TUTORIAL 1

DATE: 12th January 2015

---

1. Consider the programs shown in the class, with some modifications:

```
#include <stdio.h> /* included system
                      header files */
/*main.c*/
void swap (); /* declaration */
int buf [2] = {34,56}; /* initialised global */
int main ()  /* definition main */
{
  swap ();
  printf("buf[0]= %d buf[1]= %d\n", buf[0], buf[1]);
  return 0;
}
-----------------------------------------------
/*swap.c*/
extern int buf []; /*declaration buf*/
#define one 1
int *bufp0 = &buf[0]; /* initialized global */
int *bufp1;   /* uninitialized global */

void swap ()   /* definition swap */
{
  int temp;    /* local */
  f();
  bufp1 = &buf[one];
  temp = *bufp0;
  *bufp0 = *bufp1;
  *bufp1 = temp;
}
-----------------------------------------------
/*other.c*/
int buf[2];
void f()
{
   buf[0] = 3;
   buf[1] = 4;
}
```

First, run `readelf` with the `-s` switch to see the symboltable of `main.o`. The fields of the output mean the following:

Num = The symbol number
Value = The address of the Symbol
Size = The size of the symbol
Type = symbol type:
FUNC = Function,
OBJECT= Data object,
FILE (source file name),
SECTION = memory section,
NOTYPE = untyped absolute symbol or undefined
Bind = GLOBAL binding means the symbol is visible outside the file. LOCAL binding is visible only in the file. WEAK is like global, the symbol can be overridden.
Vis = Symbols can be default, protected, hidden or internal.
Ndx = The section number the symbol is in. ABS means absolute: not adjusted to any section address's relocation
Name = symbol name

Do a similar exercise for `swap.o` and `other.o`

Next find the relocation information for `main.o` by running `readelf` with the `-r` switch. Match it against the code obtained by doing `objdump` with the `-d` switch.

Now use the `readelf` and the `objdump` switches once again to find out how the relocatable symbols have been relocated in the executable `a.out`. Produce the excutable twice – once with and once without the switch `-static`.

2. Kernighan's C book says – The characters /* introduce a comment, which terminates with the characters */. Comments do not nest.

   Is the description unambiguous? In any case extend the lex script example1 to introduce comments.

3. Write a lex script to find tokens in the 386-assembly produced by gcc. You can limit yourself to the tokens in the following program:

```
main:
pushl %ebp
movl %esp, %ebp
andl $-16, %esp
subl $16, %esp
call swap
movl buf+4, %edx
movl buf, %eax
movl %edx, 8(%esp)
movl %eax, 4(%esp)
movl $.LC0, (%esp)
call printf
```

```
movl $0, %eax
leave
ret
```

4. Consider a language with the following tokens:

   begin - representing the lexeme `begin`
   integer - Examples: `0, -5, 250`
   identifier - Examples: `a, A1, max`

   Construct the DFA for these tokens using the direct construction method.

5. Consider the following Lex-like specification of three distinct tokens:

$(aba)+$	Token 1
$ab*a$	Token 2
$a(a\|b)$	Token 3

   (a) Draw an appropriate syntax tree for the specification which can be directly converted to a DFA. Be careful, because the resulting DFA has to distinguish between lexemes corresponding to the three tokens.

   (b) Number the leaves of the syntax tree from left to right. Now construct the DFA labeling each state with sets of positions.

   (c) Indicate the token(s) found against each accepting state. Are there any clashes? If yes, why?

6. The time taken to do lexical analysis, it is generally thought, is linear in the length of the input string. The question below shows that this is not always the case.

   Consider the following Lex-like description of two tokens:
   abc – Token 1
   $(abc)^*d$ – Token 2

   (a) Draw the DFA corresponding to this regular expression. Indicate against each final state, the token recognised by the state.

   (b) Give an input string whose complete tokenization (by complete tokenization we mean extraction of all the tokens in the given string) would take time that is quadratic in its length.

7. (a) Comments in C consist of any sequence of characters enclosed between `/*` and `*/`. The enclosed characters should not contain the sequence `*/`. As examples, `/* this is a comment */`, `/**/` and `/*/*/` are valid comments. Example of non-comments are `/*/` and `/*/*/*/`. Write a regular expression for C comments.

   (b) Do some experiments to figure out how string literals are represented in C. In particular, find out what happens if you want to represent a " in a string. Also find out how you can represent strings that spread over multiple lines.

Now write a regular expression for strings in flexc++.

For both the problems above, you must think of arguments to show why your regular expressions are correct.

8. Study the following programs and their corresponding code generated. For each do the following:

   (a) Annotate fragments of target code with the source statemets that they correspond to.

   (b) Annotate each local variable and parameter with its (relative address)

   (a) C Code for a program with a while in a file test1.c:

```
int main()
{
  int a=1,b=1;
  while(a<=10)
    {
      b=b*a;
      a++;
    }
  return b;
 }
```

```
.file "test1.c"
.text
.globl main
.type main, @function
main:
pushl %ebp
movl %esp, %ebp
subl $16, %esp
movl $1, -8(%ebp)
movl $1, -4(%ebp)
jmp .L2
.L3:
movl -4(%ebp), %eax
imull -8(%ebp), %eax
movl %eax, -4(%ebp)
addl $1, -8(%ebp)
.L2:
cmpl $10, -8(%ebp)
jle .L3
movl -4(%ebp), %eax
leave
ret
```

   (b) C Program using structures in a file test2.c

```
struct data{
int sum;
int b[5];
};

int main()
{
  struct data rec1;
  rec1.sum=0;
  rec1.b[0]=2;
  rec1.sum=rec1.sum+rec1.b[0];
  return rec1.sum;
}
```

```
.file "test2.c"
.text
.globl main
.type main, @function
main:
pushl %ebp
movl %esp, %ebp
subl $32, %esp
movl $0, -24(%ebp)
movl $2, -20(%ebp)
movl -24(%ebp), %edx
movl -20(%ebp), %eax
addl %edx, %eax
movl %eax, -24(%ebp)
movl -24(%ebp), %eax
leave
ret
```