Compilation

25 février 2011

Résumé

Les étudiants de la troisième année de licence d'informatique de la faculté des sciences et techniques de l'université de Toulon ont été sollicités pour la rédaction de ce document au format LaTeX.

Contributions : Julien Pivi, Philippe Langevin.

Table des matières

1 Introduction

Ph. Langevin 21 Octobre 2009

La compilation est acte informatique fondamental qui consiste à traduire un programme *source* en un programme *cible*.

L'objectif du cours est de fournir une initiation à la pratique des outils de compilation présensent sur le système LINUX : `flex` et `bison`.

L'ensemble des techniques est décrit dans le "dragon" [1].

2 Ordinapoche

2.1 Machine fictive

Le cours d'initiation à l'informatique de Christian Nguyen [2] s'appuie sur une machine fictive : l'ordinapoche qui comprend 16 instructions. Dans ce cours, les étudiants doivent traduire un programme en langage algorithmique, le langage du I11, en une suite d'instructions pour l'ordinapoche.

```
1  algorithme somme des N premiers entiers
2  declaration
3     n, i, S des nombres entiers
4  debut
5    demander-une-valeur-pour n
6    affecter 0 a S
7    affecter 1 a i
8    tantque (i ≤ n) faire
9       affecter i+S a S
10      affecter i+1 a i
11   fintq
12   montrer-la-valeur-de S
13 fin
```

Un compilateur pour le langage du I11 doit traduire le programme en une suite de mots machine prête à être exécutée par l'ordinapoche Tab. (**??**). La compilation passera par une phase intermédiaire

```
1  .data
2          i 0
3          S 0
4          n
5  .text
6          INP n
7  .L1
8          CLA i
9          TMP n
10         LTH S1
11         CLA S
12         ADD i
13         STO S
14         CLA i
15         ADD $1
16         STO i
17         JMP .L1
18         CLA i
19         TMP n
20         LTH
21 .S1
22         OUT S
23         HRS
```

2.2 Jeu d'instructions

Nous allons décrire, dans les paragraphes suivants, le rôle de ces instructions ainsi que leur principe de fonctionnement.

⇒ Ameliorer les explications

Tab. 1 – Somme des n premiers entiers en langage machine

adresse	code op	opérande	mot machine	commentaire
	00	00F	000000001111	
	01	20D	001000001101	
	02	A0F	101000001111	
	03	B0E	101100001110	
	04	20E	001000001110	
	05	40D	010000001101	
	06	30E	001100001110	
	07	20D	001000001101	
	08	410	010000010000	
	09	30D	001100001101	
	0A	701	011100000001	
	0B	10E	000100001110	
	0C	9	100100000000	bourrage
00001101	0D	0		variable i
00001110	0E	0		S
00001111	0F			n
00010000	10	1		constante

Tab. 2 – Les instructions de l'ordinapoche

Code	mnémo.	Signification
0	INP	entrée d'une donnée depuis un périphérique d'entrée
1	OUT	affichage d'une donnée sur un périphérique de sortie
2	CLA	chargement du contenu de la cellule mémoire
3	STO	sotckage du contenu de l'ACC en mémoire centrale
4	ADD	addition du contenu de la cellule mémoire à l'ACC
5	SUB	soustraction du contenu de la cellule mémoire à l'ACC
6	SHT	décalage à gauche (du 1er chiffre) puis à droite (du 2nd chiffre) du contenu de l'ACC
7	JMP	branchement inconditionnel à l'adresse fournie
8	TAC	si (ACC = 0) alors instruction suivante sinon CO=adresse fournie
9	HRS	fin du programme et remise à zéro des registres
A	TMP	mémorisation dans le registre TMP du contenu de la cellule mémoire
B	LTH	si (ACC < TMP) alors instruction suivante sinon CO=adresse fournie
C	GTH	si (ACC > TMP) alors instruction suivante sinon CO=adresse fournie
D	CAL	appel d'une sous-routine à l'adresse fournie et stockage de CO dans le registre ADR
E	RET	poursuite du programme à l'adresse contenue dans le registre ADR
F		non définie

INP instruction permettant la lecture d'une donnée (nécessairement numérique, d'une valeur comprise entre 0 et 999) à partir du périphérique d'entrée, ici le clavier. Pour se faire, les actions suivantes vont se succéder : 1) positionnement du bus d'adresse à l'adresse correspondant au périphérique d'entrée, 2) positionnement du bus d'adresse de la mémoire centrale à l'adresse associée à l'instruction, 3) lecture de la donnée et mémorisation à l'adresse associée à l'instruction. Exemple : INP 50 (codée 050) mémorise le nombre tapé au clavier à l'adresse 50.

OUT instruction permettant l'affichage d'une donnée sur le périphérique de sortie qu'est l'écran. Les étapes sont les mÃªmes que pour l'instruction INP précédente à ceci prés que l'on adresse l'écran au lieu du clavier et que la donnée transite de la mémoire principale au périphérique de sortie. Exemple : OUT 51 (codée 151) affiche à l'écran la donnée mémorisée à l'adresse 51.

CLA instruction initialisant le registre accumulateur (ACC) de l'unité arithmétique et logique (UAL) à la valeur fournie en argument de cette instruction. Etapes : 1) mise à zéro du registre ACC, 2) positionnement du bus d'adresse à l'adresse correspondant au registre ACC, 3) positionnement du bus d'adresse de la mémoire centrale à l'adresse associée à l'instruction, 4) lecture de la donnée et mémorisation dans le registre ACC. Exemple : CLA 52 (codée 252) initialise ACC au nombre contenu à l'adresse 52.

STO instruction mémorisant, en mémoire centrale, le contenu du registre ACC à l'adresse fournie en argument de l'instruction. Les étapes sont similaires à celles de l'instruction CLA, sans initialisation du registre ACC bien entendu. Exemple : STO 53 (codée 353) stocke le contenu d'ACC à l'adresse 53.

ADD (resp. SUB) : instruction d'addition (resp. de soustraction) du contenu de la cellule mémoire dont l'adresse est fournie en argument au contenu du registre accumulateur de l'UAL. Les étapes sont similaires à celles de l'instruction CLA. Exemple : ADD 53 (codée 453) (resp. SUB 53) ajoute (resp. soustrait) le contenu de la cellule mémoire d'adresse 53 au registre accumulateur (ce qui correspond, si on suit l'ordre des exemples, à une multiplication pas deux dans le cas de l'addition).

SHT instruction de décalage d'une unité vers la gauche puis vers la droite du nombre contenu dans le registre accumulateur, avec entrée de zéro à droite puis à gauche. Le but est d'obtenir trÃ¨s rapidement la multiplication et/ou la division par 10 (ou par 16 en hexadécimal) du nombre contenu dans l'ACC. Aucune étape particuliÃ¨re, si ce n'est la positionnement du bus d'adresse à l'adresse correspondant au registre ACC. Exemple : SHT 12 (codée 612) du nombre 12 donne comme résultat 002 soit 2. JMP : saut inconditionnel (i.e sans condition) à l'adresse fournie en argument de l'instruction. Cette instruction initialise le compteur ordinal à cette adresse. Aucune étape particuliÃ¨re, si ce n'est l'initialisation de CO. Exemple : JMP 60 (codée 960) initialise CO à la valeur (adresse) 60.

TAC comparaison du contenu du registre accumulateur (ACC) et du nombre 0. Si leur valeur sont égales, la valeur du compteur ordinal est augmentée de 1 comme d'habitude, si par contre elles diffÃ¨rent, le CO prend la valeur fournie avec l'instruction. Etapes : 1) positionnement du bus d'adresse à l'adresse correspondant au registre ACC, 2) comparaison entre ACC et 0 puis retour du résultat (un booléen) à l'unité de commande, 3) mise à jour du compteur ordinal (CO). Exemple : TAC 70 (codée A70), initialise CO à 70 si ACC != 0.

HRS instruction de fin de programme. Aucune étape particuliÃ¨re.

TMP instruction initialisant le registre temporaire (TMP) de l'unité arithmétique et logique (UAL) à la valeur fournie en argument de cette instruction. Les étapes sont similaires à celles de l'instruction CLA. Exemple : TMP 53 (codée 853) initialise TMP au contenu de la cellule mémoire d'adresse 53.

LTH (resp. GTH) : comparaison du contenu du registre accumulateur (ACC) et du contenu du registre TMP. Si la valeur de ACC est inférieure (resp. supérieure) à TMP alors la valeur du compteur ordinal est augmentée de 1 comme d'habitude, sinon CO prend la valeur fournie avec l'instruction. Etapes : similaires à celles de l'instruction TAC. Exemple : LTH 80 (codée B80), initialise CO à 80 si ACC < TMP. Attention : le registre TMP doit Ã[a]tre initialisé, par l'instruction TMP, avant le test !

CAL instruction initialisant le registre de base des adresses (ADR) de l'unité de commande (UC) à la valeur d'adresse de l'instruction et qui poursuit l'exécution du programme à l'adresse associée à l'instruction. Exemple : 20 CAL 50 initialise ADR à la valeur (adresse) 20 puis effectue un saut inconditionnel à l'adresse 50. Attention : une instruction CAL ne peut pas Ã[a]tre imbriquée dans une autre instruction CAL.

RET instruction utilisant le contenu du registre de base des adresses (ADR) pour poursuivre l'exécution du programme à l'adresse mémorisée par ce registre, en initialisant pour ce faire le registre CO. Exemple : RET (codée E00).

2.3 Limitation

L'ordinapoche est une machine fictive qui se distingue fortement des machines réelles sur plusieurs points.

⇒ Developper : CISC, RISC etc. . .

1. Son jeu d'instructions est vraiment petit. Des opérations élémentaires comme la multiplication, la division et la majorité des opéateurs logiques ne sont pas définis.
2. Il n'y a pas d'adressage indexé ce qui rend difficile l'implantation de la notion de tableau.
3. Il n'y a pas d'adressage indirect ce qui rend difficile l'implantation des pointeurs, et de la notion de tas.
4. Elle ne possède pas de pile. Les notions de procédures et fonctions seront limités. La récursivité difficile à mettre en oeuvre.

Finalement, les programmes du langage du I11, incluront des sous-programmes (routine), en exécution non concurrente, et les variables seront toutes de type scalaires.

2.4 Zones mémoires

La construction automatique d'un programme implique l'usage de règles concernant la carte mémoire d'un programme. Le plus souvent un processus utilise au moins 4 zones mémoires : données, code, pile, et tas. La pile est utilisée pour le calcul des expressions et lors de l'exécution des sous-programmes. Dans le cas de l'ordinapoche, il n'y a pas de pile. Les donn ées temporaires (sont placées dans des zones de registres virtuels, une zone pour le programme principal, une pour le sous-progamme en cours. La transmission des paramètres et des résultats d'un sous-programme est faite au travers de zones adéquates. Notons de suite que les zones de constantes, registres, paramètres, et résultats seront transparentes pour le programmeur, leurs caractéristiques (adresse et taille) seont calculées lors de l'assemblage.

zone	description
constante	la zone des constantes contient les constantes du programme
registre	zone des variables temporaires utilisées par le programme pour calculer les expressions
registre	zone des variables temporaires utilisées par les sous-programmes pour calculer les expressions
paramètre	utilisée pour la transmission des paramètres des sous-programmes
résultat	utilisée par les sous-programmes pour la transmissions des résultats
data	Elle contient les variables utilisateurs
text	La zone de text contient les instructions.

2.5 Simulateur

Un programme `ordinapoche.c` est écrit pour obtenir une commande `ordinapoche` qui prend un fichier source `foo.bin` contenant une liste de mots machine. Le premier mot indique le point d'entrée dans le programme, les autres des données et instructions. Notons qu'un mot machine est écrit sur 3 demi-octets. Si abc et def sont les deux premiers mots machine du programme ordinapoche, les trois premiers octets du fichier binaire seont $a*16+b$, $c*16+d$ et $e*16+f$. Un bourrage (padding) à FF est utilisé pour les binaires ayant un nombre impair de mot machine.

1. Lecture de la valeur initiale du pointeur d'instruction.
2. Chargement du programme en mémoire.
3. Exécution du code.

Plusieurs options permettront l'exécution en mode pas à pas, en mode normal ou encore pédagogique.

Une option de lancement permet d'obtenir un fichier de nombres `foo.txt` au format texte qui pourra être utilisé avec l'interprète `ordinapoche.tcl` de Christian Nguyen, consulter le site pour le mode d'emploi.

2.6 sources

Le simulateur d'Emilien Royer.

```
//Emilien Royer, Mars 2010
//

#include "header.h"

void
Initialisation ()
{
  R_CO = 0;
  R_ADR = 0;
  R_TMP = 0;
  R_ACC = 0;
}

void
ExecuteInstruction (unsigned char code_op, unsigned char operande)
{
```

```
/*
   RecupÃ¨re le code d'une instruction et l'opÃ©rande associÃ©e avec, puis execute
   l'instruction correspondante. Voir instruction.c pour le detail des
   operations effectuÃ©es.
*/
  switch (code_op)
    {
    case INP:
      Input (operande);
      break;
    case OUT:
      Output (operande);
      break;
    case CLA:
      ClearAndAdd (operande);
      break;
    case STO:
      Store (operande);
      break;
    case ADD:
      Add (operande);
      break;
    case SUB:
      Sub (operande);
      break;
    case SHT:
      Shift ();
      break;
    case JMP:
      Jump (operande);
      break;
    case TAC:
      TestAccContent (operande);
      break;
    case HRS:
      HaltAndReset ();
      break;
    case TMP:
      Temp ();
      break;
    case LTH:
      LessThan (operande);
      break;
    case GTH:
      GreaterThan (operande);
      break;
    case CAL:
      CallSub (operande);
      break;
```

```
    case RET:
      Return ();
      break;
    }
}

void
PrintOpCode (unsigned char code_op)
{
  /*
     RecupÃ¨re le code d'une instruction et l'opÃ©rande associÃ©e avec, puis execute
     l'instruction correspondante. Voir instruction.c pour le detail des
     operations effectuÃ©es.
   */
  switch (code_op)
    {
    case INP:
      printf ("INP");
      break;
    case OUT:
      printf ("OUT");
      break;
    case CLA:
      printf ("CLA");
      break;
    case STO:
      printf ("STO");
      break;
    case ADD:
      printf ("ADD");
      break;
    case SUB:
      printf ("SUB");
      break;
    case SHT:
      printf ("SHT");
      break;
    case JMP:
      printf ("JMP");
      break;
    case TAC:
      printf ("TAC");
      break;
    case HRS:
      printf ("HRS");
      break;
    case TMP:
      printf ("TMP");
      break;
```

```
    case LTH:
      printf ("LTH");
      break;
    case GTH:
      printf ("GTH");
      break;
    case CAL:
      printf ("CAL");
      break;
    case RET:
      printf ("RET");
      break;
    }
}

void
Execution (unsigned char beg, char s)
{
  /*
     Execute le programme: L'adresse qui initialise le compteur ordinal est
     fournie en argument (cf LoadInMemory).
   */
  R_CO = beg;
  unsigned int buff;            // entier non signÃ© car 3 demi octets ne peuvent pas
  // Ãªtres contenus dans un octet...
  unsigned char code_op, operande;

  /*
     A chaque passe dans la boucle, on rÃ©cupere l'entier contenu dans
     l'adresse courante, et grace Ã  des operations de dÃ©calage, on extrait
     le code d'opÃ©ration, puis l'opÃ©rande.
   */
  while (1)
    {
      buff = memoire[R_CO];     // Extraction du code d'opÃ©ration
      code_op = buff >> 8;

      buff = buff << 24;        // Extraction de l'opÃ©rande (on opÃ¨re sur 32 bits,
      operande = buff >> 24;

      /*
         Option du mode pas Ã  pas, affiche les diffÃ©rents registres et
         l'action en cours
       */
      if (s)
        {
          printf ("R_CO:␣%x\t", R_CO);
          PrintOpCode (code_op);
          printf ("␣␣%x\t\t", operande);
```

```
            printf ("R_ACC: %x R_ADR: %x R_TMP: %x\n", R_ACC, R_ADR, R_TMP);
            getchar ();           // Attend une entrÃ©e au clavier avant de continuer
          }
        ExecuteInstruction (code_op, operande);
        R_CO++;
      }
}

unsigned char
LoadInMemory (char *file_name)
{
/*
   Charge le code executable contenu dans le fichier binaire indiquÃ© en argument
   dans la mÃ©moire de l'ordinapoche.
   De plus, renvoit l'adresse Ã  laquelle le programme s'execute (variable beg).
*/
  int boucle = 0;
  unsigned int saisie;
  unsigned int buff, tempo;
  unsigned char beg;
  FILE *file;

  /* on ouvre le fichier binaire en lecture, constituÃ© de la forme suivante:
     Chaque octet contient 2 mots (codÃ©s sur 4 bits, le maximum est donc 15).
     Par exemple, pour l'instruction ordinapoche INPUT 0x58, nous avons donc
     0 5 8, qui se prÃ©sente sous la forme d'un premier octet: 00000101 et d'un
     deuxieme octet: 1000????. (???? Ã©tant le premier mot de la prochaine
     instruction) il s'agit donc de decomposer les octets pour recuperer ces
     mots.
   */
  printf ("%s\n", file_name);
  file = fopen (file_name, "rb");

  buff = fgetc (file);
  R_CO = buff;
  beg = buff;

  printf ("R_CO: %x\n", R_CO);

  /* Le principe de la boucle est le suivant:
     Il y a deux cas Ã  differencier, prenons l'exemple suivant:
     a b c
     d e f
     g h i
     1) Le premier octet lu contiendra les mots a et b, ce qui est insuffisant
     pour executer une instruction car il nous manque c pour avoir l'opÃ©rande.
     Il nous faut donc lire l'octet suivant, qui contient c et d. On sÃ©pare
     donc c, et on execute l'instruction. On garde d de cÃ´tÃ© pour le prochain
     passage dans la boucle.
```

```
      2) deuxieme passage: Nous avons dÃ©jÃ  le code de l'instruction, il reste Ã
      connaitre l'opÃ©rande. Il suffit donc de lire un octet. Le prochain passage
      sera identique au premier, d'ou le switch avec un indice modulo 2.
   */

  while (saisie != EOF)
    {
      switch (boucle % 2)
        {

        case 0:
          buff = fgetc (file);
          buff = buff << 4;
          tempo = fgetc (file);
          saisie = tempo;
          buff = buff + (tempo >> 4);
          memoire[R_CO] = buff;
          break;

        case 1:
          buff = tempo << 28;
          buff = buff >> 20;
          tempo = fgetc (file);
          buff = buff + tempo;
          saisie = tempo;
          memoire[R_CO] = buff;
          break;
        }
       boucle++;
       R_CO++;
    }
  printf ("\n");
  fclose (file);

  return beg;
}

void
Dump (char *file_name)
{
  /*
     CrÃ©e un fichier "logs" qui contient une version texte du code executable
     Pour comprendre les mÃ©canismes, se referer aux commentaires de la
     fonction LoadInMemory.
   */
}

void
ReadMemory (char *file_name, unsigned char end)
```

```
{
/*
   Fonction de dÃ©bugage, permet de lire la mÃ©moire Ã  partir d'une adresse donnÃ©e
   afin de verifier qu'il n'y a pas eu de problÃ¨me lors du chargement en mÃ©moire
   du code.
*/
  FILE *file;
  unsigned int buff;
  unsigned int tempo;

  file = fopen (file_name, "rb");
  buff = fgetc (file);
  R_CO = buff;
  printf ("R_CO: %d\n", R_CO);

  while (end > 0)
    {
      buff = memoire[R_CO];
      printf ("%d\t", buff);
      printf ("%x|", buff >> 8);
      tempo = buff << 24;
      tempo = tempo >> 24;
      printf ("%x", tempo);
      printf ("\n");
      R_CO++;
      end--;
    }

  fclose (file);
}

void
PrintHelp ()
{
/*
    Affiche sur la sortie standart les diffÃ©rentes options de lancement du
    programme
*/
  printf ("\nOptions available:\n");
  printf ("\t-s (Step mode, useful for debugging purpose)\n");
  printf
    ("\t-d (Dump an ASCII text file from a binary, not yet available)\n");
  printf ("\n");
}

int
main (int argc, char *argv[])
{
  int opt;
```

```
  char s = 0;
  char *optlist = "hsd";
  unsigned char beg;

  if (argc < 2)
    {
      printf ("Usage: %s Binaryfilename\n\n", argv[0]);
      printf ("\t-h (Help)\n");
      exit (1);
    }

  while ((opt = getopt (argc, argv, optlist)) > 0)
    switch (opt)
      {
      case 'h':
        PrintHelp ();
        exit (1);
      case 's':
        s = 1;
        break;
      case 'd':
        Dump (argv[argc - 1]);
        exit (1);
      default:
        printf ("Unkown option: %c\n", opt);
        printf ("\t-h (Help)\n");
        exit (1);
      }

  Initialisation ();
  beg = LoadInMemory (argv[argc - 1]);
  printf ("\n");
  Execution (beg, s);
  return 0;
}
```

3 Assembleur

3.1 Langage

Un programme en assembleur doit pouvoir être traduit sans difficulté en langage machine par un assembleur tout en étant lisible par un humain. Il est découpé en deux zones : les données et le code. On adopte une syntaxe assez proche de l'assembleur ATT. Un identificateur usuel désigne une variable, quelques caractères spéciaux préfixes des classes de symboles Tab. (3).

Le programme assembleur parcourt une source en langage assembleur `prog.asm` pour déterminer la taille des différentes zones (constante, registres, variable, paramètre résultat), puis la valeur des différentes étiquettes. Une fois déterminé ces valeurs, il produit un exécutable pour l'ordinapoche `prog.bin`.

TAB. 3 – Les caractès préfixes

.	étiquette (label)	assembleur ATT
$	constante	
%	registre virtuel	
#	un argument de sous-programme	langage TEX
@	un résultat de sous-programme	Pascal

```
                sdc.asm
1   .data
2           i
3           n
4           s
5   .text
6   .routine prod( x , y )
7           CLA $0
8           STO @0
9   .L1 :
10          CLA x
11          TAC .S1
12          CLA @0
13          ADD y
14          STO @0
15          CLA x
16          SUB $1
17          STO x
18          JMP .L1
19  .S1 :
20          RET
21
22  .main
23          INP n
24          CLA $0
25          STO s
26          CLA $1
27          STO i
28  .L2 :
29          CLA n
30          ADD $1
31          STO %1
32          CLA i
33          TMP %1
34          JLT .S2
35          CLA i
36          STO #1
37          STO #2
38          CAL prod
39          CLA @0
40          ADD s
41          STO s
42          JMP .L2
43  .S2
44          OUT s
45          HRS
```

Un tableau de paire (opération, symbole) initialisé à (NOP,NULL). Le programme est parcouru de façon séquentielle. Lors de ce parcourt, chaque symbole représentant une adresse permet une mise à jour et/ou une insértion dans la table de symboles. A la fin de ce processus, le code machine peut être produit par un parcourt du tableau des paires.

3.2 Automate

3.3 Routine

3.4 Programme

La simplicité d'un langage d'assemblage permet de réduire la programmation è celle d'un simple automate. L'outil de compilation `flex` est indiqué. La compilation de l'analyseur assemble.l se fait en deux temps :

```
flex assemble.l -o assemble.c
gcc assemble.c -o assemble -lfl
```

4 Code produit par `gcc`

Le programme `prog.c` ci-dessous

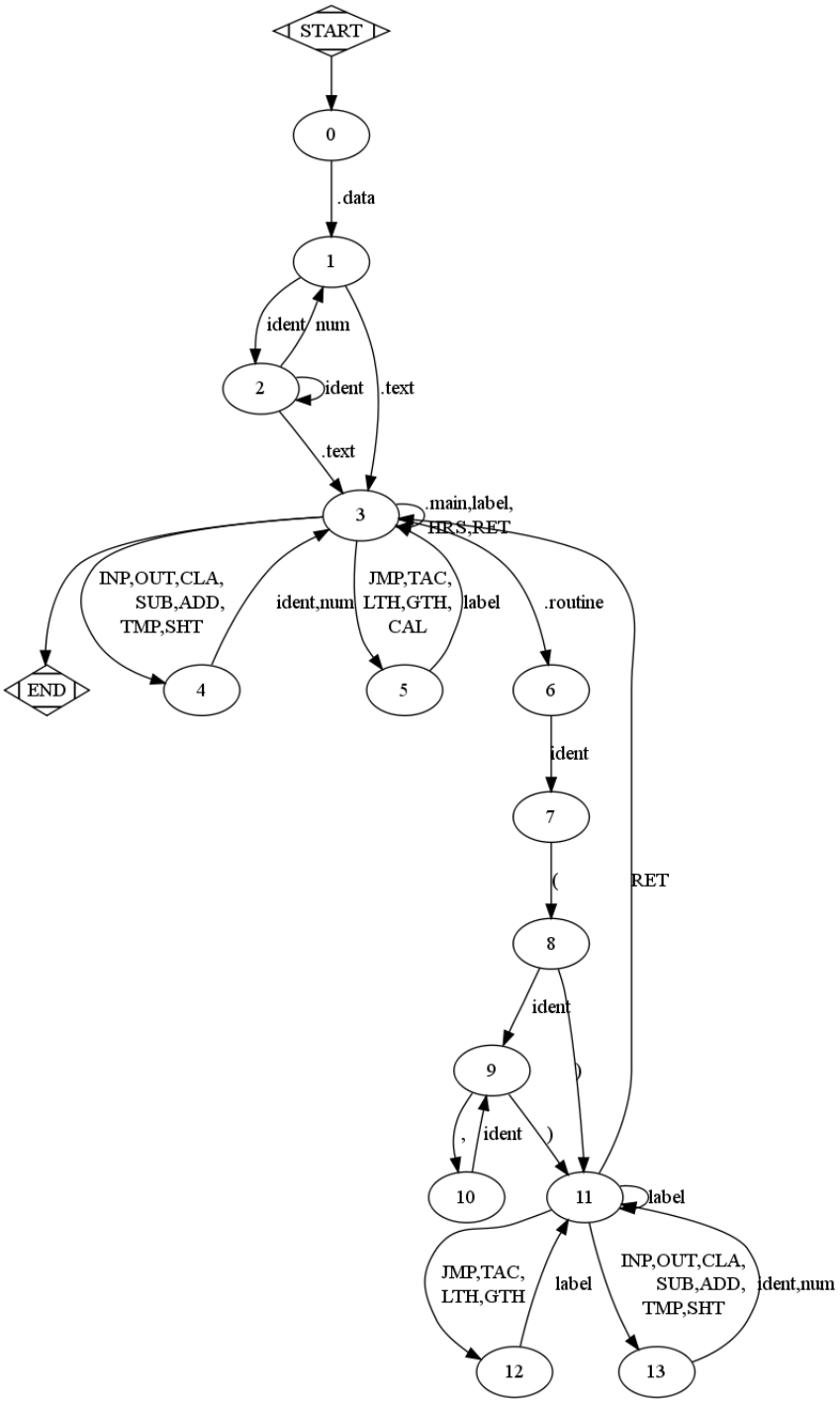

Fig. 1 – Automate.

prog.c

```
1  int proc(int x)
2  { int t[8];
3    return t[x];
4  }
5  int inf( int x, int y)
6  {
7  int t;
8  if ( x < y ) t = x;
9  else t = y;
10 return t;
11 }
12
13 int main( void )
14 {
15 int x, y, z, t;
16 int tab[10];
17 //expression arithmetique
18 t = x + y * z;
19 //expression booleenne
20 t = x > y && y > z;
21 //acces dans un tableau
22 t = tab[5];
23 //instruction iterative
24 while ( x > 0 )
25       x--;
26 //instruction alternative
27 if ( x > y )
28     z = x;
29 else
30     z = y;
31 //appel de fonction
32 z = inf(x, y);
33 //appel de proc
34 z = proc( x );
35 return 0;
36 }
```

nous permet d'illustrer la production de code du compilateur gcc. On peut obtenir un code intermédiaire en assembleur en stoppant la compilation avant la phase d'assemblage :

```
gcc -S prog.c
```

prog.s

```
1   .file "prog.c"
2   .text
3  .globl proc
4   .type proc, @function
5  proc:
6   pushl %ebp
7   movl %esp, %ebp
8   subl $32, %esp
9   movl 8(%ebp), %eax
10  movl -32(%ebp,%eax,4), %eax
11  leave
12  ret
13  .size proc, .-proc
14 .globl inf
15  .type inf, @function
16 inf:
17  pushl %ebp
18  movl %esp, %ebp
19  subl $16, %esp
20  movl 8(%ebp), %eax
21  cmpl 12(%ebp), %eax
22  jge .L4
23  movl 8(%ebp), %eax
24  movl %eax, -4(%ebp)
25  jmp .L5
26 .L4:
27  movl 12(%ebp), %eax
28  movl %eax, -4(%ebp)
29 .L5:
30  movl -4(%ebp), %eax
31  leave
32  ret
33  .size inf, .-inf
34 .globl main
35  .type main, @function
36 main:
37  pushl %ebp
38  movl %esp, %ebp
39  subl $72, %esp
40  movl -12(%ebp), %eax
41  imull -8(%ebp), %eax
42  addl -16(%ebp), %eax
43  movl %eax, -4(%ebp)
44  movl -16(%ebp), %eax
45  cmpl -12(%ebp), %eax
46  jle .L8
47  movl -12(%ebp), %eax
48  cmpl -8(%ebp), %eax
49  jle .L8
```

```
50   movl $1, %eax
51   jmp .L9
52  .L8:
53   movl $0, %eax
54  .L9:
55   movl %eax, -4(%ebp)
56   movl -36(%ebp), %eax
57   movl %eax, -4(%ebp)
58   jmp .L10
59  .L11:
60   subl $1, -16(%ebp)
61  .L10:
62   cmpl $0, -16(%ebp)
63   jg .L11
64   movl -16(%ebp), %eax
65   cmpl -12(%ebp), %eax
66   jle .L12
67   movl -16(%ebp), %eax
68   movl %eax, -8(%ebp)
69   jmp .L13
70  .L12:
71   movl -12(%ebp), %eax
72   movl %eax, -8(%ebp)
73  .L13:
74   movl -12(%ebp), %eax
75   movl %eax, 4(%esp)
76   movl -16(%ebp), %eax
77   movl %eax, (%esp)
78   call inf
79   movl %eax, -8(%ebp)
80   movl -16(%ebp), %eax
81   movl %eax, (%esp)
82   call proc
83   movl %eax, -8(%ebp)
84   movl $0, %eax
85   leave
86   ret
87   .size main, .-main
88   .ident "GCC: (GNU) 4.4.4
89  20100630 (Red Hat 4.4.4-10)"
90   .section
91  .note.GNU-stack,"",@progbits
```

Notons que `gcc` est capable d'assembler ces codes

`gcc prog.s -o prog.exe`

Notons qu'il est possible de désassembler les exécutable sous `gdb` (disassemble), ou encore avec la commande `objdump`.

4.1 Expression arithmétique

```
1   .file "prog.c"
2   .text
3  .globl proc
4   .type proc, @function
5  proc:
```

4.2 Expression booléenne

```
1   .file "prog.c"
2   .text
3  .globl proc
4   .type proc, @function
5  proc:
```

4.3 Instruction itérative

```
1   .file "prog.c"
2   .text
3  .globl proc
4   .type proc, @function
5  proc:
```

4.4 Instruction alternative

```
1   .file "prog.c"
2   .text
3  .globl proc
4   .type proc, @function
5  proc:
```

4.5 Appel de fonction

Les procédures et fonctions utilisent des conventions, on parle de *cadre de pile*, Fig. (2). Les arguments et variables locales sont adressés relativement au registre de base *BP*.

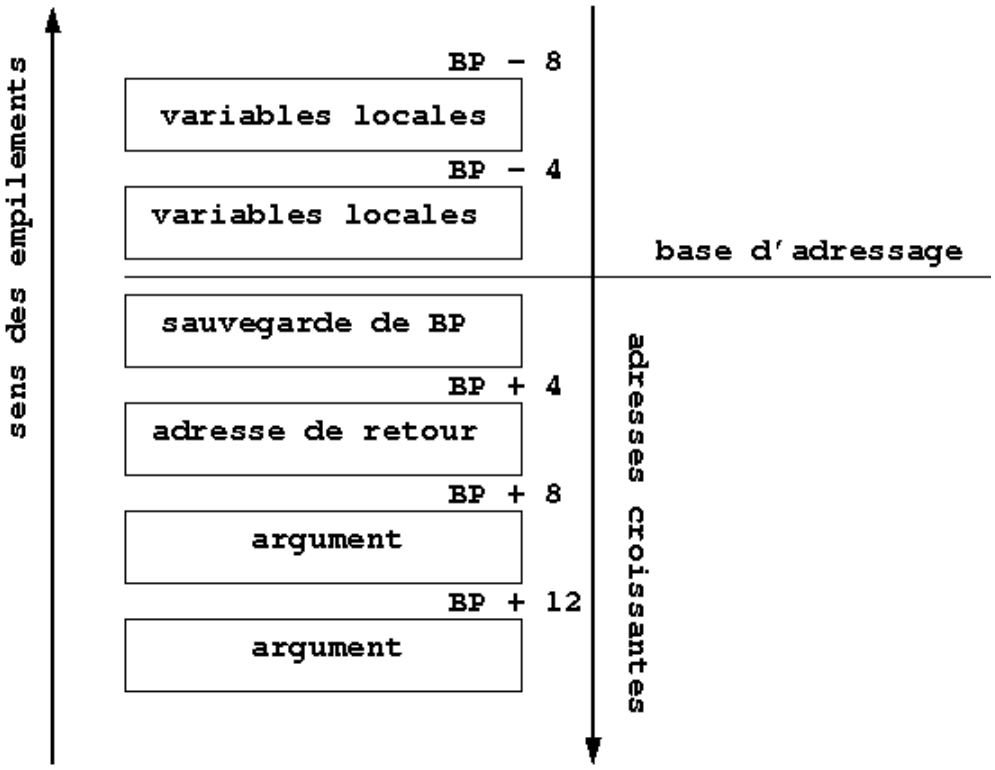

FIG. 2 – Le cadre de pile (32 bits). Les variables locales et les arguments sont adressés relativement au registre de base BP.

L'appel d'une fonction suit le processus :
- Les arguments de la fonction sont empilés.
- L'adresse de retour est empilée avant un saut 'a l'adresse de la fonction : *CALL proc*.
- Dans le prologue, le registre de base est empilé (sauvegarde) avant d'être initialisé à la valeur du pointeur de pile : *PUSH BP ; MOV SP, BP ;*.
- Le pointeur de pile est décrémenté pour réserver l'espace des variables locales : *SUB n SP ;*.
- ...
- Au cours de l'épilogue, le registre de base et le pointeur de pile sont restaurés : *LEAVE ;*, ou bien *MOV BP SP* pour récurérer le pointeur de pile, et *POP BP* pour restaurer le registe de base.
- L'adresse de retour est dépilée et les arguments éventuellement dépilés pour restaurer le pointeur de pile *RET*, *RET n*.

Le code de la fonction `inf`, fait bien apparaitre les arguments en BP+8 et BP+12, la variable locale en BP-4, le prologue sur les lignes 17,18,19 et l'épilogue sur les lignes 31, 32.

```
14  .globl inf
15   .type inf, @function
16  inf:
17   pushl %ebp
18   movl %esp, %ebp
19   subl $16, %esp
20   movl 8(%ebp), %eax
21   cmpl 12(%ebp), %eax
22   jge .L4
23   movl 8(%ebp), %eax
24   movl %eax, -4(%ebp)
25   jmp .L5
26  .L4:
27   movl 12(%ebp), %eax
28   movl %eax, -4(%ebp)
29  .L5:
30   movl -4(%ebp), %eax
31   leave
32   ret
33   .size inf, .-inf
```

et l'appel de la fonction par la fonction `main`.

```
74   movl -12(%ebp), %eax
75   movl %eax, 4(%esp)
76   movl -16(%ebp), %eax
77   movl %eax, (%esp)
78   call inf
79   movl %eax, -8(%ebp)
```

Références

[1] A.V. Aho, R. Sethi, and J.D. Ullman. *Compilers : Principles, Techniques, and Tools*. Addisson-Wesley, Reading, Massachusetts, 1986.

[2] Christian Nguyen and Valérie Gillot. Introduction à l'informatique. http://nguyen.univ-tln.fr/index.php/2009/09/10/3-introduction-a-l-informatique.