Lecture 7

Syntax Analysis

Awanish Pandey

Department of Computer Science and Engineering
Indian Institute of Technology
Roorkee

February 4, 2025

Take aways from the last class

- Overview of Syntax Analysis

Take aways from the last class

- Overview of Syntax Analysis
- Derivation of string from a grammer

Take aways from the last class

- Overview of Syntax Analysis
- Derivation of string from a grammer
- Parse Tree

Take aways from the last class

- Overview of Syntax Analysis
- Derivation of string from a grammer
- Parse Tree
- Ambiguity

Take aways from the last class

- Overview of Syntax Analysis
- Derivation of string from a grammer
- Parse Tree
- Ambiguity
- Resolving dangling `if-else` problem.

Take aways from the last class

- Overview of Syntax Analysis
- Derivation of string from a grammer
- Parse Tree
- Ambiguity
- Resolving dangling `if-else` problem.
- Top down parsing

Take aways from the last class

- Overview of Syntax Analysis
- Derivation of string from a grammer
- Parse Tree
- Ambiguity
- Resolving dangling `if-else` problem.
- Top down parsing
- Recursive Descent Parsing

Take aways from the last class

- Overview of Syntax Analysis
- Derivation of string from a grammer
- Parse Tree
- Ambiguity
- Resolving dangling `if-else` problem.
- Top down parsing
- Recursive Descent Parsing
- Left Recursion

Take aways from the last class

- Overview of Syntax Analysis
- Derivation of string from a grammer
- Parse Tree
- Ambiguity
- Resolving dangling `if-else` problem.
- Top down parsing
- Recursive Descent Parsing
- Left Recursion
- Left Factoring

Predictive Parser

- A non recursive top down parsing method

Predictive Parser

- A non recursive top down parsing method
- Parser "predicts" which production to use

Predictive Parser

- A non recursive top down parsing method
- Parser “predicts” which production to use
- It removes backtracking by fixing one production for every non- terminal and input token(s)

Predictive Parser

- A non recursive top down parsing method
- Parser “predicts” which production to use
- It removes backtracking by fixing one production for every non- terminal and input token(s)
- Predictive parsers accept LL(k) languages

Predictive Parser

- A non recursive top down parsing method
- Parser "predicts" which production to use
- It removes backtracking by fixing one production for every non- terminal and input token(s)
- Predictive parsers accept LL(k) languages
  - First L stands for left to right scan of input

Predictive Parser

- A non recursive top down parsing method
- Parser "predicts" which production to use
- It removes backtracking by fixing one production for every non- terminal and input token(s)
- Predictive parsers accept LL(k) languages
  - First L stands for left to right scan of input
  - Second L stands for leftmost derivation

Predictive Parser

- A non recursive top down parsing method
- Parser "predicts" which production to use
- It removes backtracking by fixing one production for every non- terminal and input token(s)
- Predictive parsers accept LL(k) languages
  - First L stands for left to right scan of input
  - Second L stands for leftmost derivation
  - k stands for number of lookahead token

Predictive Parser

also Predictive Parser is a Recursive descent parser with no backtracking.

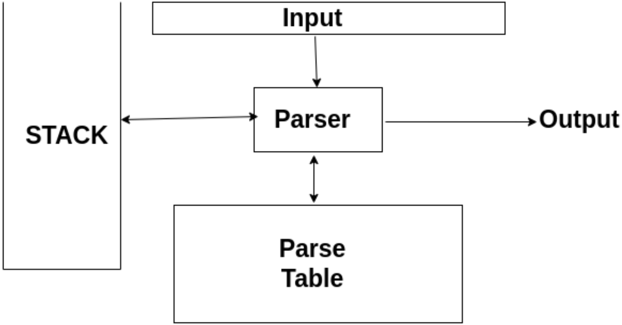
- A non recursive top down parsing method
- Parser "predicts" which production to use
- It removes backtracking by fixing one production for every non- terminal and input token(s)
- Predictive parsers accept LL(k) languages
  - First L stands for left to right scan of input
  - Second L stands for leftmost derivation
  - k stands for number of lookahead token
  - In practice LL(1) is used

Predictive Parser

Predictive parser can be implemented by maintaining an external stack

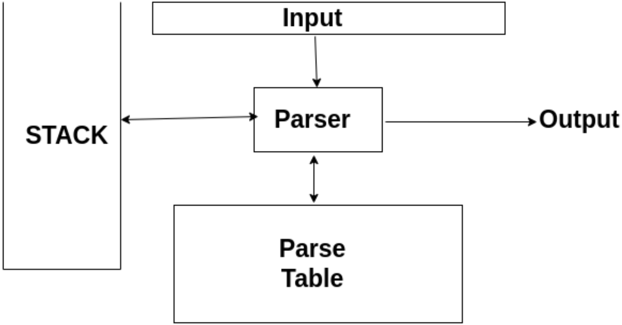
Predictive Parser

Predictive parser can be implemented by maintaining an external stack



Predictive Parser

Predictive parser can be implemented by maintaining an external stack



Parse table is a two dimensional array $M[X, a]$ where "X" is a non terminal and "a" is a terminal of the grammar

Example

Consider the following grammar:

$$E \rightarrow TE'$$
$$E' \rightarrow +TE'|\epsilon$$
$$T \rightarrow FT'$$
$$T' \rightarrow *FT'|\epsilon$$
$$F \rightarrow (E)|id$$

Parse table for the grammar

	id	+	*	(	)	$
E	$E \rightarrow TE^{'}$			$E \rightarrow TE^{'}$		
E'		$E^{'} \rightarrow +TE^{'}$			$E^{'} \rightarrow \epsilon$	$E^{'} \rightarrow \epsilon$
T	$T \rightarrow FT^{'}$			$T \rightarrow FT^{'}$		
T'		$T^{'} \rightarrow \epsilon$	$T^{'} \rightarrow *FT^{'}$		$T^{'} \rightarrow \epsilon$	$T^{'} \rightarrow \epsilon$
F	$F \rightarrow id$			$F \rightarrow (E)$		

Parse table for the grammar

	id	+	*	(	)	$
E	$E \rightarrow TE^{'}$			$E \rightarrow TE^{'}$		
E'		$E^{'} \rightarrow +TE^{'}$			$E^{'} \rightarrow \epsilon$	$E^{'} \rightarrow \epsilon$
T	$T \rightarrow FT^{'}$			$T \rightarrow FT^{'}$		
T'		$T^{'} \rightarrow \epsilon$	$T^{'} \rightarrow *FT^{'}$		$T^{'} \rightarrow \epsilon$	$T^{'} \rightarrow \epsilon$
F	$F \rightarrow id$			$F \rightarrow (E)$		

Blank entries are error states.

Parsing Algorithm

- The parser considers 'X' the symbol on top of stack, and 'a' the current input symbol.

Parsing Algorithm

- The parser considers 'X' the symbol on top of stack, and 'a' the current input symbol.
- These two symbols determine the action to be taken by the parser.

Parsing Algorithm

- The parser considers 'X' the symbol on top of stack, and 'a' the current input symbol.
- These two symbols determine the action to be taken by the parser.
- Assume that '\$' is a special token that is at the bottom of the stack and terminator for the input string

Parsing Algorithm

- The parser considers 'X' the symbol on top of stack, and 'a' the current input symbol.
- These two symbols determine the action to be taken by the parser.
- Assume that '\$' is a special token that is at the bottom of the stack and terminator for the input string
- `if X = a = $ then halt`

Parsing Algorithm

- The parser considers 'X' the symbol on top of stack, and 'a' the current input symbol.
- These two symbols determine the action to be taken by the parser.
- Assume that '$' is a special token that is at the bottom of the stack and terminator for the input string
- `if X = a = $ then halt`
- if $X = a \neq \$ \; then \; pop(x) \; and \; ip++$

Parsing Algorithm

- The parser considers 'X' the symbol on top of stack, and 'a' the current input symbol.
- These two symbols determine the action to be taken by the parser.
- Assume that '$' is a special token that is at the bottom of the stack and terminator for the input string
- `if X = a = $ then halt`
- if $X = a \neq \$ \; then \; pop(x) \; and \; ip++$
- `if X is a non terminal`

Parsing Algorithm

- The parser considers 'X' the symbol on top of stack, and 'a' the current input symbol.
- These two symbols determine the action to be taken by the parser.
- Assume that '\$' is a special token that is at the bottom of the stack and terminator for the input string
- `if X = a = $ then halt`
- if $X = a \neq \$$ *then pop*(x) *and ip*$++$
- `if X is a non terminal`
  `then if` $M[X, a] = X \rightarrow UVW$

Parsing Algorithm

- The parser considers 'X' the symbol on top of stack, and 'a' the current input symbol.
- These two symbols determine the action to be taken by the parser.
- Assume that '\$' is a special token that is at the bottom of the stack and terminator for the input string
- `if X = a = $ then halt`
- if $X = a \neq \$$ *then pop*(x) *and ip* $++$
- `if X is a non terminal`
  then if $M[X, a] = X \to UVW$
  `then begin pop(X); push(W,V,U)`

Parsing Algorithm

- The parser considers 'X' the symbol on top of stack, and 'a' the current input symbol.
- These two symbols determine the action to be taken by the parser.
- Assume that '\$' is a special token that is at the bottom of the stack and terminator for the input string
- `if X = a = $ then halt`
- if $X = a \neq \$ \ \textit{then pop}(x) \textit{ and } ip++$
- `if X is a non terminal`
  `then if` $M[X, a] = X \rightarrow UVW$
  `then begin pop(X); push(W,V,U)`
  end

Parsing Algorithm

$ is not considered part of grammar

- The parser considers 'X' the symbol on top of stack, and 'a' the current input symbol.
- These two symbols determine the action to be taken by the parser.
- Assume that '\$' is a special token that is at the bottom of the stack and terminator for the input string
- `if X = a = $ then halt`
- if $X = a \neq \$$ *then pop*(x) *and ip*$++$
- `if X is a non terminal`

  then if $M[X, a] = X \rightarrow UVW$

  ```
      then begin pop(X); push(W,V,U)
  end
  else error
  ```

push W, then V and then U

Example

initially, the stack will contain \$ and above it will be the start symbol.

Stack	Input	Action
\$E	id+id*id \$	expand by $E \rightarrow TE'$

Example

Stack	Input	Action
$E	id+id*id $	expand by $E \to TE'$
$E'T	id+id*id $	expand by $T \to FT'$

Example

Stack	Input	Action
\$E	id+id*id \$	expand by $E \rightarrow TE'$
\$E'T	id+id*id \$	expand by $T \rightarrow FT'$
\$E'T'F	id+id*id \$	expand by $F \rightarrow id$

Example

Stack	Input	Action
$E	id+id*id $	expand by $E \rightarrow TE'$
$E'T	id+id*id $	expand by $T \rightarrow FT'$
$E'T'F	id+id*id $	expand by $F \rightarrow id$
$E'T'id	id+id*id $	pop id and ip++

Example

Stack	Input	Action
$E	id+id*id $	expand by $E \rightarrow TE'$
$E'T	id+id*id $	expand by $T \rightarrow FT'$
$E'T'F	id+id*id $	expand by $F \rightarrow id$
$E'T'id	id+id*id $	pop id and ip++
$E'T'	+id*id $	expand by $T \rightarrow \epsilon$

Example

Stack	Input	Action
$E	id+id*id $	expand by $E \rightarrow TE'$
$E'T	id+id*id $	expand by $T \rightarrow FT'$
$E'T'F	id+id*id $	expand by $F \rightarrow id$
$E'T'id	id+id*id $	pop id and ip++
$E'T'	+id*id $	expand by $T \rightarrow \epsilon$
$E'	+id*id $	expand by $E' \rightarrow +TE'$

Example

Stack	Input	Action
$E	id+id*id $	expand by $E \to TE'$
$E'T	id+id*id $	expand by $T \to FT'$
$E'T'F	id+id*id $	expand by $F \to id$
$E'T'id	id+id*id $	pop id and ip++
$E'T'	+id*id $	expand by $T \to \epsilon$
$E'	+id*id $	expand by $E' \to +TE'$
$E'T+	+id*id $	pop + and ip++

Example

Stack	Input	Action
$E	id+id*id $	expand by $E \rightarrow TE'$
$E'T	id+id*id $	expand by $T \rightarrow FT'$
$E'T'F	id+id*id $	expand by $F \rightarrow id$
$E'T'id	id+id*id $	pop id and ip++
$E'T'	+id*id $	expand by $T \rightarrow \epsilon$
$E'	+id*id $	expand by $E' \rightarrow +TE'$
$E'T+	+id*id $	pop + and ip++
$E'T	id*id $	expand by $T \rightarrow FT'$

Example

Stack	Input	Action
$E'T'F	id*id $	expand by $F \rightarrow id$

Example

Stack	Input	Action
$E'T'F	id*id $	expand by $F \rightarrow id$
$E'T'id	id*id $	pop id and ip++

Example

Stack	Input	Action
$E'T'F	id*id $	expand by $F \rightarrow id$
$E'T'id	id*id $	pop id and ip$++$
$E'T'	*id $	expand by $T' \rightarrow *FT'$

Example

Stack	Input	Action
$E'T'F	id*id $	expand by $F \rightarrow id$
$E'T'id	id*id $	pop id and ip$++$
$E'T'	*id $	expand by $T' \rightarrow *FT'$
$E'T'F*	*id $	pop * and ip$++$

Example

Stack	Input	Action
$E'T'F	id*id $	expand by $F \rightarrow id$
$E'T'id	id*id $	pop id and ip$++$
$E'T'	*id $	expand by $T' \rightarrow *FT'$
$E'T'F*	*id $	pop * and ip$++$
$E'T'F	id $	expand by $F \rightarrow id$

Example

Stack	Input	Action
$E'T'F	id*id $	expand by $F \rightarrow id$
$E'T'id	id*id $	pop id and ip++
$E'T'	*id $	expand by $T' \rightarrow *FT'$
$E'T'F*	*id $	pop * and ip++
$E'T'F	id $	expand by $F \rightarrow id$
$E'T'id	id $	pop id and ip++

Example

Stack	Input	Action
$E'T'F	id*id $	expand by $F \to id$
$E'T'id	id*id $	pop id and ip++
$E'T'	*id $	expand by $T' \to *FT'$
$E'T'F*	*id $	pop * and ip++
$E'T'F	id $	expand by $F \to id$
$E'T'id	id $	pop id and ip++
$E'T'	$	expand by $T' \to \epsilon$

Example

Stack	Input	Action
$E'T'F	id*id $	expand by $F \rightarrow id$
$E'T'id	id*id $	pop id and ip++
$E'T'	*id $	expand by $T' \rightarrow *FT'$
$E'T'F*	*id $	pop * and ip++
$E'T'F	id $	expand by $F \rightarrow id$
$E'T'id	id $	pop id and ip++
$E'T'	$	expand by $T' \rightarrow \epsilon$
$E'	$	expand by $E' \rightarrow \epsilon$

Example

Stack	Input	Action
$E'T'F	id*id $	expand by $F \rightarrow id$
$E'T'id	id*id $	pop id and ip++
$E'T'	*id $	expand by $T' \rightarrow *FT'$
$E'T'F*	*id $	pop * and ip++
$E'T'F	id $	expand by $F \rightarrow id$
$E'T'id	id $	pop id and ip++
$E'T'	$	expand by $T' \rightarrow \epsilon$
$E'	$	expand by $E' \rightarrow \epsilon$
$	$	halt

Constructing Parse table

Constructing Parse table

- Table can be constructed if for every non terminal, every lookahead symbol can be handled by at most one production

what are the cases in which there can be more than one production in any entry??
If multiple entries,
1. Grammar is ambiguous
2. If grammar is not ambiguous, then parser is not strong enough.
    a. Grammar may be left recursive or left factored.

LL(K) is more powerful than LL(1) => If we are doing LL(1), and given that grammar is ambiguous and multiple entries arise, then increase the value of K.

Constructing Parse table

- Table can be constructed if for every non terminal, every lookahead symbol can be handled by at most one production
- **First(a)** for a string of terminals and non terminals *a* is Set of symbols that might begin the fully expanded (made of only tokens) version of *a*

Constructing Parse table

- Table can be constructed if for every non terminal, every lookahead symbol can be handled by at most one production
- **First(a)** for a string of terminals and non terminals *a* is Set of symbols that might begin the fully expanded (made of only tokens) version of *a*
- **Follow(X)** for a non terminal X is set of symbols that might follow the derivation of X in the input stream

Computation of FIRST set

Computation of FIRST set

- If X is a terminal symbol then $First(X) = X$

Computation of FIRST set

- If X is a terminal symbol then $First(X) = X$
- If $X \rightarrow \epsilon$ is a production then ϵ is in First(X).

Computation of FIRST set

- If X is a terminal symbol then $First(X) = X$
- If $X \rightarrow \epsilon$ is a production then ϵ is in First(X).
- If X is a non terminal and $X \rightarrow Y_l Y_2 \cdots Y_k$ is a production then if for some "i", "a" is in $First(Y_i)$ and ϵ is in all of $First(Y_j)$ (such that $j < i$) then a is in $First(X)$

Computation of FIRST set

- If X is a terminal symbol then $First(X) = X$
- If $X \rightarrow \epsilon$ is a production then ϵ is in First(X).
- If X is a non terminal and $X \rightarrow Y_l Y_2 \cdots Y_k$ is a production then if for some "i", "a" is in $First(Y_i)$ and ϵ is in all of $First(Y_j)$ (such that $j < i$) then a is in $First(X)$
- If ϵ is in $First(Y_1) \cdots First(Y_k)$ then ϵ is in $First(X)$

Example

- For the expression grammar
  $E \rightarrow TE^{'}$
  $E' \rightarrow +TE'|\epsilon$
  $T \rightarrow FT'$
  $T' \rightarrow *FT'|\epsilon$
  $F \rightarrow (E)|id$

Example

- For the expression grammar
  $E \rightarrow TE^{'}$
  $E' \rightarrow +TE'|\epsilon$
  $T \rightarrow FT'$
  $T' \rightarrow *FT'|\epsilon$
  $F \rightarrow (E)|id$
- $First(E) = First(T) = First(F) = (, id$

Example

- For the expression grammar
  $E \rightarrow TE^{'}$
  $E' \rightarrow +TE'|\epsilon$
  $T \rightarrow FT'$
  $T' \rightarrow *FT'|\epsilon$
  $F \rightarrow (E)|id$
- $First(E) = First(T) = First(F) = (, id$
- $First(E') = +, \epsilon$

Example

- For the expression grammar
  $E \to TE^{'}$
  $E' \to +TE'|\epsilon$
  $T \to FT'$
  $T' \to *FT'|\epsilon$
  $F \to (E)|id$
- $First(E) = First(T) = First(F) = (, id$
- $First(E') = +, \epsilon$
- $First(T') = *, \epsilon$