18CSC304J-Compiler Design

Preliminaries Required

- Basic knowledge of programming languages.
- Basic knowledge of FSA and CFG.
- Knowledge of a high programming language for the programming assignments.

Textbook:

Alfred V. Aho, Ravi Sethi, and Jeffrey D. Ullman,

"*Compilers: Principles, Techniques, and Tools*"

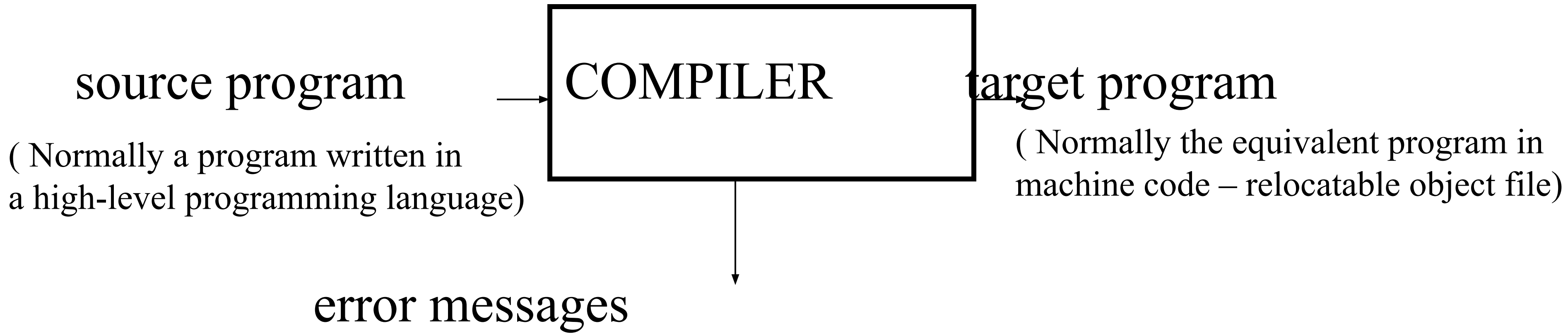
Addison-Wesley, 1986.

Course Outline

- Introduction to Compiling
- Lexical Analysis
- Syntax Analysis
  - Context Free Grammars
  - Top-Down Parsing, LL Parsing
  - Bottom-Up Parsing, LR Parsing
- Syntax-Directed Translation
  - Attribute Definitions
  - Evaluation of Attribute Definitions
- Semantic Analysis, Type Checking
- Run-Time Organization
- Intermediate Code Generation

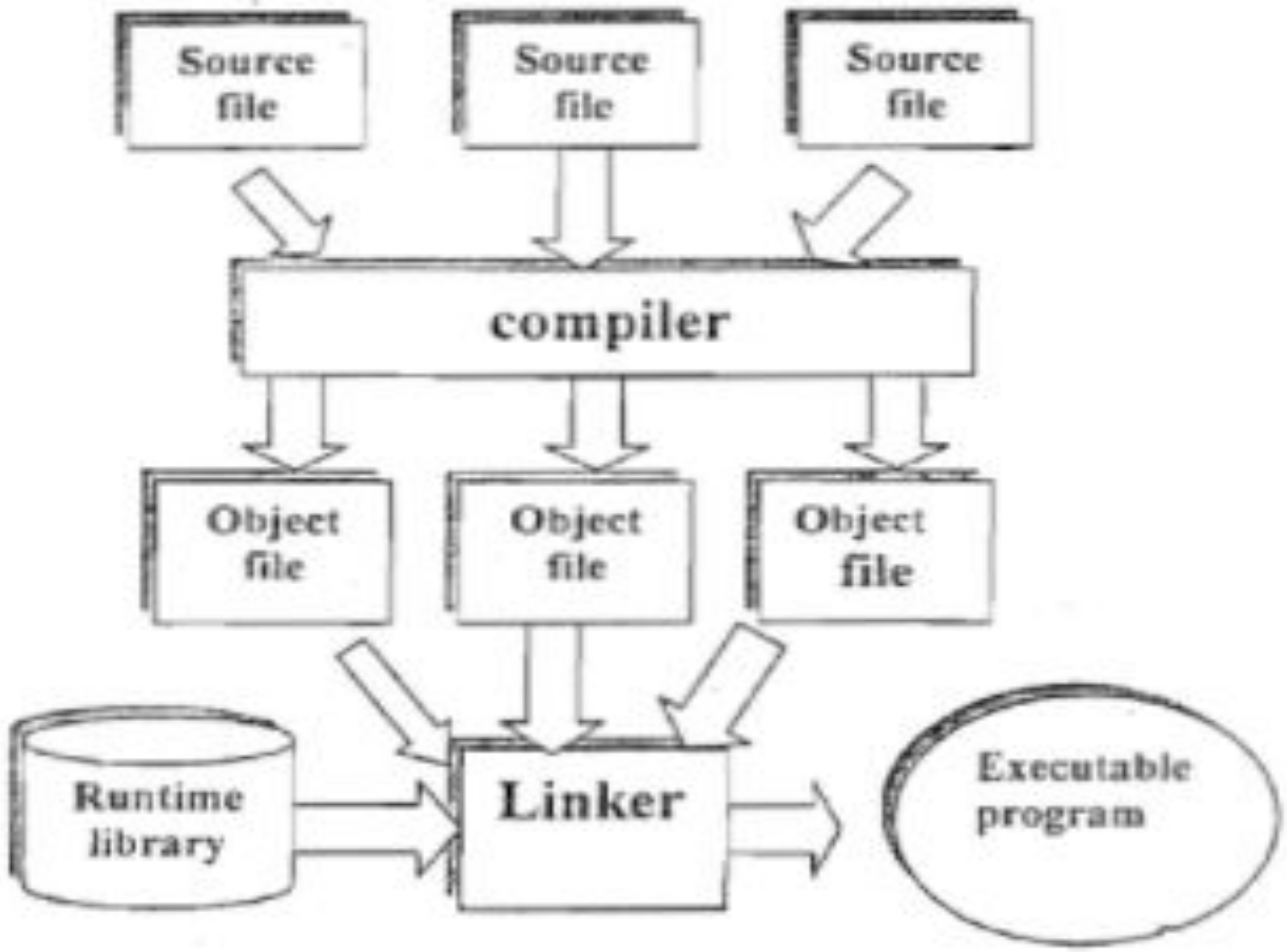
COMPILERS

- A **compiler** is a program takes a program written in a source language and translates it into an equivalent program in a target language.

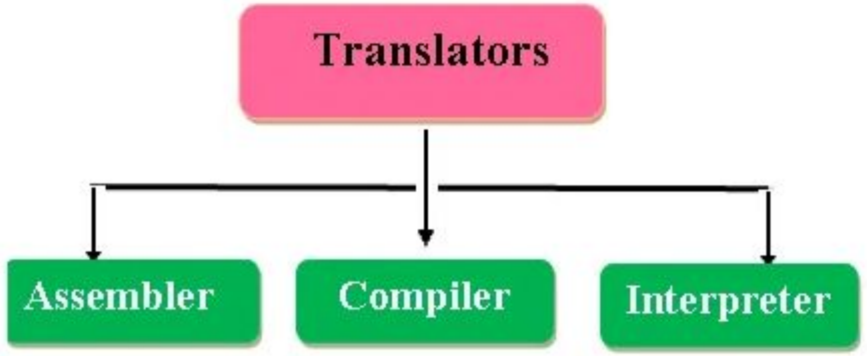


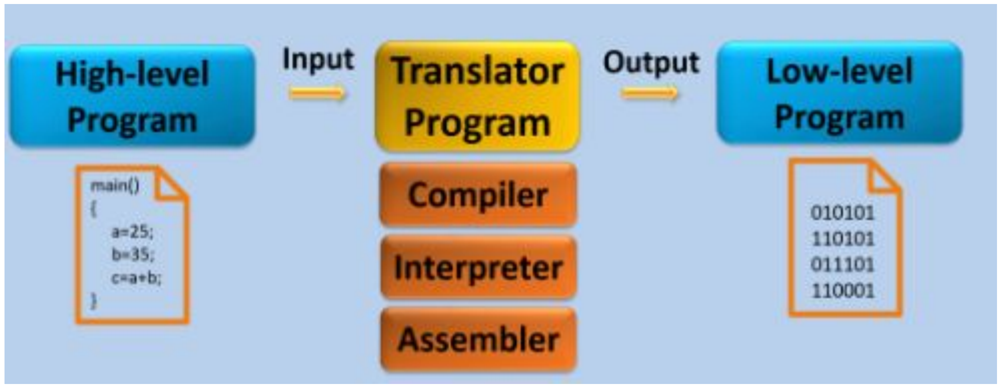
(Normally a program written in a high-level programming language)

(Normally the equivalent program in machine code – relocatable object file)

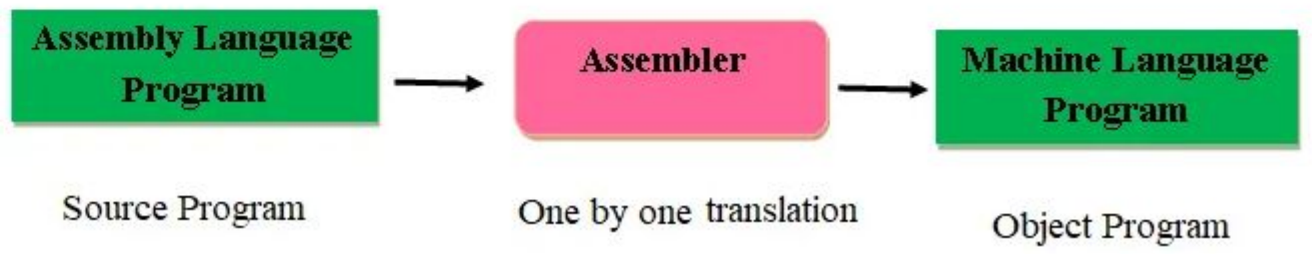

Source file
Source file
Source file
compiler
Object file
Object file
Object file
Runtime library
Linker
Executable program

Translators

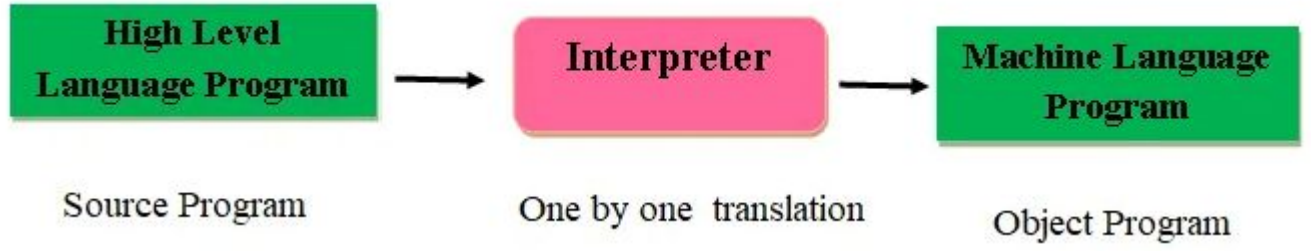




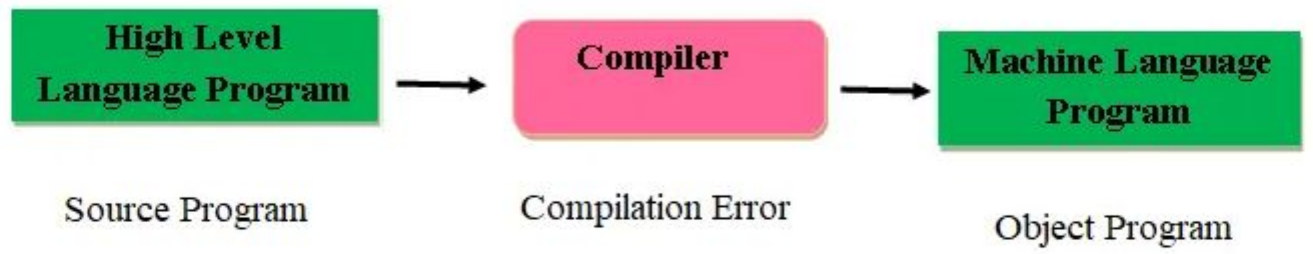
How Assembler Works



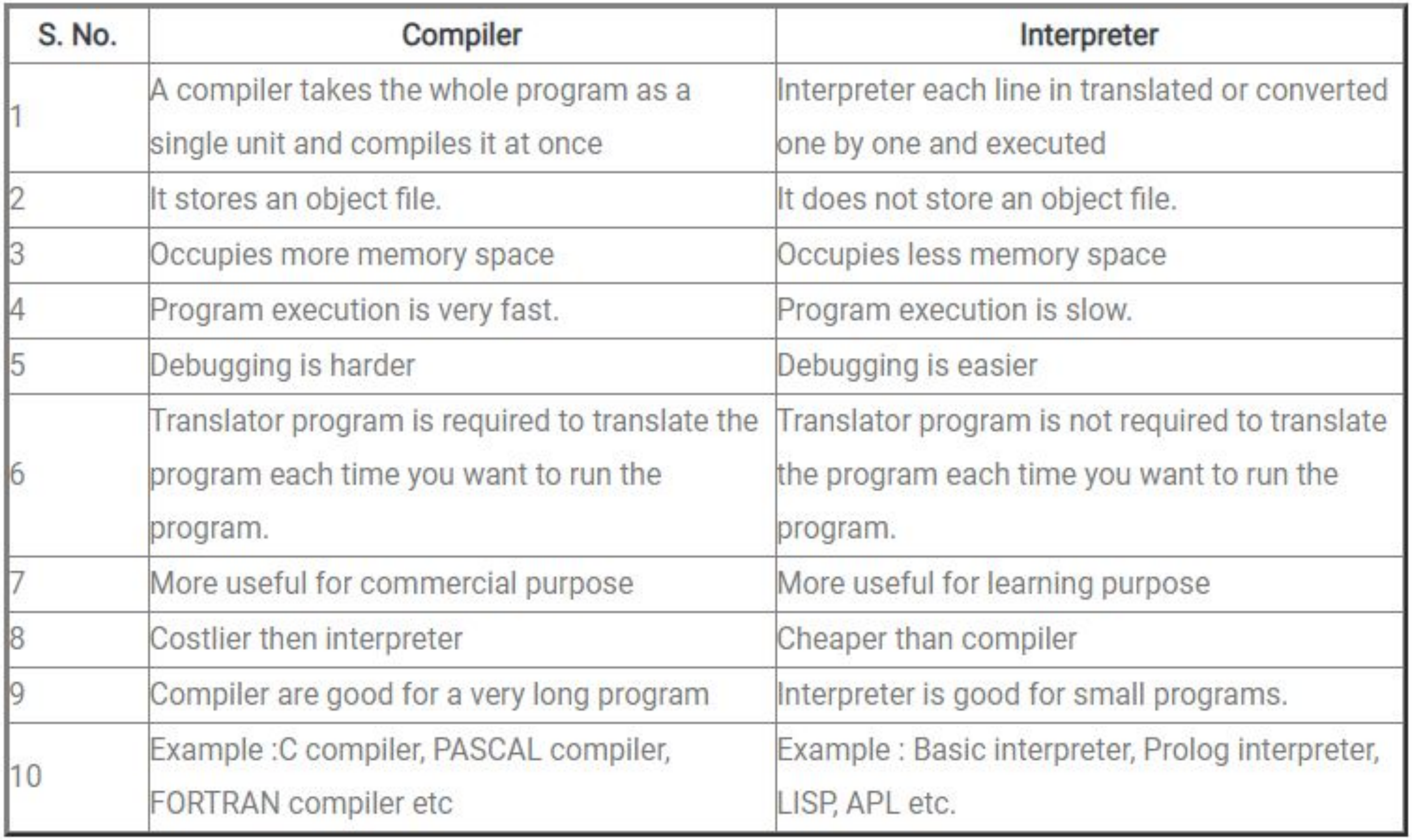
How Interpreter Works



How Compiler Works

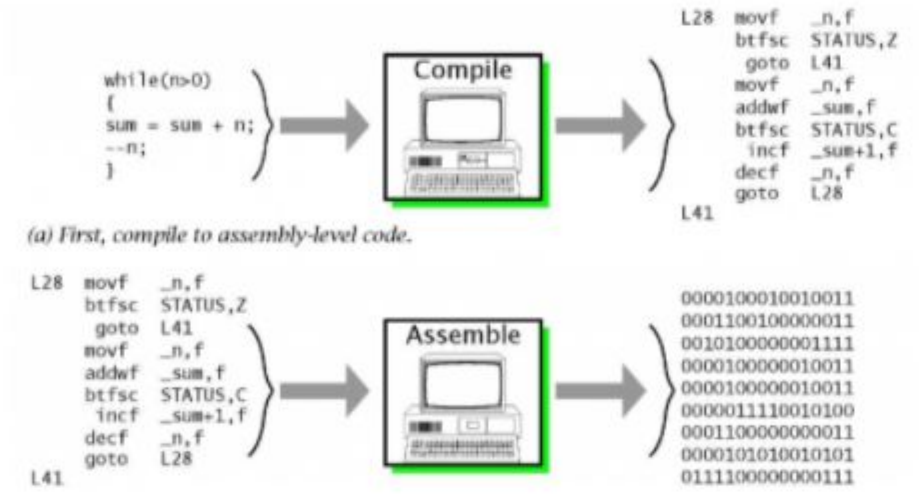


Compiler vs Interpreter



S. No.	Compiler	Interpreter
1	A compiler takes the whole program as a single unit and compiles it at once	Interpreter each line in translated or converted one by one and executed
2	It stores an object file.	It does not store an object file.
3	Occupies more memory space	Occupies less memory space
4	Program execution is very fast.	Program execution is slow.
5	Debugging is harder	Debugging is easier
6	Translator program is required to translate the program each time you want to run the program.	Translator program is not required to translate the program each time you want to run the program.
7	More useful for commercial purpose	More useful for learning purpose
8	Costlier then interpreter	Cheaper than compiler
9	Compiler are good for a very long program	Interpreter is good for small programs.
10	Example :C compiler, PASCAL compiler, FORTRAN compiler etc	Example : Basic interpreter, Prolog interpreter, LISP, APL etc.

Compilers vs Assemblers



```
while(n>0)
{
sum = sum + n;
--n;
}
```

```
L28  movf   _n,f
     btfsc  STATUS,Z
      goto  L41
     movf   _n,f
     addwf  _sum,f
     btfsc  STATUS,C
      incf  _sum+1,f
     decf   _n,f
     goto   L28
L41
```

(a) First, compile to assembly-level code.

```
L28  movf   _n,f
     btfsc  STATUS,Z
      goto  L41
     movf   _n,f
     addwf  _sum,f
     btfsc  STATUS,C
      incf  _sum+1,f
     decf   _n,f
     goto   L28
L41
```

```
0000100010010011
0001100100000011
0010100000001111
0000100000010011
0000100000010011
0000011110010100
0001100000000011
0000101010010101
0111100000000111
```

Other Applications

- In addition to the development of a compiler, the techniques used in compiler design can be applicable to many problems in computer science.
  - Techniques used in a lexical analyzer can be used in text editors, information retrieval system, and pattern recognition programs.
  - Techniques used in a parser can be used in a query processing system such as SQL.
  - Many software having a complex front-end may need techniques used in compiler design.
    - A symbolic equation solver which takes an equation as input. That program should parse the given input equation.
  - Most of the techniques used in compiler design can be used in Natural Language Processing (NLP) systems.

Analysis-Synthesis model of compilation

- There are two major parts of a compiler: **Analysis** and **Synthesis**

- In analysis phase, an intermediate representation is created from the given source program.
  - Lexical Analyzer, Syntax Analyzer and Semantic Analyzer are the parts of this phase.
- In synthesis phase, the equivalent target program is created from this intermediate representation.
  - Intermediate Code Generator, Code Generator, and Code Optimizer are the parts of this phase.

Analysis of source program

Analysis is done in 3 phases:

1.Linear Analysis:

Stream of characters are read from left to right and grouped into tokens

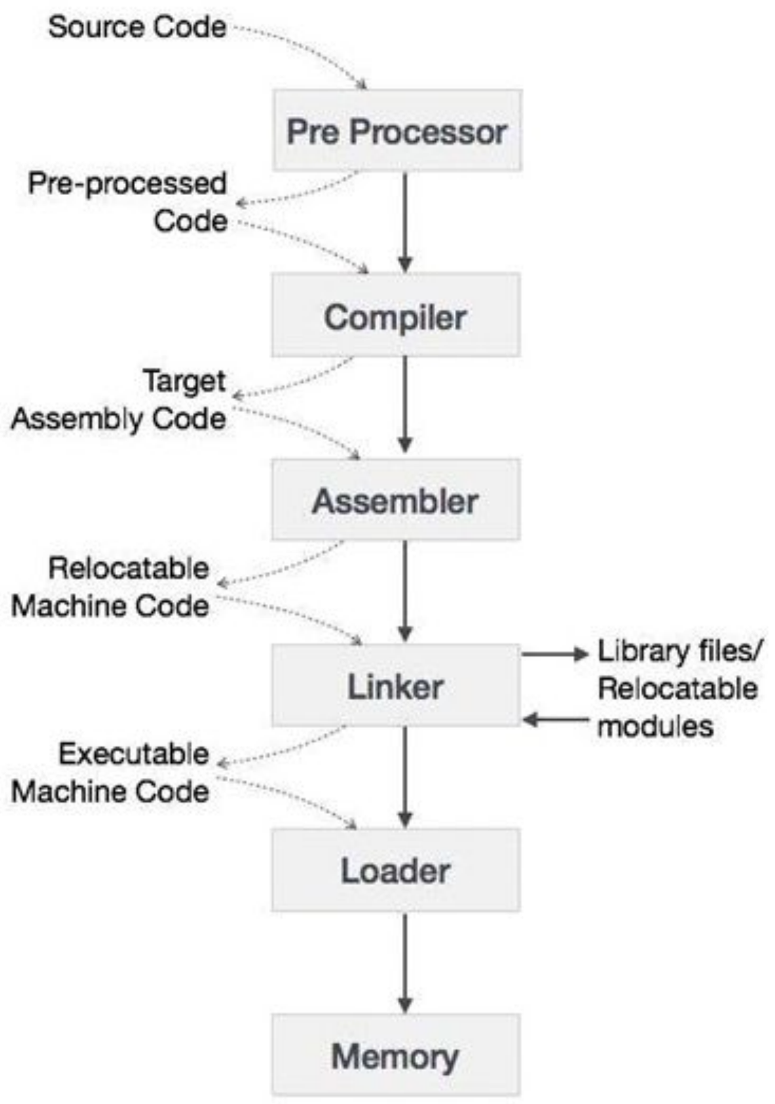
2. **Hierarchical Analysis (Syntax):**

Tokens are grouped hierarchically into nested collections

3. Semantic Analysis:

Checks inherent meaning of code.

Language Processing System



1) Preprocessor

converts the HLL (high level language) into pure high level language. It includes all the header files and also evaluates if any macro is included. It is the optional because if any language which does not support #include and macro preprocessor is not required.

2) Compiler: takes pure high level language as a input and convert into assembly code.

3) Assembler: takes assembly code as an input and converts it into assembly code.

4) Linking and loading:

It has four functions

1. Allocation: get the memory portions from operating system and storing the object data.

2. Relocation: maps the relative address to the physical address and relocating the object code.

3. Linker:

It combines all the executable object module to pre single executable file.

4. Loader:

It loads the executable file into permanent storage.

Phases of A Compiler

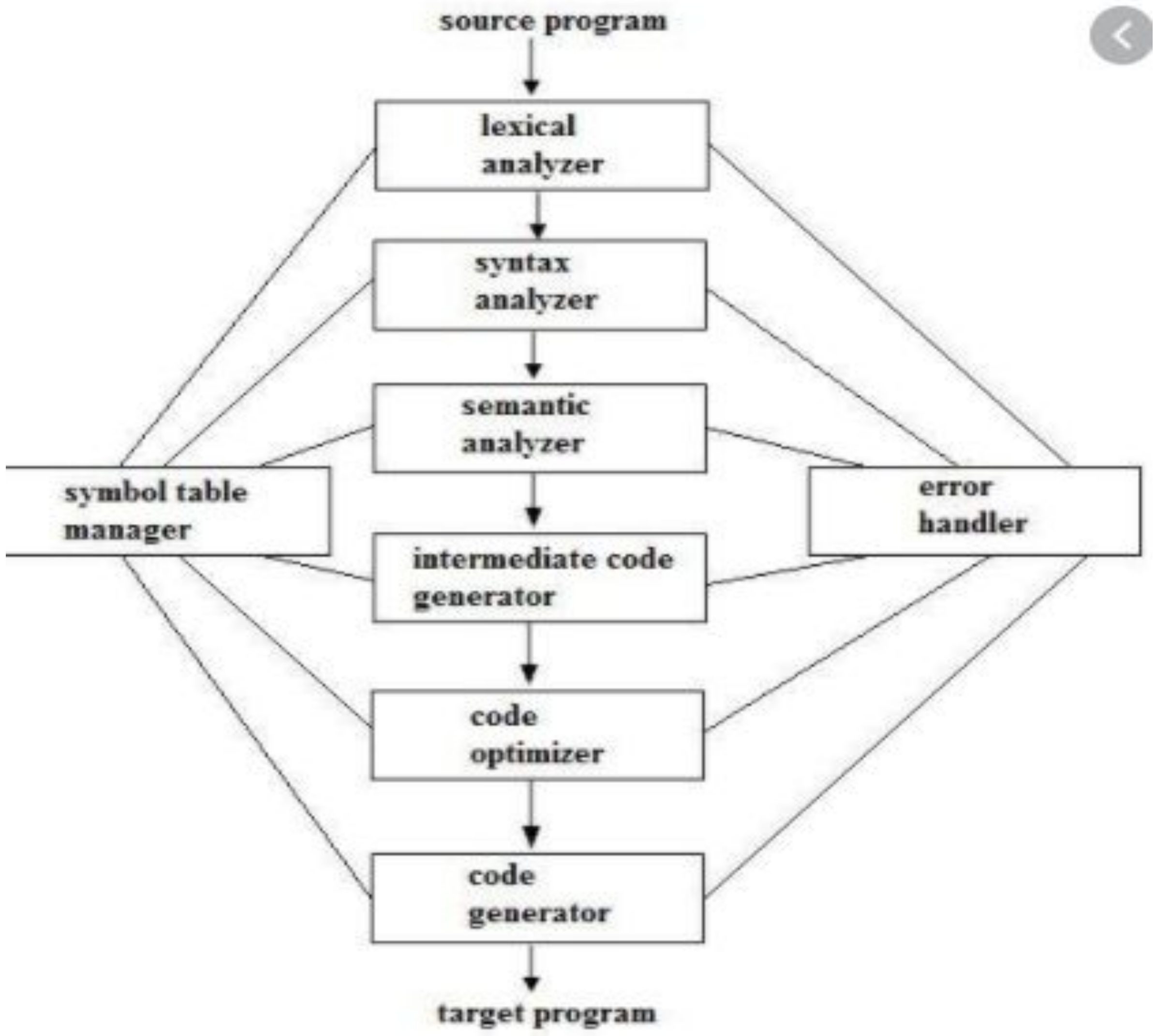
Grouping of Phases:

- **Analysis Phase:**

Lexical, Syntax, Semantic

- **Synthesis Phase:**

Intermediate Code generation, Code optimizer, Code generator



Lexical Analyzer

- **Lexical Analyzer** reads the source program character by character and returns the *tokens* of the source program.
- A *token* describes a pattern of characters having same meaning in the source program. (such as identifiers, operators, keywords, numbers, delimeters and so on)

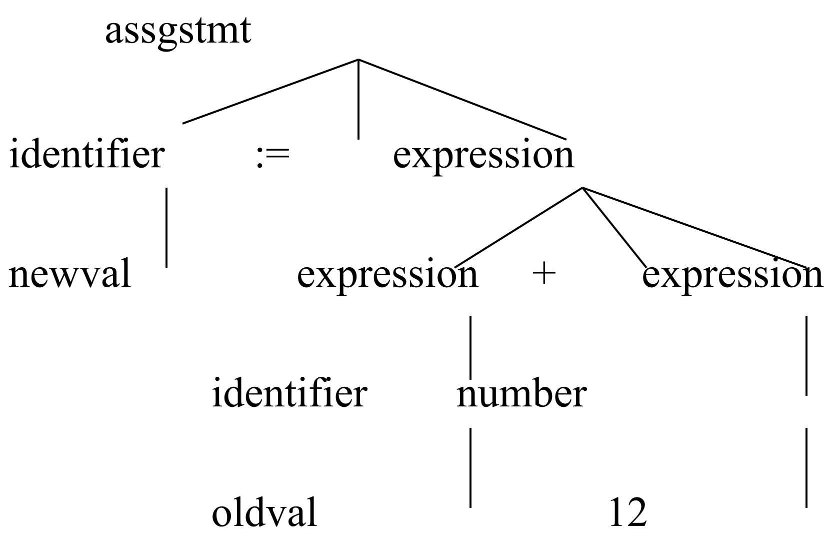
Ex: newval := oldval + 12 => tokens:

newval	identifier
:=	assignment operator
oldval	identifier
+	add operator
12	a number

- Puts information about identifiers into the symbol table.
- Regular expressions are used to describe tokens (lexical constructs).
- A (Deterministic) Finite State Automaton can be used in the implementation of a lexical analyzer.

Syntax Analyzer

- A **Syntax Analyzer** creates the syntactic structure (generally a parse tree) of the given program.
- A syntax analyzer is also called as a **parser**.
- A **parse tree** describes a syntactic structure.



- In a parse tree, all terminals are at leaves.

- All inner nodes are non-terminals in a context free grammar.

Syntax Analyzer (CFG)

- The syntax of a language is specified by a **context free grammar** (CFG).
- The rules in a CFG are mostly recursive.
- A syntax analyzer checks whether a given program satisfies the rules implied by a CFG or not.
  - If it satisfies, the syntax analyzer creates a parse tree for the given program.

- Ex: We use BNF (Backus Naur Form) to specify a CFG

```
assgstmt     ->  identifier  := expression
expression  ->  identifier
expression  ->  number
expression  ->  expression  +  expression
```

Syntax Analyzer versus Lexical Analyzer

- Which constructs of a program should be recognized by the lexical analyzer, and which ones by the syntax analyzer?
  - Both of them do similar things; But the lexical analyzer deals with simple non-recursive constructs of the language.
  - The syntax analyzer deals with recursive constructs of the language.
  - The lexical analyzer simplifies the job of the syntax analyzer.
  - The lexical analyzer recognizes the smallest meaningful units (tokens) in a source program.
  - The syntax analyzer works on the smallest meaningful units (tokens) in a source program to recognize meaningful structures in our programming language.

Parsing Techniques

- Depending on how the parse tree is created, there are different parsing techniques.
- These parsing techniques are categorized into two groups:
  - ***Top-Down Parsing,***
  - ***Bottom-Up Parsing***
- **Top-Down Parsing:**
  - Construction of the parse tree starts at the root, and proceeds towards the leaves.
  - Efficient top-down parsers can be easily constructed by hand.
  - Recursive Predictive Parsing, Non-Recursive Predictive Parsing (LL Parsing).
- **Bottom-Up Parsing:**
  - Construction of the parse tree starts at the leaves, and proceeds towards the root.
  - Normally efficient bottom-up parsers are created with the help of some software tools.
  - Bottom-up parsing is also known as shift-reduce parsing.
  - Operator-Precedence Parsing – simple, restrictive, easy to implement
  - LR Parsing – much general form of shift-reduce parsing, LR, SLR, LALR

Semantic Analyzer

- A semantic analyzer checks the source program for semantic errors and collects the type information for the code generation.
- Type-checking is an important part of semantic analyzer.
- Normally semantic information cannot be represented by a context-free language used in syntax analyzers.
- Context-free grammars used in the syntax analysis are integrated with attributes (semantic rules)
  - the result is a syntax-directed translation,
  - Attribute grammars
- Ex:

```
newval  :=  oldval  +  12
```

  - The type of the identifier *newval* must match with type of the expression *(oldval+12)*

Intermediate Code Generation

- A compiler may produce an explicit intermediate codes representing the source program.
- These intermediate codes are generally machine (architecture independent). But the level of intermediate codes is close to the level of machine codes.
- Ex:

$$\begin{tikzcd}[row sep=large]
\text{newval := oldval * fact + 1} \arrow[d] \\
\text{id1 := id2 * id3 + 1} \arrow[d] \\
\text{MULT id2,id3,temp1}
\end{tikzcd}$$

```
MULT    id2,id3,temp1
ADD     temp1,#1,temp2
MOV     temp2,,id1
```

Intermediates Codes (Quadraples)

Code Optimizer (for Intermediate Code Generator)

- The code optimizer optimizes the code produced by the intermediate code generator in the terms of time and space.

- Ex:

```
MULT    id2,id3,temp1
ADD     temp1,#1,id1
```

Code Generator

- Produces the target language in a specific architecture.
- The target program is normally is a relocatable object file containing the machine codes.

- Ex:

(assume that we have an architecture with instructions whose at least one of its operands is a machine register)

```
MOVE       id2,R1
MULTid3,R1
ADD  #1,R1
MOVE       R1,id1
```

Symbol table

- It is a data structure created and maintained by compilers in order to store information about the occurrence of various entities such as variable names, function names, objects, classes, interfaces, etc.
- Symbol table is used by both the analysis and the synthesis parts of a compiler.

Uses:

- To store the names of all entities in a structured form at one place.
- To verify if a variable has been declared.
- To implement type checking, by verifying assignments and expressions in the source code are semantically correct.
- To determine the scope of a name (scope resolution).

Error handler

- The tasks of the **Error Handling** process are to detect each error, report it to the user, and then make some recover strategy and implement them to handle error.
- An **Error** is the blank entries in the symbol table.

Types of Error –

Run-time error:

✔ error which takes place during the execution of a program, and usually happens because of adverse system parameters or invalid input data.

✔ Logic errors, occur when executed code does not produce the expected result. Logic errors are best handled by meticulous program debugging.

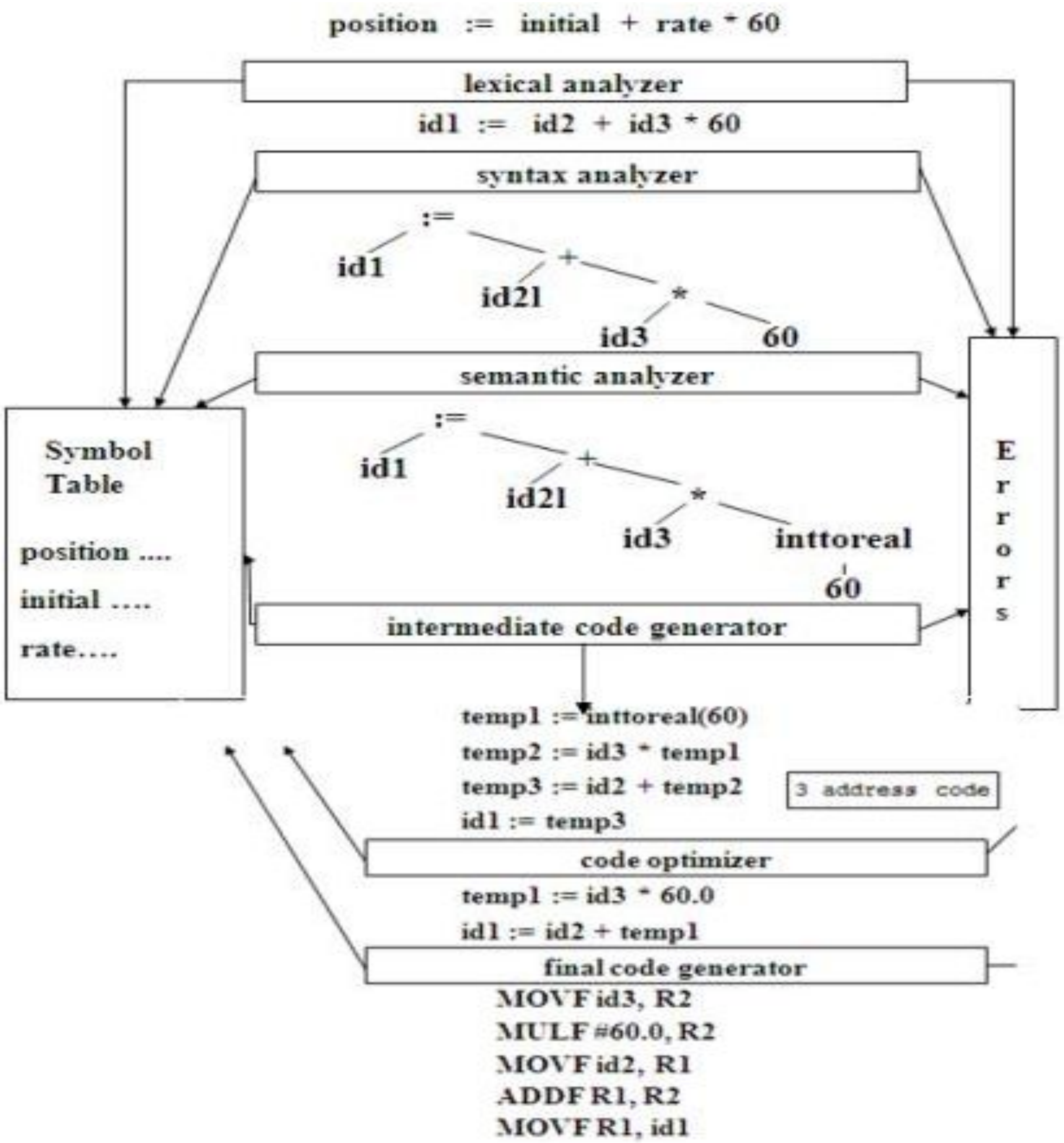
Compile-time errors:

✔ rises at compile time, before execution of the program.

Error handler (…)

Classification of Compile-time error –

- **Lexical** : This includes misspellings of identifiers, keywords or operators
- **Syntax** : missing semicolon or unbalanced parenthesis
- **Semantic** : incompatible value assignment or type mismatches between operator and operand
- **Logical** : code not reachable, infinite loop.


position := initial + rate * 60
lexical analyzer
id1 := id2 + id3 * 60
syntax analyzer
:=
id1
+
id21
*
id3
60
semantic analyzer
:=
id1
+
id21
*
id3
inttoreal
60
Symbol Table
position ....
initial ....
rate....
Errors
intermediate code generator
temp1 := inttoreal(60)
temp2 := id3 * temp1
temp3 := id2 + temp2
id1 := temp3
3 address code
code optimizer
temp1 := id3 * 60.0
id1 := id2 + temp1
final code generator
MOVF id3, R2
MULF #60.0, R2
MOVF id2, R1
ADDF R1, R2
MOVF R1, id1

Error Recovery

Four common error-recovery strategies:

Panic mode

- When a parser encounters an error anywhere in the statement, it ignores the rest of the statement by not processing input from erroneous input to delimiter, such as semi-colon.
- This is the easiest way of error-recovery and also, it prevents the parser from developing infinite loops.

Statement mode

- When a parser encounters an error, it tries to take corrective measures so that the rest of inputs of statement allow the parser to parse ahead.
- For example, inserting a missing semicolon, replacing comma with a semicolon etc.
- Parser designers have to be careful here because one wrong correction may lead to an infinite loop.

Error Recovery (…)

Error productions

•Some common errors are known to the compiler designers that may occur in the code.

•In addition, the designers can create augmented grammar to be used, as productions that generate erroneous constructs when these errors are encountered.

Global correction

•The parser considers the program in hand as a whole and tries to figure out what the program is intended to do and tries to find out a closest match for it, which is error-free.

•When an erroneous input (statement) X is fed, it creates a parse tree for some closest error-free statement Y.

•This may allow the parser to make minimal changes in the source code, but due to the complexity (time and space) of this strategy, it has not been implemented in practice yet.

Cousins of complier

1. Preprocessor:

- A preprocessor is a program that processes its input data to produce output that is used as input to another program.
- The output is said to be a preprocessed form of the input data, which is often used by some subsequent programs like compilers.
- They may perform the following functions : Macro processing, Rational Preprocessors, File Inclusion, Language extension
- **Macro processing:**

✔A macro is a rule or pattern that specifies how a certain input sequence should be mapped to an output sequence according to a defined procedure.

✔The mapping process that instantiates a macro into a specific output sequence is known as macro expansion.

Cousins of complier (…)

- **File Inclusion:**

✔ Preprocessor includes header files into the program text.

✔ When the preprocessor finds an #include directive it replaces it by the entire content of the specified file.

- **Rational Preprocessors:**

✔ These processors change older languages with more modern flow-of-control and data-structuring facilities.

- **Language extension :**

✔ These processors attempt to add capabilities to the language by what amounts to built-in macros.

✔ For example, the language Equel is a database query language embedded in C.

Cousins of complier (…)

2. Assembler

•Assembler creates object code by translating assembly instruction mnemonics into machine code.

•There are two types of assemblers:

✔**One-pass assemblers** go through the source code once and assume that all symbols will be defined before any instruction that references them. ·

✔**Two-pass assemblers** create a table with all symbols and their values in the first pass, and then use the table in a second pass to generate code

Cousins of complier (…)

3. Linker and Loader

•A linker or link editor is a program that takes one or more objects generated by a compiler and combines them into a single executable program.

•Three tasks of the linker are

1.Searches the program to find library routines used by program, e.g. printf(), math routines.

2. Determines the memory locations that code from each module will occupy and relocates its instructions by adjusting absolute references

3. Resolves references among files.

•A loader is the part of an operating system that is responsible for loading programs in memory, one of the essential stages in the process of starting a program.

Grouping of Phases

Front end

Phases: Lexical analysis, Syntax analysis, Semantic analysis, Intermediate code generation.

- Front end comprises of phases which are dependent on the input (source language) and independent on the target machine (target language).
- It includes lexical and syntactic analysis, symbol table management, semantic analysis and the generation of intermediate code.
- Code optimization can also be done by the front end.
- It also includes error handling at the phases concerned.

Grouping of Phases (…)

Back End

•Phases: Code optimizer and code generator

•Back end comprises of those phases of the compiler that are dependent on the target machine and independent on the source language.

•This includes code optimization, code generation.

•In addition to this, it also encompasses error handling and symbol table management operations.

Grouping of Phases (…)

Passes

- The phases of compiler can be implemented in a single pass by marking the primary actions viz. reading of input file and writing to the output file.
- Several phases of compiler are grouped into one pass in such a way that the operations in each and every phase are incorporated during the pass.
- 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.

Grouping of Phases (…)

Reducing the Number of Passes

•Minimizing the number of passes improves the time efficiency as reading from and writing to intermediate files can be reduced.

•When grouping phases into one pass, the entire program has to be kept in memory to ensure proper information flow to each phase because one phase may need information in a different order than the information produced in previous phase.

•The source program or target program differs from its internal representation. So, the memory for internal form may be larger than that of input and output.

Compiler construction tools

Parser Generators

Input: Grammatical description of a programming language
Output: Syntax analyzers.

Parser generator takes the grammatical description of a programming language and produces a syntax analyzer.

Scanner Generators

Input: Regular expression description of the tokens of a language
Output: Lexical analyzers.
Scanner generator generates lexical analyzers from a regular expression description of the tokens of a language.

Syntax-directed Translation Engines

Input: Parse tree.
Output: Intermediate code.
Syntax-directed translation engines produce collections of routines that walk a parse tree and generates intermediate code.

Compiler construction tools (…)

Automatic Code Generators

Input: Intermediate language.
Output: Machine language.
Code-generator takes a collection of rules that define the translation of each operation of the intermediate language into the machine language for a target machine.

Data-flow Analysis Engines

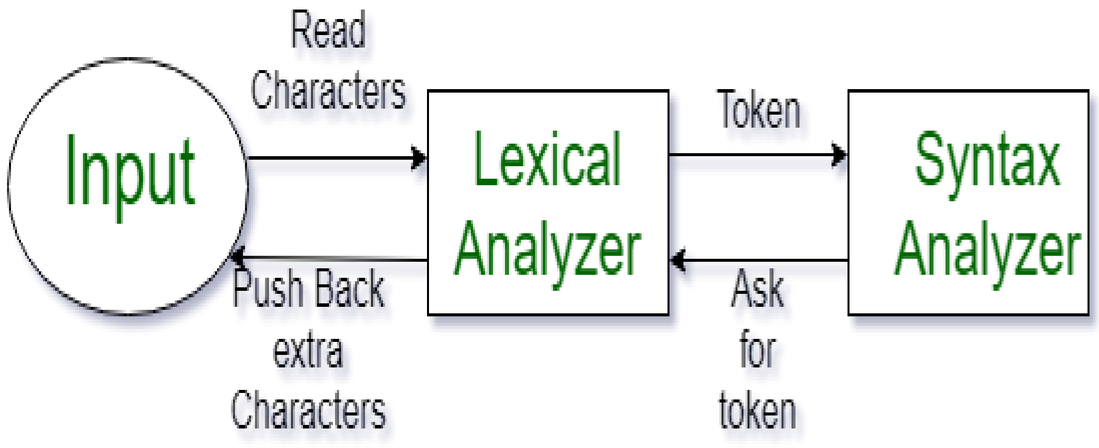
Data-flow analysis engine gathers the information that is, the values transmitted from one part of a program to each of the other parts. Data-flow analysis is a key part of code optimization.

Compiler Construction Toolkits

The toolkits provide integrated set of routines for various phases of compiler. Compiler construction toolkits provide an integrated set of routines for construction of phases of compiler.

LEXICAL ANALYSIS

- Lexical Analysis is the first phase of the compiler also known as a scanner.
- It converts the High level input program into a sequence of **Tokens**.
- Lexical Analysis can be implemented with the Deterministic finite Automata.
- The output is a sequence of tokens that is sent to the parser for syntax analysis.



What is a token?

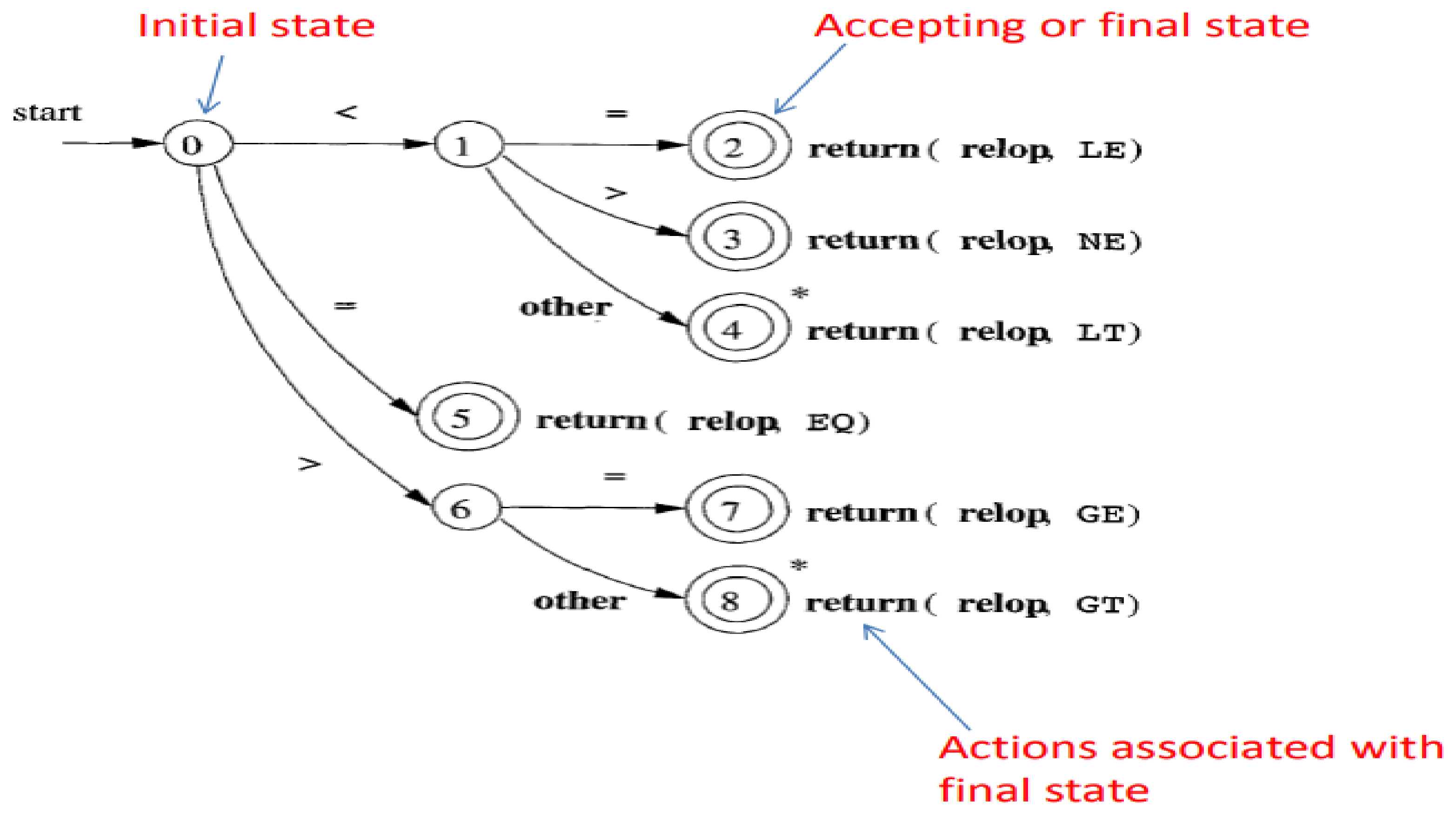
A lexical token is a sequence of characters that can be treated as a unit in the grammar of the programming languages.

Examples:

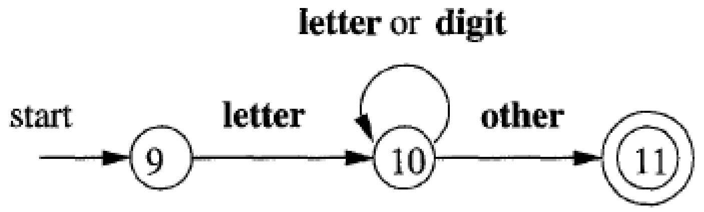
Keywords, Identifier (variable name, function name), Operators, Separators

Not tokens: Comments, preprocessor directive, macros, blanks, tabs, newline.

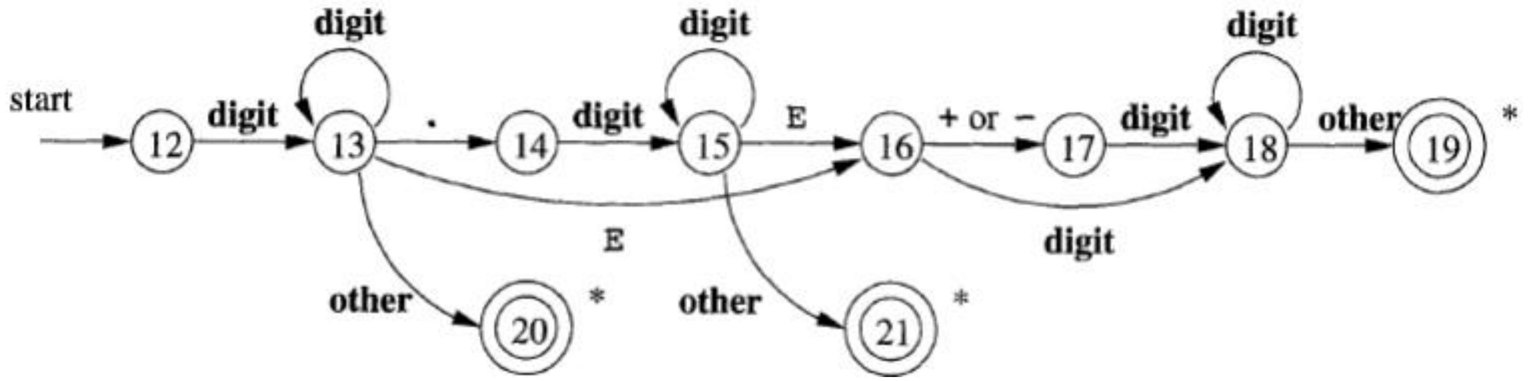
FSA for Relational operators



FSA for naming variables in Fortran



FSA for floating point numbers



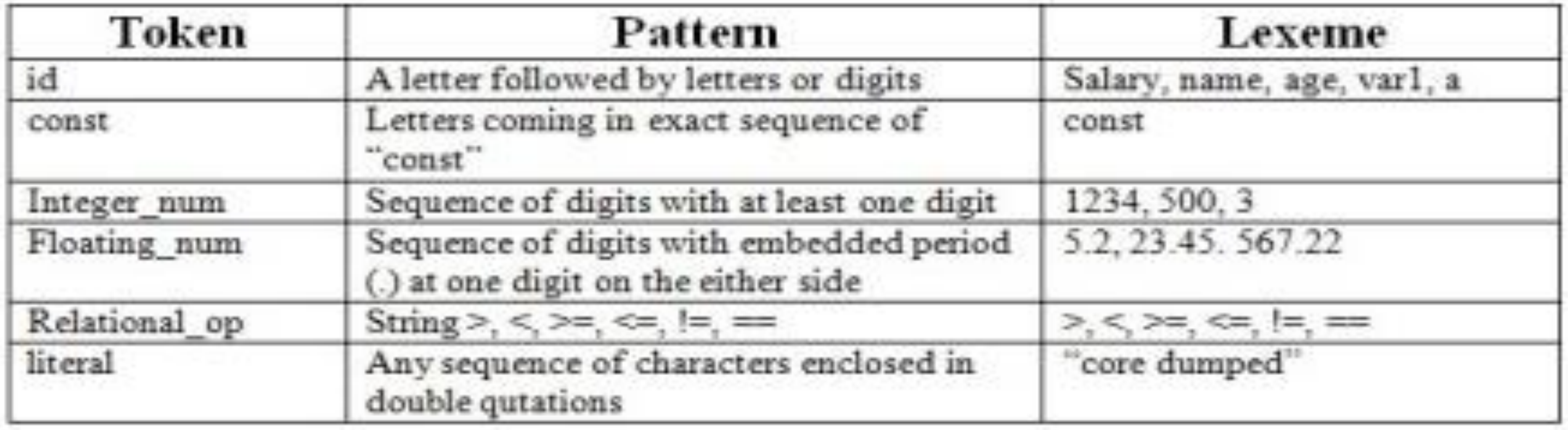
LEXICAL ANALYSIS (…)

Lexemes:

- The sequence of characters matched by a pattern to form the corresponding token or a sequence of input characters that comprises a single token is called a lexeme.
- There are some predefined rules for every lexeme to be identified as a valid token.
- These rules are defined by grammar rules, by means of a pattern.

Patterns:

- A pattern explains what can be a token, and these patterns are defined by means of regular expressions.



Token	Pattern	Lexeme
id	A letter followed by letters or digits	Salary, name, age, var1, a
const	Letters coming in exact sequence of "const"	const
Integer_num	Sequence of digits with at least one digit	1234, 500, 3
Floating_num	Sequence of digits with embedded period (.) at one digit on the either side	5.2, 23.45. 567.22
Relational_op	String >, <, >=, <=, !=, ==	>, <, >=, <=, !=, ==
literal	Any sequence of characters enclosed in double qutations	"core dumped"

LEXICAL ANALYSIS (…)

Functions of Lexical Analyser:

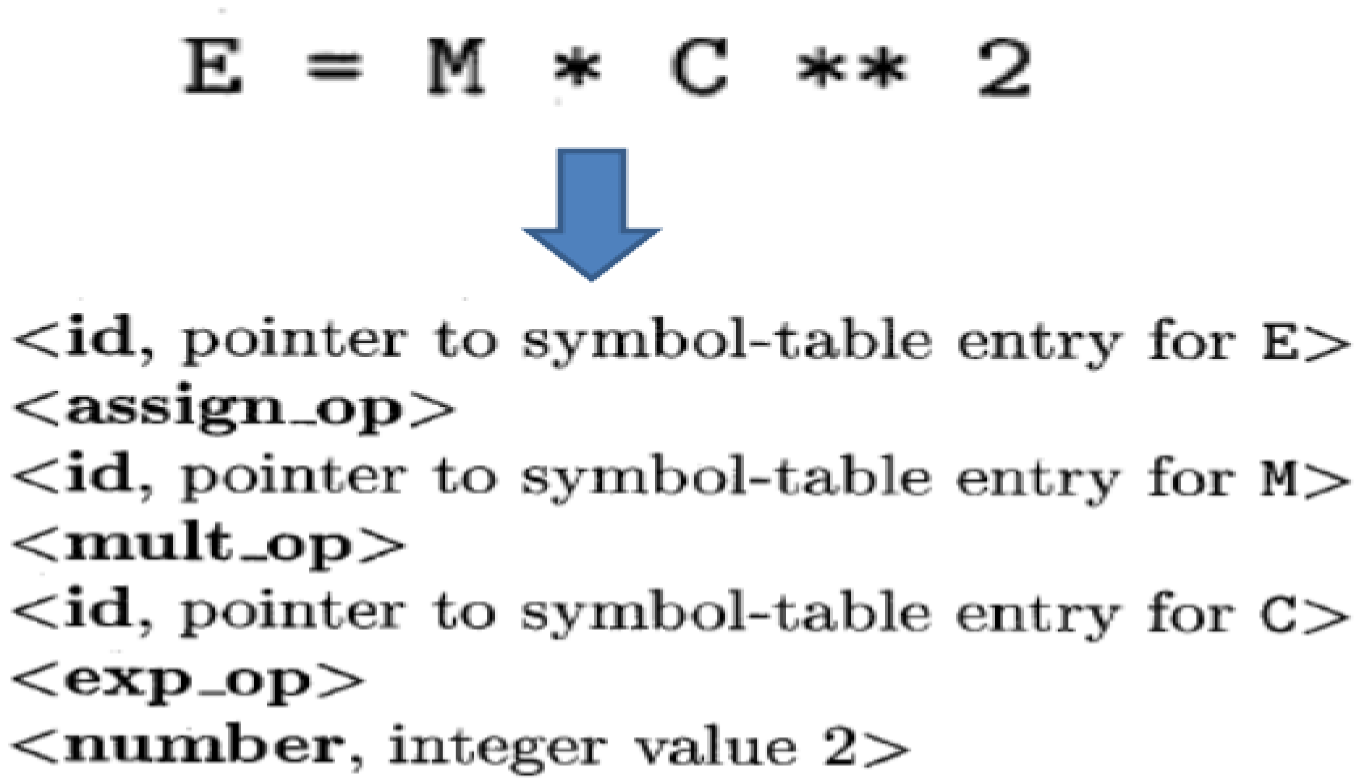
1. Tokenization i.e. Dividing the program into valid tokens.
2. Remove white space characters.
3. Remove comments.
4. It also provides help in generating error messages by providing row numbers and column numbers.

Issues in Lexical Analysis:

Reasons of separating analysis phase into lexical and syntax:

1. **Simplicity:** Separation of lexical and syntax analysis allows us to simplify these phases.
2. **Efficiency:** A large amount of time is spent reading the source program and partitioning into tokens. Separate lexical and syntax analyzer working in parallel can significantly improve the performance.
3. **Portability:** Mostly the lexical analysis phase is different for different languages and other phases are almost the same. To construct a compiler for C++ language we only have to develop the lexical analysis phase and use the other phases of pascal language.

Attributes for tokens



```
E = M * C ** 2
```

<**id**, pointer to symbol-table entry for `E`>
<**assign_op**>
<**id**, pointer to symbol-table entry for `M`>
<**mult_op**>
<**id**, pointer to symbol-table entry for `C`>
<**exp_op**>
<**number**, integer value 2>

Specification of Tokens

Alphabets: Any finite set of symbols.
Eg: {0,1} is a set of binary alphabets

Strings: Any finite sequence of alphabets is called a string. Length of the string is the total number of occurrence of alphabets. A string of zero length is known as an empty string and is denoted by ε

Language: A language is considered as a finite set of strings over some finite set of alphabets. Computer languages are considered as finite sets, and mathematically set operations can be performed on them. Finite languages can be described by means of regular expressions.

Longest Match Rule
When the lexical analyzer read the source-code, it scans the code letter by letter; and when it encounters a whitespace, operator symbol, or special symbols, it decides that a word is completed.
Eg: int intvalue;

- The Longest Match Rule states that the lexeme scanned should be determined based on the longest match among all the tokens available.
- The lexical analyzer also follows **rule priority** where a reserved word, e.g., a keyword, of a language is given priority over user input.

Specification of Tokens (…)

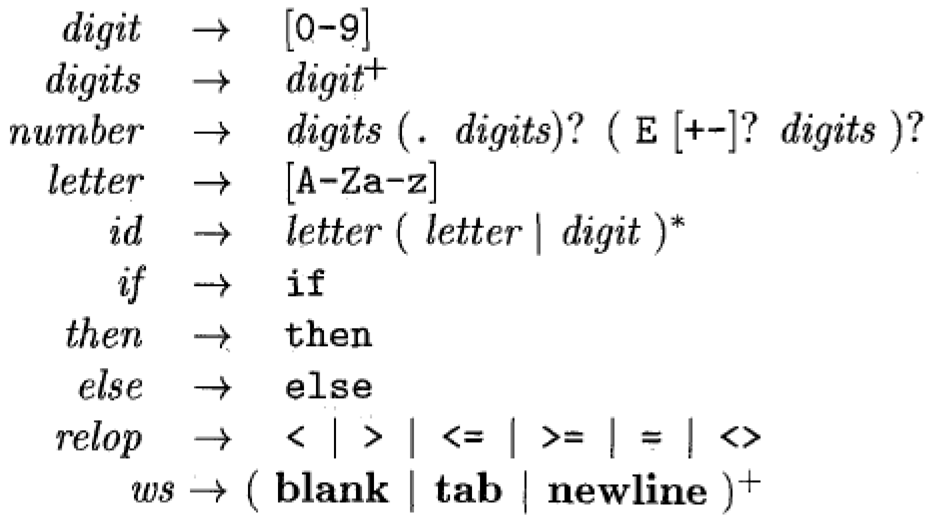
Prefix of String:
A part of a string which is obtained by zero or more trailing characters/symbols for a string
Example prefix of string “apple” is apple, appl, app, ap etc.
Suffix of String:
A part of a string which is obtained by deleting zero or more leading symbols for a string
Example of string “apple” is apple, pple, ple, le etc.
Substring :
Any string obtained by deleting a prefix or suffix from a string is called substring
Example, substring of “apple” is ppl etc.
Proper prefix, suffix, or substring:
A proper prefix, suffix or substring is that part “x” of the string “s” which is a prefix, a suffix or substring of string “s” such that $x \neq s$
Subsequence:
A string formed by deleting zero or more not necessarily contiguous symbols from string is call subsequence.

Specification of Tokens (…)

Regular Expression:

- Regular expression can be used to specify the structure of tokens used in the programming language.
- Regular expressions defines the patterns for the tokens.
- When comparing this pattern against a string, it'll either be true or false.
- The set of string describe by a regular expression is called set and the language describe by regular expression is called **regular language.**

Examples:



digit	$\rightarrow$	`[0-9]`
digits	$\rightarrow$	$digit^+$
number	$\rightarrow$	*digits* (`.` *digits*)? (`E` `[+-]`? *digits*)?
letter	$\rightarrow$	`[A-Za-z]`
id	$\rightarrow$	*letter* (*letter* | *digit*)*
if	$\rightarrow$	`if`
then	$\rightarrow$	`then`
else	$\rightarrow$	`else`
relop	$\rightarrow$	`<` | `>` | `<=` | `>=` | `=` | `<>`
ws	$\rightarrow$	(**blank** | **tab** | **newline**)$^+$

Specification of Tokens (…)

Operations on Regular Expressions

- **Union:** If L and M are two regular languages then their union L U M is also a union.

 L U M = {s | s is in L or s is in M}

- **Intersection:** If L and M are two regular languages then their intersection is also an intersection.

L ⋂ M = {st | s is in L and t is in M}

- **Kleen closure:** If L is a regular language then its Kleen closure L1* will also be a regular language.

L* = Zero or more occurrence of language L.

- **Positive Closure:** If L is a regular language then its positive closure $L1^+$ will also be a regular language.

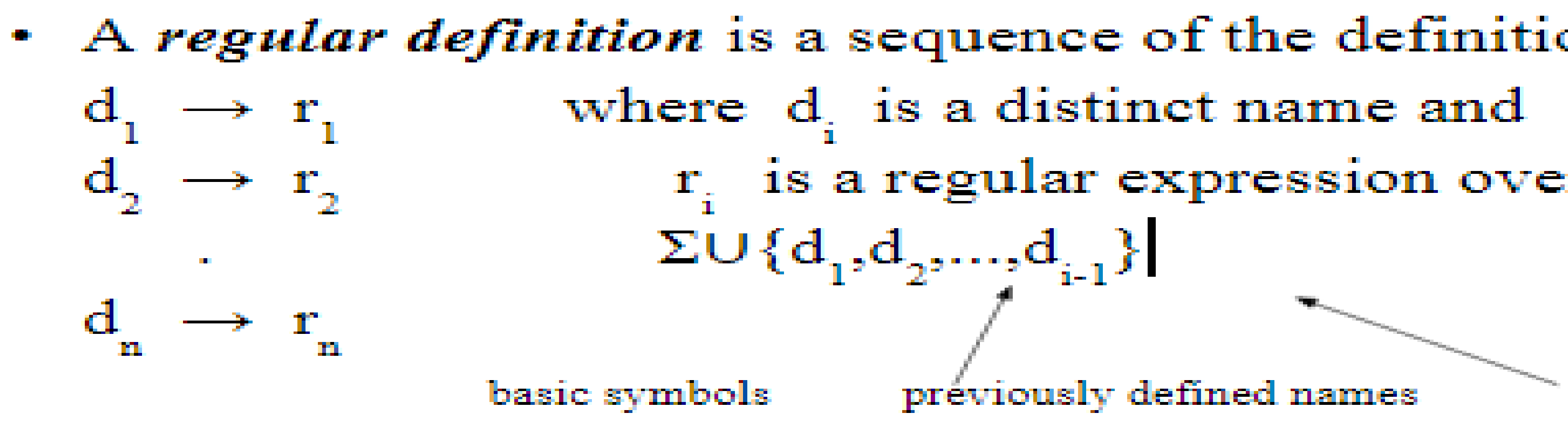
$L^{+=}$ One or more occurrence of language L.

- **?:** If L is a regular language then its positive closure $L1^?$ will also be a regular language

$L^?$ = Zero or one occurrence of language L.

Specification of Tokens (…)

Regular Definitions



- A ***regular definition*** is a sequence of the definitions of the form:

$d_1 \longrightarrow r_1$ where d_i is a distinct name and

$d_2 \longrightarrow r_2$ r_i is a regular expression over symbols in

. $\Sigma \cup \{d_1, d_2, \ldots, d_{i-1}\}$

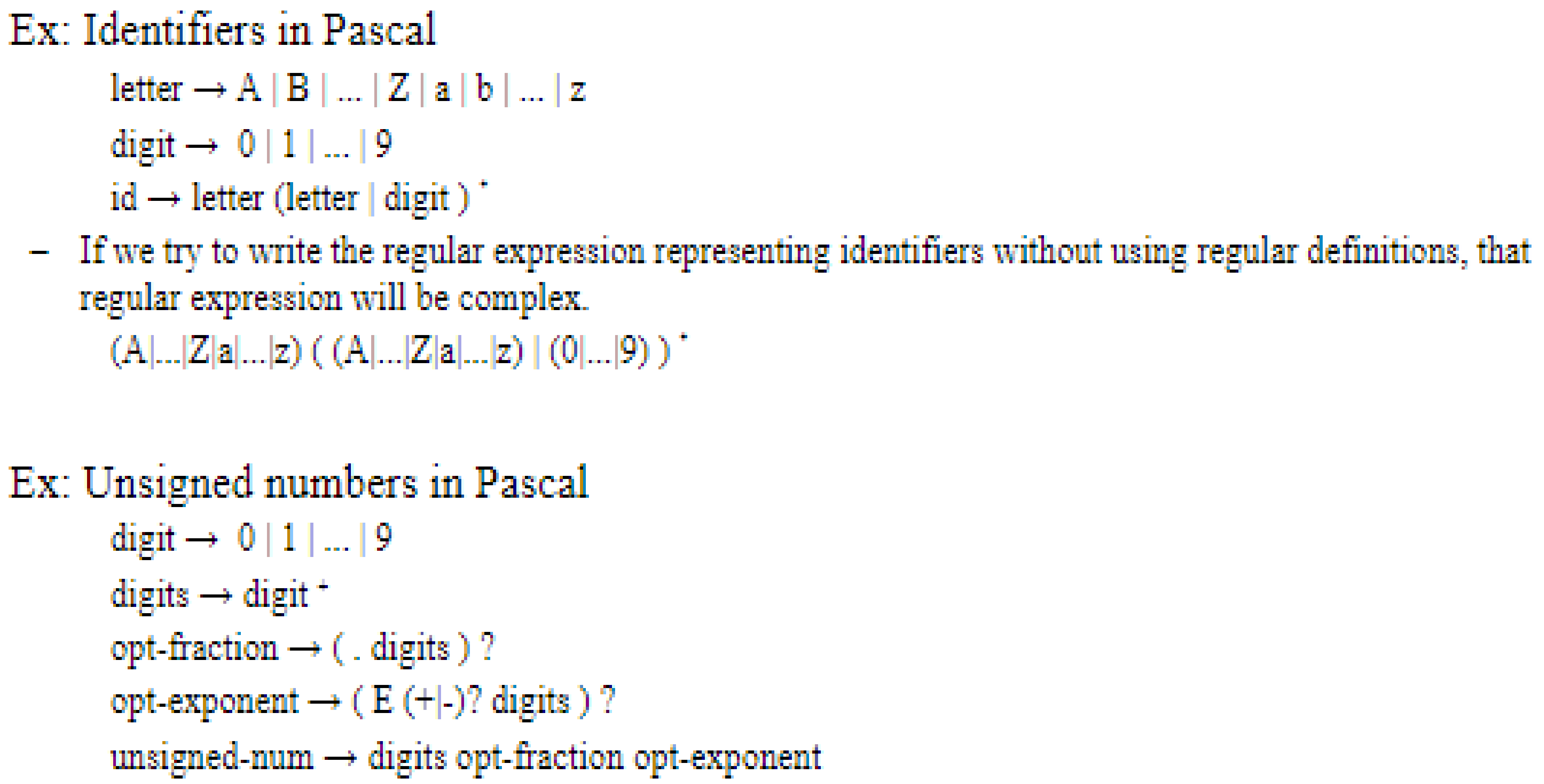
$d_n \longrightarrow r_n$

basic symbols previously defined names

We can give names to regular expressions, and we can use these names as symbols to define other regular expressions.

Specification of Tokens (…)

Examples of Regular Definitions



Ex: Identifiers in Pascal

letter → A | B | ... | Z | a | b | ... | z

digit → 0 | 1 | ... | 9

id → letter (letter | digit)*

– If we try to write the regular expression representing identifiers without using regular definitions, that regular expression will be complex.

(A|...|Z|a|...|z) ((A|...|Z|a|...|z) | (0|...|9))*

Ex: Unsigned numbers in Pascal

digit → 0 | 1 | ... | 9

digits → digit$^+$

opt-fraction → (. digits) ?

opt-exponent → (E (+|-)? digits) ?

unsigned-num → digits opt-fraction opt-exponent

INPUT BUFFERING

Need for Buffering:

- The main task of the lexical analyzer is to read the input characters of the source program group them into lexemes and produce as output a sequence of tokens for each lexeme in the source program.
- When the lexical analyzer discovers a lexeme constituting an identifier, it needs to enter that lexeme into the symbol table.
- To ensure that a right lexeme is found, one or more characters have to be looked up beyond the next lexeme
- Hence a two-buffer scheme is introduced to handle large lookaheads safely.
- The lexical analyzer not only identifies the lexemes but also pre-processes the source text like removing comments, white spaces, etc.

Lexical analyzers are divided into a cascade of two processes:

- **Scanning:** It consists of simple processes that do not require the tokenization of the input such as deletion of comments, compaction of consecutive white space characters into one.
- **Lexical Analysis:** This is the more complex portion where the scanner produces sequence of tokens as output

INPUT BUFFERING (...)

Techniques for speeding up the process of lexical analyzer such as the use of sentinels to mark the buffer end have been adopted.

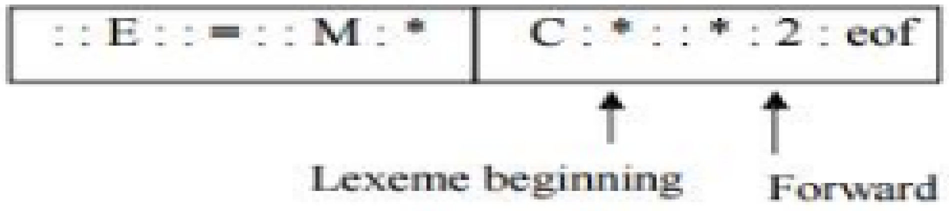
There are three general approaches for the implementation of a lexical analyzer:

1. **By using a lexical-analyzer generator:** In this, the generator provides routines for reading and buffering the input.
2. By writing the lexical analyzer in a conventional systems-programming language, using I/O facilities of that language to read the input.
3. By writing the lexical analyzer in assembly language and explicitly managing the reading of input.

INPUT BUFFERING (...)

Buffer Pairs

Specialized buffering techniques are used to reduce the amount of overhead, which is required to process an input character in moving characters



- Consists of two buffers, each consists of N-character size which are reloaded alternatively.
- Two pointers: lexemeBegin and forward
- Lexeme Begin points to the beginning of the current lexeme which is yet to be found. Forward scans ahead until a match for a pattern is found.
- Once a lexeme is found, lexeme begin is set to the character immediately after the lexeme which is just found and forward is set to the character at its right end.
- Current lexeme is the set of characters between two pointers.

INPUT BUFFERING (...)

Sentinels

- Sentinels is used to make a check, each time when the forward pointer is moved, a check is done to ensure that one half of the buffer has not moved off. If it is done, then the other half must be reloaded. The sentinel is a special character that cannot be part of the source program. (eof character is used as sentinel).
- Therefore the ends of the buffer halves require two tests for each advance of the forward pointer.

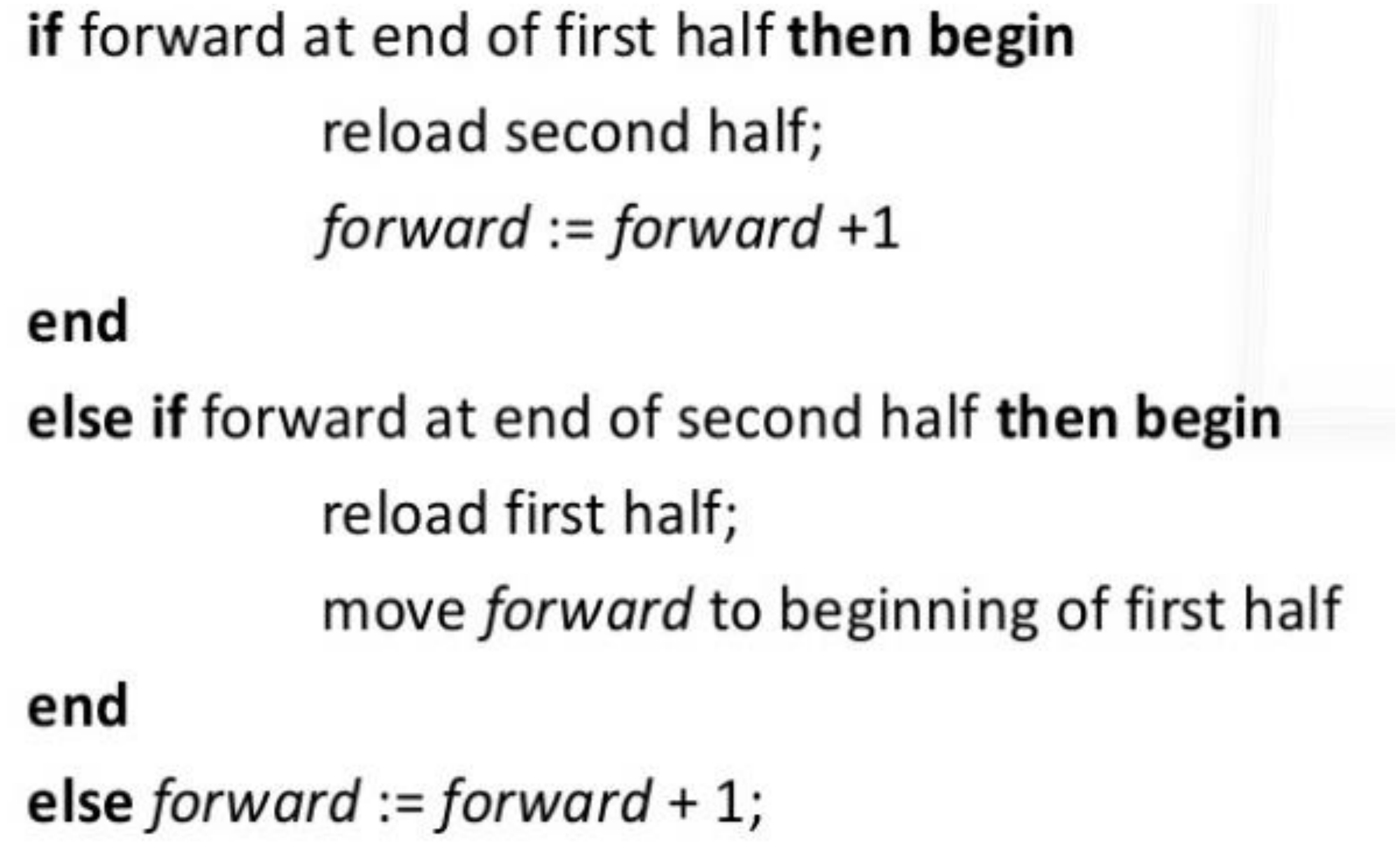
Test 1: For end of buffer.

Test 2: To determine what character is read.

- The usage of sentinel reduces the two tests to one by extending each buffer half to hold a sentinel character at the end.

INPUT BUFFERING (...)

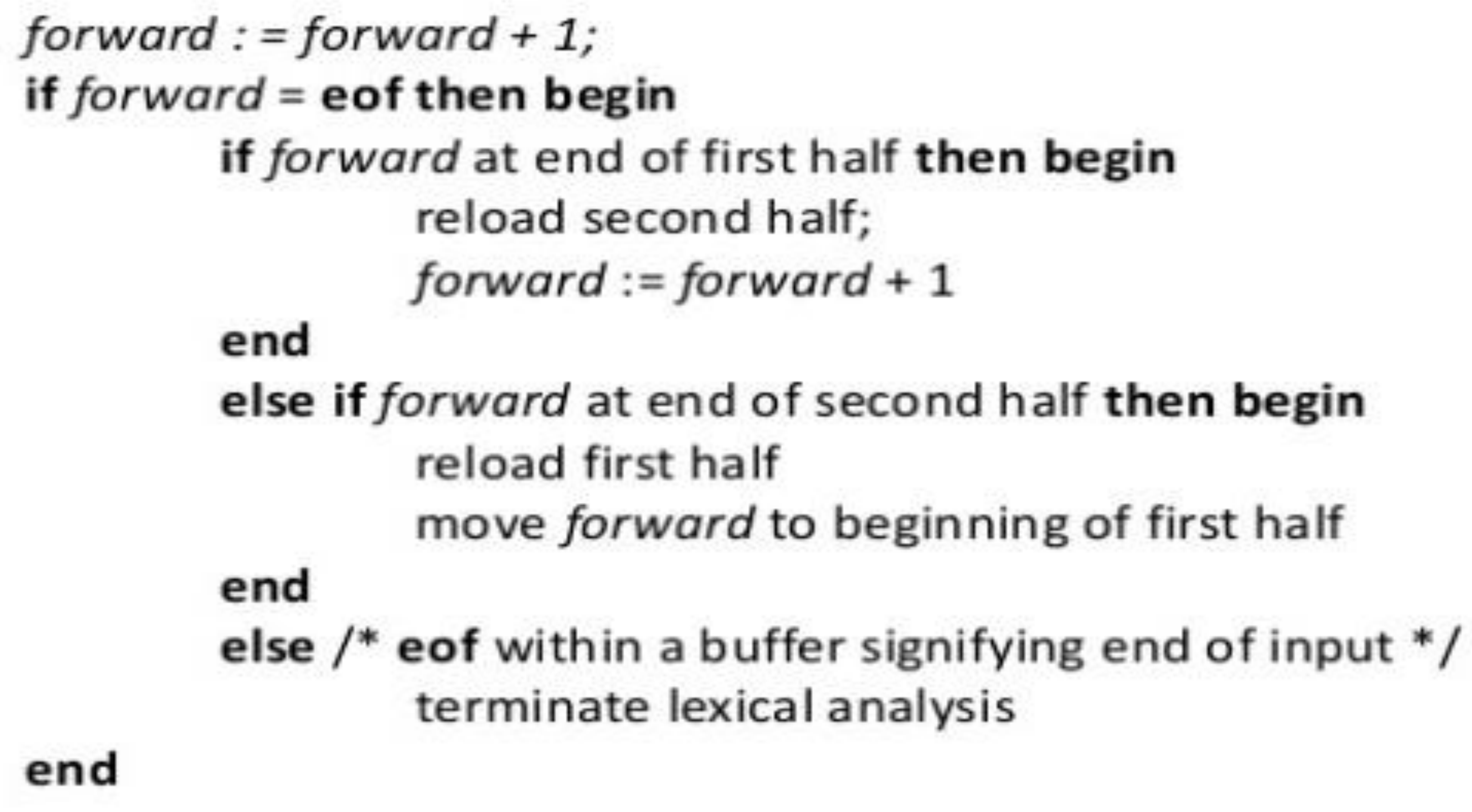
Pseudocode for input buffering



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

INPUT BUFFERING (…)

Pseudocode for input buffering (reduced no. of tests)



```
forward : = forward + 1;
if forward = eof then begin
          if forward at end of first half then begin
                    reload second half;
                    forward := forward + 1
          end
          else if forward at end of second half then begin
                    reload first half
                    move forward to beginning of first half
          end
          else /* eof within a buffer signifying end of input */
                    terminate lexical analysis
end
```

INPUT BUFFERING (...)

Disadvantages:

- 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.

Advantages

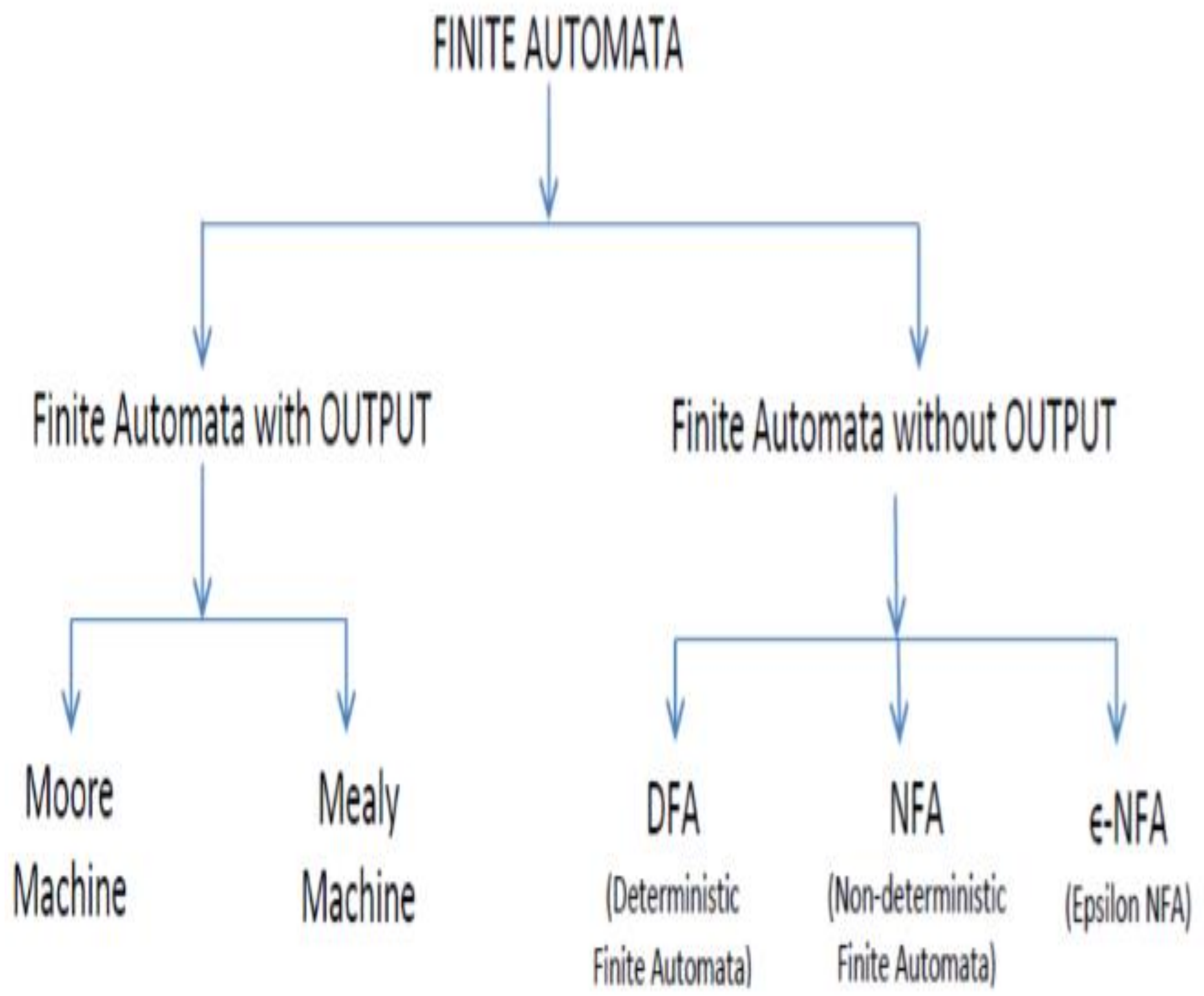
- Most of the time, It performs only one test to see whether forward pointer points to an eof.
- Only when it reaches the end of the buffer half or eof, it performs more tests.
- Since N input characters are encountered between eofs, the average number of tests per input character is very close to 1.

FINITE STATE AUTOMATA

- The finite automata or finite state machine is an abstract machine which have five elements or tuple.
- It has a set of states and rules for moving from one state to another but it depends upon the applied input symbol.
- It is an abstract model of digital computer.
- Formal specification of machine is { Q, Σ, q, F, δ }.

  Q : Finite set of states.
  Σ : set of Input Symbols.
  q : Initial state.
  F : set of Final States.
  δ : Transition Function.

- Two Types: Deterministic Finite state automata (DFA); Non Deterministic Finite State Automata (NFA)


FINITE AUTOMATA
Finite Automata with OUTPUT
Finite Automata without OUTPUT
Moore Machine
Mealy Machine
DFA (Deterministic Finite Automata)
NFA (Non-deterministic Finite Automata)
ϵ-NFA (Epsilon NFA)

FINITE STATE AUTOMATA (…)

DFA: For every state q in S and every character α in Σ, one and only one transition of the following form occurs:

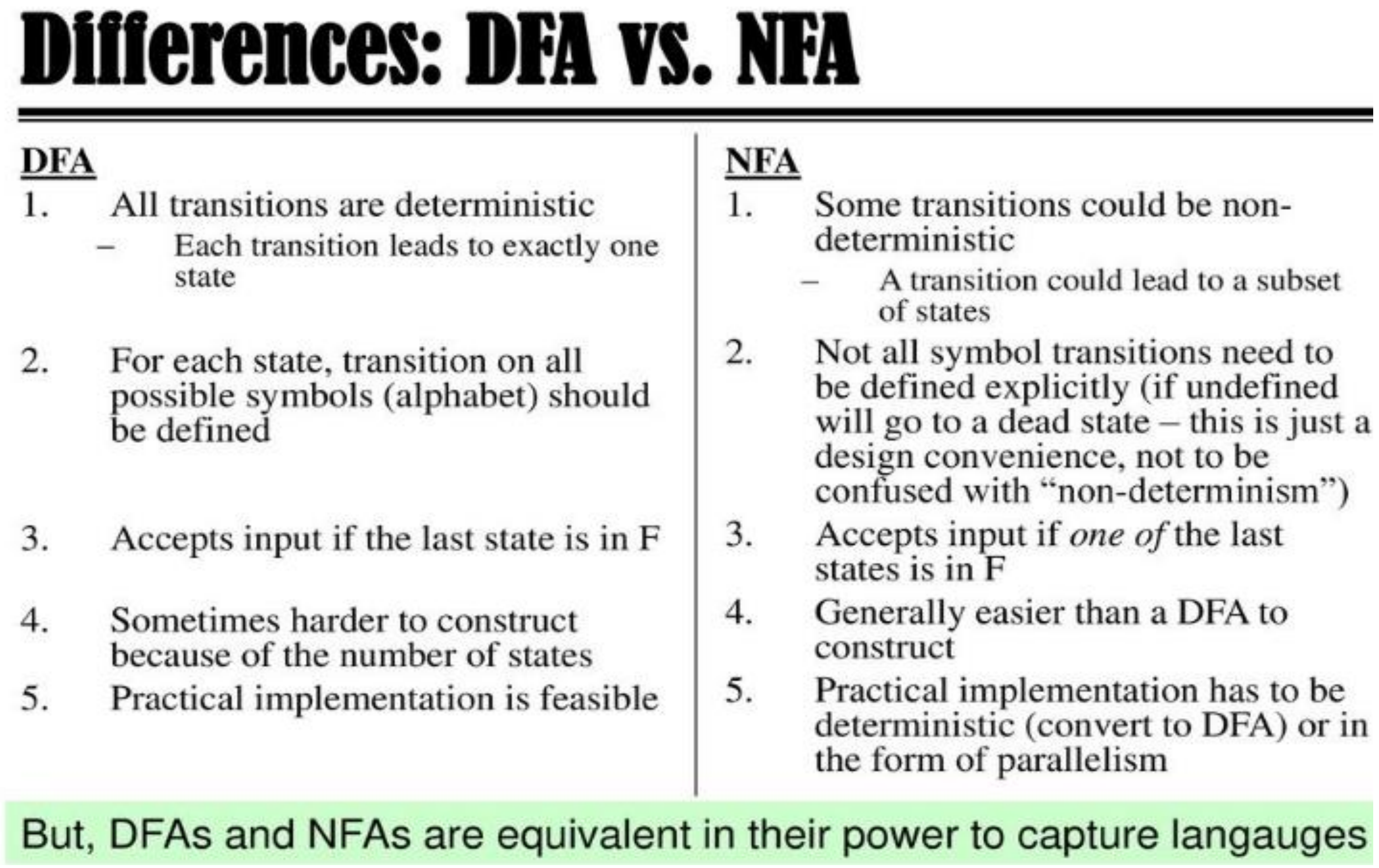
$$\begin{tikzcd} q \arrow[r, "\alpha"] & q' \end{tikzcd}$$

NFA: For every state q in S and every character α in $\Sigma \cup \{e\}$, one (or both) of the following will happen:

•No transition: $\begin{tikzcd} q \arrow[r, "\alpha"] & q' \end{tikzcd}$ occurs

•**One or more** transitions $\begin{tikzcd} q \arrow[r, "\alpha"] \arrow[dr, "\alpha"'] & q' \\ & p \end{tikzcd}$ occurs

FINITE STATE AUTOMATA (…)



Differences: DFA vs. NFA

DFA	NFA
1. All transitions are deterministic – Each transition leads to exactly one state	1. Some transitions could be non-deterministic – A transition could lead to a subset of states
2. For each state, transition on all possible symbols (alphabet) should be defined	2. Not all symbol transitions need to be defined explicitly (if undefined will go to a dead state – this is just a design convenience, not to be confused with "non-determinism")
3. Accepts input if the last state is in F	3. Accepts input if *one of* the last states is in F
4. Sometimes harder to construct because of the number of states	4. Generally easier than a DFA to construct
5. Practical implementation is feasible	5. Practical implementation has to be deterministic (convert to DFA) or in the form of parallelism

But, DFAs and NFAs are equivalent in their power to capture langauges

DFA – A Formal Definition(Rerun)

A ***finite automaton*** is a 5-tupple (Q,Σ,δ,q_0,F) where:

1. Q is a finite set called the ***states***.
2. Σ is a finite set called the ***alphabet***.
3. $\boxed{\delta : Q \times \Sigma \to Q \text{ is the } \textit{\textbf{transition function}}.}$
4. $q_0 \in Q$ is the ***start state***, and
5. $F \subseteq Q$ is the set of ***accepting states***.

NFA – A Formal Definition

A ***finite automaton*** is a 5-tupple (Q,Σ,δ,q_0,F) where:

1. Q is a finite set called the ***states***.
2. Σ is a finite set called the ***alphabet***.
3. $\boxed{\delta : Q \times \Sigma_\varepsilon^* \to P(Q) \text{ is the } \textit{\textbf{transition function}}.}$
4. $q_0 \in Q$ is the ***start state***, and
5. $F \subseteq Q$ is the set of ***accept states***.
   - $\Sigma_\varepsilon = \Sigma \cup \{\varepsilon\}$

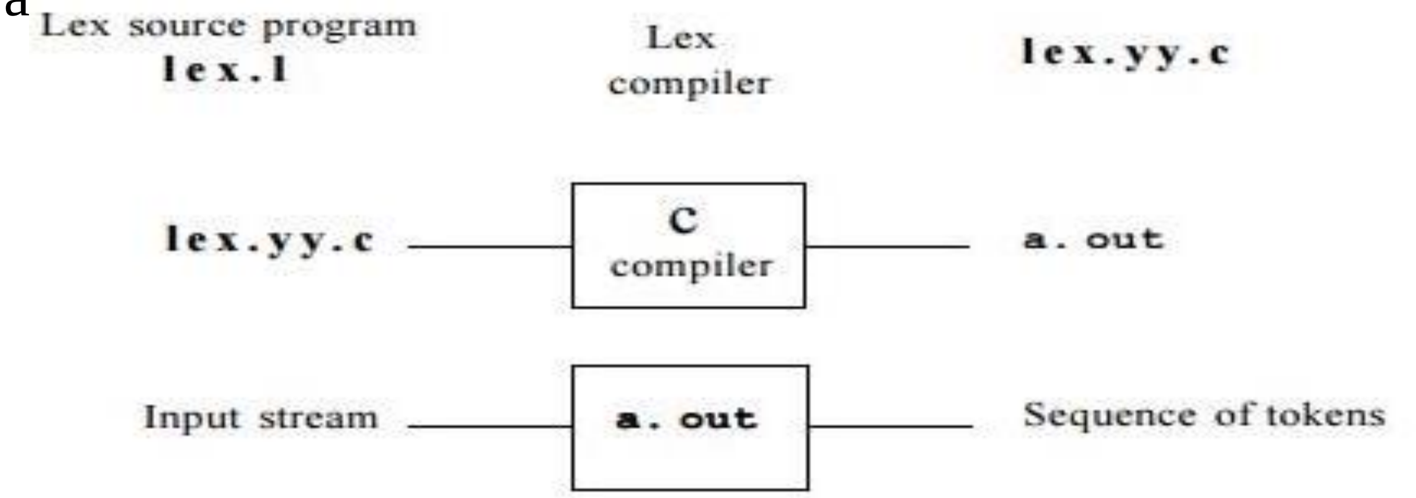
LEX: A tool for lexical analysis

- Lex is a tool in lexical analysis phase to recognize tokens using regular expression.
- Lex is a program that generates lexical analyzer. It is used with YACC parser generator.
- The lexical analyzer is a program that transforms an input stream into a sequence of tokens.
- It reads the input stream and produces the source code as output through implementing the lexical analyzer in the C program

LEX: A tool for lexical analysis

The function of Lex is as follows:

- Firstly lexical analyzer creates a program lex.1 in the Lex language. Then Lex compiler runs the lex.1 program and produces a C program lex.yy.c.
- Finally C compiler runs the lex.yy.c program and produces an object program a.out.
- a.out is lexical analyzer that transforms an input stream into a sequence of tokens.



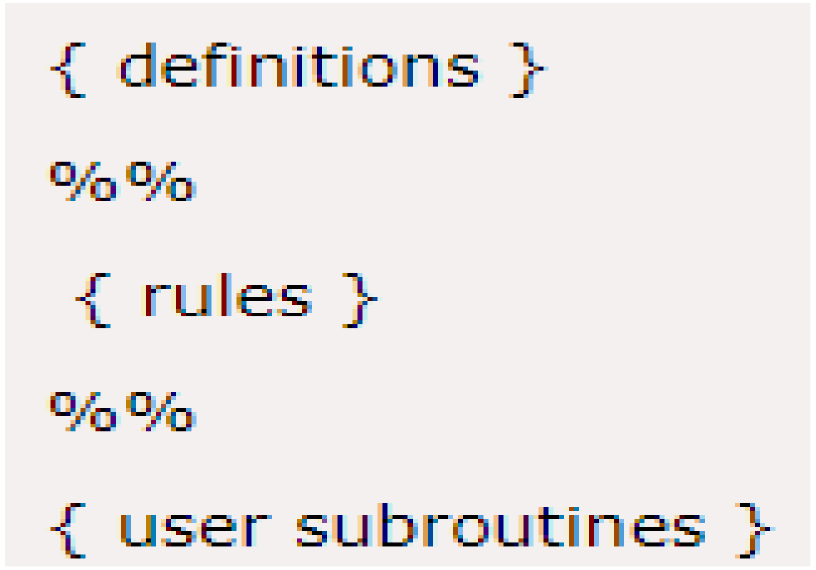
LEX: File format

Definitions include declarations of constant, variable and regular definitions.

Rules define the statement of form p1 {action1} p2 {action2}....pn {action}.

Where **pi** describes the regular expression and **action1** describes the actions what action the lexical analyzer should take when pattern pi matches a lexeme.

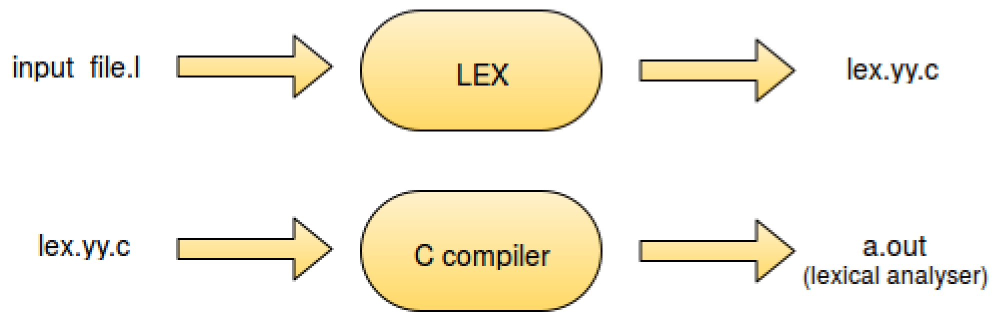
User subroutines are auxiliary procedures needed by the actions. The subroutine can be loaded with the lexical analyzer and compiled separately.



```
{ definitions }
%%
 { rules }
%%
{ user subroutines }
```

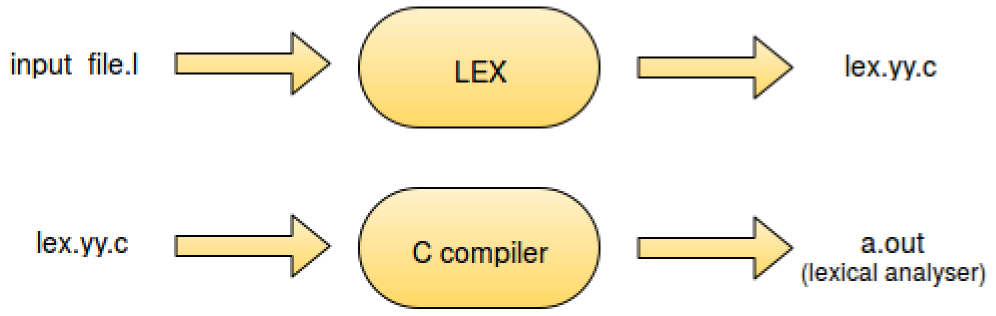
Files created by LEX

- lex.l is an a input file written in a language which describes the generation of lexical analyzer. The lex compiler transforms lex.l to a C program known as lex.yy.c.
- lex.yy.c is compiled by the C compiler to a file called a.out.
- The output of C compiler is the working lexical analyzer which takes stream of input characters and produces a stream of tokens.



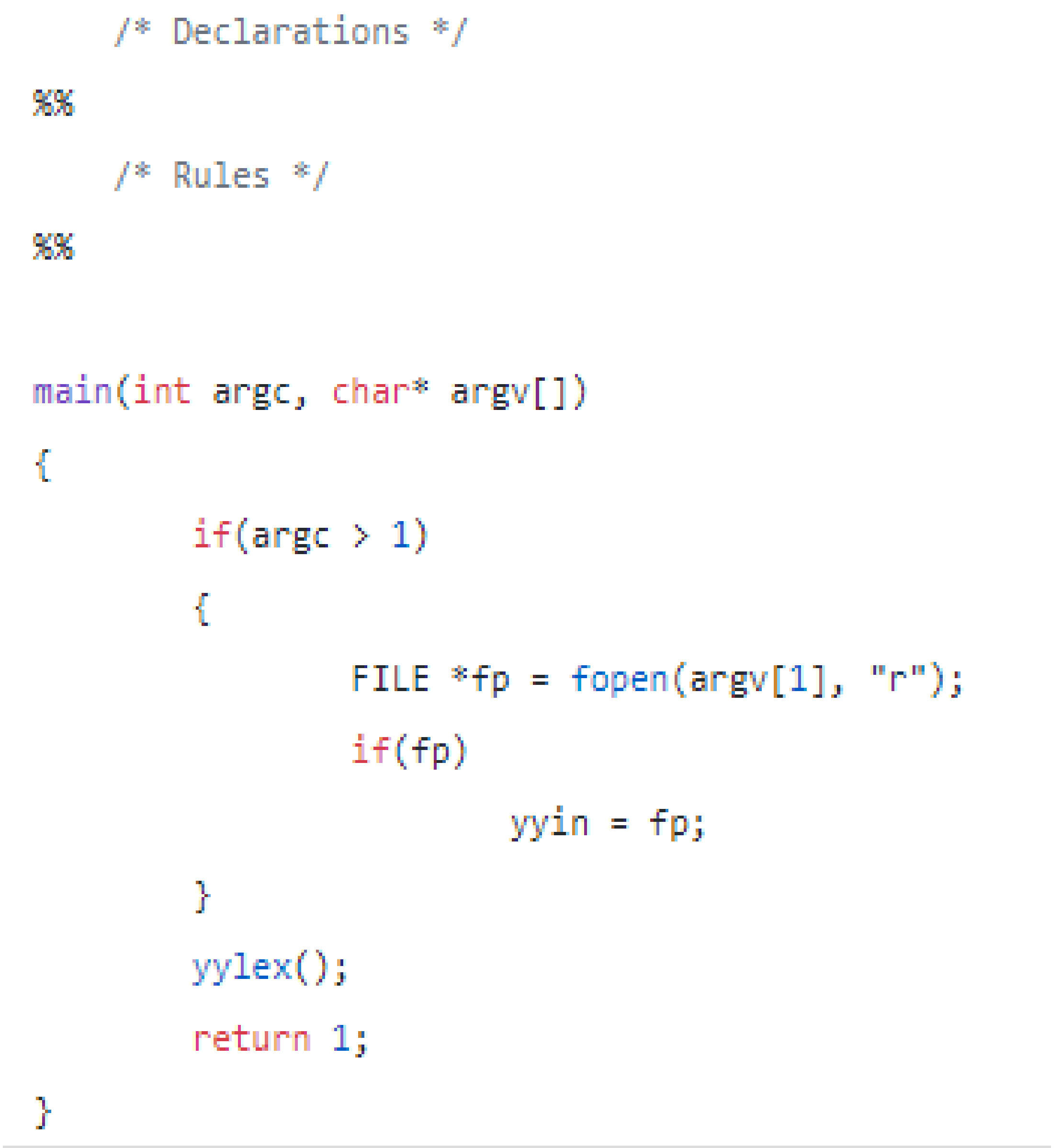
Files created by LEX

- lex.l is an a input file written in a language which describes the generation of lexical analyzer. The lex compiler transforms lex.l to a C program known as lex.yy.c.
- lex.yy.c is compiled by the C compiler to a file called a.out.
- The output of C compiler is the working lexical analyzer which takes stream of input characters and produces a stream of tokens.



LEX Variable

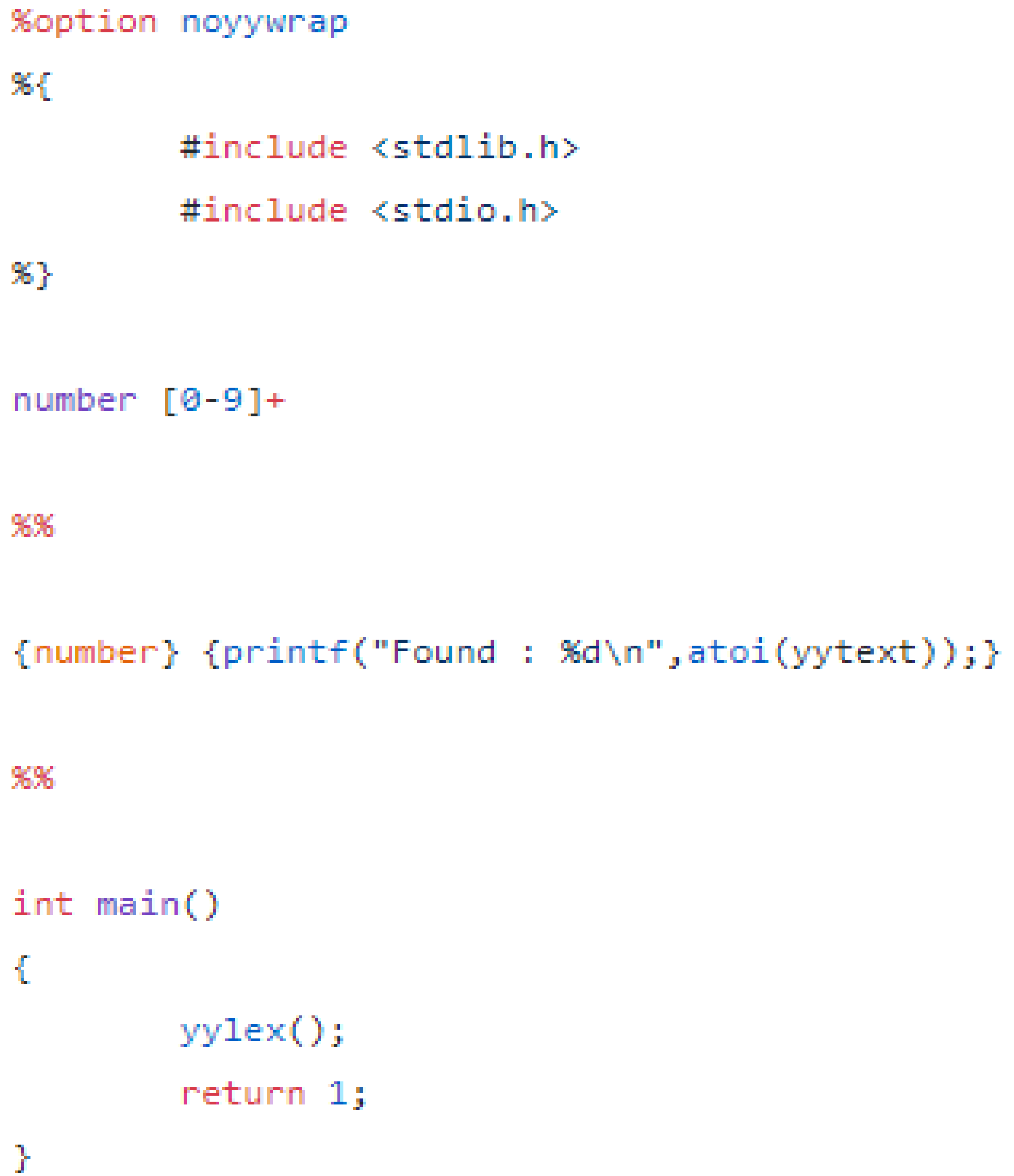
- **yyin** is a variable of the type FILE* and points to the input file. yyin is defined by LEX automatically. If the programmer assigns an input file to yyin in the auxiliary functions section, then yyin is set to point to that file. Otherwise LEX assigns yyin to stdin(console input).



```
    /* Declarations */
%%
    /* Rules */
%%

main(int argc, char* argv[])
{
        if(argc > 1)
        {
                FILE *fp = fopen(argv[1], "r");
                if(fp)
                        yyin = fp;
        }
        yylex();
        return 1;
}
```

LEX Variable

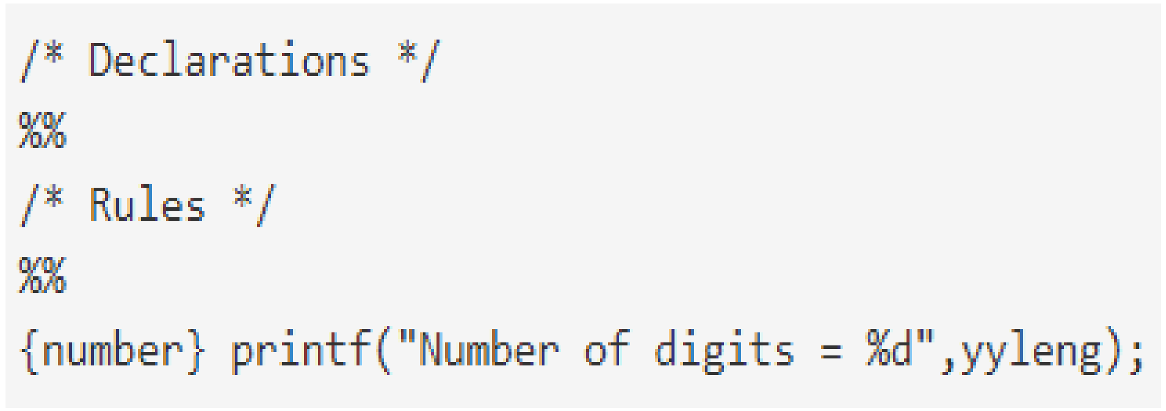
- yytext is of type char* and it contains the lexeme currently found. A lexeme is a sequence of characters in the input stream that matches some pattern in the Rules Section. Each invocation of the function yylex() results in yytext carrying a pointer to the lexeme found in the input stream by yylex(). The value of yytext will be overwritten after the next yylex() invocation.



```
%option noyywrap
%{
        #include <stdlib.h>
        #include <stdio.h>
%}

number [0-9]+

%%

{number} {printf("Found : %d\n",atoi(yytext));}

%%

int main()
{
        yylex();
        return 1;
}
```

LEX Variable

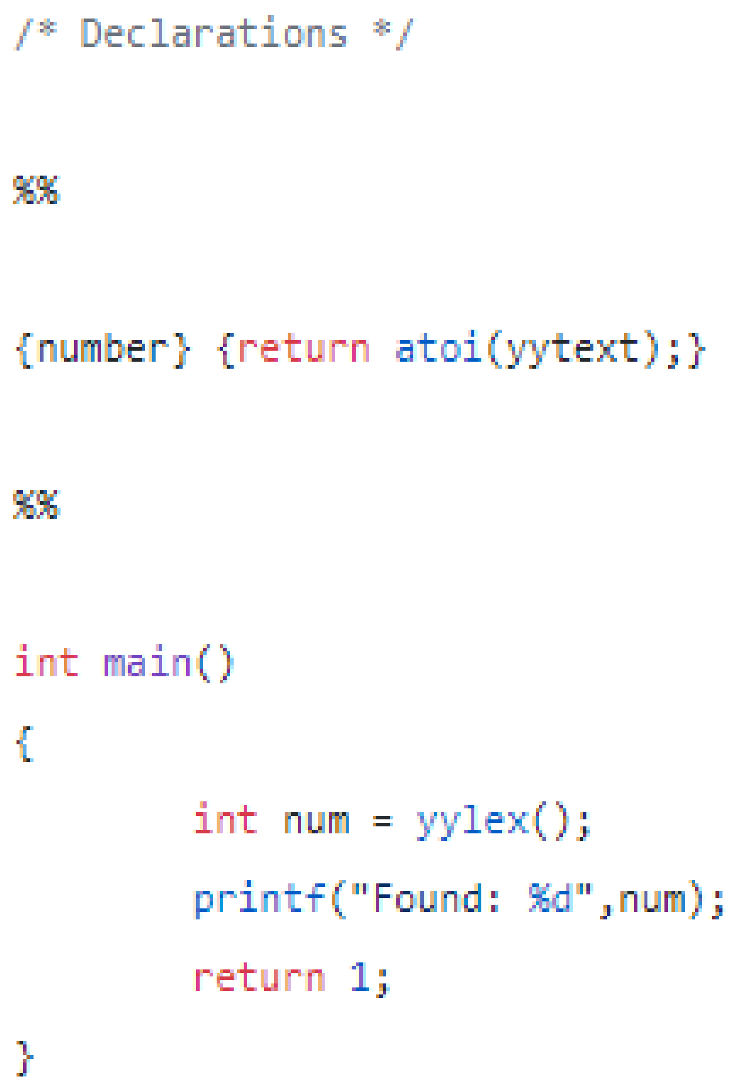
- yyleng is a variable of the type int and it stores the length of the lexeme pointed to by yytext.



```
/* Declarations */
%%
/* Rules */
%%
{number} printf("Number of digits = %d",yyleng);
```

LEX Functions

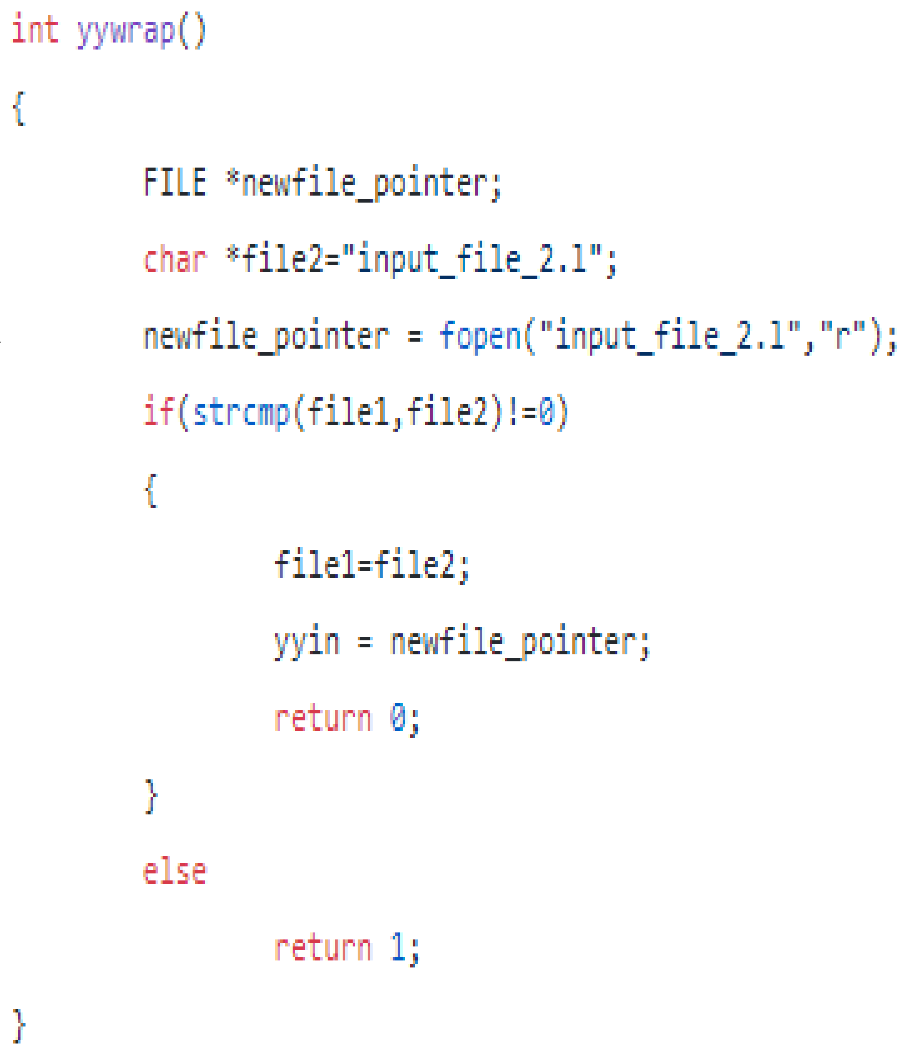
- ylex() is a function of return type int. LEX automatically defines yylex() in lex.yy.c but does not call it.
- The programmer must call yylex() in the Auxiliary functions section of the LEX program.
- LEX generates code for the definition of yylex() according to the rules specified in the Rules section.
- When yylex() is invoked, it reads the input as pointed to by yyin and scans through the input looking for a matching pattern.
- When the input or a part of the input matches one of the given patterns, yylex() executes the corresponding action associated with the pattern as specified in the Rules section.



```
/* Declarations */

%%

{number} {return atoi(yytext);}

%%

int main()
{
        int num = yylex();
        printf("Found: %d",num);
        return 1;
}
```

LEX Functions

- LEX declares the function yywrap() of return-type int in the file lex.yy.c .
- LEX does not provide any definition for yywrap(). yylex() makes a call to yywrap() when it encounters the end of input.
- If yywrap() returns zero (indicating false) yylex() assumes there is more input and it continues scanning from the location pointed to by yyin.
- If yywrap() returns a non-zero value (indicating true), yylex() terminates the scanning process and returns 0 (i.e. “wraps up”).
- If the programmer wishes to scan more than one input file using the generated lexical analyzer, it can be simply done by setting yyin to a new input file in yywrap() and return 0.

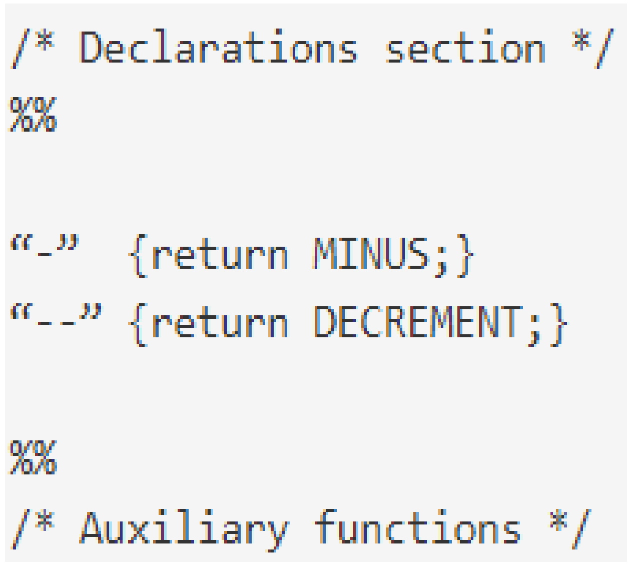


```
int yywrap()
{
        FILE *newfile_pointer;
        char *file2="input_file_2.l";
        newfile_pointer = fopen("input_file_2.l","r");
        if(strcmp(file1,file2)!=0)
        {
                file1=file2;
                yyin = newfile_pointer;
                return 0;
        }
        else
                return 1;
}
```

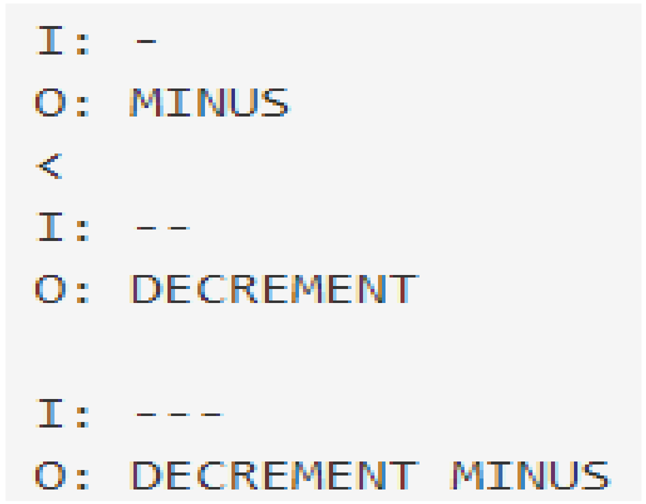
Disambiguation

- yylex() uses two important disambiguation rules in selecting the right action to execute in case there is more than one pattern that matches a string in the given input:

1. Choose the first match.
2. "Longest match" is preferred



```
/* Declarations section */
%%

“-”  {return MINUS;}
“--” {return DECREMENT;}

%%
/* Auxiliary functions */
```

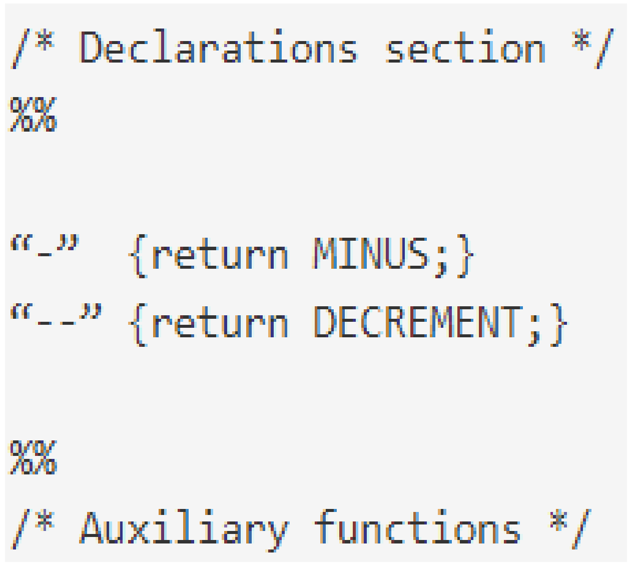


```
I: -
O: MINUS
<
I: --
O: DECREMENT

I: ---
O: DECREMENT MINUS
```

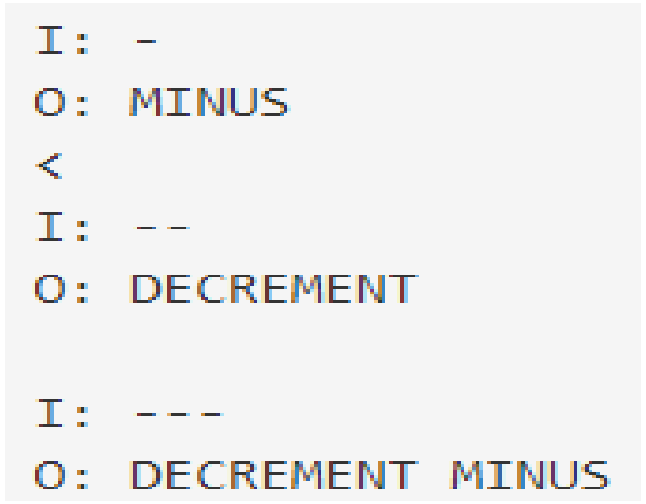
Disambiguation

- yylex() uses two important disambiguation rules in selecting the right action to execute in case there is more than one pattern that matches a string in the given input:

1. Choose the first match.
2. "Longest match" is preferred

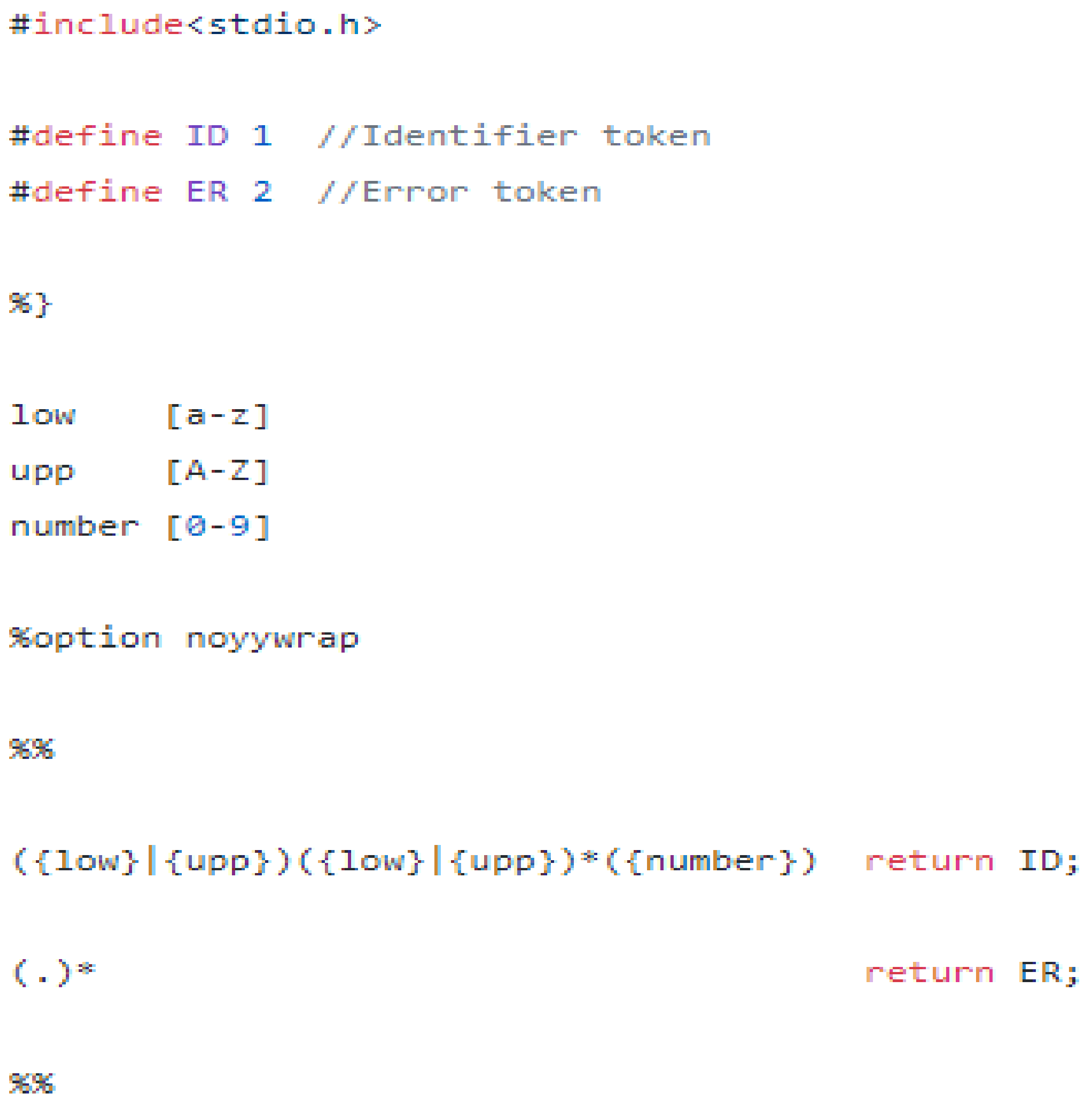


```
/* Declarations section */
%%

“-”  {return MINUS;}
“--” {return DECREMENT;}

%%
/* Auxiliary functions */
```

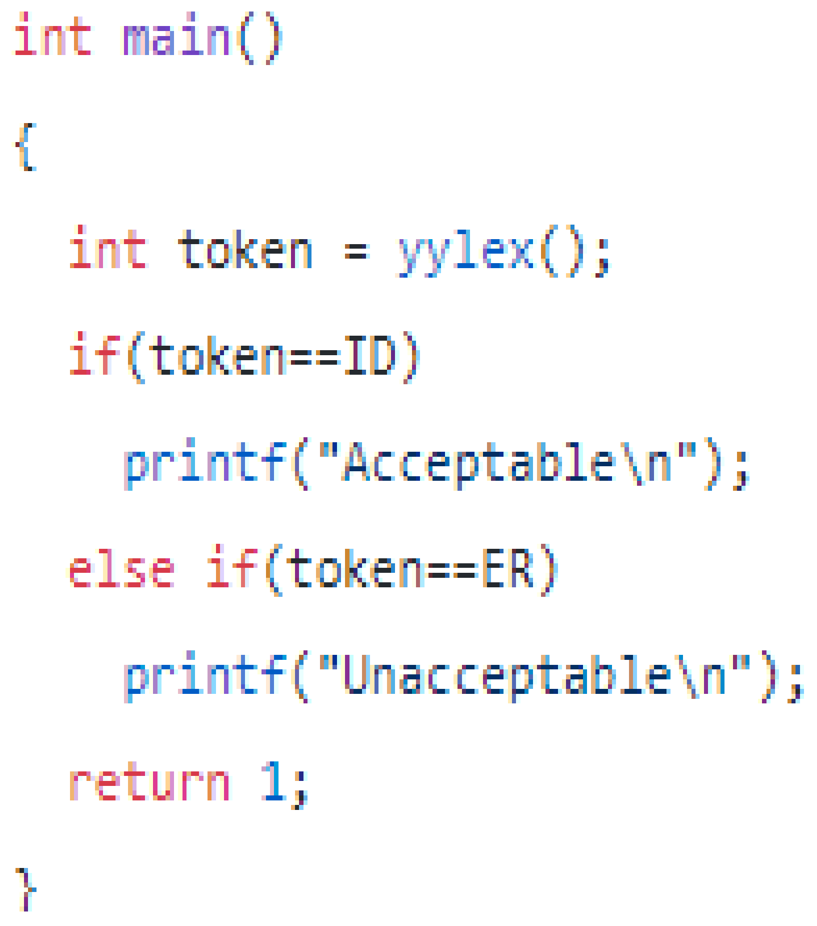


```
I:  -
O:  MINUS
<
I:  --
O:  DECREMENT

I:  ---
O:  DECREMENT MINUS
```

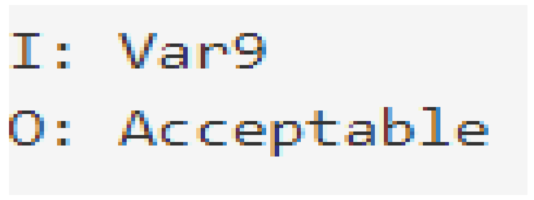
Example: Token identification



```
#include<stdio.h>

#define ID 1  //Identifier token
#define ER 2  //Error token

%}

low    [a-z]
upp    [A-Z]
number [0-9]

%option noyywrap

%%

({low}|{upp})({low}|{upp})*({number})  return ID;

(.)*                                    return ER;

%%
```



```
int main()
{
  int token = yylex();
  if(token==ID)
    printf("Acceptable\n");
  else if(token==ER)
    printf("Unacceptable\n");
  return 1;
}
```



```
I: Var9
O: Acceptable
```