ASSIGNMENT COMPILER DESIGN

Submitted To:Ms. Shweta

Submitted By:Rajanarora

Branch:CSE

Sec:B1

Roll No:071076

Que1 : - Differentiate between

i. Phase and pass of compiler

Ans:- a phase is a logically cohesive operation operation that takes as input one representation of the source program and produces as output another representation. A compiler takes input a source program and produces output an equivalent machine language instruction. It is not reasonable to consider entire process occurring in one step. So we partition compilation process into series of subprogram called phases. Listed below are various phases of compiler:-

1)lexical analysis(separate source language into groups that logically belong together called tokens)
2)Syntax analysis (group tokens into syntactic structures called expression.)
3)intermediate code generation(use structure produce to create simple instruction)
4)code optimization(improve intermediate code so that program runs faster)
5)code generation(produce final object code by deciding memory locations etc)

Portions of one or more phases are combined into a module called a pass. A pass reads the source program of the output of the previous pass, makes the transformations specified by its phases, and writes output into an intermediate file, which may then be read by a subsequent pass. The number of passes and the grouping of phases in dictated by particular language and machine. Various factors affecting number of passes in compiler are;

1) forward reference
2) storage limitation
3) optimization

ii) code optimization and code generation

Ans:- code optimization is a technique that a compiler employs to produce an improved object code of source code. It improves execution efficiency of object code. Programs that are expected to run many times optimization is necessary, it will improves its execution time. So this phase in which a faster or smaller object language program is created is called optimization phase. When applying any optimizing transformation, following criteria is applied,

1) Optimization should be such that meaning of program is not altered

2) Optimization should reduce time and space used by object code

3) Optimization should capture most improvements with less amount of effort.

Code generation phase converts the intermediate code into the sequence of the machine instruction. A simple minded code generator might map the statement

A=B+C into the machine code sequence

LOAD B

```
ADD C
STORE A
```

This target program may contain many redundant loads and stores and utilize machine recourses inefficiently. So to avoid this code generator might keep track of the run time contents of the registers. Code generator generates load and stores only when necessary. Factors affecting code generation are,

1) Target code structure (structure if target code controls the complexity of code generation)

2) Input (if operates, data types are not straight forward mapped then a good amount of effort May bee required)

iii) multipass and single pass compiler

Ans:-In computer programming, a **one-pass compiler** is a compiler that passes through the source code of each compilation unit only once. In other words, a one-pass compiler does not "look back" at code it previously processed. Another term sometimes used is **narrow compiler**, which emphasizes the limited scope a one-pass compiler is obliged to use. This is in contrast to a multi-pass compiler which traverses the source code and/or the abstract syntax tree several times, building one or more intermediate representations that can be arbitrarily refined.

While one-pass compilers may be faster than multi-pass compilers, they are unable to generate as efficient programs, due to the limited scope available. (Many optimizations require multiple passes over a program, subroutine, or basic block.) In addition, some programming languages simply cannot be compiled in a single pass, as a result of their design.

A **multi-pass compiler** is a type of compiler that processes the source code or abstract syntax tree of a program several times. This is in contrast to a one-pass compiler, which traverses the program only once. Each pass takes the result of the previous pass as the input, and creates an intermediate output. In this way, the (intermediate) code is improved pass by pass, until the final pass emits the final code.

Multi-pass compilers are sometimes called **wide compilers** referring to the greater scope of the passes: they can "see" the entire program being compiled, instead of just a small portion of it. The wider scope thus available to these compilers allows better code generation (e.g. smaller code size, faster code) compared to the output of one-pass compilers, at the cost of higher compiler time and memory consumption. In addition, some languages cannot be compiled in a single pass, as a result of their design.

iv) register allocation and assignment

Instructions involving only register operands are shorter and faster than those involving memory operands, Therefore efficient utilization of registers is particularly important I n generating good code. So the problem of deciding what name in program in should recede in registers called register allocation and kin which register each should decide is called register assignment. One approach to register allocation and assignment is to assign specific types of quantities in an object program to certain registers. For example a decision can be made to assign subroutine links to one group [p of registers, basic address to another, arithmetic computation to another, the runtime stack pointer to a fixed register.

This approach has advantage that it simplifies the design of a compiler. Its disadvantage I uses registers inefficiently, certain register may go unused and unnecessary loads and stores are generated.

v) Top down and bottom up parsing.

Top down parsing attempts to fin the leftmost derivations for an input string w, which s equivalent to constructing a parse tree for the input string w that start from root and creates a nodes of the parse tree in the predefined order.

The reason that to0p down parsing g seeks the leftmost derivation for an input string, is scanned by the Purser from left to right one symbol at a time and the left most derivation generate the leaves of the parse tree in the left tom right order which matches the input scan order. Bottom up parsing can be defined as attempt to repudiate input string w to start symbol of a grammar by tracing output the right derivation of w in reverse. This is equivalent to constructing a parse tree for the input string w by starting with leaves and proceeding toward root that is attempting the parse tree from bottom up. The reason that why bottom up parsing trace out rightmost derivation and not leftmost derivation id because the parser scans the input string w from left to right one symbol at a time. And to trace out right most derivation of a, string w the tokens of w must be made available in a left to right order.

Q1)Can a sentence have one right most derivation but two left most derevations.justify your ans with example.

Answer) Yes,it is possible to have a sentence which is having one right most derivation but two left most derivations.

Q3)-

i. What is the role of lexical analyzer in compilation process?

ii. Define regular expression and give steps to convert a regular expression into non-deterministic finite automata.

Answer)-

i. The Role of the Lexical Analyzer

As the first phase of a compiler, 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. The stream of tokens is sent to the parser for syntax analysis. It is common for the lexical analyzer to interact with the symbol table as well. When the lexical analyzer discovers a lexeme constituting an identifier, it needs to enter that lexeme into the symbol table. In some cases, information regarding the kind of identifier may be read from the symbol table by the lexical analyzer to assist it in determining the proper token it must pass to the parser.

These interactions are suggested in Fig. 3.1.1. Commonly, the interaction is implemented by having the parser call the lexical analyzer. The call, suggested by the *getNextToken*command, causes the lexical analyzer to read characters from its input until it can identify the next lexeme and produce for it the next
token, which it returns to the parser.

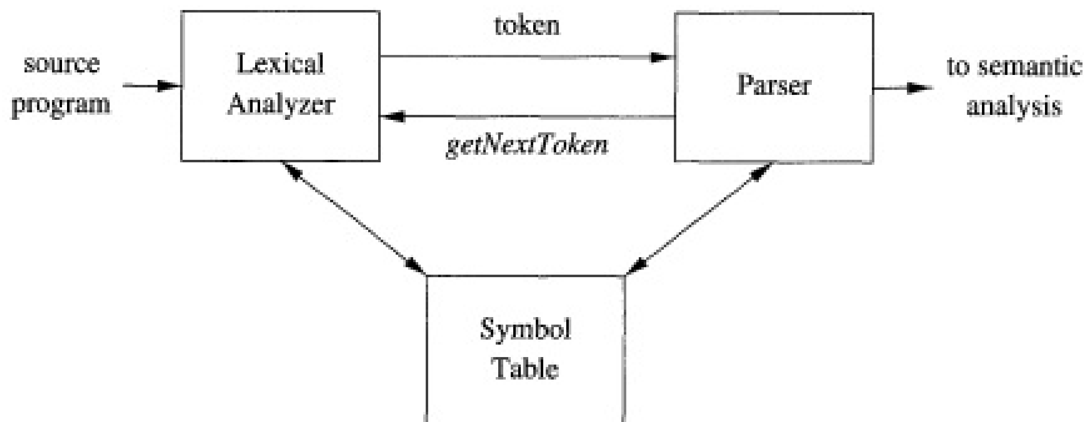

Figure 3.1.1: Interactions between the lexical analyzer and the parser

Since the lexical analyzer is the part of the compiler that reads the source text, it may perform certain other tasks besides identification of lexemes. These tasks include:

1. Stripping out comments and *whitespace* (blank, newline, tab, and perhaps other characters that are used to separate tokens in the input).

2. Correlating error messages generated by the compiler with the source program. For instance, the lexical analyzer may keep track of the number of newline characters seen, so it can associate a line number with each error message. In some compilers, the lexical analyzer makes a copy of the source program with the error messages inserted at the appropriate positions.

3. If the source program uses a macro-preprocessor, the expansion of macros may also be performed by the lexical analyzer.

Sometimes, lexical analyzers are divided into a cascade of two processes:

a) ***Scanning***consists of the simple processes that do not require tokenization of the input, such as deletion of comments and compaction of consecutive whitespace characters into one.
b) ***Lexical analysis***proper is the more complex portion, where the scanner produces the sequence of tokens as output.

Following three related but distinct terms are used when discussing lexical analysis:

a) *A **token***is a pair consisting of a token name and an optional attribute value. The token name is an abstract symbol representing a kind of lexical unit, e.g., a particular keyword, or a sequence of input characters denoting an identifier. The token names are the input symbols that the parser processes. In what follows, we shall generally write the name of a token in boldface. We will often refer to a token by its token name.

b) *A **pattern***is a description of the form that the lexemes of a token may take. In the case of a keyword as a token, the pattern is just the sequence of characters that form the keyword. For identifiers and some other tokens, the pattern is a more complex structure that is *matched* by many strings.

c) *A **lexeme***is a sequence of characters in the source program that matches the pattern for a token and is identified by the lexical analyzer as an instance of that token.

TOKEN	INFORMAL DESCRIPTION	SAMPLE LEXEMES
if	characters `i`, `f`	`if`
else	characters `e`, `l`, `s`, `e`	`else`
comparison	`<` or `>` or `<=` or `>=` or `==` or `!=`	`<=`, `!=`
id	letter followed by letters and digits	`pi`, `score`, `D2`
number	any numeric constant	`3.14159`, `0`, `6.02e23`
literal	anything but `"`, surrounded by `"`'s	`"core dumped"`

Figure 3.1.2: Examples of tokens

In many programming languages, the following classes cover most or all of the tokens:

1. One token for each keyword. The pattern for a keyword is the same as the keyword itself.
2. Tokens for the operators, either individually or in classes such as the token comparison mentioned in Fig. 3.1.2.
3. One token representing all identifiers.
4. One or more tokens representing constants, such as numbers and literal strings.

5. Tokens for each punctuation symbol, such as left and right parentheses, comma, and semicolon.

Lexical Errors

It is hard for a lexical analyzer to tell, without the aid of other components, that there is a source-code error. For instance, if the string f i is encountered for the first time in a C program in the context:

a lexical analyzer cannot tell whether fi is a misspelling of the keyword if or an undeclared function identifier. Since fi is a valid lexeme for the token id, the lexical analyzer must return the token id to the parser and let some other phase of the compiler - probably the parser in this case - handle an error due to transposition of the letters.

However, 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. We delete successive characters from the remaining input, until the lexical analyzer can find a well-formed token at the beginning of what input is left. This recovery technique may confuse the parser, but in an interactive computing environment it may be quite adequate.

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.

Transformations like these may be tried in an attempt to repair the input. The simplest such strategy is to see whether a prefix of the remaining input can be transformed into a valid lexeme by a single transformation. This strategy makes sense, since in practice most lexical errors involve a single character. A more general correction strategy is to find the smallest number of transformations needed to convert the source program into one that consists only of valid lexemes, but this approach is considered too expensive in practice to be worth the effort.

Q4)- Make left and right derivation using top down and bottom up strategy to derive a statement w=id +(id +id)* id using the following grammar:

E->E+E

E->E*E

E->(E)

E->id

Answer)-

Top Down Parsing:

It is an attempt to find the derivation for an input string.It is an attempt to construct a parse tree for input starting from root and creating the nodes of parse tree in preorder

Top Down parsing using left most derivation. We start with root node (E).

E	
E+E	(E□E+E)
Id+E	(E□id)
Id+E*E	(E□E*E)
Id+(E)*E	(E□(E))
Id+(E+E)*E	(E□E+E)
Id+(id+E)*E	(E□id)
Id+(id+id)*E	(E□id)
Id+(id+id)*id	(E□id)

Top Down parsing using right most derivation

E	
E*E	(E□E*E)
E*id	(E□id)
E+E*id	(E□E+E)
E+(E)*id	(E□(E))
E+(E+E)*id	(E□E+E)
E+(E+id)*id	(E□id)
E+(id+id)*id	(E□id)
Id+(id+id)*id	(E□id)

Bottom up Parsing:

It is an attempt to find the derivation for an input string.It is an attempt to construct a parse tree for input starting at leaves and ending towards root

Bottom up parsing using right most derivation.

Id+(id+id)*id (E→id)

Id+(id+id)*E (E→id)

Id+(id+E)*E (E→id)

Id+(E+E)*E (E→E+E)

Id+(E)*E (E→(E))

Id+E*E (E→E*E)

Id+E (E→id)

E+E (E→E+E)

E

Q5)-

i. What is meant by input buffering? How is it useful in design of lexical analyzer?

ii. Write a short note on LEX.

Answer)-

i. **Input buffering and its role in design of Lexical Analyzer**

A lexical analyzer may need to read ahead some characters before it can decide on the token to be returned to the parser. For example, a lexical analyzer for C or Java must read ahead after it sees the character >. If the next character is =, then > is part of the character sequence >=, the lexeme for the token for the "greater than or equal to" operator. Otherwise > itself forms the "greater than" operator, and the lexical analyzer has read one character too many. A general approach to reading ahead on the input, is to maintain an input buffer from which the lexical analyzer can read and push back characters. Input buffers can be justified on efficiency grounds alone, since fetching a block of characters is usually more efficient than fetching one character at a time. A pointer keeps track of the portion of the input that has been analyzed; pushing back a character is implemented by moving back the pointer.

One-character read-ahead usually suffices, so a simple solution is to use a variable, say peek, to hold the next input character. The lexical analyzer in this section reads ahead one character while it collects digits for numbers or characters for identifiers; e.g., it reads past 1 to distinguish between 1 and 10, and it reads past **t** to distinguish between **t** and **true.**

The lexical analyzer reads ahead only when it must. An operator like * can be identified without reading ahead. In such cases, peek is set to a blank, which will be skipped when the lexical analyzer is called to find the next token. The invariant assertion in this section is that when the lexical analyzer returns a token, variable peek either holds the character beyond the lexeme for the current token, or it holds a blank.
There are many situations where we need to look at least one additional character ahead. For instance, we cannot be sure we've seen the end of an identifier until we see a character that is not a letter or digit, and therefore is not part of the lexeme for **id**. In C, single-character operators like -, =, or < could also be the beginning of a two-character operator like ->, ==, or <=. Thus, we shall introduce a two-buffer scheme that handles large look-aheads safely. We then consider an improvement involving "sentinels" that saves time checking for the ends of buffers.

Buffer Pairs: Because of the amount of time taken to process characters and the large number of characters that must be processed during the compilation of a large source program, specialized buffering techniques have been developed to reduce the amount of overhead required to process a single input character. An important scheme involves two buffers that are alternately reloaded, as suggested in Fig. 5.1.1.
Each buffer is of the same size N, and N is usually the size of a disk block, e.g., 4096 bytes. Using one system read command we can read N characters inio a buffer, rather than using one system call per character. If fewer than N characters remain in the input file, then a special character, represented by **eof**, marks the end of the source file and is different from any possible character of the source program.

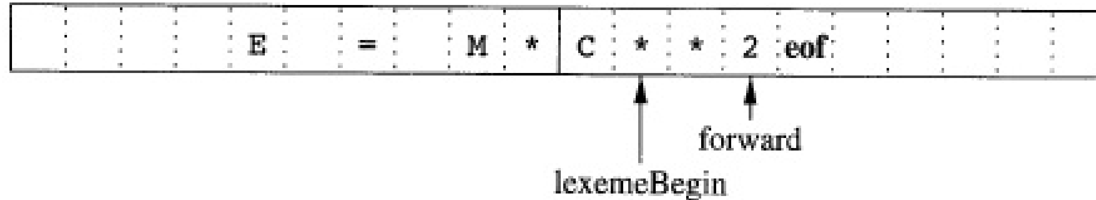

Figure 5.1.1: Using a pair of input buffers

Two pointers to the input are maintained:

1. Pointer lexemeBegin, marks the beginning of the current lexeme, whose extent we are attempting to determine.
2. Pointer forward scans ahead until a pattern match is found; the exact strategy whereby this determination is made will be covered in the balance of this chapter.

Once the next lexeme is determined, forward is set to the character at its right end. Then, after the lexeme is recorded as an attribute value of a token returned to the parser, 1exemeBegin is set to the character immediately after the lexeme just found. In Fig. 5.1.1, we see forward has passed the end of the next lexeme, ** (the Fortran exponentiation operator), and must be retracted one position to its left.
Advancing forward requires that we first test whether we have reached the end of one of the buffers, and if so, we must reload the other buffer from the input, and move forward to the beginning of the newly loaded buffer. As long as we never need to look so far ahead of the actual lexeme that the sum of the lexeme's length plus the distance we look ahead is greater than N, we shall never overwrite the lexeme in its buffer before determining it.

Sentinels: If we use the scheme of buffer pairs as described, we must check, each time we advance forward, that we have not moved off one of the buffers; if we do, then we must also reload the other buffer. Thus, for each character read, we make two tests: one for the end of the buffer, and one to determine what character is read (the latter may be a multiway branch). We can combine the buffer-end test with the test for the current character if we extend each buffer to hold a sentinel character at the end. The sentinel is a special character that cannot be part of the source program, and a natural choice is the character **eof**.
Figure 5.1.2 shows the same arrangement as Fig. 5.1.1, but with the sentinels added. Note that **eof** retains its use as a marker for the end of the entire input. Any **eof** that appears other than at the end of a buffer means that the input is at an end. Figure 5.1.3 summarizes the algorithm for advancing forward. Notice how the first test, which can be part of a multiway branch based on the character pointed to by forward, is the only test we make, except in the case where we actually are at the end of a buffer or the end of the input.

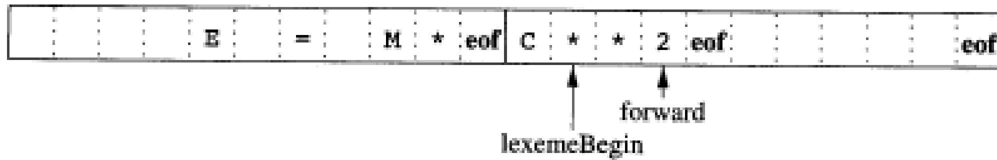

Figure 5.1.2: Sentinels at the end of each buffer

```
switch( *forward++ ) {
        caseeof:
                if(forward is at end of first buffer ) {
                        reload second buffer;
                        forward = beginning of second buffer;
                }
                else if (forward is at end of second buffer ) {
                        reload first buffer;
                        forward = beginning of first buffer;
                }
                else/* eofwithin a buffer marks the end of input */
                        terminate lexical analysis;
                break;
        Cases for the other characters
}
```

Figure 5.1.3**:** Lookahead code with sentinels

---

ii. **LEX**

Lex is a program generator designed for lexical processing of character input streams. It accepts a high-level, problem oriented specification for character string matching, and produces a program in a general purpose language which recognizes regular expressions.

The regular expressions are specified by the user in the source specifications given to Lex. The Lex written code recognizes these expressions in an input stream and partitions the input stream into strings matching the expressions. At the boundaries between strings program sections provided by the user are executed. The Lex source file associates the regular expressions and the program fragments. As each expression appears in the input to the program written by Lex, the corresponding fragment is executed.
The user supplies the additional code beyond expression matching needed to complete his tasks, possibly including code written by other generators. The program that recognizes the expressions is generated in the general purpose programming language employed for the user's program fragments. Thus, a high level expression language is provided to write the string expressions to be matched while the user's freedom to write actions is unimpaired. This avoids forcing the user who wishes to use a string manipulation language for input analysis to write processing programs in the same and often inappropriate string handling language.
Lex is not a complete language, but rather a generator representing a new language feature which can be added to different programming languages, called ``host languages.'' Just as general purpose languages can produce code to run on different computer hardware, Lex can write code in different host languages. The host language is used for the output code generated by Lex and also for the program fragments added by the user. Compatible run-time libraries for the different host languages are also provided. This makes Lex adaptable to different environments and different users. Each application may be directed to the combination of hardware and host language appropriate to the task, the user's background, and the properties of local implementations. At present, the only supported host language is C, although Fortran (in the form of Ratfor has been available in the past. Lex itself exists on UNIX, GCOS, and OS/370; but the code generated by Lex may be taken anywhere the appropriate compilers exist.
Lex turns the user's expressions and actions (called source in this memo) into the host general-purpose language; the generated program is named yylex. The yylex program will recognize expressions in a stream (called input in this memo) and perform the specified actions for each expression as it is detected. See Fig 5.2.1.

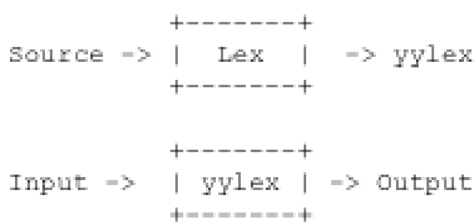

Fig 5.2.1 : An overview of Lex

For a trivial example, consider a program to delete from the input all blanks or tabs at the ends of lines.

```
%%
[ \t]+$ ;
```

is all that is required. The program contains a %% delimiter to mark the beginning of the rules, and one rule. This rule contains a regular expression which matches one or more instances of the characters blank or tab (written \t for visibility, in accordance with the C language convention) just prior to the end of a line. The brackets indicate the character

class made of blank and tab; the + indicates ``one or more ...''; and the $ indicates ``end of line,'' as in QED. No action is specified, so the program generated by Lex (yylex) will ignore these characters. Everything else will be copied. To change any remaining string of blanks or tabs to a single blank, add another rule:

```
%%
[ \t]+$ ;
[ \t]+ printf(" ");
```

The finite automaton generated for this source will scan for both rules at once, observing at the termination of the string of blanks or tabs whether or not there is a newline character, and executing the desired rule action. The first rule matches all strings of blanks or tabs at the end of lines, and the second rule all remaining strings of blanks or tabs.

Lex can be used alone for simple transformations, or for analysis and statistics gathering on a lexical level. Lex can also be used with a parser generator to perform the lexical analysis phase; it is particularly easy to interface Lex and Yacc. Lex programs recognize only regular expressions; Yacc writes parsers that accept a large class of context free grammars, but require a lower level analyzer to recognize input tokens. Thus, a combination of Lex and Yacc is often appropriate. When used as a preprocessor for a later parser generator, Lex is used to partition the input stream, and the parser generator assigns structure to the resulting pieces. The flow of control in such a case (which might be the first half of a compiler, for example) is shown in Figure 5.2.2. Additional programs, written by other generators or by hand, can be added easily to programs written by Lex.

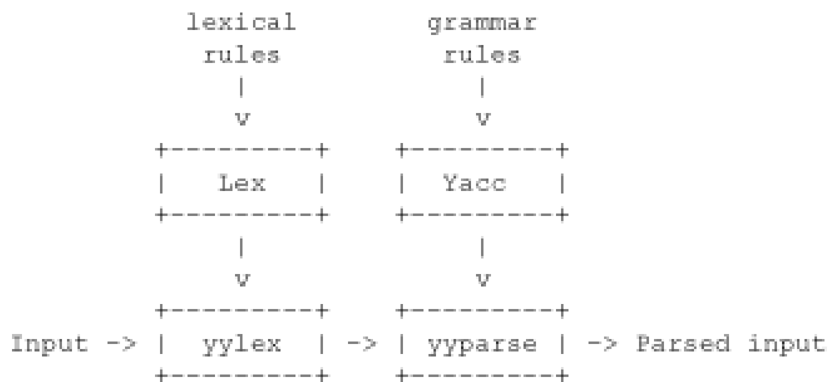

Fig 5.2.2: Lex with Yacc

Yacc users will realize that the name yylex is what Yacc expects its lexical analyzer to be named, so that the use of this name by Lex simplifies interfacing.
Lex generates a deterministic finite automaton from the regular expressions in the source. The automaton is interpreted, rather than compiled, in order to save space. The result is still a fast analyzer. In particular, the time taken by a Lex program to recognize and partition an input stream is proportional to the length of the input. The number of Lex rules or the complexity of the rules is not important in determining speed, unless rules which include forward context require a significant amount of rescanning. What does increase with the number and complexity of rules is the size of the finite automaton, and therefore the size of the program generated by Lex.

In the program written by Lex, the user's fragments (representing the actions to be performed as each regular expression is found) are gathered as cases of a switch. The automaton interpreter directs the control flow. Opportunity is provided for the user to insert either declarations or additional statements in the routine containing the actions, or to add subroutines outside this action routine.
Lex is not limited to source which can be interpreted on the basis of one character lookahead. For example, if there are two rules, one looking for ab and another for abcdefg, and the input stream is abcdefh, Lex will recognize ab and leave the input pointer just before cd. . . Such backup is more costly than the processing of simpler languages.

The general format of Lex source is:

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

where the definitions and the user subroutines are often omitted. The second %% is optional, but the first is required to mark the beginning of the rules. The absolute minimum Lex program is thus %% (no definitions, no rules) which translates into a program which copies the input to the output unchanged.

In the outline of Lex programs shown above, the rules represent the user's control decisions; they are a table, in which the left column contains regular expressions and the right column contains actions, program fragments to be executed when the expressions are recognized. Thus an individual rule might appear

```
integerprintf("found keyword INT");
```

to look for the string integer in the input stream and print the message ``found keyword INT" whenever it appears. In this example the host procedural language is C and the C library function printf is used to print the string. The end of the expression is indicated by the first blank or tab character. If the action is merely a single C expression, it can just be given on the right side of the line; if it is compound, or takes more than a line, it should be enclosed in braces. As a slightly more useful example, suppose it is desired to change a number of words from British to American spelling. Lex rules such as

```
colourprintf("color");
mechaniseprintf("mechanize");
petrolprintf("gas");
```

would be a start. These rules are not quite enough, since the word petroleum would become gaseum.