Compiler Design

Dr. Sahar kamal

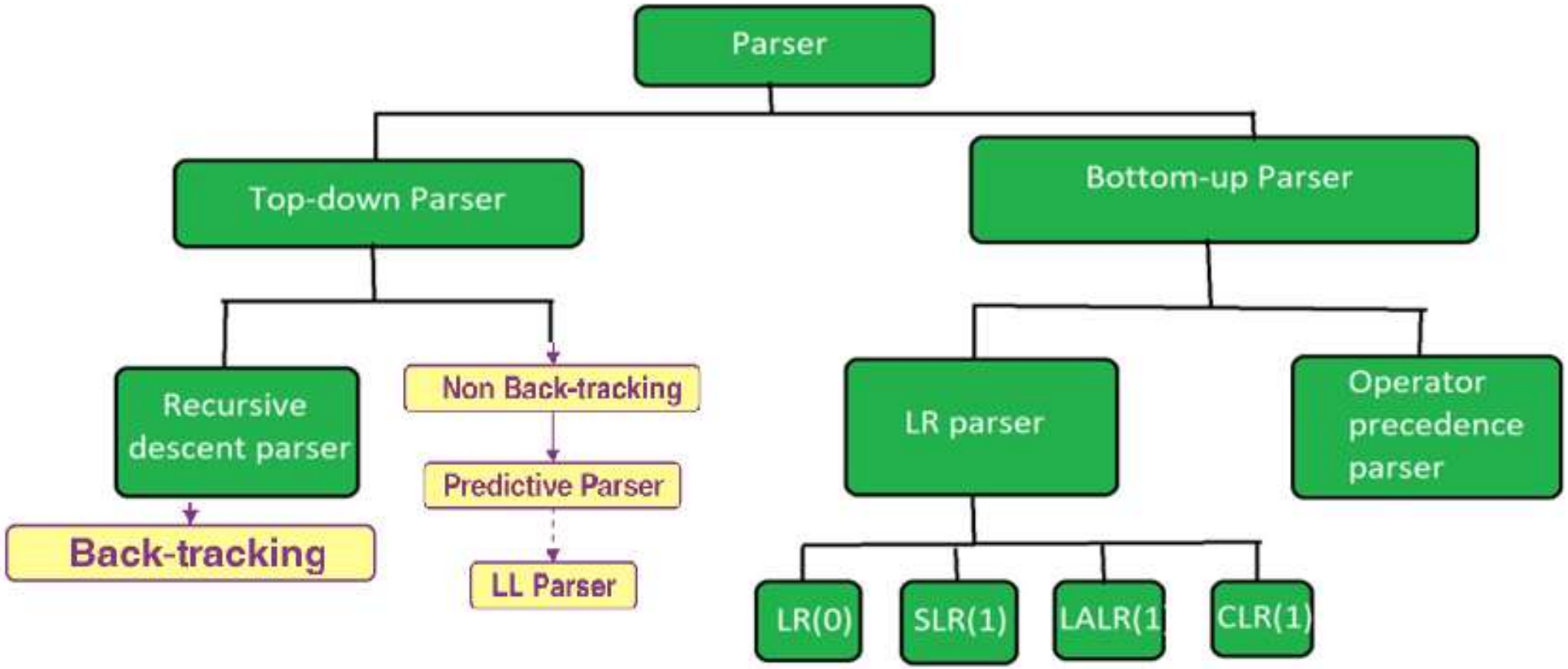
Syntax analyzer

Lecture 7

Compilers *Principles, Techniques, & Tools*

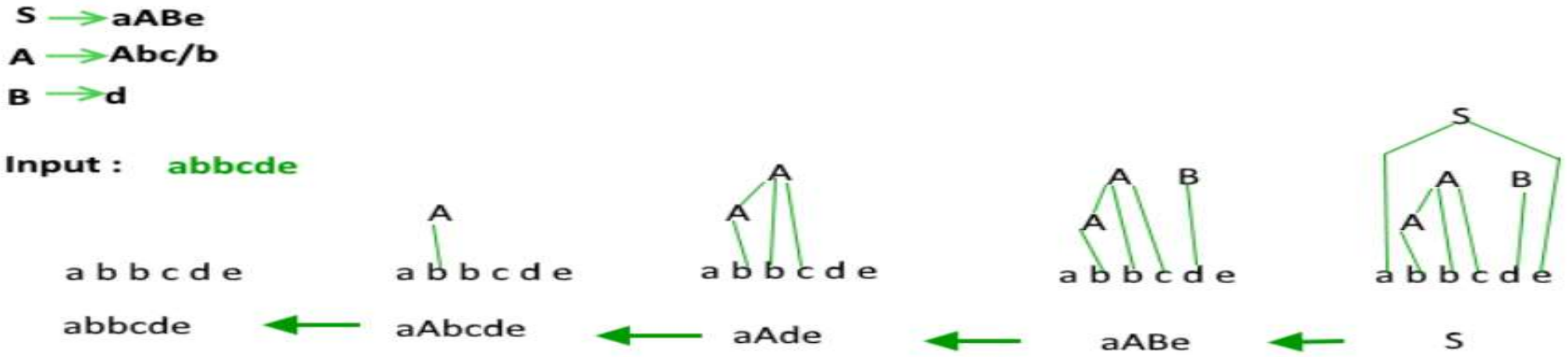
Second Edition

Types of Parsers



Bottom-Up Parsing in Compiler Design

- Bottom-Up parsing is applied in the syntax analysis phase of the compiler. Bottom-up parsing parses the stream of tokens from the lexical analyzer. And after parsing the input string it generates a parse tree.
- Bottom-up parsing is also known as **shift-reduce parsing.**
- Bottom-up parsing is used to construct a parse tree for an input string.
- In the bottom-up parsing, the parsing starts with the input symbol and construct the parse tree up to the start symbol by tracing out the rightmost derivations of string in reverse E.g.



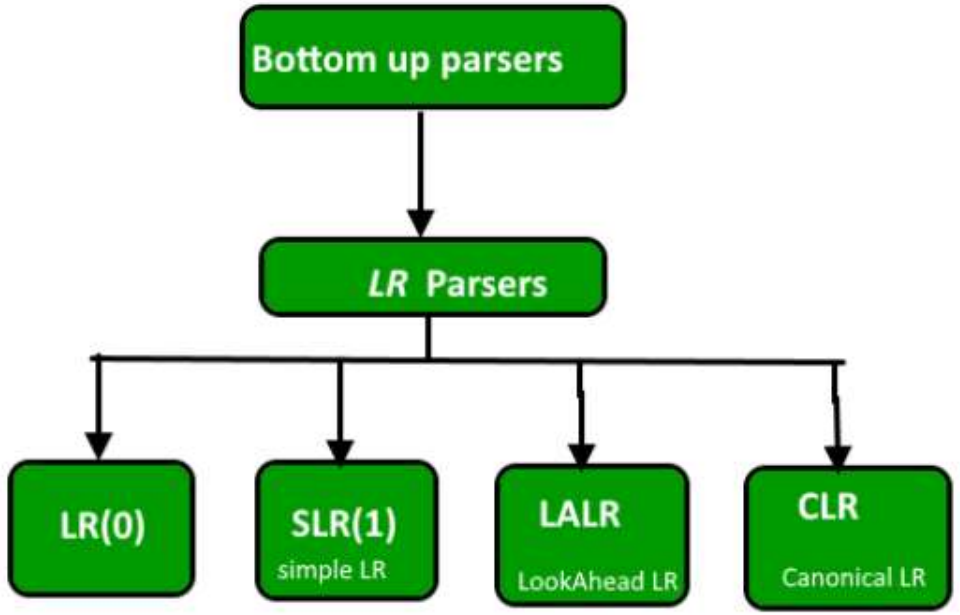
Bottom-Up Parsing in Compiler Design

- Bottom-up parsing is classified into various parsing. These are as follows:

Bottom-up parsing or Shift-Reduce Parsing

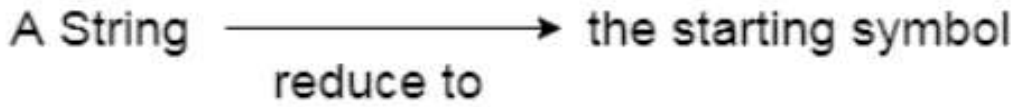
1.Operator Precedence Parsing

2.Table Driven LR Parsing



Shift reduce parsing

- Shift reduce parsing is a process of reducing a string to the start symbol of a grammar.
- Shift reduce parsing uses a stack to hold the grammar and an input tape to hold the string



- Shift reduce parsing performs the two actions: shift and reduce. That's why it is known as shift reduces parsing.
- At the shift action, the current symbol in the input string is pushed to a stack.
- At each reduction, the symbols will be replaced by the non-terminals. The symbol is the right side of the production and non-terminal is the left side of the production

Shift reduce parsing

- To perform the shift-reduce parsing we have to perform the following four functions.

Four Functions of Shift Reduce Parsing:

1. **Shift**: This action shifts the next input symbol present on the input buffer onto the top of the stack.
2. **Reduce:** This action is performed if the top of the stack has an input symbol that denotes a right end of a substring. And within the stack there exist the left end of the substring. The reduce action replaces the entire substring with the appropriate non-terminal. The production body of this non-terminal matches the replaced substring.
3. **Accept:** When at the end of parsing the input buffer becomes empty. And the stack has left with the start symbol of the grammar. The parser announces the successful completion of parsing.
4. **Error:** This action identifies the error and performs an error recovery routine.

Example:

Grammar:

1.S → S+S
2.S → S-S
3.S → (S)
4.S → a

Input string:

a1-(a2+a3)

Parsing table:

Stack contents	Input string	Actions
$	a1-(a2+a3)$	shift a1
$a1	-(a2+a3)$	reduce by S → a
$S	-(a2+a3)$	shift -
$S-	(a2+a3)$	shift (
$S-(	a2+a3)$	shift a2
$S-(a2	+a3)$	reduce by S → a
$S-(S	+a3) $	shift +
$S-(S+	a3) $	shift a3
$S-(S+a3	) $	reduce by S→a
$S-(S+S	) $	shift)
$S-(S+S)	$	reduce by S→S+S
$S-(S)	$	reduce by S → (S)
$S-S	$	reduce by S → S-S
$S	$	Accept

Let us take an example of the shift-reduce parser. Consider that we have a string id * id + id and the grammar for the input string is:

E ->E + T | T

T -> T * F | F

F -> (E) | id

Stack	Input Buffer	Action
$	id + id * id $	
$id	+ id * id $	Shift id
$F	+ id * id $	Reduce F-> id
$T	+ id * id $	Reduce T -> F
$E	+ id * id $	Reduce E -> T
$E +	id * id $	Shift +
$E + id	* id $	Shift id
$E + F	* id $	Reduce F-> id
$E + T	* id $	Reduce T -> F
$E + T *	id $	Shift *
$E + T * id	$	Shift id
$E + T * F	$	Reduce F-> id
$E + T	$	Reduce T -> F
$ E	$	Reduce E -> E + T

Shift-Reduce Parsing on Input String id + id * id

Operator precedence

- Operator precedence grammar is kinds of shift reduce parsing method. It is applied to a small class of operator grammars.

A grammar is said to be operator precedence grammar if it has two properties:

  - No R.H.S. of any production has a∈.
  - No two non-terminals are adjacent.

- Operator precedence can only be established between the terminals of the grammar. It ignores the non-terminal.
- There are the three operator precedence relations:
  - a > b means that terminal "a" has the higher precedence than terminal "b
  - a < b means that terminal "a" has the lower precedence than terminal "b".

Method − Let the input string be $a_1 a_2 \ldots \ldots . a_n$.$Initially, the stack contains $.

- Repeat forever
- If only $ is on the stack and only $ is on the input then accept and break else
- begin
  - let a be the topmost terminal symbol on the stack and let b be the current input symbols.
  - If a <. b or a =. b then shift b onto the stack /*Shift*/
  - else if a . >b then /*reduce*/
  - repeat pop the stack
  - until the top stack terminal is related by <. to the terminal most recently popped.
  - else call the error-correcting routine end

Operator Precedence Relations

	+	−	*	/	↑	id	(	)	$
+	.>	.>	<.	<.	<.	<.	<.	.>	.>
−	.>	.>	<.	<.	<.	<.	<.	.>	.>
*	.>	.>	.>	.>	<.	<.	<.	.>	.>
/	.>	.>	.>	.>	<.	<.	<.	.>	.>
↑	.>	.>	.>	.>	<.	<.	<.	.>	.>
id	.>	.>	.>	.>	.>			.>	.>
(	<.	<.	<.	<.	<.	<.	<.	=	
)	.>	.>	.>	.>	.>			.>	.>
$	<.	<.	<.	<.	<.	<.	<.		

Parsing Action

1. Both end of the given input string, add the $ symbol.
2. Now scan the input string from left right until the ⋗ is encountered.
3. Scan towards left over all the equal precedence until the first left most ⋖ is encountered.
4. Everything between left most ⋖ and right most ⋗ is a handle.
5. $ on $ means parsing is successful.

Example

step 1: Grammar:

1.E → E+T/T

2.T → T*F/F

3.F → id

Given string:

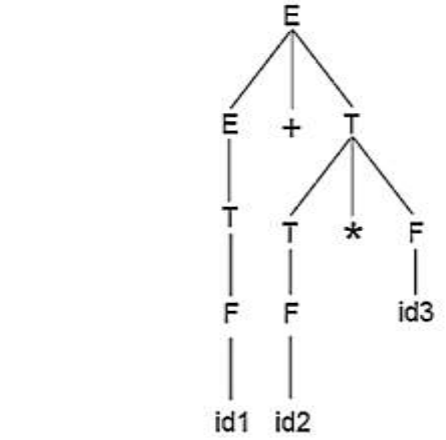
w = id + id * id

Step2:

Operator precedence relation table and Precedence Relations

	id	+	*	$
id		⋗	⋗	⋗
+	⋖	⋗	⋖	⋗
*	⋖	⋗	⋗	⋗
$	⋖	⋖	⋖	⋗

Step 4 :Let us consider a parse tree for it as follows



Step3 Now let us process the string with the help of the above precedence table:

$ ⋖ id1 ⋗ + id2 * id3 $

$ ⋖ F ⋗ + id2 * id3 $

$ ⋖ T ⋗ + id2 * id3 $

$ ⋖ E ≐ + ⋖ id2 ⋗ * id3 $

$ ⋖ E ≐ + ⋖ F ⋗ * id3 $

$ ⋖ E ≐ + ⋖ T ≐ * ⋖ id3 ⋗ $

$ ⋖ E ≐ + ⋖ T ≐ * ≐ F ⋗ $

$ ⋖ E ≐ + ≐ T ⋗ $

$ ⋖ E ≐ + ≐ T ⋗ $

$ ⋖ E ⋗ $

Accept.

Example –

Let's take an example to understand the role of operator precedence as follows.

```
E-> E+T/T
T-> T*V/V
V->a/b/c/d
string= "a+b*c*d"
```

Stack	Input	Stack Top	Current Input	Action
$	a+b*c*d$	$	a	shift a
$a	+b*c*d$	a	+	reduce using V->a
$V	+b*c*d$	V	+	reduce using T->V
$T	+b*c*d$	T	+	reduce using E->T
$E	+b*c*d$	E	+	shift +
$E+	b*c*d$	b	*	reduce using V->b
$E+V	b*c*d$	V	*	reduce using T->V
$E+T	*c*d$	T	*	shift *

$E+T*	c*d$	*	c	shift c
$E+T*c	*d$	c	*	reduce using V->c
$E+T*V	*d$	V	*	reduce using T->T*V
$E+T	*d$	T	*	shift *
$E+T*	d$	*	d	shift d
$E+T*d	$	d	$	reduce using V->d
$E+T*V	$	V	$	reduce using T->T*v
$E+T	$	T	$	reduce using E->E+T
$T	$	E	$	accept