Chapitre 4

Support pour les langages de haut niveau

4.1 Compilation et interprétation

De nos jours, les programmes ne sont jamais écrits directement en langage machine — le seul langage directement compréhensible par un processeur — mais ils sont écrits dans des langages de plus haut niveau. Ils doivent donc d'une certaine façon être *traduits* dans la langue du processeur. Cette traduction peut être faite d'un seul coup sur tout le texte du programme avant que ne débute son exécution : c'est dans ce cas que l'on parle de *traduction* ou de *compilation*.

Une autre façon de procéder consiste à lire une par une les instructions du programme écrit en langage de haut niveau et à exécuter les actions décrites par chacune des instructions aussitôt qu'elle a été lue : on parle alors d'*interprétation*.

Les langages **Pascal**, **C**, **Fortran** sont souvent compilés. Les langages **Lisp**, **APL**, **Basic** sont souvent interprétés.

4.2 La vie d'un programme

4.2.1 L'édition

La création et la modification — appelées *édition* — d'un fichier texte source d'un programme se fait avec un *éditeur de texte*.

4.2.2 La compilation

La *compilation* est le terme informatique pour désigner une traduction. La traduction se fait du langage *source* vers le langage *objet*. Le langage objet est souvent du *langage machine relogeable* pour le processeur de la machine qui compile[1]

Pour le langage C, il existe de très nombreux compilateurs. L'un d'entre eux, qui produit du code de bonne qualité et qui est disponible pour de nombreux processeurs et systèmes d'exploitation est celui de GNU. Avec ce compilateur, obtenir un fichier objet se fait en indiquant l'option `-c`, exemple :

```
gcc -c foo.c
```

Dans les systèmes Unix, le fichier objet produit par cette commande est le même que celui du fichier source où toutefois le suffixe `.c` est remplacé par le suffixe `.o`.

Les compilateurs proposent de très nombreuses options pour s'adapter à toutes sortes de particularité des matériels ou pour faciliter la vie du programmeur. Parmi ces dernières citons encore une

[1] Il existe aussi des compilateurs qui s'exécutenr sur une machine de type A et produisent du code pour des machines de type B : on parle alors de *compilateur croisé* (en anglais : cross-compiler).

fois les options du compilateur de GNU pour attirer l'attention du programmeur sur toutes sortes de construction du langage C qui sont ambiguës ou douteuses :

```
gcc -c -W -Wall -Wuninitialized -O1 foo.c
```

4.2.2.1 L'assemblage

Un *assembleur* n'est rien d'autre qu'un compilateur qui traduit du *langage d'assemblage* en *langage machine*. Pour sa plus grande partie, le *langage d'assemblage* est une forme lisible du langage machine. Toutes les instructions du processeur ont un équivalent lisible au moyen de *mnémoniques* pour représenter les codes des opérations et les opérandes avec tous leurs modes d'adressage. De plus, la plupart des assembleurs permettent de définir des *macros*. Une *macro* est définie par un morceau de texte qui est nommé. Lorsque ce nom est rencontré dans le programme source, il est remplacé par le texte qu'il représente.

L'assembleur de GNU permet de produire un listing avec les options suivantes :

```
as -adnl foo.s > foo.lst
```

Notons que les compilateurs permettent d'obtenir un équivalent en langage d'assemblage des fichiers objets qu'ils produisent habituellement. Ainsi, la commande :

```
gcc -S foo.c
```

produit un fichier `foo.s`.

Ce fichier est un fichier texte lisible avec un éditeur de texte.

4.2.3 L'édition de liens

Un *fichier objet relogeable* n'est pas directement un exécutable : les appels aux fonctions de bibliothèque n'ont pas été *résolus*, ni les appels entre modules s'il y en a plusieurs. L'*éditeur de liens* (en anglais : linker) combine (*lie*) plusieurs modules pour en faire un *exécutable*.

Souvent les éditeurs de liens sont lancés par la commande de compilation. En effet, de cette façon chaque compilateur introduira les bibliothèques standard pour son langage. Par exemple, l'éditeur de lien`ld` est appelé de la façon suivante par `gcc` sur un système Solaris :

```
/usr/ccs/bin/ld -Y P,/usr/ccs/lib:/usr/lib -Qy -o foo
/usr/local/lib/gcc-lib/sparc-sun-solaris2.8/2.95.3/crt1.o
/usr/local/lib/gcc-lib/sparc-sun-solaris2.8/2.95.3/crti.o
/usr/ccs/lib/values-Xa.o
/usr/local/lib/gcc-lib/sparc-sun-solaris2.8/2.95.3/crtbegin.o
-L/usr/local/lib/gcc-lib/sparc-sun-solaris2.8/2.95.3
-L/usr/ccs/bin -L/usr/ccs/lib -L/usr/local/lib
/var/tmp/ccVA8cH4.o -lgcc -lc -lgcc
/usr/local/lib/gcc-lib/sparc-sun-solaris2.8/2.95.3/crtend.o
/usr/local/lib/gcc-lib/sparc-sun-solaris2.8/2.95.3/crtn.o
```

4.2.4 Le chargement et l'exécution

Un programme en cours d'exécution par le processeur (un *processus*) doit être présent en mémoire centrale. La mémoire doit contenir à la fois la liste des instructions à exécuter (le texte du programme en langage machine) et les données, que ce soient des données globales ou les données de la pile (les variables locales).

Le fichier *exécutable* produit par l'éditeur de liens contient une description de ce que doit être la configuration mémoire au début de l'exécution. Le texte du programme (le *segment texte*) et les données initialisées (le *segment data*) sont présents tels quels dans le fichier. Pour les données non initialisées (le *segment bss*) et la pile, seules leurs tailles sont précisées. Par ailleurs, le fichier exécutable contient éventuellement d'autres informations comme par exemple une *table des symboles* qui peut être utilisée par un *dévermineur* (en anglais : debugger).

4.3 Structures de données

4.3.1 Les types élémentaires

Les représentations des types simples (entiers, caractères, nombres flottants) ont été vues dans le chapitre 2. Dans un langage impératif "de haut niveau", ces types simples sont accessibles à travers des noms prédéfinis ou des mots-clés. Cependant, tous les types de base d'une machine donnée ne sont pas nécessairement accessibles, et réciproquement tous les types élémentaires d'un langage ne sont pas nécessairement directement implémentables sur cette machine. Il peut y avoir de grandes variations entre les représentations d'un type élémentaire d'un langage sur différentes machines, même si le langage est normalisé. L'exemple le plus fréquent concerne la longueur des entiers. Si à peu près tous les processeurs reconnaissent une représentation en complément à deux pour les entiers signés, certaines machines favorisent des entiers sur 16 bits, d'autres sur 32 bits, et on peut s'attendre à ce que très bientôt une longueur de 64 bits ne soit pas exceptionnelle. La principale difficulté engendrée par cette absence de précision rend difficile l'écriture de programmes *portables*.

4.3.2 L'alignement

Le compilateur gère les problèmes d'*alignement* : les processeurs 32 bits actuels accèdent à la mémoire avec une adresse exprimée en octets, toutefois pour que l'accès à un mot de 32 bits soit efficace (voire possible) il faut que son adresse soit divisible par 4, ce qui permet de le transférer entre la mémoire et le processeur en utilisant la totalité de la largeur du bus de données.

4.3.3 Représentation mémoire des structures

4.3.3.1 Les enregistrements

Les enregistrements sont des aggrégats d'objets de différents types qui sont manipulés par un symbole et un sélecteur de champ. En pratique, les différents champs d'un enregistrement sont stockés dans des zones contiguës de la mémoire. Cependant, l'ordre dans lequel les différents champs vont se trouver représentés en mémoire n'est pas nécessairement le même que celui de la déclaration dans le langage de haut niveau. En effet, le compilateur peut compacter plusieurs champs de petite longueur ensemble pour éviter de laisser des "trous" lorqu'il y a des contraintes d'*alignement* sur les champs plus longs. Ainsi dans l'exemple de la déclaration suivante :

```
struct E {
    char u;
    int v;
    char w;
  };
```

Si le compilateur doit *aligner* le champ de type `int` sur une adresse (en octets) divisible par 4, et qu'il ne compacte pas les deux champs de type `char`, la structure mémoire d'un enregistrement de ce type pourrait être la suivante :

u	X	X	X
v	v	v	v
w	X	X	X

avec un gaspillage de six octets. Alors qu'avec un compactage, il pourrait décider de représenter un tel enregistrement en mémoire sous la forme :

u	X	w	X
v	v	v	v

où, cette fois, seulement deux octets sont perdus. Ces deux exemples laissent entrevoir qu'il y a en fait de très nombreuses possibilités pour stocker une structure au delà du fait que les champs eux-mêmes peuvent être représentés d'une façon différente par deux compilateurs différents[2].

[2] Ne serait-ce que pour la longueur du type `int`.

Ainsi, l'adresse du champ `c` d'un enregistrement `e` se calcule de la façon suivante : addr($e.c$) = addr(e) + offset(c). offset est une fonction qui dépend du compilateur.

4.3.3.2 Les tableaux

Les éléments successifs d'un tableau sont stockés dans des zones contigues de mémoire. Tous les éléments d'un tableau ont la même taille. En utilisant sizeof(t[0]) pour désigner la taille générique des éléments du tableau, l'adresse de l'élément d'indice `i` du tableau `t` se calcule de la façon suivante : addr($t[i]$) = addr(t) + i * sizeof(t[0]).

4.3.3.3 Les pointeurs

Un pointeur du langage C est représenté par une adresse mémoire. Pour la plupart des processeurs, une telle adresse occupe quatre octets[3]. Certains processeurs peuvent désigner une même adresse mémoire de plusieurs façons, en particulier, la famille des processeurs 80X86 de Intel, grâce à ses registres de segment, ne code pas directement une adresse mémoire dans un registre mais cette adresse mémoire est le résultat d'une addition de deux registres. Dans un tel cas, le compilateur doit en tenir compte, en particulier lorqu'il s'agit de comparer deux pointeurs, il ne suffit pas de comparer les registres, mais il faut comparer le résultat des deux additions.

4.3.3.4 Les fonctions

En langage C, l'identificateur d'une fonction est le pointeur sur la première instruction du code de cette fonction. La fonction elle-même étant l'ensemble des instructions du code de cette fonction. Ces instructions sont rangées consécutivement dans la mémoire.

4.4 Allocation mémoire

4.4.1 Variables globales et variables locales

Dans un langage impératif comme le langage C, la gestion mémoire "automatique" est limitée à la pile. Les variables globales n'appellent pas de gestion particulière mais simplement une unique réservation de place effectuée à la compilation. Voici un court extrait de programme en langage C qui définit trois variables de genres différents :

```
int u;

void tagada ()
{
int v;
static int w;
...
}
```

La variable `u` est une variable globale allouée une fois et une seule à la compilation. Lors d'une exécution de ce programme, cette variable occupe toujours le même emplacement mémoire.

Par contre, la variable `v` n'est allouée que lors de l'exécution de la fonction `tagada`, et un nouvel emplacement est alloué à cette variable pour chaque appel de la fonction `tagada`. Ainsi, entre deux appels successifs de la fonction `tagada`, on ne peut espérer retrouver une valeur significative dans une telle variable. De même, pour une fonction récursive, chaque instance d'exécution possède un emplacement propre pour cette variable.

Du point de vue de la gestion mémoire, la variable `w` est du même genre que la variable `u` : elle est allouée une fois et une seule à la compilation. Tout au long d'une exécution elle occupe toujours

[3] Ce serait cependant une grave erreur de programmation que de faire cette hypothèse dans le développement d'un programme.

le même emplacement mémoire. La différence avec une variable globale est que cette variable n'est accessible que dans le corps de la fonction `tagada`.

Ainsi les variables globales et les variables locales statiques sont allouées *statiquement* au *moment de la compilation* (en anglais : Compile time), et les variables locales automatiques sont allouées *dynamiquement* au *moment de l'exécution* (en anglais : Run time).

Segments La zone des variables globales est appelée le segment de données ; ce segment a une taille fixe déterminée à la compilation. La zone des variables locales automatiques est la pile. Cette pile a une taille qui évolue en permanence au cours de l'exécution du programme. Selon les systèmes d'exploitation, sa taille est soit fixe (avec donc un risque d'échec d'exécution du programme si cette taille fixe a été prévue trop petite), soit variable et augmentée au fur et à mesure des besoins.

4.4.2 L'allocation dynamique dans le tas

Certaines données nécessitent une gestion mémoire qui ne peut être ni statique — on ne connaît pas au moment de la compilation la taille nécessaire —, ni automatique (dans la pile d'exécution) — l'allocation et la libération ne sont pas liés aux appels de fonctions. On peut penser par exemple à un éditeur de texte : le texte lui-même doit être chargé en mémoire, cependant sa taille n'est pas connue lors de la création du programme, et les fonctions qui vont travailler dessus (recherche, remplacement, insertion, etc.) ne sont pas directement liées à l'allocation.

Dans ce cas, on utilise une allocation dynamique dans le tas, typiquement au moyen de deux fonctions. En langage C, la fonction `malloc()` et ses dérivées permet d'allouer une zone mémoire dont on donne la taille en argument ; et la fonction `free()` permet elle de libérer une zone mémoire fournie par `malloc()`. C'est l'ensemble des zones mémoires allouées par `malloc()` qu'on appelle le tas.

4.4.3 Le bloc d'exécution d'une fonction

Revenons sur l'allocation dynamique des variables locales automatiques. Ces variables sont allouées dans la pile. La notion de pile correspond à l'idée de plusieurs actions imbriquées les unes dans les autres. Il est classique de faire une analogie entre un cuisinier qui prépare un plat selon une recette et un processeur qui exécute un programme.

Le cuisinier suivant la recette de la page 253 en est à la ligne 28 qui lui ordonne de préparer une sauce béchamel. Cela est comparable à l'appel d'une procédure (routine, sous-routine, sous-programme, fonction, ...). En effet, pour préparer cette sauce, il doit se reporter à la page 27, sans pour autant oublier que lorsque la sauce sera prête, il devra revenir à la page 253, ligne 28. Il doit donc empiler cette *adresse de retour*. De même, la recette de la sauce béchamel peut à son tour l'entrainer à un autre endroit de son livre, il devra à ce moment de nouveau empiler l'*adresse de retour* à la page 27, ...

Il existe donc une pile gérée par le processeur pour ces adresses de retour, et un registre particulier qui indique le sommet de la pile[4]. Cette pile est utilisée par les instructions d'appels de sous-routines et de retour des dites sous-routines (pour un 80X86, les mnémoniques de ces instructions sont `CALL` et `RET` (*RETurn*)). Les langages impératifs associent une sous-routine à chaque fonction (procédure), et ainsi sont amenés à utiliser cette même pile pour leur passer leurs arguments, ranger les variables locales et retourner la valeur (dans le cas des fonctions). Cependant, pour des raisons d'efficacité, d'autres méthodes d'échange d'informations peuvent être utilisées, essentiellement l'utilisation des registres. Différentes combinaisons peuvent même être choisies entre la pile et les registres — c'est une des raisons qui peut conduire à des incompatibilités entre langages sur une même machine.

[4]Selon les processeurs, ce pointeur de pile indique soit le premier emplacement libre soit le dernier emplacement occupé. Une autre différence entre les processeurs concerne le sens d'évolution de la pile, selon les adresses croissantes ou selon les adresses décroissantes. Ces différences n'ont pas d'importance pour la compréhension de la suite, il suffit de savoir qu'il existe deux opérations de base : *empiler* et *dépiler*.

Selon les langages et les compilateurs, les choix de passages des arguments sont différents. Nous décrivons maintenant un des choix possibles. Prenons l'exemple d'une fonction, avec un argument `long` et un argument `int`, qui utilise trois variables locales, et qui rend un `long` :

```
long power (long x, int n)
{
int u,v;
long y;
...
}
```

Le code de cette fonction commence par réserver la place pour les variables locales — avec le compilateur utilisé, les variables de type `int` ou `long` occupent quatre octets, il faut donc douze octets pour les variables locales, nombre que l'on retrouve en opérande de l'instruction `subl $12, %esp`. A la fin de l'exécution de la fonction, cette place doit être libérée sur la pile, c'est l'objet de l'instruction `leave`. Le code de la fonction se termine par un `ret` :

```
1   power:
2           pushl   \%ebp
3           movl    \%esp, \%ebp
4           subl    \$12, \%esp
5           movl    \$1, -12(\%ebp)
6   .L3:
7           leal    12(\%ebp), \%eax
8           decl    (\%eax)
9           cmpl    \$-1, 12(\%ebp)
10          jne     .L5
11          jmp     .L4
12  .L5:
13          movl    -12(\%ebp), \%eax
14          imull   8(\%ebp), \%eax
15          movl    \%eax, -12(\%ebp)
16          jmp     .L3
17  .L4:
18          movl    -12(\%ebp), \%eax
19          movl    \%eax, \%eax
20          leave
21          ret
22  main:
23          pushl   \%ebp
24          movl    \%esp, \%ebp
25          subl    \$8, \%esp
26          subl    \$8, \%esp
27          pushl   \$5
28          pushl   \$3
29          call    power
30          addl    \$16, \%esp
31          movl    \%eax, u
32          leave
33          ret
```

Un appel de la fonction `power` se compose de quatre étapes :

- empilement des arguments, (lignes 27 et 28),
- appel de `power`, (ligne 29),
- "dépilement" des arguments, (ligne 30),
- récupération du résultat, (ligne 31).

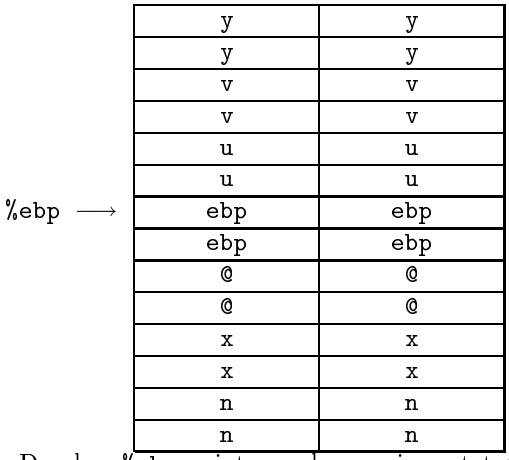
Le bloc de la pile composé :

- des arguments,
- de l'adresse de retour,
- du pointeur sur le *bloc d'exécution* précédent,
- de la place pour les variables locales,

s'appelle un *bloc d'exécution* (en anglais : frame). Les trois instructions `push %ebp ; mov %esp,%ebp ; sub 12,%esp` des lignes 2–4 permettent de fabriquer le chaînage entre le nouveau bloc d'exécution et le précédent tout en réservant la place pour les variables locales — ces trois instructions peuvent être remplacées par la seule instruction `enter`. Réciproquement l'instruction `leave` de la ligne 20 aurait pu être remplacée par `mov %ebp,%esp ; pop %ebp`. Elle remet la pile dans l'état qu'elle avait avant les trois instructions d'entrée dans la fonction.

Ainsi, un bloc d'exécution de la fonction `power` précédemment évoquée à l'allure suivante[5] :

[5] Ici, les octets ont été regroupés par deux sur une ligne



De plus, `%ebp` pointe sur le premier octet de `ebp`, ainsi la variable `u` est accessible par `[%ebp-4]`, la variable `v` par `[%ebp-8]`, et la variable `y` par `[%ebp-12]`. De même, l'argument `x` est accessible par `[%ebp+8]`, et `n` par `[%ebp+12]`. Le mode d'adressage utilisé permet ainsi d'accéder aux variables locales et aux arguments de la même façon pour toutes les instances d'appel de la fonction `power`, donc où que soit effectivement le bloc d'exécution dans la pile.

4.5 Exercices

Exercice 4.1 (La pile : arguments et variables locales) *Voici un extrait d'un programme en langage C :*

```
1   #include <stdio.h>
2
3   typedef struct { int u[100]; } UnGros;
4
5   UnGros gros;
6   int u;
7   double x;
8
9   void P (int i) { int u,v; u = i; v = u * u; }
10
11  int F (int i) { return i * i; }
12
13  void Bof (int *i) { *i = *i * *i; }
14
15  double sqr (double x) { return x * x; }
16
17  void Tehefun (UnGros g) { Tehefun(g); }
18
19  void Hadeux (UnGros *g) { Hadeux(g); }
20
21  void stack_test (int u)
22  {
23  int v;
24  printf("(%2d) %08x", u, &u);
25  printf("      %08x\n", &v);
26  if (u) stack_test(u-1);
27  }
28
29  main (int argc, char *argv[])
```

```
30  {
31    P(3);
32  L1:
33    u=F(2);
34  L2:
35    Bof(&u);
36  L3:
37    x = sqr(3.);
38  L4:
39    stack_test(3);
40  }
```

Et voici, le résultat de sa compilation en langage d'assemblage :

```
1                               .file   "asi04short.c"
2                               .version        "01.01"
3                       gcc2_compiled.:
4                       .text
5                               .align 4
6                       .globl P
7                       P:
8   0000 55                     pushl   %ebp
9   0001 89E5                   movl    %esp, %ebp
10  0003 83EC08                 subl    $8, %esp
11  0006 8B4508                 movl    8(%ebp), %eax
12  0009 8945FC                 movl    %eax, -4(%ebp)
13  000c 8B45FC                 movl    -4(%ebp), %eax
14  000f 0FAF45FC               imull   -4(%ebp), %eax
15  0013 8945F8                 movl    %eax, -8(%ebp)
16  0016 C9                     leave
17  0017 C3                     ret
18                      .Lfe1:
19                              .align 4
20                      .globl F
21                      F:
22  0018 55                     pushl   %ebp
23  0019 89E5                   movl    %esp, %ebp
24  001b 8B4508                 movl    8(%ebp), %eax
25  001e 0FAF4508               imull   8(%ebp), %eax
26  0022 89C0                   movl    %eax, %eax
27  0024 5D                     popl    %ebp
28  0025 C3                     ret
29                      .Lfe2:
30  0026 89F6                   .align 4
31                      .globl Bof
32                      Bof:
33  0028 55                     pushl   %ebp
34  0029 89E5                   movl    %esp, %ebp
35  002b 8B4D08                 movl    8(%ebp), %ecx
36  002e 8B4508                 movl    8(%ebp), %eax
37  0031 8B5508                 movl    8(%ebp), %edx
38  0034 8B00                   movl    (%eax), %eax
39  0036 0FAF02                 imull   (%edx), %eax
40  0039 8901                   movl    %eax, (%ecx)
41  003b 5D                     popl    %ebp
42  003c C3                     ret
43                      .Lfe3:
44  003d 8D7600                 .align 4
45                      .globl sqr
46                      sqr:
47  0040 55                     pushl   %ebp
48  0041 89E5                   movl    %esp, %ebp
49  0043 83EC08                 subl    $8, %esp
50  0046 8B4508                 movl    8(%ebp), %eax
51  0049 8B550C                 movl    12(%ebp), %edx
52  004c 8945F8                 movl    %eax, -8(%ebp)
```

```
53   004f 8955FC            movl    %edx, -4(%ebp)
54   0052 DD45F8            fldl    -8(%ebp)
55   0055 DC4DF8            fmull   -8(%ebp)
56   0058 C9                leave
57   0059 C3                ret
58                  .Lfe4:
59   005a 89F6              .align 4
60                  .globl Tehefun
61                  Tehefun:
62   005c 55                pushl   %ebp
63   005d 89E5              movl    %esp, %ebp
64   005f 57                pushl   %edi
65   0060 56                pushl   %esi
66   0061 81EC9001          subl    $400, %esp
67        0000
68   0067 89E7              movl    %esp, %edi
69   0069 8D4508            leal    8(%ebp), %eax
70   006c 89C6              movl    %eax, %esi
71   006e FC                cld
72   006f B9640000          movl    $100, %ecx
73        00
74   0074 F3                rep
75   0075 A5                movsl
76   0076 E8FCFFFF          call    Tehefun
77        FF
78   007b 81C49001          addl    $400, %esp
79        0000
80   0081 8D65F8            leal    -8(%ebp), %esp
81   0084 5E                popl    %esi
82   0085 5F                popl    %edi
83   0086 5D                popl    %ebp
84   0087 C3                ret
85                  .Lfe5:
86                  .align 4
87                  .globl Hadeux
88                  Hadeux:
89   0088 55                pushl   %ebp
90   0089 89E5              movl    %esp, %ebp
91   008b 83EC0C            subl    $12, %esp
92   008e FF7508            pushl   8(%ebp)
93   0091 E8FCFFFF          call    Hadeux
94        FF
95   0096 83C410            addl    $16, %esp
96   0099 C9                leave
97   009a C3                ret
98                  .Lfe6:
99                          .section        .rodata
100                 .LC1:
101  0000 28253264          .string "(%2d) %08x"
102       29202530
103       387800
104                 .LC2:
105  000b 20202020          .string "      %08x\n"
106       20202530
```

```
107         38780A00
108                     .text
109     009b 90                 .align 4
110                     .globl stack_test
111                     stack_test:
112     009c 55                 pushl   %ebp
113     009d 89E5               movl    %esp, %ebp
114     009f 83EC08             subl    $8, %esp
115     00a2 83EC04             subl    $4, %esp
116     00a5 8D5508             leal    8(%ebp), %edx
117     00a8 89D0               movl    %edx, %eax
118     00aa 50                 pushl   %eax
119     00ab FF7508             pushl   8(%ebp)
120     00ae 68000000           pushl   $.LC1
121          00
122     00b3 E8FCFFFF           call    printf
123          FF
124     00b8 83C410             addl    $16, %esp
125     00bb 83EC08             subl    $8, %esp
126     00be 8D45FC             leal    -4(%ebp), %eax
127     00c1 50                 pushl   %eax
128     00c2 680B0000           pushl   $.LC2
129          00
130     00c7 E8FCFFFF           call    printf
131          FF
132     00cc 83C410             addl    $16, %esp
133     00cf 837D0800           cmpl    $0, 8(%ebp)
134     00d3 7410               je      .L9
135     00d5 83EC0C             subl    $12, %esp
136     00d8 8B4508             movl    8(%ebp), %eax
137     00db 48                 decl    %eax
138     00dc 50                 pushl   %eax
139     00dd E8FCFFFF           call    stack_test
140          FF
141     00e2 83C410             addl    $16, %esp
142                     .L9:
143     00e5 C9                 leave
144     00e6 C3                 ret
145                     .Lfe7:
146     00e7 90                 .align 4
147                     .globl main
148                     main:
149     00e8 55                 pushl   %ebp
150     00e9 89E5               movl    %esp, %ebp
151     00eb 83EC08             subl    $8, %esp
152     00ee 83EC0C             subl    $12, %esp
153     00f1 6A03               pushl   $3
154     00f3 E8FCFFFF           call    P
155          FF
156     00f8 83C410             addl    $16, %esp
157                     .L11:
158     00fb 83EC0C             subl    $12, %esp
159     00fe 6A02               pushl   $2
160     0100 E8FCFFFF           call    F
```

```
161         FF
162    0105 83C410            addl    $16, %esp
163    0108 A3000000          movl    %eax, u
164         00
165                   .L12:
166    010d 83EC0C            subl    $12, %esp
167    0110 68000000          pushl   $u
168         00
169    0115 E8FCFFFF          call    Bof
170         FF
171    011a 83C410            addl    $16, %esp
172                   .L13:
173    011d 83EC08            subl    $8, %esp
174    0120 68000008          pushl   $1074266112
175         40
176    0125 6A00              pushl   $0
177    0127 E8FCFFFF          call    sqr
178         FF
179    012c 83C410            addl    $16, %esp
180    012f DD1D0000          fstpl   x
181         0000
182                   .L14:
183    0135 83EC0C            subl    $12, %esp
184    0138 6A03              pushl   $3
185    013a E8FCFFFF          call    stack_test
186         FF
187    013f 83C410            addl    $16, %esp
188    0142 C9                leave
189    0143 C3                ret
190                   .Lfe8:
191                           .comm   gros,400,32
192                           .comm   u,4,4
193                           .comm   x,8,8
194                           .ident  "GCC: (GNU) 2.96 20000731 (Red Hat Linux 7.0)"
```

Question 4.1.1 *Combien d'octets sont nécessaires pour stocker les variables locales de la procédure `P` ? Où voit-on apparaître cette valeur dans le listage ?*

Question 4.1.2 *Sur quelles lignes est effectuée l'appel `P(3)` du programme principal, y compris le passage de l'argument ?*

Question 4.1.3 *Dessinez le bloc d'exécution d'un appel de la procédure `P`.*

Question 4.1.4 *Sur quelles lignes est effectuée l'appel `F(2)` du programme principal, y compris le passage de l'argument ? Où est récupérée la valeur de retour de la fonction `F` ?*

Question 4.1.5 *Combien d'octets sont nécessaires pour le passage d'un argument par adresse ? Où peut-on en voir un exemple dans le listage précédent ?*

Question 4.1.6 *Où est stocké la valeur `3.0`, argument de l'appel de `sqr(3.)` ? Etudier l'appel `sqr(3.)`.*

Question 4.1.7 *Dessinez le bloc d'exécution d'un appel de la fonction `sqr`.*

Question 4.1.8 *Des deux procédures `Tehefun` et `Hadeux`, laquelle sature le plus rapidement (compté en nombre d'appel, et non en temps d'exécution) la pile ? On précisera combien d'octets sont nécessaires sur la pile pour chaque appel de ces deux procédures.*

Interpreter, n. : One who enables two persons of different languages to understand each other by repeating to each what it would have been to the interpreter's advantage for the other to have said. – Ambrose Bierce, "The Devil's Dictionary"

THE LESSER-KNOWN PROGRAMMING LANGUAGES #10 : SIMPLE

SIMPLE is an acronym for Sheer Idiot's Monopurpose Programming Language Environment. This language, developed at the Hanover College for Technological Misfits, was designed to make it impossible to write code with errors in it. The statements are, therefore, confined to BEGIN, END and STOP. No matter how you arrange the statements, you can't make a syntax error. Programs written in SIMPLE do nothing useful. Thus they achieve the results of programs written in other languages without the tedious, frustrating process of testing and debugging.

When someone says "I want a programming language in which I need only say what I wish done," give him a lollipop.