COMPILER DESIGN
CS6109 MODULE 3 & 4

BE COMPUTER SCIENCE AND ENGINEERING

REGULATION 2018 RUSA

Thanasekhar B

COURSE OBJECTIVES : CS6109 COMPILER DESIGN

- To know about the various transformations in the different phases of the compiler, error handling and means of implementing the phases
- To learn about the techniques for tokenization and parsing
- To understand the ways of converting a source language to intermediate representation
- To have an idea about the different ways of generating assembly code
- To have a brief understanding about the various code optimization techniques

CS6109 COMPILER DESIGN

- MODULE 3
  - Error handling
  - Error Detection and Recovery
  - Lexical phase error management
  - Syntax phase error management
  - Error recovery routines
- MODULE 4
  - Context-Free Grammar (CFG)
  - Derivation Trees
  - Ambiguity in Grammars and Languages
  - Need and Role of the parser
- Extra Learning Component
  - LEX & YACC for Language front end design and syntax verification

Errors

Lexical Errors and Recovery

- Scanner throws errors when a situation arises in which the it couldn't proceed because none of the patterns for tokens matches any prefix of the remaining input
- Simple Error Recovery Strategies for Scanners attempts to see whether a prefix of the remaining input can be transformed into a valid lexeme by a single transformation such as
  - Delete one character from the remaining input
  - Insert a missing character into the remaining input
  - Replace a character by another character
  - Transpose two adjacent characters
- Panic mode recovery - Delete successive characters from the remaining input, until the lexical analyser can find a well-formed token at the beginning of what input is left
- A costly strategy is to find the smallest number of transformations needed to convert the source program into one that consists only of valid lexemes

Syntactic Errors

- When the stream of tokens does not fit into the specification of the syntactic constructs according to the CFG, the parser detects and reports errors
- Parsers use the viable-prefix property, to detect that an error has occurred as soon as they see a prefix of the input that cannot be completed to form a string in the language
- The error handlers should
  - Report the presence of errors clearly and accurately
  - Recover from each error quickly enough to detect subsequent errors
  - Add minimal overhead to the processing of correct programs
  - Report the place in the source program where an error is detected
  - Print the offending line with a pointer to the position at which an error is detected

Error Recovery Strategies

Panic-mode

- On discovering an error, the parser discards input symbols one at a time until one of a designated set of synchronizing tokens is found
- The synchronizing tokens are usually delimiters, such as semicolon or *}*, whose role in the source program is clear and unambiguous
- This is simple and guaranteed not to go into an infinite loop, even though it skips a considerable amount of input without checking it for additional errors

Phrase-level

- The parser performs local correction on the remaining input, by replacing the prefix of the remaining input by some string that allows the parser to continue
- A typical local correction is to replace a comma by a semicolon, delete an extraneous semicolon, or insert a missing semicolon
- The replacements should not lead to infinite loops
- Phrase-level replacement has been used in several error-repairing compilers, as it can correct any input string

Error-productions

- Error-Productions are used for common erroneous constructs
- A parser constructed from these error productions detects the anticipated errors when an error production is used during parsing
- The parser can then generate appropriate error diagnostics about the erroneous construct that has been recognized in the input

Global-correction

- Uses algorithms for choosing a minimal sequence of changes to obtain a globally least-cost correction
- Given an incorrect input string x and grammar G, these algorithms will find a parse tree for a related string y, such that the number of insertions, deletions, and changes of tokens required to transform x into y is as small as possible
- The closest correct program may not be what the programmer had in mind

Context Free Grammar (CFG)

Definition

- CFG consists of Non terminals (Variables), Terminals, Productions, and Start symbol

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

- Non terminals are syntactic variables that denote sets of strings
- Terminals are the basic symbols from which strings are formed
- The productions of a grammar specify the manner in which the terminals and non terminals can be combined to form strings
- Start symbol is a distinguished non terminal, and the set of strings it denotes is the language generated by the grammar
- Each production consists of a non terminal called the head or left side of the production and a body or right side consisting of zero or more terminals and non terminals

$$A \rightarrow \alpha \text{ where } A \in N, \alpha \in (N \cup T)^*$$

Notational Conventions

- The capital letters A, B, C, D,… denote Non terminals and S denote the start symbol, unless otherwise stated
- The lower case letters a, b, c, d, … and digits denote terminals
- The capital letters U, V, W, X, Y and Z denote symbols that may be Terminals or Non terminals
- Lower case letters u, v, w, x, y and z denote string of terminals or sentences
- The lower case Greek letters α, β, γ and η denote string of Non terminals and Terminals, even empty

CFG - Examples

- Identifiers

  I → I L | I D | L

  L → a | b | c

  D → 0 | 1 | 2
- Arithmetic Expressions

  E → E + E | E * E | E ^ E | (E) | id
- Postfix Expression

  E → E E + | E E - | E E * | E E / | id
- Assignment Statement

  A → id = E

  E → E + E | E * E | E ^ E | (E) | id
- List

  L → (T) | id

  T → T , L | L
- Balanced Brackets

  B → B B | (B) | ∈
- IF Statement

  S → i C t S | i C t S e S | a

  C → b
- While Statement

  W → while (C) do S

  C → b

  S → a
- Declarations

  D → T id ; D | ∈

  T → int | float | char
- Function Call

  F → id (A)

  A → A , id | id

CFG and Derivations

- Assume $\alpha A\beta$ is a sequence of grammar symbols, where α and β are arbitrary strings of grammar symbols from $(N \cup T)^*$
- Suppose $A \rightarrow \gamma$ is a production, then, we write $\alpha A\beta \Rightarrow \alpha\gamma\beta$, the symbol $\Rightarrow$ means derives in one step
- If $\alpha_1, \alpha_2, \alpha_3, \ldots \alpha_m$ are strings in $(N \cup T)^*$, $m \geq 1$, and $\alpha_1 \Rightarrow \alpha_2 \Rightarrow \alpha_3 \Rightarrow \ldots \Rightarrow \alpha_m$, then we can say $\alpha_1 \overset{*}{\Rightarrow} \alpha_m$, ie. α_1 derives α_m in zero or more steps

CFG for Prefix Expression

$E \rightarrow + E\ E \mid * E\ E \mid x \mid y$

The derivation for + * x y x is

E	$\Rightarrow$ + E E	Using $E \rightarrow + E\ E$
	$\Rightarrow$ + * E E E	Using $E \rightarrow * E\ E$
	$\Rightarrow$ + * x E E	Using $E \rightarrow x$
	$\Rightarrow$ + * x y E	Using $E \rightarrow y$
	$\Rightarrow$ + * x y x	Using $E \rightarrow x$

This can be written as

$E \overset{*}{\Rightarrow} + * x\ y\ x$

+ * x y x is a string of only terminals (T^*), termed as sentence or yield of a derivation

Types of Derivations

Left Most Derivation

- In each step of the derivation, if the left most variable is expanded using its right hand side, then the derivation is called left most derivation

$$S \overset{*}{\Rightarrow} w\,A\,\alpha \Rightarrow w\,\beta\,\alpha$$

where $A \rightarrow \beta$ is a production, $w \in T^*$, $\alpha, \beta \in (N \cup T)^*$

$$
\begin{aligned}
E &\Longrightarrow E + E \\
&\Longrightarrow id + E \\
&\Longrightarrow id + E * E \\
&\Longrightarrow id + id * E \\
&\Longrightarrow id + id * id
\end{aligned}
$$

Right Most Derivation

- In each step of the derivation, if the right most variable is expanded by its right hand side, then the derivation is called right most derivation

$$S \overset{*}{\Rightarrow} \alpha\,A\,w \Rightarrow \alpha\,\beta\,w$$

where $A \rightarrow \beta$ is a production, $w \in T^*$, $\alpha, \beta \in (N \cup T)^*$

$$
\begin{aligned}
E &\Longrightarrow E + E \\
&\Longrightarrow E + E * E \\
&\Longrightarrow E + E * id \\
&\Longrightarrow E + id * id \\
&\Longrightarrow id + id * id
\end{aligned}
$$

Parse Tree/Derivation Tree

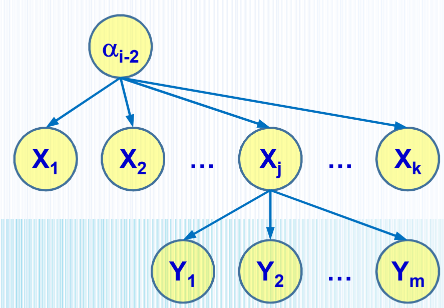
Definition

- A parse tree is a graphical representation of a derivation that filters out the order in which productions are applied to replace non terminals
- The interior node is labelled with the non terminal A in the head of the production
- The children of the node are labelled, from left to right, by the symbols in the body of the production by which this A was replaced during the derivation
- The leaves of a parse tree are labelled by terminals and, read from left to right, constitute a sentence, called the yield or frontier of the tree

Relationship with Derivation

- Let $\alpha_1 \Rightarrow \alpha_2 \Rightarrow \alpha_3 \Rightarrow \ldots \Rightarrow \alpha_m$ be a derivation where α_1 is a single nonterminal A
- For each sentential form α_i in the derivation, we can construct a parse tree whose yield is α_i
- The tree for $\alpha_1 = A$ is a single node labelled A
- Assume the parse tree with yield $\alpha_{i-1} = X_1 X_2 \ldots X_k$
- α_i is derived from α_{i-1} by replacing X_j, a non terminal by $\beta = Y_1 Y_2 \ldots Y_m$
- This means that in the i^{th} step of the derivation, production X_j is applied to α_{i-1} to derive

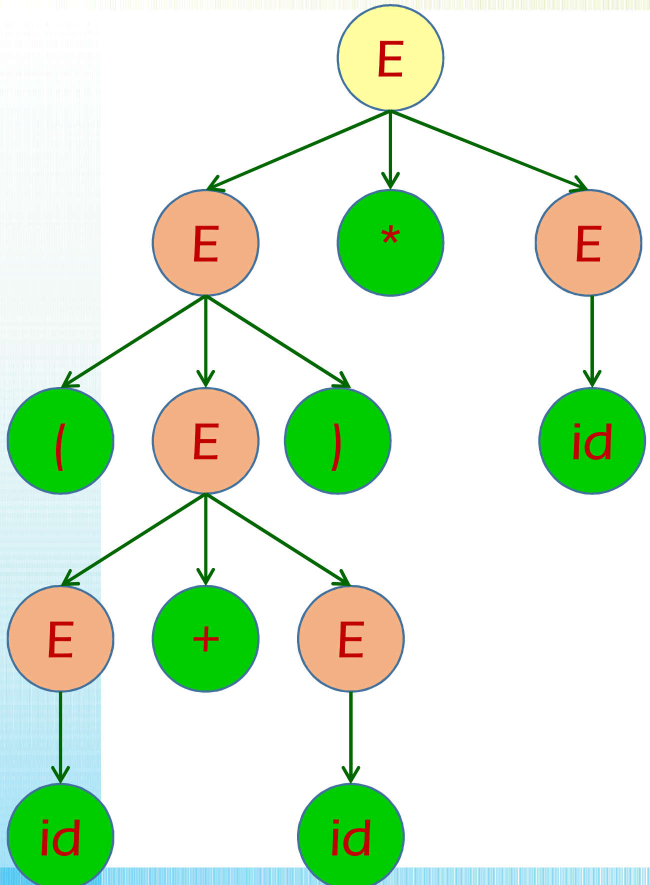
$$\alpha_i = X_1 X_2 \ldots X_{j-1} Y_1 Y_2 \ldots Y_m X_{j+1} X_{j+2} \ldots X_k$$



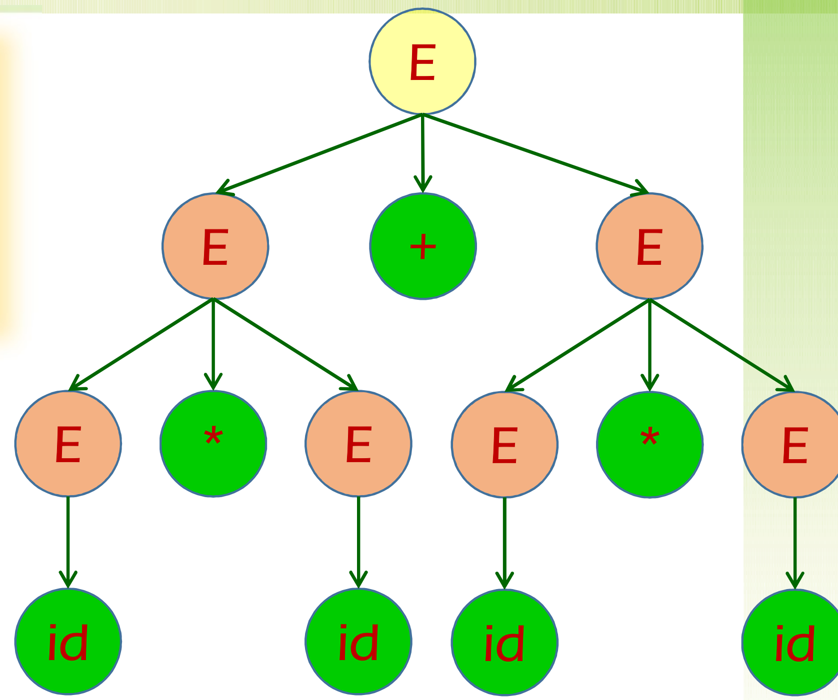
Parse Tree Examples

CFG

E → E + E
E → E * E
E → (E)
E → id



(id + id) * id



id * id + id * id

Root Node
Internal Node
Leaf Node

Ambiguity

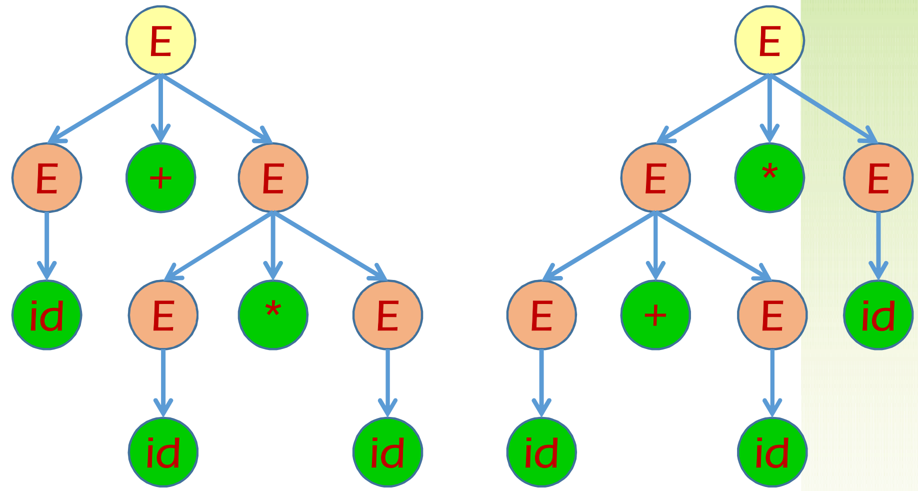
- A grammar that produces more than one parse tree for the same sentence is said to be ambiguous grammar
- Any sentence with more than one parse tree is said to be ambiguous sentence
- Ambiguous sentences have more than one left most derivation or more than one right most derivation
- The grammar should be made unambiguous, to uniquely determine which parse tree to select for a sentence
- Otherwise, disambiguating rules, that throw away undesirable parse trees, leaving only one tree for each sentence are necessary

CFG

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

Sentence

id + id * id



- The sentence id + id * id has two parse trees
- Hence the sentence is ambiguous
- The grammar becomes ambiguous

Removing Ambiguity

Undecidability of Ambiguity

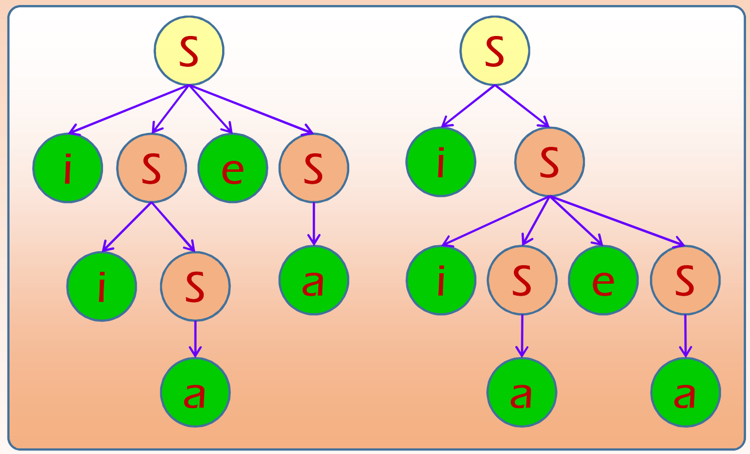
- Though removing ambiguity is needed for parser construction, there is no algorithm for ambiguity removal
- Finding a CFG is ambiguous or not is undecidable
- There are languages without unambiguous grammars for recognition and such languages are called inherently ambiguous

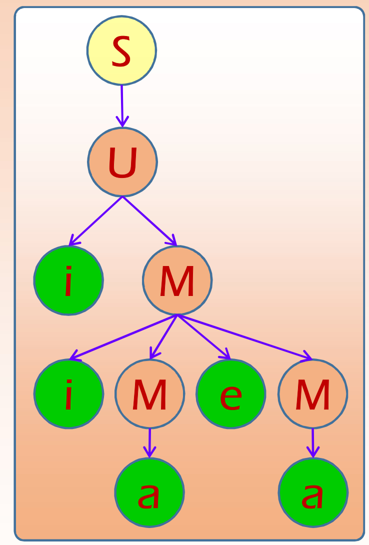
  $L = \{a^n b^n c^m d^m\} \cup \{a^n b^m c^m d^n\},\ m, n \geq 1$
- To remove ambiguity, find the productions which cause ambiguity by generating the ambiguous sentences
- Rewrite those productions, such that the new list of productions do not generate ambiguous sentences

Example

The CFG for "if" statement S → iS | iSeS | a is ambiguous as the sentence "iiaea" has two parse trees

This is due to the "else" and hence the grammar is called as dangling-else grammar





After rewriting the grammar with MATCHED-ELSE and UNMATCHED-ELSE productions, the new grammar becomes unambiguous

S → M | U   M → iMeM | a   U → iM | iMeU

Unambiguating Expression Grammar

Expression Grammar Ambiguous

E → E + E | E – E	1
E → E * E | E / E	2
E → E ^ E	3
E → (E) | I	4
I → L | ID | IL	5
D → 0 | 1 | 2	6
L → a | b | c	7

Ambiguity Removal Steps

- F denotes any expression that cannot be broken apart by any operator
- P denotes any expression that can be broken only by exponentiation operator, but not by others
- T denotes any expression that can be broken only by multiplication or division and exponentiation
- E denotes all expressions and can be broken by all the operators
- The order of the expressions that can be broken by F, P, T, E denote the priority of the operations
- The symbol on left hand side of the production appears either on the left end or right end of the right hand side, denoting the associativity

Expression Grammar Unambiguous

F → (E) | I	4
P → F ^ P | F	3
T → T * P | T / P | P	2
E → E + T | E – T | T	1
I → L | ID | IL	5
D → 0 | 1 | 2	6
L → a | b | c	7

Left Recursion

Left Recursive Grammar

- A grammar is left recursive if it has a non terminal A such that there is a derivation $A \stackrel{*}{\Rightarrow} A\alpha$ for some string α
- A parser constructing parse tree in top-down approach cannot handle left-recursive grammars, so a transformation is needed to eliminate left recursion
- Productions of the form $A \rightarrow A\alpha | \beta$ are rewritten as

  $A \rightarrow \beta A'$

  $A' \rightarrow \alpha A' | \epsilon$

- In general, when there are more left recursive rules for any variable, the rules will be of the form

  $A \rightarrow A\alpha_1 | A\alpha_2 \dots | A\alpha_m | \beta_1 | \beta_2 \dots | \beta_n$

  and they are rewritten as

  $A \rightarrow \beta_1 A' | \beta_2 A' \dots | \beta_n A'$

  $A' \rightarrow \alpha_1 A' | \alpha_2 A' \dots | \alpha_m A' | \epsilon$

Algorithm for Left Recursion Removal

Left Recursion Removal

Arrange the non terminals in some order $A_1, A_2, \dots, A_n$

for i = 1 to n do

for j = 1 to i-1 do

Substitution

Replace each production of the form $A_i \rightarrow A_j\gamma$ by the productions $A_i \rightarrow \delta_1\gamma | \delta_2\gamma | \dots | \delta_k\gamma$, where $A_j \rightarrow \delta_1 | \delta_2 | \dots | \delta_k$ are all current A_j productions

Eliminate the immediate left recursion among the A_i-productions

Example

Grammar with Left Recursion

$S \rightarrow A a \mid b$

$A \rightarrow A c \mid S d \mid \epsilon$

Grammar After Substitution

$S \rightarrow A a \mid b$

$A \rightarrow A c \mid A a d \mid b d \mid \epsilon$

Grammar without Left Recursion

$S \rightarrow A a \mid b$

$A \rightarrow b d A' \mid A'$

$A' \rightarrow c A' \mid a d A' \mid \epsilon$

Left Recursion - Examples

Grammar with Left Recursion

E → E + T | T
T → T* F | F
F → (E) | id

Grammar without Left Recursion

E → TE′
E′ → +TE′ | ∈
T → FT′
T′ → *FT′ | ∈
F → (E) | id

Grammar with Left Recursion

S → AA | 0
A → SS | 1

Grammar After Substitution

S → AA | 0
A → AAS | 0S | 1

Grammar without Left Recursion

S → AA | 0
A → 0SA′ | 1A′
A′ → ASA′ | ∈

Grammar with Left Recursion

E → EE+ | EE- | EE* | id

Grammar without Left Recursion

E → idE′
E′ → E+E′ | E-E′ | E*E′ | ∈

Grammar with Left Recursion

L → (T) | id
T → T,L | L

Grammar without Left Recursion

L → (T) | id
T → LT′
T′ → ,LT′ | ∈

Grammar with Left Recursion

I → IL | ID | L
L → a | b | c
D → 1 | 2 | 3

Grammar without Left Recursion

I → LI′
I′ → LI′ | DI′ | ∈
L → a | b | c
D → 1 | 2 | 3

Left Factoring

Left factoring in productions

- Left factoring is the problem produced when there exists common prefixes in the various alternative productions for some non terminal
- Consider the production $A \rightarrow \alpha\beta \,/\, \alpha\gamma$ where α is a common prefix in the productions of A
- Left factoring creates ambiguity in selecting the productions in top down construction of parse trees
- To avoid this, the productions are rewritten as

$$A \rightarrow \alpha A'$$

$$A' \rightarrow \beta \,/\, \gamma$$

Algorithm for Left Recursion Removal

1. For each nonterminal A, find the longest prefix α common to two or more of its alternatives
2. If $\alpha \neq \epsilon$, then replace all the A-productions
   $A \rightarrow \alpha\beta_1 \mid \alpha\beta_2 \ldots \mid \alpha\beta_m \mid \gamma$ by
   $$A \rightarrow \alpha A' \mid \gamma$$
   $$A' \rightarrow \beta_1 \mid \beta_2 \ldots \mid \beta_m$$
   where γ represents all alternatives that do not begin with α
3. Repeatedly apply this transformation until no two alternatives for a nonterminal have a common prefix

Example

Grammar with Left Factoring

$S \rightarrow i\,C\,t\,S \mid i\,C\,t\,S\,e\,S \mid a$

$C \rightarrow b$

Grammar without Left Factoring

$S \rightarrow i\,C\,t\,S\,S' \mid a$

$S' \rightarrow \epsilon \mid e\,S$

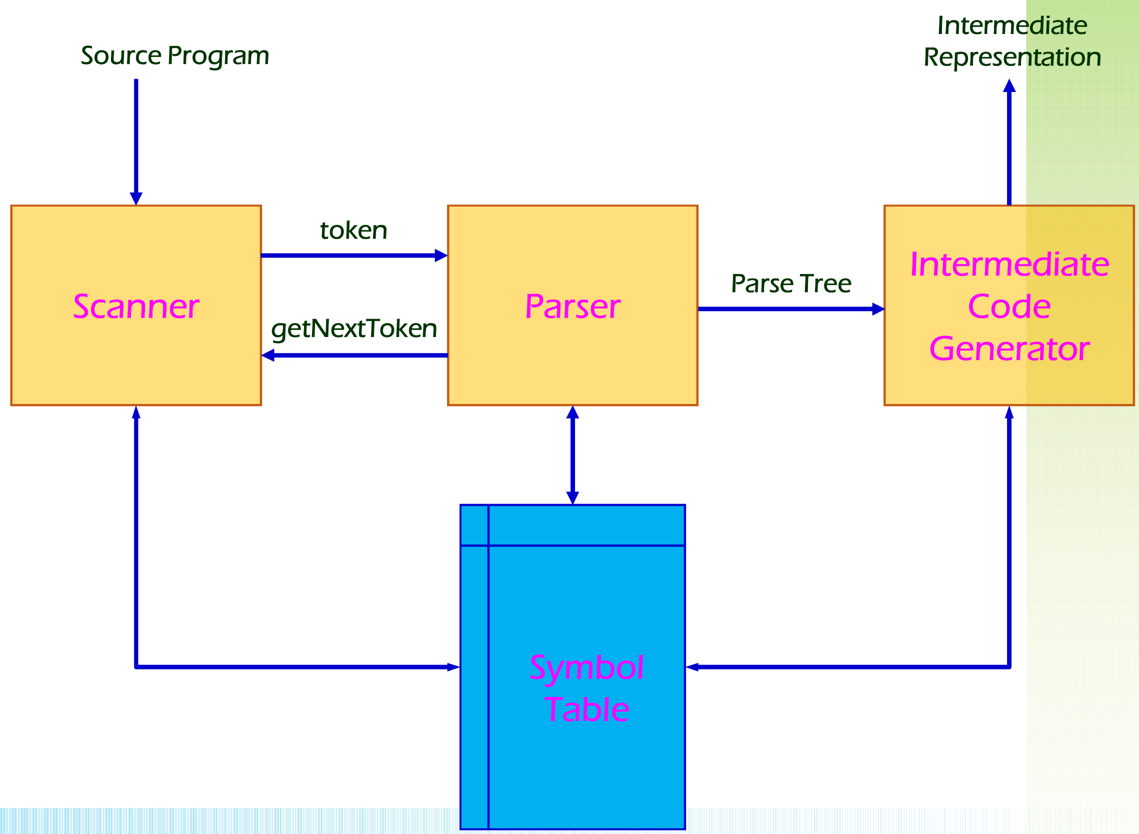
$C \rightarrow b$

Syntax Analyzer

Creation of Parsing Table from the CFG and Creation of Parse Tree while parsing

Role of the Parser

- A parser for Grammar G is a program that takes a string 'w' as input and produces a parse tree for the string w, if w is a sentence of G or an error message indicating w
- Parser reads a string of tokens from the scanner and verifies that the string of token names can be generated by the grammar for the source language
- Also, the parser reports syntax errors if any and recover from commonly occurring errors to continue processing the remainder of the program
- For the well-formed programs, the parser constructs a parse tree and passes it to the rest of the compiler for further processing
- The parse tree construction may be done in either top to bottom manner or bottom to top manner



Types of Parsers