Bottom-Up Parsing

CIS*4650 (Winter 2020)

Bottom-Up Parsing

- Basic idea:
  - Use of stack to store terminals and non-terminals
  - Store input tokens until entire RHS is reached
  - Identify production based on tokens on stack and/or next token in input
  - Reduce RHS to LHS
  - If entire input reduces to the start-symbol, the input is valid or accepted

e.g.,

```
S -> a A B e
A -> A b c | b
B -> d
```

Reductions to S:

abbcde
aAbcde
aAde
aABe
S

Bottom-Up Parsing

- Basic actions: depending on stack contents and/or lookahead, the parser will either:
  - SHIFT: push input token onto stack, or
  - REDUCE: choose a production; pop RHS symbols from stack; and push LHS symbol onto stack
- Discovers productions of a rightmost derivation in reverse order

Bottom-Up Parsing

- Also use stack for terminals, non-terminals, and possible state information:

e.g.: S′ -> S
S -> (S) S | ε

Steps	Parsing stack	Input	Action
1	$	( ) $	shift
2	$ (	) $	reduce S -> ε
3	$ (S	) $	shift
4	$ (S)	$	reduce S -> ε
5	$ (S) S	$	reduce S -> (S) S
6	$ S	$	reduce S′ -> S
7	$ S′	$	accept

Bottom-Up Parsing

- Left-recursion is not a problem, but needs to look deeper into stack:

e.g.: E′ -> E
E -> E + n | n

Steps	Parsing stack	Input	Action
1	$	n + n $	shift
2	$ n	+ n $	reduce E -> n
3	$ E	+ n $	shift
4	$ E +	n $	shift
5	$ E + n	$	reduce E -> E + n
6	$ E	$	reduce E′ -> E
7	$ E′	$	accept

Conflicts

- Shift/reduce conflict:
  - Given stack and lookahead symbol(s), can't choose between shifting and reducing
  - e.g., dangling else:

<stmt> -> if <exp> then <stmt>
| if <exp> then <stmt> else <stmt>

- Reduce/reduce conflict:
  - Given stack and lookahead symbol(s), can't choose between possible reductions
  - e.g., multiple equivalent productions:

A -> B c d | C c e   B -> xy   C -> xy

LR Parsing

- LR(k): Left-to-right parse; Rightmost derivation; k token lookahead

- Recall:
  - LL(k) parsing predicts production to use based on first k tokens of RHS
  - LR(k) defers decision-making until it has seen all tokens of RHS and a further k tokens ahead

- LR parsing is more powerful than LL (at least for a given k token lookahead)

Implementing LR Parsers

- Deterministic Pushdown Automata (DPDA): Similar to a DFA in principle, but:
  - machine has a stack
  - a symbol can be pushed onto the stack on a transition
  - transitions are based on current state, input, and top of stack

- Essentially a DFA with memory: Implemented as a DFA with a stack, plus additional logic to consider the more sophisticated transition mechanism

LR(0) Parsing

- Possible because a lookahead token is pushed onto stack before being examined
- LR(0) items: a choice of production with a designated position in the RHS of the rule (commonly a period)

e.g., S -> A   A -> (A) | a

8 LR(0) items:

Initial items

S -> . A	S -> A .
A -> . (A)	A -> (. A)
A -> (A .)	A -> (A) .
A -> . a	A -> a .

Complete items (may be handles)

Constructing LR(0) NFA

- Use LR(0) items as states in a finite automata:
  - Construct NFA of LR(0) items, convert NFA to DFA, and minimize DFA
- Creating LR(0) NFA:
  - Case 1: $A \rightarrow \alpha \,.\, X \beta$ and X is a terminal
    - Shift X and transition to state $A \rightarrow \alpha X \,.\, \beta$ on X

$$\boxed{A \rightarrow \alpha.\, X \beta} \xrightarrow{X} \boxed{A \rightarrow \alpha X \,.\, \beta}$$

  - Case 2: $A \rightarrow \alpha \,.\, X \beta$ and X is a non-terminal
    - Transition to state $A \rightarrow \alpha X \,.\, \beta$ on X

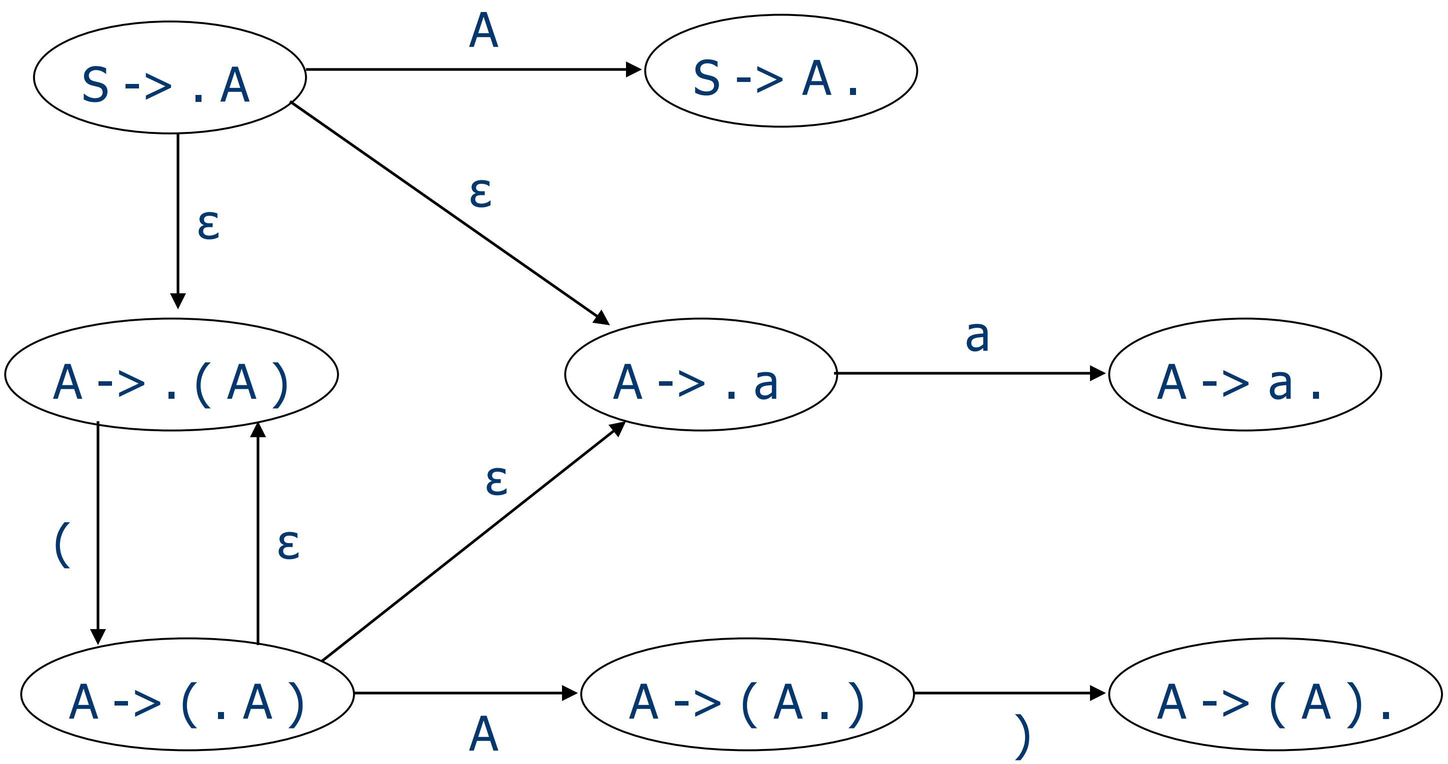
$$\boxed{A \rightarrow \alpha.\, X \beta} \xrightarrow{X} \boxed{A \rightarrow \alpha X \,.\, \beta}$$

    - ε-transition to all initial productions on X.

$$\boxed{A \rightarrow \alpha.\, X \beta} \xrightarrow{\varepsilon} \boxed{X \rightarrow \,.\, \gamma}$$

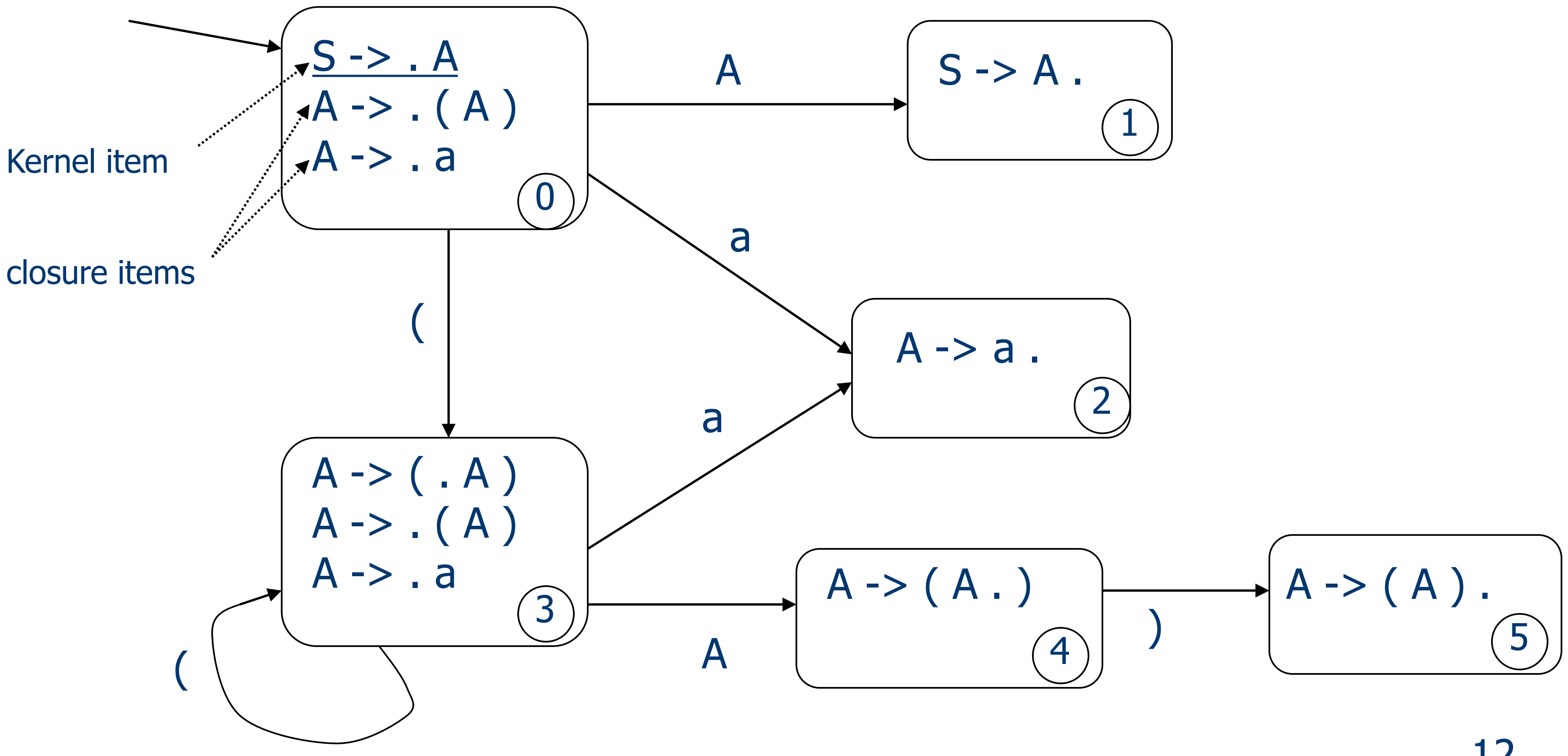
LR(0) NFA Example

e.g., S -> A   A -> (A) | a



Converting to LR(0) DFA

ε-closure: the set of items that can be reached following ε-transitions.



LR(0) DFA

- DFA states track the status of the parse, not acceptance of strings --- no final states
- Decision to shift or reduce depends entirely on the state of the DFA, not on examination of input
- Non-terminals are only shifted on a reduction (Goto)
- Missing transitions on a state represent parse errors

LR(0) Parsing Table

e.g.,   (0) S -> A   (1) A -> (A)   (2) A -> a

State	Input				Goto
	(	a	)	$	A
0	s3	s2			g1
1				a	
2	r2	r2	r2	r2	
3	s3	s2			g4
4			s5		
5	r1	r1	r1	r1	

Parsing Table Entries

- Goto function for non-terminals: map a state and a grammar symbol to a new state
  - g_n: go to state n
  - applied after a reduction action
- Action function for terminals: map a state and an input symbol to an action
  - s_n: shift to state n (push current token to stack and advance input)
  - r_k: reduce by rule k (pop stack as many time as there are symbols in RHS; in state now on top of stack, go to state for the non-terminal on LHS)
  - a: accept (stop parsing and report success)
  - empty : error (stop parsing and report failure)

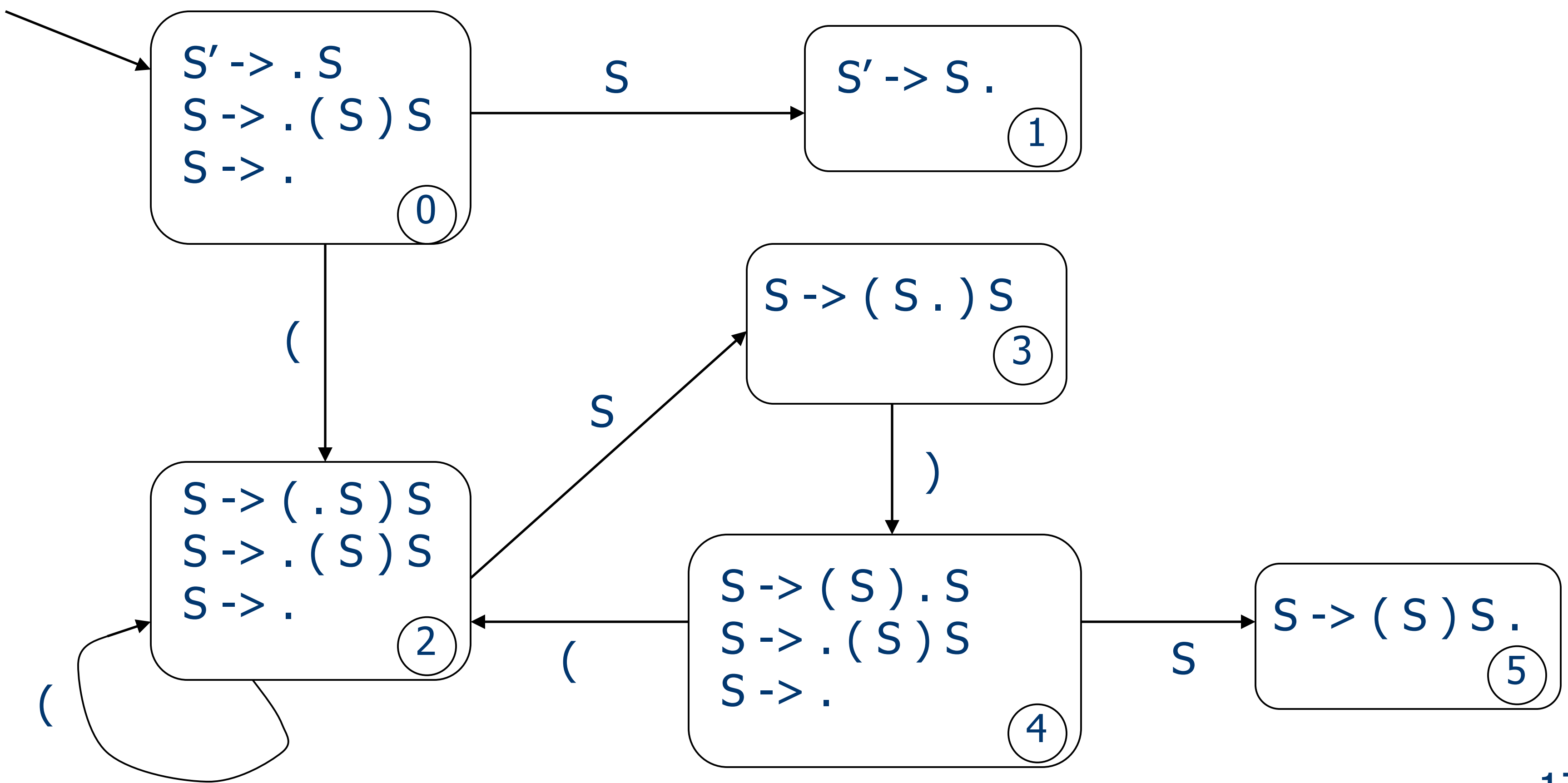
LR(0) Parsing Actions

e.g., (0) S -> A (1) A -> (A) (2) A -> a

Steps	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

Non-LR(0) Example

e.g., S′ -> S   S -> (S) S   S -> ε



Non-LR(0) Example

e.g., (0) S' -> S (1) S -> (S) S (2) S -> ε

State	Input			Goto
	(	)	$	S
0	s2, r2	r2	r2	g1
1			a	
2	s2, r2	r2	r2	g3
3		s4		
4	s2, r2	r2	r2	g5
5	r1	r1	r1	

Issues for LR(0) Parsing

- LR(0) grammar must determine a shift or a reduce based only on the current state (no lookahead)
  - Very restrictive: e.g., E -> E + n | n is not LR(0)
- If a state contains a complete LR(0) item, it can't have other items:
  - More than one completed productions: reduce/reduce conflict
  - Another item that implies a shift: shift/reduce conflict
- Very few practical grammars are LR(0)

SLR(1) Parsing

- SLR(1): Simple LR with 1 token lookahead
- Uses a DFA of LR(0) items
- Enhances power over vanilla LR(0):
  - Consults the input token before a shift to make sure that an appropriate DFA transition exists
  - Uses FOLLOW set of a non-terminal to decide if a reduction should be performed
- Many programming languages have SLR(1) grammars

SLR(1) Parsing Table

e.g., (0) S' -> S (1) S -> (S) S (2) S -> ε
FOLLOW(S) = {), $ }

State	Input			Goto
	(	)	$	S
0	s2	r2	r2	g1
1			a	
2	s2	r2	r2	g3
3		s4		
4	s2	r2	r2	g5
5		r1	r1	

Limits of SLR(1) Parsing

- Not powerful enough for some constructs:

e.g,
S -> id | V := E
V -> id
E -> V | num

Starting state:

S' -> . S
S -> . id
S -> . V := E
V -> . id

id

S -> id .
V -> id .

Reduce/reduce conflict in LR(0)

FOLLOW(S) = { $ }
FOLLOW(V) = { :=, $ }

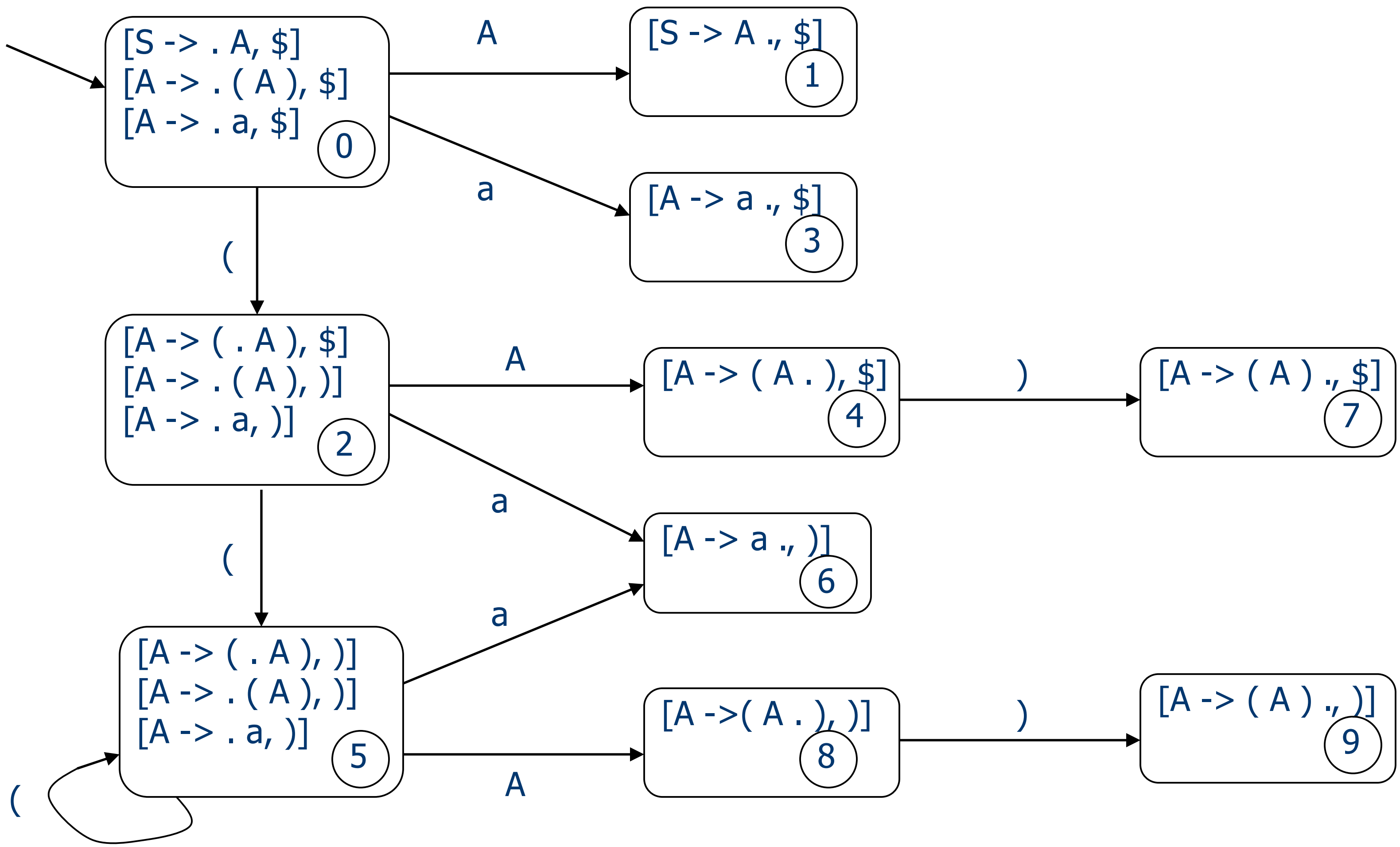
Remain to be in conflict in SLR(1)

LR(1) Parsing

- LR(1) items: contain an LR(0) item plus 1 lookahead token
- Similar process:
  - Construct NFA; translate to DFA; and minimize DFA
- Issues:
  - Increased complexity of the DFA
  - Multiplicative effect on LR(1) items due to the lookahead token
  - Shift/reduce and reduce/reduce conflicts are minimized

LR(1) Parsing

e.g., S -> A　　A -> (A)　　A -> a



LALR(1) Parsing

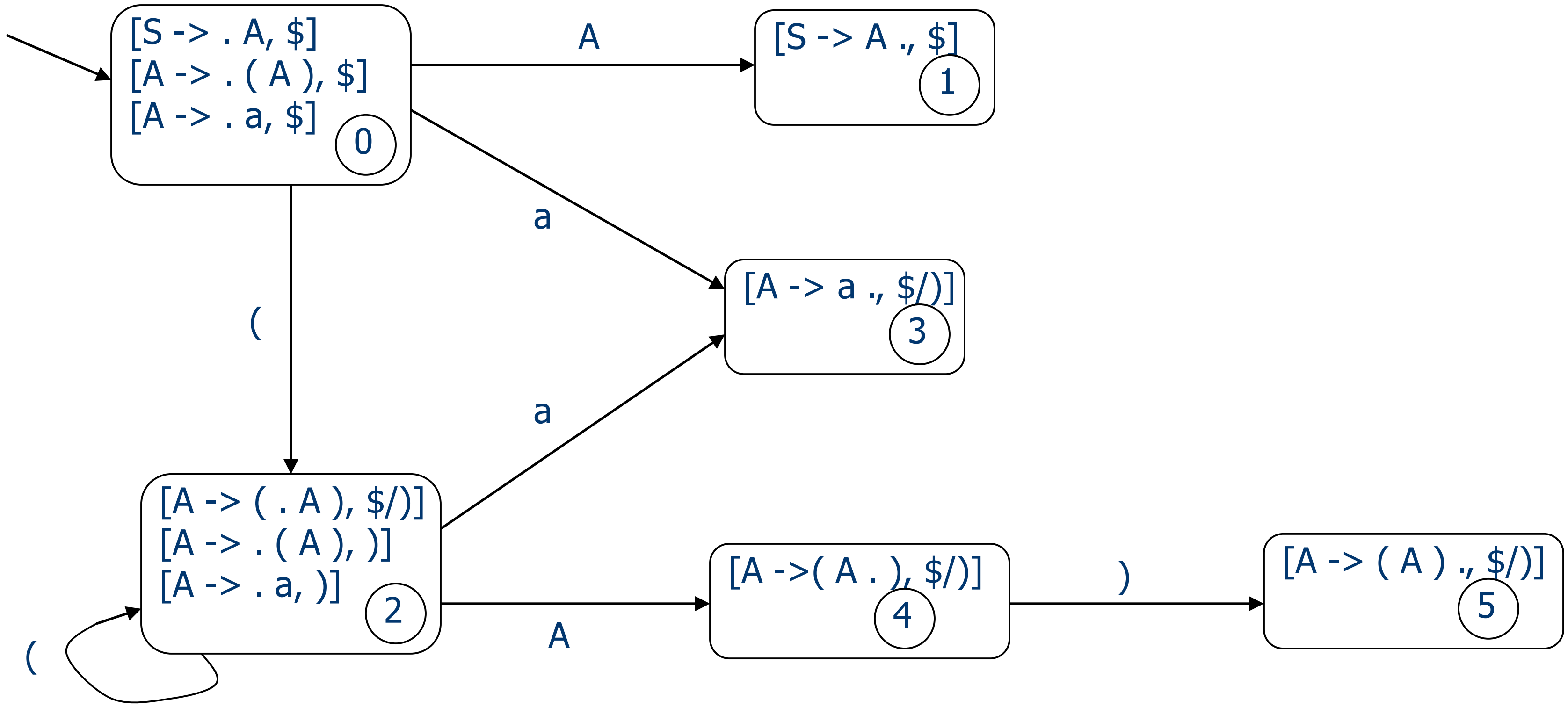
- LALR(1): Look Ahead LR with 1 lookahead token

- Large size of LR(1) tables due to many states with same set of LR(0) items, while differing only in lookahead tokens:
  - Merge states with identical LR(0) items
  - Combine lookahead tokens into a token set

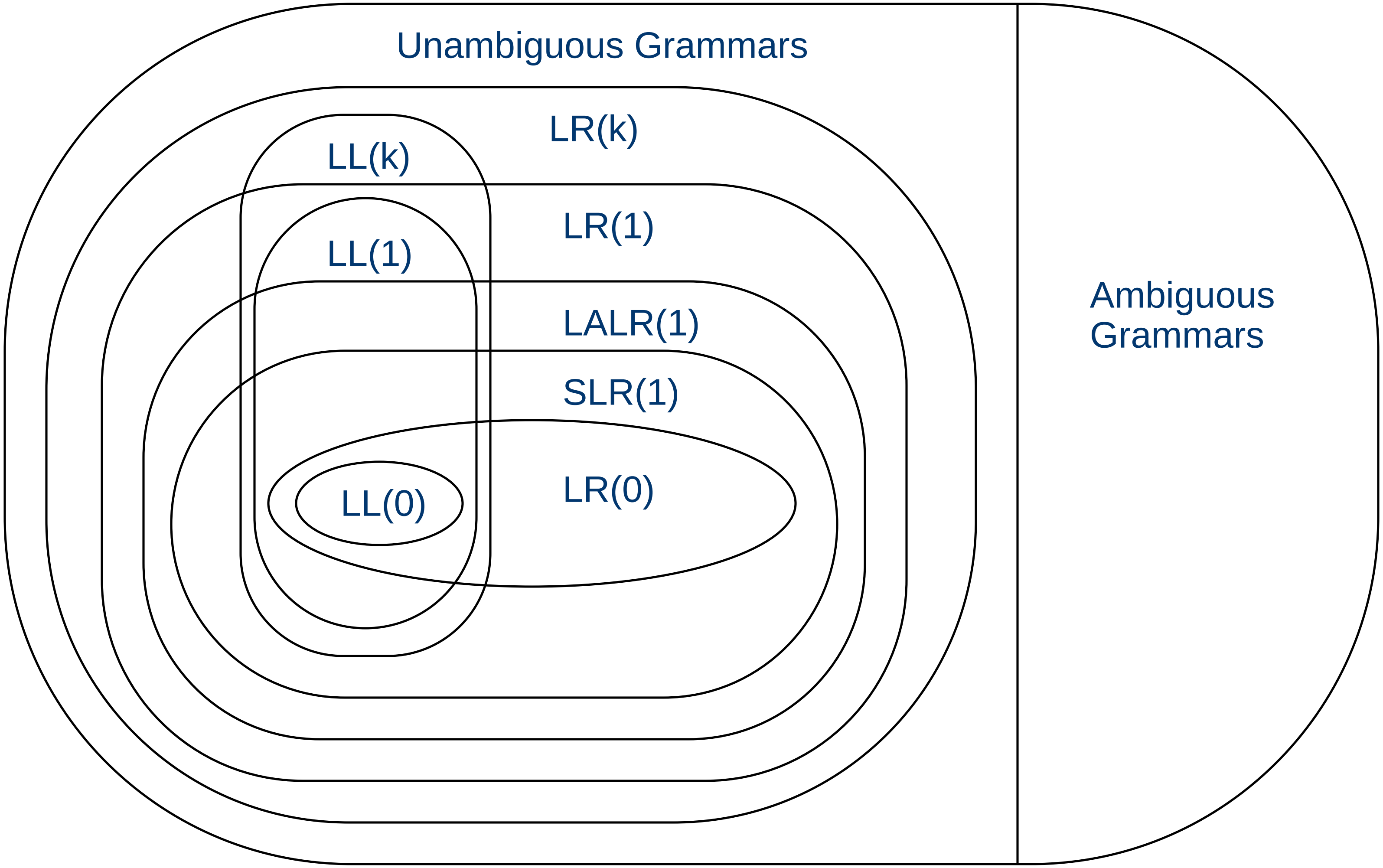
- Benefits:
  - Dramatic reduction in states over LR(1) parsing
  - Fewer chances of conflicts over SLR(1) parsing

LALR(1) Parsing

e.g., S -> A A -> (A) A -> a



Grammar Hierarchy



Parse Generators

- Parsing and scanning go hand in hand
  - Parsing is a common requirement

- Automating generation of code to implement LALR(1) parsers is relatively straightforward
  - Recall: automating recognition of tokens based on regular expressions

- Parser generators receive a CFG specification file
  - EBNF specification of a language
  - e.g., YACC and CUP

Error Recovery in Parser Tools

- Error productions: Error symbols are treated as terminals and shift actions are used for them in the parsing table

```
exp -> ID
exp -> exp + exp
exp -> ( exps )
exps -> exp
exps -> exps ; exp
exp -> ( error )
exps -> error ; exp
```

- Reducing the table size may obscure error detection: LR(1) can detect errors earlier than LALR(1), which is earlier than SLR(1), which is earlier than LR(0)

Error Recovery in Parser Tools

- When an error state is reached, do the following:
  - Pop the stack until a state is reached where the action for the error token is a shift
  - Shift the error token
  - Discard input tokens until a state is reached that has a non-error action on the current lookahead token
  - Resume normal parsing

Precedence Directives in CUP

- Given a highly ambiguous grammar:

```
exp -> exp op exp | ID | NUM | ( exp )
op -> = | <> | + | - | * | /
```

- Operator precedence directives in CUP:

```
precedence nonassoc EQ, NEQ
precedence left PLUS, MINUS
precedence left TIMES, DIV
```

Precedence Directives in CUP

- Rule precedence directives in CUP:

```
terminal        INT, PLUS, MINUS, TIMES, DIVIDES, UMINUS;
non terminal    exp;

precedence left    PLUS, MINUS;
precedence left    TIMES, DIVIDES;
precedence right   UMINUS;

start with exp;

exp   ::=  INT
        |    exp  PLUS  exp
        |    exp  MINUS  exp
        |    exp  TIMES exp
        |    exp  DIVIDES exp
        |    MINUS  exp  %prec  UMINUS;
```