Lex - A Lexical Analyzer Generator

Lex is a program generator designed for lexical processing of character input streams.

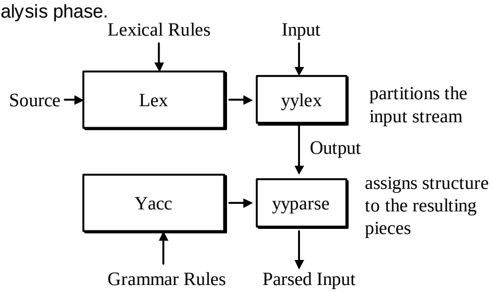
- It accepts a high-level, problem oriented specification for character string matching,
- and produces a program in a general purpose language which recognizes regular expressions.

- The regular expressions are specified by the user in the source specifications given to Lex.
  - Lex generates a deterministic finite automaton from the regular expressions in the source.
  - This automaton is, rather than compiled, in order to save space.

- 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.
- Lex turns the user's expressions and actions (called source) into the host general-purpose language; the generated program is named `yylex`.
  - The `yylex` program will recognize expressions in a stream (called input) and perform the specified actions for each expression as it is detected.

Lex can be used alone for simple transformations,
or can be used with a parser generator to perform the lexical analysis phase.



Examples

- A program to delete from the input all blanks or tabs at the ends of lines:

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

- To change any remaining string of blanks or tabs to a single blank, add another rule:

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

Lex Source

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.

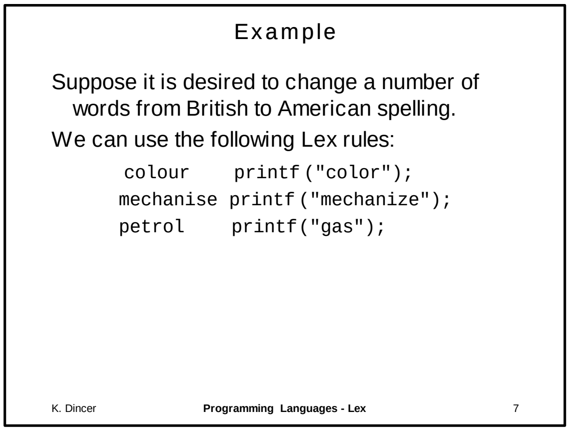
The absolute minimum Lex program is thus (no definitions, no rules)
which translates into a program
which copies the input to the output unchanged.

The Rules

- represent the user's control decisions;
- are a table, in which
  - the left column contains regular expressions (see section 3)
  - the right column contains actions, program fragments to be executed when the expressions are recognized.

Ex: `integer   printf("found keyword INT");`

- A single C expression can just be given on the right side of the line;
- A compound or multi-line expression should be enclosed in braces.

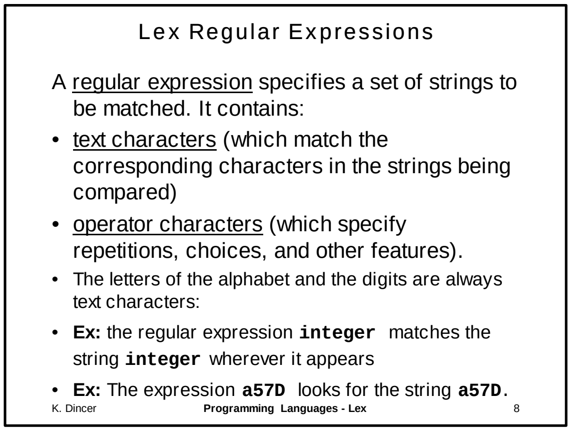


Example

Suppose it is desired to change a number of words from British to American spelling.

We can use the following Lex rules:

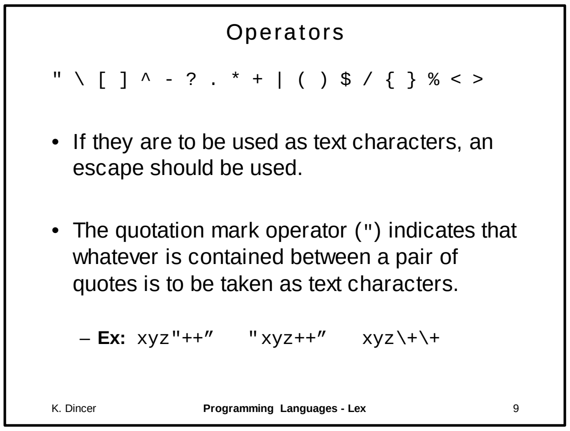
```
 colour    printf("color");
mechanise printf("mechanize");
petrol    printf("gas");
```



Lex Regular Expressions

A regular expression specifies a set of strings to be matched. It contains:

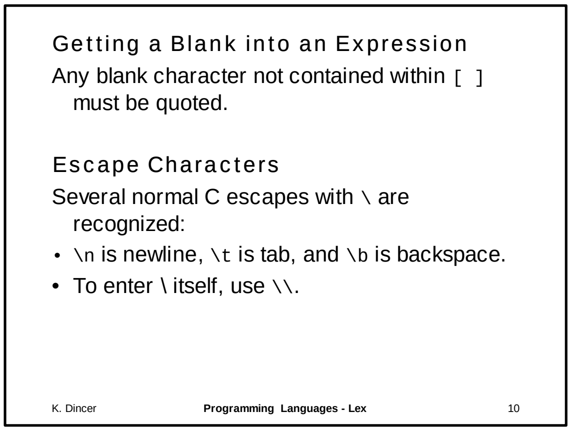
- text characters (which match the corresponding characters in the strings being compared)
- operator characters (which specify repetitions, choices, and other features).
- The letters of the alphabet and the digits are always text characters:
- **Ex:** the regular expression **`integer`** matches the string **`integer`** wherever it appears
- **Ex:** The expression **`a57D`** looks for the string **`a57D`**.



Operators

```
" \ [ ] ^ - ? . * + | ( ) $ / { } % < >
```

- If they are to be used as text characters, an escape should be used.
- The quotation mark operator (`"`) indicates that whatever is contained between a pair of quotes is to be taken as text characters.
  - **Ex:** `xyz"++”   "xyz++”   xyz\+\+`



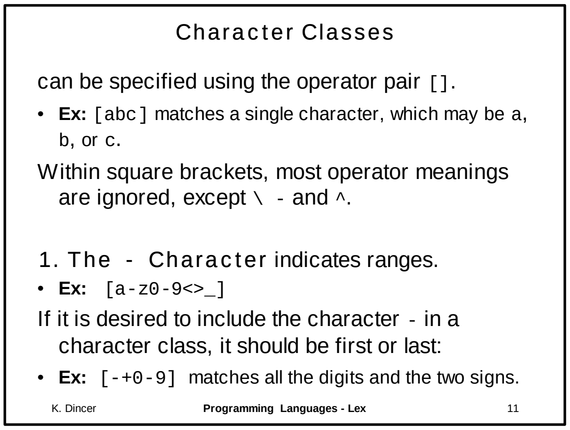
Getting a Blank into an Expression

Any blank character not contained within `[ ]` must be quoted.

Escape Characters

Several normal C escapes with `\` are recognized:

- `\n` is newline, `\t` is tab, and `\b` is backspace.
- To enter \ itself, use `\\`.



Character Classes

can be specified using the operator pair `[]`.

- **Ex:** `[abc]` matches a single character, which may be `a`, `b`, or `c`.

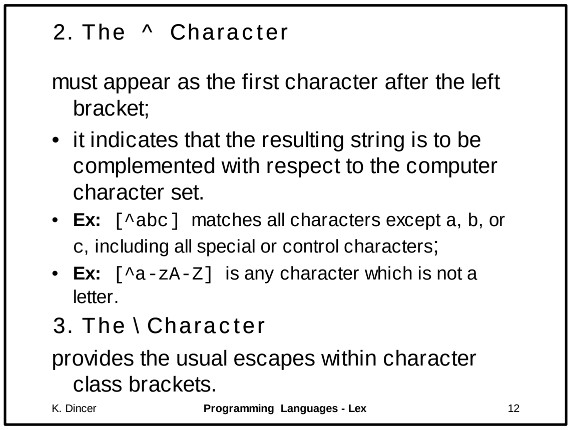
Within square brackets, most operator meanings are ignored, except `\` `-` and `^`.

1. The - Character indicates ranges.

- **Ex:** `[a-z0-9<>_]`

If it is desired to include the character `-` in a character class, it should be first or last:

- **Ex:** `[-+0-9]` matches all the digits and the two signs.



2. The ^ Character

must appear as the first character after the left bracket;

- it indicates that the resulting string is to be complemented with respect to the computer character set.
- **Ex:** `[^abc]` matches all characters except a, b, or c, including all special or control characters;
- **Ex:** `[^a-zA-Z]` is any character which is not a letter.

3. The \ Character

provides the usual escapes within character class brackets.

Arbitrary Character

To match almost any character, the operator character . is used.

- . is the class of all characters except newline.

Optional expressions.

The operator ? indicates an optional element of an expression.

- **Ex:** `ab?c` matches either `ac` or `abc`.

Repeated Expressions

Repetitions of classes are indicated by the operators `*` and `+`.

- **Ex:** `a*` is any number of consecutive a characters, including zero.
- **Ex:** `a+` is one or more instances of a.
- **Ex:** `[a-z]+` is all strings of lower case letters.
- **Ex:** `[A-Za-z][A-Za-z0-9]*` indicates all alphanumeric strings with a leading alphabetic character.
  - This is a typical expression for recognizing identifiers in computer languages.

Alternation and Grouping

The operator `|` indicates alternation

- Ex: `(ab|cd)` matches either `ab` or `cd`.
- Note that parentheses are used for grouping, although they are not necessary on the outside level; `ab|cd` would have sufficed.

Parentheses can be used for more complex expressions:

- **Ex:** `ab|cd+)?(ef)*` matches such strings as `abefef`, `efefef`, `cdef`, or `cddd`; but not `abc`, `abcd`, or `abcdef`.

Context Sensitivity

Lex will recognize a small amount of surrounding context via operators `^` and `$`:

- If the first character of an expr. is `^`, the expr. will only be matched at the beginning of a line.
  - This can never conflict with the other meaning of `^`, complementation of character classes, since that only applies within the `[]` operators.
- If the very last character is `$`, the expression will only be matched at the end of a line.

`$` operator is a special case of the / operator character, which indicates trailing context.

- Ex: `ab/cd` matches the string `ab`, but only if followed by `cd`. Thus `ab$` is the same as `ab/\n`

Repetitions and Definitions

The operators `{}` specify

- either repetitions (if they enclose numbers)
- or definition expansion (if they enclose a name). The definitions are given in the first part of the Lex input, before the rules.

Ex: `{digit}` looks for a predefined string named digit and inserts it at that point in the expression.

Ex: In contrast, `a{1,5}` looks for `1` to `5` occurrences of a.

Finally, initial `%` is special, being the separator for Lex source segments.

A Lex Example

```
%{
/* a sample bit of code */ }  Directly copied
%}
ws    [ \t]       }
nonws [^ \t\n]    }  Definitions
%%
int cc = 0, wc = 0, lc = 0;                              }
{nonws}+     cc += yyleng; ++wc;                         }
{ws}+        cc += yyleng;                               }  Rules
\n           ++lc; ++cc;                                 }
<<EOF>>  {                                               }
    printf( "%8d %8d %8d\n", lc, wc, cc);                }
    yyterminate();                                       }
 }
             Actions
```