Lecture Eight

Introduction to Syntax Analysis

The CSC318 Team

[2022]

Lecture Outline

- Introduction
- Syntax Analysers
- Context-Free Grammars
  - Derivations
  - Parse Tree
- Types of Parsers
- Top-Down Parsers
  - Ambiguity
  - Left Recursion
  - Left Factoring
- Classification of Top-Down Parsers
  - Recursive Decent Parser
  - Predictive Parsers
- LL Parsers
  - Using Parse Tables
  - Computation of First and Follow Sets
  - Construction of Parse Tables
- Limitations of Syntax Analysers

Lecture Outline | **Progress**

- **Introduction**
- Syntax Analysers
- Context-Free Grammars
  - Derivations
  - Parse Tree
- Types of Parsers
- Top-Down Parsers
  - Ambiguity
  - Left Recursion
  - Left Factoring
- Classification of Top-Down Parsers
  - Recursive Decent Parser
  - Predictive Parsers
- LL Parsers
  - Using Parse Tables
  - Computation of First and Follow Sets
  - Construction of Parse Tables
- Limitations of Syntax Analysers

Introduction

- **Lexical Analysers** can identify tokens with the help of Regular **Expressions and pattern rules.**
  - **Lexical Analyser** cannot check the **Syntax** of a given sentence due to the limitations of the **Regular Expressions**.
  - RegExps **cannot** check balancing tokens, such as **parenthesis**.

Introduction

- Most **Languages** are not regular, and are more complicated than what a **RegExps** can describe.
  - E.g. its difficult to express "for every opening parenthesis there must be a closing parenthesis" using a **RegExps**.
- RegExps are used to describe the **tokens** of a language.
  - Recall:
    - **Identifiers** in the language match the regex [a-zA-Z_][a-zA-Z0-9_]*
    - **Numbers** match the regex [0-9]$^+$
  - How those tokens fit together to form a complete program is then described in a **Grammar**

Lecture Outline | Progress

- Introduction
- **Syntax Analysers**
- Context-Free Grammars
  - Derivations
  - Parse Tree
- Types of Parsers
- Top-Down Parsers
  - Ambiguity
  - Left Recursion
  - Left Factoring
- Classification of Top-Down Parsers
  - Recursive Decent Parser
  - Predictive Parsers
- LL Parsers
  - Using Parse Tables
  - Computation of First and Follow Sets
  - Construction of Parse Tables
- Limitations of Syntax Analysers

Syntax Analysers

- A **Syntax Analyser** (or parser) takes the input from a Lexical Analyser in the form of **Token Streams**.
  - The parser analyses the source code (**token stream**) based on the **Production Rules** to detect any **error** in the code.
  - The output of this phase is a **Parse Tree**.
  - Therefore, the parser accomplishes the following:
    - Looking for **Errors (Syntax Errors)**
    - Generating a **Parse Tree**
      - Parsers parses the whole code even if there are errors in the code

Syntax Analysers

- The syntax of a language describes which programs are structurally valid or not – **Syntactic Validity**
  - Recall:
    - A **Language** is a set of sentences formed by the set of basic symbols.
    - A **Grammar** is the set of **rules** that govern how to ascertain that these sentences are part of the language or not.
    - However, a syntactically correct sentence may make no actual sense
- Therefore, **Semantic analysis** are usually handled separately
  - Such as type checking errors and runtime errors

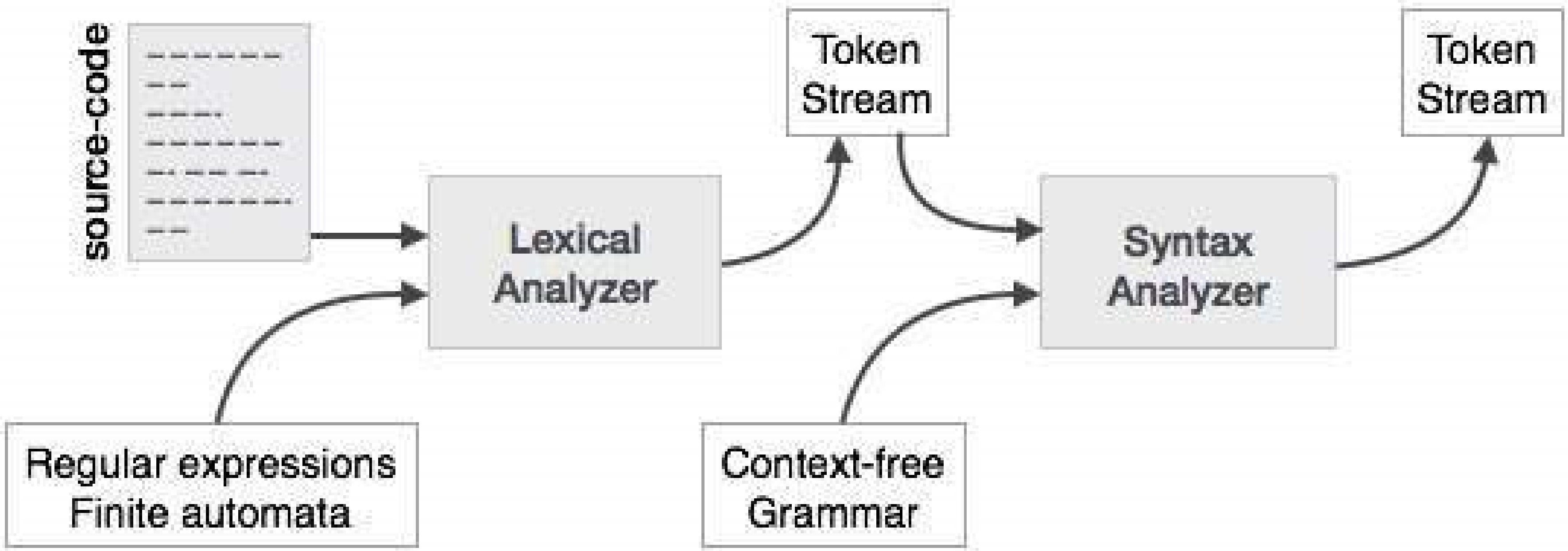
Lecture Outline | **Progress**

- Introduction
- Syntax Analysers
- **Context-Free Grammars**
  - Derivations
  - Parse Tree
- Types of Parsers
- Top-Down Parsers
  - Ambiguity
  - Left Recursion
  - Left Factoring
- Classification of Top-Down Parsers
  - Recursive Decent Parser
  - Predictive Parsers
- LL Parsers
  - Using Parse Tables
  - Computation of First and Follow Sets
  - Construction of Parse Tables
- Limitations of Syntax Analysers

Context-Free Grammar (CFG)

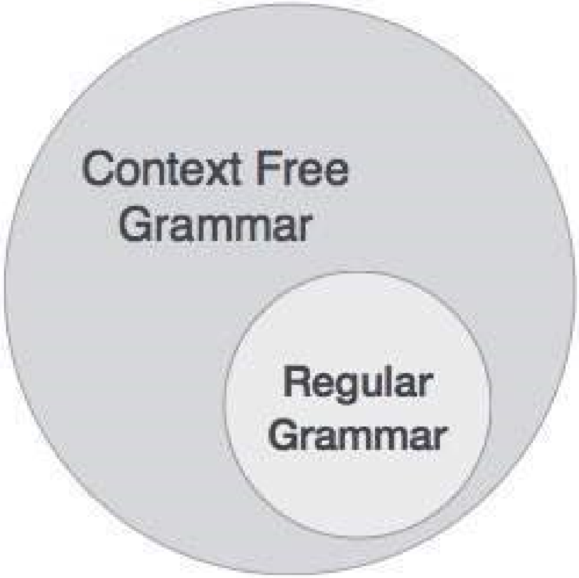
- The Syntax Analyzer (Parsers) use a class of Grammar called **Context-Free Grammar (CFG)**
  - Recognized by **Push-down Automata.**
    - **Recall: Chomsky Hierarchy of Grammars**
- **CFG** are well-suited to programming languages
  - They restrict the manner in which programming construct can be used
  - They simplify the process of analysing its use in a program.
  - They are called *Context-free* because
    - Non-terminals are parsed independent of the other symbols surrounding it (i.e., without respect to context)

Graphical Depiction



Context-Free Grammar (CFG)

- CFG is a **superset** of **Regular Grammar**
  - Every **Regular Grammar** is CFG
  - Already mentioned
    - CFG is a helpful tool in describing the syntax of programming languages.



Four components of Context-Free Grammar

- **A Set Of Non-terminals (V)**
  - Non-terminals are syntactic variables that denote sets of strings. The non-terminals define sets of strings that help define the language generated by the grammar.
- **A set of tokens (Σ) (i.e. Terminal symbols)**
  - Terminals are the basic symbols from which strings are formed.
- **A set of productions (P)**
  - The productions of a grammar specify the manner in which the terminals and non-terminals can be combined to form strings.
  - Each production consists of a non-terminal called the left side of the production, an arrow, and a sequence of tokens and/or on- terminals, called the right side of the production.
- **One of the non-terminals is designated as the start symbol (S)-** from where the production begins.
  - The strings are derived from the start symbol by repeatedly replacing a non-terminal (initially the start symbol) by the right side of a production, for that non-terminal

Context-Free Grammar

$$G = (V, \Sigma, P, S)$$

Lecture Outline | **Progress**

- Introduction
- Syntax Analysers
- **Context-Free Grammars**
  - **Derivations**
  - Parse Tree
- Types of Parsers
- Top-Down Parsers
  - Ambiguity
  - Left Recursion
  - Left Factoring
- Classification of Top-Down Parsers
  - Recursive Decent Parser
  - Predictive Parsers
- LL Parsers
  - Using Parse Tables
  - Computation of First and Follow Sets
  - Construction of Parse Tables
- Limitations of Syntax Analysers

Derivation

- Derivation is a sequence of production rules;
  - Involved in generating the input string.
- For a sentential form of inputs two decisions are taken during parsing
  - Deciding the **Non-terminal** which is to be replaced.
  - Deciding the **Production Rule**, by which, the non-terminal will be replaced.

Derivation Options

- Non-terminals can be replaced with production rules, based on either of the following:
  - **Left-Most Derivation**
    - Input is scanned and replaced from **Left** to **Right**
    - The sentential form derived is called the left-sentential form.
  - **Right-Most Derivation**
    - Input is scanned and replaced from **Right to Left**
    - The sentential form derived is called the right-sentential form.

Example

- Given the Production rules:

  F → F + F

  F → F * F

  F → id

- Given the Input string: ***id + id * id***
  - **Show the Left-most and Right-most Derivation**

Derivations

The Right-Most Derivation is

$F \rightarrow F + F$

$F \rightarrow F + F * F$

$F \rightarrow F + F * id$

$F \rightarrow F + id * id$

$F \rightarrow id + id * id$

The left-Most derivation is

$F \rightarrow F * F$

$F \rightarrow F + F * F$

$F \rightarrow id + F * F$

$F \rightarrow id + id * F$

$F \rightarrow id + id * id$

Notice that the sides that is always processed first.

Lecture Outline | **Progress**

- Introduction
- Syntax Analysers
- **Context-Free Grammars**
  - **Derivations**
  - **Parse Tree**
- Types of Parsers
- Top-Down Parsers
  - Ambiguity
  - Left Recursion
  - Left Factoring
- Classification of Top-Down Parsers
  - Recursive Decent Parser
  - Predictive Parsers
- LL Parsers
  - Using Parse Tables
  - Computation of First and Follow Sets
  - Construction of Parse Tables
- Limitations of Syntax Analysers

Parse Tree

- A parse tree is a **Graphical Depiction** of a derivation.
  - It is convenient to see how strings are derived from the start symbol.
  - The **Start Symbol** of the derivation becomes the root of the parse tree.

Parse Tree

- In a parse tree:
  - All leaf nodes are **terminals**.
  - All interior nodes are **non-terminals**.
  - In-order traversal gives original input string.
- A parse tree depicts **Associativity** and **Precedence** of operators.
  - The deepest sub-tree is traversed first,
    - Therefore the operator in that sub-tree gets precedence over the operator which is in the parent nodes.

Lecture Outline | **Progress**

- Introduction
- Syntax Analysers
- Context-Free Grammars
  - Derivations
  - Parse Tree
- **Types of Parsers**
- Top-Down Parsers
  - Ambiguity
  - Left Recursion
  - Left Factoring
- Classification of Top-Down Parsers
  - Recursive Decent Parser
  - Predictive Parsers
- LL Parsers
  - Using Parse Tables
  - Computation of First and Follow Sets
  - Construction of Parse Tables
- Limitations of Syntax Analysers

Types of Parsers

- Parsers can be **Top-down** or **Bottom-up**:
  - **Top-down parsers**
    - Build the parse-tree starting from the **root** until all the tokens are associated with a leaf on the parse tree.
  - **Bottom-up parsers**
    - Build the parse-tree starting from the **leaves**, assembling the tree fragments until the parse tree is complete.

Lecture Outline | **Progress**

- Introduction
- Syntax Analysers
- Context-Free Grammars
  - Derivations
  - Parse Tree
- Types of Parsers
- **Top-Down Parsers**
  - Ambiguity
  - Left Recursion
  - Left Factoring
- Classification of Top-Down Parsers
  - Recursive Decent Parser
  - Predictive Parsers
- LL Parsers
  - Using Parse Tables
  - Computation of First and Follow Sets
  - Construction of Parse Tables
- Limitations of Syntax Analysers

Top-Down Parsing

- Top-Down Parsers constructs from the Grammar
  - Which is free from **Ambiguity** and **Left Recursion**.
  - It allows a grammar which is **Left Factored**.
- Top Down Parsers uses **Leftmost Derivation** to construct a parse tree.

Lecture Outline | **Progress**

- Introduction
- Syntax Analysers
- Context-Free Grammars
  - Derivations
  - Parse Tree
- Types of Parsers
- **Top-Down Parsers**
  - **Ambiguity**
  - Left Recursion
  - Left Factoring
- Classification of Top-Down Parsers
  - Recursive Decent Parser
  - Predictive Parsers
- LL Parsers
  - Using Parse Tables
  - Computation of First and Follow Sets
  - Construction of Parse Tables
- Limitations of Syntax Analysers

Ambiguity (1 of 3)

- CFG is broken into
  - Ambiguous Grammar
  - Unambiguous grammars
- A grammar G is said to be **ambiguous** if it has **more than one parse tree** (*left or right derivation*) for at least one string.
  - For Example

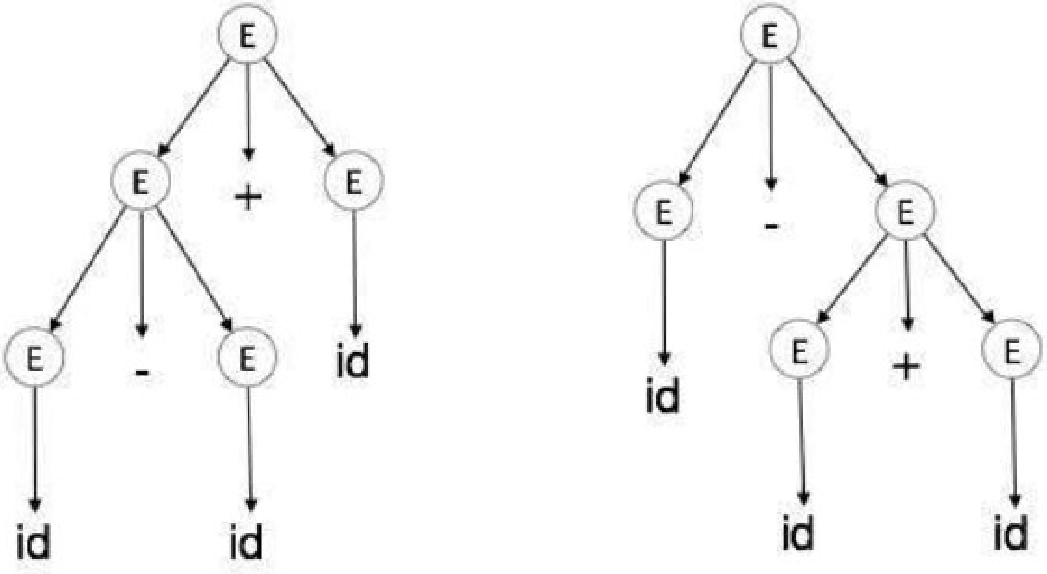
    $E \rightarrow E + E$

    $E \rightarrow E - E$

    $E \rightarrow id$

Ambiguity (2 of 3)

- For the string id + id – id, the above grammar generates two parse trees:



Ambiguity (3 of 3)

- The language generated by an ambiguous grammar is **Inherently Ambiguous**.
  - Ambiguity in grammar is **not good** for a compiler construction.
    - 2 or more parse trees structures for the same code implies two different executable programs.
    - Implies no unique structure for all its programs
- No automated method can **Detect** and **Remove** ambiguity, except by:
  1. **Re-writing** the whole grammar without ambiguity
  2. Setting and following **Associativity** and **Precedence** constraints.

Reducing Ambiguity

- These methods decrease the chances of ambiguity in a language or its grammar.
  - Associativity
  - Precedence

Associativity

- Given an operand with operators on both sides;
  - the effective operator on the operand is decided by the **associativity** of those operators.
  - If the operation is **Left-associative**, then the left operator will act on the operand
  - If the operation is **Right-associative**, the right operator will act on the operand.

Associativity: An Example

- **Given** the expression
  - id op id op id, where op can be +,-,/,* and ^
- **Left-associative Operators**
  - Addition, Multiplication, Subtraction, and Division operations
    - The expression will be evaluated as <u>(id + id)</u> + id
- **Right-associative Operators**
  - Exponentiation
    - The expression will be evaluated as id ^ <u>(id ^ id)</u>

Precedence

- Given that two operators share a common operand,
  - the **Precedence** of operators decides which will take the operand.
    - For Example 2+3*4 can have two different parse trees, (2+3)*4 and 2+(3*4).
  - The problem can be solved by precedence among operators.
    - Mathematically **Multiplication** has precedence over **Addition**, so the expression 2+3*4 will always be interpreted as: 2 + (3 * 4)

Operator Precedence

- C has 15 levels of precedence,
  - making its **expression grammar** more complex than that of most other languages
  - See: https://en.cppreference.com/w/c/language/operator_precedence

Lecture Outline | **Progress**

- Introduction
- Syntax Analysers
- Context-Free Grammars
  - Derivations
  - Parse Tree
- Types of Parsers
- **Top-Down Parsers**
  - Ambiguity
  - **Left Recursion**
  - Left Factoring
- Classification of Top-Down Parsers
  - Recursive Decent Parser
  - Predictive Parsers
- LL Parsers
  - Using Parse Tables
  - Computation of First and Follow Sets
  - Construction of Parse Tables
- Limitations of Syntax Analysers

Left Recursion

- A grammar is left-recursive if it has any non-terminal **'A'** whose derivation contains **'A'** itself as the left-most symbol.
  - S := Aα | β
  - A := Sd
- On encountering the same non-terminal in its derivation, the parser finds it hard to decide to when to stop parsing the left non-terminal and it might go into an **Infinite loop**
- Removed by Transforming the grammar to a *right-recursive one*
  - **A => βdA'**
  - **A' = αdA' | ε**

Lecture Outline | **Progress**

- Introduction
- Syntax Analysers
- Context-Free Grammars
  - Derivations
  - Parse Tree
- Types of Parsers
- **Top-Down Parsers**
  - Ambiguity
  - Left Recursion
  - **Left Factoring**
- Classification of Top-Down Parsers
  - Recursive Decent Parser
  - Predictive Parsers
- LL Parsers
  - Using Parse Tables
  - Computation of First and Follow Sets
  - Construction of Parse Tables
- Limitations of Syntax Analysers

Left Factoring

- It is possible that more than one grammar production rules has a **common prefix string**,
  - $A \Rightarrow \alpha\beta \mid \alpha\gamma \mid \ldots$
- Top-down parser cannot make a choice as to which of the production it should take to parse the string in hand
- In Left Refactoring one production for each common prefixes is obtained, while **the rest of the derivation is added by new productions**.
  - $A \Rightarrow \alpha A'$
  - $A' \Rightarrow \beta \mid \gamma \mid \ldots$

Transforming a Grammar for Predictive Parsing by Left Factoring

- Sometimes, we can transform a grammar to have this property:
  - For each non-terminal A find the longest prefix α common to two or more of its alternatives.
  - if $\alpha \neq \varepsilon$ then replace all of the A productions
    $$A \rightarrow \alpha\beta1 \mid \alpha\beta2 \mid \ldots \mid \alpha\beta n$$
    with
    $$A \rightarrow \alpha \, \mathbf{A}'$$
  - $$\mathbf{A}' \rightarrow \beta1 \mid \beta2 \mid \ldots \mid \beta n$$
    where **A´** is fresh
  - Repeat until no two alternatives for a single non-terminal have a common prefix.

Transforming a Grammar for Predictive Parsing by Left Factoring

- Given the Grammar
  - E → T + E | T
  - T → int | int * T | (E)
- Hard to predict because
  - For "*T*" two productions start with "***int***"
  - Also for "**E**" it is not clear how to predict

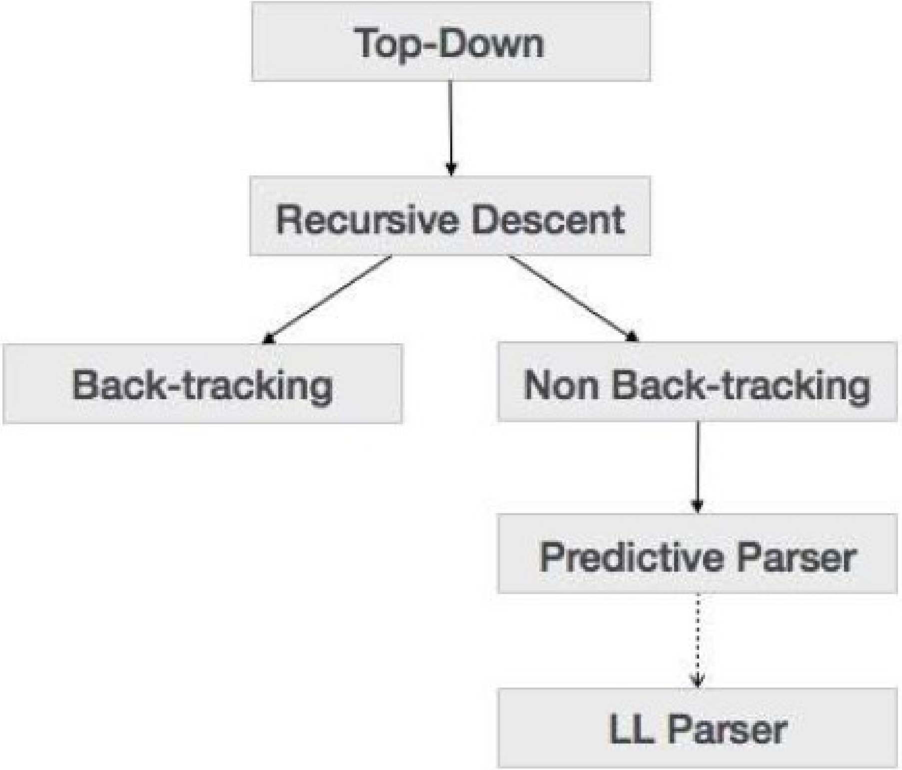
Performing Left-Factoring

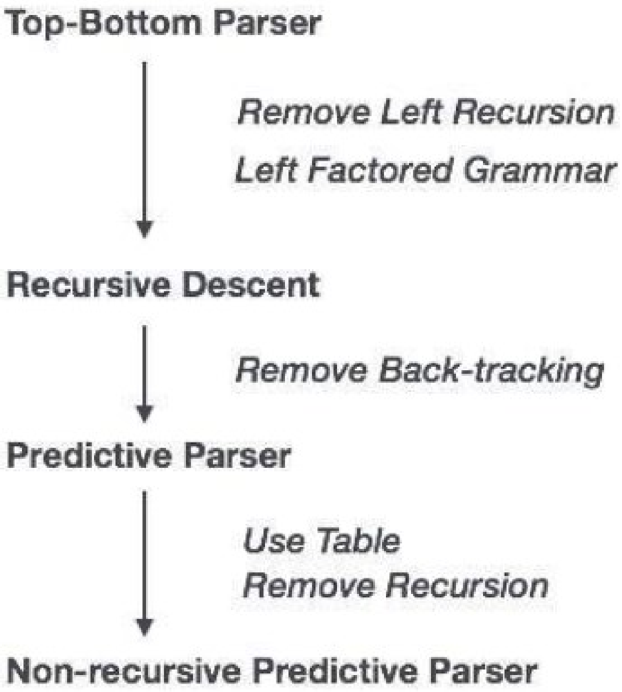
- For the Grammar
  - E → T + E | T
  - T → int | int * T | (E)
- Factor out common prefixes of productions
  - E → T X
  - X → + E | ε
  - T → (E) | int Y
  - Y → * T | ε

Lecture Outline | Progress

- Introduction
- Syntax Analysers
- Context-Free Grammars
  - Derivations
  - Parse Tree
- Types of Parsers
- Top-Down Parsers
  - Ambiguity
  - Left Recursion
  - Left Factoring
- **Classification of Top-Down Parsers**
  - **Recursive Decent Parser**
  - Predictive Parsers
- LL Parsers
  - Using Parse Tables
  - Computation of First and Follow Sets
  - Construction of Parse Tables
- Limitations of Syntax Analysers

Classification of Top-Down Parsing





Lecture Outline | **Progress**

- Introduction
- Syntax Analysers
- Context-Free Grammars
  - Derivations
  - Parse Tree
- Types of Parsers
- Top-Down Parsers
  - Ambiguity
  - Left Recursion
  - Left Factoring
- **Classification of Top-Down Parsers**
  - **Recursive Decent Parser**
  - Predictive Parsers
- LL Parsers
  - Using Parse Tables
  - Computation of First and Follow Sets
  - Construction of Parse Tables
- Limitations of Syntax Analysers

Recursive Descent Parsing

- Recursive descent is a top-down parsing technique that constructs the parse tree from the **top** and the input is read from **left to right**.
  - It uses **procedures (Routines)** for every terminal and non-terminal entity.
- This parsing technique **Recursively Parses** the input to make a parse tree, which may or may not require back-tracking.
  - But the grammar associated with it (if not left factored) cannot avoid back-tracking.

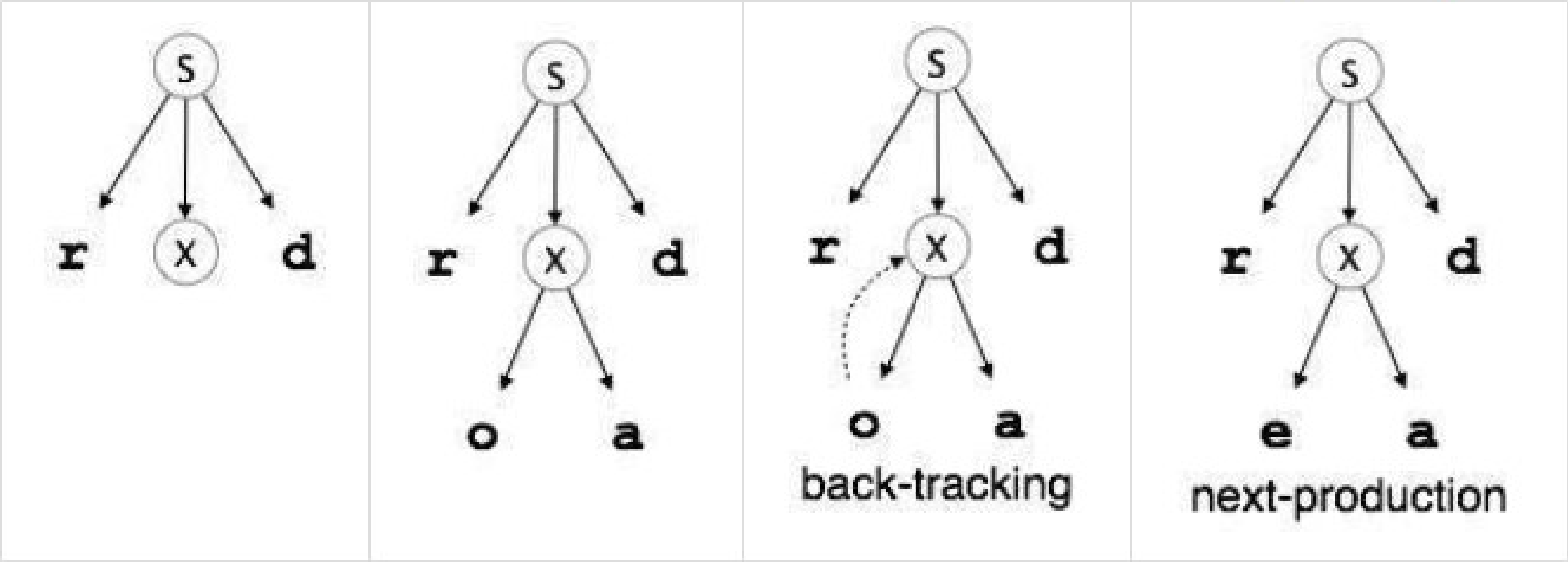
Recursive Descent Parsing

- However, a form of recursive-descent parsing that does not require any back-tracking is known as **Predictive Parsing**.
  - This parsing technique is regarded recursive as it uses **Context-Free Grammar** which is recursive in nature.

Back-tracking Example

- For Example, given the following CFG:
  - **S → rXd | rZd**
  - **X → oa | ea**
  - **Z → ai**
- For an input string: **read**
  - a top-down parser, will behave like this:

Parsing Visualization



Simple Expression Grammar

- Given the Simple Expression grammar

1.	<goal>	::=	<expr>
2.	<expr>	::=	<expr> + <term>
3.		|	<expr> – <term>
4.		|	<term>
5.	<term>	::=	<term> * <factor>
6.		|	<term> / <factor>
7.		|	<factor>
8.	<factor>	::=	`num`
9.		|	`id`

Consider the input string

`x — 2 * y`

Top-down derivation

1.	<goal>	::=	<expr>
2.	<expr>	::=	<expr> + <term>
3.		|	<expr> – <term>
4.		|	<term>
5.	<term>	::=	<term> * <factor>
6.		|	<term> / <factor>
7.		|	<factor>
8.	<factor>	::=	num
9.		|	id

Prod'n	Sentential form	Input
–	⟨goal⟩	↑x – 2 * y
1	⟨expr⟩	↑x – 2 * y
2	⟨expr⟩ + ⟨term⟩	↑x – 2 * y
4	⟨term⟩ + ⟨term⟩	↑x – 2 * y
7	⟨factor⟩ + ⟨term⟩	↑x – 2 * y
9	id + ⟨term⟩	↑x – 2 * y
–	id + ⟨term⟩	x ↑– 2 * y
–	⟨expr⟩	↑x – 2 * y
3	⟨expr⟩ – ⟨term⟩	↑x – 2 * y
4	⟨term⟩ – ⟨term⟩	↑x – 2 * y
7	⟨factor⟩ – ⟨term⟩	↑x – 2 * y
9	id – ⟨term⟩	↑x – 2 * y
–	id – ⟨term⟩	x ↑– 2 * y
–	id – ⟨term⟩	x – ↑2 * y
7	id – ⟨factor⟩	x – ↑2 * y
8	id – num	x – ↑2 * y
–	id – num	x – 2 ↑* y
–	id – ⟨term⟩	x – ↑2 * y
5	id – ⟨term⟩ * ⟨factor⟩	x – ↑2 * y
7	id – ⟨factor⟩ * ⟨factor⟩	x – ↑2 * y
8	id – num * ⟨factor⟩	x – ↑2 * y
–	id – num * ⟨factor⟩	x – 2 ↑* y
–	id – num * ⟨factor⟩	x – 2 * ↑y
9	id – num * id	x – 2 * ↑y
–	id – num * id	x – 2 * y

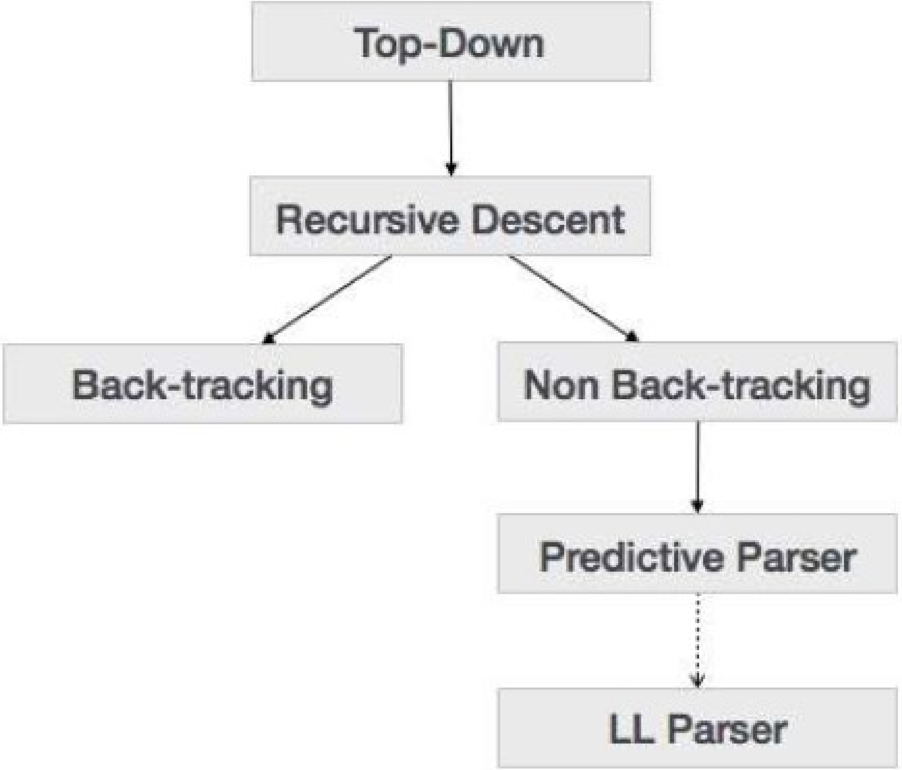
Summary of Recursive Descent Parsing

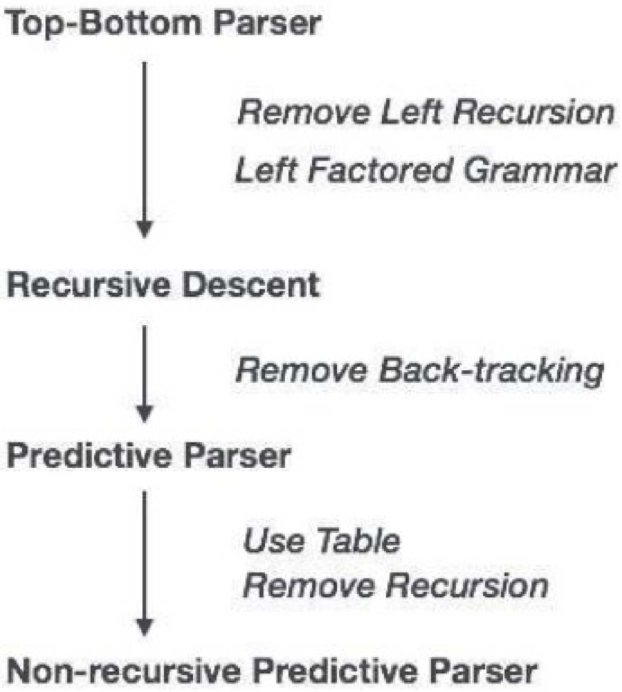
- It is a **simple** and **general** parsing strategy
- Left-recursion must be **eliminated first**
  - but that can be done automatically
- Unpopular because of **Back-tracking**
- Thought to be **too inefficient**
- In practice, backtracking is eliminated by **restricting the grammar**

Lecture Outline | **Progress**

- Introduction
- Syntax Analysers
- Context-Free Grammars
  - Derivations
  - Parse Tree
- Types of Parsers
- Top-Down Parsers
  - Ambiguity
  - Left Recursion
  - Left Factoring
- **Classification of Top-Down Parsers**
  - **Recursive Decent Parser**
  - **Predictive Parsers**
- LL Parsers
  - Using Parse Tables
  - Computation of First and Follow Sets
  - Construction of Parse Tables
- Limitations of Syntax Analysers

Classification of Top-Down Parsing





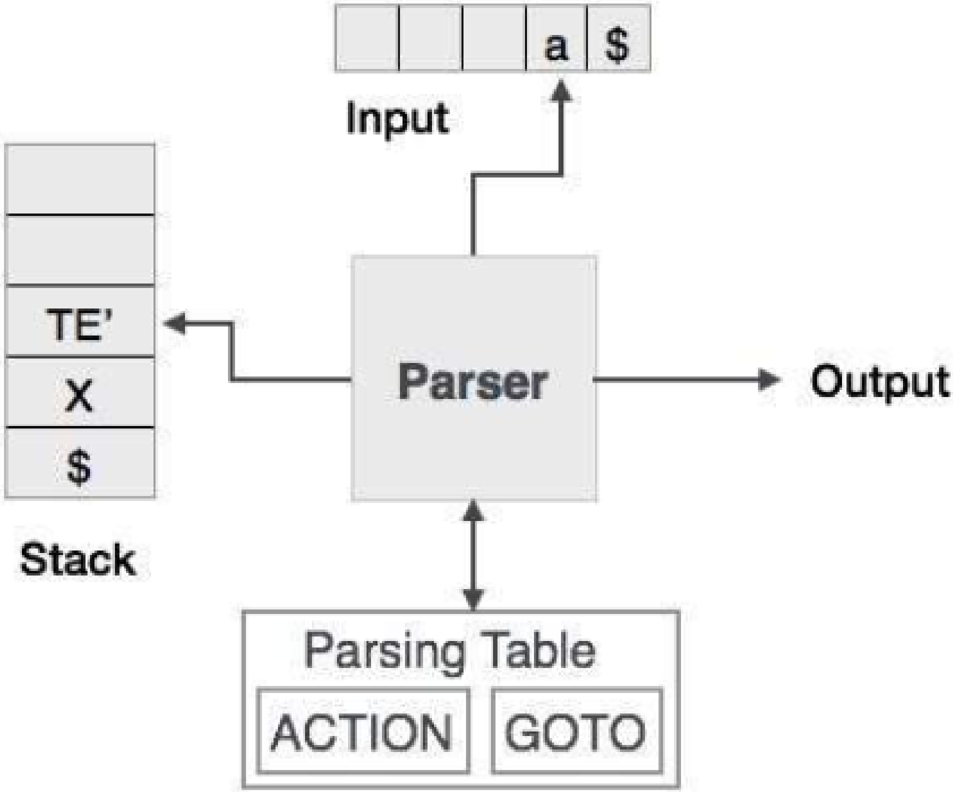
Predictive Parser

- **Predictive parser** is a **Recursive Descent Parser**,
  - It has the capability to **predict** which production is to be used to replace the **Input String**.
- The predictive parser **does not suffer** from **Backtracking**.
  - To accomplish its tasks, the predictive parser uses a **look-ahead pointer**, which points to the next input symbols.
- To make the parser back-tracking free,
  - The **Predictive Parser** puts some constraints on the Grammar;
  - Accepts only a class of grammar known as **LL(k) Grammar**.

Predictive Parser

- Predictive parsing uses a **stack** and a **parsing table** to parse the input and generate a parse tree.
  - Both the **stack** and the **input** contains an end symbol **"$"** to denote that the stack is empty and the input is consumed.
  - The parser refers to the **Parsing Table** to take any decision on the input and stack element combination.
- A grammar must be **left-factored** before use for **predictive parsing.**

Graphical Depiction



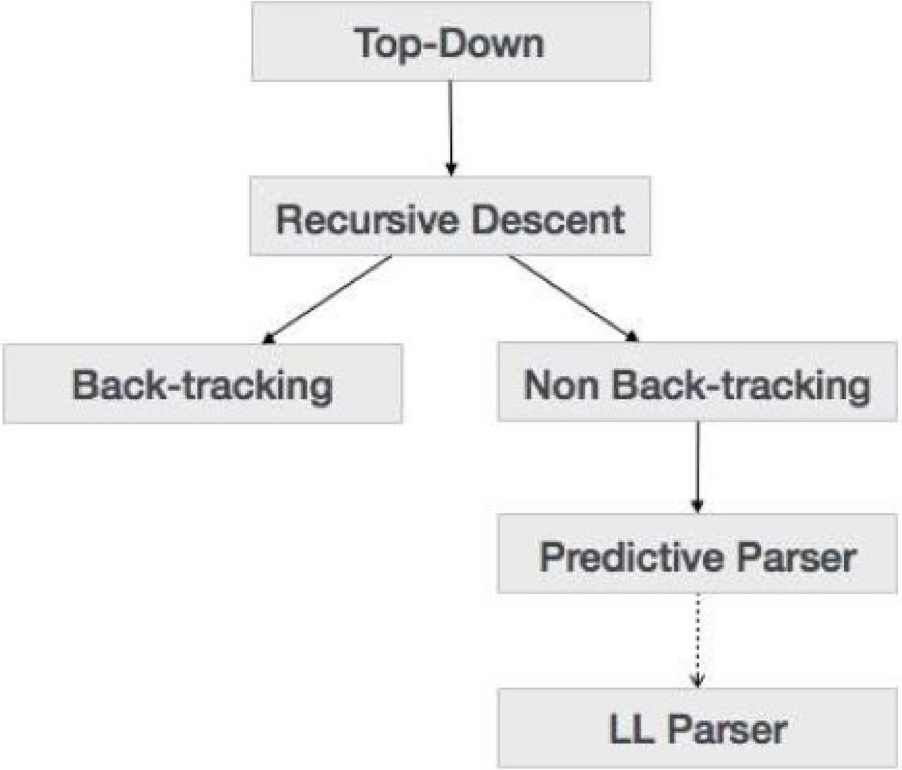
Recursive Descent Vs Predictive Parsing

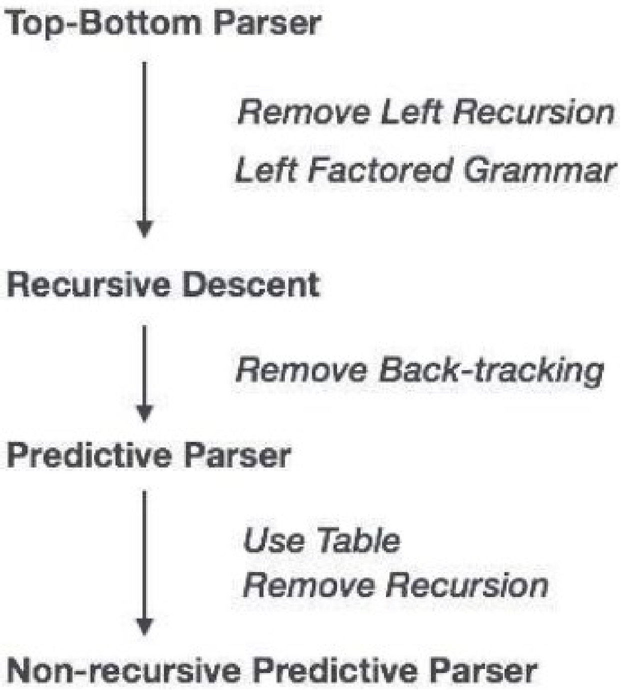
- In **Recursive Descent Parsing** with backtracking:
  - The parser may have more than one production to choose from for a single instance of input.
- In **Predictive Parsing**:
  - The parser has **At Most One Production** to choose.
    - It is possible that there might be instances where there is no production matches the input string.
    - This makes the parsing procedure to fail.

Lecture Outline | **Progress**

- Introduction
- Syntax Analysers
- Context-Free Grammars
  - Derivations
  - Parse Tree
- Types of Parsers
- Top-Down Parsers
  - Ambiguity
  - Left Recursion
  - Left Factoring
- Classification of Top-Down Parsers
  - Recursive Decent Parser
  - Predictive Parsers
- **LL Parsers**
  - Using Parse Tables
  - Computation of First and Follow Sets
  - Construction of Parse Tables
- Limitations of Syntax Analysers

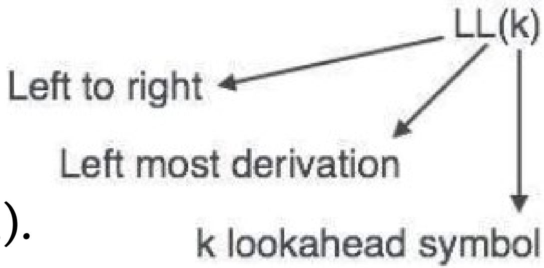
Classification of Top-Down Parsing





LL Parser

- LL parser is denoted as LL(k).
  - First L means it's parsing the input from **Left** to **Right**,
  - Second L stands for **Left-most Derivation**
  - **k** itself represents the number of look aheads
- Generally k = 1, so LL(k) may also be written as LL(1).
- An LL Parser accepts LL grammar.
  - LL grammar is a subset of **Context-free Grammar** but with some restrictions to get the simplified version.
  - The grammar must be from free from Left Recursion, Common Prefix, and Ambiguity.
- This parser is **Non-Recursive**.



Definition of a LL(1) Grammar

- A grammar G is LL(1) if **A → α | β** are two distinct productions of G:
  - for non terminal, both **α** and **β** derive strings beginning with **A**.
  - at most one of **α** and **β** can derive empty string.
  - if **β → t**, then **α** does not derive any string beginning with a terminal in **FOLLOW(A)**.
- LL grammar can be implemented using **Recursive-descent** or **Table-driven Algorithms**

LL Parsers

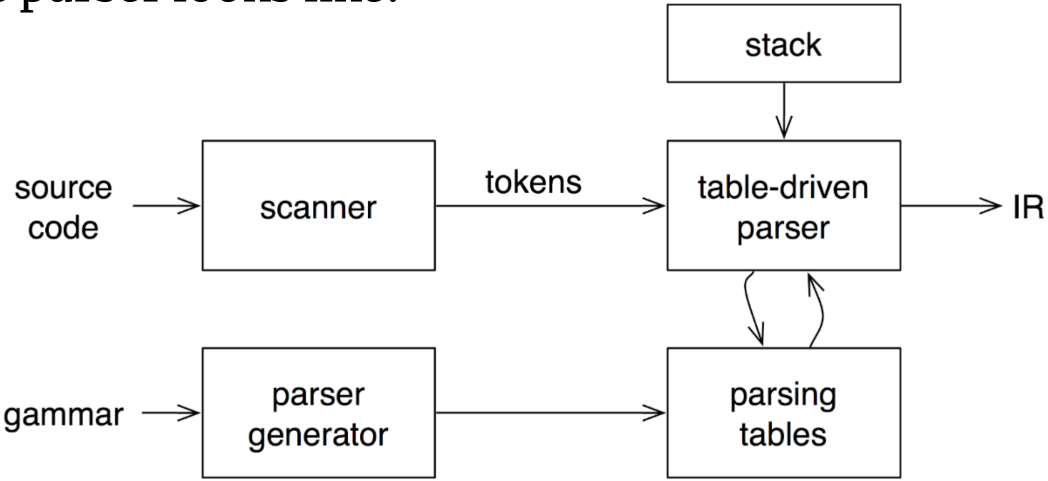
- Recall:
  - LL grammar can be implemented using **Recursive-descent** or **Table-driven Algo's**
- **Recursive descent** encodes state information in its run- time stack, or call stack.
  - Using recursive procedure calls to implement a stack abstraction may not be particularly efficient.
- This suggests other implementation methods:
  - explicit stack, hand-coded parser
  - stack-based, **table-driven** parser

Lecture Outline | **Progress**

- Introduction
- Syntax Analysers
- Context-Free Grammars
  - Derivations
  - Parse Tree
- Types of Parsers
- Top-Down Parsers
  - Ambiguity
  - Left Recursion
  - Left Factoring
- Classification of Top-Down Parsers
  - Recursive Decent Parser
  - Predictive Parsers
- **LL Parsers**
  - **Using Parse Tables**
  - Computation of First and Follow Sets
  - Construction of Parse Tables
- Limitations of Syntax Analysers

LL Parsers

- A predictive parser looks like:



- Rather than writing code, Tables can be built
  - However, building tables can be automated

Using Parsing Tables

- Method similar to recursive descent, except
  - For each non-terminal ***S***
  - We look at the next token ***a***
  - And chose the production shown at ***[S, a]***
- Use a **Stack** to keep track of pending non-terminals
  - Reject when we encounter an error state
  - Accept when we encounter end-of-input
- To construct the **Parse Table**
  - Compute **First** and **Follow** Sets

Lecture Outline | Progress

- Introduction
- Syntax Analysers
- Context-Free Grammars
  - Derivations
  - Parse Tree
- Types of Parsers
- Top-Down Parsers
  - Ambiguity
  - Left Recursion
  - Left Factoring
- Classification of Top-Down Parsers
  - Recursive Decent Parser
  - Predictive Parsers
- **LL Parsers**
  - Using Parse Tables
  - **Computation of First and Follow Sets**
  - Construction of Parse Tables
- Limitations of Syntax Analysers

First and Follow Sets

- **First Sets Algorithm:** Look at the definition of **FIRST(α)** set:
  - if α is a **terminal**, then **FIRST(α) = { α }**
  - if α is a **non-terminal** and **α → ε** is a production, then **FIRST(α) = { ε }.**
  - if α is a **non-terminal** and **$\alpha \rightarrow \gamma_1 \gamma_2 \gamma_3 \ldots \gamma_n$** and any **FIRST(γ)** contains ***t*** then ***t*** is in **FIRST(α)**

First and Follow Sets

- **Follow Set Algorithm**
  - if **α** is a start symbol, then **FOLLOW(α) = $**
  - if **α** is a non-terminal and has a production **α → AB**, then **FIRST(B)** is in **FOLLOW(A)** except **ε**.
  - if **α** is a non-terminal and has a production **α → AB**, where **B ε**, then **FOLLOW(A)** is in **FOLLOW(α)**.
    - Follow set can be seen as: **FOLLOW(α) = { t | S *αt*}**

Some Example Workings

$S \rightarrow aABb$
$A \rightarrow c \mid \varepsilon$
$B \rightarrow d \mid \varepsilon$

$S \rightarrow ACB \mid CbB \mid Ba$
$A \rightarrow da \mid BC$
$B \rightarrow g \mid \varepsilon$
$C \rightarrow h \mid \varepsilon$

$S \rightarrow aBDh$
$B \rightarrow cC$
$C \rightarrow bC \mid \varepsilon$
$D \rightarrow EF$
$E \rightarrow g \mid \varepsilon$
$F \rightarrow f \mid \varepsilon$

$E \rightarrow TE'$
$E' \rightarrow +TE' \mid \varepsilon$
$T \rightarrow FT'$
$T' \rightarrow *FT' \mid \varepsilon$
$F \rightarrow id \mid (E)$

Workings 1

S → aABb
A → c | ε
B → d | ε

Production	FIRST	FOLLOW
S	{a}	{$}
A	{c, ε}	{d , b}
B	{d , ε	{b}

Workings 2

$S \rightarrow ACB \mid CbB \mid Ba$
$A \rightarrow da \mid BC$
$B \rightarrow g \mid \varepsilon$
$C \rightarrow h \mid \varepsilon$

Production	FIRST	FOLLOW
S	{ d , g , h , b , a , ε }	{ $ }
A	{ d , g , h , ε }	{ h , g , $ }
B	{ g , ε }	{ $, a , h , g }
C	{ h , ε }	{ g , $, b , h }

Workings 3

E	**→ TE'**
E'	**→ +TE' | ε**
T	**→ FT'**
T'	**→ *FT' | ε**
F	**→ id | (E)**

Production	FIRST	FOLLOW
E	{ id , (}	{ $,) }
E'	{ + , ε }	{ $,) }
T	{ id , (}	{ + , $,) }
T'	{ * , ε }	{ + , $,) }
F	{ id , (}	{ * , + , $,) }

Workings 4

S →	aBDh
B →	cC
C →	bC | ε
D →	EF
E →	g | ε
F →	f | ε

Production	FIRST	FOLLOW
S	{ a }	{$}
B	{ c }	{ g , f , h }
C	{ b , ε }	{ g , f , h }
D	{ g , f , ε }	{h}
E	{ g , ε }	{ f, h }
F	{ f , ε }	{h}

Lecture Outline | **Progress**

- Introduction
- Syntax Analysers
- Context-Free Grammars
  - Derivations
  - Parse Tree
- Types of Parsers
- Top-Down Parsers
  - Ambiguity
  - Left Recursion
  - Left Factoring
- Classification of Top-Down Parsers
  - Recursive Decent Parser
  - Predictive Parsers
- **LL Parsers**
  - Using Parse Tables
  - Computation of First and Follow Sets
  - **Construction of Parse Tables**
- Limitations of Syntax Analysers

Table-Driven Parser| Example 1

- Construct a **Parse Table** for the Following Grammar:
  - Generate the **First Sets**
  - Generate the **Follow Sets**
  - Construct the **Parse Table**

$$
\begin{aligned}
E &\rightarrow T\ X \\
X &\rightarrow +\ E \mid \varepsilon \\
T &\rightarrow (\ E\ ) \mid \text{int}\ Y \\
Y &\rightarrow *\ T \mid \varepsilon
\end{aligned}
$$

Table-Driven Parser | Example 1

$$
\begin{aligned}
E &\rightarrow T\ X \\
X &\rightarrow +\ E \mid \varepsilon \\
T &\rightarrow (\ E\ ) \mid \text{int}\ Y \\
Y &\rightarrow *\ T \mid \varepsilon
\end{aligned}
$$

- First sets
  - First(( ) = { (}
  - First( )) = {) }
  - First(int) = { int }
  - First(+) = { + }
  - First(*) = { * }
  - First(T) = {int, (}
  - First(E) = {int, (}
  - First(X) = {+, ε }
  - First(Y) = {*, ε }

Follow sets

- Follow(+) = { int, (}
- Follow(*) = { int, (}
- Follow(( ) = { int, (}
- Follow(E) = {), $}
- Follow(X) = {$,) }
- Follow(T) = {+,) , $}
- Follow( )) = {+,) , $}
- Follow(Y) = {+,) , $}
- Follow(int) = {*, +,) , $}

Table-Driven Parser | Example

- Given the Left-factored Grammar

$$E \rightarrow T\ X$$
$$X \rightarrow +\ E \mid \varepsilon$$
$$T \rightarrow (\ E\ ) \mid \text{int}\ Y$$
$$Y \rightarrow *\ T \mid \varepsilon$$

	int	*	+	(	)	$
E	T X			T X		
X			+ E		ε	ε
T	int Y			(E)		
Y		* T	ε		ε	ε

Table-Driven Parser| Example 2

- Construct a **Parse Table** for the Following Grammar:
  - Generate the **First Sets**
  - Generate the **Follow Sets**
  - Construct the **Parse Table**

$$
\begin{aligned}
S &\rightarrow E \\
E &\rightarrow TE' \\
E' &\rightarrow +E \mid -E \mid \varepsilon \\
T &\rightarrow FT' \\
T' &\rightarrow * T \mid / T \mid \varepsilon \\
F &\rightarrow \mathtt{num} \mid \mathtt{id}
\end{aligned}
$$

Table-Driven Parser| Example 2

S → E
E → TE´
E´ → +E | —E | ε
T → FT´
T´ → * T | / T | ε
F → num | id

	id	num	+	−	*	/	$
S	$S \to E$	$S \to E$	−	−	−	−	−
E	$E \to TE'$	$E \to TE'$	−	−	−	−	−
E'	−	−	$E' \to +E$	$E' \to -E$	−	−	$E' \to \varepsilon$
T	$T \to FT'$	$T \to FT'$	−	−	−	−	−
T'	−	−	$T' \to \varepsilon$	$T' \to \varepsilon$	$T' \to *T$	$T' \to /T$	$T' \to \varepsilon$
F	$F \to \mathtt{id}$	$F \to \mathtt{num}$	−	−	−	−	−

	FIRST	FOLLOW
S	$\{\mathtt{num}, \mathtt{id}\}$	$\{\$\}$
E	$\{\mathtt{num}, \mathtt{id}\}$	$\{\$\}$
E'	$\{\varepsilon, +, -\}$	$\{\$\}$
T	$\{\mathtt{num}, \mathtt{id}\}$	$\{+, -, \$\}$
T'	$\{\varepsilon, *, /\}$	$\{+, -, \$\}$
F	$\{\mathtt{num}, \mathtt{id}\}$	$\{+, -, *, /, \$\}$
id	$\{\mathtt{id}\}$	−
num	$\{\mathtt{num}\}$	−
*	$\{*\}$	−
/	$\{/\}$	−
+	$\{+\}$	−
−	$\{-\}$	−

Table-Driven Parser| Example 3

- Construct a **Parse Table** for the Following Grammar:
  - Generate the **First Sets**
  - Generate the **Follow Sets**
  - Construct the **Parse Table**

1.	<goal>	::=	<expr>
2.	<expr>	::=	<term> <expr´>
3.	<expr´>	::=	+ <expr>
4.		|	- <expr>
5.		|	ε
6.	<term>	::=	<factor> <term´>
7.	<term´>	::=	* <term>
8.		|	/ <term>
9.		|	ε
10.	<factor>	::=	num
11.		|	id

Table-Driven Parser| Example 3

Our expression grammar :

1.	<goal>	::=	<expr>
2.	<expr>	::=	<term> <expr´>
3.	<expr´>	::=	+ <expr>
4.		|	- <expr>
5.		|	ε
6.	<term>	::=	<factor> <term´>
7.	<term´>	::=	* <term>
8.		|	/ <term>
9.		|	ε
10.	<factor>	::=	num
11.		|	id

Its parse table

	id	num	+	−	*	/	$[†]
⟨goal⟩	1	1	–	–	–	–	–
⟨expr⟩	2	2	–	–	–	–	–
⟨expr′⟩	–	–	3	4	–	–	5
⟨term⟩	6	6	–	–	–	–	–
⟨term′⟩	–	–	9	9	7	8	9
⟨factor⟩	11	10	–	–	–	–	–

[†] we use $ to represent EOF

Lecture Outline | **Progress**

- Introduction
- Syntax Analysers
- Context-Free Grammars
  - Derivations
  - Parse Tree
- Types of Parsers
- Top-Down Parsers
  - Ambiguity
  - Left Recursion
  - Left Factoring
- Classification of Top-Down Parsers
  - Recursive Decent Parser
  - Predictive Parsers
- LL Parsers
  - Using Parse Tables
  - Computation of First and Follow Sets
  - Construction of Parse Tables
- **Limitations of Syntax Analysers**

Limitations of Syntax Analyzers

- **Syntax Analysers** receive their inputs, in the form of tokens, from lexical analysers.
  - Lexical Analysers are responsible for the validity of a token supplied to the syntax analyser.

Limitations of Syntax Analyzers

- Syntax analysers have the following drawbacks:
  - It cannot determine if a token is valid
  - It cannot determine if a token is declared before it is being used
  - It cannot determine if a token is initialized before it is being used
  - It cannot determine if an operation performed on a token type is valid or not.
  - These tasks are accomplished by the **Semantic Analyser**

Lecture Outline | **End**

- Introduction
- Syntax Analysers
- Context-Free Grammars
  - Derivations
  - Parse Tree
- Types of Parsers
- Top-Down Parsers
  - Ambiguity
  - Left Recursion
  - Left Factoring
- Classification of Top-Down Parsers
  - Recursive Decent Parser
  - Predictive Parsers
- LL Parsers
  - Using Parse Tables
  - Computation of First and Follow Sets
  - Construction of Parse Tables
- Limitations of Syntax Analysers