Unit 8: LR(0) Parsing
A Worked Example LR(k) (Part 1)

SCC 312 Compilation

Aims

- To present a detailed worked example of LR(0) parsing, showing the data structures and algorithms applied

- To show how it produces a rightmost derivation of the input string

LR(0) parsing

- The parser has an
  - Input buffer
  - A stack on which it keeps a list of states it has been in
  - An action table that tells it which new state to move to
  - A goto table that tells it which grammar rule it should use given the state it is currently in and the terminal or non-terminal it has just read on the input stream

LR(0) parsing

- To help explain how this works, we'll use the following small grammar:
  - (1) $E \rightarrow E * B$
  - (2) $E \rightarrow E + B$
  - (3) $E \rightarrow B$
  - (4) $B \rightarrow 0$
  - (5) $B \rightarrow 1$

Action Table

- The action table is indexed by
  - a state of the parser and
  - a terminal (including a special non-terminal $ that indicates the end of the input stream)

Action Table

- An entry contains three types of actions:
  - a shift that is written as 'sn ' and indicates that the next state is n,
  - a reduce that is written as 'rm ' and indicates that a reduction with grammar rule m should be performed
  - and an accept that is written as 'acc' and indicates that the parser accepts the string in the input stream.

Goto Table

- The goto table is indexed by a state of the parser and a non-terminal and simply indicates what the next state of the parser will be if it has recognized a certain non-terminal.

Action and Goto Tables

state	*	+	0	1	$
0			s1	s2	
1	r4	r4	r4	r4	r4
2	r5	r5	r5	r5	r5
3	s5	s6			acc
4	r3	r3	r3	r3	r3
5			s1	s2	
6			s1	s2	
7	r1	r1	r1	r1	r1
8	r2	r2	r2	r2	r2

Action

state	E	B
0	3	4
1		
2		
3		
4		
5		7
6		8
7		
8		

Goto

The Parsing Algorithm (1)

- The LR parsing algorithm now works as follows:
  - A **stack** is initialized with [0]. The current state will always be the state that is on the top of the stack.
  - Given the current state and the current terminal on the input stream, an action is looked up in the action table.
  - There are four possible entries in this table.

0
stack

Entry options one and two

- a *shift* operation **sn**:
  - the current terminal is removed from the input stream
  - and the state n is pushed onto the stack and becomes the current state
- a *reduce* operation **rm**:
  - the number m is written to the output stream
  - for every symbol in the right-hand side of rule m a state is removed from the stack
  - given the state that is then on top of the stack and the left-hand side of rule m, a new state is looked up in the goto table and made the new current state by pushing it onto the stack.

0
stack

Entry options three and four

- an *accept*: the string is accepted as correct
- no action: a syntax error

0
stack

Worked Example

- Parsing “1 + 1”

(1) $E \rightarrow E * B$

(2) $E \rightarrow E + B$

(3) $E \rightarrow B$

(4) $B \rightarrow 0$

(5) $B \rightarrow 1$

Worked Example

- To explain why this algorithm works we now proceed with showing how a string like “1 + 1” would be parsed by such a parser. When the parser starts it always starts with the initial state 0 and the stack shown.

in

1	+	1	$

out

(1) E → E * B

(2) E → E + B

(3) E → B

(4) B → 0

(5) B → 1

0
stack

Worked Example

in

1	+	1	$

- The first terminal that the parser sees is the '1' and according to the action table it should then go to state 2 resulting in the stack shown.
- For the sake of our explanation we also show the symbol (‘1’) that caused the transition to the next state, although strictly speaking it is not part of the stack.

s	*	+	0	1	$
0			s1	s2	
1	r4	r4	r4	r4	r4
2	r5	r5	r5	r5	r4
3	s5	s6			acc
4	r3	r3	r3	r3	r3
5			s1	s2	
6			s1	s2	
7	r1	r1	r1	r1	r1
8	r2	r2	r2	r2	r2

2 (‘1’)
0
stack

Worked Example

in	1	+	1	$

- After using a shift action (s2 in this case), we fetch the next thing from the input stream (a “+”) to become the new current input

in	1	+	1	$

s	*	+	0	1	$
0			s1	s2	
1	r4	r4	r4	r4	r4
2	r5	r5	r5	r5	r4
3	s5	s6			acc
4	r3	r3	r3	r3	r3
5			s1	s2	
6			s1	s2	
7	r1	r1	r1	r1	r1
8	r2	r2	r2	r2	r2

2 (‘1’)
0
stack

Worked Example

in

1	+	1	$

- In state 2 the action table says that whatever terminal we see on the input stream we should do a *reduction* with grammar rule 5.
- If the table is correct then this means that the parser has just recognized the right-hand side of rule 5, which is indeed the case.

s	*	+	0	1	$
0			s1	s2	
1	r4	r4	r4	r4	r4
2	r5	r5	r5	r5	r5
3	s5	s6			acc
4	r3	r3	r3	r3	r3
5			s1	s2	
6			s1	s2	
7	r1	r1	r1	r1	r1
8	r2	r2	r2	r2	r2

(1) E → E * B

(2) E → E + B

(3) E → B

(4) B → 0

(5) B → 1

2 (‘1’)
0
stack

Worked Example

in

1	+	1	$

- a reduce rm: (r5)
  - the number m (5) is written to the output stream
  - for every symbol in the right-hand side of rule m a state is removed from the stack.
  - There is only one symbol on the RHS so ww remove a single state.

(5) B → 1

s	*	+	0	1	$
0			s1	s2	
1	r4	r4	r4	r4	r4
2	r5	r5	r5	r5	r5
3	s5	s6			acc
4	r3	r3	r3	r3	r3
5			s1	s2	
6			s1	s2	
7	r1	r1	r1	r1	r1
8	r2	r2	r2	r2	r2

out

5			

2 (‘1’)
0
stack

→

0
stack

Worked Example

in

1	+	1	$

- a reduce rm: (r5) (5) B $\rightarrow$ 1
  - given the state that is now on top of the stack (0) and the left-hand side of rule m (5), a new state is looked up in the goto table and made the new current state by pushing it onto the stack.
  - So we look up entry [0, B] and find the state '4'.
  - We are now in state 4

s	E	B
0	3	4
1		
2		
3		
4		
5		7
6		8
7		
8		

stack
0

$\rightarrow$

stack
4 (B)
0

State 4

in

1	*+*	1	$

s	*	+	0	1	$
0			s1	s2	
1	r4	r4	r4	r4	r4
2	r5	r5	r5	r5	r5
3	s5	s6			acc
4	*r3*	*r3*	*r3*	*r3*	*r3*
5			s1	s2	
6			s1	s2	
7	r1	r1	r1	r1	r1
8	r2	r2	r2	r2	r2

- In state 4 the action table says we do a reduction with rule 3.
  - So we write 3 on the output stream
  - pop one (because there is only one symbol on the RHS of rule 3) state from the stack

4 (B)
0
stack

→

0
stack

State 4: goto table (3) E → B

- After every reduce we go to the goto table, to find the new state for state 0 and E,
  - goto [0, E] = 3
  - which is state 3.
  - A new entry is pushed onto the stack (E,3)

s	*E*	B
0	*3*	4
1		
2		
3		
4		
5		7
6		8
7		
8		

(1) E → E * B

(2) E → E + B

(3) E → B

(4) B → 0

(5) B → 1

0
stack

→

3 (E)
0
stack

State 3

- The current input terminal is a '+' and according to the action table [3, ‘+’] it should then go to state 6. A new entry (‘+’, 6) is therefore pushed.

s	*	+	0	1	$
0			s1	s2	
1	r4	r4	r4	r4	r4
2	r5	r5	r5	r5	r5
3	s5	s6			acc
4	r3	r3	r3	r3	r3
5			s1	s2	
6			s1	s2	
7	r1	r1	r1	r1	r1
8	r2	r2	r2	r2	r2

in

1	+	1	$

3 (E)
0
stack

→

6 (‘+’)
3 (E)
0
stack

in

1	+	1	$

Finite State Automaton

- Note that the resulting stack can be interpreted as the history of a finite state automaton that has just read a non-terminal E followed by a terminal '+'.
  - The transition table of this automaton is defined by the shift actions in the action table and the goto actions in the goto table.

6 (‘+’)
3 (E)
0
stack

State 6

- The next terminal is now '1'. According to action table entry [6, '1'] we should perform a shift and go to state 2.
  - Push a new entry onto the stack ('1',2)

in	1	+	1	$

6 ('+')
3 (E)
0
stack

→

in	1	+	1	$

2 ('1')
6 ('+')
3 (E)
0
stack

s	*	+	0	1	$
0			s1	s2	
1	r4	r4	r4	r4	r4
2	r5	r5	r5	r5	r5
3	s5	s6			acc
4	r3	r3	r3	r3	r3
5			s1	s2	
6			s1	s2	
7	r1	r1	r1	r1	r1
8	r2	r2	r2	r2	r2

Back in State 2

- We've been in state 2 before, so we just go through the same steps.
  - We have to reduce by rule 5.

2 ('1')
6 ('+')
3 (E)
0
stack

s	*	+	0	1	$
0			s1	s2	
1	r4	r4	r4	r4	r4
2	r5	r5	r5	r5	r5
3	s5	s6			acc
4	r3	r3	r3	r3	r3
5			s1	s2	
6			s1	s2	
7	r1	r1	r1	r1	r1
8	r2	r2	r2	r2	r2

(5) B → 1

Reducing by rule 5 again...

- the number m (5) is written to the output stream
- for every symbol in the right-hand side of rule m a state is removed from the stack.
- There is only one symbol on the RHS so we remove a single state.

s	*	+	0	1	$
0			s1	s2	
1	r4	r4	r4	r4	r4
2	r5	r5	r5	r5	r5
3	s5	s6			acc
4	r3	r3	r3	r3	r3
5			s1	s2	
6			s1	s2	
7	r1	r1	r1	r1	r1
8	r2	r2	r2	r2	r2

2 (‘1’)
6 (‘+’)
3 (E)
0
stack

→

6 (‘+’)
3 (E)
0
stack

out	**5**	**3**	**5**	

$(5)\ B \rightarrow 1$

Reducing by rule 5 again…

- given the state that is now on top of the stack (6) and the left-hand side of rule m (5), a new state is looked up in the goto table and made the new current state by pushing it onto the stack.
  - So we look up entry [6, B] and find the state '8'.

s	E	*B*
0	3	4
1		
2		
3		
4		
5		7
6		*8*
7		
8		

6 ('+')
3 (E)
0
stack

→

8 (B)
6 ('+')
3 (E)
0
stack

Finite Automaton Trace

- Again note that the stack corresponds with a list of states of a finite automaton that has read a non-terminal E, followed by a '+' and then a non-terminal B.

8 (B)
6 (‘+’)
3 (E)
0
stack

State 8: reduce by rule 2

- In state 8 we always perform a reduce with rule 2.

s	*	+	0	1	$
0			s1	s2	
1	r4	r4	r4	r4	r4
2	r5	r5	r5	r5	r5
3	s5	s6			acc
4	r3	r3	r3	r3	r3
5			s1	s2	
6			s1	s2	
7	r1	r1	r1	r1	r1
8	*r2*	*r2*	*r2*	*r2*	*r2*

8 (B)
6 (‘+’)
3 (E)
0
stack

State 8: reduce by rule 2   E -> E + B

- the number 2 is written to the output stream
- for every symbol in the right-hand side of rule m a state is removed from the stack.
- There are three symbols on the RHS so we remove 3 states.
- Before we do so, note that the top three states on the stack have symbols that correspond to the 3 symbols in the right-hand side of rule 2.

8 (B)
6 (‘+’)
3 (E)
0
stack

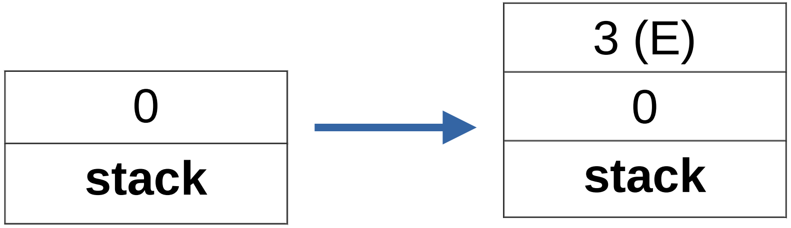
0
stack

out	**5**	**3**	**5**	**2**

State 8: reduce by rule 2   E -> E + B

- given the state that is now on top of the stack (0) and the left-hand side of rule m (2), a new state is looked up in the goto table and made the new current state by pushing it onto the stack.
- So we look up entry [0, E] and find the state '3'.
- We make a new stack entry (E, 3) and push it.

s	E	B
0	3	4
1		
2		
3		
4		
5		7
6		8
7		
8		



Example

in	1	+	1	$

out	5	3	5	2

- Finally, we process the current character '$' from the input stream which means that according to the action table (the current state is 3) the parser accepts the input string.
- The rule numbers that have been written to the output stream are indeed a rightmost derivation of the string "1 + 1" in reverse order.

s	*	+	0	1	$
0			s1	s2	
1	r4	r4	r4	r4	r4
2	r5	r5	r5	r5	r5
3	s5	s6			acc
4	r3	r3	r3	r3	r3
5			s1	s2	
6			s1	s2	
7	r1	r1	r1	r1	r1
8	r2	r2	r2	r2	r2

3 (E)
0
stack

Example

in	*1*	*+*	*1*	*$*

out	5	3	5	2

- Finally, we process the current character '$' from the input stream which means that according to the action table (the current state is 3) the parser accepts the input string.
- The rule numbers that have been written to the output stream are indeed a rightmost derivation of the string "1 + 1" in reverse order.

(1) E → E * B

(2) E → E + B

(3) E → B

(4) B → 0

(5) B → 1

3 (E)
0
stack

Constructing an LR(0) parse tree

Constructing an LR(0) parse tree

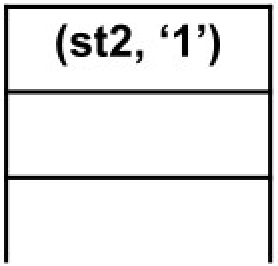
- Put on the stack a record containing the state number and appropriate other information for the parse tree being built.
- The algorithm:
  - whenever we do a “shift” we put on the stack the state number and details of the token recognised (that is the name or symbol table address of an identifier)
  - whenever we do a “reduce” we put on the stack the state number, the identification of the grammar production used (and just for info, the non-terminal on the LHS of the production as well), and a list of pointers to the component parts of the RHS of the production (they’ve just been popped off the stack)

Summary of Stack Actions

step	stack actions	stack
1 : shift 2	push (st2, ‘1’)	(st2, ‘1’)
2 : reduce 5	pop → (st2, ‘1’); push (st4, B)	(st4, B)
3 : reduce 3	pop → (st4, B); push (st3, E)	(st3, E)
4 : shift 6	push (st6, ‘+’)	(st3, E), (st6, ‘+’)
5 : shift 2	push (st2, ‘1’)	(st3, E), (st6, ‘+’), (st2, ‘1’)
6 : reduce 5	pop → (st2, ‘1’); push(st8, B)	(st3, E), (st6, ‘+’),(st8, B)
7: reduce 2	pop → (st8, B); pop → (st6, ‘+’) ; pop-> (st3, E); push (st3, E)	(st3, E)

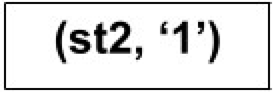
Building a tree: step1: shift 2

- push (st2, ‘1’)



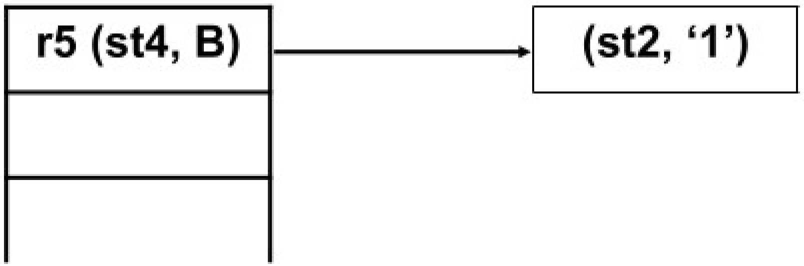
Step2: reduce 5

- pop → (st2, ‘1’);



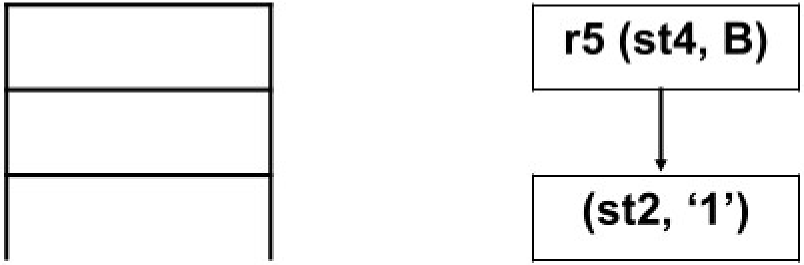
Step2: reduce 5

- push (st4, B)
  - Note that when something is pushed, it becomes the parent node of the entries that have just been popped.



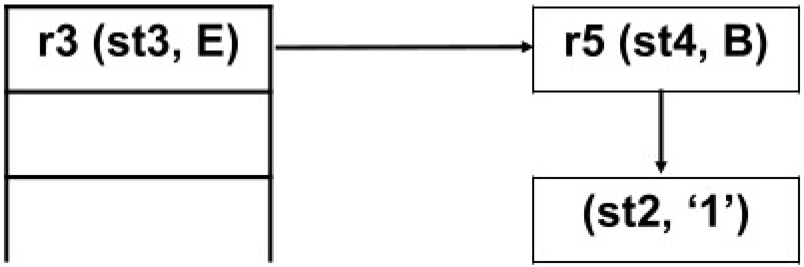
Step3: reduce 3

- pop → (st4, B)



Step3: reduce 3

- push (st3, E)



…some steps later…

- We have a rightmost derivation

$$
\begin{aligned}
E \rightarrow\ & E + B \ (2) \\
& E + 1 \ (5) \\
& B + 1 \ (3) \\
& 1 + 1 \ (5)
\end{aligned}
$$

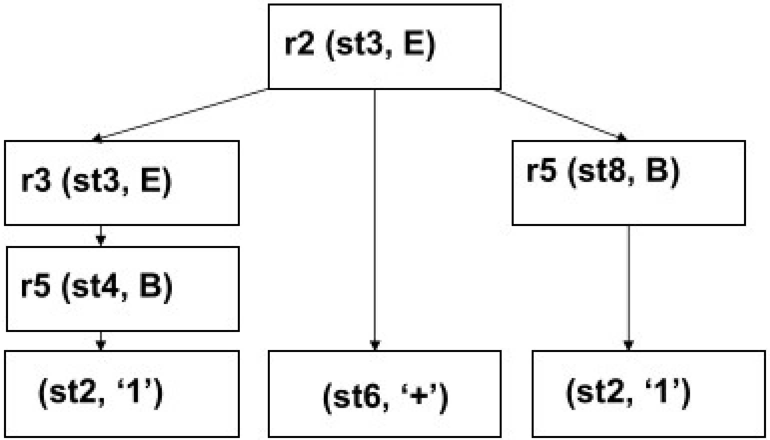
(1) $E \rightarrow E * B$

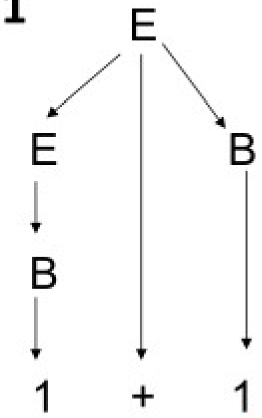
(2) $E \rightarrow E + B$

(3) $E \rightarrow B$

(4) $B \rightarrow 0$

(5) $B \rightarrow 1$





Learning Outcomes

- You should now understand the mechanics of LR(0) parsing
- You should be asking the question “Where do the Action and Goto tables come from??”