Menoufia University

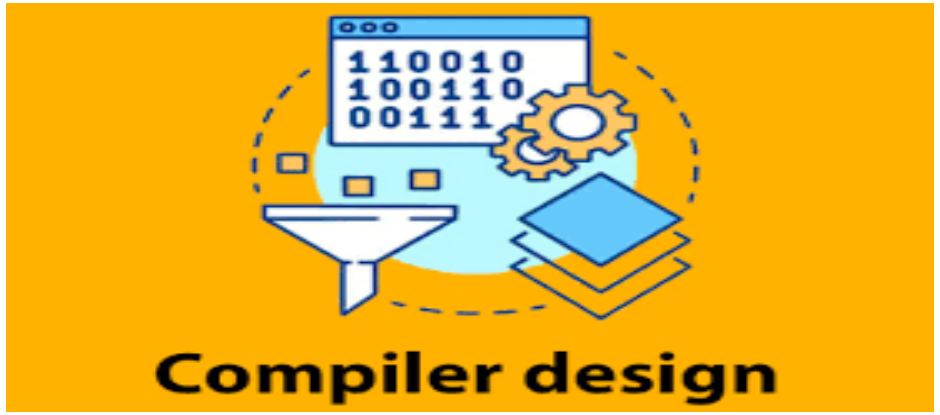
Faculty of computers & Information

Computer Science Department.

Compiler Design

4 Year – first Semester

Lecture 7



DR. Eman Meslhy Mohamed

Lecturer at Computer Science department

2023-2024

Bottom Up Parsing

- Parsing algorithms which proceed from the bottom of the derivation tree and apply grammar rules (in reverse) are called ***Bottom up parsing algorithms*** .

- First, we will begin with an empty stack.

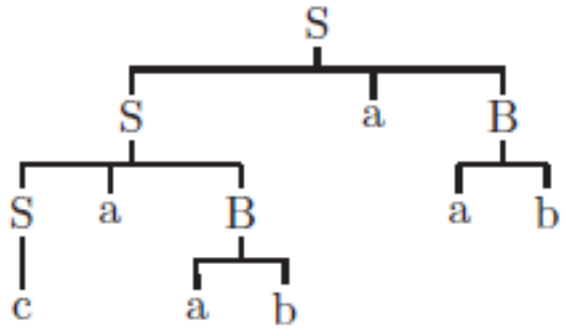
- Then, one or more input symbols are moved onto the stack, which are then replaced by nonterminals according to the grammar rules.

- The input string is acceptable If all the input symbols have been read, and the algorithm terminates with the starting nonterminal alone on the stack.

Shift Reduce Parsing

- Bottom up parsing involves two fundamental operations.
  - The process of moving an input symbol to the stack is called a ***shift*** operation,
  - and the process of replacing symbols on the top of the stack with a nonterminal is called a ***reduce*** operation
- A derivation tree for the string caabaab using the following grammar.

$S \rightarrow S\ a\ B$
$S \rightarrow c$
$B \rightarrow a\ b$

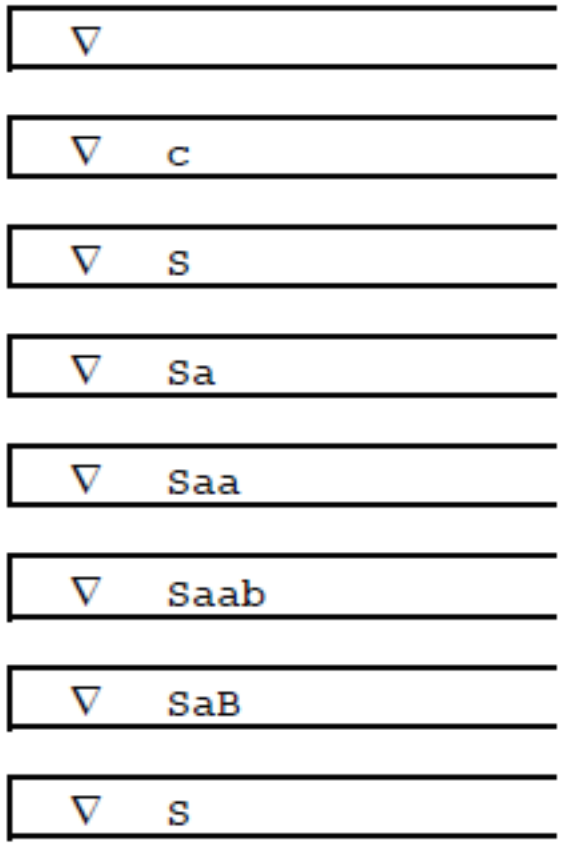


$$S \Rightarrow \underline{SaB} \Rightarrow Sa\underline{ab} \Rightarrow \underline{SaBaab} \Rightarrow Sa\underline{abaab} \Rightarrow \underline{c}aabaab$$

Example

- Show the sequence of stack and input configurations as the string caab

$S \rightarrow S\ a\ B$
$S \rightarrow c$
$B \rightarrow a\ b$

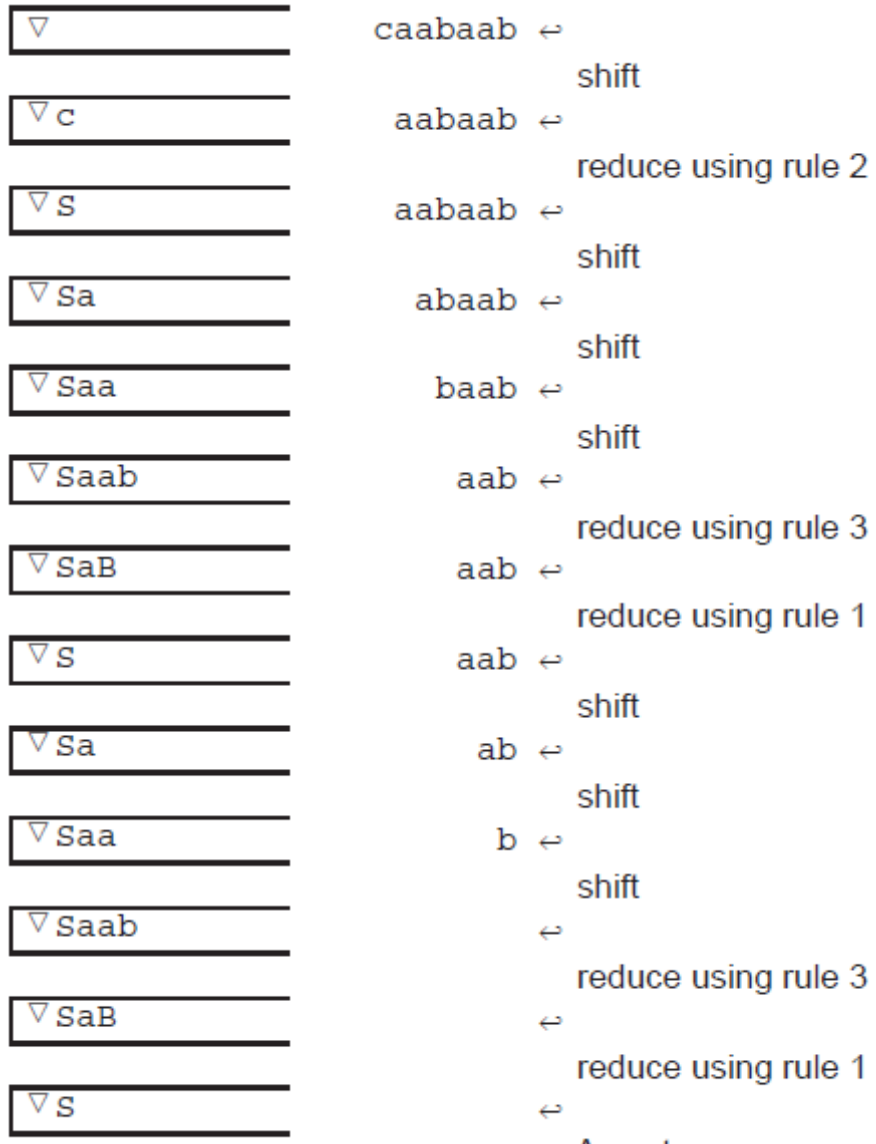


caab ↲
shift
aab ↲
reduce using rule 2
aab ↲
shift
ab ↲
shift
b ↲
shift
↲
reduce using rule 3
↲
reduce using rule 1
↲
Accept

Example

- Show the sequence of stack and input configurations as the string caabaab

$S \rightarrow S\ a\ B$
$S \rightarrow c$
$B \rightarrow a\ b$



LR Grammar

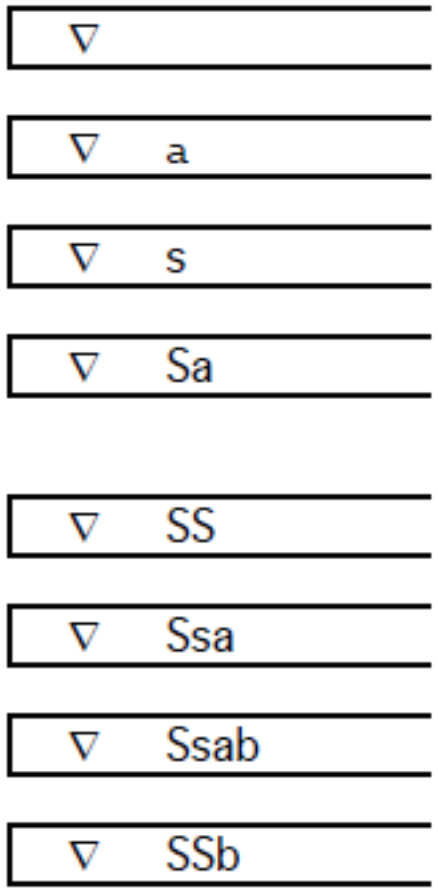
- If the grammar can be implemented with a shift reduce algorithm, we say the grammar is ***LR*** (the L indicates we are reading input from the ***left***, and the R indicates we are finding a ***rightmost*** derivation).

- **The grammar is not LR, if**
  - **Shift/reduce conflict**
  - **reduce/reduce conflict**
  - **Ambiguous grammar**

Shift/Reduce Conflict

- The parser does not know whether to shift an input symbol or reduce the handle on the stack.
- For example, Show the sequence of stack and input configurations as the string aaab

1. **S → S a B**
2. **S → a**
3. **B → a b**



aaab ↲	
	shift
aab ↲	
	reduce using rule 2
aab ↲	
	shift
ab ↲	
	shift/reduce conflict reduce using rule 2 (incorrect)
ab ↲	
	shift
b ↲	
	shift
↲	
	reduce using rule 3
↲	
	Syntax error (incorrect)

Reduce/Reduce Conflict

- It is clear that a reduce operation should be performed,
- But, there is more than one grammar rule whose right hand side matches the top of the stack, and it is not clear which rule should be used.
- For example, Show the sequence of stack and input configurations as the string aa

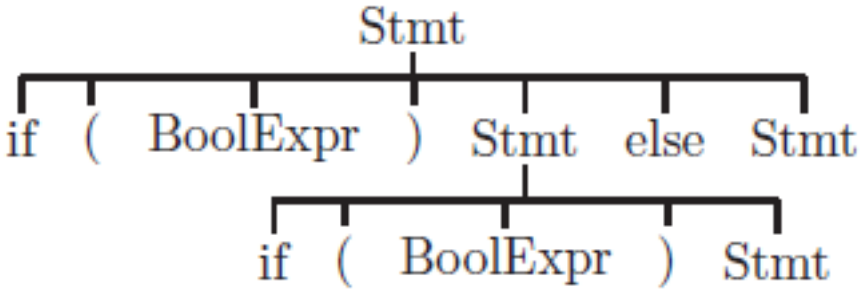
$S \rightarrow SA$
$S \rightarrow a$
$A \rightarrow a$

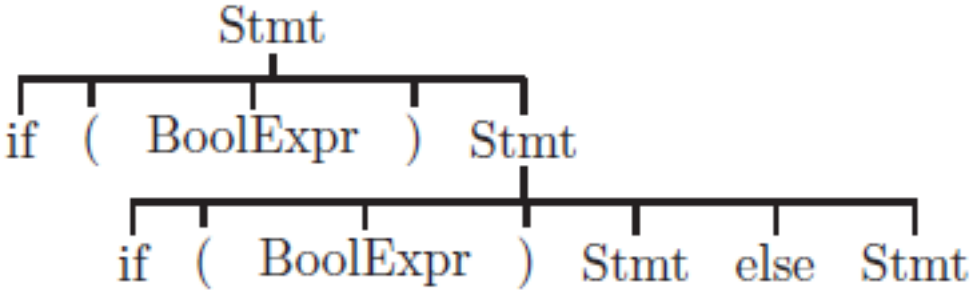
∇	aa ↵	shift
∇ a	a ↵	reduce/reduce conflict (rules 2 and 3) reduce using rule 3 (incorrect)
∇ A	a ↵	shift
∇ Aa	↵	reduce/reduce conflict (rules 2 and 3) reduce using rule 2 (rule 3 will also yield a syntax error)
∇ AS	↵	Syntax error

Ambiguous grammar

- Ambiguous grammar will always produce conflicts when parsing bottom up with the shift reduce algorithm.
- The following figure shows two different derivation trees for the statement if (BoolExpr) if (BoolExpr) Stmt else Stmt.

1. Stmt → if (BoolExpr) Stmt else Stmt
2. Stmt → if (BoolExpr) Stmt





Stack

Input

∇ . . . if (BoolExpr) Stmt

else . . . ↲

LR Parsing With Tables

- One way to implement shift reduce parsing is with tables that determine whether to shift or reduce, and which grammar rule to reduce.
- This method makes use of two tables to control the parser.
- The first table, called the ***action table***, determines whether a shift or reduce is to be invoked.
- If it specifies a reduce, it also indicates which grammar rule is to be reduced.
- The second table, called a ***goto table***, indicates which stack symbol is to be pushed on the stack after a reduction.

action table and goto table

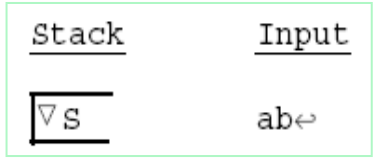
1. Expr → Expr + Term
2. Expr → Term
3. Term → Term * Factor
4. Term → Factor
5. Factor → (Expr)
6. Factor → var

	Goto Table		
	Expr	Term	Factor
▽	push Expr1	push Term2	push Factor4
Expr1			
Term1			
Factor3			
(	push Expr5	push Term2	push Factor4
Expr5			
)			
+		push Term1	push Factor4
Term2			
*			push Factor3
Factor4			
var			

	Action Table					
	+	*	(	)	var	↵
▽			shift (		shift var	
Expr1	shift +					Accept
Term1	reduce 1	shift *		reduce 1		reduce 1
Factor3	reduce 3	reduce 3		reduce 3		reduce 3
(			shift (		shift var	
Expr5	shift +			shift)		
)	reduce 5	reduce 5		reduce 5		reduce 5
+			shift (		shift var	
Term2	reduce 2	shift *		reduce 2		reduce 2
*			shift (		shift var	
Factor4	reduce 4	reduce 4		reduce 4		reduce 4
var	reduce 6	reduce 6		reduce 6		reduce 6

The operation of the LR parser can be described as follows:

1. Find the action corresponding to the current input and the top stack symbol.
2. If that action is a shift action:
   a. Push the input symbol onto the stack.
   b. Advance the input pointer.
3. If that action is a reduce action:
   a. Find the grammar rule specified by the reduce action.
   b. The symbols on the right side of the rule should also be on the top of the stack
      - pop them all off the stack.
   c. Use the nonterminal on the left side of the grammar rule to indicate a column of the goto table, and use the top stack symbol to indicate a row of the goto table.
      - Push the indicated stack symbol onto the stack.
   d. Retain the input pointer.
4. If that action is blank, a syntax error has been detected.
5. If that action is Accept, terminate.
6. Repeat from step 1.



B → Sa

Example

Show the sequence of stack, input, action, and goto configurations for the input var*var

1. Expr → Expr + Term
2. Expr → Term
3. Term → Term * Factor
4. Ter
5. Fac
6. Fac

Stack	Input	Action	Goto
▽	var*var ↩		
		shift var	
▽ var	*var ↩		
		reduce 6	push Factor4
▽ Factor4	*var ↩		
		reduce 4	push Term2
▽ Term2	*var ↩		
		shift *	
▽ Term2*	var ↩		
		shift var	
▽ Term2*var	↩		
		reduce 6	push Factor3
▽ Term2*Factor3	↩		
		reduce 3	push Term2
▽ Term2	↩		
		reduce 2	push Expr1
▽ Expr1	↩		
		Accept	

	Action Table					
	+	*	(	)	var	↩
▽			shift (		shift var	
Expr1	shift +					Accept
Term1	reduce 1	shift *		reduce 1		reduce 1
Factor3	reduce 3	reduce 3		reduce 3		reduce 3
(			shift (		shift var	
Expr5	shift +			shift)		
)	reduce 5	reduce 5		reduce 5		reduce 5
+			shift (		shift var	
Term2	reduce 2	shift *		reduce 2		reduce 2
*			shift (		shift var	
Factor4	reduce 4	reduce 4		reduce 4		reduce 4
var	reduce 6	reduce 6		reduce 6		reduce 6

	GoTo Table		
	Expr	Term	Factor
▽	push Expr1	push Term2	push Factor4
Expr1			
Term1			
Factor3			
(	push Expr5	push Term2	push Factor4
Expr5			
)			
+		push Term1	push Factor4
Term2			
*			push Factor3
Factor4			
var			

Show the sequence of stack, input, action, and goto configurations for the input (var+var)*var

1. Expr → Expr + Term
2. Expr → Term
3. Term → Term * Factor
4. Term → Factor
5. Factor → (Expr)
6. Factor → var

	Goto Table		
	Expr	Term	Factor
▽	push Expr1	push Term2	push Factor4
Expr1			
Term1			
Factor3			
(	push Expr5	push Term2	push Factor4
Expr5			
)			
+		push Term1	push Factor4
Term2			
*			push Factor3
Factor4			
var			

	Action Table					
	+	*	(	)	var	↩
▽			shift (		shift var	
Expr1	shift +					Accept
Term1	reduce 1	shift *		reduce 1		reduce 1
Factor3	reduce 3	reduce 3		reduce 3		reduce 3
(			shift (		shift var	
Expr5	shift +			shift)		
)	reduce 5	reduce 5		reduce 5		reduce 5
+			shift (		shift var	
Term2	reduce 2	shift *		reduce 2		reduce 2
*			shift (		shift var	
Factor4	reduce 4	reduce 4		reduce 4		reduce 4
var	reduce 6	reduce 6		reduce 6		reduce 6

Example

Show the sequence of stack, input, action, and goto configurations for the input (var+var)*var

1. Expr → Expr + Term
2. Expr → Term
3. Term → Term * Factor
4. Term → Factor
5. Factor → (Expr)
6. Factor → var

Action Table

	+	*	(	)	var	↩
▽			shift (		shift var	
Expr1	shift +					Accept
Term1	reduce 1	shift *		reduce 1		reduce 1
Factor3	reduce 3	reduce 3		reduce 3		reduce 3
(			shift (		shift var	
Expr5	shift +			shift)		
)	reduce 5	reduce 5		reduce 5		reduce 5
+			shift (		shift var	
Term2	reduce 2	shift *		reduce 2		reduce 2
*			shift (		shift var	
Factor4	reduce 4	reduce 4		reduce 4		reduce 4
var	reduce 6	reduce 6		reduce 6		reduce 6

GoTo Table

	Expr	Term	Factor
▽	push Expr1	push Term2	push Factor4
Expr1			
Term1			
Factor3			
(	push Expr5	push Term2	push Factor4
Expr5			
)			
+		push Term1	push Factor4
Term2			
*			push Factor3
Factor4			
var			

Stack	Input	Action	Goto
▽	(var+var)*var ↩		
		shift (	
▽ (	var+var)*var ↩		
		shift var	
▽ (var	+var)*var ↩		
		reduce 6	push Factor4
▽ (Factor4	+var)*var ↩		
		reduce 4	push Term2
▽ (Term2	+var)*var ↩		
		reduce 2	push Expr5
▽ (Expr5	+var)*var ↩		
		shift +	
▽ (Expr5+	var)*var ↩		
		shift var	
▽ (Expr5+var	)*var ↩		
		reduce 6	push Factor4
▽ (Expr5+Factor4	)*var ↩		
		reduce 4	push Term1
▽ (Expr5+Term1	)*var ↩		
		reduce 1	push Expr5
▽ (Expr5	)*var ↩		
		shift)	
▽ (Expr5)	*var ↩		
		reduce 5	push Factor4
▽ Factor4	*var ↩		
		reduce 4	push Term2
▽ Term2	*var ↩		
		shift *	
▽ Term2*	var ↩		
		shift var	
▽ Term2*var	↩		
		reduce 6	push Factor3
▽ Term2*Factor3	↩		
		reduce 3	push Term2
▽ Term2	↩		
		reduce 2	push Expr1
▽ Expr1	↩		
		Accept	

Thanks