Appendix B

Cheat-sheet: LEX

B.1 LEX Source

The general format of a Lex source is:

Listing B.1.1: format.l — C code

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

The second `%%` is optional, but the first is required to mark the beginning of the rules. The absolute minimum Lex program is described below and translates into a program which copies the input to the output unchanged.

Listing B.1.2: smallest_lex_file.l — C code

```
%%
```

The **definitions** section may contain:

- translations for repetitive regular expressions
- `includes` of libraries and global variable definitions, enclosed in `%{ %}`
- a list of start conditions
- others, such as:
  - selection of a host language
  - a character set table
  - adjustments to the default size of arrays within Lex itself

The **rules** section represents the user's control decisions. Rules are a table in which:

- the left column contains regular expressions
- the right column contains actions, program fragments to be executed when the expressions are recognized

The **user subroutines** section is often omitted, but may contain overrides for the `yywrap()` function which is called whenever Lex reaches an end-of-file, and any other user defined functions.

Note B.1.1

In the definitions and rules sections, any indented text, or text enclosed in `%{ and %}` is copied verbatim to the output (with the `%{}`'s removed). The `%{}`'s must appear unindented on lines, and by themselves.

A sample lex file containing typically used elements:

Listing B.1.3: sample.l — C code

```
%{
#include <stdio.h>
#include "y.tab.h"
int c;
%}
D               [0-9]
%%
" "             ;
[a-z]           {
                  c = yytext[0];
                  yylval.a = c - 'a';
                  return(LETTER);
                }
{D}             {
                  c = yytext[0];
                  yylval.a = c - '0';
                  return(DIGIT);
                }
[^a-z0-9\b]     {
                    c = yytext[0];
                    return(c);
                }
%%
yywrap ()
{
  < ... some post processing ... >
  return 1;
}
```

Concept B.1.1: Comments

In Lex, C-like comments such as `/* ... */` are valid, but for the rules section, only comments for lines that are indented (with whitespaces or tabs) will be correctly identified.

Listing B.1.4: comments.l C code

```
%{  /*
This is a valid lex file. The comments are all correct for Lex.  This is Comment 0.
Also note the three other comments in various locations in this code.
As always, note the use of white-space before any C code.

Comment 0:  Within %{ ...%}.
Comment 1:  In the comments section, preceded by a blank.
Comment 2:  In the body, within a compound C statement (in braces).
Comment 3:  In the body, preceded by a blank.  Also note semi-colon.
*/
%}
 /*comment1*/
int i,x,y;
D       [0-9]
H       [a-fA-F]
%%
[xX]({H}|{D})+  { /*comment2*/
               y=0;
               for (i=1 ; i < strlen (yytext) ; i++) {
                  x=yytext [i];
                  if (x >= '0' && x <= '9') {x = x - '0';}
                  if (x >= 'A' && x <= 'F') {x = x - 'A' + 10;}
                  if (x >= 'a' && x <= 'f') {x = x - 'a' + 10;}
                  y = y * 16 + x;}
              printf ("%s\t%d\t%d\n", yytext, strlen (yytext)-1 ,y);}
.* /*comment3*/;
```

B.2 Regular Expressions in LEX

Regular expressions in Lex may specify a set of characters/strings to be matched. It can contain:

- expressions specific to exact letters/digits to be matched
- expressions specific to exact operator characters `" \ [ ] ^ - ? . * + | ( $ / { } .`
- expressions with meta characters that have a special meaning
- expressions with repetitions, choices and other properties

For some of the use cases in the tables below, multiple examples are provided, separated by a semicolumn.

Table B.1: General Meta Characters

Usage	Utility	Example	Example Notes
text characters	Any Character corresponding to those in the input string (letters and digits)	`a`; `123`	matches the character a, matches the digits `1`, `2` and `3` in this order
`.`	Any Character Except New Line	`a.*b`	matches any string that begins with `a`, ends with `b`, and has any other character(s) except for newline inbetween
`-`	Specifies ranges when used in a character class. If needed as non-special character, it should be specified at the beginning or end of the character class.	`[a-z0-9<>]`; `[-+0-9]`	matches lowercase letters, digits, angular brackets; matches plus and minus signs and digits
`{x}`	Translation of x from the definitions section	`{D}`, where D is defined as `D [0-9]`	uses the translation for the Digits definition `D` to match digits
`\`	Escapes characters to avoid special behavior, or specifies a special character	`[\40-\176]`; `\n`; `\*`	matches all ASCII characters from blank to tilda; matches newline; matches the star character
`\n`	Matches New Line	`[a-z][0-9]\n`	matches lines that contain a letter and a digit
`\t`	Matches Tab	`[ \t] ;`	skips tabs and spaces in the input string

Table B.2: Anchor Meta Characters

Usage	Utility	Example	Example Notes
`^`	If `^` is the first character in an expression, then the expression will only be matched at the beginning of a line (used outside of a character class).	`^abc`	matches the string `abc` if found at the beginning of a line
`^`	Negates the set that follows if used within a character class.	`[^abc]`	matches any character except for `a`, `b` and `c`
`$`	If `$` is the last character in an expression, then the expression will only be matched at the end of a line.	`abc$`	matches the string `abc` if found at the end of a line
`x/y`	Trailing Context. If `x` and `y` are two LEX regular expressions then `x/y` is another LEX regular expression that matches `x` if `x` is followed by `y`. After use in this context, `y` is returned to the input before the action is executed (the action only sees the text matched by `x`)	`ab/cd`; `ab/$`	matches the string `ab` if followed by `cd`; equivalent to `ab/\n`
`<y>x`	x when lex is in start condition y	`<ONE>x`	matches the character `x` if start condition `ONE` is met

Table B.3: Grouping Meta Characters

Usage	Utility	Example	Example Notes
`[]`	Matches Characters specified by the set within brackets	`[xy]`; `[x-z]`	matches characters `x` or `y`; matches characters `x`, `y` or `z`
`[^ ]`	Matches Characters NOT specified by the set within the brackets	`[^0-9]`; `[^abc]`	anything but digits; anything but `a`, `b` or `c`
`x|y`	Either `x` Or `y`	`ab|cd`	matches the string `ab` or the string `cd`
`x|`	The action for `x` is the action for the next rule	`[a-z] |` followed by `[A-Z] ;`	does nothing for both lowercase and uppercase letters
`( )`	Groups the expressions specified inside. Not necessary for Either Or, but useful for complex expressions	`(ab|cd+)?(ef)*`	matches strings such as `abefef`, `efefef`, `cdef`, `cddd`, but not strings like `abc`, `abcd`, or `abcdef`

Table B.4: Quantifier Meta Characters

Usage	Utility	Example	Example Notes
`x*`	0 or More instances of x	`[A-Za-z][A-Za-z0-9]*`	all alphanumeric strings that start with a letter
`x+`	1 or More instances of x	`[a-z]+`	any string containing lowercase letters except for the empty string
`x?`	0 or One instance of x	`ab?c`	matches ac or abc
`x{m,n}`	m through n occurences of x	`[a-z]{1,3}`	any string containing 1 to 3 lowercase letters

Note B.2.1

When a special character needs to be used as a text character, it must be escaped using "\". Besides "\", you can also enclose a character within quotes to avoid escaping it.
Example: `xyz ++` is the same as `xyz "++"` and as `" xyz ++"`.