CACHAN
Département d'électronique

Cachan le 6 mars 2006

Travaux dirigés d'informatique industrielle
Automates en C

Les exercices ci-dessous seront traités en utilisant une approche « automates ». Lors de la séance de travaux pratiques (XAO) d'informatique industrielle on réalisera et testera le bon fonctionnement de certains de ces automates.

Un monte-charge

Cet automate est le plus rudimentaire. Il scrute en permanence trois entrées du port parallèle p0, p1 et appel (actifs à '1') qui indiquent respectivement :

- Que la cabine est au rez de chaussée,
- Que la cabine est à l'étage,
- Que l'utilisateur demande un changement de niveau.

Au démarrage du programme, si la cabine n'est pas au rez de chaussée, l'automate l'y envoie sans intervention externe et arrête le moteur.

L'automate fournit en sortie sur le port parallèle deux commandes, haut et bas, qui commandent la mécanique.

- Etablir un diagramme de transitions à quatre états : init, arret, monte et descend qui matérialisent le fonctionnement.
- Représenter l'état de l'automate par une variable de type énuméré pouvant prendre les quatre valeurs prévues.
- Rechercher quelque part la façon d'adresser le port parallèle. La sortie port parallèle correspond-elle à une sortie directe ou à une sortie mémorisée ?
- Coder l'automate en C, directement dans la fonction main, en suivant un algorithme simplissime :

```
déclarations
Initialiser le port parallèle
Lire la position de la cabine
Initialiser l'automate
Pour toujours
     Lire les entrées
     Actualiser l'état
     Actualiser les sorties
```

  On étudiera deux versions de réalisation de l'algorithme :
  1. une version qui utilise des instructions de contrôle (switch, if … else),
  2. une version qui utilise une table de transitions : un tableau de structures définit pour chaque état actuel et chaque entrée le couple (etat futur, sorties).
- Remplacer chacune des actions élémentaires précédentes par l'appel à une fonction ad-hoc.
- Rajouter une détection d'ouverture et de fermeture de porte qui évite d'une part de démarrer le monte charge porte ouverte, et laisse à l'utilisateur le temps de sortir quand il est arrivé à destination.
- Tester sur la carte coldfire les différentes versions du programme.

Un digicode

Pour réaliser un digicode nous adopterons une démarche un peu différente dans le traitement des entrées. Un programme principal appelle la fonction digicode, qui réalise l'automate, chaque fois qu'une nouvelle valeur d'un caractère est rentrée au clavier, en lui passant ce caractère en argument. Cette fonction retourne une valeur qui autorise (1) ou non (0) l'ouverture de la porte.

- Etablir un premier diagramme de transitions en prenant un code secret fixe, par exemple "CA94".
- Réaliser l'automate en C, et l'appeler au moyen d'une fonction principale de test.
- Placer le code secret dans un tableau plutôt que dans le code du diagramme de transitions, l'état de l'automate devenant l'index du tableau.

Le digicode précédent a pour défaut de devoir être recompilé à chaque changement de code, ce qui n'est guère raisonnable. Pour palier ce défaut on définit deux codes : un code utilisateur et un code superviseur. Les deux codes sont stockés dans des variables de type tableaux de 4 chiffres hexadécimaux, et initialisés à '0000' (utilisateur) et 'AAAA' (superviseur). Le code superviseur autorise la modification de l'un ou l'autre des codes précédents :

```
<code utilisateur> => ouverture de la porte
<code superviseur> '0' => <nouveau code utilisateur>
<code superviseur> 'A' => <nouveau code superviseur>
```

- Tenter une ébauche de diagramme de transitions. La présence de deux codes indépendants pose un problème, lequel ?
- La solution du problème précédent peut être facilitée en introduisant deux automates en interaction. Préciser ce point et proposer les diagrammes de transitions des deux automates. Discuter des actions à prendre si les deux codes sont identiques.
- Réaliser le programme de la fonction digicode en C.
- Effectuer les tests.

Où l'on transforme un PC, ou une carte à microcontrôleur, en calculette

Ce premier exercice n'est pas à proprement dit de l'informatique industrielle, il peut être réalisé sur n'importe quelle machine qui dispose d'une unité centrale, d'un clavier et d'un écran. Il faut évidemment disposer également d'un compilateur C…

On souhaite réaliser une calculatrice simple, disposant des quatre opérations de base agissant sur des nombres « à virgule », c'est à dire sans les exposants. Un tel objet doit évidemment traiter correctement les priorités entre opérateurs :

2 + 3 * 4 => 14

2 * 3 + 4 => 10 etc.

Notre machine hypothétique dispose d'une entrée qui lit un caractère au clavier. Lecture « au vol », c'est à dire que le caractère est envoyé sans attendre la frappe d'un caractère spécial du style retour chariot ou équivalent. Pour CVI cette fonction s'appelle getkey(), si vous préférez Borland vous utiliserez getch() (version sans écho, notre programme gère l'écho à l'écran).

Analyse de quelques problèmes à traiter :

Un petit problème se pose immédiatement : si je tape 3.1416+2 comment faire pour reconnaître que le chiffre '6' appartient bien au nombre Pi, mais que le symbole '+' qui le suit est un opérateur d'addition. Nous n'allons évidemment pas demander à l'utilisateur de taper un espace, ou n'importe quoi d'autre, il a le droit d'être bête. Nous nous chargerons de l'opération. Pour trouver la fin du nombre il a bien fallu lire le caractère '+', mais pour traiter la suite il faut le redéposer sur le clavier ! Les librairies C sous Unix permettent de faire cette « délecture », nous devrons ici nous en charger nous même (fonction clavier()).

Autre point amusant : comment reconnaître qu'un nombre est bien construit ? En fait il suffit de le décrire : « un nombre commence par un chiffre ou un point (décimal). Suivent des chiffres ou, si le premier caractère était un chiffre, des chiffres suivis d'un point suivi de chiffres. Le premier caractère qui viole cette règle indique la fin du nombre. ». Cette phrase suffit pour écrire un automate qui décrit la grammaire d'un nombre (fonction lire_val()).

Pour traiter la grammaire d'une expression c'est un peu plus compliqué. La priorité des opérateurs impose de mettre des opérations en attente pour traiter d'abord des opérations plus prioritaires. Une façon (il y en a des dizaines, dont trois ou quatre propres) de traiter le problème est de créer une pile d'opérations en attente. C'est ce que font certains fabriquants de calculettes pédantes qui imposent à l'utilisateur de penser « postfixé ». Nous ne suivrons évidemment pas cet exemple, nous gérerons la pile d'opérations en attente nous même.

Le programme complet (non certifié) est fourni ci-dessous. Les questions qui suivent nécessitent une lecture attentive du code fourni.

Fonction clavier :

- Etablir un diagramme de transitions qui explique l'évolution de la variable etat_tampon.
- Expliquer le fonctionnement de cette fonction de lecture de bas niveau.
- Pourquoi cette variable est-elle déclarée « statique » ?
- Proposer une généralisation de cette fonction qui gère un tampon clavier de plus de un caractère.

Fonction lire_val :

- Décrire par un diagramme de transitions la structure d'un nombre décimal.
- En déduire une interprétation de l'algorithme utilisé dans lire_val.
- Proposer un remplacement « maison » de la solution de paresse qui consiste à utiliser sscanf().
- Proposer une modification qui prenne en compte la possibilité d'agrémenter un nombre de son signe (l'un des canulars des langages : les symboles '+' et '-' peuvent représenter des opérateurs d'addition ou un signe ; comment les calculettes simples résolvent-elles ce problème ?).
- Proposer une modification qui prenne en compte la notation « scientifique » (12.345e-7).

La calculette proprement dite (fonctions du fichier calc_4op.c) :

- Dessiner les occupations des deux piles (opérateurs et opérandes) pour quelques expressions arithmétiques simples (du genre 1+2*3 ou 2*3+4*5, etc.).
- Analyser la structure d'un opérateur (struct ope), que représentent les deux membres de cette structure ?
- Déduire du programme un diagramme de transitions qui décrit l'évolution de l'automate « calc ».
- Analyser la gestion des piles d'opérateurs et d'opérandes.
- Modifier le programme pour que la calculette accepte des parenthèses.
- Pour avoir une calculatrice scientifique, il est agréable de disposer de fonctions transcendantes (sinus, cosinus etc…). Quelles modifications faudrait-il apporter au programme pour réaliser une telle calculatrice.

```
// clavier.h
#include <utility.h>
#include <ansi_c.h>

int lire_val(double *) ;
typedef enum {poser,prendre} gestion_buf ;
typedef enum {plein,vide} etat ;
void clavier(gestion_buf, int *val) ;


// clavier.c
#include "clavier.h"

void clavier(gestion_buf action, int *val)
{
   static etat etat_tampon = vide ;
   static int tampon ;
   switch(etat_tampon)
      {
         case vide : if(action == prendre)
                     {
                     *val = GetKey() ; // ou toute autre fonction équivalente...
//                     putchar((char)*val) ; // pour avoir l'echo local
                     }
                  else
                     {
                     tampon = *val ;
                     etat_tampon = plein ;
                     }
                  break ;
         case plein :if(action == prendre)
                     {
                     *val = tampon ;
                     etat_tampon = vide ;
                     }
                  break ;
      }
}


int lire_val(double *ad)
{
   double valeur ;
   static char buf[128] ;
   int caract,i ;
   typedef enum {debut,fract,ent} gestion_nbre ;
   gestion_nbre etat ;
   etat = debut ;
   i = 0 ;
   while(i < 127)
      {
      clavier(prendre,&caract) ;  // accepter un caractère
      if(caract == 'q') exit(0) ; // on peut meme terminer le programme
      switch(etat)
         {
         case  debut  :if(isdigit((char)caract  ))  //  un  nombre  peut  débuter  par  un
chiffre...
                     {
                     buf[i++] = caract ;
                     putchar(caract) ;
                     etat = ent ;  // on continue sur une partie entière
                     }
                  else if(caract == '.')  //...ou par un point décimal
                     {
                     buf[i++] = caract ;
                     putchar(caract) ;
                     etat = fract ; // on continue sur une partie fractionnaire
                     }
                  break ; // on ignore les autres caractères
         case fract :if(isdigit((char)caract )) // partie fractionnaire : des chiffres
                     {
                     buf[i++] = caract ;
                     putchar(caract) ;
                     }
                  else     // fin de la partie fractionnaire
                     {
```

```
                     buf[i] = 0 ; // on termine la chaine
                     i = 127 ;    // on force une sortie de boucle
                     clavier(poser,&caract) ; // on repose le caractère ignoré
                     }
                  break ;
         case ent :  if(isdigit((char)caract )) // partie entière : des chiffres ...
                     {
                     buf[i++] = caract ;
                     putchar(caract) ;
                     }
                  else if(caract == '.')  //...ou par un point décimal
                     {
                     buf[i++] = caract ;
                     putchar(caract) ;
                     etat = fract ; // on continue sur une partie fractionnaire
                     }
                  else     // fin d'un nombre entier
                     {
                     buf[i] = 0 ; // on termine la chaine
                     i = 127 ;    // on force une sortie de boucle
                     clavier(poser,&caract) ; // on repose le caractère ignoré
                     }
                  break ;
         }
      }
   return sscanf(buf,"%lf",ad) ;  // la paresse me perdra
}
// calc_4op.c
// calculette quatre opérations, automate et pile
#include <utility.h>
#include <ansi_c.h>
#include "clavier.h"

#define MAX_OPE 3
#define MAX_VAL 4

typedef enum {push_val,read_op,push_op,pop_op} cal_type ;
typedef struct {char op; char pri;} ope ;
cal_type calc = push_val ;
int lire_val(double *) ;
int lire_op(ope *) ;
void calcule(double *, char) ;

ope pile_ope[MAX_OPE] = {0} ;   // opérateur priorite min en fond de pile
double pile_val[MAX_VAL] ;
ope *top_ope = pile_ope ;
double *top_val = pile_val - 1 ;   // fond de pile vide

void main(void)
{
   ope code ;
   int bidon ;
   while(1)
   {
   switch(calc)
      {
      case push_val : if(lire_val(++top_val)) calc = read_op ;  // valeur correcte
                  else
                     {
                     top_val-- ;  // erreur : on annule l'entree
                     clavier(prendre,&bidon) ;
                     }
                  break ;
      case read_op :  if(lire_op(&code) != -1)  // operateur legal
                     if(code.pri > top_ope->pri) calc = push_op ;
                     else calc = pop_op ;
                  break ;
      case push_op :  *++top_ope = code ;
                  putchar('\n') ;
                  putchar(code.op) ;
                  calc = push_val ;
                  break ;
      case pop_op :   if(code.pri > top_ope->pri)
                     if(code.op == '=') // fond de pile , operateur '='
                        {
                        top_ope = pile_ope ;
```

```
                        top_val = pile_val ;
                        calc = push_val ;
                        }
                     else calc = push_op ; // on empile
                  else
                     {
                     calcule(top_val--,(top_ope--)->op) ;
                     printf("\n=%lf",*top_val) ;
                     }
                  break ;
      }
   }
}


int lire_op(ope *ad_code)
{
   int code ;
    clavier(prendre,&code) ;
   ad_code->op = code ;
   switch(code)
      {
      case '-' :
      case '+' : ad_code->pri = 2 ;
                 return 1 ;
      case '/' :
      case '*' : ad_code->pri = 3 ;
                 return 1 ;
      case '=' : ad_code->pri = 1 ;
                 return 1 ;
      case 'q' : exit(0) ;
      default  : return -1 ;
      }
}

void calcule(double *pile, char op_code)
{
   switch(op_code)
      {
      case '+' :  *(pile - 1) = *pile + *(pile - 1) ;
                  break ;
      case '-' :  *(pile - 1) = *(pile - 1) - *pile ;
                  break ;
      case '*' :  *(pile - 1) = *pile * *(pile - 1) ;
                  break ;
      case '/' :  if(*pile) *(pile - 1) = *(pile -1) / *pile ;
                   else printf("\ndivision par zero\n") ;
                  break ;
      }
}
```

Codage d’un texte en longueur variable

Pour limiter la taille des données on utilise classiquement des codages binaires de longueur variable. Par exemple, en admettant que l’alphabet ne comporte que quatre lettres : a, e, m et r :

```
e => 0
a => 10
m => 110
r => 111
```

- Imaginer un automate qui lit une à une des valeurs binaires et qui fournit en sortie le code caractère correspondant dès qu’il l’a reconnu.
- Que se passe-t-il si l’un des bits lus est faux ? Quelle conséquence cela aurait-t-il sur la robustesse des fichiers compressés si on ne prenait pas la peine de rajouter des contrôles d’erreurs ?