Chapter 2

LR grammars and languages

This chapter introduces a group of parsing algorithms which create parsing tree of the input string from bottom to top. These algorithms are named LR parsers since they read input string from left to right and they produce right parse of the input. The algorithm may use the information of the nearest k symbols of the unread part of the input string. Grammars which allow such a parser to be constructed are named $LR(k)$ grammars.

The basic principle of LR parser can be stated as follows:
Let $G = (N, T, P, S)$ be an unambiguous context-free grammar and let $w = a_1a_2...a_n$ be an input string from language $L(G)$. Then there exists rightmost derivation $S = \gamma_1 \Rightarrow \gamma_2 \Rightarrow \ldots \Rightarrow \gamma_m = w$.

Since the mentioned derivation is rightmost one, every sentential form $\gamma_i (i = 1, 2, ..., m-1)$ is of the form $\gamma_i = \alpha A a_j a_{j+1} \ldots a_n$, where $A \in N$, $\alpha \in (N \cup T)^*$ and string $a_j a_{j+1} \ldots a_n \in T^*$ is a suffix of the input string w. Suppose $\gamma_{i-1} = \alpha B z$ and a rule $B \rightarrow \beta$ be used in a derivation step $\gamma_{i-1} \Rightarrow \gamma_i$ (that is $\alpha B z \Rightarrow \alpha \beta z$). The main problem of deterministic bottom-up parsing is to find out the correct string βz in the sentential form γ_i. If the string is found, sentential form γ_i can be reduced to the sentential form γ_{i-1}.

The model of a bottom-up parser is pushdown automaton. Such an automaton is, in general, nondeterministic one, thus cannot be directly used as a parser. Let consider how a deterministic pushdown automaton can be constructed for a given grammar and what circumstances are to be satisfied. Given a context-free grammar, a pushdown automaton can be constructed. The transition mapping δ is defined as follows (remember, the top of the pushdown store is on the right-hand side):

1. $\delta(q, a, \varepsilon) \quad = \{(q, a)\} \forall a \in T$,
2. $\delta(q, \varepsilon, \alpha) \quad = \{(q, A) : A \rightarrow \alpha \in P\}$,
3. $\delta(q, \varepsilon, \#S) = \{(r, \varepsilon)\}$.

The operations are denoted *shift* (1), *reduce* (2), and *accept* (3).

The above shown construction leads to a pushdown automaton that is nondeterministic in all cases. The reason resides in the fact that the shift is defined by a transition $\delta(q, a, \varepsilon) = \{(q, a)\}$ and reduce by a transition $\delta(q, \varepsilon, \alpha) = \{(q, A)\}$. For both transitions, the string ε is a prefix (an also a suffix) of the string α. To obtain a really deterministic pushdown automaton, the construction is to be changed. The main problem of the construction is the fact that shift operations are done regardless the contents of the pushdown store. Therefore, we will attempt to modify the automaton in such a way that it can decide which operation (shift or reduce) to perform based on the symbol that is on the top of the pushdown store. We will demonstrate the technique in the following example.

Example 2.1:

Let a context-free grammar be $G = (\{S, A, B\}, \{a, b, c, d\}, P, S)$, where P contains rules:

(1) $S \rightarrow Aa$
(2) $A \rightarrow bB$
(3) $A \rightarrow Ac$
(4) $B \rightarrow d$

A pushdown automaton for that grammar can be constructed as follows:
$R = (\{q, r\}, \{a, b, c, d\}, \{S, A, B, a, b, c, d, \#\}, \delta, q, \#, \{r\})$, where transition δ is defined:

1. $\delta(q, a, \varepsilon) = \{(q, a)\}$
   $\delta(q, b, \varepsilon)) = \{(q, b)\}$
   $\delta(q, c, \varepsilon)) = \{(q, c)\}$
   $\delta(q, d, \varepsilon)) = \{(q, d)\}$
2. $\delta(q, \varepsilon, Aa) = \{(q, S)\}$
   $\delta(q, \varepsilon, bB) = \{(q, A)\}$
   $\delta(q, \varepsilon, Ac) = \{(q, A)\}$
   $\delta(q, \varepsilon, d) \quad = \{(q, B)\}$
3. $\delta(q, \varepsilon, \#S) = \{(r, \varepsilon))\}$

This pushdown automaton is nondeterministic because it performs shifts according to its definitions. Based on the contents of the pushdown store, the shifts may be performed as follows:

$\delta(q, a, A) = \{(q, Aa)\}$ - symbols a and c appear in the sentential form after
$\delta(q, c, A) = \{(q, Ac)\}$ the symbol A,
$\delta(q, b, \#) = \{(q, \#b)\}$ - symbol b can appear at the beginning of the sentential form only,
$\delta(q, d, b) = \{(q, bd)\}$ - symbol d can appear just after symbol b only.

This modification leads to a deterministic pushdown automaton for the given grammar. However, this technique is not universal, it can be used for a limited class of grammars (for strong $LR(0)$ grammars) only. For other grammars, the modification will not work and the resulting pushdown automaton will not be deterministic. The bottom-up parser is similar to top-down parser – both the parsers can use the following additional information to choose next operation while parsing:

1. the information about the not-yet read part of the input string,

2. the information about parsing in the past.

There are grammars that can be deterministically parsed by the bottom-up parser with the additional information about up to k closest symbols in the unread part of the input string. These grammars are *strong* $LR(k)$ grammars. In the next sections we will study two classes of LR grammars. First, we will introduce strong $LR(k)$ grammars. Then, weak $LR(k)$ grammars will be studied. Deterministic parsing of weak $LR(k)$ grammars must use information about the parsing history. Both classes of LR grammars use the same (except for slight modifications) parsing algorithm which is based on the pushdown automaton. For both classes of grammars, a parsing table is used to decide whether a reduction is to be performed or not and which reduction is to be choosen. The parsing table contains all necessary information. The table is constructed based on the grammar. The construction algorithm is different for both classes of LR grammars. We will describe the algorithms for individual cases in the following sections.

2.1 Strong LR grammars

Strong LR grammars are context-free grammars, for which exist a deterministic bottom-up parser that:

1. uses the information about up to k closest symbols in the not-yet read part of the input string,
2. does not use the information about the parsing history.

Prior to defining strong $LR(k)$ grammars, we will introduce functions $BEFORE$ and EFF_k.

Definition 2.2: Let $G = (N, T, P, S)$ be a context-free grammar, $X \in N$, and $\alpha \in (N \cup T)^*$. Functions $BEFORE(X)$ and $EFF_k(\alpha)$ are defined as follows:

$$\begin{aligned} BEFORE(X) &= \{Y : S \Rightarrow^* \alpha Y X \beta, Y \in (N \cup T)\} \cup \{\# : S \Rightarrow^* X\beta\}, \\ EFF_k(\alpha) &= \{w : w \in FIRST_k(\alpha), \text{and there exists rightmost derivation} \\ &\quad \alpha \Rightarrow^* \beta \Rightarrow^* wx \text{ such, that for no } \beta \text{ holds that } \beta = Awx \ \}. \end{aligned}$$

The set $EFF_k(\alpha)$ contains all strings from the set $FIRST_k(\alpha)$ that were not derived by a derivation $\alpha \Rightarrow^* \beta \Rightarrow^* wx$ such that the first nonterminal in β was substitued by an empty string. The name EFF stands for ε-free first.
Now, we can define strong $LR(k)$ grammar.

Definition 2.3:
A contex-free grammar $G = (N, T, P, S)$ is a strong $LR(k)$ grammmar, if the augmented grammar $G' = (N \cup \{S'\}, T, P \cup \{S' \to S\}, S')$ mets the following criteria:

1. If P' contains a pair of rules of the form:
   (a) $A \to \alpha X, B \to \beta X$,
   (b) $A \to \alpha X, B \to \varepsilon$ and $X \in BEFORE(B)$, or
   (c) $A \to \varepsilon, B \to \varepsilon$ and $X \in BEFORE(B)$, $X \in BEFORE(A)$
   then $FOLLOW_k(A) \cap FOLLOW_k(B) = \emptyset$.
2. If P' contains a pair of rules of the form:
   (a) $A \to \alpha X,\ B \to \alpha X \gamma$,
   (b) $A \to \varepsilon, B \to \alpha X \gamma$ and $X \in BEFORE(A)$, or
   (c) $A \to \varepsilon, B \to \gamma$ and $X \in BEFORE(A)$, $X \in BEFORE(B)$
   then $FOLLOW_k(A) \cap\ EFF_k(\gamma FOLLOW_k(B)) = \emptyset$.

The first condition ensures that in the case of a reduction, it is possible to choose the correct rule for the reduction based on up to k lookahead symbols. The second condition guarantees that it is possible to decide whether reduction or shift operation is to be performed.

Similarly to the top-down parser, the bottom-up parser uses parsing table when choosing the next operation that is to be performed. The table entries contain the appropriate operation, which is based on the topmost pushdown store symbol and on the lookahead string. The parsing table for a strong $LR(k)$ grammar can be constructed using the following algorithm.

Algorithm 2.4: Construction of a parsing table for a strong $LR(k)$ grammar.
Input: Strong $LR(k)$ grammar $G = (N, T, P, S)$.
Output: Parsing table p for G.
Method: Parsing table p is defined over $(N \cup T \cup \{\#\}) \times T^{*k}$.

1. The input grammar G is augmented:

$$G' = (N \cup \{S'\}, T, P \cup \{S' \rightarrow S\}, S').$$

2. Parsing table p is constructed:
   (a) $p(X,u) =$ *reduce(i)*, if $A \rightarrow \alpha X$ is i-th rule in P and $u \in FOLLOW_k(A)$.
   (b) $p(X,u) =$ *reduce(i)*, if $A \rightarrow \varepsilon$ is i-th rule in P, $X \in BEFORE(A)$, $u \in FOLLOW_k(A)$.
   (c) $p(S,\varepsilon) =$ *accept*.
   (d) $p(X,u) =$ *shift*, if $B \rightarrow \beta X \gamma \in P$ and $u \in EFF_k(\gamma FOLLOW_k(B))$.
   (e) $p(X,u) =$ *error* in all other cases.

Example 2.5: Given grammar $G = (\{E, E', T, T', F\}, \{a, +, *, (,)\}, P, E)$, where P contains rules below. Evaluate parsing table for G.

(1)	E	$\rightarrow$	$E'T'$	(5)	T'	$\rightarrow$	$T*$
(2)	E'	$\rightarrow$	$E+$	(6)	T'	$\rightarrow$	ε
(3)	E'	$\rightarrow$	ε	(7)	F	$\rightarrow$	(E)
(4)	T	$\rightarrow$	$T'F'$	(8)	F	$\rightarrow$	a

We augment the grammar by the rule (0) $S \rightarrow E$. Grammar G' is a strong $LR(1)$ grammar, thus parsing table may be constructed. The table is shown below. The operations in the table are denoted as follows: *Sh ... shift, R(i) ... reduce(i), A ... accept*, error entries are left blank. When constructing the table, we used that $BEFORE(E') = \{\#, (\}$ and $BEFORE(T') = \{E'\}$.

p	a	$+$	$*$	$($	$)$	ε
E		Sh			Sh	A
E'	$R(6)$			$R(6)$		
T		$R(1)$	Sh		$R(1)$	$R(1)$
T'	Sh			Sh		
F		$R(4)$	$R(4)$		$R(4)$	$R(4)$
a		$R(8)$	$R(8)$		$R(8)$	$R(8)$
$+$	$R(2)$			$R(2)$		
$*$	$R(5)$			$R(5)$		
$($	$R(3)$			$R(3)$		
$)$		$R(7)$	$R(7)$		$R(7)$	$R(7)$
$\#$	$R(3)$			$R(3)$		

Strong $LR(k)$ parsing can be done using the following algorithm.

Algorithm 2.6: Strong $LR(k)$ parsing algorithm.
Input: Parsing table p for grammar $G = (N, T, P, S)$, grammar G rules, and input string $w \in T^*$.
Output: Right parse in case string $w \in L(G)$, error signaling otherwise.
Method: The algorithm reads symbols from the input string w, makes use of the pushdown store and creates a string of numbers of rules which were used during the reductions. The initial pushdown store contents is $\#$. The algorithm repeats steps $(1a)$ and $(1b)$ until the input string is either accepted or rejected (error signaling). In the description below, let the symbol X denote the symbol on the top of the pushdown store.

1. Evaluate the lookahead string (of length k), let it be u.
   (a) If $p(X,u) =$ *shift*, one symbol is read from the input string and is stored in the pushdown store.

(b) If $p(X,u) =$ *reduce(i)*, the algorithm finds rule i, let it be $A \to \alpha$. Then remove (pop) string α from the pushdown store, push the symbol A on the pushdown store and append rule number i to the right parse. If the contents of the pushdown store was not equal to α (and thus it was not possible to remove it), an error is detected and parsing ends with an error signaling.

(c) If $p(X,\varepsilon) =$ *accept* and the contents of the pushdown store is $\#X$, the parsing was successful and the output string is correct right parse of the input string. If the contents of the pushdown store is different, an error signaling takes place instead.

(d) If $p(X,u) =$ *error*, the parsing ends with an error signaling.

A configuration of the parsing algrithm is triplet (α, x, π), where

α is the contents of the pushdown store (topmost symbol is on the right-hand side),

x is the not-yet read part of the input string,

π is the so far created part of the output,

$(\#, w, \varepsilon)$ is the initial configuration,

$(\#S, \varepsilon, \pi)$ is the final configuration.

Example 2.7: Let us demonstrate the parsing of the input string $a + a * a$. We will use parsing table evaluated in Example 2.5.

$$
\begin{array}{lrrr}
(\#, a+a*a, \varepsilon) & \vdash (\#E', & a+a*a, & 3) \\
& \vdash (\#E'T', & a+a*a, & 36) \\
& \vdash (\#E'T'a, & +a*a, & 36) \\
& \vdash (\#E'T'F, & +a*a, & 368) \\
& \vdash (\#E'T, & +a*a, & 3684) \\
& \vdash (\#E, & +a*a, & 36841) \\
& \vdash (\#E+, & a*a, & 36841) \\
& \vdash (\#E', & a*a, & 368412) \\
& \vdash (\#E'T', & a*a, & 3684126) \\
& \vdash (\#E'T'a, & *a, & 3684126) \\
& \vdash (\#E'T'F, & *a, & 36841268) \\
& \vdash (\#E'T, & *a, & 368412684) \\
& \vdash (\#E'T*, & a, & 368412684) \\
& \vdash (\#E'T', & a, & 3684126845) \\
& \vdash (\#E'T'a, & \varepsilon, & 3684126845) \\
& \vdash (\#E'T'F, & \varepsilon, & 36841268458) \\
& \vdash (\#E'T, & \varepsilon, & 368412684584) \\
& \vdash (\#E, & \varepsilon, & 3684126845841)
\end{array}
$$

A special case of strong $LR(k)$ grammars are strong $LR(0)$ grammars. For these grammars, the information about the topmost symbol on the pushdown store is sufficient when choosing the next operation during parsing. Any strong $LR(0)$ grammar has these properties:

1. the right-hand sides of all rules in the augmented grammar G' end with mutualy different symbols,
2. a symbol occuring at the end of a right-hand side of any rule does not appear in any other rule on the right-hand side.

The above properties imply that grammar G does not contain any ε-rules and that starting symbol S does not appear on right-hand side of any rule in G. Moreover, the symbols occuring at the end of right-hand sides of rules positively identify when a reduction is to be performed and what rule to use for the reduction. If a symbol X, which occurs at the end of rule $A \to \alpha X$, appears on the top of the pushdown store, then reduction by rule $A \to \alpha X$ is to be performed.

Example 2.8: Let grammar $G = (\{S, A, B\}, \{a, b, c, d\}, P, S)$ have P containing rules:

(1)	S	$\to$	Aa	(3)	A	$\to$	Ac
(2)	A	$\to$	bB	(4)	B	$\to$	d

Grammar G is a strong $LR(0)$ grammar. We augment the grammar with rule (0) $S' \to S$ and construct parsing table.

p	ε
S	A
A	Sh
B	$R(2)$
a	$R(1)$
b	Sh
c	$R(3)$
d	$R(4)$
$\#$	Sh

The parsing of the input string $bdca$ is depicted below.

$$
\begin{aligned}
(\#, bdca, \varepsilon) &\vdash (\#b, dca, \varepsilon) \\
&\vdash (\#bd, ca, \varepsilon) \\
&\vdash (\#bB, ca, 4) \\
&\vdash (\#A, ca, 42) \\
&\vdash (\#Ac, a, 42) \\
&\vdash (\#A, a, 423) \\
&\vdash (\#Aa, \varepsilon, 423) \\
&\vdash (\#S, \varepsilon, 4231)
\end{aligned}
$$

2.2 Weak LR grammars

The strong LR parsing was using information about k symbols from the not-yet read part of the input string and one symbol on the top of the pushdown store only. In the case of weak LR grammars, such an information is not enough. When choosing the next rule in the parsing, weak LR parser uses the information about parsing history in addition to the strong LR parser. Obviously, weak LR grammars include strong LR grammars as a proper subset. For that reason, we will omit the "weak" adjective in the next text.

During the parsing of an input string x generated by a grammar G, the bottom-up parser uses the pushdown store to keep a string that corresponds to a prefix of some rightmost sentential form that occures in the rightmost derivation of x in G. If a grammar $G = (N, T, P, S)$ allows a derivation $S \Rightarrow^* \alpha A w \Rightarrow \alpha \beta w \Rightarrow^* xw$, then rightmost sentential form w may be reduced using rule $A \to \beta$ to rightmost sentential form $\alpha A w$. The substring β is a *handle*[*] of sentential form $\alpha \beta w, \alpha, \beta \in (N \cup T)^*, w \in T^*$.

Definition 2.9: Let $G = (N, T, P, S)$ be a context-free grammar and let $G' = (N \cup \{S'\}, T, P \cup$

[*] redukční část

$\{S' \rightarrow S\}, S')$ be augmented grammar for G. We state that G is $LR(k)$ grammar $k \geq 0$, if (all derivations are rightmost):

1. $S' \Rightarrow_{rm}^{*} \alpha A w \Rightarrow \alpha\beta w$,
2. $S' \Rightarrow_{rm}^{*} \gamma B x \Rightarrow \alpha\beta y$,
3. $FIRST_k(w) = FIRST_k(y)$

implies that $\alpha A y = \gamma B x$, (i.e. $\alpha = \gamma$, $A = B$, and $x = y$).
In other words, it is possible to positively decide that a reduction by a rule $A \rightarrow \beta$ is to be performed based on string α and the lookahead string of length up to k symbols.

Definition 2.10:
Assume that $S \Rightarrow^{*} \alpha A w \Rightarrow \alpha\beta w$ is a rightmost derivation in a context-free grammar $G = (N, T, P, S)$. A string γ is a *viable prefix*[*] in G, if it is a prefix of $\alpha\beta$. It means that the string γ is a string which is a prefix of some rightmost sentential form and it does not include complete handle in that sentential form. If $\gamma = \alpha\beta$, then γ is a *complete viable prefix*[†].

In a bottom-up parsing, a viable prefix appears in the pushdown store during the parsing. If the pushdown store contains a complete viable prefix, a reduction can be performed.

In addition to the $LR(k)$ grammars, we will study subclasses of $LR(k)$ grammars:

- $LR(0)$ grammars,
- simple $LR(k)$ grammars, ($SLR(k)$ grammars),
- $LALR(k)$ grammars.

The reasons for defining such a subclasses of $LR(k)$ grammars are listed below:

- the construction of a parser for the subclasses is simplier than the construction for general $LR(k)$ parser,
- the tables for the parser are smaller.

2.2.1 LR(0) grammars

$LR(0)$ grammars are grammars that can be parsed by a deterministic bottom-up parser which uses only parsing history to decide the next parsing operation. It means that the parser does not use the lookahead information.

Example 2.11: Given grammar $G = (\{S, A, B\}, \{a, b\}, P, S)$, where P contains rules:

(1)	S	$\rightarrow$	aAb	(4)	A	$\rightarrow$	b
(2)	S	$\rightarrow$	$aaBba$	(5)	B	$\rightarrow$	ε
(3)	A	$\rightarrow$	aA				

We demonstrate that deterministic parsing of strings generated by this grammar can be done using information about parsing history only. This is not obvious, because symbol b, for instance, appears in G in three different places: it is the right-hand side of the rule $A \rightarrow b$, and it is a part of right hand sides of rules $S \rightarrow aAb$ and $S \rightarrow aaBba$.

The parsing of strings $abab$ and $aaba$ is demonstrated in the table below. These two examples show that reductions are choosen based on a certain contents of the pushdown store. The next table

[*] perspektivní předpona
[†] úplná perspektivní předpona

depicts the strings that will be contained in the pushdown store when a reduction is to be performed using a certain rule. Such strings are always complete viable prefixes. We can see that the example is simple, because the contents of the pushdown store is just one string (one viable prefix) for each rule.

Input	Pushdown store contents	Operation
$abab$	ε	shift a
bab	a	shift b
ab	ab	reduction $A \to b$
ab	aA	shift a
b	aAa	reduction $A \to Aa$
b	aA	shift b
ε	aAb	reduction $S \to aAb$
ε	S	accept
$aaba$	ε	shift a
aba	a	shift a
ba	aa	reduction $B \to \varepsilon$
ba	aaB	shift b
a	$aaBb$	shift a
ε	$aaBba$	reduction $S \to aaBba$
ε	S	accept

There may exist several complete viable prefixes for every rule. Even more, it is possible that there might exist infinite number of complete viable prefixes for a rule. However, it is proven that the sets of complete viable prefixes are regular. It means that it is possible to construct a finite automaton to analyze the viable prefixes. This automaton is called *characteristic automaton* for LR grammar, LR automaton for short. The usage of a LR automaton for analysis of viable prefixes makes LR parsing simplier. When using LR automaton, it is not needed to traverse the pushdown store to decide which operation to use. Instead, it is sufficient to look at state of the LR automaton. The LR automaton for G augmented by the rule $S' \to S$ is depicted in Figure 2.1.

Rule	Pushdown store contents
$S \to aAb$	aAb
$S \to aaBba$	$aaBba$
$A \to Aa$	aAa
$A \to b$	ab
$B \to \varepsilon$	aa

The automaton in Figure 2.1 was constructed such that for every complete viable prefix, there exists a sequence of transitions from the starting state to a final state. The final state corresponds to a certain reduction. Therefore the final states are different and every final state is labeled by a rule that will be used for the reduction.

We outline the $LR(0)$ parsing algorithm (the exact algorithm will be stated later):

Read input string and traverse the LR automaton accordingly. Store the reached states in the pushdown store. When a final state is reached, a reduction is performed. It means that the states that correspond to the right-hand side of the reduction rule are replaced by the nonterminal standing on the left-hand side of the reduction rule.

In the terms of the LR automaton, this can be thought as a return to a state that corresponds to the situation before treating the first symbol on the right-hand side of the reduction rule. In

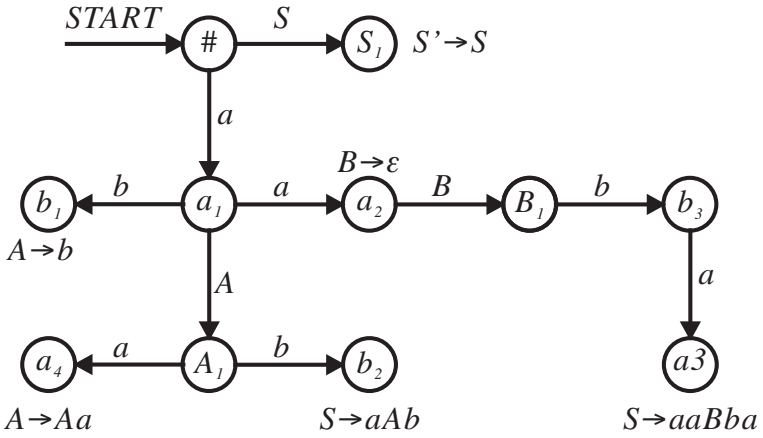

Figure 2.1: LR automaton

that symbol, an edge labeled by the nonterminal standing on the left-hand side of the reduction rule must exist. Using such an edge, new automaton state is reached.

To simply evaluate the end of the parsing, we augment the grammar with a new starting symbol S' and a new rule $S'\rightarrow S$. The complete viable prefix for that rule is always S. The reduction by this rule can be therefore considered the end of parsing and accepting of the input string.

The LR automaton states can be mnemonically labeled as follows: Edges labeled by symbol X lead to a state X_i, the subscript i is choosen so the label X_i is unique. The starting state will be labeled #.
There are three methods to construct the LR automaton:

1. By constructing the collection of sets of LR items. The function $GOTO$ is the transition function of the LR automaton.
2. Construction of LR automaton directly from the grammar.
3. By establishing a system of regular equations. The solution of the equation system are regular expressions describing the sets of complete viable prefixes. The LR automaton can be obtained by constructing finite automaton from these regular equations.

The first method will be studied in detail. The second and the third ones are described in [2,8].

Definition 2.12:
A $LR(0)$ item is a rule from grammar with a position mark on the right-hand side. We will use the symbol . (dot) to mark the position.
For instance $A\rightarrow\alpha.\beta$.

A set of $LR(0)$ items contains $LR(0)$ items which have identical symbol before the dot mark. A set of $LR(0)$ items describes parsing state at the moment when a certain symbol was pushed onto the pushdown store. The symbols after the dot mark are the symbols which might be pushed onto the pushdown store when changing to a new state. There are two important kinds of $LR(0)$ items in a set of $LR(0)$ items:

1. $LR(0)$ items where the dot mark is followed by a terminal symbol. These items represent situation where shift will be performed.
2. $LR(0)$ items where the dot mark is placed on the end of the right-hand side. These items represent reduction states.

For every set M of $LR(0)$ items, there exists a successor set of $LR(0)$ items for every symbol located after the dot mark. We start from the initial set of $LR(0)$ items when constructing the

collection of sets of $LR(0)$ items. Afterwards, all successors of the initial set are constructed. This operation is performed repeatedly for every set of $LR(0)$ items.

A set of $LR(0)$ items is constructed from the kernel and its closure. When constructing the closure, new $LR(0)$ items are added to the set. These items have dot mark before the first symbol of the right hand side and the left hand side nonterminal appears just before the dot mark in some other $LR(0)$ item belonging to the set. The construction of the set of $LR(0)$ items is described in the Algorithm 2.13.

Algorithm 2.13: Construction of the collection of sets of $LR(0)$ items.
Input: A context-free grammar $G = (N, T, P, S)$.
Output: A collection C of sets of $LR(0)$ items for grammar G.
Method:

1. Prepare augmented grammar G':
   $G' = (N \cup \{S'\}, T, P \cup \{S' \rightarrow S\}, S')$, where $S' \notin N$.
2. The initial set of $LR(0)$ items $\#$ is constructed as follows:
   (a) $\# := \{S' \rightarrow .S\}$.
   (b) If $A \rightarrow .B\alpha \in \#, B \in N$ and $B \rightarrow \beta \in P$, then $\# := \# \cup \{B \rightarrow .\beta\}$
   (c) Repeat step b) until no new item can be added to the set $\#$.
   (d) $C := \{\#\}$, $\#$ is the initial set.
3. Having constructed a set of $LR(0)$ items M_i, a new set of $LR(0)$ items X_j will be constructed for every symbol $X \in (N \cup T)$ which appears just after the dot mark in some $LR(0)$ item in M_i. The index j will be choosen so it is higher than the so-far highest index X_n.
   (a) $X_j := \{A \rightarrow \alpha X.\beta : A \rightarrow \alpha.X\beta \in M_i\}$.
   (b) If $A \rightarrow \alpha.B\beta \in X_j, B \in N$, $B \rightarrow \gamma \in P$, then $X_j := X_j \cup \{B \rightarrow .\gamma\}$.
   (c) Repeat step b) until no new item can be added to X_j.
   (d) $C := C \cup \{X_j\}, goto(M_i, X) = X_j$.
4. Repeat step 3. until no new set of $LR(0)$ items can be added to the collection C.

Note: The steps 2a) and 3a) create the kernel of a set of $LR(0)$ items. The closure is evaluated by repeating steps 2b) and 3b).

Example 2.14: Given grammar $G = (\{S, A, B\}, \{a, b, c\}, P, S)$, where P contains rules below. Evaluate the set of $LR(0)$ items.

(1)	S	$\rightarrow$	B	(4)	A	$\rightarrow$	bA
(2)	B	$\rightarrow$	aBb	(5)	A	$\rightarrow$	c
(3)	B	$\rightarrow$	A				

The set of $LR(0)$ items is:

$$
\begin{array}{rl}
\# = & \{S'\rightarrow .S \\
 & S\rightarrow .B \\
 & B\rightarrow .aBb \\
 & B\rightarrow .A \\
 & A\rightarrow .bA \\
 & A\rightarrow .c\} \\
S = & \{S'\rightarrow S.\} \\
A_1 = & \{B\rightarrow A.\} \\
b_1 = & \{A\rightarrow b.A \\
 & A\rightarrow .bA \\
 & A\rightarrow .c\}
\end{array}
\qquad
\begin{array}{rl}
B_1 = & \{S\rightarrow B.\} \\
a = & \{B\rightarrow a.Bb \\
 & B\rightarrow .aBb \\
 & B\rightarrow .A \\
 & A\rightarrow .bA \\
 & A\rightarrow .c\} \\
c = & \{A\rightarrow c.\} \\
B_2 = & \{B\rightarrow aB.b\} \\
A_2 = & \{A\rightarrow bA.\} \\
b_2 = & \{B\rightarrow aBb.\}
\end{array}
$$

Now, we are ready to introduce the *goto* function defined over the set of $LR(0)$ items for grammar G.

Definition 2.15: Function $goto(M_i, X) = X_j$, if items of the form $A\rightarrow\alpha.X\beta$ are in a set M_i and the kernel of X_j was formed by items of the form $A\rightarrow\alpha X.\beta$.

The *goto* function can be represented as an oriented, node and edge labeled graph. The transition $goto(M_i, X) = X_j$ corresponds to the graph depicted in Figure 2.2. The *GOTO* function is the transition mapping of the LR automaton.

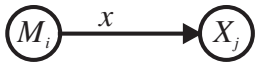

Figure 2.2: *goto* function as a graph

Example 2.16: Let us construct *goto* function for the set of $LR(0)$ items from the Example 2.14. The function is depicted in Figure 2.3.

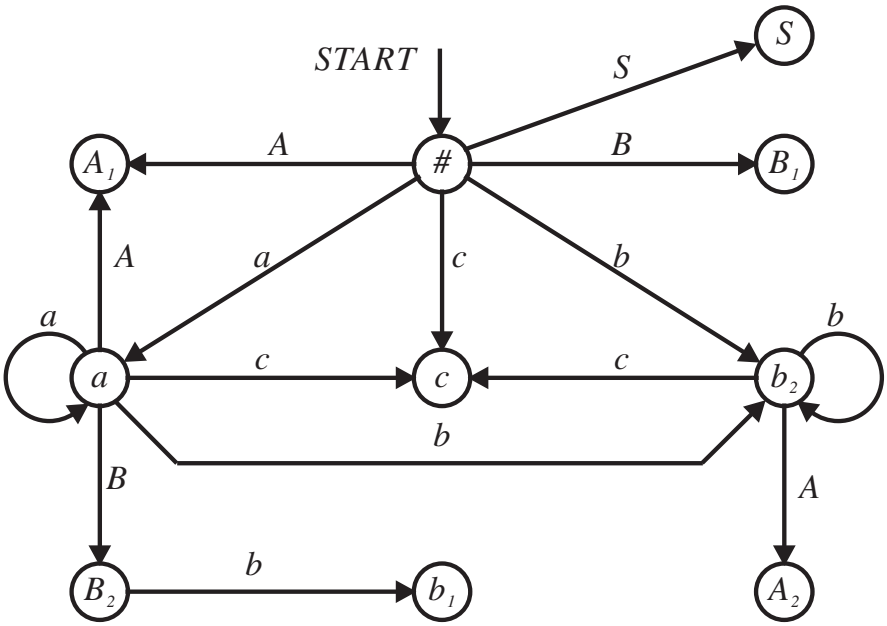

Figure 2.3: LR automaton from Example 2.16

Definition 2.17: A context free grammar $G = (N, T, P, S)$ is an $LR(0)$ grammar, if it holds:

If a set M from the collection of sets of $LR(0)$ items for the grammar G contains an item of the form $A \to \alpha.$, then the set M does not contain any other item of the form $B \to \beta.$ or $B \to \beta.\gamma$, where γ starts by a terminal.

Note: The above definition is equivalent to the definition 2.9 for the case of $k = 0$.

We will now present how to construct a parsing table based on a collection of sets of $LR(0)$ items.

Algorithm 2.18: Construction of a parsing table for $LR(0)$ grammars.
Input: A collection C of sets of $LR(0)$ items for augmented grammar $G' = (N', T, P', S')$.
Output: Parsing table p for grammar G'.
Method: The parsing table p will have rows labeled by symbols that correspond to sets from C. For all $M_i \in C$ do:

1. $p(M_i) = accept$, if $S' \to S. \in M_i$,
2. $p(M_i) = reduce(j)$, if $A \to \beta. \in M_i$ and $A \to \beta$ is the j-th rule from P', and $A \neq S'$, $\beta \neq S$,
3. $p(M_i) = shift$ in all other cases.

Example 2.19: Given grammar $G = (\{S', S, A, B\}, \{a, b, 0, 1\}, P, S')$, where P contains rules:

(0)	S'	$\to$	S	(4)	A	$\to$	0
(1)	S	$\to$	A	(5)	B	$\to$	$aBbb$
(2)	S	$\to$	B	(6)	B	$\to$	1
(3)	A	$\to$	aAb				

This grammar is not $LL(k)$ grammar for any k. We will demonstrate that it is $LR(0)$ grammar. First, we will construct the collection of sets of $LR(0)$ items for G using Algorithm 2.13.

$$
\begin{aligned}
\# = \{&S \to .A \\ &S \to .B \\ &A \to .aAb \\ &A \to .0 \\ &B \to .aBbb \\ &B \to .1 \\ &S' \to .S\} \\
a = \{&A \to a.Ab \\ &B \to a.Bbb \\ &A \to .aAb \\ &A \to .0 \\ &B \to .aBbb \\ &B \to .1\}
\end{aligned}
\qquad
\begin{aligned}
A_1 &= \{S \to A.\} \\
B_1 &= \{S \to B.\} \\
0 &= \{A \to 0.\} \\
1 &= \{B \to 1.\} \\
A_2 &= \{A \to aA.b\} \\
B_2 &= \{B \to aB.bb\} \\
b_1 &= \{A \to aAb.\} \\
b_2 &= \{B \to aBb.b\} \\
b_3 &= \{B \to aBbb.\} \\
S &= \{S' \to S.\}
\end{aligned}
$$

From the structure of the sets of $LR(0)$ items, it is clear that the grammar is $LR(0)$. We will construct the parsing table and LR automaton. The automaton is depicted in Figure 2.4.

p	action
#	shift
A_1	reduce(1)
B_1	reduce(2)
a	shift
0	reduce(4)
1	reduce(6)
A_2	shift
B_2	shift
b_1	reduce(3)
b_2	shift
b_3	reduce(5)
S	accept

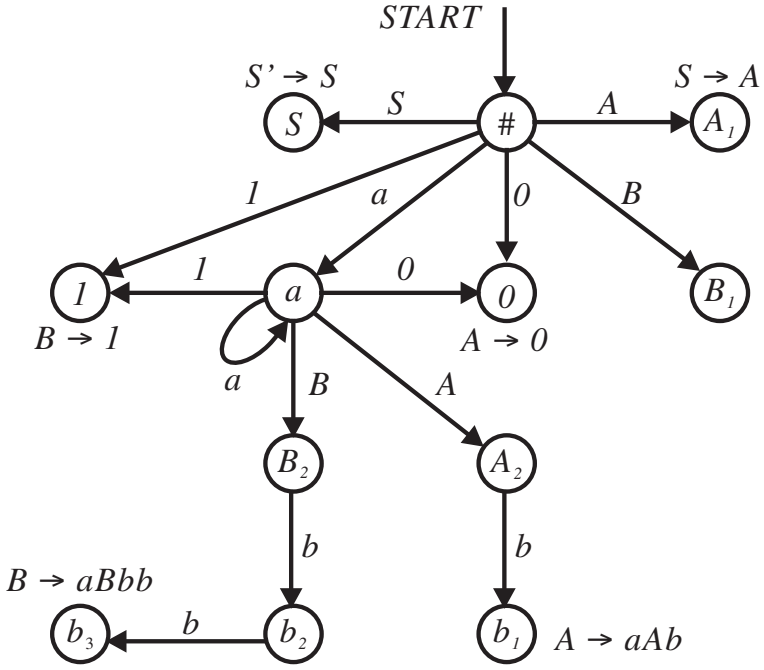

Figure 2.4: LR automaton from Example 2.19

The parsing algorithm for $LR(0)$ grammars is similar to the parsing algorithm for strong $LR(0)$ grammars. The difference is that the states of the LR automaton will be stored in the pushdown store instead of the grammar symbols.

Algorithm 2.20: Parsing algorithm for $LR(0)$ grammars.
Input: Parsing table p and LR automaton for grammar G, input string $w \in T^*$, and the initial symbol stored in the pushdown store (the label of the initial set of $LR(0)$ items, # is the conventional name used in this textbook).
Output: Right parse of w in case $w \in L(G)$, an error signaling otherwise.
Method: The algorithm reads symbols from the input string w, uses the pushdown store and creates a sequence of numbers of rules used for reductions. The initial pushdown store contents is #. The algorithm repeats steps 1., 2., ..., 6. until it either accepts the string or detects an error. In the steps below, let the X denote the topmost symbol in the pushdown store.

1. If $p(X) =$ *shift*, read one symbol and continue with step (5).
2. If $p(X) =$ *reduce(i)*, find i-th rule from P, let it be $A \to \alpha$. Pop $|\alpha|$ symbols from the pushdown store and append rule number (i) to the output. Continue with step (5).
3. If $p(X) =$ *accept* and entire input string was read, the parsing terminates, input string w is accepted and the output string is the right parse of w. If the entire input string was not read, the parsing ends with an error signaling.
4. If $p(X) =$ *error*, the parsing ends with an error signaling.
5. Let Y be the symbol that is to be pushed into the pushdown store (Y is either the input symbol read in step (1) or left-hand side of the rule used in step (2)) and let X be the symbol on the top of the pushdown store (note that step (2) could remove certain number of symbols from the pushdown store).
6. If $goto(X, Y) = Z$, then store Z onto the pushdown store and continue with step (1).
7. If $goto(X, Y)$ is not defined, the parsing ends with an error signaling.

Example 2.21: We will demonstrate $LR(0)$ parsing for input string $aa0bb$ using parsing table and LR automaton from Example 2.19.

$$\begin{array}{lcrr}
(\#, aa0bb, \varepsilon) & \vdash & (\#aa, & a0bb, & \varepsilon) \\
& \vdash & (\#aa, & 0bb, & \varepsilon) \\
& \vdash & (\#aa0, & bb, & \varepsilon) \\
& \vdash & (\#aaA_2, & bb, & 4) \\
& \vdash & (\#aaA_2b, & b, & 4) \\
& \vdash & (\#aA_2, & b, & 43) \\
& \vdash & (\#aA_2b, & \varepsilon, & 43) \\
& \vdash & (\#A_2, & \varepsilon, & 433) \\
& \vdash & (\#S, & \varepsilon, & 4331)
\end{array}$$

2.2.2 Simple $LR(k)$ grammars

In the previous section, the construction of $LR(0)$ parser was studied. The condition for $LR(0)$ grammar is fulfilled only by a small subset of grammars. Very often, there is a set of $LR(0)$ items that contain an item of the form $A \to \alpha.$, which represents reduction, as well as other item of the form $B \to \beta.$ representing some other reduction or an item of the form $C \to \gamma.a\delta$ representing a shift. Such a grammar is not an $LR(0)$ one. We will demonstrate it in the next Example.

Example 2.22: Given grammar $G = (\{E', E, T, F\}, \{+, *, (,), a\}, P, E')$, where the set of rules P contains rules:

$$\begin{array}{llcl@{\qquad}llcl}
(0) & E' & \to & E & (4) & T & \to & F \\
(1) & E & \to & E + T & (5) & F & \to & (E) \\
(2) & E & \to & T & (6) & F & \to & a \\
(3) & T & \to & T * F & & & &
\end{array}$$

The collection of sets of $LR(0)$ items for G contains the following sets: