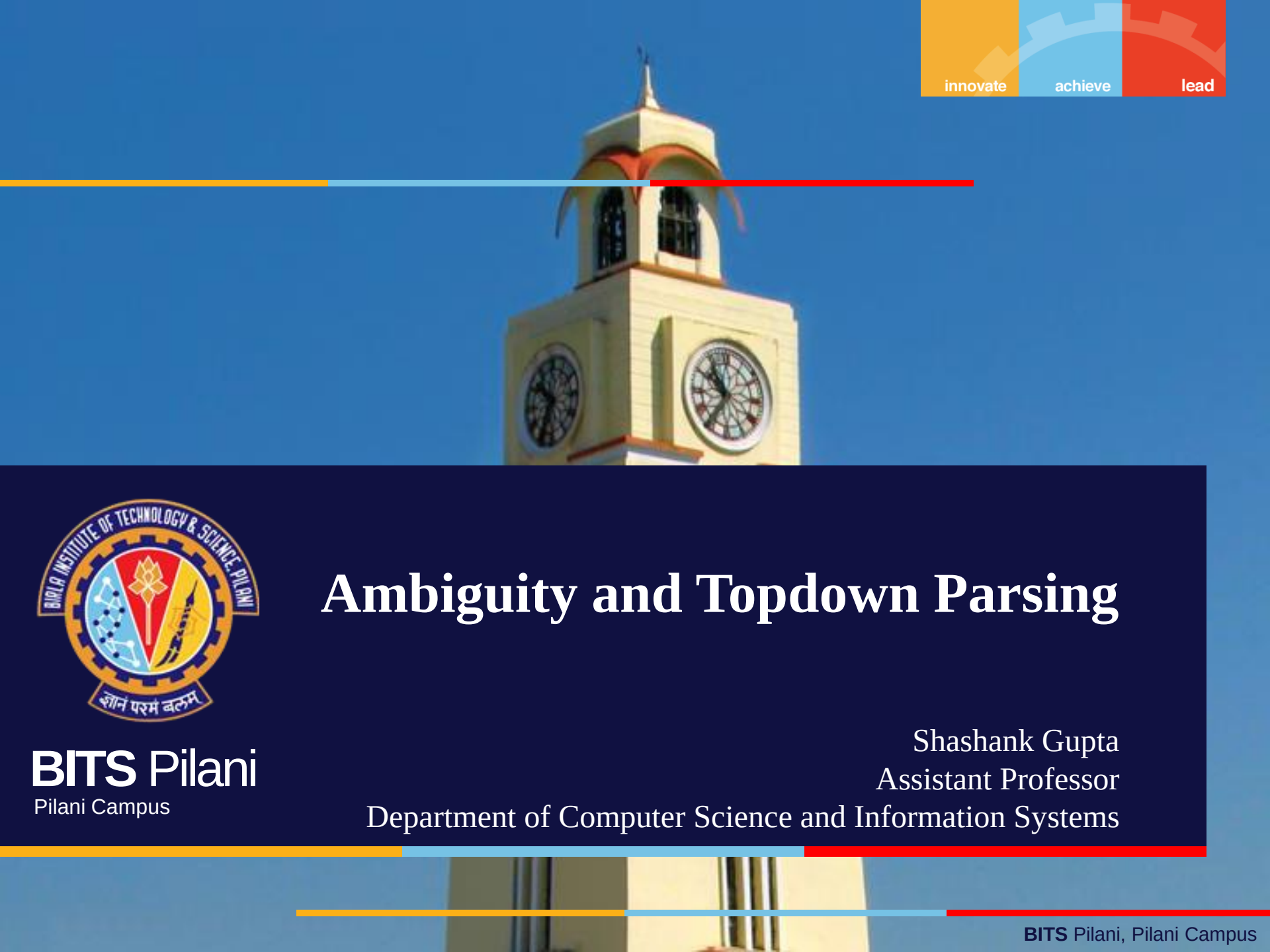


innovate achieve lead

Ambiguity and Topdown Parsing

Shashank Gupta
Assistant Professor
Department of Computer Science and Information Systems

BITS Pilani
Pilani Campus

Ambiguity

- A Grammar can have more than one parse tree for a string.
- Consider the Grammar

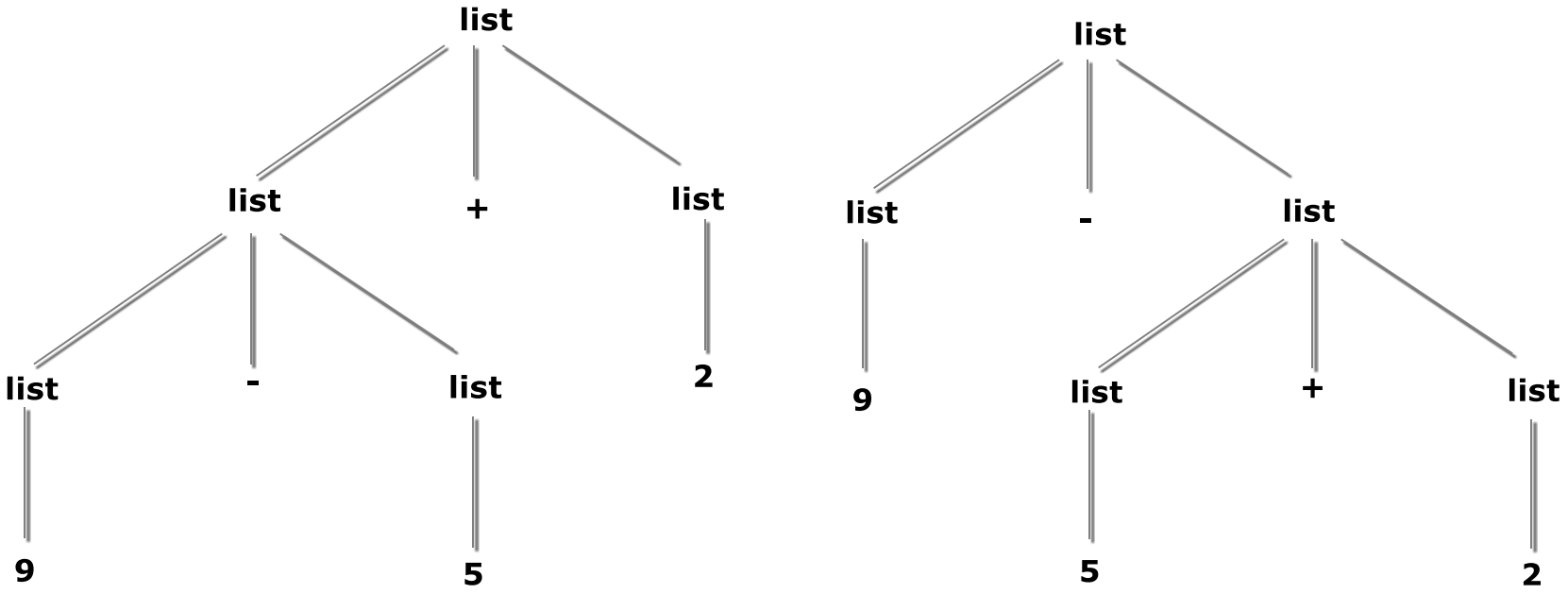
$$List \rightarrow List + List$$
$$\rightarrow List - List$$
$$Digit \rightarrow 0|1|2|---|9$$

MODIFIED GRAMMAR

$$List \rightarrow List + Digit$$
$$|List - Digit$$
$$|Digit$$
$$Digit \rightarrow 0|1|2|3---|9$$

OLD GRAMMAR

Example



String 9-5+2 has now two parse trees

Ambiguity

Ambiguity in a grammar must be handled in such a way that it should not produce multiple parse trees for the same string.

Hence, the way in which one writes the grammar is very important.

How to Handle Ambiguity?

Ambiguity is awkward since, semantics of the programs can be inappropriate.

Must be resolved by

- Rewriting the Grammar.
- Utilize some precedence and associativity rules.

Infeasible to transform an ambiguous grammar into an unambiguous grammar by using some well-defined algorithm.

Associativity Rules

If an operand has operator on both the sides, the side on which operator takes this operand is the associativity of that operator.

- `*`, `/`, `+`, `-` and `^`, `=` are all left associative and right associative operators.
- For e.g. in `P*Q*R`, `Q` must be evaluated by left `*`.

Associativity Rules

Grammar to generate the strings with right associative operators

$$right \rightarrow letter = right \mid letter$$

$$letter \rightarrow a \mid b \mid - - - \mid z$$

Precedence Rules

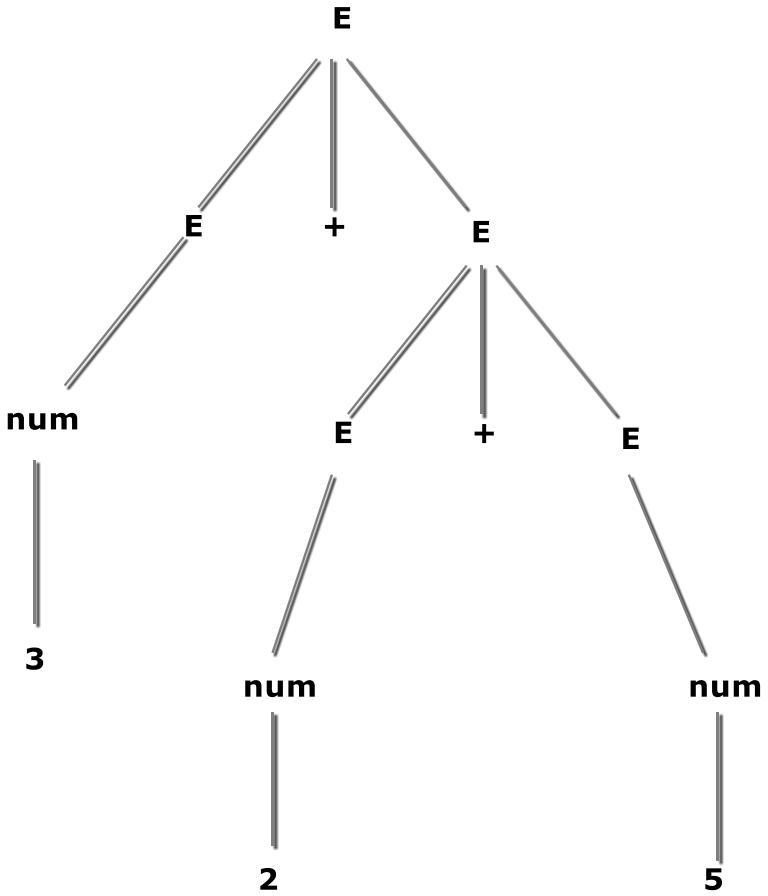
`5-10+13` has two different interpretations because of two different parse trees corresponding to

- `(5-10)+13` and `5-(10+13)`

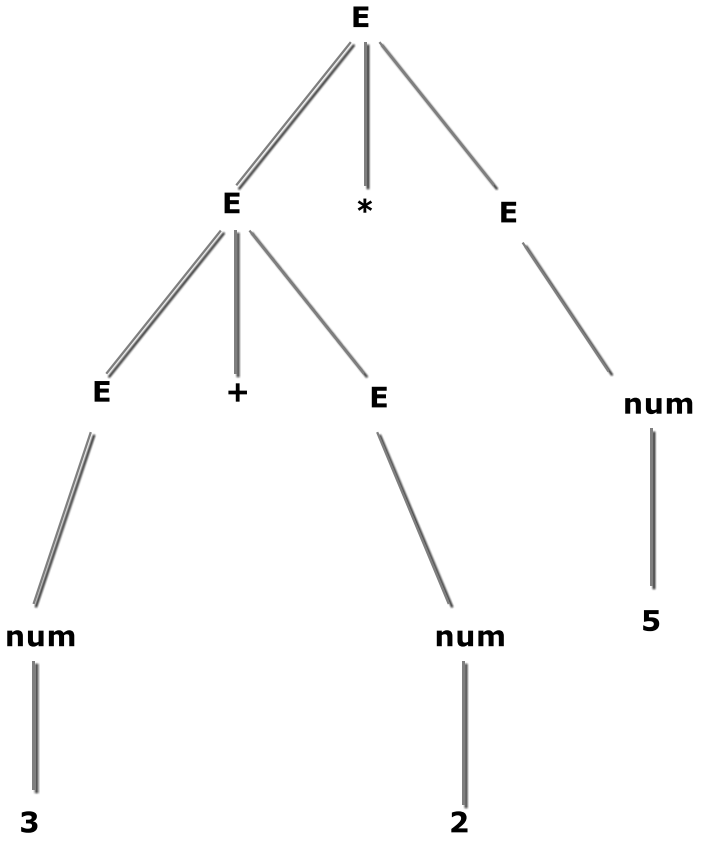
Precedence rules determine the correct interpretation.

Ambiguous Grammar for Arithmetic Expressions

$$E \rightarrow E+E \mid E*E \mid (E) \mid num$$



Parse Tree for 3+2+5



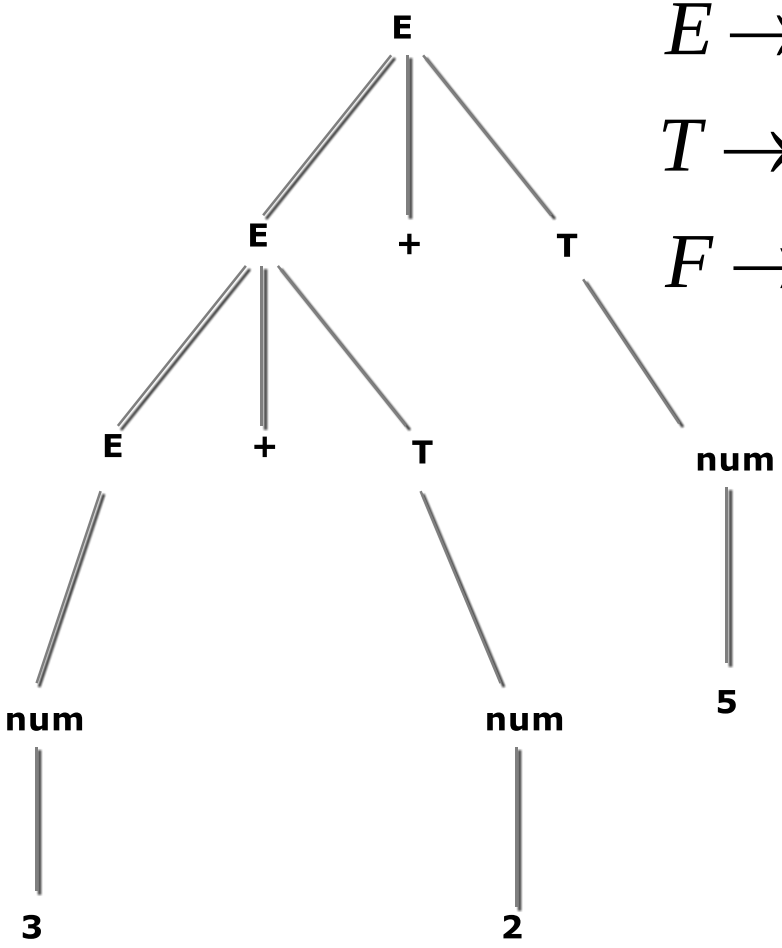
Parse Tree for 3+2*5

Another Grammar for Arithmetic Expressions

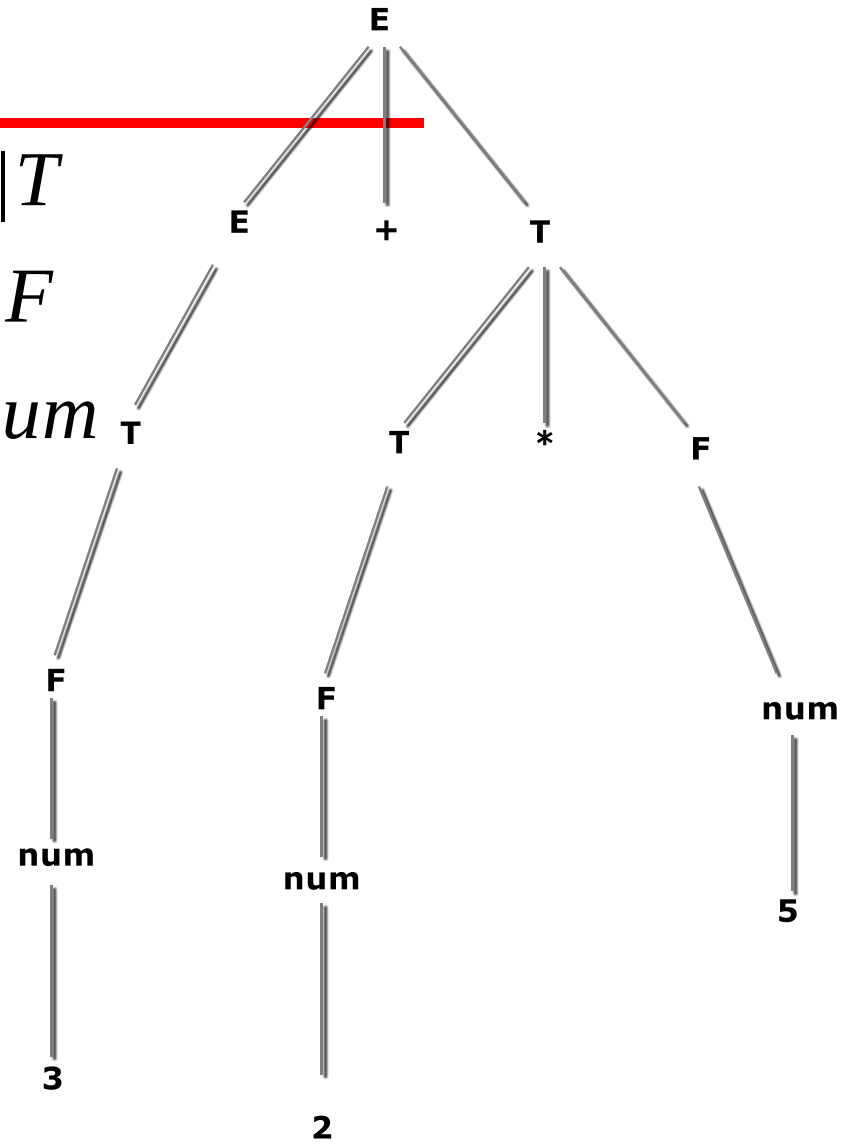
$$E \rightarrow E + T \mid T$$

$$T \rightarrow T * F \mid F$$

$$F \rightarrow (E) \mid num$$



Parse Tree for 3+2+5



Parse Tree for 3+2*5

Ambiguity in Programming Languages

Dangling Else Problem

When there are multiple IF statements and a single ELSE then the ELSE part doesn't get a clear view to go with which IF.

Ambiguity in Programming Languages

if A then if B then C1 else C2

```
If A then
{
     if B then
          C1
     else
          C2
}
```

if A then if B then C1 else C2

```
If A then
{
     if B then
          C1
}
else
          C2
```

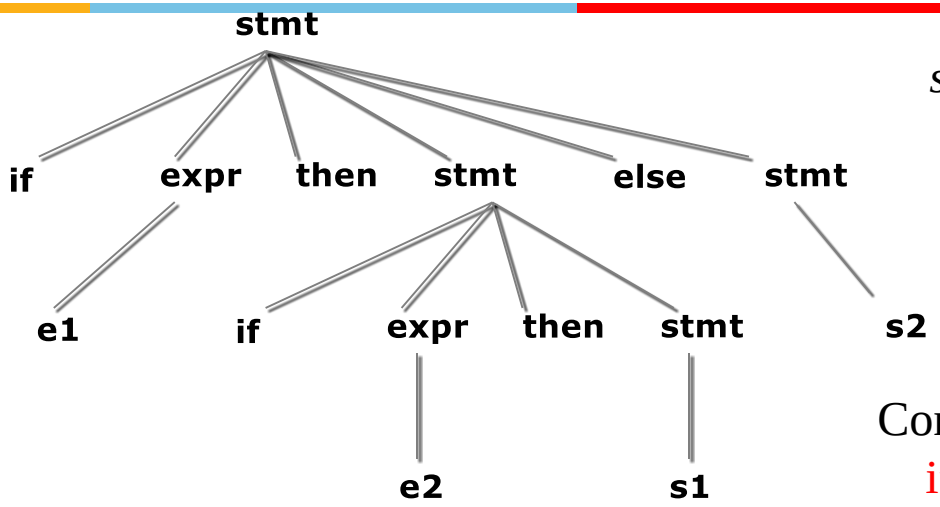
Ambiguity in Programming Languages

$$stmt \rightarrow if\ \exp\ then\ stmt$$
$$| if\ \exp\ then\ stmt\ else\ stmt$$

- Consider the following string

```
if e1 then if e2 then s1 else s2
```

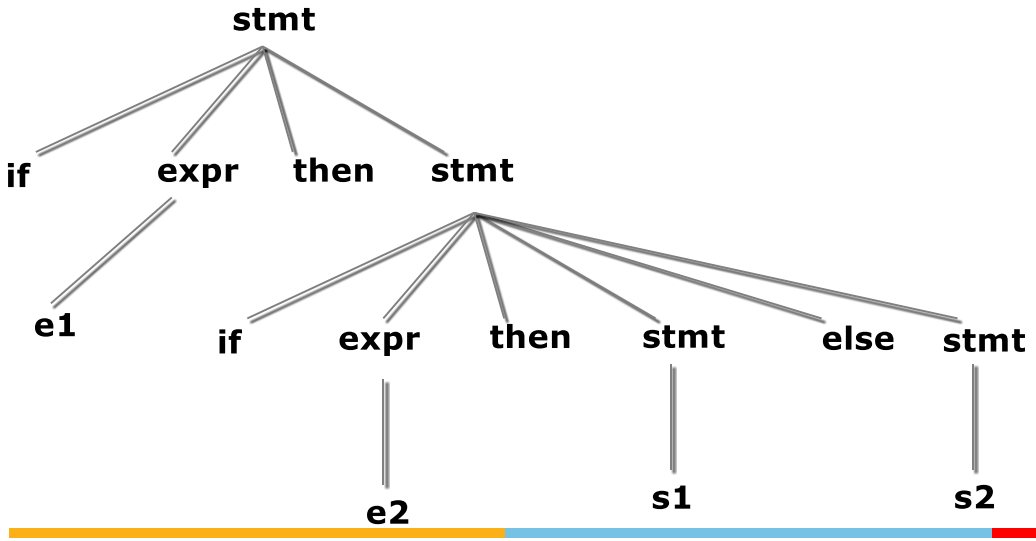
Ambiguity in Programming Languages



$$stmt \rightarrow if\ \exp\ then\ stmt$$
$$| if\ \exp\ then\ stmt\ else\ stmt$$

Consider the following string

if e1 then if e2 then s1 else s2



Resolving Ambiguity in Programming Languages

Match each `else` with the closest previous `then`.

Rewrite the grammar in proper form.

Resolving Ambiguity in Programming Languages

$$stmt \rightarrow matchedstmt$$
$$\quad | unmatchedstmt$$
$$matchedstmt \rightarrow if\ \exp\ then\ matchedstmt$$
$$\qquad\qquad\qquad\qquad else\ matchedstmt$$
$$\qquad\qquad\qquad | others$$
$$unmatchedstmt \rightarrow if\ \exp\ then\ stmt$$
$$\qquad\qquad\qquad | if\ \exp\ then\ matchedstmt$$
$$\qquad\qquad\qquad\qquad else\ unmatchedstmt$$

Parsing

Technique of finding whether a given string can be generated by the language specified by the Grammar.

Parsing has two categories

- Top-down Parsing
- Bottom-up Parsing

Parsing

Top-down Parsing

- Construction of the parse tree initiate at the root node (i.e. from the Start symbol) and proceeds towards leaves (token or terminals)

Bottom-up Parsing

- Construction of the parse tree starts from the leaf nodes (tokens or terminals of the grammar) and proceeds towards root (Start symbol).

Example

$$List \rightarrow \underline{List} + Digit$$
$$\rightarrow \underline{List} - Digit + Digit$$
$$\rightarrow \underline{Digit} - Digit + Digit$$
$$\rightarrow 9 - \underline{Digit} - Digit$$
$$\rightarrow 9 - 5 + \underline{Digit}$$
$$\rightarrow 9 - 5 + 2$$

Top-down Parsing

Construction of a parse tree is done by starting the root labelled by a start symbol.

Repeat the following two steps:

- at a node labelled with non terminal A, select one of the productions of A and construct the children nodes.
- find the next node at which the subtree is to be constructed.

Example: Top-Down Parsing

This Grammar generates the type information of Pascal

Consider a Grammar of the form {V,T,P,S} i.e.{{type, simple}, {↑, [,], id, array, of, int, char, num, dotdot}, P, type}

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

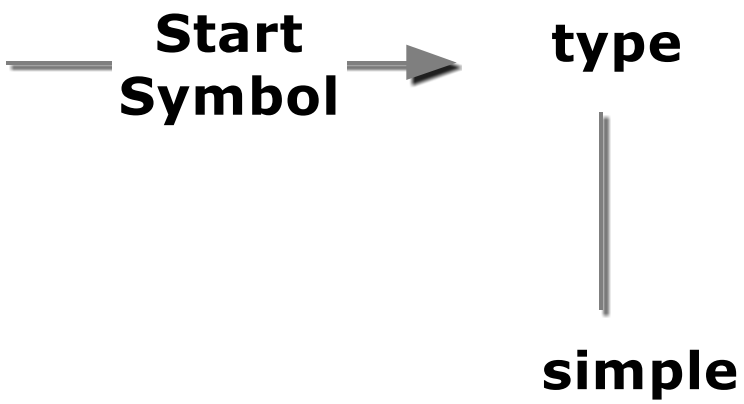
$$simple \rightarrow \text{int} \mid char \mid num\ dotdot\ num$$

Example

Parse `array [ num dotdot num ] of int`

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

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

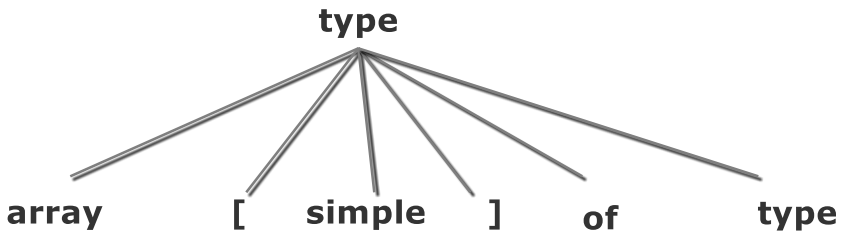


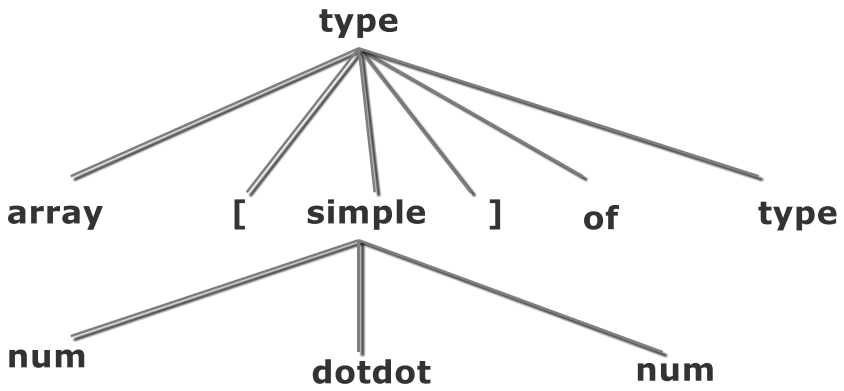
Example of Backtracking

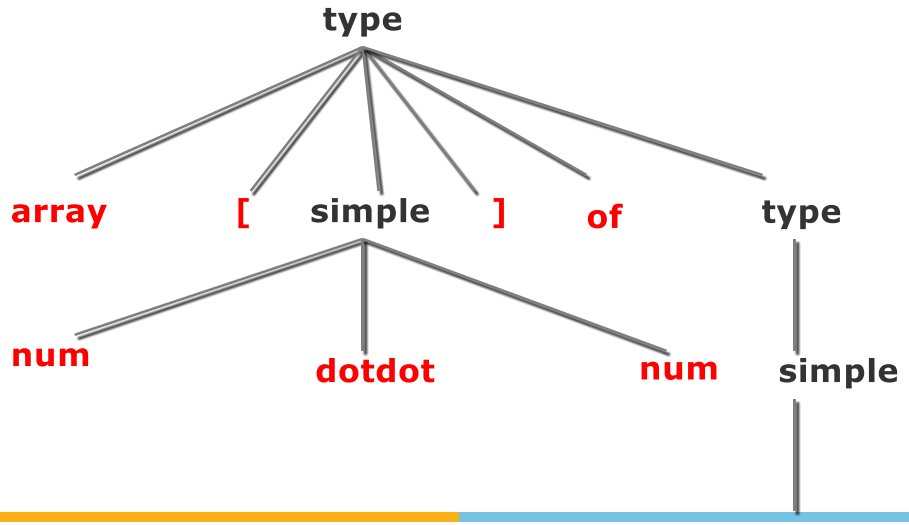
$type \rightarrow simple \mid \uparrow id \mid array[\ simple\ ]\ of\ type$

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

Parse `array [ num dotdot num ] of integer`







How to resolve Backtracking?

Backtracking is really a complex process to implement.

If parser can determine that what is the “first character of the string produced when a production rule is applied”, and comparing it to the current token in the input string it sees

- It can wisely take decision on which production rule to apply.

How to Resolve Backtracking?

```
S -> cAd
A -> bc|a
```

And the input string is "cad".

If the parser knew that after reading character 'c' in the input string and applying S->cAd, next character in the input string is 'a'.

Then, it would have ignored the production rule A->bc (because 'b' is the first character of the string produced by this production rule, not 'a'), and directly use the production rule A->a

First Set

`FIRST(X)` for a grammar symbol X is the set of tokens that begin the strings derivable from X.

Let there be a production

A -> α

- then `First ( α )` is the set of tokens that appear as the **first** token in the strings generated from α **(string of terminals and non-terminals)**.

Example

Consider a Grammar of the form {V,T,P,S} i.e.{{type, simple}, {↑, [,], id, array, of, int, char, num, dotdot}, P, type}

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

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

First (simple) = {int, char, num}
First (num dotdot num) = {num}

∈

Example

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

$$S \rightarrow ABCDE$$

$$First(S) = First(ABCDE) = (First(A) - \in) \cup (First(B) - \in) \cup (First(C)) = \{a, b, c\}$$

$$A \rightarrow a \mid \in$$

$$B \rightarrow b \mid \in$$

$$C \rightarrow c$$

$$D \rightarrow d \mid \in$$

$$E \rightarrow e \mid \in$$

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