UNIVERSITÉ PARIS-XII — FACULTÉ DE SCIENCES ÉCONOMIQUES ET DE GESTION
Master Professionnel *Méthodes Appliquées de la Statistique et de l'Économétrie pour la Recherche, l'Analyse et le Traitement de l'Information*
Année universitaire 2006-2007 — *François LEGENDRE*
F.Legendre@univ-paris12.fr — http://fj.legendre.free.fr

Projet « Assembleur UNIC »

1 Exposé du projet

Dans le document intitulé « UNIC », je propose différents programmes exécutables directement écrits en décimal. Ces programmes ne sont guère lisibles ; ils sont difficiles à mettre au point. Les programmeurs, très rapidement, ont préféré écrire les programmes dans un langage d'assemblage. C'est un langage de très bas niveau, spécifique à la machine utilisée. Les différentes instructions sont désignées par un mnémonique. Les adresses sont repérées par un symbole.

Une ligne de programme d'assemblage est composée, en général, de trois champs. Par exemple, la ligne suivante

suite crg un

comporte trois champs, respectivement ***suite***, ***crg*** et ***un***. Le premier champ, ici ***suite***, est une étiquette. Sa valeur est l'adresse de l'instruction. Ainsi, cette étiquelle pourra constituer ailleurs dans le programme l'opérande d'une instruction de branchement. Le deuxième champ, ici ***crg***, est le mnémonique d'une instruction. L'instruction dont le code est ***1*** dans UNIC est l'instruction pour charger le registre général. Plutôt que de coder ***1***, le programmeur préfère écrire ***crg*** (pour « *Charger le Registre Général* ») : c'est plus expressif et moins sujet à l'erreur. Le troisième champ, ici ***un***, est l'opérande de l'instruction. Sa valeur dépend de l'instruction : pour l'instruction ***crg***, l'opérande est l'adresse de la variable à charger dans le registre.

Les mnémoniques utilisés pour UNIC sont portés dans le tableau 1. Les deux derniers mnémoniques, dans ce tableau, ne désignent pas des instructions. La pseudo-instruction ***déf*** permet de renseigner un emplacement de la mémoire. Il s'agit en général d'initialiser un emplacement qui est réservé pour une variable. La pseudo-instruction ***syn*** permet de donner à un symbole une valeur pour rendre le programme plus expressif.

Reprenons le premier programme étudié. En C++, ce programme est le suivant.

TAB. 1 – Le jeu d'instructions et de mnémoniques d'UNIC

Instruction	Mnémonique	Explication
0 ***c***⋆	***ase***	Appeler le système d'exploitation, ***c*** étant le code de la fonction appelée
1 ***aa***¶	***crg***	Charger le registre général avec le contenu situé à l'adresse ***aa***
2 ***aa***¶	***srg***	Stocker le contenu du registre général à l'emplacement d'adresse ***aa***
3 ***aa***¶	-	Soustraire, l'opérande étant situé à l'adresse ***aa***
4 ***aa***¶	***bi***	Exécuter l'instruction située à l'adresse ***aa*** (branchement inconditionnel)
5 ***aa***¶	***bsdz***	Exécuter l'instruction située à l'adresse ***aa*** si le registre général n'est pas égal à zéro (branchement conditionnel)
6 ***d***§	***crgp***	Charger le registre général depuis la pile, avec le déplacement ***d***
7 ***d***§	***srgp***	Stocker le contenu du registre général vers la pile, avec le déplacement ***d***
8 ***aa***¶	***appel***	Appeler une fonction@, la première instruction de la fonction étant à l'adresse ***aa***
9	***ret***	Retourner (depuis une fonction)
–	***déf***	Définir un emplacement en mémoire
–	***syn***	Rendre un symbole synonyme d'un entier naturel

⋆ ***c*** est un code de 0 à 3 qui indique la fonction appelée.
¶ ***aa*** est une adresse (de 00 à 99) qui désigne l'opérande.
§ ***d*** est un déplacement (de 0 à 9) qui désigne l'opérande dans la pile.
@ Le registre général contient le nombre de données locales à ne pas écraser.

```
1  #include <iostream> // 'cin', 'cout' et 'cerr'.
2
3  int n = 0 ; // Une variable globale entière initialisée à 0.
4
5  int
6  main() // Fonction principale du programme.
7  {
8    n = 1 ;          // La valeur entière '1' est affectée à la variable 'n'.
9    std::cout << n ; // La variable 'n' est affichée sur la sortie standard.
10   return 0 ;       // Fin du programme avec un code de retour égal à 0.
11 }
```

Ce programme est codé comme suit en décimal.

Adresse	Code	Explication
—	—	Début du segment des instructions.
00	1 13	Charger le registre général.
		13 est l'adresse de la constante ***1***.
03	2 15	Stocker le contenu du registre général.
		15 est l'adresse de la variable globale ***n***.
06	0 2	Appeler le système d'exploitation.
		2 est le code « Ecriture d'un entier vers ***std::cout*** ».
08	1 14	Charger le registre général.
		14 est l'adresse de la constante ***0***.
11	0 0	Appeler le système d'exploitation.
		0 est le code « Fin du programme ».
13	1	Constante ***1***.
14	0	Constante ***0***.
—	—	Début du segment des données globales.
15	0	Variable globale ***n*** initialisée à 0.

Il faut bien comprendre que ce programme exécutable UNIC prend la forme de la suite de chiffres « 1132150211400100 » de la même manière qu'un « vrai » programme exécutable est une suite de chiffre binaires particulièrement difficile à déchiffrer. En langage d'assemblage, ce programme serait codé comme suit.

```
fin_programme   syn   0
écriture_entier syn   2

                crg   un
                srg   n
                ase   écriture_entier
                crg   zéro
                ase   fin_programme
un              déf   1
zéro            déf   0
n               déf   0
```

Le programmeur, au lieu de coder par exemple « 1 13 », va coder « ***crg un*** ». Le mnémonique ***crg*** sera remplacé par 1, le code de l'instruction ***crg***. L'opérande ***un*** est une étiquette ; celle-ci sera remplacée par l'adresse de l'emplacement ainsi désigné.

Il n'est pas très difficile de développer le programme qui, en entrée, analyse le programme d'assemblage et qui, en sortie, engendre le programme exécutable. Deux passes sont toutefois nécessaires. La première passe récole les symboles utilisés et leur donne une valeur. La seconde passe engendre effectivement le code. Dans l'exemple précédent, à l'issue de la première passe, la table des symboles est comme suit.

Symbole	Valeur
fin_programme	1
écriture_entier	2
un	13
n	15
zéro	14

On voit bien, notamment, les deux choses suivantes.

1. Une variable correspond à un emplacement de la mémoire de l'ordinateur.
2. Un symbole (ou un identificateur) est le moyen de désigner une variable ; c'est surtout le moyen de désigner une adresse de cette mémoire.

La seconde passe permet d'engendrer le code. Les pseudo-instructions ***syn*** vont alors être ignorées. La ligne de programme d'assemblage « ***crg un*** » sera par exemple traduite en le code « 1 13 ». En effet, 1 est le code de l'instruction ***crg*** et le symbole ***un*** sera remplacé par l'adresse 13. Les pseudo-instructions ***déf*** n'engendrent pas de code mais elles permettent de définir la valeur initiale de la mémoire.

Dans le projet, une correspondance est utilisée pour faire le lien entre le mnémonique de l'instruction et le code (et la taille) de l'instruction. La syntaxe « ***mnemo2inst["crg"].code()*** » donne 1 parce que le code de l'instruction *Charger le Registre Général* est égal à 1. De même, la syntaxe « ***mnemo2inst["crg"].taille()*** » donne 3.

Plus précisément, cette correspondance est définie comme un « ***map<string, Inst>*** » où ***Inst*** est un objet défini par l'utilisateur. Cet objet sert à représenter une instruction. Il comporte deux attributs, ***code_*** et ***taille_***, qui permettent de représenter les deux caractéristiques d'intérêt d'une instruction.

Pour initialiser un ***map***, on dispose de la syntaxe « ***table[clef] = valeur*** » ou de la syntaxe « ***table.insert(make_pair<TypeClef, TypeValeur>(clef, valeur))*** ». Les deux syntaxes ne sont pas exactement équivalentes. La première, si la clef n'est pas présente, revient à créer une entrée en utilisant le constructeur par défaut pour initialiser la valeur puis à affecter la valeur effective à cette entrée. Par contre, la seconde syntaxe n'utilise pas le constructeur par défaut. Il faut donc préférer la seconde syntaxe, plus lourde, pour initialiser un ***map***. En outre, il n'est alors pas nécessaire de définir un constructeur par défaut.

Par exemple, si on utilise, pour initialiser ***mnemo2inst***, la syntaxe « ***mnemo2inst["ase"] = Inst(ASE, 2)*** », le constructeur par défaut pour l'objet ***Inst*** aurait été appelé. Il faudrait donc remplacer le constructeur par défaut par la ligne suivante.

```
Inst() : code_(), taille_() {} // Constructeur par défaut.
```

Il est fréquent de retenir des noms de la forme « ***mnemo2inst*** » qui se lisent *mnemo to inst*. Il s'agit de rendre explicite une conversion. Par exemple, pour convertir une image au format ***gif*** au format ***png***, on utilise l'utilitaire ***gif2png***.

L'idiome suivant est très fréquemment utilisé pour lire un fichier ligne par ligne.

```
string ligne ;
while ( getline(fichier, ligne) ) {
  ... // Traitement avec 'ligne'.
}
```

La fonction *getline(..., ...)* retourne *false* soit si la fin de fichier est atteinte soit si une erreur de lecture est survenue.

La bibliothèque des flux prévoit aussi de lire depuis une chaîne de caractères ou d'écrire dans une chaîne de caractères. Il faut pour cela inclure le fichier d'en-tête *sstream* en utilisant la directive « *#include <sstream>* ». On dispose ainsi des objets *istringstream* et *ostringstream*. L'objet *istringstream* s'utilise par exemple comme suit.

```
istringstream mon_fichier("1 2 3") ;
size_t i ;
while ( mon_fichier >> i ) {
  cout << "Valeur de i " << i << endl ; }
```

Ce fragment affiche, bien sûr, les trois lignes suivantes.

```
Valeur de i 1
Valeur de i 2
Valeur de i 3
```

C'est ainsi que le plus simple pour lire les trois premiers items (ces items sont séparés par le caractère de tabulation) de chaque ligne du flux d'entrée revient à coder le fragment de programme suivant.

```
vector< vector<string> > lignes_lues ;
string ligne ;
while ( getline(cin, ligne) ) {
  vector<string> items ;
  istringstream lecture(ligne) ;
  string item ;
  while ( getline(lecture, item, '\t') && (items.size() < 3) ) {
    items.push_back(item) ; }
  lignes_lues.push_back(items) ; }
```

La variable *lignes_lues* est un *vector* de *vector* de *string*. La syntaxe *lignes_lues[i]* désigne un *vector* de *string*. La syntaxe *lignes_lues[i][j]* désigne la chaîne de caractères du champ numéro $j+1$ de la ligne numéro $i+1$ puisqu'en C++ les indices commencent à 0. Le premier *getline* sert à renseigner la variable *ligne*. Ensuite, cette variable est l'argument qui sert à construire un *istringstream*. Enfin, le second *getline* sert à renseigner la variable *item*. Cette variable est empilée, au moyen de la méthode *push_back(...)*, dans la variable *items* (qui est un *vector* de *string*). Cette dernière est finalement empilée dans la variable *lignes_lues*.

Plus bas dans le programme, on trouve les trois lignes suivantes.

```
// Les lignes vides sont ignorées ; de même les lignes qui débutent par '#'.
if ( items.empty() || (items[0][0] == '#') ) {
  continue ; }
```

L'expression « *items.empty()* » retourne une variable de type *bool* égale à *true* si le vecteur est vide. Cette expression est équivalente à « *items.size() == 0* ». L'opérateur « || » (ou logique) est un opérateur qui n'évalue pas son opérande de droite si l'opérande de gauche est égal à *true*. Aussi l'expression « *items.empty() || (items[0][0] == '#')* » est-elle sûre. En effet, si le vecteur *items* est vide, l'opérande de gauche (l'expression « *items.empty()* ») est égale à *true* et ainsi l'opérande de droite (l'expression « *(items[0][0] == '#')* ») n'est pas évalué. C'est donc seulement quand le vecteur comporte au moins un élément qu'il est fait une référence à *items[0]*.

L'opérateur « *&&* » (et logique) présente la même particularité. Aussi l'expression « *(x != 0) && (abs(1/x) > 2)* » n'engendre-t-elle jamais une division par zéro.

Quand on utilise un tableau associatif, la syntaxe « *table[clef]* » a l'inconvénient de créer une entrée si celle-ci n'existe pas dans le tableau. Il faut, pour savoir si une clef existe ou non dans le tableau, utiliser la méthode *find(...)*. L'expression « *table.find(clef)* » retourne un itérateur qui est égal à *table.end()* en cas d'échec. Cet itérateur pointe sur une *pair<TypeClef, TypeValeur>*. L'expression « *table.find(clef).second* », en cas de succès, désigne la valeur recherchée puisque l'attribut *second* d'une *pair* désigne le second élément de la paire.

Les adresses sont codées sur deux positions. L'adresse 9 doit par exemple être représentée sous la forme 09. Pour engendrer ces adresses, le plus simple est de coder le fragment de programme suivant.

```
string rep_adresse ;
rep_adresse.push_back('0' + adresse/10) ;
rep_adresse.push_back('0' + adresse%10) ;
```

La variable *adresse* est de type *size_t*. L'expression « *'0' + adresse/10* » est analysée en fonction des règles de priorité des différents opérateurs. L'opérateur « */* » est plus prioritaire que l'opérateur « *+* ». Le calcul effectué en premier est donc « *adresse/10* ». Le type de ce résultat intermédiaire est *size_t*. Le compilateur voit alors qu'il faut convertir le caractère « *'0'* » en un *size_t* pour pouvoir effectuer l'addition. Les caractères sont très souvent encodés en utilisant le codage ASCII pour lequel le code du caractère « *0* », par exemple, est 48.

J'exploite le fait que les chiffres sont codés consécutivement dans la plupart des codages utilisés. Ainsi, en ASCII, le code du caractère « *1* » est 49, celui du caractère « *2* » 50,

etc. Si la variable ***adresse*** est égale, par exemple, à 23, il faut tout d'abord obtenir le code du caractère « ***2*** ». Le calcul $48+2$ permet d'obtenir 50 qui est bien le code du caractère « ***2*** ». Le nombre 2 est obtenu comme le résultat de la division euclidienne de 23 par 10. Ensuite, la méthode ***push_back(...)*** appliquée à une instance de ***string*** attend comme argument un caractère. Le compilateur va donc convertir le résultat (qui est un ***size_t***) en le code d'un caractère. Enfin, il faut obtenir le code du caractère « ***3*** ». Le calcul $48+3$ permet d'obtenir 51, le code du caractère « ***3*** ». Le nombre 3 est obtenu comme le reste de la division euclidienne de 23 par 10. L'opérateur « ***%*** », en C++, donne justement le reste de cette division.

La table des symboles est renseignée à deux occasions : soit quand une ligne d'assemblage comporte une étiquette soit au cas de la pseudo-instruction ***syn***. L'expression « ***items[0].empty()*** » retourne une variable de type ***bool*** égale à ***true*** si la chaîne de caractères qui est contenue dans ***items[0]*** est vide. Quand une étiquette a été renseignée dans la ligne d'assemblage, l'expresssion « ***! items[0].empty()*** » est donc égale à ***true***. Quand la ligne d'assemblage est relative à une pseudo-instruction ***syn***, la table des symboles est simplement renseignée par l'instruction « ***table_symboles[items[0]] = items[2]*** ». Par exemple, si la ligne de programme est « ***bidule syn 0*** », cela se traduit par une instruction équivalente à « ***table_symboles["bidule"] = "0"*** ».

Deux passes, au moins, sont nécessaires. La deuxième passe sert à engendrer le code exécutable, en le jeu d'instructions de UNIC. Une passe ne suffit pas car la valeur de certains symboles ne peut pas être immédiatement résolue. Pour ce qui a trait à notre premier exemple, la valeur du symbole « ***écriture_entier*** » est connue avant son utilisation. En revanche, ce n'est pas le cas du symbole « ***un*** ». Celui-ci est utilisé dans la première instruction exécutable du programme (l'instruction « ***crg un*** ») alors que sa valeur n'est définie qu'ultérieurement (dans la pseudo instruction « ***un déf 1*** »).

2 Le programme

Ci-après figure le programme, en C++, de notre assembleur pour la machine UNIC. Une étude attentive de ce programme est l'occasion de se perfectionner en C++.

```
#include <iostream> // cin, cout et cerr
#include <sstream>  // istringstream
#include <string>   // Chaînes de caractères de la STL
#include <map>      // Tableaux associatifs de la STL
#include <vector>   // Vecteurs de la STL

using namespace std ;

// Erreur fatale : le message est affiché et le programme se termine avec un code de
// retour égal à 1.
inline void
fatal(const string & message)
{
  cerr << "Erreur fatale, message : " << message << '.' << endl ;
  exit(1) ;
}
// Type qui représente une instruction.
class Inst {
  size_t code_ ;  // Code de l'instruction.
  size_t taille_ ; // Taille de l'instruction.
  public :
  // Constructeur normal.
  Inst(const size_t & code, const size_t & taille) : code_(code), taille_(taille) {}
  // Constructeur de copie.
  Inst(const Inst & arg) : code_(arg.code()), taille_(arg.taille()) {}
  // Constructeur par défaut.
  Inst() { fatal("Erreur interne, appel du constructeur par défaut de 'Inst'") ; }
  size_t code() const { return code_ ; }    // Lecture de l'attribut 'code_'.
  size_t taille() const { return taille_ ; }  // Lecture de l'attribut 'taille_'.
} ;
// Code des différentes opérations.
const size_t ASE = 0 ;      // Appel du système d'exploitation.
const size_t CRG = 1 ;      // Charger le registre général.
const size_t SRG = 2 ;      // Stocker le registre général.
const size_t MOINS = 3 ;    // Soustraire.
const size_t BI = 4 ;       // Branchement inconditionnel.
const size_t BSDZ = 5 ;     // Branchement si différent de zéro.
const size_t CRGP = 6 ;     // Charger le registre général depuis la pile.
const size_t SRGP = 7 ;     // Stocker le registre général vers la pile.
const size_t APPEL = 8 ;    // Appeler une fonction.
const size_t RET = 9 ;      // Retourner.
const size_t DEF = 10 ;     // Définir un emplacement de mémoire.
const size_t SYN = 11 ;     // Rendre un symbole synonyme d'un naturel.

int
main()
{
  // Tableau associatif 'mnémonique de l'opération' --> 'Inst(code, taille)'.
  map<string, Inst> mnemo2inst ;
  // Initialisation de 'mnemo2inst'.
  mnemo2inst.insert(make_pair<string, Inst>("ase", Inst(ASE, 2))) ;
  mnemo2inst.insert(make_pair<string, Inst>("crg", Inst(CRG, 3))) ;
  mnemo2inst.insert(make_pair<string, Inst>("srg", Inst(SRG, 3))) ;
  mnemo2inst.insert(make_pair<string, Inst>("-", Inst(MOINS, 3))) ;
  mnemo2inst.insert(make_pair<string, Inst>("bi", Inst(BI, 3))) ;
  mnemo2inst.insert(make_pair<string, Inst>("bsdz", Inst(BSDZ, 3))) ;
  mnemo2inst.insert(make_pair<string, Inst>("crgp", Inst(CRGP, 2))) ;
```

```
58  mnemo2inst.insert(make_pair<string, Inst>("srgp", Inst(SRGP, 2))) ;
59  mnemo2inst.insert(make_pair<string, Inst>("appel", Inst(APPEL, 3))) ;
60  mnemo2inst.insert(make_pair<string, Inst>("ret", Inst(RET, 1))) ;
61  mnemo2inst.insert(make_pair<string, Inst>("déf", Inst(DEF, 1))) ;
62  mnemo2inst.insert(make_pair<string, Inst>("syn", Inst(SYN, 0))) ;
63
64  typedef map<string, string> Map_str_str ; // Pour se simplifier la vie.
65  // Tableau associatif : Table des symboles 'symbole' --> 'valeur du symbole'.
66  Map_str_str table_symboles ;
67
68  // Vecteur qui mémorise les deux ou trois items de chaque ligne lue.
69  vector< vector<string> > lignes_lues ;
70
71  // Première passe ; validation des instructions et récolement des symboles.
72  size_t adresse = 0 ;              // Adresse courante dans le programme.
73  string ligne ;                    // Ligne courante lue depuis 'cin'.
74  while ( getline(cin, ligne) ) { // 'false' si fin du fichier (ou si err. de lecture).
75    vector<string> items ;      // Vecteur pour stocker les deux ou trois items de la ligne.
76    // La ligne lue devient un 'istringstream', c'est-à-dire un 'fichier en mémoire'.
77    istringstream lecture(ligne) ;
78    string item ;
79    // Lecture de trois champs au maximum dans la ligne, séparés par un '\t'.
80    while ( getline(lecture, item, '\t') && (items.size() < 3) ) {
81      items.push_back(item) ; }
82    // Les lignes vides sont ignorées ; de même les lignes qui débutent par '#'.
83    if ( items.empty() || (items[0][0] == '#') ) {
84      continue ; }
85    if ( items.size() < 2 ) {
86      cerr << ligne << endl ;
87      fatal("Ligne, comportant moins de deux items, invalide") ; }
88    // Les deux ou trois premiers items sont stockés dans 'lignes_lues'.
89    lignes_lues.push_back(items) ;
90    // Le deuxième item 'items[1]' est le mnémonique de l'opérateur.
91    map<string, Inst>::const_iterator itr = mnemo2inst.find(items[1]) ;
92    if ( itr != mnemo2inst.end() ) {
93      // L'étiquette est égale à l'adresse courante, sauf pour l'opération 'syn' où
94      // l'étiquette est égale à l'opérande. 'itr->second' désigne l'instance de 'Inst'
95      // qui a été retrouvée.
96      if ( itr->second.code() != SYN ) {
97        // On ne renseigne la table des symboles que si l'étiquette n'est pas vide.
98        if ( ! items[0].empty() ) {
99          // Une adresse est codée sur deux positions.
100         string temporaire ;
101         temporaire.push_back('0' + adresse/10) ;
102         temporaire.push_back('0' + adresse%10) ;
103         table_symboles[items[0]] = temporaire ; } }
104     else {
105       if ( (items.size() != 3) || items[0].empty() ) {
106         cerr << ligne << endl ;
107         fatal("Ligne invalide : " + items[0] + '\t' + items[1]) ; }
108       table_symboles[items[0]] = items[2] ; }
109     // L'adresse est incrémentée en raison de la taille de l'instruction.
110     adresse += itr->second.taille() ; }
111   else {
112     cerr << ligne << endl ;
113     fatal("Opérateur '" + items[1] + "' non reconnu") ; }
114 } // Fin du 'while ( getline(cin, ligne) )'.
115
116 // Deuxième passe ; le code exécutable est engendré sur 'cout'.
117 for ( size_t i = 0 ; i < lignes_lues.size() ; ++ i ) {
118   // On renouvelle la recherche (qui doit réussir) du mnémonique de l'opérateur. C'est
119   // pour cela que l'on utilise la syntaxe 'mnemo2inst[clef]'. Si, à l'exécution, le
120   // constructeur par défaut est appelé c'est une erreur interne. C'est pour cela que
121   // le constructeur par défaut de 'Inst' engendre une erreur fatale quand il est
122   // appelé.
123   Inst inst = mnemo2inst[lignes_lues[i][1]] ;
124   // Ne pas engendrer du code pour la pseudo-instruction 'syn'.
125   if ( inst.code() == SYN ) {
126     continue ; }
127   // L'instruction 'ret' n'a pas d'opérande.
128   if ( inst.code() == RET ) {
129     cout << RET << endl ;
130     continue ; }
131   // Il faut s'assurer qu'un opérande a bien été spécifié.
132   if ( lignes_lues[i].size() != 3 ) {
133     fatal("Ligne invalide : " + lignes_lues[i][0] + '\t' + lignes_lues[i][1]) ; }
134   // L'opération 'déf' n'engendre pas de code, seulement une valeur initiale.
135   if ( inst.code() == DEF ) {
136     cout << lignes_lues[i][2] << endl ;
137     continue ; }
138   // Les autres opérations comportent un opérande, le rechercher dans la table des
139   // symboles.
140   Map_str_str::const_iterator itr = table_symboles.find(lignes_lues[i][2]) ;
141   if ( itr == table_symboles.end() ) {
142     fatal("Symbole '" + lignes_lues[i][2] + "' indéfini") ; }
143   cout << inst.code() << ' ' << itr->second << endl ; }
144
145 // Troisième passe ; l'impression de contrôle est engendrée sur 'cerr'.
146 adresse = 0 ;
147 cerr << "Adresse\tCode\tExplication" << endl ;
```

```
148  for ( size_t i = 0 ; i < lignes_lues.size() ; ++ i ) {
149    Inst inst = mnemo2inst[lignes_lues[i][1]] ;
150    string a ; // 'a' désigne soit l'adresse soit, pour une instruction 'ase', le code.
151    switch ( inst.code() ) {
152      case ASE : a = table_symboles[lignes_lues[i][2]] ;
153      cerr << adresse << '\t' << ASE << ' ' << a << '\t' <<
154        "Appeler le système d'exploitation." << endl ;
155      switch ( a[0] ) {
156        case '0' : cerr << "\t\t\t0 est le code 'Fin du programme'." << endl ;
157        break ;
158        case '1' : cerr << "\t\t\t1 est le code 'Lecture d'un entier depuis "
159          "std::cin'." << endl ;
160        break ;
161        case '2' : cerr << "\t\t\t2 est le code 'Écriture d'un entier vers "
162          "std::cout'." << endl ;
163        break ;
164        default : cerr << "\t\t\t" << a << " est un code inconnu." << endl ;
165      }
166      break ;
167
168      case CRG : a = table_symboles[lignes_lues[i][2]] ;
169      cerr << adresse << '\t' << CRG << ' ' << a << '\t' <<
170        "Charger le registre général." << endl ;
171      cerr << "\t\t\t" << a << " est l'adresse de " << lignes_lues[i][2] << '.' <<
172        endl ;
173      break ;
174
175      case SRG : a = table_symboles[lignes_lues[i][2]] ;
176      cerr << adresse << '\t' << SRG << ' ' << a << '\t' <<
177        "Stocker le contenu du registre général." << endl ;
178      cerr << "\t\t\t" << a << " est l'adresse de " << lignes_lues[i][2] << '.' <<
179        endl ;
180      break ;
181
182      case MOINS : a = table_symboles[lignes_lues[i][2]] ;
183      cerr << adresse << '\t' << MOINS << ' ' << a << '\t' << "Soustraire." << endl ;
184      cerr << "\t\t\t" << a << " est l'adresse de " << lignes_lues[i][2] << '.' <<
185        endl ;
186      break ;
187
188      case BI : a = table_symboles[lignes_lues[i][2]] ;
189      cerr << adresse << '\t' << BI << ' ' << a << '\t' << "Se brancher "
190        "inconditionnellement." << endl ;
191      cerr << "\t\t\t" << a << " est l'adresse de l'instruction à exécuter, étiquetée "
192        << lignes_lues[i][2] << '.' << endl ;
193      break ;
194
195      case BSDZ : a = table_symboles[lignes_lues[i][2]] ;
196      cerr << adresse << '\t' << BSDZ << ' ' << a << '\t' <<
197        "Sauter si le registre général n'est pas égal à zéro." << endl ;
198      cerr << "\t\t\t" << a << " est l'adresse de l'instruction à exécuter, "
199        "étiquetée " << lignes_lues[i][2] << '.' << endl ;
200      break ;
201
202      case CRGP : a = table_symboles[lignes_lues[i][2]] ;
203      cerr << adresse << '\t' << CRGP << ' ' << a << '\t' <<
204        "Charger le registre général depuis la pile." << endl ;
205      cerr << "\t\t\t" << a << " est le déplacement qui désigne la variable "
206        "locale " << lignes_lues[i][2] << '.' << endl ;
207      break ;
208
209      case APPEL : a = table_symboles[lignes_lues[i][2]] ;
210      cerr << adresse << '\t' << APPEL << ' ' << a << '\t' <<
211        "Appeler une fonction." << endl ;
212      cerr << "\t\t\t" << a << " est l'adresse de la première instruction à exécuter, "
213        "étiquetée " << lignes_lues[i][2] << '.' << endl ;
214      break ;
215
216      case RET : cerr << adresse << '\t' << RET << '\t' << "Retourner." <<
217        endl ;
218      break ;
219
220      case DEF : cerr << adresse << '\t' << lignes_lues[i][2] << '\t' <<
221        "Symbole " << lignes_lues[i][0] << '.' << endl ;
222      break ;
223
224      case SYN : break ;
225
226      default : cerr << "Erreur interne, prévenir F. Legendre" << endl ;
227    } // Fin du 'switch ( inst.code() ) {'
228   adresse += inst.taille() ; }
229
230   return 0 ; // Retour au système d'exploitation avec le code '0'.
231 }            // Fin de 'main(...)'
```

3 Compilation et utilisation du programme

Le programme est compilé au moyen de la commande suivante.

g++ Projet_assembleur_UNIC.cpp --output as

où ***as*** est donc le nom que l'on donne au programme exécutable qui réalise l'assemblage d'un programme écrit en assembleur UNIC.
Soit le fichier ***unic_1.unc*** qui contient le premier programme étudié en langage d'assemblage. Ce programme est assemblé au moyen de la commande suivante.

as <unic_1.unc >unic_1.exe 2>unic_1.txt

où ***unic_1.exe*** est le nom que l'on donne au programme exécutable qui utilise le jeu d'instructions UNIC et où ***unic_1.txt*** est le nom du fichier qui va contenir l'impression de contrôle.

En effet, le caractère « **<** » permet de relier le contenu d'un fichier au flux d'entrée du programme. Aussi les caractères du fichier ***unic_1.unc*** seront-ils lus par le programme sur le flux d'entrée ***cin*** comme s'ils avaient été entrés au clavier. Le caractère « **>** » permet lui de rediriger la sortie standard vers un fichier. Les caractères placés sur le flux de sortie ***cout*** figureront ainsi dans le fichier ***unic_1.exe***. Enfin, les deux caractères « ***2>*** » permettent de rediriger le flux de sortie des erreurs ***cerr***.

Sur mes pages personnelles sur Internet, les fichiers ***Projet_assembleur_UNIC.cpp***, ***unic_1.unc***, ***unic_2.unc***, ***unic_3.unc*** et ***unic_4.unc*** sont librement disponibles.