Universitetet i Oslo
Institutt for Informatikk
Reliable Systems
Martin Steffen

INF 5110: Compiler construction

Spring 2024 **Series 4** 26. 4. 2024

Topic: Chapter 5: LR parsing (Exercises with hints for solution)

Issued: 26. 4. 2024

Exercise 1 (LR(0)-items, SLR(1) parsing) Consider the following grammar for well-balanced parentheses:

$$S \rightarrow S\,(\,S\,) \mid \epsilon$$

1. Construct the DFA of LR(0) items for the grammar.
2. Construct the SLR(1) parsing table.
3. Show the parsing stack and the actions of an SLR(1) parser for the input string

$$(\,(\,)\,(\,)\,)\ .$$

4. Is the grammar LR(0)? If not, describe a resulting LR(0) conflict. If yes, construct the LR(0) parsing table and describe how a parse might differ from an SLR(1) parse.

Solution: In the lecture, we had almost the same grammar. The grammar here is left-recursive. That makes it non-LL(1), which is not part of this exercise, as we are now focusing on bottom-up parsing. The "other one" from the lecture, with the production $S \rightarrow (\,S\,)\,S$, is not left-recursive and was LL(1).

DFA For the construction, we have learnt two ways (with ultimately the same outcome). The first construction is via an NFA (a non-deterministic finite automaton), with subsequent determinization, the other one goes directly to the DFA. I think, the direct one is faster to do.

The construction should, for a written exam, be routine and comparatively fast. Apart from knowing what an LR(0) item is and being able to list them (which is easy), it involves the following routine steps:

- build the *closure* of a set of items. It's not directly the ϵ-closure construction as learnt earlier in the lecture,[1] at least not in the way it's done (i.e., the construction does not mention any ϵ). But it **does** correspond to the ϵ-construction *if* one would have taken the route over the NFA, which contains a lot of ϵ-transitions.

  It involves looking at (partially constructed) states containing an item where the . stands in front of a *non-terminal* and then *add* the corresponding right-hand sides *as*

[1] In the lecture, the ϵ-closure was introduced in connection with determinizing NFA with ϵ-transitions.

initial items. That process needs to be continued till saturation.[2] It's fairly easy to do in the example given in this exercise.

Once done, one can already look out for conflicts. In absence of first- and follow-sets, one can check only for LR(0)-conflicts.[3] In this particular situation, there's a **shift/reduce conflict in state 1**. Note also that states 0 and 2 do **not** indicate conflicts. It may seems that one could do either an S transition or a reduce: *but*: "S does not count": Doing a transition on S is *not a shift* (it does not eat an input). So: reduction is the only alternative in 0 and 2.

One can also ponder if there's a reduce/reduce conflict: one has to look at the completed items as they indicate the production used for reduce steps. No state contains more than one completed item, so we are gold here.

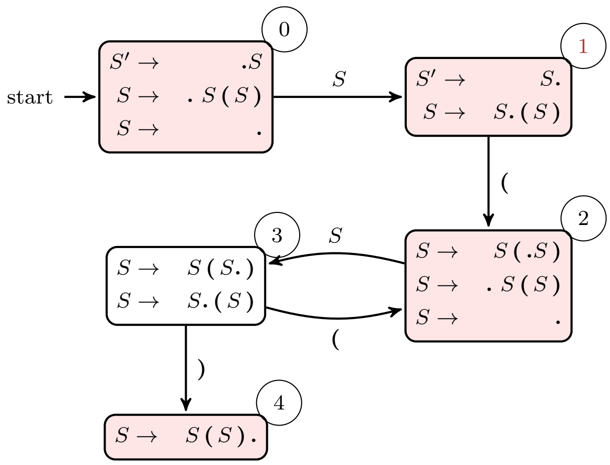

Figure 1: LR(0)-DFA

To sum up the above discussion: The automaton is **not LR(0)**. See *state* 1. It contains a *complete item* (which means a reduce step is possible) and an outgoing transition (labeled with the left-parenthesis).

States 0 and 2 are *not* problematic in the same way! Reduction there is the only way to react.

SLR(1) parsing table For the SLR(1) parsing approach: the first thing to do is always: (the first and the) follow sets.[4] Since, except for state 1, the situation is LR(0) already, we need to check if the follow-sets are enough to defuse that particular conflict. And that means, we need the follow-set of S', only, not the follow set of all symbols. See the information contained in Figure 1.

To see for SLR(1), we have to look at state 1 again, which broke the LR(0)-property: The automaton from Figure 1 *is SLR(1)*. *Shift* is done for **(** and *reduce with rule 0* is done for **$**. Alternatively one could say, the corresponding table is "unambiguous" (each slot contains at most one entry). See Table 1. In the table, the reduce steps contain an "r" for reduce *and* the rule. The rules numbered from 0 to 2, as in the following grammar:

[2] That's why it's called "closure".

[3] For LR(0) conflicts, one does not need the first- and follow sets. One will need them for more complex conditions, as for SLR(1)-parsing.

[4] For the question of SLR(1) conflicts, we are interested primarily in the follow-sets, but to determine those systematically, we need the first-sets first.

$$
\begin{array}{rcl}
S' & \rightarrow & S \\
S & \rightarrow & S\,(\,S\,) \\
S & \rightarrow & \epsilon
\end{array}
$$

Remember: we routinely need an extra start system, say S', which gives 3 rules, as opposed to only 2. In the lecture, the tables were shown containing not numbers of the rules, but the rules themselves "copied in" (but it's just a representational issue). Note also the *accept* slot. It corresponds to the reduction with the "extra start production" $S' \rightarrow S$, which is here numbered as production **0**. The numbers after the "shifts" s of course don't refer to the number of a production, but to a state (the production numbers are written **boldface** here for make clear that here are two different "numbers").[5]

state	input			goto
	(	)	**$**	S
0	$r : \mathbf{2}$	$r : \mathbf{2}$	$r : \mathbf{2}$	1
1	$s : \mathbf{2}$		accept/$r : \mathbf{0}$	
2	$r : \mathbf{2}$	$r : \mathbf{2}$	$r : \mathbf{2}$	3
3	$s : 2$	$s : 4$		
4	$r : \mathbf{1}$	$r : \mathbf{1}$	$r : 1$	

Table 1: SLR(1) table

Stack and reduction: If the rest has been done properly, this one is rather easy. See Table 2

stage	parsing stack	input	action
1	$\$_0$	$(\,(\,)\,(\,)\,)\,\$$	reduce[$S \rightarrow \boldsymbol{\epsilon}$]
2	$\$_0 S_1$	$(\,(\,)\,(\,)\,)\,\$$	shift
3	$\$_0 S_1 (_2$	$(\,)\,(\,)\,)\,\$$	reduce[$S \rightarrow \boldsymbol{\epsilon}$]
4	$\$_0 S_1 (_2 S_3$	$(\,)\,(\,)\,)\,\$$	shift
5	$\$_0 S_1 (_2 S_3 (_2$	$)\,(\,)\,)\,\$$	reduce[$S \rightarrow \boldsymbol{\epsilon}$]
6	$\$_0 S_1 (_2 S_3 (_2 S_3$	$)\,(\,)\,)\,\$$	shift
7	$\$_0 S_1 (_2 \underline{S_3 (_2 S_3)_4}$	$(\,)\,)\,\$$	reduce[$S \rightarrow S\,(\,S\,)$]
8	$\$_0 S_1 (_2 S_3$	$(\,)\,)\,\$$	...
9	$\$_0 S_1 (_2 S_3 (_2$	$)\,)\,\$$	
10	$\$_0 S_1 (_2 S_3 (_2 S_3$	$)\,)\,\$$	
11	$\$_0 S_1 (_2 \underline{S_3 (_2 S_3)_4}$	$)\,\$$	
12	$\$_0 S_1 (_2 S_3$	$)\,\$$	
13	$\$_0 \underline{S_1 (_2 S_3)_4}$	$\$$	
14	$\$_0 S_1$	$\$$	
15	accept	$\$$	

Table 2: Reduction

LL(0)? See the remark earlier (no LL(0)).

□

Exercise 2 (LR(1) parsing)

[5] Of course, in a one can also write down the rules themselves, as in the lecture. Or else have states A, B, C, and rules 0, 1, 2.

1. Show that the following grammar is not LR(1):

$$A \to \mathbf{a}A\mathbf{a} \mid \boldsymbol{\epsilon}$$

2. Is the grammar ambiguous or not?

Solution: Note that the grammar is pretty similar to the "simple parentheses" grammar we had in the lecture. In this example (unlike what we mostly concentrate on) we are doing an automaton based on *LR(1) items* not *LR(0)* items. That may lead to pretty large constructions, but in the exercise, the grammar is still simple. Also the construction of LR(1)-DFAs is rather similar to the one we concentrated on in the lecture.

We start initially with **\$** as extra element in the initial item (initial state). The lecture did not 100% cover the *direct* construction, i.e., the one that directly give the DFA, we only showed the construction of the NFA. Here we show the DFA (without detour over the NFA), but the ϵ-closure works analogous to the LR(0) case. We only have to keep in mind that the ϵ-transitions now also have do deal with the look-ahead part of the items, and deal with them in a non-trivial manner.

Indeed, the new thing is the treatment of the look-ahead in the items. We If we want to show the direct construction, we need to understand/remember how the NFA construction worked. The starting point of the whole construction are now **LR(1)-items**. Those correspond to LR(0)-items **plus one look-ahead**, one extra terminal as part of the item (written after the comma), that is interpreted as look-ahead.

The non-$\boldsymbol{\epsilon}$-transitions are easy, because they *keep* that extra non-terminal look-ahead. But not the $\boldsymbol{\epsilon}$-transitions: They change it! In a situation like

$$[A \to \alpha.B\gamma, \mathbf{a}]$$

with the . in front of a nonterminal, there will be an ϵ-transition. For aq production $B \to \beta$, there is an ϵ transition to a state with the corresponding initial item and where the look-ahead is an *arbitrary non-terminal* **b** from the **first set of the** $\gamma\mathbf{a}$ for an item of the form $A \to \alpha.B\gamma$

$$[A \to \alpha.B\gamma, \mathbf{a}] \xrightarrow{\boldsymbol{\epsilon}} [B \to .\beta, \mathbf{b}] \tag{1}$$

With that knowledge that, the construction of the DFA is straightforward. See Figure 2.

Wrt. the construction (and the effect of $\boldsymbol{\epsilon}$ from equation (1)), let's look at two states (the others work similar).

Let's start with the initial state 0. Given $A' \to .A\ \$$, we need to add closure for A's productions (there are two production. Now, since A is at the end (i.e. γ from above is empty), we have to take the first set of $\epsilon\,\$$ (= \$) into account. This first-set contains just one symbol, the look-ahead **\$** itself. That explains the second and the third item in state 0.

For *state 2*, we have to bute the $\boldsymbol{\epsilon}$-closure for $A \to \mathbf{a}.A\mathbf{a}$. This time we have to take the follows-set of $\mathbf{a}\,\$$ into account, and that contains just **a**. This explains the look-aheads in the second and third item in state 2.

Now that we have constructed the LR(1)-automaton, is the grammar LR(1)-parseable? Let's look at states 2 and 4. In both states, there's a shift and a reduce possible, with **a** as next symbol.

Thus, the grammar is not LR(1)!

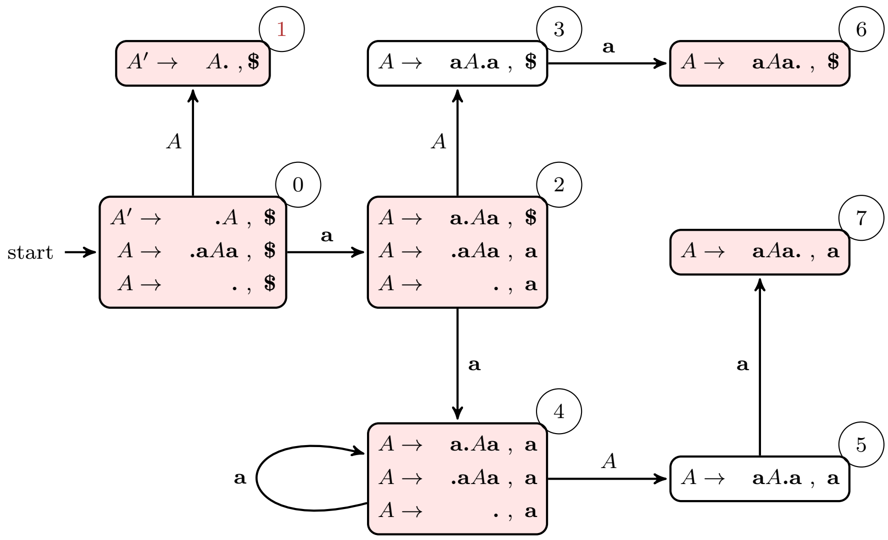

Figure 2: LR(1)-DFA

Note also state 0: there is a shift and a reduce possible, but the shift takes an **a**, but the reduce has a look-ahead of **\$** as part of the LR(1) item. Note in that context: if in that state 0, the shift is done (with **a**), what we called the look-ahead part of the item is (also) **\$**. The look-ahead of an item refers to what comes after one has parsed the right-hand side, not the first thing that happens next (like **a** in the second item of state 0.

The grammar does all words with an even number of **a**'s. For each such word, there is exactly one parse-tree of the form of Figure 3. In other words: the grammar is **unambiguous.**

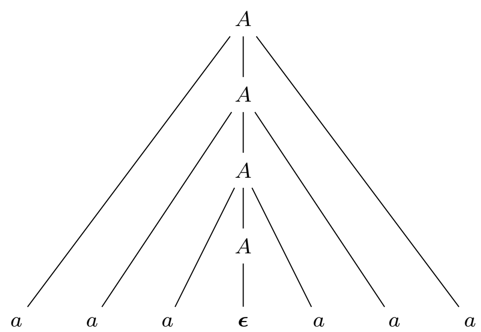

Figure 3: Parse tree

Some extra remarks beyond the actual question from the exercise: the grammar is actually *not* LR(k), for any k. It might be worthwhile to reflect upon why that intuitively is the case. It has to do (intuitively) with the fact, that while doing the parsing, the parser would not know when it has reached the "middle" of the word. It could do some guessing (using "non-determinism") but anyway, that would not be an LR-parser.

We may compare the situation also to the grammar

$$A \to \mathbf{a_1} A \mathbf{a_2} \;\mid\; \boldsymbol{\epsilon}$$

The grammar corresponds to one of the grammars for "parentheses" .... See also the following 2 grammars which produce the same language than the one from the task:

$$\begin{array}{rcl} A & \to & A\mathbf{aa} \mid \boldsymbol{\epsilon} \\ A & \to & \mathbf{aa}A \mid \boldsymbol{\epsilon} \end{array}$$

Those are SLR(1)!

$\square$

Exercise 3 (Bottom-up parsing) The following ambiguous grammar generates the same language as the grammar of Exercise 1 in this collection (namely all strings of well-balanced parentheses):

$$A \to AA \ | \ \textbf{(}\, A \,\textbf{)} \ | \ \epsilon \tag{2}$$

Will a `yacc`-generated parser using this grammar recognize all legal strings? Why or why not?

Extra: Try to change the order: put the production $A \to AA$ at the end.

Solution: The grammar accepts the same language as other well-known ones. But the question is about the grammar, not the language. The new one is clearly ambiguous. Already for a string of terminals **( )**, there is more than one parse tree (see Figure 4).

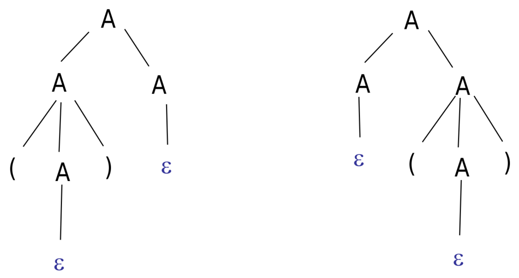

Figure 4: Two parse trees for **( )**

$$\mathit{Follow}(A) = \{\textbf{(}, \textbf{)}, \epsilon\}$$

- We see *many* shift-reduce conflicts: in those cases we do *shifts*.
- There is a *reduce/reduce* conflict in state 5.[6] The question is what would yacc (and cup) do (besides warning us about it). The answer: it would chose to reduce according to the production which comes first (among those which are available for a *reduce*-step at the given point). In the given grammar from equation (2), there are *three* productions, and the order is from left to right. In state 5, there are therefore production 1) and production 3)

  $$A \to AA \text{ and } A \to \epsilon$$

  available for reduction. Since 1) is first, that takes priority. More concretely:

  with **$** or **)** as next symbol, state 5 will reduce according to production **1**

  That choice is encoded also in Table 6. See Figure 7 for a reduction.

  Note: if we had written the grammar differently, swapping the order of 1) and 3), state 5 would obviously reduce according to $A \to \epsilon$. In that case, the original language would *not* be parsed properly!

□

[6] In state 5, there are *also* shift-reduce conflicts. As mentioned, if a shift is possible, that *uniformly* takes precedence over reduce-steps, in yacc-style conventions. Specifically discussed here is, what happens if in state 5 *no* shift is possible, in which case we have to decide between the two reduce-steps.

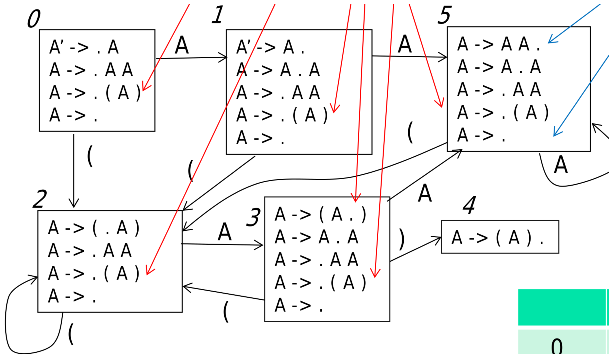

Figure 5: LR(0)-DFA

state	input			goto
	(	)	$	A
0	$s:2$	$r:3$	$r:3$	1
1	$s:2$		accept/$r:0$	5
2	$s:2$	$r:3$	$r:3$	3
3	$s:2$	$s:4$		5
4	$r:2$	$r:2$	$r:2$	
5	**s : 2**	**r : 1**	**r : 1**	5

Figure 6: Table

Exercise 4 (Priorities & associativity by manual conflict resolution) Take the following variant of the "expression grammar"

$$\begin{array}{rcl} exp' & \rightarrow & exp \\ exp & \rightarrow & exp + exp \mid exp * exp \mid \mathbf{n} \end{array}$$

and extend it with exponentiation as follows

$$\begin{array}{rcl} exp' & \rightarrow & exp \\ exp & \rightarrow & exp + exp \mid exp * exp \mid exp \uparrow exp \mid \mathbf{n} \end{array}$$

Assume that the usual associativities and precedences are intended (which includes right-associativity for exponentiation).

Now: indicate how *conflicts* in an LR-parse-table are to be resolved (if possible) to obtain the indicated behavior.

Solution:

The first Figure 8 shows the LR(0)-DFA for the grammar *without exponentiation*. Figure 9 later shows the one for the grammar *with* exponentiation (there written as `**`).

The grammar is ambiguous, but does not contain ϵ-productions. The first and follow sets are therefore rather straightforward to compute. Indirectly, the high ambiguity of the grammar is reflected by the fact that the follow-set of *exp* contains all terminals (and additionally $).

Already for the simpler DFA from Figure 8, there are consequently many conflicts. We should have a closer look at the following three states (again the simpler Figure 8 first, even if the automaton of Figure 9 is the one which the exercise is really about).

State 5: There is a shift-reduce conflict (check again the follow-set of E against the outgoing "shift"-edges). We have 3 symbols to consider $+$, $*$, and $\$$. The shift-reduce conflict is

stage	parsing stack	input	action
1	$\$_0$	( ) ( ) $	
2	$\$_0(_2$	) ( ) $	
3	$\$_0(_2 A_3$	) ( ) $	
4	$\$_0(_2 A_3)_4$	( ) $	
5	$\$_0 A_1$	( ) $	
6	$\$_0 A_1 (_2$	) $	
7	$\$_0 A_1 (_2 A_3$	) $	
8	$\$_0 A_1 (_2 A_3)_4$	$	
9	$\$_0 A_1 A_5$	$	
10	$\$_0 A_1$	$	

Figure 7: Reduction

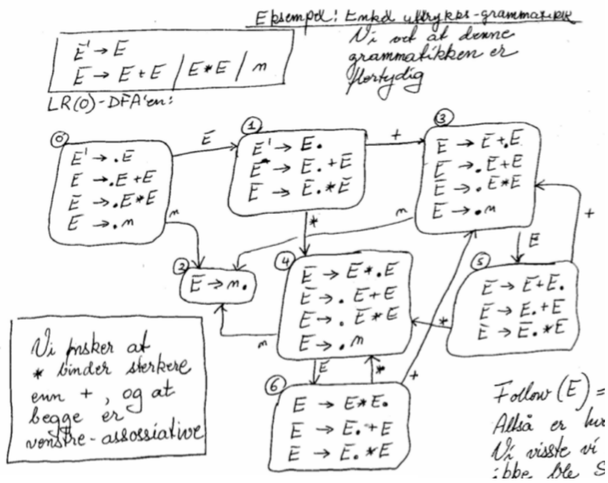

Figure 8: LR(0)-DFA

on $+$ and $*$. Now, the *manual* disambiguation, we are requested to do, is done as follows. Note, that in the given state, the stack contains

$$exp + exp$$

"on top". That observation is the key to understand what to do.

- $: reduce is the only option, we cannot shift $.
- $+$: pick *reduce*. Reason: $+$ is *left-associative*. The mental picture is as follows. We have just parsed a plus-expression, as that's now on the top of the stack. Now we see that there is another $+$ coming. To pick the right reaction then is a question of *associativity*. We should arrange the reaction in such a way that the already parsed addition, the one on the stack is "handled first". Handled means, we do a reduce. Doing that reduce step builds up the parent node of the mentioned $exp + exp$, that's the bottom-up working of the parser. Since $+$ is intended to be left-associative, that's exactly what we need to do in the current situation: a *reduce* step wrt. the corresponding production.
- $*$: *shift*. Reason: $*$ has precedence over $+$. The reason is now sort of opposite from the previous sub-case. We have to arrange it in such a way that it's *not* $(exp + exp) * \ldots$. That would be the result of a reduce-step. Instead we need to glue the "second" expression to whatever comes after the $*$, indicated by "…" but we don't have a look-ahead to know already what it concretely is. Thus we pick the shift-option

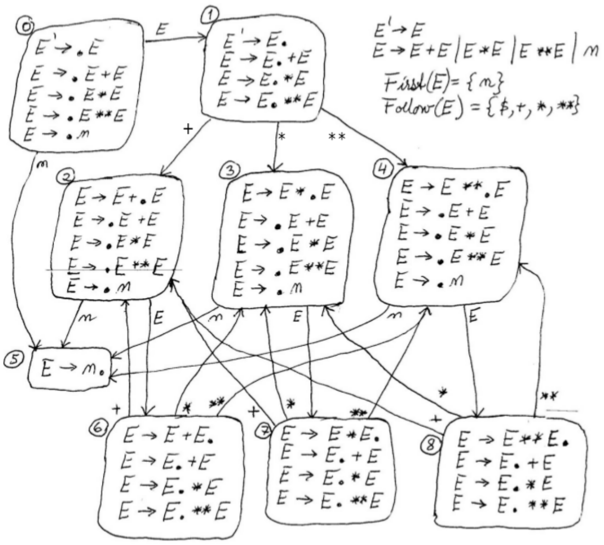

Figure 9: LR(0)-DFA

State 6: The stack now contains $exp * exp$ on top.[7] The argument of what to choose works analogous to the previous one.

- **$**: reduce, same argument as above.
- $+$: *reduce*, since $*$ as precedence over $+$, we therefore need to treat the $*$ “first”.
- $*$: *reduce*, as $*$ is left-assoc.

state	input				goto
	n	$+$	$*$	**$**	*exp*
0	$s:2$				1
1		$s:3$	$s:4$	accept	
2		$r:(exp \to \mathbf{n})$	$r:(exp \to \mathbf{n})$		
3	$s:2$				5
4	$s:2$				6
5		$r:(exp \to exp + exp)$	$s:4$	$r:(exp \to exp + exp)$	
6		$r:(exp \to exp * exp)$	$r:(exp \to exp * exp)$	$r:(exp \to exp * exp)$	

Figure 10: Parse table (simpler grammar)

For the more complex DFA from Figure 9, we do it analogously. Focus on the three states 6, 7, and 8.

State 6: The stack now contains $exp + exp$

- **$**: same argument as above only option, we cannot shift **$**
- $+$: reduce $(exp \to exp + exp)$ (left-assoc)
- $*$: *shift 3*
- $\uparrow$: *shift 4*

[7]How do we actually know that?

State 7: Note, the top of the stack now contains $exp * exp$

- **\$**: same argument as above
- $+$: *reduce* $(exp \to exp * exp)$,
- $*$: *reduce* $(exp \to exp * exp)$ (left-assoc). Note it's the same production than the first case (of course).
- $\uparrow$: *shift 4*

State 8: Note, the top of the stack now contains $exp \uparrow exp$

- **\$**: same argument as above
- $+$: *reduce* $(exp \to exp \uparrow exp)$
- $*$: *reduce* $(exp \to exp \uparrow exp)$
- $\uparrow$: *shift* 4 (right-assoc)

Note: all that is done automatically (in yacc, CUP etc), if one gives the associativities and precedences appropriately

Exercise 5 (Bottom-up parsing routine) [8] Consider the following grammar G, where S is the start symbol, and the terminals as $\#$ and **a**

$$
\begin{array}{rcl}
S & \to & T\ S \\
S & \to & T \\
T & \to & \#\, T \\
T & \to & \mathbf{a}
\end{array}
$$

Now do:

1. calculate the first and follow sets of S and T. Use, as in the lecture, **\$** to stand for the end-of-input.
2. formulate, in your own words, which words of terminals are derivable from S.[9]
3. Decide if you can formulate a regular expression that captures words of $\#$ and **a** derivable from S.[10] If the answer is yes, give a regular expression that captures the language.
4. Introduce a new start symbol S' and construct the LR(0)-DFA for G directly from that grammar. Enumerate the states.
5. Give the parsing table for that grammar, and let the type of the grammar should determine the form of the parsing table.
6. Show how

   $$\mathbf{a}\,\#\,\mathbf{a}$$

   is being parsed; do that in the form presented in the book/lecture, making use of the yet-to-parse input and the stack and indicate the shift and stack operations appropriately during the parsing process.

Solution:

1. The first and follow set are easy, since ϵ is not used (which implies there are no nullable symbols.)

[8] It corresponds to an exam question from 2006, minus one sub-question.
[9] The "language of S".
[10] Is $\mathcal{L}(G)$ regular?

	First	*Follow*
S	**a**, #	**$**
T	**a**, #	**a**, #, **$**

Table 3: First and follow

2. The language could be described as follows following form,

   It consists of words containing one or more **a**'s, where each **a** is preceded by *zero* or more #.

3. As could be seen by the previous informal description, a regular expression capturing the same language could be
$$(\#^*\mathbf{a})^+ \ .$$

4. The LR(0)-DFA is given in Figure 11. Note: the grammar does not contain an ϵ-production. Nonetheless: the states *do contain ϵ-closure.* You may reflect on that.

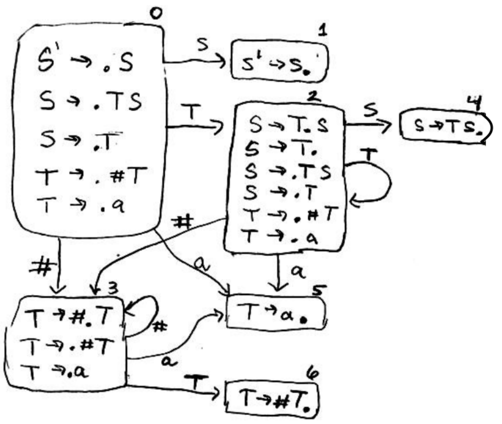

Figure 11: LR(0)-DFA

Looking at the automaton: the grammar is *not* LR(0). We see that in state 2 (there's a shift-reduce conflict). There is no reduce-reduce conflict.

For SLR(1): concentrate in the conflicting state 2 and see if it "goes away" using the technique which SLR(1)-parsing is built upon. That requires looking at the *Follow*-sets (which are done earlier in this exercise).

Important: One common trap here is to apply the *Follow*-set considerations onto the "wrong" symbols. See the slide called "Resolving LR(0) shift-reduce conflicts". In particular: in the conflicting state 2, the follow set of T is *irrelevant*, it's only the follow set of S which counts, as S is the left-hand side of the production which corresponds to the *complete* item in the state!

In an exam, making an argument about the follow-set of non-relevant symbols (like T here) would reduce points not only if this would lead a wrong outcome. In the example here: considering T would lead to the erroneous conclusion that there seem to be a SLR-conflict.

But: since $\mathit{Follow}(S) \cap \{\#, \mathbf{a}\} = \emptyset$, everything is fine, the grammar is SLR(1).

Since the grammar is SLR(1), it's immediately also LALR(1) and LR(1).

5. The corresponding SLR(1) parsing table is given in Figure 12. Note that the line for state 2 contains *shifts* and *reduce* steps (actually one reduce entry). That's not directly taken from the automaton, of course. It's the disambiguation done via the follow-set consideration from above (in particular here for S).

state	input			goto	
	a	$\#$	$\$$	S	T
0	$s:5$	$s:3$		1	2
1			accept		
2	$s:5$	$s:3$	$r:(S \to T)$	4	2
3	$s:5$	$s:3$			6
4			$r:(T \to T\ S)$		
5	$r:(T \to \mathbf{a})$	$r:(T \to \mathbf{a})$	$r:(T \to \mathbf{a})$		
6	$r:(T \to \# T)$	$r:(T \to \# T)$	$r:(T \to \# T)$		

Figure 12: SLR(1) table

6. The requested reduction is given in Figure 13

stage	parsing stack	input	action
1	$\$_0$	$\mathbf{a}\,\#\,\mathbf{a}\,\$$	shift
2	$\$_0\mathbf{a}_5$	$\#\,\mathbf{a}\,\$$	reduce $[T \to \mathbf{a}]$
3	$\$_0T_2$	$\#\,\mathbf{a}\,\$$	shift
4	$\$_0T_2\#_3$	$\mathbf{a}\,\$$	shift
5	$\$_0T_2\#_3\mathbf{a}_5$	$\$$	reduce $[T \to \mathbf{a}]$
6	$\$_0T_2\#_3T_6$	$\$$	reduce $[T \to \# T]$
7	$\$_0T_2T_2$	$\$$	reduce $[S \to T]$
8	$\$_0T_2S_4$	$\$$	reduce $[S \to T\ S]$
9	$\$_0S_1$	$\$$	accept (reduce $[S' \to S]$)

Figure 13: Reduction of $\mathbf{a}\,\#\,\mathbf{a}$

$\square$