Génération de Code (2)

CS410 - Langages et Compilation

Julien Henry
Catherine Oriat

Grenoble-INP Esisar

2013-2014

Génération de code : déjà vu

- Organisation de la mémoire
- Génération de code pour les expressions arithmétiques
- Allocation des registres

Ce qu'il reste à voir :

- Structures de contrôle : `if-then-else`, boucles, etc.
- Evaluation des expressions Booléennes
- Appels de fonction

Summary

Codage des booléens

Pour coder des expressions booléennes, on peut utiliser des valeurs entières :

- On code *false* par 0, et *true* par 1.
- On code *false* par 0, et *true* par tous les entiers non nuls (ex : C).

Evaluation des opérations *or* et *and*

Exemple :
On veut évaluer l'expression C suivante :

```
(p != NULL) and (*p == 42)
```

2 possibilités :

- Evaluation **stricte** : on évalue A = `(p != NULL)` et B = `(*p == 42)`, puis on calcule A `and` B
- Evaluation **paresseuse** : on évalue A = `(p != NULL)`. Si A = *false*, alors l'expression vaut *false* sans avoir besoin d'évaluer B. Si A = *true*, on évalue B = `(*p == 42)` et on renvoie B.

C'est le même principe avec l'opérateur `or`

Conséquences

Dans le cas d'une évaluation paresseuse :

- Gain de performance car on évite d'évaluer la seconde opérande
- Il faut savoir si le compilateur évalue l'opérande de gauche ou l'opérande de droite en premier !
- La sémantique du programme est différente selon le type d'évaluation choisi. Le langage source spécifie le mode d'évaluation dans son document de référence.

Exemple : En Java,

- `A | B` calcule `A or B` de manière stricte, en commençant par évaluer `A` puis `B`.
- `A || B` calcule `A or B` de manière paresseuse, en commençant par évaluer `A`, puis si `A` est faux évalue `B`.

Génération de code pour les expressions booléennes

Lorsque l'on fait une évaluation paresseuse, certaines portions de code ne sont pas toujours exécutées.

On va donc utiliser des instructions de branchement !

Instructions de comparaison et de branchement

On suppose que le langage d'assemblage dispose de l'instruction de comparaison suivante

`CMP dval, Ri` : compare la valeur dval à la valeur stockée dans le registre `Ri`

Cette instruction positionne les codes conditions suivant le résultat de la comparaison (flags du processeur) :

- `EQ` $\Leftrightarrow$ `dval` $=$ `Ri`
- `NE` $\Leftrightarrow$ `dval` $\neq$ `Ri`
- `GT` $\Leftrightarrow$ `dval` $>$ `Ri`
- `GE` $\Leftrightarrow$ `dval` $\geq$ `Ri`
- `LT` $\Leftrightarrow$ `dval` $<$ `Ri`
- `LE` $\Leftrightarrow$ `dval` $\leq$ `Ri`

Instructions de comparaison et de branchement

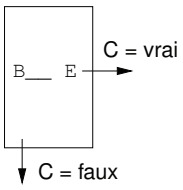
Chaque code condition a une instruction de branchement associée :

- `BEQ E` : saut vers l'étiquette `E` si `EQ` est positionné à *true*.
- `BNE E` : idem pour `NE`
- `BGT E` : idem pour `GT`
- `BGE E` : idem pour `GE`
- `BLT E` : idem pour `LT`
- `BLE E` : idem pour `LE`
- `BRA` : branchement inconditionnel

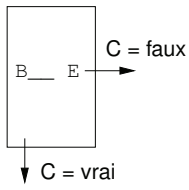
Codage d'une expression booléenne

Le code d'une expression booléenne est une suite d'instruction contenant un certain nombre de sauts conditionnels.

- On peut terminer la suite d'instruction si on peut évaluer paresseusement que l'expression est vraie (schéma de gauche) ou fausse (schéma de droite)



Saut = vrai (ex : or)



Saut = faux (ex : and)

Fonction de génération de code

On définit une fonction de génération de code pour une expression booléenne :

```
Etiquette get_etiquette(); // crée une étiquette
void generer_etiquette(Etiquette E); // crée une étiquette

void coder_cond(Arbre C, Boolean Saut, Etiquette E) {...}
```

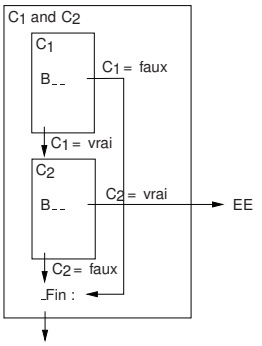
- Si `Saut = true`, il y a branchement à l'étiquette `E` lorsque `C` est vrai.
- Si `Saut = false`, il y a branchement à l'étiquette `E` lorsque `C` est faux.

Implémentation de `coder_cond`

- `coder_cond(true, true, E)` = `generer(BRA E)`
  On se branche à `E` si `true` s'évalue à *vrai*, ce qui est toujours le cas.
- `coder_cond(true, false, E)` = `null`
  On se branche à `E` si `true` s'évalue à *faux*, ce qui est toujours faux.
- `coder_cond(false, true, E)` = `null`
- `coder_cond(false, false, E)` = `generer(BRA E)`

Implémentation de `coder_cond`

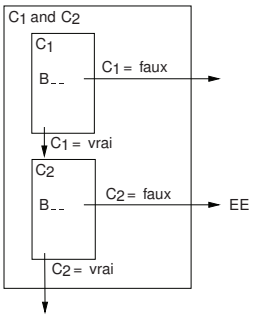
- `coder_cond(C1 and C2, true, E) =`




```
E_Fin = get_etiquette()
coder_cond(C1, false,
E_Fin)
coder_cond(C2, true, E)
generer_etiquette(E_Fin)
```


Implémentation de `coder_cond`

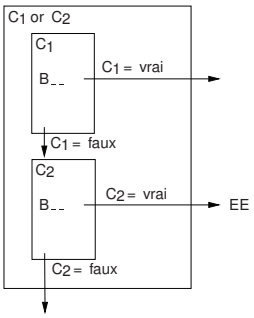
- `coder_cond(C1 and C2, false, E) =`




```
coder_cond(C1, false, E)
coder_cond(C2, false, E)
```


Implémentation de `coder_cond`

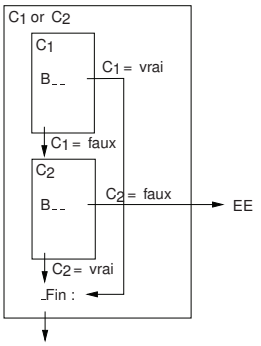
- `coder_cond(C1 or C2, true, E) =`




```
coder_cond(C1, true, E)
coder_cond(C2, true, E)
```


Implémentation de `coder_cond`

- `coder_cond(C1 or C2, false, E) =`




```
E_Fin = get_etiquette()
coder_cond(C1, true,
E_Fin)
coder_cond(C2, false, E)
generer_etiquette(E_Fin)
```


Implémentation de `coder_cond`

- `coder_cond(idf, true, E) =`
  ```
  generer(LOAD @idf, R0)
  generer(CMP #0, R0)
  generer(BNE E)
  ```
- `coder_cond(idf, false, E) =`
  ```
  generer(LOAD @idf, R0)
  generer(CMP #0, R0)
  generer(BEQ E)
  ```


Implémentation de `coder_cond`

- `coder_cond(E1 < E2, true, E)`
  ```
  coder_exp(E2,R0)
  coder_exp(E1,R1)
  generer(CMP R1, R0)
  generer(BLT E)
  ```
- `coder_cond(E1 < E2, false, E)`
  ```
  coder_exp(E2,R0)
  coder_exp(E1,R1)
  generer(CMP R1, R0)
  generer(BGE E)
  ```

On fait de même pour tous les opérateurs de comparaison.

Summary

Noeuds if-then-else

On veut générer le code correspondant à une conditionnelle *if-then-else*. On doit donc faire un test de la condition, et brancher vers le bloc *then* ou *else*.

```
coder_inst(if C then T else F) =
        E_end = get_etiquette()
        E_else = get_etiquette()
        coder_cond(C,false,E_else)
        coder_inst(T)
        generer(BRA E_end)
        generer_etiquette(E_else)
        coder_inst(F)
        generer_etiquette(E_fin)
```


Noeuds if-then

On veut générer le code correspondant à une conditionnelle *if-then*.

```
coder_inst(if C then T) =
        E_end = get_etiquette()
        coder_cond(C,false,E_end)
        coder_inst(T)
        generer_etiquette(E_fin)
```


Noeuds while

On veut générer le code correspondant à boucle *while* (C) {I}.

- Méthode 1 :

```
E_debut:
   <code de C avec branchement vers E_fin
   si C est faux>
   <code de I>
   BRA E_debut
E_fin:
```

- Méthode 2 :

```
   BRA E_cond
E_debut:
   <code de I>
E_cond:
   <code de C avec branchement vers E_debut
   si C est vrai>
```


Noeuds while

```
coder_inst(while C do T) =
        E_cond = get_etiquette()
        E_debut = get_etiquette()
        generer(BRA E_cond)
        generer_etiquette(E_debut)
        coder_inst(T)
        generer_etiquette(E_cond)
        coder_cond(C,true,E_debut)
```


Exercice

Comment ferait-on pour générer le code correspondant à une boucle for ?

```
for (I; C; N) { E }
```

Summary

Rappel

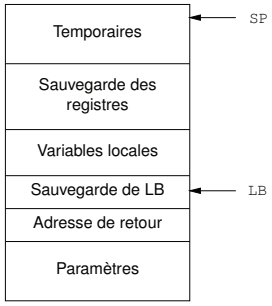
La mémoire associée au programme contient une pile, qui permet de gérer les données nécessaires à l'exécution d'une fonction.

Ce qu'on stocke sur la pile

- Paramètres de la fonction
- Adresse de retour : un fois que la fonction a terminé, il faut retourner à l'endroit du code d'où la fonction a été appelée
- Variables locales
- Sauvegarde des registres que la fonction utilise
- les variables temporaires que l'on crée lorque le nombre de registres est insuffisant
- l'ancienne valeur de LB pour pouvoir dépiler le nombre correct de cases mémoire quand la fonction termine

Allocations sur la pile

Toutes les choses précédentes ont une position bien définie dans la pile. On accède à chacunes des cases mémoire grâce aux registres `LB` et `SP`. L'ensemble de ces données s'appelle le **bloc d'activation** de la fonction.



Exemple : Factorielle

```
void Fact(int n, int* r) {
  int s;
  if (n <= 1) {
    *r = 1;
  } else {
    Fact(n-1,&s);
        *r = n*s;
  }
}
```

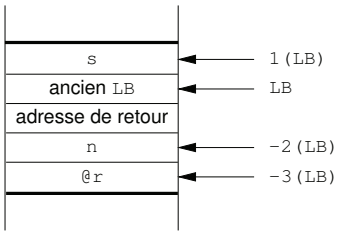
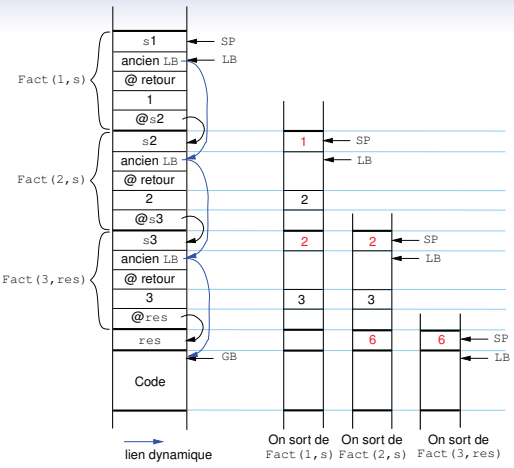


FIGURE: Bloc d'activation d'un appel de la fonction Fact



Code généré lors d'un appel de fonction

Lors d'un appel de fonction, il faut donc ajouter sur la pile le bloc d'activation de cette fonction.

- On commence par donner les paramètres avec des instruction `PUSH`
- On empile ensuite l'adresse de retour, qui se calcule à partir du registre `PC` (program counter)
- On empile ensuite la valeur du registre `LB` : `PUSH LB`
- puis on se branche vers la fonction appelée, dont le début est défini par une étiquette.

Code généré en début de fonction

- On commence par réserver dans la pile la place pour les variables locales. On incrémente donc le registre `SP`.
- On empile les registres que l'on va utiliser
- On initialise les variables locales si besoin.

Code généré en fin de fonction

- On restaure les registres pour qu'il reprennent la valeur qu'ils avaient dans la fonction appelante.
- On dépile les variables locales : on décrémente donc `SP`.
- On se branche à l'adresse de retour

Au retour de la fonction (dans la fonction appelante), on doit encore effectuer quelques instructions :

- On dépile l'adresse de retour et les paramètres de la fonction.

Exemple avec le langage d'assemblage ARM

```
void f(int x) {
        x=x+1;
}
void g() {
        int x = 42;
        f(x);
}
```

Remarque :

- quand on empile, les adresses diminuent
- `bl` branche et met à jour le registre `lr` (link register)

```
f:
        push    {r0}
        add     r0, r0, #1
        str     r0, [sp], #4
        mov     pc, lr
    ...
g:
        push    {lr}
        sub     sp, sp, #4
        mov     r0, #42
        str     r0, [sp]
        mov     r0, #42
        bl      f
        add     sp, sp, #4
        pop     {lr}
        mov     pc, lr
    ...
```


Passage des arguments par valeur/référence

On peut passer les paramètres d'une fonction par référence ou par valeur :

- Par **référence** : dans la pile, c'est l'adresse de l'objet qui est ajoutée. Les modifications de l'objet affecteront donc le reste du programme quand la fonction terminera.
- Par **valeur** : dans la pile, c'est la valeur de l'objet qui est ajoutée. Si on modifie cette valeur, la case mémoire va être modifiée dans la pile. Á la fin de la fonction, on dépile son bloc d'activation et les changements de valeur du paramètre est donc sans effet.

Summary

Données composées

On peut vouloir représenter des données *composées* :

- tableaux
- structures

Ces éléments peuvent être manipulés comme expressions, passés en paramètres de fonction, etc.

Ils occupent une place de plus d'un mot dans la mémoire : il faut préciser le mode de représentation de ces objets.

Tableaux à deux dimensions

On veut représenter dans la mémoire un tableau à deux dimensions, par exemple la matrice :

$$A = \begin{pmatrix} a_{1,1} & a_{1,2} & a_{1,3} \\ a_{2,1} & a_{2,2} & a_{2,3} \end{pmatrix}$$

Les langages de programmation utilisent des représentations différentes pour ce type de structures.

Tableaux à deux dimensions

Codage contigu par colonnes :

A(2,3)
A(1,3)
A(2,2)
A(1,2)
A(2,1)
A(1,1)

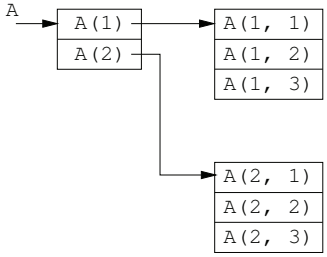
Codage contigu par lignes :

A(2,3)
A(2,2)
A(2,1)
A(1,3)
A(1,2)
A(1,1)

L'adresse de `A(i,j)` dépend du nombre de lignes et de colonnes.

Tableaux à deux dimensions

En Java, les tableaux à deux dimensions sont stockés de la façon suivante :



C'est en fait un vecteur de pointeurs sur un tableau à $n-1$ dimensions. Les données sont alors non-contigües.

Passage par référence / valeur

Si on veut passer en paramètre à une fonction un élément de type complexe (structure, tableau, etc) :

- Passage par référence : on empile uniquement l'adresse du premier élément du tableau/structure
- Passage par valeur : on empile tous les éléments du tableau/structure.

Summary

Résumé

1. On a un programme écrit dans notre langage de programmation préféré.
2. Le processus de compilation permet de générer un programme en langage d'assemblage.

Mais :

Un fichier en langage d'assemblage n'est toujours pas exécutable !

Exemple

bonjour.c :

```
int main (void) {
        int x;
        x = f(42);
        x++;
        return x;
}
```

f.c :

```
int f(int x) {
        return x + 5;
}
```


Exemple : gcc -S bonjour.c

```
        .file   "bonjour.c"
        .text
        .globl  main
        .type   main, @function
main:
.LFB0:
        .cfi_startproc
        pushl   %ebp
        .cfi_def_cfa_offset 8
        .cfi_offset 5, -8
        movl    %esp, %ebp
        .cfi_def_cfa_register 5
        andl    $-16, %esp
        subl    $32, %esp
        movl    $42, (%esp)
        call    f
        movl    %eax, 28(%esp)
        addl    $1, 28(%esp)
        movl    28(%esp), %eax
        leave
        .cfi_restore 5
        .cfi_def_cfa 4, 4
        ret
        .cfi_endproc
.LFE0:
        .size   main, .-main
        .ident  "GCC: (Ubuntu/Linaro 4.7.2-2ubuntu1) 4.7.2"
        .section        .note.GNU-stack,"",@progbits
```


gcc -c bonjour.s

`gcc -c bonjour.s` crée le fichier objet bonjour.o
On peut voir le code binaire avec `od -x bonjour.o`

```
0000000 457f 464c 0101 0001 0000 0000 0000 0000
0000020 0001 0003 0001 0000 0000 0000 0000 0000
0000040 0114 0000 0000 0000 0034 0000 0000 0028
0000060 000c 0009 8955 83e5 f0e4 ec83 c720 2404
0000100 002a 0000 fce8 ffff 89ff 2444 831c 2444
0000120 011c 448b 1c24 c3c9 4700 4343 203a 5528
0000140 7562 746e 2f75 694c 616e 6f72 3420 372e
0000160 322e 322d 6275 6e75 7574 2931 3420 372e
0000200 322e 0000 0014 0000 0000 0000 7a01 0052
0000220 7c01 0108 0c1b 0404 0188 0000 001c 0000
...
```


objdump

```
#objdump -d bonjour.o

bonjour.o:     file format elf32-i386
Disassembly of section .text:
00000000 <main>:
   0: 55                    push   %ebp
   1: 89 e5                 mov    %esp,%ebp
   3: 83 e4 f0              and    $0xfffffff0,%esp
   6: 83 ec 20              sub    $0x20,%esp
   9: c7 04 24 2a 00 00 00  movl   $0x2a,(%esp)
  10: e8 fc ff ff ff        call   11 <main+0x11>
  15: 89 44 24 1c           mov    %eax,0x1c(%esp)
  19: 83 44 24 1c 01        addl   $0x1,0x1c(%esp)
  1e: 8b 44 24 1c           mov    0x1c(%esp),%eax
  22: c9                    leave
  23: c3                    ret
```


Exemple

bonjour.c :

```
int main (void) {
        int x;
        x = f(42);
        x++;
        return x;
}
```

f.c :

```
int f(int x) {
        return x + 5;
}
```

Ici, `bonjour.o` ne connait pas le symbole `f` déclaré dans `f.o`.

Symboles globaux

- Un programme est séparé en plusieurs fichiers source
- On crée un fichier assembleur par fichier source
- Chaque fichier assembleur a des symboles globaux : ils correspondent aux identificateurs déclarés dans le fichier et qui devront être visibles par les autres fichiers

Dans un fichier assembleur, on peut faire appel à des fonctions qui ne sont pas déclarées dans ce même fichier (venant d'un autre fichier du projet, d'une librairie, etc)

Conclusion :

- Un fichier assembleur définit des symboles globaux
- Un fichier assembleur utilise des symboles globaux

Symboles globaux : nm et objdump

```
#nm bonjour.o
         U f
00000000 T main


#objdump -t bonjour.o
bonjour.o:     file format elf32-i386

SYMBOL TABLE:
00000000 l    df *ABS* 00000000 bonjour.c
00000000 l    d  .text 00000000 .text
00000000 l    d  .data 00000000 .data
00000000 l    d  .bss 00000000 .bss
00000000 l    d  .note.GNU-stack 00000000 .note.GNU-stack
00000000 l    d  .eh_frame 00000000 .eh_frame
00000000 l    d  .comment 00000000 .comment
00000000 g     F .text 00000024 main
00000000         *UND* 00000000 f
```


Edition de lien

La compilation en fichiers objets laisse l'identification de certains symboles à plus tard. L'édition de lien permet de résoudre la liaison de ces symboles externes :

- liaison **statique** : le fichier objet et la bibliothèque sont liés dans le même fichier exécutable par le **linker** (ou **éditeur de lien**).
- liaison **dynamique** : le fichier objet est lié avec la bibliothèque, mais pas dans le même fichier exécutable. Les liens sont établis lors du lancement de l'exécutable par le **chargeur**.

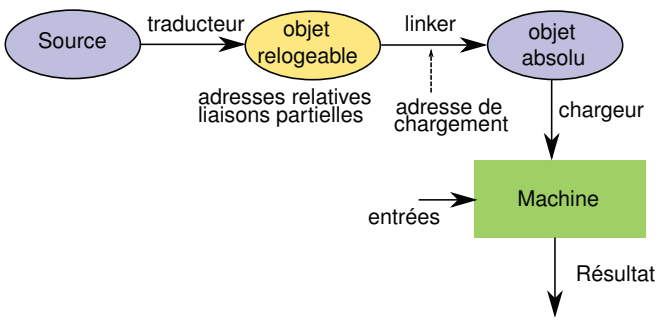
Fichier objet relogeable

Un fichier objet (`.o`) est **relogeable** : Toutes les adresses mémoire qui sont manipulées sont relatives.

- Dans le `.o`, les adresses mémoires sont toutes relatives à l'adresse de début de la section (`.text`, `.data`)
- les adresses que vont occuper par `.text` et `.data` ne sont pas encore connues

L'**édition de lien** (**link**) "fusionne" les différents `.o`, assigne une adresse aux débuts des sections, et remplace toutes les adresses relatives par des adresses absolues.

Etape de la vie d'un programme unique



Etape de la vie d'un programme composé

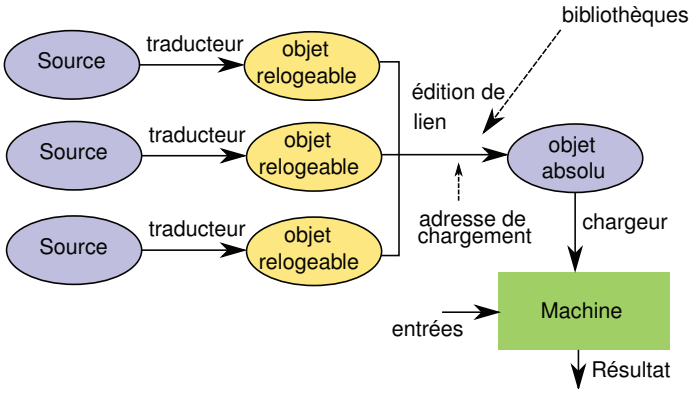


Table de relocation, table des symboles

Pour pouvoir faire les calculs des emplacements mémoire des différents symboles, le chargeur a besoin d'informations.

Un fichier objet dispose d'une table de relocation et d'une table des symboles.

- **table des symboles** : liste les adresses (relatives) des symboles dans leur section.
- **table de relocation** : liste des "trous" à remplir (chacunes des utilisations de symboles)

Exemple

```
#  adr.
#relative   code hexa       code asm
                            .section .data
    0000    03000000    i:    .int 3
    0004    FF          j:    .byte 0xff

                            .section .text
                            .global main
    0000    B800000000  main: movl $i,%eax
    0005    3A0504000000      cmpb j,%al
    000b    3D04000000        cmpl $main,%eax
    0010    C3                ret
```

Table des Symboles :

data	0x00	i
data	0x04	j
text	0x00	main

Table de relocation :

0x01	text	.data(->i)
0x07	text	.data(->j)
0x0c	text	main

Erreur de link

En compilant un programme en C, on peut avoir une erreur de **link** :
Exemple :

- Déclaration dans un fichier `fichier1.h` d'une fonction `f`.
- Utilisation de la fonction `f` dans un fichier `fichier2.c`
- On oublie d'implémenter `f` dans `fichier1.c`