COMPILER DESIGN

Bottom-up parsing: Canonical LR and LALR

Error recovery in LR parsing

Parsing: concluding comments

Problems with LR parsers

- The SLR parser breaks down when there are conflicts between shift/recude:
  - Shift-reduce conflict: it cannot decide whether to shift or reduce

```
S →  if E then S
  |  if E then S else S
```

  - Reduce-reduce conflict: A given configuration implies more than one possible reduction, e.g.

State 3 : V[**id**]	$S \to$ **id**•
	 $E \to$ **id**•
		FOLLOW(S) = {\$}
		 FOLLOW(E) = {\$,+,=}

```
Z → S
S → E = E
S → id
E → E + id
E → id
```

Canonical LR parsers

- For example, the previous example problem is not without solution:
  - if the next token is either + or =, reduce **id** to E
  - if the next token is $, reduce **id** to S
- Solution: use *lookahead sets* in the items generation process to eliminate ambiguities

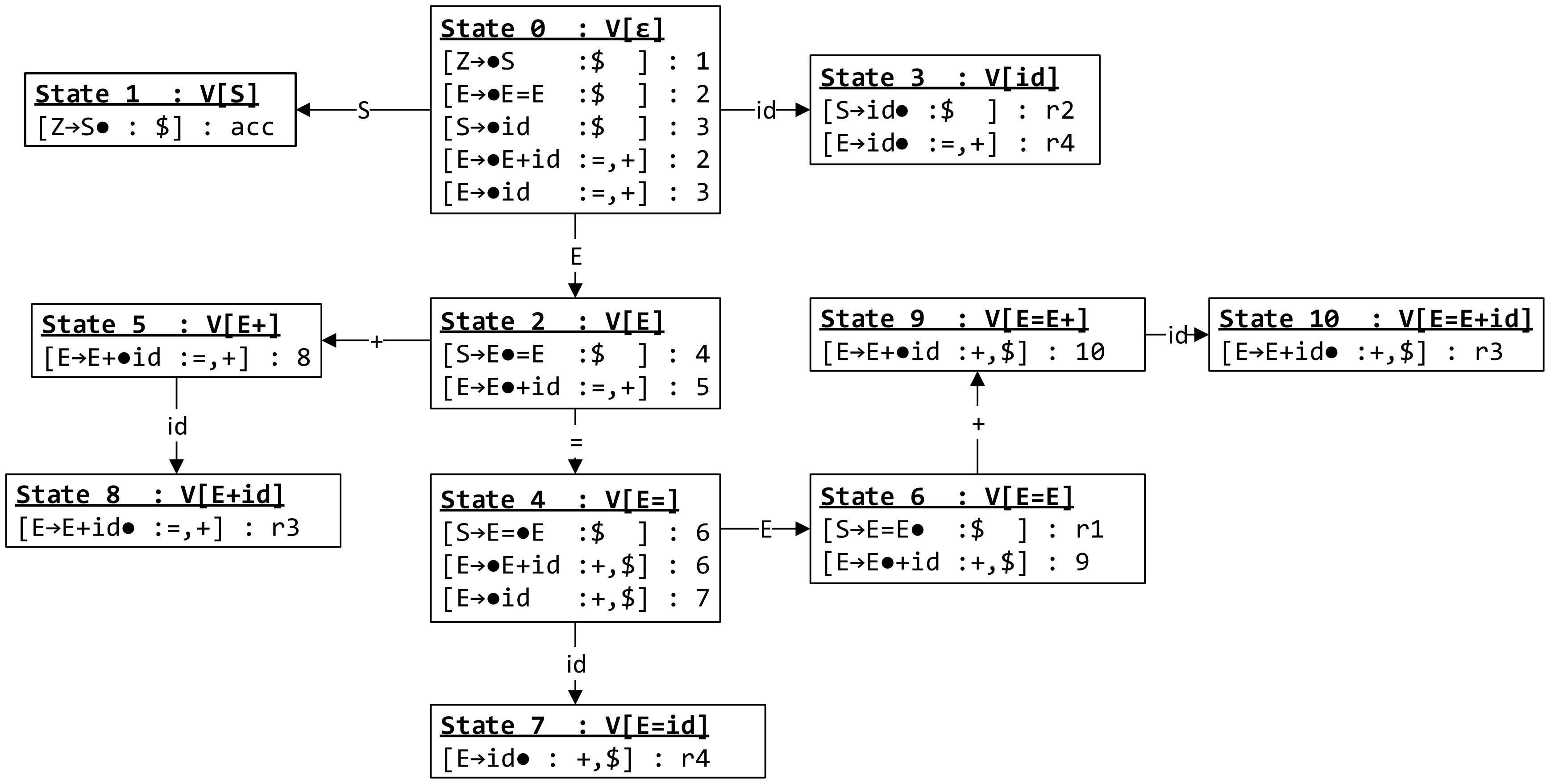
Canonical LR parsing: item sets generation

```
State 0: [Z → •S    :{$}]   : [Z → •S    :{$}]    : V[S]     : State 1
V[ε]     [S → •id   :{$}]     [S → •E=E  :{$}]    : V[E]     : State 2
         [S → •E=E  :{$}]     [E → •E+id :{=,+}]  : V[E]     : State 2
         [E → •E+id :{=}]     [E → •id]  :{=,+}]  : V[id]    : State 3
         [E → •id   :{=}]     [S → •id   :{$}]    : V[id]    : State 3
         [E → •E+id :{+}]
         [E → •id   :{+}]

State 1: [Z → S•    :{$}]   : [Z → S•    :{$}]    : accept   : Final State
V[S]

State 2: [S → E•= E :{$}]   : [S → E•=E  :{$}]    : V[E=]    : State 4
V[E]     [E → E•+id :{=,+}    [E → E•+id :{=,+}]  : V[E+]    : State 5

State 3: [E → id•   :{=,+}] : [E → id•   :{=,+}]  :          : handle (r4)
V[id]    [S → id•   :{$}]     [S → id•   :{$}]    :          : handle (r2)

State 4: [S → E=•E  :{$}]   : [S → E=•E  :{$}]    : V[E=E]   : State 6
V[E=]    [E → •E+id :{$}]     [E → •E+id :{+,$}]  : V[E=E]   : State 6
         [E → •id   :{$}]     [E → •id   :{+,$}]  : V[E=id]  : State 7
         [E → •E+id :{+}]
         [E → •id   :{+}]
```

Canonical LR parsing: item sets generation

```
State 5: [E → E+•id :{=,+}] : [E → E+•id :{=,+}]   : V[E+id]   : State 8
V[E+]

State 6: [S → E=E•  :{$}]   : [S → E=E•  :{$}]     :           : handle (r1)
V[E=E]   [E → E•+id :{+,$}]   [E → E•+id :{+,$}]   : V[E=E+]   : State 9

State 7: [E → id•   :{+,$}] : [E → id•   :{+,$}]   :           : handle (r4)
V[E=id]

State 8: [E → E+id• :{=,+}] : [E → E+id• :{=,+}]   :           : handle (r3)
V[E+id]

State 9: [E → E+•id :{+,$}] : [E → E+•id :{+,$}]   : V[E=E+id] : State 10
V[E=E+]

State 10:[E → E+id• :{+,$}] : [E → E+id• :{+,$}]   :           : handle (r3)
V[E=E+id]
```

Canonical LR parsing: corresponding DFA



```
State 0   : V[ε]
[Z→•S     :$   ] : 1
[E→•E=E   :$   ] : 2
[S→•id    :$   ] : 3
[E→•E+id :=,+] : 2
[E→•id    :=,+] : 3

State 1   : V[S]
[Z→S• : $] : acc

State 2   : V[E]
[S→E•=E   :$   ] : 4
[E→E•+id :=,+] : 5

State 3   : V[id]
[S→id• :$   ] : r2
[E→id• :=,+] : r4

State 4   : V[E=]
[S→E=•E   :$   ] : 6
[E→•E+id :+,$] : 6
[E→•id    :+,$] : 7

State 5   : V[E+]
[E→E+•id :=,+] : 8

State 6   : V[E=E]
[S→E=E•   :$   ] : r1
[E→E•+id :+,$] : 9

State 7   : V[E=id]
[E→id• : +,$] : r4

State 8   : V[E+id]
[E→E+id• :=,+] : r3

State 9   : V[E=E+]
[E→E+•id :+,$] : 10

State 10   : V[E=E+id]
[E→E+id• :+,$] : r3
```

Canonical LR parsing: parsing table

state	action				goto	
	id	=	+	$	S	E
0	s3				1	2
1				acc		
2		s4	s5			
3		r4	r4	r2		
4	s7					6
5	s8					
6			s9	r1		
7			r4	r4		
8		r3	r3			
9	s10					
10			r3	r3		

0	Z → S
1	S → E = E
2	S → id
3	E → E + id
4	E → id

LALR

- Problem: CLR generates more states than SLR, e.g. for a typical programming language, the SLR table has hundreds of states whereas a CLR table has thousands of states
- Solution: merge similar states, i.e. states that have the same item sets, but not necessarily with the same lookahead sets. Merge such states into one state, and merge the lookahead sets of items that are duplicated.

LALR

```
A.        State 0:  {[Z → •S:{$}],[S → •id:{$}],[S → •E=E:{$}],
                     [E → •E+id:{=,+}],[E → •id]:{=,+}]}
B.        State 1:  {[Z → S•:{$}]}
C.        State 2:  {[S → E•=E:{$}],[E → E•+id:{=,+}]}
D.        State 3:  {[E → id•:{=,+}],[S → id•:{$}]}
E.        State 4:  {[S → E=•E:{$}],[E → •E+id:{+,$}],[E → •id:{+,$}]}
F.        State 5:  {[E → E+•id:{=,+}]}
          State 9:  {[E → E+•id:{+,$}]}
G.        State 6:  {[S → E=E•:{$}],[E → E•+id:{+,$}]}
H.        State 7:  {[E → id•:{+,$}]}
I.        State 8:  {[E → E+id•:{=,+}]}
          State 10: {[E → E+id•:{+,$}]}
```

```
          State 0:  {[Z → •S:{$}],[S → •id:{$}],[S → •E=E:{$}],
                     [E → •E+id:{=,+}],[E → •id]:{=,+}]}
          State 1:  {[Z → S•:{$}]}
          State 2:  {[S → E•=E:{$}],[E → E•+id:{=,+}]}
          State 3:  {[E → id•:{=,+}],[S → id•:{$}]}
          State 4:  {[S → E=•E:{$}],[E → •E+id:{+,$}],[E → •id:{+,$}]}
          State 5:  {[E → E+•id:{=,+,$}]}
          State 6:  {[S → E=E•:{$}],[E → E•+id:{+,$}]}
          State 7:  {[E → id•:{+,$}]}
          State 8:  {[E → E+id•:{=,+,$}]}
```

LALR parsing table

state	action				goto	
	id	=	+	$	S	E
0	s3				1	2
1				acc		
2		s4	s5			
3		r4	r4	r2		
4	s7					6
5	s8					
6			s5	r1		
7			r4	r4		
8		r3	r3	r3		

1	$Z \rightarrow S$
2	$S \rightarrow E = E$
3	$S \rightarrow id$
4	$E \rightarrow E + id$
5	$E \rightarrow id$

Error recovery in LR parsing

Error detection in LR parsers

- An error is detected when the parser consults the action table and finds an empty entry
- Each empty entry potentially represents a different and specific syntax error
- If we come onto an empty entry on the goto table, it means that there is an error in the table itself

Error recovery in LR parsers

```
onError()
    1. pop() until a state s is on top of the stack and T is not empty
        where
         s has at least one entry in the goto table under non-terminal N
         S is the set of states in the goto table for s
         T is the set of terminals for which the elements of S
           have a shift or accept entry in the action table
    4. lookahead = nextToken() until lookahead x is in T
    5. push the N corresponding to x and its corresponding state in S
    6. resume parse
```

Error recovery in LR parsers

- This method of error recovery attempts to eliminate the remainder of the current phrase derivable from a non-terminal A which contains a syntactic error.
- Part of that A was already parsed, which resulted in a sequence of states on top of the stack.
- The remainder of this phrase is still in the input.
- The parser attempts to skip over the remainder of this phrase by looking for a symbol on the input that can legitimately follow A.
- By removing states from the stack, skipping over the input, and pushing GOTO(s, A) on the stack, the parser pretends that it has successfully reduced an instance of A and resumes normal parsing.

- Example: in many programming languages, statements end with a ";"
  - if an error is found in a statement, the stack is popped until we get V[<statement>] on top of the stack
  - nextToken() until the next ";" is found
  - resume parse

Error recovery in LR parsers: example

state	action						goto		
	id	+	*	(	)	$	E	T	F
0	s5	e1	e1	s4	e1	e0	1	2	3
1	e2	s6	ee	e2	e3	acc			
2	e2	r2	s7	e2	r2	r2			
3	e2	r4	r4	e2	r4	r4			
4	s5	e1	e1	s4	e1	e0	8	2	3
5	e2	r6	r6	e2	r6	r6			
6	s5	e4	e4	s4	e4	e0		9	3
7	s5	e5	e5	s4	e5	e0			10
8	e2	r1	s7	e2	s11	e0			
9	e2	r1	s7	e2	r1	r1			
10	e2	r3	r3	e2	r3	r3			
11	e2	r5	r5	e2	r5	r5			

1	E → E + T
2	E → T
3	T → T * F
4	T → F
5	F → (E)
6	F → id

Error recovery in LR parsers: example

	stack	input	action
1	0	id)id*id)$	shift 5
2	0id5	)id*id)$	reduce (F → id)
3	0F3	)id*id)$	reduce (T → F)
4	0T2	)id*id)$	reduce (E → T)
5	0E1	)id*id)$	e3
6	0T2	*id)$	shift 7
7	0T2*7	id)$	shift 5
8	0T2*7id5	)$	reduce (F → id)
9	0T2*7F10	)$	reduce (T → T * F)
10	0T2	)$	reduce (E → T)
11	0E1	)$	e3
12	0E1	$	accept

e0	unexpected end of program
e1	missing operand
e2	missing operator
e3	mismatched parenthesis
e4	missing term
e5	factor expected
e6	+ or) expected
ee	parser error

1	E → E + T
2	E → T
3	T → T * F
4	T → F
5	F → (E)
6	F → id

Parsing: general comments

Parser generators

- For real-life programming languages, construction of the table is extremely laborious and error-prone (several thousands of states)
- Table construction follows strictly defined rules that can be implemented as a program called a parser generator
- Yacc (Yet Another Compiler-Compiler) generates a LALR(1) parser code and table
- Grammar is given in input in (near) BNF
- Detects conflicts and resolves some conflicts automatically
- Requires a minimal number of changes to the grammar
- Each right hand side is associated with a custom semantic action to generate the symbol table and code

Which method to use?

- We have seen
  - Top-down: recursive descent predictive, table-driven predictive
  - Bottom-up: SLR, CLR, LALR
  - Parser generator
- For real-life languages, the recursive descent parser lacks maintainability
- The need for changing the grammar is a disadvantage for predictive parsers
- Code generation and error detection is more difficult and less accurate in top-down parsers
- Most compilers are now implemented using the LR method, using parser generators
- Other more recent ones are generating top-down parsing methods (e.g. JavaCC)