Chapter 1: Introduction

1.1 Language Processors:

Language processors are

1. Preprocessor
2. Compiler / Interpreter
3. Assembler
4. Loader / Linker

Preprocessor:

- It collects the source program before actual translation begins
- The output of preprocessor may be given as the input to compilers
- Preprocessor allows user to use macros in the program
- Preprocessor allows user to include the header files which may be required by the program.
- Preprocessor can delete comments, include other files and perform macro definition

Structure

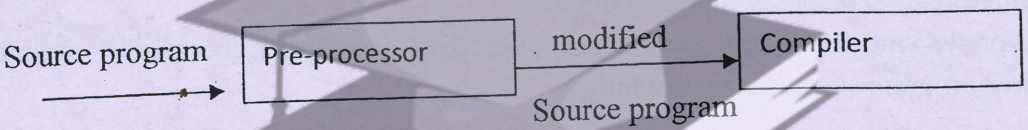


Fig: 1.1 A preprocessor

Compiler:

- It is a program that can read a program in one language – the source language – and translate it into an equivalent program in another language – the target language.
- Fig 1.2 shows an important role of the compiler is to report any errors in the source program that it detects during the translation process.

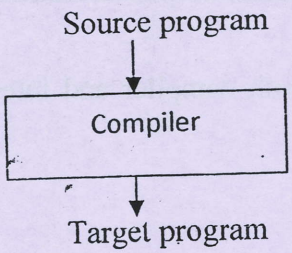


Fig: 1.2 A compiler

- If the target program is an executable machine language program it can then be called by the user to process inputs and produce outputs see fig 1.3.

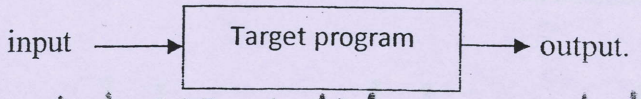


fig: 1.3 Running of target program

Interpreter:

It is another common kind of language processor instead of producing a target program as a translation, an interpreter appears directly execute the operations specified in the source program on the inputs supplied by the user as shown in fig 1.4.

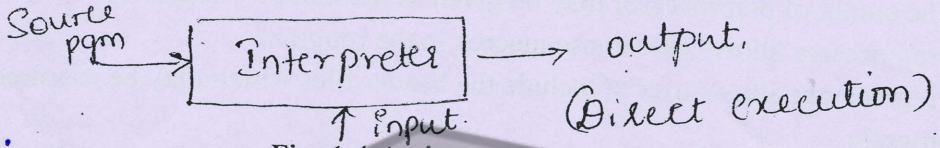


Fig: 1.4 An interpreter

- The machine language target program produced by compiler is usually much faster than an interpreter at mapping inputs to outputs.
- Interpreter can give better error diagnostics than a compiler, because it executes the source program statement by statement.

Difference between compiler and interpreter

Slno	Compiler	Interpreter
1.	Execution of whole program at a time	Execution of program line by line
2.	Program execution is relatively fast	Program execution is slow
3.	It gives an error diagnostic messages	It gives better error diagnostic message compare to compiler.

Note: JAVA language processor combines both compiler and interpreter for the execution is called Hybrid compiler, as shown in fig 1.5

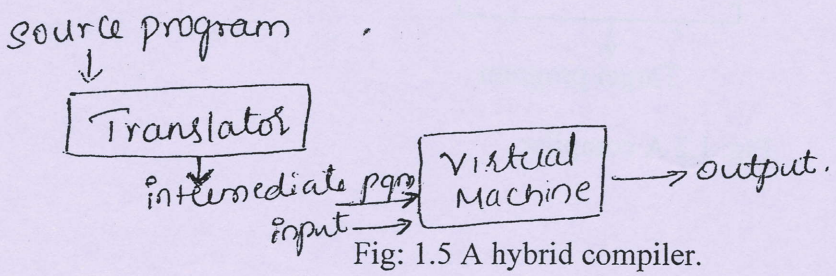


Fig: 1.5 A hybrid compiler.

- A JAVA source program may first be compiled into an intermediate form called byte codes.
- The byte codes are then interpreted by a virtual machines.
- A benefit of this arrangement is that byte codes compiled on one machine can be interpreted on another machine perhaps across a network.
- In order to achieve faster processing of inputs to outputs, some Java compilers called Just_in_time compiler, translate the byte codes into machine language immediately before they run the intermediate program to process the input.

Assembler:

The compiler produces an assembly code as the output, which is given as the input to the assembler.

The assembler is a kind of translator which accepts assembly language as the input and produce relocatable machine code as its output is shown in the fig 1.6.

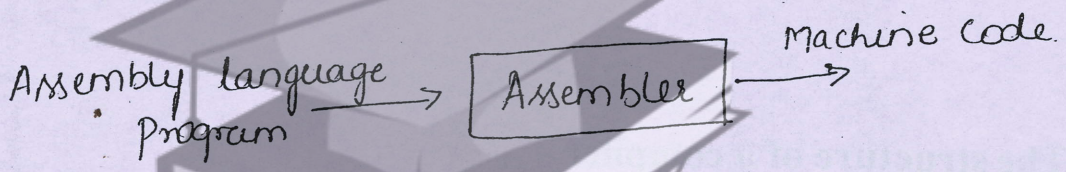


Fig 1.6 an Assembler

Loaders and Linkers:

- A linking is a process which collects separately compiled different object code files into a single file i.e. relocatable machine code is directly executable.
- Linker resolves external memory address where the code in one file may refer's the location in another file.
- Loading is a process in which the relocatable machine code is read and the relocatable address are modified.
- The loader then puts together the entire executable object files into memory for execution.

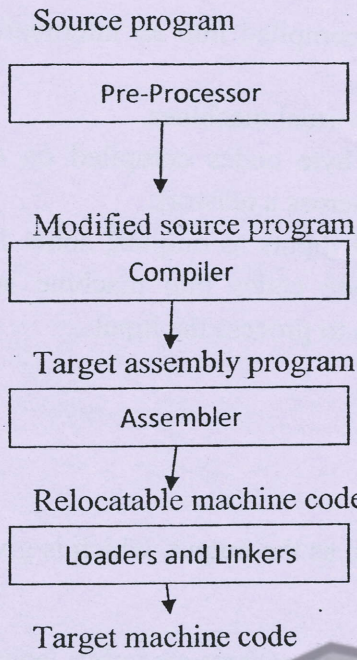


Fig: 1.7 A language –Processing system

1.2 The structure of a compiler

The compilation can be performed using two major parts of a compiler is shown in fig 1.8

1. Analysis Part
2. Synthesis Part.

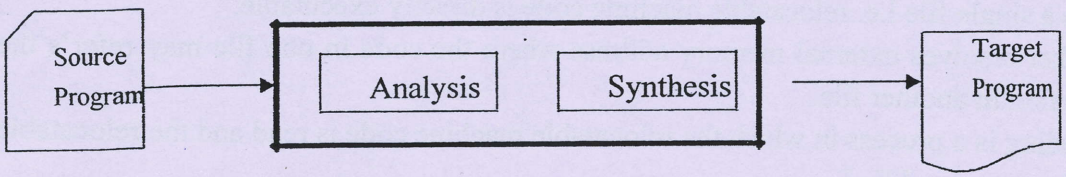


Fig 1.8 analysis and synthesis model.

Analysis Model: (Front end of compiler)

- It breaks up the source program into constituent pieces and imposes a grammatical structure on them.
- It then uses this structure to create an intermediate representation of the source program.
- If the analysis part detects that the source program is either syntactically ill formed or semantically unsound, then it must provide informative messages, so the user can take a corrective action.
- This part also collects information about the source program and stores it in a data structure called a symbol table. Which is passed along with the intermediate representation to the synthesis part?
- Analysis part consists of 4 sub parts. Such as
    a. Lexical analysis
    b. Syntax analysis
    c. Semantic analysis
    d. Intermediate code generation.

Synthesis Model: (Back end of a compiler)

- This part constructs the desired target program from the intermediate representation and the information in the symbol table.
- Synthesis part consists of 3 sub parts. Such as
    a. Machine independent code optimizer
    b. Code generator
    c. Machine dependent code generator.

Note: code optimization phases are an optional phases (the purpose of this optimization phase is to perform transformation on the intermediate representation, so that the back end can produce a better target program than it would have otherwise produced from an unoptimized intermediate representation.) Since optimization is optional, one or the other of the two optimization phases shown in fig 1.9

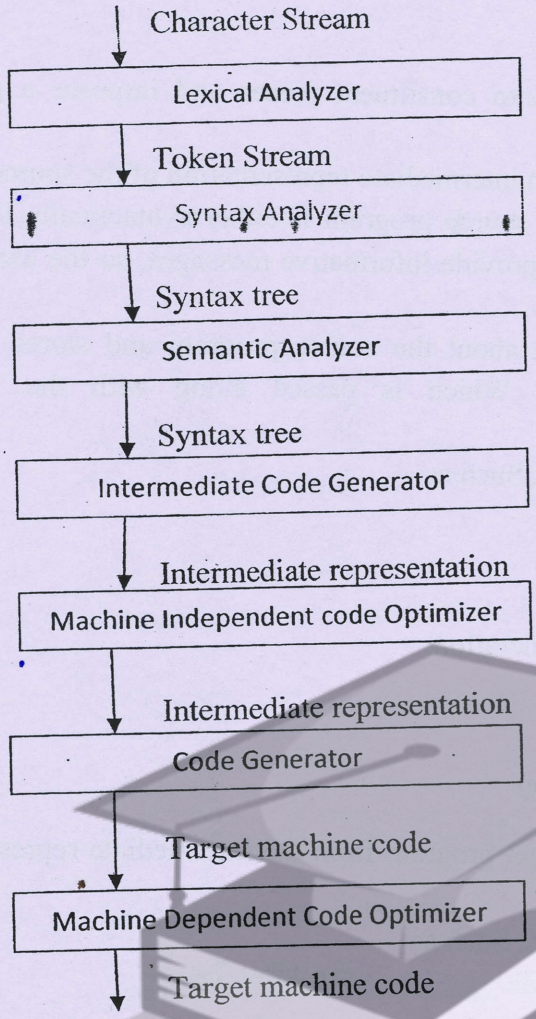


Fig 1.9 Phases of a Compiler

Lexical Analyzer: (Scanner)

- The 1st phase of a compiler is called ***lexical analysis or scanning***.
- The lexical analyzer reads the stream of characters making up the source program and groups the characters into meaning full sequence called **lexemes.**
- For each lexeme, the lexical analyzer produces the token as the output. Token is of the form

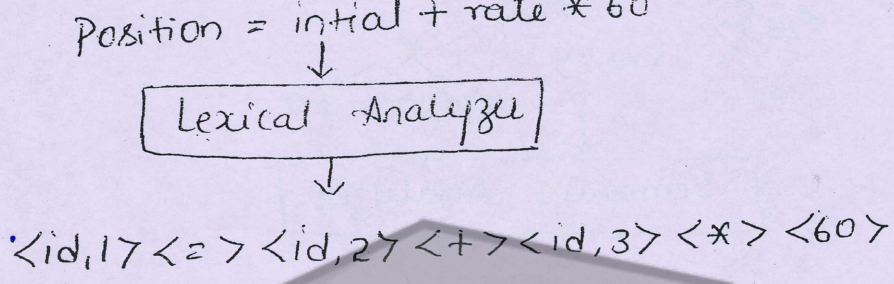
  <Token -name, attribute-value>

  - **Token name:** it is an abstract symbol that is used during syntax analysis
  - **Attribute value**: it points to an entry in the symbol table for the token.
- Information from the symbol table entry is needed for semantic analysis and code generation.

Functions of lexical analyzer:

1. Lexical analyzer collects the source program character by character
2. It divides the source program into meaning full unit is called lexeme.
3. It imposes a grammatical structure on each lexeme is called tokens.
4. It stores the information of the source program into symbol table.
5. Lexical analyzer is used to remove comment lines, white spaces (blanks, tabs and newlines).
6. Lexical analyzer also keeps track of line numbers which are helpful for location and display of errors.

Ex:



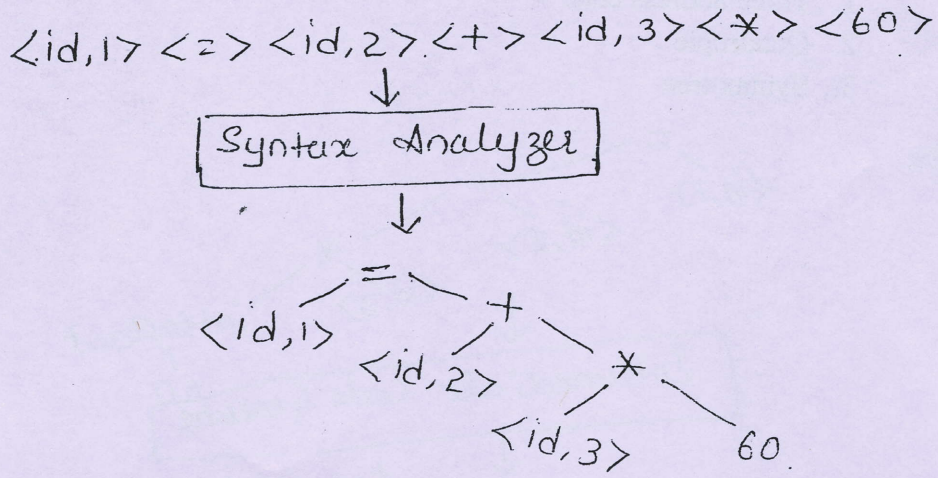
Syntax Analyzer: (Parser)

- The syntax analyzer is the second phase of the compiler also called ***parsing or syntax analysis.***

Functions of Syntax analyzer

1. it collects the token as the input
2. it constructs syntax tree as output (Syntax tree: all root nodes are operators and leaf nodes are operands)
3. it recognizes the syntactic errors and sends an error messages to user.

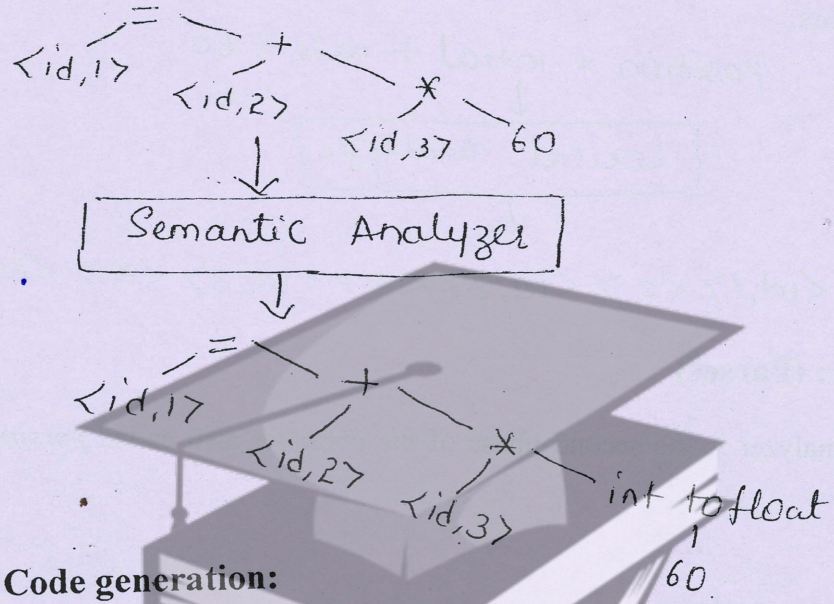
Ex:



Semantic Analyzer:

- It checks the source program for semantic errors, and collects the type information for the code generation.
- It checks for ensuring the components of programs suits together meaningfully (type checking)
- It performs type conversion (Implicit- by itself, Explicit- by user)

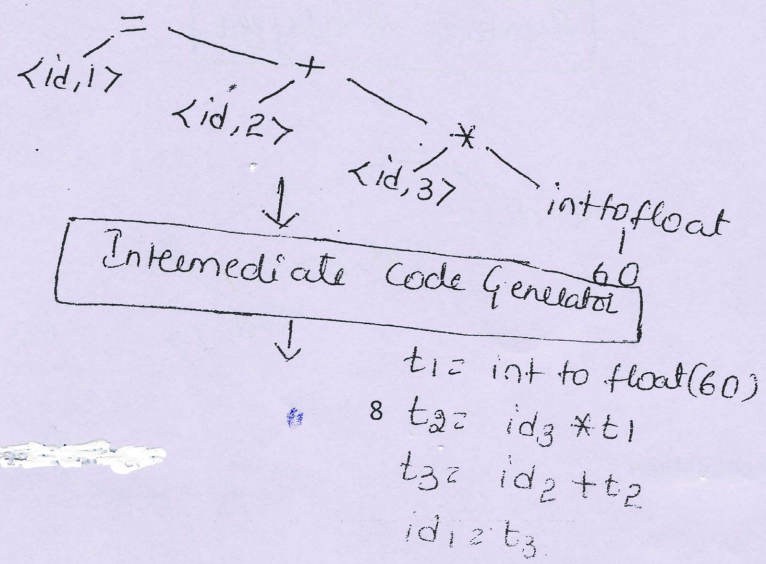
Ex: pos, rate and initial are real and 60 is an integer so the type conversion from int to float.



Intermediate Code generation:

- The intermediate code is a kind of code which is easy to generate and this code can be easily converted to target code.
- The intermediate codes are generally machine independent, but the level of intermediate code is close to the level of machine code.
- Intermediate code has variety of forms such as
    1. Three address code
    2. Quadruple
    3. Syntax tree

Ex:



We consider an intermediate form called 3- address code, which consists of a sequence of assembly like instructions with 3 operands per instruction.

Code optimization:

- The machine independent code-optimization phase attempts to improve the intermediate code so that better (faster) target code will result.
- The objective of code optimization is to get the shorter code, target code that consumes less power.

Ex:

$t_1 = inttofloat(60)$
$t_2 = id_3 * t_1$
$t_3 = id_2 + t_2$
$id_1 = t_3$

↓

Code optimizer

↓

$t_1 = id_3 * 60.0$
$id_1 = id_2 + t_1$

- Machine dependent code optimization improves the running time of the target program.

Code Generation:

- The code generator takes the input as intermediate representation of the source program and maps it into the target language.
- If the target language is machine code, Registers or memory locations are selected for each of the variables used by the program.
- The intermediate statements are translated into sequence of machine instruction that performs the same task.

Ex:

$t_1 = id_3 * 60.0$
$id_1 = id_2 + t_1$

↓

Code Generator

↓

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

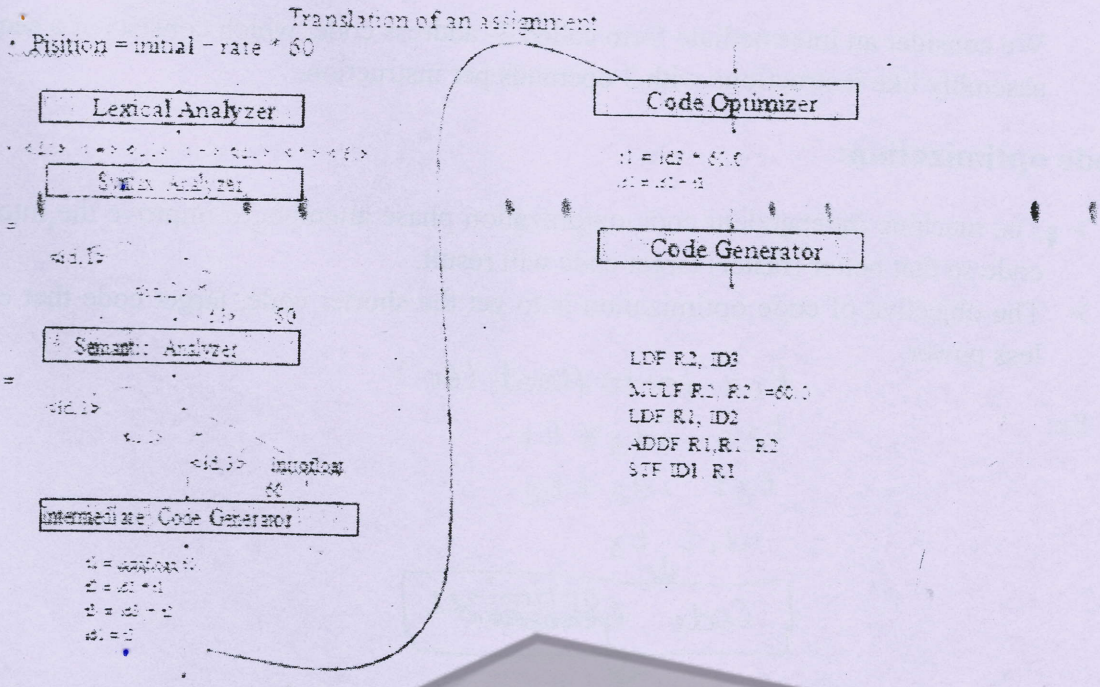


fig 1.10: Translation of an assignment statement.

Symbol Table Management:

- The symbol table is a data structure containing record for each variable name, with fields for the attributes of the name.
- The attributes may provide information about the storage allocation for a name, its type and its scope.
- In case of procedure names, such things as the number and types of its arguments, the method of passing each argument, and the type returned.
- The data structure should be defined to allow the compiler to find the record for each name quickly and to store or retrieve data from that record quickly.

The grouping of phases into passes:

Implementation activities from several phases may be grouped together into a passes that reads an input file and writes an output file.

Ex: Front-end phases

1. Lexical Analyzer
2. Syntax Analyzer
3. Semantic Analyzer
4. Intermediate code Generation

Optional phases	1. Machine independent code optimization
	2. Machine dependent code optimization
Back-end phases	1. Code Generation

Compiler construction tools:

Compiler writer use a modern software development environment containing tools such as language editors, debugging, version managers, text harnesses other more sophisticated tools have been created to help the implementation of various phases of compiler

The following are some of the compiler construction tools

1. **Parser Generator:** That automatically produces syntax analyzer from a grammatical description of a programming language.
2. **Scanner generator:** that produces lexical analyzer from a regular Expression description of the tokens of a language.
3. **Syntax directed translation engine:** it produces collections of routines for walking a parse tree and generating intermediate code
4. **Code generator generators:** that produces a code generator from a collection of rules for translating each operation of the intermediate language into machine language for a target machine.
5. **Data flow analysis engine:** that facilitates the gathering of information about hoe values are transmitted from one part of a program to other part.
6. **Compiler – construction toolkits:** that provides an integrated set of routines for constructing various phases of a compiler.

1.3 The evaluation of programming language:

1. Move to high level language
2. Impacts on compilers

1.3.1 Move to High level language:

- There are thousands of programming languages available currently we can classify languages based on generation, commutation etc….

Classification by generation:

1st generation: M/c language

2nd generation: Assembly level language

3rd generation: High level languages like C,C++,JAVA etc…

4th generation: Specific applications like NOMAD for report generation, SQL for data base queries.

5th generation: Logic and constraint based languages like prolog and oops5.

Classification by computation

Imperative language: in which a program specifies how computation should be done

Ex: C, C++, Java etc…..

Declarative language: it specifies what computation is to be done

Ex: prolog.

The Von-Neumann language: it is applied to programming languages whose computational model is based on Von-Neumann computer architecture.

The object oriented languages: this supports object oriented programming

Ex: C++, Java etc…

The Scripting Language: there are interpreted languages both high level languages operators designed for "Gluing together" computation. Theses computations are called scripts.

Ex: Java, PHP, Perl scripts.

1.3.2 Impacts on compilers:

- Compiler writing task is challenging. New algorithms have to be designed and representation to translate and support new language features. They have to desire new translation algorithm that would take maximal advantage of the new hardware capabilities.

- The performance of a computer system is so dependent on compiler technology that computers are used as tool in evaluating architectural concept before a compiler is built.
- Compiler writing is challenging, many modern language processing system handle several source languages and target machines within some frame work in collection of compilers.
    1. Millions of lines of code may be there in input program.
    2. A compiler must translate correctly the infinite set of programs that could be written in source language.

1.4 The science of building a compiler:

1.4.1 Modeling in compiler design and implementation:

Study of compilers is a study of how we design the right mathematical models and choose right algorithms while balancing. The need for generality and power against simplicity and efficiency some of the most fundamental models are "Finite State Machines and Regular Expressions" for Lexical units and "Context free Grammars" for Syntactic units.

1.4.2 The Science of Code Optimization

Optimization in compiler design refers to attempts that a compiler makes to produces code that is more efficient than obvious code is "optimization" there is no way that code produced by a compiler can be guaranteed to bear fast or faster than any other code.

Code optimization must meet the following design objectives.

1. Optimization must be correct. That is preserves the meaning of the compiled program
2. Optimization must improve the performance of many programs.
3. Compilation time must be reasonable.
4. Engineering effort required must be manageable.

1.5 Applications of Compiler technology

Compiler design impacts several areas of computer science. Such as

1. Implementation of High Level Programming Language.
2. Optimization for Computer Architecture
3. Design of new computer architecture
4. Program Translations
5. Software Productivity Tools

1.5.1 Implementation of High Level Programming Language

Different Programming Languages supports different levels of abstraction.

- C: Predominant programming language in 1980's. In order to provide additional features in 1990 introduced object oriented programming language (C++, Java).

- Key ideas behind object oriented programming languages are
  1. Data abstraction
  2. Inheritance
  3. Polymorphism.

- Object oriented programs are difficult from those written in many other languages, in that they consist of many more, but smaller, procedures.

- Java Language has many features that make programming easier, many of which have been introduced previously in other languages. Such as

  1. Type-safe: that is, an object cannot be used as an object of unrelated type.
  2. Boundary checking: All array accesses are checked to ensure that they lie within the bounds of the array.
  3. Java has no pointers and does not allow pointer arithmetic.
  4. Java has built in garbage collection facility: that automatically frees the memory of the variables that are no longer in use.
  5. Java is designed to support portable and mobile code.

1.5.2 Optimization for Computer Architecture

- The rapid evaluation of computer architecture has led to an insatiable demand for new compiler technology.

- Almost all high-performance systems take advantage of the same two basic techniques:

1. Parallelism
2. Memory hierarchy

Parallelism: It can be found at several levels

1. Instruction level Parallelism
2. Processor level Parallelism

Instruction level Parallelism: Where multiple operations are executed simultaneously. In this the compilers arrange the instructions to make it very effective.

Ex: VLIW (Very Long Instruction Word) machines have the instructions that can issue multiple optimizations in parallel.

Processor level Parallelism: where different threads of same application are run on different processors.

Memory hierarchy: it consists of several levels of storage with different speeds and sizes, with the level closest to the processor being fastest and smallest. Average memory access time is reduced if most of its accesses are satisfied by faster levels of hierarchy.

- Memory hierarchy is found in all machines.
- We can build very fast storage or very large storage, but not storage i.e., both fast and large.
- Processor usually has small number of registers consisting of bytes, next several levels of Cache containing kilobytes to megabytes, then primary memory containing megabytes to gigabytes, and finally secondary storage that contains gigabytes and beyond.

1.5.3 Design of new computer architecture

In modern computer architecture development compilers are developed in processor design stage, the different computers architecture are

1. SIC
2. SIC/XE
3. CISC
4. RISC

SIC: simplified instructional computer

SIC/XE: simplified instructional computer with extra equipment

CISC: complex instruction set computer

RISC: reduced instruction set computer, Computer optimization can reduce the instructions to a small number of simpler operations by eliminating the redundancy across complex instructions. Most general processor architectures based on RISC are Power PC, SPARC, ALPHA machines etc....

Specialized architecture: over the last 3 decades many architectural concepts have been proposed, namely Dataflow machines, Vector machines, VLIW machines, SIMD arrays of processors, multi-processor with shared memory and multi processor with distributed memory.

Compile technology is needed not only to support programming for their architectures, also to evaluate proposed architectural designs.

1.5.4 Program Translations

Compiler technology can be applied to translate between different kinds of languages.

1. **Binary translation:** Compiler technology can be used to translate binary code of one machine to that of another machine, used in companies to increase the availability of software for their machines.
2. **Hardware Synthesis:** Most software is written in High level languages; even hardware design is mostly described in hardware level. Hardware description languages are verilog, VHDL (Very high speed integrated circuit Hardware Description Language).
3. **Data Base Query interpreter:** besides software and hardware languages are useful in many other applications.
   Ex: SQL
4. **Compiled simulation:** Simulation is a general technique used in many scientific and engineering disciplines to understand a problem or to validate a design, but they are expansive. Therefore, instead of writing a simulator that interprets design, it is faster to compile the design to produce machine code that simulate that particular design compiled simulation can run faster.

1.5.5 Software Productivity tools

Software is efficient if it is error free, we can guarantee this by testing. An interesting and easy approach is to use dataflow analysis to locate statically, can find errors among all possible execution paths.

Several ways in which program analysis, build upon techniques was originally developed to optimize code.

1. **Type checking:** an effective and well established technique to catch inconsistencies in programs, it can be used to catch errors. It can be used to catch variety of security holes in program.
2. **Bounds checking:** an easier to make mistake in low level languages then High level languages. Ex: C does not have array bounds check. Data flow analysis can be used to locate buffer overflow.
3. **Memory management tools:** garbage Collection is an excellent example of software reliability. Automatic memory management removes all memory related errors which are major problems in languages like C, C++.

1.6 Programming Language Basics

1.6.1 Static and dynamic distinction

- The most important issue that we face during designing a compiler for a language is what decisions can the compiler made a program.
- If a language uses a policy that allows the compiler to decide an issue, then we say that the language uses a static policy or that the issue can be decided at compile time.
- A policy that allows a decision to be made when we execute the program is said to be a dynamic policy or to require a decision at run time.
- Here we need to concentrate in the scope of declaration and scope of declaration of x is the region of the program in which uses of x refers declaration. A language uses static scope if it is possible to determine the scope of declaration by loading only at the program.

1.6.2 Environments and states

The association of names with locations in memory and then with values can be described by two mappings that change as the program runs.

The environment: environment is mapping from names to locations in the store. Since variables refer to locations, we would alternatively define an environment as a mapping from names to variables.

The state is a mapping from locations in store to their values.

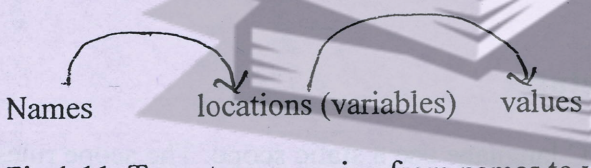


Fig:1.11 Two stage mapping from names to values.

Environments change according to the scope rules of a language.

```
....
int i;                  /* global i */
.....
Void f(.....) {
int i;                  /* local i*/
.....
```

```
i=3;                    /*use of local i */
.....
}
.....
X=i+1;                  /* use of global i */
```

Fig:1.12 two declarations of the name i.

- Consider the above C program fragment integer i is declared a global variable, and also declared as a variable local to functions f.
- When f is executing, the environment adjusts so that name I refers to the location reserved for the i that is local to f, and any use of i, such as the assignment i=3 shown explicitly refers to that location.
- Whenever function g other than f is executing, uses of i cannot refers to the i that is local to f.

The environment and state mappings are dynamic, but there are few exceptions

1. Static versus dynamic binding of names to locations: most binding of names to locations is dynamic. Some declarations such as the global i in the above fig can be given a location in the store once and for all, as the compiler generates the object code.
2. Static versus dynamic binding of locations to values: The binding of locations to values is generally dynamic as well, since we cannot tell the value in a location until we run the program.

1.6.3 Static Scope and block structure:

Most of the languages, including C language use a static scope. The scope rules for C are based on program structure, and it is determined implicitly by where the declaration appears in the program. Later advanced languages such as C++ and Java uses explicit access control of using keywords Public, Private and Protected.

C static scope policy is as follows:

1. A C program consists of a sequence of top level declarations of variables and functions.
2. Functions may have variable declarations within them, where variables include local variables and parameters. The scope of each such declaration is restricted to the function in which it appears.
3. The scope of a top-level declaration of a name x consists of the entire program that follows, with the exception of those statements that lie within a function that also has a function declaration of x.

In C, the syntax of blocks is given by

1. One type of statement is a block. Blocks can appear anywhere that other types of statements, such as assignment statements, can appear.
2. A block is a sequence of declarations followed by a sequence of statements, all surrounded by braces.

```
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;
}
```

Fig:1.13 Blocks in a C++ program

Declaration	Scope
int a=1;	B1-B3
int b=1;	B1-B2
int b=2;	B2-B4
int a=3;	B3
int b=4;	B4

Fig: 1.14 Scope of declaration

1.6.4 Explicit Access Control:

Classes and structures provide it in different programming Languages. In classes, explain private, public and protected Access specifies. In C++, a class definition may be separated from the definitions of some or all of its methods. Therefore, a name x associated with the class C may have a region that is within its scope. In fact, regions inside and outside the scope may alternate, until all the methods have been defined.

1.6.5 Dynamic Scope:

Technically, any scoping policy if it is based on factor (S) that can be known only when the program executes. The term dynamic scope, however, usually refers to the following policy: A use of a name x refers to the declaration of x in the most recently called procedure with such a declaration. Dynamic scoping of this type appears only in special situations.

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

Fig:1.15 A macro whose names must be scoped dynamically

in above example the function main first calls function b. As b executes, it prints the value of a macro a. since (x+1) must be substituted for a, we resolve this use of x to the declaration int x=1 in function b. the reason is that b has a declaration of x, so the (x+1) in the printf in b refers to this x. thus the value printed is 2.

After b finishes, and c is called, we again need to print the value of macro a. however, the only x accessible to c is the global x. the printf statement in C thus refers to this declaration of x, and value 3 is printed.

1.6.6 Parameter passing Mechanisms:

All programming languages have a notion of a procedure, but they can after in how these procedures get their arguments.

1. **Call by Value:**
   - A calling function arguments value are passed to called function, there value are copied into the called function.

- Any Modification in the called function will not reflect in the calling function values i.e. before calling the function whatever value present in the arguments.
- Call-by-value has the effect that all computation involving the formal parameters done by the called procedures are local to that procedure and the actual parameter themselves cannot be changed.

2. **Call by Reference:**
   - In call-by-reference, the actual value are not passed, instead their address are passed from the calling function.
   - In the function header, the formal parameters accept the address of actual parameters.
   - Here, no values are copied as the memory locations but these values are referenced and it contains an address, it is called pointer variable.
   - If any modification in the value takes place in the called function it reflects in the calling function original value will get changed.
   - Using the pointer variables, the value of the actual parameters can be changed. This process is called pass by reference.

3. **Call by Name:**
   - A third Mechanism- call-by-name-was used in the early programming language. Algol 60.
   - It requires that the callee executes as if the actual parameter were submitted literally for the formal parameter in the code of the callee, as if the formal parameter were a macro standing for the actual parameter.
   - When the actual parameter is an expression rather than a variable, some unintuitive behaviors occur, which is one reason this mechanism is not favored today.

1.6.7 Aliasing:

There is an interesting consequence of call-by-reference parameter passing or its simulation, as in java, where references to objects are passed by value. It is possible that two formal parameters can refer to the same location; such variables are said to be aliases of one another. As a result, any two variables, which may appear to take their values from two distinct formal parameters, can become aliases of each other, as well.

Lexical Analysis

The Role of Lexical Analyzer.

Lexical analyzer is the first phase of compiler. The main function of lexical analyzer is to read the input characters of the source program from left to right one character at a time, group them into lexeme, and produces as output a sequence of tokens for each lexeme in the source program.

The role of lexical analyzer during the procen of Compilation is shown in the fig 1.1

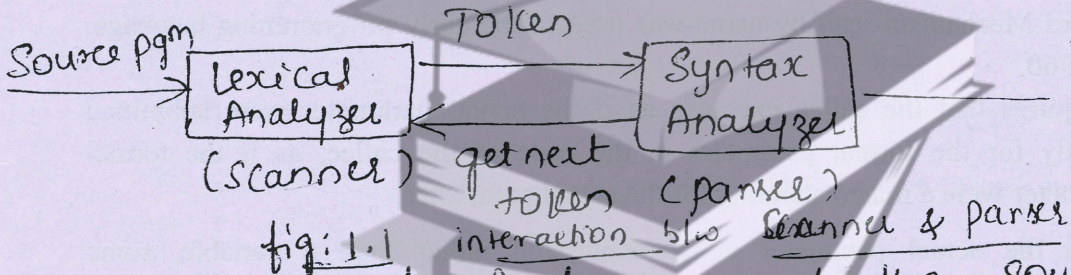


fig: 1.1 interaction b/w Scanner & parser.

* The lexical Analyzer reads the source program character by character to produce tokens.

* Normally a lexical Analyzer doesn't return a list of tokens at one shot, it returns a token when the parser asks a token from the 'get next token' Command helps lexical Analyzer to read a character from its input.

Functions of lexical Analyzer.

1. It collects the source program as input.
2. It produces a stream of tokens as output.
3. It eliminates Comments and white Spaces (blanks, newline, tab etc---)

4. It keeps track of line numbers.
5. It recognizes the lexical errors and also correct the lexical errors.
6. It stores the info of source program to Symtab.

The lexical Analyzer are divided into a cascade of two process.

1. Scanning.
2. Lexical Analysis

Scanning: It consists of simple process, it does not require tokenization of the input.
Ex:- deletion of comments, Combining consecutive white Spaces characters into one.

Lexical Analysis: It is a more complex operation, it produces sequences of tokens as output.

Lexical Analysis Vs Parsing

There are number of reasons, why the analysis Portion of a compiler normally seperated into lexical analysis and parsing (Syntax analysis)

1. Simplicity :- Techniques for lexical Analysis can be simpler than those required for Syntax analysis. DFA Vs CFG seperation also simplifies the Syntax analyzer. the Lexical Analyzer works with simple non-recursive constructs of the language and parser works with recursive constructs of the language.

2. Efficiency:- Efficiency of compiler is improved, Seperation into different modules make it easier to perform simplifications and optimizations unique to the different paradigms.
Ex:- Buffering technique.

3. Portability:- Due to input/output and character that variations, lexical Analyzer are not always machine independent.

Token, Patterns and Lexemes

Token:

* It is a pair consisting of a token name and an optional attribute value.
* The token name is an abstract symbol representing a kind of lexical unit.
  Ex:- Particular Keyword, Sequence of input characters (Identifiers).
* The token names are the input symbols that the Parser processes.

Pattern:

* It is a description of the form that the lexemes of a token may take.
* Pattern is a more complex structure that is matched by many strings.
* Ex:- Keyword as a token - pattern is just the sequence of characters that form the keyword.

3. Lexeme :

* It is a sequence of characters in the source program that matches the pattern for a token.
* It is identified by the lexical Analyzer as an instance of that token.

Token	Informal description	Sample example Lexemes
if	characters i, f	if
else	characters e, l, s, e	else
Comparision	< or > or <= or >= or <> or ==	<, >, <=, >=, <>, ==
id	letter followed by letters / digits	score, pos, rate, initial
number	any numeric constant	0, 1, 10, 3.14 etc--
Literal	any thing but surrounded by double Quotes	"core dumped",

Attributes of tokens

* When more than one lexeme can match a pattern, the Subsequent phase of Compiler should be provided with additional information by lexical Analyzer regards particular lexeme that matched.
* In such situations a lexical analyzer returns attribute value represented by the token and token name that describes the lexeme.

* Some attributes:

<id, attr> where attr is a pointer to the symtab

<assgop> no attribute is needed (if there is only one assignment operator)

<num, val> where val is the actual value of the number.

Ex:-

E = M * C ** 2.

are written below as a sequence of pairs.

<id, pointer to SYMTAB entry for E>
<assign-op>
<id, pointer to SYM-TAB entry for M>
<mult-op>
<id, pointer to SYM-TAB entry for C>
<exp-op>
<number, integer value 2>.

Lexical Errors

* The lexical Analyzer concerned issues are: skipping the comments, symbol table interface and position of the token in the file.

* The LA needs other components support for indicating errors are present in the source code.

Input Buffering

Buffer pairs

* In order to reduce the amount of overhead required to process a single input character a special buffering scheme has developed. to ~~reduce~~ process large number of characters during the compilation of large source program.

Types of Input Buffering techniques

1. One buffer technique.
2. Two buffer technique.

One buffer technique :- only one buffer is used to store the input string.

disadvantage :- if the lexeme is very long then it crosses the buffer has to be refilled. that makes overwriting the 1st part of lexeme.

Two buffer technique :- Two buffers are used to store the input string.

* 1st buffer and 2nd buffers are scanned alternately
* When end of the 1st buffer is reached, then reload the 2nd buffer and vicevarsa,
* disadvantage

  * if length of the lexeme is longer than length of the buffer then scanning input cannot be scanned

* Ex:-

if the string 'fi' is encountered for the 1st time in a C-program in the context.

fi (x == f(a))

* Here, it is very difficult for LA to tell whether 'fi' is misspelled keyword "if" or an undeclared identifier "fi". fi is a valid lexeme for the token id.

* In such situations, LA showed return the token id to the parser and let other phases of the compiler to handle such errors due to letters interchange.

* Suppose a situation arises in which the lexical analyzer is unable to proceed because none of the patterns for tokens matches any prefix of the remaining input.

* The simplest recovery strategy is "panic mode" recovery.

* It deletes the successive characters from the remaining input, until the lexical analyzer can find a well formed token at the beginning of what input is left.

* other possible error recovery actions are.

1. Delete one character from the remaining input.
2. Insert a missing character into the remaining input.
3. Replace a character by another character.
4. Transpose two adjacent characters.

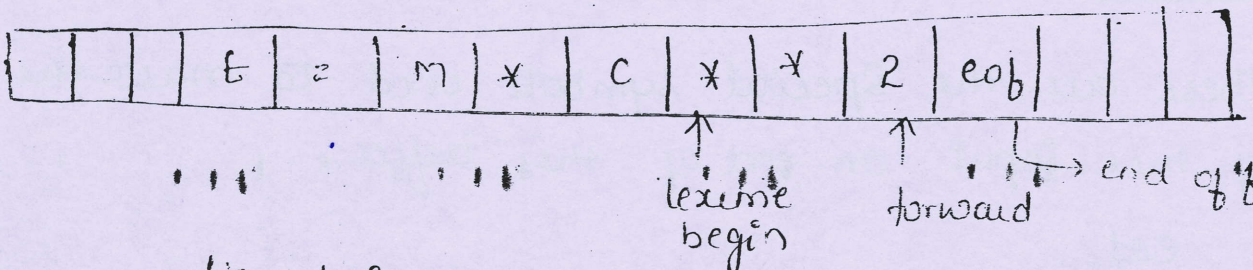


fig 1.2.

* fig 1.2 shows the input buffer.. each buffer is of the same size N, and N is usually the size of a disk block i.e., 4096 bytes.

*. Using one system read command we can read N characters into a buffer rather than using one system call per character.

*. A special character, represented by eof marks the end of the source file or end of the buffer.

*. Buffering technique maintains two pointers.

1. Lexeme begin pointer:- Marks the beginning of the current lexeme,
2. Forward pointer:- Scans ahead until pattern mater is found.

*. Initially, both pointers i.e lexeme begin pointer & forward pointer pointing to the current symbol. forward pointer will be moving towards right one symbol at a time if once the blank character is encountered the string between forward & lexeme begin it's called "lexeme"

Sentinels

*. These are the special symbols used to mark the end of the input or end of the buffer.

Ex:- eof.

* When forward encounters 1st eof. (after 4096 bytes) then one can recognize end ~~is obtained for~~ 1st buffer and hence filling up of second buffer is started.

* When second eof is obtained (after 4096 bytes) then it indicates end of the second buffer and hence filling up of 1st buffer is started.

* Alternatively both the buffers ~~are~~ can be filled up until end of the input program and token stream is identified.

Algorithm for lookahead code with sentinels

```
Switch (*forward++)
{
  Case eof:
        if (forward at the end of 1st buffer)
        {
            reload second buffer
            forward = beginning of second buffer;
        }
      else if (forward at the end of 2nd buffer)
        {
            reload first buffer
            forward = beginning of first buffer);
```

else
    terminate lexical analysis
    break;
    cases for the other characters
}.

Note: * For one buffer technique maximum 2 Sentinels can use. (1 - end of the ilp, & 1 - End of the buffer)
* For two buffer technique maximum 3 Sentinel can use.
(1 - end of the ilp, 2 - end of the buffers).

Specifications of Tokens

* To Specify tokens, a regular expressions are used.
* Regular expressions are an important notations for specifying lexeme patterns to recognize the tokens.

Strings & Languages

Alphabet :- It is any finite set of symbols (Σ) the typical ex:- letters, digits, & punctuation, ASCII

String :- A string is a finite sequence of symbols from ~~the~~ Σ the language theory.

* The terms 'sentence' and 'word' are often used as synonyms of strings

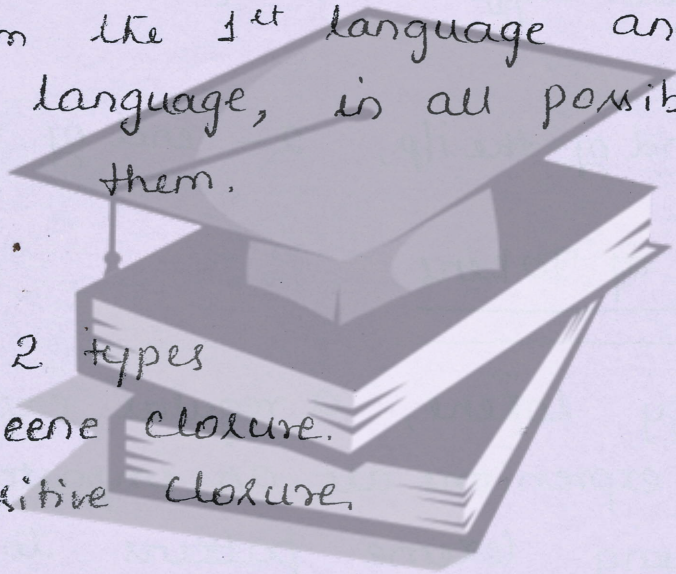
Language:- It is an countable set of strings over some fixed alphabet.

Operations on Languages

In lexical Analysis most important operations are Union, Concatenation and closure.

Union:- It is the familiar operation on set.

Concatenation:- It is all strings formed by taking a string from the 1st language and a string from the second language, in all possible ways, and concatenating them.



Closure :- 2 types

1. Kleene closure.
2. Positive closure.

Kleen closure:- The Kleen closure of a language L, denoted L^*, is the set of strings you get by concatenating L zero or more times.

Note: L^0 → the Concatenation of L zero times is defined to be $\{\varepsilon\}$.

Positive closure:- it is denoted L^+, is the same as the Kleen closure, but without the term L^0.

That is, ε will not be in L^+ unless it is in L itself.

Operation	Defination and Notation
Union of L and M	$L \cup M = \{s \mid s \text{ is in } L \text{ or } s \text{ is in } M\}$
Concatenation of L & M.	$LM = \{st \mid s \text{ is in } L \text{ and } t \text{ is in } M\}$
Kleen closure	$L^* = \bigcup_{i=0}^{\infty} L^i$
Positive closure	$L^+ = \bigcup_{i=1}^{\infty} L^i$

Regular Expressions

* It is a collection of Metacharacters.
* Metacharacters are special symbols they are having their own meaning.

  Ex: + → zero or more occurance

  ? → one or zero occurance.
* Regular expressions are used to describe the tokens

  Ex: Any valid C identifiers.

  id = letter(letter | digit)*
* A language that can be defined by a regular expression is called "regular set"

Algebraic LAWS for regular expressions

LAW	Description
1. $r \mid s = s \mid r$	$\mid$ is commutative.
2. $r \mid (s \mid t) = (r \mid s) \mid t$	$\mid$ is associative.
3. $r(st) = (rs)t$	Concatenation is associative.
4. $r(s \mid t) = rs \mid rt$; $(s \mid t)r = sr \mid tr$	Concatenation distributes over $\mid$
5. $\varepsilon r = r\varepsilon = r$	ε is the identity for concatenation
6. $r^* = (r \mid \varepsilon)^*$	ε is guaranteed in a closure.
7. $r^{**} = r^*$	$*$ is idempotent.

Regular Definitions

* We may wish to give name to regular expressions and use these names in subsequent expressions, as if the names were themselves symbols.

* If Σ is an alphabet then regular definition is a sequence of definations of the form

$$d_1 \rightarrow r_1$$
$$d_2 \rightarrow r_2$$
$$\vdots$$
$$d_n \rightarrow r_n$$

Where
$d_i \rightarrow$ new symbol
$r_i \rightarrow$ is a regular exp^n.

Ex:-

- Regular definition
  - letter → A|B---|Z|a|b|---|z.
  - digit → 0|1|2---|9
  - id → letter (letter|digit)*

Extensions of Regular Expressions

* Kleen introduced Regular expressions in 1950s.
* Unix utilities such as Lex uses notational extensions useful in specification & lexical Analyzer.

1. one / more instances → +
2. zero / more instances → *
3. zero / one instance → ?
4. Character class → [ ]

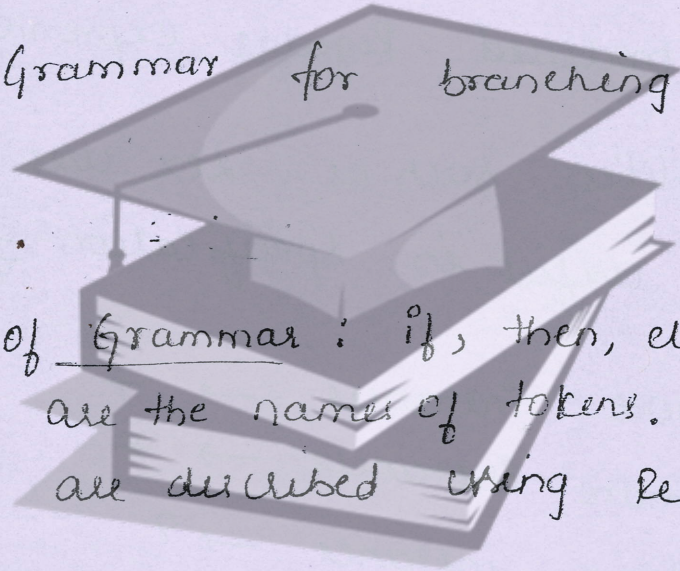
Recognition of Tokens

* We must study how to take the patterns for all needed tokens and build a piece of code that examines the input string and finds a prefix that is a lexeme matching one of the patterns.

Ex:-

```
Stmt → if expr then Stmt
     | if expr then Stmt else Stmt
     | ε
expr → term relop term
     | term
term → id
     | number.
```

fig: Grammar for branching Statement.

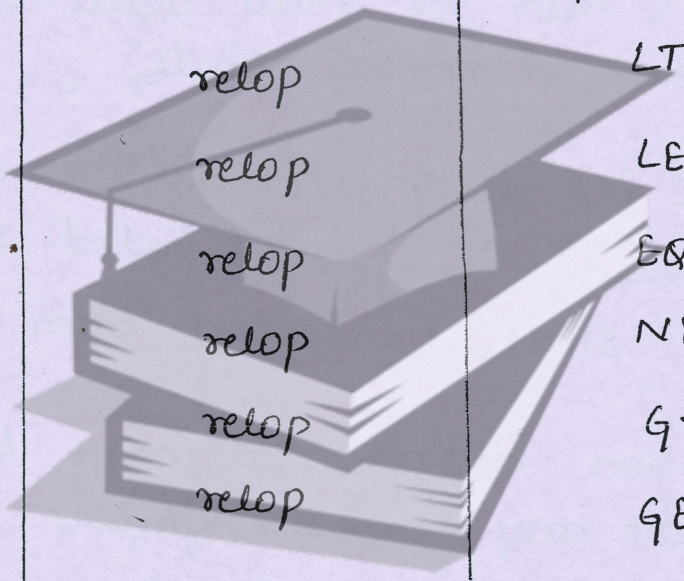


Terminals of Grammar : if, then, else, relop, id and number are the names of tokens. Pattern for these tokens are described using Regular definitions as

```
digit  → [0-9].
digits → [digit]+
number → digits (. digits)? (E [+-]? digits)?
letter → [A-Z a-z]
id     → letter (letter | digit)*
if     → if
then   → then
else   → else
relop  → < | > | >= | <= | = | <>
```

Tokens, their patterns, and attribute values.

LEXEMES	TOKEN_NAME	ATTRIBUTE VALUE
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.



Transition diagrams.

* Transition diagrams have a collection of nodes or circles, called states.
* Each state represents a condition that could occur during the process of scanning the input looking for a lexeme that matches one of several patterns.
* Edges are directed from one state of the transition diagram to another.
* each edge is labelled by a symbol or set of symbols

Some important conventions about transition diagrams are

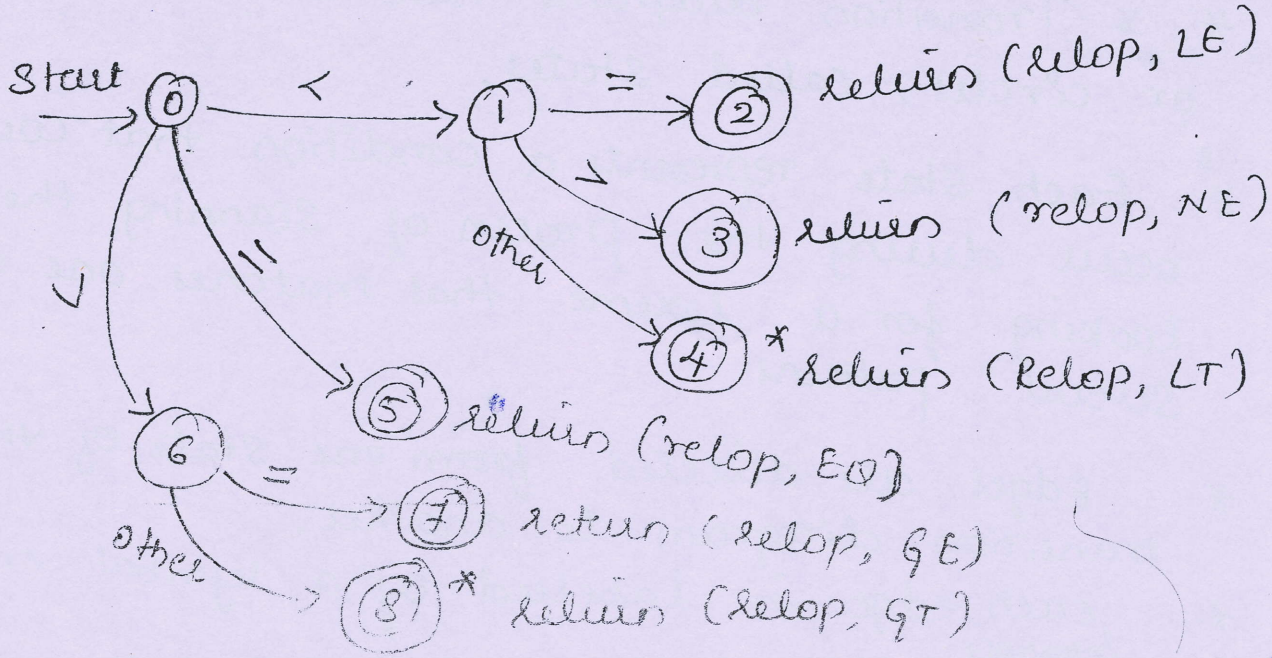
1. Certain states are said to be accepting or final. These states indicate that a lexeme has been found, although the actual lexeme may not consists of all positions between the lexeme Begin & forward pointers

2. If it is necessary to retract the forward pointer one position, then we shall 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, labelled "start" (entering from nowhere).

The transition diagram always begins in the start state before any input symbols have been read.

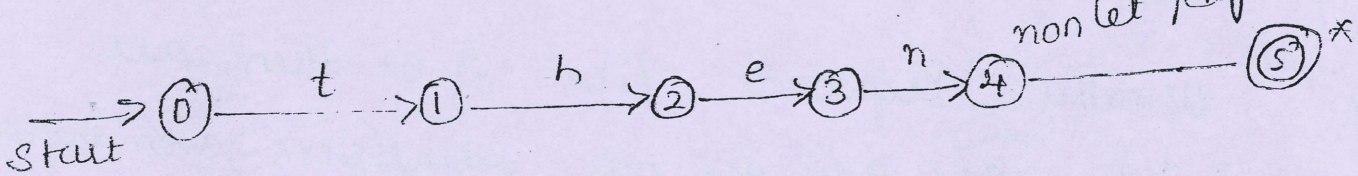
Transition diagram for relop



2. Create Seperate transition diagram for each keyword;

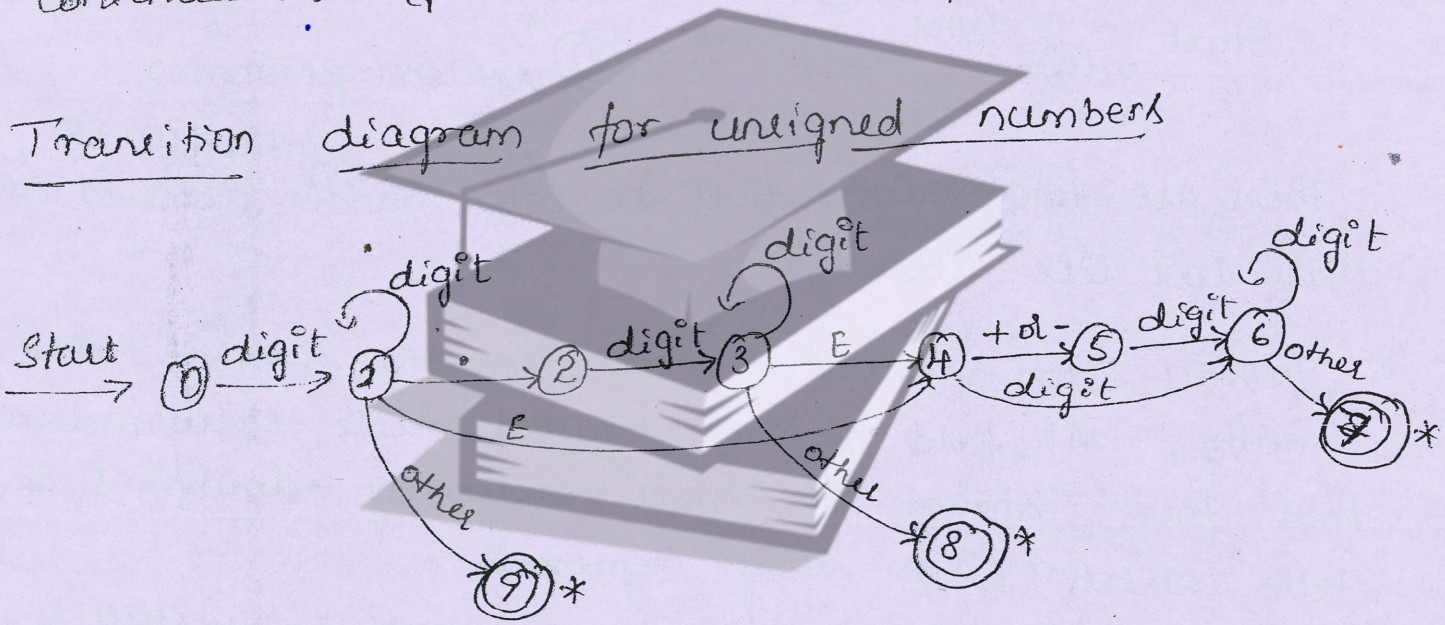
Ex:-

Keyword then.

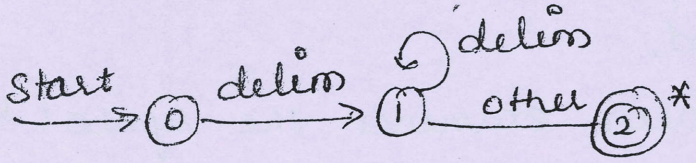


nonlet /dig → Any character that cannot be the continuation of an identifier.

Transition diagram for unsigned numbers



Transition diagram for whitespaces.

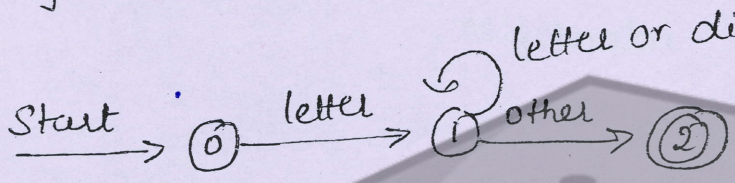


delim → (white Space, newline, blank).

Recognition of Reserved words and Identifiers.

* Recognizing keywords and identifiers presents a problem.

* Usually keywords like if or then are reserved, so they are not identifiers even though they look like identifiers.



There are two ways that we can handle reserved words that look like identifiers.

1. Install the reserved words in the symbol table initially. A field of the symbol table entry indicates that these strings are never ordinary identifiers & tells which token they represent.

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 symtable entry for the lexeme found.

Function get token examines the symbol table entry for the lexeme found, and returns whatever token name the symbol table says this lexeme represents either id or one of the keyword tokens that was initially installed in the table.

Architecture of a Transition-diagram Based Lexical Analyzer.

* There are several ways that a collection of transition diagrams can be used to build a lexical analyzer.

* Each state is represented by a piece of code.

* Variable state holding the number of the current state for a transition diagram.

* A switch based on the value of the state takes us to code for each of the possible states, where we find the action of that state.

```
TOKEN getRelop()
{
    TOKEN retTOKEN = new (relop);
    While(1)
    {
        switch (state)
        {
            case 0:  c= nextchar();
                     if (c== '<' ) state =1;
                     else if (c== '=') state = 5;
                     else if (c== '>' ) state =6;
                     else fail();
                     break;

            case # 1 : ___ - - -

                       - - - - - -

            case 8 :  retract();
                      retToken.attribute = GT;
                      return (retToken);
```