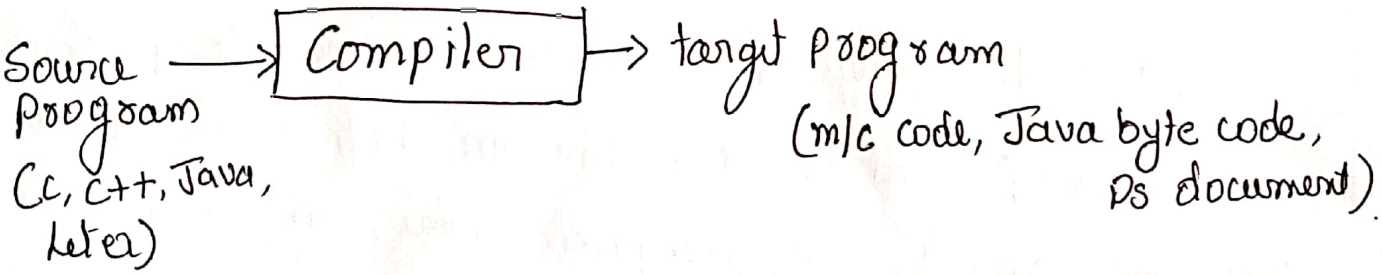
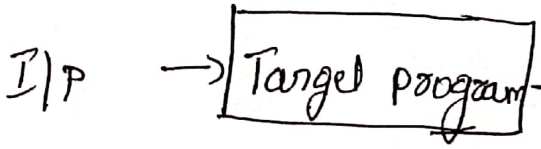
Savitha S.K.
Assistant Professor
CSE Dept. BIT

Translator: Is a program (s/w) Converts a high level language program in to a Target code. Compiler & Interpreters are the translators.

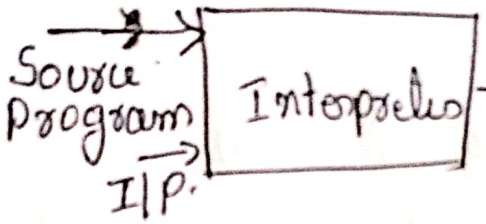
Compiler: is a program that ~~can~~ reads Source program and translate into equivalent program in another language i.e target language. Target code can be Assembly code or machine code. An important role of the Compiler is to report any errors in the Source program that it detects during the translation process.



If the target program is an executable m/c code, it can then be called by the user to process inputs and produce outputs.



Interpreter: Is also a language processor. Instead of producing a Target Program as a translation, An interpreter appears to directly execute the operation specified in source program on inputs supplied by the user.

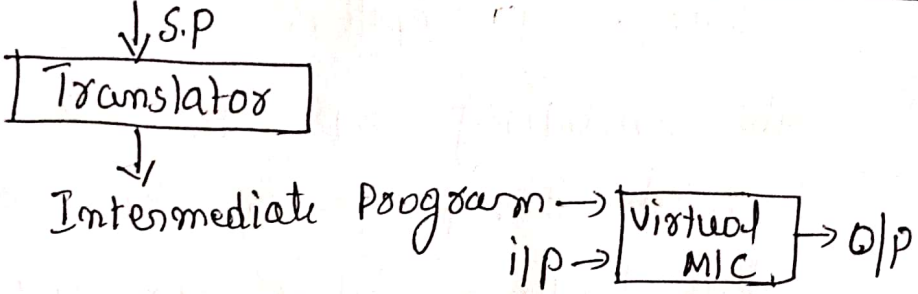


The m/c lang Target program produced by a compiler is usually much faster than an interpreter at mapping input to output. An interpretin, however, can usually give better error diagnostics than a compiler.
It executes source program statement by statement.

Hybrid Compiler

Java language processors combine compilation and Interpretation as shown below. a Java S.p may first be compiled into an intermediate form called byte codes. The byte codes are than interpreted by a virtual m/c.

Hybrid compiler



Interpreter Vs Compiler.

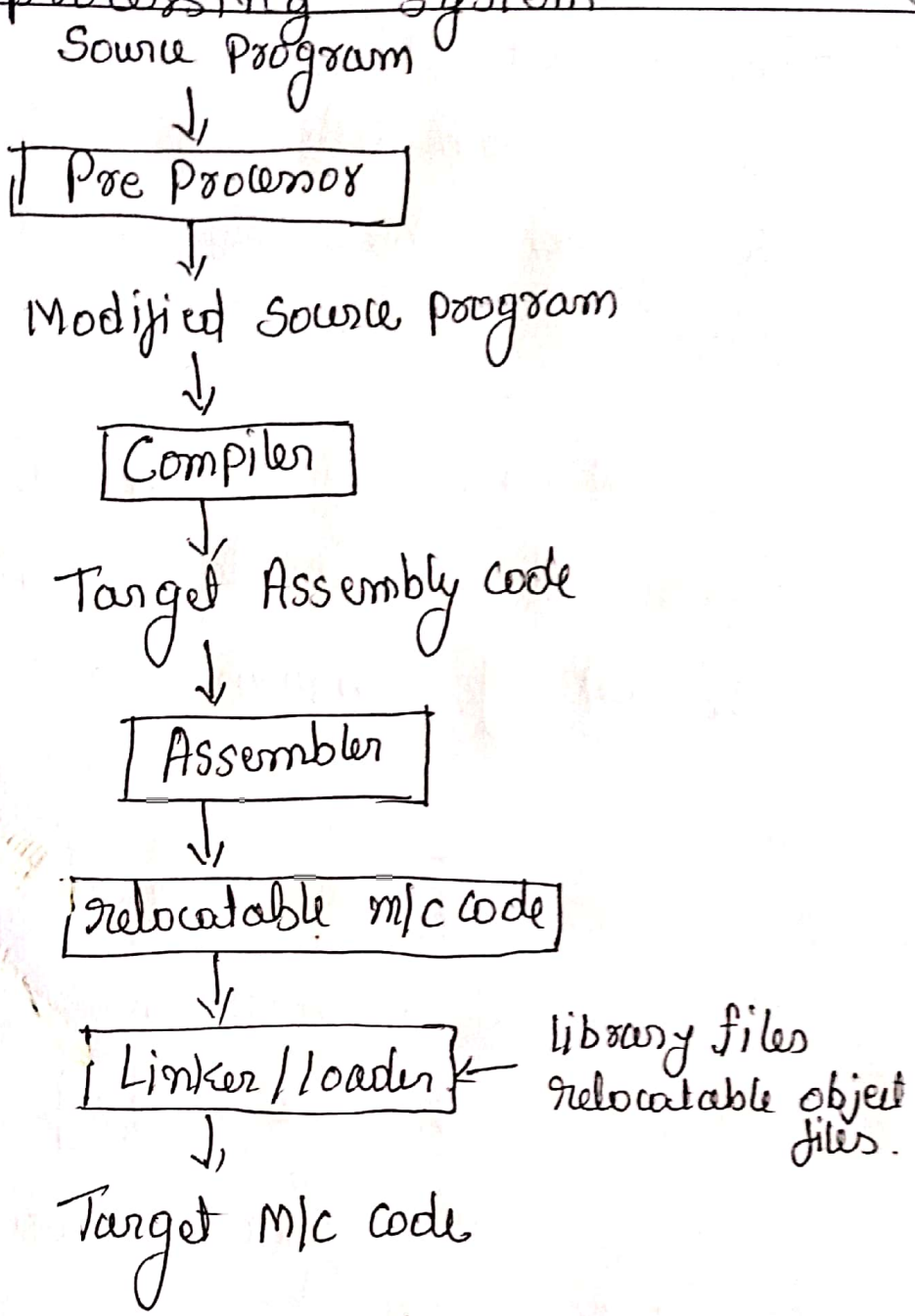
* Compiler translate the entire program into target program & then it executes it. Where as interpreter executes instruction of sp one by one.
* Compilers are 5 times faster than the interpreter.
* Interpreter requires much more memory than the m/c code generated by the compiler.
* Interpreters are easy to write and can provide better error message.
* Compiler produces efficient object code where as no intermediate code is generated by interpreter.
* Difficult to use compiler for beginners compare to interpreter.

Cousines of Compilers

while Converting S.P into T.P in addition to compiler we need other programs like preprocessor, assemblers, loaders, linkers, editors as shown below.

Preprocessor expands macros, brings different files which has different modules of same application together. The modified S.P is then fed to a Compiler. The Compiler may produce an assembly language Program as its output. The assemble will process this assembly code & produces relocatable m/c code.

Large programs are often compiled in pieces, so the relocatable m/c code may have to be linked together with other relocatable object files and library files into the code that actually runs on the m/c. The linker resolves external memory address, where where the code in one file may refer to a location in another file. The loader then puts together all of the executable object files into memory for execution.



Structure of a Compiler:

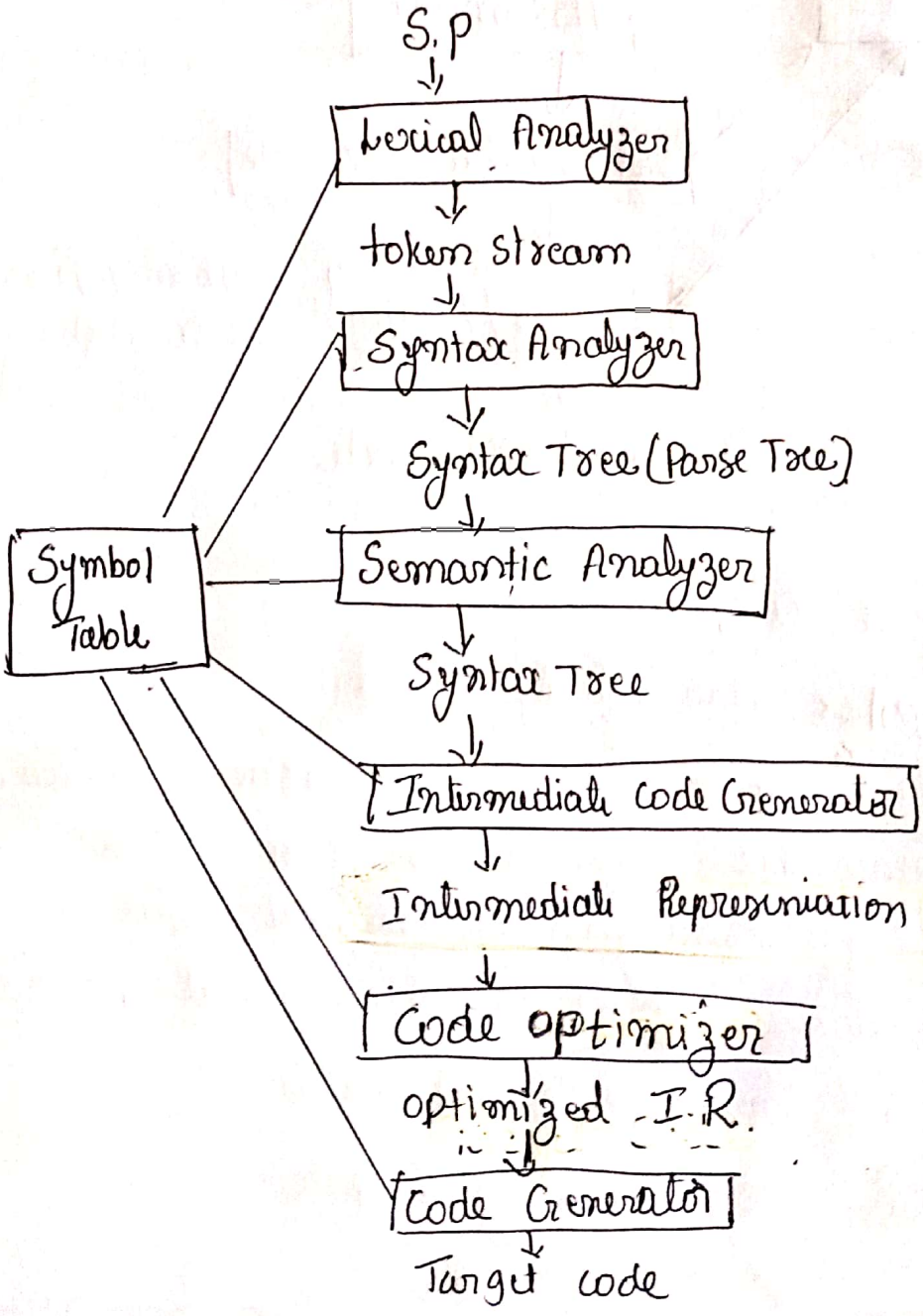
The Compiler has mainly 2 parts.

1) Analysis: Breaks up the S.P into constituent pieces and imposes a gramatical structure on them. Then it uses this structure to create an intermediate representation of S.P. It also collects information about program and stores it in the "Symbol table".

It is called front end of compiler.

② Synthesis Part: It constructs the desired target p... from the intermediate representation and the info... in the symbol table. This is called Back end of the Compiler.

The different phases of a compiler. The 1st 3 phases forms analysis part and last 3 phases forms synthesis ~~Phase~~ Part of compiler.



1) Lexical Analysis phase

The lexical Analyzer reads the stream of characters ~~makt~~ from the source program and groups the characters in to meaningful sequence called lexemes (units).

For each lexeme, the lexical Analyzer produces a token of the form <token name, attribute value> & passes it into 2nd phase of compiler i.e Syntax Analysis. The first component of token is token name which is an abstract symbol used during parsing. The second component is attribute value that points to an entry in the symbol table for this token.

ex: $a = b + c * 12$

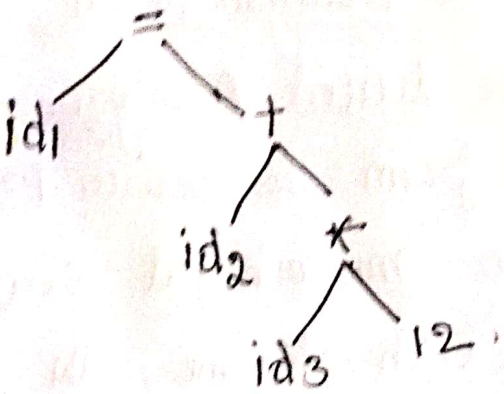
<id1, 1>, <=>, <id2, 2>, <+>, <id3, 3>, <*> <60>

2) Syntax Analysis (Parsing) : It uses 1st component of the token produced by lexical analyzer. to create a tree like intermediate representation that depicts the gramatical structure of the token stream. It is called Syntax tree, where internal nodes represents operation and the children nodes represents arguments of the operation. This phase is used to find syntax errors in S.P. It uses Context Free Grammar for generation of Parse tree.

ex:

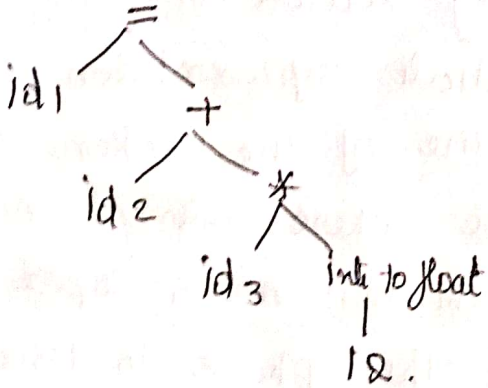
$E \to E \mid id \mid E+E \mid E*E \mid num$

↑ Grammar for Expression



3) Semantic Analysis: This phase uses syntax tree and the information in the symbol table to check the source program for Semantic errors. It also gathers type information and saves it in either the syntax tree or the symbol table, for subsequent use during intermediate code generation. The important part of semantic analysis is type checking, where the compiler checks that each operator has matching operands. If the operator is applied to floating pt and integer, the compiler may convert or coerce int to floating pt number.

ex:



Intermediate code generation:

After syntax & semantic analysis, some compilers generate an explicit intermediate ~~representation~~

representation of the source program. which is a program for an abstract machine. This intermediate representation must have 2 imp properties; it should be easy to produce, & easy to translate into the target program.

The intermediate representation can have a variety of forms. one of which is an "Three address code" which is like an assembly language for a m/c in which every memory location can act like a register. Three address code consists of a sequence of instructions, each of which has at most 3 operands.

ex: for the statement $\langle id,1\rangle := \langle id,2\rangle + \langle id,3\rangle * 60$

$$\left.\begin{array}{l} temp1 := int\ to\ real(60) \\ temp2 := id3 * temp1 \\ temp3 := id2 * temp2 \\ id1 := temp3. \end{array}\right\} \rightarrow (1)$$

This intermediate form has several properties.

① each instruction has at most one operator in addition to the assignment.

② compiler must generate a temporary name to

hold the value computed by each instruction

③ Some "three address" instructions have fewer than 3 operands eg. 1st & last instruction

In general, these representations must do more than compute expressions, they must also handle flow-of-control constructs & procedure calls.

Code optimization:

This phase attempts to improve the intermediate code, so that faster-running machine code will result. Some optimizations are trivial eg., a natural algorithm generates the intermediate code, using an instruction for each operator in the tree representation after semantic analysis, even though there is a better way to perform the same calculation; using the 2 instructions.

```
temp1 := id3 * 60.0
id1   := id2 + temp1
```

(2)

There is nothing wrong with this algorithm, since the problem can be fixed during

during the code - optimization phase. i.e the compiler can deduce that the conversion of 60 from int to real representation can be done once & for all at compile time, so the inttoreal opt can be eliminated. temp3 is used only once, to transmit its value to id1, & then becomes safe to substitute id1 for temp3, whereupon the last statement of above is not needed & the ~~below~~ above code ~~is~~ results in (2)

[Note: It optimizes the code in terms of no. of instructions]

Code generation:

The final phase of the compiler is the generation of target code, consisting of relocatable m/c code or assembly code. Memory locations are selected for each of the variables used by the program. Then, intermediate instructions are each translated into a sequence of m/c instructions that perform the same task. eg above code (2) can be translated as shown below using 2 registers R1 & R2.

```
LDF   R2, id3
MOLF  R2, R2, #12.0
LDF   R1, id2
ADDF  R1, R1, R2
STF   id1, R1
```

1st operand is destination & F specifies floating point numb
specifies that 12.0 is a constant.

Symbol Table:

The S.table is a data structure containing a record for each variable name, with fields for the attributes of the name such as storage allocated, its type, its scope and in case of procedure names, number and types of its arguments, the method of passing each argument (By value or By reference) and the type returned.

ex:

1	a	Real	...
2	b	"	...
3	c	"	...

F = C * 1.8 + 32

↓

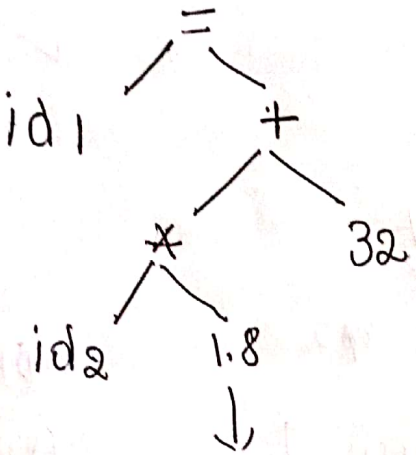
L.A

↓

⟨id1, 1⟩ ⟨=⟩ ⟨id2, 2⟩ ⟨*⟩ ⟨1.8⟩ ⟨+⟩ ⟨32⟩

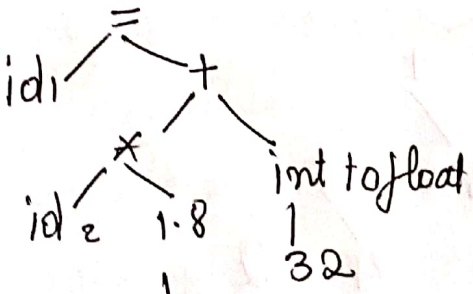
↓

S.A



↓

Semantic Analysis



↓

$t_1 = id_2 * 1.8$

$t_2 = int\ to\ float\ (32)$

$t_3 = t_1 + t_2$

$id_1 = t_3$

↓

code optimizer

↓

$t_1 = id_2 * 1.8$

$id_1 = t_1 + 32.0$

↓

code generator

↓

LDF R_2, id_2

MOLF R_2, R_2, 1.8

ADDF R_2, R_2, 32.0

STF id1, R_2

2) a[index] = index + 4 + 2.

↓

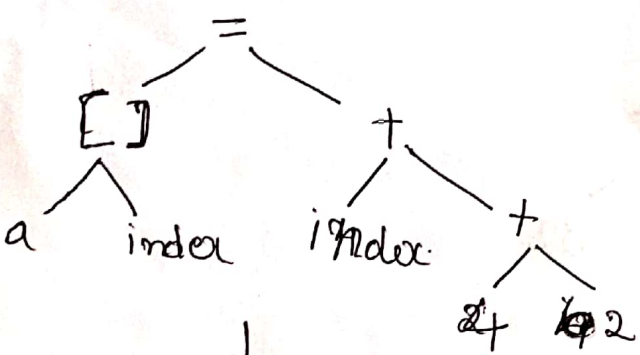
Lexical Analysis

↓

<a, 1>, <[>, <id2, 2>, <]>, [=], <id2, 2>, <+>, <4>, <+>, <2>

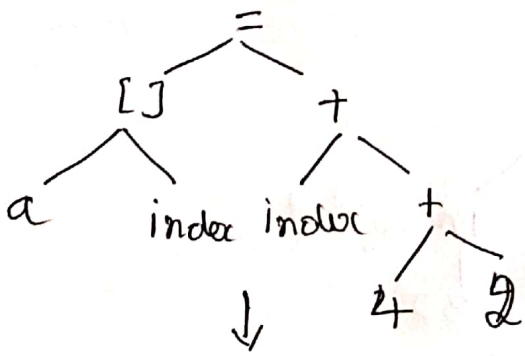
↓

Syntax Analysis



↓

Semantic Analysis



↓

Intermediate code Generation

$t_1 = 4 + 2$

$t_2 = id_2 + t_1$

$id_1[index] = t_2$.

↓

Code optimization

$id_1[index] = 6 + id_2$.

↓

Code Generation

↓

MOV index, R_0

MOV #a, R_1

add R_0, R_1

MOV #6, R_2

add R_0, R_2

MOV R_2, R_1

Grouping of Phases into Passes

The discussion of phases deals with the logical organization of a compiler. In an implementation, activities from diff phases are grouped together into pass that reads an i/p file & writes an o/p file. ex 1st 4 phases are grouped into one pass & right as another pass.

Compiler Construction Tools

The writers (s/w developers) of compilers as to use diff. s/w tools such as lang editors, debuggers, version managers, profilers, test harnesses & so on. In addition to their general s/w development tools, often more specialized tools have been created to help implement various phases of compilers.

1. Parser generators: that automatically produce syntax analyzers from a grammatical description of a P.L.
2. Scanner generators: produces lexical analyzers from a R.E description of the tokens of a lang
3. Syntax-directed translation engines: that produce collection of routines for walking a P.T & generating intermediate code.
4. Code-generator: produces a code generator from a collection of rules for translating each opt of the intermediate lang into M/C lang
5. Data-flow analysis engines: gathers info about how values are transmitted from one part of a prog to

each other part. It 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.

The Evolution of P.L:

* During 1950's 1st people friendly P.lang developed is mnemonic assembly lang.
* Then the high level lang like Fortran for scientific computation, Cobol for business d/a processing & Lisp for Symbolic computation. as been developed during holy of 1950's. The philosophy behind these lang was to create higher level notations with which programmers could more easily write numerical computations, bussiness applications & Symbolic programs.
* Then usualy these diff lang are classied among diff generations.

1st generation languages are m/c lang, 2nd gen the assembly lang, 3rd generation the higher level lang like Fortran, cobol, lisp, C, C++, Java C#. 4th gener lang are lang designed for specific applications like NOMAD for report generating, SQL for d/b queries & postscript for text formatting. The 5th gen lang are logic & constraint based lang like prolog & OPS5

one more way we can classify the lang as Imperative lang & declarative lang, object oriented lang, scripting lang & Java script, Perl, PHP, Python, Ruby & Tcl. programs written in scripting lang are much shorter than equivalent prog written in lang like C.

Imperative lang → C, C++, C#
declarative lang → ML, Haskell, Prolog
object oriented lang → C++, C#, Java

The Science of Building a Compiler.

Modeling in CD & Implementation

Some of fundamental models used in CD are

* R.E which describes lexical units of prog (k.w, identifiers etc) & also describes algorithms used by the compiler to recognize those units.
* CFG: used to describe syntactic structure of P.L. such as the nesting of parenthesis or control constructs.
* Trees are also imp models for representing the structure of prog & their translation into object code.

The Science of code optimization:

The 'optimization' in CD refers to the attempts that a compiler makes to produce code that is more efficient than the obvious code.

The compiler optimization must meet the following design objectives.

- The optimization must be correct, i.e preserve the meaning of the compiler prog.

- The optimized must improve the performance of many
- The compilation time must be kept reasonable.
- The engineering effort required must be manage

The use of a rigorous mathematical foundation allows us to show that an optimization is correct & that it produces the desirable effect for all possible i/p's. The diff models ~~use~~ [such as graphs, matrices, & linear programming are necessary if the compiler is to produce well optimized code.]

Applications of Compiler Technology:

1. Implementation of High-level P.L: [C, C++, Java etc]
2. Optimizations for Computer architectures.

(a) Parallelism: It can be found at several levels at the Instruction level where multiple opt are executed simultaneously and at the processor level, where diff threads of same application are run on diff processors. All modern mp have instruction level parallelism. where as this parallelism is hidden from the programmer. progs are written as if all inst were executed in sequence; the h/w checks for dependencies & issues them in parallel when possible. In some cases there will be ~~H/w h/w~~ H/w scheduler that can change the inst ordering to increase the parallelism in the program. whether the h/w reorders the inst or not, compiler can rearrange the inst to make instruction level parallelism more effective.

Memory Hierarchies:

① M·H consists of several levels of storage with diff speeds & sizes. with the level closest to the processor being the fastest but smallest.

A processor usually have small no of registers consisting of hundreds of bytes, several levels of caches containing kilobytes to megabytes, physical memory containing mega bytes to gigabytes & finally secondary storage that contains gigabytes & beyond. correspondingly the speed of access b/w adjacent levels of the hierarchy also differes. while compilers traditionally focus on optimizing the processor execution, more emphasis is now placed on making the memory hierarchy more effective.

② Design of New Computer Architecture

In the early days of computer architecture design, compilers were developed after the m/c were built. Now, that has been changed because the performance of a computer s/m is determined not by its raw speed but also by how well compilers can exploit its features. Thus in modern computer architecture development, compilers are developed in the processor design stage. Ex: of how compiler influenced the ⁱⁿ of Computer Architecture was the invention of the RISC (Reduced Inst set computer) architecture over last 3 decades, many architectural concepts have

been proposed like d/a flow m/c's, vector m/c, [very long Inst word] m/c, SIMD (Single inst, multiprocessor with shared & distributed memory. The dev development of each of these architectural concepts was accompanied by the research & development of corresponding compiler tech.

④ Program Translation

The compiler not only used to convert H.L.L to LLL it is also used to translate b/w diff kinds of lang. ex.

ⓐ Binary Translation:

Compiler Tech can be used to translate the binary code for one m/c to that of another, allowing a m/c to run prog originally compiler for another inst set. Binary Translation Tech has been used by various computer companies to increase the availability of s/w for their m/c. In particular, because of the domination of the X86 PC market, most s/w titles are available as X86 code. Binary Translators have been developed to convert X86 code into both Alpha & Sparc code.

ⓑ H/w Synthesis:

How the most s/w's written in HLL, even H/w design are mostly described in HLL like Verilog & VHDL Techniques to translate design at higher levels, such as the behavior of an ckt also exist.

ⓒ Database Query Interpreters:

as SQL are used to search db. DB queries consist of predicates containing relational & boolean operators.

they can be interpreted or compiled into commands to search a db for records satisfying that predicate.

Compiled Simulation:

Simulation is a general tech used in many scientific & engineering disciplines to understand a phenomenon or to validate a design. Inputs to a simulator usually include the description of the design & specific i/p parameters for that particular simulation run. Simulations can be very expensive as takes more time. Instead of writing a simulator that interprets the design, it is faster to compile the design to produce m/c code that simulates that particular design natively. Compiled simulation can run orders of magnitude faster than an interpreter-based approach. Compiled simulation is used in many state-of-the art tools that simulate designs written in verilog or VHDL.

(5) S/w productivity Tools:

The diff prog we write may contain diff errors. Testing is the primary tech for locating errors in programs. one more approach is to use data flow analysis to locate errors statically. It finds errors in all possible execution paths. Many of the data flow analysis techniques, originally developed for compiler optimization can be used to find errors like type checking, Bounds checking, memory management errors.

Programming lang Basics

① Parameter Passing Mechanisms:

Call-by-value: Here, the actual parameter is evaluated (if an exp) or copied (if variable). The value is placed in the location belonging to the corresponding formal parameter of the called procedure. This method is used in C, C++ & Java.

Here changes made to the formal parameter does not effect the actual parameter. But if we pass name of array like a then if we do x[i]=2 it also changes a[i]=2 in actual parameter ∵ array name is ptr. Same case in Java when we pass object name.

Call-by-reference: Here the address of the actual parameter is passed to the callee as the value of the corresponding formal parameter. The changes made to the formal parameter also effects the actual parameter ∵ here the address of actual parameters are sent.

If the actual parameter is an expression, however then the exp is evaluated before the call, & its value stored in a location of its own. changes to the formal parameter change the value in its location, but can have no effect on the d/a of the caller.

call by name: A 3rd mechanism - call-by-name was used in the early programming lang Algol 60. It requires that the callee execute as if the actual parameter were substituted literally for the formal parameter in the code of the callee, as if the formal parameter were a macro standing for the a.p (with renaming of local names in the called procedure, to keep them distinct) when the A.p is an expression rather than a variable, some unintuitive behaviour occur, which is the one reason this mechanism is not

Programming Lang Basics

* ~~Static~~/ Dynamic Distinction. → with in class.

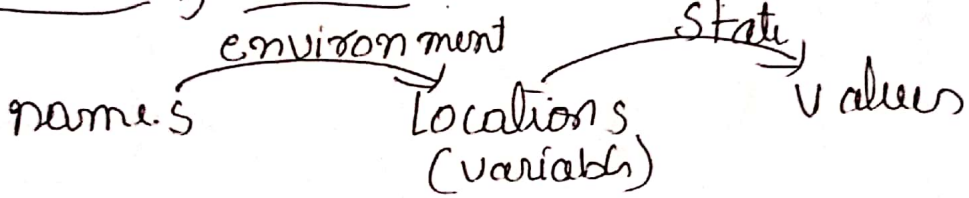
Public static int X;

Static scope means only one the memory is allocated & it is fixed any updations we are doing will affect only that location.

Dynamic ~~sta~~ scope means several declaration of x.

The objects of class will have diff copies of x ~~x. locations~~

Environments & Statis



when program runs we need to check whether ~~changes~~ it affects the value of data elements of) affects names of d/a. ex $X = Y + 1$ changes t value denoted by X.

Environments change according to the scope rules of a lang.

```
....
int i;          // global i
..
void f(...)
{ int i;        // local i
  ..
  i=3;          // use of local i
  ...
}
X = i+1;        // use of global i
```

environment & state mapping are usually dynamic. only if we used static i.e. or global variable then environment mapping is static. & the state mapping is dynamic ; we can't tell the value in a location until we run the program. but if we have declaration like #define Arraysize 1000

This is static state mapping.

Static scope & Block structure :

blocks in C & C++ are represented by { & }. the blocks is the grouping of declarations & statements.

```
main()
{ int a=1;                                   B1
  int b=1;
  {
      int b=2;                               B2
      {  int a=3;                            B3
         cout<<a<<b;
      }
      {  int b=4;                            B4
         cout<<a<<b;
      }
      cout<<a<<b;
  }
  cout<<a<<b;
}
```

Declaration	Scope
int a=1;	$B_1 - B_3$
int b=1;	$B_1 - B_2$
int b=2;	$B_2 - B_4$
int a=3;	B_3
int b=4;	B_4

Exercise: 1.6.1 For block structured code below indicate the values assigned to ~~values~~ w, x, y, z.

```
int w, x, y, z;
int i=4; int j=5;
{ int j=7;
  i=6;
  w=i+j; ⇒ (13)
}
x=i+j; (11)
{ int i=8;
  y=i+j; (13)
}
z=i+j; (9)(11)
```

```
int w, x, y, z;
int i=3; int j=4;
{ int i=5; (9)
  w=i+j;
}
x=i+j; (7)
{ int j=6;
  i=7;
  y=i+j; (13)
}
z=i+j; (11)
```

```
#define a (a+1)
int x=2;
void b() { int x=1; printf("%d\n", a); }
void c() { printf("%d\n", a); }
void main() { b(); c(); }
```

o/p 2, 3.

Lexical Analysis

This chapter deals with techniques for specifying & implementing lexical Analyzers. A simple way to build L.A is to construct a diagram that illustrates the structure of the tokens of the source lang. & then to hand-translate the diagram into a program for finding tokens.

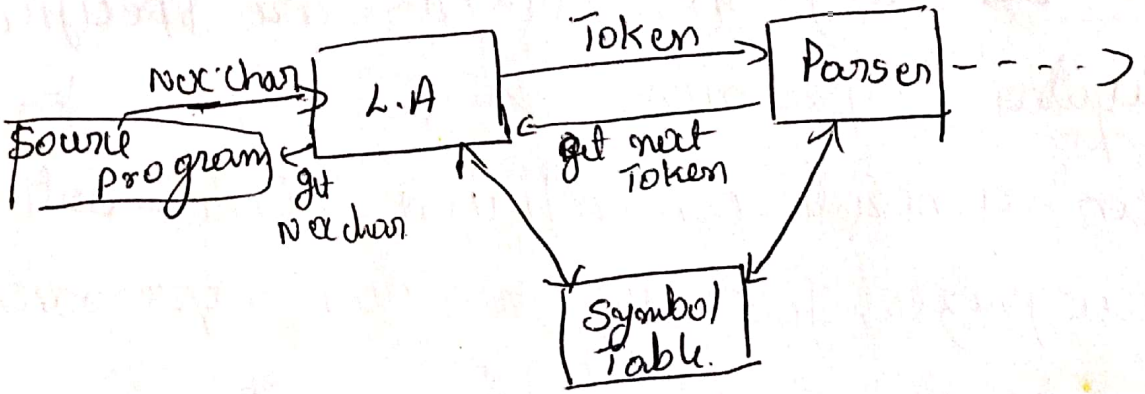
Since Pattern directed programming is widely useful, we introduce a pattern-action lang called Lex for specifying lexical analyzers. In this language, patterns are specified by Regular expressions, & a compiler for Lex can generate an efficient Finite-automaton recognizer for the regular expressions.

A software tool lexical-analyzer generator can be used for constructing the L.A. which uses best known pattern-matching algorithms & thereby creates efficient L.A's for people who are not experts in pattern-matching techniques.

The Role of the lexical Analyzer:

It is the 1st Phase of a Compiler, which reads the i/p characters & produce as o/p a sequence of tokens that the parser uses for syntax analysis. Usually it is implemented as a subroutine or co-routine of a ~~Comp~~ Parser. Upon ~~receiving~~ receiving a "get next token" command from the Parser, the L.A reads the i/p characters until it can identify the next token.

Interaction b/w L.A & Parser



Since L.A is a part of Compiler that reads the source text, it may also perform certain secondary tasks at the user interface like stripping out from the S.P comments, white space in the form of blanks, tab & newline characters, keeps track of number of new

The Programming language is defined by

1. Syntax

Decides whether sentence in a lang is well formed.

2. Semantics

Determines the meaning, if any, of a syntactically well formed sentence.

3. Grammer:

A formal s/m that provides a generative finite description of the lang

Lexical Analyzer or Parser of a compiler handle the syntax of the Programming Language.

Token examples

Let us consider the program segment

```
void main()
{
printf("Hello world\n");
}
```

The tokens of this program segment are:

1. void,
2. main,

Token Type	Example
ID	foo, n14, a, temp...
NUM	73, 0, 00, 515, +2...
REAL	66.1, .5, 10. 1e67 5.5e-10...
KEYWORDS	IF DO WHILE INT...
SYMBOLS	, (Comma), != (Not equal), ((Lparand)...

How are tokens formally defined & Recognized?

By using R-E to define a token as a formal Regular language.

A language in real life is made of

1. words made up of alphabets &
2. Sentences made up of words arranged according to the grammar of that lang

Formal def of languages.

(E) Alphabet → A Finite (non-empty) set of symbols

(Σ) String → A finite sequence of symbols from an alphabet which includes even the empty sequence. (ε)

language → A set of finite strings. The set of all possible finite strings of elements of alphabet Σ (including) is denoted by Σ^*.

Finite specification of language is possible with

1. Automaton: a Recognizer; a m/c that accepts all strings in a language (& rejects all other strings)

2. Grammar: a generator; a sys for producing all strings in the lang (& no other strings)

Note: A lang may be specified by many diff grammars & automata But. A grammar or automaton specifies only one lang.

line characters seen so that a line number can be associated with an error message.

In some compilers L.A is in charge of making a copy of S.P with the error m/g's marked in it. If the S.P supports some macro Processor fun, then this preprocessor fn may also be implemented as L.Analysis takes place

L.A are divided into a cascade of 2 Phases "Scanning" & "Lexical Analysis": where Scanner performs simple fn like eliminate blanks from i/p & L.Analysis generates tokens.

Issues in Lexical Analysis:

There are several reasons for separating the analysis phase of compiling into ~~L.A & S.P~~ Lexical Analysis & Parsing.

1. Simplify the design of Phases: This is an most imp consideration of seperation which makes the phases simple ex: a parser embodying the conventions for comments &

white space is significantly more complex one that can assume comments & white sp have already been removed by a L.A.

2. Compiler efficiency is improved:

A seperate L.A allows us to construct a specialized & potentially more efficient process for the task. A large amount of time is spent reading the S.P & partitioning it into tokens. specialized buffering tech for reading i/p characters & processing tokens can significantly speed up the performance of a compiler.

3. Compiler portability is enhanced:

I/P alphabet peculiarities & other device-specific anomalies can be restricted to the lexical analyzer. The representation of symbols ↑ in pascal can be isolated in the L.A.

Tokens, Patterns, Lexemes:

In general, there is a set of strings in the i/p for which the same token is produced as o/p. This set of strings is described by a rule called a **pattern** associated with the token. The pattern is said to match each ~~each~~ string in the set. **A lexeme** is a sequence of characters in the sp i.e matched by the pattern for a token. (also called as lexical unit)

ex. In Pascal

literal.

```
Const  Pi = 3.1416;   [ printf ("Total = %d\n", score) ]
                        ID                      ID
```

the substring Pi is a lexeme for the token "identifier".

Token	Sample lexeme	Informal description of Pattern
const	Const	const
if	if	if
relation	<, <=, >=, =, <>, >	< or <= or > or >= or = or <>
id	Pi, count, D2	letter followed by letters & digits
num	3.14, 0, 10, 6.4	any numeric constant
literal	"core dumped"	any characters b/w " & " except "

Tokens are treated as terminal symbols in the grammar for the source lang, using boldface names to represent tokens. The lexemes i.e strings of chars in the s.p that can be treated together as a lexical unit.

In most P.L the following constructs are treated as tokens: keywords, operators, identifiers, constants, literal strings, & punctuation symbols such as parentheses, commas, & semicolons. In the example above when the character sequence pi appears in the s.p a token representing an identifier is returned to the parser. The returning of a token is often implemented by passing an integer corresponding to the token. It is this integer i.e referred to in Fig 3.2 as boldface id.

A pattern is a rule describing the set of lexemes that can represent a particular token in s.p. To represent Pattern of ~~some~~ tokens like num, identifier we use R.E.

arrangement of lexems may be imp in L. Analysis.

Blanks are treated in different ways in diff lang. & it will complicate the task of identifying tokens.

ex: In Fortran

DO 5 I = 1.25

we can't tell until we have seen the decimal point that DO is not a k.w but rather part of the identifier DO5I. on the other hand

DO 5 I = 1, 25

we have 7 tokens, corresponding to the k.w DO, the statement label 5, Identifier I, operator =, Constant 1, the comma & the Constant 25. Here we can't be sure until we have seen the comma that DO is a k.w.

Attributes for tokens:

when more than one pattern matches a lexeme, the lexical analyzer must provide additional info about the particular lexeme that matched to the subsequent phases of the compiler. ex, the pattern num

matches both the strings 0 & 1, but it essential for the code generator to know what string was actually matched.

The lexical analyzer collects info about tokens into their associated attributes. A token has only a single attribute i.e Pointer to the symbol table entry in which the info. about the token is kept; the ptr becomes the attribute for the token. We may be interested in both the lexeme for an identifier & the line number on which it was 1st seen. Both these items of information can be stored in the symbol table entry for the identifier.

ex. The tokens & associated attribute-values for the fortran statement.

$$E = M * C ** 2$$

< id, ptr to S.T entry for E>
< assign-op, >
< id, ptr to S.T entry for M>
< mult-op, >

<id, ptr to S T entry for C)

<exp-op,)

<num, integer value 2)

Note that in certain pairs there is no need for an attribute value, the 1st component is sufficient to identify the lexeme.

Lexical Errors:

Few errors ~~are~~ can't be detected by lexical anal alone, ∵ a L.A has a very localized view of a S.P. If the string fi is encountered in a C program for the first time in the context

fi(a==f(x)) . . .

A L.A can't tell whether fi is a misspelling of the K.W if or an undeclared fn identifier. Since fi is a valid identifier L.A must return token for an identifier & hdl some other phases of compiler handle any error.

Suppose L.A is unable to proceed ∵ none of the patterns for tokens matches a prefix of the remaining i/p.

The simplest recovery method is "panic mode recovery. we delete successive characters from the remaining i/p until the LA can find a well formed token.

Other possible error-recovery actions are

1. deleting an extraneous character
2. Inserting a missing character
3. Replacing an incorrect character by a correct character. ~~by a correct character~~
4. Transposing two adjacent characters.

error transformations like these may be tried in our attempt to repair the i/p. The simplest method is to see whether a prefix of the remaining i/p can be transformed into a valid lexeme by just a single error transformation. This strategy assumes most lexical errors are the result of a single error transformation, an assumption usually but not always, borne out in practice.

One way of finding the errors in a program is to compute the minimum no of error transformations required to t/s the erroneous prog into one i.e syntactically well-formed.

Input Buffering:

There are 3 general approaches to the implementation of a L.A.

1. Use a L.A generator, S.A Lex Compiler to produce the L.A from a R.E based specifications. Here generator provides routines for reading & buffering the i/P.
2. Write the L.A in a conventional s/m programming language, using the I/O facilities of that lang to read the i/p.
3. Write the L.A in assembly lang & explicitly manage the reading of i/p.

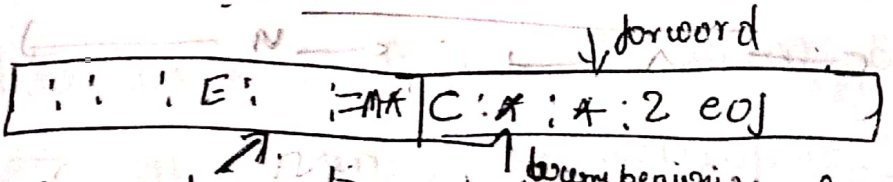
Since L.A is the only phase of the compiler that reads the SP character by character, it

is possible to spend a considerable amt of time in the lexical Analysis phase, even though the later phases are conceptually more complex. Thus the speed of L. Analysis is a concern in compiler design. & we use 1st method i.e generators to implement L.A.

Buffer pairs:

For many source lang, there are times when the L.A needs to look ahead several characters beyond the lexeme for a pattern before a match can be announced. There are diff specialized buffering techs are available but we will see one such tech.

We use a buffer divided into 2N-character halves. where N is the number of characters one one disk block. ex 1024 or 4096



Read N i/p characters into each half of the buffer with one read command instead of using read command for each i/p character

If fewer than N characters remain in the i/p then a special character eof is read into the buffer after the i/p characters.

Two ptrs to the i/p buffer are maintained. The string of characters b/w the 2 ptrs is the current lexeme. Initially both ptrs point to the 1st character of the next lexeme to be found. One called forward ptr, scans ahead until a match for a pattern is found. once the next lexeme is found the forward ptr is set to the character at its right end. After the lexeme is Processed, both ptrs are set to the char immediately past the lexeme, with this scheme the comments & the white space can be treated as patterns that yield no token.

If the forward ptr is about to move past the halfway mark, the right half is filled with N new i/p characters

If the forward ptr is about to move past the right end of the buffer, the left half is filled with N new characters & the forward ptr wraps around to the beginning of the buffer.

This buffering scheme works quite well most of the time, but with it the amount of lookahead is limited, & this limited lookahead may make it impossible to recognize tokens in situations where the distance that the forward ptr must travel is more than the length of the buffer. ex if we see

DECLARE (ARG1, ARG2 ... ARGm) E

in a PL/I program, we can't determine whether Declare is a K.W or an array name until we see the character that follows the right parenthesis. In either case, the lexeme ends at the second E, but the amount of lookahead needed is ∝ to the number of arguments,

which in principle is unbounded.

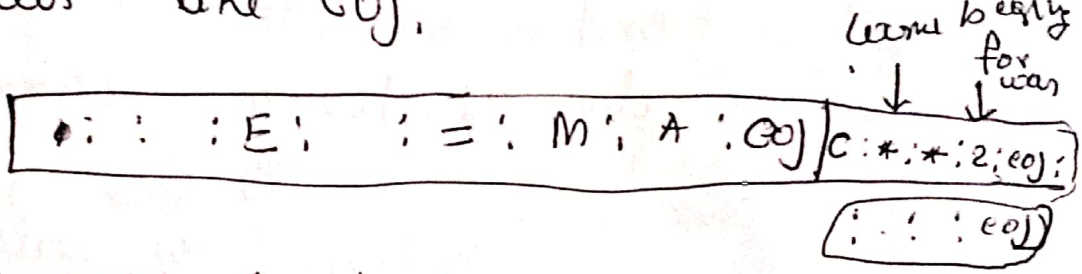
Code to advance forward ptr is

```
if forward at end of first half then
                                    begin
    reload second half;
    forward := forward + 1
end
else if forward at end of 2nd half then
                                    begin
    reload 1st half
    move forward to beginning of
                                    1st half.
end
else forward := forward + 1
```

Sentinels:

If we use the above scheme then we must check each time we move the FP that we have not moved off one half of the buffer; if we do, then we must reload the other half. i.e our code for advancing the forward ptr performs tests like above.

Except at the ends of the buffer the code above requires 2 tests for each advance of the forward ptr. we can reduce the 2 sets to one if we extend each buffer half to hold a sentinel character at the end. ~~The sen~~ i.e special character like eof.



with this arrangement ~~of fig~~ we can use below code to advance the F.p.

```
forward := forward + 1;
if forward↑ = eof then begin
  if forward at end of 1st half then
                                  begin
      reload 2nd half;
      forward := forward + 1
  end
  else if forward at end of 2nd half
                                  then begin
      reload 1st half;
      move forward to beginning of 1st half
  end
  else /* eof within a buffer signifying end of i/p
      terminate L. Analysis
end.
```

we also need to decide how to process the characters scanned by the forward ptr; does it mark the end of a token, does it represent progress in finding a particular token or what? One way to structure these tests is to use a case statement, if the implementation lang has one. The test

if forward ↑ = eof

can be implemented as one of the diff cases.

Specifications of tokens

Patterns are represented by Regular Expression.

Some ~~Basic~~ formal definitions of language

Alphabet [character] [Σ]

It is a finite (non empty) set of symbols.

where symbols can be ~~characters~~ digits or punctuations or letters.

The set {0, 1} is the binary alphabet.

ASCII & ~~EBCDIC~~ EBCDIC are 2 examples of computer alphabets.

(1) String: It is a finit sequence of symbols from a alphabet. It is also know as Sentince or a word. |s| is the length of string i.e the number of occurrences of symbols in s. ex banana the |s| = 6. The empty string, denoted by ϵ, is a special string of length zero.

language [Σ^*]

A set of finite strings over some fixed alphabets. The set of all possible finite strings of elements of alphabet Σ is denoted by Σ^*. The Abstract lang like ϕ, the empty set, or {ϵ} the set containing only the empty string. are called.

If x & y are strings, then Concatenation is given by xy. ex x = dog y = house then xy = doghouse. The empty string is the identity element under concatenation.

i.e $S\epsilon = \epsilon S = S$.

If we think concanting notion as a product then a string "exponentiation" is defined as follows. Define S^0 to be ϵ & for i > 0 define S^i to be $S^{i-1}S$. Since ϵS is S itself, $S^1 = S$. Then $S^2 = SS$ $S^3 = SSS$ & so on.

ex of language

1) language of all strings consisting of n 0's followed by n 1's for some $n \geq 0$ is

$\{\epsilon, 01, 0011, \ldots\}$.

2) ϕ is empty language, but $\phi \neq \{\epsilon\}$ where former is empty language & latter is a language with empty string.

3) set of binary numbers whose value is a prime

$\{10, 11, 101, 111, 1011 \ldots\}$.

Some common terms associated with parts of a string are given below.

Term	Definitions
Prefix of s	A string obtained by removing zero or more ~~trailing~~ symbols from the ~~end~~ of strings ex ban is a prefix of banana.
Suffix of s	A string obtained by removing 0 or more ~~leading~~ symbols from beginning of s. ex nana is suffix of banana

substring of S : A string obtained by deleting prefix & a suffix from S; ex ban is a substring of banana. every prefix & every suffix is a substring, but not every substring of S is a prefix or a suffix of S. for every string S, both S & ∈ are prefix, suffixes & substrings of S.

Proper prefix, suffix or substring of S

Any non empty string x i.e respectively, a prefix, suffix or substring of S s.t $S \neq x$.

Subsequences of S : Any string formed by deleting 0 or more not necessarily contiguous symbols from S. ex baaa is a subsequence of banana.

operations on languages:

operations	Definitions
union of L & M (L∪M)	$L \cup M = \{S \mid S \text{ is in } L \text{ or } S \text{ is in } M\}$
Concantination of L & M (LM)	$LM = \{St \mid S \text{ is in } L \& t \text{ is in } M\}$
Kleen closure of L (L^*)	$L^* = \bigcup_{i=0}^{\infty} L^i$ [L^* denotes "zero or more concantination of" L
+ve closure of L (L^+)	$L^+ = \bigcup_{i=1}^{\infty} L^i$ L^+ denotes "one or more concantination of L"

Ex: Let L = {A, B, ... Z, a, b, ... z} & D = {0 ... 9}

Some new languages created from L & D by applying different operations are

1. L ∪ D is the set of letters & digits
2. LD is the set of strings consisting of a letter followed by a digit.
3. L^4 is the set of all 4 letter strings
4. L^* is the set of all strings of letters including ∈, the empty string.
5. $L(L \cup D)^*$ is the set of all strings of letters & digits beginning with a letter.
6. D^+ is the set of all strings of one or more digits.

Regular Expressions: The text strings can be represented by Regular Expressions. R.E can be described by Finite Automata but style & notations are different.

The R.E are another type of language defining notation. R.E n denotes a language $L(n)$.

R.E are built recursively out of smaller R.E. using below rules.

Basis: 1) ϕ is a R.E denoting an empty language $L(\phi) = \emptyset$.

2) G is a R.E denoting $L(\epsilon) = \{\epsilon\}$ i.e lang with empty string.

3) a is a R.E denoting lang containing only $\{a\}$ i.e $L(a) = \{a\}$

Induction:

Suppose r & s are R.E denoting $L(r)$ & $L(s)$ respectively then

1) $(r)|(s)$ is a R.E denoting the lang $L(r) \cup L(s)$
2) $(r)(s)$ " " " $L(r)L(s)$
3) $(r)^*$ " " " $(L(r))^*$
4. (r) is a R.E denoting $L(r)$. it says we can add additional pairs of Parentheses around exp without changing the lang they denote.

R.E may also contain some unnecessary pairs of parentheses. we may drop certain pairs of parenthesis by adopting some conventions like

1) unary * has highest precedence & is left associative
2) Concatenation has 2nd highest precedence & is left associative
3) | has lowest precedence & left associative

ex: $(a)|((b)^*(c)) \Longrightarrow a|b^*c$

both represent same lang i.e string that are either a single a or zero or more b's followed by one c.

Let $\Sigma = \{a, b\}$

1. The R.E a/b denotes the lang $\{a, b\}$
2. $(a/b)(a/b)$ denotes $\{aa, ab, ba, bb\}$ lang of all string of length 2 over Σ.
3. a^* denotes the lang consisting of all strings of 0 or more a's i.e $\{\epsilon, a, aa..\}$
4. $(a/b)^*$ denotes the set of all string consisting of zero or more instances of a or b i.e all string of a's & b's $\{\epsilon, a, b, aa, ab, bb, ba, aaa...\}$
   OR $(a^*b^*)^*$
5. a/a^*b denotes the lang $\{a, b, ab, aab, aaab...\}$ i.e the string a & all strings consisting of zero or more a's & ending in b.

A lang that can be defined by R.E is called Regular set. suppose if 2 R.E r & s defines same lang then we say both are equivalent $r = s$ or $(a/b) = (b/a)$

Regular Definitions:

We can use names to certain R.E. eg:

id → letter (letter | digit)*
letter → A|B|..|Z|a|b|..|z|-
digit → 0|1|...|9

Unsigned numbers (integer or Fpt) are strings like such as 5280, 0.01234, 6.336E4 or 1.8E-4
The Regular definitions.

digits → digit digit*
optionalFraction → . digits | ∈
optionalExeption → (E(+|-|∈) digits | ∈
number → digits optionalFraction optionalExponent

1. is not valid.

using shorthands we can rewrite the regular def as.bel

letter → [A-Za-z-]
digit → [0-9]
id → letter (letter | digit)*

&

digit → [0-9]
digits → digit+
number → digits (. digits)? (E[+-]? digits)?

Recognition of tokens:

Here Symbols characters

Take the patterns for all needed tokens & build a piece of code that examines the i/p string & finds a prefix that is a lexeme matching one of the Patterns.

ex:
```
stmt → if expn then stmt
     | if expr then stmt else stmt
     | ∈
expn → term relop term
     | term
term → id
     | number.
```

The terminals of the grammar, which are if, then, else, relop, id & number are the names of tokens as far as the L.A is concerned.

Patterns for

```
if → if
else → else
then → then
relop → <|>|<=|>=|=|<>
```

Lexical Analyzer skips the white spaces by recognizing the "token" ws defined by:

ws → (blank | tab | newline) +

Here blank, tab & newline are abstract symbols that we use to express the ASCII characters of the same names. when LA recognize the token as it will not return it to the parser, but rather restarts the lexical Analysis from the character that follows the white space.

Transition diagram (Finite Automata)

Before constructing an L.A 1st we have to convert the patterns (R.E) into transition diagrams which are deterministic i.e only one edge for one ilp from each state. (NFA)

The T.D have a collection of states (nodes) each state represents a condition that could occur during the process of scanning the i/p looking for a lexeme that matches one of the several patterns. We may think of a state as summarizing all we need to know about what characters we have seen b/w lexem begin ptr & forward ptr.

Each edges are directed from one state of the T.D to another, which are labeled by a symbol or set of symbols. If we are in state s & i/p symbol is a, we look for an edge out of state s labeled by a. if we find such an edge advance the forward ptr & enter the state of the transition diagram to which that edge leads.

Imp conventions about the transition diagrams are;

1. Certain states are said to be accepting or final. which indicates that lexeme has been found, although the actual lexeme may not consist of all positions b/w the lp & fp. represent it by double circle & attach action to be taken. to it.
2. If it is necessary to retract the F.ptr one position (i.e the lexeme does not include the symbol that got us to the accepting state) then we additionally place a * near the accepting state.
3. one state is considered as start state or initial state it is indicated by an edge labeled

~~#include~~

main( )

{ ~~dec~~ int a, b
~~Printf ("   ")~~
}

a = b+10 if (a>10
Printf (" %d ", a)
else Printf (" Fail")
}

Str -> if exp then stat

Prog -> #include main(){<decla list> Statlist}

decla-list -> <dec> | <dec-list> ; <dec>

dec -> ~~<id-list> :~~ <type> <id-list>

type -> Integer

id-list -> id | (id-list), id

Statlist -> stm | (stm-list), stmt

Stm -> <assign> | <printf> | if--else

assign -> id : = <exp>

exp -> | term | exp + term | exp - term

term -> Factor | term * Fact | term Div Fact

Factor -> id | int | (<exp>)

Printf -> Printf (" id-list")

if -> if (~~condition~~ ifexp) ~~then~~

ifexp -> ~~term~~ Factor relop ~~term~~ Factor Statm
| ~~term~~ Factor else stat | E

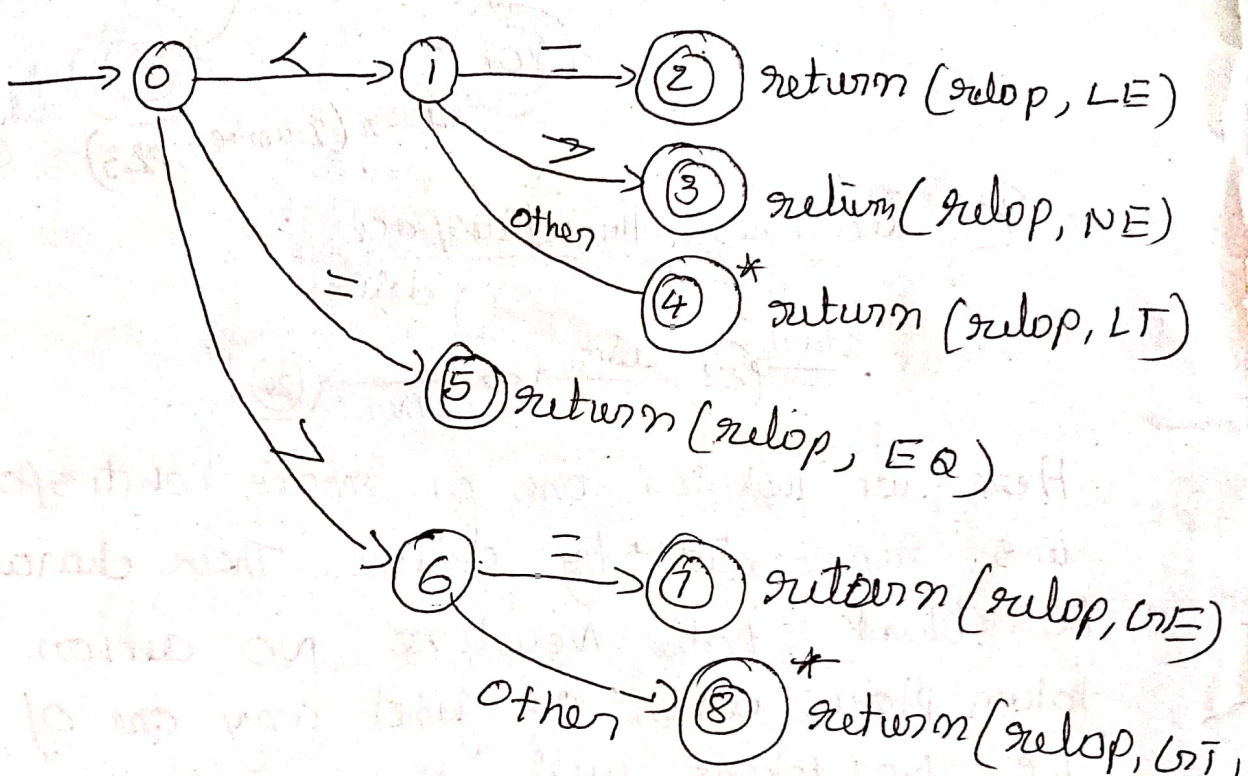
relop -> +| -| >| >=| <=| <|

Tokens
main()
~~Printf~~ PROGRAM (
~~main~~ {,
}
Int
if
else
printf
+
*
DIV
(
)
id
Factor
relop

netreat@yahoo.com

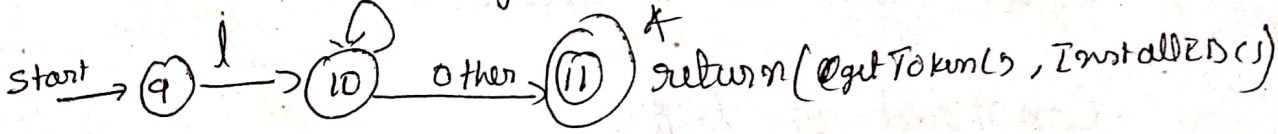
"start". The T.D always begins in the start state before any i/p symbols have been read.

Eg: T.D For ~~relation operator~~ that recognizes the lexemes matching the token relop.



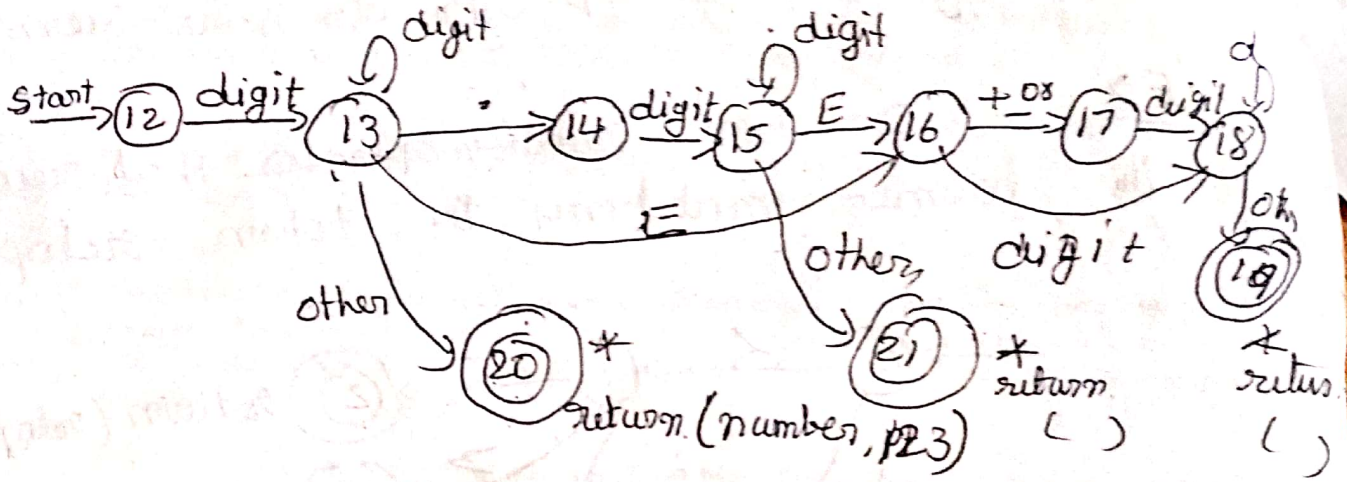
To Recognize an Identifier & K.W

letter/digit

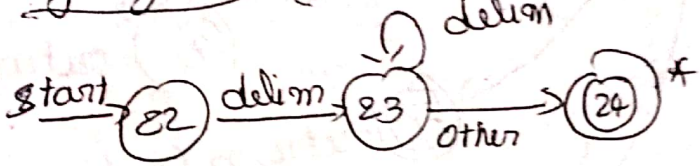


To Recognize K.W.

To Recognize the Number



To Recognize the whitespace



Here we look for one or more "whitespace" characters represented by delim. These characters could be blank, tab, Newline. No action will be taken place when we find any one of the above i.e. No tokens will be sent to the parser. Rather we must restart the lexical Analysis after the white space.

Constructing L.A:

To construct an L.A we need the collection of T. Diagrams for different terminals of a Grammar. In each state is represented by a piece of code.

ex: Take getrelop() which is a C++ fn whose job is to simulate the T.D for relop and return an object of type TOKEN, i.e pair consisting of token name & attribute value.

getrelop() 1st creates a new object retToken & initializes the 1st component to RELOP.

If State = 0 then fn nextchar() obtains the next character from the i/p & assigns it to the local variable C. We then check C for the three characters we expect to find, ex if next character is = then we go to state 5.

If the next character is not the one that can begin a comparision operator then the fn fail() is called. & it should reset the forward pointer to become Begin, to allow another T.D to be applied to the true beginning of the unprocessed i/p.

It might then change the value of State to be the start state for another T.D, which will search for another token,

Alternatively if there is no other F.D that remains unused, fail() could initiate an error-correction phase that will try to repair the i/p & find a lexeme.

If we are in state 8 we must retract the i/p ptr one position. which is done by fn retract(). Since state 8 represents the recognition of lexeme >= we set the second component of the return object, which we name as GT

```
TOKEN getrelop()
{
  TOKEN retToken = new (RELOP);
  while (1) { /* repeat character processing until
                 a return or failure occurs */
    switch (state)
    { case 0: c = nextchar();
              if (c == '<') state = 1;
              else if (c == '=') state = 5;
              else if (c == '>') state = 6;
              else fail(); /* lexeme is
                              not relop */
              break;
```