Rosalie Duteuil (n° étudiant : 22000654)

Conception de programmes l'approche impérative

Chapitre 5

Index lexicographique - cx17

Types et constantes

```
typedef unsigned idx ;    // type index
typedef char * str ;    // type string
typedef enum {False, True} bool ;

#define maximum 1024    // espace mémoire pour lire une ligne de texte
#define max_mots 4096   // nombre max d'éléments dans la table mot
```

Représentation des données

```
typedef struct node { idx car ; struct node * cdr ; } node, * list ;
typedef struct snode { str car ; struct snode * cdr ; } snode, * slist ;

#define nil NULL          // pour faire plus lisp

struct ndex { str mot ; list refs; } ;
```

Variables globales

```
struct ndex mots[max_mots] ;  // table des mots indexés
idx mot_libre = 0 ;       // indice du premier emplacement libre
slist stoplist = nil ;      // mots ignorés --> à placer dans "stoplist.data"
```

Gestion des listes

```
// ajoute un élément en début de liste
list cons(idx car, list L)
{ list new = malloc(sizeof(node)) ;
  if (! new) usage("cons : manque de RAM") ;
  new -> car = car ;
  new -> cdr = L ;
  return new ; }

// même chose avec des listes de chaînes
slist scons(str car, slist L)
{ slist new = malloc(sizeof(snode)) ;
  if (! new) usage("cons : manque de RAM") ;
  new -> car = car ;
  new -> cdr = L ;
  return new ; }

// affiche les éléments d'une liste d'entiers
void putlist(list L)
{ if (! L) return ;
  putlist(L -> cdr) ;
  printf("%i ", L -> car) ; }

// Cherche un mot dans une liste
int dans_list(str mot, slist L)
{ if (! L) return 0 ;   // on a pas trouvé le mot
  return (pareil(L -> car, mot) || dans_list(mot, L -> cdr)) ; }
```

Fonction main

```
int main(int k, const str ldc[])
{ ini_stoplist(k, ldc) ;          // récupération de la stopliste

  if (k < 2) usage("Usage : indiquer le nom du fichier à indexer...") ;
  char ligne[maximum] ;          // espace pour une ligne
  FILE * flux = fopen(ldc[1], "r") ;    // ouvre le texte
  if (! flux) usage("Erreur : problème avec le fichier...") ;

  idx x = 0 ;               // à partir de l'index 0
  while (fgets(ligne, maximum, flux))   // tant qu'il y a du texte
    indexe(ligne, ++x) ;        // indexe la ligne suivante
  fclose(flux) ;

  dump(mot_libre) ;           // affiche - jusqu'au mot inutilisé

  return 0 ;  }
```

```
void usage(str message) { fprintf(stderr, "%s\n", message) ; exit(1) ; }
```

Fonction indexe()

```
const str split_chars = "(., ;'!?/\n\t\")" ;

// effectue l'indexage pour une ligne, fournie en entrée
void indexe(str ligne, idx ref)
{ str mot = strtok(strdup(ligne), split_chars) ;    // récupère le premier mot de la ligne

  while (mot)
  { if (!dans_list(mot, stoplist))  // si le mot n'est pas dans la stoplist, on indexe :
    { int x = indice(mot) ;   // est-ce que le mot est déjà indexé ?
      if (x < 0) ajoute_mot(mot_libre, mot, ref) ; // si non ajout à l'index
      else ajoute_ref(x, ref) ; } ;   // si oui ajout de la nouvelle réf
    mot = strtok(NULL, split_chars) ; } } // on passe au mot suivant
```

Fonction indice()

```
// Renvoie l'indice d'un mot donné dans l'index, -1 s'il n'est pas indexé
int indice(str mot)
{ idx x ;
  for (x = 0; mots[x].mot ; ++x)          // ce mot est-il déjà mémorisé ?
    if (pareil(mot, mots[x].mot)) return x ;  // oui : retourne cet index
  return -1 ; }                 // non : retourne négatif

// compare deux chaînes de caractères, renvoie un booléen (insensible à la casse)
bool pareil(str x, str y) { return strcasecmp(x, y) ? False : True ; }
```

Fonctions ajouts_mot() et ajoute_ref()

```
// ajoute un mot nouveau dans les tables mots et refs, à l'indice donné ; màj indice de
    prochain emplacement libre
void ajoute_mot(idx x, str mot, idx ref)
{ mots[x].mot = mot ;         // stocke le mot à l'emplacement x
  mots[x].refs = cons(ref, nil) ;   // ajoute la première référence pour ce mot (ligne
    actuelle)
  ++mot_libre ; }

// ajoute une référence pour un mot déjà indexé (+ màj du compteur ref_libre)
void ajoute_ref(idx x, idx ref)
{ if (mots[x].refs -> car != ref)         // on vérifie que la ligne ne soit pas déjà
    indexée
    mots[x].refs = cons(ref, mots[x].refs) ;  } // ajout de la référence
```

Fonction dump()

```
// affichage final
void dump(idx k)  // indice du dernier élément de mot [pas nécessaire]
{ qsort(mots, mot_libre, sizeof(struct ndex), compare);
  for (idx x = 0 ; x < k ; ++x)     // scanne les mots un par un
  { if (mots[x].mot)          // test redondant, en l'état
    { printf("%s : ", mots[x].mot) ;  // affiche le mot
    putlist(mots[x].refs) ;       // parcourt et affiche ses références
    printf("\n") ; } } }        // termine par un alinéa

// fonction de comparaison de deux entités ndex (classement alphabétique entre les deux
    mots)
int compare(void const * E1, void const * E2)
{ struct ndex const *pE1 = E1;
  struct ndex const *pE2 = E2;
  return strcasecmp(pE1 -> mot, pE2 -> mot); }
```

Gestion de la stopliste

```
// remplit la variable stoplist avec le contenu du fichier spécifié en LDC (ou
    "stoplist.data" par défaut)
void ini_stoplist(int argc, const str args[])
{ int i = 0;
  while (i < argc && strcmp(args[i++],"-s")) ;  // on parcourt les arguments à la recherche
    de l'option -s
  if (i != argc) mots_en_liste(args[i]) ;     // si -s était spécifié, on récupère les mots
    de la stoplist
  else mots_en_liste("stoplist.data") ; }     // sinon stoplist par défaut


//stocke les mots contenus dans fichier dans la variable globale stoplist.
int mots_en_liste(str fichier)
{ FILE * R = fopen(fichier, "r") ;          // ouverture du fichier
  if (! R) usage ("Erreur : problème avec la stop-liste. Vérifier le fichier stoplist.data
    ou préciser le chemin avec l'option -s") ;
  char buffer[32] ;                 // espace pour un mot
  while (fscanf(R, "%s", buffer) != EOF)        // tant qu'il reste des mots
    stoplist = scons(strdup(buffer), stoplist) ;  // on enregistre le mot
  return 0 ;}
```

Tentative de fusion des listes en type générique

```
typedef struct node { void *car ; struct node * cdr ; } node, * list ;

list cons(void * car, list L)
{ list new = malloc(sizeof(node)) ;
  if (! new) usage("cons : manque de RAM") ;  // enfin, un peu de sérieux !
  new -> car = car ;
  new -> cdr = L ;
  return new ; }

// affiche les éléments d'une liste d'entiers
void putlist(list L)
{ if (! L) return ;
  putlist(L -> cdr) ;
  printf("%i ", *(int *) L -> car) ; }

// Cherche un mot dans une liste
int dans_list(str mot, list L)
{ if (!L) return 0 ;    // on a pas trouvé le mot
  return (pareil((str) L -> car, mot) || dans_list(mot, L -> cdr)) ; }
```

Problème des entiers

```
void ajoute_mot(idx x, str mot, idx ref)
{ mots[x].mot = mot ;         // stocke le mot à l'emplacement x
  int * ligne = malloc(sizeof(int)) ;
  if (! ligne) usage("cons : manque de RAM") ;
  *ligne = ref ;
  mots[x].refs = cons(ligne, nil) ; // ajoute la première référence pour ce mot (ligne
    actuelle)
  ++mot_libre ; }


void ajoute_ref(idx x, idx ref)
{ if (*(int *)(mots[x].refs -> car) != ref) {     // on vérifie que la ligne ne soit pas
    déjà indexée
    int * ligne = malloc(sizeof(int)) ;
    if (! ligne) usage("cons : manque de RAM") ;
    *ligne = ref ;
    mots[x].refs = cons(ligne, mots[x].refs) ;  }}  // ajout de la référence
```

Chapitre 8

Émulation - cx25

Types de données et variables globales

```
typedef char * str ;        // type string
typedef unsigned char adr ;   // type adresse (0 à 255)

#define taille_memoire 256

adr memoire[taille_memoire] ; // variables stockées comme entiers non signés sur 8 bits
int debut_ram = 32 ;    // adresses 0 à 31 = ROM
adr PC = 32 ;          // début du programme
adr A = 0 ;            // accumulateur
int iter_max = -1 ;     // nombre max d'itérations ; mettre -1 pour  une boucle infinie
```

Récupération du programme

Format d'entrée : fichier texte composé de bigrammes hexadécimaux. La taille du programme ne doit pas dépasser la RAM disponible.

```
int get_prgm(str fichier)
{ FILE * flux = fopen(fichier, "r") ;   // ouvre le texte
  if (! flux) return 1 ;          // impossible de lire le fichier

  int i = debut_ram, lu ;       // index (prochain emplacement disponible en mémoire)
  unsigned int code ;           // opcode (bigramme hexa) lu dans le fichier
  while (i < taille_memoire + 1)
  { if (!(lu = fscanf(flux, "%2X", &code))) return 2 ;    // caractère non conforme
    else if (lu == -1) break ;      // fin du fichier
    else memoire[i++] = (adr)code; }  // sinon : inscription de l'octet en mémoire
  fclose(flux) ;
  return (i == taille_memoire + 1) ? 3 : 0 ;}   // si i est > la mémoire disponible, le
    programme est trop long, sinon tout s'est bien passé
```

Boucle d'exécution

```
int execute()
{ for(int k = 0 ; k != iter_max ; k++)    // boucle finie ou non (cf iter_max)
  { int OP = memoire[PC++] ;        // on récupère le code opération et on avance
    if (OP / 16 == 1) saut(OP) ;      // codes qui commencent par 1 = sauts
    else if (exe_commande(OP, memoire[PC])) // sinon exécution (code opération + opérande)
      return -1 ;             // instruction non reconnue -> erreur

    if (PC == 0) break ;          // on a atteint la fin du programme
    else if (PC < debut_ram) return -1 ; }      // tentative accès ROM -> erreur
  return 0 ; }
```

Les fonctions de branchements

```
int saut(int OP)
{ if (OP == 16 || (OP == 17 && (char)A < 0) || (OP == 18 && (char)A == 0))  // vérification
    de la condition du saut
    PC = memoire[PC] ;      // mise à jour du PC (saut)
  else PC++ ;           // incrémentation du PC (pas de saut)
  return 0 ;}
```

Les autres opérations

```
int exe_commande(int OP, adr AD)  // opcode et opérande
{ adr vrb ;               // variable utilisée dans l'opération

  switch ((int)(OP / 16))     // récupère la variable avec + ou - d'indirection(s)
  {   case 0 : case 2 : vrb = AD ; break ;        // codes 0x ou 2x -> opérande = valeur
    case 4 : case 6 : vrb = memoire[AD] ; break ; // codes 4x ou 6x -> opérande = adresse
    case 12 : case 14 : vrb = memoire[memoire[AD]] ; break ;  // codes Cx ou Ex -> opérande
    = adresse d'un pointeur
    default : return -1 ; }                  // erreur : code opération non conforme

  switch (OP)           // exécution de la commande
  { case 0 : case 64 : case 192 :   A = vrb ; break ;     // inscrit vrb dans A
    case 32 : case 96 : case 224 :  A += vrb ; break ;    // ajoute vrb à A
    case 65 : case 193 : printf("%i\n", vrb); break;      // envoie vrb en sortie (entier
    signé)
    case 33 : case 97 : case 225 :  A -= vrb ; break ;      // soustrait vrb à A
    case 34 : case 98 : case  226 : A =  ~(A & vrb) ; break ; // fait non-et entre A et vrb
    case 72 : memoire[AD] = A ; break ;             // inscrit A en mémoire
    case 200 : memoire[memoire[AD]] = A ; break ;       // inscrit A en mémoire (avec
    indirection)
    case 73 : store(AD); break ;                // inscrit l'entrée utilisateur en mémoire
    case 201 :store(memoire[AD]); break ;           // inscrit l'entrée utilisateur en
    mémoire (avec indirection)
    default : return -1 ; }

  PC++ ;              // incrémentation du PC
  return 0; }
```

Récupération de la saisie utilisateur

```
void store(adr adresse)
{ int lu;
  char buffer[3] ;
  while (1)     // attente d'une saisie correcte
  { fgets(buffer, 3, stdin) ;
    if (sscanf(buffer, "%2X", &lu) == 1) break ; }
  memoire[adresse] = (adr)lu ; }  // stockage en mémoire
```

La fonction main()

```
int main(int argc, str argv[])
{ if (argc < 2) {usage(4) ; return -1 ;}  // manque le nom de fichier
  int err = get_prgm(argv[1]) ;     // résultat de la lecture du programme
  if (err) {usage(err) ; return -1 ;}   // échec de lecture -> erreur et fin
  if (execute()) usage(5);        // exécution (affichage si erreur)
  return 0; }
```

```
void usage(int erreur)
{ switch(erreur)
  { case 1 : fprintf(stderr, "Erreur : le fichier ne peut pas être lu.\nArrêt du
    programme...\n") ; break ;
    case 2 : fprintf(stderr,"Erreur : le code contient des caractères non conformes.\nArrêt
    du programme...\n") ; break ;
    case 3 : fprintf(stderr,"Erreur : le code est trop long pour la mémoire (%i o de
    RAM).\nArrêt du programme...\n", taille_memoire - debut_ram) ; break ;
    case 4 : fprintf(stderr,"Usage : indiquer le nom du fichier à indexer...\n") ; break ;
    case 5 : fprintf(stderr,"Erreur : problème dans le code (opération invalide, ou accès
    mémoire interdit)\n") ; } }
```

Exercice cx25.1

La fonction execute()

```
// exécution d'une instruction du programme
int execute()
{ int OP = memoire[PC++] ;    // on récupère le code opération et on avance
  printf("\nExécution de l'instruction : %02X %02X ...\n", OP, (memoire[PC])) ;

  if (OP / 16 == 1) saut(OP) ;  // codes qui commencent par 1 = sauts
  else if (exe_commande(OP, memoire[PC]))   // sinon exécution (code opération + opérande)
    return -1 ;       // instruction non reconnue -> erreur

  printf("PC : 0x%02X \t A : 0x%02X (%i)\n\n", PC, A, (char)A); // info registres,
    instruction

  if (PC == 0) return 1 ;       // on a atteint la fin du programme
  else if (PC < debut_ram) return -1 ;      // tentative accès ROM -> erreur
  return 0 ; }
```

Fonction main()

```
printf("Lancement de l'exécution du programme...\n(Taper 'm' pour consulter l'état de la
    mémoire)\n") ;

while(1)
{ switch (execute())
  {   case -1 : usage(5) ; return -1 ;
    case 1 : printf("Fin du programme.\n") ; return 0 ; }
  if (interac()) break; }
```

Fonction interact

```
// pause, attente d'une saisie de l'utilisateur pour avancer, arrêter, ou afficher la
    mémoire
int interac()
{ char buffer[64] ;
  printf("Continuer avec la prochaine instruction (o/n) ? ") ;
  while(1)              // boucle jusqu'à une saisie conforme
  { fgets(buffer, 64, stdin) ;    // récupération de la saisie
    if (buffer[0] == 'o') break ; // on continue l'exécution
    else if (buffer[0] == 'n') {puts ("Arrêt de l'émulateur...") ; return -1 ; }  // arrêt
    du stepper
    else if (buffer[0] == 'm') affiche_mem() ;  // affichage mémoire
    else printf("Saisie invalide. ") ;      // saisie non conforme
    printf("Entrer \"o\" pour continuer, \"n\" pour quitter : ") ;} // confirmer les options
  return 0 ; }
```

L'affichage est assuré par la fonction suivante :

```
void affiche_mem()
{ for (int i = 0 ; i < 16 ; i++)
  { for (int j = 0 ; j < 16 ; j++) printf("%i ", memoire[16*i+j]);
    puts("") ; } }
```

Améliorations possibles

- Faire saisir PC, et iter_max à l'utilisateur (pour ne pas recompiler le programme à chaque fois)
- Dump mémoire : afficher les numéros d'adresse en début de lignes (ou de colonnes), pouvoir préciser la (ou les) case(s) mémoire à afficher, …
- Stepper : afficher la traduction des opcodes
- …