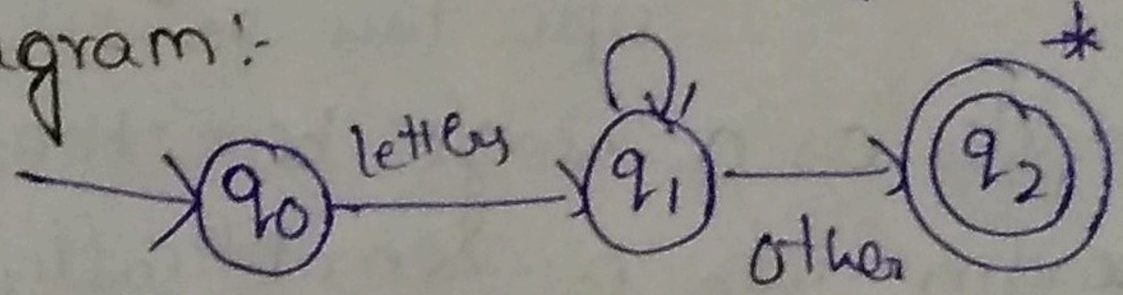
① a) Define regular expressions. Write a regular expression for identifier, keyword & design a transition diagram?

The lexical analyzer needs to scan & identify only a finite set of valid string (or) token (or) lexeme that belong to the language in hand. it searches for the pattern defined by the language rules.

Regular expressions have the capability to express finite languages by defining a pattern for finite strings of symbols. The grammar defined by RE is know as regular grammar. The language defined by regular grammar is know as regular language.

Identifier:- An identifier can be composed of letters such as uppercase, lowercase letters, underscore, digits but the starting letter should be an alphabet (or) underscore.

Transition diagram:-

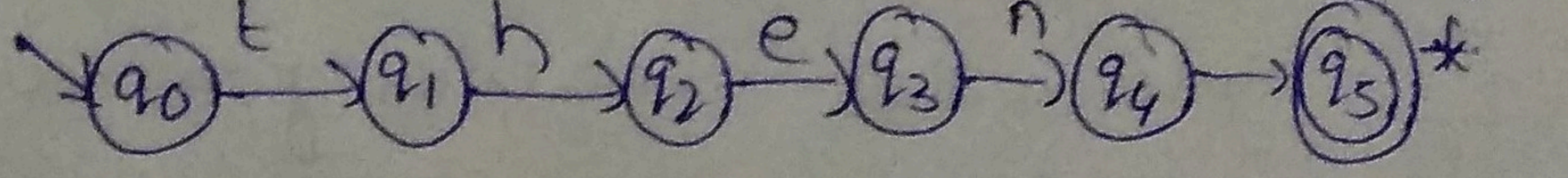


Representing language tokens using Regular Expression:-

Identifier = (letter)(letter/digit)*

Keywords:- Keywords are predefined, reserved words used in programming that have special meanings to the compiler keywords are part of syntax and they cannot be used as identifier.

Ex:- Then



(1)b) Write short notes on the following:-
a) Token b) pattern c) lexeme

Token:- A token 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. Ex:- A particular keyword (or) a sequence of input characters denoting an identifier.

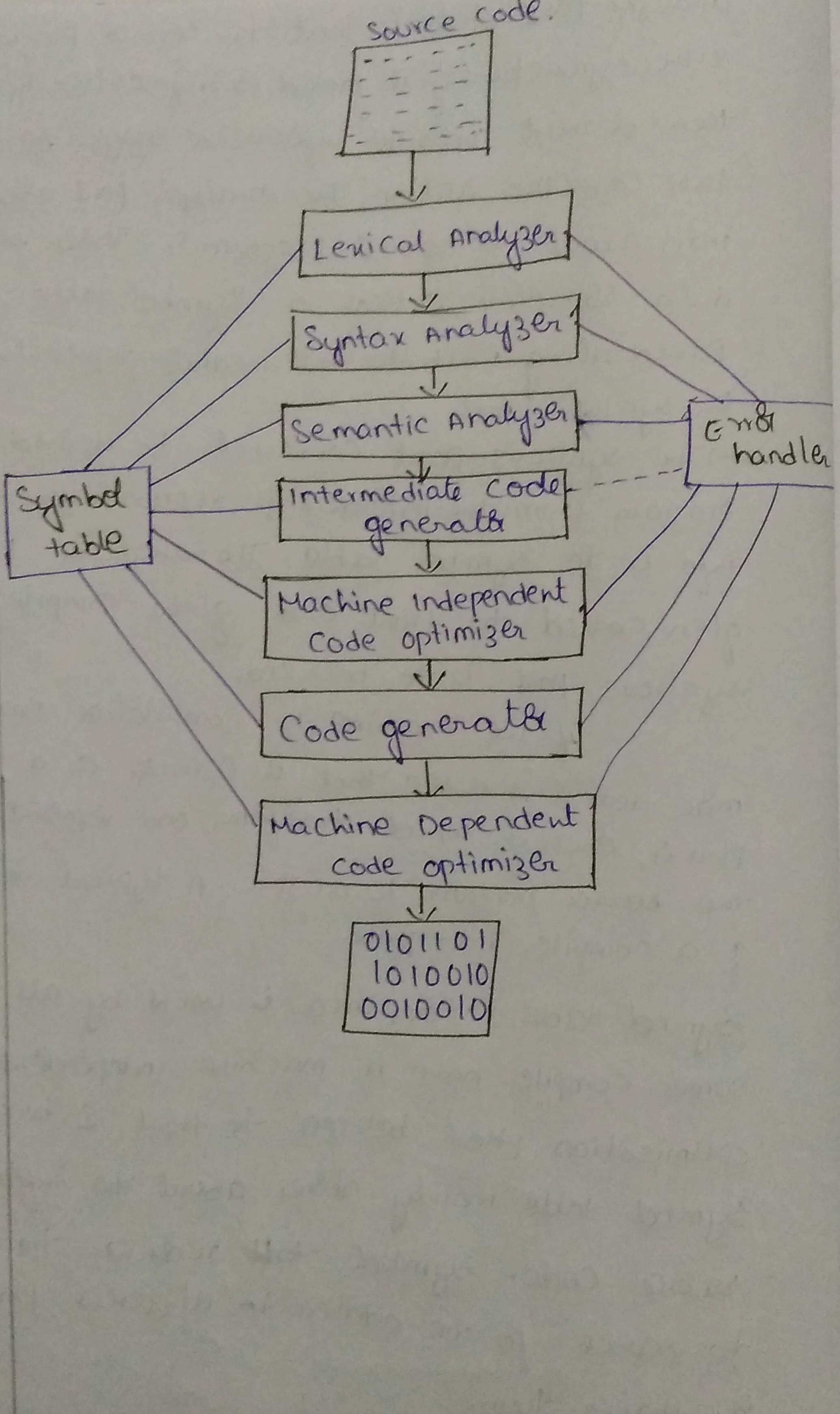
Pattern:-
-> A pattern is a description of the form that the lexemes of a token may take
-) The pattern is just the sequence of characters that form the keyword.

Lexeme:- A lexeme is a sequence of characters in the source program that matches the pattern for a token & is identified by the lexical analyzer as an instance of that token.

(2) a) Explain the structure of a compiler

upto this point we have treated a compiler as a single box that maps a source program into a semantically equivalent target program. if we open up this box a little, we see that ~~are~~ there are 2 parts to this mapping. they are
-> Analysis
-> Synthesis

Analysis:- The analysis part breaks up the source prog into constituent pieces & imposes a grammatic stru on them. it then uses this structure to create an intermediate representation of the source program. if


Source code.
Lexical Analyzer
Syntax Analyzer
Semantic Analyzer
Error handler
Symbol table
Intermediate Code generator
Machine Independent Code optimizer
Code generator
Machine Dependent Code optimizer
01011101
1010010
0010010

② b) Consider the following Fragment of 'c' code

```
Float i,j;
i=i*70+j+2';
```

Write the output at all phases of a compiler.
By considering a expression, i = i * 70 + j + 2

* Lexical Analysis:-

The primary functions of this phase are:

-> Identify the lexical units in a source code
-> classify lexical units into classes like constants, reserved words, & enter them in different tables. it will ignore comments in the source program.
-> Identify the token which is not a part of the language.

O/P = $id_1 = id_1 * 70 + id_2 + 2$ (Stream of tokens)

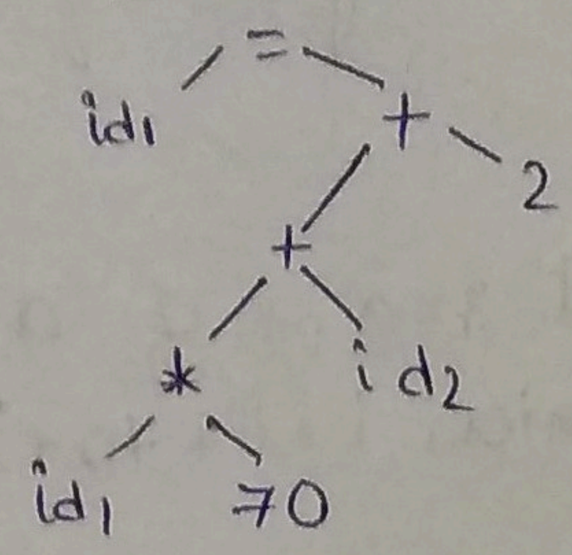
i = identifier.
= = Assignment Operator
70 = number
+ = Addition operator
* = multiplication operator
j = identifier.

Syntax Analysis :- It accepts the collection of tokens as inputs & it produces a parse tree as O/P.
-> The operators with higher precedence is used first, operators = Internal nodes.

Symbol table :- info about identifiers are stored i.e, no of identifiers name & value of its.

'*' is having high priority than '+'.

Parse tree :-



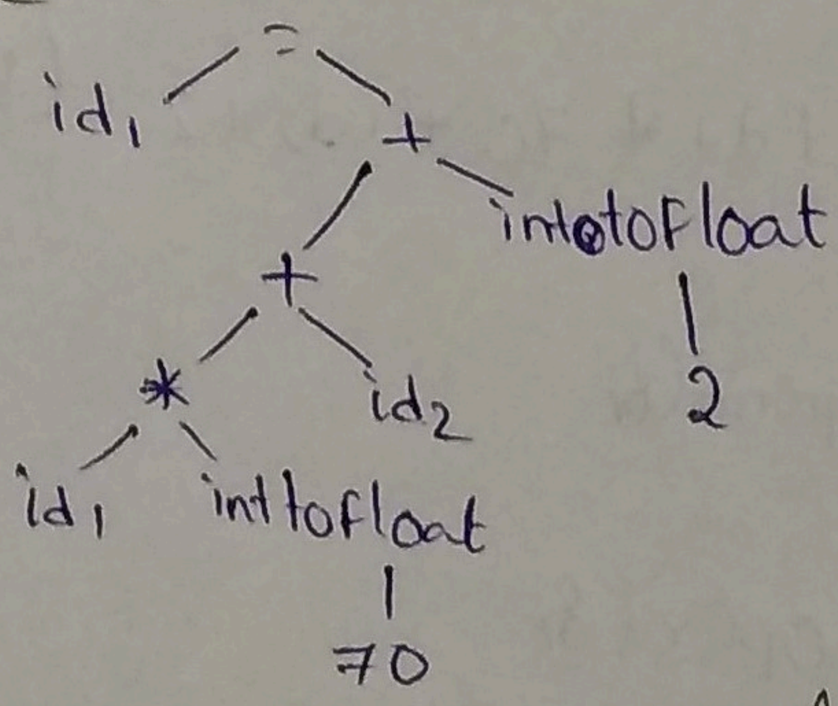
= Assignment operator left id_1 = right side value after computin all the operations.

Semantic Analyzer :- It accepts parse tree as input & produces correct parse tree as output.

Type Conversions are performed here, Hence i & j are floating values, ten result will be Float.

we Convert 70 & 2 as float i.e 70.0 & 2.0.

Parse tree :-



Intermediate code generator :- It accepts correct parse tree as input & produces intermediate code as output

3- Address code notation

The corresponding operations contain 3 oper

This requires 3 rules to be followed :-

-) t& assignment operation, te maximum g 1 operato should be present in right side.

-) it should generate tempoary variable fa storing result.

-) some instruction may contain fewer than 3 operands.

(3) a) Explain in detail the difficulties in top down Parsing

The following are the problems associated with top-down Parsing:-

-> Backtracking
-> left recursion
-> left factoring
-> Ambiguity

Problems with top-DOWN parser:-

-> Only judges grammatically
-> Stops when it finds a single derivation.
-> NO semantic knowledge employed.
-> problems with left-recursive rules
-> problems with ungrammatical sentences.

(3) b) construct first & follow set from the flg grammar

S -> aAB | bA | ∈

A -> aAb | ∈

B -> bB | ∈

TO calculate first & follow set, grammar should not have left recursive.

Productions	First set	Followset
S -> aAB	{a, b, ∈}	{$}
S -> bA | ∈		{b, a, $}
A -> aAb	{a, ∈}	
A -> ∈		{$}
B -> bB	{b, ∈}	
B -> ∈		

Intermediate Code :-

$t_1 = \text{inttoFloat}(70)$

$t_2 = id_1 * t_1$

$t_3 = t_2 * id_2$

$t_4 = \text{inttoFloat}(2)$

$t_5 = t_3 + t_4$

$id_1 = t_5$.

Code Optimizer :- It accepts intermediate code & re
the unimportant (&) unnecessary steps.

$t_1 = id_1 * 70.0$

$t_2 = t_1 + id_2$

$t_3 = t_2 + 2.0$

$id_1 = t_3$.

Hence from 6 lines of operations are reduced to
line of code.

Code generator :- It produces assembly level lang

```
LDF   R1, id1
MULF  R1, R1, #70.0
LDF   R2, id2
ADDF  R1, R1, R2
ADDF  R1; R1, #2.0
STF   id1, R1
```

(4) a) Define left recursion. Eliminate left recursion from the following grammar.

$A \rightarrow Ba / Aa / c$

$B \rightarrow Bb / Ab / d$.

Left Recursion :- A production of grammar is said to have left Recursion if the left most variable of its RHS is same as variable of its LHS.

-> A grammar containing a production having left recursion is called as left Recursion Grammar.

Ex:- $S \rightarrow Sa / \epsilon$

-> left Recursion is considered to be a problematic situation for Top down parsers.

-> Therefore, left recursion has to be eliminated from the grammar.

Elimination of left Recursion:- left recursion is eliminated by converting the grammar into a right recursive grammar.

if we have the left recursive pair of production

$$A \rightarrow A\alpha / \beta$$

(where β doesnot begin with an A).

Then we eliminate left recursion by replacing the Pair of production with

$$A \rightarrow \beta A'$$

$$A' \rightarrow \alpha A' / \epsilon$$

(Right Recursive grammar)

This functions same as a left recursion grammar.

The given production are:-

$A \to Ba/Ae/c$

$B \to Bb/Ab/d$

This is a case of indirect left recursion.

1) $A \to BaA'/cA'$

$A' \to aA'/\epsilon$

$B \to Ab/Bb/d$

2) substituting the productions of A in $B \to Ab$, we get the following grammar.

$A \to BaA'/cA'$

$A' \to aA'/c$

$B \to Bb/BaA'b/cA'b/d$

3) By eliminating left recursion we get

$A \to BaA'/cA'$

$A' \to aA'/e$

$B \to cA'bB' \; AB'$

$B \to bB'/aA'bB'/e$

(4) b) Explain error recovery strategies in predective Parsing.

Error Recovery Methods & Techniques:-

-) Panic Mode Recovery

-) phrase level Recovery

-) Erroneous Production.

Panic Mode Recovery:-

-) let us think the parser has succesfully scanned & created a parse tree till 'a' & next to that it has found an error $w = x_1 x_2 \cdots a \cdots x_n$.

-) Panic mode error recovery is based on the idea

skipping symbols on the i/p until a token in a selected set of synchronizing tokens appears.
-> After detection of error the parser should be restored to state start, where it can restart again.
-> its effectiveness depends on the choice of synchronizing set. the set should be chose so that the parser recovers quickly from errors that are likely to occur in practise.
-> Good Example for specific symbol in C is ;.

Phrase level Recovery:-
-> On discovering an error perform a local fix to allow the parser to continue.
-> Simple cases are exchanging ; with , and = with ==, delete an extraneous semicolon (or) insert a missing semicolon. Difficulties occur when the real error occured long before an error was detected.
-> The choice of the local correction is left to the compiler design.

Erroneous productions:- Include productions for common errors. we can augment the grammar for the language at hand with production that generate the erroneous constructs.
-> A Parser Constructed from a grammar augmented by these errors productions detect the anticipated errors.
-> The Parser can then generated appropriate error diagnostics about the erroneous construct that has been recognized in the input.