Compiladores: Cheat Sheet

Eduardo Morgado

fevereiro, 2020

Conteúdo:

Figuras:

Tabelas:

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 C[1]

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

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

[2] É 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,

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

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

- Nullable(ε) = true
- Nullable(a) = false (a é símbolo terminal)
- Nullable($\alpha\beta$) = Nullable(α) $\wedge$ Nullable(β)
- Nullable(N) = Nullable(α_1) $\vee$ ... $\vee$ Nullable(α_n)
  onde N$\longrightarrow \alpha_1 \mid ... \mid \alpha_n$

2.2 First

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

- First(ε) = $\emptyset$
- First(a) = {a}
- First($\alpha\beta$) = $\begin{cases} First(\alpha) \cup \text{First}(\beta) & \text{se Nullable}(\alpha) \\ First(\alpha) & \text{se } \alpha \notin \text{Nullable}(\alpha) \end{cases}$
- First(N) = First(α_1) $\cup$ ... $\cup$ First(α_n)
  onde N$\longrightarrow \alpha \mid ... \mid n$

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

2.3 Follow

Follow(N) é o conjunto das produções que podem directa ou indirectamente utilizar/seguir a regra N, dessa forma $\alpha \in$ Follow(N) sse S $\Longrightarrow^{*} \alpha$Na$\beta$, ou seja, α está no conjunto 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 First(β) $\subseteq$ Follow(N)[3]:
   2.1. Calcular First(β)
   2.2. Adicionar First(β) a Follow(N)
3. Depois de todas as restrições 2 serem resolvidas, avançar para passo 4
4. Resolver restrições da forma Follow(M) $\subseteq$ Follow(N)[4]:
   4.1. Adicionar Todos os elementos de Follow(M) a 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'$\longrightarrow$ 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$\longrightarrow \alpha$Nβ, onde β pode ser vazio e M pode ser igual a N:
      2.1.1. Se $\beta \notin$ **Nullable(β)**, **m = First(β)** e é adicionada a restrição **m $\subseteq$) Follow(N)**
      2.1.2. Se $\beta \in$ **Nullable(β)** e se a o símbolo M for **diferente** de N adicionar a restrição **Follow(M) $\subseteq$ Follow(N)**
3. Resolver as restrições segundo os passos apresentados anteriormente.

[3]Neste caso β é uma sequência de símbolos e N um símbolo não terminal.
[4]Neste 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$\longrightarrow \alpha$, onde α é uma sequência de símbolos, esta produção é escolhida para o input **c** se:

- c $\in$ First(α)
  ou
- Nullable(α) $\wedge$ c $\in$ 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'$\longrightarrow$ 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$\longrightarrow \alpha$ e símbolo de entrada **c**, escolher essa produção quando:
   i. c $\in$ First(α)
   ou
   ii. Nullable(α) $\wedge$ c $\in$ Follow(N)
   guardar a escolha na tabela de correspondências

[5]Isto significa que, iremos, para alem 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 $\longrightarrow$ (S)S Para esta produção β =)S
     m = First(**)S**) = {)} $\subseteq$ Follow(S)
   - 2.2. S $\longrightarrow \varepsilon$
   - 2.3. S' $\longrightarrow$ S$ Para esta produção β = $
     m = First($) = {$} $\subseteq$ Follow(S)
3. Gerar a tabela de restrições 1
4. resolver restrições
   - 4.1. Para restrições do formato m $\subseteq$ Follow(N), m = First(β), dessa forma as restrições $\begin{cases} \{\$\} \subseteq \text{Follow(S)} \\ \{)\} \subseteq \text{Follow(S)} \end{cases}$ são transformadas em {$,)} $\subseteq$ Follow(S)
5. Gerar tabelas de correspondências para símbolos não terminais (,) e $, considerar entradas:
   - M[S,(] = S $\longrightarrow$ (S)S
     a. Considerar N = S, α) (S)S e c = (, ($\in$ First(α), ($\in$ First(**(S)S**) logo a produção S$\longrightarrow$ (S)S é escolhida
   - M[S,)] = S $\longrightarrow \varepsilon$
     a. Considerar N = S, α = (S)S e c =)
        i.) $\notin$ First(α)
        ii. Nullable(**(S)S**) $\wedge$) $\in$ Follow(S), false $\wedge$ false = false
     b. Considerar N = S, $\alpha = \varepsilon$, c =)
        i.) $\notin$ First(ε)
        ii. Nullable(ε) $\wedge$) $\in$ Follow(S), true $\wedge$ true = true, logo S $\longrightarrow \varepsilon$ é escolhida
   - M[S,$] = S $\longrightarrow \varepsilon$
     a. Considerar N = S, α = (S)S e c = $
        i. $ $\notin$ First(S)
        ii. Nullable(α) = false, produção não é escolhida
     b. Considerar N = S, $\alpha = \varepsilon$, c = $
        i. $ $\notin$ First(S)
        ii. Mullable(α) $\wedge$ $ $\in$ Follow(S), true $\wedge$ true = true logo produção S $\longrightarrow \varepsilon$ 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 quatidade 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'$\longrightarrow$ 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 **.** não estiver no fim da produção, ou seja, ainda falta consumir mais símbolos. Um REDUCE acontece se **.** 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 **.** 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' $\longrightarrow$ A
  - 2: a $\longrightarrow$ (A)
  - 3: A $\longrightarrow$ a
- SHIFT para estado x $\longrightarrow$ S-x
- REDUCE da produçao x $\longrightarrow$ 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 possuirmos **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

[6]Este algoritmo verifica se duas variáveis estão "vivas" ao mesmo tempo
[7]Ligados a um maior numero de varáveis que o número de registos
[8]Regra A $\longrightarrow$ a.

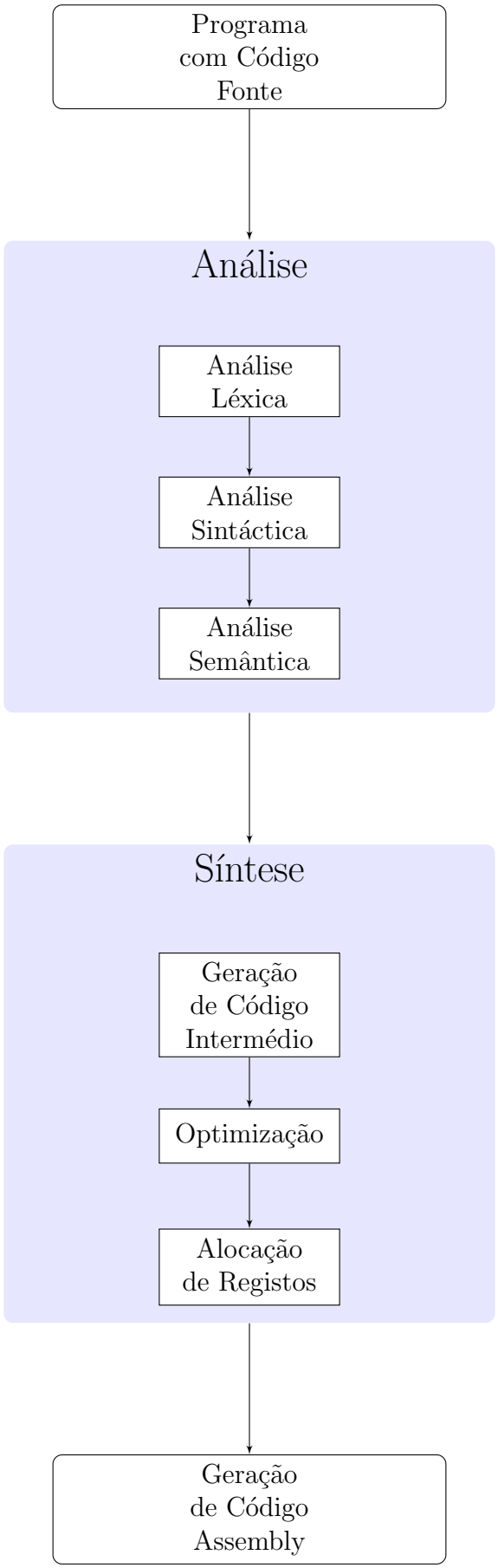

Figure 1: Fases de Compilação

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

Figure 2: Gramática 1

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

Figure 3: Gramática 2

$$A' \longrightarrow A$$
$$A \longrightarrow (A) \mid a$$

Figure 4: Gramática 3

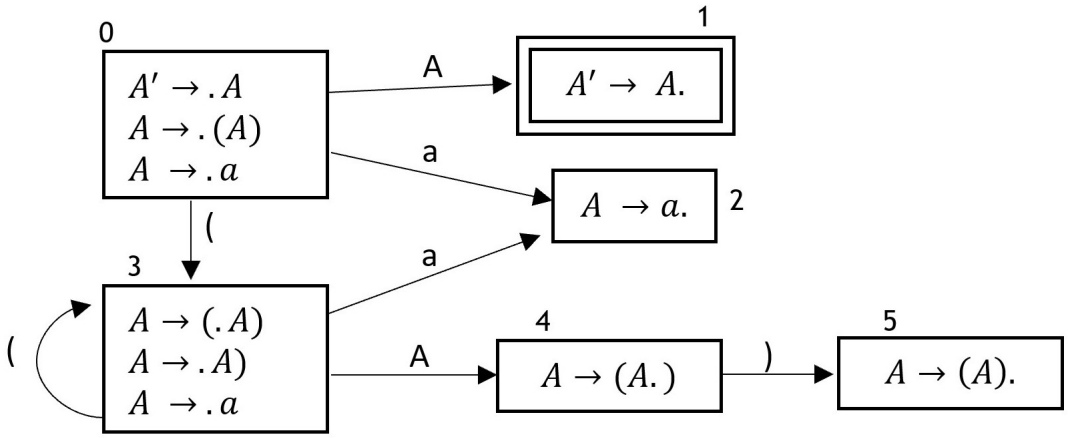

Figure 5: Autómato LR(0) da gramática 3

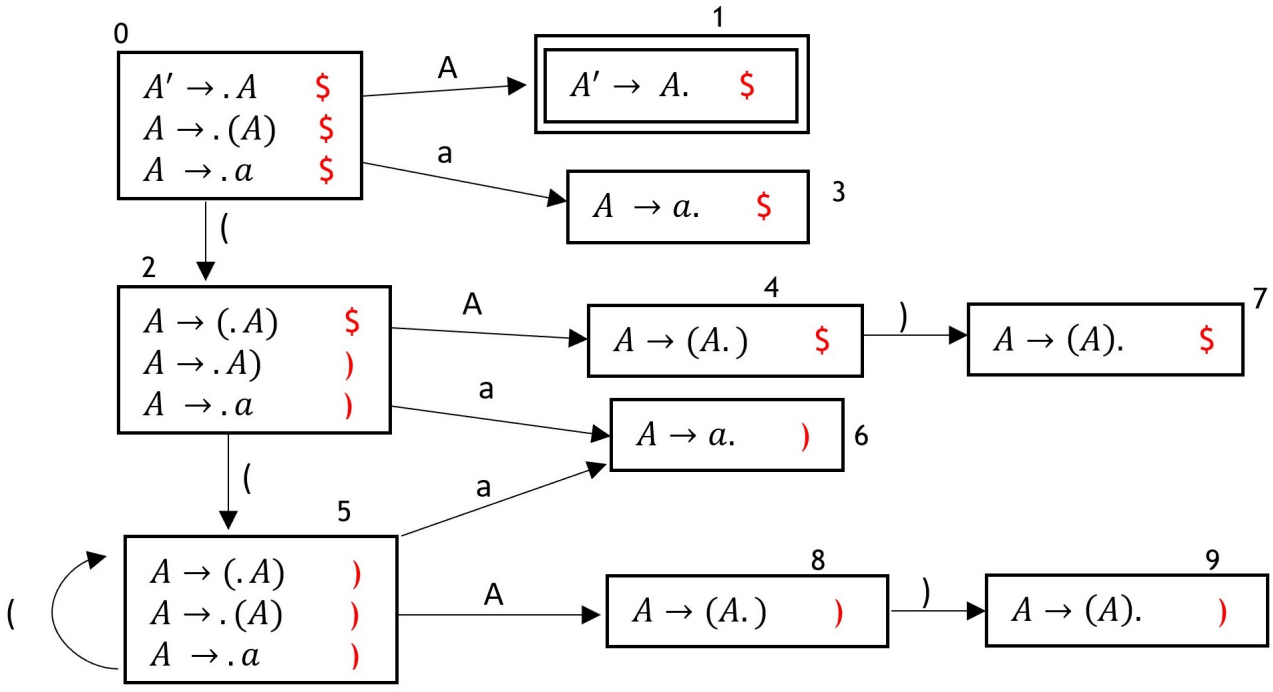

Figure 6: Autómato LR(1) da gramática 3

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

Table 1: Restrições a gramática 2

	$($	$)$	$\$$
S'	$S' \longrightarrow S\$$	$S' \longrightarrow S\$$	$S' \longrightarrow S\$$
S	$S \longrightarrow (S)S$	$S \longrightarrow \varepsilon$	$S \longrightarrow \varepsilon$[1ex] height

Table 2: Tabela de correspondências de gramática 2

Stack	**Input**	**Action**
S $	()$	S ⟶ (S)S
(S) S $	()$	match
S) S $	)$	S ⟶ ε
) S $	)$	match
S $	$	S ⟶ ε
$	$	ACCEPT (pilha vazia)

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

[9]Como no estado 3 existe a opção goto A e foi obtido o estado A, seguimos esse estado para o 4
[10]Regra A ⟶ (A).
[11]Regra A ⟶ (A).

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

[12]goto A, para estado 1
[13]Regra A' $\longrightarrow$ A.
[14]*Parsing* atinge raiz de AST tendo percorrido todo o *input*, sendo assim a linguagem aceita a palavra.

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