OVERVIEW OF COMPILATION:

Compilation is a process that translates a program in one language (the source language) into an equivalent program in another language (the object or target language).

A compiler is a computer program (or set of programs) that transforms source code written in a programming language (the source language) into another computer language (the target language, often having a binary form known as object code).

The most common reason for wanting to transform source code is to create an executable program.

The name "compiler" is primarily used for programs that translate source code from a high-level programming language to a lower level language (e.g., assembly language or machine code).

If the compiled program can run on a computer whose CPU or operating system is different from the one on which the compiler runs; the compiler is known as a cross-compiler.

A program that translates from a low level language to a higher level one is a decompiler.

JKDirectory!

A program that translates between high-level languages is usually called a source-to-source compiler or transpiler.

A language rewriter is usually a program that translates the form of expressions without change of language. More generally, compilers are sometimes called translators.

Compilers enabled the development of programs that are machine-independent. Before the development of FORTRAN, the first higher-level language, in the 1950s, machine-dependent assembly language was widely used.

While assembly language produces more abstraction than machine code on the same architecture, just as with machine code, it has to be modified or rewritten if the program is to be executed on different computer hardware architecture.

With the advent of high-level programming languages that followed FORTRAN, such as COBOL, C, and BASIC, programmers could write machine-independent source programs. A compiler translates the high-level source programs into target programs in machine languages for the specific hardware. Once the target program is generated, the user can execute the program.

A compiler is likely to perform many or all of the following operations: lexical analysis, preprocessing, parsing, semantic analysis (Syntax-directed translation), code generation, and code optimization.

STRUCTURE OF A COMPILER:

Compilers bridge source programs in high-level languages with the underlying hardware. A compiler verifies code syntax, generates efficient object code, performs run-time organization, and formats the output according to assembler and linker conventions. A compiler consists of:

1) ***The Front End:***

    a. Verifies syntax and semantics, and generates an intermediate representation or IR of the source code for processing by the middle-end.

    b. Performs type checking by collecting type information.

    c. Generates errors and warning, if any, in a useful way.

    d. Aspects of the front end include lexical analysis, syntax analysis, and semantic analysis.

JK Directory!

2) ***The Middle End:***

    Performs optimizations, including removal of useless or unreachable code, discovery and propagation of constant values, relocation of computation to a less frequently executed place (e.g., out of a loop), or specialization of computation based on the context; Generates another IR for the backend.

3) ***The Back End:***

    a) Generates the assembly code, performing register allocation in process.

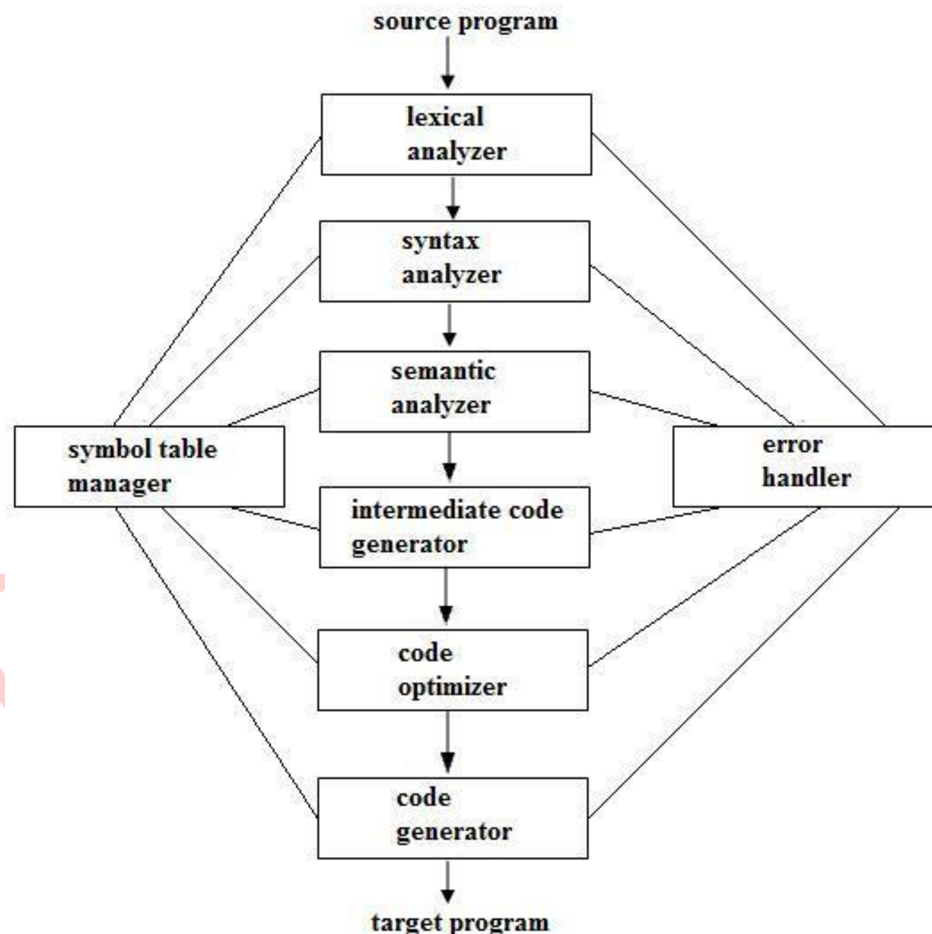
    b) Optimizes target code utilization of the hardware by figuring out how to keep parallel execution units busy, filling delay slots.

PHASES OF COMPILER:

The process of compilation is split up into six phases, each of which interacts with a symbol table manager and an error handler. This is called the analysis/synthesis model of compilation. There are many variants on this model, but the essential elements are the same.

1) Lexical Analyzer
2) Syntax Analyzer

3) Semantic Analyzer
4) Intermediate Code Generation
5) Code Optimization
6) Code Generation



LEXICAL ANALYSIS:

A lexical analyzer or scanner is a program that groups sequences of characters into lexemes, and outputs (to the syntax analyzer) a sequence of tokens. Here:

a) *Tokens* are symbolic names for the entities that make up the text of the program e.g. *if* for the keyword **if**, and *id* for any identifier. These make up the output of the lexical analyzer.

b) A *pattern* is a rule that specifies when a sequence of characters from the input constitutes a token; e.g. the sequence **i, f** for the token *if*, and *any sequence of alphanumeric starting with a letter* for the token *id*.

c) A *lexeme* is a sequence of characters from the input that match a pattern (and hence constitute an instance of a token); for example **if** matches the pattern for *if*, and **foo123bar** matches the pattern for *id*.

For example, the following code might result in the table given below:

program foo(input,output);var x:integer;begin
readln(x);writeln('value read =',x) end.

Lexeme	*Token*	*Pattern*
program	*program*	p, r, o, g, r, a, m
		newlines, spaces, tabs
foo	*id* (foo)	*letter followed by seq. of alphanumerics*
(	*leftpar*	*a left parenthesis*
input	*input*	i, n, p, u, t
,	*comma*	*a comma*
output	*output*	o, u, t, p, u, t
)	*rightpar*	*a right parenthesis*
;	*semicolon*	*a semi-colon*
var	*var*	v, a, r
x	*id* (x)	*letter followed by seq. of alphanumerics*
:	*colon*	*a colon*
integer	*integer*	i, n, t, e, g, e, r
;	*semicolon*	*a semi-colon*
begin	*begin*	b, e, g, i, n
		newlines, spaces, tabs
readln	*readln*	r, e, a, d, l, n
(	*leftpar*	*a left parenthesis*
x	*id* (x)	*letter followed by seq. of alphanumerics*
)	*rightpar*	*a right parenthesis*
;	*semicolon*	*a semi-colon*
writeln	*writeln*	w, r, i, t, e, l, n
(	*leftpar*	*a left parenthesis*
'value read ='	*literal* ('value read =')	*seq. of chars enclosed in quotes*
,	*comma*	*a comma*
x	*id* (x)	*letter followed by seq. of alphanumerics*
)	*rightpar*	*a right parenthesis*
		newlines, spaces, tabs
end	*end*	e, n, d
.	*fullstop*	*a fullstop*

It is the sequence of tokens in the middle column that are passed as output to the syntax analyzer.

This token sequence represents *almost* all the important information from the input program required by the syntax analyzer.

Whitespace (newlines, spaces and tabs), although often important in separating lexemes, is usually not returned as a token.

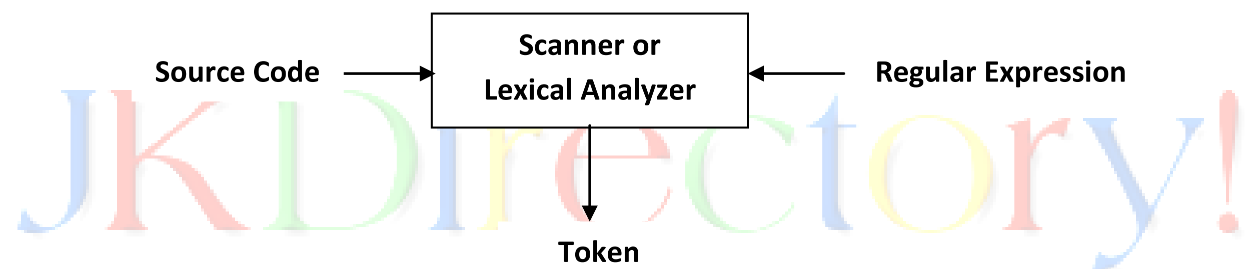
Also, when outputting an *id* or *literal* token, the lexical analyzer must also return the value of the matched lexeme (shown in parentheses) or else this information would be lost.

For the lexical analysis, specifications are traditionally written using regular expressions which is an algebraic notation for describing sets of strings. The generated lexical analyzers or lexers are in a class of extremely simple programs called finite automata.

Lexical analyzer also called as Lexer is a state machine which does lexing. Like an iterator Lexer typically has one more method ***next()*** that returns next token in sequence.

Inside the Lexer there is a state machine that processes the characters of input stream in order to generate the token sequence. Lexer uses another method called ***hasNext()*** to take decision about whether the end of the token sequence is reached or not.

In the token sequence, the possible tokens are reserved words (keywords), identifiers, literals, comments, operators, separators and white spaces.



Construction of scanner or lexical analyzer:

1) Define all tokens as regular expressions
2) Construct a finite automata for each token
3) Combine all these automata to a new automaton (in general NFA).
4) Translate it to a corresponding DFA.
5) Minimize the DFA.
6) Implement the DFA.

***SYMBOL TABLE MANAGEMENT*:**

A symbol table is a data structure containing all the identifiers (i.e. names of variables, procedures etc.) of a source program together with all the attributes of each identifier.

For variables, typical attributes include:

- Its type,

- How much memory it occupies,
- Its scope.

For procedures and functions, typical attributes include:

- The number and type of each argument (if any),
- The method of passing each argument, and
- The type of value returned (if any).

The purpose of the symbol table is to provide quick and uniform access to identifier attributes throughout the compilation process. Information is usually put into the symbol table during the lexical analysis and/or syntax analysis phases.

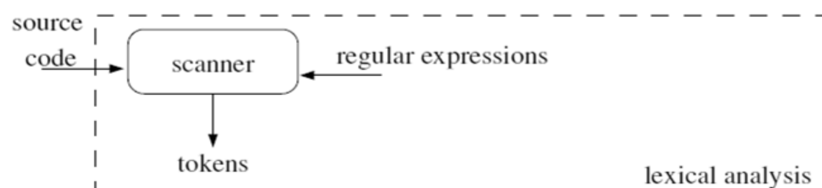
Typically the tokens are:

1) ***Reserved words (keywords)*** –example lexemes are *if then else begin*
2) ***Identifiers*** –example lexemes are *i alpha k10*
3) ***Literals*** –example lexemes are *123 3.1416 'A' "text"*
4) ***Operators*** –example lexemes are *+ ++ !=*
5) ***Separators*** –example lexemes are *; , (*

JKDirectory!

Non tokens are:

1) ***Whitespace*** –example lexemes are *' ' tab newline formfeed*
2) ***Comments*** –example lexemes are */* comment */ //eol comment*



REGULAR GRAMMAR AND REGULAR EXPRESSION FOR COMMON PROGRAMMING LANGUAGE FEATURES

REGULAR GRAMMAR:

The syntactic structure of a language is defined using grammars. Grammars specify a set of strings over an alphabet. A grammar defines a set of sentences which is a sequence of symbols likes tokens (also called as terminals). A grammar is a set of productions where each production defines a non-terminal which is a variable that stands for a set of sentences.

Efficient recognizers (automata) can be constructed to determine whether a string is in the language. By convention, non-terminals are capitalized, and terminals are lowercase. Production has form:

non-terminal ::= expression of terminals and non-terminals and operators

Regular grammar has a special property: by substituting every non terminal except the root one with its right hand side you can reduce it down to a single production for the root with only terminals and operators on right hand side. This compiled form of regular grammar is called a regular expression.

REGULAR EXPRESSION:

The notation of regular expression is a mathematical formalism ideal for expressing "patterns" and thus ideal for expressing the "lexical structure of programming languages".

Regular expression represents the patterns of strings of symbols. A regular expression ***r*** matches a set of strings over an alphabet (∑). This set is denoted as L(***r***) and is called the language determined or generated by ***r***.

Suppose we have the alphabet ∑ = {0,1} then the example language is:

Set of possible combinations of zeros and ones i.e. L_0 = {0, 1, 00, 01, 10, 11, . . .}

JkDirectory!

Suppose we have the alphabet ∑ as the set of UNICODE characters then the example languages are:

1) All possible java keywords i.e. {"class", "import", "public"}
2) All possible lexemes corresponding to java tokens.
3) All possible lexemes corresponding to java white spaces.
4) All binary numbers etc…

With respect to class of regular languages and apply these concepts to practical problem of lexical analysis we define the notion of regular expression and show how regular expressions determine regular languages.

DFA is the class of algorithms that solve decision problems for regular languages. With regular expressions and DFA we can specify and implement lexical analyzer.

REGULAR EXPRESSION	Read	Is called
a	a	Symbol
M|N	M or N	Alternative
MN	M followed by N	Concatenation
Ɛ	The empty string	Epsilon

REGULAR EXPRESSION	Read	Means
M*	Zero or more M	Repetition
M^+	At least one. . .	MM*
M?	Optional	Ɛ/M
[a-zA-Z]	One of . . .	a|b| . . . |z|A|B| . . . |Z
~[0-9]	Not . . .	One character but not anyone of those listed
"a+b"	The string	a \+ b

IMPLEMENTATION OF SCANNERS:

Let us construct the automata for IF and ID as:

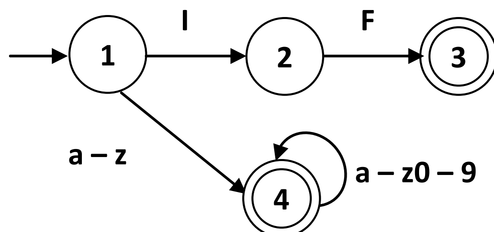
Automata for the whole language:

Combine the automata for individual tokens:

1) Merge the start states
2) Re-enumerate the states so they can get unique numbers
3) Mark each final state with the token i.e. matched.

JKDirectory!

After combining the automata's for IF and ID we get the automata as:



PASS AND PHASES OF TRANSLATION:

A ***pass*** is the group of several phases of compiler to perform analysis or synthesis of source program. Passes refer to the number of times the compiler has to traverse through the entire program. There are several types of passes as given below:

1) **Single pass:** it is also called as one-pass compiler, which is a compiler that passes through the parts of each compilation unit only once, immediately translating each part into its final machine code.
2) **Multi-pass:** It converts the program into one or more intermediate representations steps in between source code and machine code, and which reprocesses the entire compilation unit in each sequential pass.

One-pass compilers are smaller and faster than multi-pass compilers. One-pass compilers are unable to generate as efficient programs, due to the limited scope of available information. Many effective compiler optimizations require multiple passes over a basic block, loop, subroutine, or entire module. Some require passes over an entire program. Some programming languages simply cannot be compiled in a single pass, as a result of their design.

Phases of a compiler are the sub-tasks that must be performed to complete the compilation process. The discussion of phases deals with the logical organization of the compiler.

In an implementation, activities from several phases may be grouped together in to a pass that reads an input file, and writes the output file.

For example the front-end phases of lexical analysis, syntax analysis, semantic analysis and intermediate code generation might be grouped together in to one pass.

Code optimization might be an optional pass. Then there could be a back-end pass consisting of code generation for a particular target machine.

Since writing a compiler is a non-trivial task, it is good idea to structure the work. A typical way of doing this is to split the compilation in to several phases with well-defined interfaces.

JKDirectory!

Conceptually these phases operate in sequence, each phase taking the output from the previous phase as its input.

A phase is a logically interrelated operation that takes source program in one representation and produces output in another representation. There are two phases of compilation.

a) Analysis (Machine Independent/Language Dependent)
b) Synthesis(Machine Dependent/Language independent)

Compilation process is partitioned into no-of-sub processes called ‘phases’.

INTERPRETATION:

An interpreter is another way of implementing a programming language. Interpretation shares many aspects with compiling. Lexing, parsing and type-checking are in an interpreter done just as in a compiler.

But instead of generating code from the syntax tree, the syntax tree is processed directly to evaluate expressions and execute statements and so on.

An interpreter may need to process the same piece of syntax tree (e.g. body of the loop) many times and hence interpretation is typically slower than executing a compiled program. But writing an interpreter is often simpler than writing a compiler and the interpreter is easier to move to a different machine.

Compilation and interpretation may be combined to implement a programming language. The compiler may produce intermediate-level code which is interpreted than compiled to machine code.

An interpreter generally uses one of the following strategies for program execution:

1) Parse the source code and perform its behavior directly
2) Translate source code into some efficient intermediate representation and immediately execute this
3) Explicitly execute stored precompiled code made by a compiler which is part of the interpreter system

The process of interpretation can be carried out in Lexical analysis, Syntax analysis, Semantic analysis, and Direct Execution.

BOOTSTRAPPING:

Bootstrapping is the process of writing a compiler (or assembler) in the target programming language which it is intended to compile. Applying this technique leads to a self-hosting compiler.

Bootstrapping a compiler has the following advantages:

1. Compiler developers only need to know the language being compiled.
2. Compiler development can be done in the higher level language being compiled.
3. Improvements to the compiler's back-end improve not only general purpose programs but also the compiler itself.
4. It is a comprehensive consistency check as it should be able to reproduce its own object code.

DATA STRUCTURES IN COMPILATION:

The compilation process can be thought of as taking some input data structure and transforming it to produce an output data structure which in some sense is equivalent to but

more desirable than the input form. The input data is often called the source program and the output the object code.

One of the simplest forms the compilation process can take is to have the input data structure coming from a peripheral device (such as a card reader), and the compiler producing as output a machine code program in the store of the computer ready for execution. In most compilers, before the output form is arrived at, the input data structure may well be transformed into several internal data structures.

The compiler itself may require storing additional information or alternatively be driven by information stored internally. Finally, the data structures defined in the source program in the specific programming language must be mapped into some storage structure on the background computer that is to execute the program. The problems which need to be resolved when designing a compiler are ones such as finding the most transparent abstract data structure to use at each stage and also the most efficient internal storage structure into which to map the abstract data structure.

ABSTRACT DATA STRUCTURES:

A data structure is defined as a set of rules and constraints which show the relationships that exist between individual pieces of data. The structure says nothing about the individual pieces of data which may occur. It may require them to hold the structure in some sense, but any information contained in the data items is independent of the structure.

JKDirectory!

The term item used here will denote a piece of an abstract data structure. The item itself may be another data structure so that a hierarchical set of data structures is built up.

String: A string is an ordered set of items. The string may either be of variable length or fixed.

Array: An array A is a set of items which are so arranged that an ordered set of integers uniquely defines the position of each item of the array and provides a method of accessing each item directly.

Queues and Stack: Queues and stacks are dynamically changing data structures. At any time, the queue or stack contains an ordered set of items. In the case of a queue, if an item is added, it is placed at the end of the ordered set. Items can only be accessed or removed from the front of the ordered set defining the queue.

The first item added to the queue is the only one accessible and must be the first removed. For a stack, items are similarly added to the end of the ordered set. The difference is that the accessing or removal of items from the ordered set is from the end of the set. The last item

added is the only one accessible and is the first to be removed. The items may well be of variable length and contain the length defined within the item.

Table: A table consists of a set of items, and associated with each item is a unique name, or key. In general, each item will consist of its key, together with some information associated with the key. An item can be added to the table by presenting its key together with the associated information. Items are accessed by presenting the key to the table.

Tree: A tree is a structure consisting of a set of nodes. Each node (or item) has the property that, apart from information that it may carry, it also contains pointers to lower-level nodes. At the lowest level of the tree, the nodes point to leaves which consist of data structures external to the tree. The idea of level comes from the fact that each tree must have one top-most node which has no pointers to it from other nodes (this is often called the root of the tree) and also no node can point to a node previously defined. This latter condition ensures that each node has a unique path to it from the root.

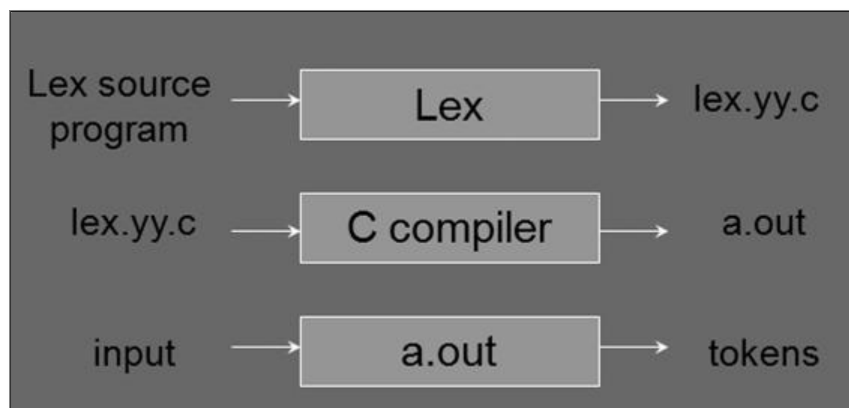
LEX LEXICAL ANALYZER GENERATOR:

The UNIX utility lex parses a file of characters. It uses regular expression matching; typically it is used to 'tokenize' the contents of the file. Lex uses patterns that match strings in the input and converts the strings to tokens. Lex generates C code for a lexical analyzer, or scanner.

JKDirectory!

An overview of Lex:

Initially to the lex utility a lex source program with extension .l is given as input, which will generates the C file (lex.yy.c) as output. Later this C file is compiled with the C compiler that produces the output "a.out". To this output if we give the input as the stream of characters (lexemes) then it will produce the output as tokens.



STRUCTURE OF A LEX FILE:

A lex file is divided in to three sections by %% delimiters. The general format of lex source is given as below:

{Definitions}

%%

{Transition rules}

%%

{User subroutines}

DEFINITION SECTION:

There are three things that can go in the definitions section:

C code Any indented code between %{ and %} is copied to the C file. This is typically used for defining file variables, and for prototypes of routines that are defined in the code segment.

Definitions A definition is very much like #define cpp directive. For example

```
letter [a-zA-Z]
digit [0-9]
punct [,.:;!?]
nonblank [ˆ \t]
```

These definitions can be used in the rules section: one could start a rule as below:

```
{letter}+ {...
```

RULE SECTION:

The rules section has a number of pattern-action pairs. The patterns are regular expressions and the actions are either a single C command, or a sequence enclosed in braces.

If more than one rule matches the input, the longer match is taken. If two matches are the same length, the earlier one in the list is taken.

If the application of a rule depends on context, there are a couple of ways of dealing with this.

We distinguish between 'left state' and 'right state', basically letting a rule depend on what comes before or after the matched token.

Left state

A rule can be prefixed as *<STATE>(some pattern) {...* meaning that the rule will only be evaluated if the specified state holds; Switching between states is done in the action part of the rule: *<STATE>(some pattern) {some action; BEGIN OTHERSTATE;}* The initial state of lex is INITIAL.

Right state

It is also possible to let a rule depend on what follows the matched text. For instance abc/de {some action} means ‘match abc but only when followed by de. This is different from matching on abcde because the de tokens are still in the input stream, and they will be submitted to matching next.

USER CODE:

If the lex program is to be used on its own, this section will contain a main program. If you leave this section empty you will get the default main:

```
int main()
{
yylex();
return 0;
}
```

Where yylex is the parser that is built from the rules

BUILDING AN APPLICATION WITH LEX

General commands to create an application using lex.

lex lexFile.l

Generates (outputs) a C file ***lex.yy.c*** and constructs a C function *yylex()*.

Compile and link with lex library.

cc -o scanner lex.yy.c –ll

SPECIAL LEX VARIABLES

yytext is an external variable that contains the matched string.

yyleng is an external variable that contains the length (the number of characters) of the matched string.

yylineno is an external variable that contains the number of the current input line (version dependent).

Example:

Find a name and bracket it with < and > if it is encountered.

nameMatch.l

```
%{
#include <stdio.h>
%}
%%
Dexter|DeeDee printf( "<%s>", yytext );
%%
```

Input: cartoon.dat

Animaniacs: Yakko, Wakko, Dot

Bugs Bunny: Bugs Bunny

Dexter’s Laboratory: Dexter, DeeDee

Speed Racer: Speed, ChimChim

Spongebob Squarepants: Spongebob

Output

```
% ./example2 < cartoon.dat
```

```
Animaniacs: Yakko, Wakko, Dot
Bugs Bunny: Bugs Bunny
<Dexter>’s Laboratory: <Dexter>, <DeeDee>
Speed Racer: Speed, ChimChim
Spongebob Squarepants: Spongebob
```

JKDirectory!