University of Mosul
College Computer Science & Mathematics
Department of Computer Science

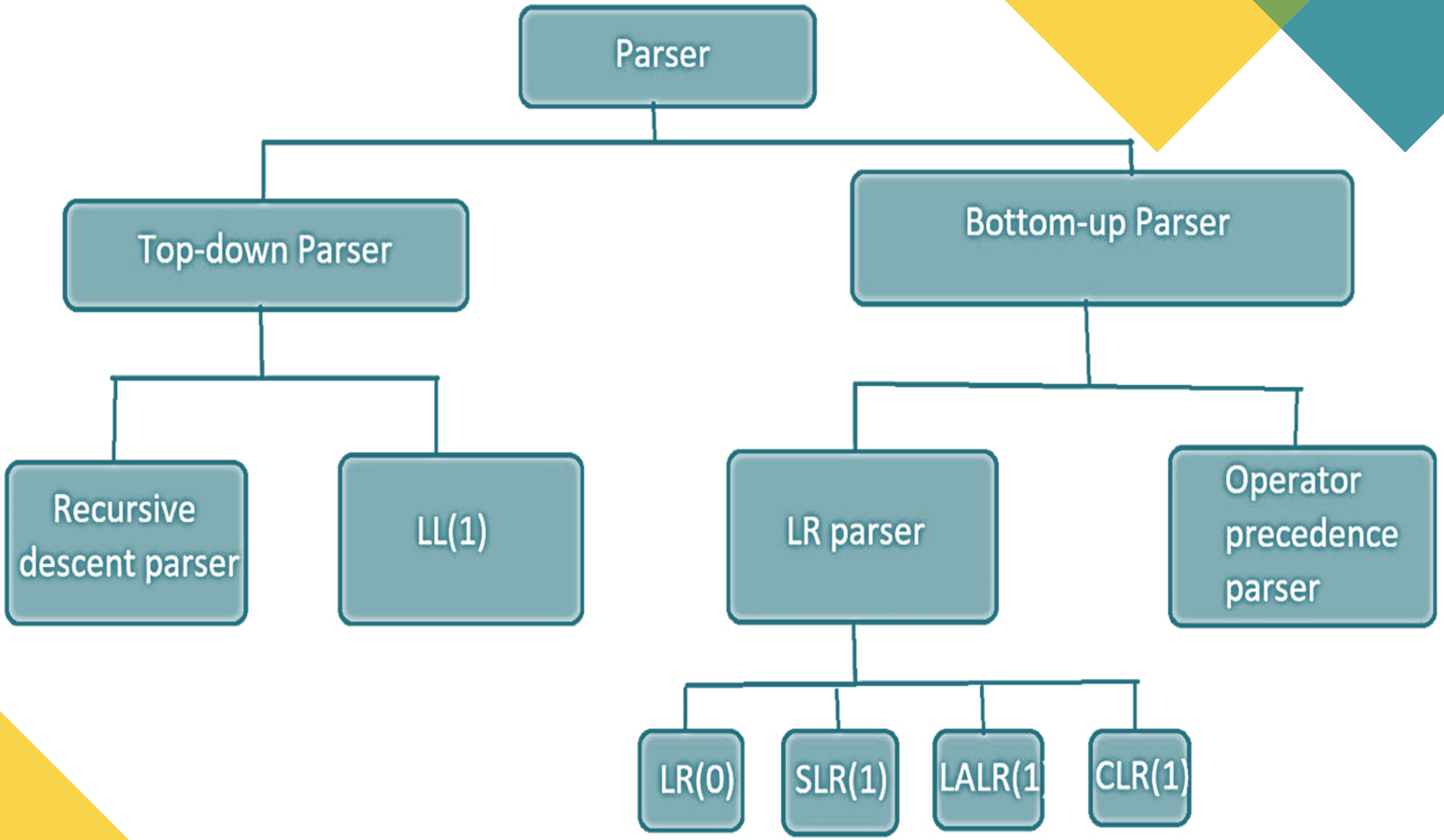
Compiler 2

Lecture 1:

Types of parsers in compiler design

BY: Assist Prof. Dr. Ielaf O Abdul Majjed

The **parser** is that phase of the compiler which takes a token string as input and with the help of existing grammar, converts it into the corresponding Intermediate Representation. The parser is also known as *Syntax Analyzer*



Types of Parser:

The parser is mainly classified into two categories, i.e. Top-down Parser, and Bottom-up Parser. These are explained below:

1. *Top-Down Parser:*

The top-down parser is the parser that generates parse for the given input string with the help of grammar productions by expanding the non-terminals i.e. it starts from the start symbol and ends on the terminals. It uses left most derivation.

Further Top-down parser is classified into **2 types**: *Recursive descent parser*, and *Non-recursive descent parser*.

Recursive descent parser is also known as the Brute force parser or the backtracking parser. It basically generates the parse tree by using brute force and backtracking.

Non-recursive descent parser is also known as LL(1) parser or predictive parser or without backtracking parser or dynamic parser. It uses a parsing table to generate the parse tree instead of backtracking.

2. Bottom-up Parser:

Bottom-up Parser is the parser that generates the parse tree for the given input string with the help of grammar productions by compressing the non-terminals i.e. it starts from terminals and ends on the start symbol. It uses the reverse of the rightmost derivation.

Further Bottom-up parser is classified into two types: LR parser, and Operator precedence parser.

LR parser is the bottom-up parser that generates the parse tree for the given string by using unambiguous grammar. It follows the reverse of the rightmost derivation.

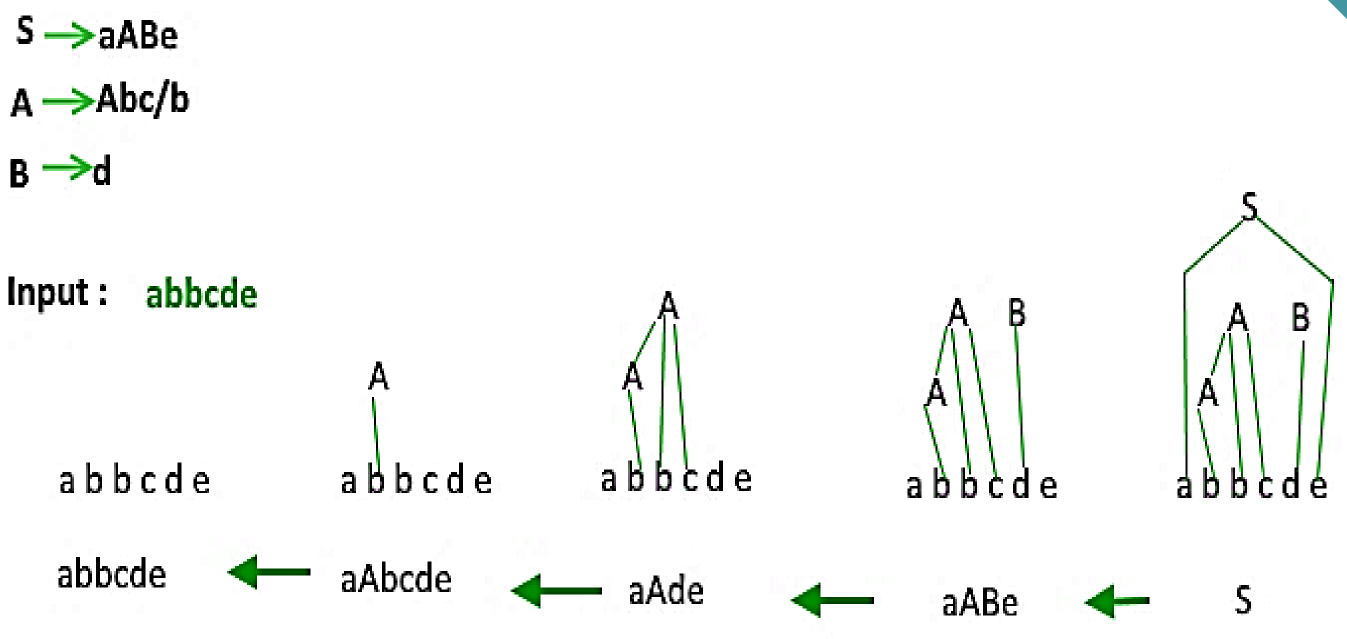
LR parser is of four types:

a- LR(0) b- SLR(1) c-LALR(1) d-CLR(1)

Operator precedence parser generates the parse tree form given grammar and string but the only condition is two consecutive non-terminals and epsilon never appear on the right-hand side of any production.

Bottom Up Parsers / Shift Reduce Parsers

Bottom up parsers start from the sequence of terminal symbols and work their way back up to the start symbol by repeatedly replacing grammar rules' right hand sides by the corresponding non-terminal. This is the reverse of the derivation process, and is called "reduction"



Example:1 consider the grammar

S→ aABe

A→ Abc|b

B→ d

“**abbcde**” can be reduced to S by the following steps:

Sol:

abbcde

aAbcde

aAde

aABe

S

Example:2 consider the grammar

S→ aABe

A→ Abc|bc

B→ dd

“**abcbcdde**” can be reduced to S by the following steps:

Sol:

abcbcdde

aAbcdde

aAdde

aABe

S

Example:3 Using the following arithmetic grammar

E → E+T | T

T → T*F | F

F → (E) | id

Illustrates the bottom-up parse for string w= id * id

The reductions will be discussed in terms of the sequence of strings

$\underline{\textbf{id}}\ \textbf{*}\ \textbf{id}$

$\underline{\textbf{F}}\ \textbf{*}\ \textbf{id}$

$\textbf{T}\ \textbf{*}\ \underline{\textbf{id}}$

$\underline{\textbf{T * F}}$ **handle**

$\underline{\textbf{T}}$ **handle pruning start**

E start

The following *derivation* corresponds to the parse

E → T

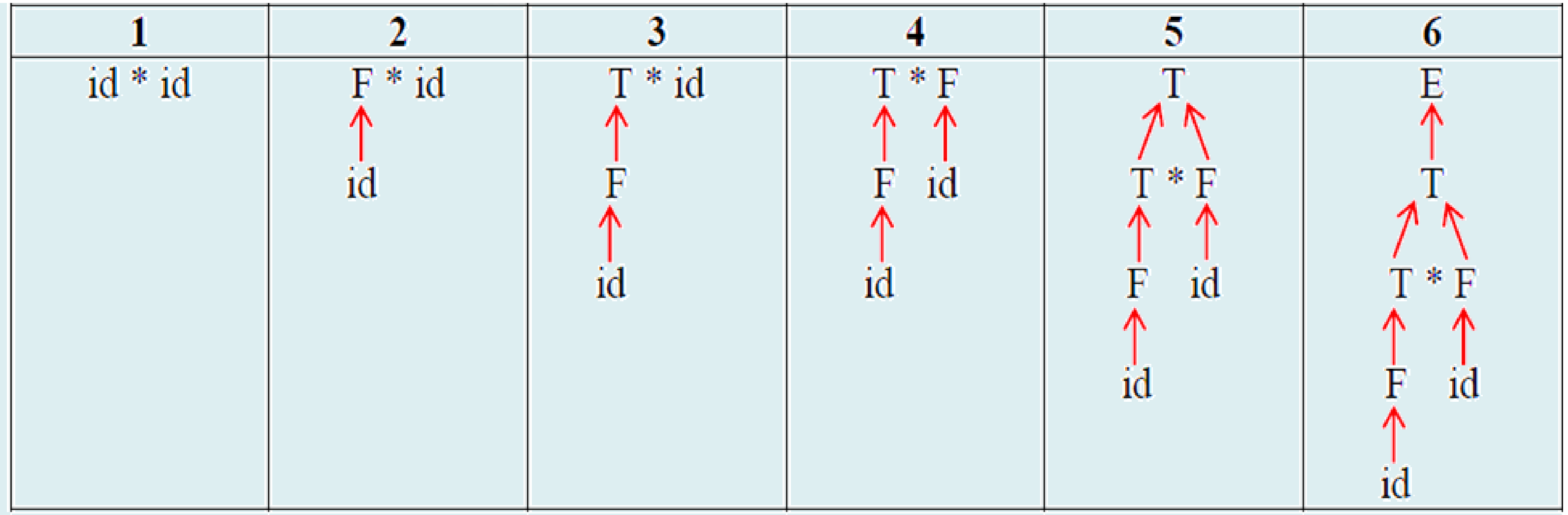
→T * F

→ T * id

→ F * id

→ id * id

This derivation is in fact a ***RightMost Derivation*** (RMD):



We can think of bottom-up parsing as the process of "reducing" a string w to the start symbol of the grammar.

At each reduction step, a specific substring matching the body of a production is replaced by the *nonterminal* at the head of that production.

The key decisions during bottom-up parsing are *about when to reduce* and about *what production to apply*, as the parse proceeds.

The grammar is the expression grammar in example 3:

The reductions will be discussed in terms of the sequence of strings

id * id →   F * id →   T * id →   T * F →   T →   E … (reductions)

By definition, a ***Reduction*** is the reverse of a step in a derivation (recall that in a derivation, a nonterminal in a sentential form is replaced by the body of one of its productions).

The goal of Bottom-Up Parsing is therefore to construct a derivation in reverse. The following derivation corresponds to the parse in example 3:

E → T → T * F → T * id → F * id → id * id … (RMD derivation**)**

Handle Pruning:

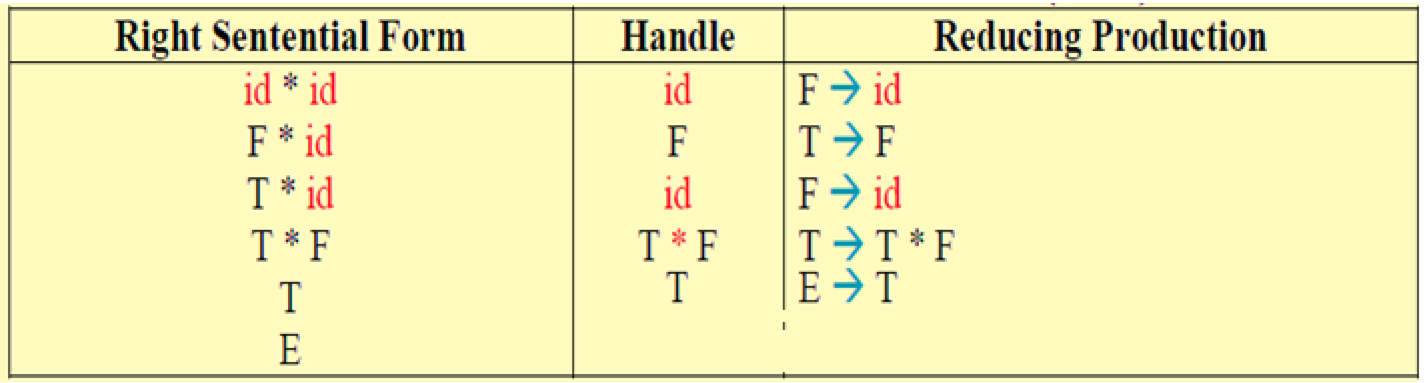
Bottom-up parsing during a left-to-right scan of the input constructs a rightmost derivation in reverse. Informally, a "**handle**" is a substring that matches the body of a production, and whose reduction represents one step along the reverse of a rightmost derivation.

For example, adding subscripts to the tokens **id** for clarity, the handles during the parse of **id_1** * **id_2** according to the expression grammar

E → E+T | T
T → T*F | F
F → (E) | id

Although **T** is the body of the production **E → T**, the symbol **T** is not a handle in the sentential form **T * id_2**.
If **T** were indeed replaced by **E**, we would get the string **E * id_2**, which cannot be derived from the start symbol **E**.
Thus, the leftmost substring that matches the body of some production need not be a handle.

E → T → T * F → F * F → F * id_1 → id_1 * id_2 … (derivation**)**



Right Sentential Form	Handle	Reducing Production
id * id	id	F → id
F * id	F	T → F
T * id	id	F → id
T * F	T * F	T → T * F
T	T	E → T
E		

Example: consider the grammar:

E' → E
E → E+T | E–T | T
T → (E) | id

By using RMD derivation **derive id + (id – id)**

Solution:

E' → E

→ E + T

→ E + (E)

→ E + (E – T)

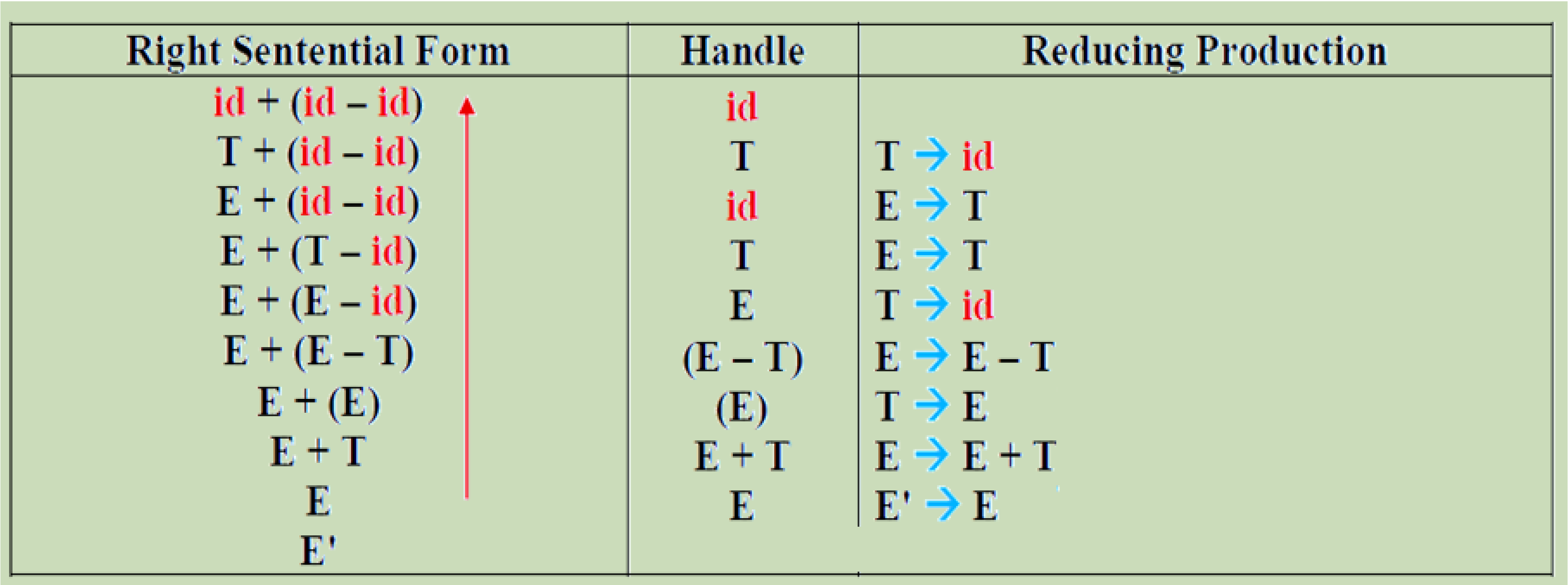
→ E + (E – id)

→ E + (T – id)

→ E + (id – id)

→ T + (id – id)

→ id + (id – id)



Right Sentential Form	Handle	Reducing Production
id + (id – id)	id	
T + (id – id)	T	T → id
E + (id – id)	id	E → T
E + (T – id)	T	E → T
E + (E – id)	E	T → id
E + (E – T)	(E – T)	E → E – T
E + (E)	(E)	T → E
E + T	E + T	E → E + T
E	E	E' → E
E'		

H.W. : For he following grammar

E ⟶ E+T | T

T ⟶ T*F | F

F ⟶ id | (E)

Parse the input **id * id + id**