FLEX TUTORIAL

1. Getting started

Flex is an open-source lexical analyzer generator. These scanners parse a data stream (a file or a buffered string) looking for patterns, and executing some actions when said patterns are found. This guide is just an introduction to a basilar flex usage: for more details, please read the official manuale.

2. The input file

In order to generate the C scanner, flex needs a formatted input file, defining the rules. This file uses the .lpp extensions, and it's divided into three parts:

```
definitions
%%
rules
%%
code
```

2.1 Definitions

If you like to have some code before anything else in the scanner's code, you can define one or more top blocks. They are usually written to insert important preprocessor directives.

```
%top
{
  all the code in here will be before anything else
}
```

It's then possible to define particular behavious for the flex scanner. Here some options to control that:
%option header-file=”header.h” creates an header
%option noyywrap when to the EOF, the scanner assumes there are no more file to parse (if the scanner is not called manually again with another file pointer as input stream).
%option outfile=”scanner.c” scanner's name (default lex.yy.c)
%option warn enables warnings

Sometimes it's useful to match regular expressions with a name easy to remember, so that the code is more readable. Example:

```
DIGIT [0-9]
```

Every use of {DIGIT} inside a rule will be translated into [0-9].

2.2 Rules

They form the heart of the scanner. The classical row is:

```
regexp   { action }
```

For more informations about regular expressions in general, and about the extended set supported by flex, read more specific manuals.
When flex matches a pattern in the stream, it is copied in char *yytext (or in char yytext[] if you used %array), with its length in yyleng.
Lines of C code form the action, separated from the regexp by at least one whitespace. If no action matches a given pattern, all the occurrences of that pattern will be deleted from the output.

Everything that doesn't match any pattern is printed in output as it is (default rule).

There are several special instructions that can be used in actions.
ECHO: prints out yytext;
input(): reads the next character;
yyterminate() exits with success.

3. Start conditions

Start conditions allow you to control the execution of actions, thus creating a sort of state machine inside your scanner. All the conditions must be declared in the definition section:

```
%s cond1 cond2
```

With

```
<cond1>pattern1 {action1;}
```

action1 is executed only if, when the scanner found pattern1, flex was in that condition. At the beginning, flex is in the INITIAL condition.

```
pattern0 {BEGIN(cond1);}
```

moves flex in the cond1 condition when pattern0 is found. You can also write BEGIN(INITIAL).

4. Buffers

By default, the scanner reads its tokens from the standard input. If we want to trigger manually the parsing, making the scanner reads from a particular file:

```
yyset_in(fpointer);
int ret = yylex();
```

Don't forget to include the header if this instructions are in a different file. Other utility functions include:

YY_BUFFER_STATE yy_create_buffer(FILE *f, int size); creates a buffer file with the given size (if unsure, use YY_BUF_SIZE). The returned variable is useful for other functions;
void yy_delete_buffer (YY_BUFFER_STATE buf); deallocates the given buffer;
YY_BUFFER_STATE yy_scan_string(const char *s); scans a string.

5. Conclusion

Every comment, error or feedback to <syn.shainer@gmail.com>
http://giudoku.sourceforge.net

Lisa