Lex Practice

Table 1: Special Characters

Pattern	Matches
`.`	any character except newline
`\.`	literal `.`
`\n`	newline
`\t`	tab
`^`	beginning of line
`$`	end of line

Table 2: Operators

Pattern	Matches
`?`	zero or one copy of the preceding expression
`*`	zero or more copies of the preceding expression
`+`	one or more copies of the preceding expression
`a|b`	`a` or `b` (alternating)
`(ab)+`	one or more copies of `ab` (grouping)
`abc`	`abc`
`abc*`	`ab abc abcc abccc ...`
`"abc*"`	literal `abc*`
`abc+`	`abc abcc abccc abcccc ...`
`a(bc)+`	`abc abcbc abcbcbc ...`
`a(bc)?`	`a abc`

Table 3: Character Class

Pattern	Matches
`[abc]`	one of: `a b c`
`[a-z]`	any letter a through 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`

Regular expressions are used for pattern matching. A character class defines a single character and normal operators lose their meaning. Two operators supported in a character class are the hyphen ("`-`") and circumflex ("`^`"). When used between two characters the hyphen represents a range of characters. The circumflex, when used as the first character, negates the expression. If two patterns match the same string the longest match wins. In case both matches are the same length, then the first pattern listed is used.

... *definitions* ...
`%%`
... *rules* ...
`%%`
... *subroutines* ...

Input to Lex is divided into three sections with `%%` dividing the sections. This is best illustrated by example. The first example is the shortest possible lex file:

```
%%
```

Input is copied to output one character at a time. The first `%%` is always required as there must always be a rules section. However if we don't specify any rules then the default action is to match everything and copy it to output. Defaults for input and output are **`stdin`** and **`stdout`**, respectively. Here is the same example with defaults explicitly coded:

```
%%
    /* match everything except newline */
.   ECHO;
    /* match newline */
\n  ECHO;

%%
```

```
int yywrap(void) {
    return 1;
}

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

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 verbatim to the generated C file. We may take advantage of this behavior to specify comments in our lex file. In this example there are two patterns, "**.**" and "**\n**", with an **ECHO** action associated for each pattern. Several macros and variables are predefined by lex. **ECHO** is a macro that writes code matched by the pattern. This is the default action for any unmatched strings. Typically, **ECHO** is defined as:

```
#define ECHO fwrite(yytext, yyleng, 1, yyout)
```

Variable **yytext** is a pointer to the matched string (NULL-terminated) and **yyleng** is the length of the matched string. Variable **yyout** is the output file and defaults to **stdout**. Function **yywrap** is called by lex when input is exhausted. Return 1 if you are done or 0 if more processing is required. Every C program requires a **main** function. In this case we simply call **yylex** that is the main entry-point for lex . Some implementations of lex include copies of **main** and **yywrap** in a library thus eliminating the need to code them explicitly. This is why our first example, the shortest lex program, functioned properly.

Table 4: Lex Predefined Variables

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	wrapup, return 1 if done, 0 if not

yywrap(void)	done
FILE *yyout	output file
FILE *yyin	input file
INITIAL	initial start condition
BEGIN condition	switch start condition
ECHO	write matched string

Here is a program that does nothing at all. All input is matched but no action is associated with any pattern so there will be no output.

```
%%
.
\n
```

The following example prepends line numbers to each line in a file. Some implementations of lex predefine and calculate **yylineno**. The input file for lex is **yyin** and defaults to **stdin**.

```
%{
    int yylineno;
%}
%%
^(.*)\n    printf("%4d\t%s", ++yylineno, yytext);
%%
int main(int argc, char *argv[]) {
    yyin = fopen(argv[1], "r");
    yylex();
    fclose(yyin);
}
```

The definitions section is composed of substitutions, code, and start states. Code in the definitions section is simply copied as-is to the top of the generated C file and must be bracketed with "%{" and "%}" markers. Substitutions simplify pattern-matching rules. For example, we may define digits and letters:

```
digit     [0-9]
letter    [A-Za-z]
%{
    int count;
%}
%%
    /* match identifier */
{letter}({letter}|{digit})*    count++;
%%
int main(void) {
    yylex();
```

```
    printf("number of identifiers = %d\n", count);
    return 0;
}
```

Whitespace must separate the defining term and the associated expression. References to substitutions in the rules section are surrounded by braces (**{letter}**) to distinguish them from literals. When we have a match in the rules section the associated C code is executed. Here is a scanner that counts the number of characters, words, and lines in a file (similar to Unix wc):

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