Lecture 9

Syntax Analysis

Awanish Pandey

Department of Computer Science and Engineering
Indian Institute of Technology
Roorkee

February 7, 2025

Take aways from the last class

- Calculation of *follow* set

Take aways from the last class

- Calculation of *follow* set
- Creation of parse table

Take aways from the last class

- Calculation of *follow* set
- Creation of parse table
- Error recovery technique.

Take aways from the last class

- Calculation of *follow* set
- Creation of parse table
- Error recovery technique.
- Bottom up parsing

Take aways from the last class

- Calculation of *follow* set
- Creation of parse table
- Error recovery technique.
- Bottom up parsing
- Shift-Reduce parser

Example

Example

- Assume grammar is $E \rightarrow E + E | E * E | id$

Example

- Assume grammar is $E \rightarrow E + E | E * E | id$
- Parse id*id+id

Example

- Assume grammar is $E \rightarrow E + E|E * E|id$
- Parse id*id+id

String	Action

Example

- Assume grammar is $E \rightarrow E + E | E * E | id$
- Parse id*id+id

String	Action
.id*id+id	shift

Example

- Assume grammar is $E \rightarrow E + E | E * E | id$
- Parse id*id+id

String	Action
.id*id+id	shift
id.*id+id	reduce $E \rightarrow id$

Example

- Assume grammar is $E \rightarrow E + E | E * E | id$
- Parse id*id+id

String	Action
.id*id+id	shift
id.*id+id	reduce $E \rightarrow id$
E.*id+id	shift

Example

- Assume grammar is $E \rightarrow E + E | E * E | id$
- Parse id*id+id

String	Action
.id*id+id	shift
id.*id+id	reduce $E \rightarrow id$
E.*id+id	shift
E*.id+id	shift

Example

- Assume grammar is $E \rightarrow E + E | E * E | id$
- Parse id*id+id

String	Action
.id*id+id	shift
id.*id+id	reduce $E \rightarrow id$
E.*id+id	shift
E*.id+id	shift
E*id.+id	reduce $E \rightarrow id$

Example

- Assume grammar is $E \rightarrow E + E | E * E | id$
- Parse id*id+id

String	Action
.id*id+id	shift
id.*id+id	reduce $E \rightarrow id$
E.*id+id	shift
E*.id+id	shift
E*id.+id	reduce $E \rightarrow id$
E*E.+id	reduce $E \rightarrow E * E$

Example

- Assume grammar is $E \rightarrow E + E | E * E | id$
- Parse id*id+id

String	Action
.id*id+id	shift
id.*id+id	reduce $E \rightarrow id$
E.*id+id	shift
E*.id+id	shift
E*id.+id	reduce $E \rightarrow id$
E*E.+id	reduce $E \rightarrow E * E$
E.+id	shift

Example

- Assume grammar is $E \rightarrow E + E | E * E | id$
- Parse id*id+id

String	Action
.id*id+id	shift
id.*id+id	reduce $E \rightarrow id$
E.*id+id	shift
E*.id+id	shift
E*id.+id	reduce $E \rightarrow id$
E*E.+id	reduce $E \rightarrow E * E$
E.+id	shift
E+.id	shift

Example

- Assume grammar is $E \rightarrow E + E|E * E|id$
- Parse id*id+id

String	Action
.id*id+id	shift
id.*id+id	reduce $E \rightarrow id$
E.*id+id	shift
E*.id+id	shift
E*id.+id	reduce $E \rightarrow id$
E*E.+id	reduce $E \rightarrow E * E$
E.+id	shift
E+.id	shift
E+id.	reduce $E \rightarrow id$

Example

- Assume grammar is $E \rightarrow E + E | E * E | id$
- Parse id*id+id

String	Action
.id*id+id	shift
id.*id+id	reduce $E \rightarrow id$
E.*id+id	shift
E*.id+id	shift
E*id.+id	reduce $E \rightarrow id$
E*E.+id	reduce $E \rightarrow E * E$
E.+id	shift
E+.id	shift
E+id.	reduce $E \rightarrow id$
E+E.	reduce $E \rightarrow E + E$

Example

- Assume grammar is $E \to E + E | E * E | id$
- Parse id*id+id

String	Action
.id*id+id	shift
id.*id+id	reduce $E \to id$
E.*id+id	shift
E*.id+id	shift
E*id.+id	reduce $E \to id$
E*E.+id	reduce $E \to E * E$
E.+id	shift
E+.id	shift
E+id.	reduce $E \to id$
E+E.	reduce $E \to E + E$
E.	ACCEPT

string successfully reduced to start symbol.

Shift reduce parsing

Shift reduce parsing

- Symbols on the left of "." are kept on a stack

Shift reduce parsing

- Symbols on the left of "." are kept on a stack
  - Top of the stack is at "."

Shift reduce parsing

- Symbols on the left of "." are kept on a stack
    - Top of the stack is at "."
    - Shift pushes a terminal on the stack

Shift reduce parsing

- Symbols on the left of "." are kept on a stack
  - Top of the stack is at "."
  - Shift pushes a terminal on the stack
  - Reduce pops symbols (rhs of production) and pushes a non terminal (lhs of production) onto the stack

Shift reduce parsing

- Symbols on the left of "." are kept on a stack
  - Top of the stack is at "."
  - Shift pushes a terminal on the stack
  - Reduce pops symbols (rhs of production) and pushes a non terminal (lhs of production) onto the stack
- The most important issue: when to shift and when to reduce

Shift reduce parsing

- Symbols on the left of "." are kept on a stack
  - Top of the stack is at "."
  - Shift pushes a terminal on the stack
  - Reduce pops symbols (rhs of production) and pushes a non terminal (lhs of production) onto the stack
- The most important issue: when to shift and when to reduce
- Reduce action should be taken only if the result can be reduced to the start symbol

Bottom up parsing

- A more powerful parsing technique

Bottom up parsing

- A more powerful parsing technique
- LR grammars – more expensive than LL

Bottom up parsing

- A more powerful parsing technique
- LR grammars – more expensive than LL
- Can handle left recursive grammars

Bottom up parsing

- A more powerful parsing technique
- LR grammars – more expensive than LL
- Can handle left recursive grammars
- Can handle virtually all the programming languages

Bottom up parsing

- A more powerful parsing technique
- LR grammars – more expensive than LL
- Can handle left recursive grammars
- Can handle virtually all the programming languages
- Natural expression of programming language syntax

Bottom up parsing

- A more powerful parsing technique
- LR grammars – more expensive than LL
- Can handle left recursive grammars
- Can handle virtually all the programming languages
- Natural expression of programming language syntax
- Automatic generation of parsers (Yacc, Bison etc.)

Bottom up parsing

- A more powerful parsing technique
- LR grammars – more expensive than LL
- Can handle left recursive grammars
- Can handle virtually all the programming languages
- Natural expression of programming language syntax
- Automatic generation of parsers (Yacc, Bison etc.)
- Detects errors as soon as possible

Bottom up parsing

LR grammars accept more class of languages compared with LL grammars.

- A more powerful parsing technique
- LR grammars – more expensive than LL
- Can handle left recursive grammars
- Can handle virtually all the programming languages
- Natural expression of programming language syntax
- Automatic generation of parsers (Yacc, Bison etc.)
- Detects errors as soon as possible
- Allows better error recovery

Issues in bottom up parsing

Issues in bottom up parsing

- How do we know which action to take

Issues in bottom up parsing

- How do we know which action to take
    - whether to shift or reduce

Issues in bottom up parsing

- How do we know which action to take
  - whether to shift or reduce
  - which production to use for reduction?

Issues in bottom up parsing

- How do we know which action to take
  - whether to shift or reduce
  - which production to use for reduction?
- Sometimes parser can reduce but it should not: $X \rightarrow \epsilon$ can always be reduced!

Issues in bottom up parsing

- How do we know which action to take
  - whether to shift or reduce
  - which production to use for reduction?
- Sometimes parser can reduce but it should not: $X \rightarrow \epsilon$ can always be reduced!
- Given stack δ and input symbol '*a*', should the parser

Issues in bottom up parsing

- How do we know which action to take
  - whether to shift or reduce
  - which production to use for reduction?
- Sometimes parser can reduce but it should not: $X \rightarrow \epsilon$ can always be reduced!
- Given stack δ and input symbol 'a', should the parser
  - Shift a onto stack (making it δa)

Issues in bottom up parsing

- How do we know which action to take
  - whether to shift or reduce
  - which production to use for reduction?
- Sometimes parser can reduce but it should not: $X \rightarrow \epsilon$ can always be reduced!
- Given stack δ and input symbol 'a', should the parser
  - Shift a onto stack (making it δa)
  - Reduce by some production $A \rightarrow \beta$ assuming that stack has form $a\beta$ (making it aA)

Issues in bottom up parsing

- How do we know which action to take
  - whether to shift or reduce
  - which production to use for reduction?
- Sometimes parser can reduce but it should not: $X \rightarrow \epsilon$ can always be reduced!
- Given stack δ and input symbol 'a', should the parser
  - Shift a onto stack (making it δa)
  - Reduce by some production $A \rightarrow \beta$ assuming that stack has form $a\beta$ (making it aA)
  - How to keep track of length of β?

Handle

Handle

- A string that matches right hand side of a production and whose replacement gives a step in the reverse right most derivation

Handle

- A string that matches right hand side of a production and whose replacement gives a step in the reverse right most derivation
- If $S \rightarrow^{rm*} aAw \rightarrow^{rm} a\beta w$ then β (corresponding to production $A \rightarrow \beta$) in the position following 'a' is a handle of $a\beta w$. The string w consists of only terminal symbols

Handle

- A string that matches right hand side of a production and whose replacement gives a step in the reverse right most derivation
- If $S \rightarrow^{rm*} aAw \rightarrow^{rm} a\beta w$ then β (corresponding to production $A \rightarrow \beta$) in the position following 'a' is a handle of $a\beta w$. The string w consists of only terminal symbols
- We only want to reduce handle and not any rhs

Handle

- A string that matches right hand side of a production and whose replacement gives a step in the reverse right most derivation
- If $S \rightarrow^{rm*} aAw \rightarrow^{rm} a\beta w$ then β (corresponding to production $A \rightarrow \beta$) in the position following 'a' is a handle of $a\beta w$. The string w consists of only terminal symbols
- We only want to reduce handle and not any rhs
- **Handle pruning**: If β is a handle and $A \rightarrow \beta$ is a production then replace β by A

Handle

- A string that matches right hand side of a production and whose replacement gives a step in the reverse right most derivation
- If $S \rightarrow^{rm*} aAw \rightarrow^{rm} a\beta w$ then β (corresponding to production $A \rightarrow \beta$) in the position following 'a' is a handle of $a\beta w$. The string w consists of only terminal symbols
- We only want to reduce handle and not any rhs
- **Handle pruning**: If β is a handle and $A \rightarrow \beta$ is a production then replace β by A
- A right most derivation in reverse can be obtained by handle pruning

Handle

- A string that matches right hand side of a production and whose replacement gives a step in the reverse right most derivation
- If $S \rightarrow^{rm*} aAw \rightarrow^{rm} a\beta w$ then β (corresponding to production $A \rightarrow \beta$) in the position following 'a' is a handle of $a\beta w$. The string w consists of only terminal symbols
- We only want to reduce handle and not any rhs
- **Handle pruning**: If β is a handle and $A \rightarrow \beta$ is a production then replace β by A
- A right most derivation in reverse can be obtained by handle pruning
- Handles always appear at the top of the stack and never inside it. This makes stack a suitable data structure

Handle

- A string that matches right hand side of a production and whose replacement gives a step in the reverse right most derivation
- If $S \rightarrow^{rm*} aAw \rightarrow^{rm} a\beta w$ then β (corresponding to production $A \rightarrow \beta$) in the position following 'a' is a handle of $a\beta w$. The string w consists of only terminal symbols
- We only want to reduce handle and not any rhs
- **Handle pruning**: If β is a handle and $A \rightarrow \beta$ is a production then replace β by A
- A right most derivation in reverse can be obtained by handle pruning
- Handles always appear at the top of the stack and never inside it. This makes stack a suitable data structure
- Bottom up parsing is based on recognizing handles

Conflicts

Conflicts

- The general shift-reduce technique is:

Conflicts

- The general shift-reduce technique is:
  - if there is no handle on the stack then shift

Conflicts

- The general shift-reduce technique is:
  - if there is no handle on the stack then shift
  - If there is a handle then reduce

Conflicts

- The general shift-reduce technique is:
  - if there is no handle on the stack then shift
  - If there is a handle then reduce
- However, what happens when there is a choice

Conflicts

- The general shift-reduce technique is:
  - if there is no handle on the stack then shift
  - If there is a handle then reduce
- However, what happens when there is a choice
  - What action to take in case both shift and reduce are valid. **Shift-Reduce conflict**

Conflicts

- The general shift-reduce technique is:
  - if there is no handle on the stack then shift
  - If there is a handle then reduce
- However, what happens when there is a choice
  - What action to take in case both shift and reduce are valid. **Shift-Reduce conflict**
  - Which rule to use for reduction if reduction is possible by more than one rule? **Reduce-Reduce conflict**

Conflicts

- The general shift-reduce technique is:
  - if there is no handle on the stack then shift
  - If there is a handle then reduce
- However, what happens when there is a choice
  - What action to take in case both shift and reduce are valid. **Shift-Reduce conflict**
  - Which rule to use for reduction if reduction is possible by more than one rule? **Reduce-Reduce conflict**
- Conflicts come either because of ambiguous grammars or parsing method is not powerful enough

if there are multiple entries in bottom up parse table, then grammar can be ambiguous, or parser technique is not powerful enough.

Shift-Reduce conflict

Shift-Reduce conflict

- Consider the grammar $E \rightarrow E + E | E * E | id$

Shift-Reduce conflict

- Consider the grammar $E \rightarrow E + E|E * E|id$
- input $id + id * id$

Shift-Reduce conflict

- Consider the grammar $E \rightarrow E + E | E * E | id$
- input $id + id * id$

stack	input	action
E+E	*id	reduce E→E+E
E	*id	shift
E*	id	shift
E*id		reduce E→id
E*E		reduce E→E*E
E		

Shift-Reduce conflict

- Consider the grammar $E \rightarrow E + E | E * E | id$
- input $id + id * id$

stack	input	action
E+E	*id	reduce E→E+E
E	*id	shift
E*	id	shift
E*id		reduce E→id
E*E		reduce E→E*E
E		

stack	input	action
E+E	*id	shift
E+E*	id	shift
E+E*id		reduce E→id
E+E*E		reduce E →E*E
E+E		reduce E→ E+E
E		

considering the precedence of * over +, this one is correct.

Reduce-Reduce conflict

Reduce-Reduce conflict

- Consider the grammar $M \rightarrow R + R | R + c | R$
  $R \rightarrow c$

Reduce-Reduce conflict

- Consider the grammar $M \rightarrow R + R | R + c | R$
  $R \rightarrow c$
- input $c + c$

Reduce-Reduce conflict

- Consider the grammar $M \rightarrow R + R | R + c | R$
  $R \rightarrow c$
- input $c + c$

stack	input	action
	c+c	shift
c	+c	reduce R→C
R	+c	shift
R+	c	shift
R+c		reduce R→c
R+R		reduce M→ R+R
M		

Reduce-Reduce conflict

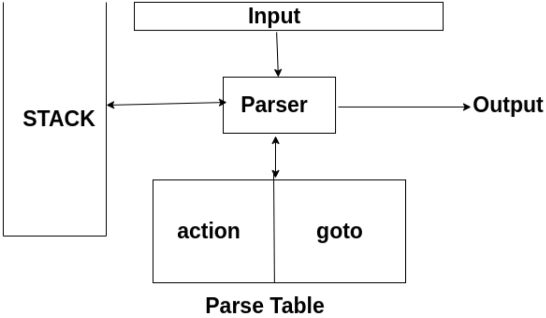
- Consider the grammar $M \rightarrow R + R | R + c | R$
  $R \rightarrow c$
- input $c + c$

stack	input	action
	c+c	shift
c	+c	reduce R→C
R	+c	shift
R+	c	shift
R+c		reduce R→c
R+R		reduce M→ R+R
M		

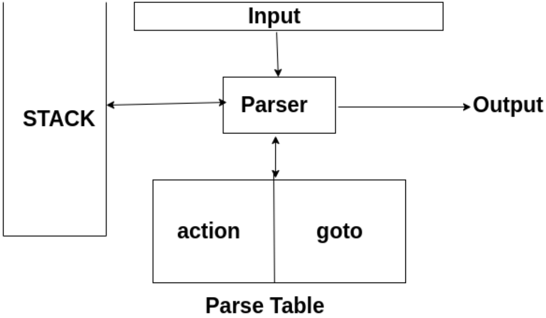
stack	input	action
	c+c	shift
c	+c	reduce R→C
R	+c	shift
R+	c	shift
R+c		reduce M→R+c
M		

LR Parsing

LR Parsing

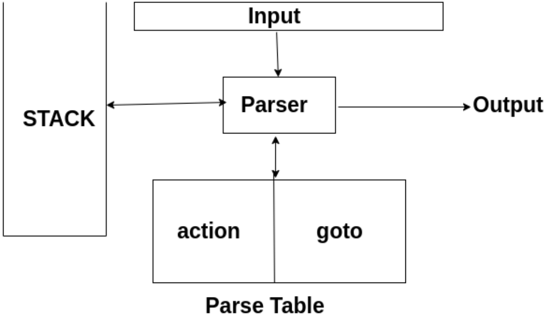


LR Parsing



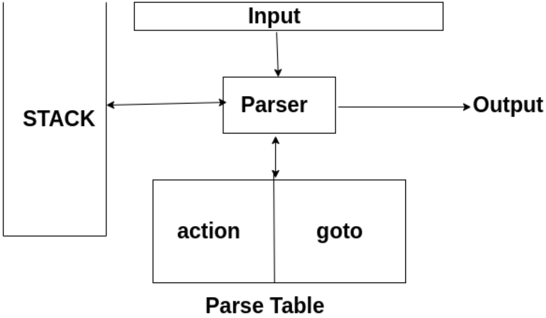
- Input contains the input string.

LR Parsing



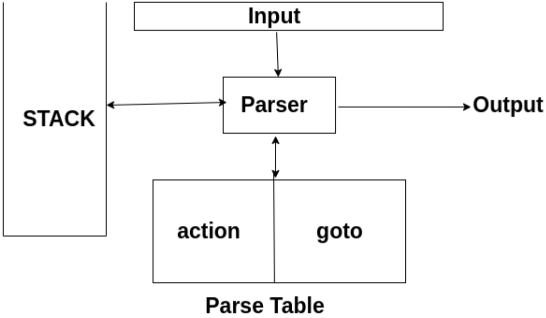
- Input contains the input string.
- Stack contains a string of the form $S_0X_1S_1X_2\cdots X_nS_n$ where each X_i is a grammar symbol and each S_i is a state.

LR Parsing



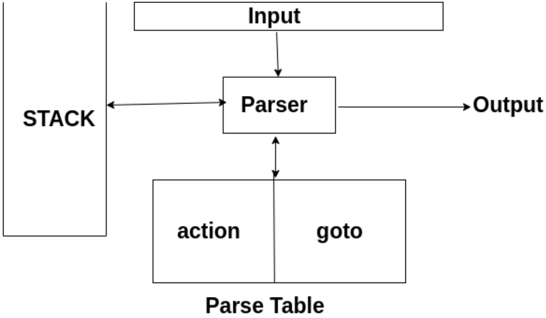
- Input contains the input string.
- Stack contains a string of the form $S_0X_1S_1X_2\cdots X_nS_n$ where each X_i is a grammar symbol and each S_i is a state.
- Tables contain **action** and **goto** parts.

LR Parsing



- Input contains the input string.
- Stack contains a string of the form $S_0 X_1 S_1 X_2 \cdots X_n S_n$ where each X_i is a grammar symbol and each S_i is a state.
- Tables contain **action** and **goto** parts.
- action table is indexed by state and terminal symbols.

LR Parsing



- Input contains the input string.
- Stack contains a string of the form $S_0X_1S_1X_2\cdots X_nS_n$ where each X_i is a grammar symbol and each S_i is a state.
- Tables contain **action** and **goto** parts.
- action table is indexed by state and terminal symbols.
- goto table is indexed by state and non terminal symbols.

Actions in an LR (shift reduce) parser

Actions in an LR (shift reduce) parser

- Assume S_i is top of stack and a_j is current input symbol

Actions in an LR (shift reduce) parser

- Assume S_i is top of stack and a_j is current input symbol
- Action $[S_i, a_j]$ can have four values

Actions in an LR (shift reduce) parser

- Assume S_i is top of stack and a_j is current input symbol
- Action $[S_i, a_j]$ can have four values
  1. shift a_j to the stack and goto state S_k

Actions in an LR (shift reduce) parser

- Assume S_i is top of stack and a_j is current input symbol
- Action $[S_i, a_j]$ can have four values
  1. shift a_j to the stack and goto state S_k
  2. reduce by a rule

Actions in an LR (shift reduce) parser

- Assume S_i is top of stack and a_j is current input symbol
- Action $[S_i, a_j]$ can have four values
  1. shift a_j to the stack and goto state S_k
  2. reduce by a rule
  3. accept

Actions in an LR (shift reduce) parser

- Assume S_i is top of stack and a_j is current input symbol
- Action $[S_i, a_j]$ can have four values
  1. shift a_j to the stack and goto state S_k
  2. reduce by a rule
  3. accept
  4. error

Configurations in LR parser

$Stack : S_0 X_1 S_1 X_2 \cdots X_m S_m \quad Input : a_i a_{i+1} \cdots a_n \$$

Configurations in LR parser

$Stack : S_0 X_1 S_1 X_2 \cdots X_m S_m \quad Input : a_i a_{i+1} \cdots a_n \$$

- If $action[S_m, a_i] = shift\ S$

Configurations in LR parser

$Stack : S_0 X_1 S_1 X_2 \cdots X_m S_m \quad Input : a_i a_{i+1} \cdots a_n\$$

- If $action[S_m, a_i] = shift\ S$

  Then the configuration becomes

  $Stack : S_0 X_1 S_1 \cdots X_m S_m a_i S \quad Input : a_{i+1} \cdots a_n\$$

Configurations in LR parser

$Stack : S_0X_1S_1X_2 \cdots X_mS_m \quad Input : a_ia_{i+1} \cdots a_n\$$

- If $action[S_m, a_i] = shift\ S$

  Then the configuration becomes

  $Stack : S_0X_1S_1 \cdots X_mS_ma_iS \quad Input : a_{i+1} \cdots a_n\$$
- If $action[S_m, a_i] =$ reduce $A \rightarrow \beta$

Configurations in LR parser

$Stack : S_0X_1S_1X_2 \cdots X_mS_m \quad Input : a_ia_{i+1} \cdots a_n\$$

- If $action[S_m, a_i] = shift\ S$

  Then the configuration becomes

  $Stack : S_0X_1S_1 \cdots X_mS_ma_iS \quad Input : a_{i+1} \cdots a_n\$$
- If $action[S_m, a_i] =$ reduce $A \rightarrow \beta$

  Then the configuration becomes

  $Stack : S_0X_1S_1 \cdots S_{m-r}X_{m-r}S \quad Input : a_ia_{i+1} \cdots a_n\$$

Configurations in LR parser

$Stack : S_0X_1S_1X_2 \cdots X_mS_m \quad Input : a_ia_{i+1} \cdots a_n\$$

- If $action[S_m, a_i] = shift\ S$

  Then the configuration becomes

  $Stack : S_0X_1S_1 \cdots X_mS_ma_iS \quad Input : a_{i+1} \cdots a_n\$$

- If $action[S_m, a_i] =$ reduce $A \to \beta$

  Then the configuration becomes

  $Stack : S_0X_1S_1 \cdots S_{m-r}X_{m-r}S \quad Input : a_ia_{i+1} \cdots a_n\$$

  $Where \quad r = |\beta| and \quad S = goto[S_{m-r}, A]$

Configurations in LR parser

$Stack : S_0 X_1 S_1 X_2 \cdots X_m S_m \quad Input : a_i a_{i+1} \cdots a_n\$$

- If $action[S_m, a_i] = shift\ S$

  Then the configuration becomes

  $Stack : S_0 X_1 S_1 \cdots X_m S_m a_i S \quad Input : a_{i+1} \cdots a_n\$$
- If $action[S_m, a_i] =$ reduce $A \rightarrow \beta$

  Then the configuration becomes

  $Stack : S_0 X_1 S_1 \cdots S_{m-r} X_{m-r} S \quad Input : a_i a_{i+1} \cdots a_n\$$

  $Where \quad r = |\beta| and \quad S = goto[S_{m-r}, A]$
- If $action[S_m, a_i] = accept$

Configurations in LR parser

$Stack : S_0 X_1 S_1 X_2 \cdots X_m S_m \quad Input : a_i a_{i+1} \cdots a_n \$$

- If $action[S_m, a_i] = shift\ S$

  Then the configuration becomes

  $Stack : S_0 X_1 S_1 \cdots X_m S_m a_i S \quad Input : a_{i+1} \cdots a_n \$$
- If $action[S_m, a_i] =$ reduce $A \rightarrow \beta$

  Then the configuration becomes

  $Stack : S_0 X_1 S_1 \cdots S_{m-r} X_{m-r} S \quad Input : a_i a_{i+1} \cdots a_n \$$

  $Where \quad r = |\beta| and \quad S = goto[S_{m-r}, A]$
- If $action[S_m, a_i] = accept$

  Then parsing is completed. *HALT*

Configurations in LR parser

$Stack : S_0 X_1 S_1 X_2 \cdots X_m S_m \quad Input : a_i a_{i+1} \cdots a_n\$$

- If $action[S_m, a_i] = shift\ S$

  Then the configuration becomes

  $Stack : S_0 X_1 S_1 \cdots X_m S_m a_i S \quad Input : a_{i+1} \cdots a_n\$$

- If $action[S_m, a_i] =$ reduce $A \rightarrow \beta$

  Then the configuration becomes

  $Stack : S_0 X_1 S_1 \cdots S_{m-r} X_{m-r} S \quad Input : a_i a_{i+1} \cdots a_n\$$

  $Where \quad r = |\beta| and \quad S = goto[S_{m-r}, A]$

- If $action[S_m, a_i] = accept$

  Then parsing is completed. $HALT$

- If $action[S_m, a_i] = error$

Configurations in LR parser

$Stack : S_0 X_1 S_1 X_2 \cdots X_m S_m \quad Input : a_i a_{i+1} \cdots a_n\$$

top of stack will be always stack symbol

- If $action[S_m, a_i] = shift\ S$

  Then the configuration becomes

  $Stack : S_0 X_1 S_1 \cdots X_m S_m a_i S \quad Input : a_{i+1} \cdots a_n\$$

- If $action[S_m, a_i] =$ reduce $A \rightarrow \beta$

  Then the configuration becomes

  $Stack : S_0 X_1 S_1 \cdots S_{m-r} X_{m-r} S \quad Input : a_i a_{i+1} \cdots a_n\$$

  $Where \quad r = |\beta| and \quad S = goto[S_{m-r}, A]$

X{m-r} will be A

- If $action[S_m, a_i] = accept$

  Then parsing is completed. *HALT*

No need to check anything when "accept" comes. Checks have been done during parse table creation.

- If $action[S_m, a_i] = error$

  Then invoke error recovery routine

LR parsing Algorithm

Algorithm LR$_p$*arsing*$_A$*lgorithm*

```
Initial State $Stack : S_0$   $Input : w\$$
for $TRUE$ do
    if $action[S, a] = shift \quad S'$ then
        push(a); push(S'); ip++
    else if  action[S,a] = reduce $A \rightarrow \beta$ then
        pop $(2 * |\beta|)$ symbols;
        push(A); push (goto[S",A]) {S"  is the state after popping symbols}
    else if action[S,a] = accept then
        exit
    else
        error()
    end if
end for
```

Example

- Consider the grammar

Example

- Consider the grammar $E \to E + T | T \qquad T \to T * F | F \qquad F \to (E) | id$

state	id	+	*	(	)	$	E	T	F
0	s5			s4			1	2	3
1		s6				acc			
2		r2	s7		r2	r2			
3		r4	r4		r4	r4			
4	s5			s4			8	2	3
5		r6	r6		r6	r6			
6	s5			s4				9	3
7	s5			s4					10
8		s6			s11				
9		r1	s7		r1	r1			
10		r3	r3		r3	r3			
11		r5	r5		r5	r5			

Parse id + id * id

stack	input	action
0	id + id * id$	s5
0id5	+ id*id$	reduce F→id
0F3	+id*id$	reduce T→ F
0T2	+id*id$	reduce E→T
0E1	+id*id$	shift 6
0E1+6	id*id $	shift 5
0E1+6id5	*id $	reduce F→id
0E1+6F3	*id $	reduce T→F
0E1+6T9	*id $	shift 7
0E1+6T9*7	id $	shift 5
0E1+6T9*7id5	$	reduce F → id
0E1+6T9*7F10	$	reduce T →T*F
0E1+6T9	$	reduce E→E+T
0E1	$	acc