Compiler Design (CSE)

Syntax Analysis-I

Role of Parsers:

In the syntax analysis phase, a compiler verifies whether or not the tokens generated by the lexical analyser are grouped according to the syntactic rules of the language. This is done by a parser.

It is a program that takes input string w (obtain set of strings tokens from the lexical analyser) and produces as o/p either a parse tree or error message.

The goal is to determine the syntactic validity of a source string is valid, a tree is build for use by the subsequent phases of the computer.

1. It verifies the structure generated by the tokens based on the grammar.
2. It constructs the parse tree.
3. It reports the errors.
4. It performs error recovery.

Parser cannot detect errors such as:
1. Variable re-declaration
2. Variable initialization before use
3. Data type mismatch for an operation.

The above issues are handled by Semantic Analysis phase.

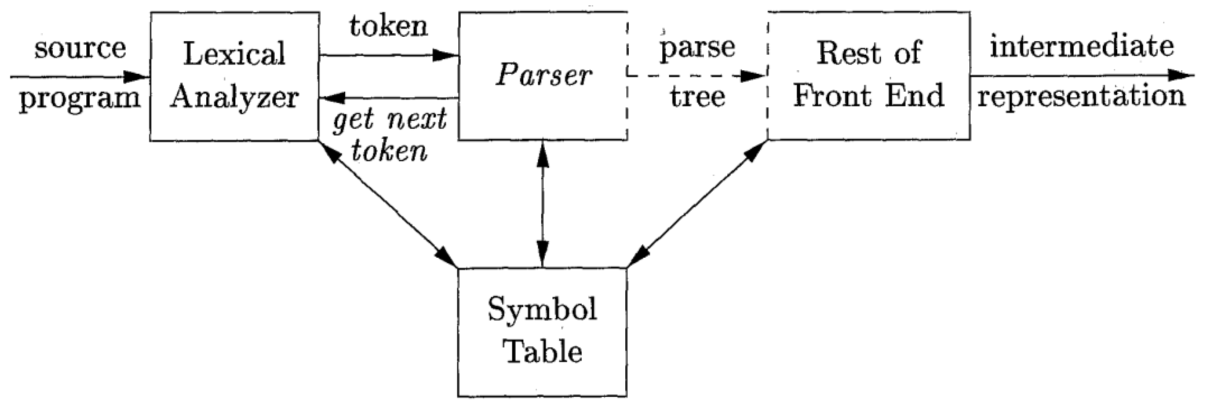


Figure 4.1: Position of parser in compiler model

Grammars

Defn:

$G = (N, T, P, S)$

- S — Start symbol
- P — Production rules
- T — set of terminals
- N — set of variables.

Determination of languages from Grammar.

Ex 1: P: $S \to aSb \mid \epsilon$

Derivation:

$S \to aSb$
$\Rightarrow ab \quad (S \to \epsilon)$

$S \to aSb$
$\Rightarrow aaSbb$
$\Rightarrow a^n b^n$ (applying n-times)

$L = \{a^n b^n \mid n >= 0\}$

Ex 2: P: $S \to aS \mid bS \mid a$

Ex 3: P : $S \rightarrow aSb \mid aAb$

$A \rightarrow aA \mid a$

Generating Grammers from Languages.

Ex:- 1 $L = \{a^n \mid n >= 1\}$

P: $S \rightarrow aS$
$S \rightarrow \epsilon$

Ex 2: All strings over $\{ab\}$

P: $S \rightarrow aS$
$S \rightarrow bS$
$S \rightarrow \epsilon$

Ex 3: set of all strings starting and ending with different symbol.

Step 1: Find RE : $a(a+b)^*b + b(a+b)^*a$.

Step 2: Production Rules:

$S \rightarrow aAb \mid bAa$
$A \rightarrow aA \mid bA \mid \epsilon$

Ex 4: $L = \{a^n b^n \mid n \geq 1\}$

Ex 5: $L = \{ a^n c^m b^n \mid n, m >= 1 \}$

Ex 6: $L = \{ a^n b^n c^m d^m \mid n, m >= 1 \}$

Chomsky's Classification of grammars:

Type-3 (Regular Grammar)

~~(i)~~ ~~$A \to \alpha B$~~

(i) $\boxed{A \to \alpha B \mid \beta \quad , \quad A, B \in V \quad \alpha, \beta \in T^*}$ Right Linear Grammar

or (ii) $\boxed{A \to B\alpha \mid \beta \quad , \quad A, B \in V \quad \alpha, \beta \in T^*}$ Left Linear Grammar.

So, in Type 3 production may be of the form above.

$S \to \lambda$ is allowed only if S-does not appear on the RHS of any production.

Form: $V \to VT^* \mid T^*$ (LLG)

or $V \to T^*V \mid T^*$ (RLG)

Ex:- (i) $A \to aB \mid a$ | $A \to Ba \mid a$
$B \to aB \mid bB \mid a \mid b$ | $B \to Ba \mid Bb \mid a \mid b$.

(ii) $A \to Ba \mid a$
$B \to aB \mid a$ (cannot be combination of both LLG & RLG)
so not T3.

Type - 2.

Context Free Grammar

Form : $A \rightarrow \alpha$, $A \in V$
$\alpha \in (V \cup T)^*$ [set of all terminals and non-terminals including NULL]

Note:
- No need to be worried about the Left and Right context, directly replace by terminals.
- All regular grammars are context free.
- It should be of type 1
- LHS of production can have only one variable.
  if $A \rightarrow B$
  then $|A| = 1$
- No restrictions on B. (can be of any length)

Ex 1:- $A \rightarrow aAb \mid ab$

Ex 2:-
$S \rightarrow AB$
$A \rightarrow a$
$B \rightarrow b$

Type 1

Context Sensitive Grammar

$\alpha \rightarrow \beta$
$\alpha \in (T \cup V)^* V (T \cup V)^*$
$\beta \in (T \cup V)^+$

Restriction: $|\alpha| \leq |\beta|$

$\alpha A \beta \rightarrow \alpha \delta \beta$
$\alpha, \beta \in (T \cup V)^*$
$A \in V$
$\delta \in (T \cup V)^+$

- We are producing different δ based on Left and Right context
- RHS should not be NULL (ϵ - not allowed in RHS).
- Null is applicable only for production, $S \rightarrow \epsilon$, where S does not appear in RHS.

Ex:- Standard Examples

1) $L = \{a^n b^n c^n : n \geq 1\}$
2) $L = \{a^n : n \text{ is prime}\}$
3) $L = \{a^{n!}, n \geq 0\}$
4) $L = \{a^n : n \text{ is non-prime}\}$
5) $L = \{ww : w \in (a,b)^+\}$
6) $L = \{w w w^R : w \in (a,b)^+\}$

Type-0 (Unrestricted)
Recursive Enumerable Grammar

form: $\alpha \rightarrow \beta$

$\alpha \in (T \cup V)^* V (T \cup V)^*$

$\beta \in (T \cup V)^*$

- only restriction is, LHS should have atleast one non-terminal.

Parse Trees.

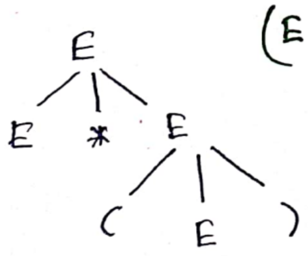
In compiler design, parse tree is the data structure of choice to represent the source program. This facilitates the translation of the source program into executable code by allowing natural, recursive functions to perform this translation process.

construction:

- Each interior node → Non-terminals. (V)
- Each leaf node → Terminal or ϵ (T or ϵ)

Ex: 1 $E \rightarrow E * E \mid (E)$

$(E \Rightarrow E * (E))$



$E \Rightarrow E * E$

$\Rightarrow E * (E)$

Ex-2 $P \rightarrow 0P0 \mid 1P1 \mid \epsilon$ $\quad (P \overset{*}{\Rightarrow} 0110)$

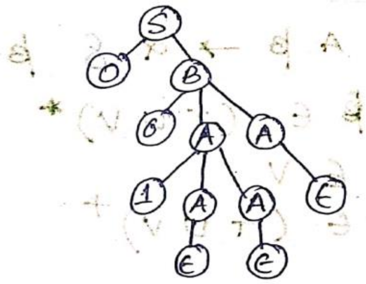
$P \Rightarrow 0P0$

$\Rightarrow 01P10$

$\Rightarrow 0110 \quad (P \rightarrow \epsilon)$

Ex 3 $G = \{V, T, P, S\}$ where $S \to 0B$, $A \to 1AA \mid \epsilon$, $B \to 0AA$

$T = \{0, 1\}^*$, $V = \{A, B\}$



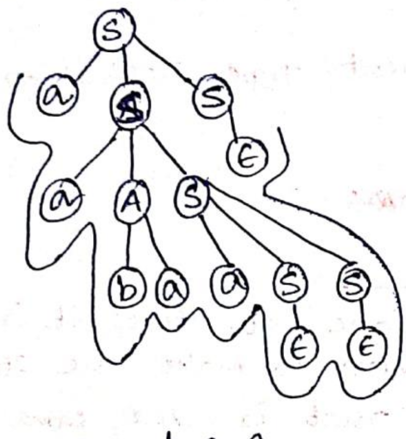
Left Derivation and Right Derivation for a string.

Ex:

$$\boxed{\begin{array}{l} S \to aAS \mid aSS \mid \epsilon \\ A \to SbA \mid ba \end{array}}$$

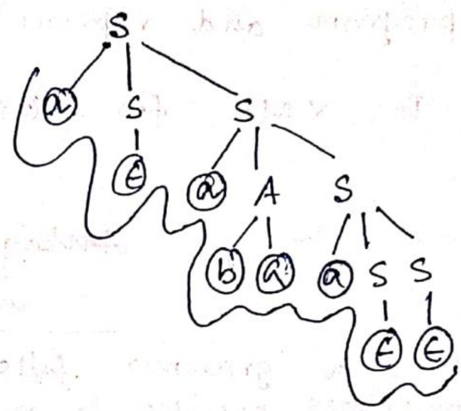
string: aabaa

lm



aabaa

rm



aabaa

Q. Write a CFG to generate all possible palindromes over {a,b}.

Ambiguity in Grammars and Languages.

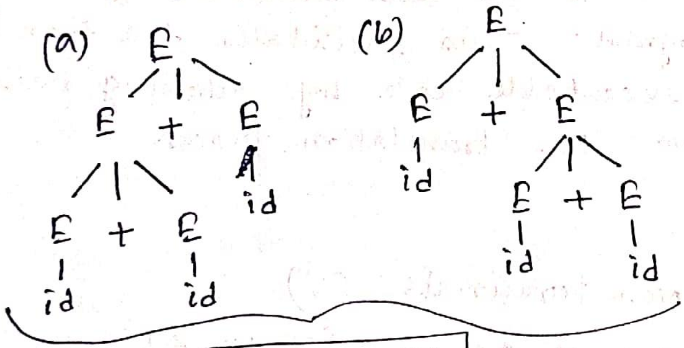
When a grammer fails to provide unique structures, it is sometimes possible to redesign the grammar to make the structure unique for each string in the language. That is, the same internal string may be the yield of two derivation trees.

Ambiguous Grammer :-

$$E \rightarrow E + E$$
$$E \rightarrow E * E$$
$$E \rightarrow id$$

1. String :- $\boxed{id + id + id}$

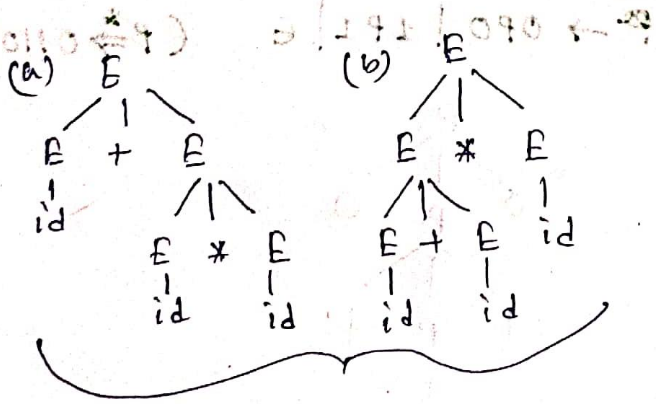
Now we get 2- Parse trees,



(b) Failed due to Associativity Rules.

If the operator has left Ass.- then the grammar should be Left Recursive.

2. string : $\boxed{id + id * id}$



(b) Failed due to Precedence Rules.

In Ex. 2(a) It properly groups * before the + operator. That is maintaining the precedence rules of operators in an expression.

again, In Ex, 1(a), the associativity is maintained, '+' is used as left associative.

$$(((E+E)+E)+E)\ldots$$

so, As the solution to the problem of precedence and associativity, we need to introduce several different variable to give the level of "binding strength".

$E \to E+T \,/\, T$

$T \to F \,/\, T * F$ (Removing ambiguity)

$F \to id \,/\, (E)$

$id \to a|b|ida|idb|id0|id1$ (Letter (Letter | digit)*)

[* takes precedence over + ; * must be grouped before adjacent '+''s on either side].

[Both * and + left-associative].

Factors ⇒ identifiers or paranthesis. (cannot be broken apart by * or +

Term ⇒ cannot be broken by +.
so, (a * b) * a1 can be broken by left Associativity.
But a + b * a1, cannot broken by +, so, always proper grouping be ~~a+b~~. a + (b * a1).

Expression ⇒ cannot be broken by either adjacent * or adjacent +.
simply sum of one or more example.

∴ The grammar is unambiguous.

Applications of CFG

- There is a mechanical way of turning the language description as a CFG into a parser, the component of the compiler that discovers the structure of the source program and represents that structure by a parse tree.
- In XML for designing Document Type Definition (DTD)