Contents

Chapter 9

Deterministic bottom-up parsing

9.1 The basics of Shift–Reduce parsing

The *deterministic LR(k)* parsing method, introduced by Knuth [5], is applicable to a strictly larger subclass of ordinary grammars than the LL parsing. Like an LL parser, an LR(k) parser reads the string from left to right using a stack memory. The stack memory is used to represent a partial parse of the read portion of the input. When the parser is in a configuration (η, v), where $w = uv$ is the entire input string and $\eta \in (\Sigma \cup N)^*$ represents the current stack contents, this means that the parser has already parsed the earlier part of the input u, and found its representation as the concatenation η, with $u \in L_G(\eta)$.

A deterministic LR parser may use two operations: (i) shifting the next input symbol to the stack, and (ii) reducing a right-hand side of a rule $A \to \alpha$ at the top of the stack to a single symbol A:

$$(\eta, av) \xrightarrow{\text{Shift } a} (\eta a, v)$$
$$(\eta\alpha, v) \xrightarrow{\text{Reduce } A \to \alpha} (\eta A, v)$$

In order to decide which operation to apply in a configuration, a deterministic LR parser may use the next k symbols of the input, where k is fixed, and the entire contents of its stack. The exact dependence of the action on these data shall be derived and explained in this section; in the end, the action at each step shall be computed in contant time.

Example 9.1. *Grammar for* $\{ a^n c b^n \mid n \geqslant 0 \} \cup \{ a^n d b^{2n} \mid n \geqslant 0 \}$*:*

$$\begin{array}{lcl} S & \to & C \mid D \\ C & \to & aCb \mid c \\ D & \to & aDbb \mid d \end{array}$$

The string *aacbb* is recognized as follows:

$$(\varepsilon, aacbb) \xrightarrow{\text{SHIFT } a} (a, acbb) \xrightarrow{\text{SHIFT } a} (aa, cbb) \xrightarrow{\text{SHIFT } c} (aac, bb) \xrightarrow{\text{REDUCE } C \to c}$$
$$(aaC, bb) \xrightarrow{\text{SHIFT } b} (aaCb, b) \xrightarrow{\text{REDUCE } C \to aCb} (aC, b) \xrightarrow{\text{SHIFT } b} (aCb, \varepsilon) \xrightarrow{\text{REDUCE } C \to aCb}$$
$$(C, \varepsilon) \xrightarrow{\text{REDUCE } S \to C} (S, \varepsilon)$$

Brief outline: At each step, the parser guesses a path in a possible parse tree, with its stack contents representing a parse forest to the left of that path. This guess is made by an NFA, which reads the stack from left (bottom) to right (top) and emits an action in the end. That NFA can be determinized. The resulting DFA can be executed at each step to compute the action. One can avoid scanning the entire stack at each step by storing the intermediate states of the DFA in the stack.

9.2 Parsing tables used by LR parsers

First, a DFA processing the stack from the bottom to the top. For a grammar $G = (\Sigma, N, R, S)$, a DFA $\mathcal{A} = (\Sigma \cup N, Q, q_0\delta)$. Instead of a set of accepting states, there is an table of actions performed by the parser after reading the stack up to the top. The action is determined by the state of the DFA and by the first k symbols of the unread portion of the input. ACTION: $Q \times \Sigma^{\leqslant k} \to \{\text{SHIFT}\} \cup \{\text{REDUCE } A \to \alpha\}_{A\to\alpha\in R}$

9.3 Representation by pushdown automata

9.3.1 Deterministic pushdown automata

Pushdown automata as a reformulation of ordinary grammars (Chomsky and Schützenberger, 1963). Their deterministic case is important as a model for LR parsers.

Definition 9.1 (Ginsburg and Greibach [4])**.** *A* deterministic pushdown automaton *(DPDA) is a septuple* $\mathcal{A} = (\Sigma, Q, \Gamma, q_0, \bot, \delta, F)$*, in which*

- Σ *is a finite input alphabet,*
- Q *is a finite set of states,*
- Γ *is a finite pushdown alphabet,*
- $q_0 \in Q$ *is the initial state*
- $\bot \in \Gamma$ *is the bottom pushdown symbol,*
- *the transition function is* $\delta\colon Q \times (\Sigma \cup \{\varepsilon\}) \times \Gamma \to Q \times \Gamma^*$*, and for each pair of a state* $q \in Q$ *and a stack symbol* $s \in \Gamma$*, either* $\delta(q, \varepsilon, s)$ *is not defined, or* $\delta(q, a, s)$ *is undefined for all* $a \in \Sigma$*;*
- $F \subseteq Q$ *is the set of accepting states.*

The configurations of the automaton are triples (q, w, x)*, where* $q \in Q$ *is the current state,* $w \in \Sigma^*$ *is the remaining input string and* $\gamma \in \Gamma^*$ *represents the pushdown contents. The transition relation on the set of these configurations is defined as* $(q, uw, s\gamma_0) \to (q', w, \gamma\gamma_0)$*, where* $\delta(q, u, s) = (q', \gamma)$*. The language recognized by the PDA is*

$$L(\mathcal{A}) = \{\, w \in \Sigma^* \mid (q_0, w, \bot) \to^* (q_{acc}, \varepsilon, \gamma) \text{ for some } q_{acc} \in F \text{ and } \gamma \in \Gamma^*\bot \}.$$

Theorem 9.1 (Knuth [5])**.** *For every language* $L \subseteq \Sigma^*$*, the following statements are equivalent:*

1. *L is described by an LR(k) grammar for some* $k \geqslant 1$*;*
2. *L is described by an LR(1) grammar;*
3. *L is recognized by a DPDA.*

9.3.2 Grammars to automata

An LR(k) grammar describing a language L to a DPDA recognizing the same language. First, a weaker transformation: to a DPDA recognizing $L\$$, with a dedicated end-marker \$.

Lemma 9.1 (Knuth [5])**.** *Let* $G = (\Sigma, N, R, S)$ *be an LR(k) grammar, let the end-marker* \$ *be a new symbol not in* Σ*. Then there exists and can be effectively constructed a DPDA over the alphabet* $\Sigma \cup \{\$\}$ *that recognizes the language* $L(G)\$$*.*

Proof. Let $M = (\Sigma \cup N, \widehat{Q}, \widehat{q_0}, \widehat{\delta}, \widehat{F})$ be a DFA processing the LR parser's stack, let $\textsc{Action}\colon \widehat{Q} \times \Sigma^{\leqslant k} \to \{\textsc{Shift}\} \cup \{\textsc{Reduce}\ A \to \alpha\}_{A \to \alpha \in R}$ be the parser's action table. Denote the greatest number of symbols in the right-hand side of a rule by $m = \max_{A \to \alpha \in R} |\alpha|$.

The DPDA is defined as a septuple $(\Sigma, Q, \Gamma, q_0, \bot, \delta, F)$, where

$$\begin{aligned} Q &= \Sigma^{\leqslant k} \cup \left(\Sigma^{\leqslant k} \times \{0, 1, \ldots, m\} \times N\right) \cup \{\textsc{Acc}\}, \\ \Gamma &= \{\bot\} \cup \widehat{Q}, \\ q_0 &= \varepsilon, \\ F &= \{\textsc{Acc}\}, \end{aligned}$$

An internal state of the form $x \in \Sigma^{\leqslant k}$ is a buffer containing a look-ahead string, necessary for simulating the LR(k) parser. The automaton begins its computation by filling the buffer with symbols:

$$\delta(x, a, \bot) = (xa, \bot) \qquad (|x| < k)$$

Once the buffer is filled, the automaton writes down in its stack the parser's initial stack configuration, and becomes ready so simulate the parsing:

$$\delta(x, \varepsilon, \bot) = (x, \bot\widehat{q_0}) \qquad (|x| = k)$$

If the end of the input is reached before the buffer is completely filled, the automaton proceeds with the simulation all the same:

$$\delta(x, \$, \bot) = (x, \bot\widehat{q_0}) \qquad (|x| < k)$$

Next, for every look-ahead string $x \in \Sigma^{\leqslant k}$ and for every state $\widehat{q}$ of the LR-DFA, the transitions of the DPDA are defined on the basis of the LR parser's action table. If $\textsc{Action}(\widehat{q}, x) = \textsc{Shift}$, then let $x = ay$ (assuming LR(k) with $k \geqslant 1$). The DPDA should push a symbol $\widehat{\delta}(\widehat{q}, a)$ onto the stack, remove a from the buffer and update the buffer. If there is input symbol b coming, it should be appended to the end of the buffer:

$$\delta(ay, b, \widehat{q}) = (yb, \widehat{q}\widehat{\delta}(\widehat{q}, a)) \qquad (|ay| = k)$$

If the DPDA reads an end-marker instead, then it has just seen the end of the input, and all remaining symbols of the input are kept in the buffer:

$$\delta(ay, \$, \widehat{q}) = (y, \widehat{q}\widehat{\delta}(\widehat{q}, a)) \qquad (|ay| = k)$$

And if the end of the input has already been seen, the DPDA no longer attempts reading any symbols from the input and runs entirely on its buffer contents:

$$\delta(ay, \varepsilon, \widehat{q}) = (y, \widehat{q}\widehat{\delta}(\widehat{q}, a)) \qquad (|ay| < k)$$

Assume that $\textsc{Action}(\widehat{q}, x) = \textsc{Reduce}\ A \to \alpha$. Then, in order to simulate the behaviour of an LR(1) parser, the DPDA should pop as many symbols from the stack as there are symbols in α, then pop one more stack symbol $\widetilde{q}$ and push the state $\widehat{\delta}(\widetilde{q}, A)$ onto the stack, all without touching the input or the buffer. This is done in the intermediate states of the form (x, i, A), where the number $i \in \{0, 1, \ldots, m\}$ indicates the number of stack symbols left to discard.

$$\begin{aligned} \delta(x, \varepsilon, \widehat{q}) &= ((x, |\alpha|, A), \widehat{q}) \\ \delta((x, i, A), \varepsilon, \widetilde{q}) &= ((x, i-1, A), \varepsilon) && (i > 0) \\ \delta((x, 0, A), \varepsilon, \widetilde{q}) &= (x, \widehat{\delta}(\widetilde{q}, A)) \end{aligned}$$

Alternatively, if the LR(k) parser finds its initial state $\widetilde{q_0}$ underneath, and $A = S$, then the parser accepts; this is implemented by the DPDA in the following transition:

$$\delta((x, 0, A), \varepsilon, \widetilde{q}) = (\textsc{Acc}, \varepsilon)$$

□

It remains to remove the end-marker.

Lemma 9.2 (Ginsburg and Greibach [4]). *Let $\$ \notin \Sigma$. If a language $L\$ \subseteq \Sigma^*\$$ is recognized by a DPDA, then so is L.*

9.3.3 Automata to grammars

Lemma 9.3 (Knuth [5]). *For every DPDA recognizing a language $L \subseteq \Sigma^*$, there exists an LR(1) grammar describing the same language.*

9.5 Input-driven automata and grammars

Idea: the string is already given in a parsed form, and a grammar is expected to define some fine details of the syntax.

Example: XML.

An input-driven pushdown automaton has an input alphabet split into three classes, and the type of the current symbol determines whether the automaton must push onto the stack, pop from the stack, or ignore the stack. The model was studied in two waves: first considered by Mehlhorn [8], later rediscovered and further studied by Alur and Madhusudan [1] in 2004 under the name of "visibly pushdown automata".

9.5.1 Input-driven pushdown automata

A *(deterministic) input-driven pushdown automaton* (IDPDA) is a special case of a deterministic pushdown automaton, in which the input alphabet is split into three classes, Σ_{+1}, Σ_{-1} and Σ_0, and the type of the input symbol determines the type of the operation with the stack. For an input symbol in Σ_{+1}, the automaton always pushes one symbol onto the stack. If the input symbol is in Σ_{-1}, the automaton must pop one symbol. Finally, for a symbol in Σ_0, the automaton may not use the stack: that is, neither modify it, nor even examine its contents.

Definition 9.2 (Mehlhorn [8]; Alur and Madhusudan [1]). *A deterministic input-driven pushdown automaton (IDPDA) is an octuple $M = (\Sigma_{+1}, \Sigma_0, \Sigma_{-1}, Q, \Gamma, q_0, \langle\delta_a\rangle_{a\in\Sigma}, F)$, in which:*

- $\Sigma = \Sigma_{+1} \cup \Sigma_0 \cup \Sigma_{-1}$ *is an input alphabet split into three disjoint classes;*
- Q *is a finite set of (internal) states of the automaton;*
- Γ *is the pushdown alphabet;*
- $q_0 \in Q$ *is the initial state;*
- *the transition function by each left bracket symbol $< \in \Sigma_{+1}$ is a partial function $\delta_<\colon Q \to Q \times \Gamma$, which, for a given current state, provides the next state and the symbol to be pushed onto the stack;*
- *for each neutral symbol $c \in \Sigma_0$, the state change is described by a partial function $\delta_c\colon Q \to Q$;*

- *for every right bracket symbol* $> \in \Sigma_{-1}$*, there is a partial function* $\delta_> : Q \times \Gamma \to Q$ *specifying the next state, assuming that the given stack symbol is popped from the stack;*

- $F \subseteq Q$ *is the set of accepting states.*

A configuration of A *is a triple* (q, w, x)*, where* $q \in Q$ *is the state,* $w \in \Sigma^*$ *is the remaining input and* $x \in \Gamma^*$ *is the stack contents. The initial configuration on an input string* $w_0 \in \Sigma^*$ *is* (q_0, w_0, ε)*. For each configuration with at least one remaining input symbol, the next configuration is uniquely determined by a single step transition function defined as follows:*

- *for each left bracket* $< \in \Sigma_{+1}$*, let* $(q, <w, x) \vdash_A (q', w, \gamma x)$*, where* $\delta_<(q) = (q', \gamma)$*;*

- *for every right bracket* $> \in \Sigma_{-1}$*, let* $(q, >w, \gamma x) \vdash_A (\delta_>(q, \gamma), w, x)$*;*

- *for a neutral symbol* $c \in \Sigma_0$*, define* $(q, cw, x) \vdash_A (\delta_c(q), w, x)$*.*

Once the input string is exhausted, the last configuration (q, ε, x) *is accepting if* $q \in F$ *and* $x = \varepsilon$*. The language* $L(A)$ *recognized by the automaton is the set of all strings* $w \in \Sigma^*$*, on which the computation from* (q_0, w, ε) *is accepting.*

Under this definition, IDPDA work only on well-nested strings. Alur and Madhusudan [1] gave a slightly relaxed definition that also applies to ill-nested strings.

Example 9.2. *Consider the language* $L = \{ a^n b^n | n \geqslant 0 \} \cup \{ a^n b^c | n \geqslant 0 \}$ *defined over the alphabet* $\Sigma = \Sigma_{+1} \cup \Sigma_{-1} \cup \Sigma_0$*, where* $\Sigma_{+1} = \{a\}$*,* $\Sigma_{-1} = \{b, c\}$ *and* $\Sigma_0 = \varnothing$*. This language is recognized by a DIDPDA* $\mathcal{A} = (\Sigma_{+1}, \Sigma_0, \Sigma_{-1}, Q, \Gamma, q_0, \langle \delta_a \rangle_{a \in \Sigma}, F)$*, with the set of states* $Q = \{q_0, q_1, q_b, q_c, q_{acc}\}$ *and with the pushdown alphabet* $\Gamma = \{s_0, s_1\}$*. In the initial state* q_0*, the automaton reads the first* a *and pushes* s_0*.*

$$\delta_a(q_0) = (q_1, s_0)$$

All subsequent symbols a *are read in the state* q_1*, where the stack symbol pushed is* s_1*.*

$$\delta_a(q_1) = (q_1, s_1)$$

Thus, after reading a^n*, the stack contains* $s_1^{n-1} s_0$*. Once the automaton reads the first* b *or* c*, it enters a state, in which it will only read symbols of the same type.*

$$\delta_b(q_1, s_1) = q_b$$
$$\delta_c(q_1, s_1) = q_c$$

Each remaining symbol b *or* c *is matched to the corresponding* a *by popping a stack symbol.*

$$\delta_b(q_b, s_1) = q_b$$
$$\delta_c(q_c, s_1) = q_c$$

Once the last matching b *or* c *is read, the automaton knows that by the symbol* s_0*, and accordingly enters the unique accepting state.*

$$\delta_b(q_b, s_0) = \delta_c(q_c, s_0) = q_{acc}$$

9.5.2 Nondeterministic input-driven pushdown automata

Definition 9.3 (von Braunmühl and Verbeek [2]; Alur and Madhusudan [1])**.** *A nondeterministic input-driven pushdown automaton is a tuple* $M = (\Sigma_{+1}, \Sigma_0, \Sigma_{-1}, Q, Q_0, \Gamma, \langle \delta_a \rangle_{a \in \Sigma}, F)$*, in which:*

- Q_0 *is a set of initial states;*
- *for each push-symbol* $< \in \Sigma_{+1}$*, there is a nondeterministic transition function* $\delta_<\colon Q \to 2^{Q \times \Gamma}$*, which, for a given current state, provides zero or more transitions of the form (next state, symbol to be pushed);*
- *for each neutral symbol* $c \in \Sigma_0$*, the state change is nondeterministic, via a function* $\delta_c\colon Q \to 2^Q$*;*
- *for every pop-symbol* $> \in \Sigma_{-1}$*, there is a nondeterministic transition function* $\delta_>\colon Q \times \Gamma \to 2^Q$*.*

9.5.3 Determinization

One of the most important facts about input-driven automata is that their nondeterministic variant can be determinized. This possibility relies on the property that all nondeterministic computations on the same input string must use exactly the same sequence of push and pop operations. A simulating deterministic IDPDA follows the same sequence of stack operations, and can trace all possible computations of the nondeterministic IDPDA. Differing from the well-known *subset construction* for finite automata, this simulation keeps track of a *set of pairs of states* of the nondeterministic machine (rather than just a subset of the set of states), where a pair (p, q) refers to a computation on a well-nested substring that begins in state p and ends in state q.

Theorem 9.2 (von Braunmühl and Verbeek [2])**.** *For every nondeterministic IDPDA* M *defined over an alphabet* $\Sigma = \Sigma_{+1} \cup \Sigma_{-1} \cup \Sigma_0$*, with a set of states* Q *and a pushdown alphabet* Γ*, there exists a deterministic IDPDA* M' *with the set of states* $Q' = 2^{Q \times Q}$ *and with the pushdown alphabet* $\Gamma' = 2^{Q \times Q} \times \Sigma_{+1}$*, which recognizes the same language.*

Notably, the number of stack symbols in the original NIDPDA does not affect the size of the simulating DIDPDA, because the latter never stores those stack symbols.

Proof. Let $A = (\Sigma_{+1}, \Sigma_0, \Sigma_{-1}, Q, Q_0, \Gamma, \langle \delta_a \rangle_{a \in \Sigma}, F)$ be an NIDPDA. Construct a DIDPDA $B = (\Sigma_{+1}, \Sigma_0, \Sigma_{-1}, Q', Q_0, \Gamma', \langle \delta'_a \rangle_{a \in \Sigma}, F')$, with $Q = 2^{Q \times Q}$ and $\Gamma' = 2^{Q \times Q} \times \Sigma_{+1}$, as follows. Every state $P \subseteq Q \times Q$ of B is a set of pairs of states of A, each corresponding to the following situation: whenever $(p, q) \in P$, both p and q are states in one of the computations of A, where p was reached just after reading the most recent left bracket, whereas q is the current state of that computation.

The initial state of B, defined as $q'_0 = \{\, (q, q) \mid q \in Q_0 \}$, represents the behaviour of A on the empty string, which begins its computation in an initial state, and remains in the same state. The set of accepting states reflects all computations of A ending in an accepting state.

$$F' = \{\, P \subseteq Q \times Q \mid \text{there is a pair } (p, q) \in P, \text{ with } q \in F \}$$

The transition functions δ'_a, with $a \in \Sigma$, are defined as follows.

- On a left bracket $< \in \Sigma_{+1}$, the transition in a state $P \in Q'$ is $\tau_<(P) = (P', (<, P))$, where

$$P' = \{\, (q', q') \mid \text{there is a pair } (p, q) \in P) : (\exists s \in \Gamma) : (q', s) \in \delta_<(q) \}.$$

Thus, B pushes the current context of the simulation onto the stack, along with the current left bracket, and starts the simulation afresh at the next level of brackets, where it will trace the computations from all states q' reachable by A at this point.

- For a neutral symbol $c \in \Sigma_0$ and a state $P \in Q'$, the transition $\delta'_c(P) = \{ (p, q') \mid \exists (p, q) \in P : q' \in \delta_c(q) \}$ directly simulates one step of A in all currently traced computations.

- For a right bracket $> \in \Sigma_{-1}$ and a state $P' \subseteq Q'$, the automaton pops a stack symbol $(<, P) \in \Gamma'$ containing a matching left bracket and the context of the previous simulation. Then, each computation in P is continued by simulating the transition by the left bracket, the behaviour inside the brackets stored in P', and the transition by the right bracket.

  $$\delta'_>(P', (<, P)) = \{ (p, q'') | (\exists (p, q) \in P)(\exists (p', q') \in P')(\exists s \in \Gamma) : (p', s) \in \delta_<(q), q'' \in \delta_>(q', s) \}$$

- For an unmatched right bracket $> \in \Sigma_{-1}$, the transition in a state $P \in Q'$ advances all currently simulated computations of A in the same way as for a neutral symbol: $\tau_>(P, \bot) = \{(p, q') \mid \exists (p, q) \in P : q' \in \delta_>(q', \bot)\}$.

The correctness of the construction can be proved by induction on the bracket structure of an input string. □

Matching lower bound.

Theorem 9.3 (Alur and Madhusudan [1]). *Consider an alphabet $\Sigma = \Sigma_{+1} \cup \Sigma_0 \cup \Sigma_{-1}$ with $\Sigma_{+1} = \{<\}$, $\Sigma_{-1} = \{>\}$ and $\Sigma_0 = \{0, 1, \#\}$. Then, for each $n \geqslant 1$, there exist a language L_n over Σ, which is recognized by an NIDPDA with $O(n)$ states and n stack symbols, while every DIDPDA for L_n needs at least 2^{n^2} states.*

9.5.4 Input-driven grammars

Definition 9.4 (Alur and Madhusudan [1]). *An input-driven grammar is a quadruple $G = (\Sigma, N, R, S)$, where*

- *$\Sigma = \Sigma_{+1} \cup \Sigma_0 \cup \Sigma_{-1}$ is the alphabet, split into three disjoint classes;*
- *N is the set of nonterminal symbols;*
- *R is the set of rules, each of the form $A \to <B>C$, $A \to aC$ or $A \to \varepsilon$, with $A, B, C \in N$, $< \in \Sigma_{+1}$, $> \in \Sigma_{-1}$ and $a \in \Sigma$.*
- *$S \in N$ is the initial symbol.*

Definition 9.5. *An input-driven Boolean grammar if a quadruple $G = (\Sigma, N, R, S)$, where there exists a partition of the alphabet into three disjoint classes $\Sigma = \Sigma_{+1} \cup \Sigma_0 \cup \Sigma_{-1}$, and all rules are of the following form,*

$$A \to <C_1>B_1 \,\&\, \ldots \,\&\, <C_m>B_m \,\&\, \neg<E_1>D_1 \,\&\, \ldots \,\&\, \neg<E_n>D_n$$
$$A \to cB_1 \,\&\, \ldots \,\&\, cB_m \,\&\, \neg cD_1 \,\&\, \ldots \,\&\, \neg cD_n$$
$$A \to \varepsilon$$

Any such grammar is called conjunctive, *of $n = 0$ in all rules, and* ordinary, *if furthermore $m = 1$.*

All these grammar models are equivalent to IDPDA.

9.5.5 Operations on input-driven automata

Closed under Boolean operations, concatenation and Kleene star (for binary operations, assuming the same partition of the alphabet in both arguments).

Exercises

9.5.1. (by Chistikov) For every $n \geqslant 0$, consider the singleton language $L_n = \{a^n b^n\}$. Construct an IDPDA recognizing L_n, with $\Theta(\sqrt{n})$ states and with a fixed number of stack symbols.

9.6 Generalized LR parsing

On the one hand, simulates nondeterminism in LR parsing. On the other hand, this is a more practical implementation of tabular algorithms.

9.6.1 Nondeterminism in LR parsing

Generalized LR parsing, first proposed by Lang [7] and later independently discovered and developed by Tomita [10], is a polynomial-time method of simulating nondeterminism in the deterministic LR. Every time a deterministic LR parser has to choose an action to perform (to *shift* an input symbol or to *reduce* by one or another rule), a generalized LR parser performs both actions, storing all possible contents of an LR parser's stack in the form of a graph, which contains $O(n)$ vertices and therefore fits in $O(n^2)$ memory. The algorithm is applicable to every ordinary grammar, and its complexity is bounded by cubic. It works for conjunctive grammars with minimal modifications, while maintaining its cubic-time complexity. A further extension to Boolean grammars requires more significant modifications, and runs in time $O(n^4)$. The main advantage of this algorithm over the tabular algorithms is that it works much faster on "good" grammars: for instance, in linear time on the Boolean closure of the deterministic languages.

9.6.2 Example

Example 9.3. *Consider the following grammar describing the language* $\{ a^n b^n \mid n \geqslant 0\} \cup \{ a^n b^{2n} \mid n \geqslant 0\}$.

$$S \to A \mid B$$
$$A \to aAb \mid \varepsilon$$
$$B \to aBbb \mid \varepsilon$$

*The grammar is not LR(k) for any k, (***ref earlier example***)*
****TBW: nondeterministic LR table for this grammar****
****TBW: how the Generalized LR parser works on this grammar, on the input* $w = aaabbbbbb$.

9.6.3 Graph-structured stack

The Generalized LR uses a *graph-structured stack* to represent the contents of the linear stack of a standard LR parser in all possible branches of a nondeterministic computation. This is a directed graph with a designated *source node*, representing the bottom of the stack. Each arc of the graph is labelled with a symbol from $\Sigma \cup N$. The nodes are labelled with the states of a DFA processing the labels on the path from the source node. This would typically be one of the DFAs defined for the Deterministic LR, and it is used merely to help the parser handle at least some decisions deterministically. In particular, the source node is labelled with the initial state. There is a non-empty collection of designated nodes, called *the top layer* of the stack. Every arc

leaving one of these nodes has to go to another node in the top layer. The labels of these nodes should be pairwise distinct, and hence the number of top layer nodes is bounded by a constant.

9.6.4 The algorithm

Initially, the stack contains a single source node, which at the same time forms the top layer. The computation of the algorithm is an alternation of *reduction phases*, in which the arcs going to the top layer are manipulated without consuming the input, and *shift phases*, where a single input symbol is read and consumed, and a new top layer is formed as a successor of the former top layer.

The shift phase is carried out as illustrated in Figure 9.1. Let a be the next input symbol. For each top layer node labelled with a state q, the algorithm follows the transition of the DFA from q by the symbol a. If the transition leads to a certain state q', a node labelled with this state is created in the new top layer. If it is undefined, this branch of the graph-structured stack is removed.

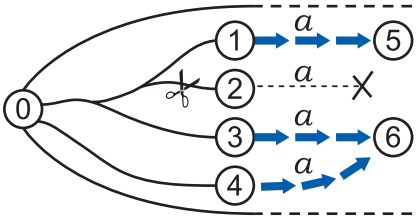

Figure 9.1: The shift phase in the Generalized LR parsing algorithm.

The reduction phase for an ordinary grammar is a sequence of additions of arcs labelled by nonterminals and leading to the top layer. If a grammar has a rule $A \to \alpha$, then, whenever the graph contains a node q, which is connected to the top layer by the path α, the parser may perform a *reduction* by this rule, adding an arc labelled A from q to a node in the top layer labelled by the appropriate state of the DFA. This is illustrated in Figure 9.2(a).

In the case of a conjunctive grammar, for a rule $A \to \alpha_1 \,\&\, \ldots \,\&\, \alpha_m$, the reduction requires m paths from q to the top layer, labelled with α_1, …, α_m, as shown in Figure 9.2(b).

For a Boolean grammar, a rule $A \to \alpha_1 \,\&\, \ldots \,\&\, \alpha_m \,\&\, \neg\beta_1 \,\&\, \ldots \,\&\, \neg\beta_n$. further requires that there are no paths $\beta_1, \ldots, \beta_n$, see Figure 9.2(c). Furthermore, a Generalized LR parser for a Boolean grammar may remove ("invalidate") an arc if the conditions for its existence no longer hold.

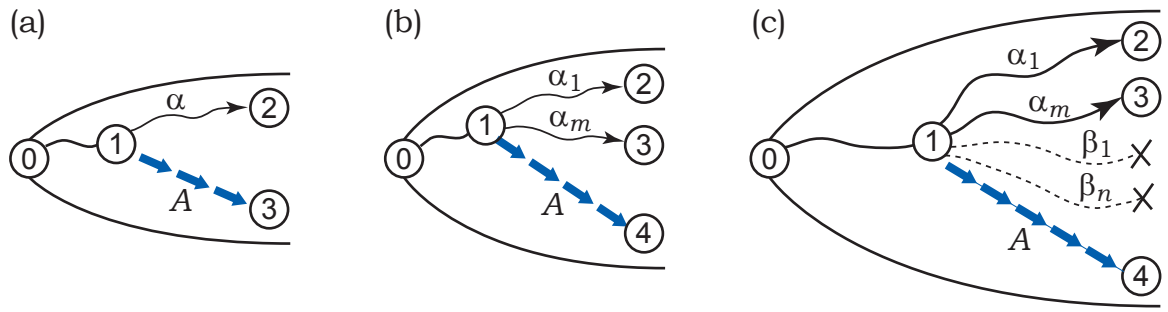

Figure 9.2: Reduction in Generalized LR: (a) by a rule $A \to \alpha$ in an ordinary grammar; (b) by a rule $A \to \alpha_1 \,\&\, \ldots \,\&\, \alpha_m$ in a conjunctive grammar; (c) by a rule $A \to \alpha_1 \,\&\, \ldots \,\&\, \alpha_m \,\&\, \neg\beta_1 \,\&\, \ldots \,\&\, \neg\beta_n$ in a Boolean grammar.

The input string is accepted, if the last reduction phase (the one after shifting the last input

symbol) produces an arc labelled S from the source node to the top layer, which represents a parse of the entire string according to the grammar. The algorithm works correctly—that is, accepts if and only if the string is in the language—for every conjunctive grammar. It also works correctly for most Boolean grammars, although a grammar expressing a contradiction, such as $S \to \neg S$, would bring a parser into an infinite cycle of reductions and invalidations.

9.6.5 Searching in the stack

The worst-case running time of the algorithm depends on how efficiently the operations on the graph are implemented. For ordinary and for conjunctive grammars, can be done in time $O(n^3)$. However, one of the main benefits of this algorithm is that it works much faster on "better" grammars: for instance, in linear time on Boolean combinations of LR(1) grammars.

Bibliography

[1] R. Alur, P. Madhusudan, "Visibly pushdown languages", *ACM Symposium on Theory of Computing* (STOC 2004, Chicago, USA, 13–16 June 2004), 202–211.

[2] B. von Braunmühl, R. Verbeek, "Input driven languages are recognized in $\log n$ space", *North-Holland Mathematics Studies*, 102 (1985), 1–19.

[3] F. DeRemer, "Simple LR(k) grammars", *Communications of the ACM*, 14:7 (1971), 453–460.

[4] S. Ginsburg, S. A. Greibach, "Deterministic context-free languages", *Information and Control*, 9:6 (1966), 620–648.

[5] D. E. Knuth, "On the translation of languages from left to right", *Information and Control*, 8:6 (1965), 607–639.

[6] A. J. Korenjak, "A practical method for constructing LR(k) processors", *Communications of the ACM* 12:11 (1969), 613–623.

[7] B. Lang, "Deterministic techniques for efficient non-deterministic parsers", *ICALP 1974*, LNCS 14, 255–269.

[8] K. Mehlhorn, "Pebbling mountain ranges and its application to DCFL-recognition", *Automata, Languages and Programming* (ICALP 1980, Noordweijkerhout, The Netherlands, 14–18 July 1980), LNCS 85, 422-435.

[9] A. Okhotin, "Generalized LR parsing algorithm for Boolean grammars", *International Journal of Foundations of Computer Science*, 17:3 (2006), 629–664.

[10] M. Tomita, "An efficient augmented context-free parsing algorithm", *Computational Linguistics*, 13:1 (1987), 31–46.

Index