Parser

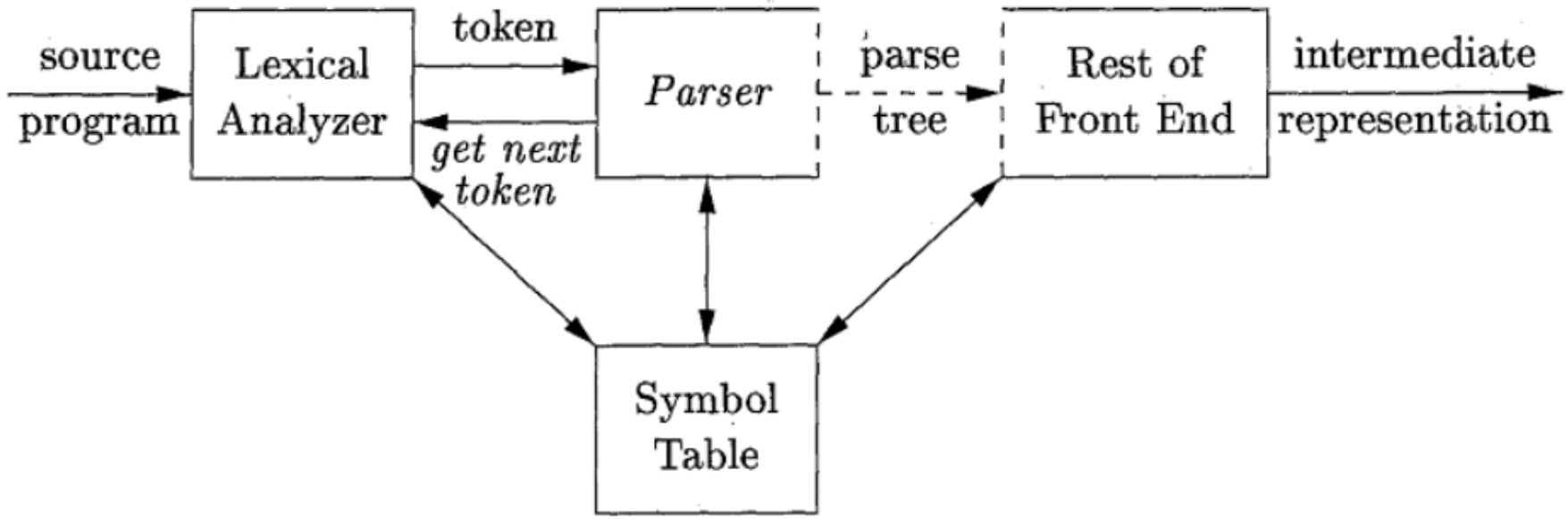
Dr. Meera Thapar Khanna

CSE Department

Pandit Deendayal Energy University

Parsing

- 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 Parse Tree representation.
- The parser is also known as Syntax Analyzer or Hierarchical Analysis.
- Parser report any syntax errors in the program and to recover from commonly occurring errors to continue processing the remainder of the program


source
program
Lexical
Analyzer
token
get next
token
Parser
parse
tree
Rest of
Front End
intermediate
representation
Symbol
Table

Types of Parsing

- Top Down Parsing
- Bottom Up Parsing

Top Down Parser

- Top-down parser find the left most derivation for an input string.
- It construct a parse tree for the input starting from the root and creating the nodes of the parse tree in preorder.

Types of Top-Down Parsing

- Recursive descent parsing
- Predictive parsing

RECURSIVE DESCENT PARSING

- It is a type of Top down parsing that uses a set of recursive procedures to scan the input,
- It may involve backtracking i.e. making repeated scanning of the input.

A top-down parser starts with the root of the parse tree, labelled with the start symbol of the grammar. To build a parse, it repeats the following steps until the input string matches the parse tree

1. At a node labelled A, select a production A → α and construct the appropriate child for each symbol of α

2. When a terminal is added that doesn't match the input string, backtrack

3. Find the next node to be expanded.

The key is selecting the right production in step 1 that should be guided by input string.

Example

Grammar

G :

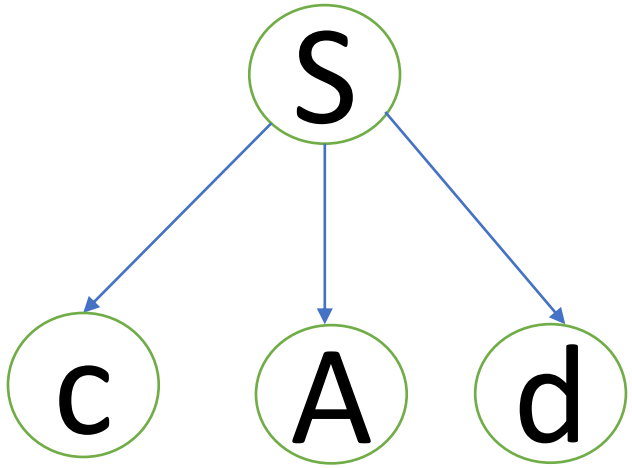
$S \rightarrow cAd$

$A \rightarrow ab \mid a$

and the input string w=cad.

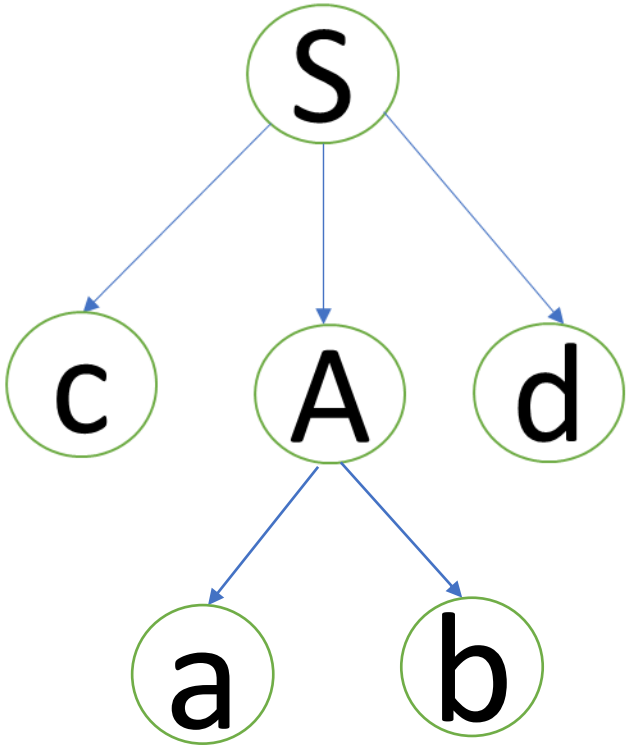
Step 1

- Initially, tree is created with single node labeled S. An input pointer points to ‘c’, the first symbol of string w.
- Now, expand the tree with the production of S.



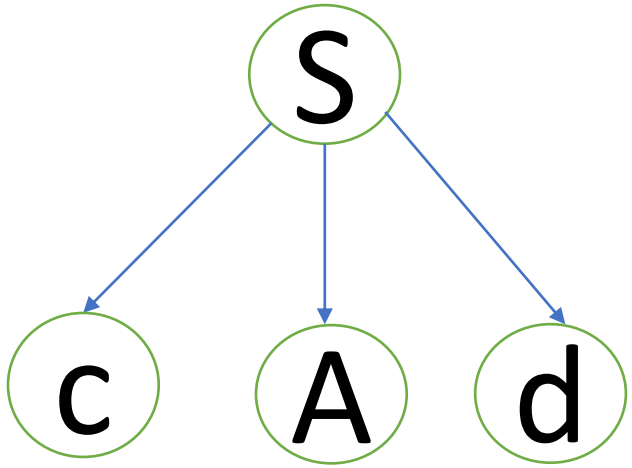
Step 2

- The leftmost leaf ‘c’ matches the first input symbol of string w, so advance the input pointer to the second input symbol of string w ‘a’ and consider the next leaf ‘A’. So go for the first alternative i.e. expand ‘A’.



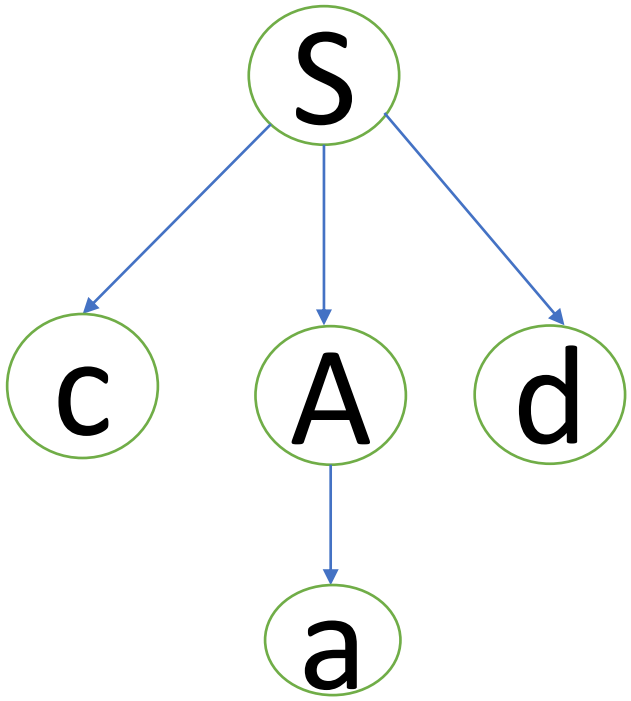
Step 3

- The second symbol ‘a’ of w also matches with second leaf of tree. So advance the input pointer to third symbol of string w ‘d’. But the third leaf of tree is b which does not match with the input symbol d.
- Hence discard this production and reset the pointer to second position. This is called backtracking.



Step 4

- Now try the second alternative for A.



Note

- A left-recursive grammar can cause a recursive-descent parser to go into an infinite loop.
- Hence, elimination of left-recursion must be done before parsing.

Example

Grammar

E -> E + T | T

T -> T * F | F

F -> -E | (E) | id

Remove Left Recursion

First eliminate the left recursion for E as

$E \rightarrow TE'$

$E' \rightarrow +TE' \mid \varepsilon$

Then eliminate for T as

$T \rightarrow FT'$

$T' \rightarrow *FT' \mid \varepsilon$

New Grammar

$E \rightarrow TE'$

$E' \rightarrow +TE' \mid \varepsilon$

$T \rightarrow FT'$

$T' \rightarrow *FT' \mid \varepsilon$

$F \rightarrow (E) \mid id$

Procedure for Recursive Descent Parsing

- Each non-terminal becomes a function that tells about the RHS of the productions associated with it and choses a particular RHS:
- This is decided on the basis of a look-ahead symbol
- and throws an exception if no alternative applies

Recursive Procedure for Example

Non terminals are **E, E', T, T', F** so name the functions like E(), E_dash(), T(), T_Dash(), F(), Next()

First on encountering symbol **E:**

```
E()
{
        T()
        E_Dash()
}
```

For symbol **E’**:

```
E_Dash()
{
    If input_symbol = ’+’ then
        Next( )
        T( )
        E_Dash( )
    else
        error()
}
```

For Symbol **T**:

```
T()
{
    F()
    T_Dash()
}
```

For Symbol **T'**:

```
T_Dash()
{
    If input_symbol = '*' then
        Next( )
        F( )
        T_Dash( )
    else
        error()
}
```

For Symbol **F**:

```
F()
{
        If input_symbol = ’id’ then
                Next( )
        Else if input_symbol = ‘(’ then
                        Next()
                        E( )
        Else if input_symbol = ‘)’ then
                        Next( )
                else
                        error()
}
```

```
Next()
{
        input = next token;
}
```

For input id+id*id, sequence of procedure call

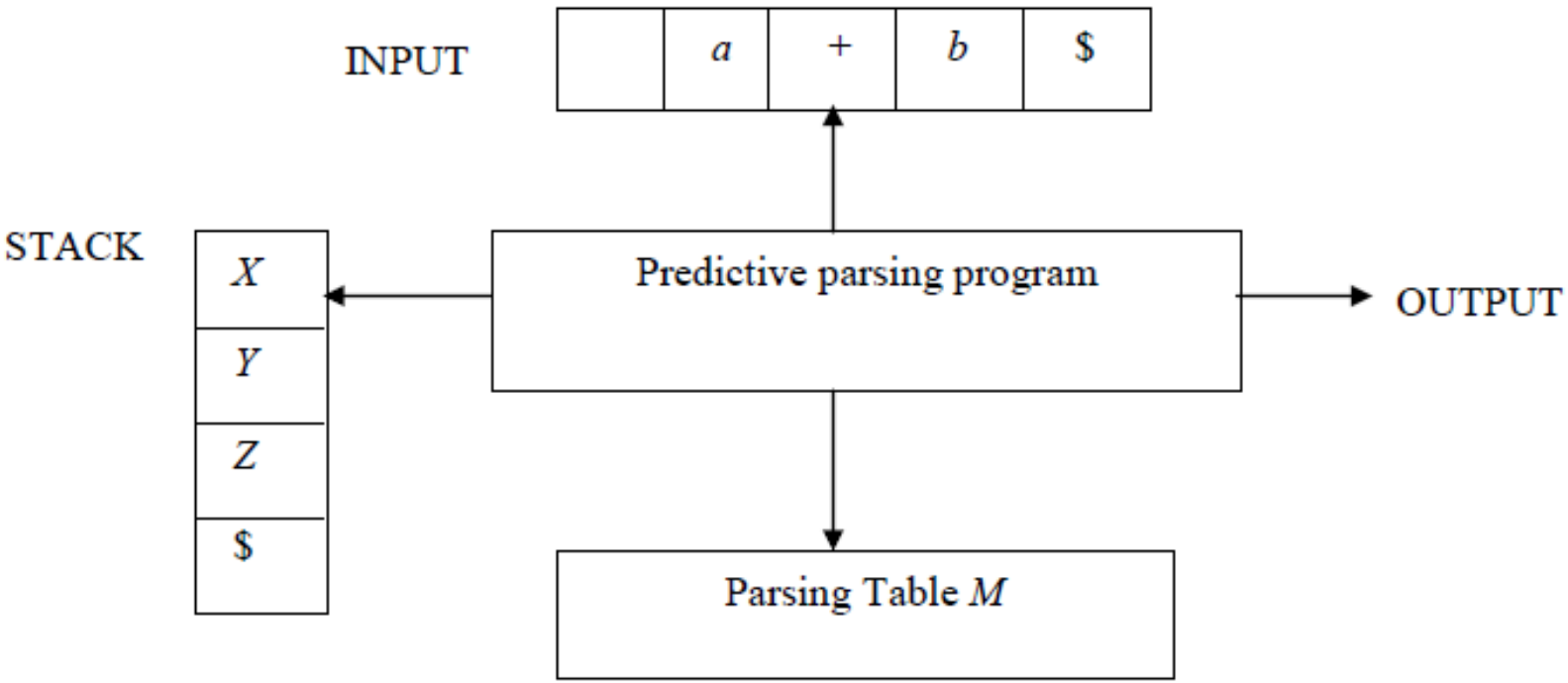
PROCEDURE	INPUT STRING
E( ) E->TE’	**id**+id*id
T( ) T->FT’	**id**+id*id
F( ) F->id	**id**+id*id
Next( )	id**+**id*id
T_Dash( ) T’-> ε	id**+**id*id
E_Dash( ) E’->+TE’	id**+**id*id
Next( )	id+**id***id
T( ) T->FT’	id+**id***id

PROCEDURE	INPUT STRING
F( ) F->id	id+**id***id
Next( )	id+id*****id
T_Dash( ) T’->*FT’	id+id*****id
Next( )	id+id*****id
F( ) F->id	id+id*****id
Next( )	id+id*__id__
T_Dash( ) T’-> ε	id+id*__id__
E_Dash() E’-> ε	id+id*__id__

PREDICTIVE PARSING

- Predictive parsing is a special case of recursive descent parsing where no backtracking is required.
- The key problem of predictive parsing is to determine the production to be applied for a non-terminal in case of alternatives.

Non-recursive predictive parser



Input buffer:

- It consists of strings to be parsed, followed by $ to indicate the end of the input string.

Stack:

- It contains a sequence of grammar symbols preceded by $ to indicate the bottom of the stack.
- Initially, the stack contains the start symbol on top of $.

Parsing table:

- It is a two-dimensional array M[A, a], where ‘A’ is a non-terminal and ‘a’ is a terminal.

Parsing Program

The parser program considers X, the symbol on top of stack, and ‘a’, the current input symbol. These two symbols decide the parser action.

There are three possibilities:

- If X = a = $, the parser halts and announces successful completion of parsing.
- If X = a ≠ $, the parser pops X off the stack and advances the input pointer to the next input symbol.
- If X is a non-terminal , the program consults entry M[X, a] of the parsing table M. This entry will either be an X-production of the grammar or an error entry.
  - If M[X, a] = {X → UVW}, the parser replaces X on top of the stack by UVW
  - If M[X, a] = error, the parser calls an error recovery routine.

Step for constructing Predictive Parser

- Write the Context Free grammar for given input String
- Check for Ambiguity. If ambiguous remove ambiguity from the grammar
- Check for Left Recursion. Remove left recursion if it exists.
- Check For Left Factoring. Perform left factoring if it contains common prefixes in more than one alternates.
- Compute FIRST and FOLLOW sets
- Construct Parser Table
- Using Parsing Algorithm generate Parse tree as the Output