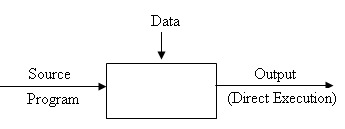
**UNIT – I**

**INTRODUCTION**

* Programming languages are notations for describing computations to people and to machines. The world as we know it depends on programming languages, because all the software running on all the computers was written in some programming language. But, before a program can be run, it first must be translated into a form in which it can be executed by a computer.
* The software systems that do this translation are called *compilers.*
* In this subject we have to cover that a few basic ideas can be used to construct translators for a wide variety of languages and machines. Besides compilers, the principles and tech­niques for compiler design are applicable to so many other domains that they are likely to be reused many times in the career of a computer scientist. The study of compiler writing touches upon programming languages, machine ar­chitecture, language theory, algorithms, and software engineering.
* First to explain the basic idea of different forms of language translators, give a high level overview of the structure of a typical compiler, and discuss the trends in programming languages and machine architecture that are shaping compilers. We include some observations on the relationship between compiler design and computer-science theory and an outline of the applications of compiler technology that go beyond compilation.

***1. Interpreter:***

* An interpreter is a program that appears to execute a source program as if it were machine language.

[](http://1.bp.blogspot.com/-2ztuqTtPxsk/T-Cq3XIsnlI/AAAAAAAAAMA/ukazcuEASFw/s1600/interpreter.bmp)

* Languages such as BASIC, SNOBOL, and LISP can be translated using interpreters. JAVA also uses interpreter. The process of interpretation can be carried out in following phases.

1. Lexical analysis

2. Syntax analysis

3. Semantic analysis

4. Direct Execution

**Advantages:**

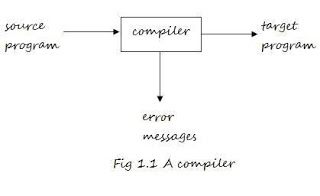
* Modification of user program can be easily made and implemented as execution proceeds.
* Type of object that denotes various may change dynamically.
* Debugging a program and finding errors is simplified task for a program used for interpretation.
* The interpreter for the language makes it machine independent.

**Disadvantages:**

* The execution of the program is slower.
* Memory consumption is more.

**2. Compilers:**

* A compiler is a program that reads a program written in one language (*the source language*) and translates it into an equivalent program in another language (*the target language*).
* As part of this translation process, the compiler reports to its user the presence of errors in the source program.

[](http://4.bp.blogspot.com/-o5vNfH-gcwc/TzC47MqHCSI/AAAAAAAAAE4/TzcMS1MSUec/s1600/untitled.JPG)

* ***The Analysis-Synthesis Model of Compilation:***
* There are two parts to compilation: ***analysis*** and ***synthesis***.
* The ***analysis part*** breaks up the source program into constituent pieces and creates an intermediate representation of the source program.
* The ***synthesis part*** constructs the desired target program from the intermediate representation.
* During analysis, the operations implied by the source program are determined and recorded in a hierarchical structure called ***tree (syntax tree)***.
* In which each node represents an operation and the children of a node represent the arguments of the operation.

* Software tools that manipulates the source program first perform some kind of analysis:

1. ***Structure Editors*:** takes input a sequence of commands to build a source program.

* Performs text creation and modification, analyzes the program text – hierarchical structure (check the i/p is correctly formed).

1. ***Pretty Printers***: analyzes the program and prints the structure of the program becomes clearly visible.
2. ***Static Checkers***: reads a program, analyzes it, and attempts to discover potential bugs without running the program.
3. ***Interpreters***: Instead of producing a target program as a translation, it performs the operations implied by the source program.

* Some examples where the analysis portion is similar to conventional compiler:

1. ***Text Formatters***: takes input that is a stream of characters and indicates commands to indicate paragraphs, figures, or mathematical structures like subscripts and superscripts.
2. ***Silicon Compilers***: has a source language similar to a conventional programming language.

* However, the variables of the language represent, not locations in memory, but logical signals (0 or 1) or groups of signals in a switching circuit.

1. ***Query Interpreters***: translates a predicate containing relational and boolean operators into commands to search database for records satisfying that predicate.

* ***Preprocessor***: The task of collecting the source program is sometimes entrusted to a distinct program.

**3. Analysis of the Source Program**

* In compiling, analysis consists of three phases:
  1. ***Linear Analysis*:** in which the stream of characters (source program) is read from left to right and grouped into tokens; that are sequence of characters having a collective meaning.
  2. ***Hierarchical Analysis***: in which characters or tokens are grouped hierarchically into nested collections with collective meaning.
  3. ***Semantic Analysis***: in which certain checks are performed to ensure that the components of a program fit together meaningfully.

* ***Lexical Analysis***: (*Linear Analysis or Scanning*)

–     For example, in lexical analysis the characters in the

assignment statement

•      *Position:=initial + rate \* 60*

–     Would be grouped into the following tokens:

•     The *identifier*; position

•     The *assignment symbol*; :=

•     The *identifier*; initial

•     The *plus symbol*; +

•     The *identifier*; rate

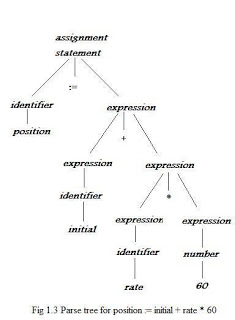
•     The *multiplication symbol*; \*

•     The *number*; 60

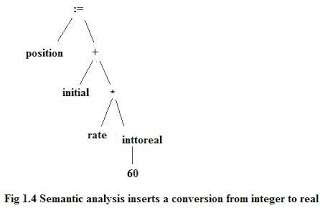
•     The blanks separating the characters of these tokens would normally

be eliminated during lexical analysis.

* ***Syntax Analysis***: (*Hierarchical Analysis or Parsing*)
* It involves grouping the tokens of the source program into grammatical phrases that are used by the compiler to synthesize output.
* The grammatical phrase is represented by a parse tree.

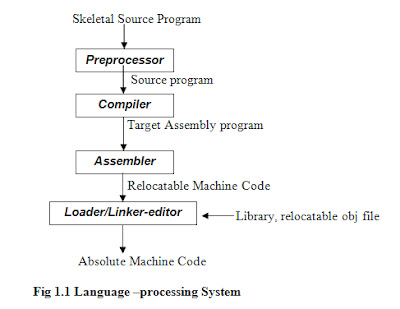
[](http://3.bp.blogspot.com/-kV9ThXnGfQM/TzC5gdEWf7I/AAAAAAAAAFA/gA4Y1atdoy4/s1600/untitled1.JPG)

* ***Semantic Analysis***:
* This phase checks the source program for semantic errors and gathers type information for the subsequent code-generation phase.
* It uses the hierarchical structure determined by the syntax-analysis phase to identify the operators and operands of expressions and statements.
* An important component of semantic analysis is ***type checking***.

[](http://2.bp.blogspot.com/-tK3m5Jqbgb8/TzC5qnMX1vI/AAAAAAAAAFI/uFOUYOSXR0M/s1600/untitled2.bmp)

**4. Overview of Language Processing:**

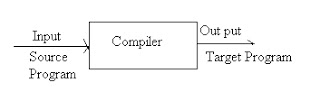
A hardware device designed or used to perform tasks, such as processing program code to machine code. Language processors are found in languages such as [FORTRAN](http://www.computerhope.com/jargon/f/fortran.htm) and [COBOL](http://www.computerhope.com/jargon/c/cobol.htm). To get the source program to machine code we have some phased.



**Pre-processor:** A pre-processor produce input to compilers. They may perform the following functions.

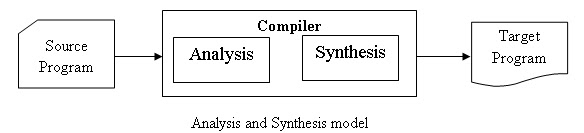
1. *Macro processing:* A pre-processor may allow a user to define macros that are short hands for longer constructs.
2. *File inclusion:* A pre-processor may include header files into the program text.
3. *Rational pre-processor:* these pre-processors’ augment older languages with more modern flow-of-control and data structuring facilities.
4. *Language Extensions:* These pre-processor attempts to add capabilities to the language by certain amounts to build-in macro.

C**ompiler:** A compiler is a program that accepts a program written in a High Level Language and produces an object (low-level) program.

[](http://3.bp.blogspot.com/-jU66EsujzhY/T-CYRDoF_-I/AAAAAAAAALg/RkXDkPcccFo/s1600/compiler.bmp)

The compilation can be done in two parts: **Analysis** and **Synthesis**.

* In Analysis part the source program is read and broken down into constituent pieces. The syntax and the meaning of the source string are determined and then an intermediate code is created from input source program.
* In Synthesis part this intermediate form of the source language is taken and converted into an equivalent target program. During this process if certain code has to be optimized for efficient execution then the required code is optimized.

[](http://2.bp.blogspot.com/-DtHS41n3AY0/T-CcmoEIFBI/AAAAAAAAALs/zJg647UkklE/s1600/aandSmodel.bmp)

***Properties of Compiler***

1. The compiler itself must be bug-free.
2. It must generate correct machine code.
3. The generated machine code must run fast.
4. The compiler itself must run fast.
5. The compiler must be portable.
6. It must give good diagnostics and error messages.
7. The generated code must work well with existing debuggers.
8. It must have Consistent optimization.

***Features of good compiler***

Various features of good compiler are as given below:-

1. The good compiler compiles the large amount code in *less time*.
2. The good compiler requires *less amount of memory* space to compile the source language.
3. The good compiler can c*ompile only the modified code* segment if frequent modifications are required in the source program.
4. While handling the hardware interrupts the good compilers *interact* closely with operating system.

**Assembler:***P*rogrammers found it difficult to write or read programs in machine language. They begin to use a mnemonic (symbols) for each machine instruction, which they would subsequently translate into machine language. Such a mnemonic machine language is now called an assembly language. Programs known as assembler were written to automate the translation of assembly language in to machine language. The input to an assembler program is called source program, the output is a machine language translation (object program).

**Loader and Link-editor:** Once the assembler procedures an object program, that program must be placed into memory and executed. The assembler could place the object program directly in memory and transfer control to it, thereby causing the machine language program to be execute. This would waste core by leaving the assembler in memory while the user’s program was being executed. Also the programme would have to retranslate his program with each execution, thus wasting translation time. To overcome this problems of wasted translation time and memory. System programmers developed another component called loader.

“A loader is a program that places programs into memory and prepares them for execution.”

It would be more efficient if subroutines could be translated into object form the loader could “relocate” directly behind the user’s program. The task of adjusting programs o they may be placed in arbitrary core locations is called relocation. Relocation loaders perform four functions.

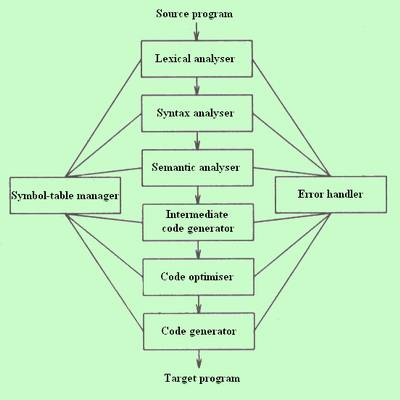
1. Allocate space in memory for the programs.
2. Resolve symbolic references between object decks (linking).
3. Adjust all address-dependent locations, such as address constants, to correspond to the allocation space (relocation).
4. Physically place the machine instructions and data into memory (loading).

**5. Phases of a compiler**

***Phases of a compiler:*** A compiler operates in phases. A phase is a logically interrelated operation that takes source program in one representation and produces output in another representation. The phases of a compiler are shown the following.

There are two phases of compilation.

1. Analysis (Machine Independent/Language Dependent)
2. Synthesis(Machine Dependent/Language independent)

****

**Lexical analysis:** It is the process of converting a sequence of characters into a sequence of tokens. A program or function which performs lexical analysis is called a **lexical analyzer**, or **scanner**.

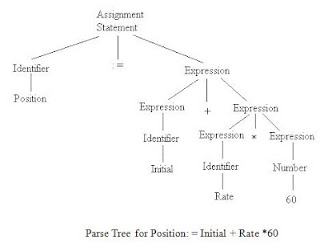
As translation progress the compiler's internal representation of source program change

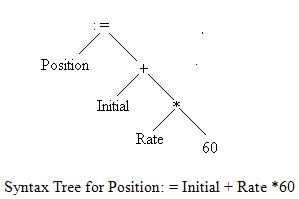
Position: = initial=RATE\*60.

* The lexical analyzer phase needs the character in the source program and groups them into a stream of tokens in which each token represents a logically cohesive sequence of character, such as an identifier, keyword (if, while, etc....) punctuation character, or a multi-character operator like:= the character sequence forming a tokens is called the lexeme for the tokens certain tokens will be augmented by "a lexical value".
* For example when an identifier like rate in found, the lexical analyzer not only generates a tokens say id, but also enters the lexeme rate into the symbol table, if it is not already there. The lexical value associated with this occurrence of 'id' point to the symbol table entry for rate.

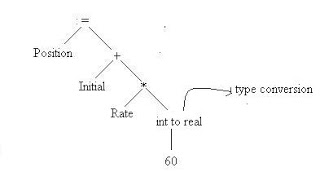
**Syntax Analysis:** The syntax analyzer does syntax analysis. It groups the tokens of the source program into grammatical phrase that r used by the compiler to generates output this is represented by the parse tree (hierarchical structure).

* A syntax tree is a comprised representation of the parse in which the operators appear as the interior nodes and operands of an operator are the children of the node for that operator.

[[](http://4.bp.blogspot.com/-spCVjyMB0Nc/T99So8zDUyI/AAAAAAAAAK4/Te55Gia0wvQ/s1600/fig4.JPG)](http://4.bp.blogspot.com/-spCVjyMB0Nc/T99So8zDUyI/AAAAAAAAAK4/Te55Gia0wvQ/s1600/fig4.JPG)

Syntax tree:  
[](http://1.bp.blogspot.com/-a7t68xs8Z1I/T99TvRUl3uI/AAAAAAAAALA/8MxSik2K684/s1600/fig5.JPG)

**Semantic analysis:** Semantic analysis checks the source program for somatic errors. It uses the hierarchical structure deter main by the syntax analysis phase to identify the operator and operands of expression and statements. Semantic analysis performs type checking i.e, it checks that each operator has operands that are permitted by the source language specification.

[](http://1.bp.blogspot.com/-GgcLdxiMVuA/T99U19KyoaI/AAAAAAAAALI/7S9ms5PKnsM/s1600/fig6.JPG)

**Intermediate Code Generator:** After syntax and semantics analysis some compiler generates an intermediate representation of the source program. This representation should be easy to produce and easy to translate into the target program.

* Thus the intermediate code generation phase transforms the parse tree into an intermediate language. On popular type of intermediate language is called as "three-address code" which is like the assembly language the three-address code for the statement.

Position: = initial + rate\*60

temp1:=into real (60)

temp2:=id3\*temp1

temp3:=id2+temp2

id1:=temp3

**Code Optimization:** Code optimization phase improve the intermediate code ie., It reduces the code by removing the repeated or unwanted instructions from the intermediate code. The above giving intermediate code can be optimized to the following

Temp1:=id3\*60.0

id1:=id2+temp1

* Compiler can deduce that the conversion of 60 from integer to real can be done all at once at compile time. So that into real operations can be eliminated, temp3 is used to transmit its value to id1 instead which can be directly substituted to id1.

**Code Generation:** Code generation converts the intermediate code into the target code consisting of sequenced machine code or assembly codec that performs the same task. The above optimized code can be written using registers R1 and R2 which is given below

MOVF id3, R2

MULF #60.0, R2

MOVF id2, R1

ADDF R2, R1

MOVF R1, id1

* First and second operands of each instruction specify source and destination respectively. The F in each instruction tells that instruction delays with floating point number.

**Error Detection Reporting**

* Each phase can encounter errors. However after detecting an error, a phase must somehow deal with the error so that compilation can process, allowing further error in the source program to be detected.
* The syntax and semantic analysis phase usually handle a large function of the error detection by the compiler. The lexical phase can detect error when the character remaining in the input does not form any tokens of the language.

(Eg:-if we try to add two identifiers one of which is the name of an array, and the other the name of a procedure)

**Symbol Table Management**

* An essential function of compiler is to record the identifier used in the source program and collect information about various attributes of each identifier.
* A symbol table is a data structure containing a record for each identifier, with field for the attribute of identifier. The data structure allows us to find the record for each identifier.
* Quickly and to store or retrieve data form that’s record quickly. When an identifier in source program detected the lexical analyzers, the identifiers entered into symbol table.

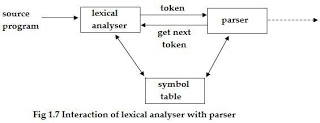
EXAMPLE:- var position, initial, rate, real........

**6. Lexical Analysis:**

* A simple way to build a lexical analyseris to construct a diagram that illustrates the structure of the tokens of the source language, and then to translate the diagram into a program for finding tokens.
* The **software tool** that automates the construction of lexical analysers is **lexical analyser generator.**
* The **major advantage of a lexical analyser generator** is that it can utilize the best-known pattern-matching algorithms and there by create efficient lexical analysers for people who are not experts in pattern-matching techniques.

**6.1 Role of the Lexical Analyser:**

* The lexical analyser is the first phase of a compiler.
* Its main task is to read the input characters and produce as output a sequence of tokens.

[](http://2.bp.blogspot.com/-DRPxl2ndjjA/TzIXBmB7OxI/AAAAAAAAAFg/HXxtRbZWSP0/s1600/aa.JPG)

* Upon receiving a “get next token” command from the parser, the lexical analyser reads input character until it can identify next token.
* Lexical analysers read source text and also perform certain secondary task.

–     Stripping out from the source program comments and white space in the

form of blank, tab, and newline characters.

–     Correlating error messages from the compiler with the source program.

•      For example, the lexical analyser may keep track of the number of

newline characters seen, so that a line number is associated with

an error message

–     If the source language supports some macro pre-processor functions, then

these pre-processor functions may also be implemented as lexical analysis

takes place.

***Issues in lexical analysis*:**

* Reasons for separating the analysis phase into lexical analysis and parsing:

1. Simpler design

2. Compiler efficiency is improved

•      Specialized buffering techniques for reading input characters and

processing tokens can significantly speed up the performance of

the computer.

3. Compiler portability is enhanced

***Tokens, lexeme and pattern*:**

* Set of strings in the input for which the same *token* is produced as output.
* Set of strings is described by a rule called a *pattern* associated with the token.
* A *lexeme* is a sequence of characters in the source program that is matched by the pattern for a token.

For example: **const pi = 3.1416;**

**Examples of tokens**

|  |  |  |
| --- | --- | --- |
| **Token** | **Sample Lexemes** | **Informal description of pattern** |
| const | const | const |
| if | if | if |
| relation | <, <=, =, >, >=, <> | <or<=or=or>or>=or<> |
| id | pi, count, D2 | letter followed by letters and digits |
| num | 3.1416, 0, 6.02E23 | any numeric constant |
| literal | “core dumped” | any characters between “ and “ except" |

**Lexical errors:**

* Misspelling of the keyword; e.g. fi(a==f)
* Undeclared function identifier;

**Error-recovery actions:**

* Deleting an extraneous character
* Inserting a missing character
* Replacing an incorrect character by a correct character
* Transporting two adjacent characters

**7. Difference between compiler and interpreter**

* A compiler converts the high level instruction into machine language while an interpreter converts the high level instruction into an intermediate form.
* Before execution, entire program is executed by the compiler whereas after translating the first line, an interpreter then executes it and so on.
* List of errors is created by the compiler after the compilation process while an interpreter stops translating after the first error.
* An independent executable file is created by the compiler whereas interpreter is required by an interpreted program each time.
* The compiler produce object code whereas interpreter does not produce object code.
* In the process of compilation the program is analyzed only once and then the code is generated whereas source program is interpreted every time it is to be executed and every time the source program is analyzed. Hence interpreter is less efficient than compiler.
* Examples of interpreter: A *UPS Debugger* is basically a graphical source level debugger but it contains built in C interpreter which can handle multiple source files. Example of compiler: *Borland c compiler* or Turbo C compiler compiles the programs written in C or C++.

**8. Cousins of Compiler**

* The term ‘cousins of compiler’ refers to the type of programs which are required for the execution of the source program. These are the programs along with which compiler operates. The cousins of compilers are pre-processors, assemblers, and loaders and link editors.

**9. The Grouping of Phases**

* Logically each phase is viewed as a separate program that reads input and produced output for the next phase. In practice some phases are combined.

***Front and Back Ends***

* Modern compilers contain two parts, each which is often subdivided. These two parts are the front end and back end.
* The front end consists of those phases, or parts of phases, that depends primarily on the source language and is largely independent of the target machine. These normally include lexical and syntactic analysis, the creation of the symbol table, semantic analysis, and the generation of intermediate code. A certain amount of code optimization can be done by the front end as well. The front end also includes the error handling that goes along with each of these phases.
* The back end includes those portions of the compiler that depend on the target machine, and generally, these portions do not depend on the source language, just the intermediate language. In the back end, we find aspects of the code optimization phase, and we find code generation, along with the necessary error handling and symbol-table operations.

***Passes***

* Several phases of compilation are usually implemented in a single pass consisting of reading an input file and writing an output file. In practice, there is great variation in the way the phases of a compiler are grouped into passes, so we prefer to organize our discussion of compiling around phases rather than passes.
* It is common for several phases to be grouped into one pass, and for the activity of these phases to be interleaved during the pass. For example, lexical analysis, syntax analysis, semantic analysis, and intermediate code generation might be grouped into one pass.
* If so, the token stream after lexical analysis may be translated directly into intermediate code. In more detail, we may think of the syntax analyzer as being "in charge". It attempts to discover the grammatical structure on the tokens it sees; it obtains tokens as it needs them, by calling the lexical analyzer to find the next token. As the grammatical structure is discovered, the parser calls the intermediate code generator to perform semantic analysis and generate a portion of the code.

***Reducing the Number of Passes***

* It is desirable to have relatively few passes, since it takes time to read and write intermediate files. If we group several phases into one pass, we may be forced to keep the entire program in memory, because one phase may need information in a different order than a previous phase produces it. The internal form of the program may be considerably larger than either the source program or the target program, so this space may not be a trivial matter.
* For some phases, grouping into one pass presents few problems. For example, as we mentioned above, the interface between the lexical and syntactic analyzers can often be limited to a single token.
* On the other hand, it is often very hard to perform code generation until the intermediate representation has been completely generated. For example, languages like PUI and *Algol 68* permit variables to be used before they are declared. We cannot generate the target code for a construct if we do not know the types of variables involved in that construct. Similarly, most languages allow goto's that jump forward in the code. We cannot determine the target address of such a jump until we have seen the intervening source code and generated target code for it.
* In some cases, it is possible to leave a blank slot for missing information, and fill in the slot when the information becomes available. In particular, intermediate and target code generation can often be merged into one pass using a technique called *"back patching."*

**10. Compiler - Construction Tools:**

The compiler writers use software tools such as, debuggers, version managers, profilers, and so on.

The following is a list of some useful compiler-construction tools:

  1. Parser generators

   2. Scanner generators

   3. Syntax-directed translation engines

   4. Automatic code generators

  5. Data-flow engines

**1. Parser generators:** These produce syntax analyser from context free grammar as input.

**2. Scanner generators**: These automatically produce lexical analyser from a specification based on regular expressions.

**3. Syntax-directed translation engines**: These produce collection of routines from parse tree, generating the intermediate code.

**4. Automatic code generators**: Takes collection of rules that define the translation of each operation of the intermediate language into the machine language for the target machine.

**5. Data-flow engines**: To perform good code optimization involves data-flow analysis – gathering of information about how values are transmitted from one part of a program to other part.

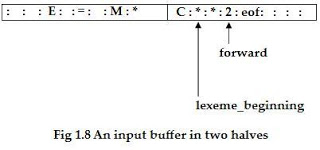
# 11. [Buffering](http://compilerdesigndetails.blogspot.com/2012/02/buffering.html)

**Input Buffering:**

* Some efficiency issues concerned with the buffering of input.
* A two-buffer input scheme that is useful when look ahead on the input is necessary to identify tokens.
* Techniques for speeding up the lexical analyser, such as the use of sentinels to mark the buffer end.
* There are three general approaches to the implementation of a lexical analyser:

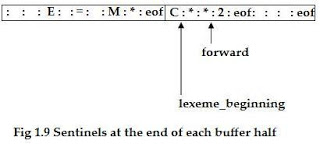
1. Use a lexical-analyser generator, such as Lex compiler to produce the lexical analyser from a regular expression based specification. In this, the generator provides routines for reading and buffering the input.
2. Write the lexical analyser in a conventional systems-programming language, using I/O facilities of that language to read the input.
3. Write the lexical analyser in assembly language and explicitly manage the reading of input.

* ***Buffer pairs*:**
* Because of a large amount of time can be consumed moving characters, specialized buffering techniques have been developed to reduce the amount of overhead required to process an input character.
* The scheme to be discussed:
* Consists a buffer divided into two N-character halves.

[](http://1.bp.blogspot.com/-NYKw_BY9dew/TzIYGQ4IDFI/AAAAAAAAAFw/-F0HO0fGFX0/s1600/aa2.JPG)

N – Number of characters on one disk block, e.g., 1024 or 4096.

* Read N characters into each half of the buffer with one system read command.
* If fewer than N characters remain in the input, then eof is read into the buffer after the input characters.
* Two pointers to the input buffer are maintained.
* The string of characters between two pointers is the current lexeme.
* Initially both pointers point to the first character of the next lexeme to be found.
* Forward pointer, scans ahead until a match for a pattern is found.
* Once the next lexeme is determined, the forward pointer is set to the character at its right end.
* If the forward pointer is about to move past the halfway mark, the right half is filled with N new input characters.
* If the forward pointer is about to move past the right end of the buffer, the left half is filled with N new characters and the forward pointer wraps around to the beginning of the buffer.
* *Disadvantage of this scheme*:
* This scheme works well most of the time, but the amount of lookahead is limited.
* This limited lookahead may make it impossible to recognize tokens in situations where the distance that the forward pointer must travel is more than the length of the buffer.
* For example: DECLARE ( ARG1, ARG2, … , ARGn ) in PL/1 program;
* Cannot determine whether the DECLARE is a keyword or an array name until the character that follows the right parenthesis.
* ***Sentinels*:**
* In the previous scheme, must check each time the move forward pointer that has not moved off one half of the buffer. If it is done, then must reload the other half.
* Therefore the ends of the buffer halves require two tests for each advance of the forward pointer.
* This can reduce the two tests to one if it is extend each buffer half to hold a sentinel character at the end.
* The sentinel is a special character that cannot be part of the source program. (eof character is used as sentinel).

[](http://3.bp.blogspot.com/-e6EAs0IXKc8/TzIYBmcjemI/AAAAAAAAAFo/KNM5hWVn9DI/s1600/aa1.JPG)

* In this, most of the time it performs only one test to see whether forward points to an eof.
* Only when it reach the end of the buffer half or eof, it performs more tests.
* Since N input characters are encountered between eof’s, the average number of tests per input character is very close to 1.

# 12. [Specification of Tokens:](http://compilerdesigndetails.blogspot.com/2012/02/specification-of-tokens.html)

* Regular expressions are notation for specifying patterns.
* Each pattern matches a set of strings.
* Regular expressions will serve as names for sets of strings.
* *Strings and Languages:*
* The term *alphabet or character class* denotes any finite set of symbols.
* e.g., set {0,1} is the binary alphabet.
* The term *sentence and word* are often used as synonyms for the term string.
* The *length of a string* *s* is written as |s| - is the number of occurrences of symbols in *s*.
* e.g., string “banana” is of length six.
* The *empty string* denoted by *ε* – length of empty string is zero.
* The term *language* denotes any set of strings over some fixed alphabet.
* e.g., {ε} – set containing only empty string is language under φ.
* If *x* and *y* are strings, then the *concatenation* of *x* and *y* (written as *xy*) is the string formed by appending *y* to *x*. *x* = *dog* and *y* = *house*; then *xy* is *doghouse*.
* *sε* = *εs* = *s*.
* *s0* = *ε*, *s1* = *s*, *s2* = *ss*, *s3* = *sss*, … so on.

|  |  |
| --- | --- |
| **TERM** | **DEFINITION** |
| Prefix of s | A string obtained by removing zero or more trailing symbols of string s; e.g., ban is a prefix of banana. |
| Suffix of s | A string formed by deleting zero or more of the leading symbols of s; e.g., nana is a suffix of banana. |
| Substring of s | A string obtained by deleting a prefix and a suffix from s; e.g., nan is a substring of banana. |
| Proper prefix, suffix, or substring of s | Any nonempty string x that is a prefix, suffix or substring of s that s <> x. |
| Subsequence of s | Any string formed by deleting zero or more not necessarily contiguous symbols from s; e.g., baaa is a subsequence of banana. |

* *Operations on Languages*:
* There are several operations that can be applied to languages:

|  |  |
| --- | --- |
| **OPERATION** | **DEFINITION** |
| Union of L and M. written LυM | L υ M = { s | s is in L or s is in M } |
| Concatenation of L and M. written LM | LM = { st | s is in L and t is in M } |
| Kleene closure of L.  written L\*. | L\* denotes “zero or more concatenation of” L. |
| Positive closure of L.  written L+. | L+ denotes “one or more Concatenation of” L. |

* *Regular Expressions*:
* It allows defining the sets to form tokens precisely.
* e.g., *letter ( letter | digit) \**
* Defines a Pascal identifier – which says that the identifier is formed by a letter followed by zero or more letters or digits.
* A regular expression is built up out of simpler regular expressions using a set of defining rules.
* Each regular expression r denotes a language L(r).
* The rules that define the regular expressions over alphabet ∑.
* (Associated with each rule is a specification of the language denoted by the regular expression being defined)
* *1. ε* is a regular expression that denotes {ε}, i.e. the set containing the empty string.
* 2. If *a* is a symbol in ∑, then *a* is a regular expression that denotes {a}, i.e. the set containing the string *a*.
* 3. Suppose r and s are regular expressions denoting the languages L(r) and L(s). Then

1. (r) | (s) is a regular expression denoting the languages L(r) U L(s).
2. (r)(s) is a regular expression denoting the languages L(r)L(s).
3. (r)\* is a regular expression denoting the languages (L(r))\*.
4. (r) is a regular expression denoting the languages L(r).

* A language denoted by a regular expression is said to be a regular set.
* The specification of a regular expression is an example of a recursive definition.
* *Rule (1) and (2)* form the basis of the definition.
* *Rule (3)* provides the inductive step.

|  |  |
| --- | --- |
| **AXIOM** | **DESCRIPTION** |
| r|s = s|r | | is commutative |
| r|(s|t)= (r|s)|t | | is associative |
| (rs)t = r(st) | Concatenation is associative |
| r(s|t) = rs|rt  (s|t)r = sr|tr | Concatenation distributes over | |
| εr = r,  rε = r | ε is the identity element for concatenation |
| r\* = (r|ε)\* | Relation between \* and ε |
| r\*\* = r\* | \* Is idempotent |

* *Regular Definition*:
* If ∑ is an alphabet of basic symbols, then a regular definition is a sequence of definitions of the form
* d1 → r1
* d2 → r2
* …
* dn → rn
* Where each di is a distinct name, and each ri is a regular expression over the symbols in ∑ U {d1, d2, … , di-1}, i.e., the basic symbols and the previously defined names.
* e.g. (regular definition in bold):
* letter → A|B|…|Z|a|b|…|z
* digit → 0|1|…|9
* id → letter ( letter | digit ) \*
* *Notational Shorthand*:
* This shorthand is used in certain constructs that occur frequently in regular expressions.
  + *one or more instance*: unary postfix operator *+* means *“one or more instances of”*. If *r* is a regular expression that denotes the language L(r), the *r+* is a regular expression that denotes the language (L(r))+. Similarly unary postfix operator *\** means *“zero or more instances of”*. The two algebraic identities *r\** and *r+* relate the *kleene* and *positive closure* *operators*.
  + *zero or one instance*: unary postfix operator ? means *“zero or one instance of”*. The notation *r?* is a shorthand for r|ε.
  + *character class*: the notation [abc] is a shorthand for a|b|c.