CSE216
Foundations of Computer Science

Instructor: Zhoulai Fu

State University of New York, Korea

Plan
(no change; reminder)

- ~~11.14 Tu:~~ Ocaml -
- ~~11.16~~ Ocaml ungraded homework; Ocaml in REC
- ~~11.21 Tu~~ Review Midterm 2 **-**
- ~~11.23~~ **Midterm 2** no homework; C in REC
- 11.28 Tu C
- 11.30 C C; ungraded homework; Final Review in REC
- 12.05 Tu Final Review -
- 12.07 Final Review - Final Review
- 12.11 Tu Final -

Agenda

- Midterm 2

- C

Problem 1. Lambda Calculus (Points = 20. No partial points.)

1. What is the beta-normal form of the lambda expression: (λx.λy.yx)(λz.λw.zw)?

2. What is the beta-normal form of the lambda expression: (λx.λy.xy)(λz.λw.zw)?

3. What is the beta-normal form of the lambda expression: (λx.λy.yxy)(λz.λw.zw)?

4. What is the beta-normal form of the lambda expression: ((λx.λy.y) y) (λx.x a) ?

Problem 2. Ocaml Types (Points = 20. No partial points.)

Give the type of the following OCaml expressions:

1. `["hello"; "world"]`
2. `[[1];[1]]`
3. `(1,"bar")`
4. `[(1,2,"foo");(3,4,"bar")]`
5. `let f x y z = x+y+z in f 1 2`

Give the type of the function defined below:

1. `let f x = x *. 3.14`
2. `let f (x, y) = x + y`
3. `let f x y = if y then x else x`
4. `let f x y z = if x then y else y`
5. `let rec f x = if (x = 0) then 1 else 1 + f(x - 1)`

Problem 3. Ocaml Functions (Points = 40. Partial points allowed.)

1. Write an Ocaml function `lucas: int -> int` that calculates the nth number in the sequence of *Lucas numbers*. Lucas numbers begin with 2 and 1, and each subsequent number is the sum of the two immediately preceding it. Namely, 2, 1, 3, 4, 7, 11, 18, 29, 47, 76, 123, 199 ... Consider the sequence starts from 0-th number. Assume the input is equal to or larger than 0.

2. Implement Ocaml function `rev: 'a list -> 'a list` that computes a reverse of a list. Do not use `List.rev` or anything involving `fold`.

```
# rev [5;6;7] ;;
- : int list = [7; 6; 5]
```

3. Write a function `last : 'a list -> 'a option` that returns the last element of a list.

```
# last ["a" ; "b" ; "c" ; "d"];;
- : string option = Some "d"
# last [];;
- : 'a option = None
```

4. Write an Ocaml function `compress: 'a list -> 'a list` that eliminates consecutive duplicates of list elements.

```
# compress ["a"; "a"; "a"; "a"; "b"; "c"; "c"; "a"; "a"; "d"; "e"; "e";
"e"; "e"];;
- : string list = ["a"; "b"; "c"; "a"; "d"; "e"]
```

Problem 4. Higher Order Functions (Points = 20. Partial points allowed.)

A common pattern of the functions that accumulate something over a list can be captured by the higher-order function `fold_right`:

```
# let rec fold_right f lst acc =
    match lst with
    | [] -> acc
    | hd :: tl -> f hd (fold_right f tl acc)

val fold_right : ('a -> 'b -> 'b) -> 'a list -> 'b -> 'b = <fun>
```

1. Write the function `length: 'a list -> int` that calculates the length of a list using `fold_right`.

1. Write the function `filter: ('a -> bool) -> 'a list -> 'a list` that takes a predicate (namely, a function that returns true or false) and a list, and returns a new list containing only those elements of the original list for which the predicate returns true. Implement the function `filter` using `fold_right` above.

```
# filter (fun x -> x mod 2 = 0) [1;2;3;4] ;;
- : int list = [2; 4]
```

Defining Constants

Using #define preprocessor.

```c
#include <stdio.h>

#define LENGTH 10
#define WIDTH  5
#define NEWLINE '\n'

int main()
{

   int area;

   area = LENGTH * WIDTH;
   printf("value of area : %d", area);
   printf("%c", NEWLINE);

   return 0;
}
```

Using const keyword.

```c
#include <stdio.h>

int main()
{
   const int  LENGTH = 10;
   const int  WIDTH  = 5;
   const char NEWLINE = '\n';
   int area;

   area = LENGTH * WIDTH;
   printf("value of area : %d", area);
   printf("%c", NEWLINE);

   return 0;
}
```

#define LENGTH vs const int LENGTH ?

const int LENGTH vs extern int LENGTH vs. static int LENGTH ?

Storage classes

- auto:
- register:
- static:
- extern:

Storage classes

- auto: Variable allocated when the block in which they are defined is entered, and deallocated when it is exited.
- register: local variables that should be stored in a register instead of RAM.
- static: existence during the life-time
- extern: give a reference of a global variable that is visible to ALL the program files.

Auto

- auto: Variable allocated when the block in which they are defined is entered, and deallocated when it is exited.

```
void function() {
    auto int x = 0; // Here, `auto` is redundant because `x` is a local variable
    // ...
}
```

Register

- register: local variables that should be stored in a register instead of RAM.

```c
#include <stdio.h>

int main() {
    register int counter;
    for(counter=0; counter<100000; counter++) {
        printf("%d\n", counter);
    }
    return 0;
}
```

Static

- static: existence during the life-time

```c
#include <stdio.h>

void increment() {
    static int count = 0;
    count++;
    printf("%d\n", count);
}

int main() {
    increment();  // prints 1
    increment();  // prints 2
    increment();  // prints 3
    return 0;
}
```

extern

- extern: give a reference of a global variable that is visible to ALL the program files.

First File: main.c

```c
#include <stdio.h>

int count ;
extern void write_extern();

main()
{
    write_extern();
}
```

Second File: write.c

```c
#include <stdio.h>

extern int count;

void write_extern(void)
{
    count = 5;
    printf("count is %d\n", count);
}
```

a missing DEFINITION won't have issues during Compilation but will result in a LINKAGE error;

the extern keyword in the second file PROMISES that int count will be defined (i.e., declared & malloc'd) somewhere else.

BUT, if int count doesn't exist anywhere else, that promise is broken.

also, no more than one definition:

X extern int count;
:
int count = ... ;

do not need to be in the same directory, BUT the same compilation unit.

Lab exercise 1:
Implementing a Caesar Cipher in C

Introduction

- n this lab exercise, you will be implementing a simple Caesar cipher in the C programming language. A Caesar cipher is a type of substitution cipher where each character in the plaintext is 'shifted' a certain number of places down the alphabet. For example, with a shift of 3, A would be replaced by D, B would become E, and so on.

Background

- **Character arrays in C:** In C, strings are typically represented as arrays of characters. For example, the string "HELLO" can be declared as **char str[] = "HELLO";**. Note that all strings in C are null-terminated, which means they end with a special character '\0'.

- **Character pointers in C (char*)**: A character pointer in C can also be used to represent a string. It can point to the first character of a string, and the string is assumed to continue until a null character is encountered. For example, **char* str = "HELLO";**.

- **String manipulation in C**: C provides several functions for manipulating strings, such as **strcpy** for copying strings and **strlen** for finding the length of a string. However, in this exercise, you will be manipulating strings directly.

Problem Statement

```c
int main() {
    char str[] = "KENNEDY";
    caesarCipher(str);
    printf("%s\n", str); // Should print "NHQQHGB"
    return 0;
}
```

- Write a C function **void caesarCipher(char* str)** that performs a Caesar cipher on an input string. The string will consist of capital letters only, and the cipher should shift each letter 3 places to the right in the alphabet, wrapping around to the beginning of the alphabet if necessary.

- For example, the input string "KENNEDY" should produce the output "NHQQHGB".

Lab exercise 2:
Sentence Title Case Verification in C

Problem

- Your task is to write a C function that checks whether a sentence is in 'Title Case'. In other words, the function should return true if each word in the sentence starts with a capital letter and continues with lowercase letters. Here are the specific requirements:

    - The function should take a single argument - a string, representing the sentence to check. This string consists only of letters and blank spaces.

    - The function should return a boolean value (in C, typically represented as an int with 0 for false and non-zero for true).

    - The function should return true if and only if each word in the sentence starts with a capital letter and continues with lowercase letters. Otherwise, it should return false.

- Write the function as described above. Test your function with several test sentences to ensure that it works correctly.

Background

- In C, strings are represented as arrays of characters. You can use array indexing to access individual characters in a string, similar to how you'd access elements in an array. For example, sentence[0] would give you the first character in the string sentence.

- C provides functions to manipulate and check characters. You might find the following functions from the ctype.h library useful:

  - **isupper(int c)** checks if the given character is uppercase.

  - **islower(int c)** checks if the given character is lowercase.

  - **isspace(int c)** checks if the given character is a whitespace character.

- Reminder: A string in C is null-terminated, meaning it ends with the special null character '\0'. You can use this fact to iterate through the string.