símbolos não terminais
7. Para símbolo não terminal N e produção $N \rightarrow \alpha$ e símbolo de entrada c , escolher essa produção quando:
 - i $c \in \text{First}(\alpha)$
ou
 - ii $\text{Nullable}(\alpha) \wedge c \in \text{Follow}(N)$
guardar a escolha na tabela de correspondências

⁵Isto significa que, iremos, para além de se verificar o símbolo actual na sequência, irá ser também considerado o próximo símbolo na sequência de entrada.

2.5 Aplicação de LL(1)

Considerar a gramática 1 como a gramática utilizada para a aplicação do algoritmo e considerar como sequência de entrada a palavra ().

Aplicar LL(1):

1. Primeiro a gramática deve ser modificada para a gramática 2
2. Gerar tabela de restrições, para esta gramática o único símbolo não terminal é S coma as produções:
 - 2.1. $S \rightarrow (S)S$ Para esta produção $\beta =)S$
 $m = \text{First}()S) = \{) \} \subseteq \text{Follow}(S)$
 - 2.2. $S \rightarrow \varepsilon$
 - 2.3. $S' \rightarrow S\$$ Para esta produção $\beta = \$$
 $m = \text{First}(\$) = \{ \$ \} \subseteq \text{Follow}(S)$
3. Gerar a tabela de [restrições 1](#)
4. resolver restrições
 - 4.1. Para restrições do formato $m \subseteq \text{Follow}(N)$, $m = \text{First}(\beta)$, dessa forma as restrições
$$\begin{cases} \{ \$ \} \subseteq \text{Follow}(S) \\ \{) \} \subseteq \text{Follow}(S) \end{cases}$$
são transformadas em $\{ \$,) \} \subseteq \text{Follow}(S)$
5. Gerar tabelas de correspondências para símbolos não terminais (,) e \$, considerar entradas:
 - $M[S, (] = S \rightarrow (S)S$
 - a Considerar $N = S$, $\alpha = (S)S$ e $c = ($, $(\in \text{First}(\alpha)$, $(\in \text{First}((S)S)$ logo a produção $S \rightarrow (S)S$ é escolhida
 - $M[S,)] = S \rightarrow \varepsilon$
 - a Considerar $N = S$, $\alpha = (S)S$ e $c =)$
 - i $) \notin \text{First}(\alpha)$
 - ii $\text{Nullable}((S)S) \wedge) \in \text{Follow}(S)$, $\text{false} \wedge \text{false} = \text{false}$
 - b Considerar $N = S$, $\alpha = \varepsilon$, $c =)$
 - i $) \notin \text{First}(\varepsilon)$
 - ii $\text{Nullable}(\varepsilon) \wedge) \in \text{Follow}(S)$, $\text{true} \wedge \text{true} = \text{true}$, logo $S \rightarrow \varepsilon$ é escolhida
 - $M[S, \$] = S \rightarrow \varepsilon$
 - a Considerar $N = S$, $\alpha = (S)S$ e $c = \$$
 - i $\$ \notin \text{First}(S)$
 - ii $\text{Nullable}(\alpha) = \text{false}$, produção não é escolhida
 - b Considerar $N = S$, $\alpha = \varepsilon$, $c = \$$
 - i $\$ \notin \text{First}(S)$
 - ii $\text{Mullable}(\alpha) \wedge \$ \in \text{Follow}(S)$, $\text{true} \wedge \text{true} = \text{true}$ logo produção $S \rightarrow \varepsilon$ é escolhida
6. Gerar a [tabela de correspondências 2](#)
7. Aplicar a tabela e o algoritmo LL(1) ao *input* () obtendo a tabela de [análise 3](#)

3 Bottom-Up

Neste tipo de análise, o objectivo é, a partir das folhas da AST, tentar chegar à produção inicial, caso seja possível, a gramática e por consequência, a sua linguagem reconhece a palavra de entrada. Neste tipo de algoritmos iremos focar apenas no Left-To-Right LR(k), onde k especifica a quantidade de símbolos *lookahead* onde a complexidade do algoritmo será $O(n^k)$, apenas iremos focar no LR(0) e LR(1).

O algoritmo apresenta três passos:

- Adicionar nova produção inicial $S' \rightarrow S$ à gramática
- Gerar autómato a partir da gramática
- Gerar a tabela de *parsing* LR

3.1 LR(0) - Um exemplo

Para este exemplo iremos utilizar a gramática 3. No primeiro passo iremos gerar o autómato a partir da [gramática 3](#), [autmato 1](#), irá depois ser gerada a tabela de [parsing LR\(0\)](#).

Depois de gerar a tabela LR(0) podemos aplicar o algoritmo a um *input*, por exemplo $((a))\$$, obtemos a tabela de [análise LR\(0\)](#), sendo a palavra aceite.

Nota: Um SHIFT acontece se \cdot não estiver no fim da produção, ou seja, ainda falta consumir mais símbolos. Um REDUCE acontece se \cdot estiver no fim da produção, ou seja, já foram consumidos todos os símbolos necessários para se poder reduzir a produção ao símbolo não terminal que a origina. Um conflito **SHIFT/REDUCE** acontece se \cdot ainda não estiver no fim estando também no fim simultâneamente. Caso uma LR(k) produza um conflito, por regra geral o LR(k+1) resolve esse conflito.

3.2 LR(1) - Um exemplo

Para este exemplo iremos utilizar a gramática 3. No primeiro passo iremos gerar o autómato a partir da [gramática 3](#), [autómato 2](#), irá depois ser gerada a tabela de [parsing LR\(1\)](#) com a seguinte notação:

- Regras:
 - 1: $A' \rightarrow A$
 - 2: $a \rightarrow (A)$
 - 3: $A \rightarrow a$
- SHIFT para estado $x \rightarrow S-x$
- REDUCE da produção $x \rightarrow R-x$

Depois de gerar a tabela LR(1) podemos aplicar o algoritmo a um *input*, por exemplo $((a))\$$, obtemos a tabela de [análise LR\(1\)](#), sendo a palavra aceite.

4 Alocação de Registos

Para um conjunto de instruções MIPS dispomos de um número **limitado** de registos, dessa forma, a escolha e utilização dos mesmos no processo de geração de código Assembler deve ser **eficiente**, dessa forma é utilizado o **grafo de interferência**⁶, este grafo pode ser analisado por algoritmos de coloração aproximados sendo gerada a matriz de variáveis, por exemplo, caso o grafo apresente três variáveis, a, b e c e existam caminhos bidireccionais (a,c), (c,b), a matriz de variáveis será a [tabela 8](#).

Este algoritmo de coloração apresenta quatro fases:

1. **Simplificação**: Neste passo se possuírmos **k** registos, é escolhido arbitrariamente um nó no grafo de interferências com menos que k vizinhos, retirando esse nó do grafo e inserindo-o numa pilha.
2. **Spill**: Nós com grau maior ou igual a k ⁷ são colocados em memória.
3. **Select**: É feito *pop* à pilha e atribuída cor ao nó.
4. **Substituição**: O código é construído com os registos em memória.

Nota: Este algoritmo é executado num ciclo com um número arbitrário de iterações.

5 Optimizações

A optimização é um processo fundamental para aumentar a eficiência de um compilador, as fases de optimização ocorrem durante o processo de optimização nas seguintes fases:

- Formação de AST
- Geração de Código intermédio
- Geração de código Assembler

Existem dois tipos de optimizações:

1. Locais:
 - 1.1. Eliminação de Instruções redundantes
 - 1.2. Optimização de Controlo de Fluxo
 - 1.3. Simplificações Algébricas
 - 1.4. Redução de Custo no Assembler
2. Globais:
 - 2.1. Function-Preserving Transformations
 - 2.2. Copy Propagation
 - 2.3. Loop Optimization

⁶Este algoritmo verifica se duas variáveis estão "vivas" ao mesmo tempo

⁷Ligados a um maior numero de varáveis que o número de registos

⁸Regra $A \rightarrow a$.

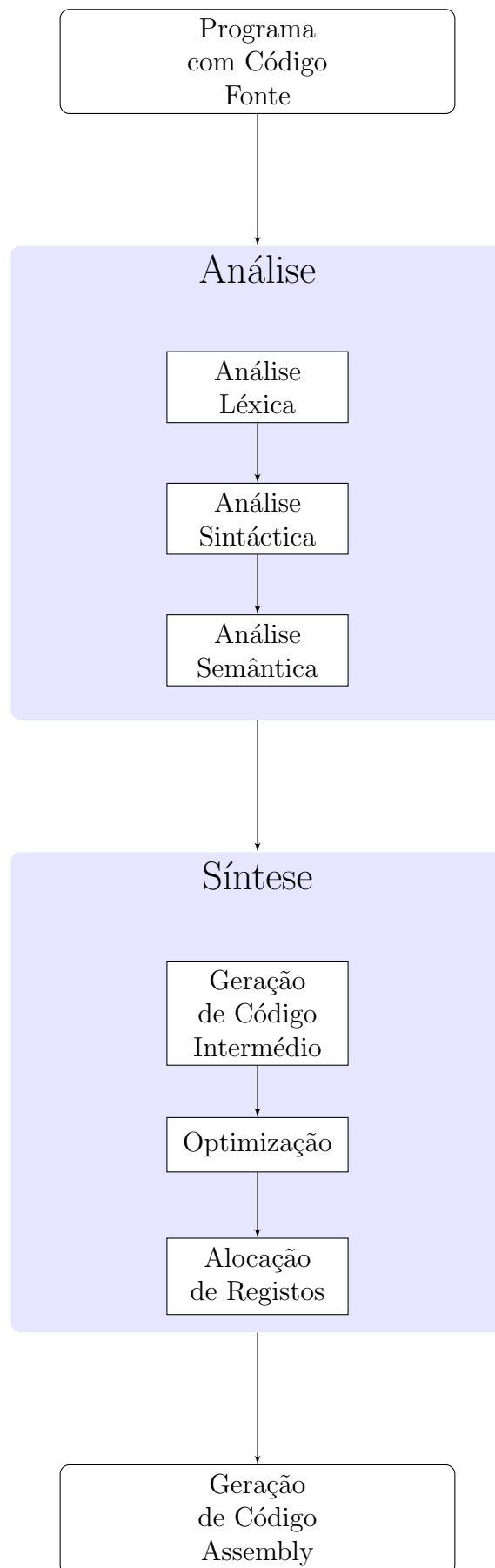


Figure 1: Fases de Compilação

$$S \longrightarrow (S)S \mid \varepsilon$$

Figure 2: Gramática 1

$$\begin{aligned} S' &\longrightarrow S\$ \\ S &\longrightarrow (S)S \mid \varepsilon \end{aligned}$$

Figure 3: Gramática 2

$$\begin{aligned} A' &\longrightarrow A \\ A &\longrightarrow (A) \mid a \end{aligned}$$

Figure 4: Gramática 3

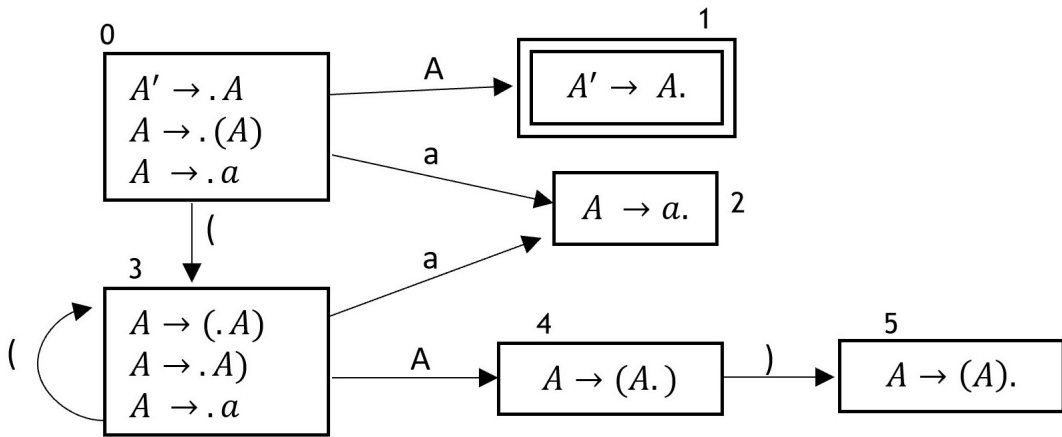


Figure 5: Autômato LR(0) da [gramática 3](#)

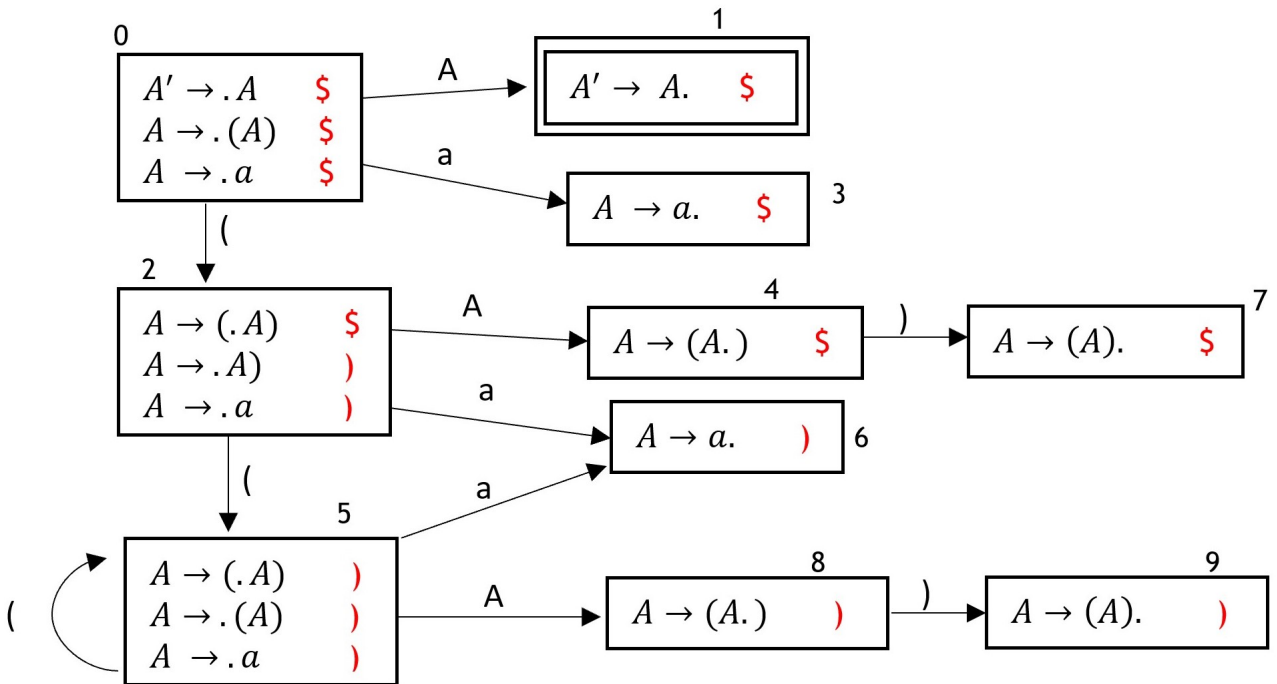


Figure 6: Autômato LR(1) da gramática 3

Produções	Restrições
$S' \rightarrow S\$$	$\{\$ \} \subseteq FOLLOW(S)$
$S \rightarrow (S)S$	$\{ \} \subseteq FOLLOW(S)$
$S \rightarrow \varepsilon$	

Table 1: Restrições a [gramática 2](#)

	()	\$
S'	$S' \rightarrow S\$$	$S' \rightarrow S\$$	$S' \rightarrow S\$$
S	$S \rightarrow (S)S$	$S \rightarrow \varepsilon$	$S \rightarrow \varepsilon[1ex] \text{ height}$

Table 2: Tabela de correspondências de [gramática 2](#)

Stack	Input	Action
S \$	()\$	$S \rightarrow (S)S$
(S) S \$	()\$	match
S) S \$)\$	$S \rightarrow \varepsilon$
) S \$)\$	match
S \$	\$	$S \rightarrow \varepsilon$
\$	\$	ACCEPT (pilha vazia)

Table 3: Tabela de análise LL(1) para [gramática](#), com entrada ()\$ e [tabela de correspondências 2](#)

⁹Como no estado 3 existe a opção goto A e foi obtido o estado A, seguimos esse estado para o 4

¹⁰Regra $A \rightarrow (A)$.

¹¹Regra $A \rightarrow (A)$.

¹²goto A, para estado 1

¹³Regra $A' \rightarrow A$.

¹⁴*Parsing* atinge raiz de AST tendo percorrido todo o *input*, sendo assim a linguagem aceita a palavra

Estados	Input			Acção	Regra	Goto A
	(a)			
0	3	2		SHIFT		1
1			REDUCE	$A' \rightarrow A$		
2			REDUCE	$A \rightarrow A$		
3	3	2		SHIFT		4
4			5	SHIFT		
5			REDUCE	$A \rightarrow (A)$		

Table 4: Tabela de *parsing* LR(0)

Stack	Input	Acção
\$0	((a))\$	SHIFT
\$0(3	(a))\$	SHIFT
\$0(3(3	a))\$	SHIFT
\$0(3(3a2))\$	REDUCE ⁸
\$0(3(3A4 ⁹))\$	SHIFT
\$0(33A4)5)\$	REDUCE ¹⁰
\$0(3A4)\$	SHIFT
\$0(3A4)5	\$	REDUCE ¹¹
\$0A1 ¹²	\$	REDUCE ¹³
\$A'	\$	ACCEPT ¹⁴

Table 5: *Parsing* LR(0) a *input* ((a))\$

Estados	Input				Goto A
	(a)	\$	
0	S-2	S-3			S-1
1				ACCEPT	
2	S-5	S-6			S-4
3				R-3	
4			S-7		
5	S-5	S-6			S-8
6			R-3		
7				R-2	
8			S-9		
9			R-2		

Table 6: Tabela de *parsing* LR(1)

Stack	Input	Acção	Lookahead
\$0	((a))\$	SHIFT	\$
\$0(2	(a))\$	SHIFT	\$
\$0(2(5	a))\$	SHIFT)
\$0(2(5a6))\$	REDUCE)
\$0(2(5A8))\$	SHIFT)
\$0(2(5A8)9)\$	SHIFT)
\$0(2A4)\$	REDUCE	\$
\$0(2A4)7	\$	SHIFT	\$
\$0A	\$	REDUCE	\$
\$0A1	\$	ACCEPT	

Table 7: *Parsing LR(1) a input ((a))\$*

	a	b	c
a	0	0	1
b	0	0	1
c	1	1	0

Table 8: Matriz de Variáveis