

Syntax



Syntax

- Syntax is the structure of a language.
- In early days syntax is described by lengthy English explanation and many examples.
- One of the great advances in programming languages has been the development of a formal system for describing syntax by Context Free Grammar (CFG).
- Backus-naur form (BNF) is a notational system used to describe the CFG.

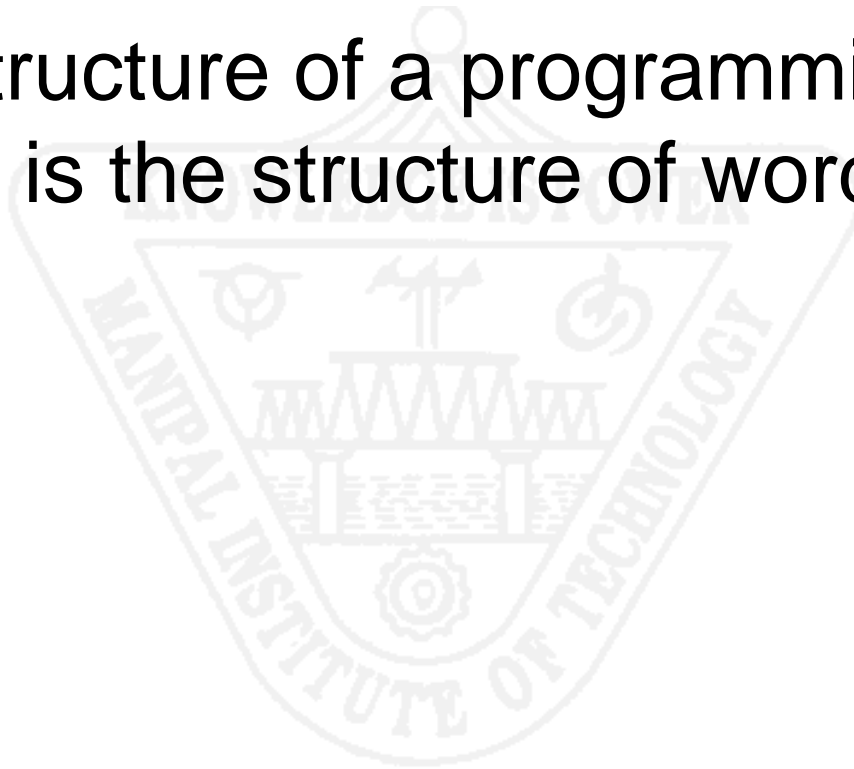
Variations of BNF

Three basic forms:

- Original BNF
- Extended BNF (EBNF)
- Syntax diagrams

Lexical structure of a programming language

- Lexical structure of a programming language is the structure of words or tokens.



Categories of Tokens

- Reserved words / keywords (if, while etc)
- Literals / constants (42, “hello” etc)
- Special symbols (; , + etc)
- Identifiers (x24, monthly_balance etc)

Tokens

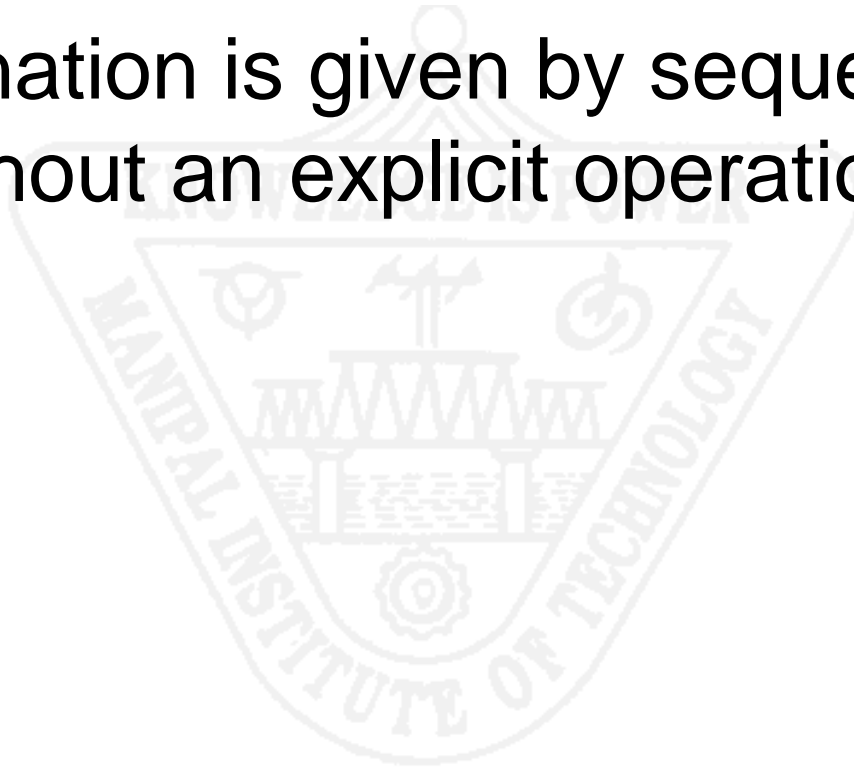
- Principle of longest substring is applied in determining tokens.
- Eg:- doif single identifier where in do, if are two reserved words
- The format of a program can affect the way tokens are recognized.
- For example, the principle of longest substring requires that certain tokens be separated by **token delimiters or white space.**

Regular expression

- Tokens in a programming language are often described in English, but they can also be described formally by **regular expressions**, which are descriptions of patterns of characters.
- Regular expressions have 3 basic operations
 - 1] concatenation
 - 2] repetition
 - 3] choice or selection

Concatenation

- Concatenation is given by sequencing the items without an explicit operation.



Repetition

- Repetition is indicated by the use of an asterisk after the item or a pattern of items which is / were to be repeated.

Choice / Selection

- Choice is indicated by a vertical bar between the items from which the selection is to be made.

$(a|b)^*c$

- Is a regular expression indicating 0 or more repetitions of either the characters a or b (choice), followed by a single character c (concatenation);
- Strings that match this regular expression include ababaac, aac, babbc, and just c itself, but not aaaab or bca.

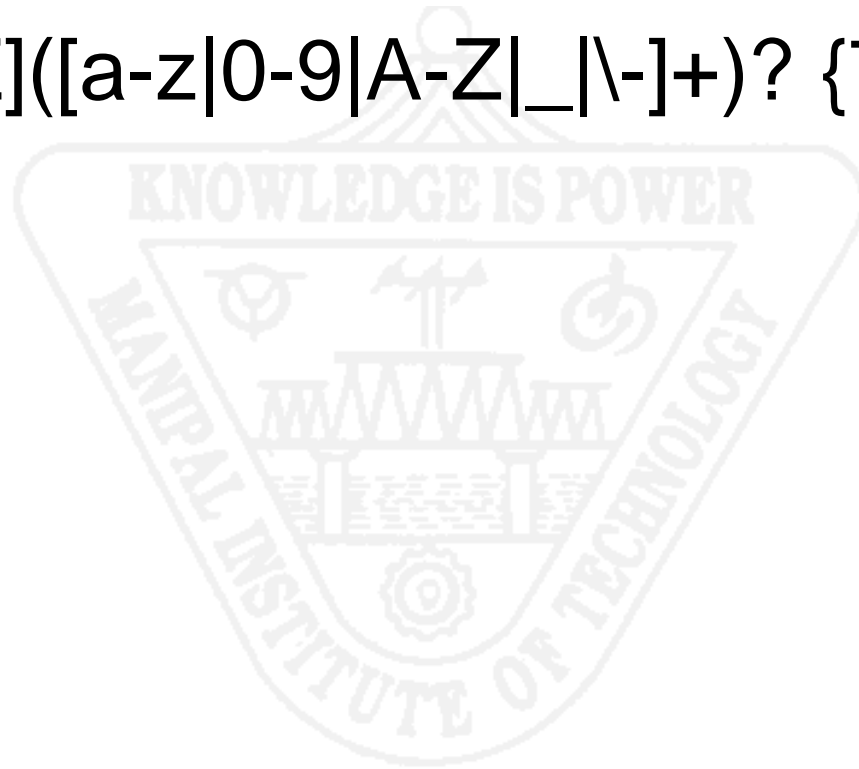
Regular expression notations extended by additional operations

- Square brackets with a hyphen indicate a range of characters.
- + indicates one or more repetitions
- ? Indicates an optional item
- A period (.) indicates any character.
- \ is used as escape sequence.
- { } number of letters in tokens
- { 3,5} minimum of 3 and maximum of 5 letters

- $[0-9]^+$
- Is a regular expression for simple integer constants consisting of one or more digits (characters between 0 and 9)
- $[0-9]^+(\backslash.[0-9]^+)?$
- The above regular expression represent the Unsigned floating point number....one or more digits followed by an optional fractional part consisting of a decimal point

User Name

- $[a-z \mid A-Z]([a-z|0-9|A-Z|_|\-|+])? \{7,15\}$



E-mail ID

- $[a-z|A-Z] [a-z|A-Z|0-9|_|\.|\\-]^+ @$
 $[a-z|A-Z|0-9|_|\.|\\-]^+ \\.([a-z|A-Z|\\.|]) \{2,6\}$

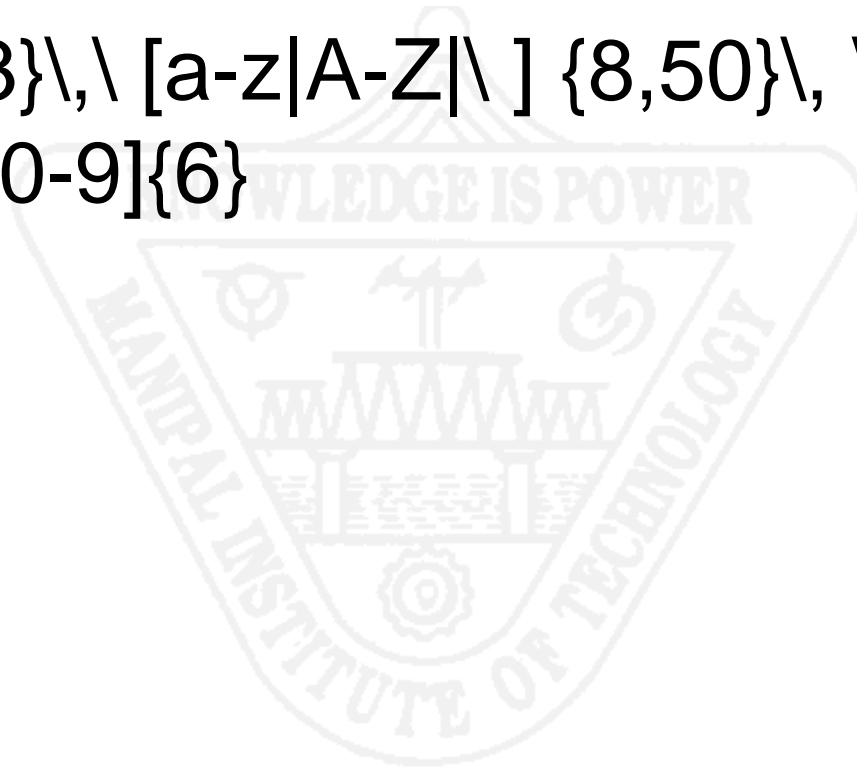
Postal Address

- #32, downstream, USA -123456



Solution

- $\#[0-9]\{1,3\}\backslash,\backslash[a-zA-Z]\backslash\]\{8,50\}\backslash,\backslash[a-zA-Z]\backslash\]\{3,15\}\backslash-[0-9]\{6\}$



More Tokens

- **String and character constants**
- string \rightarrow " char* "
- character \rightarrow ' char '
- char \rightarrow *not*(" | ' | \) | escape
- escape \rightarrow \(" | ' | \ | n | r | t | v | b | a)
- **White space**
- whitespace \rightarrow <space> | <tab> | <newline> |
- **comment**
- comment \rightarrow /* *not*(*/) */

Scanner

- The UNIX lex utility can be used to automatically turn a regular expression description of the tokens of a language into a scanner. (A successor to lex, called flex, or fast lex, is freely available and runs on most operating systems.)
- Scanners can be constructed by hand relatively easily.
- To give you a sense of such a scanner construction, C code for a simple scanner is illustrated.

Flex

- **FLEX** (Fast LEXical analyzer generator) is a tool for generating **scanners**.
- A scanner, sometimes called a tokenizer, is a program which recognizes lexical patterns in text.
- The flex program reads user-specified input files, or its standard input if no file names are given, for a description of a scanner to generate.
- The description is in the form of pairs of regular expressions and C code, called rules.
- Flex generates a C source file named, "lex.yy.c", which defines the function yylex().
- The file "lex.yy.c" can be compiled and linked to produce an executable.
- When the executable is run, it analyzes its input for occurrences of text matching the regular expressions for each rule.
- Whenever it finds a match, it executes the corresponding C code.

WC in Flex

```
%{  
int chars = 0;  
int words = 0;  
int lines = 0;  
%}  
%%  
[a-zA-Z]+ { words++; chars += strlen(yytext); }  
\n { chars++; lines++; }  
. { chars++; }  
\t { }  
%%  
main(int argc, char **argv)  
{  
  yylex();  
  printf("%8d%8d%8d\n", lines, words, chars);  
}
```

Compiling and running

run this command from windows CMD and generate lex.yy.c successfully

- flex filename.l or flex filename.lex

Then, lex.yy.c should compile and link with the "-lfl" library to produce an executable a.out.

- cc lex.yy.c -lfl
- ./a.out

```
#include<ctype.h>
#include<stdio.h>
typedef enum {PLUS,TIMES,MINUS,DIVIDE,OR,NUMBER,NEWLINE,MYSTERIOUSCHARACTER}TokenType;
int numValue;
```

```
int currChar;
TokenType getToken()
{
    while(currChar==' ')
        currChar=getChar();
    if(isdigit(currChar))
    {
        numValue=0;
        while(isdigit(currChar))
        {
            numValue=10*numValue +currChar -'0';
            currChar=getchar();
        }
        return NUMBER;
    }
    else
    {
        switch(currChar)
        {
            case '+' : return PLUS;break;
            case '-' : return MINUS;break;
            case '*' : return TIMES;break;
            case '/' : return DIVIDE;break;
            case '|' : return OR;break;
            case '\n' : return NEWLINE;break;
            default: return MYSTERIOUSCHARACTER;
        }
    }
}
```

main()

{

TokenType token;

currChar = getchar();

do

{

token=getToken();

switch(token)

{

case PLUS: printf("PLUS\n");break;

case MINUS: printf("MINUS\n");break;

case TIMES: printf("TIMES\n");break;

case DIVIDE: printf("DIVIDE\n");break;

case NEWLINE: printf("NEWLINE\n");break;

case OR: printf("OR\n");break;

case NUMBER: printf("NUMBER=%d\n",numValue);break;

case MYSTERIOUSCHARACTER:

printf("MYSTERIOUSCHARACTER=%c\n",currChar);

}

}while (token != NEWLINE);

return 0;

}

Flex scanner

```
TT_TIMES  
TT_PLUS  
TT_LPAREN  
TT_RPAREN  
TT_NUMBER: 42  
TT_ERROR: #  
TT_NUMBER: 345  
TT_EOL
```

- Given the input line:
 * + () 42 # 345
- The program produces the following output
- First, the code contains a main function to test drive the scanner (which is represented by the getToken function).
- Given a line of input, the main function simply calls getToken repeatedly and prints each token as it is recognized;
- A number is also printed with its numeric value and
- An error is printed with the offending character.

CFGs & BNFs

- (1) *sentence* \rightarrow *noun-phrase verb-phrase* .
- (2) *noun-phrase* \rightarrow *article noun*
- (3) *article* \rightarrow a | the
- (4) *noun* \rightarrow girl | dog
- (5) *verb-phrase* \rightarrow *verb noun-phrase*
- (6) *verb* \rightarrow sees | pets

A grammar for simple english sentences

CFGs & BNFs

- Sometimes a metasymbol is also an actual symbol in a language. In that case the symbol can be surrounded by quotation marks to distinguish it from the metasymbol.
- For example, rule 1 has a period in it. While a period is not part of any metasymbol described, it can easily be mistaken for one. Hence, it might be better to write the rule as follows:
 - *sentence* \rightarrow *noun-phrase verb-phrase* ‘.’

CFGs & BNFs

- If the quotation marks also become metasymbols.
- Some notations also rely on metasymbols (such as angle brackets) rather than italics or fonts to distinguish phrases from tokens, which can also be useful in situations where formatting is not available (such as in text files or handwritten text).
- In that case, the arrow is also often replaced by a metasymbol that is also pure text (such as two colons and an equal sign).
- $\langle \text{sentence} \rangle ::= \langle \text{noun-phrase} \rangle \langle \text{verb-phrase} \rangle \text{“.”}$

CFGs & BNFs

sentence \Rightarrow noun-phrase verb-phrase . (rule 1)
 \Rightarrow article noun verb-phrase . (rule 2)
 \Rightarrow the noun verb-phrase . (rule 3)
 \Rightarrow the girl verb-phrase . (rule 4)
 \Rightarrow the girl verb noun-phrase . (rule 5)
 \Rightarrow the girl sees noun-phrase . (rule 6)
 \Rightarrow the girl sees article noun . (rule 2)
 \Rightarrow the girl sees a noun . (rule 3)
 \Rightarrow the girl sees a dog . (rule 4)

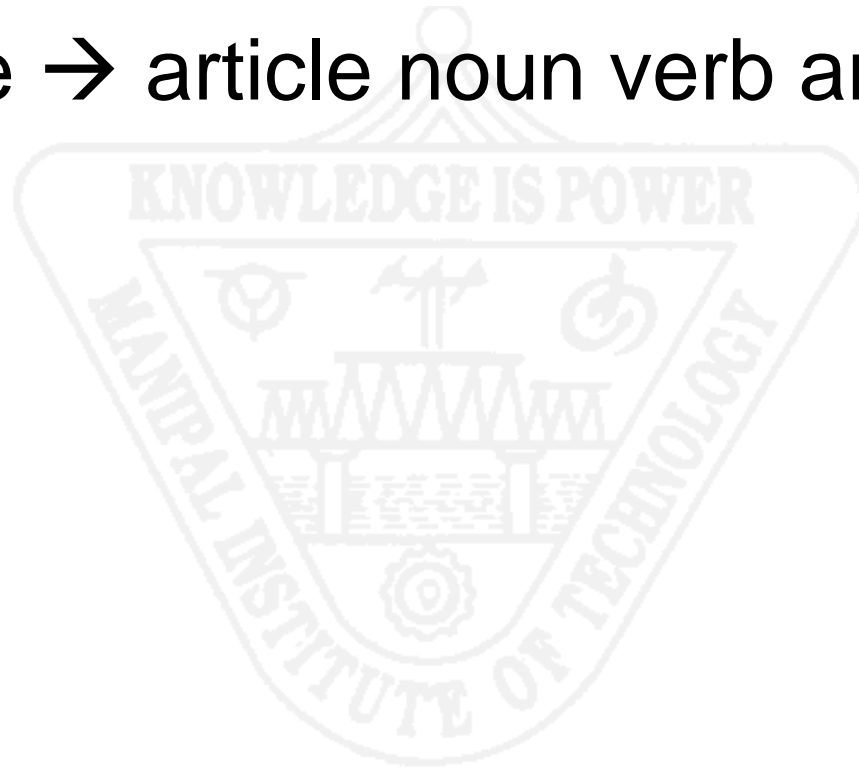
A derivation using the grammar given.

Rewritten Grammar

- Sentence \rightarrow CapitalNounPhrase VP .
- CapitalNounPhrase \rightarrow CA noun
- CA \rightarrow A | The
- noun \rightarrow girl | dog
- VP \rightarrow verb nounphrase
- nounphrase \rightarrow article noun
- article \rightarrow a | the
- verb \rightarrow sees | pets

Number of sentences

- Sentence → article noun verb article noun .



CFG and BNF

- Left-hand side (before the \rightarrow) are called nonterminals or structure names.
- Right-hand sides (after the \rightarrow) are strings of tokens and nonterminals, sometimes called symbols. (Metasymbols with special meanings can also sometimes appear.)
- Tokens are sometimes called terminals.
- Grammar rules themselves are sometimes called productions, since they "produce" the language.
- Metasymbols are the arrow \rightarrow ("consists of") and the vertical bar $|$ (choice).
- One nonterminal is singled out as the start symbol: it stands for a complete unit in the language (sentence or program).

Legal sentence of grammar

sentence \Rightarrow *noun-phrase verb-phrase* . (rule 1)
 \Rightarrow *article noun verb-phrase* . (rule 2)
 \Rightarrow the *noun verb-phrase* . (rule 3)
 \Rightarrow the girl *verb-phrase* . (rule 4)
 \Rightarrow the girl *verb noun-phrase* . (rule 5)
 \Rightarrow the girl sees *noun-phrase* . (rule 6)
 \Rightarrow the girl sees *article noun* . (rule 2)
 \Rightarrow the girl sees a *noun* . (rule 3)
 \Rightarrow the girl sees a dog . (rule 4)

Legality!!!

- “the dog pets the girl.” is a legal sentence as well.



Question

- Write syntax of a language such that there will be no re-declaration of variable. If finitely many identifiers are allowed.

solution

- StmtSeq \rightarrow DeclStat; OtherStat|OtherStat;
- OtherStat \rightarrow OtherStat;OtherStat| ϵ
- DeclStat \rightarrow Type Identifier
- Type \rightarrow int
- Identifier \rightarrow a

Language defined by the CFG

- The language defined by the CFG is the set of all strings of terminals for which there exists a derivation beginning with the start symbol and ending with string of terminals.

BNF form

- By expressing the syntax of any programming language in BNF form makes it easy to write translator for the language, since the parsing stage can be automated.

Use of CFG in programming language

$expr \rightarrow expr + expr \mid expr * expr$
 $\mid (expr) \mid number$

$number \rightarrow number digit \mid digit$

$digit \rightarrow 0 \mid 1 \mid 2 \mid 3 \mid 4 \mid 5 \mid 6 \mid 7 \mid 8 \mid 9$

Derivation

- Expr \rightarrow number
 - \rightarrow number digit
 - \rightarrow digit digit
 - \rightarrow 2 digit
 - \rightarrow 23
- Expr \rightarrow number
 - \rightarrow number digit
 - \rightarrow number digit digit
 - \rightarrow digit digit digit
 - \rightarrow 2 digit digit
 - \rightarrow 23 digit
 - \rightarrow 234

Parse and abstract syntax trees

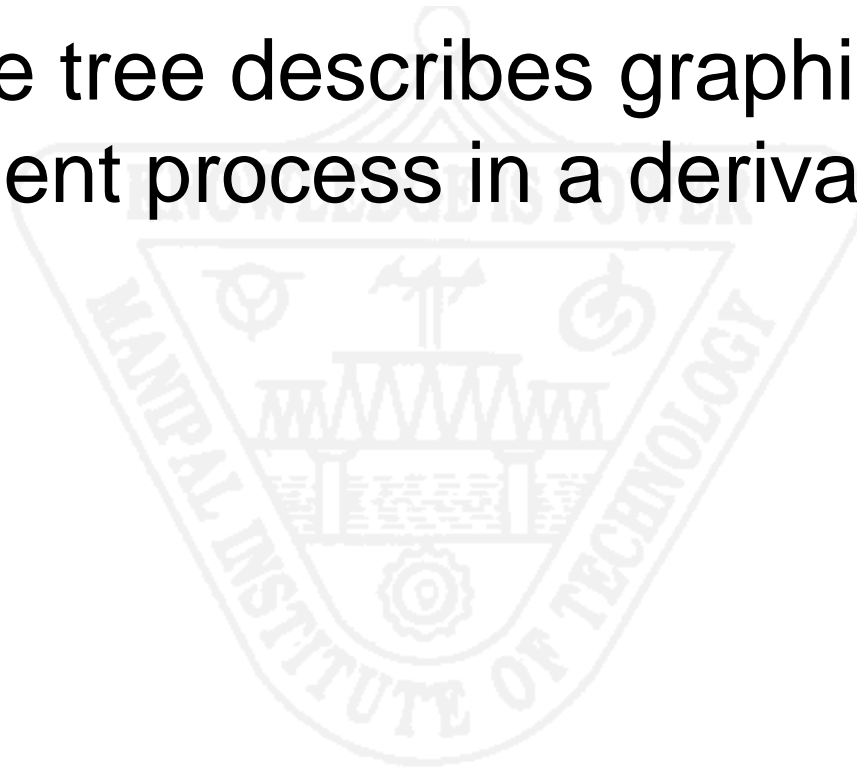
- Syntax establishes structure not meaning. But the meaning of a sentence (program) is related to its syntax.

Syntax directed semantics

- The process of attaching the semantics of a construct to its syntactic structure is called syntax directed semantics.

Parse Tree

- The parse tree describes graphically the replacement process in a derivation.



Parse Tree

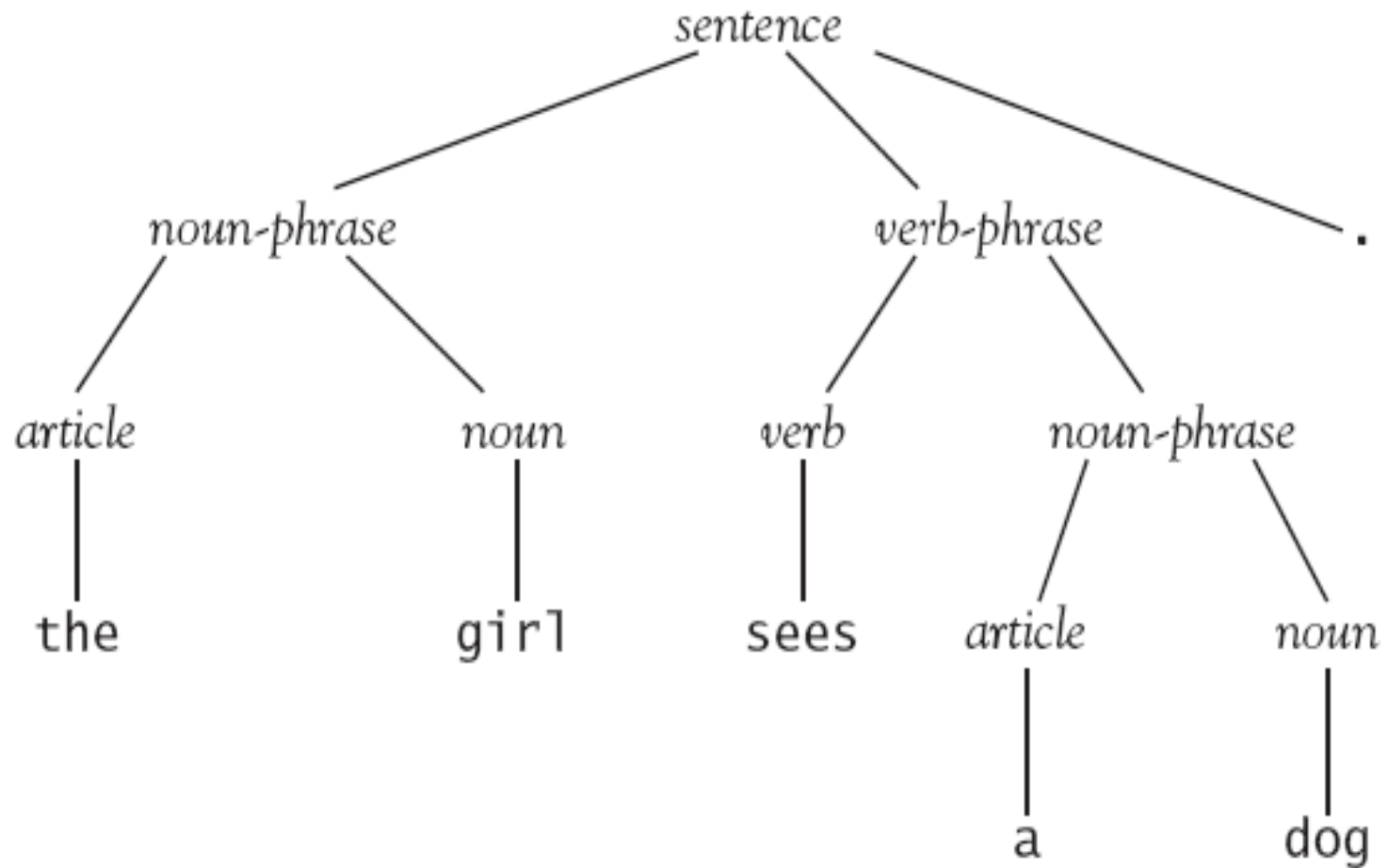
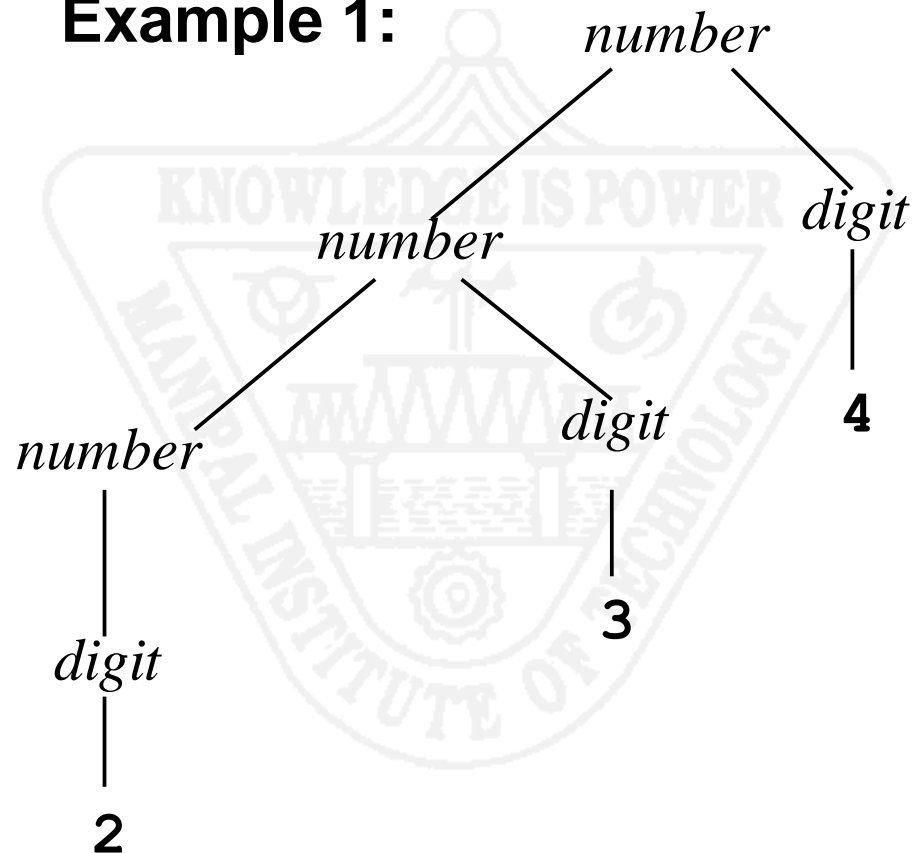


Figure 6.7: Parse tree for the sentence "the girl sees a dog."

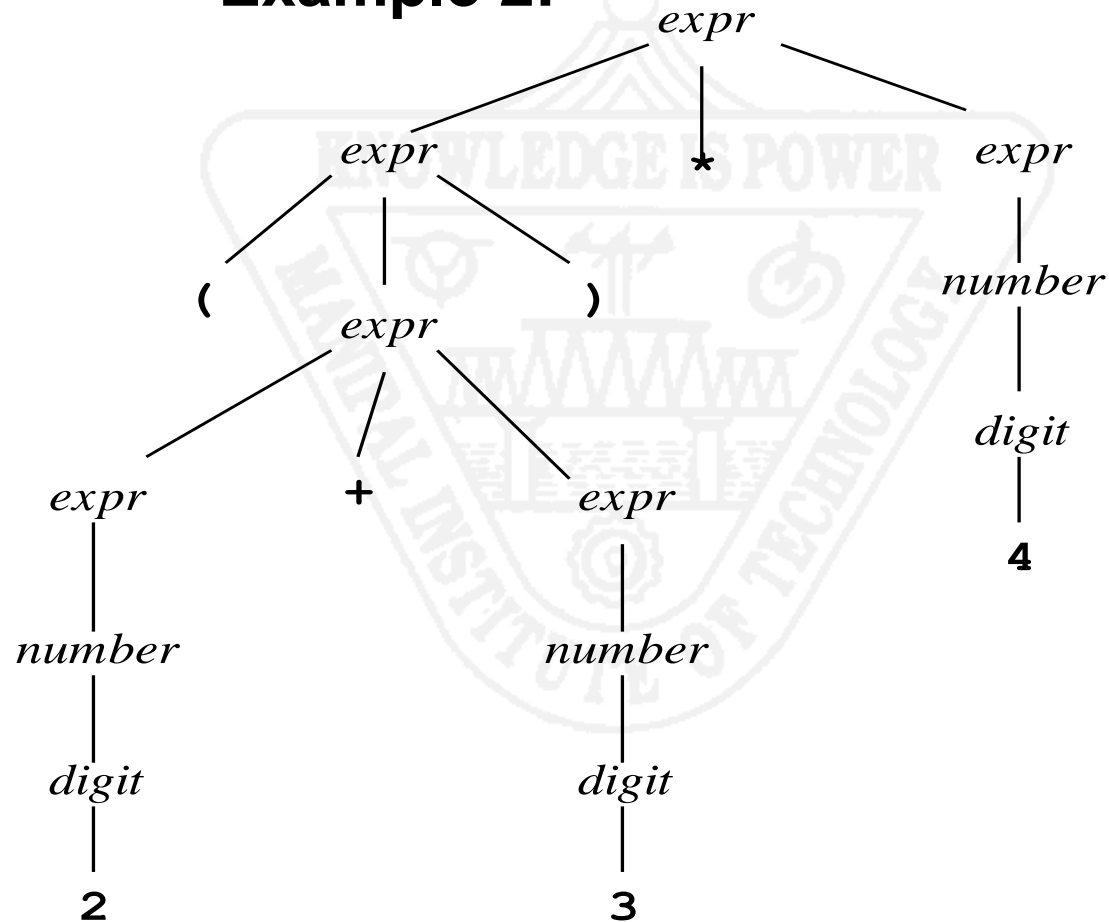
Parse Tree

Example 1:



Parse Tree

Example 2:



Parse Tree

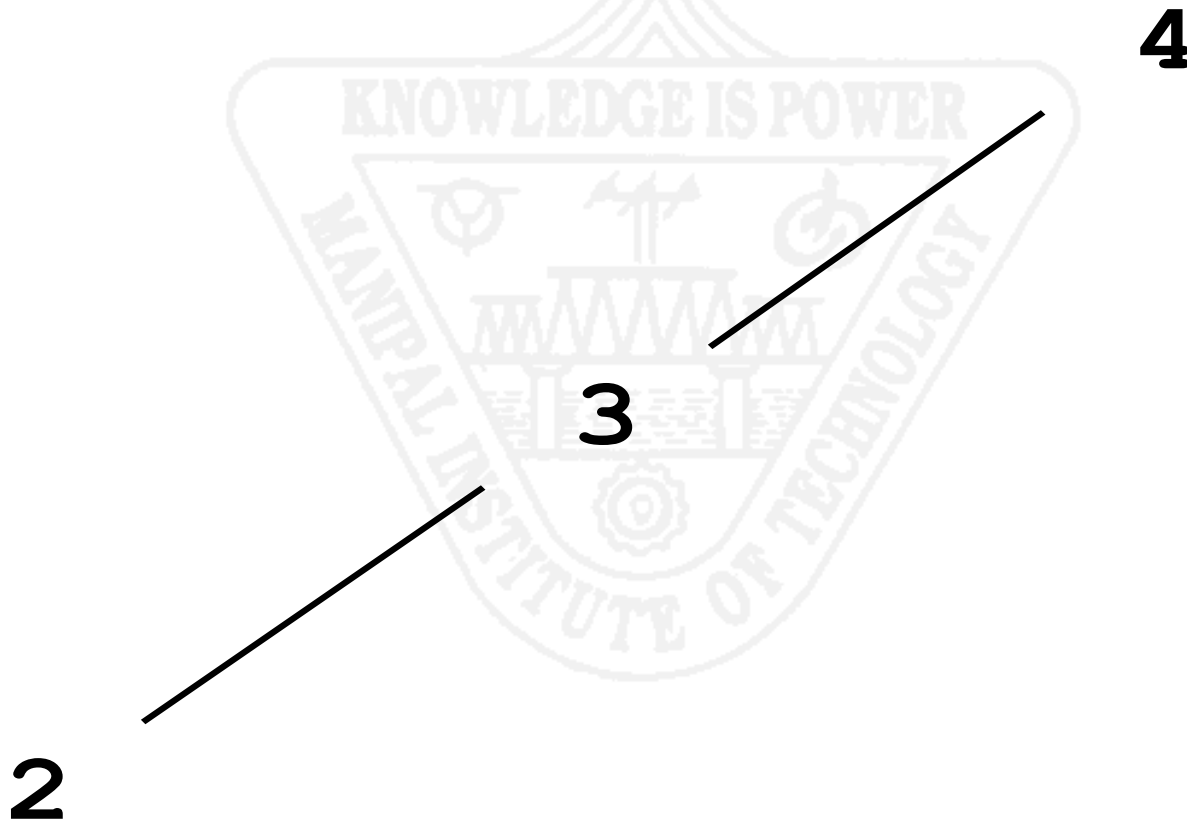
- A parse tree is labeled by non terminals at interior nodes and terminal at leaves.
- The structure of the parse tree is completely specified by the grammar rules of the language and derivation of a particular sequence of terminals.
- The grammar rule specifies the structure of each interior node and derivation specifies which interior nodes are constructed.

Parse Tree

- All the terminals and non terminals in a derivation are included in the parse tree. But not all the terminals and non terminals may be necessary to determine completely the syntactic structure of an expression or a sentence.

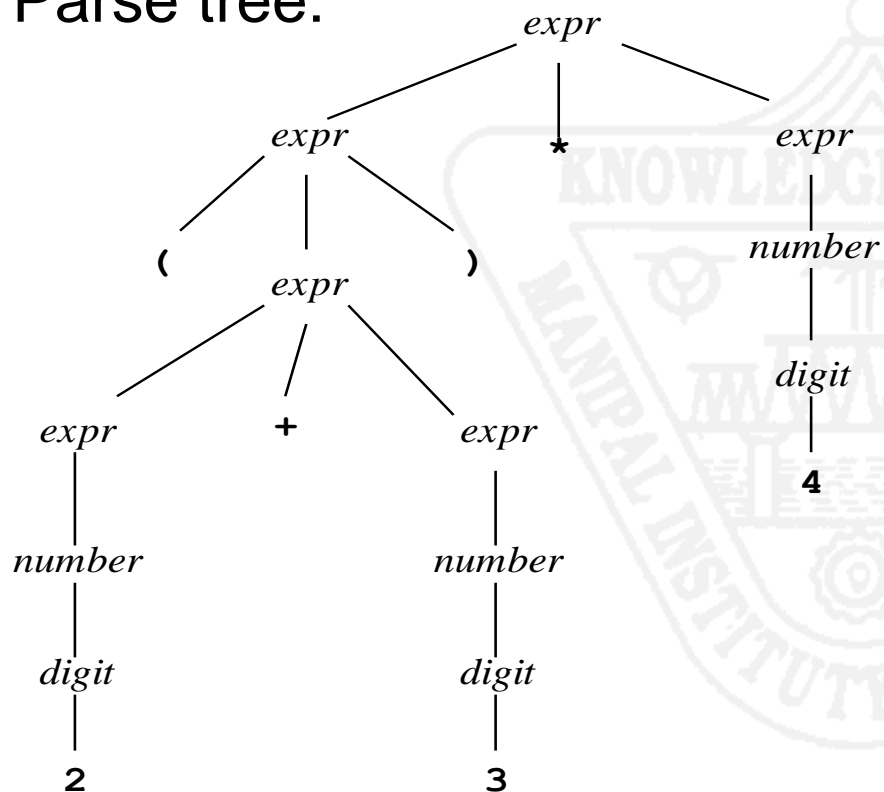
Abstract Syntax Trees

Example 1:



Abstract Syntax Trees

Parse tree:



Condensed Parse Tree:

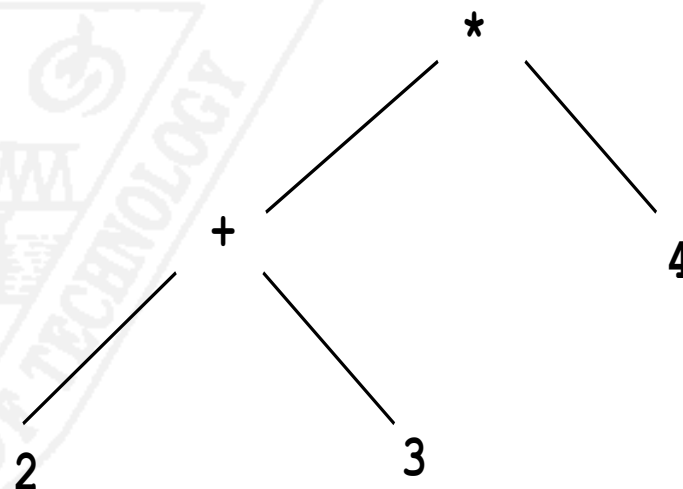


Figure shows how a parse tree for $(2 + 3) * 4$ can be condensed.

Such trees are called abstract syntax trees or just syntax trees, because they abstract the essential structure of the parse tree.

Abstract Syntax Trees

- Abstract syntax trees commonly known as Syntax trees.
- They extract the essential structure of the parse trees.

Abstract Syntax Trees

- Abstract Syntax Tree is of less importance to the programmer where as it is of great importance to the language designer and translator writer.

Ambiguity

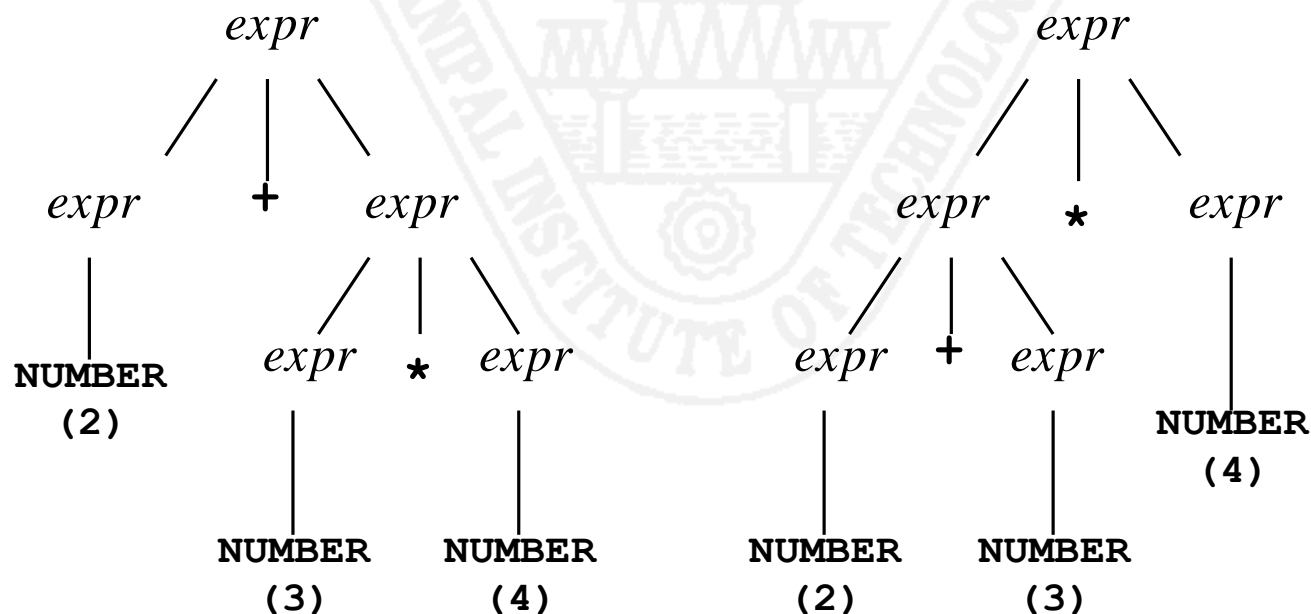
- Grammars don't always specify unique parse trees for every string in the language: a grammar is ambiguous if some string has two distinct parse (or abstract syntax) trees (not just two distinct derivations).
- Ambiguity is usually bad and must be removed.
- Semantics help in determining which parse tree is correct.
- Often the grammar can be rewritten to make the correct choice.

Example of Ambiguity

- Grammar:

$$expr \rightarrow expr + expr \mid expr * expr$$

$$\mid (expr) \mid \text{NUMBER}$$
- Expression: $2 + 3 * 4$
- Parse trees:

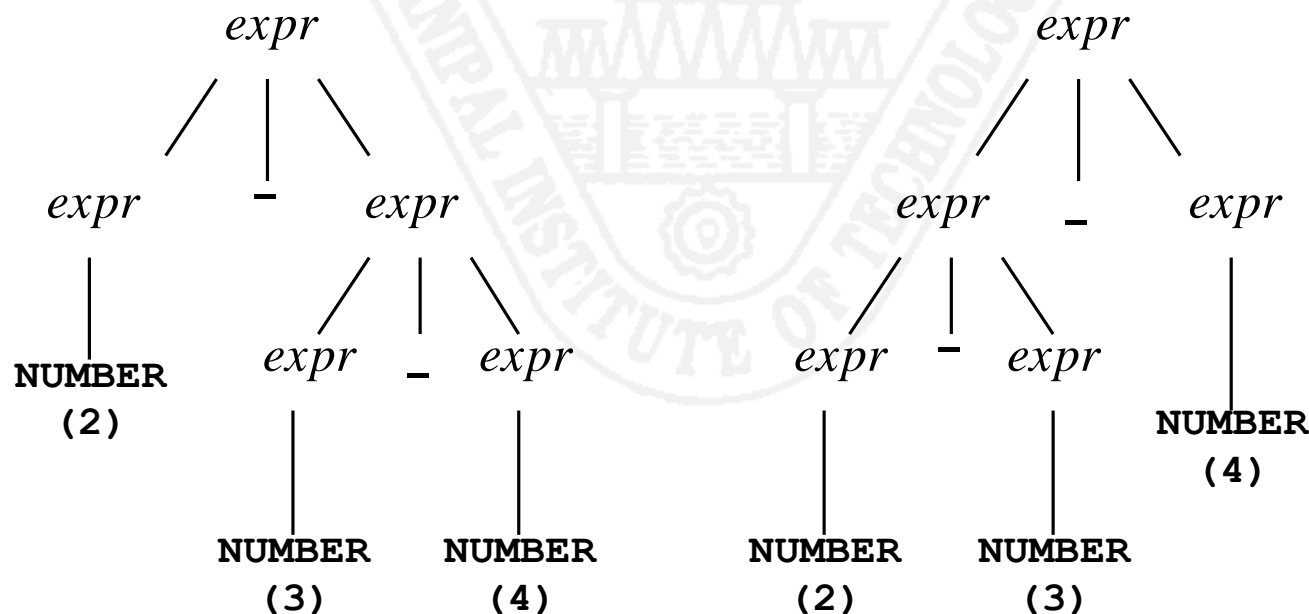


Example of Ambiguity

- Grammar (with subtraction):

$$expr \rightarrow expr + expr \mid expr - expr$$

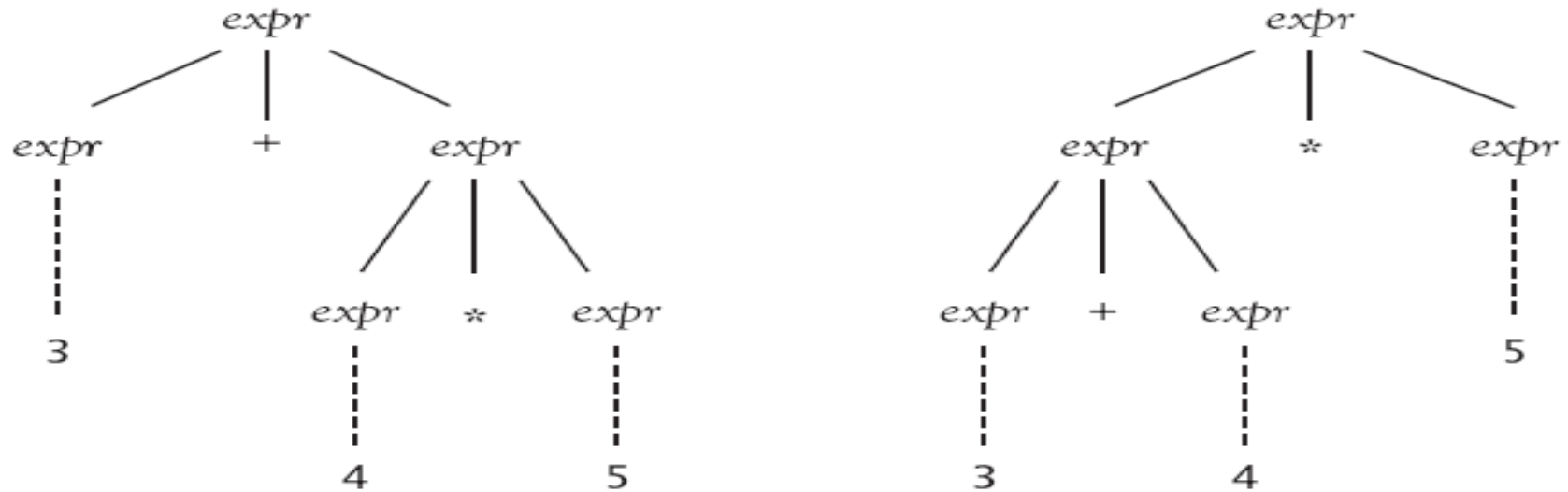
$$\mid (expr) \mid \text{NUMBER}$$
- Expression: 2 - 3 - 4
- Parse trees:



Draw Parse Tree Using Grammar3

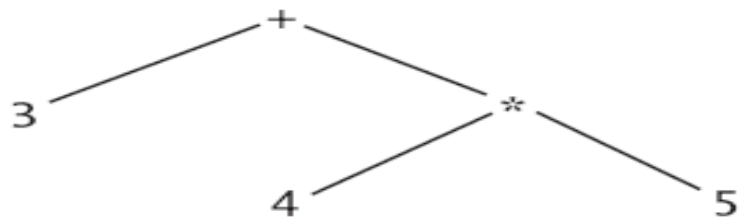
- $((2))$
- $3+4*5+6*7$
- $3*(4+5)*(6+7)$
- $(2+(3+(4+5)))$

Ambiguities

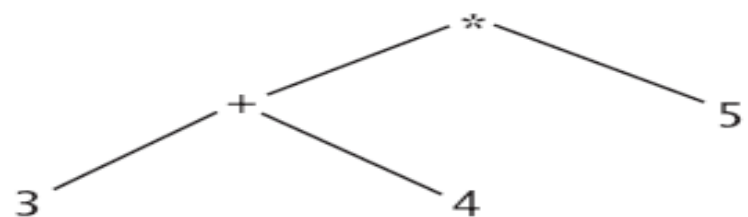


Two parse trees for $3 + 4 * 5$

Tree 1



Tree 2



Two abstract syntax trees for $3 + 4 * 5$, indicating the ambiguity of the grammar

Ambiguities

- Ambiguity can also be expressed directly in terms of derivations.
- Even though many derivations may in general correspond to the same parse tree, certain special derivations that are constructed in a special order can only correspond to unique parse trees.
- One kind of derivation that has this property is a leftmost derivation, where the leftmost remaining nonterminal is singled out for replacement at each step.

Ambiguities

- For example, the derivations in the following figures are leftmost

sentence \Rightarrow *noun-phrase verb-phrase* . (rule 1)
 \Rightarrow *article noun verb-phrase* . (rule 2)
 \Rightarrow *the noun verb-phrase* . (rule 3)
 \Rightarrow *the girl verb-phrase* . (rule 4)
 \Rightarrow *the girl verb noun-phrase* . (rule 5)
 \Rightarrow *the girl sees noun-phrase* . (rule 6)
 \Rightarrow *the girl sees article noun* . (rule 2)
 \Rightarrow *the girl sees a noun* . (rule 3)
 \Rightarrow *the girl sees a dog* . (rule 4)

number \Rightarrow *number digit*
 \Rightarrow *number digit digit*
 \Rightarrow *digit digit digit*
 \Rightarrow *2 digit digit*
 \Rightarrow *23 digit*
 \Rightarrow *234*

and the following derivation (deriving the same string as in the above right) is not.

number \Rightarrow *number digit*
 \Rightarrow *number 4*
 \Rightarrow *number digit 4*
 \Rightarrow *number 34*
...

Resolving these ambiguities

- Each parse tree has a unique leftmost derivation, which can be constructed by a preorder traversal of the tree.
- Thus, the ambiguity of a grammar can also be tested by searching for two different leftmost derivations of the same string.
- If such leftmost derivations exist, then the grammar must be ambiguous, since each such derivation must correspond to a unique parse tree.

Resolving these ambiguities

- For example, the ambiguity of the same grammar is demonstrated by the two leftmost derivations for the string $3 + 4 * 5$ as given

Leftmost Derivation 1
(Corresponding to
Tree 1 of Figure 6.13)

$expr \Rightarrow expr + expr$
 $\Rightarrow number + expr$
 $\Rightarrow digit + expr$
 $\Rightarrow 3 + expr$
 $\Rightarrow 3 + expr * expr$
 $\Rightarrow 3 + number * expr$
 $\Rightarrow \dots$ (etc.)

Leftmost Derivation 2
(Corresponding to
Tree 2 of Figure 6.13)

$expr \Rightarrow expr * expr$
 $\Rightarrow expr + expr * expr$
 $\Rightarrow number + expr * expr$
 $\Rightarrow \dots$ (etc.)

Two leftmost derivations for $3 + 4 * 5$, indicating
the ambiguity of the grammar

Resolving these ambiguities

- Ambiguous grammars present difficulties, since no clear structure is expressed.
- To be useful, either the grammar must be revised to remove the ambiguity or a “**disambiguating rule**” must be stated to establish which structure is meant.
- Which of the two parse trees (or leftmost derivations) is the correct one for the expression $3 + 4 * 5$?
- If we take the usual meaning of the expression $3 + 4 * 5$ from mathematics, we would choose the first tree over the second, since multiplication has precedence over addition.

Resolving these ambiguities

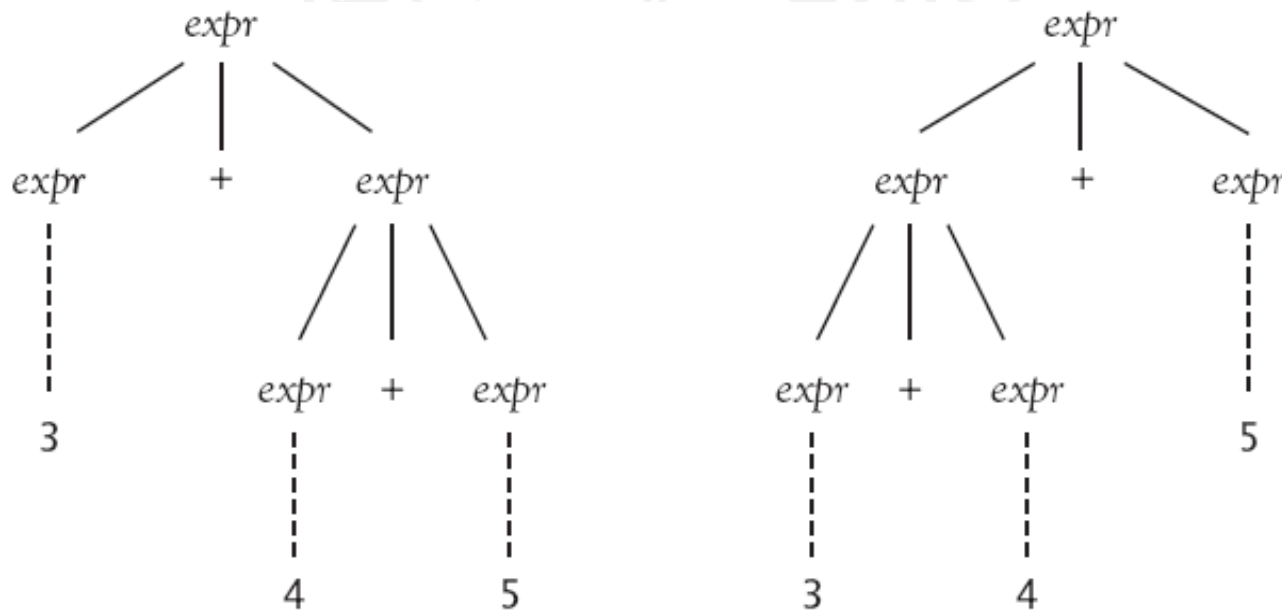
- To resolve the ambiguity express the fact that multiplication should have precedence over addition.
- The usual way to revise the grammar is to write a new grammar rule (called a “term”) that establishes a “precedence cascade” to force the matching of the “*” at a lower point in the parse tree:
- Use recursion to specify associativity, new rules (a “precedence cascade”) to specify precedence.

$expr \rightarrow expr + expr \mid term$

$term \rightarrow term * term \mid (expr) \mid number$

Resolving these ambiguities

- We have not completely solved the ambiguity problem, however, because the rule for an *expr* still allows us to parse $3 + 4 + 5$ as either $(3 + 4) + 5$ or $3 + (4 + 5)$.
- In other words, we can make addition either **right- or left-associative**, as shown in Figure



Addition as either right- or left-associative

Resolving these ambiguities

- $8 - 4 - 2 = 2$ if “-” is left-associative, but $8 - 4 - 2 = 6$ if “-” is right-associative.
- Replace the rule:
 » $expr \rightarrow expr + expr$
with either:
 » $expr \rightarrow expr + term$
or:
 » $expr \rightarrow term + expr$
- The first rule is **left-recursive**, while the second rule is **right-recursive**.

Resolving these ambiguities

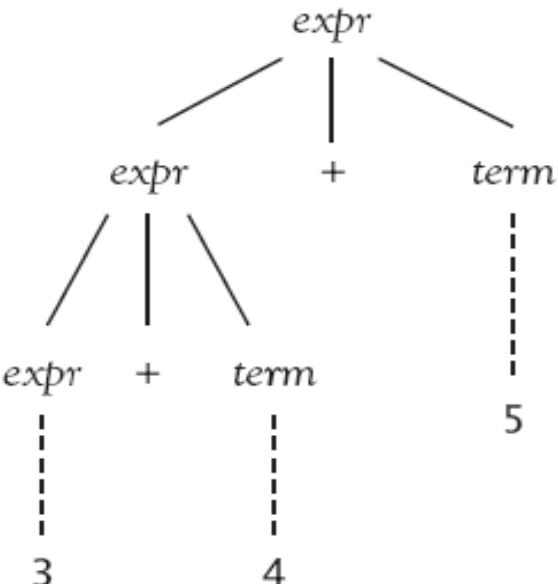
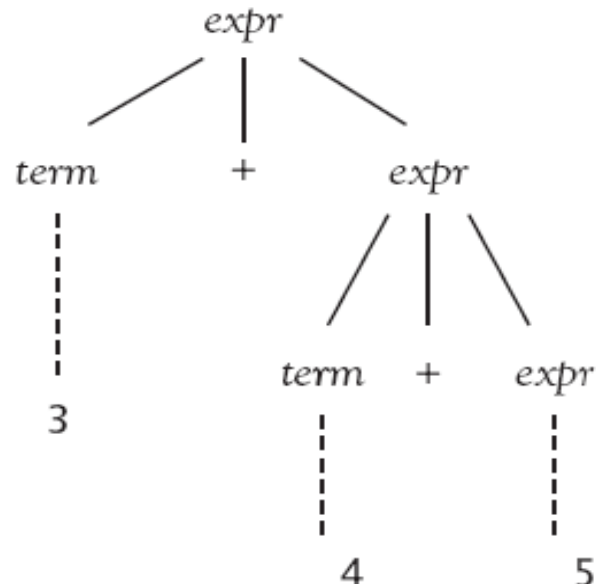
After resolving the ambiguity

$$expr \rightarrow expr + term \mid term$$
$$term \rightarrow term * factor \mid factor$$
$$factor \rightarrow (expr) \mid \text{NUMBER}$$

- Note how left recursion expresses left associativity, and higher precedence means “lower” in the cascade.

Resolving these ambiguities

- A **left-recursive rule** for an operation causes it to left-associate, similarly, a right-recursive rule causes it to right-associate, as shown in the figure below.

A left-recursive rule for an operation causes it to left-associate.	A right-recursive rule for an operation causes it to right-associate.
 <pre> graph TD E1[expr] --- E2[expr] E1 --- P1[+] E1 --- T1[term] E2 --- E3[expr] E2 --- P2[+] E2 --- T2[term] E3 --- E4[expr] E3 --- P3[+] E3 --- T3[term] E4 --- L1[3] T2 --- L2[4] T1 --- L3[5] </pre> <p>The parse tree for the left-recursive rule shows the expression $(3 + 4) + 5$. The root node <i>expr</i> expands to <i>expr</i> + <i>term</i>. The left <i>expr</i> expands to <i>expr</i> + <i>term</i>, and the leftmost <i>expr</i> expands to <i>expr</i> + <i>term</i>. The leaf nodes are 3, 4, and 5, representing the left-associative evaluation.</p>	 <pre> graph TD E1[expr] --- T1[term] E1 --- P1[+] E1 --- E2[expr] T1 --- L1[3] E2 --- T2[term] E2 --- P2[+] E2 --- E3[expr] T2 --- L2[4] E3 --- T3[term] E3 --- P3[+] E3 --- E4[expr] T3 --- L3[5] </pre> <p>The parse tree for the right-recursive rule shows the expression $3 + (4 + 5)$. The root node <i>expr</i> expands to <i>term</i> + <i>expr</i>. The right <i>expr</i> expands to <i>term</i> + <i>expr</i>, and the rightmost <i>expr</i> expands to <i>term</i> + <i>expr</i>. The leaf nodes are 3, 4, and 5, representing the right-associative evaluation.</p>

Parse trees showing results of left- and right-recursive rules

Resolving these ambiguities

- Therefore, the revised grammar that expresses both precedence and associativity is
 - $expr \rightarrow expr + term \mid term$
 - $term \rightarrow term * factor \mid factor$
 - $factor \rightarrow (expr) \mid number$
 - $number \rightarrow number digit \mid digit$
 - $digit \rightarrow 0 \mid 1 \mid 2 \mid 3 \mid 4 \mid 5 \mid 6 \mid 7 \mid 8 \mid 9$

Question

- To the same grammar add % and ^ operators. ^ is having higher precedence than * and right associative and % has equal precedence as * and left associative.

solution

- $E \rightarrow E+T|T$
- $T \rightarrow T*P|T\%P|P$
- $P \rightarrow F\wedge P|F$
- $F \rightarrow (E)|N$
- $N \rightarrow N D|D$
- $D \rightarrow 0|1|2|\dots|9$

Extended BNF Notation

We noted earlier that the grammar rule:

$$\textit{number} \rightarrow \textit{number digit} \mid \textit{digit}$$

generates a number as a sequence of digits:

$$\textit{number} \Rightarrow \textit{number digit}$$
$$\Rightarrow \textit{number digit digit}$$
$$\Rightarrow \textit{number digit digit digit}$$
$$\Rightarrow \textit{digit} \dots \textit{digit}$$
$$\dots$$
$$\Rightarrow (\text{arbitrary repetitions of digit})$$

Extended BNF Notation

Similarly, the rule:

$$\textit{expr} \rightarrow \textit{expr} + \textit{term} \mid \textit{term}$$

generates an expression as a sequence of terms separated by “+’s”:

$$\textit{expr} \Rightarrow \textit{expr} + \textit{term}$$

$$\Rightarrow \textit{expr} + \textit{term} + \textit{term}$$

$$\Rightarrow \textit{expr} + \textit{term} + \textit{term} + \textit{term}$$

. . .

$$\Rightarrow \textit{term} + . . . + \textit{term}$$

Extended BNF Notation

- This situation occurs so frequently that a special notation for such grammar rules is adopted that expresses more clearly the repetitive nature of their structures:

- $number \rightarrow digit \{digit\}$

and:

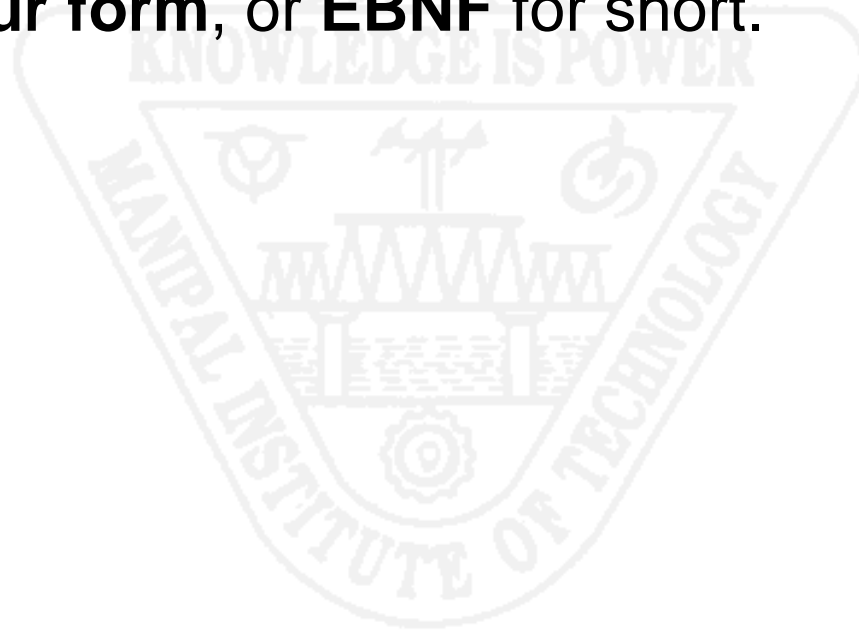
- $expr \rightarrow term \{+ term\}$

the curly brackets $\{ \}$ stand for “zero or more repetitions of.”

- Thus, the above rules express that a number is a sequence of one or more digits, and an expression is a term followed by zero or more repetitions of a $+$ and another term.

Extended BNF Notation

- In this notation, the curly brackets have become new metasymbols, and this notation is called **Extended Backus-Naur form**, or **EBNF** for short.



Extended BNF Notation

- This new notation obscures the left associativity of the + operator that is expressed by the left recursion in the original rule in BNF.
- Any operator involved in a curly bracket repetition is left-associative.
- Indeed, if an operator were right-associative, the corresponding grammar rule would be right-recursive. Right-recursive rules are usually not written using curly brackets.
- Parse trees and syntax trees cannot be written directly from the grammar. Therefore, we will always use BNF notation to write parse trees

Extended BNF Notation

- Notation for repetition and optional features.
- $\{...\}$ expresses repetition:
 $expr \rightarrow expr + term \mid term$ becomes
 $expr \rightarrow term \{ + term \}$
- $[...]$ expresses optional features:
 $if-stmt \rightarrow if (expr) stmt$
 $\quad \mid if (expr) stmt \text{ else } stmt$
becomes
 $if-stmt \rightarrow if (expr) stmt [\text{ else } stmt]$

Convert the following EBNF into BNF

Program \rightarrow int main() \{ Declarations
Statements \}

Declarations \rightarrow \{ Declaration \}

Declaration \rightarrow Type Identifier

Type \rightarrow int | char

Statements \rightarrow \{ Statement \}

Statement \rightarrow ; | Block | Assignment

Block \rightarrow \{ Statements \}

Assignment \rightarrow Identifier [Expression] = Expression;
Expression;

Expression \rightarrow Conjunction

Conjunction \rightarrow Equality { && Equality }

Equality \rightarrow Relation [Equop Relation]

Equop \rightarrow == | !=

Relation \rightarrow Addition [Relop Addition]

Relop \rightarrow < | >

Addition \rightarrow Term { AddOp Term }

AddOp \rightarrow + | -

Term \rightarrow Factor { MulOp Factor }

MulOp \rightarrow * | / | %

Factor \rightarrow Primary

Primary \rightarrow Identifier

Identifier \rightarrow Letter { Letter | Digits }

Letter \rightarrow a | b | c | d | ... | z

Solution

- Program \rightarrow int main() { Declarations Statements }
- Declarations \rightarrow Declarations Declaration | ϵ
- Declaration \rightarrow Type Identifier
- Type \rightarrow int | char
- Statements \rightarrow Statements Statement | ϵ
- Statement \rightarrow ; | Block | Assignment
- Block \rightarrow {Statements}

Solution (continued)

- Assignment \rightarrow Identifier Expression = Expression; | Identifier = Expression;
- Expression \rightarrow Conjunction
- Conjunction \rightarrow Equality | Conjunction && Equality
- Equality \rightarrow Relation | Relation Equop Relation
- Equop \rightarrow == | !=

Solution (continued)

- Relation \rightarrow Addition | Addition Relop Addition
- Relop \rightarrow $<|>$
- Addition \rightarrow Term | Addition Addop Term
- Addop \rightarrow $+ \mid -$
- Term \rightarrow Factor | Term Mulop Factor
- Mulop \rightarrow $* \mid / \mid \%$
- Factor \rightarrow Primary

Solution (continued)

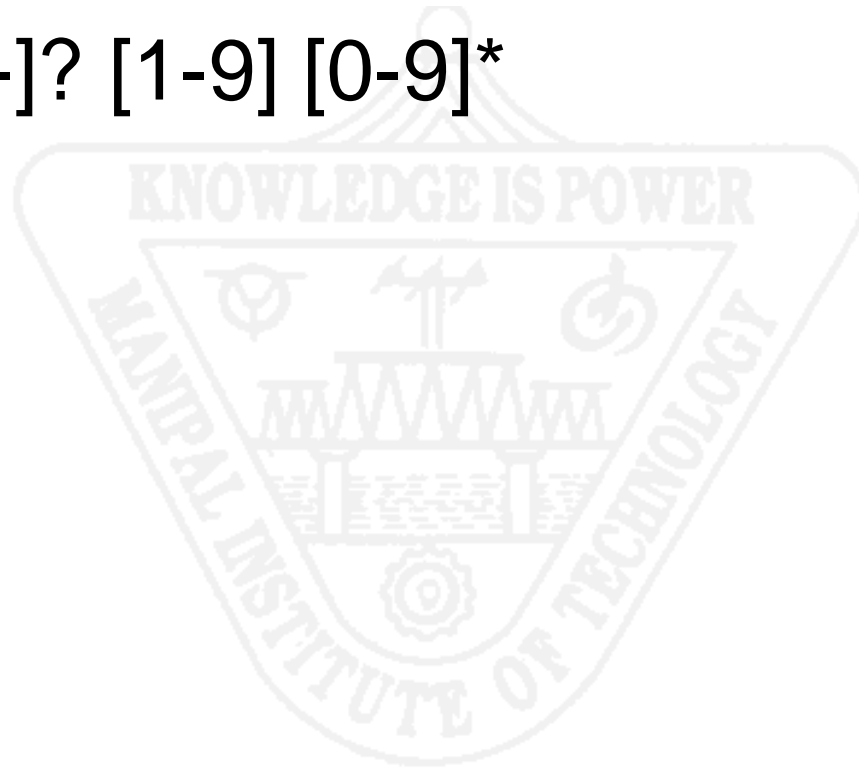
- Primary \rightarrow Identifier
- Identifier \rightarrow Letter | Identifier Letter | Identifier Digits
- Letter \rightarrow a|b|c|.....|z

Question

- Write the regular Expression for valid signed integers. Eg:- -0 is invalid where as -909 is valid.

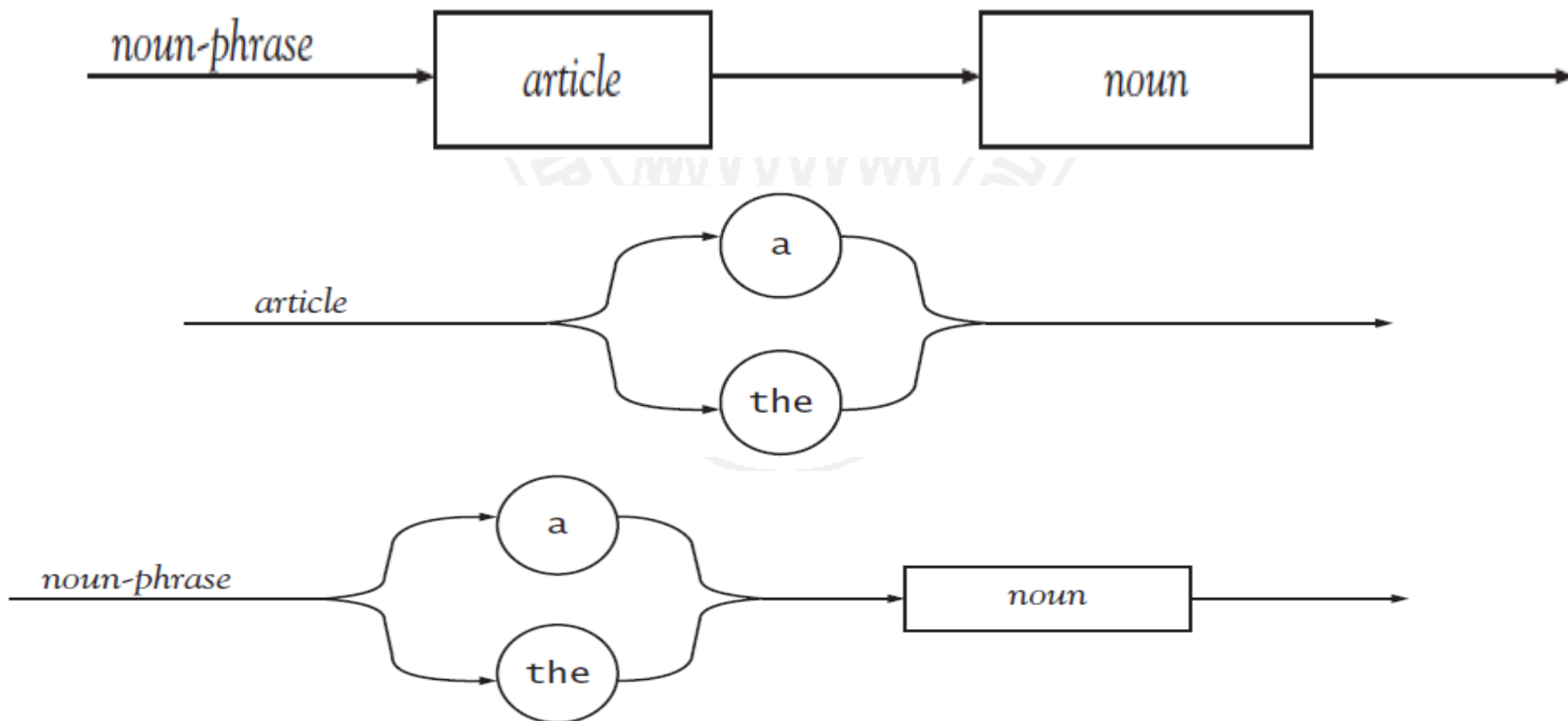
Solution

- $0 \mid [\backslash+ \mid \backslash-]? [1-9] [0-9]^*$



Syntax Diagram

- Graphical representation for a grammar rule is the **syntax diagram**, which indicates the sequence of terminals and nonterminals encountered in the right-hand side of the rule.
- Ex: Syntax diagrams for *noun-phrase* and *article*

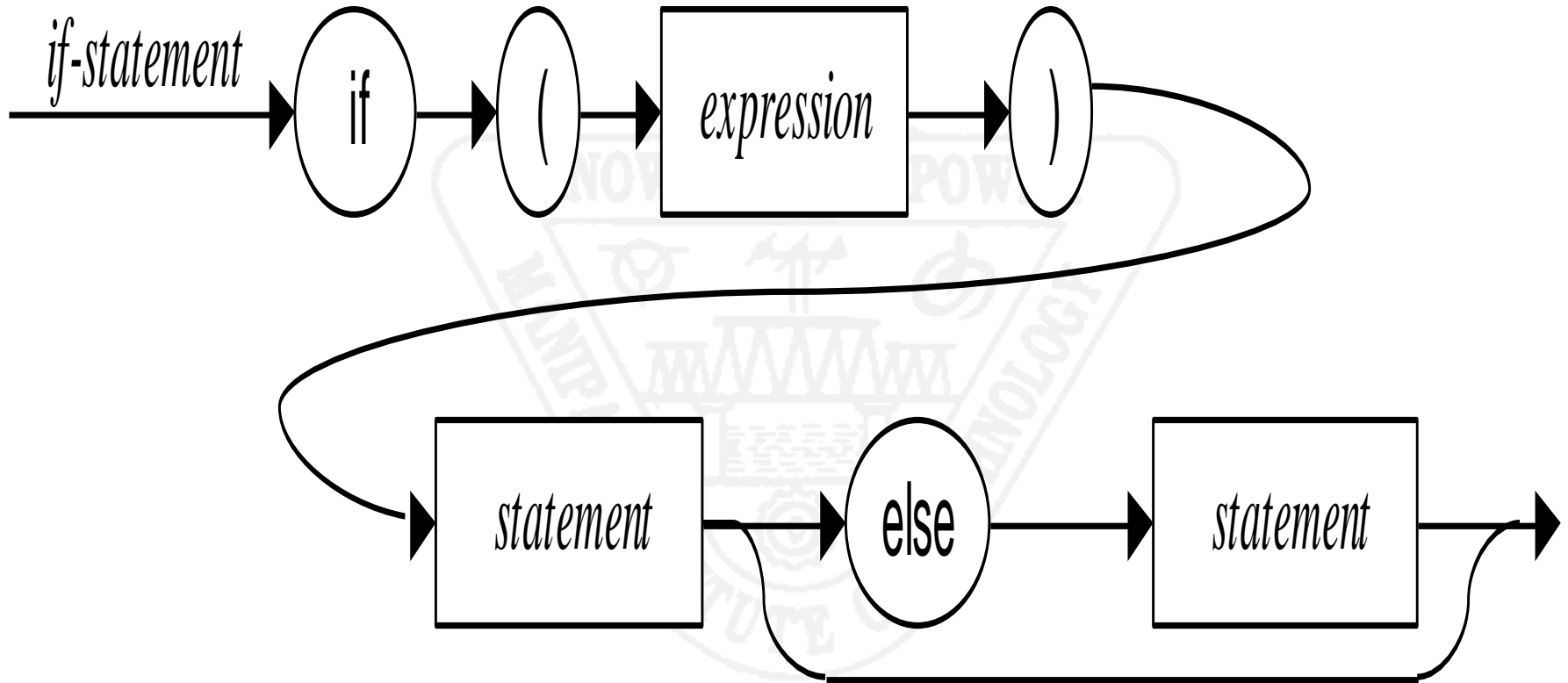


Syntax Diagram

```
expr → term { + term }  
term → factor { * factor }  
factor → ( expr ) | number  
number → digit { digit }  
digit → 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9
```

- Note the use of loops in the diagrams to express the repetition given by the curly brackets in the EBNFs.

Syntax Diagram



Parsing Techniques and Tools

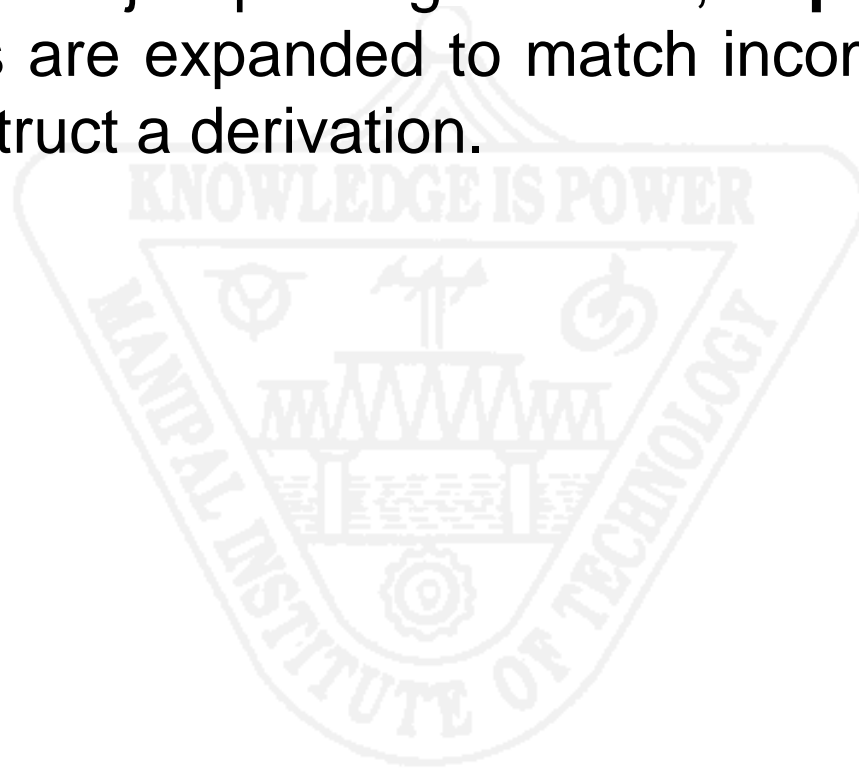
- A grammar written in BNF, EBNF, or as syntax diagrams describes the strings of tokens that are syntactically legal in a programming language.
- The grammar, thus, also implicitly describes the actions that a parser must take to parse a string of tokens correctly and to construct, either implicitly or explicitly, a derivation or parse tree for the string.

Parsing Techniques and Tools

- Given a grammar in one of the forms we have discussed, how does it correspond to the actions of a parser?
- **One method of parsing** attempts to match an input with the right-hand sides of the grammar rules.
- When a match occurs, the right-hand side is replaced by, or **reduced** to, the nonterminal on the left.
- Such parsers are **bottom-up** parsers, since they construct derivations and parse trees from the leaves to the root..
- They are sometimes also called **shift-reduce** parsers, since they shift tokens onto a stack prior to reducing strings to nonterminals.
- $2 \rightarrow D \rightarrow N \rightarrow F \rightarrow P \rightarrow T \rightarrow E$, Considering the input is 2. from the unambiguous grammar derived in slide no. 72.

Top-Down parsing

- In the other major parsing method, **top-down parsing**, nonterminals are expanded to match incoming tokens and directly construct a derivation.



Recursive-Descent Parsing

```
void sentence() {
    nounPhrase();
    verbPhrase();
}

void nounPhrase() {
    article();
    noun();
}

void article() {
    if (token == "a") match("a", "a expected");
    else if (token == "the") match("the", "the expected");
    else void match(TokenType expected, char* message) {
        if (token == expected) getToken();
        else error(message);
    }
}
```

Recursive-Descent Parsing

- This type of parsing cannot be applied to left recursive grammar since it enters into an infinite recursive calls.
- Ex: $\text{expr} \rightarrow \text{expr} + \text{term} \mid \text{term}$
- There are two difficulties in the above grammar with respect to Recursive decent parsing.
- 1] the first choice in this rule would have the procedure immediately calling itself recursively, which would result in an infinite recursive loop.
- 2] there is no way to decide which of the two choices to take ($\text{expr} \rightarrow \text{expr} + \text{term}$ or $\text{expr} \rightarrow \text{term}$) until a + is seen (or not) much later in the input.
- The problem is caused by the existence of the left recursion in the first alternative of the grammar rule,
- since the right-recursive rule: $\text{expr} \rightarrow \text{term} + \text{expr} \mid \text{term}$ has only a very modest problem and can be turned into a `term()` and then testing for a "+":

```
void expr() {  
    term();  
    if (token == "+") {  
        match("+", "+ expected");  
        expr();  
    }  
}
```

Recursive-Descent Parsing

- Unfortunately, this grammar rule (and the associated parsing procedure) make + into a right-associative operator.
- How can we remove the problem caused by the left recursion, while at the same time retaining the left associative behavior of the + operation?
- The solution is implicit in the EBNF description of this same grammar rule, which expresses the recursion as a loop: $expr \rightarrow term \{ + term \}$
- This form of the grammar rule for *expr* corresponds to the recursive-descent code:

```
void expr() {  
    term();  
    while (token == "+") {  
        match("+", "+ expected");  
        term();  
        /* perform left associative operations here */  
    }  
}
```

Recursive-Descent Parsing

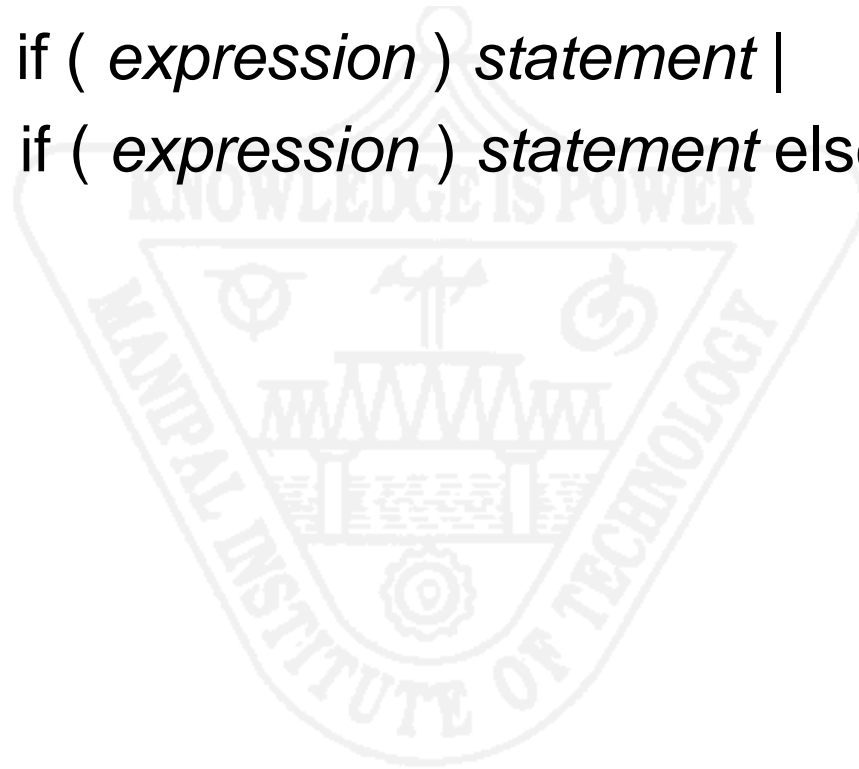
```
void match(TokenType expected, char*  
message){  
    if (token == expected) getToken();  
    else error(message);  
}
```

Left factoring

- Right-recursive rules on the other hand, as we have already noted, present no such problem in recursive-descent parsing. However, the code that we write for a right-recursive rule such as:
- $expr \rightarrow term @ expr \mid term$
- $expr \rightarrow term [@ expr]$
- The production is rewritten such that term is factored out and @expr becomes an optional entity. The process is called as left factoring.

RDP

if-statement \rightarrow *if (expression) statement |*
if (expression) statement else statement



RDP

```
void ifStatement() {  
    match("if", "if expected");  
    match("(", "(" expected");  
    expression();  
    match(")", ")" expected");  
    statement();  
    if (token == "else") {  
        match("else", "else expected");  
        statement();  
    }  
}
```


RDP

- To make it easy to print the computed value, and also to make sure only a single expression is entered in the input line, a new grammar rule is added to the grammar
- *command* \rightarrow *expr* '\n'
- The above grammar also makes new line as end of input.

Program

- (1) `#include <ctype.h>`
- (2) `#include <stdlib.h>`
- (3) `#include <stdio.h>`
- (4) `int token; /* holds the current input character for the parse */`

Program

- (5) /* declarations to allow arbitrary recursion */
- (6) void command();
- (7) int expr();
- (8) int term();
- (9) int factor();
- (10) int number();
- (11) int digit();

Program

- (12) void error(char* message){
- (13) printf("parse error: %s\n", message);
- (14) exit(1);
- (15) }

Program

- (16) void getToken(){
- (17) /* tokens are characters */
- (18) token = getchar();
- (19) }

Program

- (20) void match(char c, char* message){
- (21) if (token == c) getToken();
- (22) else error(message);
- (23) }

Program

- (24) void command(){
- (25) /* command -> expr '\n' */
- (26) int result = expr();
- (27) if (token == '\n') /* end the parse and print the result */
- (28) printf("The result is: %d\n",result);
- (29) else error("tokens after end of expression");
- (30) }

Program

- (31) int expr(){
- (32) /* expr -> term { '+' term } */
- (33) int result = term();
- (34) while (token == '+'){
- (35) match('+', "+ expected");
- (36) result += term();
- (37) }
- (38) return result;
- (39) }

Program

- (40) int term(){
- (41) /* term -> factor { '*' factor } */
- (42) int result = factor();
- (43) while (token == '*'){
- (44) match('*', "* expected");
- (45) result *= factor();
- (46) }
- (47) return result;
- (48) }

Program

- (49) int factor(){
- (50) /* factor -> '(' expr ')' | number */
- (51) int result;
- (52) if (token == '('){
- (53) match('(', "(expected");
- (54) result = expr();
- (55) match(')', ") expected");
- (56) }
- (57) else
- (58) result = number();
- (59) return result;
- (60) }

Program

- (61) int number(){
- (62) /* number -> digit { digit } */
- (63) int result = digit();
- (64) while (isdigit(token))
- (65) /* the value of a number with a new trailing digit
- (66) is its previous value shifted by a decimal place
- (67) plus the value of the new digit
- (68) */
- (69) result = 10 * result + digit();
- (70) return result;
- (71) }

Program

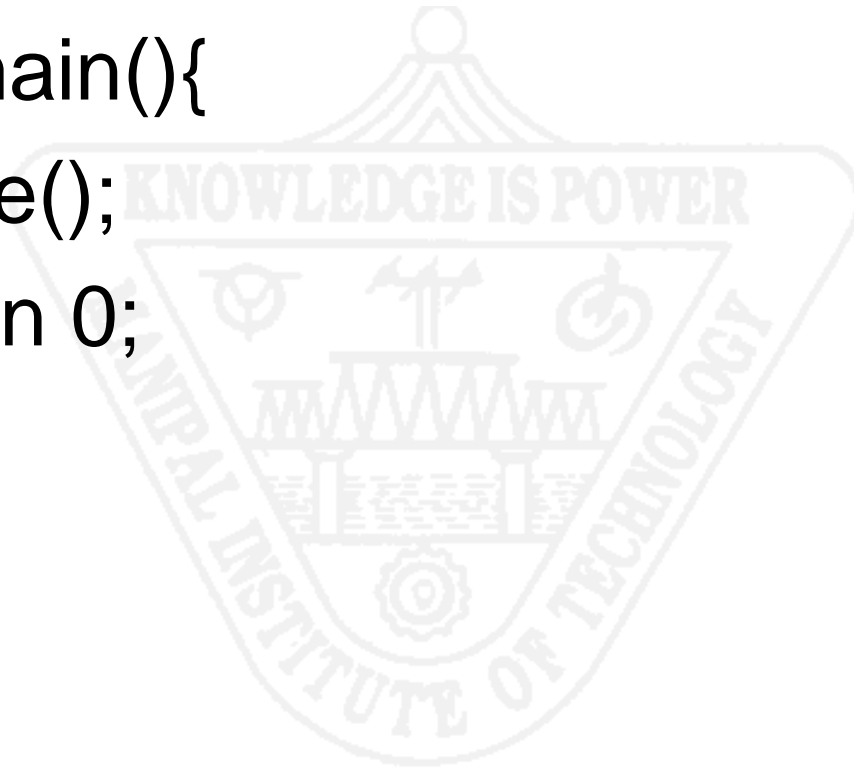
- (72) int digit(){
- (73) /* digit -> '0' | '1' | '2' | '3' | '4'
- (74) | '5' | '6' | '7' | '8' | '9' */
- (75) int result;
- (76) if (isdigit(token)){
- (77) /* the numeric value of a digit character
- (78) is the difference between its ascii value and the
- (79) ascii value of the character '0'
- (80) */
- (81) result = token - '0';
- (82) match(token, "(expected");
- (83) }
- (84) else
- (85) error("digit expected");
- (86) return result;
- (87) }

Program

- (88) void parse(){
- (89) getToken(); /* get the first token */
- (90) command(); /* call the parsing procedure for the start symbol */
- (91) }

Program

- (92) int main(){
- (93) parse();
- (94) return 0;
- (95) }



Predictive Parser

- In this method for converting a grammar into a parser, the resulting parser bases its actions only on the next available token in the input stream (stored in the token variable in the code).
- This use of a single token to direct a parse is called **single-symbol lookahead**.
- A parser that commits itself to a particular action based only on this lookahead is called a **predictive parser**.
- Predictive parsers require that the grammar to be parsed satisfies certain conditions so that this decision-making process will work.

Conditions to be satisfied to write a RDP

- Given $A \rightarrow \alpha_1 \mid \alpha_2 \mid \dots \mid \alpha_n$
 - $\text{First}(\alpha_i) \cap \text{First}(\alpha_j) = \emptyset$ for all $i \neq j$
 - Given $A \rightarrow \beta [\alpha] \sigma$
 $\text{First}(\alpha) \cap \text{Follow}(\alpha) = \emptyset$

First

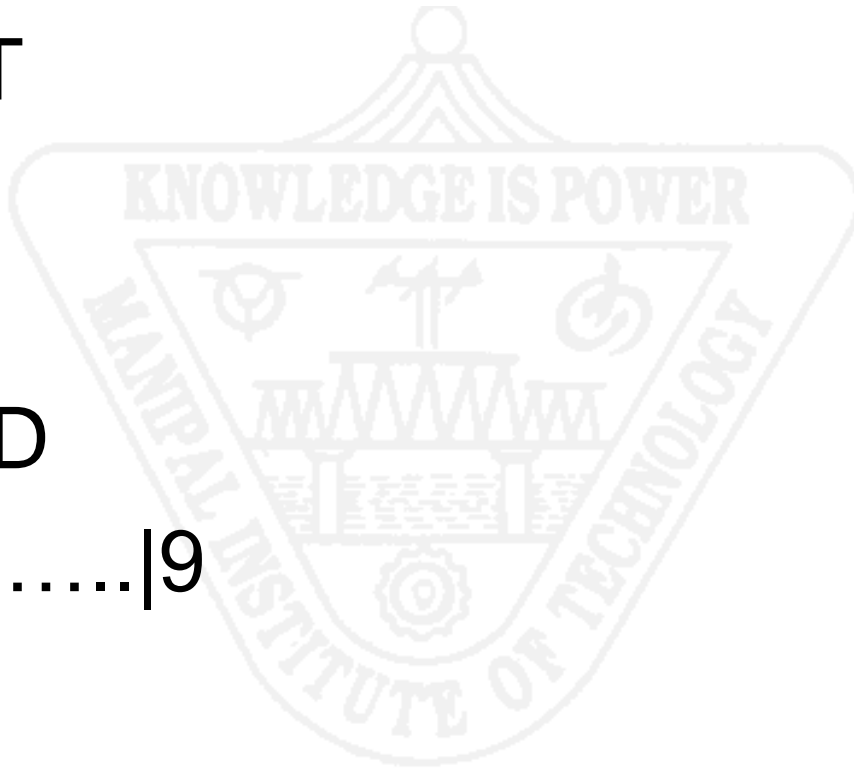
- **Rules for First Sets**
- If X is a terminal **then** $\text{First}(X)$ is just X !
- If there is a Production $X \rightarrow \epsilon$ **then** add ϵ to $\text{first}(X)$
- If there is a Production $X \rightarrow Y_1Y_2..Y_k$ **then** add $\text{first}(Y_1Y_2..Y_k)$ to $\text{first}(X)$
- $\text{First}(Y_1Y_2..Y_k)$ is **either**
 - $\text{First}(Y_1)$ (if $\text{First}(Y_1)$ doesn't contain ϵ)
 - **OR** (if $\text{First}(Y_1)$ does contain ϵ) then $\text{First}(Y_1Y_2..Y_k)$ is everything in $\text{First}(Y_1)$ <except for ϵ > as well as everything in $\text{First}(Y_2..Y_k)$
 - If $\text{First}(Y_1) \text{ First}(Y_2).. \text{First}(Y_k)$ all contain ϵ **then** add ϵ to $\text{First}(Y_1Y_2..Y_k)$ as well.

Follow

- **Rules for Follow Sets**
- First put \$ (the end of input marker) in Follow(S) (S is the start symbol)
- If there is a production $A \rightarrow \beta B \alpha$, (where β and α can be a whole string) **then** everything in FIRST(α) except for ϵ is placed in FOLLOW(B).
- If there is a production $A \rightarrow \beta B$, **then** everything in FOLLOW(A) is in FOLLOW(B)
- If there is a production $A \rightarrow \beta B \alpha$, where FIRST(α) contains ϵ , **then** everything in FOLLOW(A) is in FOLLOW(B)

Grammar

- $E \rightarrow E + T \mid T$
- $T \rightarrow T * F \mid F$
- $F \rightarrow (E) \mid N$
- $N \rightarrow ND \mid D$
- $D \rightarrow 0 \mid 1 \mid 2 \mid \dots \mid 9$



First

- $\text{First}(D) = \{0, 1, \dots, 9\}$
- $\text{First}(N) = \{0, 1, \dots, 9\}$
- $\text{First}(F) = \{(\text{'}, 0, 1, \dots, 9)\}$
- $\text{First}(T) = \{(\text{'}, 0, 1, \dots, 9)\}$
- $\text{First}(E) = \{(\text{'}, 0, 1, \dots, 9)\}$

Follow

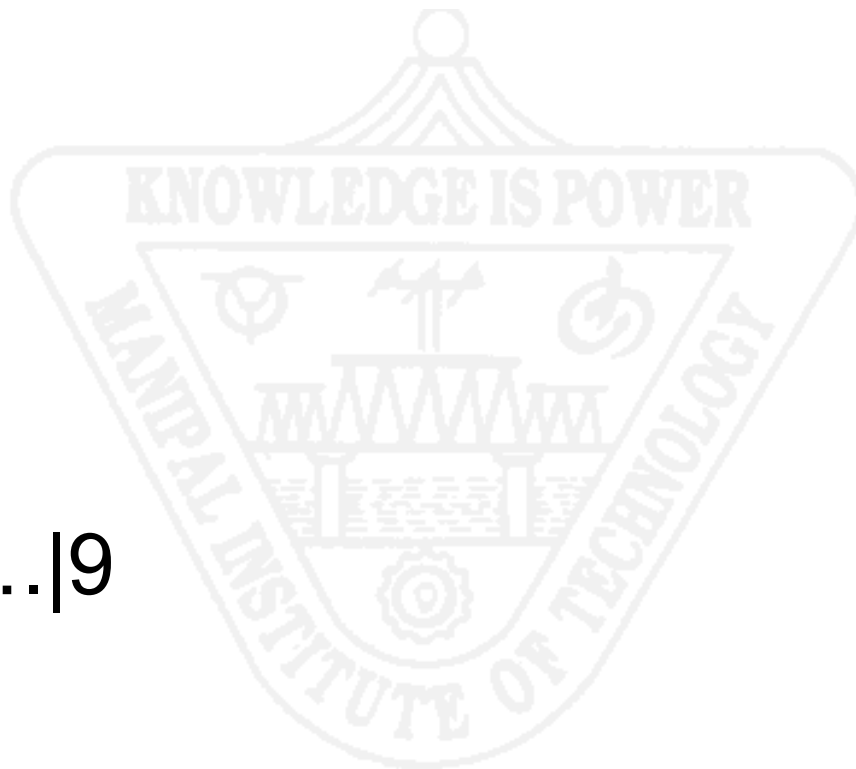
- $\text{Follow}(E) = \{+,), \$\}$
- $\text{Follow}(T) = \{+, *,), \$\}$
- $\text{Follow}(F) = \{*, +,), \$\}$
- $\text{Follow}(N) = \{*, +,), \$\}$
- $\text{Follow}(D) = \{*, +,), \$\}$

Parsing

- Since $\text{First}(E+T) \cap \text{First}(T) \neq \emptyset$ this grammar cannot be parsed recursive descently.

Grammar

- $E \rightarrow T\{+T\}$
- $T \rightarrow F\{*F\}$
- $F \rightarrow (E) | N$
- $N \rightarrow D\{D\}$
- $D \rightarrow 0 | 1 | \dots | 9$



First

- $\text{First}(D) = \{0, 1, \dots, 9\}$
- $\text{First}(N) = \{0, 1, \dots, 9\}$
- $\text{First}(F) = \{(\text{,} 0, 1, \dots, 9)\}$
- $\text{First}(T) = \{(\text{,} 0, 1, \dots, 9)\}$
- $\text{First}(E) = \{(\text{,} 0, 1, \dots, 9)\}$

Follow

- $\text{Follow}(E) = \{), \$\}$
- $\text{Follow}(T) = \{+,), \$\}$
- $\text{Follow}(F) = \{*, +,), \$\}$
- $\text{Follow}(N) = \{*, +,), \$, 0, 1, ---, 9\}$
- $\text{Follow}(D) = \{*, +,), \$, 0, 1, ---, 9\}$

Grammar

- $\text{if-stmt} \rightarrow \text{if}(E)\text{Statement}[\text{else Statement}]$
- $\text{First}(\text{else Statement}) = \{\text{else}\}$
- $\text{Follow}(\text{else Statement}) = \{;\}$
- Thus it is possible to write the Recursive descent parser for if statement.

The format of YACC specification

- `%{ /* code to insert at the beginning of the parser */ %}`
- `/* other YACC definitions if necessary */`
- `%%`
- `/* grammar and associated actions */`
- `%%`
- `/* auxiliary procedures */`

STEPS TO EXECUTE YACC PROGRAM

- `bison -d filename.y`
- `gcc filename.tab.c`
- `gcc -o filename filename.tab.o`
- `./filename`

References

Text book

- Kenneth C. Louden “Programming Languages Principles and Practice” second edition Thomson Brooks/Cole Publication.

Reference Books:

- Terrence W. Pratt, Masvin V. Zelkowitz “Programming Languages design and Implementation” Fourth Edition Pearson Education.
- Allen Tucker, Robert Noonan “Programming Languages Principles and Paradigms second edition Tata MC Graw –Hill Publication.