複習程式設計一

10193 - moocHW6a, 10194 - moocHW6b

這些題目主要是透過 list 來複習 C struct，以及指標與記憶體。
除了標準的 head, tail, cons 這些功能之外，我們還要做 zip, unzip, take, drop, split_at。

List 的實作可以有很多形式，範例程式只是其中一種可能，不是最好的實作方式，只是為了方便練習與理解。程式範例中對於 list 的操作，除了取得 head, tail 這個動作是基於原有的 list 之外，其餘的操作都會長出一份新的 list 複本，所以在處理記憶體的時候要注意，要盡量避免 memory leak。

C 程式如果牽涉到動態處理記憶體，程式很容易出錯，常見問題例如指標指到已經不存在的記憶體位址，把不該 free 的記憶體 free，或是取得了記憶體、但是卻沒有用指標持續記住那塊記憶體的位址，最後變成 memory leak。

題目的程式碼頗長，用到不少指標和遞迴呼叫，要先有耐心看過一遍，然後再想想看如何完成`zip` `unzip` `take` `drop` `split_at` 這幾個函數。

範例程式碼有個地方需要修正，

```
strncpy(aptr->str, str, STR_LEN);
aptr->str[STR_LEN] = '\0';
```

`strncpy`的運作方式是，假如`str`的長度超過`STR_LEN`，只會把`str`的前`STR_LEN`個字元，複製到`aptr->str`，這種情況`strncpy`並不會幫`aptr->str`補上字串結束符號`'\0'`，必須自己補。

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <ctype.h>

#define STR_EXPAND(tok) #tok
#define STR(tok) STR_EXPAND(tok)
#define STR_LEN 20

/* incomplete struct Atom */
struct Atom;

/* a pair of atom */
typedef struct Pair {
    struct Atom *left;
    struct Atom *right;
} Pair;
```

```c
/* a list of atoms */
typedef struct List {
    struct Atom *data;
    struct List *next;
} List;

/* an atom can be a string, an int, a pair of atoms, or a list */
typedef struct Atom {
    char str[STR_LEN+1];
    int val;
    Pair *pair;
    List *lst;
    int dtype; /* 0 for string, 1 for int, 2 for pair, 3 for list */
} Atom;
```

```c
/* functions for creating an atom */
Atom* atom_str(char *str);
Atom* atom_val(int val);

Atom* copy_atom(Atom *aptr);
void print_atom(Atom *aptr);
void free_atom(Atom *atpr);

/* convert an atom to a single-atom list */
List* atom_to_list(Atom *a);

/* this function is important */
/* useful for creating a new copy of an existing list */
List* cons(Atom *a, List *b);

void print_list(List *lptr);
void free_list(List *lptr);
int length(List *lptr);
List* read_list_helper(List *l, int n);
List* read_list(void);

Atom* head(List *lptr);
List* tail(List *lptr);

List* zip(List *lptr1, List *lptr2);
Pair* unzip(List *lptr);

List* take(int n, List *l);
List* drop(int n, List *l);

Pair* pair_list(List *lptr1, List *lptr2);
Pair* split_at(int n, List *l);

void print_pair(Pair* p);
```

```
void free_pair(Pair* p);

int main(void)
{
    List *s1, *s2, *s3;
    Pair *p;
    int N, i;

    s1 = read_list();
    s2 = read_list();

    scanf("%d", &N);
    for (i=0; i<N; i++) {
        s3 = zip(s1, s2);
        free_list(s1);
        s1 = s2;
        s2 = s3;
    }

    print_list(s3);
    printf("\n");
    p = unzip(s3);
    print_pair(p);
    printf("\n");
    return 0;
}
```

```
List* read_list_helper(List *l, int n)
{
    Atom *a;
    int x;
    char str[STR_LEN+1];
    List *l1, *l2;

    if (n==0) {
        return l;
    } else {
        if (scanf("%d", &x)==1) {
            a = atom_val(x);
        } else {
            scanf("%"STR(STR_LEN)"s", str);
            while(!isspace(getchar()));
            a = atom_str(str);
        }
        l1 = read_list_helper(l, n-1);
        l2 = cons(a, l1);
        free_list(l1);
        return l2;
    }
}
```

```
List* read_list(void)
{
    int ndata;

    scanf("%d", &ndata);

    return read_list_helper(NULL, ndata);
}
/* Given a string, create a new atom */
Atom* atom_str(char *str)
{
    Atom *aptr = (Atom*) malloc(sizeof(Atom));
    if (str==NULL) {
        aptr->str[0] = '\0';
    } else {
        strncpy(aptr->str, str, STR_LEN);
        aptr->str[STR_LEN] = '\0';
    }
    aptr->val = 0;
    aptr->pair = NULL;
    aptr->lst = NULL;
    aptr->dtype = 0;
    return aptr;
}

/* Given a value, create a new atom */
Atom* atom_val(int val)
{
    Atom *aptr = (Atom*) malloc(sizeof(Atom));
    aptr->str[0] = '\0';
    aptr->val = val;
    aptr->pair = NULL;
    aptr->lst = NULL;
    aptr->dtype = 1;
    return aptr;
}
```

```
Atom* copy_atom(Atom *aptr)
{
    Atom *aptr_new;

    if (aptr==NULL) return NULL;

    aptr_new = (Atom*) malloc(sizeof(Atom));
    if (aptr_new==NULL) return NULL;

    aptr_new->dtype = aptr->dtype;

    if (aptr->dtype == 0) {
        strncpy(aptr_new->str, aptr->str, STR_LEN);
        aptr_new->str[STR_LEN] = '\0';
```

```
    } else if (aptr->dtype == 1) {
        aptr_new->val = aptr->val;
    } else if (aptr->dtype == 2) {
        if (aptr->pair == NULL) {
           aptr_new->pair = NULL;
        } else {
            aptr_new->pair = (Pair *) malloc(sizeof(Pair));
            aptr_new->pair->left = copy_atom(aptr->pair->left);
            aptr_new->pair->right = copy_atom(aptr->pair->right);
        }
    } else if (aptr->dtype == 3) {
        if (aptr->lst==NULL) {
            aptr_new->lst = NULL;
        } else {
            aptr_new->lst = cons(head(aptr->lst), tail(aptr->lst));
        }
    }
    return aptr_new;
}
```

```
void print_atom(Atom *aptr)
{
    if (aptr==NULL) {
        printf("Empty");
        return;
    }
    switch (aptr->dtype) {
    case 0:
        printf("\"%s\"", aptr->str);
        break;
    case 1:
        printf("%d", aptr->val);
        break;
    case 2:
        print_pair(aptr->pair);
        break;
    case 3:
        print_list(aptr->lst);
        break;
    default:
        printf("Undefined.");
    }

}
/* Given an atom, create a list containing the atom. */
List* atom_to_list(Atom *a)
{
    List *b;
    b = (List*) malloc(sizeof(List));
```

```
    b->next = NULL;
    b->data = copy_atom(a);
    return b;
}

/* create a new list and add the atom to the head */
List* cons(Atom *a, List *b)
{
    List *c;

    c = atom_to_list(a);
    if (b!=NULL) {
        c->next = cons(head(b), tail(b));
    }

    return c;
}

void print_list(List *lptr)
{
    printf("[");
    while (lptr!=NULL) {
        print_atom(lptr->data);
        if (lptr->next != NULL)
            printf(",");
        lptr = lptr->next;
    }
    printf("]");
}

int len_helper(List *lptr, int len)
{
    if (lptr==NULL) return len;
    else return len_helper(lptr->next, len+1);
}
int length(List *lptr)
{
    return len_helper(lptr, 0);
}

void free_atom(Atom *aptr)
{
    if (aptr != NULL) {
        if (aptr->dtype==2) {
            free_pair(aptr->pair);
        } else if (aptr->dtype==3) {
            free_list(aptr->lst);
        }
        free(aptr);
    }
}
```

```
void free_list(List *lptr)
{
    List *p;
    if (lptr!=NULL) {
        p = tail(lptr);
        if (head(lptr)!=NULL) {
            free_atom(head(lptr));
        }
        free_list(p);
    }
}

Atom* head(List *lptr)
{
    return lptr->data;
}

List* tail(List *lptr)
{
    return (lptr==NULL) ? NULL : lptr->next;
}

/* create a new list combining a string list and a value list
If the two input lists have different lengths, the new list will be
of the same length as the shorter one.
*/

List* zip(List *lptr1, List *lptr2)
{
    Atom a;
    List *l1, *l2;

    if ( ??? ) { //lptr1 或 lptr2 兩者有一個是 NULL
        return NULL;
    } else {
        a.pair = ???; // malloc 取得一塊 Pair 空間
        a.pair->left = ???; // 利用  copy_atom  複製 lptr1 的開頭第一個 atom
        a.pair->right = ???; // 利用  copy_atom  複製 lptr2 的開頭第一個 atom
        a.dtype = 2;
        l1 = ??? // 利用 zip 處理剩下的 lists  (提示: 用遞迴)
        l2 = cons(&a, l1); // 利用 cons 造出新的 list
        free_pair(a.pair);
        free_list(l1);
        return l2;
    }
}
```

```
Pair* unzip(List *lptr)
{
    Atom *h;
    Pair *p, *newp;
```

```
    List *l1, *l2;
    if (lptr==NULL) {
        newp = pair_list(NULL, NULL);
        return newp;
    }

    h = ???; // // 取出 lptr 的開頭第一個 atom
    p = ???; // 利用 unzip 對剩下的 lptr 遞迴繼續處理
    l1 = ???; // 兩個 lists 分別來自兩個 cons
    l2 = ???; // 一個 cons 組合左邊的 list 另一個組合右邊的 list
    newp = ???; // // 利用 pair_list 從兩個 lists 產生新的 pair
    free_list(l1);
    free_list(l2);
    free_pair(p);
    return newp;
}
```

```
List* take(int n, List *lptr)
{
    /*
    判斷 n 的值決定是否繼續
    利用 cons 搭配 take 的遞迴呼叫  建構出回傳的 list
    */
}

List* drop(int n, List *lptr)
{
    /*
    判斷 n 的值決定是否繼續
    利用 cons 搭配 drop 的遞迴呼叫  建構出回傳的 list
    */

}

Pair* pair_list(List *lptr1, List *lptr2)
{
    Pair *p;
    Atom a;

    p = (Pair*)malloc(sizeof(Pair));
    a.dtype = 3;

    a.lst = lptr1;
    p->left = copy_atom(&a);

    a.lst = lptr2;
    p->right = copy_atom(&a);

    return p;
```

```
}
```

```
Pair* split_at(int n, List *lptr)
{
    Pair *p;
    Atom a;
    /* 參考 pair_list 的寫法
    利用 take 和 drop 還有 copy_atom
    做出 split_at
    */
    return p;
}
```

```
void print_pair(Pair *p)
{
    printf("(");
    print_atom(p->left);
    printf(",");
    print_atom(p->right);
    printf(")");
}

void free_pair(Pair* p)
{
    if (p!=NULL) {
        free_atom(p->left);
        free_atom(p->right);
        free(p);
    }
}
```

HW6a

題目描述

zip函數的作用是將兩個 lists 合併，對應位置的元素組成一個 pair，然後產生一個新的 list 包含所有的 pairs。也就是從兩個 lists 變成一個 list of pairs。
底下的表示法，方括號 [] 代表 list，圓括弧 () 代表 pair。 譬如第一個 list 是 [1,2,3] 而第二個是 ["a","b","c"]，經過 zip 作用之後變成
[(1,"a"),(2,"b"),(3,"c")]。 接著如果再用第二個 list 和第一次 zip 之後的結果，再做一次 zip，結果會變成 [("a",(1,"a")),("b",(2,"b")),("c",(3,"c"))] 然後第一次 zip 和第二次 zip 得到的兩個 lists，再做第三次 zip 就變成
[((1,"a"),("a",(1,"a"))),((2,"b"),("b",(2,"b"))),((3,"c"),("c",(3,"c")))] 上述的操作已經寫在範例程式中，只要將 zip 正確寫出，就能產生上述的執行結果。

輸入格式

輸入資料會有五行。
第一行是一個整數 M1，代表第一個 list 包含的資料筆數，第二行則是 M1 筆資料，每筆用空白隔開。
第三行是一個整數 M2，代表第二個 list 包含的資料筆數，第四行則是 M2 筆資料，每筆用空白隔開。
第五行則是一個整數 N。代表執行 zip 的次數。

輸出格式

輸出只有一行，列出經過 `zip` 之後的 list 內容。最後有換行。

輸入範例

4 aa bb cc dd 4 1 2 3 4 2

輸出範例

```
[(1,("aa",1)),(2,("bb",2)),(3,("cc",3)),(4,("dd",4))]
```

HW6b

題目描述

同樣是以範例程式，完成其中缺少的 `unzip` 函數。這個函數的作用是將 a list of pairs 變成 a pair of lists。底下的表示法，方括號 `[]` 代表 list，圓括弧 `()` 代表 pair。譬如原本的 list of pairs 是 `[(1,"a"),(2,"b"),(3,"c")]`，經過 `unzip` 之後變成 `([1,2,3],["a","b","c"])`。 如果有兩個 lists `[1,2,3]` 和 `["a","b","c"]`，先做第一題描述的兩次 `zip`，變成 `[("a",(1,"a")),("b",(2,"b")),("c",(3,"c"))]` 之後，如果再做一次 `unzip` 就變成 `(["a","b","c"],[(1,"a"),(2,"b"),(3,"c")])`
這一題要做的就是，輸入兩個 lists，經過 N 次 `zip`之後，再做一次 `unzip`，然後輸出最後產生的 pair。 上述的操作已經寫在範例程式中，只要將 `unzip` 正確寫出，就能產生上述的執行結果。

輸入格式

與第一題相同。輸入資料會有五行。
第一行是一個整數 M1，代表第一個 list 包含的資料筆數，第二行則是 M1 筆資料，每筆用空白隔開。
第三行是一個整數 M2，代表第二個 list 包含的資料筆數，第四行則是 M2 筆資料，每筆用空白隔開。
第五行則是一個整數 N。代表執行 zip 的次數。

輸出格式

輸出只有一行，列出經過 N 次 zip 之後再做一次 upzip 得到的 pair 內容。

輸入範例

4 aa bb cc dd 4 1 2 3 4 2

輸出範例

```
([1,2,3,4],[("aa",1),("bb",2),("cc",3),("dd",4)])
```

參考寫法

如果已經熟悉範例程式提供的各個函數的作用，例如copy_atom cons head tail pair_list，只要照著註解的提示，應該就能寫得出來。
開始寫之前可以先回頭確認一下相關struct的定義。 struct Atom;

```
/* a pair of atom */
typedef struct Pair {
    struct Atom *left;
    struct Atom *right;
} Pair;

/* a list of atoms */
typedef struct List {
    struct Atom *data;
    struct List *next;
} List;

/* an atom can be a string, an int, a pair of atoms, or a list */
typedef struct Atom {
    char str[STR_LEN+1];
    int val;
    Pair *pair;
    List *lst;
    int dtype; /* 0 for string, 1 for int, 2 for pair, 3 for list*/
} Atom;
```

zip和unzip的參考寫法如下

```
List* zip(List *lptr1, List *lptr2)
{
    Atom a;
    List *l1, *l2;

    if ( lptr1==NULL || lptr2==NULL ) { //lptr1 或 lptr2 兩者有一個是 NULL
        return NULL;
    } else {
        a.pair = (Pair*) malloc(sizeof(Pair)); // malloc 取得一塊 Pair 空間
        a.pair->left = copy_atom(head(lptr1)); // 利用  copy_atom  複製 lptr1 的開頭第一
個 atom
        a.pair->right = copy_atom(head(lptr2)); // 利用  copy_atom  複製 lptr2 的開頭第
一個 atom
        a.dtype = 2;
        l1 = zip(tail(lptr1),tail(lptr2));// 利用 zip 處理剩下的 lists  (提示: 用遞迴)
        l2 = cons(&a, l1); // 利用 cons 造出新的 list
        free_pair(a.pair);
        free_list(l1);
```

```
        return l2;
    }
}
```

```
Pair* unzip(List *lptr)
{
    Atom *h;
    Pair *p, *newp;
    List *l1, *l2;
    if (lptr==NULL) {
        newp = pair_list(NULL, NULL);
        return newp;
    }

    h = head(lptr); // // 取出 lptr 的開頭第一個 atom
    p = unzip(tail(lptr)); // 利用 unzip 對剩下的 lptr 遞迴繼續處理
    l1 = cons(h->pair->left, p->left->lst); // 兩個 lists 分別來自兩個 cons
    l2 = cons(h->pair->right, p->right->lst); // 一個 cons 組合左邊的 list 另一個組合右邊
的 list
    newp = pair_list(l1,l2); // // 利用 pair_list 從兩個 lists 產生新的 pair
    free_list(l1);
    free_list(l2);
    free_pair(p);
    return newp;
}
```