PARSING

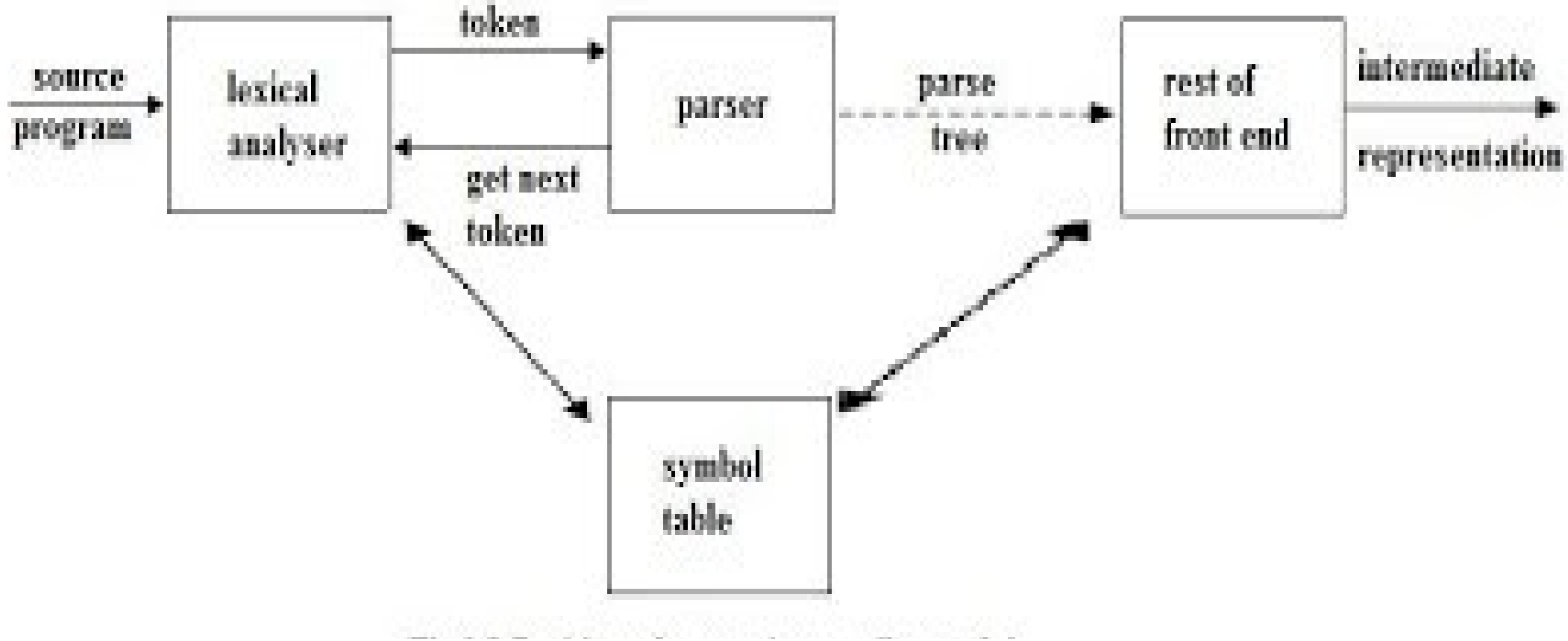
BY: KHUSHBOO JETHWA

ENROLLMENT NO.: 140950107028

DEFINITION OF PARSING

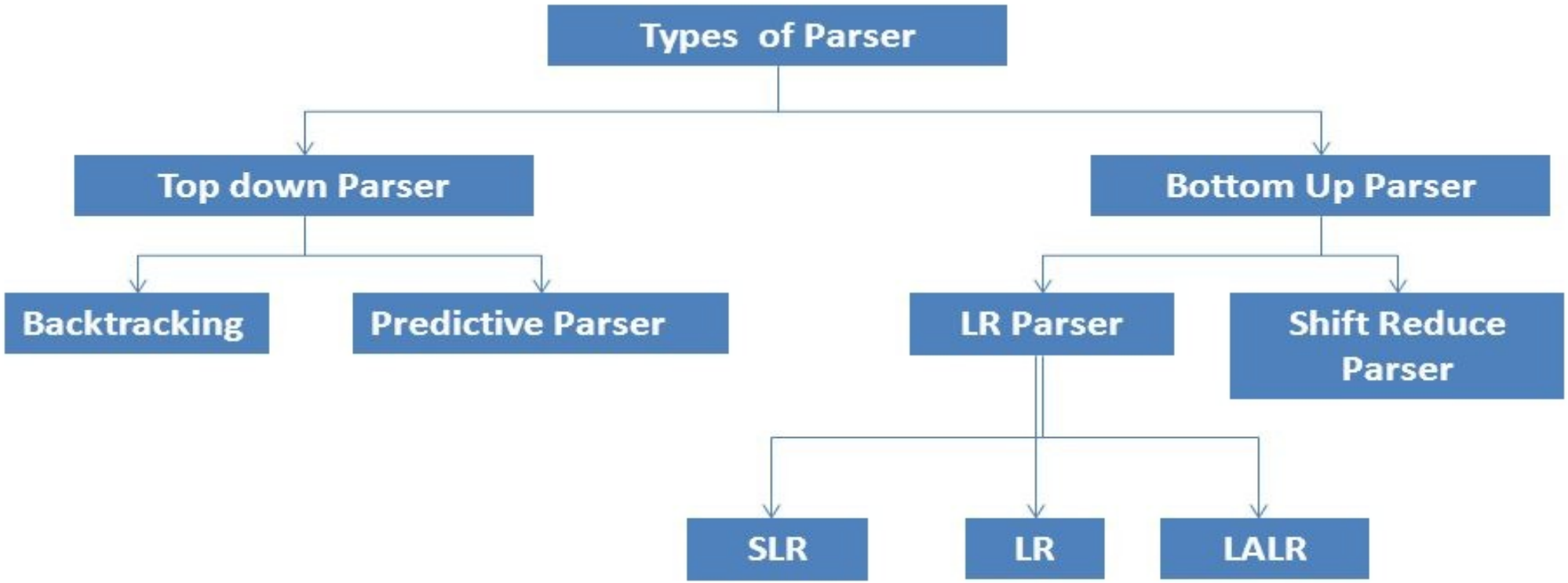
- A **parser** is a **compiler** or interpreter component that breaks data into smaller elements for easy translation into another language.
- A **parser**takes input in the form of a sequence of tokens or program instructions and usually builds a data structure in the form of a **parse** tree or an abstract syntax tree.

ROLE OF PARSER



- **In the compiler model, the parser obtains a string of tokens from the lexical analyser, and verifies that the string can be generated by the grammar for the source language.**
- **The parser returns any syntax error for the source language.**
- **It collects sufficient number of tokens anf builds a parse tree**

Types of Parser



- *There are basically two types of parser:*

- *Top-down parser:*
  - starts at the root of derivation tree and fills in
  - picks a production and tries to match the input
  - may require backtracking
  - some grammars are backtrack-free (*predictive*)

- *Bottom-up parser:*
  - starts at the leaves and fills in
  - starts in a state valid for legal first tokens
  - uses a *stack* to store both state and sentential forms

TOP DOWN PARSING

- A top-down parser starts with the root of the parse tree, labeled with the start or goal symbol of the grammar.

- To build a parse, it repeats the following steps until the fringe of the parse tree matches the input string
  - STEP1: At a node labeled *A*, select a production $A \rightarrow \alpha$ and construct the appropriate child for each symbol of α
  - STEP2: When a terminal is added to the fringe that doesn't match the input string, backtrack
  - STEP3: Find the next node to be expanded.

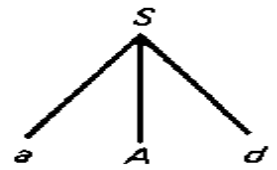
- The key is selecting the right production in step 1

EXAMPLE FOR TOP DOWN PARSING

- Supppose the given production rules are as follows:
- S-> aAd|aB
- A-> b|c
- B->ccd

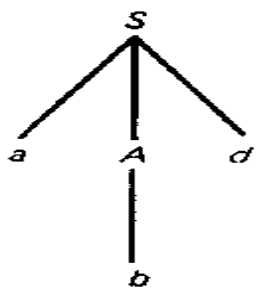
generated:
input: | *accd*

(*a*)



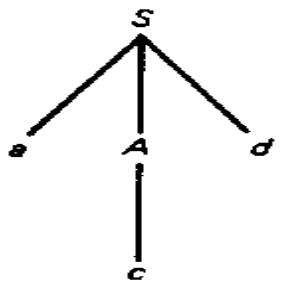
generated: *a*
input: *a* | *ccd*

(*b*)



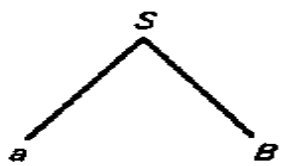
generated: *ab*
input: *ac* | *cd*

(*c*)



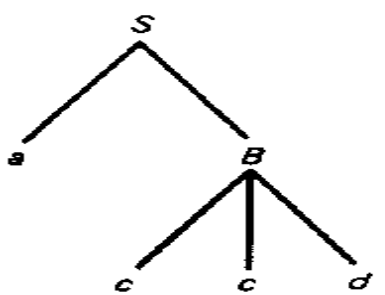
generated: *ac* | *acd*
input: *ac* | *cd* | *acc* | *d*

(*d*)



generated: *a*
input: *a* | *ccd*

(*e*)



generated: *ac* | *acc* | *accd*
input: *ac* | *cd* | *acc* | *d* | *accd*

(*f*)

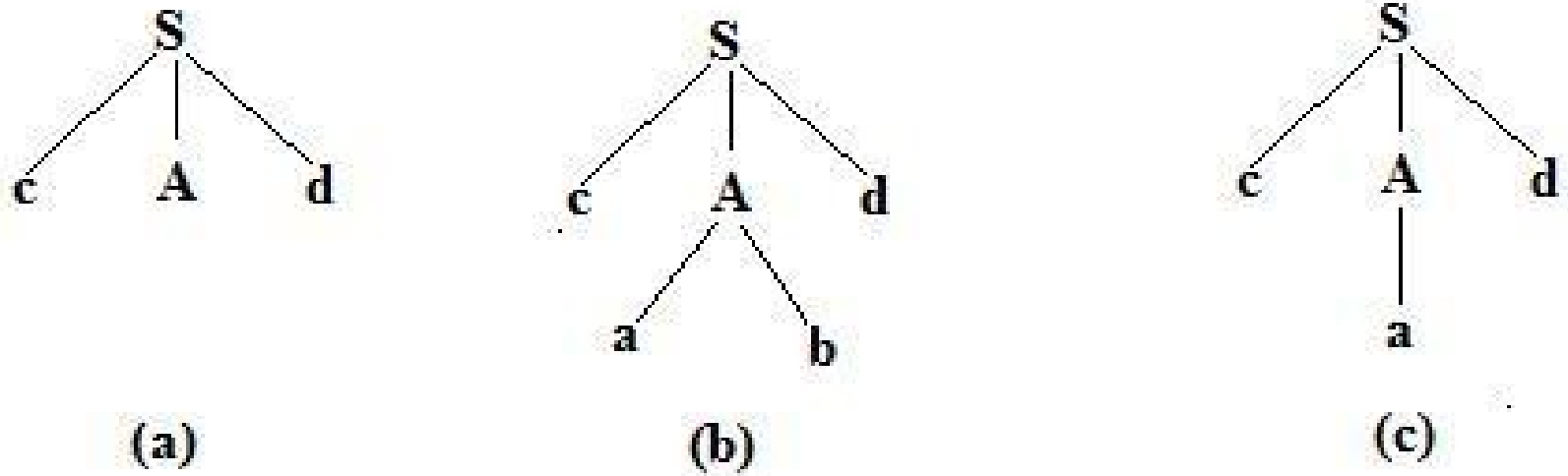
PROBLEMS WITH TOPDOWN PARSING

1) BACKTRACKING

- Backtracing is a technique in which for expansion of non-terminal symbol we choose one alternative and if some mismatch occurs then we try another alternative if any.
- If for a non-terminal there are multiple production rules beginning with the same input symbol then to get the correct derivation we need to try all these alternatives.

EXAMPLE OF BACKTRACKING

- Suppose the given production rules are as follows:
- S->cAd
- A->a|ab



- 2) LEFT RECURSION
- Left recursion is a case when the left-most non-terminal in a production of a non-terminal is the non-terminal itself(direct left recursion) or through some other non-terminal definitions, rewrites to the non-terminal again(indirect left recursion). Consider these examples -
- (1) A -> Aq (direct)
- (2) A -> Bq

  B -> Ar (indirect)
- Left recursion has to be removed if the parser performs top-down parsing

REMOVING LEFT RECURSION

- To eliminate left recursion we need to modify the grammar. Let, G be a grammar having a production rule with left recursion
- A-> Aa
- A->B
- Thus, we eliminate left recursion by rewriting the production rule as:
- A->BA’
- A’->aA’
- A’->c

3) LEFT FACTORING

- Left factoring is removing the common left factor that appears in two productions of the same non-terminal. It is done to avoid back-tracing by the parser. Suppose the parser has a look-ahead ,consider this example-

- A -> qB | qC

where A,B,C are non-terminals and q is a sentence. In this case, the parser will be confused as to which of the two productions to choose and it might have to back-trace. After left factoring, the grammar is converted to-

- A -> qD
- D -> B | C

RECURSIVE DESCENT PARSING

- A **recursive descent parser** is a kind of top-down **parser** built from a set of mutually **recursive** procedures (or a non-**recursive** equivalent) where each such procedure usually implements one of the productions of the grammar.

EXAMPLE OF RECURSIVE DESCENT PARSING

- Suppose the grammar given is as follows:
- E->iE’
- E’->+iE’

Program:

```
E()
{
        if(l==‘i’)
        {
                match(‘i’);
                E’();
        }
} l=getchar();
```

```
E’()
{
                    if(l==‘+”)
                    {
                                match(‘+’);
                                match(‘i’);
                                E’();
                    }
                    else
                                return ;
}
Match(char t)
{
                    if(l==t)
                                l=getchar();
                    else
                                printf(“Error”);
}
```

```
Main()
 {
  E();
  If(l==‘$’)
  {
                printf(“parsing successful”);
  }
  }
```

PREDICTIVE LL(1) PARSING

- The first “L” in LL(1) refers to the fact that the input is processed from left to right.
- The second “L” refers to the fact that LL(1) parsing determines a leftmost derivation for the input string.
- The “1” in parentheses implies that LL(1) parsing uses only one symbol of input to predict the next grammar rule that should be used.
- The data structures used by LL(1) are 1. Input buffer 2. Stack
  3. Parsing table

- The construction of predictive LL(1) parser is bsed on two very important functions and those are First and Follow.
- For construction of predictive LL(1) parser we have to follow the following steps:
  - STEP1: computate FIRST and FOLLOW function.
  - STEP2: construct predictive parsing table using first and follow function.
  - STEP3: parse the input string with the help of predictive parsing table

FIRST

- If X is a terminal **then** First(X) is just X!
- If there is a Production X → ε **then** add ε to first(X)
- If there is a Production X → Y1Y2..Yk **then** add first(Y1Y2..Yk) to first(X)
- First(Y1Y2..Yk) is **either**
  - First(Y1) (if First(Y1) doesn't contain ε)
  - **OR** (if First(Y1) does contain ε) then First (Y1Y2..Yk) is everything in First(Y1) <except for ε > as well as everything in First(Y2..Yk)
  - If First(Y1) First(Y2)..First(Yk) all contain ε **then** add ε to First(Y1Y2..Yk) as well.

FOLLOW

- First put $ (the end of input marker) in Follow(S) (S is the start symbol)
- If there is a production A → aBb, (where a can be a whole string) **then** everything in FIRST(b) except for ε is placed in FOLLOW(B).
- If there is a production A → aB, **then** everything in FOLLOW(A) is in FOLLOW(B)
- If there is a production A → aBb, where FIRST(b) contains ε, **then** everything in FOLLOW(A) is in FOLLOW(B)

EXAMPLE OF FIRST AND FOLLOW

The Grammar

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

- FIRST(E) = {'(',id}
- FIRST(E') = {+,ε}
- FIRST(T) = {'(',id}
- FIRST(T') = {*,ε}
- FIRST(F) = {'(',id}

- FOLLOW(E) = {$,)}
- FOLLOW(E') = {$,)}
- FOLLOW(T) = {+,$,)}
- FOLLOW(T') = {+,$,)}
- FOLLOW(F) = {*,+,$,)}

PROPERTIES OF LL(1) GRAMMARS

1. No left-recursive grammar is LL(1)
2. No ambiguous grammar is LL(1)
3. Some languages have no LL(1) grammar
4. A ε–free grammar where each alternative expansion for A begins with a distinct terminal is a *simple* LL(1) grammar.

Example:

$S \rightarrow aS \mid a$

is not LL(1) because FIRST(aS) = FIRST(a) = { a }

$S \rightarrow aS'$

$S' \rightarrow aS \mid \varepsilon$

accepts the same language and is LL(1)

PREDICTIVE PARSING TABLE

Method:

1. $\forall$ production $A \rightarrow \alpha$:
   - α) $\forall a \in$ FIRST(α), add $A \rightarrow \alpha$ to M[A,a]
   - b) If $\varepsilon \in$ FIRST(α):
     - I. $\forall b \in$ FOLLOW(A), add $A \rightarrow \alpha$ to M[A,b]
     - II. If \$ $\in$ FOLLOW(A), add $A \rightarrow \alpha$ to M[A,\$]
2. Set each undefined entry of M to `error`

If $\exists$M[A,a] with multiple entries then G is not LL(1).

EXAMPLE OF PREDICTIVE PARSING LL(1) TABLE

The given grammar is as follows

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

	FIRST	FOLLOW
S	{num, id}	{$}
E	{num, id}	{$}
E'	{ε, +, −}	{$}
T	{num, id}	{+, −, $}
T'	{ε, *, /}	{+, −, $}
F	{num, id}	{+, −, *, /, $}
id	{id}	−
num	{num}	−
*	{*}	−
/	{/}	−
+	{+}	−
−	{−}	−

	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	$F \to$ id	$F \to$ num	−	−	−	−	−

BOTTOM UP PARSING

- Bottom-up parsing starts from the leaf nodes of a tree and works in upward direction till it reaches the root node.
- we start from a sentence and then apply production rules in reverse manner in order to reach the start symbol.
- Here, parser tries to identify R.H.S of production rule and replace it by corresponding L.H.S. This activity is known as reduction.
- Also known as LR parser, where L means tokens are read from left to right and R means that it constructs rightmost derivative.

EXAMPLE OF BOTTOM-UP PARSER

- E → T + E | T
- T → int * T | int | (E)
- Consider the string: int * int + int

int * int + int	T → int
int * T + int	T → int * T
T + int	T → int
T + T	E → T
T + T	E → T
E	

SHIFT REDUCE PARSING

- Bottom-up parsing uses two kinds of actions: 1.Shift 2.Reduce
- Shift: Move | one place to the right , Shifts a terminal to the left string
  ABC|xyz ⇒ ABCx|yz
- Reduce: Apply an inverse production at the right end of the left string
  If A → xy is a production, then Cbxy|ijk ⇒ CbA|ijk

EXAMPLE OF SHIT REDUCE PARSING

- |int * int + int shift
- int | * int + int shift
- int * | int + int shift
- int * int | + int reduce T → int
- int * T | + int reduce T → int * T
- T | + int shift
- T + | int shift
- T + int | reduce T → int
- T + T | reduce E → T
- T + E | reduce E → T + E
- E |

OPERATOR PRECEDENCE PARSING

- *Operator grammars* have the property that no production right side is empty or has two adjacent nonterminals.
- This property enables the implementation of efficient *operator-precedence parsers*.
- These parser rely on the following three precedence relations:

Relation	Meaning
a <· b	a yields precedence to b
a =· b	a has the same precedence as b
a ·> b	a takes precedence over b

- These operator precedence relations allow to delimit the handles in the right sentential forms: <· marks the left end, =· appears in
- the interior of the handle, and ·> marks the right end.
- . Suppose that $ is the end of the string, Then for all terminals we can write: $ <· *b* and *b* ·> $
- If we remove all nonterminals and place the correct precedence relation:<·, =·, ·> between the remaining terminals, there remain strings that can be analyzed by easily developed parser.

EXAMPLE OF OPERATOR PRECEDENCE PARSING

For example, the following operator precedence relations can be introduced for simple expressions:

	id	+	*	$
id		·>	·>	·>
+	<·	·>	<·	·>
*	<·	·>	·>	·>
$	<·	<·	<·	·>

Example: The input string:
id_1 + id_2 * id_3
after inserting precedence relations becomes
$ <· **id_1** ·> + <· **id_2** ·> * <· **id_3** ·> $

THANK YOU