Compiler course

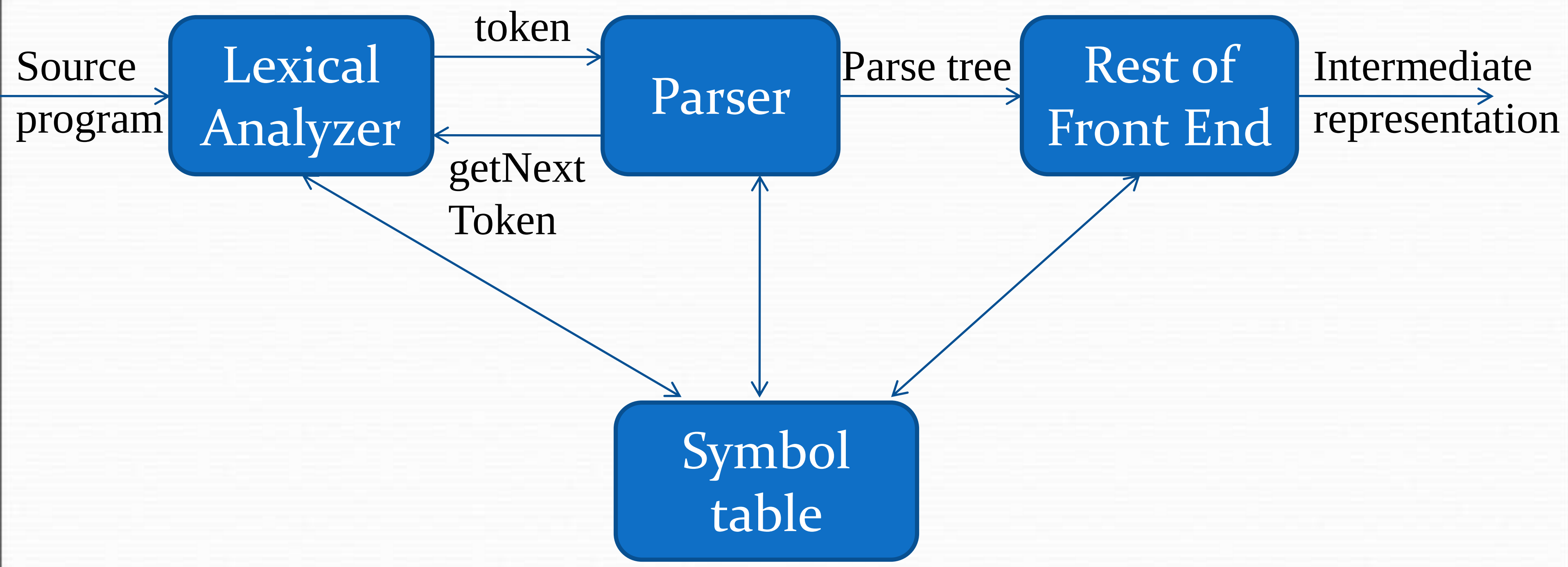
Chapter 4

Syntax Analysis

Outline

- Role of parser
- Context free grammars
- Top down parsing
- Bottom up parsing
- Parser generators

The role of parser



Uses of grammars

E -> E + T | T
T -> T * F | F
F -> (E) | **id**

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

Error handling

- Common programming errors
  - Lexical errors
  - Syntactic errors
  - Semantic errors
  - Lexical errors
- Error handler goals
  - Report the presence of errors clearly and accurately
  - Recover from each error quickly enough to detect subsequent errors
  - Add minimal overhead to the processing of correct progrms

Error-recover strategies

- Panic mode recovery
  - Discard input symbol one at a time until one of designated set of synchronization tokens is found
- Phrase level recovery
  - Replacing a prefix of remaining input by some string that allows the parser to continue
- Error productions
  - Augment the grammar with productions that generate the erroneous constructs
- Global correction
  - Choosing minimal sequence of changes to obtain a globally least-cost correction

Context free grammars

- Terminals
- Nonterminals
- Start symbol
- productions

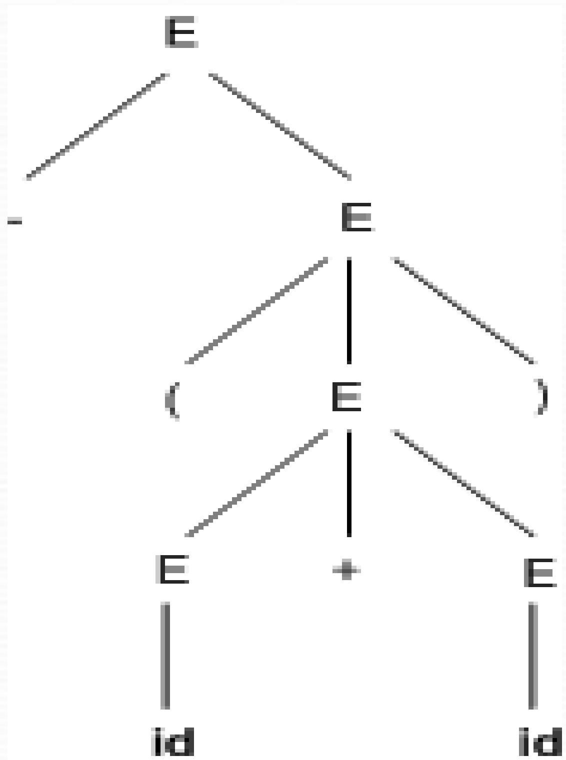
expression -> expression + term
expression -> expression – term
expression -> term
term -> term * factor
term -> term / factor
term -> factor
factor -> (expression)
factor -> **id**

Derivations

- Productions are treated as rewriting rules to generate a string
- Rightmost and leftmost derivations
  - E -> E + E | E * E | -E | (E) | **id**
  - Derivations for **–(id+id)**
    - E => -E => -(E) => -(E+E) => -(**id**+E)=>-(**id**+**id**)

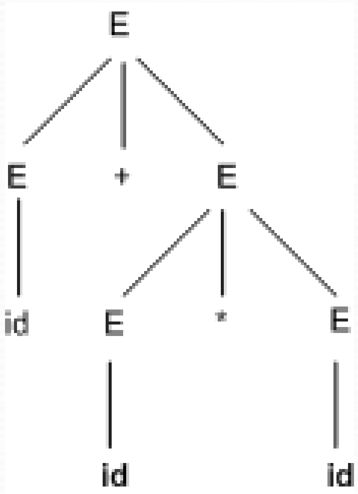
Parse trees

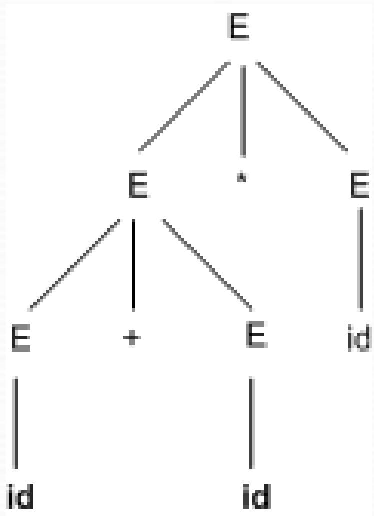
- **-(id+id)**
- E => -E => -(E) => -(E+E) => -(**id**+E)=>-(**id**+**id**)



Ambiguity

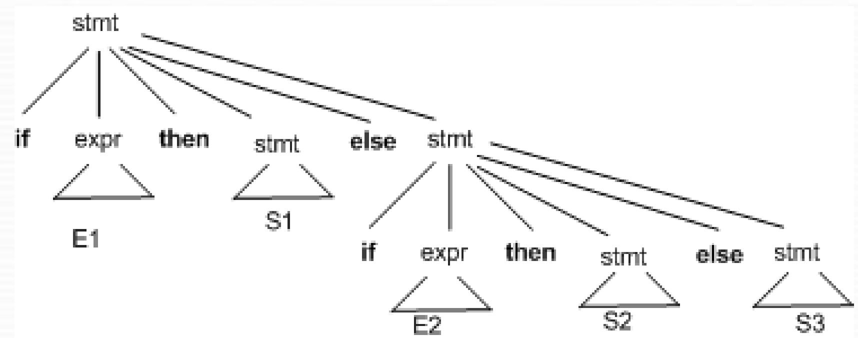
- For some strings there exist more than one parse tree
- Or more than one leftmost derivation
- Or more than one rightmost derivation
- Example: id+id*id

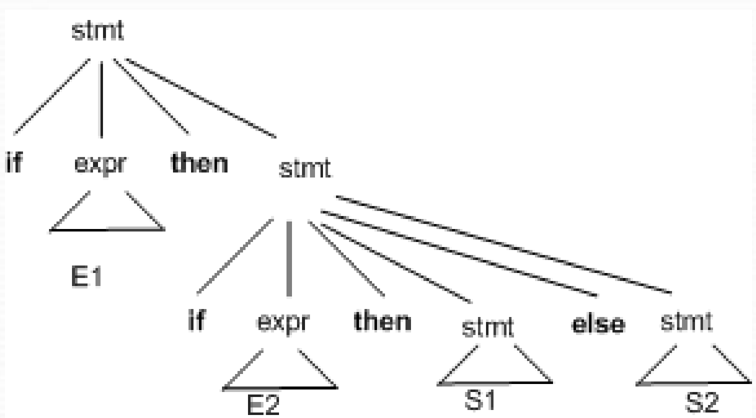


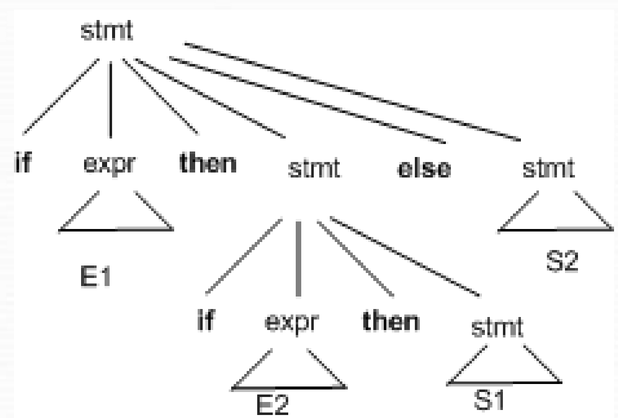


Elimination of ambiguity

stmt $\longrightarrow$ **If** expr **then** stmt
| **If** expr **then** stmt **else** stmt
| **other**







Elimination of ambiguity (cont.)

- Idea:
  - A statement appearing between a **then** and an **else** must be matched

stmt → matched_stmt
  | open_stmt

matched_stmt → **If** expr **then** matched_stmt **else** matched_stmt
  | **other**

open_stmt → **If** expr **then** stmt
  | **If** expr **then** matched_stmt **else** open_stmt

Elimination of left recursion

- A grammar is left recursive if it has a non-terminal A such that there is a derivation $A \overset{+}{=}> A\,\alpha$
- Top down parsing methods cant handle left-recursive grammars
- A simple rule for direct left recursion elimination:
  - For a rule like:
    - $A \to A\ \alpha \mid \beta$
  - We may replace it with
    - $A \to \beta\ \ A'$
    - $A' \to \alpha\ A' \mid \varepsilon$

Left recursion elimination (cont.)

- There are cases like following
  - S -> Aa | b
  - A -> Ac | Sd | ε
- Left recursion elimination algorithm:
  - Arrange the nonterminals in some order A1,A2,…,An.
  - For (each i from 1 to n) {
    - For (each j from 1 to i-1) {
      - Replace each production of the form Ai-> Aj γ by the production Ai -> $\delta_1 \gamma \mid \delta_2 \gamma \mid \ldots \mid \delta_k \gamma$ where Aj-> $\delta_1 \mid \delta_2 \mid \ldots \mid \delta_k$ are all current Aj productions
      - }
      - Eliminate left recursion among the Ai-productions
    - }

Left factoring

- Left factoring is a grammar transformation that is useful for producing a grammar suitable for predictive or top-down parsing.
- Consider following grammar:
  - Stmt -> **if** expr **then** stmt **else** stmt
  - | **if** expr **then** stmt
- On seeing input **if** it is not clear for the parser which production to use
- We can easily perform left factoring:
  - If we have A-> $\alpha\ \beta 1 \mid \alpha\ \beta 2$ then we replace it with
    - A -> α A'
    - A' -> $\beta 1 \mid \beta 2$

Left factoring (cont.)

- Algorithm
  - For each non-terminal A, find the longest prefix α common to two or more of its alternatives. If α <> ε, then replace all of A-productions A-> $\alpha \beta_1 \mid \alpha \beta_2 \mid \ldots \mid \alpha \beta_n \mid \gamma$ by
    - A -> α A' | γ
    - A' -> $\beta_1 \mid \beta_2 \mid \ldots \mid \beta_n$
- Example:
  - S -> I E t S | i E t S e S | a
  - E -> b

Tutorial (Set 1)

- Eliminate left recursion from following grammar.

  G=({S,A,B,C},{a,b,@},P,S)

  P:

  S→ABC

  A→Aa|@

  B→Bb|@

  C→Cc|@

Problem 2

$S \rightarrow aSc|B$

$B \rightarrow bdB|C$

$C \rightarrow b$

Find out string "abc" from above grammar using leftmost derivation process.

Left factoring rule 1

- When Grammar is

$$A \rightarrow \alpha\beta | \alpha\gamma$$

This can be equivalent to following grammar

$$A \rightarrow \alpha A`$$

$$A` \rightarrow \beta | \gamma$$

Left factoring rule 2

- When grammar is

$X \rightarrow \alpha A \gamma$

$A \rightarrow \beta 1 | \beta 2 | \beta 3 | ---- \beta n$

This can be equivalent to following grammar

$X \rightarrow \alpha \beta 1 \gamma | \alpha \beta 2 \gamma | \alpha \beta 3 \gamma | --- | \alpha \beta n \gamma$

Tutorial(Problem 3)

- Find out left factoring grammar for following grammar.

  $G=(\{S,A,B\},\{a,b\},P,S\})$

  P:

  $S \rightarrow aAbB|aAb$

  $A \rightarrow aA|a$

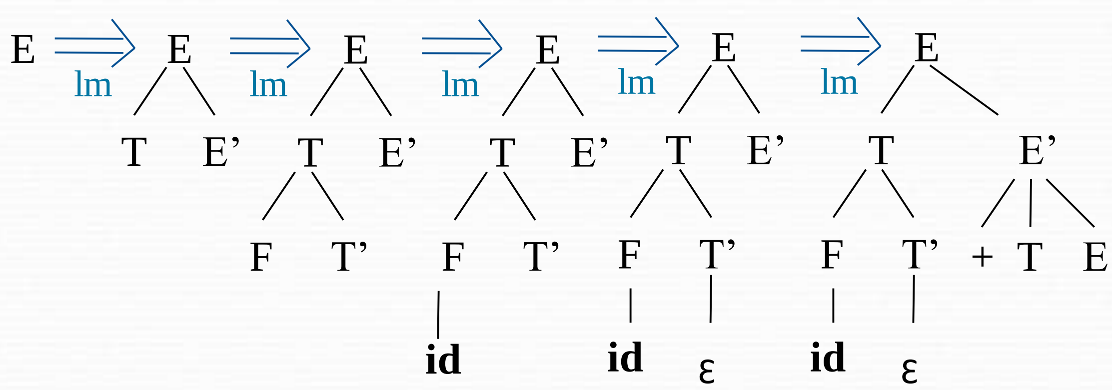
  $B \rightarrow bB|b$

Top Down Parsing

Introduction

- A Top-down parser tries to create a parse tree from the root towards the leafs scanning input from left to right
- It can be also viewed as finding a leftmost derivation for an input string
- Example: id+id*id

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



Recursive descent parsing

- Consists of a set of procedures, one for each nonterminal
- Execution begins with the procedure for start symbol
- A typical procedure for a non-terminal

```
void A() {
        choose an A-production, A->X1X2..Xk
        for (i=1 to k) {
                if (Xi is a nonterminal
                        call procedure Xi();
                else if (Xi equals the current input symbol a)
                        advance the input to the next symbol;
                else /* an error has occurred */
        }
}
```

Recursive descent parsing (cont)

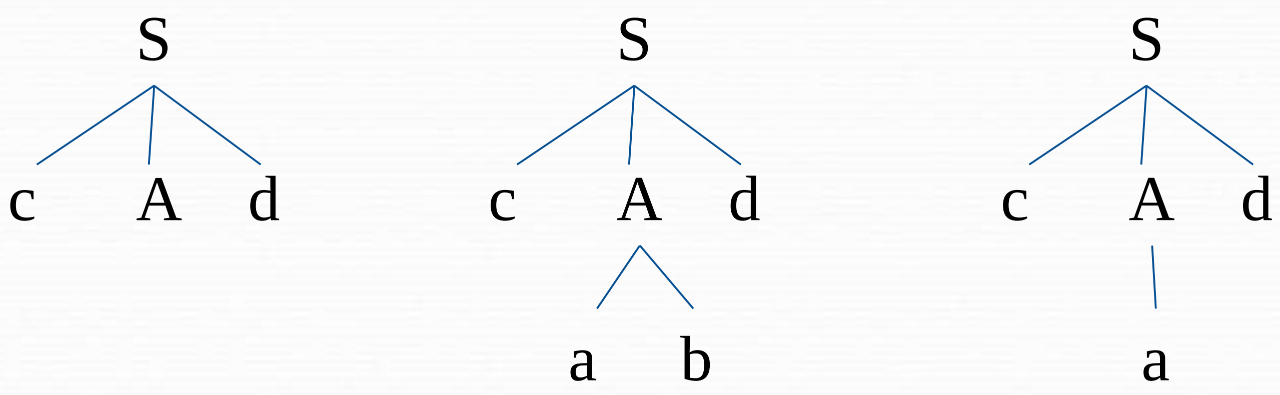
- General recursive descent may require backtracking
- The previous code needs to be modified to allow backtracking
- In general form it cant choose an A-production easily.
- So we need to try all alternatives
- If one failed the input pointer needs to be reset and another alternative should be tried
- Recursive descent parsers cant be used for left-recursive grammars

Example

S->cAd

A->ab | a

Input: cad



Problem 4

- Construct Recursive Decent Parser for following grammar

G=(NT,T,P,S)

NT=(S,E,E',T,T',V)

T=(+,*,id)

P:

$S \rightarrow E$

$E \rightarrow TE'$

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

$T \rightarrow VT'$

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

$V \rightarrow id$

```
S()
{         E();
}
E()
{        T();
         Edash();
}
Edash()
{
         if(ip==‘+’)
{        ADVANCE();
         T();
         Edash();
}
```

First and Follow

- First() is set of terminals that begins strings derived from
- If $\alpha \overset{*}{=}> \varepsilon$ then is also in First(ε)
- In predictive parsing when we have A-> $\alpha \mid \beta$, if First(α) and First(β) are disjoint sets then we can select appropriate A-production by looking at the next input
- Follow(A), for any nonterminal A, is set of terminals a that can appear immediately after A in some sentential form
  - If we have S $\overset{*}{=}>$ α Aa β for some α and β then a is in Follow(A)
- If A can be the rightmost symbol in some sentential form, then $ is in Follow(A)

Computing First

- To compute First(X) for all grammar symbols X, apply following rules until no more terminals or ε can be added to any First set:
  1. If X is a terminal then First(X) = {X}.
  2. If X is a nonterminal and $X \rightarrow Y_1 Y_2 \ldots Y_k$ is a production for some $k>=1$, then place a in First(X) if for some i a is in First(Y_i) and ε is in all of First(Y_1),…,First(Y_{i-1}) that is $Y_1 \ldots Y_{i-1} \stackrel{*}{\Rightarrow} \varepsilon$. if ε is in First($Y_j$) for $j=1,\ldots,k$ then add ε to First(X).
  3. If $X \rightarrow \varepsilon$ is a production then add ε to First(X)
- Example!

Computing follow

- To compute First(A) for all nonterminals A, apply following rules until nothing can be added to any follow set:
  1. Place \$ in Follow(S) where S is the start symbol
  2. If there is a production A-> $\alpha B \beta$ then everything in First(β) except ε is in Follow(B).
  3. If there is a production A->B or a production A-> α B β where First(β) contains ε, then everything in Follow(A) is in Follow(B)
- Example!

LL(1) Grammars

- Predictive parsers are those recursive descent parsers needing no backtracking
- Grammars for which we can create predictive parsers are called LL(1)
  - The first L means scanning input from left to right
  - The second L means leftmost derivation
  - And 1 stands for using one input symbol for lookahead
- A grammar G is LL(1) if and only if whenever A-> $\alpha \mid \beta$ are two distinct productions of G, the following conditions hold:
  - For no terminal a do α and β both derive strings beginning with a
  - At most one of α or β can derive empty string
  - If $\alpha \overset{*}{\Rightarrow} \varepsilon$ then β does not derive any string beginning with a terminal in Follow(A).

Construction of predictive parsing table

- For each production A-> α in grammar do the following:
  1. For each terminal a in First(α) add A-> in M[A,a]
  2. If ε is in First(α), then for each terminal b in Follow(A) add A-> ε to M[A,b]. If ε is in First(α) and $ is in Follow(A), add A-> ε to M[A,$] as well
- If after performing the above, there is no production in M[A,a] then set M[A,a] to error

Example

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

	First	Follow
F	{(,id}	{+, *,), $}
T	{(,id}	{+,), $}
E	{(,id}	{), $}
E'	{+,ε}	{), $}
T'	{*,ε}	{+,), $}

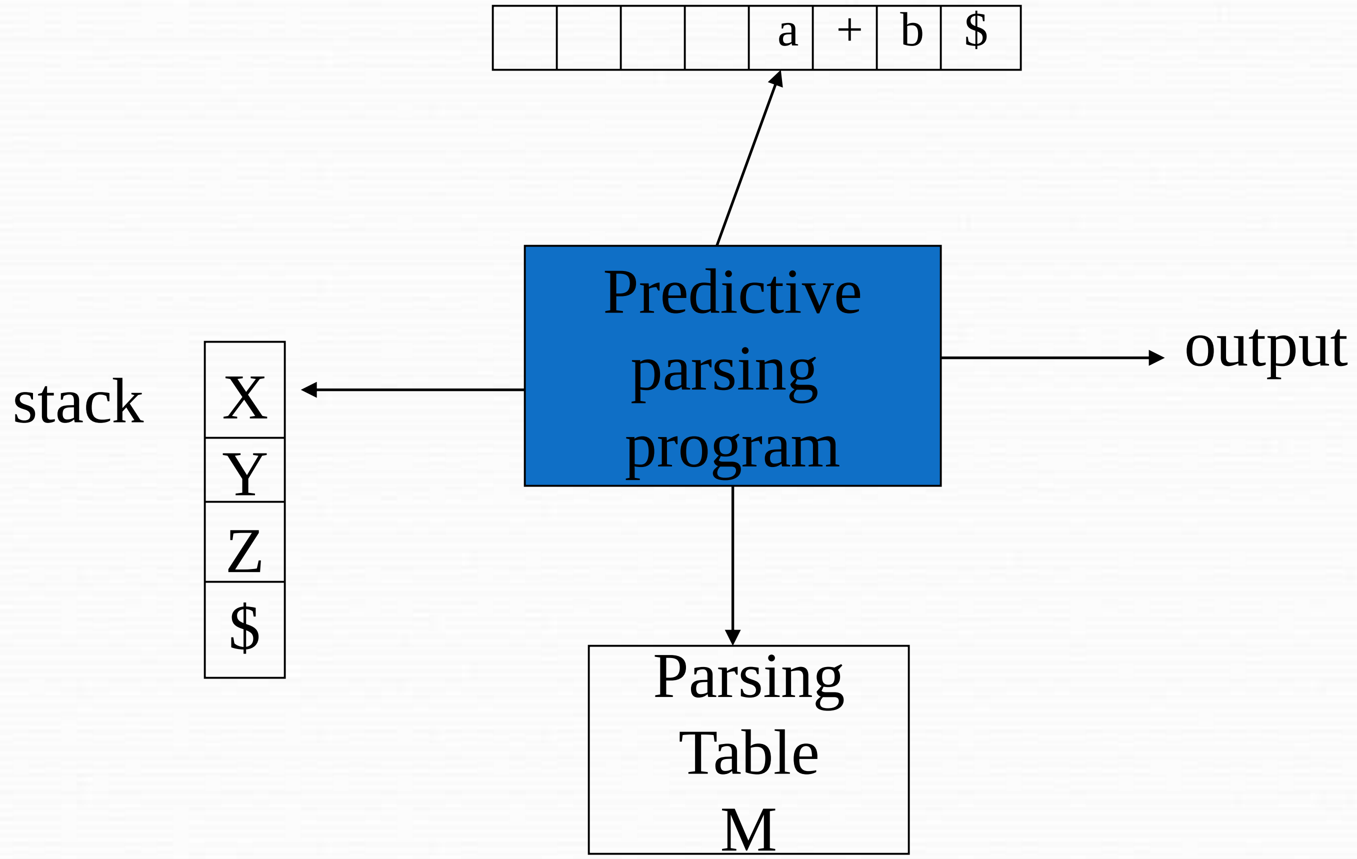
Non - terminal	Input Symbol					
	id	+	*	(	)	$
E	E -> TE'			E -> TE'		
E'		E' -> +TE'			E' -> ε	E' -> ε
T	T -> FT'			T -> FT'		
T'		T' -> ε	T' -> *FT'		T' -> ε	T' -> ε
F	F -> **id**			F -> (E)		

Another example

S -> iEtSS' | a
S' -> eS | ε
E -> b

Non - terminal	Input Symbol					
	a	b	e	i	t	$
S	S -> a			S -> iEtSS'		
S'			S' -> ε S' -> eS			S' -> ε
E		E -> b				

Non-recursive predicting parsing



Predictive parsing algorithm

```
Set ip point to the first symbol of w;
Set X to the top stack symbol;
While (X<>$) { /* stack is not empty */
   if (X is a) pop the stack and advance ip;
   else if (X is a terminal) error();
   else if (M[X,a] is an error entry) error();
   else if (M[X,a] = X->Y1Y2..Yk) {
          output the production X->Y1Y2..Yk;
          pop the stack;
          push Yk,...,Y2,Y1 on to the stack with Y1 on top;
   }
   set X to the top stack symbol;
}
```

Example

- id+id*id$

Matched	Stack	Input	Action
	E$	id+id*id$	

Error recovery in predictive parsing

- Panic mode
  - Place all symbols in Follow(A) into synchronization set for nonterminal A: skip tokens until an element of Follow(A) is seen and pop A from stack.
  - Add to the synchronization set of lower level construct the symbols that begin higher level constructs
  - Add symbols in First(A) to the synchronization set of nonterminal A
  - If a nonterminal can generate the empty string then the production deriving can be used as a default
  - If a terminal on top of the stack cannot be matched, pop the terminal, issue a message saying that the terminal was insterted

Example

Non - terminal	id	+	*	(	)	$
E	E -> TE’			E -> TE’	synch	synch
E’		E’ -> +TE’			E’ -> ε	E’ -> ε
T	T -> FT’	synch		T -> FT’	synch	synch
T’		T’ -> ε	T’ -> *FT’		T’ -> ε	T’ -> ε
F	F -> **id**	synch	synch	F -> (E)	synch	synch

Stack	Input	Action
E$	)id*+id$	Error, Skip)
E$	id*+id$	id is in First(E)
TE’$	id*+id$	
FT’E’$	id*+id$	
idT’E’$	id*+id$	
T’E’$	*+id$	
*FT’E’$	*+id$	
FT’E’$	+id$	Error, M[F,+]=synch
T’E’$	+id$	F has been poped