Bottom-up parsing

SLR parsing (2.13 [M])

Simple | Rightmost derivation

Input read Left-to-Right

A kind of bottom-up parsing

A simplified version of LALR(1) parsing, which is the technique most commonly used in parser generators.

LR parsers are also called "shift-reduce" parsers.

shift: A symbol is read from the input and pushed on the stack.

reduce: The top N elements of the stack hold symbols identical to the N symbols on the right-hand side of a specified production. These N symbols are by the reduce action replaced by the nonterminal at the left-hand side of the specified production. Contrary to LL(1) parsers, the stack holds the right-hand-side symbols such that the *last* symbol on the right-hand side is at the top of the stack.

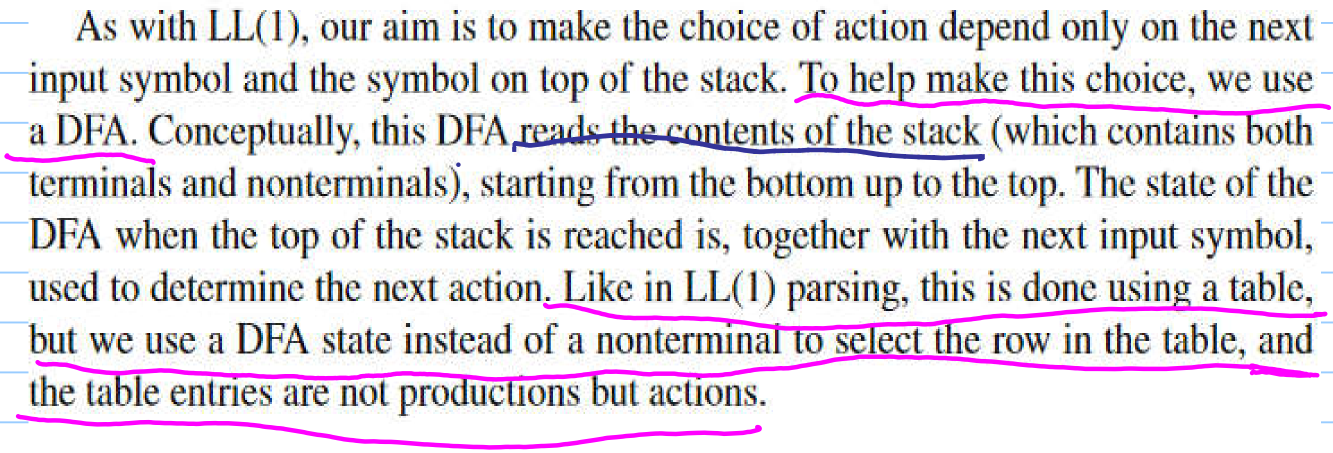
stack	input	action
	aabbbcc	shift 5 times
aabbb	cc	reduce with $R \rightarrow$
aabbb$R$	cc	reduce with $R \rightarrow$ bR
aabb$R$	cc	reduce with $R \rightarrow$ bR
aab$R$	cc	reduce with $R \rightarrow$ bR
aa$R$	cc	reduce with $T \rightarrow R$
aa$T$	cc	shift
aa$T$c	c	reduce with $T \rightarrow$ aTc
aT	c	shift
aTc	c	reduce with $T \rightarrow$ aTc
T		Success!

$$
\begin{aligned}
T &\rightarrow R \\
T &\rightarrow \mathrm{a}T\mathrm{c} \\
R &\rightarrow \\
R &\rightarrow \mathrm{b}R
\end{aligned}
$$

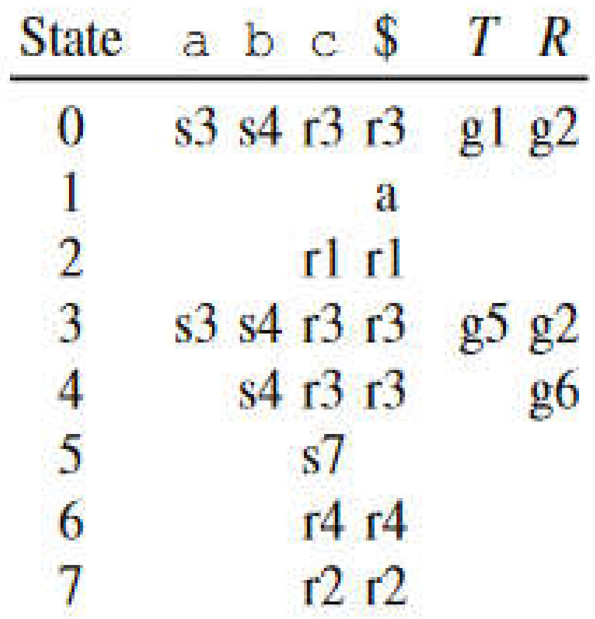
$T \Rightarrow aTc \Rightarrow aaTcc \Rightarrow aaRcc \Rightarrow aabRcc \Rightarrow$

$\Rightarrow aabbRcc \Rightarrow aabbbRcc \Rightarrow aabbbcc$

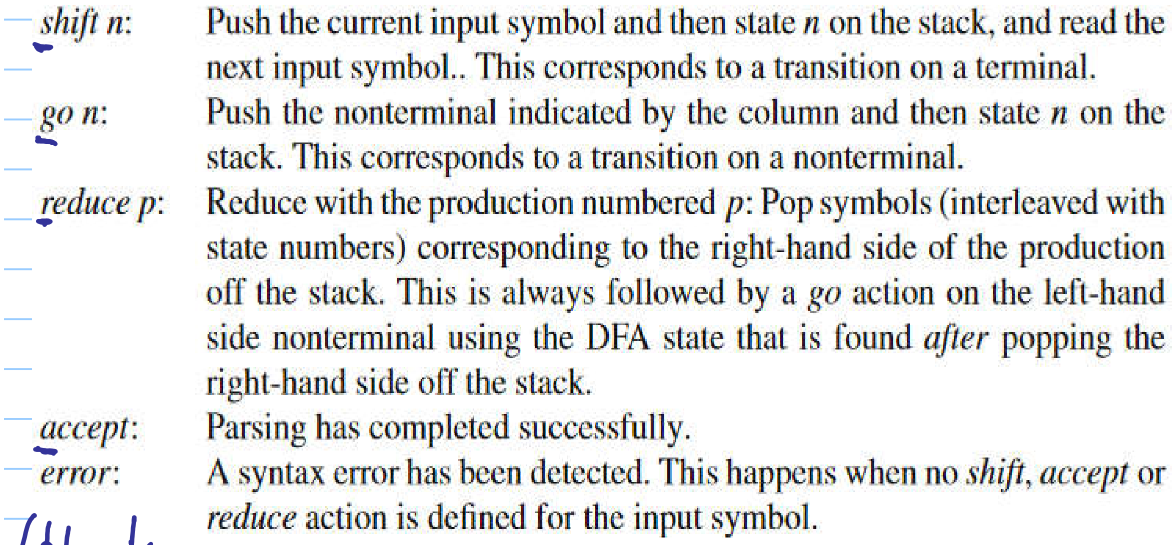
The choice of action (shift or reduce) depends on the next input symbol and the symbols on the stack,



As with LL(1), our aim is to make the choice of action depend only on the next input symbol and the symbol on top of the stack. To help make this choice, we use a DFA. Conceptually, this DFA reads the contents of the stack (which contains both terminals and nonterminals), starting from the bottom up to the top. The state of the DFA when the top of the stack is reached is, together with the next input symbol, used to determine the next action. Like in LL(1) parsing, this is done using a table, but we use a DFA state instead of a nonterminal to select the row in the table, and the table entries are not productions but actions.



State	a	b	c	$	*T*	*R*
0	s3	s4	r3	r3	g1	g2
1				a		
2			r1	r1		
3	s3	s4	r3	r3	g5	g2
4		s4	r3	r3		g6
5			s7			
6			r4	r4		
7			r2	r2		



shift n: Push the current input symbol and then state *n* on the stack, and read the next input symbol.. This corresponds to a transition on a terminal.

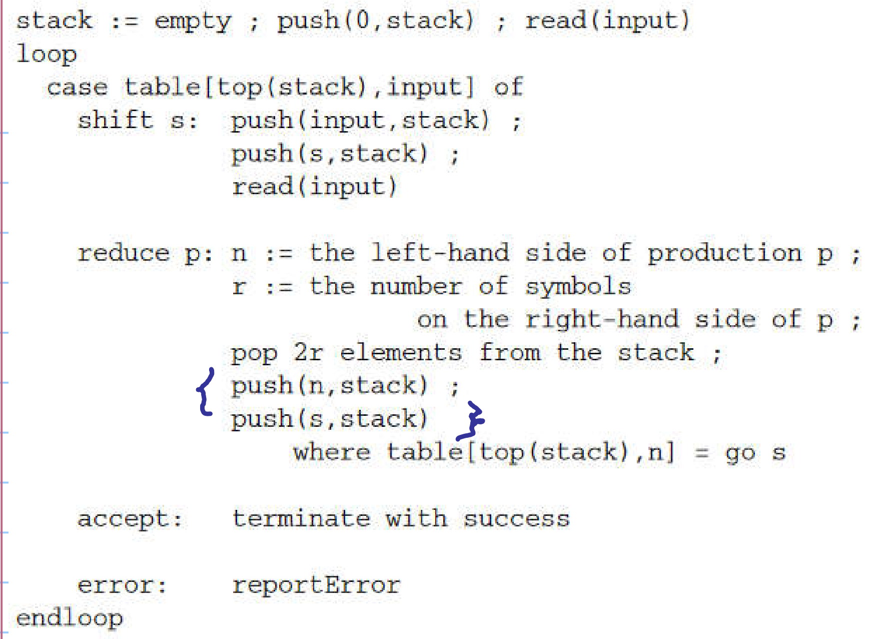
go n: Push the nonterminal indicated by the column and then state *n* on the stack. This corresponds to a transition on a nonterminal.

reduce p: Reduce with the production numbered *p*: Pop symbols (interleaved with state numbers) corresponding to the right-hand side of the production off the stack. This is always followed by a *go* action on the left-hand side nonterminal using the DFA state that is found *after* popping the right-hand side off the stack.

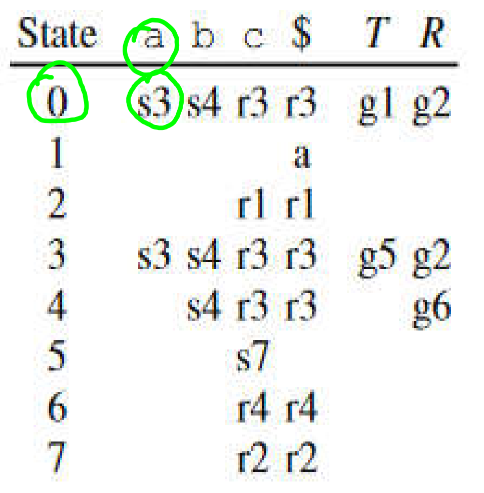
accept: Parsing has completed successfully.

error: A syntax error has been detected. This happens when no *shift*, *accept* or *reduce* action is defined for the input symbol.

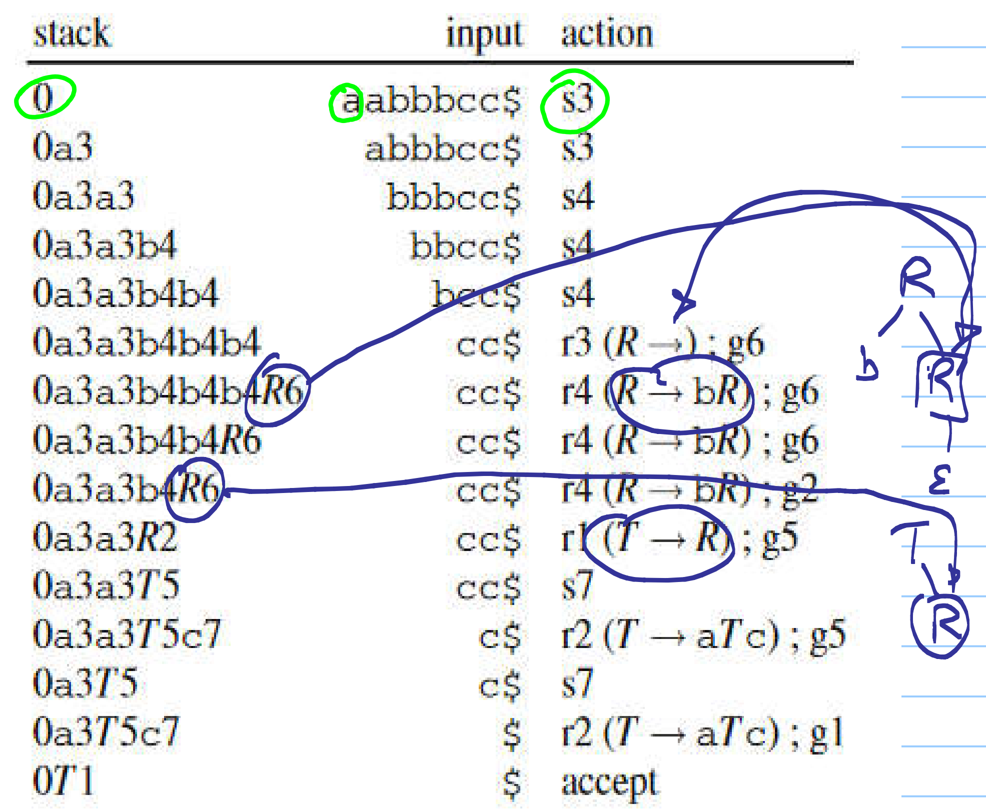
(blank entries in the table)



```
stack := empty ; push(0,stack) ; read(input)
loop
  case table[top(stack),input] of
    shift s:  push(input,stack) ;
              push(s,stack) ;
              read(input)

    reduce p: n := the left-hand side of production p ;
              r := the number of symbols
                           on the right-hand side of p ;
              pop 2r elements from the stack ;
              push(n,stack) ;
              push(s,stack)
                  where table[top(stack),n] = go s

    accept:   terminate with success

    error:    reportError
endloop
```



State	a	b	c	$	*T*	*R*
0	s3	s4	r3	r3	g1	g2
1				a		
2			r1	r1		
3	s3	s4	r3	r3	g5	g2
4		s4	r3	r3		g6
5			s7			
6			r4	r4		
7			r2	r2		



stack	input	action
0	aabbbcc$	s3
0a3	abbbcc$	s3
0a3a3	bbbcc$	s4
0a3a3b4	bbcc$	s4
0a3a3b4b4	bcc$	s4
0a3a3b4b4b4	cc$	r3 ($R \to$) ; g6
0a3a3b4b4b4*R*6	cc$	r4 ($R \to$ bR) ; g6
0a3a3b4b4*R*6	cc$	r4 ($R \to$ bR) ; g6
0a3a3b4*R*6	cc$	r4 ($R \to$ bR) ; g2
0a3a3*R*2	cc$	r1 ($T \to R$) ; g5
0a3a3*T*5	cc$	s7
0a3a3*T*5c7	c$	r2 ($T \to$ aTc) ; g5
0a3*T*5	c$	s7
0a3*T*5c7	$	r2 ($T \to$ aTc) ; g1
0*T*1	$	accept

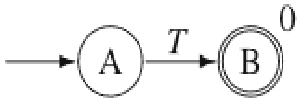
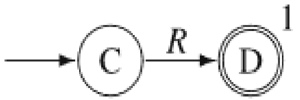
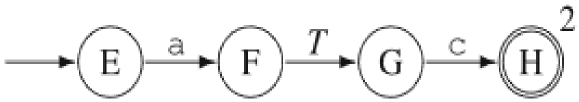
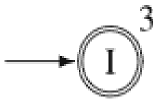
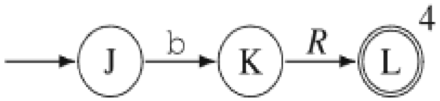
2.14 Constructing SLR Parse Tables

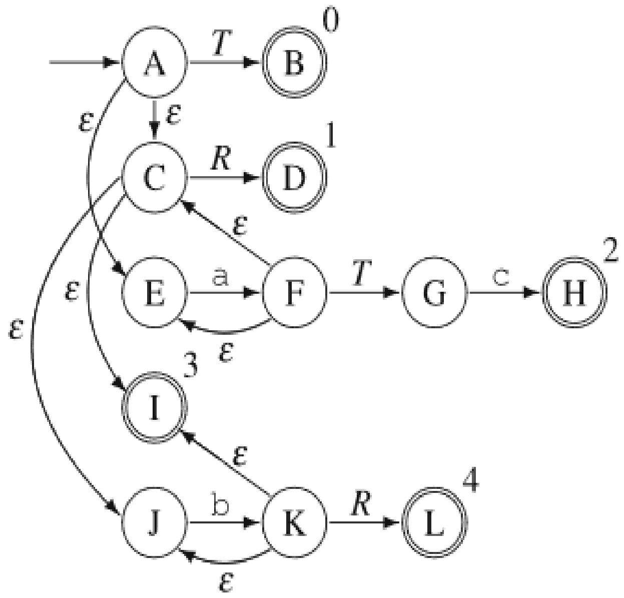
An SLR parse table has a DFA as its core. Constructing this DFA from the grammar is similar to constructing a DFA from a regular expression, as shown in this chapter:

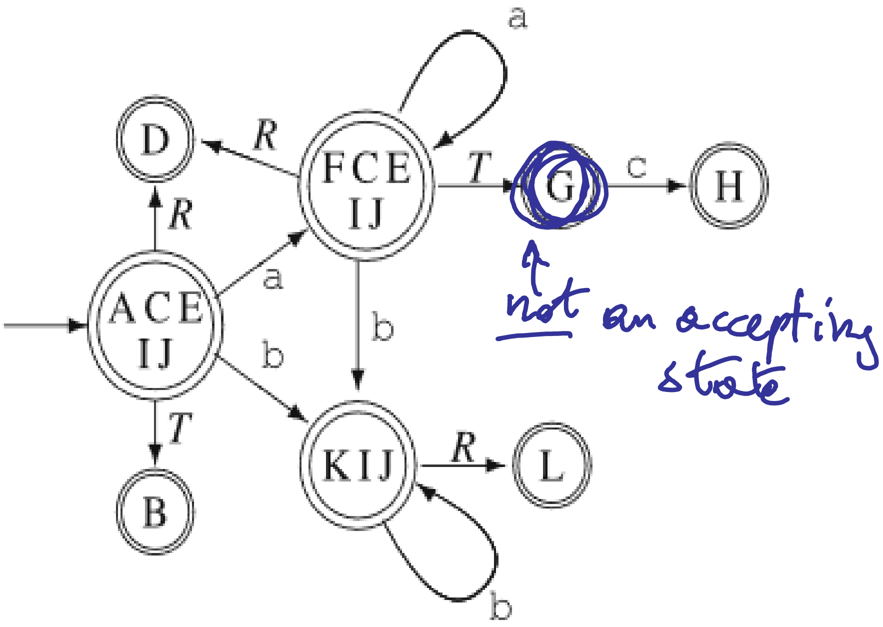
We first construct an NFA using techniques similar to those in Sect. 1.3 and then convert this into a DFA using the construction shown in Sect. 1.5.

First, we add "production 0":

$$\begin{array}{lcl} T & \to & R \\ T & \to & \mathtt{a}T\mathtt{c} \\ R & \to & \\ R & \to & \mathtt{b}R \end{array} \quad \Longrightarrow \quad \begin{array}{rlcl} 0: & T' & \to & T \\ 1: & T & \to & R \\ 2: & T & \to & \mathtt{a}T\mathtt{c} \\ 3: & R & \to & \\ 4: & R & \to & \mathtt{b}R \end{array}$$

Production	NFA
$T' \to T$	
$T \to R$	
$T \to \mathtt{a}T\mathtt{c}$	
$R \to$	
$R \to \mathtt{b}R$	

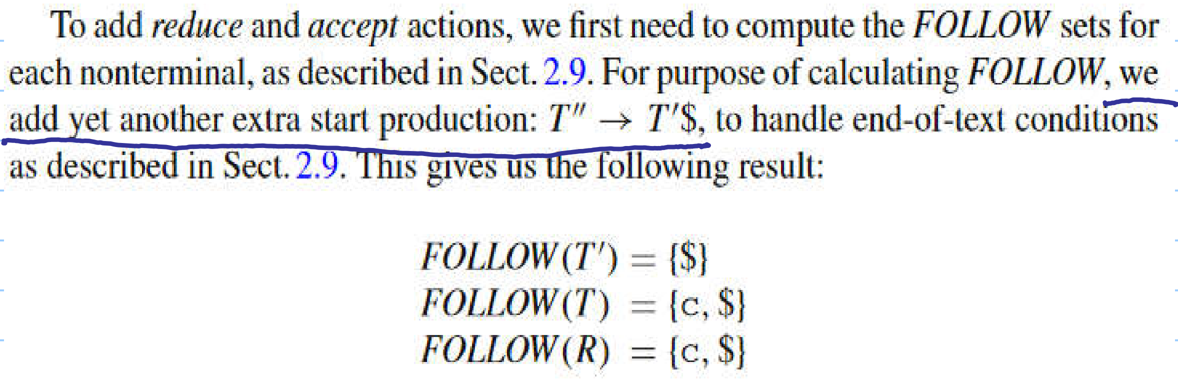
Production	Combined NFA
$T' \to T$	 A T B 0 ε ε C R D 1 ε E a F T G c H 2 ε ε ε I 3 ε J b K R L 4 ε
$T \to R$	
$T \to \mathtt{a}T\mathtt{c}$	
$R \to$	
$R \to \mathtt{b}R$	



not an accepting state

DFA state	NFA states	Transitions				
		a	b	c	*T*	*R*
0	A, C, E, I, J	s3	s4		g1	g2
1	B					
2	D					
3	F, C, E, I, J	s3	s4		g5	g2
4	K, I, J		s4			g6
5	G			s7		
6	L					
7	H					

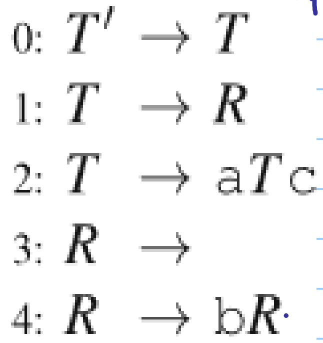
We need to add reduce and accept actions.



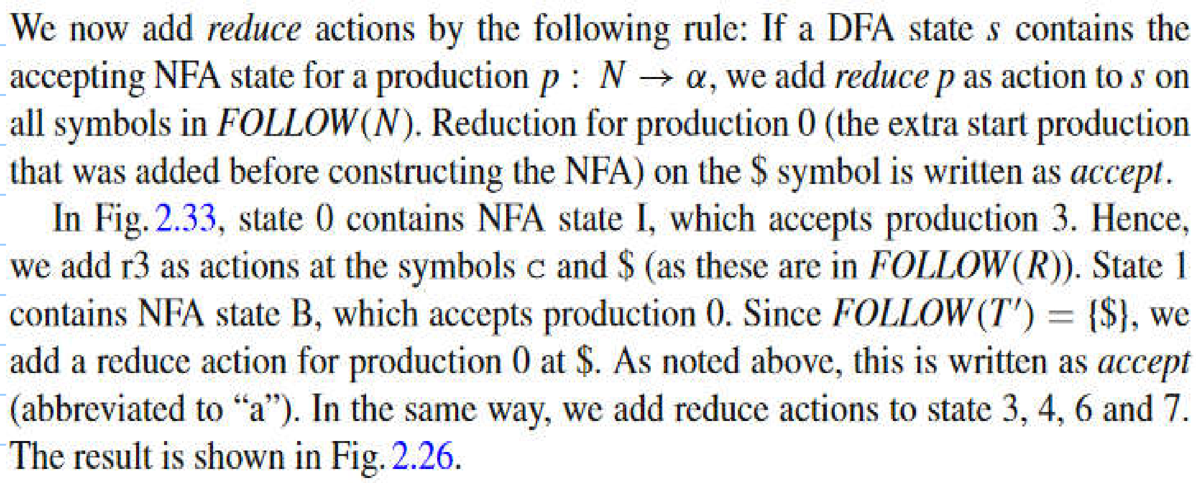
To add *reduce* and *accept* actions, we first need to compute the *FOLLOW* sets for each nonterminal, as described in Sect. 2.9. For purpose of calculating *FOLLOW*, we add yet another extra start production: $T'' \rightarrow T'\$$, to handle end-of-text conditions as described in Sect. 2.9. This gives us the following result:

$$\begin{aligned} FOLLOW(T') &= \{\$\} \\ FOLLOW(T) &= \{\mathtt{c}, \$\} \\ FOLLOW(R) &= \{\mathtt{c}, \$\} \end{aligned}$$

$T'' \rightarrow T'\$$

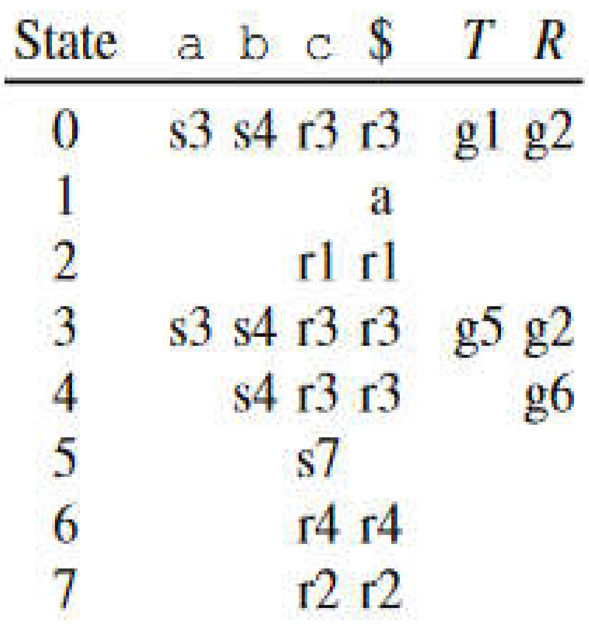


$$\begin{aligned} 0:&\ T' \rightarrow T \\ 1:&\ T \rightarrow R \\ 2:&\ T \rightarrow \mathtt{a}T\mathtt{c} \\ 3:&\ R \rightarrow \\ 4:&\ R \rightarrow \mathtt{b}R \end{aligned}$$



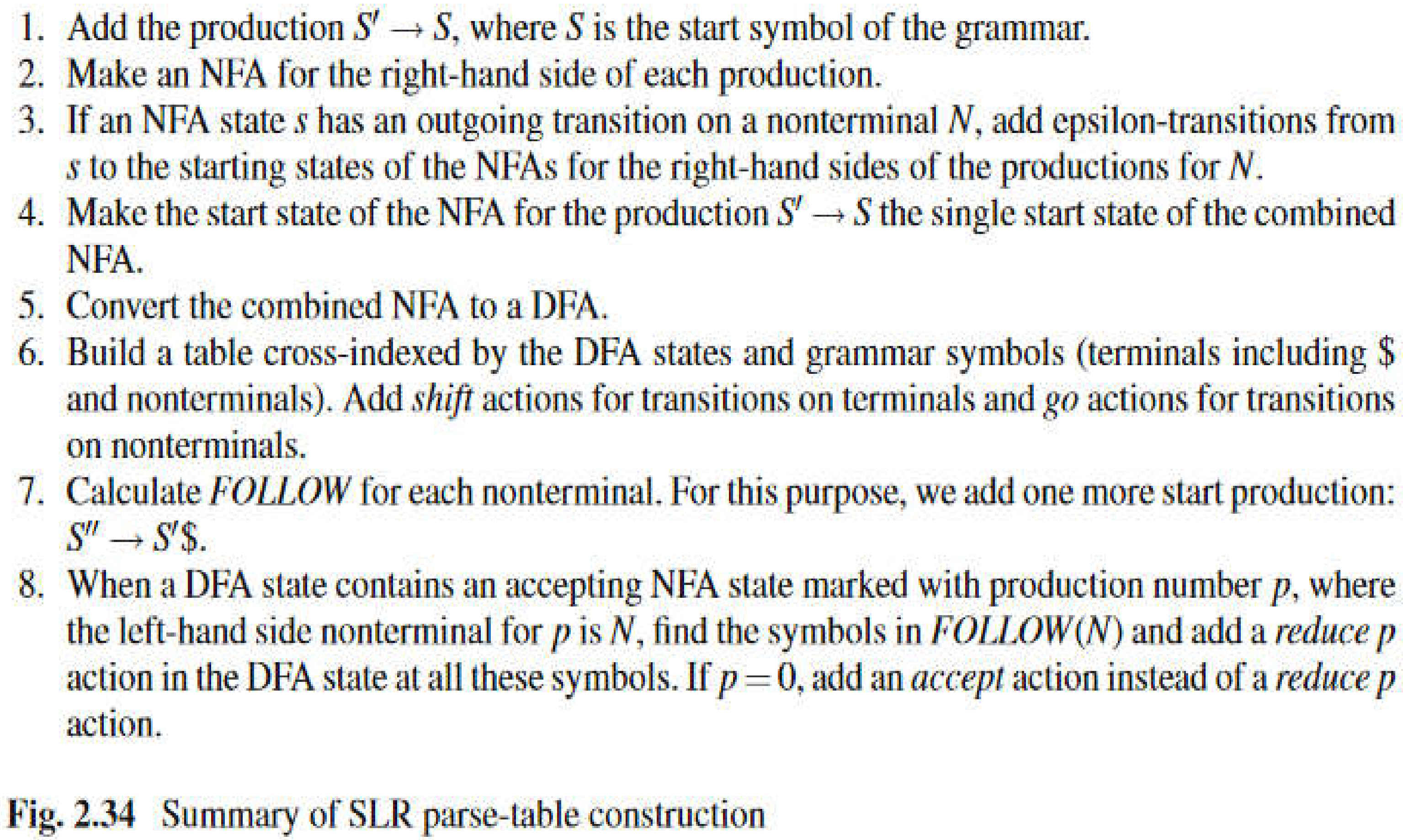
We now add *reduce* actions by the following rule: If a DFA state *s* contains the accepting NFA state for a production $p: N \rightarrow \alpha$, we add *reduce p* as action to *s* on all symbols in $FOLLOW(N)$. Reduction for production 0 (the extra start production that was added before constructing the NFA) on the $ symbol is written as *accept*.

In Fig. 2.33, state 0 contains NFA state I, which accepts production 3. Hence, we add r3 as actions at the symbols c and $ (as these are in $FOLLOW(R)$). State 1 contains NFA state B, which accepts production 0. Since $FOLLOW(T') = \{\$\}$, we add a reduce action for production 0 at $. As noted above, this is written as *accept* (abbreviated to "a"). In the same way, we add reduce actions to state 3, 4, 6 and 7. The result is shown in Fig. 2.26.



State	a	b	c	$	*T*	*R*
0	s3	s4	r3	r3	g1	g2
1				a		
2			r1	r1		
3	s3	s4	r3	r3	g5	g2
4		s4	r3	r3		g6
5			s7			
6			r4	r4		
7			r2	r2		

Figure 2.26



1. Add the production $S' \rightarrow S$, where S is the start symbol of the grammar.
2. Make an NFA for the right-hand side of each production.
3. If an NFA state s has an outgoing transition on a nonterminal N, add epsilon-transitions from s to the starting states of the NFAs for the right-hand sides of the productions for N.
4. Make the start state of the NFA for the production $S' \rightarrow S$ the single start state of the combined NFA.
5. Convert the combined NFA to a DFA.
6. Build a table cross-indexed by the DFA states and grammar symbols (terminals including \$ and nonterminals). Add *shift* actions for transitions on terminals and *go* actions for transitions on nonterminals.
7. Calculate *FOLLOW* for each nonterminal. For this purpose, we add one more start production: $S'' \rightarrow S'\$$.
8. When a DFA state contains an accepting NFA state marked with production number p, where the left-hand side nonterminal for p is N, find the symbols in $FOLLOW(N)$ and add a *reduce p* action in the DFA state at all these symbols. If $p = 0$, add an *accept* action instead of a *reduce p* action.

Fig. 2.34 Summary of SLR parse-table construction

2.14.1 Conflicts in SLR Parse-Tables

Shift-reduce and reduce-reduce conflicts are possible.

$$\begin{array}{l} A \rightarrow \alpha\, B\, \beta \\ A \rightarrow \alpha\, \gamma_1\, \delta \\ \\ B \rightarrow \gamma_1 \\ \quad\vdots \\ B \rightarrow \gamma_n \end{array}$$

and there is overlap between $\mathit{FIRST}(\delta)$ and $\mathit{FOLLOW}(B)$, then there will be a shift-reduce conflict after reading $\alpha\, \gamma_1$, as both reduction with $B \rightarrow \gamma_1$ and shifting on any symbol in $\mathit{FIRST}(\delta)$ is possible, which gives a conflict for all symbols in $\mathit{FIRST}(\delta) \cap \mathit{FOLLOW}(B)$. This conflict can be resolved by splitting the first production above into all the possible cases for B:

$$\begin{array}{l} A \rightarrow \alpha\, \gamma_1\, \beta \\ \quad\vdots \\ A \rightarrow \alpha\, \gamma_n\, \beta \\ A \rightarrow \alpha\, \gamma_1\, \delta \end{array}$$

2.15 Using Precedence Rules in LR Parse Tables

$$
\begin{aligned}
Exp &\rightarrow Exp + Exp\\
Exp &\rightarrow Exp - Exp\\
Exp &\rightarrow Exp * Exp\\
Exp &\rightarrow Exp / Exp\\
Exp &\rightarrow \mathbf{num}\\
Exp &\rightarrow (\ Exp\ )
\end{aligned}
$$

(1) A conflict between shifting on + and reducing by the production $Exp \rightarrow Exp + Exp$.

(2) A conflict between shifting on + and reducing by the production $Exp \rightarrow Exp * Exp$.

(3) A conflict between shifting on * and reducing by the production $Exp \rightarrow Exp + Exp$.

(4) A conflict between shifting on * and reducing by the production $Exp \rightarrow Exp * Exp$.

(1) This conflict arises from expressions like a+b+c. After having read a+b, the next input symbol is +. We can now either choose to reduce a+b, grouping around the first addition before the second, or shift on the plus, which will later lead to b+c being reduced, and hence grouping around the second addition before the first. Since the convention is that + is left-associative, we prefer the first of these options and, hence, eliminate the shift-action from the table and keep only the reduce-action.

(2) The offending expressions here have the form a*b+c. Since convention make multiplication bind stronger than addition, we, again, prefer reduction over shifting.

(3) In expressions of the form a+b*c, the convention, again, makes multiplication bind stronger, so we prefer a shift to avoid grouping around the + operator and, hence, eliminate the reduce-action from the table.

(4) This case is identical to case 1, where an operator that by convention is left-associative conflicts with itself. We, as in case 1, handle this by eliminating the shift.

In general, elimination of conflicts by operator precedence declarations can be summarised into the following rules:

(a) If the conflict is between two operators of different priority, eliminate the action with the lowest priority operator in favour of the action with the highest priority. In a reduce action, the operator associated with a reduce-action is an operator used in the production that is reduced. If several operators are used in the same production, the operator that is closest to the end of the production is used.[4]

(b) If the conflict is between operators of the same priority, the associativity (which must be the same, as noted in Sect. 2.3.1) of the operators is used: If the operators are left-associative, the shift-action is eliminated and the reduce-action retained. If the operators are right-associative, the reduce-action is eliminated and the shift-action retained. If the operators are non-associative, both actions are eliminated.

$a - b - c = (a - b) - c$ (left associative)

$1:2:3:[\,] = (1:(2:(3:[\,]))) = 1:(2:[3]) =$

$= 1:[2,3] = [1,2,3]$ (right associative)

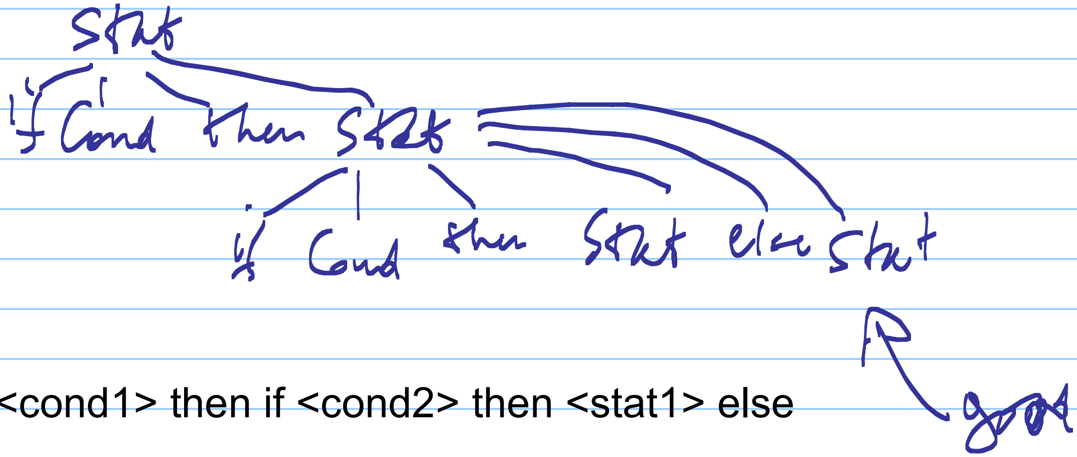
$a + b - c$

$(a+b)$

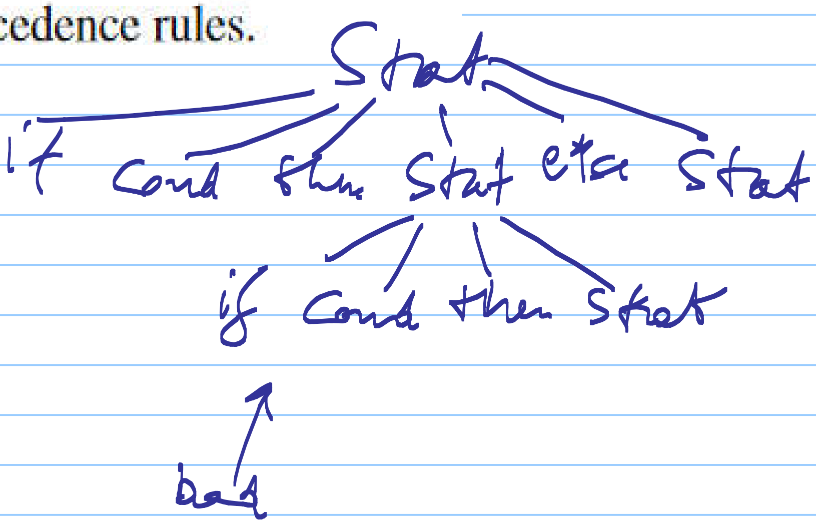
$(b - c)$

The dangling-else ambiguity (Sect. 2.4) can also be eliminated using precedence rules. If we have read `if` *Exp* `then` *Stat* and the next symbol is a `else`, we want to shift on `else`, so the `else` will be associated with the `then`. Giving `else` a higher precedence than `then` or giving them the same precedence and making them right-associative will ensure that a shift is made on `else` when we need it.

Not all conflicts should be eliminated by precedence rules. If you blindly add precedence rules until no conflicts are reported, you risk eliminating actions that are required to parse certain strings, so the parser will accept only a subset of the intended language. Normally, you should only use precedence declarations to specify operator hierarchies, unless you have analysed the parser actions carefully and found that there is no undesirable consequences of adding the precedence rules.

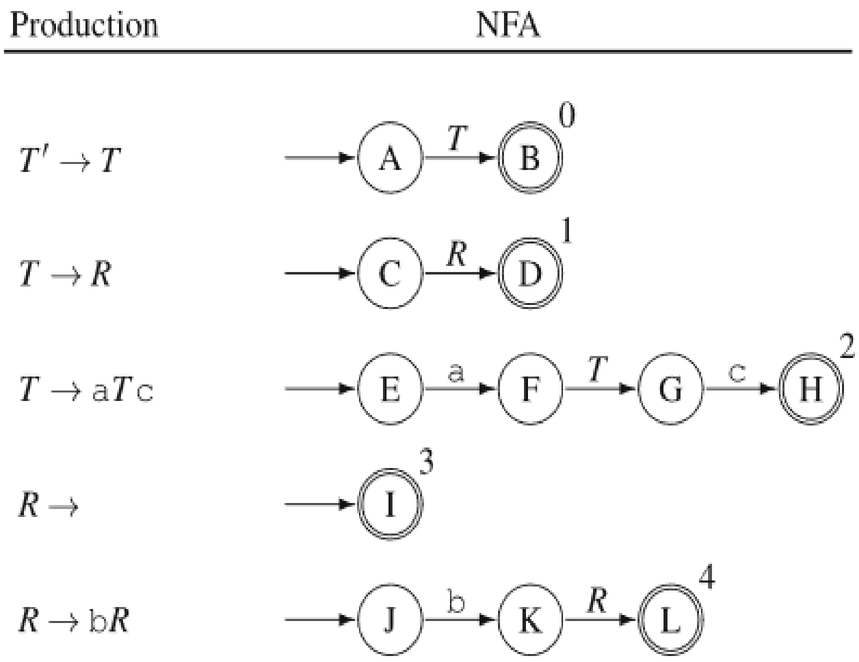


If <cond1> then if <cond2> then <stat1> else



SLR vs, LALR(1)

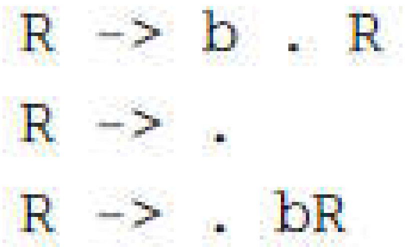
Textual representation of NFA states in parser generators (e.g., Alex, Lex)



0: $T' \to T$
1: $T \to R$
2: $T \to \mathtt{a}T\mathtt{c}$
3: $R \to$
4: $R \to \mathtt{b}R$

NFA-state	Textual representation
A	`T’ -> . T`
B	`T’ -> T .`
C	`T -> . R`
D	`T -> R .`
E	`T -> . aTc`
F	`T -> a . Tc`
G	`T -> aT . c`
H	`T -> aTc .`
I	`R -> .`
J	`R -> . bR`
K	`R -> b . R`
L	`R -> bR .`

DFA state	NFA states
0	A, C, E, I, J
1	B
2	D
3	F, C, E, I, J
4	K, I, J
5	G
6	L
7	H

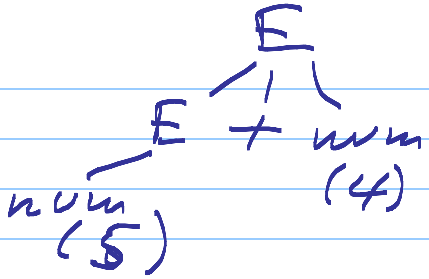


```
R -> b . R
R -> .
R -> . bR
```

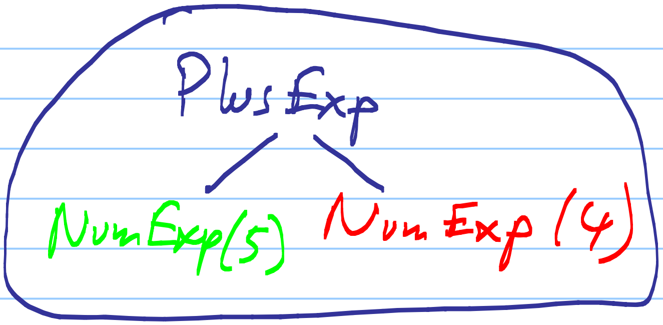
Textual representation of DFA state 4.

NFA-state	Textual representation
A	`T’ -> . T`
B	`T’ -> T .`
C	`T -> . R`
D	`T -> R .`
E	`T -> . aTc`
F	`T -> a . Tc`
G	`T -> aT . c`
H	`T -> aTc .`
I	`R -> .`
J	`R -> . bR`
K	`R -> b . R`
L	`R -> bR .`

Declarations and actions



5 + 4



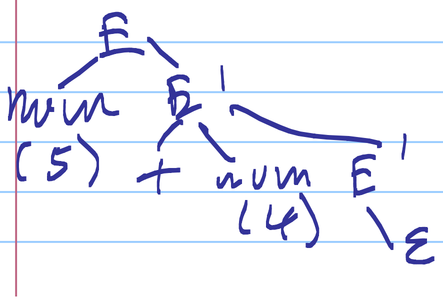
$$\begin{array}{llll} \overset{\$0}{E} & \rightarrow & \boxed{\overset{\$1}{E}} \overset{\$2}{+} \overset{\$3}{\textbf{num}} & \{ \texttt{PlusExp(}\underline{\texttt{\$1}}\texttt{,}\underline{\texttt{NumExp(\$3)}}\texttt{)} \} \\ E & \rightarrow & \textbf{num} & \{ \underline{\texttt{NumExp(\$1)}} \} \end{array}$$

Actions can be used to build abstract syntax trees, as above

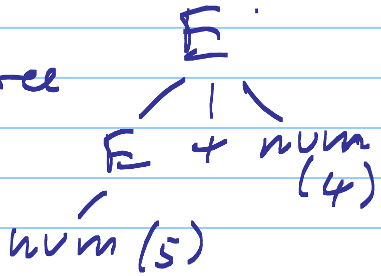
Actions can be used to reshape syntax trees into abstract ones.

$$\begin{array}{lll} E & \rightarrow & \textbf{num}\, E' \\ E' & \rightarrow & +\, \textbf{num}\, E' \\ E' & \rightarrow & \end{array}$$

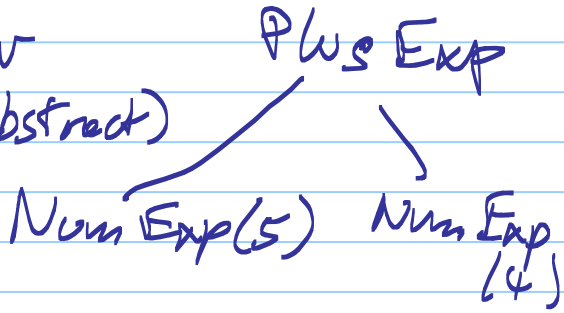
Grammar after left-recursion elimination



vs. the desired tree

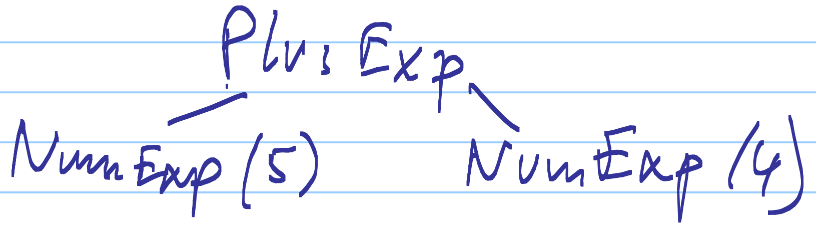


or (abstract)

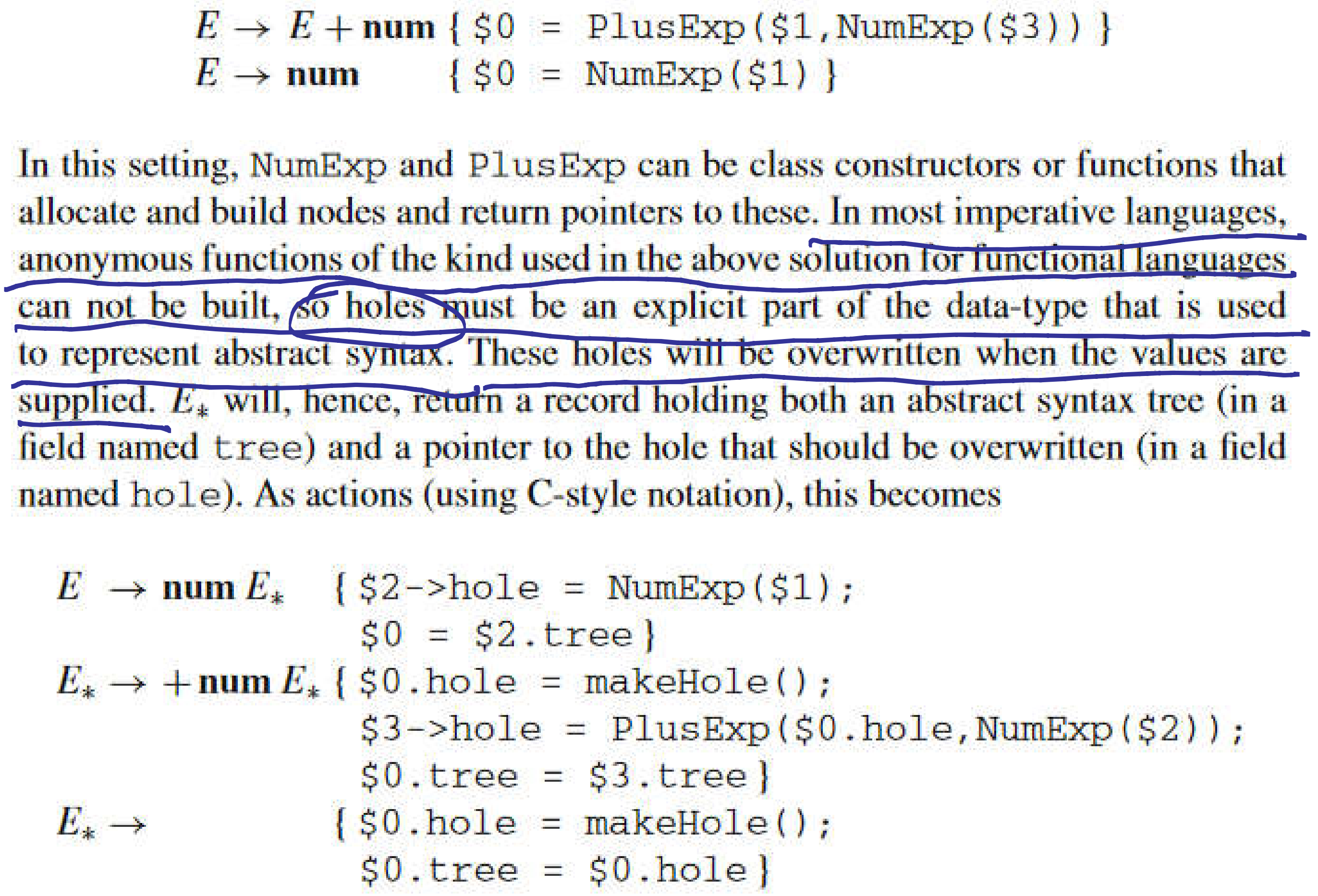


$$
\begin{array}{lcll}
E & \rightarrow & \overset{5}{\textbf{num}}\, \overset{\$2}{E'} & \{\ \$2(\boxed{\texttt{NumExp(\$1)}})\ \} \\
E' & \rightarrow & \overset{\$1}{+}\, \underset{4}{\overset{\$2}{\textbf{num}}}\, \overset{\$3}{E'} & \{\ \lambda\texttt{x.}\boxed{\texttt{\$3}}\texttt{(PlusExp(}\boxed{\texttt{x}}\texttt{,NumExp(\$2)))}\ \} \\
E' & \rightarrow & & \{\ \boxed{\lambda\texttt{x.x}}\ \}
\end{array}
$$

$$\lambda x . \underline{(\lambda x . x)}\ PlusExp(NumExp(5), NumExp(4))$$



- The second production for E' returns just a hole.
- In the first production for E', the $+$ and **num** terminals are used to produce a tree for a plus-expression (*i.e.*, a *PlusExp* node) with a hole in place of the first subtree. This tree is used to fill the hole in the tree returned by the recursive use of E', so the abstract syntax tree is essentially built outside-in. The result is a new tree with a hole.
- In the production for E, the hole in the tree returned by the E' nonterminal is filled by a *NumExp* node with the number that is the value of the **num** terminal.



```
E  → E + num { $0 = PlusExp($1,NumExp($3)) }
E  → num     { $0 = NumExp($1) }
```

In this setting, NumExp and PlusExp can be class constructors or functions that allocate and build nodes and return pointers to these. In most imperative languages, anonymous functions of the kind used in the above solution for functional languages can not be built, so holes must be an explicit part of the data-type that is used to represent abstract syntax. These holes will be overwritten when the values are supplied. E_* will, hence, return a record holding both an abstract syntax tree (in a field named tree) and a pointer to the hole that should be overwritten (in a field named hole). As actions (using C-style notation), this becomes

```
E   → num E*     { $2->hole = NumExp($1);
                   $0 = $2.tree }
E*  → + num E*   { $0.hole = makeHole();
                   $3->hole = PlusExp($0.hole,NumExp($2));
                   $0.tree = $3.tree }
E*  →            { $0.hole = makeHole();
                   $0.tree = $0.hole }
```

Java now allows anonymous functions