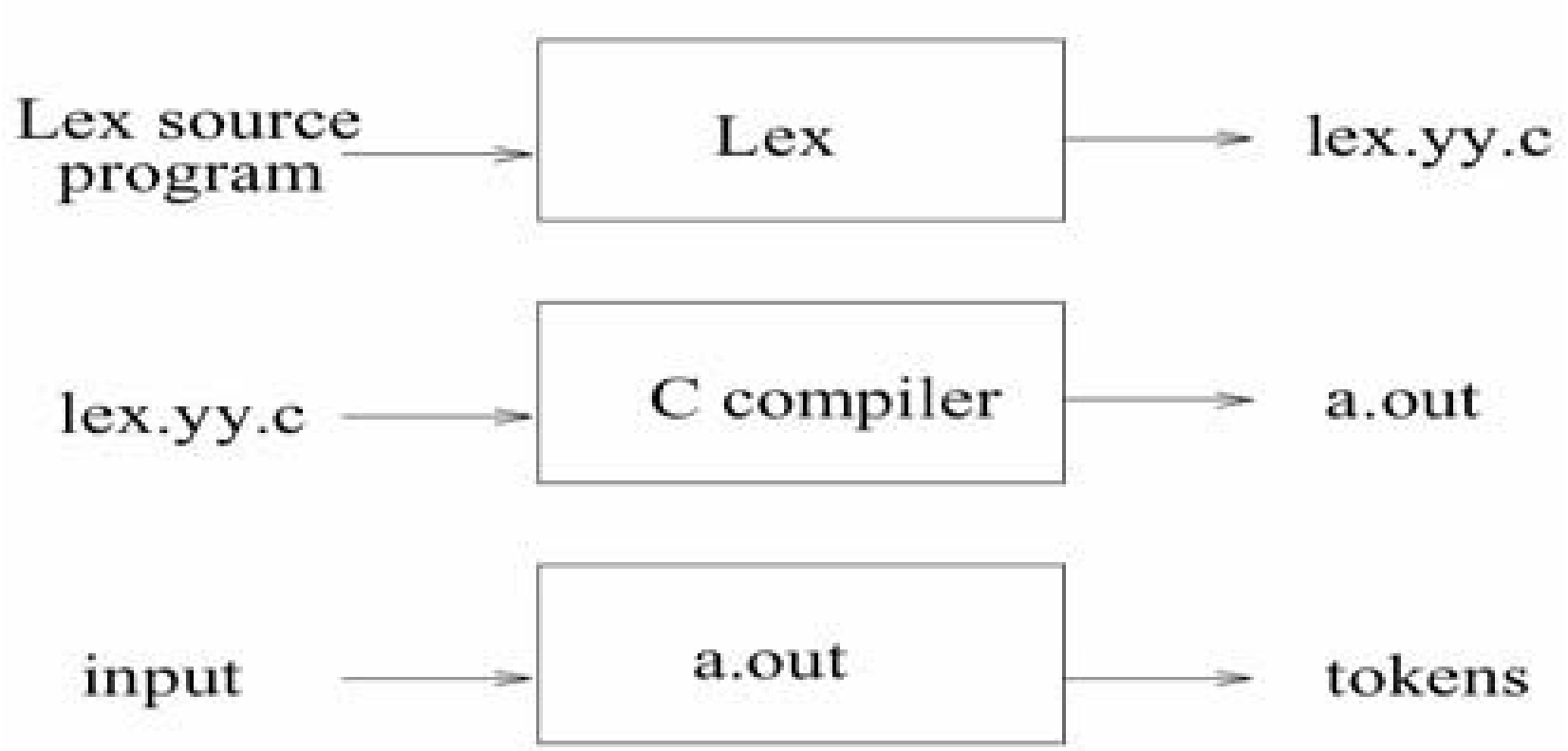
Tutorial on Lex

Lex Overview

- Lex is a tool for automatically generating a lexical analyzers or scanner given a lex specification (.l file)
- Lexical analyzers *tokenize input streams.*
- Tokens are the *terminals of a language.*
- Regular expressions define *tokens.*

Usage Paradigm of Lex



Lex Internals Mechanism

- Converts regular expressions into DFAs.
- DFAs are implemented as table driven state machines.

General Format of Lex Source (.l file)

source.l → *.c is generated after running

%{

< C global variables, prototypes, comments > → This part will be embedded into *.c

%}

[DEFINITION SECTION] → substitutions, code and start states; will be copied into *.c

%%

[RULES SECTION] → define how to scan and what action to take for each token

%%

< C auxiliary subroutines> → any user code. For example, a main function to call the scanning function yylex().

General Format of Lex Source (.l file)

- Input specification file is divided in three parts:
  - Definitions: Declarations
  - Rules: Token Descriptions and actions
  - Subroutines: User-Written code
- These three parts are separated by %%
- The first %% is always required as there must be a rules section
- If any rule is not specified,then by default everything on input will be copied to output
- Defaults for input and output are stdin and stdout

Sample Program

```
%%
                /* match everything except newline */
.               ECHO;
                /* match newline */
\n      ECHO;
%%
int yywrap(void) {
return 1;
}
int main(void) {
        yylex();
        return 0;
}
```

Sample Program

- Two patterns have been specified in the rules section.
- Each pattern must begin in column one.
- This is followed by whitespace (space, tab or newline) and an optional action associated with the pattern.
- The action may be a single C statement, or multiple C statements, enclosed in braces.
- Anything not starting in column one is copied as it is to the generated C file.

How to compile and run a lex program

- lex filename (.l)
- cc lex.yy.c –o executable_filename
- ./executable_filename

The three section

```
%{
        C declarations and includes
%}
<name>          <regexp>
<name>          <regexp>
……
%%
<regexp>        { <action to take when matched> }
<regexp>        { <action to take when matched> }
……
%%
User subroutines (C Code)
```

Sample program to identify letters

```
%{

%}
letter   [A-Za-z]
%%
                        /* match letters */
{letter}+               { printf("Letter Read");}
%%
int yywrap(void) {
    return 1;
}
int main(void) {
    yylex();
    printf("Program ends\n");
    return 0;
}
```

Metacharacter	Matches
`.`	any character except newline
`\n`	newline
`*`	zero or more copies of the preceding expression
`+`	one or more copies of the preceding expression
`?`	zero or one copy of the preceding expression
`^`	beginning of line
`$`	end of line
`a|b`	`a` or `b`
`(ab)+`	one or more copies of `ab` (grouping)
`"a+b"`	literal "`a+b`" (C escapes still work)
`[]`	character class

Table 1: Pattern Matching Primitives

Expression	Matches
abc	abc
abc*	ab abc abcc abccc ...
abc+	abc abcc abccc ...
a(bc)+	abc abcbc abcbcbc ...
a(bc)?	a abc
[abc]	one of: a, b, c
[a-z]	any letter, a-z
[a\-z]	one of: a, -, z
[-az]	one of: -, a, z
[A-Za-z0-9]+	one or more alphanumeric characters
[\t\n]+	whitespace
[^ab]	anything except: a, b
[a^b]	one of: a, ^, b
[a|b]	one of: a, |, b
a|b	one of: a, b

Table 2: Pattern Matching Examples

Name	Function
`int yylex(void)`	call to invoke lexer, returns token
`char *yytext`	pointer to matched string
`yyleng`	length of matched string
`yylval`	value associated with token
`int yywrap(void)`	wrapup, return 1 if done, 0 if not done
`FILE *yyout`	output file
`FILE *yyin`	input file
`INITIAL`	initial start condition
`BEGIN`	condition switch start condition
`ECHO`	write matched string

Table 3: Lex Predefined Variables

Meta-characters

- Meta-characters (do not match themselves)
  - ( ) [ ] { } < > + / , ^ * | . \ " $? - %
- To match a meta-character, prefix with "\"
- To match a backslash, tab or new line, use \\, \t, or \n

Regular Expression Examples

- an integer : [1-9][0-9]*
- a word : [a-zA-Z]+
- a (possibly) signed integer : [-+]?[1-9][0-9]*
- a floating point number : [0-9]*.[0-9]+

Lex Regular Expressions

Lex uses an extended form of regular expression:
(c: character, x,y: regular expressions, s: string, *m*,*n* integers and i:identifier).

- c — Any character except meta-characters
- [...] — The list of enclosed chars (may be a range)
- [∧...] — The list of chars not enclosed
- . — Any ASCII char except newline
- xy — Concatenation of x and y
- x* — Same as x*
- x+ — Same as x^+
- x? — An optional x

Default Rules and Actions

- The first and second part must exist, but may be empty, the third part and the second %% are optional.
- If the third part does not contain a main(), It will link a default main() which calls yylex() then exits.
- Unmatched patterns will perform a default action, which consists of copying the input to the output.
- Lex will always match the longest (number of characters) token possible.
- If two or more possible tokens are of the same length, then the token with the regular expression that is defined first in the lex specification is favored.

Special Variables and Functions

- yytext : Where text matched most recently is stored
- yyleng : Number of characters in text most recently matched
- yylval : Associated value of current token
- yymore() : Append next string matched to current contents of yytext
- yyless(n) : Remove from yytext all but the first n characters
- unput(c) : Return character c to input stream
- yywrap() : May be replaced by user

  The yywrap method is called by the lexical analyzer whenever it inputs an EOF as the first character when trying to match a regular expression

Example: Program to count number of lines, words, characters

```
%{
    int nchar, nword, nline;
%}
%%
\n                      { nline++; nchar++; }
[^ \t\n]+               { nword++, nchar += yyleng; }
.                       { nchar++; }
%%
int yywrap(void) {
    return 1;
}
int main(void) {
    yylex();
    printf("%d\t%d\t%d\n", nchar, nword, nline);
    return 0;
}
```