COMPILER CONSTRUCTION

Principles and Practice

Kenneth C. Louden

5. Bottom-Up Parsing

PART TWO

Contents

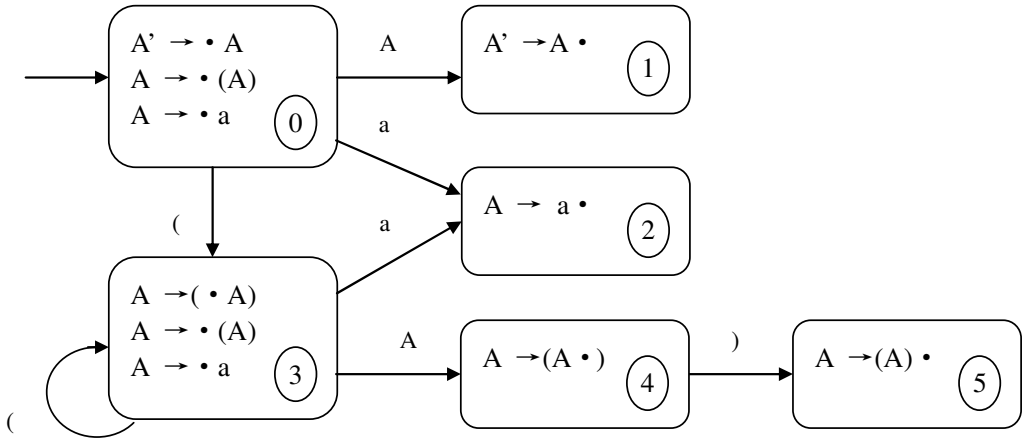
LR(0) Items of A Grammar

A' → A
A → (A)|a

This grammar has eight items:

A' → ·A
A' → A·
A → ·(A)
A → (·A)
A → (A·)
A → (A)·
A → ·a
A → a·

DFA and the LR(0) Parsing of The grammar: A → (A) | a



	Parsing stack	input	Action
1	$ 0	((a))$	shift
2	$ 0 (3	(a))$	shift
3	$ 0 (3 (3	a)$	shift
4	$ 0 (3 (3 a 2	))$	reduce A→a
5	$ 0 (3 (3 A 4	))$	shift
6	$ 0 (3 (3 A 4) 5	)$	reduce A→(A)
7	$ 0 (3 A 4	)$	shift
8	$ 0 (3 A 4) 5	$	reduce A→(A)
9	$ 0 A 1	$	accept

The LR(0) parsing table

State	Action	Rule	Input			Goto
			(	a	)	A
0	shift		3	2		1
1	reduce	A'→A				
2	reduce	A→a				
3	shift		3	2		4
4	shift				5	
5	reduce	A→(A)				

1. One column is reserved to indicate the *actions* for each state;
2. A further column is used to indicate the grammar choice for the reduction;
3. For each token, there is a column to represent the new state;
4. Transitions on non-terminals are listed in the Goto sections.

5.3 SLR(1) Parsing

SLR(1), called simple LR(1) parsing, uses the DFA of sets of LR(0) items as constructed in the previous section

SLR(1) increases the power of LR(0) parsing significant by using the next token in the input string

- **First, it consults the input token *before* a shift to make sure that an appropriate DFA transition exists**
- **Second, it uses the Follow set of a non-terminal to decide if a reduction should be performed**

5.3.1 The SLR(1) Parsing Algorithm

Definition of The SLR(1) parsing algorithm(1)

Let s be the current state,

actions are defined as follows: .

1.If state s contains any item of form $A \rightarrow \alpha \cdot X\beta$

where X is a terminal, and

X is the next token in the input string,

then to shift the current input token onto the stack,

and push the new state containing the item

$A \rightarrow \alpha X \cdot \beta$

2. If state s contains the complete item $A \rightarrow \gamma \cdot$,

and the next token in input string is in Follow(A)

then to reduce by the rule $A \rightarrow \gamma$

Definition of The SLR(1) parsing algorithm(2)

2. (Continue)

   A reduction by the rule S' →S, is equivalent to acceptance;

   – This will happen only if the next input token is $.

   In all other cases, Remove the string γ and a corresponding states from the parsing stack

   – Correspondingly, back up in the DFA to the state from which the construction of γ began.

   – This state must contain an item of the form B → α·Aβ.

   Push A onto the stack, and the state containing the item

   B → αA·β.

Definition of The SLR(1) parsing algorithm(3)

3. If the next input token is such that neither of the above two cases applies,
   - an error is declared

A grammar is an SLR(l) grammar if the application of the above SLR(1) parsing rules results in *no ambiguity*

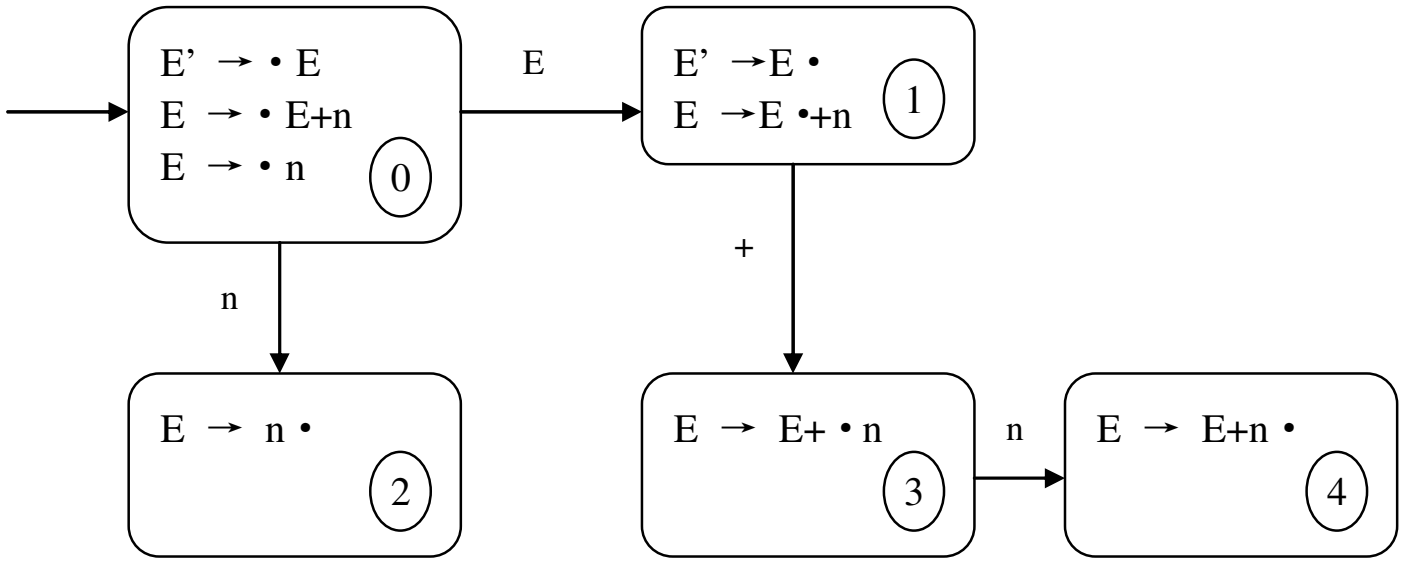
A grammar is SLR(1) if and only if, for any state s, the following two conditions are satisfied:

- For any item $A \rightarrow \alpha \cdot X\beta$ in s with X a terminal,
  There is no complete item $B \rightarrow \gamma.$ in s with X in Follow(B).
- For any two complete items $A \rightarrow \alpha \cdot$ and $B \rightarrow \beta \cdot$ in s,
  Follow(A) ∩ Follow(B) is empty.

A violation of the first of these conditions represents a **shift-reduce conflict**

A violation of the second of these conditions represents a **reduce-reduce conflict**

- **Example 5. 10 Consider the grammar:**
  - **E'→E**
  - **E→E + n | n**
- **This grammar is not LR(0), but it is SLR(l).**
  - **The Follow sets for the nonterminals:**
  - **Follow(E') ={$ } and Follow(E) = { $, +} .**



The SLR(1) parsing table for above Grammar

State	Input			Goto
	n	+	$	E
0	s2			1
1		s3	accept	
2		r($E \rightarrow n$)	r($E \rightarrow n$)	
3	s4			
4		r($E \rightarrow E+n$)	r($E \rightarrow E+n$)	

- A shift is indicated by the letter s in the entry, and a reduction by the letter r
  - In state 1 on input +, a shift is indicated, together with a transition to state 3
  - In state 2 on input +, a reduction by production E → n is indicated
  - The action "accept" in state 1 on input $ instead of r (E' →E)

The parsing process for n+n+n

	Parsing stack	Input	Action
1	*$0*	*n + n + n $*	*shift 2*
2	*$0 n 2*	*+n + n $*	*reduce E →n*
3	*$0E1*	*+n + n $*	*shift3*
4	*$0E1+3*	*n + n $*	*shift4*
5	*$0E1+3n4*	*+ n $*	*reduce E → E+n*
6	*$0E1*	*+ n $*	*shift3*
7	*$0E1+3*	*n $*	*shift4*
8	*$0E1+3n4*	*$*	*reduce E → E+n*
9	*$0E1*	*$*	*accept*

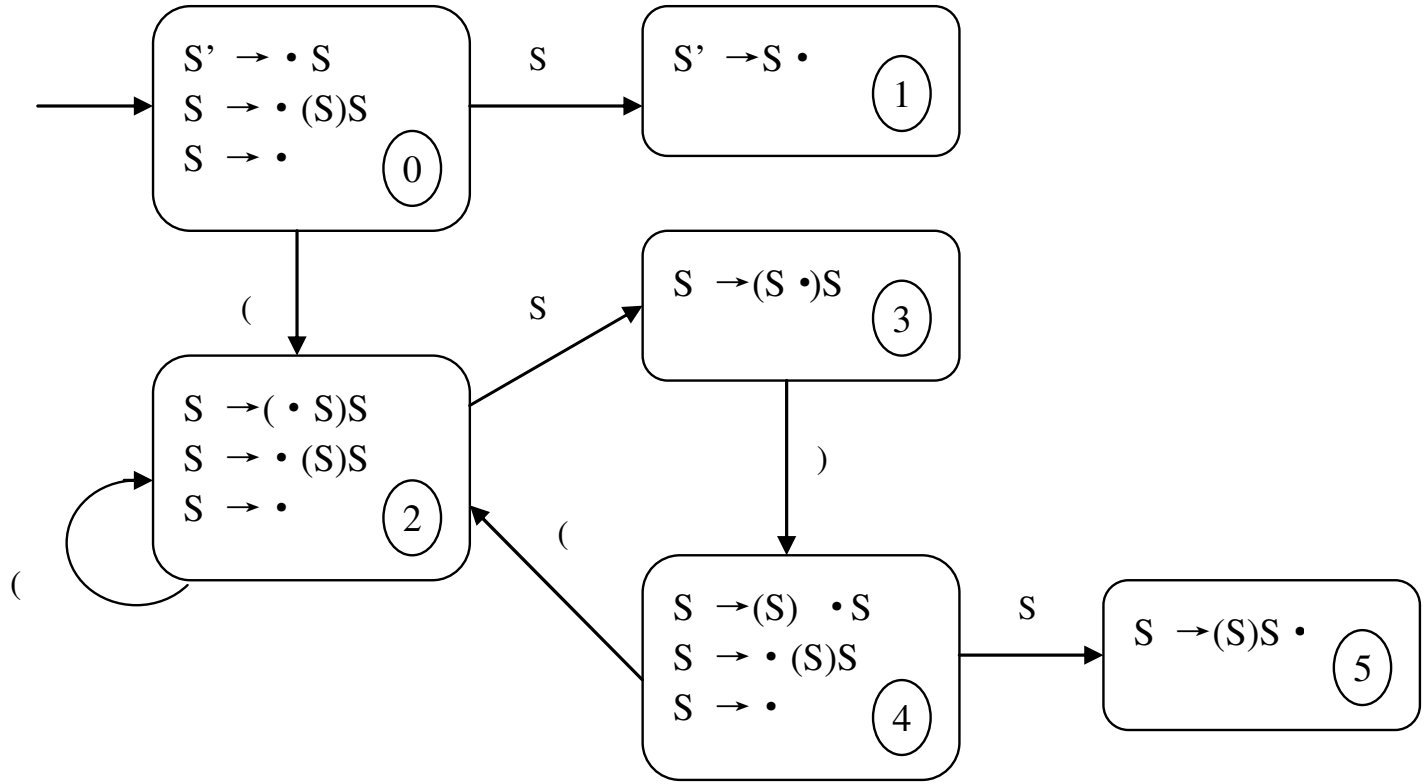
Example 5. 11 Consider the grammar of balanced parentheses

S' → S

S → (S)S|ε

Follow sets computation yields:

– Follow(S') = {$} and Follow(S)={$,)}.



- **The SLR(l) parsing table is as follows, where**
  - **The non-LR(0) states 0, 2, and 4 have both shifts and**
  - **reductions by the ε-production $S \rightarrow \varepsilon$**

State	Input			Goto
	(	)	$	S
0	s2	r($S \rightarrow \varepsilon$)	r($S \rightarrow \varepsilon$)	1
1			accept	
2	s2	r($S \rightarrow \varepsilon$)	r($S \rightarrow \varepsilon$)	3
3		s4		
4	s2	r($S \rightarrow \varepsilon$)	r($S \rightarrow \varepsilon$)	5
5		r($S \rightarrow (S)S$)	r($S \rightarrow (S)S$)	

- **The steps to parse the string ( ) ( )**
  - **The stack continues to grow until the final reductions**
  - **This is characteristic of bottom-up parsers in the presence of right-recursive rules such as S → (S)S**
  - **Thus, right recursion can cause stack overflow, and so is to be avoided if possible**

	Parsing stack	Input	Action
1	*$0*	*( ) ( )$*	*shift 2*
2	*$0(2*	*) ( )$*	*reduce* S → ε
3	*$0(2S3*	*) ( )$*	*shift4*
4	*$0(2S3)4*	*( )$*	*shift2*
5	*$0(2S3)4(2*	*)$*	*reduce* S → ε
6	*$0(2S3)4(2S3*	*) $*	*shift4*
7	*$0(2S3)4(2S3)4*	*$*	*reduce* S → ε
8	*$0(2S3)4(2S3)4S5*	*$*	*reduce* S → (S)S
9	*$0(2S3)4S5*	*$*	*reduce* S → (S)S
10	*$0S1*	*$*	*accept*

5.3.2 Disambiguating Rules for Parsing Conflicts

- **Parsing conflicts in SLR(l) parsing can be of two kinds:**
  - **shift-reduce conflicts and reduce-reduce conflicts.**
- **In the case of shift-reduce conflicts, there is a natural disambiguating rule,**
  - **Which is to always *prefer the shift over the reduction***
  - **Most shift-reduce parsers therefore automatically resolve shift-reduce conflicts by preferring the shift over the reduction.**
- **The case of reduce-reduce conflicts is more difficult;**
  - ***Such conflicts often (but not always) indicate an error in the design of the grammar.***

- **Example 5. 12 Consider the grammar of simplified if statements.**

  statement →if-stmt| other

  if-stmt → if (exp) statement

  |if (exp) statement else statement

  exp→0|1

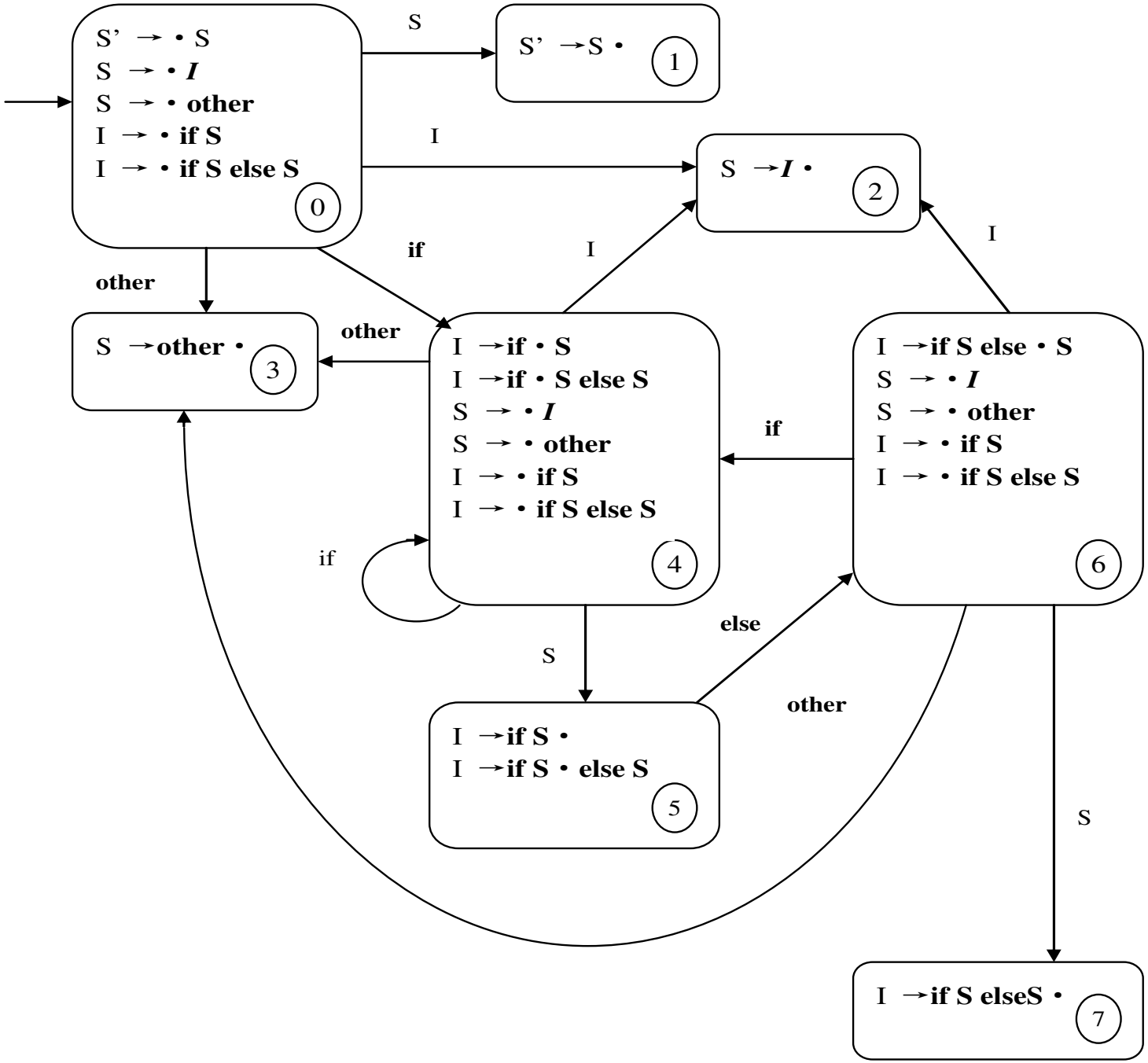
- **Write the grammar as follows**

  S→I| other

  I→if S | if S else S

- **The DFA of sets of items is shown in next page. To construct the SLR(1) parsing actions need the Follow sets for S and I.**

  Follow(S)=Follow(I)={$, *else*}


S' → • S
S → • I
S → • other
I → • if S
I → • if S else S
0
S
S' → S •
1
I
S → I •
2
if
I
other
other
S → other •
3
I → if • S
I → if • S else S
S → • I
S → • other
I → • if S
I → • if S else S
4
if
I → if S else • S
S → • I
S → • other
I → • if S
I → • if S else S
6
if
I
if
S
else
other
I → if S •
I → if S • else S
5
S
I → if S elseS •
7

5.3.3 Limits of SLR(1) Parsing Power

- Example 5. 13 Consider the following grammar rules for statements.

  stmt → *call-stmt* | *assign-stmt*

  call-stmt → ***identifier***

  assign-stmt → *var* ***:=****exp*

  var →*var [exp]* |***identifier***

  exp → *var* | ***number***

- Simplify this situation to the following grammar without changing the basic situation:

  S → ***id*** | *V* ***:=*** *E*

  V→ ***id***

  E → *V* | ***n***

- To show how this grammar results in parsing conflict in SLR(l) parsing, consider the start state of the DFA of sets of items:

  $S' \rightarrow \cdot S$

  $S \rightarrow \cdot \mathbf{id}$

  $S \rightarrow \cdot V \mathbf{:=} E$

  $V \rightarrow \cdot \mathbf{id}$

- This state has a shift transition on id to the state

  $S \rightarrow \mathbf{id} \cdot$

  $V \rightarrow \mathbf{id} \cdot$

- Now, Follow(S)={$} and Follow(V)={:=, $}

  := because of the rule $S \rightarrow V \mathbf{:=} E$, and $ because an E can be a V

- Thus, the SLR(l) parsing algorithm calls for a reduction in this state by both the rule $S \rightarrow \mathbf{id}$ and the rule $V \rightarrow \mathbf{id}$ under input symbol $.
  - (this is a reduce-reduce conflict)

5.3.4 SLR(k) Grammars

- As with other parsing algorithms, the SLR(1) parsing algorithm can be extended to SLR(k) parsing where parsing actions are based on k >=1 symbols of lookahead.
- Using the sets First*k* and Follow*k* as defined in the previous chapter, an SLR(k) parser uses the following two rules:

  1. If state s contains an item of the form $A \rightarrow \alpha \cdot X\beta$(X a token), and $Xw \in$ First*k* $(X\beta)$ are the next k tokens in the input string,

  then the action is to shift the current input token onto the stack, and the new state to be pushed on the stack is the state containing the item $A \rightarrow \alpha X \cdot \beta$

  2. If state s contains the complete item $A \rightarrow \alpha \cdot$, and $Xw \in$ Follow*k*(A) are the next *k* tokens in the input string,

  then the action is to reduce by the rule $A \rightarrow \alpha$.

Back

5.4 General LR(1) and LALR(1) Parsing

5.4.1 Finite Automata of LR(1) Items

- The SLR(1) method:
  - Applies lookaheads after the construction of the DFA of LR(0) items
  - **The construction of DFA ignores lookaheads**

- The general LR(1) method:
  - **Using a new DFA with the lookaheads built into its construction**

    The DFA items are an extension of LR(0) items

    LR(1) items include a single lookahead token in each item.
  - A pair consisting of an LR(0) item and a lookahead token.

    LR(1) items using square brackets as $[A \rightarrow \alpha \cdot \beta, a]$

    where $A \rightarrow \alpha \cdot \beta$ is an LR(0) item and a is a lookahead token

- The definition the transitions between LR(l) items
  - Similar to the LR(0) transitions except keeping track of lookaheads
  - As with LR(0) items including ε-transitions, to build a DFA's states are sets of items that are ε-closures
  - **The difference between the LR(0)and LR(1) automata comes in the definition of the ε-transitions**
- We give the definition of the easier case (the non-ε-transitions) first, which are essentially identical to those of the LR(0) case

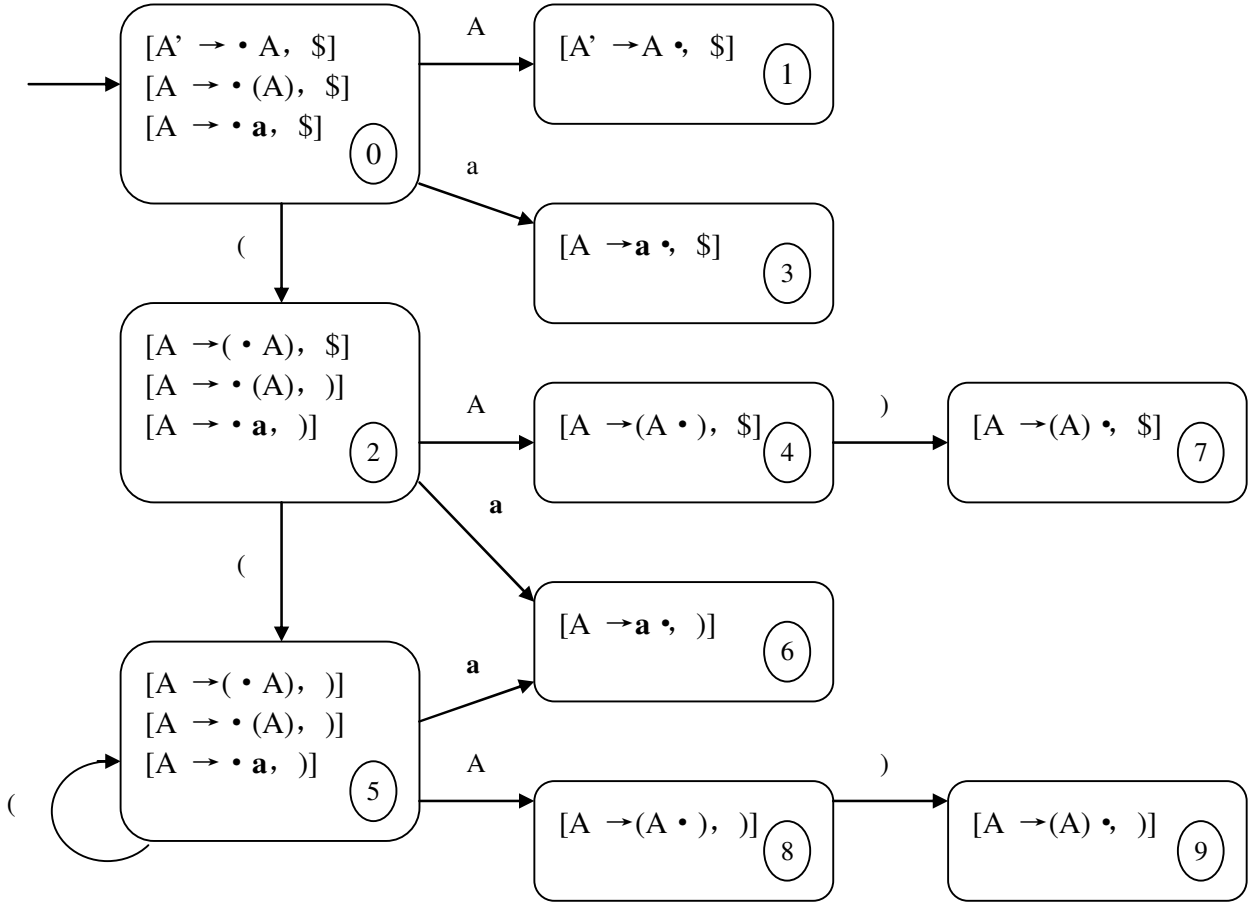
Definition

- Definition of LR(1) transitions (part 1).
  - Given an LR(1) item [A→α·Xγ,a], where X is any symbol (terminal or nonterminal),
  - There is a transition on X to the item [A→ αX·γ,a]
- Definition of LR(1) transitions (part 2).
  - Given an LR(1) item [A→α·Bγ,a], where B is a nonterminal,
  - There are ε-transitions to items [B→·β,b] for every production B →β and *for every token b* in First(γa).

- **Example 5.14** Consider the grammar

  A→(A) | a

- The DFA of sets of LR(1) items



5.4.2 The LR(1) Parsing Algorithm

- Need to complete the discussion of general LR(1) parsing by restating the parsing algorithm based on the new DFA construction
- Only need to restatement of the SLR(1) parsing algorithm, except that it **uses the lookahead tokens in the LR(1) items instead of the Follow set**

The General LR(1) parsing algorithm(1)

Let s be the current state (a the top of the parsing stack),

Then actions are defined as follows:

1. If state s contains any LR(1) item of the form [A→α·Xβ,a], where X is a terminal, and X is the next token in the input string, then the action is to shift the current input token onto the stack,

   and the new state to be pushed on the Stack is the state containing the LR(1) item [A→αX·β,a].

2. If state s contains the complete LR(1) item [A→α·,a] , and the next token in the input string is a.

   then the action is to reduce by the rule A→α.

   - A reduction by the rule S'→S, where S is the start state, is equivalent to acceptance.
   - (This will happen only if the next input token is $.)

The General LR(1) parsing algorithm(2)

In the other cases, the new state is computed as follows.

- Remove the string αand all of its corresponding states from the parsing stack;
- back up in the DFA to the state from which the construction of a began.
- By construction, this state must contain an LR(l) item of the form [B→α·Aβ,b].
- Push A onto the stack, and push the state containing the item [B→αA·β,b].

3. If the next input token is such that neither of the above two cases applies,

- an error is declared.

- **A grammar is an LR(1) grammar if the application of the above general LR(l) parsing rules results in no ambiguity**

- A grammar is LR(I) if and only if, for any state s, the following two conditions are satisfied.

  l. For any item [A→α·Xβ,a] in s with X a terminal, there is no item in s of the form [B→γ·,X] (otherwise there is a shift-reduce conflict).

  2. There are no two items in s of the form [A→α·,a] and [B→β·,a] (otherwise, there is a reduce-reduce conflict).

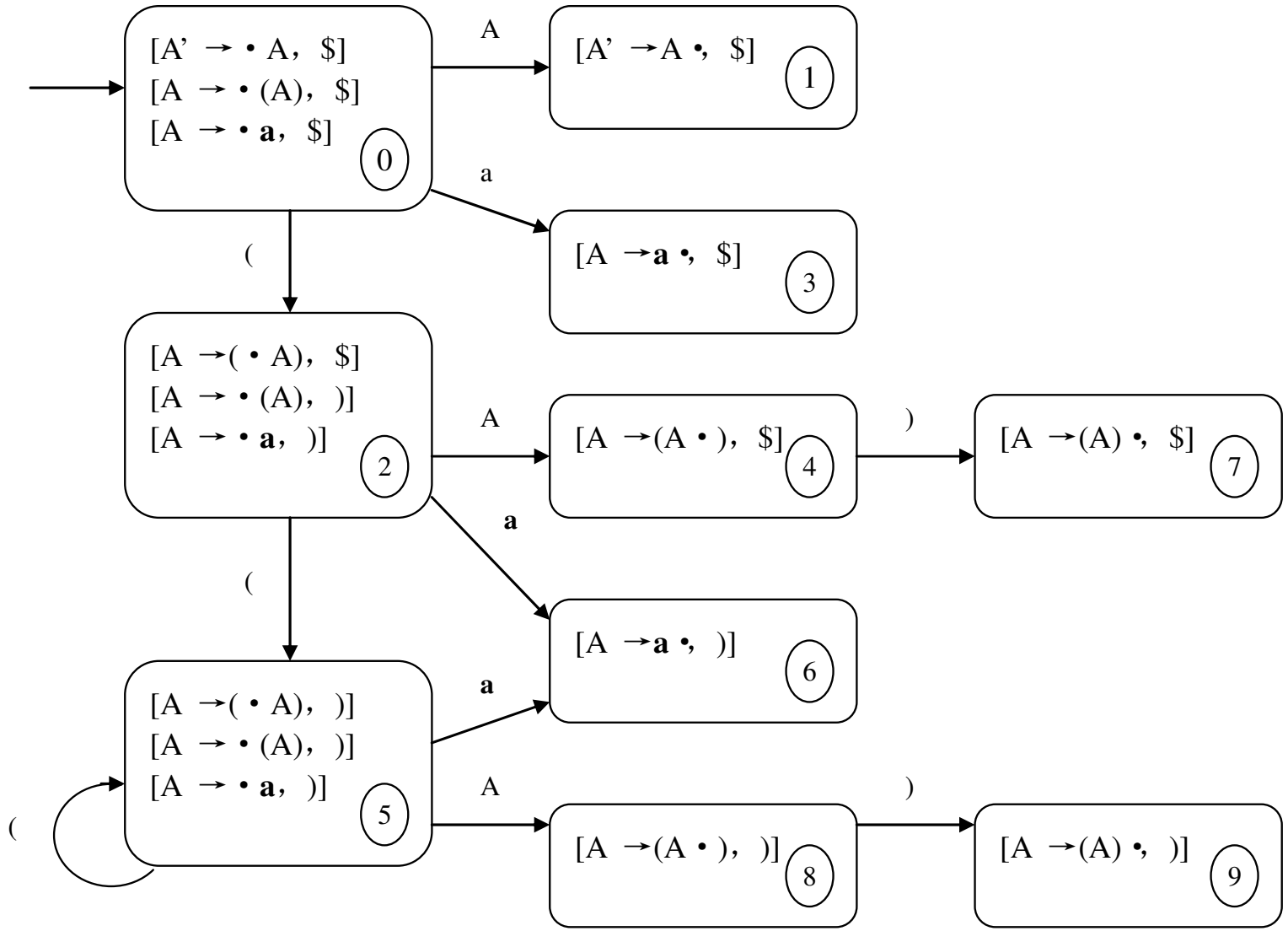
The Grammar:

(l) A→(A)

(2) A→**a**

State	Input				Goto
	(	a	)	$	A
0	s2	s3			1
1				accept	
2	s5	s6			4
3				r2	
4			s7		
5	s5	s6			8
6			r2		
7				r1	
8			s9		
9			r1		

The DFA of LR(1) item

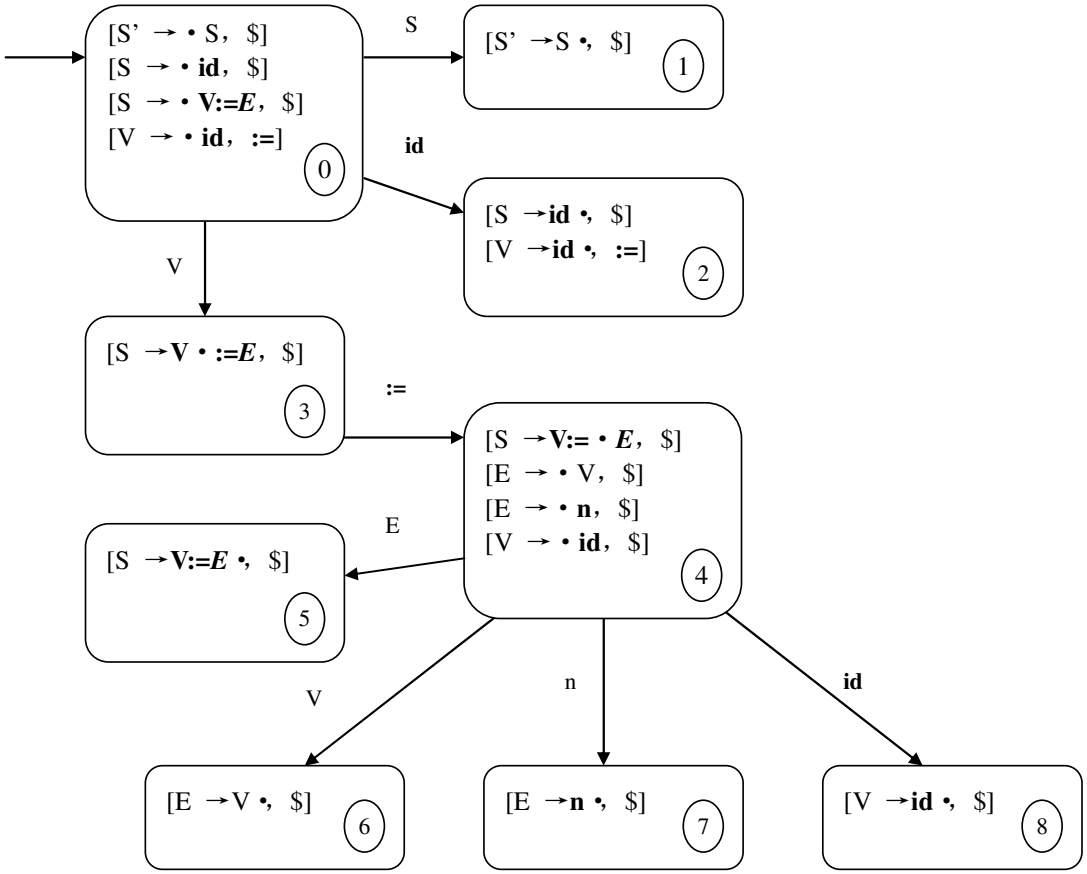


- **Example 5.16** The grammar of Example 5. 13 in simplified form:

  $S \rightarrow \mathbf{id} \mid V \mathbf{:=} E$

  $V \rightarrow \mathbf{id}$

  $E \rightarrow V \mid \mathbf{n}$



5.4.3 LALR(1) Parsing

- In the DFA of sets of LR(1) items, many different states have the same set of first components in their items (the LR(0) items), and different second components (the lookahead symbols)

- The LALR(1) parsing algorithm:
  - Identify all such states and combine their lookaheads;
  - Then, we have a DFA identical to the DFA of LR(0) items, except the each state consists of items with sets of lookaheads.
- In the case of complete items these lookahead sets are often smaller than the corresponding Follow sets.

- LALR(l) parsing retains some of the benefit of LR(l) parsing over SLR(l) parsing, while preserving the smaller size of the DFA of LR(0) items
- Formally, the core of a state of the DFA of LR(l) items is the set of LR(0) items consisting of the first components of all LR(1) items in the state

The following two facts form the basis for the LALR(l) parsing construction:

(1) FIRST PRINCIPLE OF LALR(1) PARSING

- The core of a state of the DFA of LR(l) items is a state of the DFA of LR(0) items

(2) SECOND PRINCIPLE OF LALR(1) PARSING

- Given two states s1 and s2 of the DFA of LR(l) items that have the same core
- suppose there is a transition on the symbol X from s1 to a state t1
- Then there is also a transition on X from state s2 to a state t2, and the states t1 and t2 have the same core

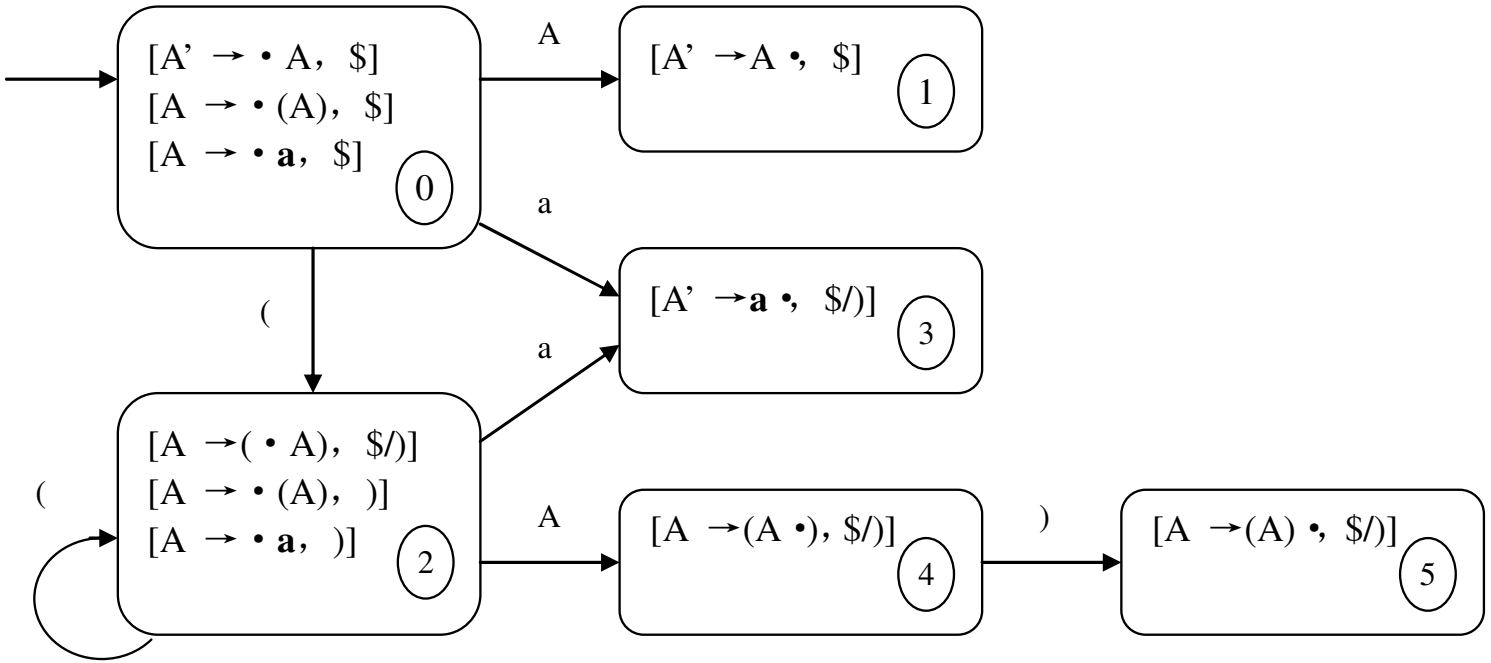
- Constructing the DFA of LALR(l) items:
  - Constructed from the DFA of LR(l) items by identifying all states that have the same core
  - And forming the union of the lookahead symbols for each LR(0) item

- Each LALR(l) item in this DFA will have an LR(0) item as its first component and a set of lookahead tokens as its second component.

- **Example 5.17** Consider the grammar of Example 5.14.

  (l) A→(A)

  (2) A→**a**



- **The algorithm for LALR(l) parsing** using the condensed DFA of LALR(l) items is identical to the general LR(1) parsing algorithm described in the previous section

- A grammar is LALR(l) grammar if no parsing conflicts arise in the LALR(l) parsing algorithm

- It is possible for the LALR(l) construction to create parsing conflicts that do not exist in general LR(1) parsing, but this rarely happens in practice

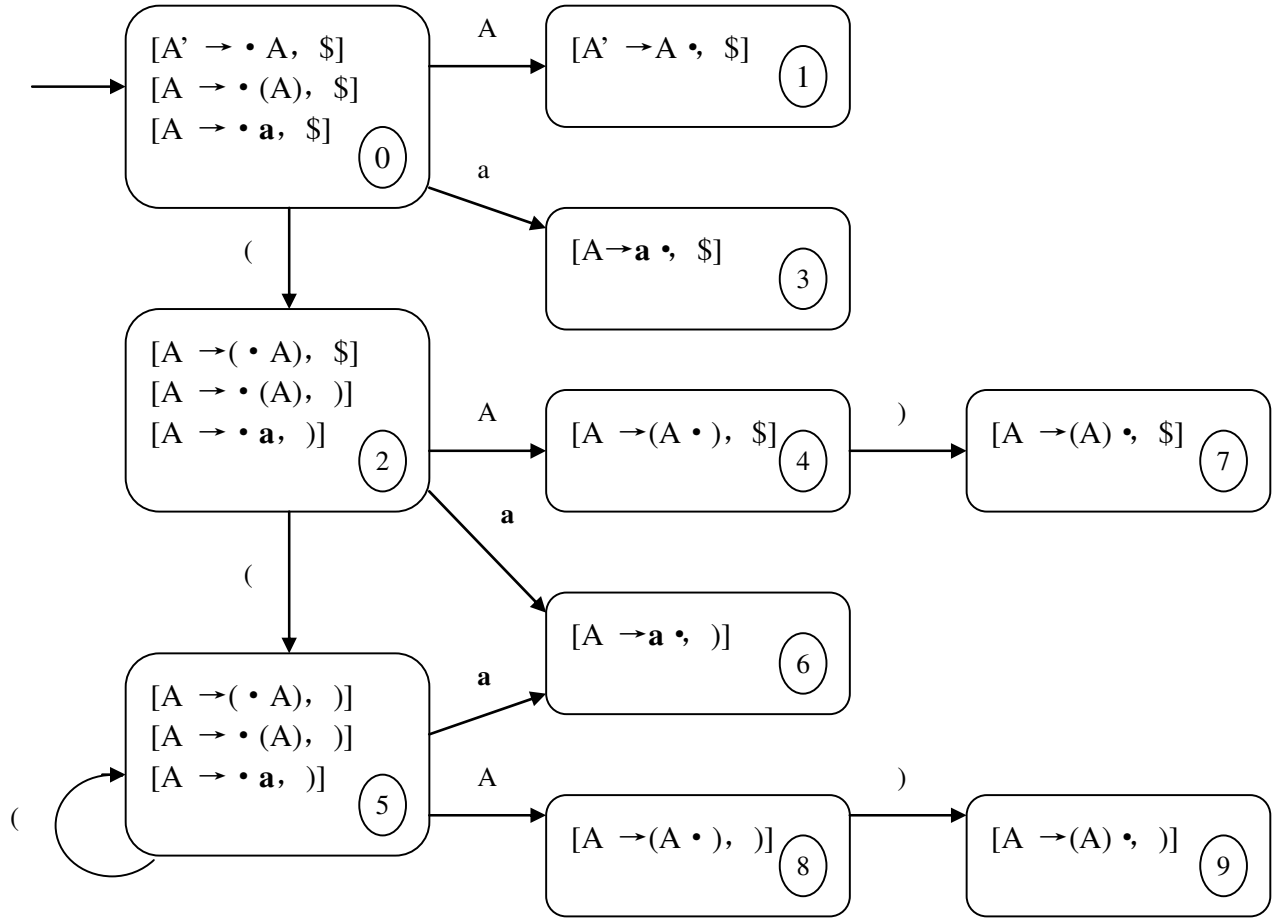
- Indeed, if a grammar is LR(l), then the LALR(l) parsing table cannot have any shift-reduce conflicts, there may be reduce-reduce conflicts.

- If a grammar is SLR(l), then it certainly is LALR(l)
- LALR(1) parsers often do as well as general LR(I) parsers in removing typical conflicts that occur in SLR(l) parsing

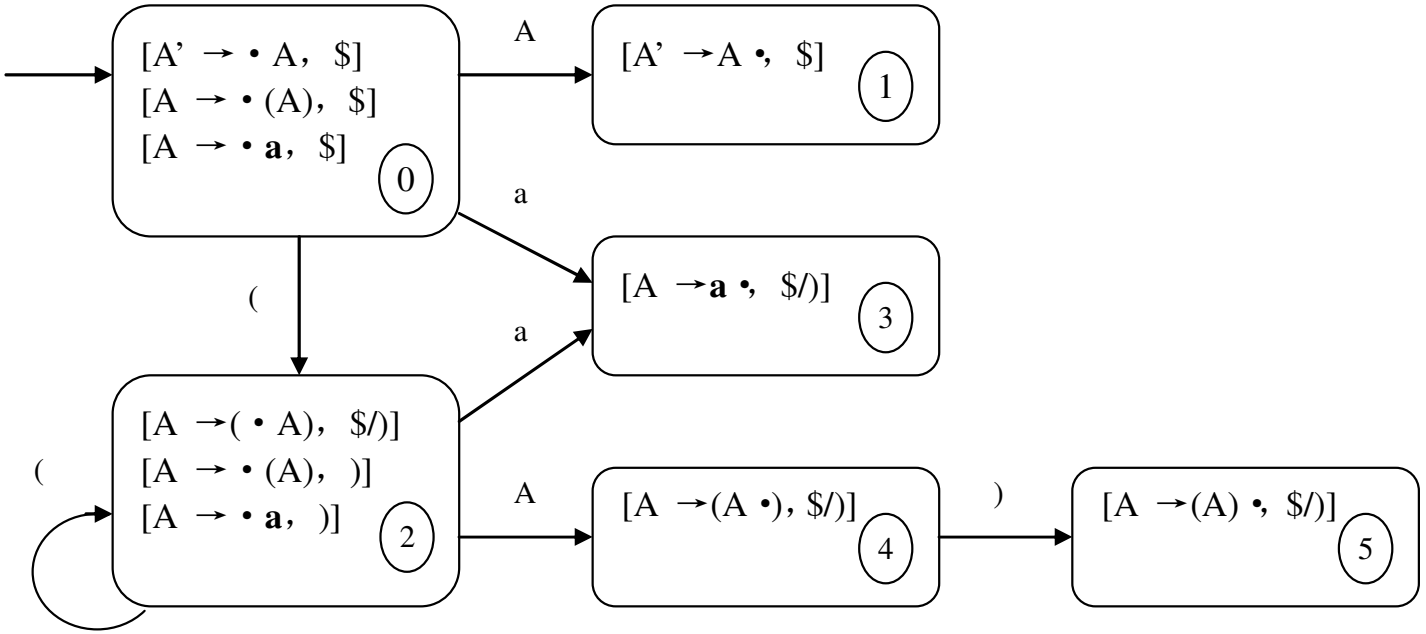
- If the grammar is already LALR(l), the only consequence of using LALR(l) parsing over general LR parsing is that, in the presence of errors, some spurious reductions may be made before error is declared
- For example:

  Given the erroneous input string **a)**

The DFA of LR(1)



The DFA of LALR(1)



Note:

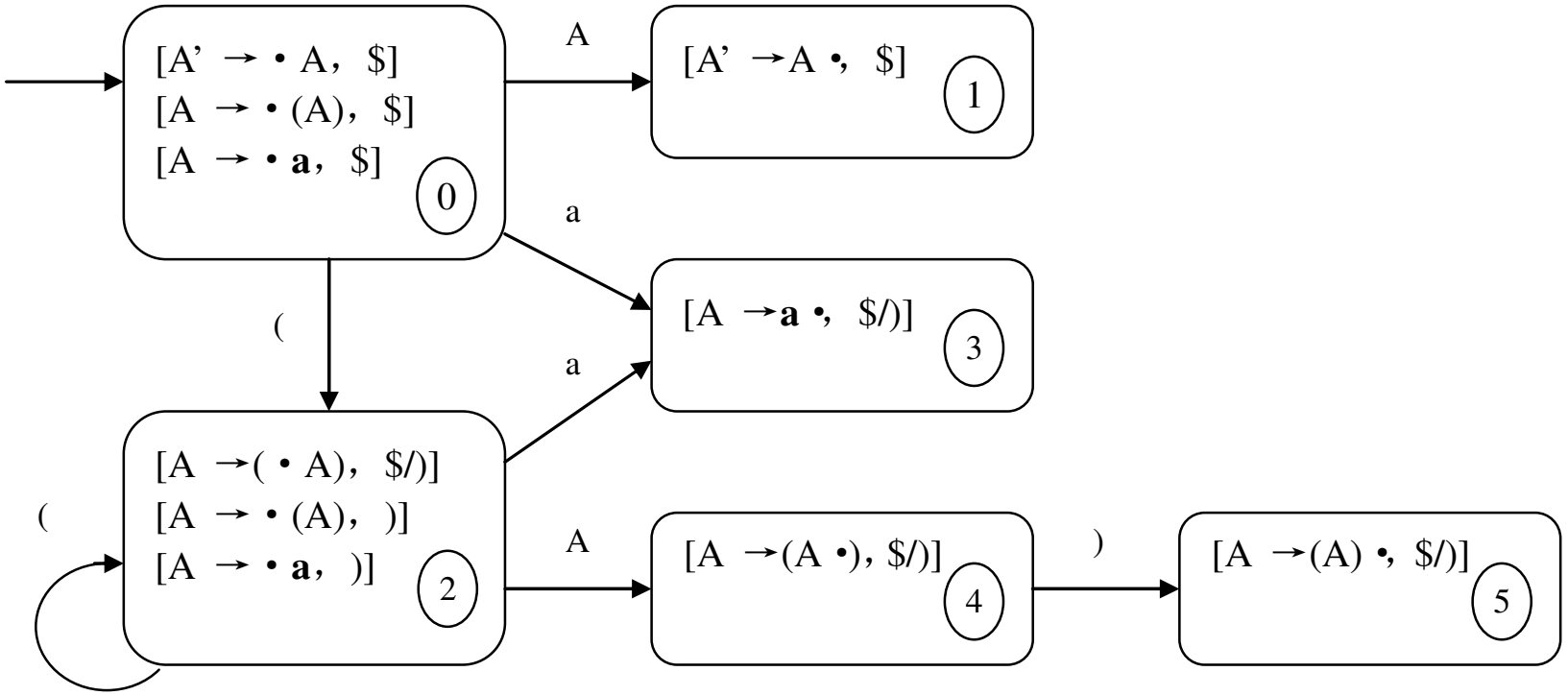
An LALR(l) parser will perform the reduction A→a , before declaring error;

A general LR(l) parser will declare error immediately after a shift of the token a.

- Combining LR(1) states to form the DFA of LALR(1) items solves the problem of large parsing tables, but it still requires the entire DFA of LR(1) items to be computed

- It is possible to compute the DFA of LALR(1) items directly from the DFA of LR(0) items through **a process of propagating lookaheads**
- We will not describe this process formally, it is instructive to see how this can be done easily
- Consider the LALR(1) DFA in following page

Computing the DFA of LALR(l) items directly from the DFA of LR(0) items



- Begin constructing lookaheads by adding the end marker $ to the lookahead of the augmentation item A'→·A in state 0
- By the rules of ε—closure, the $ propagates to the two closure items (the A on the right-hand side of the kernel item A'→·A is followed by the empty string)
- By following the three transitions from state 0, the $ propagates to the kernel items of states 1,3 and 2
- Continuing with state 2, the closure items get the lookahead). again by spontaneous generation (because A on the right-hand side of the kernel item A→(·A) comes before a right parenthesis)

- The transition on **a** to state 3 causes the) to be propagated to the lookahead of the item in that state
- The transition on (from state 2 to itself causes the) to be propagated to the lookahead of the kernel item (this is why the kernel item has both $ and) in its lookahead set)
- The lookahead set $/) propagates to state 4 and then to state 5
- Through this process, we have obtained the DFA of LALR(l) items of Figure 59 directly from the DFA of LR\0) items

Back

5.5 Yacc: LALR(1) PARSING GENERATOR

- A parser generator is a program taking a specification of the syntax of a language and producing a parser procedure for the language

- A parser generator is called compiler-compiler

- One widely used parser generator incorporating the LALR(1) parsing algorithm is called YACC

5.5.1 Yacc Basics

- Yacc takes a *specification file*
  - (usually with a **.y** suffix);
- Yacc produces *an output file consisting of C source code for the parser*
  - (usually in a file called **y.tab.c** or **ytab.c** or, more recently <filename>**.tab.c**, where <filename>**.y** is the input file).
- A Yacc specification file has the basic format.
  - **{definitions}**
  - **%%**
  - **{rules}**
  - **%%**
  - **{auxiliary routines}**
- **There are three sections separated by lines containing double percent signs**

- **The example:** A calculator for simple integer arithmetic expressions with the grammar

  exp → *exp addop term* | *termx*

  addop → ***+*** | ***-***

  term → *term mulop factor* | *factor*

  mulop → *******

  factor → (*exp*) | ***number***

```
%{
#include <stdio.h>
#include <ctype.h>
%}
%token NUMBER

%%
command :exp { printf ("%d\n",$1);}
; /*allows printing of the result */
exp:  exp '+' term {$$ = $1 + $3;}
        | exp '-' term {$$ = $1 - $3;}
        | term {$$ = $1;}
;
```

```
term:  term '*' factor {$$ = $1* $3;}
        | factor {$$ = $1;}
;
factor :NUMBER{$$ = $1;}
        | '('exp')' {$$=$2;}
;
 %%
main ( )
{ return yyparse( );
}
```

```
int yylex(void)
{ int c;
   while( ( c = getchar ( ) )== ‘ ’ );
/*eliminates blanks */
   if ( isdigit(c) ) {
      unget (c,stidin) ;
      scanf (“%d”,&yylval ) ;
     return (NUMBER ) ;
    }
   if (c== ‘\n’) return 0;
   /* makes the parse stop */
  return ( c ) ;
}
```

```
int yyerror (char * s)
{ fprintf (stderr, “%s\n”,s ) ;
   return 0;
}/* allows for printing of an error message */
```

- The definitions section (can be empty)
  - Information about the tokens, data types, grammar rules;
  - Any C code that must go directly into the output file at its beginning.
- The rules section
  - Grammar rules in a modified BNF form;
  - Actions in C code executed whenever the associated grammar rule is recognized
    - (i.e.. used in a reduction. according to the LALR(1) parsing algorithm)
  - The metasymbol conventions:
    - The vertical bar is used for alternatives;
    - The arrow symbol →is replaced in Yacc by a colon;
    - A semicolon must end each grammar rule.

- The auxiliary routines section (also can be empty):
  - Procedure and function declarations

- A minimal Yacc specification file would consist only of %% followed by grammar rules and actions.

- Notes:
- Two typical #include directives are set off from other Yacc declarations in the definitions section by the surrounding delimiters *%{ and %}*.
- Yacc has two ways of recognizing token.
  - Single-character inside single quotes as itself;
  - Symbolic tokens as a numeric value that does not conflict with any character value.

- Notes:
- *%start command* in the definition section will indicate that *command* be start symbol, otherwise the rule listed first in the rule section indicates the start symbol.
- The variable yylval is defined internally by Yacc; yyparse is the name of the parsing procedure; yylex is the name of Lex scanner generator.

5.5.2 Yacc Options

(1) –d option

- It will produce the header file usually named y.tab.h or ytab.h.
- Yacc needs access to many auxiliary procedures in addition to **yylax** and **yyerror**; They are often placed into external files rather than directly in the Yacc specification file.
- Making Yacc-specific definitions available to other files.
  - *For example, command in the file calc.y*
- ***yacc –d calc.y***

(1)–d option

Which will produce (in addition to the y.tab.c file) the file y.tab.h (or similar name), whose contents vary, but typically include such things as the following:

```
#ifndef YYSTYPE
#define YYSTYPE int
#endif
#define NUMBER 258
extern YYSTYPE yylval;
```

(2) –v option

- This option produces yet another file, with the name **y. output** (or similar name).
  - This file contains a textual description of the LALR(l)parsing table that is used by the parser.
- As an example, consider the skeletal version of the Yacc specification above.
  - **yacc -v calc .y**

(2) –v option

```
%taken NUMBER
%%
command :exp
;
exp:  exp ‘+’ term
| exp ‘-‘ term
| term
;
term:  term ‘*’ factor
| factor
;
factor :NUMBER
| ‘(‘exp’)’
;
```

A typical y.output file generated for the Yacc specification of figure 5.10 using the verbose option

```
state 0
        $accept :     _command $end
        NUMBER        shift 5
        (             shift 6
         •            error

        command       goto 1
        exp           goto 2
        term          goto 3
        factor        goto 4
state 1
        $accept :     command_ $end
        $end          accept
         •            error
state 2
        command :     exp_      (1)
        exp   :       exp_ + term
        exp:          exp_ - term

        +             shift 7
          -           shift 8
           •          reduce 4
```

state 3

	exp:	**term_ (4)**
	term:	**term_*factor**
	*****	**shift 9**
	•	**reduce**

state 4

	term:	**factor_ (6)**
	•	**reduce 6**

state 5

	factor:	**NUMBER_ (7)**
	•	**reduce 7**

state 6

	factor:	**(_exp)**
	NUMBER	**shift 5**
	(	**shift 6**

```
        •           error

        exp         goto 10
        term        goto 3
        factor      goto 4
state 7
        exp:        exp +_term

        NUMBER      shift 5
        (           shift 6
        •           error

        term        goto 11
        factor      goto 4
state 8
        exp:        exp -_term

        NUMBER      shift 5
        (           shift 6
        •           error

        term        goto 12
        factor      goto 4
```

state 9

temr:	term *_ factor
NUMBER	shift 5
(	shift 6
•	error
factor	goto 13

state 10

exp :	exp_ + term
exp:	exp_ - term
factor:	(exp_)
+	shift 7
-	shift 8
)	shift 14
•	error

state 11

exp :	exp + term_ (2)
temr:	term_ * factor
*	shift 9
•	reduce 2

state 12

exp : exp - term_ (3)

temr: term_ * factor

*** shift 9**

• reduce 3

state 13

temr: term * factor_ (5)

• reduce 5

state 14

factor: (exp)_ (8)

• reduce 8

The parsing table appears as follow:

State	Input							Goto			
	NUMBER	**(**	**+**	**-**	*****	**)**	**$**	*commad*	*exp*	*term*	*factor*
0	S5	s6						1	2	3	4
1							accept				
2	r1	r1	s7	s8	r1	r1	r1				
3	r4	r4	r4	r4	s9	r4	r4				
4	r6	r6	r6	r6	r6	r6	r6				
5	r7	r7	r7	r7	r7	r7	r7				
6	s5	s6							10	3	4
7	s5	s6								11	4
8	s5	s6									13
9	s5	s6									
10			s7	s8		s14					
11	r2	r2	r2	r2	s9	r2	r2				
12	r3	r3	r3	r3	s9	r3	r3				
13	r5	r5	r5	r5	r5	r5	r5				
14	r8	r8	r8	r8	r8	r8	r8				

5.5.3 Parsing Conflicts and Disambiguating Rule

5.5.4 Tracing the Execution of a Yacc Parsing

5.5.5 Arbitrary Value Types in Yacc

5.5.6 Embedded Action in Yacc

Back

5.6 GENERATION OF A TINY PARSER USING Yacc

5.7 ERROR RECOVERY IN BOTTOM-UP PARSERS

5.7.1 Detecting Errors in Bottom-up Parsing

- A bottom-up parser will detect an error when a blank (or error) entry is detected in the parsing table
- **Goal:**
  - Errors should be detected as soon as possible;
  - Error messages can be more meaningful and specific.
  - A parsing table should have as many blank entries as possible.
- **This goal conflicts with an equally important one: reducing the size of the parsing table.**

- **An additional feature of bottom-up parsing is that the power of the particular algorithm used can affect the ability of the parser to detect errors early**
  - An LR(l) parser detects errors earlier than an LALR(l) or SLR(1) parser;
  - These latter can detect errors earlier than an LR(0) parser.

Given the Grammar : **A→ (A)| a** The LR(1) parsing table

State	Input				Goto
	(	a	)	$	S
0	s2	s3			1
1				accept	
2	s5	s6			4
3				r2	
4			s7		
5	s5	s6			8
6			r2		
7				r1	
8			s9		
9			r1		

Given the **incorrect input string (a $,:**
The LR(l) parsing **will shift (and a onto the stack and move to state 6. In state 6 there is no entry under $, so an error will be reported**.

Given the Grammar : **A→ (A)| a** The LR(0) parsing table

State	Action	Rule	Input			Goto
			(	a	)	A
0	shift		3	2		1
1	reduce	A'→A				
2	reduce	A→a				
3	shift		3	2		4
4	shift				5	
5	reduce	A→(A)				

Given the **incorrect input string (a $,:**

The LR(0) algorithm (and the SLR(1) algorithm as well) **will reduce by A→a before discovering the lack of a balancing right parenthesis.**

5.7.2 Panic Mode Error Recovery

- As in top-down parsing, it is possible to achieve **reasonably good error recovery** in bottom-up parsers by judiciously **removing symbol from either the parsing stack or the input or both**.
- Three possible alternative actions that might be contemplated:

  1. Pop a state **from the stack**;
  2. Successively pop tokens **from the input** until a token is seen for which we can restart the parser;
  3. Push a **new state onto the stack**.

Method for choosing which action to take:

1. **Pop states** from the parsing stack until a state with nonempty Goto entries
2. If there is **a legal action** on the current input token from one of the Goto states, **push that state** onto the stack and restart the parse
   - If there are several such states, **prefer a shift to a reduce**
   - The reduce actions, **prefer one whose associated nonterminal is least genera**l
3. If there is no legal action on the current input token from one of the Goto states, **advance the input** until there is a legal action or the end of the input is reached

- The above rules have the effect of forcing the recognition of a construct when the error occurred, and restarting the parse immediately thereafter
- Error recovery that uses these or similar rules could be called **panic mode error recovery**
  - It is similar to the top-down panic mode described in Section 4.5.
- **Unfortunately, these rules can result in an infinite loop**

- Since step 2 pushes new states onto the stack. In that case, there are several possible solutions.
  - **One is to insist on a shift action from a Goto state in step 2.** This may be too restrictive, however.
  - Another solution is, if **the next legal move is a reduction**, to **set a flag** that causes the parser to keep track of the sequence of states during the following reductions,
- If the same state recurs, **to pop stack states until the original state** is removed at which the error occurred, and **begin again with step 1.**
- If, at any time, **a shift action occurs, the parser resets the flag** and continues with a normal parse

Example 5.19 Consider the simple arithmetic expression grammar with following parsing table

State	**Input**							**Goto**			
	NUMBER	**(**	**+**	**-**	*****	**)**	**$**	*commad*	*exp*	*term*	*factor*
0	S5	s6						1	2	3	4
1							accept				
2	r1	r1	s7	s8	r1	R1	r1				
3	r4	r4	r4	r4	s9	R4	r4				
4	r6	r6	r6	r6	r6	R6	r6				
5	r7	r7	r7	r7	r7	R7	r7				
6	s5	s6							10	3	4
7	s5	s6								11	4
8	s5	s6								12	4
9	s5	s6									13
10			s7	s8		s14					
11	r2	r2	r2	r2	s9	R2	r2				
12	r3	r3	r3	r3	s9	R3	r3				
13	r5	r5	r5	r5	r5	R5	r5				
14	r8	r8	r8	r8	r8	R8	r8				

The erroneous input (2+*), the parse proceeds normally until the * is seen.

Panic mode would cause the following actions to take place on the parsing stack:

Parsing stack	Input	Action
...	...	...
$0 (6E10+7	*) $	*error:*
	$	*push T, goto 11*
$0 (6E10+7T11	*) $	*shift 9*
*$0 (6E10+7T11*9*	) $	*error:*
	) $	*push F, goto 13*
*$0 (6E10+7T11*9F13*	$	*reduce T->T*F*
...	...	...

- At the first error, the parser is in state 7, which has legal Goto states 11 and 4. Since **state11 has a shift on the next input token ***, that Goto is preferred, and the token is shifted

- At the point the parser is in state 9, with a right parenthesis on the input. This is again an error. In state 9 there is a single Goto entry (to state 13), and state 13 does have a legal action on) (albeit a reduction).
- The parse then proceeds normally to its conclusion

5.7.3 Error Recovery in Yacc

5.7.4 Error Recovery in TINY

End of Part Two

THANKS