Imperial College London
Department of Computing

Compilers (221)

Exercises – LR Parsing

Check your answers with the tutorial helpers during tutorials and with each other on Piazza.

Suggested order to do the questions: 4, 5, 6, 7, 14, 15 and then the remaining questions. Let me know if you find a mistake or have a better answer.

1. Consider the following grammar with start symbol S: L1

```
S → B a | b C
B → d | e B f
C → g C | g S
```

For each of the following strings, give a derivation for the string or say whether the string can or cannot be generated by the grammar:

(i) da (ii) bddf (iii) eedffa (iv) bggda

++

```
(i)    S => Ba => da
(ii)   S => bC => FAIL
(iii)  S => Ba => eBfa => eeBffa => eedffa
(iv)   S => bC => bgC => bggS => bggBa => bggda
```

2. Show that all binary strings generated by the following grammar have values divisible by 3. Hint: use induction on the numerical values for nodes in the parse tree. L3

```
num → 11 | 1001 | num 0 | num num
```

++

Base case: If the parse tree only has a single production, then it must be either 11 or 1001. This gives us either 3 or 9, both of which are divisible by 3.

Inductive case: We assume that we have a parse tree for a string that is divisible by 3.

If we replace the root of the parse tree with the production num 0 this produces a new string that is the old string with a zero appended to the right hand side. This is the same as multiplying the number by 2. Since, the old string was divisible by 3, the new string will still be divisible by 3.

The num num case requires two parse trees, each for a number that is divisible by 3. We can think of these as 3x and 3y. If there are n bits in 3y, this results in a new number that is equal to $(3x)*2^n + 3y$, which is equal to $3*(2^n x+y)$, which is still divisible by 3.

3. For the rewritten **if** statement grammar in the slides (the grammar with rules for Matched and Unmatched statements about slide 37), draw the parse tree for L2

if 1 **then** **if** 0 **then** *other* **else** *other*

```
           S
           |
   --------U----------
   |                 S
   |                 |
   |         --------M----------
   |         |       |          |
   E         E       M          M
   |         |       |          |
if 1 then if 0 then other else other
```

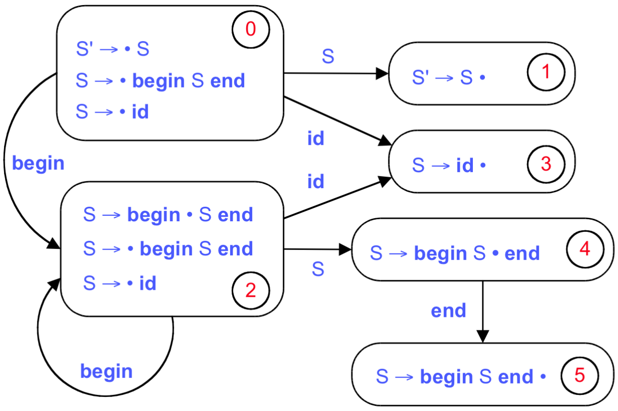
4. For the following grammar: L3

```
Statement → begin Statement end | id
```

construct the DFA of LR(0) items and the LR(0) Parsing Table.

You should build the DFA directly without first building the NFA. Your DFA should have 6 states. For conciseness, use the letter `S` in your items instead of `Statement`. Remember to augment the grammar with an auxiliary rule and give numbers for your rules when used in reduce actions in your Parsing Table.

++



```
r1:  Statement → begin Statement end
r2:  Statement → id
```

State	ACTION				GOTO
	begin	*id*	*end*	$	S
0	s2	s3			g1
1				a	
2	s2	s3			g4
3	r2	r2	r2	r2	
4			s5		
5	r1	r1	r1	r1	

5. Construct the FIRST set and FOLLOW set for the rules (non-terminals) of the following grammar: L2

```
Statement      → IfStatement | other
IfStatement    → if '(' Expression ')' Statement ElsePart
ElsePart       → else Statement | ε
Expression     → 0 | 1
```

++

Non-terminal	FIRST Set	FOLLOW Set
Statement	*if*, *other*	$, *else*
IfStatement	*if*	$, *else*
ElsePart	*else*, ε	$, *else*
Expression	0, 1	)

6. Construct the FIRST set and FOLLOW set for the rules (non-terminals) of the following grammar: L2

```
Program        → Statements
Statements     → Statement Statements | ε
Statement      → id '=' Expression | read id | write Expression
Expression     → Term TermTail
TermTail       → AddOp Term TermTail | ε
```

```
Term          → Factor FactorTail
FactorTail    → MultOp Factor FactorTail | ε
Factor        → '(' Expression ')' | id | num
AddOp         → '+' | '–'
MultOp        → '*' | '/'
```

+++

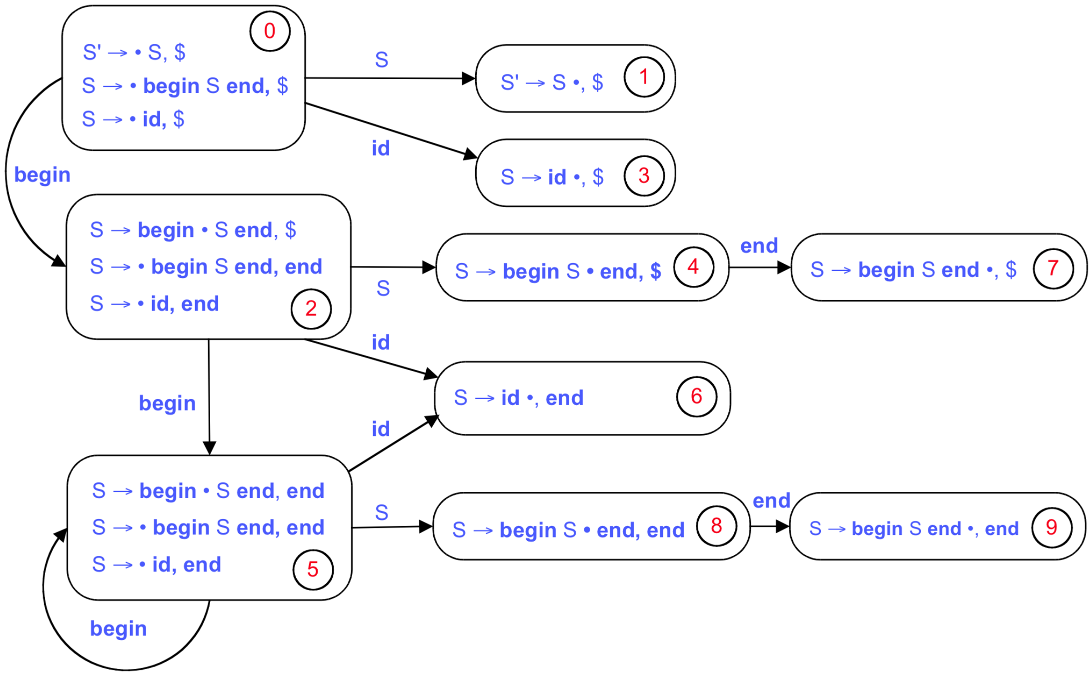
Non-terminal	FIRST Set	FOLLOW Set
Program	*id*, *read*, *write*, ε	$
Statements	*id*, *read*, *write*, ε	$
Statement	*id*, *read*, *write*	$, *id*, *read*, *write*
Expression	(, *id*, *num*	$, *id*, *read*, *write*,)
TermTail	+, -, ε	$, *id*, *read*, *write*,)
Term	(, *id*, *num*	$, *id*, *read*, *write*,), +, -
FactorTail	*, /, ε	$, *id*, *read*, *write*,), +, -
Factor	(, *id*, *num*	$, *id*, *read*, *write*,), +, -, *, /
AddOp	+, -	(, *id*, *num*
MultOp	*, /	(, *id*, *num*

7. For the following grammar: L4

```
Statement → begin Statement end | id
```

construct the DFA of LR(1) items and the LR(1) Parsing Table. Your DFA should have 10 states.

+++



```
r1:  Statement → begin Statement end
r2:  Statement → id
```

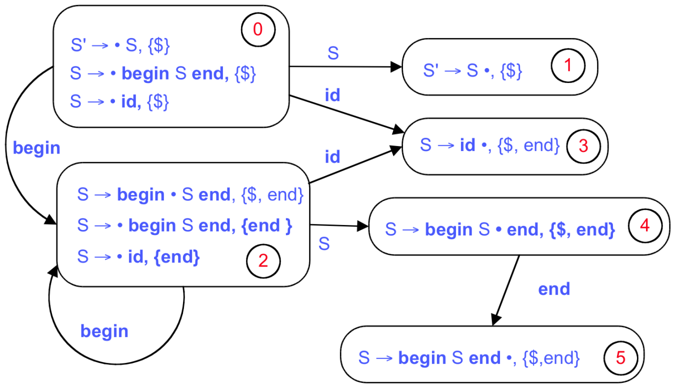
State	ACTION				GOTO
	begin	*id*	*end*	$	S
0	s2	s3			g1
1				a	
2	s5	s6			g4
3				r2	
4			s7		
5	s5	s6			g8
6			r2		
7				r1	
8			s9		
9			r1		

8. For the following grammar: L1

```
Statement → begin Statement end | id
```

construct the DFA of LALR(1) items from your solution to the previous question. Your DFA should have 6 states.

++



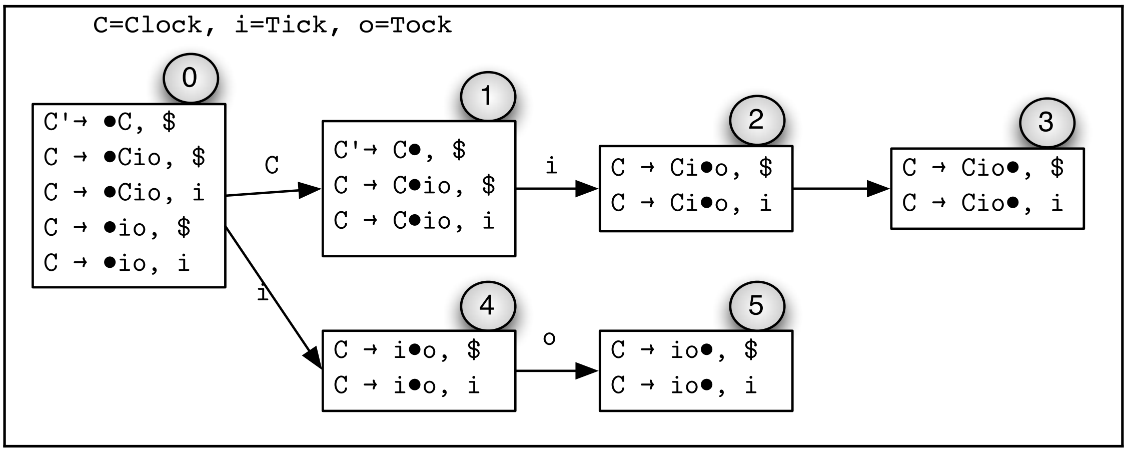
9. For the following grammar: L3

```
Clock → Clock tick tock | tick tock
```

i) Construct the DFA of LR(1) items. Use `C` for `Clock`, `i` for **`tick`**, `o` for **`tock`**. Your DFA should have 6 states.

ii) Construct the parse table from the DFA of LR(1) items and explain whether the grammar is LR(1).

++



Rule numbers:

```
R1: C → Cio
R2: C → io
```

	ACTION			GOTO
	i	o	$	C
0	S4			G1
1	S2		A	
2		S3		
3	R1		R1	
4		S5		
5	R2		R2	

10. Consider a robot arm that accepts two commands: **down** that puts an apple in a basket, and **up** that takes an apple out of the basket. Assume the robot arm starts with an empty basket. A valid command sequence for the robot arm should have no prefix that contains more **down** commands than **up** commands, i.e. taking from an empty basket is not permitted. L3

As examples, **down down up up** and **down up down** are valid command sequences, but **up down** and **down up up down** are not.

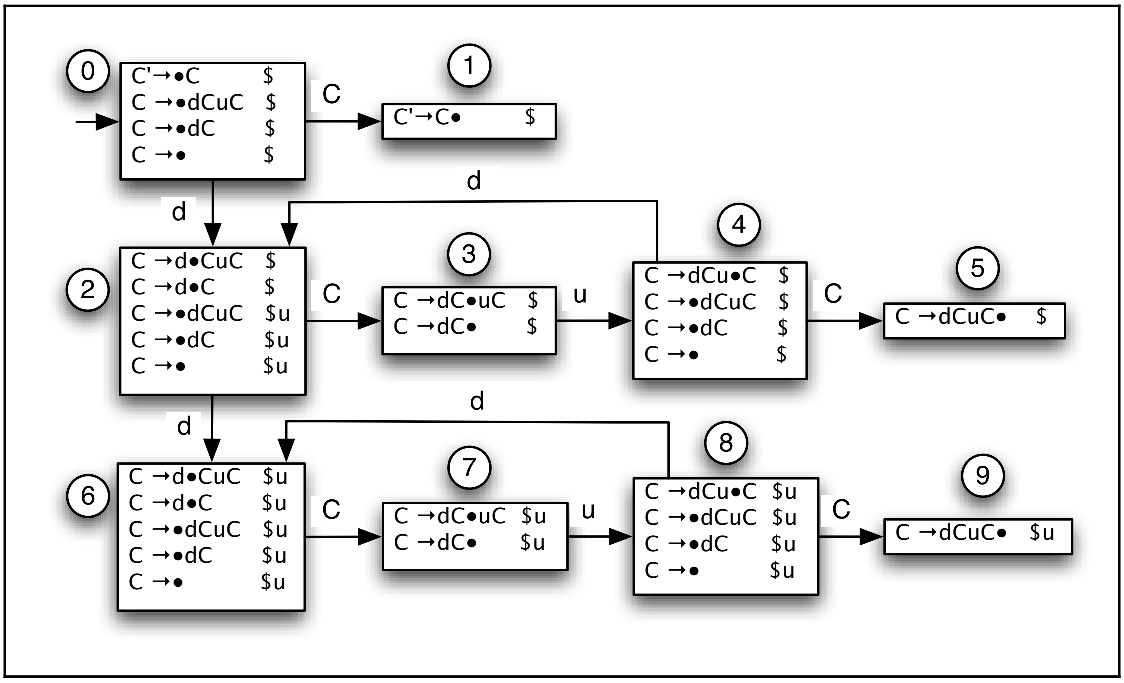
Devise a context-free grammar for all valid command sequences. For your grammar construct the DFA of LR(1) items and explain whether the grammar is LR(1).

++

The following context-free grammar represents all the valid command sequences:

```
C'  →  C
C   →  d C u C
C   →  d C
C   →  ε
```

where d is **down** and u is **up**



Should also be a ‘d’ arrow from state 6 to state 6. The grammar isn’t LR(1) since we have a shift-reduce conflict in State 7 for u, e.g. the input up up down has 2 derivations. We could also show that the grammar is ambiguous i.e. how more than one derivation for the same string.

11. Explain which, if any, of SLR(1) and LR(1) can parse the following grammar with start symbol G: L2

```
G → S | T
S → x | z
T → y | z
```

++

The grammar is ambiguous; there are two possible derivations of the string “z". Strictly neither SLR(1) nor LR(1) (nor LL(1)) is able to parse any ambiguous grammar. While it is possible to work through complete SLR(1), and LR(1) constructions to show exactly where they fail, that is not necessary. As soon as you show that a grammar is ambiguous you immediately know that none of SLR(1), or LR(1) can possibly work.

12. For the following grammar: L4

```
S → C C
C → a C | b
```

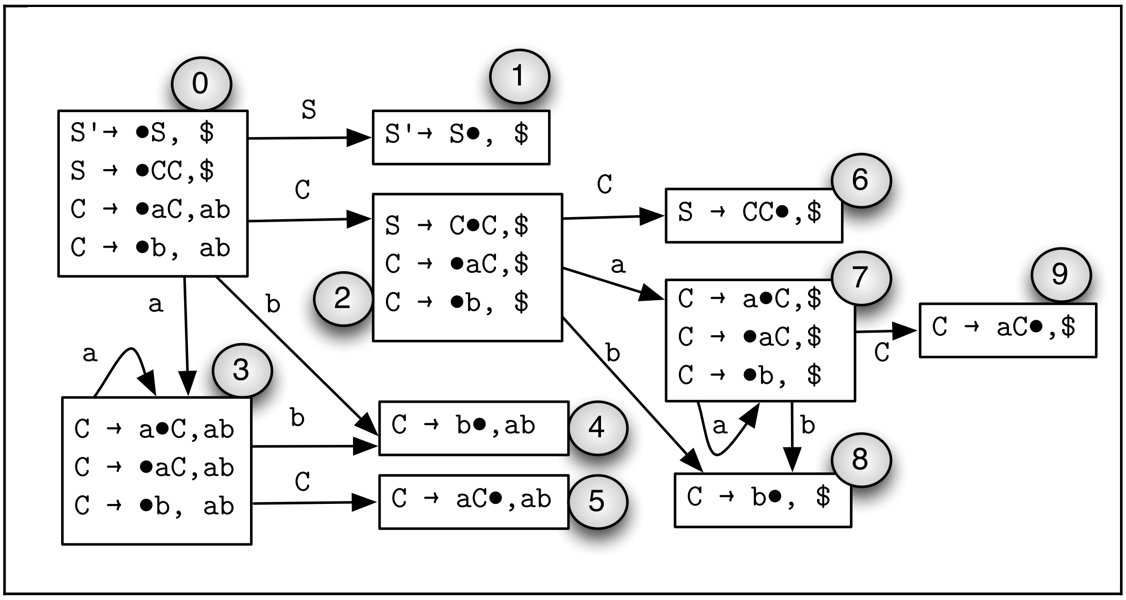
ii) Construct the DFA of LR(1) items.

ii) Construct the parse table from the DFA of LR(1) items. Your DFA should have 10 states.

iii) How many states would the DFA of LALR (1) items have? Explain your answer.

iv) Give a regular expression for the strings that S recognises.

++



Rule numbers:

```
  1: S → CC
  2: C → aC
  3: C → b
```

	ACTION			GOTO	
	a	b	$	S	C
0	s3	s4		g1	g2
1			a		
2	s7	s8			g6
3	s3	s4			g5
4	r3	r3			
5	r2	r2			
6			r1		
7	s7	s8			g9
8			r3		
9			r2		

The DFA of LALR(1) items would have 7 states, because we can combine State 3 with State 7, State 4

	with State 8, State 5 with state 9. The grammar recognises the regular language `a*b a*b`	
13.	For the LR(1) example in the slides describe how the input `id = int` is matched and the AST built by the DFA (slide 31) Left to reader.	L4
14.	Download the calc example from the website and make the parser with flex and bison. On a Mac, download and install Xcode from the App Store. 1) Execute the example with your own expression (warning there may be a command called calc). 2) How many LALR(1) states are did bison generate? Hint: look at the .output file 15 (states are numbered from 0) 3) Draw the DFA from the states. Left to the reader. 4) Adapt the example to handle division.	L4
15.	Download the extended parser example from the website and make the parser 1) How many LALR(1) states did bison generate? 82 2) Which state had the most shift/reduce conflicts? 60 3) Type in a program consisting of a legal if-then-else statement, i.e. your program should not produce Error: syntax error. Check parser.y for correct syntax.	L2
16.	Download the ANSI C parser example from the website and make the C language parser 1) How many LALR(1) states were generated? 350 2) Which rule generated a shift-reduce conflict? The classical . before 'else' for the if-then-else rule.	L1