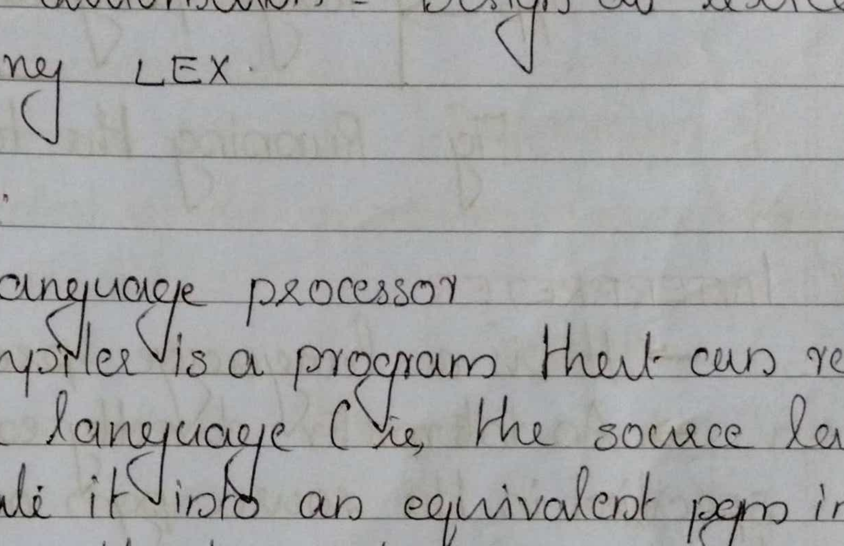
MODULE - 1.

Introduction to Compilers:- Phases of a Compiler - Analysis and synthesis phases - Lexical analysis and its role - Review of finite automation and Regular Expressions - Specification of tokens using regular expressions - Implementing lexical analyser using finite automation - Design of lexical analyser using LEX.

1.1 COMPILER:

→ Is a language processor

→ A compiler is a program that can read a pgm in one language (ie, the source language) and translate it into an equivalent pgm in another language (ie, the target language)



An important role of the compiler is to report any errors in the source pgm that it detects during

The translation process.

If the target pgm is an executable machine language pgm, it can then be called by the user to process I/ps and produce o/ps.

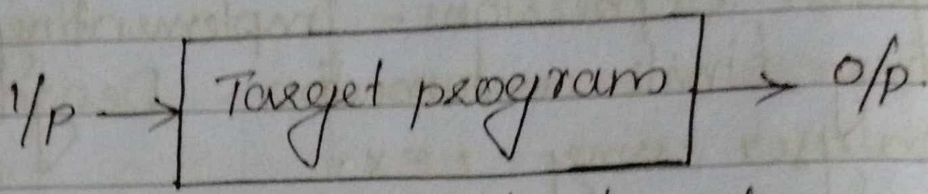


Fig: Running the target program.

1.1.1 INTERPRETER:

→ It is a language processor.

→ An Interpreter directly executes the opns specified in the source pgm on inputs supplied by the user.

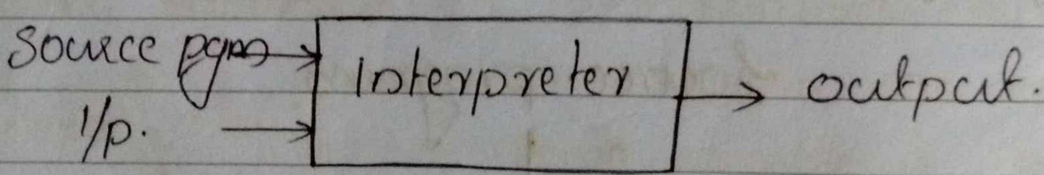
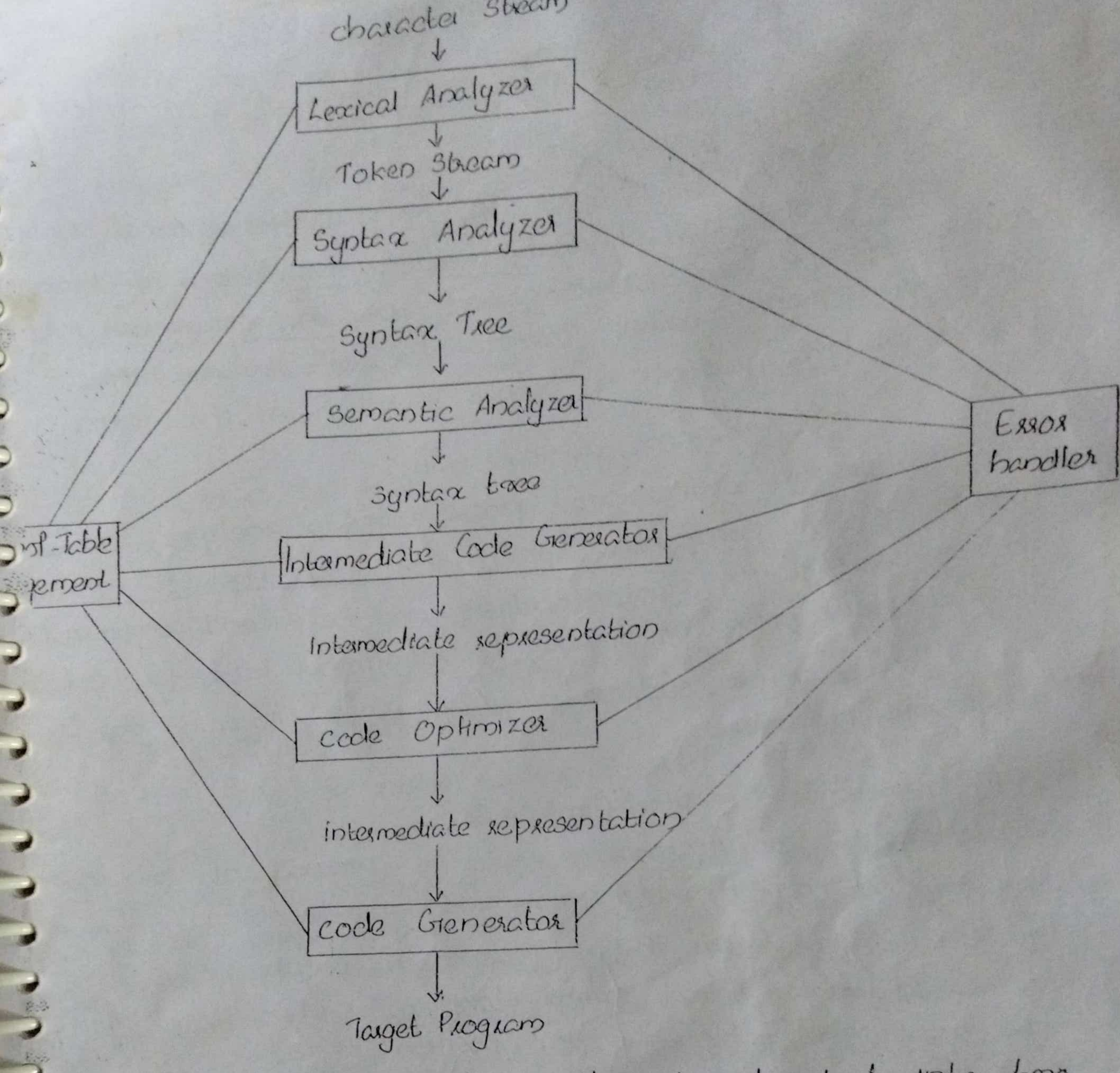


Fig: An Interpreter.

An Interpreter give better error diagnostics than a compiler, because it executes the source pgm statement by statement.

Phases of a Compiler.



Compiler can be broadly divided into two phases

1. Analysis phase [often called front end of the compiler]
2. Synthesis phase. [often called back end of the compiler].

The analysis phase breaks up the source program into pieces (tokens) and imposes a grammatical structure on them. It then uses this structure to create an intermediate representation. The analysis phase also collects information about the source program and store it in a data structure called a symbol table. If the analysis part detects that the source pgm is either syntactically ill formed or semantically unsound, then it provide error msgs.

1.2.1 ## Lexical Analysis / Scanning

- → Performed by Lexical analyser or Lexer.
- → The first phase of a compiler is called lexical Analysis or scanning.
- → The lexical analyser reads the stream of characters making up the source pgm 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, attribute-value>

which it passes to the next phase which is syntax analysis.

Here,

token-name is an abstract symbol that is used during syntax analysis.

attribute-value ⇒ points to an entry in the symbol table for this token.

Eg: If the source pgm contain the following assignment

position = initial + rate * 60.

(The characters in this assignment could be grouped into the following lexemes and mapped into the following tokens passed on to the syntax analyzer.)

After lexical analysis, the sequence of tokens are

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

- id is an abstract symbol standing for identifier.
- 1, 2, and 3 are points to the symbol table entries for position, initial and rate respectively.
- +, = and * are abstract symbols for addition, assignment and multiplication operators. They

do not need any attribute value

1	position	---
2	initial	---
3	rate	---

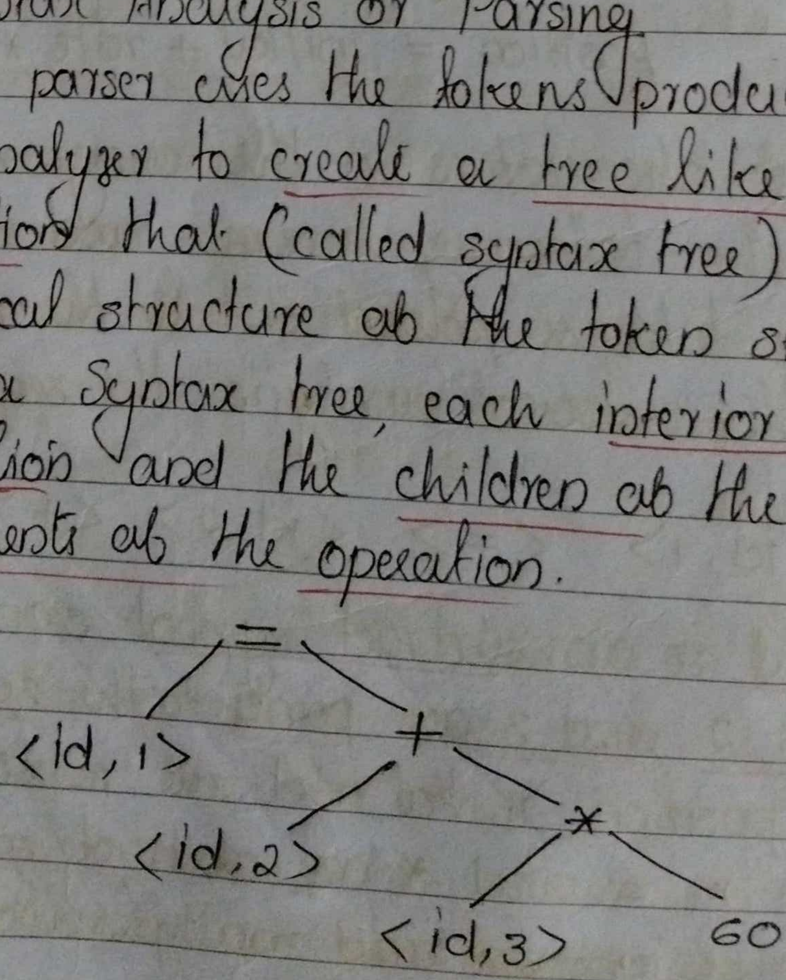
Symbol Table

1.2.2 Syntax Analysis / Parsing

- Performed by syntax Analyzer or Parser
- The second phase of the compiler is called Syntax Analysis or Parsing

→ The parser uses the tokens produced by the lexical analyzer to create a tree like intermediate representation that (called syntax tree) depicts the grammatical structure of the token stream.

→ In a Syntax tree, each interior node represents an operation and the children of the node represent the arguments of the operation.



1.2.3 Semantic Analysis:

– The Semantic Analyzer uses the syntax tree and the information in the symbol table to check the source pgm for semantic consistency with the language definition.

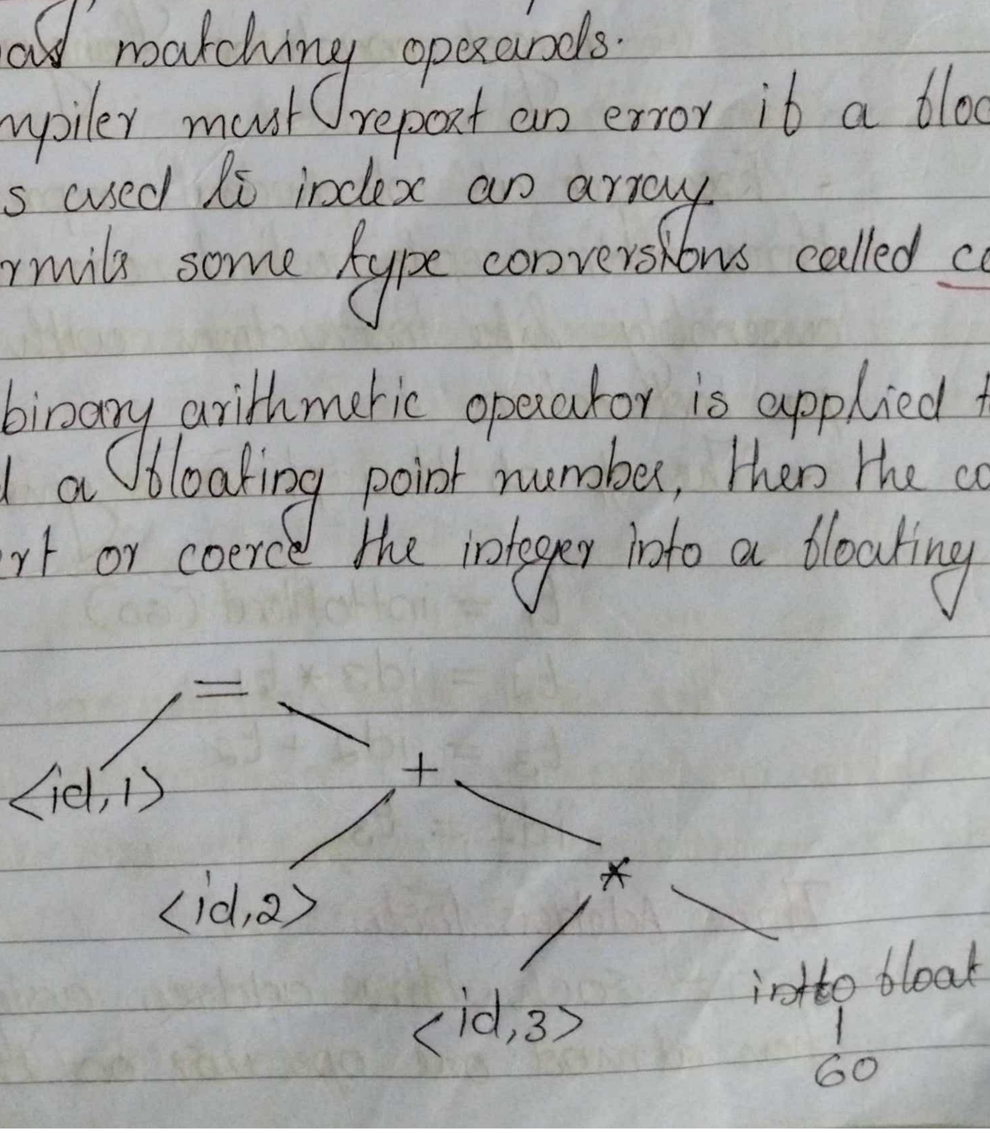
– It also gathers type information and save it in either the Syntax tree or symbol table for further use.

– An important part of semantic analysis is type checking, where the compiler checks that each operator has matching operands.

Eg: The compiler must report an error if a floating point no is used to index an array.

– It permits some type conversions called coercions.

Eg: If a binary arithmetic operator is applied to an integer and a floating point number, then the compiler may convert or coerce the integer into a floating point number.



classmate
Date
Page

1.2.4 Intermediate Code Generation.

- Syntax trees are a form of intermediate representation; They are commonly used during syntax and semantic analysis.

- compiler generates an explicit low-level or machine-like intermediate representation.

- This intermediate representation should have two important properties.

* It should be easy to produce
* It should be easy to translate into the target machine. language.

- Example of intermediate representation is called three address code. which consists of a sequence of assembly-like instructions with three operands per instruction

- The o/p of this phase is given below

t_1 = inttofloat (60)
t_2 = id3 * t_1
t_3 = id2 + t_2
id1 = t_3

Three Address Instn:

- Each three address assignment instruction has atmost one operator on the right side

- These instns fix the order in which operations are to be done.

  Eg: multiplication precedes the addition in the source pgm.

- The compiler must generate a temporary name to hold the value computed by a three-address instruction.
- Some three address instns have fewer than three operands.

1.2.5 Code Optimization.

- Code optimization phase attempts to improve the intermediate code so that better target code will result.
- Better means shorter, faster etc.

Eg: inttofloat operation can be eliminated by replacing the integer 60 by the floating point number 60.0

$$t_1 = id3 * 60.0$$
$$id1 = id2 + t_1$$

- Different optimization techniques are :-

  Local Optimization, loop optimization, deadcode elimination, strength reduction, frequency reduction etc.

1.2.6 Code Generation:-

- The code generator takes as i/p an intermedi representation of the source pgm and maps it in the target language.

- The intermediate instructions are translated into sequence of machine instructions that perfo the same task.

- A crucial aspect of code generation is the judicious assignment of registers to hold variab (registers or memory locations are selected for each of the variables used by the pgm)

Eg. Using registers R_1 and R_2, the intermediate code might get translated into the machine code

```
LDF R2, id3
MULF R2, R2, #60.0   multiplies with floating point con
LDF R1, id2   move id2 into regr R1
ADDF R1, R1, R2   Adds it with value in regr R2
STF id1, R1
```

- The first operand of each instruction specifie a destination.

- The F in each instn tells that it deals with floating point numbers.

- LDF in first line - loads contents of address id3 into regr R2.

signifies that 60.0 is to be treated as an immediate constant.
STF - Stores the value in regr R1 into the address of id1.

1.2.7 Symbol - Table Management
- (An essential feature of a compiler is to record the variable names used in the source pgm and collect information about various attributes of each name. (such as storage allocated for a name, its type, its scope number and type of arguments etc).
- A symbol table is a data structure containing a record for each variable name, with fields for the attributes of the name)
- The data structure should be designed to allow the compiler to find the record for each name quickly and to store or retrieve data from that record quickly.
- These attributes associated with a name provide information about the storage allocated for an identifier, its type, its scope and in case of procedural names such things as the no and types of its arguments, the method of passing each arguments and the type returned.

Error Handler:
- Each phase can encounter errors. Compiler stops when it finds the Ist error.

Eg: 1. Lexical Analyser : next token in the source pgm is mispelled.

2. Syntax Analyser : Syntactic errors such as a missin paranthesis.

3. IC generator : May detect an error operator whose operands have incompactible type.

4. Code Optimizer : May detect that certain stmts can never be reached.

5. Code Generator : May bind a compiler created const that is too large to fit in word of the target machine.

6. Symbol Table : May discover an identifier that has been multiply declared with contradictory attributes.

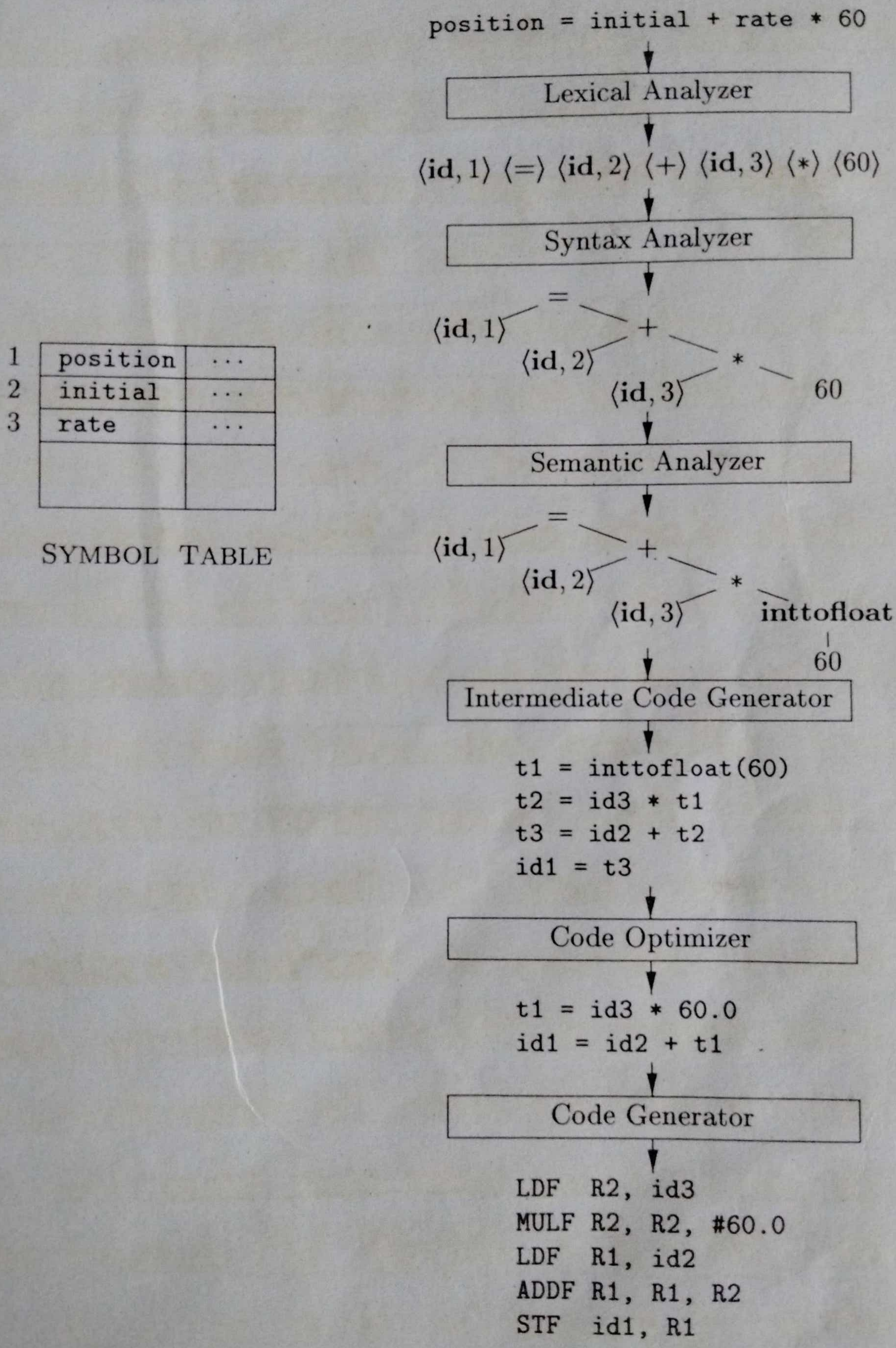


Figure 1.7: Translation of an assignment statement

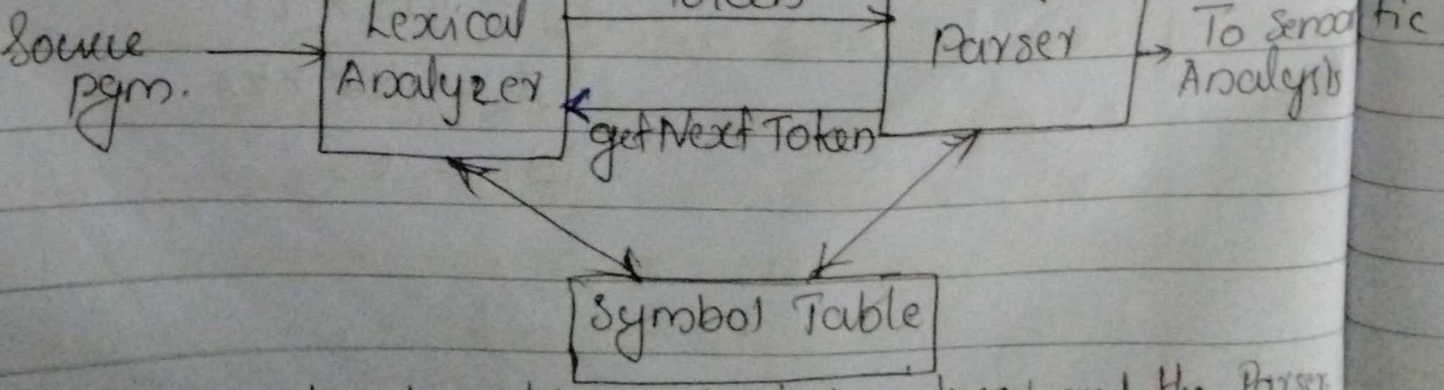
1.3 LEXICAL ANALYSIS & ITS ROLE

- First phase of a compiler.
- It reads the i/p characters of the source pgm group them into lexemes and produce as o/p a sequence of tokens for each lexeme in the source pgm.
- The stream of token is sent to the parser for syntax Analysis.

Source pgm → $\boxed{\text{Lexical Analyzer}}$ → sequence of tokens

Fig. Lexical Analyser

- It also interact with the symbol table.
- when the lexical analyser discovers a lexeme constituting an identifier, it needs to enter that lexeme into the symbol table.
- In some cases the information regarding the kind of identifier may be read from the symbol table by the lexical analyser to assist it in determining the proper token it must pass.



Interactions b/w the lexical analyzer and the Parser

Other tasks of Lexical Analyzer:

1. Stripping out comments and whitespaces (blank, new line, tab and perhaps other characters that are used to seperate tokens in the i/p).
2. Correlating error messages generated by the compiler with the source pgm. For instance, the lexical analyzer may keep track of the no. of newline characters seen, so it can associate a line no. with each error message.
3. If the source pgm uses a macro preprocessor, the expansion of macros may also be performed by the lexical analyzer.

→ Lexical analyzers are divided into 2 processes
  1. Scanning – that donot require tokenization of i/p such as deletion of comments and compaction of consecutive whitespaces characters into one.
  2. Lexical Analysis – It produce tokens from the o/p of the scanner.

1.3.1 Tokens, Patterns and Lexemes

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. eg: a particular keyword, or a sequence of i/p

characters denoting an identifier.

- The token names are the i/p symbols that the parser processes.

Pattern: — It is a description of the form that the lexemes of a token may take. In the case of a keyword as a token, the pattern is just the sequence of characters that form the keyword.

Lexeme: — Is a sequence of characters in the source pgm that matches the pattern for a token and is identified by the lexical analyzer as an instance of that token.

- The word generalised by the linear analysis may be of different kinds:
  * identifier
  * keyword (if, while, ...)
  * punctuation
  * numbers
  * literals
- Such a kind is called a TOKEN and an element of this kind is called LEXEME.
- A word is recognised to be a lexeme for a certain token by PATTERN MATCHING.

eg: letter followed by letters and digits is a pattern that matches a word like x or y with the token id (= identifier)

Token	Lexemes	Pattern
if	if	if
else	else	else
ID	x, y, no	letter followed by letters & digits
number	1, 3.14, 2.02e23	any numeric constant
comparison	<=, !=	comparison operators
literal	"Hello", "Hi"	any string of characters between " and "

- During the analysis, the compiler manages a symbol table by
  * recording the identifier of the source pgm
  * collecting information (called ATTRIBUTES) about them: storage allocation, type, scope (for fns) signature.
- when the identifier x is found by lexical analyzer
  * It generates token id
  * enters lexeme x into symbol table (if not present)
  * associates to the generated token a pointer to the symbol table entry x. This pointer is called

The lexical value ab the token.

1.3.2 Attributes bor Tokens.

When more than one lexeme can match a pattern, the lexical analyzer must provide the subsequent compiler phases additional information about the particular lexeme that matched.

eg: Pattern bor token number matches both 0 & 1 but it is important bor code generator to know which lexeme was bound in the source pgm.

So the lexical analyzer returns to the parser not only a token-name, but also an attribute value that describes the lexeme.

eg: for the token id, we need to associate with the token a great deal ab info such as its lexeme, type, location (where it is bound). This is stored in the symbol Table.

Thus, the appropriate attribute value bor an identifier is a pointer to the symbol-table entry bor that identifier

Example

$E = M * C ** 2$

Tokens generated bor the above line ab code

<id, pointer to the symbol-table entry bor E>

<assign-op>

<id, pointer to symbol-table entry for M>
<mult-op>
<id, pointer to symbol-table entry for c>
<exp-op>
<number, integer value 2>
or
pointer to symbol table entry for 2.

Note:

- In certain pairs, es especially operators, punctuations, and keywords, there is no need for attribute value.
- Token number has been given an integer-valued attribute.

1.4 INPUT BUFFERING.

It is the technique to improve the execution speed of the lexical analysis phase by speeding up the reading of source pgm.

- Here we use two buffers that are alternatively reloaded
- Each buffer is of the same size N, and N is usually the size of a disk block, eg: 4096 byte.
- Using one system read command we can read N characters into a buffer, rather than using one system call per character.
- If fewer than N characters remain in the i/p file, then a special character, represented by eof, marks

the end of the source file.
- Two pointers to the i/p are maintained.
1) Pointer lexemeBegin, marks the beginning of current lexeme, whose extent we are attempting to determine.
2) Pointer forward scans ahead until a pattern match is found. If the forward has passed to the end of the lexeme, it must be retracted one posn to the left

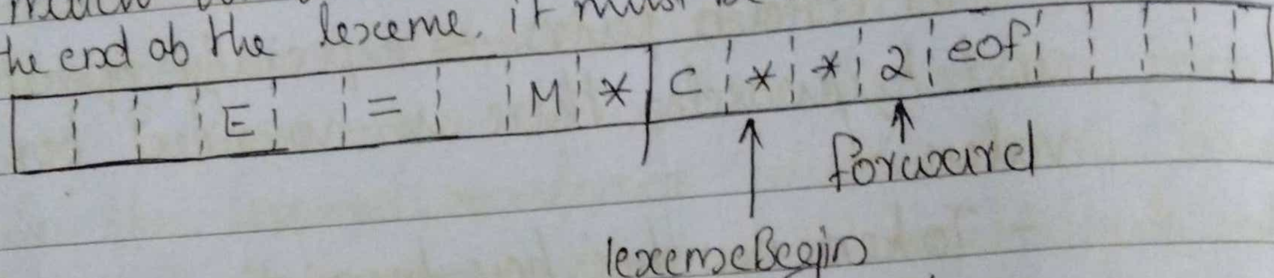


Fig: using a pair of i/p buffers

Working
- Initially both pointers points the first character of the next lexeme
- Forward pointer scans. If a lexeme is found.
- It is set to the last character of the lexeme found.
- Thus for each character read, two tests are made one for the end of buffer and the other to determine the character that is read.
- Both tests can be combined by extending the buffer to hold a sentinel character at the end.

⇒ Sentinel is a special character that cannot be part of source pgm. such as eof.

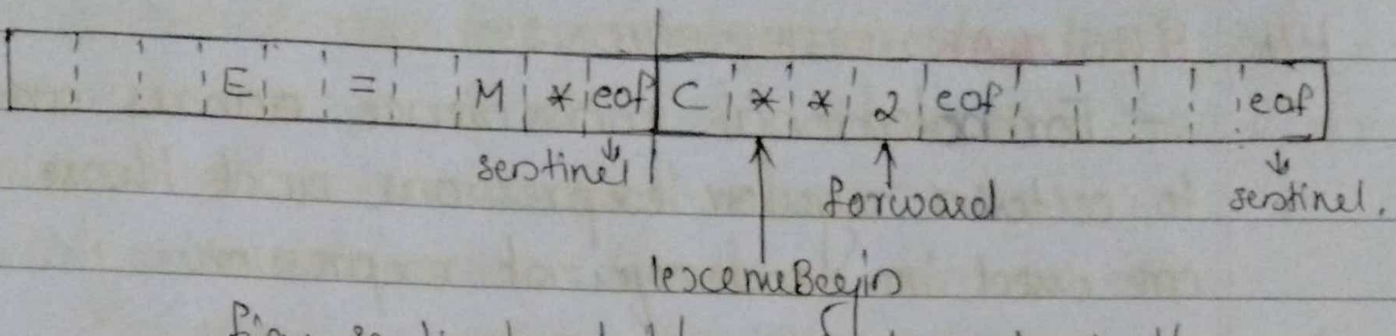


fig: sentinels at the end ab each bubber.

Fig. shows Algorithm bor advancing forward.

```
Switch (* forward + +)
{
 case eof :
     if (forward is at end ab birst bubber)
     {
         reload second bubber;
         forward = beginning ab second bubber;
     }
     else if (forward is at end ab second bubber)
     {
         reload birst bubber;
         forward = beginning ab birst bubber;
     }
     else /* eof within bubber marks the end ab input */
         terminali lexical analysis;
     break;
 case bor other characters;
}.
```

1.4 REGULAR DEFINITIONS:

- For notational convenience, names are given to certain regular expressions and those names are used in subsequent expressions. (These names are also symbols).

- If Σ is an alphabet of basic symbols, then a regular definition is a sequence of definitions of the form:

$$d_1 \rightarrow r_1$$

$$d_2 \rightarrow r_2$$

$$\cdots$$

$$d_n \rightarrow r_n.$$

where:

1) Each d_i is a new symbol, not in Σ and not the same as any other of the d's, and
2) Each r_i is a regular expression over the alphabet $\Sigma \cup \{d_1, d_2, \cdots, d_{i-1}\}$

Eg: C Identifiers are strings of letters, digits and underscores. The regular definition for the language of C-identifiers are;

letter_ → A/B/···/Z/a/b/···/z/-

digit → 0/1··/9

id → letter_(letter_/digit)*

Unsigned numbers (integer or floating point) are strings such as 5280, 0.01234, 6.336E4 or 1.89E-4.

Regular definition for unsigned numbers

digit → 0|1|...|9

digits → digit digit*

optionalFraction → .digits|ε

OptionalExponent → (E(+|-|ε) digits)|ε

number → digits optionalFraction optionalExponent

- i.e., an OptionalFraction is either a decimal point (dot) followed by one or more digits, or it is missing (the empty string)
- An OptionalExponent, if not missing, is the letter E followed by an optional + or - sign, followed by one or more digits
- Note

Atleast one digit must follow the dot. so the number doesnot match 1., but match 1.0

1.4.1 Extensions of Regular Expressions.

1. One or More Instances.

The unary, postfix operator + represents the positive closure of a regular expression and its language. That is, if r is a R.E, then $(r)^+$ denotes the language $(L(r))^+$. The operator + has the same precedence and associativity as the operator *

The following algebraic laws relates the Kleene closure and positive closure

$$r^* = r^+ / \epsilon$$

$$r^+ = r r^* = r^* r$$

2. Zero or One Instance

The unary postfix operator '?' means zero or one occurrence

ie,

$$r? = r / \epsilon$$

$$L(r?) = L(r) \cup \{\epsilon\}$$

3. character classes

- A r.e $a_1 | a_2 | \cdots | a_n$, where the a_i's are each symbols of the alphabet, can be replaced by the shorthand $[a_1 a_2 \cdots a_n]$.
- [abc] is shorthand for a|b|c. and [a-z] is shorthand for a|b|···|z

-

- Using shorthand, the regular defn. for c-identifiers are as follows:

letter_ → [A-Za-z_]
digit → [0-9]
id → letter_ (letter_ | digit)*

Regular defn for unsigned nos:

$$digit \rightarrow [0-9]$$
$$digits \rightarrow digit^{+}$$
$$number \rightarrow digits(.\,digits)?(E[+-]?\,digits)?$$

1.5 Specification of Tokens using Regular Expressions

A grammar for branching stmts,

stmt → if expr then stmt
  | if expr then stmt else stmt
  | ∈

expr → term relop term
  | term

term → id
  | number

The terminals of the grammer which are if, then, else, relop, id and number, are the names of tokens as far as lexical analyzer is concerned.

- The patterns for these tokens are described using regular defns as follows:

(patterns for id & number are same as before)

$$digit \rightarrow [0-9]$$
$$digits \rightarrow digit^{+}$$

number → digits (. digits)? (E [+-]? digits)?
letter → [A-Za-z]
id → letter (letter|digit)*
if → if
then → then
else → else
relop → < | > | <= | >= | = | <>

- For the language, lexical analyzer will recognize the keywords if, then, and else as well as lexemes that match the patterns for relop, id and number.
- In addition, lexical analyser is given the job of stripping out white space, by recognising the token ws defined by

ws → (blank|tab|newline)+

→ Here blank, tab and newline are abstract symbols that is used to express the ASCII characters of the same name.
→ Token ws is different from other tokens. As when ws is recognised it is not returned to parser, but instead lexical analyser restarts from the character that follows the ws (white space)
- After performing lexical analysis a set of tokens and attribute value (if present) are passed to the parser.

The following table shows, for each lexeme or family of lexemes, which token name is returned to parser and what attribute value.

Lexemes	Token names	Attribute value
Any ws	—	—
if	if	—
then	then	—
else	else	—
Any id	id	Pointer to table entry
Any number	number	Pointer to table entry
<	relop	LT
<=	relop	LE
=	relop	EQ
<>	relop	NE
>	relop	GT
>=	relop	GE

For the 6 relational operators, symbolic constants LT, LE etc are used as attribute value, inorder to indicate the instance of the token relop found.

1.6 Implementing Lexical Analyser using FA.

Transition diagrams:

- Transition diagrams have a collection ab nodes or circles, called states. Each state represents a condition that could occur during the process ab scanning the i/p looking for a lexeme that matches one ab several patterns.

- Edges are directed from one state ab the transition diagram to another. Each edge is labeled by a symbol or set ab symbols.

- If we are in some state s, and the next i/p symbol is a, we look for an edge out ab state s labeled by a. If we bind such an edge, advances the forward pointer and enter the state ab the transition diagram to which that edge ~~leads~~ leads

Some important conventions about transition diagrams are:

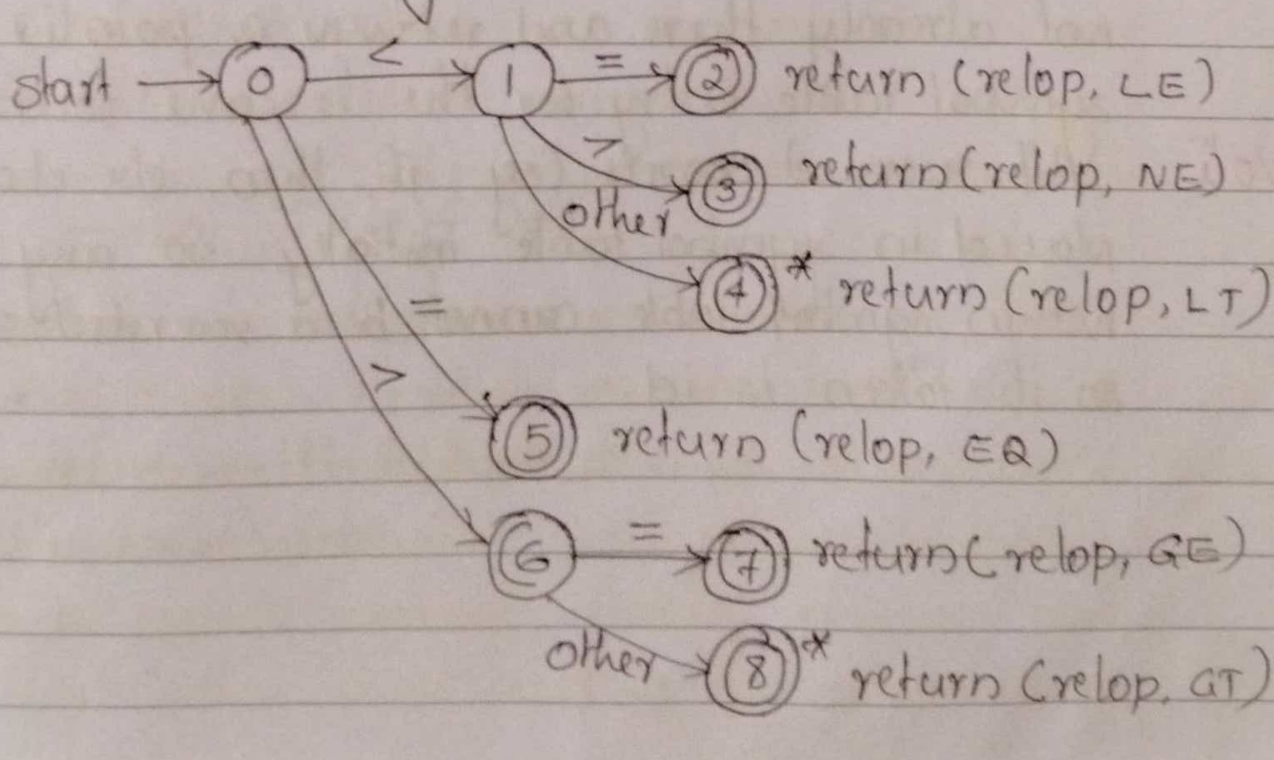
1. Certain states are said to be accepting, or final. These states indicates that a lexeme has been found, although the actual lexeme may not consists ab all positions between the lexemeBigin and forward pointers. Indicate an accepting state by a double circle and if there is an action to be taken-

typically returning a token and an attribute value to the parser - attach that action to the accepting state.

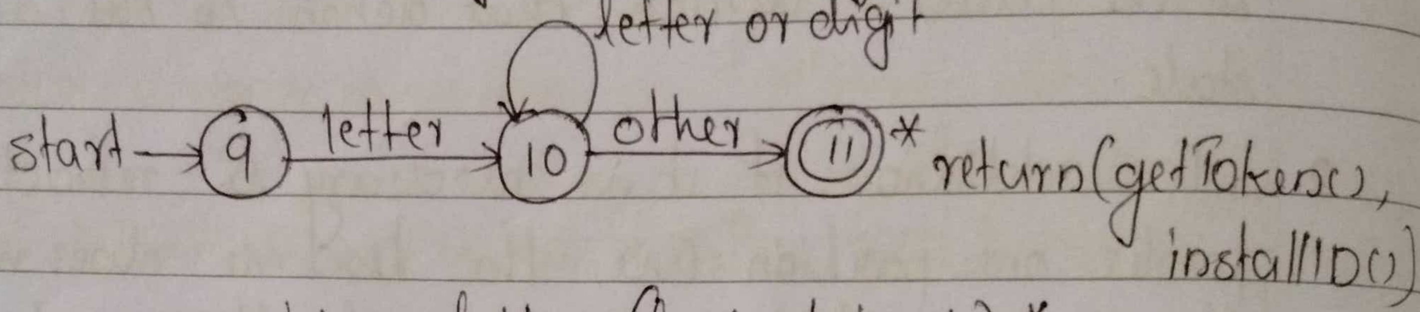
2. In addition, if it is necessary to retract the forward pointer one position (i.e. the lexeme does not include the symbol that ~~goes~~ got us to the accepting state) then we shall ~~adding~~ additionally place a * near that accepting state.

3. One state is designated as the start state, or initial state. It is indicated by an edge, labeled "start" entering from nowhere. The transition diagram always begins in the start state before any i/p symbol have been read.

Eg Transition diagram for relop :



Transition diagram for keywords & Identifier



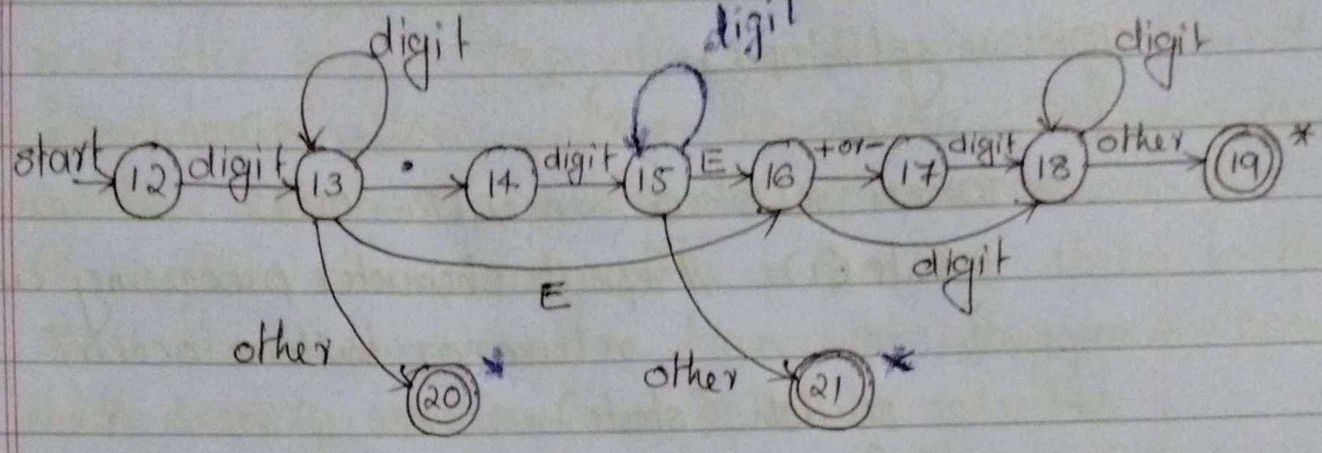
id = letter (letter|digit) *

getToken() → examines the symbol table entry for the lexeme found, and returns the token name that the symbol table entry says this lexeme represents. i.e, either id or one of the key word tokens that was initially installed in the table.

installID() → when we find an identifier a call to installID places it in the symbol table if it is not already there and returns a pointer to the symbol table entry for the lexeme found.

Note: All reserved words (eg: if, then, else etc) are placed in symbol table initially. So any identifier not in symbol table cannot be a reserved word, so its token is id.

Transition diagram for unsigned numbers



digit → 0 | 1 | . . . | 9

digits → digitdigit*

optionalFraction → .digits | ε

optionalExponent → (E(+ | -) digits) | ε

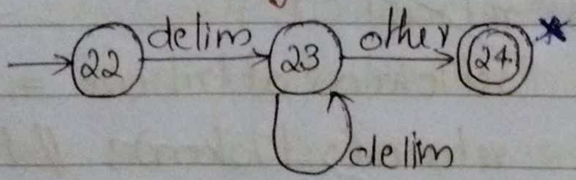
number → digit. optional Fraction. optionalExponent

Eg: 12.3

123

123.45E-6

Transition diagram for whitespace



delim → blank | tab | newline

ws → dilim+

Compiler - Construction Tools

There are specialized tools that help to implement various phases of a compiler. Some commonly used compiler - construction tools include

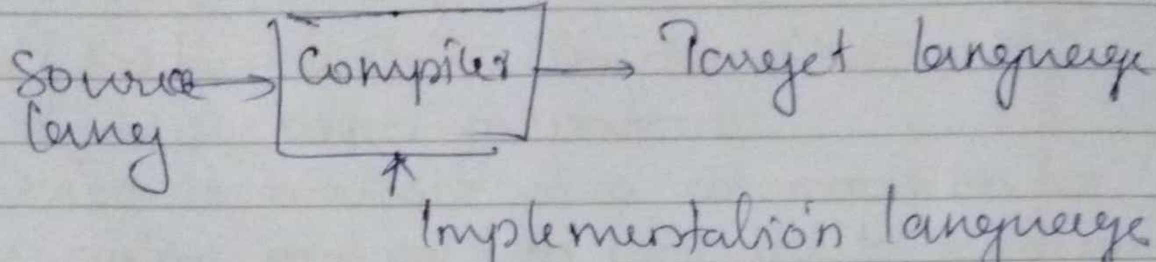
1. Parser generators → automatically produce syntax analyzers from a gramatical description of a programming language.
2. Scanner generators → produce lexical analyzers from regular expression description of the token of a language
3. Syntax - directed translation engine → produce collections of routines for walking a parse tree and generating intermediate code.
4. Code generator generators → produce a code generator from a collection of rules for translating each operation of the intermediate language into the machine language for a target machine.
5. Data - flow analysis engine → gathering of information about how values are transmitted from one part of a program to each other part Data - flow analysis is a key part of code optimization.
6. Compiler - construction toolkits that provide an integrated set of routines for constructing various phases of a compiler.

Bootstrapping

Process
- Is used to design a compiler

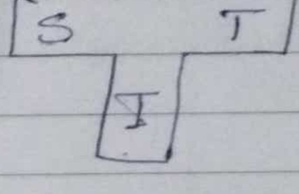
Generally a compiler can be represented using three languages

1) Source lang
2) Target lang
3) Implementation lang
 - The language in which the compiler is written.



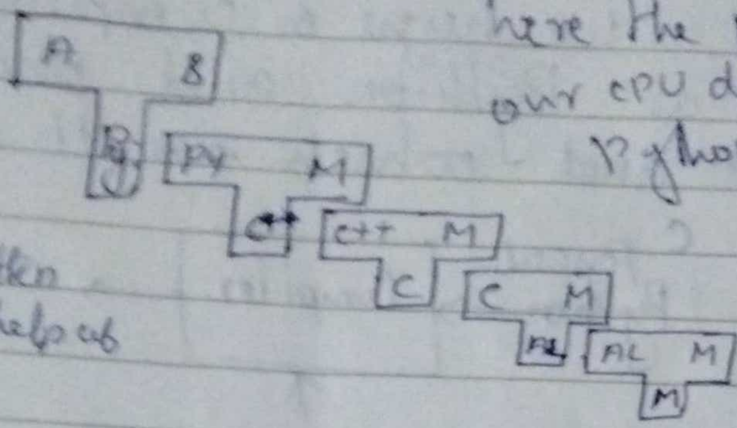
The compilation process can be explained with the help of T-diagram.

T- diagram.



S - Source lang
T - Target
I - Implementation lang

Eg: Let, we have a compiler with A as input B as o/p and the compr is written in Python lang.



here the pblm is our CPU doesn't understand Python lang.

Compiler written with the help of

Here the Assembly level code is written with the help of m/c code

Bootstrapping is a process in which a simple language is used inorder to translate a complicated pgm. but this complicated pgm ~~pd~~ produces even more complicated pgm ~~and~~ likewise the process will continue and this process is known as bootstrapping

Cross Compiler

- A compiler which runs on one m/c produces an object code for another m/c.

For a source language statement $a = b * c - 2$ where a, b and c are floating point variables * and - represent multiplication and subtraction on the same data type. Show the i/p and o/p at each of the complete phases.

$a = b * c - 2$

↓

$\boxed{\text{lexical Analyzer}}$

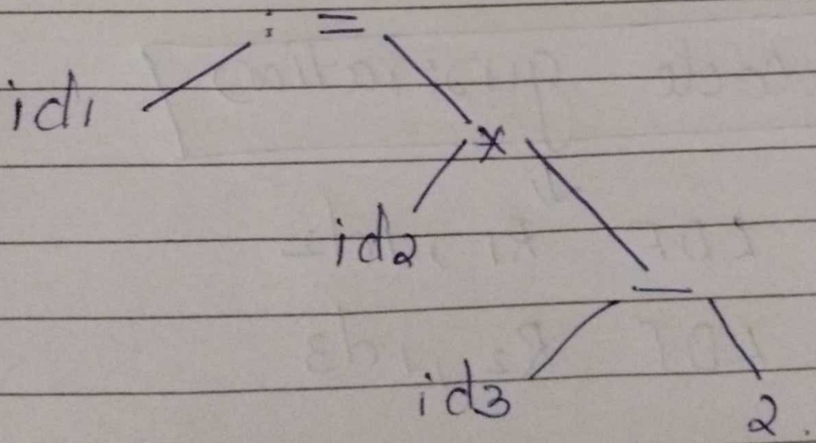
↓

$id_1 = id_2 * id_3 - 2$.

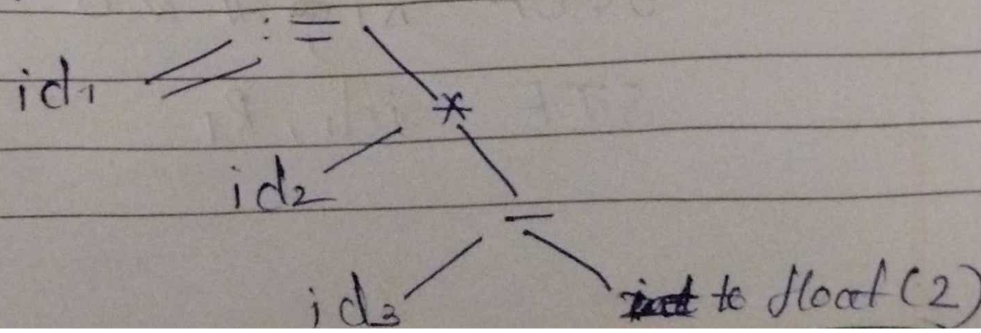
~~<id_1,> <=> <id_2> <*> <id_3> <-> <2>~~

↓

$\boxed{\text{syntax Analyzer}}$



$\boxed{\text{symantic Analyzer}}$



↓

intermediate code representation

↓

$t_1 = id_2 \times id_3$

$t_2 = \text{int to float}(2)$

$t_3 = t_1 - t_2$

$id_1 = t_3$

↓

code optimizer

↓

$t_1 = id_2 * id_3$

$id_1 = t_1 - 2.0$

↓

code generation

↓

```
LDF  R1, id2
LDF  R2, id3
MUL  R1, R2
SUBF R1, #2.0
STF  id1, R1
```

Explain how the regular expressions and finite state automata can be used for the specification and recognition of tokens

The lexical analyzer needs to scan and identify only a finite set of valid string/token/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. Regular expression is an important notation for specifying patterns. Each pattern matches a set of strings, so regular expressions serve as names for a set of strings.

The only problem left with the lexical analyzer is how to verify the validity of a regular expression used in specifying the patterns of keywords of a language. A well-accepted solution is to use finite automata for verification

Finite automata is a state machine that takes a string of symbols as input and changes its state accordingly. Finite automata is a recognizer for regular expressions. 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