When properly used, Lex & YACC allow you to parse complex languages with ease. Although these programs shine when used together, they each serve a different purpose. The examples below will explain what each part does.

The program Lex generates a so called `Lexer'. This is a function that takes a stream of characters as its input, and whenever it sees a group of characters that match a key, takes a certain action. **Example 1** illustrates this. (You may wonder how the program runs, as we didn't define a main() function. This function is defined for you in libl (liblex) which we compiled in with the -ll command or -lfl).

Example 2 shows how to use regular expressions in Lex.

Example 3 is a more complicated example for a C-like syntax. Let's say we want to parse a file that looks like this:

```
logging {
        category lame-servers
{ null; };
        category cname { null; };
};

zone "." {
        type hint;
        file "/etc/bind/db.root";
};
```

The examples we have seen so far show that Lex is able to read arbitrary input, and determine what each part of the input is. This is called '**Tokenizing**'.
YACC can parse input streams consisting of tokens with certain values. This clearly describes the relation YACC has with Lex, YACC has no idea what 'input streams' are, it needs preprocessed tokens.

Example 4 considers the problem of a thermostat that we want to control using a simple language.

We note two important changes. First, we include the file 'y.tab.h', and secondly, we no longer print stuff, we return names of tokens. This change is because we are now feeding it all to YACC, which isn't interested in what we output to the screen. File y.tab.h has definitions for these tokens.
The function *yywrap()* can be used to continue reading from another file. It is called at EOF and you can then open another file, and return 0. Or you can return 1, indicating that this is truly the end. Then there is the main() function, that does nothing but set everything in motion.

The last line simply defines the tokens we will be using. These are output using *y.tab.h* if YACC is invoked with the '-d' option.

As we've seen, we now parse the thermostat commands correctly, and even flag mistakes properly. But as you might have guessed by the weaselly wording, the program has no idea of what it should do, it does not get passed any of the values you enter.

Let's start by adding the ability to read the new target temperature. In order to do so, we need to learn the NUMBER match in the Lexer to convert itself into an integer value, which can then be read in YACC.

Whenever Lex matches a target, it puts the text of the match in the character string 'yytext'. YACC in turn expects to find a value in the variable 'yylval'. In **Example 5**, we see the obvious solution.