UNIVERSITÉ CÔTE D'AZUR

Examen de Programmation C

L'usage ou la possession de tout appareil électronique sera considéré comme de la fraude. Documents interdits.

Tous les codes devront être écrits en **Langage C ANSI**.

La notation est donnée à titre indicatif.

DATE : 19/05/2026 DURÉE : 2h00 NOTE : ...../20

NOM : .....

PRÉNOM : .....

NUMÉRO D'ÉTUDIANT : .....

Exercice 1 : Fonctions de bases (2,5 points)

1. Sans utiliser la fonction `strlen`, écrire une fonction `int ascii_length(char *ch)` qui calcule la longueur d'une chaîne de caractères.

```c
int ascii_length(char* c) {
    int i = 0;
    for (i=0; c[i] != '\0'; i++);
    return i;
}
```

2. Écrire une fonction `char *substring(char *ch, int start, int length)` avec 3 paramètres : une chaîne de caractères, un indice de début et une longueur et qui renvoie une nouvelle chaîne allouée sur le tas correspondant aux `length` caractères de `ch` à partir de l'indice `start`. En cas d'erreur, on renverra `NULL`.

Exemple : `substring("ABCD", 1, 3)` renvoie la chaîne de longueur 3 `"BCD"`. Alors que `substring("ABCD", 1, 4)` renverra `NULL`.

```c
char *substring(char *ch, int start, int length) {
    int n = longueur_ascii(ch);
    char *res;
    int i;
    if (start+length > n) return NULL;
    res = malloc(length+1);
    for (i=0; i<length; i++) {
        res[i] = ch[start+i];
    }
    res[i] = '\0';
    return res;
}
```

Une chaîne de caractères ASCII est une suite d'octets dont la valeur est comprise entre 0 et 128 (exclue). Un caractère ASCII est donc un octet dont le premier bit est à 0 (zéro). Ce codage ne permet de représenter que les symboles de bases. L'objectif de cet examen est d'écrire les fonctions permettant de gérer les chaînes codées en UTF-8 qui peuvent contenir des symboles accentués ou d'autres alphabets. Un caractère en UTF-8 peut être codé sur 1, 2, 3 ou 4 octets. Le premier octet d'un caractère UTF-8 peut commencer, selon la taille du caractère (en nombre d'octets), soit par `0`, `110`, `1110` ou `11110`. Un octet commençant par `10` correspond à un octet successeur.

- Préfixe 0 `0xxx xxxx` caractère ASCII sur un octet
- Préfixe 110 `110x xxxx` `10xxx xxxx` caractère sur deux octets, le suivant commençant par 10
- Préfixe 1110 `1110 xxxx` `10xxx xxxx` `10xxx xxxx` Trois octets, les deux autres commençant par 10
- Préfixe 11110 `1111 0xxx` `10xxx xxxx` `10xxx xxxx` `10xxx xxxx` Quatre octets

Exercice 2 : Calcul de la longueur d'une chaîne UTF-8 (2,5 points)

1. Écrire une fonction `int get_prefix(unsigned char c, int n)` qui renvoie le nombre construit à partir des `n` premiers bits de l'octet `c`. On attend une fonction d'une ligne avec un simple décalage de bits. Exemple : avec l'octet 215 qui se code `1101 0111`, l'appel de `get_prefix(215, 3)` renverra 6 (`110` en binaire).

```c
int get_prefix(unsigned char c, int n) {
    int d = 8 - n;
    return c>>d;
}
```

2. Écrire une fonction `int prefix(int n)` qui construit le nombre dont l'écriture en binaire est de longueur `n` avec `n-1` bits `1` suivit d'un unique bit `0`. On attend là aussi une fonction d'une ligne avec une simple expression binaire.

- `prefix(1)` doit renvoyer 0 (`0` en binaire)
- `prefix(2)` doit renvoyer 2 (`10` en binaire)
- `prefix(3)` doit renvoyer 6 (`110` en binaire)
- `prefix(4)` doit renvoyer 14 (`1110` en binaire)
- `prefix(5)` doit renvoyer 30 (`11110` en binaire)

```c
int prefix(int n) {
    return (1<<n) - 2; /* (1<<n) - 1 vaut en binaire 11..1 avec n fois le chiffre 1 */
}
```

3. En utilisant les deux fonctions précédentes, écrire une fonction `int prefix_p(unsigned char c, int n)` qui renvoie l'équivalent C ANSI du booléen True si l'octet `c` commence par les bits du préfixe associé à `n`. Par exemple, `prefix_p(c, 3)` vérifie si c commence par `110` (cf question précédente).

```c
int prefix_p(unsigned char c, int n) {
    return  get_prefix(c, n) == prefix(n);
}
```

4. On peut maintenant calculer la véritable longueur d'une chaîne et non simplement le nombre d'octets comme le faisait la fonction `ascii_length`. Écrire la fonction `int utf8_length(char *ch)` qui compte le nombre de caractères Unicode de la chaîne. Pour cela, il suffit de compter le nombre d'octets qui ne sont pas successeur (octet successeur : octet commençant par le préfixe binaire de taille 2 : 10). On aura ainsi `utf8_length("été")` qui vaudra 3 (codé sur 5 octets) et `utf8_length("ワンピース")` qui vaudra 5 (codé sur 15 octets).

```
int utf8_length(char *c) {
    int n;
    int i;
    n=0;
    for (i=0; c[i] != '\0'; i++) {
        if (!prefix_p(c[i],2)) n++;
    }
    return n;
}
```

Exercice 3 : Construire une chaîne Unicode (8 points)

On souhaite maintenant aller plus loin dans notre gestion de l'UTF-8 en découpant une chaîne de caractères (sous forme de tableau d'octets) en un tableau de caractères Unicode (chaque caractère Unicode étant codé sous forme de chaîne d'octets). On parlera de chaîne d'octet pour le type `char *` et de chaîne Unicode pour le futur type `utf8_str` implémentant un tableau de chaînes d'octets, chacune ne codant qu'un caractère (sur 1, 2, 3 ou 4 octets, *cf* exercice précédent).

Ci-dessous, la chaîne d'octets codant `"météo"` en C.

01101101	11000011	10101001	01110100	11000011	10101001	01101111	00000000

Ci-dessous, la chaîne Unicode codant `"météo"`. C'est ce découpage que l'on cherche à obtenir dans cet exercice. Ici, on obtient un tableau de 5 caractères Unicode codés eux même sous forme de chaînes d'octets traditionnelles.

0. `"m"` : | 01101101 | 00000000 |
1. `"é"` : | 11000011 | 10101001 | 00000000 |
2. `"t"` : | 01110100 | 00000000 |
3. `"é"` : | 11000011 | 10101001 | 00000000 |
4. `"o"` : | 01101111 | 00000000 |

1. Commencer par créer une structure `utf8_str` contenant deux champs, un tableau `tab` de chaînes de caractères (c'est-à-dire un tableau de chaînes d'octets) ainsi qu'un entier `length` correspondant à la taille du tableau.

```
typedef struct {
    int length;
    char **tab;
} utf8_str;
```

2. Écrire maintenant une fonction `utf8_str utf8_init(int n)` qui initialise une nouvelle chaîne Unicode (de type `utf8_str`) en allouant, sur le tas, un tableau de la bonne taille et en initialisant en conséquence les deux champs `length` et `tab`. Pour l'instant, le tableau est créé sans être rempli.

```
utf8_str utf8_init(int n) {
    utf8_str s;
    s.length = n;
    s.tab = malloc(n * sizeof(char*));
    return s;
}
```

3. Écrire une fonction `char *utf8_read_one_char(char *ch, int *i)` qui lit le caractère Unicode située dans la chaîne d'octets à l'indice `i` et le renvoie sous forme de chaîne. L'algorithme regardera le préfixe de l'octet à l'indice i et en déduira la longueur du caractère Unicode en nombre d'octets. On pourra alors extraire la sous-chaîne en utilisant la fonction de l'exercice 1.

On mettra à jour la valeur de l'entier `i` en y affectant l'indice de début du prochain caractère. En cas d'erreur (préfixe invalide ou chaîne trop courte), on renverra le pointeur `NULL` et on affectera `-1` à l'entier i.

Dans l'exemple de la chaîne `"météo"` de la page précédente, en commençant à lire à partir de l'indice `i=0`.

- `read_one_char` avec `i=0` renvoie la chaîne `"m"` et affecte `1` à `i`.
- `read_one_char` avec `i=1` renvoie la chaîne `"é"` et affecte `3` à `i`.
- `read_one_char` avec `i=3` renvoie la chaîne `"t"` et affecte `4` à `i`.

```
char *utf8_read_one_char(char *ch, int *i) {
    int longueur;
    char *resultat;
    if      (prefix_p(ch[*i], 1)) { longueur = 1; }
    else if (prefix_p(ch[*i], 3)) { longueur = 2; }
    else if (prefix_p(ch[*i], 4)) { longueur = 3; }
    else if (prefix_p(ch[*i], 5)) { longueur = 4; }
    else {
        *i=-1;
        return NULL;
    }
    resultat = substring(ch, *i, longueur);
    if (resultat != NULL) {
        *i = *i+longueur;
    } else {
        *i = -1;
    }
    return resultat;
}
```

4. En déduire une fonction `utf8_str utf8_read(char *ch)` qui crée et initialise la chaîne Unicode associée. En cas d'erreur, on affectera au champs `length` le nombre de caractères lus sans erreurs. On pourra (et devra) utiliser les fonctions `utf8_*` définies précédemment.

```
utf8_str utf8_read(char *ch) {
  int n = utf8_length(ch);
  utf8_str s = utf8_init(n);
  int i,j;
  i = 0;
  for(j=0; j<n; j++) {
      s.tab[j] = utf8_read_one_char(ch, &i);
      if (i<0) {
          s.length = j;
          break;
      }
  }
  return s;
}
```

5. Écrire la fonction `void utf8_printf(utf8_str s)` qui affiche tous les caractères de la chaîne sans retour à la ligne final.

```
void utf8_printf(utf8_str s) {
    int i;
    for (i=0; i<s.length; i++) {
        printf("%s", s.tab[i]);
    }
}
```

6. Écrire `void utf8_free(utf8_str s)` qui libère tous les éléments alloués sur le tas accessible depuis `s`.

```
void utf8_free(utf8_str s) {
    int i;
    for (i=0; i<s.length; i++) {
        free(s.tab[i]);
    }
    free(s.tab);
}
```

Exercice 4 : Les fonctions ord et chr (3 points)

Dans le codage Unicode, chaque caractère a un unique numéro. Les caractères de l'intervalle $[0, 128[$ (128=0x80 en hexadécimal) sont les caractères ASCII codés sur un octet. Ils sont égals à leur propre numéro. Les caractères de l'intervalle $[128, 2048[$ (2048=0x800) sont les caractères que l'on code sur deux octets. Ils correspondent aux caractères permettant de coder les écritures européennes et arabes. Les caractères de l'intervalles $[2048, 65536[$ (65536=0x10000) correspondent au caractères asiatiques (chinois, japonais, etc.). Enfin les caractères de l'intervalle $[65536, 1114111[$ (1114111=0x10FFFF) correspondent au reste, (émojis, langues rares) .

Pour trouver le numéro associé à un caractère, il suffit de ne garder que les bits que ne sont pas associés à des préfixes pour obtenir l'écriture en base 2 du numéro. Par exemple, le caractère `"é"` est codé par `11000011` `10101001`. Si on enlève les préfixes, il ne reste que `00011 101001` ce qui donne 233 en base 10.

1. Écrire la fonction `int ord(char *ch)` qui à partir d'une chaîne représentant un unique caractère (de l'intervalle $[0, 2048[$ et donc codé sur 1 ou 2 octets) renvoie le numéro associé. On aura, par exemple, `ord("é") == 233`. On renverra -1 pour les caractères au delà de 2048 *Bonus : le faire pour tous les caractères Unicode, même au delà de 2048.*

```c
int ord(char * ch) {
    int a = 0, b = 0, c = 0, d = 0;
    if (prefix_p(ch[0], 1)) {
        d = ch[0];
        return d;
    } else if (prefix_p(ch[0], 3)) {
        c = ch[0] & ((1<<5)-1) ;
        d = ch[1] & ((1<<6)-1) ;
        return (c<<6) + d;
    } else if (prefix_p(ch[0], 4)) { /* Non demandé */
        b = ch[0] & ((1<<4)-1) ;
        c = ch[1] & ((1<<6)-1) ;
        d = ch[2] & ((1<<6)-1) ;
    } else if (prefix_p(ch[0], 5)) { /* Non demandé */
        a = ch[0] & ((1<<3)-1) ;
        b = ch[1] & ((1<<6)-1) ;
        c = ch[2] & ((1<<6)-1) ;
        d = ch[3] & ((1<<6)-1) ;
    } else {
        return -1;
    }
    return (a<<18) + (b<<12) + (c<<6) + d;
}
```

2. Écrire la fonction inverse qui construit un caractère à partir du numéro `char *chr(int n)`. On supposera que n sera dans l'intervalle $[0, 2048[$. *Bonus : le faire pour tous les caractères Unicode, même au delà de 2048.*

```c
char *chr(int i) {
    char * res = NULL;
    int masque = (1<<6) -1; /* masque = 0b111111 => 6 uns */
    if (0<=i && i<0x80) {
        res = malloc(2);
        res[0] = i;
        res[1] = 0;
    } else if (i<0x800) {
        res = malloc(3);
        res[2] = 0;
        res[1] = 128 + (i & masque); /* 128 = 0b10000000 => 10 suivi de 6 zéros*/
        i = (i>>6);
        res[0] = 192 + i; /* 192 = 0b11000000  */
    } else if (i<0x10000) { /* Non demandé */
        res = malloc(4);
        res[3] = 0;
        res[2] = 128 + (i & masque); i = (i>>6);
        res[1] = 128 + (i & masque); i = (i>>6);
        res[0] = 224 + i; /* 224 = #b11100000  */
    } else if (i<0x10FFFF) { /* Non demandé */
        res = malloc(5);
        res[4] = 0;
        res[3] = 128 | (i & masque); i = (i>>6);
        res[2] = 128 | (i & masque); i = (i>>6);
        res[1] = 128 | (i & masque); i = (i>>6);
        res[0] = (unsigned char) (240 | i); /* 240 = #b11110000  */
    }
    return res;
}
```

Exercice 5 : Un peu de vaλisp (4 points)

1. On cherche à calculer des statistiques sur la gestion de la mémoire de notre allocateur vaλisp. Pour cela on définit une structure contenant trois champs qui permettent de compter le nombre de blocs libres, le nombre de blocs alloués pour l'utilisateur et le nombre de bloc utilisés par l'allocateur (ce sont les blocs dans lesquelles sont stockées les informations de la liste doublement chaînées de l'allocateur).

On se donne le type `struct statistiques` déjà défini page suivante avec trois champs entiers. Écrire une fonction `struct statistiques stat(void)` qui renvoie une telle structure correctement remplie. Ci-dessous, sont rappelées les fonctions écrites en TP que vous pouvez réutiliser.

```
bloc cons_bloc(int rm, int precedant, int libre, int suivant);
int bloc_suivant(int i);  int bloc_precedant(int i);
int usage_bloc(int i);    int rm_bloc(int i);       int taille_bloc(int i);
struct statistiques { int none;     /* Nombre de blocs libres */
                      int user;     /* Nombre de blocs réservés */
                      int alloc ;}; /* Nombre de blocs utilisés par l'allocateur  */
```

```
struct statistiques stat(void) {
    bloc i;
    struct statistiques stats;
    stats.none = 0, stats.user = 0; stats.alloc= 0;
    for (i=0; i!=bloc_suivant(i) ; i=bloc_suivant(i)) {
        stat.alloc ++;
        if (usage_bloc(i)) { stat.none += taille_bloc(i); }
        else { stat.user += taille_bloc(i); }
    }
      return stats;
}
```

2. Écrire la fonction `sexpr supprimer_env(sexpr env, sexpr symbole)` qui modifie et renvoie l'environnement donnée en paramètre en supprimant le couple (la liaison) correspondant au symbole. On pourra utiliser les fonctions ci-dessous. Par exemple, supprimer le symbole y dans l'environnement `(x . 3)` → `(y . 10)` → `(z . 10)` → `nil` renverra l'environnent `(x . 3)` → `(z . 10)` → `nil`

```
sexpr cons( sexpr e1, sexpr e2);
sexpr car(sexpr e);
sexpr cdr(sexpr e);
void set_car(sexpr e, sexpr nouvelle); /* remplace le car de e par nouvelle */
void set_cdr(sexpr e, sexpr nouvelle); /* remplace le cdr de e par nouvelle */
char * get_symbol(sexpr  val);
bool symbol_match_p(sexpr val, const char *chaine);
```

```
sexpr supprimer_env(sexpr env, sexpr symbole) {
    sexpr e, couple, variable, valeur, prec;
    char* nom = get_symbol(symbole);
    prec = NULL;
    for (e=env; e; e=cdr(e)) {
        couple = car(e);
        variable = car(couple);
        if (symbol_match_p(variable, nom)) {
            if (prec==NULL) return cdr(env);
            set_cdr(prec, cdr(e)); /* Avant : prec->e->cdr(e) Après : prec->cdr(e) */
            return env; }
        prec = e;
    }
}
```