Федеральное государственное автономное образовательное учреждение высшего образования
Университет ИТМО

Кафедра Вычислительной Техники

Дисциплина: Низкоуровневое программирование

Лабораторная работа №5

Выполнил: **Доморацкий Эридан Алексеевич**

Группа: **P33113**

Преподаватель: **Логинов Иван Павлович**

2020г

Задание

11.7 Assignment: Higher-Order Functions and Lists

11.7.1 Common Higher-Order Functions

In this assignment, we are going to implement several higher-order functions on linked lists, which should be familiar to those used to functional programming paradigm.

These functions are known under the names `foreach`, `map`, `map_mut`, and `foldl`.

- `foreach` accepts a pointer to the list start and a function (which returns void and accepts an `int`). It launches the function on each element of the list.
- `map` accepts a function *f* and a list. It returns a new list containing the results of the *f* applied to all elements of the source list. The source list is not affected.

For example, $f(x) = x + 1$ will map the list (1, 2, 3) into (2, 3, 4).

- `map_mut` does the same but changes the source list.
- `foldl` is a bit more complicated. It accepts:
  - The accumulator starting value.
  - A function $f(x, a)$.
  - A list of elements.

It returns a value of the same type as the accumulator, computed in the following way:

1. We launch *f* on accumulator and the first element of the list. The result is the new accumulator value a'.
2. We launch *f* on a' and the second element in list. The result is again the new accumulator value a''.
3. We repeat the process until the list is consumed. In the end the final accumulator value is the final result.

For example, let's take $f(x, a) = x * a$. By launching `foldl` with the accumulator value 1 and this function we will compute the product of all elements in the list.

- `iterate` accepts the initial value *s*, list length *n*, and function *f*. It then generates a list of length *n* as follows:

$$\left[s, f(s), f\left(f(s)\right), f\left(f\left(f(s)\right)\right)\dots\right]$$

The functions described above are called **higher-order functions**, because they do accept other functions as arguments. Another example of such a function is the array sorting function `qsort`.

```
void qsort( void *base,
            size_t nmemb,
            size_t size,
            int (*compar)(const void *, const void *));
```

It accepts the array starting address base, elements count `nmemb`, size of individual elements `size`, and the comparator function `compar`. This function is the decision maker which tells which one of the given elements should be closer to the beginning of the array.

11.7.2 Assignment

The input contains an arbitrary number of integers.

1. Save these integers in a linked list.
2. Transfer all functions written in previous assignment into separate `.h` and `c` files. Do not forget to put an include guard!
3. Implement `foreach`; using it, output the initial list to `stdout` twice: the first time, separate elements with spaces, the second time output each element on the new line.
4. Implement `map`; using it, output the squares and the cubes of the numbers from list.
5. Implement `foldl`; using it, output the sum and the minimal and maximal element in the list.
6. Implement `map_mut`; using it, output the modules of the input numbers.
7. Implement `iterate;` using it, create and output the list of the powers of two (first 10 values: 1, 2, 4, 8, …).
8. Implement a function `bool save(struct list* lst, const char* filename);`, which will write all elements of the list into a text file `filename`. It should return `true` in case the write is successful, `false` otherwise.
9. Implement a function `bool load(struct list** lst, const char* filename);`, which will read all integers from a text file filename and write the saved list into `*lst`. It should return `true` in case the write is successful, `false` otherwise.
10. Save the list into a text file and load it back using the two functions above. Verify that the save and load are correct.
11. Implement a function `bool serialize(struct list* lst, const char* filename);`, which will write all elements of the list into a *binary* file filename. It should return `true` in case the write is successful, `false` otherwise.
12. Implement a function `bool deserialize(struct list** lst, const char* filename);`, which will read all integers from a *binary* file filename and write the saved list into *lst. It should return `true` in case the write is successful, `false` otherwise.
13. Serialize the list into a binary file and load it back using two functions above. Verify that the serialization and deserialization are correct.
14. Free all allocated memory.

You will have to learn to use

- Function pointers.
- `limits.h` and constants from it. For example, in order to find the minimal element in an array, you have to use `foldl` with the maximal possible `int` value as an accumulator and a function that returns a minimum of two elements.
- The `static` keyword for functions that you only want to use in one module.

You are guaranteed, that

- Input stream contains only integer numbers separated by whitespace characters.
- All numbers from input can be contained as `int`.

It is probably wise to write a separate function to read a list from `FILE`.

The solution takes about 150 lines of code, not counting the functions, defined in the previous assignment.

Выполнение

```c
// linked_list.h

#pragma once

/* [Int] */
struct list;
typedef struct list list_t;

/* list_create :: Int -> [Int] */
list_t * list_create(int value);

/* list_node_at :: [Int] -> Int -> [Int] */
list_t * list_node_at(list_t * list, unsigned int index);

/* list_free :: [Int] -> IO () */
void list_free(list_t * list);

/* list_length :: [Int] -> Int */
unsigned int list_length(const list_t * list);

/* list_get :: [Int] -> Int -> Int */
int list_get(const list_t * list, unsigned int index);

/* list_sum :: [Int] -> Int */
long list_sum(const list_t * list);

/* list_add_front :: Int -> [Int] -> [Int] */
void list_add_front(int value, list_t ** list);

/* list_add_back :: Int -> [Int] -> [Int] */
void list_add_back(int value, list_t ** list);

/* list_foreach :: [Int] -> (Int -> IO ()) -> IO () */
void list_foreach(const list_t * list, void (* f)(int));

/* list_map :: (Int -> Int) -> [Int] -> [Int] */
list_t * list_map(int (* f)(int), const list_t * list);

/* list_map_mut :: (Int -> Int) -> State [Int] */
void list_map_mut(int (* f)(int), list_t * list);

/* list_foldl :: Int -> (Int -> Int -> Int) -> [Int] -> Int */
void * list_foldl(void * acc, void * (* f)(int, void *), const list_t * list);

/* iterate :: Int -> Int -> (Int -> Int) -> [Int] */
list_t * list_iterate(int value, unsigned int length, int (* f)(int));
```

```c
// linked_list.c

#include "linked_list.h"

#include <stdlib.h>

#define __list_new (malloc(sizeof(list_t)))
#define __list_foreach(_lst) for (; _lst; _lst = _lst->next)

struct list {
   int value;
   list_t * next;
};

list_t * list_create(int value) {
   list_t * list = __list_new;

   list->value = value;
   list->next = NULL;
   return list;
}

list_t * list_node_at(list_t * list, unsigned int index) {
   while (index-- > 0) {
       if (list == NULL) {
           return NULL;
       }

       list = list->next;
   }

   return list;
}

void list_free(list_t * list) {
   list_t * prev = list;

   while ((list = list->next)) {
       free(prev);

       prev = list;
   }

   free(prev);
}

unsigned int list_length(const list_t * list) {
   unsigned int length = 0;

   __list_foreach (list) {
       ++length;
   }

   return length;
}

int list_get(const list_t * list, unsigned int index) {
   const list_t * node = list_node_at((list_t *) list, index);

   if (node == NULL) {
       return 0;
   }

   return node->value;
}

long list_sum(const list_t * list) {
   long sum = 0;

   __list_foreach (list) {
       sum += list->value;
   }

   return sum;
}
```

```c
void list_add_front(int value, list_t ** list) {
    list_t * new_list;

    if (!list) {
        return;
    }

    new_list = __list_new;
    new_list->value = value;
    new_list->next = *list;
    *list = new_list;
}

void list_add_back(int value, list_t ** list) {
    list_t * last = NULL;
    list_t * current;

    if (!list) {
        return;
    }

    current = *list;
    __list_foreach (current) {
        last = current;
    }

    if (!last) {
        list_add_front(value, list);
        return;
    }

    last->next = list_create(value);
}

void list_foreach(const list_t * list, void (* f)(int)) {
    __list_foreach (list) {
        f(list->value);
    }
}

list_t * list_map(int (* f)(int), const list_t * list) {
    list_t * prev_node = NULL;
    list_t * new_list = NULL;
    list_t * next_node;

    __list_foreach (list) {
        next_node = list_create(f(list->value));

        if (prev_node) {
            prev_node->next = next_node;
        } else {
            new_list = next_node;
        }

        prev_node = next_node;
    }

    return new_list;
}

void list_map_mut(int (* f)(int), list_t * list) {
    __list_foreach (list) {
        list->value = f(list->value);
    }
}

void * list_foldl(void * acc, void * (* f)(int, void *), const list_t * list) {
    __list_foreach (list) {
        acc = f(list->value, acc);
    }

    return acc;
}
```

```c
list_t * list_iterate(int value, unsigned int length, int (* f)(int)) {
   list_t * prev_node;
   list_t * next_node;
   list_t * new_list;

   if (!length) {
       return NULL;
   }

   new_list = list_create(value);
   prev_node = new_list;

   while (length-- > 0) {
       value = f(value);

       next_node = list_create(value);
       prev_node->next = next_node;
       prev_node = next_node;
   }

   return new_list;
}

// main.c

#include <stdbool.h>
#include <stdio.h>

#include "linked_list.h"

struct sum_min_max {
   long sum;
   int min;
   int max;
};

void print_int_spaced(int a) {
   printf("%d ", a);
}

void print_int_lined(int a) {
   printf("%d\n", a);
}

int square_int(int a) {
   return a * a;
}

int cube_int(int a) {
   return a * a * a;
}

int abs_int(int a) {
   if (a < 0) {
       return -a;
   }

   return a;
}

bool save(list_t * lst, const char * filename) {
   FILE * f = fopen(filename, "wb");

   if (f == NULL) {
       return false;
   }

   for (; lst; lst = list_node_at(lst, 1)) {
       if (fprintf(f, "%d ", list_get(lst, 0)) <= 0) {
           return false;
       }
   }

   return !fclose(f);
}
```

```c
bool load(list_t ** lst, const char * filename) {
   FILE * f = fopen(filename, "rb");
   int value;

   if (f == NULL) {
       return false;
   }

   while (fscanf(f, "%d", &value) != 1) {
       list_add_back(value, lst);
   }

   if (ferror(f)) {
       return false;
   }

   return !fclose(f);
}

void * update_sum_min_max(int a, void * fold_data) {
   struct sum_min_max * smm = fold_data;

   smm->sum += a;

   if (smm->min > a) {
       smm->min = a;
   }

   if (smm->max < a) {
       smm->max = a;
   }

   return smm;
}

int dup_int(int a) {
   return 2 * a;
}

int main() {
   struct sum_min_max smm = { 0, 2147483647, -2147483648 };
   list_t * list = NULL;
   list_t * squares;
   list_t * cubes;
   list_t * bits;
   int value;

   while (scanf("%d", &value) == 1) {
       list_add_back(value, &list);
   }

   list_foreach(list, print_int_spaced);
   putchar('\n');

   list_foreach(list, print_int_lined);

   squares = list_map(square_int, list);
   list_foreach(squares, print_int_spaced);
   list_free(squares);
   putchar('\n');

   cubes = list_map(cube_int, list);
   list_foreach(cubes, print_int_spaced);
   list_free(cubes);
   putchar('\n');

   list_foldl(&smm, update_sum_min_max, list);
   printf("Sum: %ld. Min: %d. Max: %d.\n", smm.sum, smm.min, smm.max);

   list_map_mut(abs_int, list);
   list_foreach(list, print_int_spaced);
   putchar('\n');

   bits = list_iterate(1, 10, dup_int);
   list_foreach(bits, print_int_spaced);
   list_free(bits);
   putchar('\n');
```

```
    list_free(list);
    return 0;
}
```

Вывод

В ходе выполнения данной лабораторной работы были реализованы функции высшего порядка для работы со списком и работа с файловой системой на языке C.