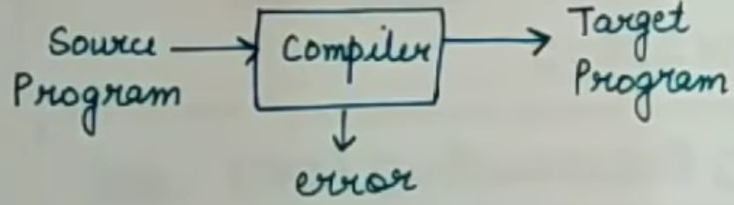
COMPILERS: An Introduction

What is a compiler ?

A compiler is a program that reads a program written in one language and translates it into an equivalent program in another language.

Input Language : Source Language
Output Language : Target Language.

A compiler also reports errors present in the source program as a part of its translation process.



Types of Compilers → Single Pass Compiler
↓
Multi-Pass Compiler

The 2 parts of the compilation process

Analysis

It takes the input source program and breaks it into parts.

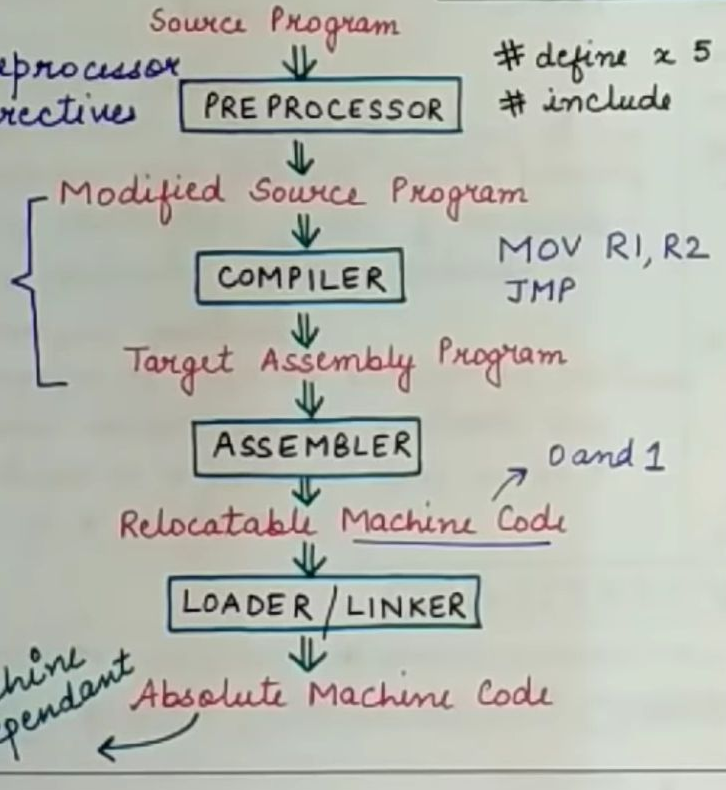
It then creates an intermediate representation of source program

Synthesis

It takes the intermediate representation as the input.

And creates the desired target program.

COUSINS OF A COMPILER



In addition to a compiler, many other programs are needed to create absolute m/c code.

PREPROCESSOR:

↳ Macro Expansion
↳ File Inclusion

ASSEMBLER:

↳ It takes the assembly code that has been generated by the compiler and converts it into relocatable Machine Code.

LINKER / LOADER:

↓ library routines, relocatable object files are linked

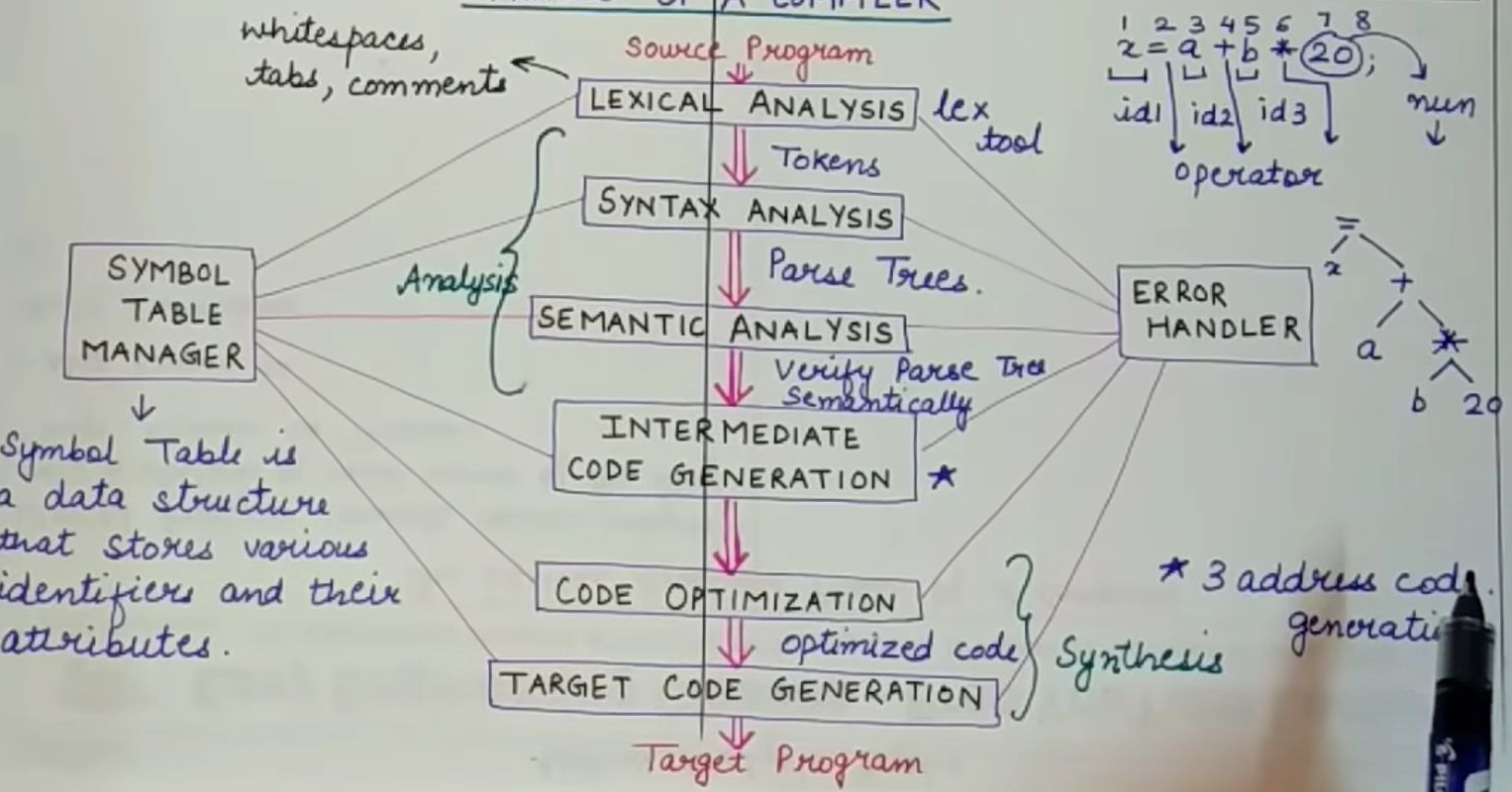
→ takes the entire program to main memory for execution.

Lecture 3

Easy Engineering Classes – Free YouTube Lectures

EEC Classes For Engineering Students of GGSIPU, UPTU and Other Universities, Colleges of India EEC Classes

PHASES OF A COMPILER



Symbol Table is a data structure that stores various identifiers and their attributes.

* 3 address code generation

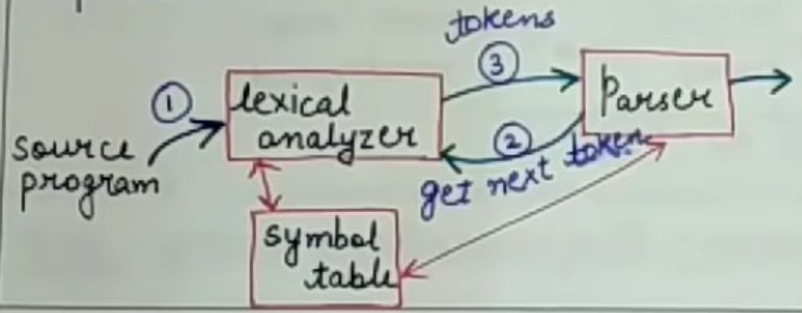
The Lexical Analysis Phase Of A Compiler

Lexical Analysis converts source program into a stream of valid words of the language, known as Tokens.

- Also known as Scanning

Basic Functions

1) Reads the input program character by character → produces a stream of tokens which is used by the parser



2) Removes comments from source program.

3) Removes whitespace characters (blanks, newline chars, tab)

4) Handling of preprocessor directives.
#define m 5; #include
↳ Macro expansion
↳ File inclusion

5) Displays errors (if present) in source program → along line & numbers.

PATTERN, TOKENS, LEXEMS

Tokens are terminal symbols of the source language.
↳ identifiers, numbers, keywords, punctuation symbols etc.

Pattern is a rule describing all those lexemes that can represent a particular token in a source language.
ids : Start with an alphabet or _, followed by any alphanumeric char.

Lexemes are matched against the pattern
↳ specific instance of a token.

`count = count + temp ;` =, +

Token : id, operator, punctuation (;)
↳ count, temp.

eg : 31 + 28 − 59

Tokens : Number : $[0\text{-}9]^+$ → 31, 28, 59.
Operator : +, −

LEXICAL ERRORS : A Lexical Analyzer may not proceed if no rule / pattern (for tokens) matches the prefix of the remaining input.

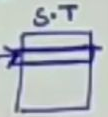
Solution :
- deleting an extraneous character(s)
- inserting a missing character.
- transposing 2 adjacent character.
- replacement of a particular char. by another char.

Attributes for a Token

Whenever a lexeme is encountered in a source program, it is necessary to keep a track of other occurences of the same lexeme i.e if this lexeme has been seen before or not.

↳ Symbol Table → Lexemes are stored in S.T.

* The pointer to this S.T entry becomes an attributes of that particular token.



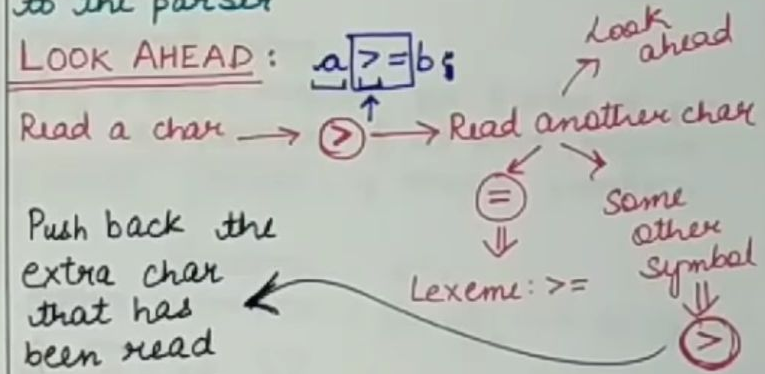
eg: e = m * c ^ 2

↓ pointer for these lexemes

The process followed by Lexical Analyzer

Take source program as input ⟹ scan it char by char ⟹ group these chars into lexemes ⟹ pass the tokens & attributes to the parser

LOOK AHEAD: a >= b;



SYNTAX ANALYSIS (Parsing)

Syntax of a language refers to the structure of valid programs / statements of that language.

- specified using certain rules known as Productions
- collection of such productions (rules) is known as grammar.

Parsing or Syntax Analysis

- is a process of determing if a string of tokens can be generated by the grammar.

(1) Parsers / Syntax Analyzer gets string of tokens from Lexical Analyzer and verifies if that string of tokens is a valid sequence i.e whether its structure is syntactically correct.

Other Tasks Of A Parser

2. Report syntactic errors.
3. Recovery from such errors so as to continue the execution process.

Output of Parser

- A representation of parse tree generated using the stream of tokens provided by the L.A.

Type checking also performed during Parsing.

GRAMMARS

A grammar consists of 4 components

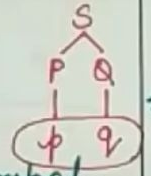
1. set of Tokens / Terminals.
2. set of Non Terminals
3. set of productions

⇓

each production has a NT, followed by arrow, followed by seq. of T and/or NT (RHS)

(4) Start symbol.

$S \to PQ$
$P \to p$
$Q \to q$



A grammar is specified by listing its productions, with the production for the start symbol appearing first

eg: expression with single digits having either + or − b/w them.

$L \to D+D \mid D-D \mid D$

$D \to 0 \mid 1 \mid 2 \mid 3 \mid 4 \mid 5 \mid 6 \mid 7 \mid 8 \mid 9$

9 + 5 etc.

$L \to D+D$
9 + 5

How To Derive strings from the Grammar

↳ Start w/ the production having the start symbol on the LHS.

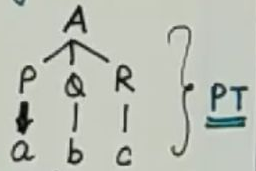
- Repeatedly replace all the NT (on RHS) by their productions.

All the strings that can be derived from a grammar belong to the language specified by that grammar.

Language → Grammar (Rules) → Productions
NT (S) T

PARSE TREES

A parse tree is a pictoral depiction of how a start symbol symbol of a grammar derives a string in the language. eg: $A \to PQR$, $P \to a$, $Q \to b$, $R \to c|d$



PROPERTIES

- Root is always labelled with the start symbol.
- Each leaf is labelled with a Terminal (Tokens)
- Each interior node is labelled by a NT.

Yield of Tree: The leaves of a Parse Tree when read from left to right form the yield.

Language defined by a grammar
↳ set of all strings that are generated by some P.T formed by that gmr.

General Types of Parsers

1. Universal Parsers
   - ↳ can parse any kind of grammar
   - ↳ Not very efficient
   - ↳ CYK Algo, Earley's algo.
2. Top Down Parsers
   - ↳ builds the Parse Tree from the root (top) to leaves (bottom).
3. Bottom-Up Parsers
   - ↳ builds the Parse tree by starting at the leaves and ending at root.

ERROR HANDLING DURING SYNTAX ANALYSIS

When the stream of tokens coming from Lexical Analyzer disobeys the syntactic (grammatical) rules of a language, syntactic errors occur.

Handling Syntactic Errors

- Report errors
- Recover from discovered errors
- Aim at not slowing down the processing of the remaining pgm.

Error Recovery Strategies

1. PANIC MODE RECOVERY : The Parser discards the input symbols one at a time untill one of the designated set of synchronizing tokens is found.
   - Simple and it does not go into ∞ loop.
   - Adequate when the presence of multiple errors in same stmt is rare.
2. PHRASE LEVEL RECOVERY : The Parser performs local correction on remaining input when the error is discovered.
   - The parser replaces the prefix of the remaining input by some string that allows the parser to carry on its execution.

   drawback : Error correction is difficult when actual error occured before the point of detection

3. ERROR PRODUCTIONS

If we know the common errors that can be encountered, we can augment the grammar for the language with productions that generate erroneous constructs.

- Use the new grammar (with the augmented productions) for the parser.
- In case an error production is used by parser, generate appropriate diagnostics to indicate the errors/erroneous constructs.

4. GLOBAL CORRECTION

The aim is to make as few changes as possible while converting an incorrect input string to a valid string.

- Given an incorrect input x, find a parse tree for a related string w (using the given grammar) such that the no. of changes (insertions/deletion) required to transform x to w is minimum
- Too Costly To Implement.

L11: Grammar for arithmetic expressions, Generation, Deriving Strings | Compiler Design Lectures

A GRAMMAR FOR ARITHMETIC EXPRESSIONS

cstmt → if expr then stmt else stmt

NT T NT T NT T

op → + | - | * | / | ↑

d → 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9

expr : d op d , -d , (d)

expr op expr , - expr , (expr)

{expr → e}

e → e op e | (e) | -e | d

d → 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9

op → + | - | * | / | ↑

grammar

eg: -5

e → -e
→ -d
→ -5.

7+5

e → e op e
→ d op d
→ 7 + 5.

-(7+5)

e → -e
→ -(e)
→ -(e op e)
→ -(d op d)
→ -(7 + 5)

-(7)

7+5 - 9*2

e → e op e
→ e op e op e
→ e op e op e op e
→ d op d op d op d
→ 7 + 5 - 9 * 2

$S \overset{*}{\Rightarrow} abc$ $A \Rightarrow a$

COMMON NOTATIONS

$\rightarrow$ or $\Rightarrow$: derives in 1 step

$\overset{*}{\Rightarrow}$: derives in 0 or more steps

$\overset{+}{\Rightarrow}$: derives in 1 or more steps

If we have a grammar G then the language generated by this grammar is denoted by L(G)

* A string w will be present in L(G) iff $S \overset{+}{\Rightarrow} w$

Context Free Language and Grammar

↓ ↓

The languages derived from CFG are called CFL. Context Free Grammar.

Equivalent Grammar

If 2 grammars generate same language then they are equivalent.

Sentential Form of Grammar : If $S \overset{*}{\Rightarrow} \alpha$ where α may contain any Non Terminal (S is the start symbol) then α is called sentential form of G.

$E \Rightarrow -E \Rightarrow -(E) \Rightarrow -(d) \Rightarrow -(7)$

$\Rightarrow -(d+d) \Rightarrow \ldots \Rightarrow (7+2)$

$E \Rightarrow d+d$

$A \rightarrow BDC$
$B \rightarrow b$
$D \rightarrow d$
$C \rightarrow c$

$\Leftrightarrow$ bdc

$A \rightarrow XY$
$X \rightarrow bd$
$Y \rightarrow c$

bdc.

Leftmost Derivation

The derivations in which only the leftmost Non Terminal is replaced at each step. (in any sentential form.)

eg: $A \to XYZ$, $X \to a$, $Y \to b$, $Z \to c$

$A \xrightarrow[lm]{} XYZ \xrightarrow[lm]{} aYZ \xrightarrow[lm]{} abZ \xrightarrow{lm} abc$ (abc: Sentence)

Canonical Derivation.

Rightmost Derivation

The derivations in which only the rightmost Non Terminal in any sentential form, is replaced at each step.

eg: $A \Rightarrow XYZ \Rightarrow XYc \Rightarrow Xbc \Rightarrow abc$

AMBIGUOUS GRAMMAR

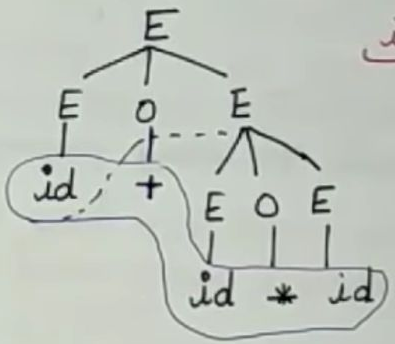
A grammar that produces more than one Parse Tree for some sentence is said to be ambiguous.

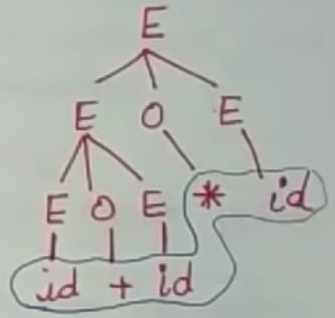
$E \to E\ O\ E \mid -E \mid (E) \mid id$ (id → Terminal)

$O \to + \mid - \mid * \mid / \mid \uparrow$.

id + id * id

id + id * id





1 string $\in L(G)$ — More than 1 LMD or More than 1 RMD.

ASSOCIATIVITY OF OPERATORS

When an operand has operators on both its sides (left and right) then we need rules to decide with which operator we will associate this operand.

Left Associative & Right Associative

$(1+2)+3$

+ : left associative

−, *, / : L.A

$4*5*6$

⇓

Parse Trees

for left associative operators are more towards the left side in length.

=, ↑

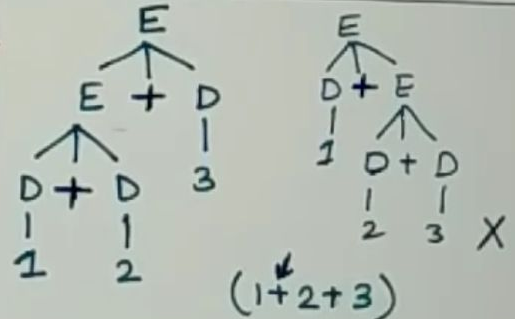
$a=b=5$

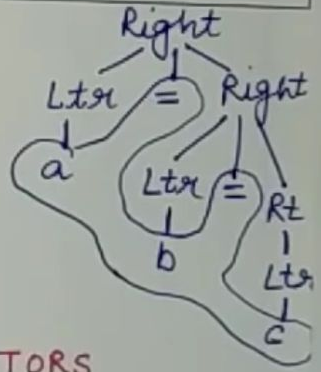
$a=b$; $2\uparrow 3\uparrow 5$

$(2)^{(3^5)}$

$1\uparrow 2\uparrow 3$

$(1)^8$





PRECEDENCE OF OPERATORS

Whenever an operator has a higher precedence than the other operator, it means that the first operator will get its operands before that operator with lower precedence.

$1+(2*3)$

$1+2-3$

+, − < *, /

CONVERTING AN AMBIGUOUS GRAMMAR INTO UNAMBIGUOUS GRAMMAR

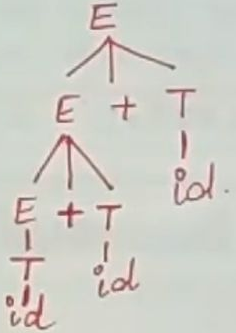
* / → Left Higher
\+ − → Left Lower

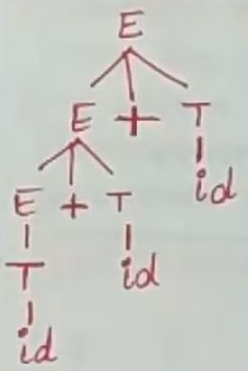
$E \to E+T \mid E-T \mid T$
$T \to T*F \mid T/F \mid F$
$F \to id$

(1+2)+3
id + id + id.

- Left Recursion : If $A \overset{+}{\Rightarrow} A\alpha$
  Right Recursion : $A \overset{+}{\Rightarrow} \alpha A$

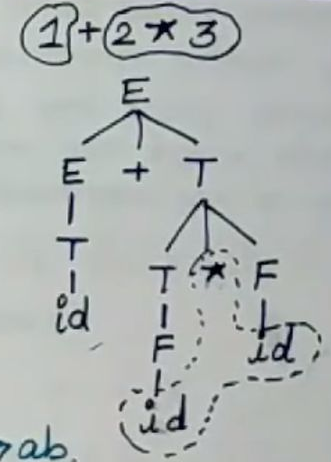
- $E \to E+T$
  $\to E+T+T$
  $\to T+T+T$
  $\to id+id+id.$





$A \to XY$
$X \to a$
$Y \to b$

$L = \{ab\}$



$A \to ab.$

$L = \{ab\}$

LEFT RECURSION

A grammar is left recursive if it has a Non Terminal A such that there is a derivation $A \overset{+}{\Rightarrow} A\alpha$ for some string α.

Direct Left Recursion : $A \to A\alpha$

Indirect Left Recursion : $S \to Aa$, $A \to Sb$

$S \overset{+}{\Rightarrow} Sb$

REMOVING LEFT RECURSION

Why ? Top Down Parsers cannot handle left recursion / grammars with LR

```
A()
{  A()
   α
}
```

How ?

$A \to A\alpha \mid \beta \Rightarrow$

$A \to \beta A'$

$A' \to \alpha A' \mid \epsilon$

```
A()
{  β
   A()
}
```

If there are multiple A productions

$A \to A\alpha_1 \mid A\alpha_2 \mid A\alpha_3 \mid \dots \mid A\alpha_m \mid \beta_1 \mid \beta_2 \mid \dots \mid \beta_n$

$A \to \beta_1 A' \mid \beta_2 A' \mid \dots \mid \beta_n A'$

$A' \to \alpha_1 A' \mid \alpha_2 A' \mid \dots \mid \alpha_m A' \mid \epsilon$

No β_i begins w/ A.

Advantage :

↳ We are able to generate the same language even after removing LR.

Disadvantage :

↳ The above procedure only eliminates Direct LR but not indirect LR

EXAMPLES OF LEFT RECURSION

1) $E \to E + T \mid T$

$T \to T * F \mid F$

$F \to (E) \mid id$ ✓

$A \to A\alpha \mid \beta$

$\Downarrow$

$A \to \beta A'$

$A' \to \alpha A' \mid \epsilon$

$E \to E + T \mid T$ (α: $+T$, β: T)

$\Rightarrow$ $\{ E \to TE' \quad E' \to +TE' \mid \epsilon \}$ ✓

$T \to T * F \mid F$ (α: $*F$, β: F)

$\Rightarrow$ $\{ T \to FT' \quad T' \to *FT' \mid \epsilon \}$ ✓✓

2) $S \to S0S1S \mid 01$ (α: $0S1S$, β: 01)

$S \to 01S'$

$S' \to 0S1SS' \mid \epsilon$

3) $L \to L,S \mid S$ (α: $,S$, β: S)

$L \to SL'$

$L' \to ,SL' \mid \epsilon$

4) $S \to SX \mid SSb \mid XS \mid a$ (α_1: X, α_2: Sb, β_1: XS, β_2: a)

$\Rightarrow$ $S \to SX \mid SSb \mid XS \mid a$

$S \to XSS' \mid aS'$

$S' \to XS' \mid SbS' \mid \epsilon$

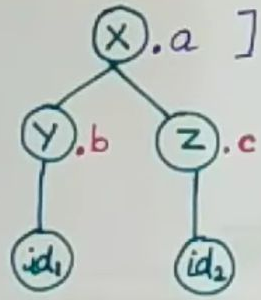
5) $A \to AA \mid Ab$ (α_1: A, α_2: b)

$A' \to AA' \mid bA' \mid \epsilon$.

SYNTAX DIRECTED DEFINITION

With every symbol present in grammar Syntax Directed Definition associates a set of attributes & with each production, a set of semantic rules for computing values of the attributes associated with the symbols appearing in that production. The grammar and the set of semantic rules make SDD.

$X \to YZ$
$Y \to id_1$
$Z \to id_2$



X.a denotes that node X has an attribute called 'a'.

⇓

Value of X.a is found using X production semantic rule for the used at this node.

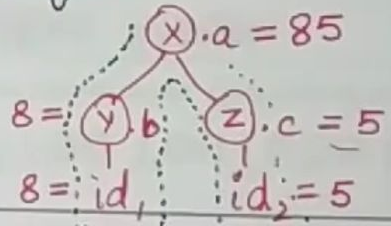
Annotated Parse Tree:

A Parse Tree with attribute values at each node, is Annotated Parse Tree.

Synthesized Attributes:

An attribute is said to be a synthesized attr. if its value at a PT node is determined from the attribute values of the children of the node.

Advantage:

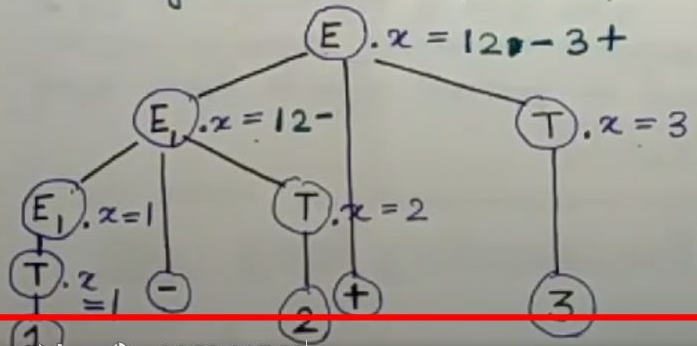


S.A can be evaluated during a single Bottom-Up Traversal.

Infix To Postfix

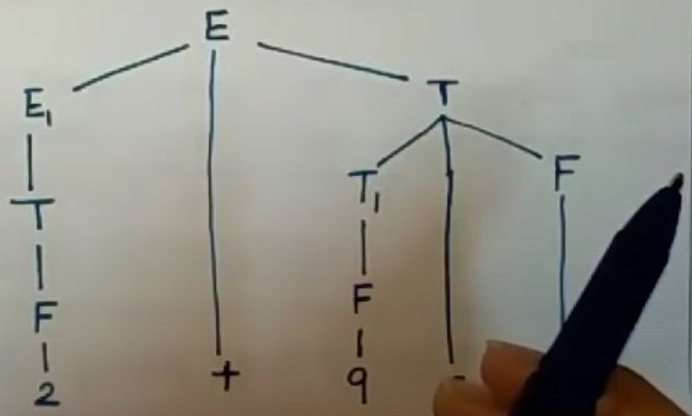
$E \to E_1 + T$	$E.x \Rightarrow E_1.x \parallel T.x \parallel '+'$
$E \to E_1 - T$	$E.x \Rightarrow E_1.x \parallel T.x \parallel '-'$
$E \to T$	$E.x \to T.x$
$T \to 0$	$T.x \to '0'$
$T \to 1$	$T.x \to '1'$
$T \to 2$	$T.x \to '2'$
⋮	⋮
$T \to 9$	$T.x \to '9'$

eg: 12-3+ (1-2+3)



Syntax Directed Definition Example

$E \to E_1 + T$	$E.val = E_1.val + T.val$
$E \to T$	$E.val = T.val$
$T \to T_1 * F$	$T.val = T_1.val * F.val$
$T \to F$	$T.val = F.val$
$F \to (E)$	$F.val = E.val$
$F \to digit$	$F.val = digit.lexval$



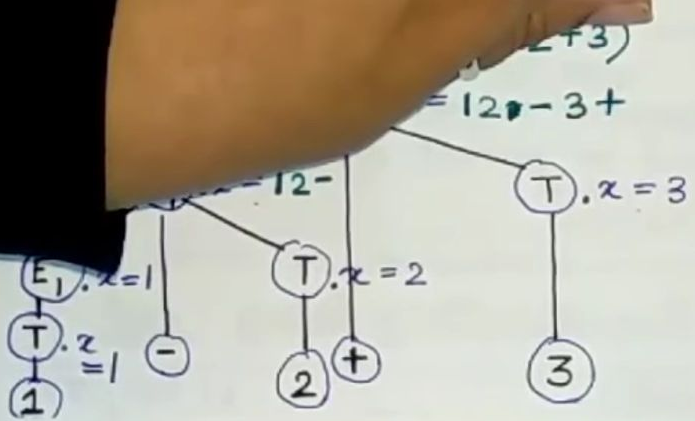
Infix To Postfix

$E \to E_1 + T$ $\quad E.x \Rightarrow E_1.x \,\|\, T.x \,\|\, '+'$

$E \to E_1 - T$ $\quad \Rightarrow E_1.x \,\|\, T.x \,\|\, '-'$

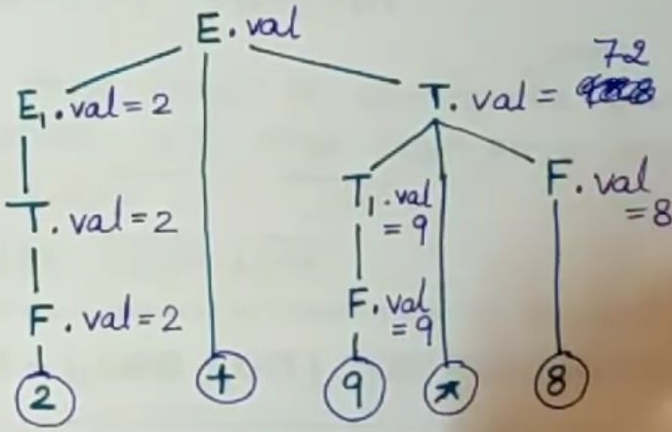
$E \to T$

$T \to$



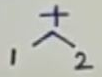
Syntax Directed Definition Example

$E \to E_1 + T$	$E.val = E_1.val + T.val$
$E \to T$	$E.val = T.val$
$T \to T_1 * F$	$T.val = T_1.val * F.val$
$T \to F$	$T.val = F.val$
$F \to (E)$	$F.val = E.val$
$F \to digit$	$F.val = digit.lexval$



Abstract Syntax Tree : Each node in an AST represents an operator and the children of the operator are the operands (of this oper^{r})

eg: 1+2



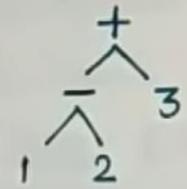
↳ No unimportant details present.

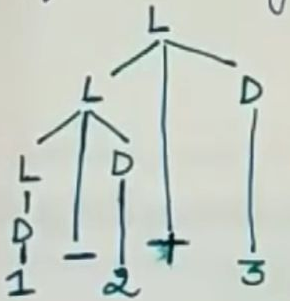
Concrete Syntax Tree

↳ It is a normal Parse Tree and the underlying grammar is called concrete syntax for the language.

eg: 1-2+3

AST

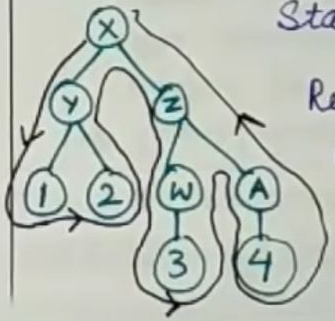




SDD does not impose any specific order for the evaluation of attributes on Parse Tree. Any evaluation order that computes 'a' after all the attributes that 'a' depends on, is acceptable.

Traversal of a tree starts at the root and visits each node of the tree in some particular order.

Depth First Traversal



Start with root

⇓

Recursively visit children of each node in left to right manner.

Translation: Given an input string x, when we construct a Parse Tree for x and convert it into Annotated Parse Tree (O/P), this mapping from input to output is called translation.

Translation Scheme:

It is a context free grammar in which program pieces/fragments are embedded in RHS of the productions (called Semantic Actions)

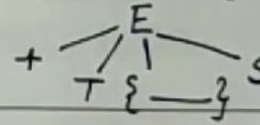
↳ Similar to SDD except the fact that Translation scheme also specifies explicitly: the evaluation order of semantic actions.

⇒ Translation scheme generates an output for each string x present in the language generated by the grammar, by executing the actions in the order in which they appear in the Depth First Traversal of PT for x.

eg: $E \rightarrow + T \{ print('*') \} S$

For making a Parse Tree for a Translation Scheme :-

1. Make an extra node for semantic actions.
2. Nodes for semantic actions do not have children.
3. Evaluate node for SA whenever that node is seen.

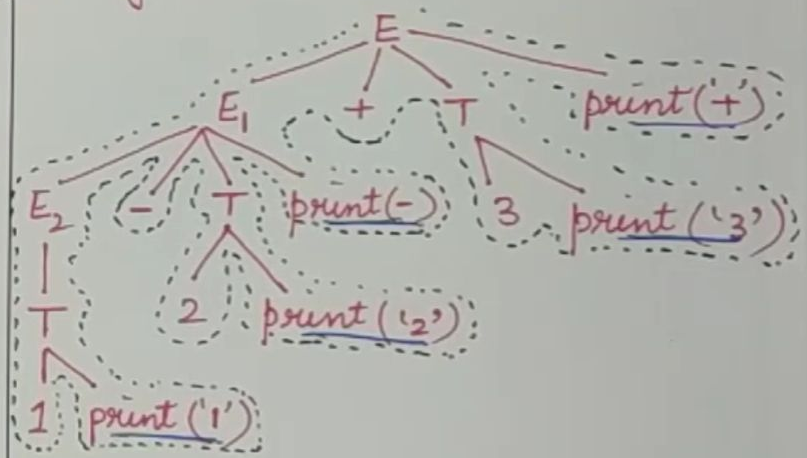


INFIX TO POSTFIX TRANSLATION

$E \to E_1 + T \longrightarrow E.x = E_1.x \parallel T.x \parallel '+'$
$E \to E_1 - T \longrightarrow E.x = E_1.x \parallel T.x \parallel '-'$
$E \to T \longrightarrow E.x = T.x$
$T \to 0 \longrightarrow T.x = '0'$
$T \to 1 \longrightarrow T.x = '1'$
⋮ ⋮
$T \to 9 \longrightarrow T.x = '9'$

1 $E \to E_1 + T$: { print('+') }
2 $E \to E_1 - T$: { print('-') }
3 $E \to T$: —
4 $T \to 0$: { print('0') }
5 $T \to 1$: { print('1') }
⋮
$T \to 9$: { print('9') }

eg : 1 - 2 + 3



1 2 - 3 +

Finding FIRST()

If α is any string of grammar symbols then FIRST(α) is the set of terminals that begin the string derived from α.
If $\alpha \overset{*}{\Rightarrow} \epsilon$ then ϵ is also in FIRST(α)

Steps To Find FIRST()

1. If X is a terminal then FIRST(X) is {X}.
2. If X is a Non Terminal and $X \to Y_1 Y_2 \dots Y_k$ is a production then
   - (a) Add '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_2) ..... FIRST(Y_{i-1}) i.e $Y_1, Y_2, \dots, Y_{i-1} \overset{*}{\Rightarrow} \epsilon$
   - (b) If ϵ is in FIRST(Y_j) for all $j = 1, 2, \dots, k$ then add ϵ to FIRST(X)
3. If $X \to \epsilon$ is a production then add ϵ to FIRST(X)

$X \overset{*}{\Rightarrow} abc$, $X \overset{*}{\Rightarrow} def$ $= \{a, d\}$ 'a' → a, $\epsilon \to \epsilon$

(aABc)

(AB) A → c

$X \to AB$
$A \to a \mid \epsilon$
$B \to b \mid \epsilon$
$\{a, b\}$

$X \to aA$. $\{a\}$

$X \to AB \to aB \to ab$.
$X \to \epsilon B \to (B) \to b$

$X \to \epsilon B \to B \to \epsilon$

$E \to (TE') = \{id, (\}$
$E' \to +TE' \mid \epsilon$ $\{+, \epsilon\}$
$T \to (FT')$ First(T) = {id, (}
$T' \to (*FT') \mid \epsilon$ First(T') = {*, ε}
$F \to id \mid (E)$ First(F) = First(id) = {id, (}
T T

SOLVED EXAMPLES

$S \to aABb$: First(S) = First(aABb) = {a}
$A \to c \mid \epsilon$ First(A) = First(c) = {c, ϵ}
$B \to d \mid \epsilon$ First(B) = First(d) = {d, ϵ}

$S \to aBDh$ First(S) = First(aBDh) = {a}
$B \to cC$ {c}
$C \to bC \mid \epsilon$ {b, ϵ}
$D \to EF$ First(D) = First(EF) = {g, f, ϵ} (First(E) − ϵ, First(F))
$E \to g \mid \epsilon$ First(E) = {g, ϵ}
$F \to f \mid \epsilon$ First(F) = {f, ϵ}

$D \to EF \to \epsilon F \to \epsilon$

$S \to Bb \mid Cd$ First(S) = {a, b, c, d}
$B \to aB \mid \epsilon$ {a, ϵ}
$C \to cC \mid \epsilon$ {c, ϵ}

$A \to da \mid BC$ = {d, g, h, ϵ}
$S \to ACB \mid CbB \mid Ba$ = {d, g, h, ϵ, b, a}
$B \to g \mid \epsilon$ = {g, ϵ}
$C \to h \mid \epsilon$ = {h, ϵ}

$S \to AB$ {b, a, c}
$A \to Ca \mid \epsilon$ {b, a, ϵ}
$B \to BaAC \mid c$ {c}
$C \to b \mid \epsilon$ {b, ϵ}

$S \to ABCDE$ = {a, b, c}.
$A \to a \mid \epsilon$ {a, ϵ}
$B \to b \mid \epsilon$ {b, ϵ}
$C \to c$ {c}
$D \to d \mid \epsilon$ {d, ϵ}
$E \to e \mid \epsilon$ {e, ϵ}

Finding FOLLOW()

For a Non Terminal A, Follow(A) is the set of terminals 'a' that can appear immediately to the right of A in some sentential form i.e the set of terminals 'a' such that there exists a derivation of the form $S \overset{*}{\Rightarrow} \alpha A a \beta$ for some α and β

There may be symbols between A and 'a' which derived ϵ and disappeared

Rules for finding FOLLOW()

1) Put $ in FOLLOW(s) where S is start symbol and $ is input end marker
2) If $A \to \alpha B \beta$ then everything in First(β) is placed in Follow(B) except ϵ.
3) If $A \to \alpha B$ or $A \to \alpha B \beta$ where First(β) contains ϵ, then everything in Follow(A) is in Follow(B).

ϵ never appears in FOLLOW().

⇒ To find Follow(A), look at the productions that have A present at the right hand side.

First(E') = {+, ϵ}
First(T') = {*, ϵ}

E → TE' {$,)} (E: start symbol)
E' → +TE' | ϵ Follow(E') = Follow(E) = {$,)}
T → FT' Follow(T) = First(E') = {+, $,)}
T' → *FT' | ϵ Follow(T') = {+, $,)}
F → (E) | id Follow(F) = {*, +, $,)}

SOLVED EXAMPLES

$S \to aABb = \{\$\}$
$A \to c|\epsilon = \{d, b\}$
$B \to d|\epsilon = \{b\}$

First(B) = d, ϵ

$S \to aBDh = \{\$\}$
$B \to cC = \{g, f, h\}$
$C \to bC|\epsilon = \{g, f, h\}$
$D \to EF = \{h\}$
$E \to g|\epsilon = \{f, h\}$
$F \to f|\epsilon = \{h\}$

Follow(B) = First(D) = First(EF)
First(E) → g, ϵ
First(F) = f, ϵ

$S \to Bb|Cd = \{\$\}$
$B \to aB|\epsilon = \{b\}$
$C \to cC|\epsilon = \{d\}$

$A \to da|BC = \{h, g, \$\}$
$S \to ACB|CbB|Ba \quad \{\$\}$
$B \to g|\epsilon = \{\$, a, h, g\}$
$C \to h|\epsilon = \{g, \$, h, b\}$

First(C) = {h, ϵ}
First(B) = {g, ϵ}

$S \to xyz|aBC \quad \{\$\}$
$B \to c|cd \quad \{e, d\}$
$C \to eg|df \quad \{\$\}$

First(C) = e, d

$S \to ABCDE = \{\$\}$
$A \to a|\epsilon \to \{b, c\}$
$B \to b|\epsilon \to \{c\}$
$C \to c \to \{d, e, \$\}$
$D \to d|\epsilon \to \{e, \$\}$
$E \to e|\epsilon \to \{\$\}$.

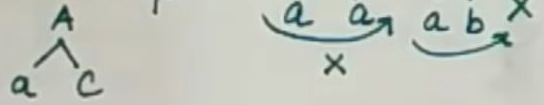
First(B) = b, ϵ

LEFT FACTORING

At times, it is not clear which out of 2 (or more) productions to use to expand a Non-Terminal because multiple productions begin with same lookahead.

$A \to aa|$
$\to ab|$
$\to ac|$

IP = a c



A grammar with left factoring present is a NON DETERMINISTIC Grammar.

Removing Left Factoring

Why?
↳ Top Down Parsers cannot work with gmr. having L.F.

How? $A \to \alpha\beta_1 | \alpha\beta_2$

$A \to aA'$
$A' \to a|b|c$

$\{ A \to \alpha A' \}$
$\{ A' \to \beta_1 | \beta_2 \}$

$A \to \alpha\beta_1 | \alpha\beta_2 | \dots | \alpha\beta_m | \gamma$

$\Downarrow$

$A \to \alpha A' | \gamma$
$A' \to \beta_1 | \beta_2 | \dots | \beta_m$.

eg: stmt → if expr then stmt, else stmt
| if expr then stmt,

stmt → if expr then stmt A
A → else stmt | ∈

LEFT FACTORING EXAMPLES

$S \to iEtS \mid iEtSeS \mid a$
$E \to b$

$S \to iEtSS' \mid a$
$S' \to eS \mid \epsilon$
$E \to b$

$X \to X+X \mid X*X \mid D$
$D \to 1 \mid 2 \mid 3$

$X \to XY \mid D$
$Y \to +X \mid *X$
$D \to 1 \mid 2 \mid 3$

$\alpha = X$
$\beta_1 = +X$
$\beta_2 = *X$
$\gamma = D$

$E \to T + E \mid T \longrightarrow \alpha = T,\ \beta_1 = +E,\ \beta_2 = \epsilon$
$T \to int \mid int * T \mid (E)$

$A \to \alpha\beta_1 \mid \alpha\beta_2$
$\Downarrow$
$A \to \alpha A'$
$A' \to \beta_1 \mid \beta_2$

$E \to TE'$
$E' \to +E \mid \epsilon$
$T \to int\ T' \mid (E)$
$T' \to *T \mid \epsilon$.

$S \to aSSbS \mid aSaSb \mid abb \mid b$
$S \to aS' \mid b$ ✓
$S' \to SSbS \mid SaSb \mid bb \Rightarrow \{S' \to SS'' \mid bb,\ S'' \to SbS \mid aSb\}$

$\alpha = aS \quad \beta_1 = SbS \quad \beta_2 = aSb$
$S \to aSS' \mid abb \mid b \Rightarrow S \to aS'' \mid b$ ✓
$S' \to SbS \mid aSb$ ✓ $\quad S'' \to SS' \mid bb$. ✓

$A \to aA$
$B \to aB$
} No common Non Terminal on LHS.