

Compiler Construction

SMD163

Lecture 5: Shift-reduce Parsing

Viktor Leijon & Peter Jonsson with slides by Johan Nordlander.
Contains material generously provided by Mark P. Jones



COMPUTER SCIENCE
AND ELECTRICAL ENGINEERING
LULEÅ UNIVERSITY OF TECHNOLOGY

1

Practicalities

- ◆ Now is deadline for second homework assignment.
- ◆ The deadline for the next assignment 8.15 next Monday. *Unless* the book doesn't arrive before Thursday, in which case there'll be an extension.
- ◆ Two labs this week. Wednesday morning and Friday afternoon.

2

Vocabulary – ambiguous

1 a : doubtful or uncertain especially from obscurity or indistinctness <eyes of an *ambiguous* color> **b** : **INEXPLICABLE**

2 : capable of being understood in two or more possible senses or ways

3

Last Time:

- ◆ Parsing;
- ◆ Context-Free Grammars;
- ◆ Parse Trees;
- ◆ Strategies for Parsing.

4

Top-down Parsing:

- ◆ Start looking for a particular nonterminal, pick a production, break up the tokens, continue ...
- ◆ In practice ... we need:
 - To read the tokens from left to right.
 - Rules to help us decide which production to use at each step ...
- ◆ A genuinely useful parsing technique, especially for handwritten parsers for simple grammars.

5

Bottom-up Parsing:

- ◆ Shifts tokens from the input stream onto the parser's stack.
- ◆ Reduces parse tree fragments on the top of the stack that match productions in the grammar.
- ◆ But, in practice, how do we know when to shift and when to reduce?

6

Past and Future:

- ◆ The key is to use information about:
 - What we've already seen;
 - And what we are about to see ...
- ◆ The more information we use, the better our decisions will be ...
- ◆ ... but the more we'll have to pay.
- ◆ In practice, our goal is to find a compromise: reliable decisions, but relatively inexpensive.

7

A Language of Expressions:

- ◆ An even simpler language of expressions:
 - $E \rightarrow n$ (n is an integer literal)
 - $E \rightarrow E + E$
 - $E \rightarrow (E)$

8

LR(0) Items:

- ◆ An item is a production with a marked position on its right hand side.
- ◆ Example: if $N \rightarrow w_1w_2$ is a production, then $N \rightarrow w_1_w_2$ is an item.
- ◆ Items correspond to intermediate steps during parsing.
- ◆ For each production $N \rightarrow w$ with n symbols on the right hand side, there are $(n+1)$ items.

9

For Example:

- ◆ The items for our language of expressions:

$E \rightarrow _n$	$E \rightarrow _ (E)$
$E \rightarrow n_$	$E \rightarrow (_ E)$
$E \rightarrow _ E + E$	$E \rightarrow (E _)$
$E \rightarrow E _ + E$	$E \rightarrow (E) _$
$E \rightarrow E + _ E$	
$E \rightarrow E + E _$	

10

Augmented Grammars:

Given a context-free grammar:

$$N_1 \rightarrow w_1$$

...

we can construct an augmented grammar:

$$S \rightarrow N_1 \$$$
$$N_1 \rightarrow w_1$$

...

where:

- $\$$ is a new terminal symbol;
- S is a new nonterminal symbol.

11

For Example:

- ◆ The augmented grammar for our language of expressions has one extra production:

$$S \rightarrow E \$$$

- ◆ ... and two extra items:

$$S \rightarrow _ E \$$$
$$S \rightarrow E _ \$$$

- ◆ (Technically, there is a third item, $S \rightarrow E \$ _$, but we won't actually need it ...)

12

Or, Marking the End:

- ◆ A simple change that:
 - Allows us to talk about “the next token”, without having to make a special case for the end of the token stream;
 - Makes parsing easier to implement.
- ◆ In practice, we just have to mark the end of the input stream:

1 + (2 + 3) \$

13

And to Begin ...

The initial stage in parsing our augmented grammar can be described by the item:

$$S \rightarrow _E \$$$

14

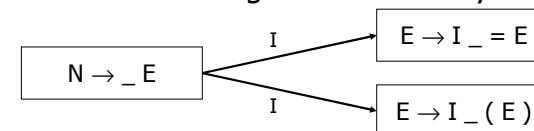
Dealing with Uncertainty:

- ◆ Consider the following two Java expressions:
`life = 42` and `life(42)`
- ◆ The grammar for a simplified Java might include:
 - $E \rightarrow I = E$ (I is an identifier)
 - $E \rightarrow I (E)$
- ◆ Suppose that we are in a state described by an item:
 - $N \rightarrow _ E$
 and that the next input token is an identifier...?

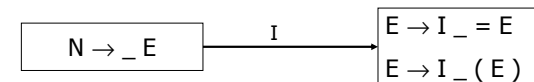
15

Using Item Sets:

- ◆ Instead of making a decision early: *Familiar?*



- ◆ Use sets of items to delay the decision:



- ◆ And now we can tell which way to go by looking at the next character ...

16

To Begin:

$S \rightarrow _ E \$$

1	+	(2	+	3)	\$
---	---	---	---	---	---	---	----

Here is the input token stream ...

Boxes represent "states", and are described by sets of items.

17

Closure:

$S \rightarrow _ E \$$
 $E \rightarrow _ E + E$
 $E \rightarrow _ (E)$
 $E \rightarrow _ n$

1	+	(2	+	3)	\$
---	---	---	---	---	---	---	----

We calculate the closure of a set of items I as follows:

While there is an item $(N \rightarrow w_N'w')$ in I and an item $(N' \rightarrow _v)$ not in I add $(N' \rightarrow _v)$ to I.

The result is called the closure of I.

18

Transitions:

$S \rightarrow _ E \$$
 $E \rightarrow _ E + E$
 $E \rightarrow _ (E)$
 $E \rightarrow _ n$

(

$E \rightarrow _ n$

If there is an item $(N \rightarrow w_t w)$ in I, then we can make a transition from that state when we see the symbol t.

The state that we make a transition to will include the item: $(N \rightarrow w t _ w)$

19

Shift!

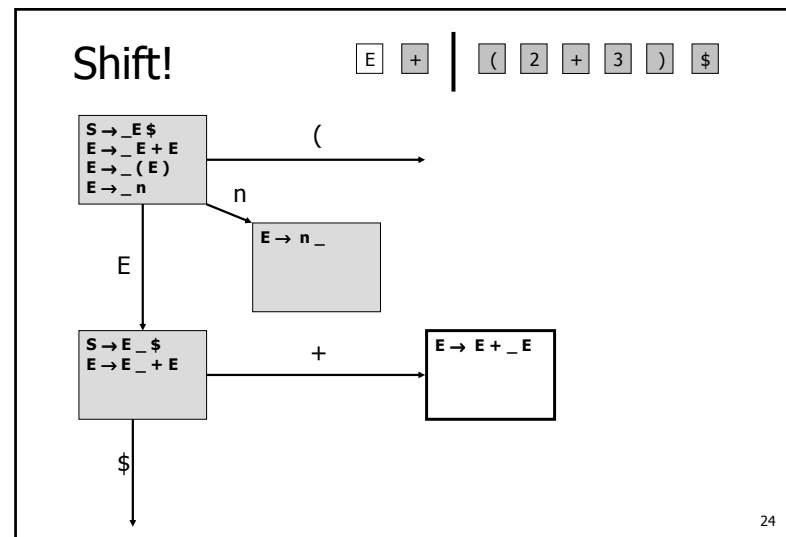
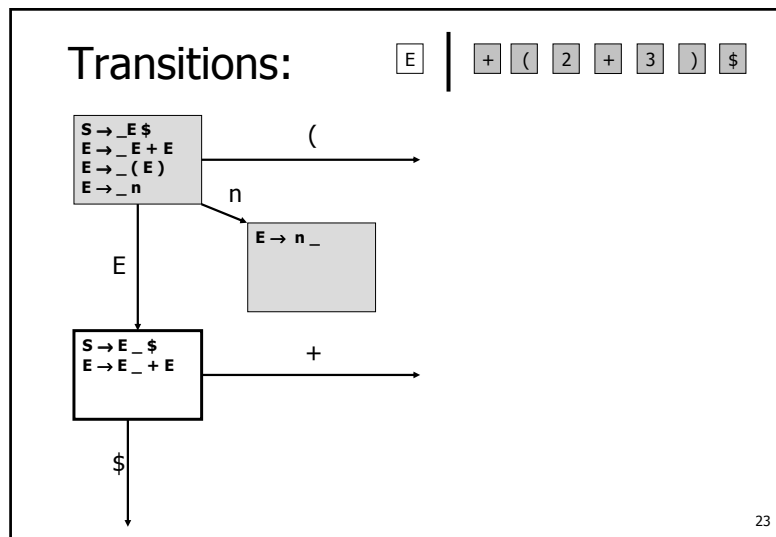
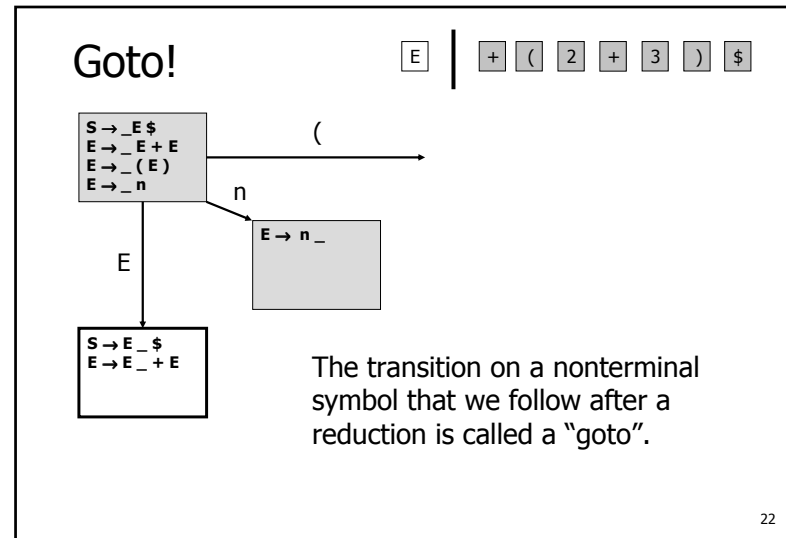
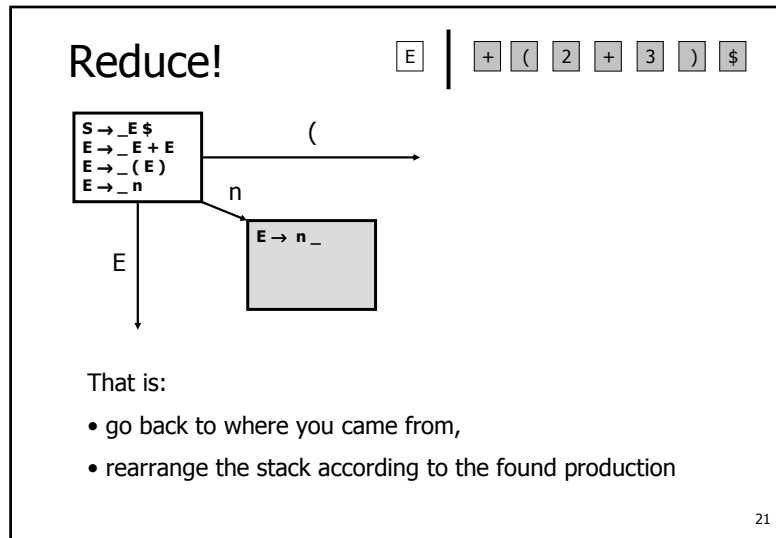
$S \rightarrow _ E \$$
 $E \rightarrow _ E + E$
 $E \rightarrow _ (E)$
 $E \rightarrow _ n$

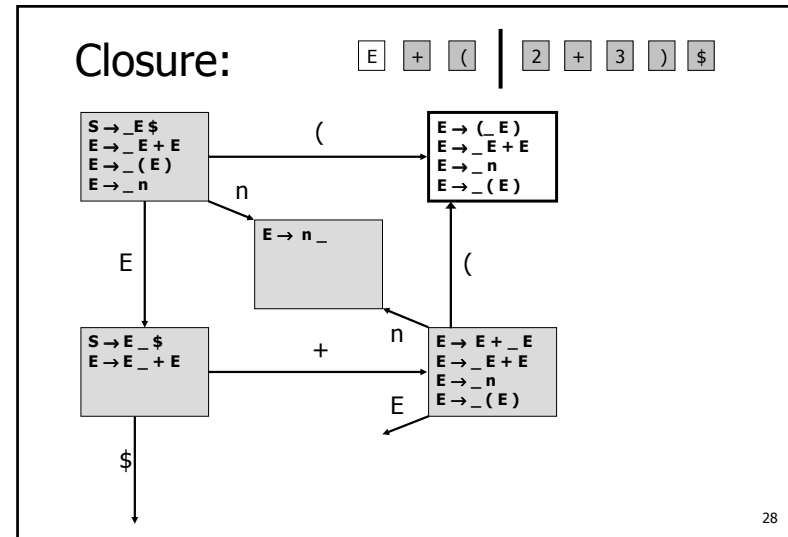
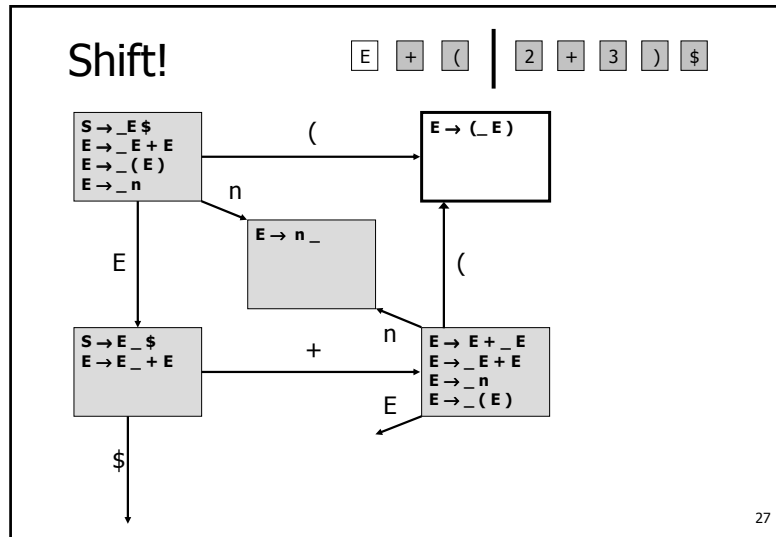
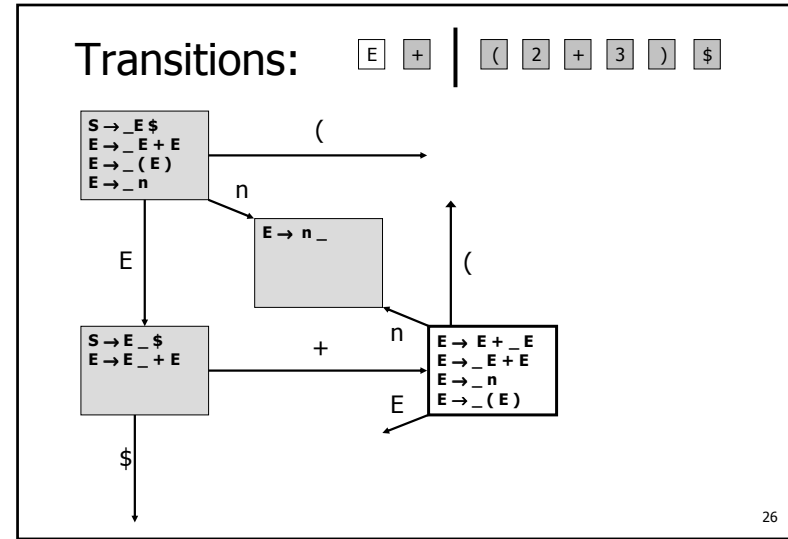
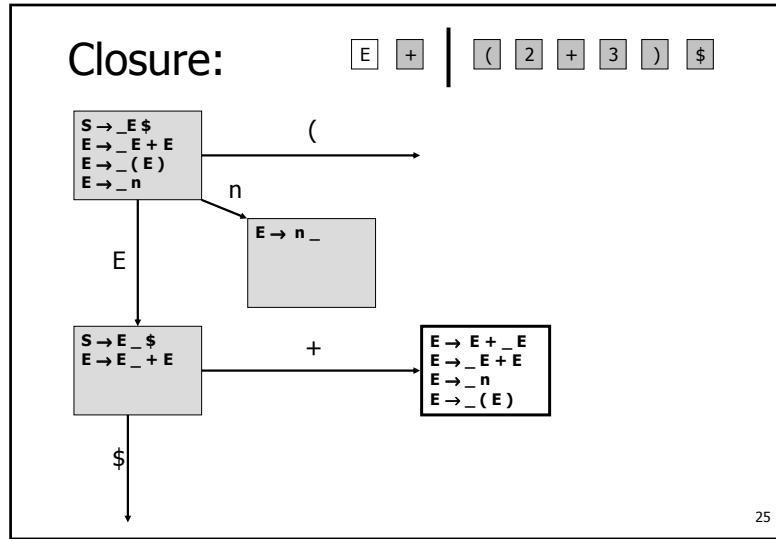
(

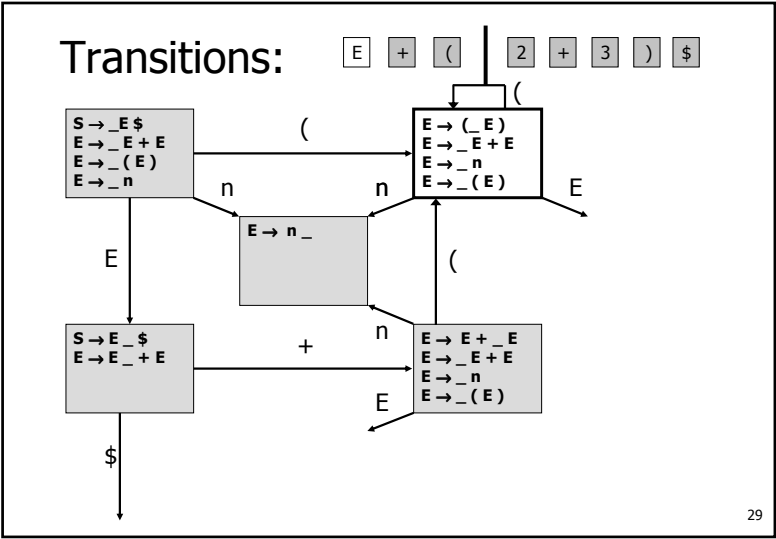
$E \rightarrow n _$

Nothing more to add to the closure
And the marker is at the end of the production...

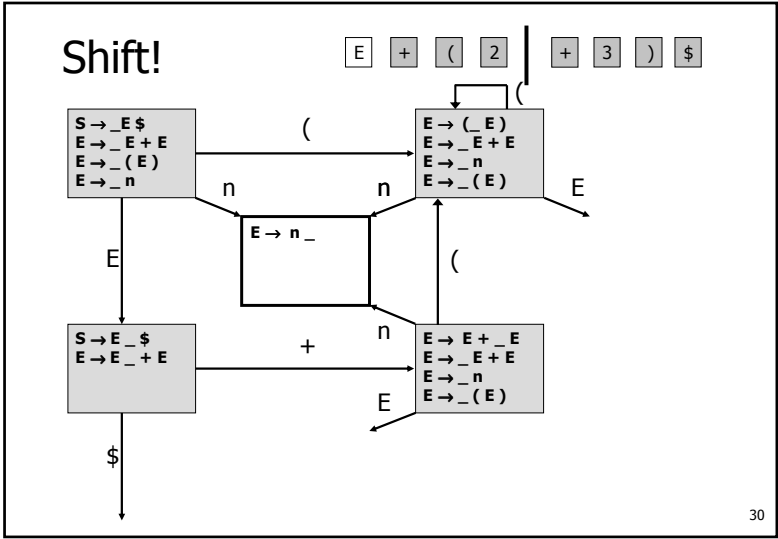
20



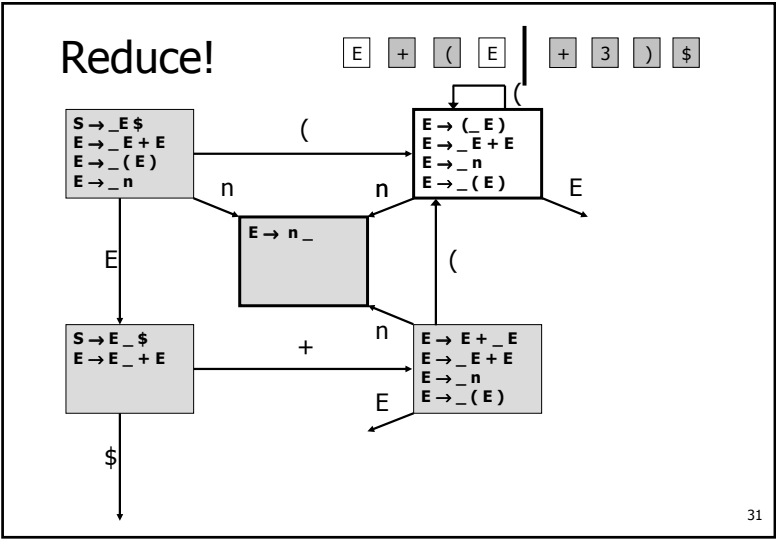




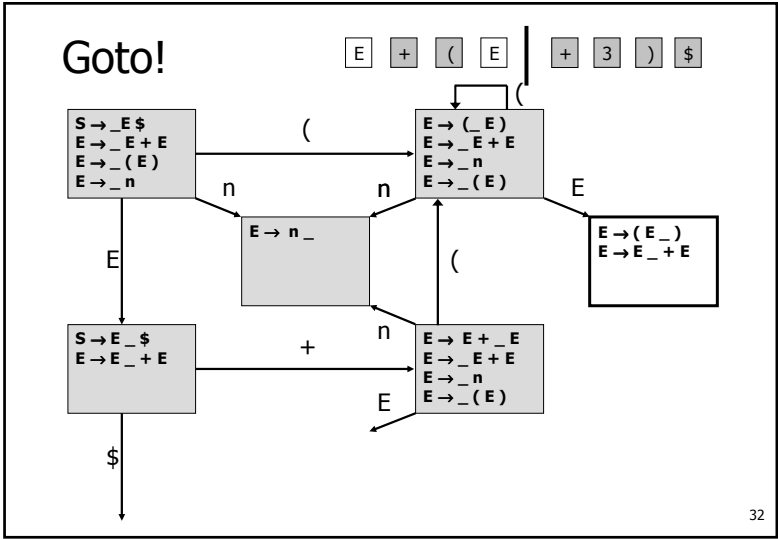
29



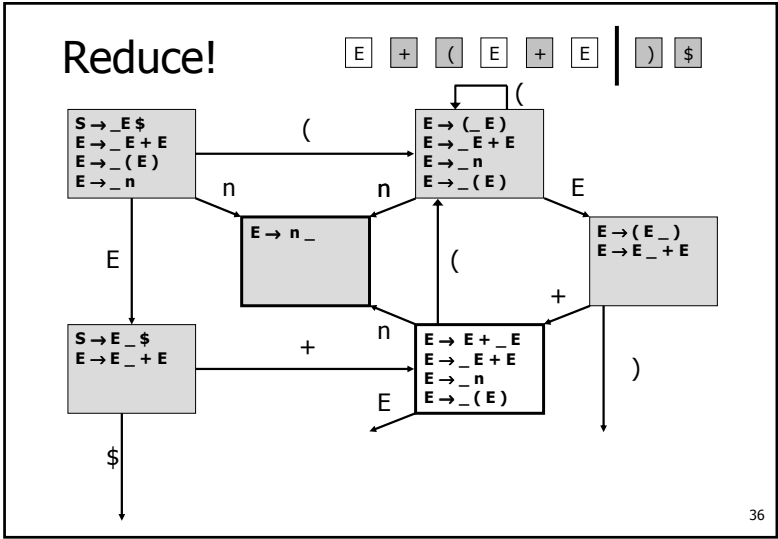
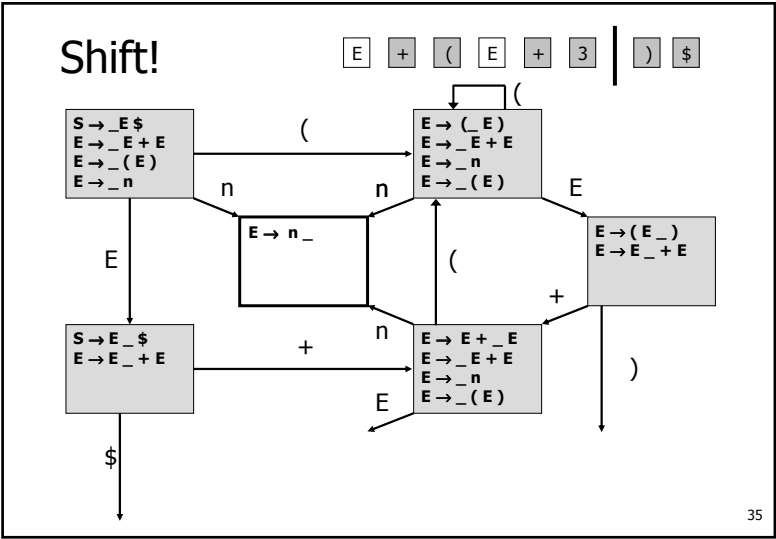
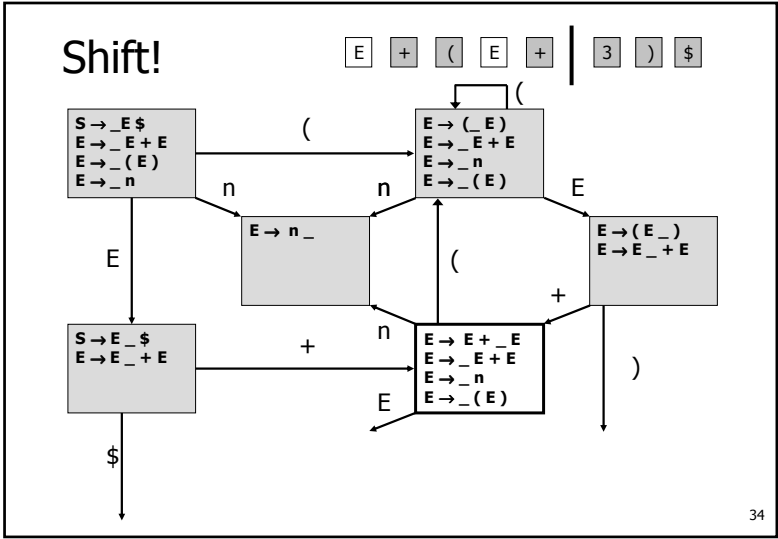
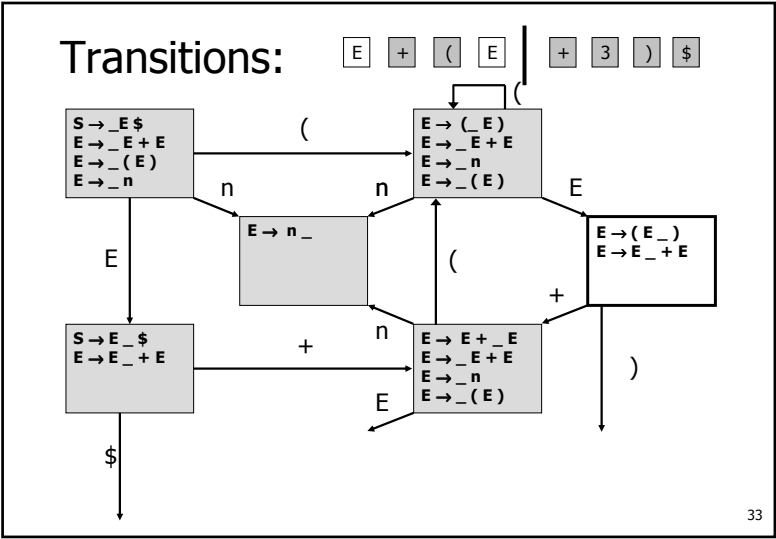
30

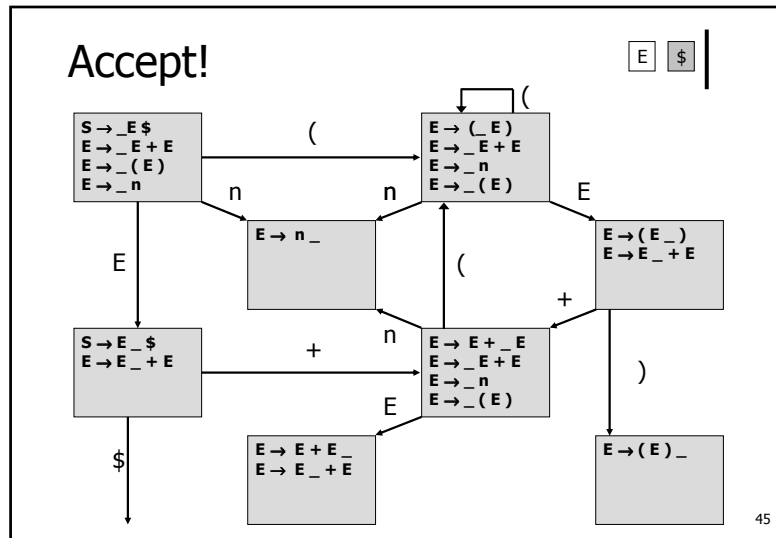


31



32

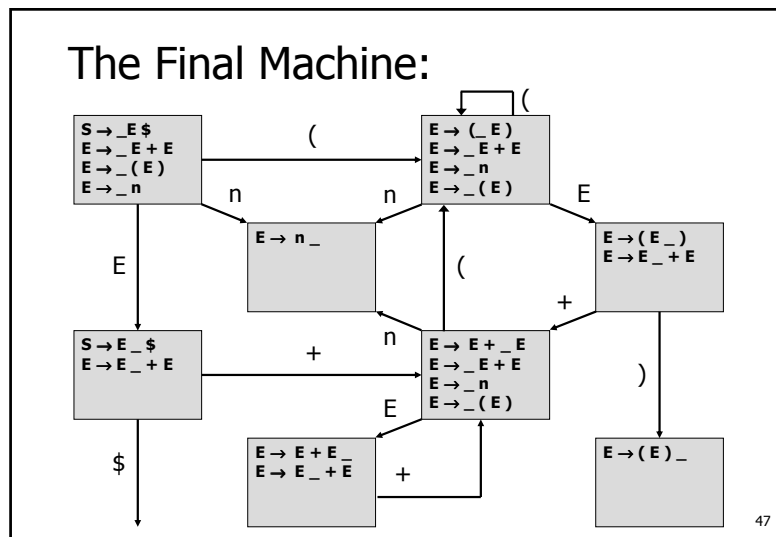




A Recognizer for Expressions:

- ◆ We can describe a recognizer for our language of expressions using a finite-state machine.
 - The machine uses a stack to hold parse tree fragments. (The semantic stack)
 - The current path through the machine is also described by a stack. (The state stack)
 - The input stream is read one token at a time, left to right.
- ◆ We could do the same thing for any context-free grammar...

46

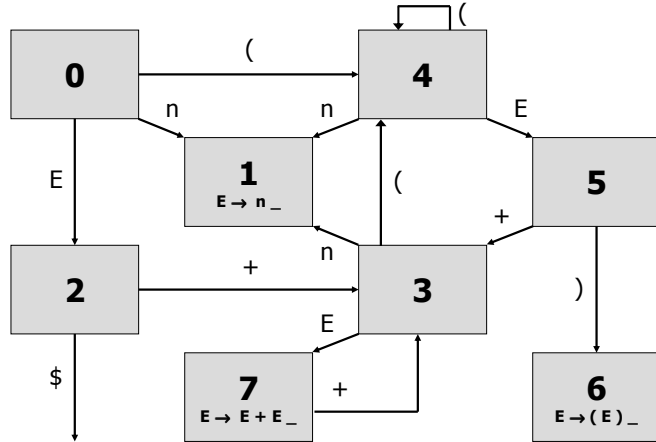


Implementing the Recognizer:

- ◆ The sets of items were used to help us calculate the machine ... they are not actually needed at parse-time.
- ◆ The machine can be implemented:
 - By generating executable (spaghetti) code.
 - By describing the machine as a table and using a generic, table-driven routine.

48

The Final Machine:



49

A Parse Table:

State #	Terminals					Reduce	Goto
	+	(n)	\$		
0		s4	s1				2
1						$E \rightarrow n$	
2	s3				A		
3		s4	s1				7
4		s4	s1				5
5	s3				s6		
6						$E \rightarrow (E)$	
7	s3					$E \rightarrow E + E$	

50

A Parse Table:

State #	Terminals					Reduce	Goto
	+	(n)	\$		
0		s4	s1				2
1						$E \rightarrow n$	
2	s3				A		
3		s4	s1				7
4		s4	s1				5
5	s3				s6		
6						$E \rightarrow (E)$	
7	s3					$E \rightarrow E + E$	

51

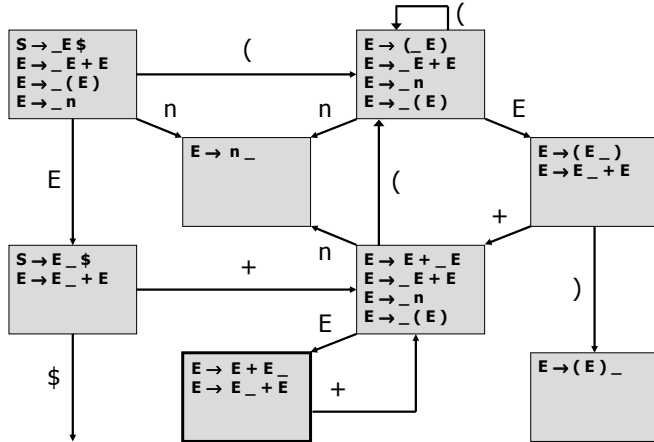
A Parse Table:

State #	Terminals					Reduce	Goto
	+	(n)	\$		
0		s4	s1				2
1							
2	s3				A		
3		s4	s1				7
4		s4	s1				5
5	s3				s6		
6						$E \rightarrow (E)$	
7	s3					$E \rightarrow E + E$	

There is a shift/reduce conflict

52

Recall the Final Machine:



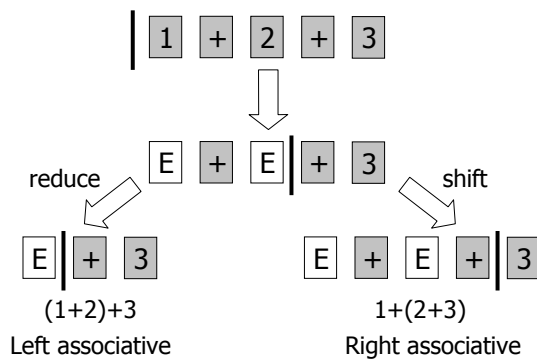
53

Conflicts:

- ◆ A conflict occurs when there are two possible ways for the machine to react in a given state.
- ◆ Conflicts come in two forms:
 - Shift/reduce;
 - Reduce/reduce.
- ◆ And if we have conflicts, then we can't be sure we'll make the right decisions at each step.

54

Our shift/reduce Conflict:



55

Ambiguity Causes Conflicts:

- ◆ If we start with an ambiguous grammar, there will be conflicts.
- ◆ Often, we can use disambiguating rules to resolve conflicts. Typical policies might include:
 - Use operator precedence and associativity, if provided;
 - If all else fails, favor shift over reduce.
 - Etc...

56

LR(0):

- ◆ If the machine has no conflicts, then we say that the grammar is LR(0).
- ◆ LR(0) stands for:
 - Left-to-right,
 - Right-most derivation (in reverse),
 - 0 characters lookahead.
- ◆ The set of languages that can be described by LR(0) grammars is also called LR(0).
- ◆ But most programming language grammars are not LR(0) ...

57

Lookahead:

Don't reduce unconditionally...

First check the next input token(s) to see if reduction is at all sensible!

58

Using Lookahead:

$S' \rightarrow D ; B ;$

$B \rightarrow D$

$D \rightarrow B !$

$D \rightarrow x$

New grammar – let's construct it's recognizer!

59

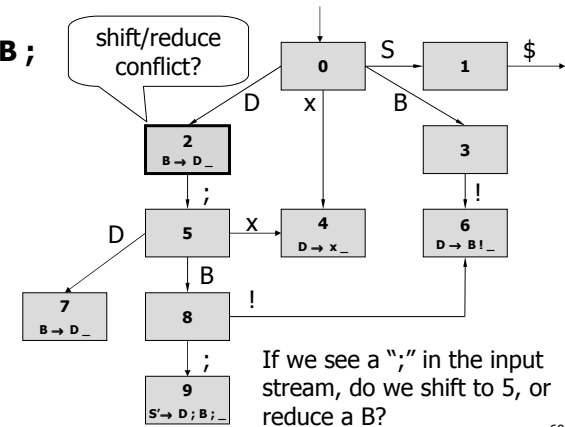
Using Lookahead:

$S' \rightarrow D ; B ;$

$B \rightarrow D$

$D \rightarrow B !$

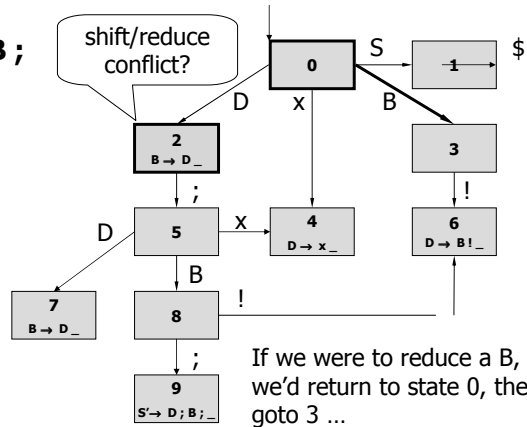
$D \rightarrow x$



60

Using Lookahead:

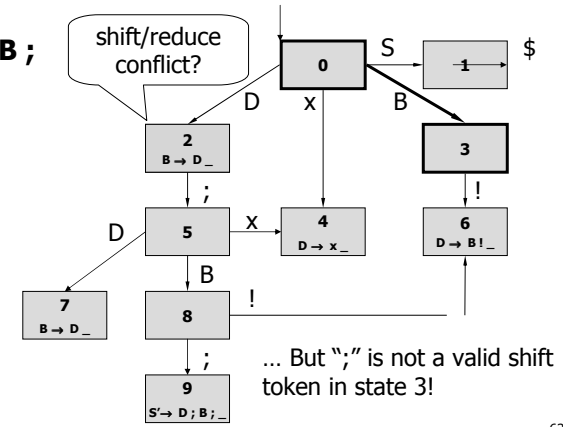
$S' \rightarrow D ; B ;$
 $B \rightarrow D$
 $D \rightarrow B !$
 $D \rightarrow x$



61

Using Lookahead:

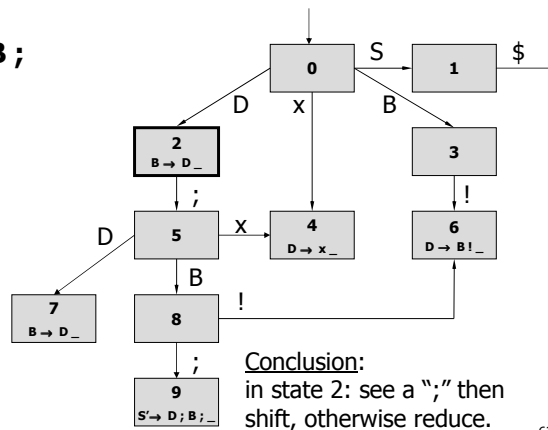
$S' \rightarrow D ; B ;$
 $B \rightarrow D$
 $D \rightarrow B !$
 $D \rightarrow x$



62

Using Lookahead:

$S' \rightarrow D ; B ;$
 $B \rightarrow D$
 $D \rightarrow B !$
 $D \rightarrow x$

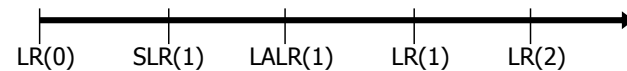


63

How Much Lookahead to Use?

- ◆ We can use lookahead to resolve conflicts.
 - But how much lookahead, and how precise do we want the lookahead to be?

- ◆ There are many choices:



- ◆ Each of these points corresponds to a different strategy for resolving conflicts.
- ◆ For example, we say that a grammar is LR(1) if the LR(1) strategy resolves all conflicts.

64

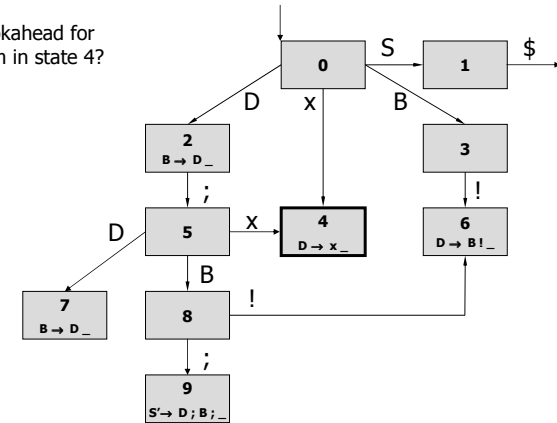
LALR(1)

- ◆ SLR(1) calculates a lookahead set that is independent of the current state
- ◆ LALR(1) calculates one lookahead set for item $(N \rightarrow w_)$ in state i by looking at all the gotos that we can reach by backtracking along a path from i labeled with w
- ◆ LR(1) calculates a different lookahead for each path...
- ◆ LR(2) calculates a two symbol lookahead for each path...
- ◆ For practical purposes, LALR(1) is a good compromise.

65

For example:

What is the lookahead for the reduce item in state 4?

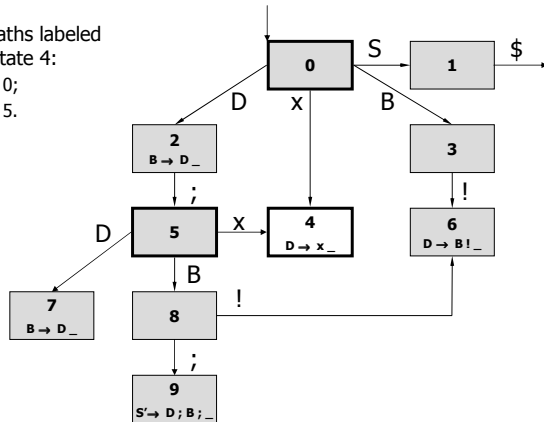


66

For example:

There are two paths labeled x that arrive in state 4:

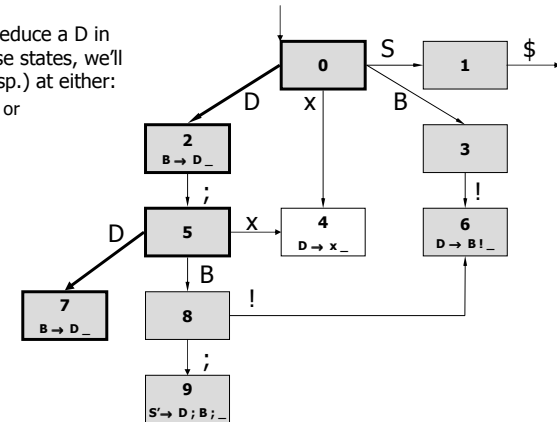
- One from 0;
- One from 5.



67

For example:

- ◆ When we reduce a D in one of those states, we'll end up (resp.) at either:
 - State 2; or
 - State 7.

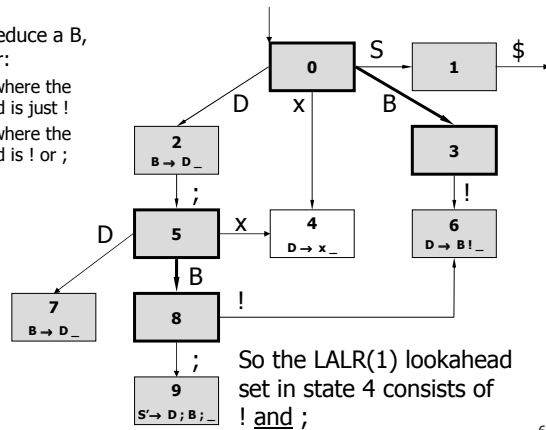


68

For example:

And then we'll reduce a B, arriving at either:

- State 3, where the lookahead is just !
- State 8, where the lookahead is ! or ;



69

Summary:

- ◆ Shift-reduce parsing is an effective technique for bottom-up parsing.
- ◆ Conflicts can occur, and must be resolved in some appropriate manner.
- ◆ The machines that we build can have 100s of states, and calculating lookaheads can be tricky ...
- ◆ ... but these tasks can be automated!

70