innovate achieve lead

Recursive Descent Parsing

Dr. Shashank Gupta
Assistant Professor
Department of Computer Science and Information Systems

BITS Pilani
Pilani Campus

Calculation of First Sets

- **if** p is a token/terminal symbol then First (p) = {p}

- **if** P -> ε is a production then First (P) = {ε}

Calculation of First Sets

- **if** P is a non-terminal and P -> Q_1 Q_2 Q_3-----Q_k is a production then

  if for some i, First (Q_i) = {x} and ε is in all of First(Q_j) (such that j < i) then

  First (P) = {x}

- **if** ε is in First (Q_1) ------ First (Q_k) then First (P) = {ε}

∈

Example

Calculate the First of all non-terminals in the following grammar
`{{S,B,C},{a,b,c,d}, P, S}.`

$$S \rightarrow Bb \mid Cd$$

$$B \rightarrow aB \mid \in$$

$$C \rightarrow cC \mid \in$$

Variables/Non Terminals	First
S	{a, b, c, d}
B	{a, ∈}
C	{c, ∈}

Example

Calculate the First of all non-terminals in the following grammar
`{{S,A,B,C}, {a, b, d, g, h}, P, S}.`

$$S \rightarrow ACB \mid CbB \mid Ba$$

$$A \rightarrow da \mid BC$$

$$B \rightarrow g \mid \in$$

$$C \rightarrow h \mid \in$$

Variables/Non-terminals	First
S	{d, g, h, ε, b, a}
A	{d, g, h, ε}
B	{g, ε}
C	{h, ε}

Recursive Descent Parser

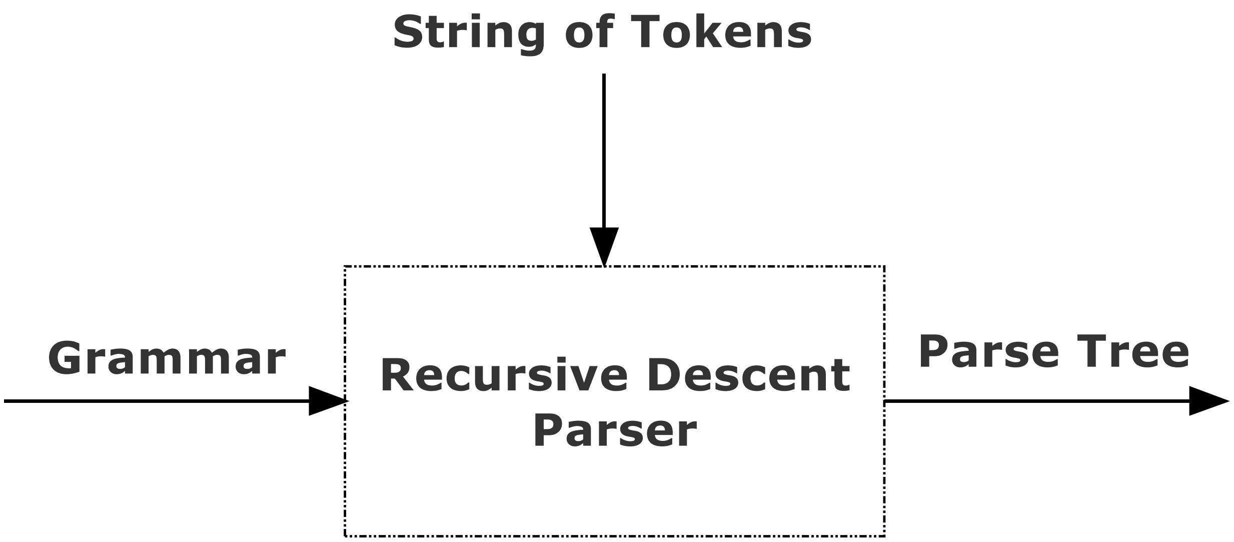
Recursive Descent Parser

It is a top down method of syntax analysis in which a set of recursive procedures are executed to parse the stream of tokens.

- A procedure is associated with each non-terminal of the grammar.

It is usually built from a set of mutually-recursive procedures or a non-recursive equivalent where each such procedure usually implements one of the production rules of the grammar.

Recursive Descent Parser



How to Develop the Procedures?

Develop a procedure for each Non-terminal of the Grammar rule.

- This procedure will capture all the specifications on the RHS of the grammar rule.

In each procedure, perform a match operation on hitting any token (in the right hand side of the grammar) with the current token in the input that needs to be parsed.

- If match occurs, increment the lookahead pointer to the next input token that needs to be parsed else, throw syntax error.

Syntax of Procedure for each Non-terminal

```
void A() {
Choose an A-production A → X1X2 … Xk
for i ← 1 … 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
 // error }
```

Example

Write procedures for each of the non-terminals of this grammar {**{E, E'}, {i, +}, P, E}** using recursive descent parsing and parse the following input: ***i + i $***

$$E \rightarrow i E^{'}$$

$$E^{'} \rightarrow + i E^{'} \mid \in$$

Recursive Descent Parser

String to be Parsed: ***i + i $***

$$E \rightarrow i\, E'$$

$$E' \rightarrow +\, i\, E' \,|\, \in$$

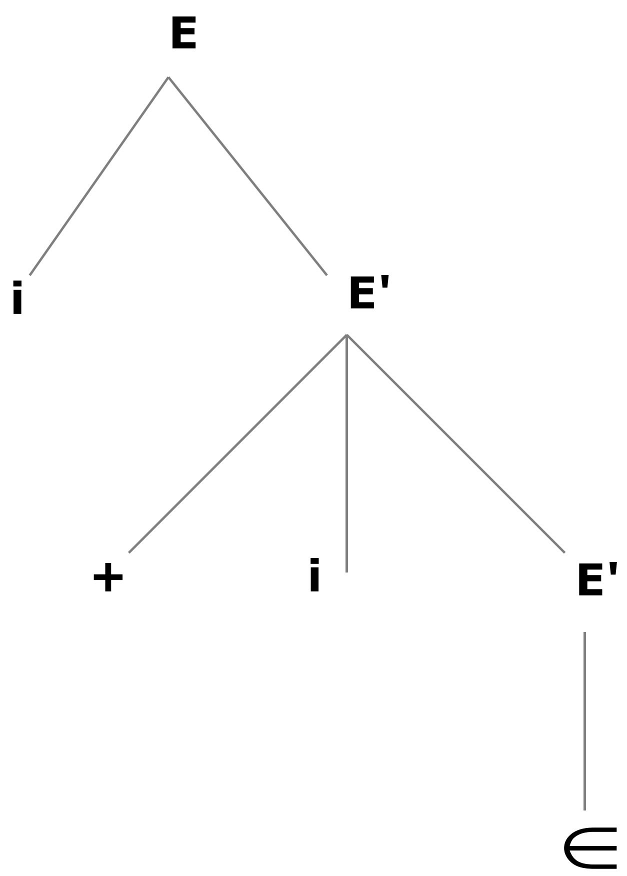
```
main(){
        E();
        if lookahead == '$'
                Printf("Parsing
Successful");}
```

```
E(){
        if lookahead == 'i'{
                match (i);
                E'();}
}
```

```
match (char t){
   if lookahead == 't'
        lookahead = nexttoken();
   else
        Error;}
```

```
E′ (){
        if lookahead == '+'
                match (+);
                match (i);
                E′();
        else
                return;}
```

Parse Tree for `i + i $`



Example

Write procedures for each of the non-terminals of this grammar `{{type, simple}, {↑, [, ], id, array, of, int, char, num, dotdot}, P, type}` using recursive descent parsing and parse the following input:

array [num dotdot num] of int

$$type \rightarrow simple \,|\, \uparrow id \,|\, array[\, simple \,]\, of\ type$$

$$simple \rightarrow \mathrm{int} \,|\, char \,|\, num\ dotdot\ num$$

Recursive Descent Parser

$$type \rightarrow simple \mid \uparrow id \mid array[\, simple \,] of\ type$$

$$simple \rightarrow \text{int} \mid char \mid num\ dotdot\ num$$

array [num dotdot num] of int

```
type(){
if lookahead == First (simple)
        simple ();
else if lookahead == '↑'{
        match (↑);
        match (id);}
else if lookahead == 'array'{
        match (array);
        match ([);
        simple ();
        match (]);
        match (of);
        type ();}
else
        Error;}
```

```
simple () {
if lookahead == 'int'
        match (int);
else if lookahead == 'char'
        match (char);
else if lookahead == 'num'{
        match (num);
        match (dotdot);
        match (num);}
else
        Error;
}
```

```
match (char t){
if lookahead == 't'
        lookahead = nexttoken();
else
        Error;
}
```

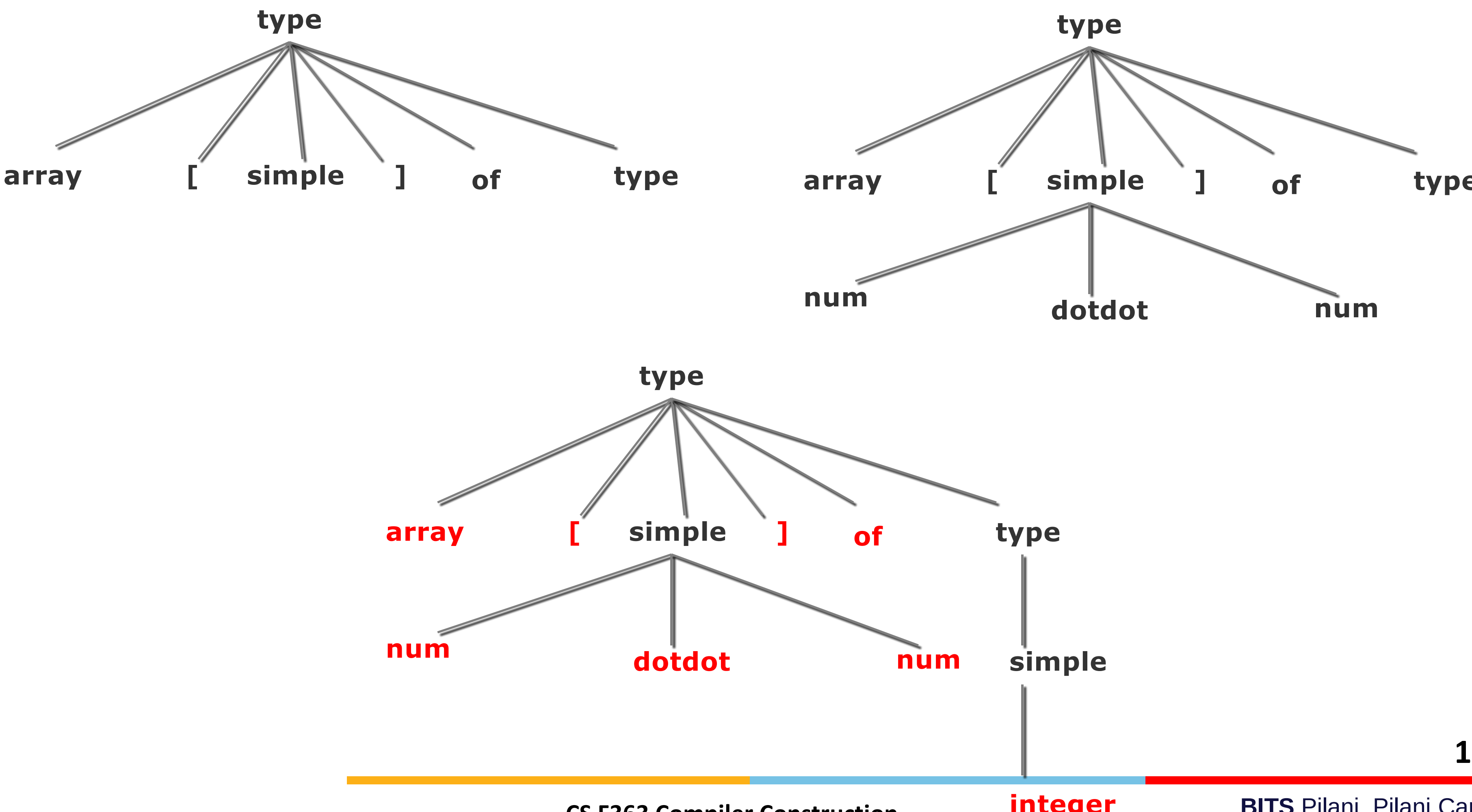
```
main () {
        type();
        if lookahead == '$'
                "Parsing Success"
}
```

Example

$$type \rightarrow simple \mid \uparrow id \mid array[\ simple\ ]of\ type$$

$$simple \rightarrow \mathrm{int} \mid char \mid num\ dotdot\ num$$

Parse `array [ num dotdot num ] of integer`



ISSUES IN RECURSIVE DESCENT PARSER

Limitations with Recursive-Descent Parsing

- Consider a grammar with two productions

$$X \rightarrow \gamma 1$$

$$X \rightarrow \gamma 2$$

- Suppose $\text{FIRST}(\gamma 1) \cap \text{FIRST}(\gamma 2) \neq \phi$
- Say a is the common terminal symbol
- Function corresponding to X will not know which production to use on input token α.

Recursive Descent Parser with Backtracking

To support backtracking

- All productions should be tried in some order
- Failure for some production implies we need to try remaining productions
- Report an error only when there are no other rules

Left Recursion

- A recursive descent parser may loop forever for the following production of the form:

$$A \rightarrow A\alpha$$

- From the Grammar $A \rightarrow A\alpha \mid \beta$
- Left recursion can be removed by rewriting the grammar as

$$A \rightarrow \beta A'$$

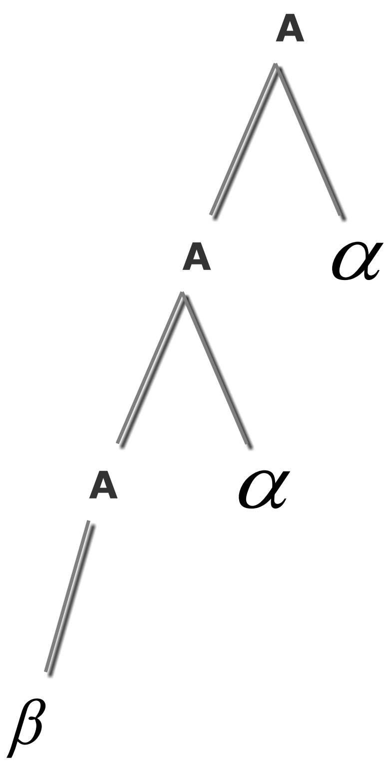
$$A' \rightarrow \alpha A' \mid \in$$

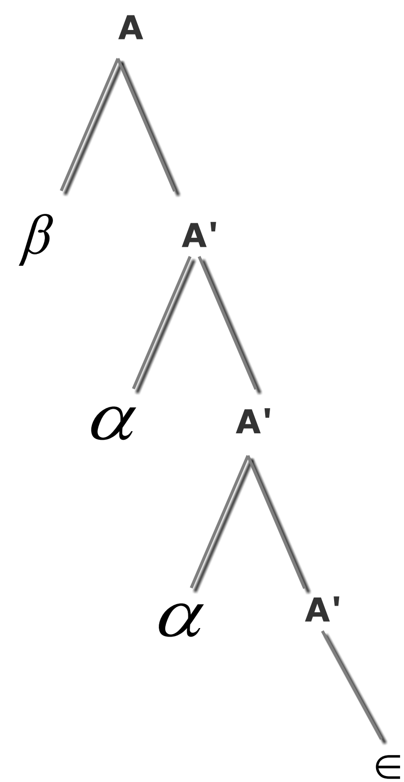
Example

$$A \rightarrow A\alpha \mid \beta$$

$$A \rightarrow \beta A'$$

$$A' \rightarrow \alpha A' \mid \in$$





Example

$$E \rightarrow E + T \mid T$$
$$T \rightarrow T * F \mid F$$
$$F \rightarrow (E) \mid id$$

$$E \rightarrow T E'$$
$$E' \rightarrow + T E' \mid \epsilon$$
$$T \rightarrow F T'$$
$$T' \rightarrow * F T' \mid \epsilon$$
$$F \rightarrow (E) \mid id$$