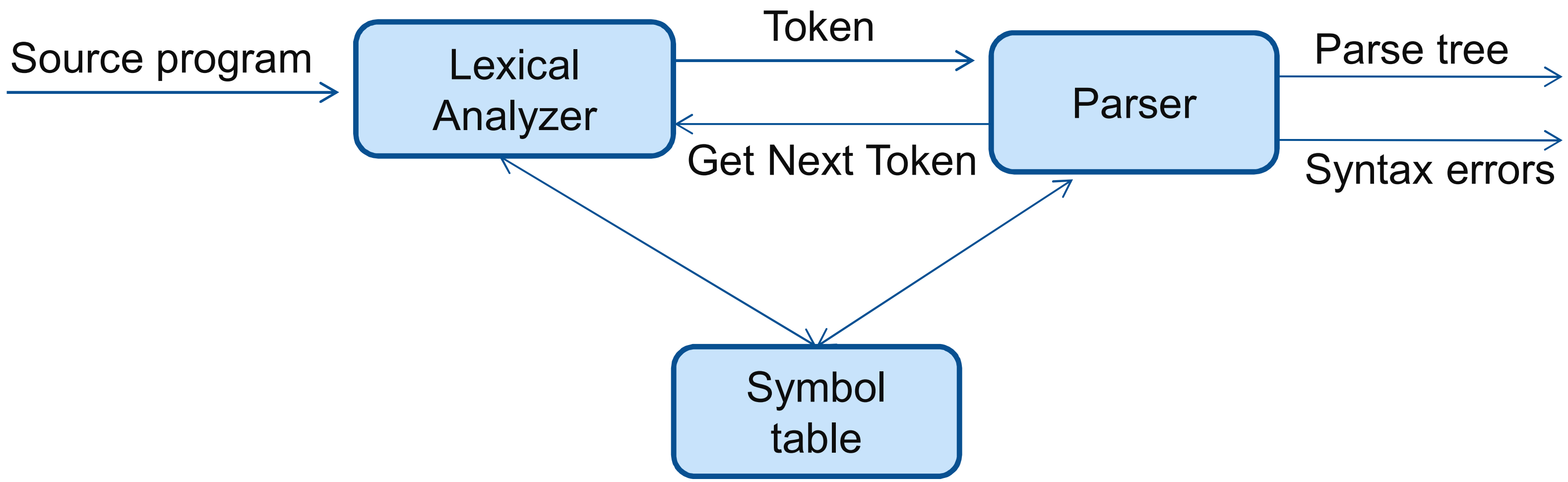
Syntax Analysis
(Part-1)

Role of Parser

- The parser obtains a string of tokens from the lexical analyzer and verifies that the string can be generated by the grammar for the source language to form a ***parse tree***.
- The parser provides report for any ***syntax errors*** in an intelligible fashion. It should also recover from commonly occurring errors so that it can continue processing the remainder of its input.



Error Handling

- The programmers frequently write incorrect programs, and a good compiler should assist the programmer in identifying and locating errors.
- The programs can contain errors at many different levels. For example, errors can be:
  - **lexical**, such as misspelling an identifier, keyword, or operator
  - **syntactic**, such as an arithmetic expression with unbalanced parentheses
  - **semantic**, such as an operator applied to an incompatible operand
  - **logical**, such as an infinitely recursive call
- Often ***much of the error detection*** and recovery in a compiler is centered around the syntax analysis phase.
- One reason for this is that ***many errors are syntactic in nature*** or are exposed when the stream of tokens coming from the lexical analyzer disobeys the grammatical rules.

Context-free Grammars

A context free grammar (grammar for short) consists of terminals, non-terminals, a start symbol, and productions.

- Terminals are the basic symbols from which strings are formed. The word "**token**" is a synonym for "**terminal**" when we are talking about grammars for programming languages. Each of the keywords **if**, **then** and **else** is a terminal.
- **Non-terminals** are syntactic variables that denote sets of strings. The non-terminals define sets of strings that help to define the language generated by the grammar.
- In a grammar, one non-terminal is distinguished as the **start symbol**. And the set of strings it denotes is the language defined by the grammar.
- 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, followed by an arrow (sometimes the symbol := is used), followed by a string of non-terminals and terminals.

Grammar for Arithmetic Expression

expr -> expr op expr

expr -> (expr)

expr -> - expr

expr-> id

op -> +

op -> -

op -> *

op -> /

In this grammar, the terminal symbols are **id + - / ( )**

and the non-terminal symbols are **expr** and **op**.

expr is the start symbol.

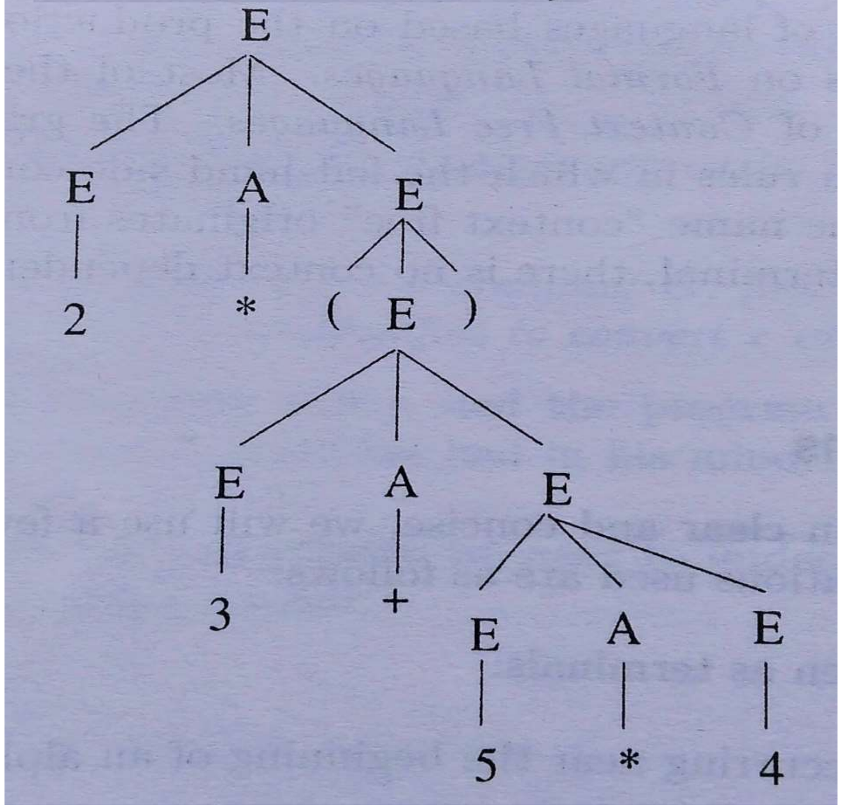
Derivation Tree/ Parse Tree

- The sequence of intermediary strings generated to expand the start symbol of the grammar to a desired string of terminals is called a **derivation**.
- The derivation can be represented by a tree is called **parse tree**.

E -> E A E | (E) | - E | id
A -> + | - | * | /

Derivation for "2*(3+5*4)"

E -> EAE -> 2AE -> 2*E
-> 2*(E) -> 2*(EAE)
-> 2*(3AE) -> 2*(3+E)
-> 2*(3+EAE)-> 2*(3+5AE)
-> 2*(3+5*E) -> 2*(3+5*4)



Parse tree for "2*(3+5*4)"

Leftmost and Rightmost Derivations

- The derivation in which the leftmost nonterminal is always replaced at each step is called ***leftmost derivation***.
- The derivation in which the rightmost nonterminal is always replaced at each step is called ***rightmost derivation***.
- In leftmost derivation, the intermediate strings are called ***left sentential forms***.
- In rightmost derivation, the intermediate strings are called ***right sentential forms***.
- The rightmost derivation is also called ***canonical representation***.

For example

expr -> expr op expr

expr -> (expr)

Leftmost derivation

expr -> expr op expr

-> (expr) op expr

Rightmost derivation

expr -> expr op expr

-> expr op (expr)

Ambiguous Grammar

- A grammar is said to be ambiguous if there exists more than one parse tree for the same sentence.
- An ambiguous grammar can have more than one leftmost and rightmost derivations.

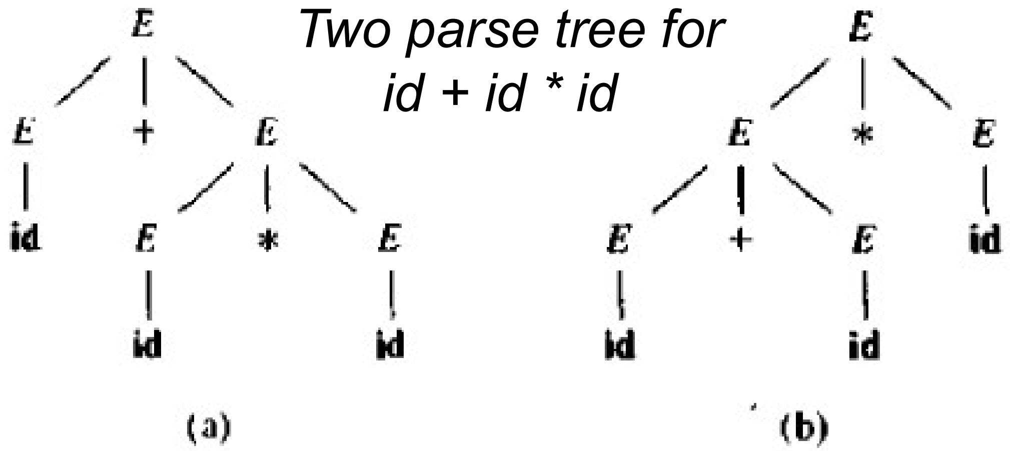
E -> E + E | E - E | E * E | E / E | (E) | - E | id

The sentence **id + id * id** has the two distinct leftmost derivations.

$$
\begin{aligned}
E &\Rightarrow E+E \\
&\Rightarrow \mathbf{id}+E \\
&\Rightarrow \mathbf{id}+E*E \\
&\Rightarrow \mathbf{id}+\mathbf{id}*E \\
&\Rightarrow \mathbf{id}+\mathbf{id}*\mathbf{id}
\end{aligned}
\qquad
\begin{aligned}
E &\Rightarrow E*E \\
&\Rightarrow E+E*E \\
&\Rightarrow \mathbf{id}+E*E \\
&\Rightarrow \mathbf{id}+\mathbf{id}*E \\
&\Rightarrow \mathbf{id}+\mathbf{id}*\mathbf{id}
\end{aligned}
$$

Operator * having higher precedence than +.

Expression a+b*c should be considered as a+(b*c) rather than as (a+b)*c.



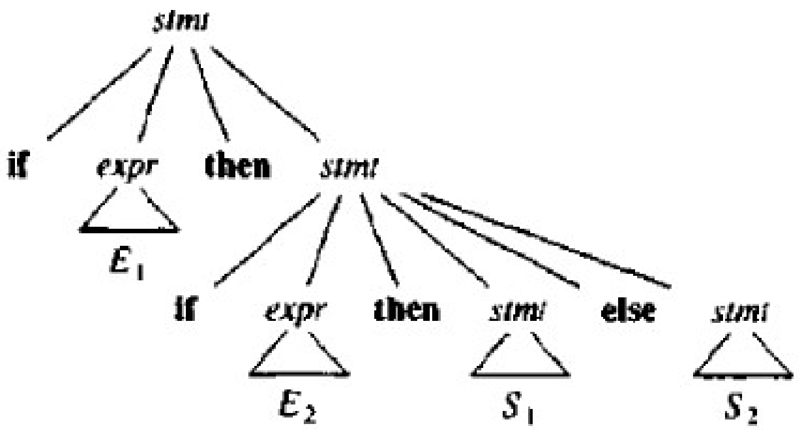
*Two parse tree for id + id * id*

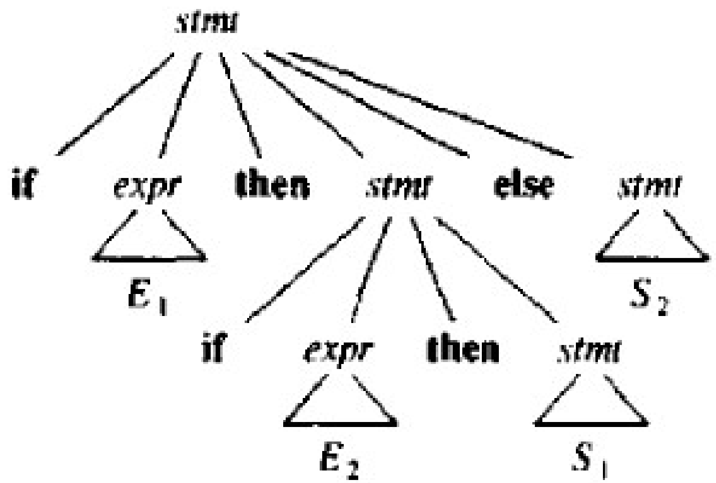
Ambiguous Grammar

stmt -> **if** expr **then** *stmt* | **if** expr **then** *stmt* **else** *stmt* | **other**

Here "other" stands for any other statement.

For example: **if** E_1 **then** **if** E_2 **then** S_1 **else** S_2






```
if  E1  then
      if  E2  then
          S1
      else
          S2
```

```
if  E1  then
     if  E2  then
         S1
else
     S2
```

Eliminating Ambiguity

- In all programming languages with conditional statements of this form, the first parse tree is preferred.
- The general rule is "Match each else with the closest previous unmatched then.
- So, we can rewrite the grammar to eliminate ambiguity.

stmt -> *matched_stmt* | *unmatched_stmt*

matched_stmt -> **if** *expr* **then** matched_*stmt*
else matched_*stmt* | **other**

unmatched_stmt -> **if** *expr* **then** *stmt* |
if expr **then** *matched_stmt* **else** *unmatched_stmt*

For example: **if** E_1 **then** **if** E_2 **then** S_1 **else** S_2

stmt -> *unmatched_stmt* -> **if** *expr* **then** *stmt* -> **if** E_1 **then** *stmt*

-> **if** E_1 **then** *matched_stmt*

-> **if** E_1 **then** **if** *expr* **then** matched_*stmt* **else** matched_*stmt*

-> **if** E_1 **then** **if** E_2 **then** S_1 **else** S_2

Eliminating Ambiguity

- To resolve the ambiguity we can add a matching ***endif*** with an if statement. So the grammar should be

  stmt -> **if** expr **then** *stmt* **endif** |
  if expr **then** *stmt* **else** *stmt* **endif** | **other**

For example: **if** E_1 **then** **if** E_2 **then** S_1 **else** S_2 **endif** **endif**

stmt-> **if** expr **then** *stmt* **endif** -> **if** E_1 **then** *stmt* **endif**

-> **if** E_1 **then if** expr **then** *stmt* **else** *stmt* **endif** **endif**

-> **if** E_1 **then if** E_2 **then** S_1 **else** S_2 **endif** **endif**

For example: **if** E_1 **then** **if** E_2 **then** S_1 **endif** **else** S_2 **endif**

stmt-> **if** expr **then** *stmt* **else** *stmt* **endif**

-> **if** E_1 **then** *stmt* **else** *stmt* **endif**

-> **if** E_1 **then if** expr **then** *stmt* **endif** **else** *stmt* **endif**

-> **if** E_1 **then if** E_2 **then** S_1 **endif** **else** S_2 **endif**

Left Recursion

- A grammar is left recursive if it has a nonterminal *A* such that there is a derivation *A -> Aα for some string α.*
- Top-down parsing methods cannot handle left-recursive grammars, *so* a transformation that eliminates left recursion is needed.

Elimination of Immediate Left Recursion

- The left-recursive pair of productions **A -> Aα | β** could be replaced by the non-left-recursive productions as follows:

$$A \rightarrow \beta A'$$

$$A' \rightarrow \alpha A' \mid \epsilon$$

Thus, the rule $A \rightarrow A\alpha_1 \mid A\alpha_2 \mid \ldots \mid A\alpha_m \mid \beta_1 \mid \beta_2 \mid \ldots \mid \beta_n$ can be modified as,

$$A \rightarrow \beta_1 A' \mid \beta_2 A' \mid \ldots \mid \beta_n A'$$

$$A' \rightarrow \alpha_1 A' \mid \alpha_2 A' \mid \ldots \mid \alpha_m A' \mid \epsilon$$

Elimination of Left Recursion

Eliminate Immediate Left Recursion from the following grammar:

E -> E + T | T

T -> T * F | F

F -> (E) | id

Solution:

E -> T E '

E ' -> + T E ' | ϵ

T -> F T '

T ' -> * F T ' | ϵ

F -> (E) | id

Elimination of Left Recursion

For example, consider the grammar, **S-> Aa, A->Sb | c**

Here, S is left recursive, because S-> Aa -> Sba. This form of general recursion can be eliminated with the following algorithm.

Algorithm for elimination of left recursion

1. Arrange non terminals in some order, say $A_1, A_2, \ldots, A_m$.

2. For i = 1 to m do

For j =1 to i-1 do

For each set of productions $A_i \rightarrow A_j\gamma$ and $A_j \rightarrow \delta_1 \mid \delta_2 \mid \ldots \mid \delta_k$

Replace $A_i \rightarrow A_j\gamma$ by $A_j \rightarrow \delta_1\gamma \mid \delta_2\gamma \mid \ldots \mid \delta_k\gamma$

3. Eliminate immediate left recursion from all productions

So, for the grammar **S-> Aa, A->Sb | c**

Step 1: Order of non-terminals are S, A.

Step 2: For i=1, S->Aa (there is no immediate left recursion)

For i=2, A->Sb | c is modified as, A->Aab|c

Step 3: Finally, S-> Aa, A->cA', A'->abA' | ϵ

Elimination of Left Recursion

Assignment No. 3: Eliminate Left Recursion of the following grammars.

a) A -> Ac | Aad | bd | ϵ

b) E -> E + E | E * E | (E) | id

Left Factoring

- ***Left factoring*** is a grammar transformation that is useful for producing a grammar suitable for ***predictive parsing***.
- The basic idea is that when it is not clear which of two alternative productions to use to expand a nonterminal *A*, we may be able to rewrite the A-productions to defer the decision until we have seen enough of the input to make the right choice.
- For example, if we have the two productions

  stmt -> **if** expr **then** *stmt* **else** *stmt* | **if** expr **then** *stmt*

- On seeing the input token **if**, we cannot immediately tell which production to choose to expand *stmt*. Only after **then**, if token **else** is found, we can decide the first rule to be used.
- This necessitates backtracking if token **else** is absent in the input stream, that is, it is an if-then statement. To **eliminate** this problem, the grammar is **left-factored** to take out the common portion separately as follows:

  stmt -> **if** expr **then** *stmt* *else-clause*

  else_clause **->** ***else*** *stmt* **| ϵ**

Algorithm for Left Factoring

- *Input:* Grammar G.
- Output: An equivalent left-factored grammar.
- Method: For each nonterminal *A* find the longest prefix α common to two or more of its alternatives. If $\alpha \neq \epsilon$, i.e., there is a nontrivial common prefix, replace all the A productions **$A \rightarrow \alpha\beta_1 \mid \alpha\beta_2 \mid \ldots \mid \alpha\beta_n \mid \gamma$** where γ represents all alternatives that do not begin with **α** by

  $A \rightarrow \alpha A' \mid \gamma$

  $A' \rightarrow \beta_1 \mid \beta_2 \mid \ldots \mid \beta_n$

Here A' is a new nonterminal. Repeatedly apply this transformation until no two alternatives for a nonterminal have a common prefix.

For example: ***S->iEtS | iEtSeS | a***

E->b

After Left Factoring:

S->iEtSS' | a

S'->eS | ϵ

E->b

Top-Down Parsing

- Top-down parsers get the name from the fact that they try to find a derivation of the input stream from the start symbol of the grammar.
- Equivalently, it can be viewed as an attempt to construct the parse tree rooted at the start symbol of the grammar for the input stream. There are two main approaches for top-down parsing.
  - **Recursive descent parsing**
  - **Predictive parsing**

Recursive Descent Parsing

- Top-down parsing can be viewed as an attempt to find a leftmost derivation for an input string.
- It can be viewed as an attempt to construct a parse tree for the input starting from the root and creating the nodes of the parse tree in preorder.
- A general form of top-down parsing, called recursive descent, that may involve backtracking, that is, making repeated scans of the input.
- However, backtracking parsers are not seen frequently due to inefficiency.

Recursive Descent Parsing

Example: Consider the grammar

S->abA, A->cd | c | ϵ

For the input stream **ab**:

- The parser starts by constructing a parse tree representing S->abA as shown in Fig. (a).
- The tree is expanded with the production A->cd as shown in Fig. (b).
- Since it does not match the string ab, the parser backtracks and then, tries the alternative A->c as shown in Fig. (c). However, the parse tree does not match the string ab.
- So, the parser backtracks and tries out the alternative A->ϵ as shown in Fig. (d). This time it finds a match. Thus, the parsing is complete and successful.

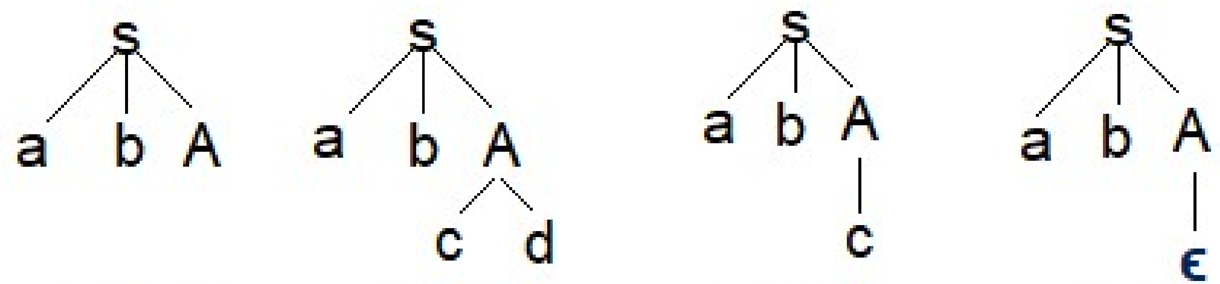


Fig. (a) Fig. (b) Fig. (c) Fig. (d)

Recursive Descent Parsing

- If the grammar is ***left-recursive***, a recursive descent parser ***may fall into an infinite loop*** even in the presence of backtracking.
- This happens because of the fact that for a left-recursive rule, the parser has to expand without consuming any further input symbol.
- A parser construction strategy, known as ***predictive parser*** is developed to create a recursive descent parser that ***does not need backtracking***.
- The ***predictive parser*** can be constructed in both ***recursive and non-recursive*** manner.

Recursive Predictive Parsing

- In many cases, by carefully writing a grammar, eliminating left recursion from it, and left factoring the resulting grammar, we can obtain a grammar that can be parsed by a recursive-descent parser that needs no backtracking, i.e., a predictive parser.
- To construct a predictive parser, we must know, given the current input symbol ***a*** and the nonterminal ***A*** to be expanded, which one of the alternatives of production **$A \rightarrow \alpha_1 \mid \alpha_2 \mid \ldots \mid \alpha_n$** is the unique alternative that derives a string beginning with ***a***. That is, the proper alternative must be detectable by looking at only the first symbol it derives.
- Flow-of-control constructs in most programming languages, with their distinguishing keywords are usually detectable in this way.
- For example, if we have the productions

  stmt -> **if** expr **then** *stmt* **else** *stmt* | **while** *expr* ***do*** *stmt*

  then the keywords **if** and **while** tell us which alternative is the only one that could possibly succeed if we are to find a statement.

Recursive Predictive Parsing

- **After left factoring**, the resultant rules can also be represented in the form of a set of transition diagrams. For this purpose we can **create a diagram for each nonterminal** *A*.
  1. Create an initial and final (return) state.
  2. For each production A -> X_1 X_2 … X_n, create a path from the initial state to the final state, with edge labeled X_1 ,X_2, … ,X_n.
- The predictive parser begins in the start state **s** for the start symbol. If after some actions it moves to a state **t** with an edge label of terminal **a**, and if the next input symbol is **a**, then the parser moves the input cursor one position right and goes to state **t**.
- If the edge is labeled by a nonterminal **A**, the parser instead goes to the start state for **A**, without moving the input cursor. If it ever reaches the final state for **A**, it immediately goes to state **t**, in effect having "read" **A** from the input during the time it moved from state **s** to **t**.
- Finally, if there is an edge from **s** to **t** labeled **ϵ**, then from state **s** the parser immediately goes to state **t**, without advancing the input.

Recursive Predictive Parsing

For example: exp -> exp + term | term

term -> term * factor | factor

factor -> (exp) | **id**

After removal of left-recursion we get

exp -> term exp_tail

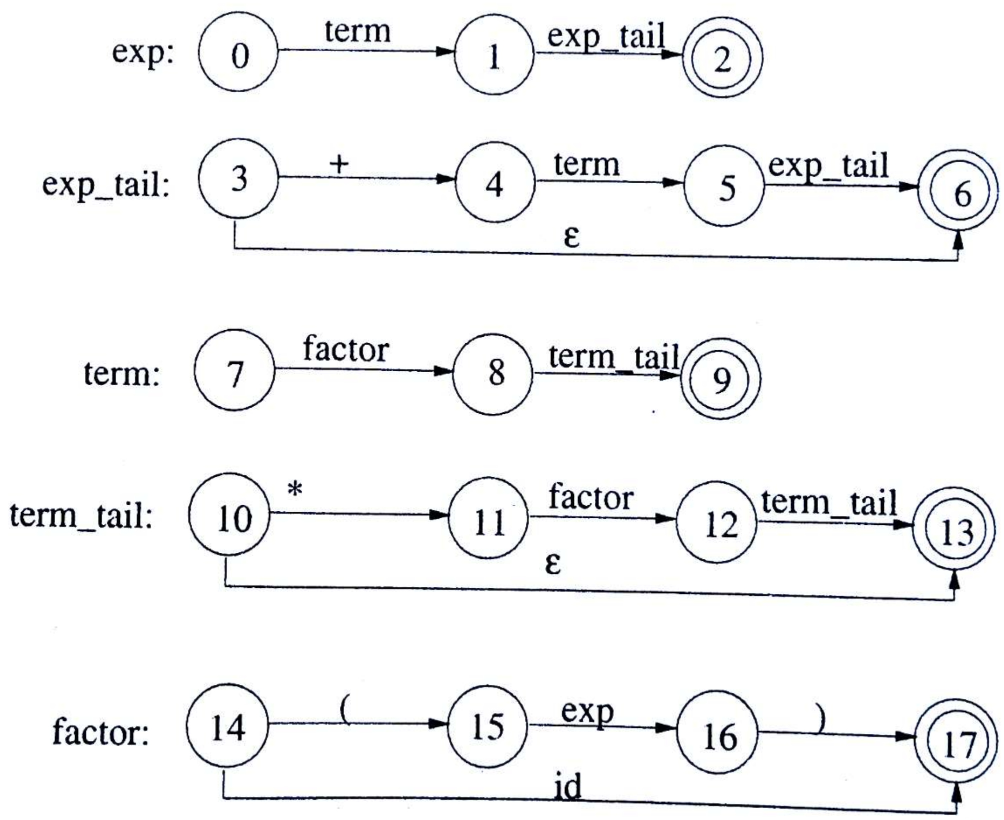
exp_tail -> + term expr_tail | ϵ

term -> factor term_tail

term_tail -> * factor term_tail | ϵ

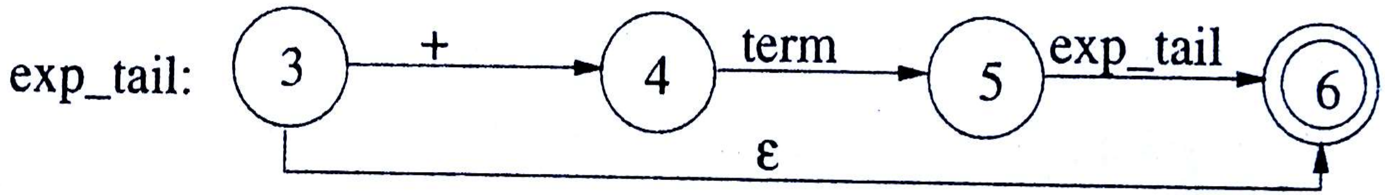
factor -> (exp) | **id**

Transition Diagrams

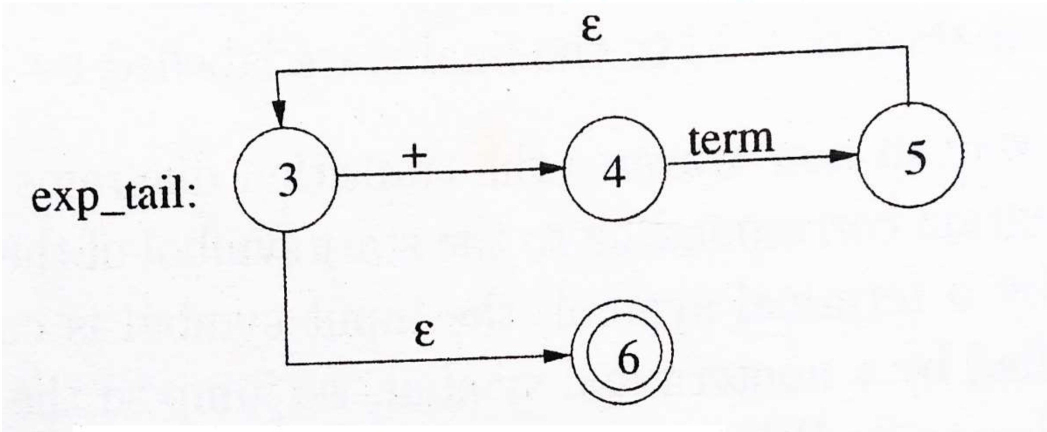


Recursive Predictive Parsing

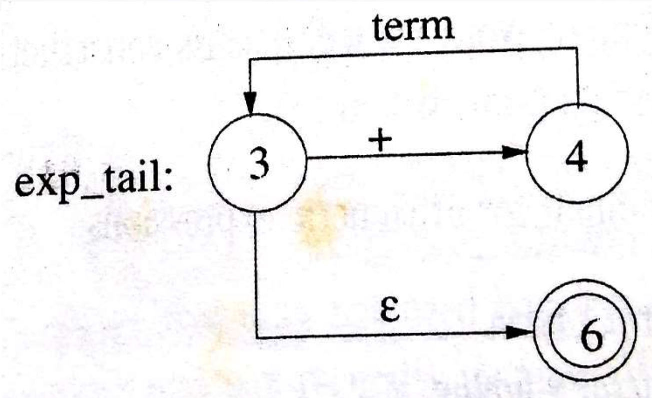
Simplification of diagrams may create a more compact and efficient parser.



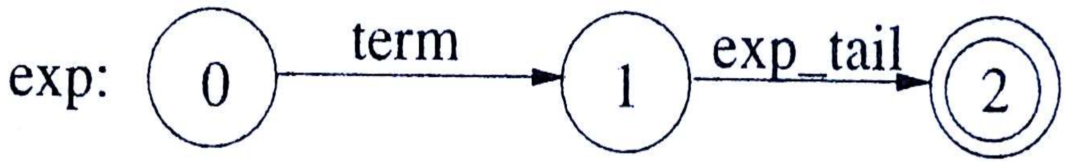
First, we eliminate self-recursion in the exp_tail transition diagram, substituting an iterative model.



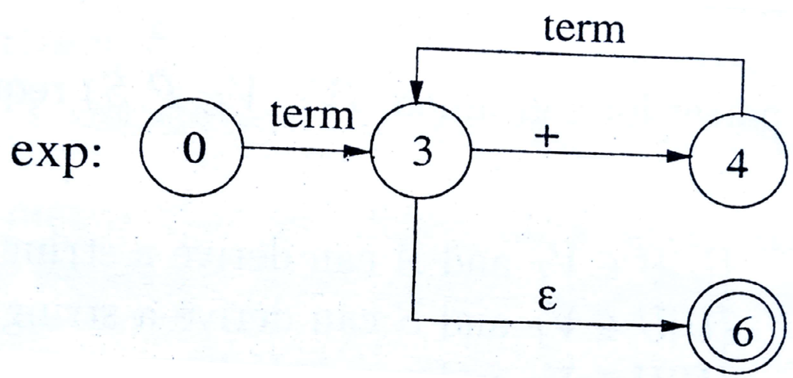
Further simplification by removing the redundant ϵ-edge.



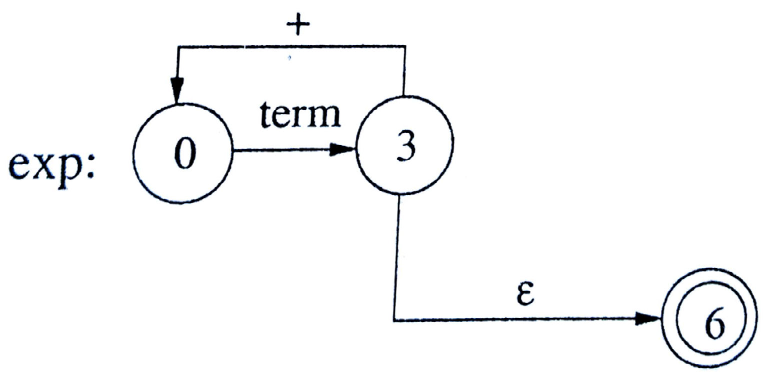
Recursive Predictive Parsing



If we substitute the exp_tail diagram in the exp one, replacing the exp_tail edge from 1 to 2, we get



By further simplification we get



Recursive Predictive Parsing

Applying the same approach to term and term_tail, we get a reduced set of diagrams for arithmetic expressions as shown below.

Final Set of Transition diagrams for expression grammar