Parsing

LR Parsing

Laura Kallmeyer

Heinrich-Heine-Universität Düsseldorf

Summer 2021

Table of contents

Introduction

- Problem of pure bottom-up parsing: we reduce categories that cannot be reached from the start symbol.
- LR parsing: top-down restricted shift-reduce parsing.
- In contrast to Earley, the top-down predictions are compiled into the states of an automaton.
- Depending on how deterministic the parser is (how many lookaheads are needed), we distinguish $LR(0), LR(1), \ldots$ grammars.

The idea (1)

Shift-reduce parsing

NP $\rightarrow$ Det N, NP $\rightarrow$ John, Det $\rightarrow$ the, N $\rightarrow$ apple, start symbol NP

Γ	remaining input	action
	the apple	
the	apple	shift
Det	apple	reduce
Det apple		shift
Det N		reduce
NP		reduce

This grammar is $LR(0)$ since, by looking only at Γ, we can decide which action to perform.

But: a simple shift reduce parser would have a conflict (shift vs. reduce) after the second configuration.

The idea (2)

Idea of LR Parsing: Shift Reduce Parser guided by precompiled predictions.

Observation: In an Earley Parser,

- The *Predict* operation is independent from the actual input. It therefore can be precompiled.
- The *Scan* and the *Complete* operations depend on the input. We can precompile them with respect to the next terminal or the next category.

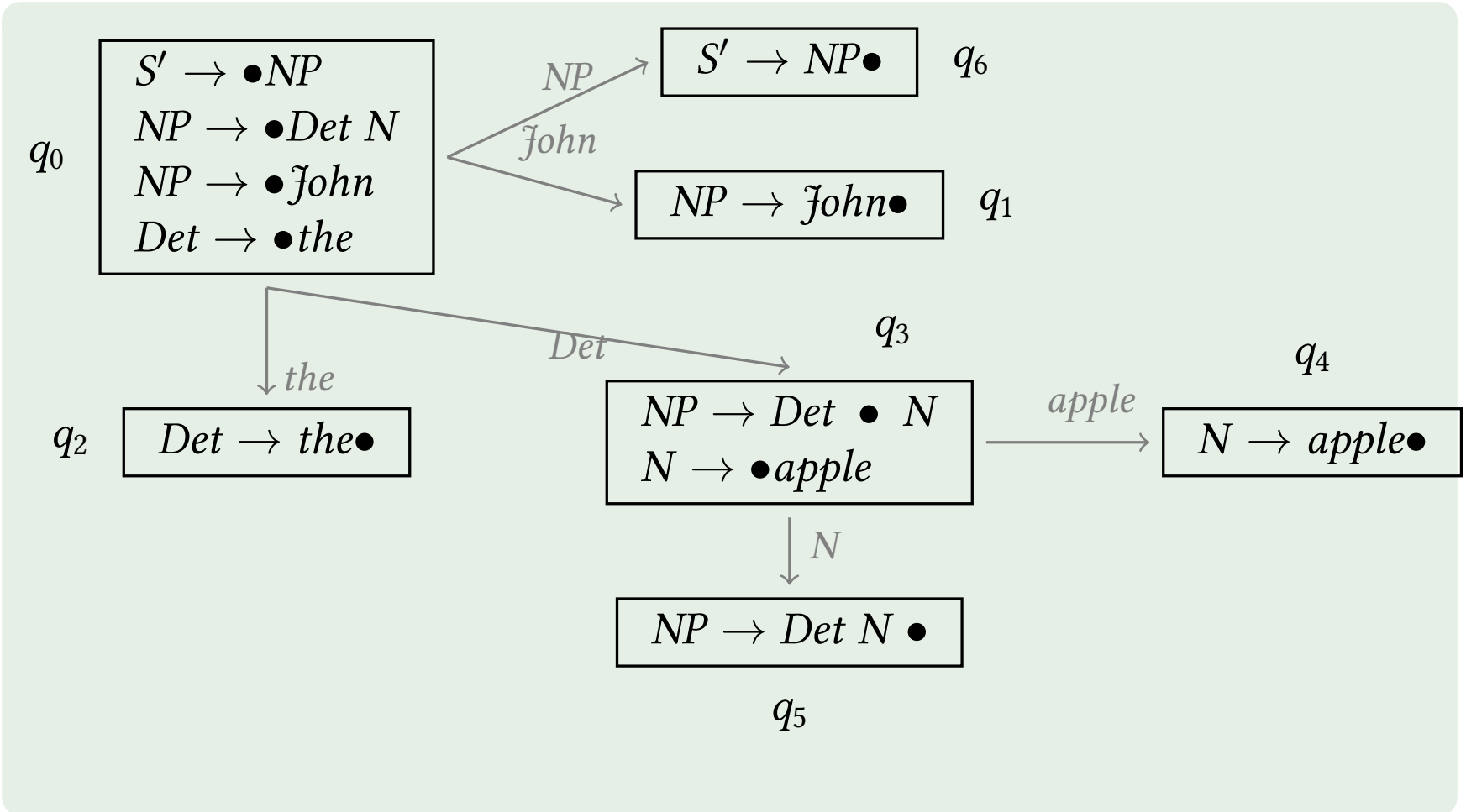
In an LR Parser, all dotted productions we can reach by predicting are precompiled into a single state. We change from one state to another by performing a *Scan* or a *Complete*, i.e., by moving the dot over a terminal or a non-terminal.

The idea (3)

Idea of the LR automaton:

- States are sets of dotted productions with, eventually, the k input symbols that may follow the production as lookaheads.
- States are transitive closures of the predict operation on dotted productions: if $A \to \alpha \bullet B\beta \in q$, then $B \to \bullet\gamma \in q$ for all $B \to \gamma$,
- The construction of the states starts with a new production $S' \to \bullet S$.
- We obtain new states by moving the dot over an $X \in N \cup T$.

The idea (4)


q_0
$S' \to \bullet NP$
$NP \to \bullet Det\ N$
$NP \to \bullet John$
$Det \to \bullet the$
NP
John
$S' \to NP\bullet$
q_6
$NP \to John\bullet$
q_1
Det
the
q_3
q_4
q_2
$Det \to the\bullet$
$NP \to Det \bullet N$
$N \to \bullet apple$
apple
$N \to apple\bullet$
N
$NP \to Det\ N \bullet$
q_5

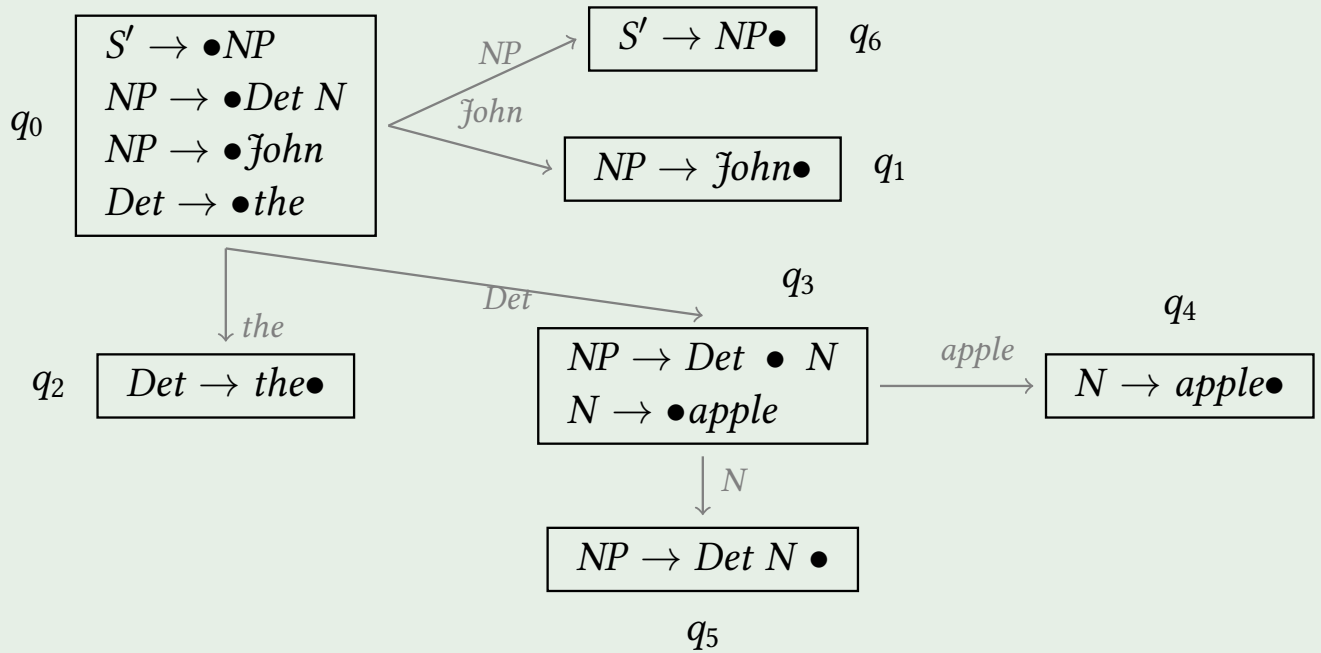
The idea (4)

At every moment, the parser is in one of the states.

- We can *shift* an $a \in T$ if there is an outgoing a-transition. The state changes according to the transition.
- We can *reduce* if the state contains a completed $A \to \alpha\bullet$. We then reduce with $A \to \alpha$. The new state is obtained by following the A-transition, starting from the state reached before processing that rule.

The stack Γ now contains additional states: $\Gamma \in (Q(N \cup T))^* Q$.

The idea (5)



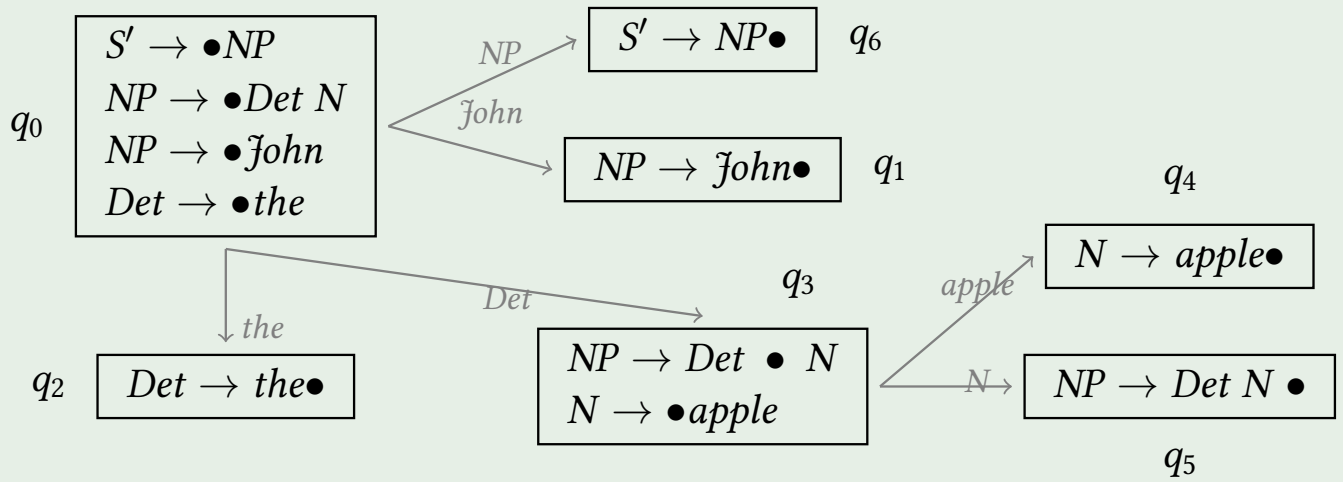
Γ	remaining input	action
q_0	the apple \$	
q_0 the q_2	apple \$	shift
q_0 Det q_3	apple \$	reduce
q_0 Det q_3 apple q_4	\$	shift
q_0 Det q_3 N q_5	\$	reduce
q_0 NP q_6	\$	reduce

The parse table (1)

The information encoded in the automaton is stored in a parse table,
The parse table has two parts:

- the action table that tells us, depending on state and next input symbol, whether a) to shift and, if so, how to change state or b) to reduce and, if so, which production to use or c) to accept.
- the goto table that lists the A-transitions, $A \in N$.

The parse table (2)



	action			goto		
	the	apple	John	NP	Det	N
0	s2		s1	6	3	
1			r2			
2			r3			
3		s4				5
4			r4			
5			r1			
6			acc			

1. NP → Det N
2. NP → John
3. Det → the
4. N → apple

The parse table (3)

- If a grammar does not allow for deterministic $LR(0)$ parsing, we can instead construct an $LR(k)$ parse table (with k lookahead symbols).
- Usually, $k = 1$ is used. Even if this is not deterministic, we can do the construction while ending up with a table with more than one entry in some of the fields (in case of shift/reduce or reduce/reduce conflicts).
- Idea of the lookahead: The lookahead of a dotted production is the next terminal that might follow that production.
  The lookaheads serve to restrict the application of reduce.
- The lookahead of $S' \to \bullet S$ is \$. Whenever predicting new items, their lookaheads are computed using the *First* sets.

The parse table (4)

Constructing a state (transitive closure of predict operation) with 1 lookahead:

```
closure(q)
    do until q does not change any more:
        for every ⟨A → α • Bβ, a⟩ ∈ q,
        every B → γ and every b ∈ First(βa):
            add ⟨B → •γ, b⟩ to q
```

The state one can reach from q by moving the dot over $X \in N \cup T$:

```
goto-state(q,X)
    return closure({⟨A → αX • β, a⟩ | ⟨A → α • Xβ, a⟩ ∈ q})
```

The parse table (5)

Construction of the set of states:

```
compute-states()
    Q = {closure({⟨S′ → •S, $⟩})}
    do until Q does not change any more:
        for every X ∈ N ∪ T and every q ∈ Q:
            Q = Q ∪ {goto-state(q, X)}
```

Now we can construct the table.

We assume the states to be numbered, $q_0 =$`closure(`$S' \to \bullet S$`, $)`. Furthermore, we assume the productions to be numbered.

The parse table (6)

Canonical LR(1) construction of the parse tables *action* and *goto*, both initialized with $\emptyset$ in all fields:

```
for every q_i ∈ Q, i ∈ [0..|Q| − 1]:
  for every a ∈ T and every q_j ∈ Q such that
    q_j = goto-state(q_i, a): add sj to action(i, a)
  if ⟨A → α•, a⟩ ∈ q_i and
    j number of A → α, then add rj to action(i, a)
  if ⟨S′ → S•, $⟩ ∈ q_i, then add acc to action(i, $)
for every q_i ∈ Q, i ∈ [0..|Q| − 1]:
  for every A ∈ N and every q_j ∈ Q such that
    q_j = goto-state(q_i, A): add j to goto(i, A)
```

The parse table (7)

Example

1. $S \to a$ 2. $S \to a\ S$

$q_0 =$ `closure`$(\{\langle S' \to \bullet\ S, \$\rangle\}) =$
$S' \to \bullet\ S, \$ \quad S \to \bullet\ a, \$ \quad S \to \bullet\ a\ S, \$$

$q_1 =$ `goto-state`$(q_0, S) =$
$S' \to S\ \bullet, \$$

$q_2 =$ `goto-state`$(q_0, a) =$
$S \to a\ \bullet, \$ \quad S \to a\ \bullet\ S, \$ \quad S \to \bullet\ a, \$ \quad S \to \bullet\ a\ S, \$$

$q_3 =$ `goto-state`$(q_2, S) =$
$S \to a\ S\ \bullet, \$$

`goto-state`$(q_1, a) = \emptyset$, `goto-state`$(q_1, S) = \emptyset$, `goto-state`$(q_2, a) = q_2$, `goto-state`$(q_3, a) = \emptyset$, `goto-state`$(q_3, S) = \emptyset$.

The parse table (8)

Example continued

Constructing the table:

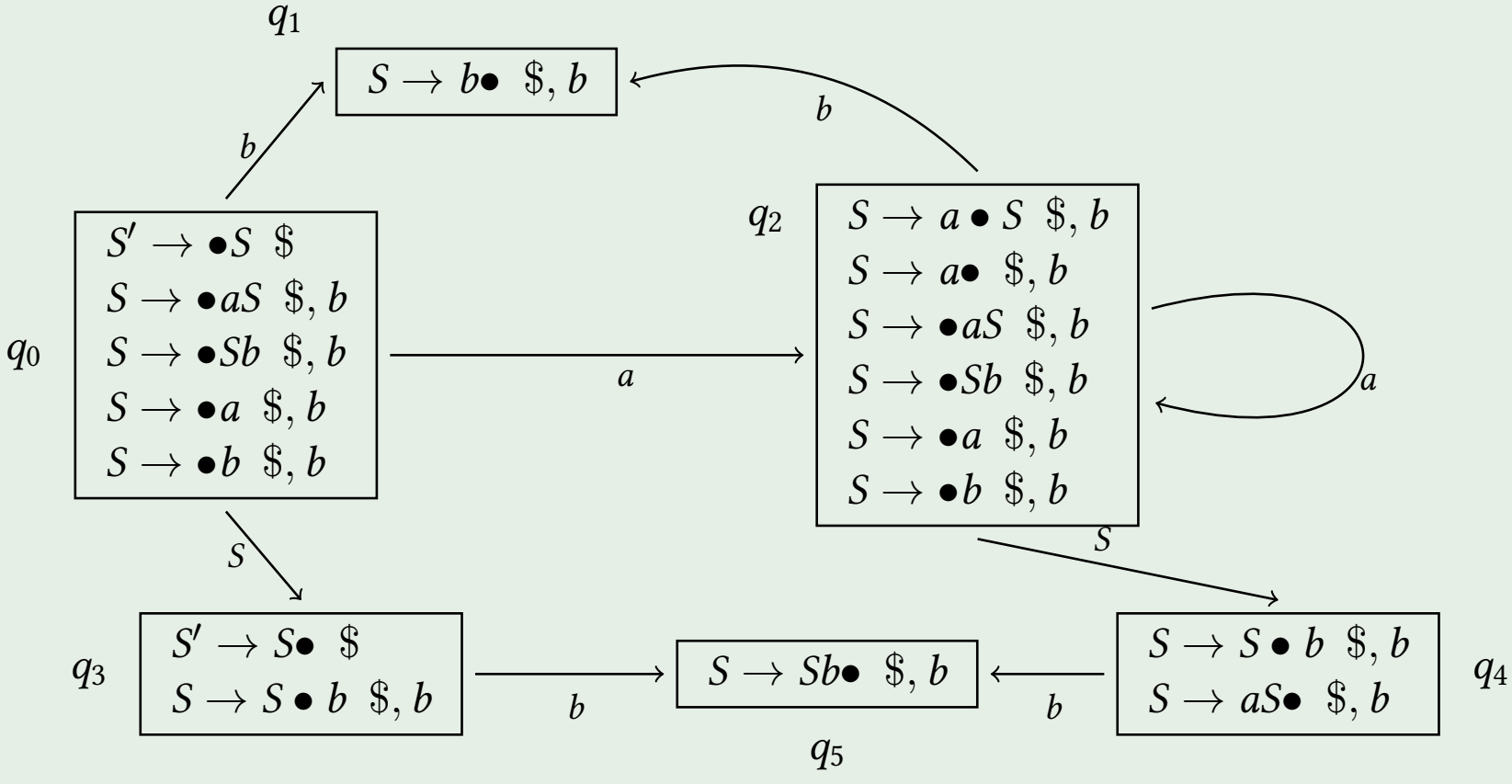
state	*action*		*goto*
	a	$	S
0	s2		1
1		acc	
2	s2	r1	3
3		r2	

The parse table (9)

Further example: canonical LR(1)

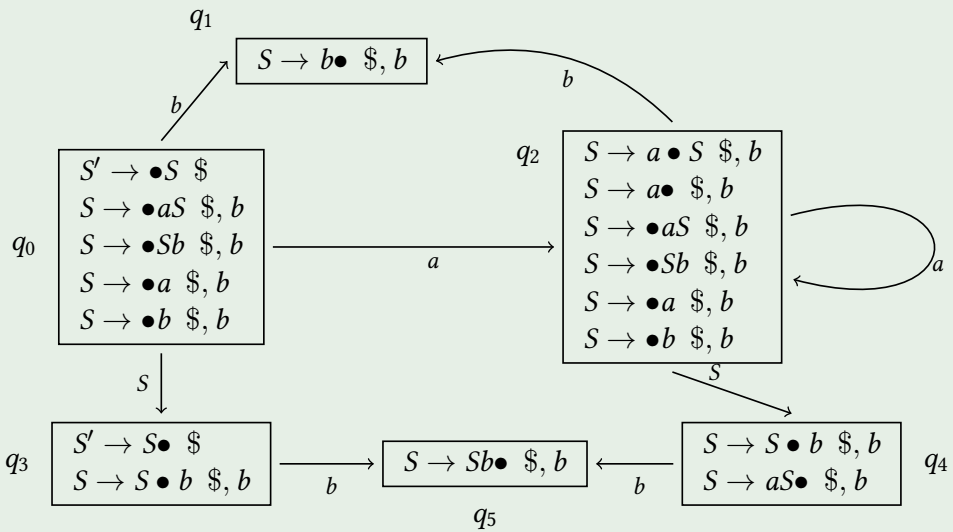
G with non-terminals $N = \{S\}$, terminals $T = \{a, b\}$, start symbol S and productions $S \to aS \mid Sb \mid a \mid b$

LR(1)-automaton for this grammar with canonical lookahead



The parse table (10)

Example continued



Parse table:

	a	b	$	S
0	$s2$	$s1$		3
1		$r4$	$r4$	
2	$s2$	$s1, r3$	$r3$	4
3		$s5$	acc	
4		$s5, r1$	$r1$	
5		$r2$	$r2$	

The grammar is not LR(1).

The parser

- The parse table determines the possible reductions and shifts we can do at every moment.
- We start with $\Gamma = q_0$. In a shift, we push the new terminal followed by the state indicated in the *action*-table on Γ.
- In a reduction, we use the production indicated in the *action* table. We pop its rhs (in reverse order) and push its lhs. The new state is the *goto*-value of 1. the state preceding the rhs of this production in Γ and 2. the lhs category of the production.
- In addition, we can push the number of the rules in reduction steps on a stack Δ_{rm} that gives us then the steps of the corresponding rightmost derivation.

LR(k)-grammars (1)

Question: How can we characterize the type of CFGs that allow for deterministic LR(k)-parsing?

- Deterministic means no shift-reduce or reduce-reduce conflicts.
- To avoid these, we must make sure that for every possible Γ, there is at most one reduce and, if so, then no shift can lead to a successful parse.

LR(k)-grammars (2)

We use the notion of the left context of a non-terminal and of a production to define LR(0)-grammars:

LR(0)

1. For each $X \in (N \cup T)^*$, the **left context of** X is $C_l(X) := \{\gamma \mid S \Rightarrow^*_{rm} \gamma X \alpha \text{ with } \gamma, \alpha \in (N \cup T)^*\}$.
   ($\Rightarrow_{rm}$ = rightmost derivation)
2. For every $A \to \alpha \in P$, the **left context of** $A \to \alpha$ is $C_l(A \to \alpha) := \{\gamma\alpha \mid \gamma \in C_l(A)\}$.
3. G is a **LR(0) grammar** iff for all pairwise different $A \to \alpha, A' \to \alpha' \in P$: no $\gamma \in C_l(A \to \alpha)$ is a prefixe of a $\gamma' \in C_l(A' \to \alpha')$.

LR(k)-grammars (3)

Example: LR(0)

NP $\rightarrow$ Det N, NP $\rightarrow$ John, Det $\rightarrow$ the, N $\rightarrow$ apple, start symbol NP

$C_l(N) = \{Det\}, \quad C_l(Det) = \{\epsilon\}, \quad C_l(NP) = \{\epsilon\}.$

$C_l(NP \rightarrow Det\ N) = \{Det\ N\}, \quad C_l(NP \rightarrow N) = \{N\},$
$C_l(Det \rightarrow the) = \{the\}, \quad C_l(N \rightarrow apple) = \{Det\ apple\},$
$C_l(NP \rightarrow John) = \{John\}.$

The grammar is LR(0).

LR(k)-grammars (4)

In an LR(k)-grammar, in addition, we consider the first k symbols following the rhs of the productions we compare:

LR(k)

A CFG G is LR(k) ($k \geq 0$) iff for every pair of right-most derivations

$S \Rightarrow^*_{rm} \alpha A w \Rightarrow_{rm} \alpha\beta w$ and

$S \Rightarrow^*_{rm} \gamma B x \Rightarrow_{rm} \alpha\beta y$

with $w, x, y \in T^*$, $A, B \in N$:

If $init_k(w) = init_k(y)$, then $\alpha A y = \gamma B x$, i.e. $\alpha = \gamma, A = B, x = y$.

($init_k$ = first k symbols)

LR(k)-grammars (5)

There are CFGs G such that there is no $k \geq 0$ with G being LR(k).

Example

$\mathrm{S} \to \mathrm{A\ X}$, $\mathrm{S} \to \mathrm{Y\ C}$, $\mathrm{X} \to \mathrm{B\ C}$, $\mathrm{Y} \to \mathrm{A\ B}$,
$\mathrm{A} \to \mathrm{a\ A}$, $\mathrm{A} \to \mathrm{a}$, $\mathrm{B} \to \mathrm{b\ B}$, $\mathrm{B} \to \mathrm{b}$, $\mathrm{C} \to \mathrm{c\ C}$, $\mathrm{C} \to \mathrm{c}$

For all $k \geq 0$, this grammar is not LR(k) since there are always right-most derivations

$$\mathrm{S} \Rightarrow \mathrm{Y\ C} \overset{k}{\Rightarrow} \mathrm{Y\ c}^k \Rightarrow \mathrm{A\ B\ c}^k$$

and

$$\mathrm{S} \Rightarrow \mathrm{A\ X} \Rightarrow \mathrm{A\ B\ C} \overset{k-1}{\Rightarrow} \mathrm{A\ B\ C\ c}^{k-1} \Rightarrow \mathrm{A\ B\ c}^k$$

LR(k)-grammars (6)

The following holds:

- Every LR(k)-grammar ($k \geq 0$) can be transformed into an equivalent LR(0)-grammar.
- There are CFLs that cannot be generated by any LR(0)-grammar.
- The class of languages generated by LR(0)-grammars is the class of deterministic CFLs, i.e., the class of languages recognized by deterministic PDAs.

Compact representations of the parse table

Deterministic LR-parsing is linear in the length of the input string. However, the construction of the parse table is quite expensive in time and space since the parse tables can get very large.

Compact representations of the parse table that preserve most of the look-ahead power:

- Look Ahead LR (LALR) automata. Idea: collapse states that distinguish one from the other only concerning their lookaheads. Replace lookahead information of a dotted production with sets containing all lookaheads for this item.
- Simple LR (SLR) automata. Idea: Instead of computing the lookahead sets based on the actual predictions, take the *Follow* set of the lhs of the dotted production as lookahead set.

Conclusion

- Bottom-up (shift-reduce) parsing with top-down restriction.
- Parsing is determined by a precomputed parse table.
- Can be done with different numbers of lookaheads.
- Deterministic for certain grammars but not for all CFGs.
- Problem: parse tables can get quite large.