Details of Lex

EXAMPLE 1

As a first exercise, we shall write the previous example as a lex script:

```
%%
a*b printf("Token 1 found\n");
c+  printf("Token 2 found\n");
%%
main()
{ yylex();
}
```

Assume that this example is stored in a file `example1.l`. To make a lexical analyser from this lex specification:

```
>lex example1.l
```

This creates a file called `lex.yy.c`. This has to be compiled and linked to the lex library.

```
>cc -o example1 lex.yy.c -ll
```

Now one can invoke the lexical analyser:

```
>example1
aabbccabc
Token 1 found
Token 1 found
Token 2 found
Token 1 found
Token 2 found
```

Note the following:

- **The core of the lex script lies between the two %%s. This consists of a series of *pattern action* pairs.** `a*b` **is an example of a pattern. An action is any valid C statement.**

- **Lex takes a lex script and produces a file called** `lex.yy.c`**. Among other things, this contains the definition of a function** `yylex()`**, which is the lexical analyser.**

- **The part after the second %% contains the function** `main` **calling** `yylex`**.**

- **Linking to the lex library is required because** `lex.yy.c` **uses certain predefined functions like** `yywrap`**,** `yyless`**, and** `yymore`**.**

- **An invocation of** `yylex` **goes through the *match pattern* $\rightarrow$ *perform action* cycle repeatedly till the entire input string is over. Thus all tokens in the input string are returned in a single invocation of** `yylex`**. As we have seen, it is sometimes necessary to return a single token for every invocation of** `yylex`**. To do this:**

```
%%
a*b {printf("Token 1 found\n"); return;}
c+  {printf("Token 2 found\n"); return;}
%%
main()
```

```
{ yylex();yylex
}
```

This behaves as:

```
>example1
aabbccabc
Token 1 found
Token 1 found
```

- **Segments of the input string which do not match any pattern are copied directly onto the output.**

```
abbcsdcf
Token 1 found
Token 1 found
Token 2 found
sdToken 2 found
f
```

- **By default,** `yylex` **reads its input from stdin, and writes it output to stdout.**

STRUCTURE OF A LEX SPECIFICATION

A lex program has three parts:

. . . *definition section*. . .
%%
. . . *rules section*. . .
%%
. . . *user defined functions*. . .

- **The *definition section* may contain, among other things, *literal blocks* and *pattern names*. Literal blocks provide a way for initialising variables used in the actions. While pattern names are shorthand for complex patterns.**

- **As stated earlier, a *rule* is a *pattern* – *action* pair. We shall examine the set of allowed patterns later.**

- **We have also seen the function `main()` being defined in the user defined section. We shall see examples of other functions which can go in at this place.**

- **Why cannot these function be placed in the literal block?**

Structure of lex.yy.c

Global Definitions: The C definitions are inserted verbatim over here. Other lex defined functions and data structures also inserted here.
yylex(); uses patterns and actions are used here.
User defined functions: main () yywrap()

EXAMPLE 2

We want to count the number of characters, words and lines in a piece of text. The resulting lexical analyser should be able to read its input from a file whose name is to be supplied along with the command to invoke the lexical analyser.

```
%{
/* Initialisation of variables to be used in actions */
unsigned charCount = 0, wordCount = 0, lineCount = 0;
%}

word [^ \t\n]+
eol \n

%%

{word} {wordCount++; charCount += yyleng;}
{eol}  {charCount++; lineCount++;}
.      charCount++;

%%

main(argc, argv)
int argc;
char ** argv;
{ if (argc >1) {
     FILE* file;
     file = fopen(argv[1], "r");
     if (!file){
         fprintf(stderr, "could not open %s\n", argv[1]);
```

```
        exit(1);
    }
    yyin = file;
  }
  yylex();
  printf("%d %d %d\n", charCount, wordCount, lineCount);
  return 0;
}
```

Assume that the resulting lexical analyser is in a file called `example2`.

```
>example2 lex.yy.c
>36269 5143 1526
```

Based on the program we can make the following observations:

- **The portion of the code between `%{` and `%}` is inserted verbatim in the early part of the generated lexical analyser. Thse generally consist of declrations that would be used by the later parts.**

- **The definition part also contains names for complex regular expressions. The names are substituted by the corresponding regular expressions in the rules section.**

- **The example contains some examples of patterns. They are**

  1. `[\^ \t\n]+` **– One or more repetetions of any character except a blank, tab or newline.**

2. . – any character except a newline

- To read from a file instead of `stdin`, first use `fopen()` to get a file pointer to the file named in the command line argument. This file pointer is assigned to `yyin`.

EXAMPLE 3

We want to extend the predvious example so that the counting can be done over multiple files.

```
%{
unsigned charCount = 0, wordCount = 0, lineCount = 0;
%}

word [^ \t\n]+
eol \n

%%

{word} {wordCount++; charCount += yyleng;}
{eol}  {charCount++; lineCount++;}
.      charCount++;

%%
int currentfile = 1;
char **filelist;


main(argc, argv)
```

```
int argc;
char ** argv;
{filelist = argv;
 if (argc ==1)
    yylex();
  else
    {FILE * file;
     file = fopen(filelist[currentfile],"r");
     yyin = file;
     yylex();
    };
printf("%d %d %d \n",charCount, wordCount, lineCount);
}

int yywrap()
{
    FILE* file;
    /*    fclose(yyin);
     */    if (filelist[++currentfile] != (char*) 0)
{
    file = fopen(filelist[currentfile], "r");
            yyin = file;
    return 0;
    }
    else return 1;
}
```

- **This example illustrates a non-trivial use of a user defined function.**
- **The function** `yywrap` **has a defualt definition in the lex library. We want override this definition for our**

purpose.

- When `yylex` hits the end of the current file, it calls `yywrap`. If `yywrap` returns 1, `yylex` exits. Otherwise it assumes that there is more input to be read from the file associated with `yyin`.
- The key idea is to redefine `yywrap` so that when it is called, `yyin` is bound to the next file supplied in the command line and 0 is returned. This is done until there are no more files to be read, when a 1 is returned.

PATTERNS

We have seen the definition section and the section containing the user defined functions in a lex script. We shall now examine the ***rules section***. As stated earlier, the rules section consist of ***pattern*** – ***action*** pairs. It is not hard to see that the power of the generated lexical analyser depends on the descriptive power of patterns.

1. ***Pattern*** – `x`, `x` is a single character not in the lex meta-character set.
   Meaning – The character `x` itself.
   Example – `a`

2. ***Pattern*** – `\x`, `x` is a single character in the lex meta-character set.
   Meaning – The character `x` itself.
   Example – `\.`

3. ***Pattern*** – `"s"`, `s` is a string of characters.
   Meaning – The string `s` literally.
   Example – `"[]"`, the string `[]`

4. ***Pattern*** – `[s]`, `s` is a string of characters..
   Meaning – Any single character occurring in `s`.
   Example – `[abc]`, the character `a`, `b` or `c`.

5. ***Pattern*** – `[x-y]`, `x` and `y` are characters.
   Meaning – A shorthand for a character class. Denotes any character in the range `x` to `y`.
   Example – `[a-c]`, the character `a`, `b` or `c`.

6. ***Pattern*** – `[^s]`, `s` is a string of characters.
   Meaning – Any characters except those in string `s`.

Example – `[^ \t\n]`, **All characters except for a blank, a tab or a newline.**

7. ***Pattern*** – `.`
   Meaning – **Any character except for a newline**
   Example – **See Example 3.**

8. ***Pattern*** – ***p***`{m,n}`, ***p*** **is a pattern,** `m` **and** `n`**are integers**
   Meaning – ***m*** **through** ***n*** **occurrences of the pattern** ***p*****.**
   Example – `a{1,3}`, **The strings** `a`, `aa` **and** `aaa`

9. ***Pattern*** – `(`***p***`)`, ***p*** **is a pattern**
   Meaning – **The same as** ***p*****. Parenthesis is used to clarify association**
   Example – `(abc)+`, **One or more occurrences of** `abc`

10. ***Pattern*** – p_1`|`p_2, p_1 **and** p_2 **are patterns**
    Meaning – **The strings represented by** p_1 **or** p_2
    Example – `(abc)+|de`, **One or more occurrences of** `abc` **or the string** `de`

11. ***Pattern*** – p_1p_2, p_1 **and** p_2 **are patterns**
    Meaning – **The strings represented by** p_1 **concatenated with those represented by** p_2
    Example – `(abc)+de`, **One or more occurrences of** `abc` **followed by** `de`

12. ***Pattern*** – p?, p **is a pattern**
    Meaning – **Zero or one occurrences of the strings represented by** p
    Example – `(abc)*`, **Zero or one p occurrences of** `abc`**.**

13. ***Pattern*** – $p*$, p is a pattern
    Meaning – Zero or more occurrences of the strings represented by p
    Example – `(abc)*`, Zero or more occurrences of `abc`.

14. ***Pattern*** – p+, p is a pattern
    Meaning – One or more occurrences of the strings represented by p
    Example – `(abc)*`, One or more occurrences of `abc`.

15. ***Pattern*** – {***n***}, ***n*** is a name defined in the definition section
    Meaning – The strings defined by the pattern corresponding to n
    Example – See Example 2.

EXAMPLES OF PATTERN USAGE

1. We want to represent real numbers through patterns. Here are some examples of reals: `.36`, `45.`, `3.414`, `.21e-6`. The lex script defining reals is:

```
DIGIT [0-9]
%%
({DIGIT}+\.{DIGIT}*|{DIGIT}*\.{DIGIT}+)(e-?{DIGIT}+)?
%%
```

Another way of doing the same thing is:

```
DIGIT [0-9]
%%
{DIGIT}+\.{DIGIT}*(e-?{DIGIT}+)?    |
{DIGIT}*\.{DIGIT}+(e-?{DIGIT}+)?
%%
```

2. In C, a comment is defined as follows: The characters /* introduce a comment, which terminates with the characters */. Comments do not nest.

```
%%
"/*"[^*]*(\*([^/*][^*]*)?)*"*/"  {ECHO;printf("\n");}
%%
main ()
{
    yylex();
}
```

For the input

```
/*///*/****/
/*///**/
/*///***/
/*///*a*b//**/
/**/
/*/*
```

The output is:

```
/*///*/
****/
/*///**/

/*///***/

/*///*a*b//**/

/**/

/*/*
```

To understand the pattern for comments, note that the most general form of a comment is:

1. The string *

2. A (possibly empty) string of characters not containing *.

3. Zero or more repetetions of:

- **a. A * followed by zero or one occurrences of:**
  - **i. Any single character other than \\ or * .**
  - **ii. A (possibly empty) string not containing *.**

4. The string *\\.

3. As a third example we shall consider the definition of strings. A string is a sequence of characters enclosed within " and ". A " within a string is represented as ".

```
%%
\"[^"]*(\"\"[^"]*)*\" {ECHO;printf("\n");}
%%
main ()
{
    yylex();
}
```

The input

```
""
""""
"jghg""nbv"
""""""
"fgb"""
```

gives as output

```
""

""""

"jghg""nbv"

""""""

"fgb"""
```

MORE PATTERNS

Certain patterns are context dependent in the sense that are active only if certain conditions ar fulfilled.

1. ***Pattern*** – **ˆp, p is a pattern**
   Meaning – **Any string represented by p, provided it occurs at the beginning of a line.**
   Example – **Consider the lex specification:**

   ```
   %%
   ^#define   {printf("<");ECHO;printf(">\n");}
   %%
   ```

   For the input:

   ```
   #define   #define
   ```

   The output is:

   ```
   <#define>
      #define
   ```

2. ***Pattern*** – p\$, p is a pattern
   Meaning – Any string represented by p, provided it occurs at the end of a line
   Example – `end$`, the string `end` provided it occurs at the end of a line.

3. ***Pattern*** – p_1/p_2, p_1 and p_2 are patterns
   Meaning – Any string represented by p_1, provided it is followed by a string represented by p_2
   Example – Recall that FORTRAN ignores all spaces except those in comments and Hollerith strings. Therefore it is usual in a Fortran compiler to remove all spaces in a preprocessor pass, and then start lexical analysis. Then a DO statement will appear as:

   ```
   DO50k=1,20,1
   ```

   And an assignment statement might look like:

   ```
   DO50k=1.20
   ```

Here is a lex script that distinguishes between the two:

```
letter [a-zA-Z]
digit  [0-9]
num    {digit}+
id     {letter}({letter}|{digit})*
%%
DO/{num}{id}=({num}|{id}), {printf("<");
                            ECHO;printf(">\n");}
{id}                       {printf("<");
                            ECHO;printf(">\n");}
%%
main ()
{
    yylex();
}
```

For the input:

```
DO101J=1,25
DO101J=1.25
```

The output is:

```
<DO>
101<J>
=1,25
<DO101J>
=1.25
```

START STATES

Sometimes the context condition cannot be captured by the patterns given above. For example, the interpretation of a token might depend on the tokens already seen. In such a case one needs *states*
As an example, in C, the string `<stdio.h>` should be interpreted as a file name, if it is preceded `#include`. This is done by introducing a state, say INCLSTATE. We then want to say that when the lexical analyser is in the state INCLSTATE, interpret the string within the angular brackets as a filename.

```
%s INCLSTATE
%%
^#include  {BEGIN INCLSTATE;}
<INCLSTATE>"<"[^>\n]+">" {..process filename..}
<INCLSTATE>\n {BEGIN INITIAL;}
%%
```

- **In a "normal" state, the patterns `"<"[^>\n]+">"` and `\n` are inactive.**
- **However, the rest of the patterns are valid when the scanner is in INCLSTATE. As an example, we also had a token consisting of some characters enclosed within < and >. Then the following script does not work.**

```
%s INCLSTATE
%%
\<.*\>                     {printf("some token found\n");}
^#include                  {BEGIN INCLSTATE;}
<INCLSTATE>"<"[^>\n]+">"   {printf("<"); ECHO;
                            printf(">\n");}
<INCLSTATE>\n              {BEGIN INITIAL;}
%%
```

For the input:

```
#include <stdio.h>
```

We get the output:

```
 some token found
```

To rectify the situation, we use exclusive states. When the scanner is in an exclusive state `s`**,** ***only*** **the patterns beginning with** `<s>` **are valid.**

```
%x INCLSTATE
%%
\<.*\>                     {printf("some token found\n");}
^#include                  {BEGIN INCLSTATE;}
<INCLSTATE>"<"[^>\n]+">"   {printf("<"); ECHO;
                            printf(">\n");}
<INCLSTATE>\n              {BEGIN INITIAL;}
%%
```

So for the same input:

```
#include <stdio.h>
```

We get the output:

```
 <<stdio.h>>
```

PUTTING IT ALL TOGETHER

We now show a complete lexical analyser for a small programming language:

```
%{
#define LT 1
#define LE 2
#define EQ 3
#define NE 4
#define GT 5
#define GE 6
#define IF 7
#define THEN 8
#define ELSE 9
#define ID 10
#define NUMBER 11
#define RELOP 12
#define LPAREN 13
#define RPAREN 14

#define SYMTABSIZE 211


typedef struct {char* idstring;
                int other_attributes;} symtab_entry;
symtab_entry symtab[SYMTABSIZE];

extern int yylval;

%}
```

```
ws [ \t\n]+
letter [A-Za-z]
digit  [0-9]
id {letter}({letter}|{digit})*
number {digit}+(\.{digit}+)?(E[+\-]?{digit}+)?

%%

{ws}     {/*no action, no return*/}
if       {return(IF);}
then     {return(THEN);}
else     {return(ELSE);}
{id}     {yylval=install_id(); return(ID);}
{number} {yylval=atof(yytext); return(ID);}
"<"      {yylval=LT; return(RELOP);}
"<="     {yylval=LE; return(RELOP);}
"="      {yylval=EQ; return(RELOP);}
"<>"     {yylval=NE; return(RELOP);}
">"      {yylval=GT; return(RELOP);}
">="     {yylval=GE; return(RELOP);}
"("      {return(LPAREN);}
")"      {return(RPAREN);}

%%
```

```
int hash(s)
   /* given a string s returns the hash value */
char* s;
{
  char* p;
  unsigned h = 0, g;
  for (p = s; *p != '\0'; p++) {
    h = (h << 4) + (*p);
    if (g = h&0xf0000000) {
      h = h ^ (g >> 24);
      h = h ^ g;
    }
  }
  return h % SYMTABSIZE;
}


int install_id () {

    /* installs the identifier string in yytext
       in the symbol table symtab */

  int  symtab_index;
  symtab_index = hash(yytext);
  strcpy(symtab[symtab_index].idstring, yytext);
  symtab[symtab_index].other_attributes = yyleng;
  return(symtab_index);
}
```

ODDS AND ENDS

We end by explaining a some of the functions and macros which are provided by lex but have not appeared in the example:

1 The function `yyless(n)` pushes back all but the first `n` characters of the current token into the input stream.

2 The function `yymore(n)` appends the next token to the current one. Instead of overwriting `yytext`, the next token is appended to it.

As an example, consider a pattern to match quoted strings, where a quotation mark within a string can be escaped with backslash.

```
%%
\"[^"]*\"      {if(yytext[yyleng-2]=='\\'){
                 yyless(yyleng-1);
                 yymore();
                } else printf("<%s>\n", yytext);
               }

%%
```

3 The macro `REJECT` tells the lexical analyser to execute the rule that contains it, and then, before executing the next rule, restore the position of the input pointer to where it was before the current rule was executed.

This is used in situations when one token is a substring of another.

```
%%
she {printf("<she found>\n");REJECT;}
he  {printf("<he found>\n");}
%%
```

The effect of `REJECT` is not the same as `yyless(0)`. (Why?)