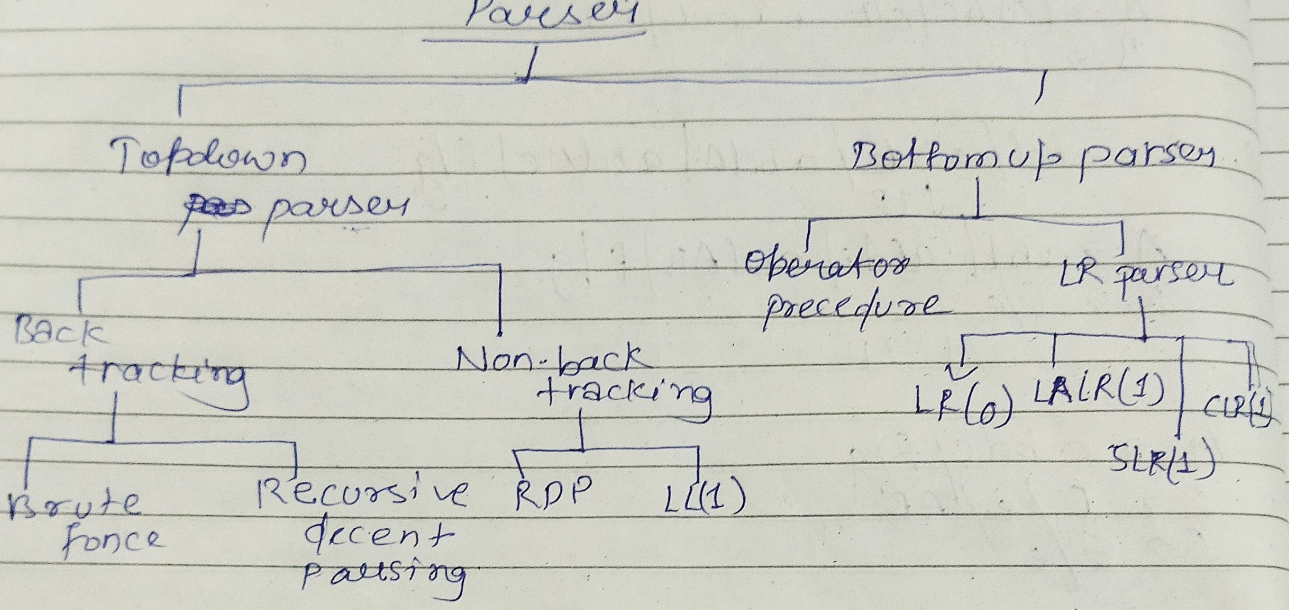
PARSER

⇒ Parser is a peace of Software that can perform Syntax Analysis



□ Top down parser

→ purpose of parsers:- check wheather this word (w) belongs language generated by a grammar or not.

→ we start with starting Symbol (s) and gradually move to down until we check the given word is belong to the language generated by grammar or not.

→ For Top-down parser we want parser free from left recursion do not have unambigous grammar

Ambigous and unambigous grammar

→ Ambigous grammar

* An ambigous grammar is a grammar, where exist more than one derivation for any word, that belongs to the language generated by the grammar ($W \in L(G)$)

** Ambigous means → Not sure/confusion

→ Unambigous grammar

* where exist only one derivation for any word that belong to the language generated by the grammar ($W \in L(G)$)

PROPERTIES

(i) for unambigous grammar left most derivation is equal to Right most derivation (LMDT = RMDT) for ambigous you have different (LMDT ≠ RMDT)

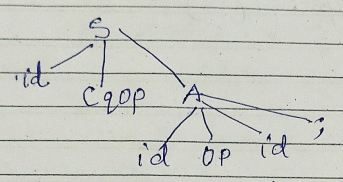
(ii) If grammar is both left recursive and Right recursive then this grammar is ambigous

(iii) Ambigous at CFG (context free grammar) is undecidable : means these does not exist an Algorith is an algorithm to determine

Parse tree :- Parse tree is a pictorial representation of the process ~~to~~ of deriving w(word) from this grammar.

Ex :- Take above eq (i) which is derived from the grammar w.

S → id eq op id op id;



(parse tree)

Classification of CFGr (context free Grammar)

Recursive CFGr | NON-Recursive CFGr

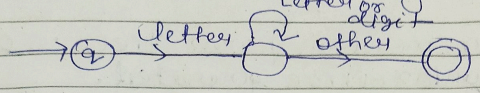
Recursive CFGr	NON-Recursive CFGr
If there exist at least one production having some variable on L.H.S and R.H.S.	If there does not exist same variable on LHS & R.H.S. then it is non-Recursive CFGr.
Ex :- S → Sa/b	Ex :- S → AaB
:- S → aS/b	A → b/C
:- S → aSb/E	B → c/d
	S → AB/BA
	A → aB/b
	B → b/c

Lexeme :- A Lexeme is a sequence of characters in the source program that is matched by pattern by pattern for a token

Lexical analyzer design :-

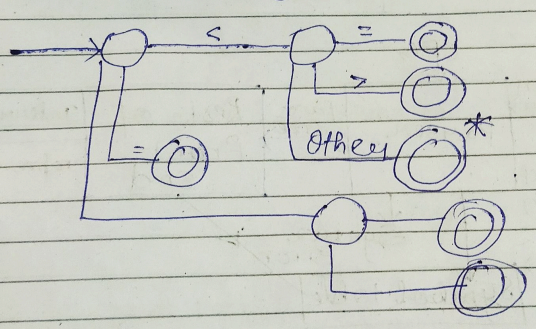
specification of token → Regular Expression
Recognizing tokens → finite automata

① Finite Automata for keyword and identifiers :-



R.E : letter (letter + digits)*

Transition diagram for operator :



Tokens Ex :-

1. main ( )	→ 3
2. {	— 1
3. int x, y;	— 5
4. fl /* gate, */ oat z;	— 9
5. x = /* exam */ 10;	— 4
6. y = 20;	— 4
7 }	— 1
	22

- int (keyword)
- x (identifier)
- = (Assignment operator)
- 10 (literal)
- ; (Delimiter)

Patterns

→ A pattern is a rule or a regular expression that defines the structure of a token in the source code.

→ The lexical analyzer uses these patterns to match sequences of characters (Lexemes) and identify them as tokens.

Input buffering

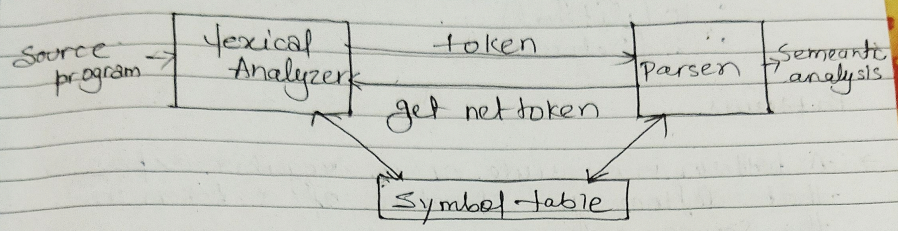
→ Input buffering is a technique used in lexical analysis to efficiency read and process the source code from an input stream.

→ It minimize the number of interactions with the input file and optimizes the performance of the lexical analyzer.

Two-buffer Scheme

A common approach to input buffering is the two-buffer scheme. Where the input is divided into two buffers of Equal Size.

The Role of lexical analysis



(i) lexical analyser divides the given program into meaningful word which is known as tokens.

(ii) Tokens are generally identifier, keywords operators, constant and special symbol.

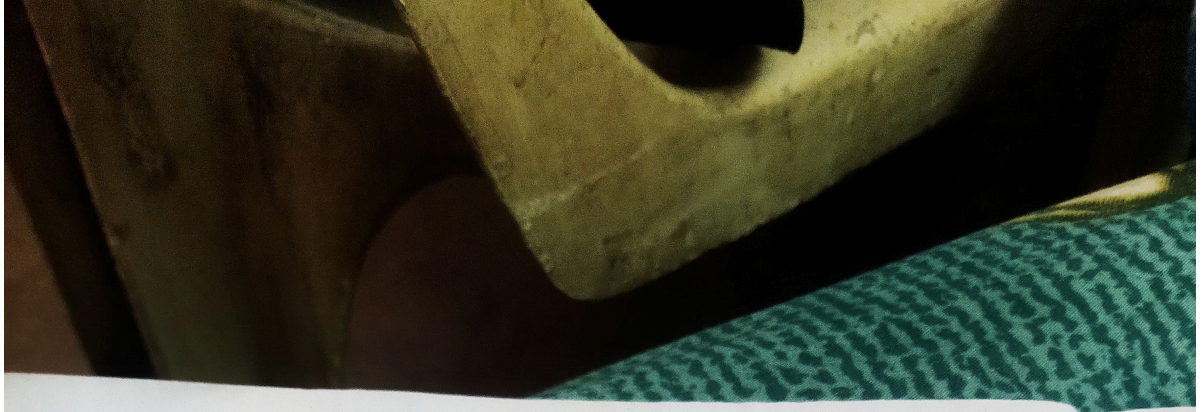
(iii) lexical Analyzer eliminate the comment lines from the given program.

(iv) L.A help in giving error massege by providing row no & column no.

Tokens :-

A token is a sequence of characters from the source code that matches a predefined pattern and is related as a single logical unit during lexical analysis. Tokens are the building blocks of a program's syntax.

Ex :- int x = 10;



code generation phase:-

1. It is a program that translates optimized intermediate code into assembly language or (target code).

Symbol table:- It is a data structure that contains information about identifiers and constants present in the program.

Error Handles:- If any phase of compiler detects error then It is stored in error handler

front end

front end comprises of phases which are dependent on the input (source language) and

Seg 5

Semantic Analysis phase:-

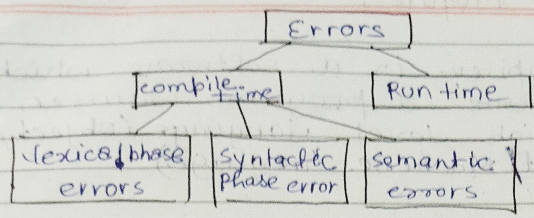
1. It is a program that take parse tree as input and produced annoted parse tree as output.
2. It also detects semantic errors present in the program. (type checking)
3. Syntax directed Translation is used.

Intermediate code generation phase:-

1. It is a program that translates high level code into intermediate code.
2. Advantages of generating intermediate code is to perform optimization in simple way.
3. Syntax directed Translation is used

Code optimization phase:-

1. It is a program that reduces time and space required by the target machine by removing some unnecessary code.
2. There are two type of optimization performed by compilers known as machine independent optimization and machine dependent optimization



Infinite Recursive calls;
→ Idefine by zero
→ Infinite loop
→ wild pointers
→ Segementation

lexical Analysis phase:-

1. It is a program that takes high level language as input and produces token as output.
2. It also detects lexical errors present in the program.
3. Regular expression & finite automata used to design.

Syntax Analysis phase:-

1. It is a program that takes token as input and produces parse tree as output
2. It detects syntax error present in the program.
3. context-free grammar & push down automata used.

3) Writing a Grammar

9) LR Parsers

* LR Parsers: left-to-right scan, Right-most derivation.

Types

a) SLR (Simple LR): Basic form; uses follow sets for reductions.

b) LALR (Look-Ahead LR): Combines similar states in SLR to reduce table size.

c) Canonical LR: Most powerful but complex.

* Steps:
  a) Construct a DFA for ~~vosts~~ viable prefixes.
  b) Build a parsing table with states and transitions.

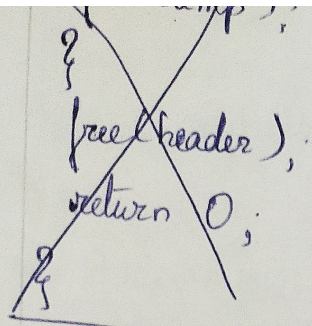
10) Parser Generators (YACC)

* YACC: Yet another compiler compiler.
  * A tool for automatic parser generation.
  * Input: Grammar specification.
  * Output: C code for a parser.
* Applications: Building parsers for custom languages.

11. Error Recovery Strategies

* Goal: continue parsing despite syntax errors.
* Techniques:
  a) Panic mode: Skip tokens until a synchronizing token is found.
  b) Phrase-level Recovery: Replace / insert / delete tokens locally to fix errors.
  c) Error Productions: Extend grammar with rules to handle common errors.
  d) Global correction: use algorithms to find the minimal changes to the input.

Reclaim unused memory.



6. Bottom-up Parsing.

Definition: Constructs a parse tree starting from leaves (terminals) and works upward toward the root.

Techniques:

a) Shift Reducing Parsing: Shift tokens to stack and reduce based on grammar rules.

b) Handles both left-recursion and right-recursion.

7. Handles and viable Prefixes

* Handles: A substring of the input that matches the RHS of a production and can be reduced to a non-terminal.

* viable prefix: A prefix of the input string that can appear on the stack of a shift-reduce parser.

8.) Operator Precedence Parsing

* use case: Simplified parsing for expressions with operators like +, -, *, /.

* Steps

a) Define Operator Precedence and associativity.

b) Create a precedence table.

c) Parse based on precedence relationships.

Module - 3

Syntax Analysis in a compiler translates the sequence of token produced by the lexical analyzer into a structured representation (parse tree) using grammatical rules.

1. The Role of a Parser

* Parser's job: To check the syntax of a program according to the given grammar (usually a context-free Grammar - CFG) and create a parse tree.
* Inputs: Tokens from the lexical analyzer.
* Output: Parse tree or syntax tree.
* Key Tasks:
  a) Syntax checking: Ensure the code follows grammar rules.
  b) Error detection: Identify and report errors when the syntax doesn't match the grammar.
  c) Structure building: Represent program structure for further phases like semantic analysis.

2) Context-free grammars (CFG)

* Definition: A set of production rules used to describe the syntax of a language.
  * Components:
    * Terminals (T): Basic symbols (like keywords, operators, etc).
    * Non-terminals (N): Variable that represent syntactic constructs.
    * Start Symbol (S): The root of the grammar.
    * Production Rules (P): Rules defining how terminals and non-terminals combine.

Example: Grammar for arithmetic expressions:

$E \rightarrow E + T \mid T$
$T \rightarrow T * F \mid F$
$F \rightarrow (E) \mid id.$

(Formerly West Bengal University of Technology)

Syllabus for B. Tech in Computer Science & Engineering

(Applicable from the academic session 2020-2021)

6	Develop algorithms to generate code for a target machine

2.) Context Start

5. Recognition of Tokens

* The lexical analyzer uses finite automata to recognize tokens:
* Deterministic Finite Automata (DFA): A simple machine with fixed states that determines if a lexeme matches a pattern.
* Non-Deterministic Finite Automata (NFA): Similar to DFA but allows multiple possible states at a time.

6. Conversion from Regular Expressions to Automata

* Regular Expression to NFA:
  * NFAs are built step by step from regular expressions using like:
    * Concatenation (e.g., ab → a path for a followed by b)
    * Union (e.g., a|b → Paths for a and b).
    * Kleene Star (e.g., a^* → loops on a).

* NFA to DFA:
  * Convert an NFA into a DFA using a method called subset construction to make it more efficient for token recognition.

7. Design of a lexical analyzer generator (Lex)

* Lex is a tool used to generate lexical analyzers automatically.
* Input: A file containing regular expressions and corresponding action.
* Output: A C program that performs lexical analysis.
* Example: Defining Patterns for keywords, identifiers,

Summary:
lexical analysis is about breaking source code into tokens using pattern and automata. Tools like lex automate this process, making compiler design more efficient.

Sanjana Mathur

...lana Abul Kalam Azad University of Technology, West Bengal
West Bengal University of Tech...

2. Key Concepts

* **Tokens:** Smallest meaningful elements of a program, like. if, while, +, or 123.
* **Patterns:** Descriptions of what a token looks like, often defined using regular expressions.
  Examples: A pattern for an identifier might be $[a-zA-Z][a-zA-Z0-9]^*$.
* **Lexemes:** Actual sequences of characters that match a pattern and form a token. Example: for the pattern $[a-zA-Z][a-zA-Z0-9]^*$, the lexeme could be x, count, or variable1.

3. Input Buffering

* **Need:** Reading the source code character can be slow. Input buffering allows reading in chunks for efficiency.
* **Mechanisms:** It uses two pointers:
  1. Start pointer: Marks the beginning of the current lexeme.
  2. Forward pointer: Moves ahead to find the end of the lexeme.

4. Specification of a Token

Tokens are defined using regular expressions. For example
* A number: $[0-9]+$
* An identifier: $[a-zA-Z_][a-zA-Z0-9_]^*$

Example for compiler design content

```
int a = 10;
float b = 20.5;
float c = a+b;
```

Type Checking:

* a is an integer, and b is a float. The operation a+b is valid because of implicit conversion (int to float).

* Type conversion:
  * a is automatically promoted to float before addition.

Eg:

```
int   a = 10;
Char  b = 'c';
float c = a+b;
```

The type checker will:

1.) Check b is a Char and a is a int.
2.) coerce (force) b into an int
3.) Promote both to float before storing the result in c.

Module - 6

Run-time environments:- This refers to the setup or framework where a program runs after it is compiled. It involves managing memory, variables and execution during program runtime.

Source language issues:- These are challenges that arises due to the nature of the programming language itself, like how it handles memory, variables, and scopes.

1.) Activation trees:- This is a structure that shows how different functions (or procedures) call each other during the execution of a program. It helps visualize the order and relationships of function calls.

2	**Lexical Analysis** [6L] The role of the lexical analyzer, Tokens, Patterns, Lexemes, Input buffering, Specifications of a token, Recognition of a tokens, Finite automata, From a regular expression to an NFA, From a regular expression to NFA, From a regular expression to DFA, Design of a lexical analyzer generator (Lex).	6	
3	**Syntax Analysis** [9L] The role of a parser, Context free grammars, Writing a g... Top-down Parsing, Non-recursive P... (LL), Bott... Operator p... LALR) Pa...	9	

Storage Allocation Strategies

How memory is allocated to variables and s...

2.) **Control Stack:** A stack used to keep track of active function calls. When a function is called, its details are added (Pushed) onto the stack, and when it ends, those details are removed (Popped).

3.) **Scope of Declaration:** Determines where a variable or function is accessible in the program.

- **Local Scope:** Variable is accessible only inside a specific function or block.
- **Global Scope:** variable is accessible throughout the program.

4.) **Binding of Names:** The process of linking a variable or function name to its value or definition. This can happen at:

- Compile-time (Static Binding): Fixed before the program runs.
- Run-time (Dynamic Binding): Decided during program execution.

Storage Organization: This is how memory is managed during the execution of a program.

1.) Subdivision of Run-time memory:

Memory is divided into sections for different purposes:

- Code section: Stores program instructions.
- Static section: Stores global variables.
- Stack: Stores function call details and local variables.
- Heap: Used for dynamic memory allocation (memory requested at run time).

2.) Activation Record: A block of memory created on the stack for each active function call. It stores:

- parameters.
- Local variable
- Return address (where the program goes after the function ends).

3.) Equivalence of type expressions

Type equivalence determines whether two type expressions are the same.

* Structural equivalence: Types are equivalent if they have the same structure, regardless of their names.

Example:

```
type A = {int, float};
type B = {int, float};
```

A and B are structurally equivalent.

* Name equivalence: Types are equivalent only if explicitly declared as the same.

Example:

```
type A = int;
type B = int;
```

A and B are not name-equivalent.

4.) Type conversions : when types don't match but the operation is still valid, the compiler uses type conversion (or type casting) to make them compatible.

* Implicit Type Conversion (coercion): The compiler automatically converts one type to another.

Example:

```
int a = 10;
float b = a; // Implicitly converts int to float
```

* Explict Type conversion (casting) : The programmer explictly specifies the type to convert to.

Example:

```
float a = 10.5;
int b = (int)a; // explictly converts float to int.
```

Error can occur if the conversion loses data or changes meaning, e.g., truncating a float to an int.

Storage Allocation

Module-5

Type Checking: is a critical aspect of compiler design, ensuring that the type of variables, expressions, and functions in a program are used consistently and correctly.

1.) Type Systems: A type system defines the rules that assign a type to each variable, expression, and function in a program. It ensures that operations are performed on compatible data types to prevent type-related errors.

- Static Typing: Types are checked at compile-time (e.g. C, Java).
- Dynamic Typing: Types are checked at runtime (e.g., Python, JavaScript).

A type system has the following properties:

- Soundness: Ensures no type errors occurs during execution.
- Completness: Every valid program can be assigned a type.

2.) Specification of a Simple Type Checker

A type checker ensures that the program adheres to the type system rules. Here's how it works step-by-step:

1.) Syntax tree Traversal: The compiler generates an abstract syntax tree (AST) from the code. The type checker traverses this tree.

2.) Type Rules Application: Each node in the AST is checked against predefined rules.

- For example, in an expression a+b, the type checker verifies that both a and b are numeric types.

3.) Symbol Table use: It uses the symbol table to retrive type information of variables and functions.

4.) Error Reporting: If a type mismatch is found, it reports an error.

Example (in Pseudocode):

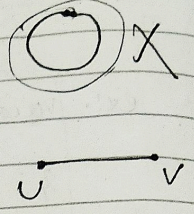
```
if (a+b) {
    Check a.type == int && b.type == int;
} else {
    report_error ("Type mismatch");
}
```

for syntax tree batch system

graph has no cycles no path that start & ends at same vertex.

DAG (Directed Acyclic Graph)

edges in graph have direction (they go from one vertex to another)



Properties :- 1) Topological sorting
Linear ordering of vertices
for every directed edge from vertex u, to v
u comes before v

2) Dependency Resolution
∵ no cycle, you can resolve dependencies in a clear, step-by-step manner.

3) Version control
systems like Git, DAGs used - commit history.

Type Checking

→ ensure types of values used in a prog. match the expected types.
→ imp. concept in prog. lang.
→ ensure - operations you perform is valid
→ prevents errors caused by mismatched types.

1) Static TC → correctness is checked at compile time before prog. is run.
→ Type of every variable, fun: & expression is determined & checked by compiler.
C, C++, Java

ex → int x = 5;
x = "Hello"; // Error.

less flexibility.

2) Dynamic TC → correctness checked at runtime. (prog. execute)
variable can be assigned values of diff. types during execution.

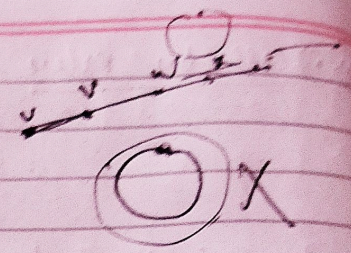
X = 5
X = "Hello" # NO error, X can change type

for syntax tree batch system

graph has no cycles no path that start & ends at same vertex.

DAG (Directed Acyclic Graph)

edges in graph have direction (they go from one vertex to another)



Properties :- 1) Topological sorting

Linear ordering of vertices

for every directed edge from vertex u, to v

u comes before v

2) Dependency Resolution

∵ no cycle, you can resolve dependencies in a clear, step-by-step manner.

3) Version control

systems like Git, DAGs used - (commit history)

③ Compiler

→ used in C, C++, Java, etc.
→ compile programs at one go.
→ CPU utilization is more.
→ compilation time is more.
→ Execution time is less.
→ overall better efficiency.

Interpreter

→ used in Python, Ruby,
→ Interpretation is step-by-step.
→ less
→ Interpretation time is less.
→ Execution time is more
→ less efficient.

④ a) Symbol table

→ Crucial DS used in compilers
→ to store info. about identifiers (variables, functions, objects, etc) in source program.
→ features → Key info.. stored :-
- Name - (of identifier)
- Type - (data type, return type, etc.)
- Scope - (global, local, block level)
- memory location/address
- Size - (based on data type).
- attributes - (parameters for func., modifiers, etc.)

→ function - facilitates semantic checking (type checking)
- supports code generation.
- detects & handles errors.

→ Str. of symbol table → implemented as (hash table / tree-based structure for efficient lookup, insertion, and deletion.

→ Symbol table →

Identifier	Type	Scope	Size
x	int	Global	4 bytes
y	float	"	4
func	function	"	-
a	int	Local	4
z	int	Local	4

⑥ Lexical Analyzer (design?)

→ 1st phase of the compiler.
→ 1° task - read the source code & convert it into a seq. of tokens.
→ Functions - Tokenization breaks the source code into tokens.
ex: if, int, =
→ Removing whitespaces and comments → eliminates irrelevant details like (11, 1st ++)
→ Symbol table maintenance → add identifiers
→ Error detection → identifies invalid tokens and generates appropriate error messages.

→ Working of Lexical Analyzer :-
· Scan the source code character by character.
· uses patterns (regular expression) to identify tokens & categorizes them based on predefined rules.

ex:- input :- int x = 10;
Tokens :- int → keyword, x → identifier, = → assignment operator.
10 → literals, ; → Delimiter.

→ advantages :
Reduces the complexity of parsing.
Simplifies syntax analysis phase (converting source code into tokens).

Source code → [Lexical Analyzer] → Tokens → [Syntax Analyzer]

⑦ Lookahead Operator

→ type of assertion in RE.
→ check if something specific comes next without actually grabbing it as part of the match.

→ Types :-

Positive lookahead (?=) :
ex: cat(?=s)
Find "cat" but only if it is immediately followed by an "s".
Text :- cats are cute.
Match :- It finds cat (bcz. it's followed by "s").
But "s" is not included in the match.

Negative lookahead
ex: cat(?!s)
Find "cat" but only if it's not followed by an "s".
Text :- cat nap is nice.
Match :- It finds cat (bcz. it's not followed by "s".)

Identifier
→ Name given to variables, functions, etc.
→ User-defined? Yes
→ Yes, it can be reused as long as it's not a keyword.
→ Ex. count, sum, total
→ represent user defined entities

Keyword
→ Reserved word with predefined meaning in the lang.
→ NO
→ NO
→ int, for, if
→ Define lang. syntax & behaviour.

(8) Error Handling
→ detecting & responding to errors, occurs during execution of program.
→ Allows program to continue running / fail gracefully, instead of crashing / producing incorrect results.
→ helps ensure that prog. behave as expected, even when unexpected events occur.
→ provides meaningful feedback to users & developers about issues, making it easier to debug.
→ Types of error →
1) Syntax error :- code is written incorrectly & the prog. lang. can't understand it.
ex:- print("Hello world" (missing closing parenthesis)

2) Runtime error :- Occur while the prog. is running
ex:- 5/0 (dividing by zero, accessing a undefined variable).

3) Logic errors :- when prog. runs without crashing, but O/P is incorrect due to flaw/logic
ex- using wrong formula to calculate area of circle.

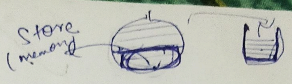
Common Error handling Techniques:-
1. Try & Catch (Exception Handling).
2. Error codes
3. Assertions
4. Logging

(9) Regular Expressions
→ powerful tool for
→ Allows to search, specific patterns
→ used in tasks like
→ Literal character
→ Meta characters →
→ Quantifiers →

Dead state
→ in context of FA
→ refers to a state
or valid state
→ Once the m/c
will not
→ useful? →

(10) LEX → (lexical
YACC → (Yet
→ tool used
→ YACC
→ Takes
and
key features

Thompson's
→ used
→ algo.
→ used
→ steps
→ helps



Register Allocation and Assignment

→ Efficient use of CPU registers → crucial for performance.

→ determine which variable should reside in registers during prog. execution.

→ Assign specific registers to variables.

Techniques :-

(1) Graph colouring → Variable → nodes in graph. node edge variable

if variable interfere :- edge connects 2 nodes

→ assign registers (color) to nodes such that adj. nodes have diff. colors.

(2) Spilling →

If there are more variables than registers, Store some variables in memory.

Challenges :- 1) Optimization ⟷ balance ⟷ compilation speed

2) handles diverse archites. (RISC vs CISC)

3) manages limited register availability.

~~Code Optimization~~

Top Down parsing

→ starts from the root of the parse tree & attempts to construct tree down to the leaves

→ Bottom-up Parsing

MOD - IX

Code Generations

(1) final phase
(2) Intermediate → target m/c code.
(3) focus - producing efficient & correct code.

Issues →

(1) correctness → a+b*c | preserve (semantics) of source program.
Order of oper^n & variable must remain unchanged.

(2) Efficiency → execution time, space, resource → minimize
(↑) memory usage
resource consump^n

(3) Error Handling → manage runtime errors (eg. divide by zero)

(4) Handling complex str. → array, pointer, function calls } generate code effectively.

(5) Register Allocation → efficiently manage limited no. of registers in target m/c.

Simple Code Generator

- Input: Intermediate representation (3-address code)
  Output: Target m/c code / assembly instruc^n
- Steps :- (1) Instruc^n selec^n → each IR statement → m/c instruction.
  [t1 = a+b] → LOAD R1, a; ADD R1, b; STORE R1, t1
  (2) Register Allocation → Assign variables to register while minimizing memory.
  (3) Optimization → Apply peephole optimiz^n to improve efficiency.
  (4) Emit code → Output the final instructions.

Ex:- IR code :-
t1 = a+b
t2 = t1 * c
d = t2

Generated code.
LOAD R1, a
ADD R1, b
STORE R1, t1
MUL R2, c
STORE R2, d

Register Allocation
→ Efficient use of
→ determine wh
should reside in registers du
execution.

Techniques :-
(1) Graph colouri
(2) Spilling

Challenges :- 1) Opt
2)
3)

Top / Dow
→ starts from the roo
attempt to compu

① Three Address code generation (TAC)

→ intermediate representation used in compilers
→ to simplify the process of code generation.
→ breaks down complex expressions into simpler statements, where each statement involves at most 3 operands.
→ each instruction has at most 1 operator and 3 operands (2 source operands : 1 destination)
→ intermediate results are stored in temporary variables (t1, t2, ...)
→ It acts as a bridge b/w HL code and m/c code.

ex:- High-level code :- a = b + c * d;

Step by Step TAC :- t1 = c * d // multiple c and d & store result in t1
t2 = b + t1 // add b and t1, store result in t2
a = t2 // assign t2 to a.

→ components of TAC :-

i) Operators :- Arithmetic (+, -, *, /)
Relational (<, >, <=, >=, ==, !=)
Logical (&&, ||).
Assignment (=)

ii) Operands :- variables (a, b)
constants (10, 3.14)
Temporary Var. (t1, t2).

→ Benefits → simplifies code generation
→ improves Optimization
→ m/c - independent.

YACC → Yet Another Compiler Compiler
→ prog that generates parser for a given grammar.
→ I/P to YACC → context-free grammar. (CFG)
→ automatically assigns no. for token, but can be overridden.
→ generates (LALR(1)) parser.
→ parser generated through this has something to do with syntax definition of prog. lang.

→ Detect Syntax errors.
→ Able to continue parsing after errors have occurred

Left to Right Right most derivation with 1 lookAhead token. (what type of parser YACC produces)

② Quadruples, Triples, Indirec

Aspect	④ Quadruples
Represent:	Each inst: has 4 (Operator, Arg1, Arg2
Fields	1. Operator 2. Argument 1 3. Argument 2 4. Result (Stores name
Ex. code	a = b + c
Operator	+
Arg1	b
Result field	Explicit
Use	Easy to impl (Straightforw handling.

Q> i) Call by value → actual value of argument is passed to the function.
- any changes made to parameter inside the function don't effect the original variable.

ex:
```
void func(int x.){
    x = 10; // not affect the original variable.
}
int main(){
    int a = 5;
    func(a); // a remains 5 after fun^n call.
}
```

ii) Call by Reference → address (reference) of argument is passed to the fun^n.
→ changes → affect original
bcz. both fun^n & caller refer to same memory location.

```
Void func (int &x){
```

iii) Call by ~~Value~~ name → more commonly seen in lang. like ALGOL
→ rarely used in modern prog. lang.
→ arg. not evaluated before fun^n is called.
→ like textual substitution.

```
procedure func(a){
    print(a);
}
begin
    func(2+3); // 2+3 is substituted directly.
end
```

a) {0,1,2}
= (0|1|2)

b) b*ab*ab*

c) (a|b)* (aaa|bbb) (a|b)*

Ram (A+B)
Ram (C+D)
Ram (A+B+C+D)

Q a) Constant Folding:-
Calculating const. expressions during compilation instead of at runtime.

Ex → · Before optimization :- x = 2+3
y = x*4

· After optimization (fold constant) :- x = 5
y = 5*4

Compiler precomputes 2+3, replacing it with 5.

b) Common Sub-expression elimination (SSE)
Identifying & removing duplicate calculations by reusing the result of a previous calculation.

Before optimiz^n. :- t1 = a+b
t2 = a+b
t3 = t1*c

After optim^n. (eliminate duplication) :- t1 = a+b
t3 = t1*c.

c) Loop Unrolling
This expands the body of a loop to reduce the no. of iterations which can make the prog. run faster.

Before Optimiz^n.

```
for (i=0; i<4; i++){
    x[i] = i*2;
}
```

After optim^n.

```
x[0] = 0*2;
x[1] = 1*2;
x[2] = 2*2;
x[3] = 3*2;
```

loop is replaced with direct calculations for each iteration.

d) Loop Jamming
This combines multiple loops into one when they share the same iteration range, reducing overhead.

ex:- Before

```
for (i=0; i<n; i++){
    a[i] = b[i] + c[i];
}
for (i=0; i<n; i++){
    d[i] = e[i] * f[i];
}
```

After (Jam the loops).

```
for (i=0; i<n; i++){
    a[i] = b[i] + c[i];
    d[i] = e[i] * f[i];
}
```

Q) Redundant load/store
→ elements in
data from mem
→ even though the
behaviour.
→ waste time &
→ can be remo
x = a
y = a
↳ redundant.
& reduced
Why? → ① faster exe
② efficient
③ better

Q) Stack
LIFO
Limited to fun^n.
Automatic
Local var., fu
automatic, mana
Size - limited,
Speed - faster
Deallocation →
use case →

(2) Quadruples, Triples, Indirect Triples

Aspect	Quadruples (4)	Triples (3)	Indirect Triples (3)
Represent:	Each inst: has 4 fields (Operator, Arg1, Arg2, Result)	3 fields (Operator, Arg1, Arg2)	uses a table of pointers to triples. (Pointer to Triple Table) ☆☆
Fields	1. Operator 2. Argument 1 3. Argument 2 4. Result (Stores temp. / Variable name)	1. Operator 2. Argument 1 3. Argument 2	1. Pointer to triple index 2. Argument 1 3. Argument 2
Ex. code	a = b + c	a = b + c	a = b + c
Operator	+	+	Pointer to triple of +
Arg 1	b	b	As in Triple.
Result field	Explicit	No result field (implicitly referenced)	uses pointer table to reference triples.
Use	Easy to implement Straightforward result handling.	More compact repres:, harder to manage during optimization.	Best for code reordering & advanced optimization.

a + b) * c

State transitions.

State I_1 → F → id.

State I_2 → T → F.
T → T * . F

State I_3 → T * . F
F → . id

State I_4 → T → T * F.

State I_5 → E → T.
E → E + . T
T → . T * F
T → . F
F → . id

State I_6 → T → F.

State I_7 → E → E + T.

(5) y : (a + b) * c

1. understand the operator's precedence:-
( ) → highest precedence
* → higher than +
+ → lowest precedence.

2.

(6) First

Set of terminals that appear at the beginning of any String derived from non-terminal.

determines which terminal to choose (1st) when parsing.

Symbol:- First(x)

First (ε) → ε

Ex:- If X → ab.
First(x) → a

Follow

Set of terminals that can appear immediately after a non-terminal in any derivation.

determines which terminal can follow a non-terminal in parse.

Follow (X)

Follow (start) → $

If A → αBβ
Follow(B) → First(β-ε) ∪ Follow(β)

ex:- Void fu
{
int
}

(i) > Call by value →

(ii) > Call by Re

(iii) > Call b

Top-Down → Recursive Descent → Predictive parsing

Bottom-up → LR → LR(0), SLR(1), LALR(1), CLR(1)
↓ Shift reduce

③ Top Down Parsing → starts from root of the parse tree
→ attempts to construct the tree down to the leaves
→ parse i/p from L to R
→ derives the leftmost derivation of the string.

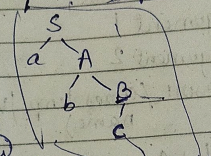
Types

1) Recursive descent Parsing → involves recursion procedure for each non-terminal in the grammar.
→ simple but fail if the grammer has LR or ambiguity

2) Predictive Parsing → A non-recursive form of top-down parsing that uses lookahead symbol to decide which prod^n to use.
Req. gramm. to be LL(1) } no left rec., left factored)
uses a parsing table to guide parsing process.

ex:- S → aA
A → bB | ε
B → c

Parse tree →



Bottom-up Parsing → starts from leaves → attempts to construct the parse tree up to the root
→ parse i/p from L to R
→ describes Rightmost derivation in reverse

Types

1) Shift-Reduce Parsing → uses a stack to store symbols and a pointer to scan the I/p.
→ alternates b/w shift (push i/p to stack) and reduce (replace stack symbols with non-terminal based on prod^n rule)

2) (Left-to-Right) LR Parsing →
LR(0) → basic version with no lookahead.
SLR(1) → (Simple LR) → use lookahead to improve decision making
LALR(1) → (Look-Ahead LR) → a more memory-efficient variant of SLR.
CLR(1) → (Canonical LR) → most powerful but complex.

ex:- S → AB
A → a
B → b

1. Shift a onto the stack.
2. Reduce a to A.
3. Shift b onto the stack
4. Reduce b to B
5. Reduce AB to S.

④ E → E+T | T
T → T*F | F
F → id.

Now, To

E → E+T
F → T+T
T → F
F → id
T → F*
F → id

To constru

① Augmented

E → E+T
T → T*F
F → id

② We add a

E'

③ Initial st

E' → .E
E → .E
E → .T
T → .T
T → .F
F → .id

$S \to CC$ → $\cdot CC, \$$
$C \to cC$ → $\cdot cC, \$$
$C \to d$ → $\cdot d, \$$
I_0

$S' \to S \cdot, \$$
I_1

$S \to C \cdot C, \$$
$C \to \cdot cC, \$$
$C \to \cdot d, \$$
I_2

Thus, I_0 : $S' \to \cdot S, \$$
$S \to \cdot CC, \$$
$C \to \cdot cC, \$$
$C \to \cdot d, \$$

compute GOTO (I_0, S) to create I_1 (move dot over S in S') → GOTO (I_0, S) to create I_2

Thus, I_1 : $S' \to S \cdot, \$$

Similarly, $S \to C \cdot C, \$$

I_2 :- $S \to C \cdot C, \$$
$C \to \cdot cC, \$$
$C \to \cdot d, \$$

GOTO (I_0, C) to create I_3

$C \to C \cdot C, \$$
$C \to \cdot cC, \$$
$C \to \cdot d, \$$
I_3

GOTO (I_0, d) to create I_4

I_4 : $C \to d \cdot, \$$
I_4

Final table of LR(1) ~~parser~~ items.

State	Items
I_0	$S' \to \cdot S, \$$, $S \to \cdot CC, \$$, $C \to \cdot cC, \$$, $C \to \cdot d, \$$
I_1	$S' \to S \cdot, \$$
I_2	$S \to C \cdot C, \$$, $C \to \cdot cC, \$$ $C \to \cdot d, \$$
I_3	$C \to c \cdot C, \$$, $C \to \cdot cC, \$$, $C \to \cdot d, \$$
I_4	$C \to d \cdot, \$$

Q) Peephole Optimization :→ focuses on improving small chunks rather than the entire program.
→ compiler "peeks" at a small part of the code
→ Simple & quick method used to improve small part of the code during compilation.
→ compiler looks at a small "window" (or peephole) of a few instructions at a time & tries to improve them without changing what the code does.
Goals → make the code run faster
→ use less memory
→ Reduce ~~Redunt~~ unnecessary instruc.

S → Aa | b
A → Ac | Sd

Top-down and Bottom-up parsing :-

Remove Left Recursion from
G1 : { S → Aa | b, A → Ac | Sd | }

Formula :- A → Aα | β
A → βA'
A' → αA' | ε

$(x+y)^2 = x^2+2xy+y^2$
$(a+b)^2 =$
$(2+3)^2 =$

Now, S → Aa | b
A → Ac | Sd
A → Sd Ac
A' → Ac

{ given }

Removing Left Recursion
S → bA' , A → dA'
A' → cA' | ε

Q) Syntax-Directed Translation (SDT)
→ SDT is like giving instructions for a recipe, where each step tells how to combine ingredients (data) to make something (O/P).
→ Similarly, SDT is a technique used in CD to associate semantic rules with grammar of a prog. lang.
→ uses Syntax directed definitions. (SDD)
→ Specify tasks like type checking, code generation, intermediate rep! construction.

S - Attribute

i) Info. moves upward in tree, like passing results from child to parent.
ii) Ex- A child tells parent "Here's my answer."
iii) Easier - only one direc. (UP)
iv) Bottom-up parsers
v) Synthesized attributes.
vi) Suitable for LR parsers

L - Attribute

i) Info. moves upward / sideways (Siblings) or downward (from parent to child).
ii) A parent tells child, "Use this info.!" or siblings share notes.
iii) A bit more complex - info can flow in multiple directions.
iv) Top-down parsers
v) Synthesized & inherited attributes.
vi) LL) Parsers

COMPILERS.

What is a Compiler?

A prog. that reads a prog. written in one lang. & translates it into an equivalent prog. in another lang.

I/P lang. → Source lang. O/P lang. → Target lang.

A compiler also reports error present in the source ~~code~~ prog. as a part of its translation process.

Source program → [compiler] → Target prog.
↓
error.

Types of compiler → Single Pass compiler.
↓
Multi-Pass compiler.

The 2 parts of compilation process

Analysis

It takes the I/P source prog. & breaks it into parts. It then creates an interme. represent. of source program.

Synthesis

It takes the intermediate represent. as the I/P. And creates the desired target program.

Cousins of a compiler

Source Prog.
⇓
Preprocessor directives → [Pre PROCESSOR] ← #define #include
⇓
Modified source Program.
⇓
[COMPILER] → MOV R1,R2 JMP
⇓
Target Assembly Program
⇓
[ASSEMBLER] → 0 and 1
⇓
Relocatable M/C code

In addn. to compiler, many other prog. are needed to create absolute m/c code.

PREPROCESSOR → macro expansion
→ file inclusion

ASSEMBLER → takes assembly code that has been generated by compiler & converts it into relocatable m/c code.

LINKER / LOADER → takes the entire prog. to main memory for execution
↓
library routines, relocatable object files are linked

Module-2

Lexical Analysis

→ Fundamental phase of the compilation process where the source code

28/12/22

(1) Overview of Lang. Processing System

HLL
↓
Pre-processor
↓ HLL
Compiler
↓ Assembly lang.
Assembler
↓ Relocatable M/c code.
Loader / Linker
↓
Absolute M/c code

(Cousins of compiler)

Compiler	Interpreter
→ C, C++,	→ Python.
→ Compile at one run.	→ Interpret step-by-step.
→ CPU utilization is more.	→ CPU utilization is less.
→ Compilation time is more.	→ Interpretation time is less.
→ Execution time is less.	→ more.
→ Overall better efficiency	→ overall less efficient

(2) Phases of compiler.

HLL
↓
Lexical Analysis ← Breaks I/p into tokens
↓ Tokens
Syntax Analysis ← verifies grammar
↓ Parse Tree
Semantic Analysis ← Checks logical consistency
↓ SDT (Syntax Directed Translation)
Intermediate code generation ← converts to intermediate form
↓ 3-address code
Code optimization ← improves efficiency
↓
Target code generation ← Produces m/c code
↓
LLL (Machine code)

Symbol Table → is a DS that stores various identifiers and their attributes

Error Handling

Frontend (Analysis phase)
- m/c independent
- language dependent

Backend (Synthesis phase)
- m/c dependent
- lang. independent

int i n t
int main
sentence
37