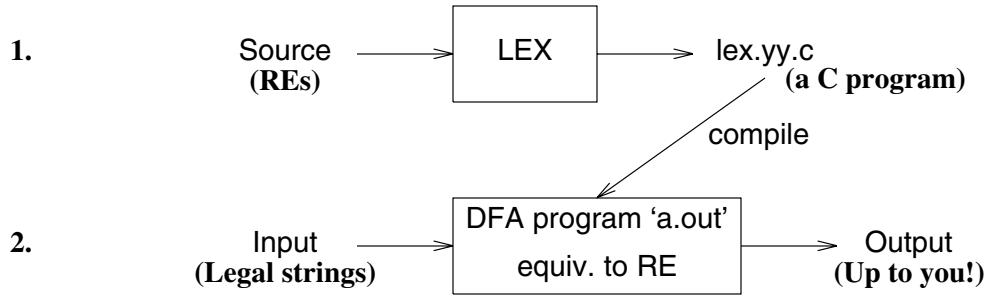
Notes on LEX — a lexical analyzer generator

1. Background

- Written in the middle 1970s by Mike Lesk and Eric Schmidt at Bell Laboratories. Classical compiler 'front end' for recognising things like keywords and identifiers in a programming language.
- LEX is one of the classic UNIX software tools. Sometimes called a 'supertool' because it doesn't *directly* perform a task — it *generates a program* to perform a given task.
- For the most part *lex* is usually used with *yacc* to generate a compiler. Recall that *lex* is a tool that can cope with Chomsky Type 3 grammars, expressed in Regular Expression (RE) form. Indeed REs go a long way towards writing grammers for programming languages. They are almost, but not quite, sufficient because nested constructs in programming languages need Chomsky Type 2 capabilities (and *yacc* provides these).
- GNU versions of these (e.g. under LINUX) now available as *flex* and *bison*‡ .
- So, *lex* accepts an RE definition of legal strings, or 'tokens', to be recognised.
- Provided the RE's are valid then *lex* builds a program in the C language, that can accept strings of characters that are valid within the RE definitions — in other words it builds for you a C simulator of the Deterministic Finite-State Automaton (DFA) that is equivalent to the input RE definitions.

The general picture is:



† Yes, this is the same Eric Schmidt who went on to become Chairman of Google. He complains from time to time that Google employees still delight in sending him bug reports for problems they encounter when using LEX.

‡ Yes, it's all rather silly. It started when Richard Stallman began the GNU project at MIT and, when asked what the acronym meant, demonstrated the true Computer Scientist's love of recursively unending definitions by replying: " It stands for 'GNU's Not UNIX' ". Then, having noted that a gnu is a wildebeeste and that *yacc* sounds like 'yak' (the Tibetan ox) there was irresistable pressure from the UNIX guru community to keep up the tradition of exotic animal names, by naming the GNU version of *yacc* as being *bison*.

The general form of the RE rules for input to *lex* is:

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

where `definitions` and `subroutines` are often omitted.

- The second `%%` is optional, but the first marks the beginning of the rules and must be there.
- Alongside each rule you can place an action (written in C) which is obeyed whenever the corresponding pattern is matched.
- Example: consider the following file, `test.l`

```
%%
a        {ECHO;printf("   rule1\n");}
ab       {ECHO;printf("   rule 2\n");}
a*b+     {ECHO;printf("   rule3\n");}
(a|b)*abb        {ECHO;printf("  rule4\n");}
.*       {ECHO;printf("   not recognised\n");}
\n      {printf(" NEXT R.E. PLEASE\n");}
%%
```

- Notice the `printf` command for printing to the UNIX standard output. Also the pre-defined macro called `ECHO`, in the *lex* environment, which echoes the token that has been recognised.
- To run this program:
  1. Create yourself a UNIX window.
  2. Then do the following:

```
$ lex test.l
$ gcc lex.yy.c -ll
$ mv a.out test
$ test
```

  Note: if '-ll' causes an error message from the linker the try '-lfl' because in a GNU environment it's very likely to be *flex* that you are actually using. Now type in any strings you want to be analysed, terminated by `RETURN` e.g

```
ab
abbbb
aaa
```

  and finish with `<ctrl>C` or `<ctrl>D` to quit the program.
- Note that `$` in the above represents the UNIX prompt. The `gcc` is a call of the GNU C compiler and `-ll` (or `-lfl`) links in the *lex* libraries which contained pre-declared macros and functions for *lex* to use (e.g. for the `ECHO` command).
- The `mv` command can be used to rename `a.out` to something more meaningful to you e.g. `test` or `myprog` or whatever. Alternatively you can use the `-o` option on the `gcc` command line for this renaming e.g. `-o myprog`
- The file extension `.l` is reserved for input to *lex*.
- Metasymbols used by *lex* in the RE syntax are:

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

- If you want any of these symbols to mean themselves then you must either escape them with a \ or enclose them in quotes If in doubt quote the entire string e.g. `"xyz++"`.
- If more than one RE could match the string *lex* uses the (textually) earliest possible rule in its rule set first.
- Another example to recognise (small!) Roman numbers. The input numbers are to be in lower case letters.

```
          int sum=0;
%%
iv        {sum= sum+4;}
v         {sum = sum+5;}
i         {sum = sum+1;}
\n      {printf("answer is %d\n",sum); sum=0; /*prepare sum for next
                                                number. Note %d is pattern
                                                for printing an integer in
                                                a printf statement*/
          }
%%
```

Note the pre-declaration of the integer `sum` to hold the result. This must be indented by at least one space (better still, use a tab).

- Note also the use of `\n` which is standard C/UNIX shorthand for the 'newline' character. Also that RE patterns to be matched can have an *action* associated with them. This is a chunk of C code conventionally laid out one tab stop away from the RE.
- Try this out for yourselves e.g in a file called `roman.l` and see if you can get it working. When you do you can congratulate yourself on writing your first ever 'compiler' (and it *is* a compiler of sorts. Conventional compilers have actions that squirt out machine code equivalents to your high-level language inputs. This one has actions that 'add up' your Roman number).
- You could always write the DFA for your grammar directly in C. But *lex*'s advantage is that the grammar for the REs and the actions is easy to change.
- The RE rules in *lex* allow you to cheat slightly and go outside the strict Type 3 domain e.g. the *lex* rule `ab/cd` means 'accept `ab` but only if it is followed by `cd`'.
- You can cheat also because the actions are in C, which has the full power of a Type 0 grammar (i.e. it has a Turing machine's capabilities). Thus if you wanted your roman number compiler to disallow certain numbers you can either put in an extra RE rule or you could exit the program by checking the value of `sum` in an action such as

```
if (sum==20) {printf("That's plenty thank you!\n"); exit();}
```

- **Exercise**: Adapt `roman.l` to accept roman numbers in either lower or upper case (but not a mixture). Add extra rules to allow valid roman numbers up to one hundred to be recognised.