Compilation: Variable, conditions, branchements

Gaétan Richard
gaetan.richard@unicaen.fr

Langages et compilation — L3 informatique

1. Variables (globales)

Portée

Portée d'une variable.

- En général, une variable globale est déclarée en début du fichier source et est valable jusqu'à la fin de l'exécution;
- Dans certains langages, il est possible de définir des variables à l'intérieur de blocs. Leur portée est limitée à ce bloc.

Cas du C

```
1 #include <stdio.h>

3 int main (void) {
    int j;
5   j = 0;
    for (int i = 0; i <10; i++) {
7     j = i * j + 2;
    }
9   printf("%d,%d",j,j);
}
```

Mise en place

On peut :

- Empiler la valeur d'une variable globale qui est contenue dans la pile
- Recopier la valeur en sommet de pile dans la variable globale

Mise en place

On peut :

- Empiler la valeur d'une variable globale qui est contenue dans la pile
- Recopier la valeur en sommet de pile dans la variable globale

Accès direct à la mémoire.

Code	Pile	sp	pc	Condition
PUSHG n	P[sp] := P[n]	sp+1	pc+2	n entier t.q. n $<$ sp
STOREG n	P[n] := P[sp-1]	sp-1	pc+2	n entier t.q. n $<$ sp

Variables et compilation

Besoin:

Déclaration: réservation de l'espace mémoire et mémorisation du liens entre nom et emplacement mémoire.

Assignation / Utilisation: retrouver l'emplacement mémoire et l'utiliser pour stocker ou charger la valeur.

Structure:

Pour stocker ces informations, on utilise une table des symboles qui contient l'ensemble des symboles présents dans le source et permet en cas de besoin d'associer entre autre une adresse.

En Antlr

Membres: Il est possible de déclarer des structures utilisées sur l'ensemble de l'analyse:

```
@members {
HashMap<String, Integer> memory = new HashMap<String, Integer>();
}
```

et d'inclure des headers java:

```
@header {
import java.util.HashMap;
}
```

En Antlr

Membres: Il est possible de déclarer des structures utilisées sur l'ensemble de l'analyse:

```
@members {
HashMap<String, Integer> memory = new HashMap<String, Integer>();
}
```

et d'inclure des headers java:

```
@header {
import java.util.HashMap;
}
```

On utilisera cette facilité pour manipuler notre table de symboles.

2. Branchements

Branchement inconditionnel

Changer: pour changer l'exécution, il suffit de changer la valeur du pc.

Code MVàP:

Code	Pile	sp	pc	Condition
JUMP label		sp	instr(label)	

Pour déterminer où aller, on indique des étiquettes dans le code MVàP.

```
1 <...>
  JUMP blop
3 PUSHI 1
  LABEL blop
5 <...>
```

Coût: Dans un processeur réel, une telle opération est coûteuse car elle nécessite de "casser" le pipeline.

Branchement conditionnel

Principe: L'instruction de branchement est exécutée suivant une condition.

Code MVàP:

Code	Pile	sp	pc	Condition
JUMPF label		sp-1	pc+2 si P[sp-1] $\neq$ 0 instr(label) sinon	

Conditions dans MVàP

Instructions MVàP:

Code	Pile	sp	pc
INF	P[sp-2]:= 1 si P[sp-2] $<$ P[sp-1], 0 sinon	sp-1	pc+1
INFEQ	P[sp-2]:= 1 si P[sp-2] $\leq$ P[sp-1], 0 sinon	sp-1	pc+1
SUP	P[sp-2]:= 1 si P[sp-2] $>$ P[sp-1], 0 sinon	sp-1	pc+1
SUPEQ	P[sp-2]:= 1 si P[sp-2] $\geq$ P[sp-1], 0 sinon	sp-1	pc+1
EQUAL	P[sp-2]:= 1 si P[sp-2] = P[sp-1], 0 sinon	sp-1	pc+1
NEQ	P[sp-2]:= 0 si P[sp-2] = P[sp-1], 1 sinon	sp-1	pc+1

Retour vers la machine à pile ...

Exemple de code:

```
1  <...>
   PUSHI 12
3  PUSHI 24
   INFEQ
5  JUMPF suite
   PUSHI 13
7  JUMP fin
   LABEL suite
9  PUSHI 2
   LABEL fin
11 PUSHI 3
   ADD
13 WRITE
   <...>
```

... et le C

```
<...>
LBB0_1:                                 ## =>This Inner Loop Header:
        Depth=1
        cmpl    $20, -8(%rbp)
        jge     LBB0_4
## BB#2:                                ##   in Loop: Header=BB0_1
    Depth=1
        movl    -12(%rbp), %eax
        imull   $3, -8(%rbp), %ecx
        addl    %ecx, %eax
        addl    $5, %eax
        movl    %eax, -12(%rbp)
## BB#3:                                ##   in Loop: Header=BB0_1
    Depth=1
        movl    -8(%rbp), %eax
        addl    $1, %eax
        movl    %eax, -8(%rbp)
        jmp     LBB0_1
LBB0_4:
<...>
```

Boucles

Structures classiques:

```
<...>
   | 'if' '(' c=condition ')' blocthen=instruction ('else' blocelse=
       instruction)?
   | 'while' '(' c=condition ')' i=instruction
   | 'for' '(' init=assignation? ';' c=condition? ';' incr=assignation?
       ')' i=instruction
  | 'do' i=instruction 'until' '(' c=condition? ')' finInstruction
```

Compilation:

- Évaluer les conditions;
- Mettre les étiquettes;
- Ajouter les sauts conditionnels;
- Compiler le code.

Assemblage

Label vers adresse:

- L'assemblage transforme les labels en adresses;
- On peut ajouter deux codes sources mvap à la suite mais pas deux codes assemblés;
- On peut *relativiser* chaque adresse par rapport à un registre global (code **PIC**).

Exemple avec la machine à pile

```
int i
i = 6
while (i < 10) i = i + 1
i
```

```
  PUSHI 0
  JUMP 0
LABEL 0
  PUSHI 6
  STOREG 0
LABEL 1
  PUSHG 0
  PUSHI 10
  INF
  JUMPF 2
  PUSHG 0
  PUSHI 1
  ADD
  STOREG 0
  JUMP 1
LABEL 2
  PUSHG 0
  WRITE
  POP
  HALT
```

```
 Adr |  Instruction
-----+--------------
   0 | PUSHI 0
   2 | JUMP  4
   4 | PUSHI 6
   6 | STOREG 0
   8 | PUSHG 0
  10 | PUSHI 10
  12 | INF
  13 | JUMPF 24
  15 | PUSHG 0
  17 | PUSHI 1
  19 | ADD
  20 | STOREG 0
  22 | JUMP  8
  24 | PUSHG 0
  26 | WRITE
  27 | POP
  28 | HALT
```

3. 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'itérations;
- 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: 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.

4. 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.

5. 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