Chapter 4
Syntax Analysis

Predictive Parsing: Recursive

E → i EP
EP → + i EP | ∈

```
int E()
{
    if (l == 'i') {
        l = getchar();
        if(l=='#')
            return 1;
        if(!EP())
            return 0;
        else
            return 1;
    }
    else
        return 0;
}

int main()
{   l=getchar();
if(E())
    if (l == '#')
        printf("Parsing Successful");
    else
        printf("Error");
else
    printf("Error");
return 0;
}
```

Predictive Parsing: Recursive

E→ i EP
EP → + i EP | ∈

```
#include<stdio.h>

char l;
int EP()
{    if(l=='#')
         return 0;
     if (l == '+')
     {
         l = getchar();
         if(l=='#')
              return 0;
         if(l=='i')
          {
          l = getchar();
          if(l=='#')
               return 1;
          if(!EP())
                    return 0;
          else
                    return 1;
          }
          else
            return(0);
     }
     else
         return (0);
}
```

Recursive Descent Parser

```
int Expr()
 {
 if(!Term())
    return 0;
 if(next=='+')
  {
    next=Get_Char();
    if(next=='#')
      return 0;
    if(!Expr())
      return 0;
    else
      return 1;
  }
 else
     return 1;
}
```

E→ T + E | T
T→ F * T | F
F→ (E) | i

```
void main()
{
cursor=0;
gets(input);
next=Get_Char();
if(Expr())
  {
  if(next=='#')
    printf("\nValid Statement");
  else
    printf("\nInvalid Statement");
  }
else
  printf("\nInvalid Statement");
getch();
}
```

Recursive Descent Parser

```
int Factor()
{
if(next=='#')
   return 0;
if(next=='(')
 {
   next=Get_Char();
   if(next=='#')
     return 0;
   if(!Expr())
     return 0;
   if(next!=')')
     return 0;
   else
    {
     next=Get_Char();
      return 1;
    }
 }
if(next!='i')
   return 0;
else
 {
   next=Get_Char();
   return 1;
 }
}
```

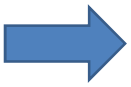
E→ T + E | T
T→ F * T | F
F→ (E) | i

```
int Term()
{
if(!Factor())
    return 0;
if(next=='*')
 {
   next=Get_Char();
   if(next=='#')
     return 0;
   if(!Term())
     return 0;
   else
     return 1;
 }
else
    return 1;
}
```

Predictive Parsing : Recursive

1. **To eliminate backtracking, what must we do/be sure of for grammar?**
   1. **no left recursion**
   2. ***apply left factoring***

E→ E + T | T
T→ T * F | F
F→ (E) | i



E → TE'
E' → + TE' | ∈

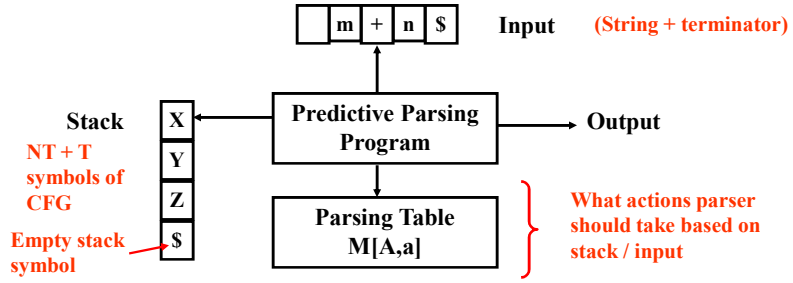
T → FT'
T' → * FT' | ∈

F → (E) | id

2. **Frequently, when grammar satisfies above conditions: current input symbol in conjunction with current non-terminal *uniquely determines* the production that needs to be applied.**

6

Non-Recursive / Table Driven



General parser behavior: X : top of stack a : current input

1. When X=a = $ halt, accept, success
2. When X=a ≠ $, POP X off stack, advance input, go to 1.
3. When X is a non-terminal, examine M[X,a]
   - if it is an error → call recovery routine
   - if M[X,a] = {X → UVW}, POP X, PUSH W,V,U
   - DO NOT expend any input

7

Algorithm for Non-Recursive Parsing

Set *ip* to point to the first symbol of w$;
repeat
  let X be the top stack symbol and *a* the symbol pointed to by *ip*;
  if X is terminal or $ **then**
    if X=a **then**
      pop X from the stack and advance *ip*
    else *error()*
  else /* X is a non-terminal */
    if M[X,a] = $X \rightarrow Y_1 Y_2 \ldots Y_k$ **then begin**
      pop X from stack;
      push $Y_k, Y_{k-1}, \ldots, Y_1$ onto stack, with Y_1 on top
      output the production $X \rightarrow Y_1 Y_2 \ldots Y_k$
    end
    else *error()*
until X=$ /* stack is empty */

Input pointer

May also execute other code based on the production used

8

Example

E → TE'
E' → + TE' | ∈
T → FT'
T' → * FT' | ∈
F → (E) | id

Our well-worn example !

Table M

Non-terminal	INPUT SYMBOL					
	id	+	*	(	)	$
E	E→TE'			E→TE'		
E'		E'→+TE'			E'→∈	E'→∈
T	T→FT'			T→FT'		
T'		T'→∈	T'→*FT'		T'→∈	T'→∈
F	F→id			F→(E)		

9

Trace of Example

STACK	INPUT	OUTPUT
$E	id + id * id$	
$E'T	id + id * id$	E→ TE'
$E'T'F	id + id * id$	T→ FT'
$E'T'id	id + id * id$	F → id
$E'T'	+ id * id$	
$E'	+ id * id$	T' → ∈
$E'T+	+ id * id$	E' → +TE'
$E'T	id * id$	
$E'T'F	id * id$	T→ FT'
$E'T'id	id * id$	F → id
$E'T'	* id$	
$E'T'F*	* id$	T' → *FT'
$E'T'F	id$	
$E'T'id	id$	F → id
$E'T'	$	
$E'	$	T' → ∈
$	$	E' → ∈

Expend Input

10

Leftmost Derivation for the Example

The leftmost derivation for the example is as follows:

$$E \Rightarrow TE' \Rightarrow FT'E' \Rightarrow id\ T'E' \Rightarrow id\ E' \Rightarrow id + TE' \Rightarrow id + FT'E'$$
$$\Rightarrow id + id\ T'E' \Rightarrow id + id * FT'E' \Rightarrow id + id * id\ T'E'$$
$$\Rightarrow id + id * id\ E' \Rightarrow id + id * id$$

11