Compiler Design CD

Q. Various Compiler Applications
- used for translation of programs
- used with S/w productivity tools
- helps implementation of high-level language

Date: __/__/____

PHASES OF COMPILER

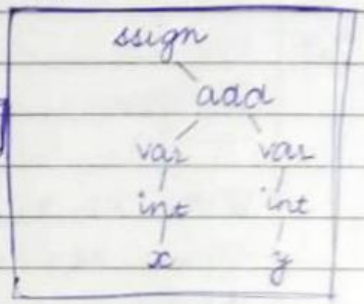
Lexical
↓ tokens
Syntax
↓
Semantic
↓
Intermediate code generation
↓
Target code generation
↓
code optimization [local & global optimization]
↓
Symbol Table Management
↓
Error handling & recovery

They all work hand-in-hand & are interlinked

errors v/s warning

```
int x=10;
int y=20;
z= x+y;
```

Parse Tree



Lexical Analysis (Scanning)
- Interface of the compiler to outside world
- scans input source program, identifies valid words of language in it
- Removes cosmetics like extra white spaces, comments etc from the program
- expands user-defined macros
- Reports presence of user-defined word.
- May perform case-conversion.
- generates a sequence of integers, called tokens to be passed to the syntax analysis phase - later phase need not worry about program text.
- generally implemented as finite automata.

Syntax Analysis
- Takes words/tokens from lexical analyzer.
- Works hand-in-hand with lexical analyzer
- Checks syntactic (grammatical) correctness

- Identifies sequence of grammar rules to derive input program from the start symbol.
- A parse tree is constructed.
- Error messages are flashed for syntactically incorrect program.

Date __/__/____

Semantic analysis

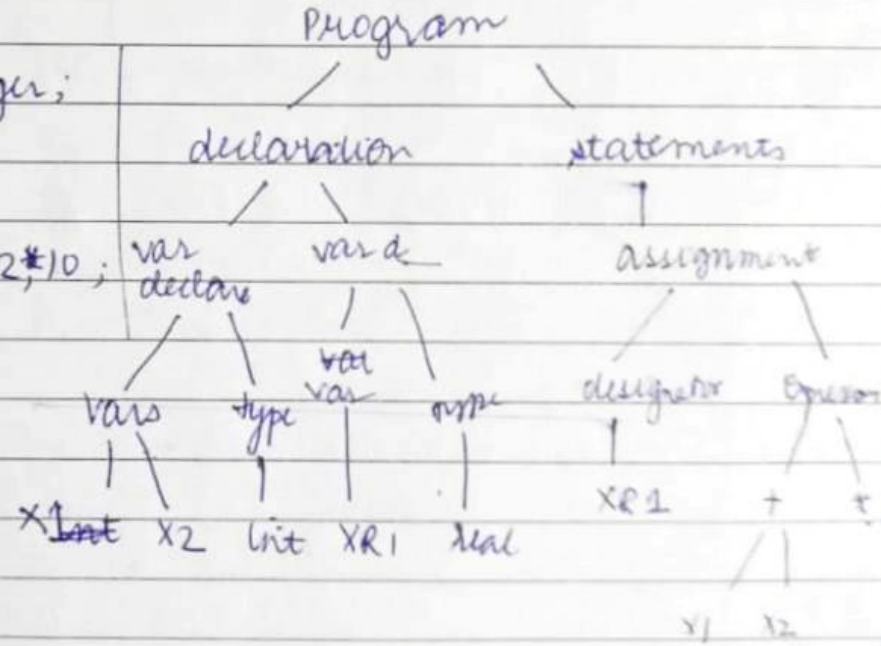
- Semantics of a program is dependent on language
- A common check is for types of variables and expressions.
- Applicability of operators to operands.
- Scope rules of language are applied to determine types - static scope & dynamic scope.

syntax

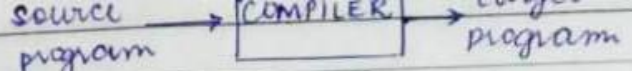
Program

```
var X1, X2: integer;
var XR1 real;
begin
    XR1 = X1 + X2*10;
end
```

PARSE TREE

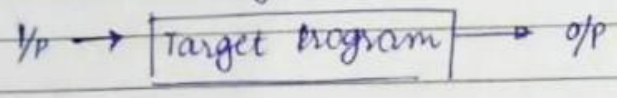


Intermediate code generation



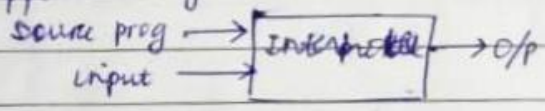
Compiler

It's a program that can read a program in 1 language (source language) and translate it into equivalent program in another language (target language). The main role of the compiler is to report any errors in the source program that it detects during the translation process.



Interpreter (line by line)

An interpreter, instead of producing a target program as a translation, it directly executes the operations in source program supplied by the user.



Assembler, Linker, Loader.

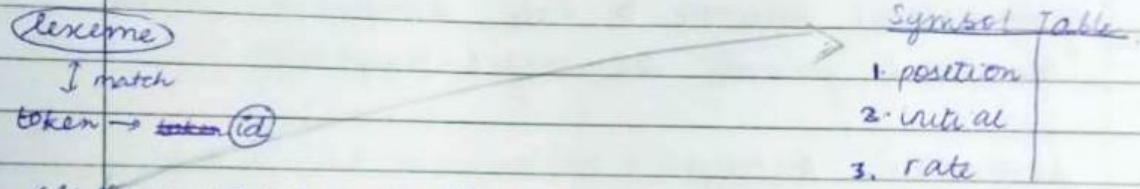
Lexical Analysis (scanning) # FIRST PHASE OF CD

- Lexical analyzer reads the string of characters making up the source program and groups the characters into meaningful sequences, called lexemes. For each lexeme, lexical analyzer produces as o/p a token of the form <token_name, token attr> where

Date: __/__/____

1. Token_name is an abstract symbol i.e. used during syntax analysis
2. Attribute value points to an entry in the symbol table for this token.

e.g. position = initial + rate * 60



A) Position is a lexeme that would be mapped into a token <id, 1> where id is an abstract symbol for identifier and 1 points to the symbol table entry for position.

B) The assignment symbol = is a lexeme that is mapped into the token = .

Syntax Analysis (Parsing) (IInd Phase of CD)

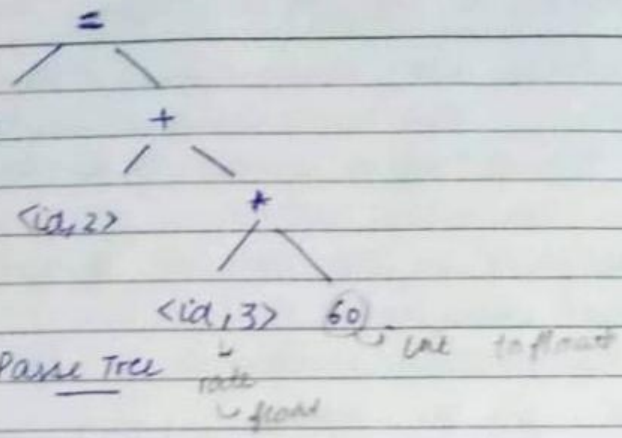
- This phase uses the output of the lexical analysis phase (tokens) to create a tree-like intermediate representation that depicts the grammatical structure of the token stream (syntax tree / parse tree) in which each interior node represents an operation and its children represent the arguments of the operation.

<id, 1>, <=>, <id, 2>, <id, 3>, <*>, <60>

↓ syntax / parse tree

<id,1> = <id,2> + <id,3> * 60

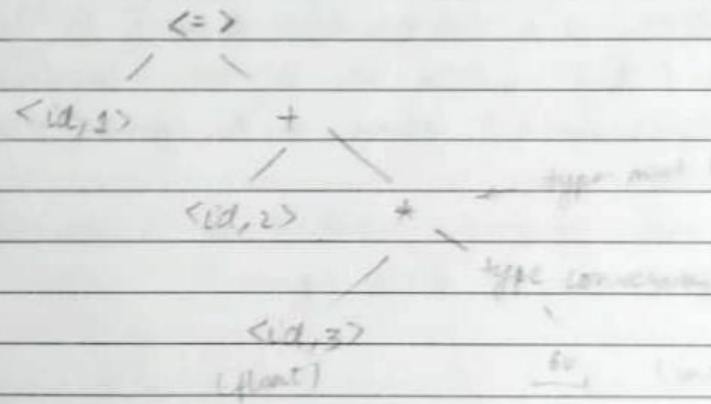
Date __/__/____



The subsequent phases of the compiler use the grammatical structure to help analyse the source program & generate the target program.

Semantic Analysis

It will add type conversion in parse tree



- It uses the syntax tree and the information in the symbol table to check the source program for semantic consistency with the language definition
- Important part of this phase is typechecking. where the compiler checks that each operator has matching operands e.g.
if in C languages array index must be an integer so the compiler will generate an error

If a floating pt. number is used as array index.

ii) Division by zero

Date: __/__/____

Intermediate code generation

```
t1 = int to float (60)
t2 = t1 * id3
t3 = t2 + id2
t4 = t3

id1 = t3
```

3 address codes will be generated. At most only 3 operands

Triples, quadraples, etc

Code optimization

```
t2 = int to float (60) * id3
id1 = t2 + id2
```

Target code generation

Code generator

It takes as input an intermediate representation of the source program & maps it into a target language.

```
LDF   R2, id3
MULF  R2, R2, #60.0
LDF   R1, id2
ADDF  R1, R1, R2
STR   id1, R1
```

Symbol table management

Unit-2 Lexical Analysis

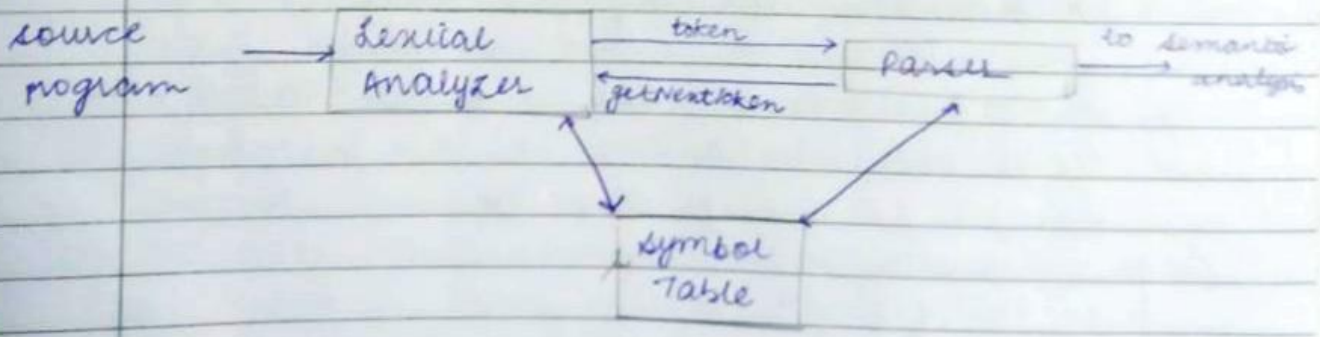
- Role of Lexical Analyzer - Tokens, Patterns & lexemes
- Attributes for Tokens - Input Buffering - Lex specifications
- Specification of Tokens - ~~Lexical~~ Recognition of tokens
- Lexical Analyzer Generator - LEX - Finite Automata.

Lexical Analysis

One can produce a lexical analyzer automatically by specifying the lexeme patterns to a lexical analyzer generator and compiling those patterns into code that functions as a lexical analyzer.

Role of Lexical Analyzer: the main task is to read the I/P characters of the source program, group them into lexemes and produce as output a sequence of tokens for each lexeme in the source program. The stream of tokens is sent to ~~a~~ parser for syntax analysis. It is common for the lexical analyzer to interact with the symbol table as well.

When the lexical analyzer discovers a lexeme constituting an identifier, it needs to enter that lexeme into the symbol table.



getNextToken command causes the lexical Analyzer to read characters from its input until it can identify the next lexeme & produce for it the next token, which it returns to the Parser.

Various tasks of Lexical Analyzer

- Stripping out comments and white space (blank, new line, tab, ~~etc~~ & other characters which are used to separate tokens in I/P.
- ~~code~~ Correlating error messages generated by the compiler with the source program
  e.g. it ~~may~~ keep track of the number of newline characters seen, so as to associate each error with line number.
- If source program uses a macro pre-processor, the expansion of macros is performed by lexical analyzer.
- We can divide the process of lexical analyzer into 2 processes:
  i) scanning ii) Lexical Analysis

Question: Discuss lexical analysis v/s Parsing.

#2 Tokens: It is a pair consisting of a token name and an optional attribute value. The token name is an abstract symbol representing a kind of lexical unit.
e.g. a particular keyword or a sequence of I/P characters denoting an identifier.

<id, 1>
↑ identifier ↑ attribute value

KEYWORD → return → NFA

Pattern
It is a description of the form that the lexemes of a token may take. e.g. in Date __/__/__
case of a keyword as a token, the pattern is just the sequence of characters that form the keyword.

For identifiers and some other tokens, the pattern is a more complex structure i.e. matched by many strings.

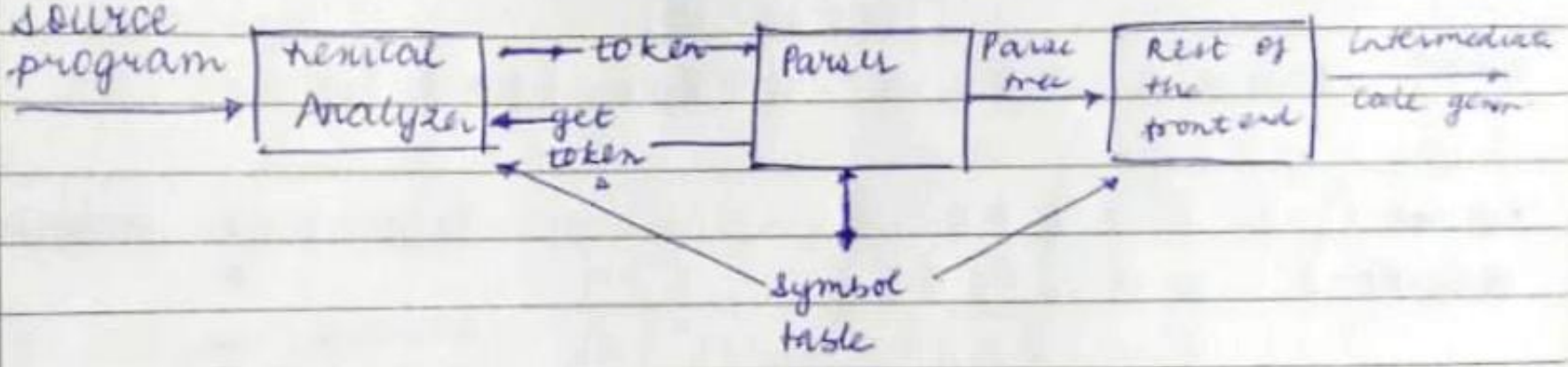
position = initial + rate * 60
e.g. <id, 1> <=> <id, 2> <+> <id, 3> <*> <6
(lexeme)
↳ SYMBOL TABLE

new pattern has match like token angle>

A lexeme or ~~~~> is a sequence of characters in the source program that matches the pattern for the token and is identified by the lexical analyzer as an instance of that token

Attributes for Tokens
When more than 1 lexeme can match a pattern, the lexical analyzer must provide the subsequent compiler phases, additional information about the particular lexeme that matched
e.g. the pattern for token number matches both 0 and 1, so it is extremely crucial for the code generator to know which lexeme was found in the source program.
∴ lexical analyzer not only returns token name to the parser, but also an attribute value that describes the lexeme represented by the token.
The token name influences parsing decisions, while the attribute value influences translation of tokens after the parsing
For certain pairs especially operators, punctuations & keywords, there is no need for an attribute value. Information about an identifier, e.g. its lexeme, its type and the location at which it is first found is kept in the symbol table.

It builds parse tree from the top (root) to the bottom.
While bottom-up starts from the leaves and works its way towards the root
I/P to the parser is scanned from left to right & that too 1 symbol at a time



CFG (content-free grammars)

$A \to aA$

In this productions consist of non-terminal (LHS) on RHS, it consists of zero or more terminals & non-terminals

Derivations:
- Left most Derivation → left-most always replaced
- Right most Derivation → right-most always replaced

Left most variable at each step

sentential form
string

intermediate steps

can contain both terminals & variables

intermediate

$E \to E+T \mid E-T \mid T$
$T \to T*F \mid T \mid F \mid F$
$F \to (E) \mid id$

at every step, you will expand only 1 non-terminal [variable]

#4 Input Buffering

- In this phase of lexical analysis, we often have to look for 1 or more characters beyond the next lexeme before we can be sure that we have the right lexeme.
- There may be situations where we need to look at atleast 1 additional character ahead e.g. to identify the
* end of an identifier, we can't be sure until we see a character i.e. not a letter or a digit
- In C, single character operators like -, =, < can also be the beginning of 2 character operators like ==, <=, >=

4b. ⇒ Buffer Pairs (eof)

P1 → lexeme begin → notes the beginning of the lexeme
P2 → forward → scans

#5 Sentinels

Date: __/__/____

#6 Specification of Tokens

- Strings & Languages - Operations on Languages - Regular Expression
- Regular definitions.

Regular exp : $0^*(11)1^*(0)$ ← end with zero (×2 consecutive)

$0^+ = \{0^* - \Lambda\}$ (atleast 1 time use keyen)

$0^* = \{\Lambda, 0, 00, 000, \ldots\}$

minimum length ~~string~~ 110	not accepted: 1010
01110	1111
accepted: 0011110	0101
00011110	
0011110