Date: 23/09/2021
Page No: 1

Pre-requisite Lecture note for Theory of Compiler Design:-

- Languages - → the alphabets of M/C language is 0, 1

1. Low level / Machine level (0, 1) signal level.

- program look of Machine language :-

```
00001101111001101
00101111011000110
11010011110001110
01101101101111011
  :       :       :
```

ADD → 0011
NUL → 0100
ADD A, B → m/c code for operation
→ are operands.

2. Middle level language - ~~0, 1 Binary~~

↳ Assembly language → Hexadecimal

| A | | 5 | | C |

→ nemonic-code
Addition → ADD
multiplication → NUL
~~program~~ load → LOAD (Load value in Registers)

→ program look of Assembly lang

```
LOAD  B
LOAD  A
ADD   A, B
MUL   B, C
PRINT C
```

NOTE :- Assembler → convert assembly language program into → m/c language.

3. High level language :-
example :- Basic, Pascal, C, C++, Java.
- English like code :-

Program Look of HLL :→

```
# ....
# ....
void main()
{ float ....
    . . . .
    . . . .
    . . . .
    . . . .
}
```

- Compiler :→ convert High level language into machine language.

23/09/2021

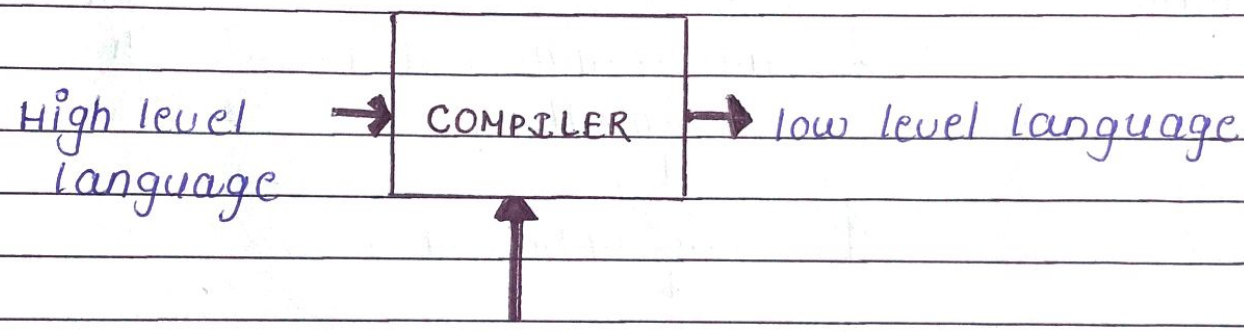
Date:
Page No: 3

THEORY OF COMPILER DESIGN

- Compiler Design :- Analysis + synthesis

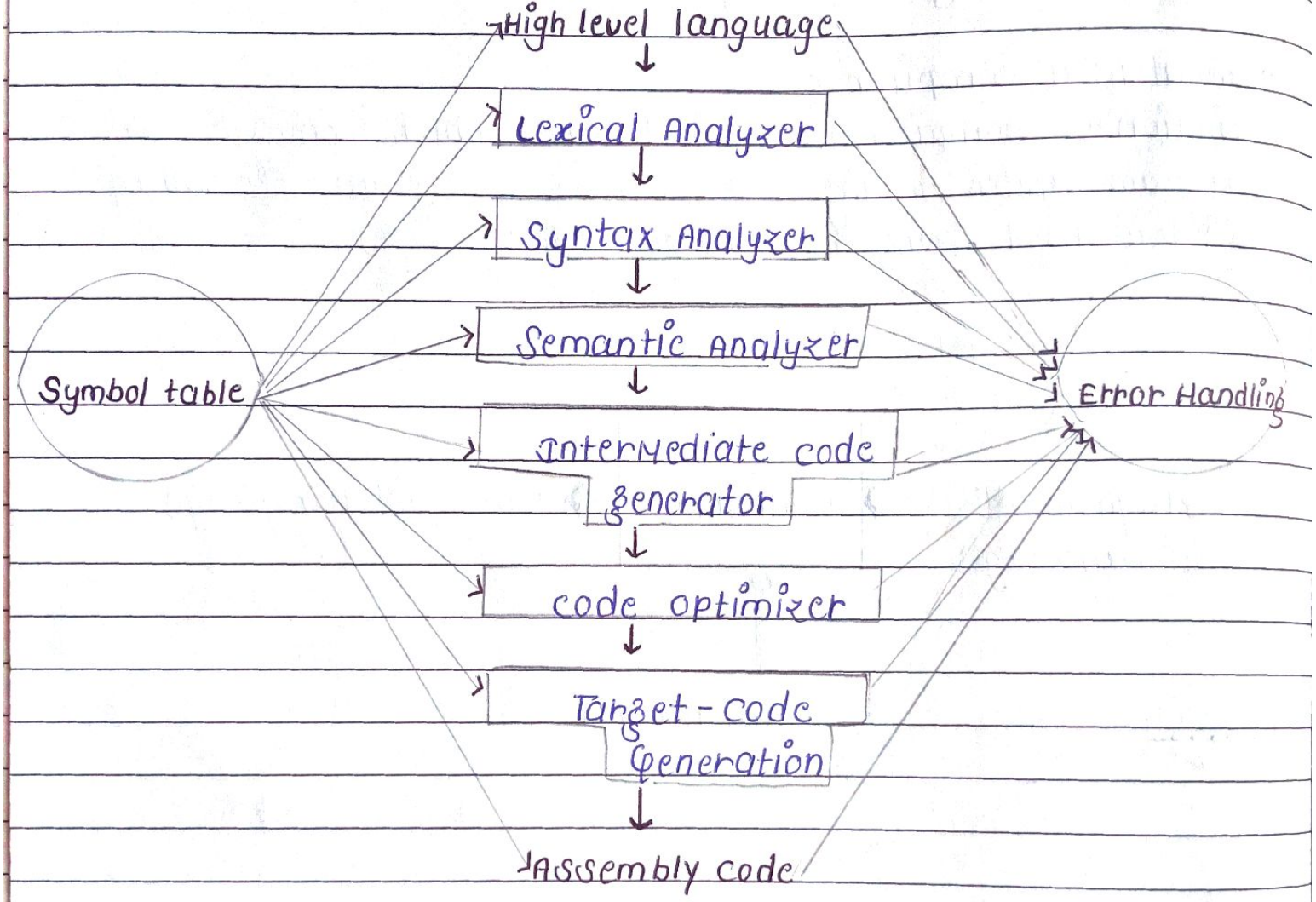
- what is a compiler ?

Definition :- compiler is a software which converts a program written in high level language (source language) to low level language (machine language).



• PHASES OF A COMPILER :-

→ Structure of compiler :-



• Analysis-Synthesis Model of analyzer :-

* Broadly compile design divide into two part :-

1. Analysis
2. Synthesis (Design)

In detail these two part can be extended (divided) into seven phases as shown in the above diagram.

1. Lexical Analyzer -:

- The Role of lexical analyzer -:
- lexical analyzer is module of compiler lexical analyzer divides the program into "tokens" example keywords, userdefined words, delimeters, special symbol etc. Scan the code is character by character.
- The lexical analyzer breaks these syntaxes into a series of tokens, by removing any whitespace or comments in the source code.
- if the lexical analyzer finds a token Invalid. it generates an error.
- it reads character streams from the source code, checks for legal tokens. and passes the data to the syntax analyzer when it demands.

- **Tokens** -: In programming language keywords, constants Identifiers, strings, numbers, operators and punctuations symbols can be considered as tokens.

example -:

int value = 100;

contain the tokens -:

int(keyword), value(Identifier), =(operator),

100(constant), ;(symbol)

- **Input Buffering** :- Buffer is a short memory as a in the RAM where a certain part of a source code are placed taking a input of the a source program in this buffer lexical analyzer scans the source code and this process. (taking input part of the source code- analyzing / scanning) gets repeated until the whole program is analyzed.

- **Regular expression** :- The lexical analyzer needs to scan and identify only a finite set of valid string/token lexeme that belong to language in hand. it searches for the pattern defined by the language rules.

Example :-

$01(1)^*011$

01011
011011
0111011
01111011
valid string of this expression.

$(1)^*$ → { }, {11}, {1111}

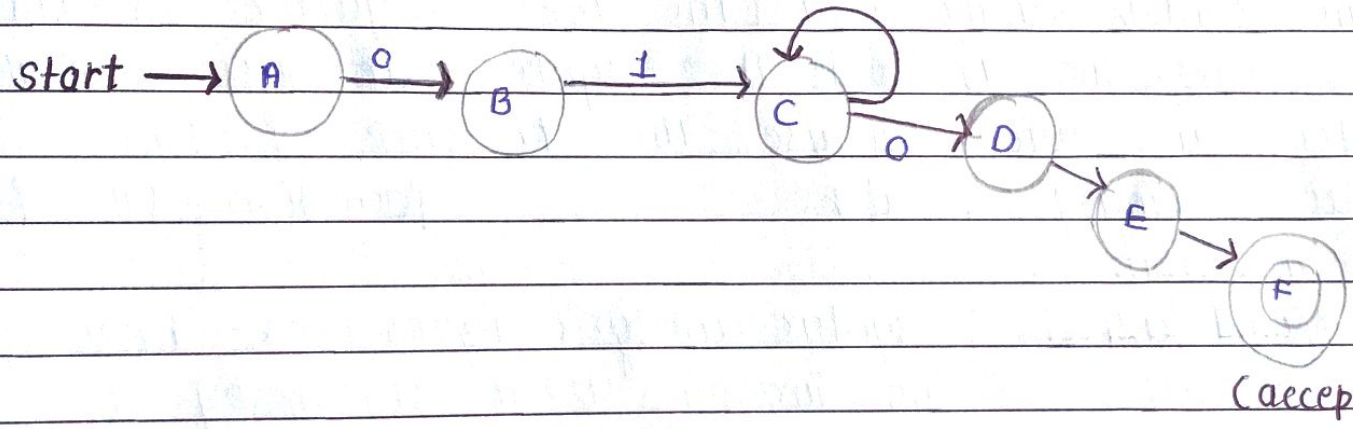
$(1)^+ = \{1\}, \{11\}$

{111}
→ at least one value compulsory.

- some of the Invalid tokens :-

0110011
01110111

• **Finite automata** :- Finite automata is state machine that takes a string of symbols as input and changes its state accordingly finite automata is a recognizer for regular expression. when a regular expression string is fed into finite automata, it changes its state for each literal if the input string is successfully processed and the automata reaches its final state it is accepted i.e. the string just fed was said to be a valid token of the language in Hand.



• **Lex** -: Lex reads on input stream specifying the lexical analyzer and writes source code which implements the lexical analyzer in the c programming language.

* structure of lex file -:

There are following section.

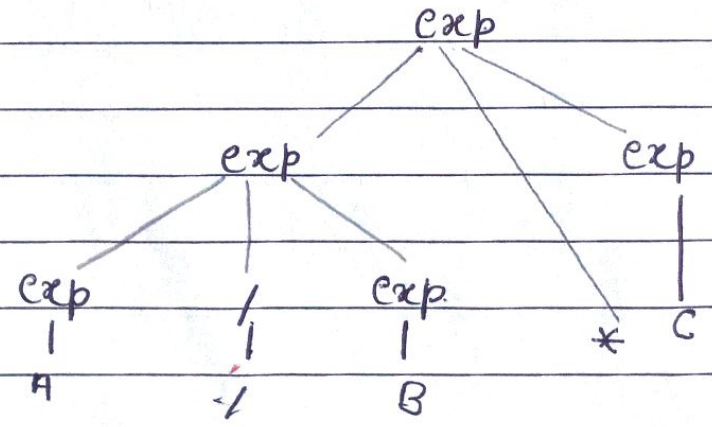
→ definition section
→ The rules section
→ The c code section.

• Syntax Analysis :- (Analysis of the grammer / Rule of grammer)
The module which do the syntax analysis is called parser
The parser has two function it checks that the token appearing in input occur in pattern that are permitted by specification of the source language. It also imposes on the token a tree-like structure that is used by subsequent phases of the compiler.
For example :- if a program contains the expression

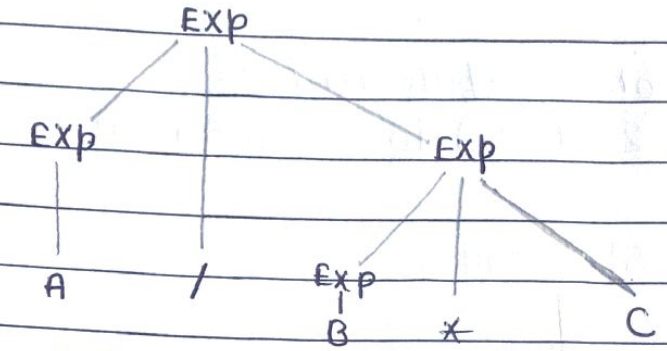
A+/B

then, after lexical analysis the expression might appear to the syntax analyser as the token sequence id + / id on seeing the (/) divide the syntax analyzer should detect an error because the presence of two adjacent binary operator. formation rule of the expression.
The second aspect of syntax analysis is to make hierarchi-cal structure of the incoming token stream by the identifying which part of the token stream grouped together.

A / B * C



• parse tree (a)



- parse tree (b)

for example :- A/B*C has two possible interpretation.

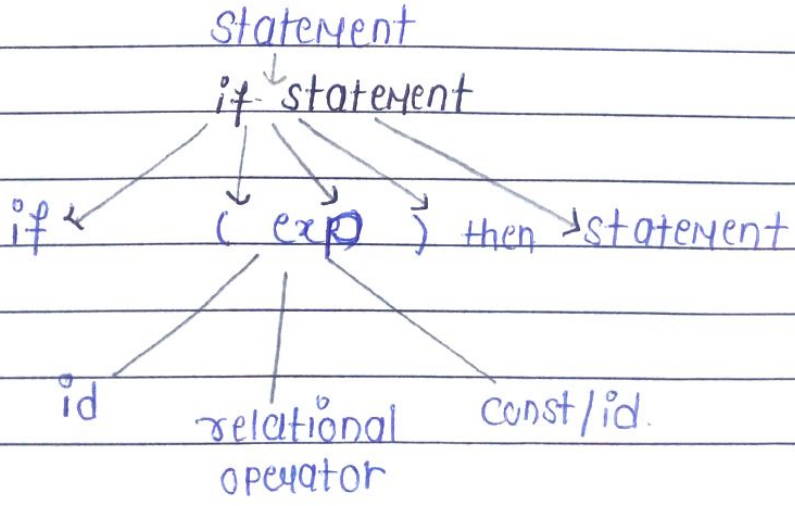
- (a) divide A/B then multiplied by C.
- (b) Multiply B by C (B*C) and then used the result to divide A.

These two interpretation can be represent in term of a parse tree as shown above.

The language specification must tell us to the (compiler) which of the presentation (a) and (b) is to be used).

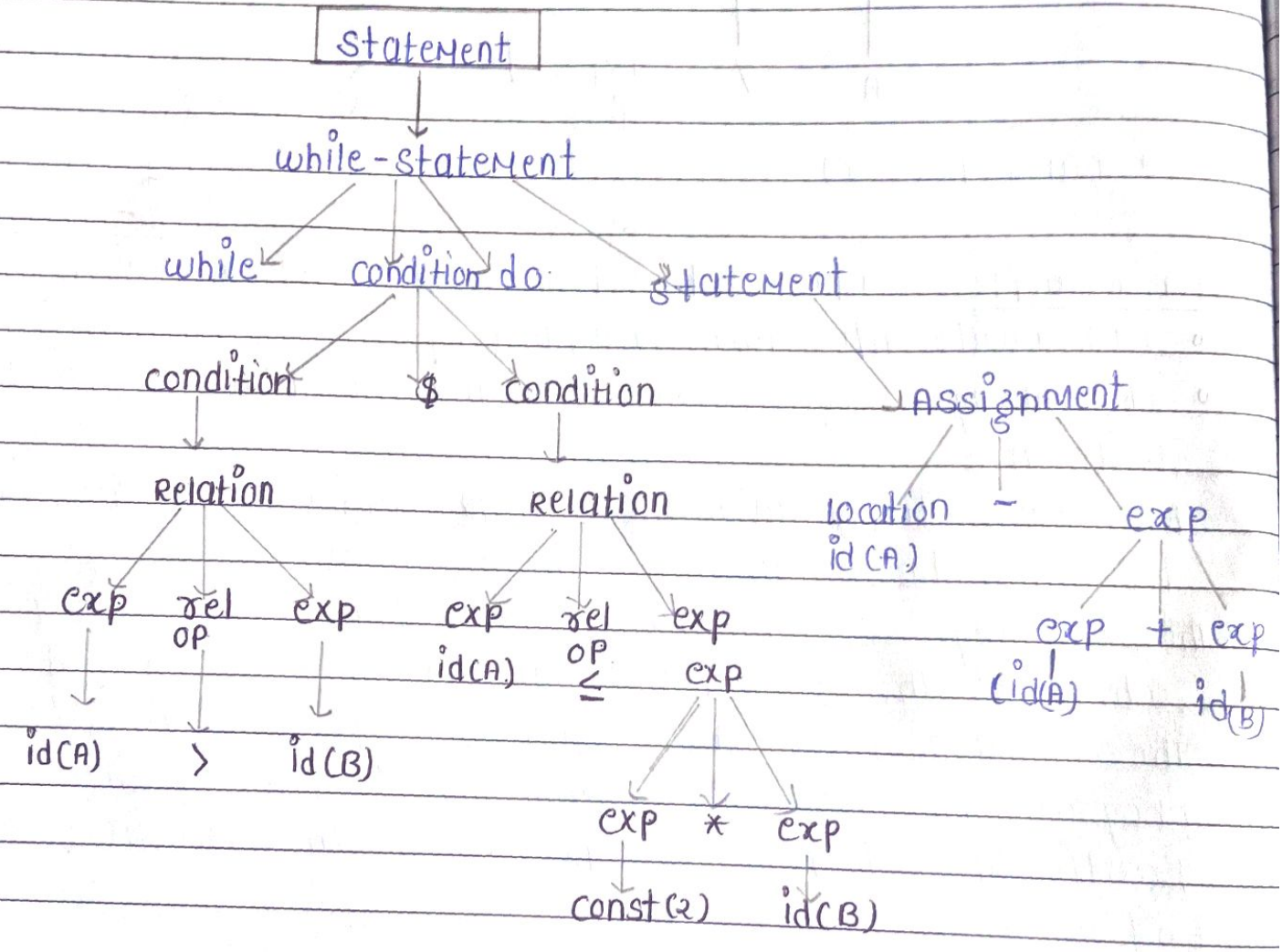
These rules form the synthetic specification of the programming language and context free grammer are particularly helpful in specifying the sythetic structure of a grammer.

* The parse tree for if statement :-

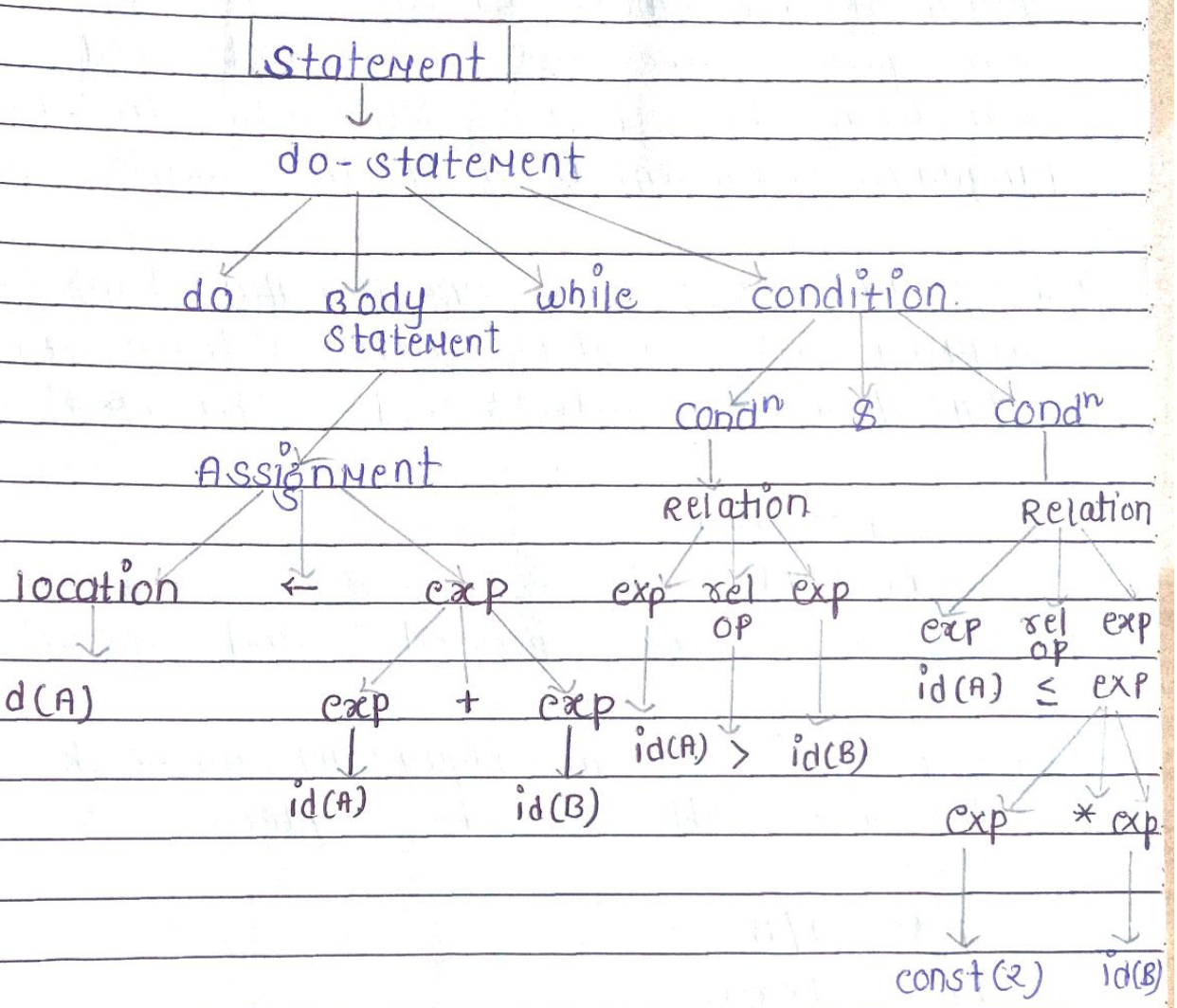


* parse tree for while statement -:

while (A>B && A<=2*B-5) do , A = A+B;



- parse tree of do-while statement :-



- Semantic Analysis :- By semantic analysis we draw the correct meaning of the source statement and accordingly the parser generate parse tree there fore to same extense the responsibility of the parser is do semantic analysis also.

- Intermediate code generation :- on a logical level the output of the syntax analyser is some representation of a parse tree the intermediate code generator transform this parse tree into an intermediate language representation of the source program.

→ Three address code :- one popular type of intermediate language is what is called "three-address code" a typical three address code statement is.

$$A \leftarrow B + C$$

$$id \leftarrow id\ op\ id$$

where A, B, C are operands and operator is binary operator.

* The parse tree in figure (A) might be converted into the three address code sequence.

$$T_1 \leftarrow A / B$$

$$T_2 \leftarrow T_1 * C$$

where, T_1, T_2 are names of temporary variable.

- INTERMEDIATE CODE GENERATION -: (while loop)

```
L1 : if A>B goto L2
     goto L3
L2 : T1 := 2*B
     T2 := T1-5
     if A ≤ T2 goto L4
     goto L3
L4 : A := A+B
     goto L1
L3 :
```

fig 1.7 Intermediate code generation.

- Optimization -: object programs that are frequently executed should be fast and small. certain compilers have within them a phase that tries to apply transformation to the output of the intermediate code generator, in an attempt to produce an intermediate-language version of the source program from which a faster or smaller object language program can ultimately be produced this phase is popularly called the optimization phase.

- Local optimisation -: There are local transformation that can be apply to a program to attempt an improvement.

for example -:

In fig 1.7 we saw two instance of jumps over jumps in the intermediate code.
i.e.

```
if A>B goto L2
   goto L3          —— (1.3)
L2:
```

This sequence could be replaced by the single statement

```
if A≤B goto L3      —— (1.4)
```

sequence (1.3) would typically be replaced in the object program b machine statements which
1. compare A and B to set the condition codes.
2. jump to L2 if the code for > is set, and
3. jump to L3.

sequence (1.4) on the other hand would be translated to machine instructions which
4. compare A and B to set the condition codes and
5. jump to L3 if the code for < or = is set.

if we assume A>B is true half the time then for (1.3) we execute (1) and (2) all the time and (3) half the time, for an average of 2.5 instructions for (1.4) are always execute two instructions, a 20% saving Also, (1.4) provides a 33% space saving if we crudely assume that all

instructions require same space another important local optimization is the elimination of common subexpressions provided A is not an alias for B or C. the assignments.

A := B+C+D
E := B+C+F

might be evaluated as

T1 := B+C
A := T1+D
E := T1+F

Taking advantage of common subexpression B+C, common subexpression written explicitly by the programmer are relatively rare, however,

A more productive source of common subexpression arises from computations generated by the compiler itself. chief among these is subscript calculation for example, the assignment.

A[i] := B[i] + C[i]

will, if the machine memory is addressed by bytes and there are, say four byte per word require 4*I to be computed three times.
An optimizing compiler can modify the intermediate program so that the calculation of 4*I is done only once.

Note that it is impossible for the programmer to specify that this calculations are not explicit at the source level.

- **Loop optimisation** -: Another important source of optimisation concerns speed ups of loops. loop are especially good targets for optimisation because programs spend most of their time in inner loops. A typical loop improvement is to move a computation that produces the same result each time around the loop to a point in the program just before the loop is entered, Then this computation is done only once each time the loop is entered, rather than one for each iteration of the loop. Such a computation is called loop invariant.

Code Generation -: The code generation phase converts the intermediate code into a sequence of machine instructions for example.
a simple code generator might map the statement

A := B+C

into the machine code sequence

```
LOAD B
ADD C
STORE A
```

However, Such a straight forward micro like expansion of intermediate code into machine code usually

produces a target program that contained many redundant LOADS and STORES and that utilizes the resources of target machine inefficiently.
To avoid this redundant loads and stores, a code generator might keep track of the run time containeds of registers.
knowing what quantities resive in the register the code generator can be generates loads and stores only when necessary.

10/11/2021

- Table management / Book keeping / symbol table management:

A compiler needs to collect information about all the data objects that appear in the source program for example :-
a compiler needs to know weather a variable represent an integers or a real number, what size and array has, How many arguments a function expects and so on. The information about data objects may be explicit as in declaration or Implicit as in the first letter of an identifier or in the context in which an identifier is used for example in FORTRAN, A(I) is a function call if A has not be declare to be an array.

11/11/2021

- The information about data objects is collected by the early phases of the compiler that lexical and sentitical analysis.

- and entered into the symbol table for example when a lexical analyzer sees an identifier MAX, it may entered the name MAX into the symbol table if it is not already there, an produce as output a token whose value component is an index to this entry of the symbol table.

If the syntax analyzer recognizes a declaration Int MAX, the action of the syntax analyser will be do not in the symbol table that MAX has type "Integer" No intermediate code is generated for this statement.

ERROR HANDLINCH :- one of the most important function of a compiler is the detection or reporting of error In the source program the error message should allow the programmer to determine where the error have occure. errors can be encountered by virtually all the phases of a compiler.

For example :- 1) The lexical analyzer may be enable to proceed because the next token in the source program is misspelled.

2) The syntax analyzer may be enable to infer a structure for its input because a syntatic error such as a missing parenthesis has occured.

3) The intermediate code generator may detect an operator whose operands have incompatible types.

4) The code optimiser, doing control flow analysis may detect that certain statements can never be reached.

5] The code generator may find a compiler created constant that is to large to fit in a word of a target machine.

6] while entering information into symbol table the book keeping routine may discover an identifier that has been multiple declaration with contradictory attributes.

whenever the phase of compiler discover an error it must report error to the error handler which issues an appropriate diagnostics message.

- **Boot straping** :- A compiler is characterised by three languages: Its source language, object language and language which it return. These language may all be quite different.

For example :-

A compiler may run on one machine and produce object code for another machine such a compiler is called a cross compiler.

many mini computer or microprocessor compiler are implemented these way: they run on a bigger machine and produce object code for smaller machine.

17/11/2021

Context free - Grammer :- it is natural to define certain programming language constructs recursively.

for example :-

we might state :-

if S_1 & S_2 are statements and E is an expression then,

```
IF (E) THEN
    S1
ELSE
    S2;
```

is a statement --- (i)

OR

if S1, S2, S3 ---- Sn are statements then,

```
BEGIN
    S1;
    S2;
    :
    Sn;
END
```

is also a statement ---- (ii)

As a third example

if E1 & E2 are expressions then,

E1 + E2 is an expression --- (iii)

if we use the syntatic category "statement" to denote the class of statements and "Expressions", then eq(i) can be expresed by the rewriting rule or production

statement :- if expression then, statement else statament — (iv)

similarly, eqn (iii) can be written as.

Expression → Expression + Expression —(v)

eqn (ii) presents a small problem we could write statement :-

```
BEGIN
    STATEMENT;
    STATENENT;
      |
      |
END;
```

But the use of (---) would create problem when we attempt to define problem based on these description for this reason we required that each rule have a known number of symbol with no (---) permitted To express eqn (ii) by recurring rules we can introduce a new syntatic category "statement-LIST" denoting any sequence of statement separated by semicolon then the recurring rules become

statement → BEGIN
statement-list ;
END;

statement-list → statement / statement ; statement-list.
—(vi)

Date: 18/11/2021
Page No:

set of rules such as eqn(vi) is an example of grammer in general, a grammer involve four quantities :-

1. Teminals 2. Non-teminals 3. a start symbols 4. productions.

- **Terminals** :- The basic symbols of which strings in the language are composed is called teminals the word "token" is a synonym for teminal when we talking about programming languages examples like, certain keywords such as begin and else are teminals.
similarly, punctuation symbols like semicolon (;) and operators like "+" are teminals.

- **Non-teminals** :- Non-teminals are special symbol that denote sets of strings the term syntatic variable and syntatic category are synonym of non-teminals.

For example :-

The syntatic category statement, expression statement-list are non-teminals one non-teminal is selected as the 3. staut symbol and its denote the language in which we are truly interested the other non-teminal are used to define other set of strings and these help to define the language.

Date:
Page No:

- **Productions** :- The productions "rewriting rule" define the way in which the syntatic category may be built-up from one-another and from the terminals each production consist of a non-terminal, followed by an arrow, followed by a string of terminals and non-terminals.

For example :- eqn (iv), (v), (vi) are the productions rules.

Example 1 :-

consider the following grammer for simple arithmet expressions.

The non-terminal symbols are expression and operator, with expression as the start symbol The terminal symbols are Id, t, −, *, /, ↑, (,). The productions are :-

expression → expression operator expression
expression → (expression)
expression → − expression
expression → Id expression
operator → +
operator → −
operator → *
operator → /
operator → ↑

Date: 23/11/2021
Page No:

OR

Start Symbol →	$E \rightarrow EAE \mid (E) \mid -E \mid id$ $A \rightarrow + \mid - \mid * \mid / \mid \uparrow$

- **Derivations and parse trees :-**

→ How does a context free grammer (CFG) define a language? The central idea is that productions may be applied repeatedly to expand the non-terminals in a string of non-terminals and terminals.

For example :- consider the following grammer of awithmetic expressions.

$E \rightarrow EAE \mid (E) \mid -E \mid id$

$A \rightarrow + \mid - \mid * \mid / \mid \uparrow$

we can take a single E and repeatedly apply productions in any order to obtain a sequence of replacement.

For example :-

$E \rightarrow -E \rightarrow -(E) \rightarrow -(id)$

we call such a sequence of replacement a derivation of $-(id)$ from E this derivation provide a proof that one particular instance of an expression is the string $-(id)$.

NOTE :- → * means 0 or more number of time applied derivation.

→ + Means one or more than one number time applied derivation.

Date:
Page No:

- Given a CFG $G \rightarrow (V, T, P, S)$ with start symbol S we can use the $\overset{+}{\Rightarrow}$ this relation to define $L(G)$, the language generated by G.

Strings in $L(G)$ may contain only terminals symbol of G. we say a string of terminals w is in $L(G)$ if and only if $S \overset{+}{\Rightarrow} w$.

The string w is called a sentence of G if $S \overset{*}{\Rightarrow} \alpha$, where α may contain non-terminals. then we say α is a sentential form of G.

Example :-

The string - (id+id) is a sentence of grammer beccuse,

$$E \Rightarrow -E \rightarrow -(E) \Rightarrow -(E+E)$$
$$\Rightarrow -(id+E) \Rightarrow -(id+id)$$

The strings E, -E, -(E) ... -(id+id) appearing in this derivation are all sentential form of this grammer we can write $E \overset{*}{\Rightarrow} -(id+id)$. it is not hard to show that the language of grammer is the set of all awithmetic expressions involving the binary operations + and *, The unary operator -, and the operand id.