Compilation: Procédures, fonctions

Florent Madelaine
florent.madelaine@unicaen.fr

Langages et compilation — L3 informatique

1. Boucles

While

Syntaxe : `'while' '(' c=condition ')' i=instruction`

Notes :

- Dans tous les langages de programmation ;
- Utile pour le cas de boucles dont on ne connaît pas le nombre d'itération ;
- Attention à bien faire changer la condition pour éviter une boucle infinie.

Until

Syntaxe :

```
'do' i=instruction 'until' '(' c=condition ')' finInstruction
```

Notes :

- Assez peu utilisée ;
- Très pratique pour un bloc devant au moins être effectué une fois ;
- Permet parfois une meilleur compréhension.

For

Syntaxe : `'for''(' init=assignation? ';' c=condition? ';' incr=assignation?')' i=instruction`

Notes :

- Une boucle présente partout ;
- Certains langages utilisent un `'for' IDENTIFIANT 'in' expression`
- Attention : pas de règles sur les éléments ;
- Attention : mieux vaux ne pas modifier la variables sur laquelle porte la boucle.

Abus de for

Altération : De part son caractère très générique, il existe de nombreux abus de la boucle for.

- Transformer un 'while' en 'for' ;
- Mettre un corps vide ;
- Mélanger condition et action ;
- ...

Utilisation. En dehors des cas sur les chaînes, il est préférable d'éviter ces utilisations ou d'indiquer leur principe sur un commentaire.

2. Conditions et évaluation

Optimisation

Exemple :

```
false and (y == 4 + x or x = y* y*y*y*y)
```

On sait dès le départ qu'il est inutile d'évaluer la partie droite.

Ordre d'évaluation

Et en C ?

```
1  int x = 4;
   if ( 1 && x++) {}
3  printf("%d\n",x);
```

Attention. Certains comportements sont détaillés dans la norme mais d'autres sont laissés au choix du compilateur.

3. Vers les fonctions

Saut dans le code et partage

Idée :

- On peut isoler une portion de code entre un label et un jump ;
- on pourrait vouloir utiliser ce code depuis deux endroit différents de notre programme.

Saut dans le code et partage

Idée :

- On peut isoler une portion de code entre un label et un jump ;
- on pourrait vouloir utiliser ce code depuis deux endroit différents de notre programme.

Problème : on ne peut pas revenir à deux endroit différents.

Retour au départ

Solution : on va sauvegarder dans un endroit connu l'adresse de l'endroit d'où l'on part.

Retour au départ

Solution : on va sauvegarder dans un endroit connu l'adresse de l'endroit d'où l'on part.

Endroit connu pour nous : sur le haut de la pile.

Support des variables locales

Problème : on peut arriver dans le code depuis des endroits ayant des taille et contenu de pile différentes.

Problème : comment faire si on veut parler de variables locales (paramètres de la fonction).

fp

Solution : avoir une variable qui donne la position de la pile au moment de l'arrivée : fp

fp

Solution : avoir une variable qui donne la position de la pile au moment de l'arrivée : fp

Travail supplémentaire : sauvegarder l'ancienne valeur avant d'appeler la fonction, calculer la nouvelle valeur, restaurer la valeur à la sortie.

fp

Solution : avoir une variable qui donne la position de la pile au moment de l'arrivée : fp

Travail supplémentaire : sauvegarder l'ancienne valeur avant d'appeler la fonction, calculer la nouvelle valeur, restaurer la valeur à la sortie.

Où ça : toujours sur la pile.

Arguments

Question : comment passer des arguments ?

Arguments

Question : comment passer des arguments ?

Solution : les positionner en haut de la pile juste avant de passer au code de la fonction.

Valeur de retour

Question : et pour la valeur de retour ?

Valeur de retour

Question : et pour la valeur de retour ?

Réponse : La positionner en haut de la pile (réserver de la place) juste avant de passer au code de la fonction.

En résumé : que doit-on sauvegarder/restaurer ?

- Lors du retour normal de la procédure la suite de l'exécution doit se poursuivre à l'instruction suivant l'instruction d'appel.
  => Le compteur de programme `pc` doit donc être sauvegardé à chaque appel
- Les données locales à la procédure s'organisent dans la pile à partir d'une adresse appelée *frame pointer* qui est conservée dans un registre spécial `fp`
  Lorsqu'une nouvelle procédure est appelée, cette valeur change, il est donc nécessaire de sauvegarder la valeur courante qui devra être restaurée à la fin de la procédure.
- Les opérations à effectuer lors d'un appel de procédure se partagent entre l'appelant et l'appelé.

L'appelant et l'appelé doivent avoir une vision cohérente de l'organisation de la mémoire

4. Fonctions

Assembleur

Principe : des instructions assembleur dédié à l'appel de fonction.

Opcode :

Code	Pile	sp	pc
CALL label		...	instr(label)
RETURN		...	...

call

Action :

- L'instruction **CALL** prend comme argument une adresse dans le code d'instructions (en argument ou sur la pile).
- Le compteur d'instructions pc se place alors à cette adresse, son ancienne valeur est sauvegardée.
- **CALL** positionne le registre fp (adresse du bloc d'activation) à la valeur courante de sp.

return

Action :

- L'instruction **RETURN** retrouve l'ancienne valeur du compteur de programme et se place à l'instruction suivante.
- Elle repositionne sp à la valeur courante de fp et restaure fp à son ancienne valeur

Exemple

Exemples d'utilisation de CALL et RETURN

Exemple de code pour le passage des arguments

Appel de `f(1,2)`, l'instruction suivante de l'appel étant à l'adresse 15 (valeur courante de `pc`). On suppose que `fp` vaut 0.

Qui	Quoi	Pile
Appelant	Empile les valeurs des arguments appelle `CALL` en donnant l'adresse de la fonction	[] [1, 2]
Machine	Le `CALL` empile l'adresse de retour et la valeur du frame pointer	[1, 2, 15, 0]
Appellé	Exécute son code qui se termine par `RETURN`	[1, 2, 15, 0]
Machine	Le `RETURN` dépile tout ce que la procédure a empilé et n'a pas dépilé jusqu'à dépiler le frame pointeur et le compteur ordinal qu'elle restaure	[1, 2]
Appelant	Dépile les arguments qu'il avait empilé	[]

Avec valeur de retour

C'est similaire mais il faut garder de la place sur la pile avant l'appel de la fonction.

Qui	Quoi	Pile
Appelant	Laisse de la place pour la valeur de retour et empile les valeurs des arguments appelle `CALL` en donnant l'adresse de la fonction	[] [0, 1, 2]
Machine	Le `CALL` empile l'adresse de retour et la valeur du frame pointer	[0, 1, 2, 15, 0]
Appellé	Exécute son code qui se termine par `RETURN` met à jour la valeur de retour	[42, 1, 2, 15, 0]
Machine	Le `RETURN` dépile tout ce que la procédure a empilé et n'a pas dépilé jusqu'à dépiler le frame pointeur et le compteur ordinal qu'elle restaure	[42, 1, 2]
Appelant	Dépile les arguments qu'il avait empilé et utilise la valeur de retour (ou pas)	[42]

Si appel depuis une autre fonction que le main ?

Appel de `f(1,2)`, l'instruction suivante de l'appel étant à l'adresse 15 (valeur courante de `pc`). On suppose que `fp` vaut 51 (valeur courante de `fp` pour la fonction appelante).

Qui	Quoi	Pile
Appelant	Laisse de la place pour la valeur de retour et empile les valeurs des arguments appelle `CALL` en donnant l'adresse de la fonction	[] [0, 1, 2]
Machine	Le `CALL` empile l'adresse de retour et la valeur du frame pointer	[0, 1, 2, 15, 51]
Appellé	Exécute son code qui se termine par `RETURN` met à jour la valeur de retour	[42, 1, 2, 15, 51]
Machine	Le `RETURN` dépile tout ce que la procédure a empilé et n'a pas dépilé jusqu'à dépiler le frame pointeur et le compteur ordinal qu'elle restaure	[42, 1, 2]
Appelant	Dépile les arguments qu'il avait empilé et utilise la valeur de retour (ou pas)	[42]

Manipulation des variables locales

Le registre fp est à jour au début de l'appel de procédure et permet de référencer les valeurs locales.

Aux instructions STOREG, PUSHG correspondent les instructions STOREL, PUSHL qui ont le même comportement mais en remplaçant gp par fp.

Manipulation des variables locales

Le registre fp est à jour au début de l'appel de procédure et permet de référencer les valeurs locales.

Aux instructions STOREG, PUSHG correspondent les instructions STOREL, PUSHL qui ont le même comportement mais en remplaçant gp par fp.

Rappel.

Code	Pile	sp	pc	
PUSHG n	P[sp] := P[gp+n]	sp+1	pc+1	n entier t.q. gp+n < sp
STOREG n	P[gp+n] := P[sp-1]	sp-1	pc+1	n entier t.q. gp+n < sp

Assigner une variable locale

Code	Pile	sp	pc	
PUSHG n	P[sp] := P[gp+n]	sp+1	pc+1	n entier t.q. gp+n < sp
PUSHL n	P[sp] := P[fp+n]	sp+1	pc+1	n entier t.q. fp+n < sp

En résumé

- Les opérations à effectuer lors d'un appel de procédure se partagent entre l'appelant et l'appelé.
- L'appelant effectue la réservation pour la valeur de retour dans le cas de fonctions et évalue les paramètres effectifs de la procédure.
- L'appelé initialise ses données locales et commence l'exécution du corps de la procédure. Au moment du retour, l'appelé place éventuellement le résultat de l'évaluation à l'endroit réservé par l'appelant et restaure les registres.

sur la pile le bloc d'activation de la fonction ressemble à

... <out> <args> <back adr> <old fp>

Le registre fp contient l'adresse de la pile de la dernière case de ce bloc d'activation, ce qui permet d'accéder aux arguments et à la valeur de retour (adresse négative par rapport à fp).

Exemple

```
function f(x, y)
   { return 2 * x + y; }
f(20, 2);
```

L'appellant empilera donc trois valeurs,

- une pour réserver la place pour le résultat
- une deuxième pour la valeur du premier paramètre
- une troisième pour la valeur du second paramètre

Le code de la fonction est en début du programme.
Le programme commence par un saut au début de la première instruction principale (*main*).

Exemple

Commentaire

```
On saute dans le ≪main≫
Début fonction f
2
x
2 * x
y
2 * x + y
stocké dans la pile comme valeur de retour
Fin fonction f


Début programme principal
On réserve la place pour la valeur de retour
On empile l'argument entier 20
On empile l'argument entier 2
On appelle f
On dépile le 2e argument
On dépile le 1er argument
On écrit le résultat
On dépile le résulat
Arrêt de la machine
```

Code MVàP

```
  JUMP 1
LABEL 0
  PUSHI 2
  PUSHL -4
  MUL
  PUSHL -3
  ADD
  STOREL -5
  RETURN
RETURN

LABEL 1
  PUSHI 0
  PUSHI 20
  PUSHI 2
  CALL 0
  POP
  POP
  WRITE
  POP
  HALT
```

Exemple

Commentaire

```
On saute dans le ≪main≫
Début fonction f
2
x
2 * x
y
2 * x + y
stocké dans la pile comme valeur de retour
Fin fonction f


Début programme principal
On réserve la place pour la valeur de retour
On empile l'argument entier 20
On empile l'argument entier 2
On appelle f
On dépile le 2e argument
On dépile le 1er argument
On écrit le résultat
On dépile le résulat
Arrêt de la machine
```

Pourquoi 2 RETURN ?

Code MVàP

```
  JUMP 1
LABEL 0
  PUSHI 2
  PUSHL -4
  MUL
  PUSHL -3
  ADD
  STOREL -5
  RETURN
RETURN

LABEL 1
  PUSHI 0
  PUSHI 20
  PUSHI 2
  CALL 0
  POP
  POP
  WRITE
  POP
  HALT
```

Exemple

Commentaire

```
On saute dans le ≪main≫
Début fonction f
2
x
2 * x
y
2 * x + y
stocké dans la pile comme valeur de retour
Fin fonction f


Début programme principal
On réserve la place pour la valeur de retour
On empile l'argument entier 20
On empile l'argument entier 2
On appelle f
On dépile le 2e argument
On dépile le 1er argument
On écrit le résultat
On dépile le résulat
Arrêt de la machine
```

Code MVàP

```
  JUMP 1
LABEL 0
  PUSHI 2
  PUSHL -4
  MUL
  PUSHL -3
  ADD
  STOREL -5
  RETURN
RETURN

LABEL 1
  PUSHI 0
  PUSHI 20
  PUSHI 2
  CALL 0
  POP
  POP
  WRITE
  POP
  HALT
```

Pourquoi 2 RETURN ? C'est juste parce qu'on a du code MVàP compilé.

Exemple

Code assemblé

```
 Adr |  Instruction
-----+--------------
   0 | JUMP       14

   2 | PUSHI       2
   4 | PUSHL      -4
   6 | MUL
   7 | PUSHL      -3
   9 | ADD
  10 | STOREL     -5
  12 | RETURN
  13 | RETURN

  14 | PUSHI       0
  16 | PUSHI      20
  18 | PUSHI       2
  20 | CALL        2
  22 | POP
  23 | POP
  24 | WRITE
  25 | POP
  26 | HALT
```

Trace

```
  pc |                |    fp   pile
=================================================
   0 | JUMP       14 |     0 [ ] 0
  14 | PUSHI       0 |     0 [ ] 0
  16 | PUSHI      20 |     0 [ 0 ] 1
  18 | PUSHI       2 |     0 [ 0 20 ] 2
  20 | CALL        2 |     0 [ 0 20 2 ] 3
   2 | PUSHI       2 |     5 [ 0 20 2 22 0 ] 5
   4 | PUSHL      -4 |     5 [ 0 20 2 22 0 2 ] 6
   6 | MUL           |     5 [ 0 20 2 22 0 2 20 ] 7
   7 | PUSHL      -3 |     5 [ 0 20 2 22 0 40 ] 6
   9 | ADD           |     5 [ 0 20 2 22 0 40 2 ] 7
  10 | STOREL     -5 |     5 [ 0 20 2 22 0 42 ] 6
  12 | RETURN        |     5 [ 42 20 2 22 0 ] 5
  22 | POP           |     0 [ 42 20 2 ] 3
  23 | POP           |     0 [ 42 20 ] 2
  24 | WRITE         |     0 [ 42 ] 1
42
  25 | POP           |     0 [ 42 ] 1
  26 | HALT          |     0 [ ] 0
```

Récursion

Principe : faire appel à une fonction à l'intérieur d'elle même.

Récursion

Principe : faire appel à une fonction à l'intérieur d'elle même.

Mise en place : cela marche tout seul.

Exemple de fonction récursive

```
function fact(n) {
   if(n <= 1) return 1;
   return n*fact(n-1);
}
write(fact(3));
```

Exemple de fonction récursive

Commentaire

```
On saute à la ≪première≫ instruction
Début fonction fact

  si (n <= 1)

    retour 1
    stocké dans la pile

  fin alors
  sinon
    rien
  finsi
  n
  valeur retour
  n
  1
  n - 1
  appel fact(n-1)
  on dépile (n-1)
  n * fact(n-1)
  stocké dans la pile

Fin fonction fact
Début programme principal
  valeur retour
  paramètre 3
  appel fact(3)
  on dépile 3

  on dépile fact(3)
```

Code MVàP

```
  JUMP 3
LABEL 0
  PUSHL -3
  PUSHI 1
  INFEQ
  JUMPF 2
  PUSHI 1
  STOREL -4
  RETURN
  JUMP 1
LABEL 2

LABEL 1
  PUSHL -3
  PUSHI 0
  PUSHL -3
  PUSHI 1
  SUB
  CALL 0
  POP
  MUL
  STOREL -4
  RETURN
RETURN
LABEL 3
  PUSHI 0
  PUSHI 3
  CALL 0
  POP
  WRITE
  POP
  HALT
```

Code assemblé

```
 Adr |  Instruction
-----+---------------
   0 | JUMP       33
   2 | PUSHL      -3
   4 | PUSHI       1
   6 | INFEQ
   7 | JUMPF      16
   9 | PUSHI       1
  11 | STOREL     -4
  13 | RETURN
  14 | JUMP       16

  16 | PUSHL      -3
  18 | PUSHI       0
  20 | PUSHL      -3
  22 | PUSHI       1
  24 | SUB
  25 | CALL        2
  27 | POP
  28 | MUL
  29 | STOREL     -4
  31 | RETURN
  32 | RETURN

  33 | PUSHI       0
  35 | PUSHI       3
  37 | CALL        2
  39 | POP
  40 | WRITE
  41 | POP
  42 | HALT
```

Trace

```
  pc |               |    fp   pile
=====================================================
   0 | JUMP       33 |     0 [ ] 0
  33 | PUSHI       0 |     0 [ ] 0
  35 | PUSHI       3 |     0 [ 0 ] 1
  37 | CALL        2 |     0 [ 0 3 ] 2
   2 | PUSHL      -3 |     4 [ 0 3 39 0 ] 4
   4 | PUSHI       1 |     4 [ 0 3 39 0 3 ] 5
   6 | INFEQ         |     4 [ 0 3 39 0 3 1 ] 6
   7 | JUMPF      16 |     4 [ 0 3 39 0 0 ] 5
  16 | PUSHL      -3 |     4 [ 0 3 39 0 ] 4
  18 | PUSHI       0 |     4 [ 0 3 39 0 3 ] 5
  20 | PUSHL      -3 |     4 [ 0 3 39 0 3 0 ] 6
  22 | PUSHI       1 |     4 [ 0 3 39 0 3 0 3 ] 7
  24 | SUB           |     4 [ 0 3 39 0 3 0 3 1 ] 8
  25 | CALL        2 |     4 [ 0 3 39 0 3 0 2 ] 7
   2 | PUSHL      -3 |     9 [ 0 3 39 0 3 0 2 27 4 ] 9
   4 | PUSHI       1 |     9 [ 0 3 39 0 3 0 2 27 4 2 ] 10
   6 | INFEQ         |     9 [ 0 3 39 0 3 0 2 27 4 2 1 ] 11
   7 | JUMPF      16 |     9 [ 0 3 39 0 3 0 2 27 4 0 ] 10
  16 | PUSHL      -3 |     9 [ 0 3 39 0 3 0 2 27 4 ] 9
  18 | PUSHI       0 |     9 [ 0 3 39 0 3 0 2 27 4 2 ] 10
  20 | PUSHL      -3 |     9 [ 0 3 39 0 3 0 2 27 4 2 0 ] 11
  22 | PUSHI       1 |     9 [ 0 3 39 0 3 0 2 27 4 2 0 2 ] 12
  24 | SUB           |     9 [ 0 3 39 0 3 0 2 27 4 2 0 2 1 ] 13
  25 | CALL        2 |     9 [ 0 3 39 0 3 0 2 27 4 2 0 1 ] 12
   2 | PUSHL      -3 |    14 [ 0 3 39 0 3 0 2 27 4 2 0 1 27 9 ] 14
   4 | PUSHI       1 |    14 [ 0 3 39 0 3 0 2 27 4 2 0 1 27 9 1 ] 15
   6 | INFEQ         |    14 [ 0 3 39 0 3 0 2 27 4 2 0 1 27 9 1 1 ] 16
   7 | JUMPF      16 |    14 [ 0 3 39 0 3 0 2 27 4 2 0 1 27 9 1 ] 15
   9 | PUSHI       1 |    14 [ 0 3 39 0 3 0 2 27 4 2 0 1 27 9 ] 14
  11 | STOREL     -4 |    14 [ 0 3 39 0 3 0 2 27 4 2 0 1 27 9 1 ] 15
  13 | RETURN        |    14 [ 0 3 39 0 3 0 2 27 4 2 1 1 27 9 ] 14
  27 | POP           |     9 [ 0 3 39 0 3 0 2 27 4 2 1 1 ] 12
  28 | MUL           |     9 [ 0 3 39 0 3 0 2 27 4 2 1 ] 11
  29 | STOREL     -4 |     9 [ 0 3 39 0 3 0 2 27 4 2 ] 10
  31 | RETURN        |     9 [ 0 3 39 0 3 2 2 27 4 ] 9
  27 | POP           |     4 [ 0 3 39 0 3 2 2 ] 7
  28 | MUL           |     4 [ 0 3 39 0 3 2 ] 6
  29 | STOREL     -4 |     4 [ 0 3 39 0 6 ] 5
  31 | RETURN        |     4 [ 6 3 39 0 ] 4
  39 | POP           |     0 [ 6 3 ] 2
  40 | WRITE         |     0 [ 6 ] 1
6
  41 | POP           |     0 [ 6 ] 1
  42 | HALT          |     0 [ ] 0
```

Récursion terminale

Optimisation. Dans le cas où la fonction récursive s'appelle elle-même en dernière instruction. On peut réutiliser l'appel en cours pour économiser la pile.

... <out> <args> <back adr> <older fp> <out> <args> <back adr> <old fp>

devient

... <tmp> <args> <back adr> <older fp>

On triche en réutilisant le bloc d'activation du premier appel, le CALL récursif, devient un simple JUMP. Il faut le cas échéant ajouter un argument à la fonction pour calculer sans attendre des résultats qu'on aurait laissé en attente. Dans l'exemple qui suit, on triche encore plus en utilisant l'emplacement de la valeur de retour comme accumulateur (multiplicatif dans le cas de factoriel).

Exemple de fonction récursive

Récursion non terminale

```
  JUMP 3
LABEL 0
  PUSHL -3
  PUSHI 1
  INFEQ
  JUMPF 2
  PUSHI 1
  STOREL -4
  RETURN
  JUMP 1
LABEL 2

LABEL 1
  PUSHL -3
  PUSHI 0
  PUSHL -3
  PUSHI 1
  SUB
  CALL 0
  POP
  MUL
  STOREL -4
  RETURN
RETURN
LABEL 3
  PUSHI 0
  READ
  CALL 0
  POP
  WRITE
  POP
  HALT
```

Récursion terminale

```
  JUMP 3
LABEL 0
  PUSHL -3
  PUSHI 1
  INFEQ
  JUMPF 2
  RETURN
  JUMP 1
LABEL 2

LABEL 1
  PUSHL -3
  PUSHL -4
  MUL
  STOREL -4
  PUSHL -3
  PUSHI 1
  SUB
  STOREL -3
  JUMP 0
  RETURN
RETURN
LABEL 3
  PUSHI 1
  READ
  CALL 0
  POP
  WRITE
  POP
  HALT
```

Imbrication

Question. Que se passe-t-il si on peut définir une fonction à l'intérieur d'une fonction.

Problème. Accès aux variables locale de la fonction englobante.
Solution possible à l'aide des *fp* sauvegardés dans la pile.

5. Problèmes et fonctions

Passage des arguments

Il existe deux façon de passer des arguments :

- le **passage par valeur**, qui recopie l'argument ; et,
- le **passage par référence**, qui utilise juste un pointeur.

Passage par valeur

Principe :

- Dans le passage de paramètre par valeur, x est une nouvelle variable allouée localement par la procédure dont la valeur est le résultat de l'évaluation de e.
- Après la fin de la procédure, la place mémoire allouée à la variable x est libérée. Les modifications apportées à x ne sont donc plus visibles.
- En l'absence de pointeurs, les seules variables modifiées sont les variables non locales à la procédure explicitement nommées dans les instructions du programme.
- Il est nécessaire de réserver une place proportionnelle à la taille du paramètre ce qui peut être coûteux dans le cas de tableaux.

Passage par référence ou par adresse

Principe :

- On calcule l'adresse de e (la valeur gauche de l'expression)
- La fonction alloue une variable x qui est initialisée par la valeur gauche de e.
- Toute référence à x dans le corps de la fonction est interprétée comme une opération sur l'objet situé à l'adresse stockée en x.
- Ce mode de passage occupe une place indépendante de la taille du paramètre (une adresse).

Note :

- En C, le passage par référence est explicitement programmé par le passage d'un pointeur (adresse mémoire) par valeur.
- En Java, le passage se fait par valeur mais les objets ont pour valeur une référence.

Fonctions et bibliothèques

Problème : l'appelant positionne les variables ; et, L'appelé les utilise. Il faut donc que les deux soient d'accord sur l'organisation de la mémoire.

Incompatibilités : il faut donc que les conventions soient les même pour pouvoir utiliser une fonction venant de l'extérieur.

Fonctions et compilations séparée

Remarque. Lors de la compilation d'une bibliothèque, le compilateur utilisé n'est pas le même que celui qui sera utilisé pour la compilation du programme.

Problème. Cette différence peut engendrer des différence de compilation lors de l'appel de fonction (optimisations, ...) qui rendent les codes incompatibles.