

M1IF12, TD 5 : Génération de code intermédiaire + optims

Hinde Bouziane et Laure Gonnord

27 octobre 2008

Remplissage des tables

Un petit exercice pour vérifier que l'on sait bien remplir les tables :

```
program trier;

type tarray : array [ 1.. 10 ] of integer;
var t0, t1, t2 : tarray;

procedure remplirTab ( var dest : tarray );
var i : integer;
begin
(* bla bla bla *)
end;

procedure afficherTab ( src : tarray );
var i : integer;
begin
(* bla bla bla *)
end;

procedure copyTab ( src : tarray ; var dest : tarray );
var i : integer;
begin
for i := 1 to 10 do
dest [ i ] := src [ i ];
end;

(* tri par insertion du minimum *)
procedure triMinimum ( src : tarray ; var dest : tarray );
(* recupere la valeur min d'un tableau entre 2 indices *)
var i : integer;
function getMinimum ( src : tarray ; i0, i1 : integer ) : integer;
var i, indice : integer;
begin
indice := i0;
for i := i0+1 to i1 do begin
if src [ indice ] > src [ i ] then
indice := i
end
getMinimum := indice
end;
(*
echange deux valeurs dans un tableau
```

```

*)
procedure swap ( i0, i1 : integer ; var src : tarray )
var v : integer;
begin
v := src [ i0 ];
src [ i0 ] := src [ i1 ];
src [ i1 ] := v
end;
begin
copyTab ( src, dest );
for i := 1 to 10 do
swap ( i, getMinimum ( dest, i, 10 ), dest )
end;

(* tri a bulles *)
procedure triABulles ( src : tarray ; var dest : tarray );
var i,j,t:Integer;
begin
copyTab ( src, dest );
for i := 1 To 9 do
for j := i+1 to 10 do
If dest[i]>dest[j] then begin
t:=dest[i];
dest[i]:=dest[j];
dest[j]:=t;
end;
end;

begin
remplirTab ( t0 );
triMinimum ( t0, t1 );
triABulles ( t0, t2 );
afficherTab ( t1 );
afficherTab ( t2 );
end.

```

CORRECTION : Pour les ids :

ident	nom
0	trier
1	tarray
2	t0
3	t1
4	t2
5	remplirTab
6	dest
7	i
8	afficherTab
9	src
10	copyTab
11	triMinimum
12	getMinimum
13	i0
14	i1
15	indice
16	swap
17	v
18	triABulles
19	j
20	t

et pour les symboles :

TS0 (du programme)	parent=NULL
ident	categorie parametres
0	programme (code=???,TS=TS0)
1	type (type=array,indices=(1..10),elements=integer)
2	variable (type=1)
3	variable (type=1)
4	variable (type=1)
5	procedure (arite=1,code=???,TS=TS1)
6	procedure (arite=1,code=???,TS=TS2)
7	procedure (arite=2,code=???,TS=TS3)
11	procedure (arite=2,code=???,TS=TS4)
18	procedure (arite=2,code=???,TS=TS7)
TS1 (remplirTab)	parent=TS0
ident	categorie parametres
6	argument (modifiable=vrai,type=1)
7	variable (type=integer)
TS2 (afficherTab)	parent=TS0
ident	categorie parametres
9	argument (modifiable=faux,type=1)
7	variable (type=integer)
TS3 (copyTab)	parent=TS0
ident	categorie parametres
9	argument (modifiable=faux,type=1)
6	argument (modifiable=vrai,type=1)
7	variable (type=integer)
TS4 (triMinimum)	parent=TS0
ident	categorie parametres
9	argument (modifiable=faux,type=1)
6	argument (modifiable=vrai,type=1)
7	variable (type=integer)
12	fonction (arite=3,retour=integer,code=???,TS=TS5)
16	procedure (arite=3,code=???,TS=TS6)

TS5 (getMinimum)		parent=TS4
ident	categorie	parametres
9	argument	(modifiable=faux,type=1)
13	argument	(modifiable=faux,type=integer)
14	argument	(modifiable=faux,type=integer)
7	variable	(type=integer)
15	variable	(type=integer)
TS6 (swap)		parent=TS4
ident	categorie	parametres
13	argument	(modifiable=faux,type=integer)
14	argument	(modifiable=faux,type=integer)
6	argument	(modifiable=vrai,type=1)
17	variable	(type=integer)
TS7 (triABulles)		parent=TS0
ident	categorie	parametres
9	argument	(modifiable=faux,type=1)
6	argument	(modifiable=vrai,type=1)
7	variable	(type=integer)
19	variable	(type=integer)
20	variable	(type=integer)

Génération de code intermédiaire

Le code 3 adresses est utilisé ici comme représentation intermédiaire. La traduction est dirigée par la syntaxe («à la yacc»). Dans un premier temps, on considère que l'on a un nombre de registre infini.

On considère la syntaxe suivante pour les expressions

$E \rightarrow id=E \mid id[E]=E \mid E+E \mid id \mid n \mid true \dots \mid E=E \mid E \text{ ou } E \mid \dots$

On utilise les attributs «code» pour le code (!) et «place» pour le nom du registre dans lequel on stocke le calcul, et on se réfère à la feuille distribuée pour la production du code.

Pour les programmes on utilise le «mini langage while» suivant :

Prog - > ListeInstructions

ListeInstructions -> Instruction ; ListeInstruction | eps

..

Instruction -> id:=E | If E then ListeInstruction else ..
| while E do ... done;

EXERCICE 1 Génération de code 3 adresses : expressions, instructions

En utilisant les fonctions de génération de code pour machine à registre, écrivez la séquence de code produite pour les fragments de programme suivants :

```
x1 := 3; x2 := 3 + x1 + x2; | x2 := 0;
if not (x2 = x1) then      | while x3
    x1 := x2                |     x3 := not x3 and (x2 = (x2 + 2));
else                        |
    x2 := 0                 |
```

```
var x,y:int
while not x=y do
    if x>y then x:=x-y else y:=y-x
done
```

CORRECTION :

Code A :

On doit générer le code d'une séquence d'instructions, donc on va faire des appels concaténés

Code-prog-A =

```
x1 := 3          /* E → entier { E.code = ''; E.val = entier.val}
                  I → id := E { I.code = E.code Id.place ':=' E.val}
                  */

t1 := 3 + x1     /* E → entier { E.code = ''; E.val = entier.val}
                  E → Id {E.place = Id.place; E.code = ''}
                  E0 → E1 + E2 { E0.place = nouvTemp()
                                E0.code = E0.place ':=' E1.val '+' E2.place
                                }
                  */

t2 := t1 + x2    /* E → Id {E.place = Id.place; E.code = '' pour x2}
                  E0 → E1 + E2 { E0.place = nouvTemp()
                                E0.code = E0.place ':=' E1.place '+' E2.place
                                }
                  */

t3 := x2 == x1   /* E0 → E1 == E2 {E0.place = nouvTemp();
                  E0.code = '' E0.place ':=' E1.place '==' E2.place}

t4 := ! t3       /* E0 → non E1 {E0.place = nouvTemp();
                  E0.code = '' E0.place ':=' '!' E1.place}

si t4 aller à lvrai1 /* pour étiq on aurait du mettre l1, l2... mais pour lisibilité...*/
x2 := 0          /* I → id := E { I.code = '' Id.place ':=' E.val}
aller à lsuivant1
lvrai1:
x1 := x2         /* I → id := E { I.code = '' Id.place ':=' E.place}
lsuivant1:      /* I0 → if E then I1 esle I2
                  {lsuivant = nouvelleEtiquette()
                   lvrai = nouvelleEtiquette()
                   I0.code =
                     E.code
                   si E.place aller à lvrai
```

```

        L2.code
        aller à lsuivant
    lvrai:
        L1.code
    lsuivant:
    }
*/

#### Code B :

Code-prog-B =

    x2 := 0 /* voir ci dessus*/

ldebutBoucle1:
    /* code expression booleene = ''*/
    si x3 aller à lbody1
    aller à lfinBoucle1
lbouble1:
    t1 := ! x3
    t2 := x2 + 2
    t3 := x2 == t2 /* x3 := .... même principe qu'avant*/
    t4 := t1 & t3
    x3 := t4
    aller à ldebutBoucle1
lfinBoucle1: /*
    I0 → while E do I1 done
    { ldebutBoucle = nouvEtiquette()
      lboucle = nouvEtiquette()
      lfinBoucle = nouvEtiquette()
      I0.code =
      I0.ldebutBoucle:
      E.code
      si E.place aller à lboucle
      aller à lfinBoucle
      lboucle:
      I1.code
      aller à ldebutBoucle
      lfinBoucle:
    }
*/

### Code C :
ldebutBoucle:
    t1 := x == y
    t2 := ! t1
    si t2 aller à lboucle
    aller à lfinBoucle

lboucle:
    t3 := x > y
    si t3 aller à lvrai
    t5 := x - y
    y := t5 /* attention: indices des temporaires */

```

```

    aller à lsuivant
lvrai:
    t4 := x - y
    x := t4
lsuivant:
    aller à ldebutBoucle
lfinBoucle

```

EXERCICE 2 Génération de code 3 adresses : xor et repeat until

On étend le langage `while` en lui ajoutant :

- l'opérateur booléen "ou exclusif" (`e1 xor e2` est vrai ssi un seul des deux opérandes est vrai);
- l'instruction `repeat C until E`.

Complétez les fonctions de génération de code pour prendre en compte ces extensions.

CORRECTION :

XOR:

plusieurs façons de faire:

1) traduire le xor en fonction de and et or

```

E → E1 xor E2 {
    v1 = nouvTemp()
    v2 = nouvTemp()
    E.code =
        E1.code
        E2.code
    v1 := E1.place '|' E2.place
    v2 := E1.place '&' E2.place
    v3 := '!' v2
    E.place := v1 '&' v3
}

```

2) selon E1 vrai ou faux

```

E → E1 xor E2 {
    lvrai1 = nouvEtiquette()
    lsuivant = nouvEtiquette()
    E.code =
        E1.code
        E2.code
    si E1.place aller à lvrai1
    E.place ':=' E2.place
    aller à lsuivant
lvrai1:
    E.place ':=' ! E2.place
lsuivant:
}

```

REPEAT:

```

I → repeat c until e { ldebutBoucle = nouvEtiquette()
    lfinBoucle = nouvEtiquette()
    I.code =
    ldebutBoucle:

```

```

    c.code
    E.code
    si E.place aller à lfinBoucle
    aller à ldebutBoucle
lfinBoucle:
}

```

EXERCICE 3 Génération de code 3@ - for, pointeurs, tableaux, ...

Pour chacun des (bouts de) programmes suivants, repérer les difficultés, construire les règles, générer le code

1. Boucle for :

```

var i,tmp : int;
tmp:=0;
for (i=0,i<10, i++) do
    tmp := tmp+i;
done;

```

2. Tableaux :

```

a[i]:=a[i-1];

```

3. Pointeurs :

```

var u:integer;
var x,p : ^integer;
p^ := 4;
x := p;
u := p^;

```

CORRECTION :

For:

```

I → for (I1, e, I2) do I3 done {ldebuBoucle = nouvEtiquette()
                                lboucle = nouvEtiquette()
                                lfinBoucle = nouvEtiquette()
                                I.code = I1.code
                                ldebuBoucle:
                                    e.code
                                    si e.place aller à lboucle
                                    aller à lfinBoucle
                                lboucle:
                                    I3.code
                                    I2.code
                                    aller à ldebuBoucle
                                lfinBoucle:
}

```

Sur l'exemple:

```

tmp := 0
i := 0
ldebuBoucle1:
    t1 := i < 10
    si t1 aller à lboucle1
    aller à lfinBoucle1

```

lboucle1:

```

t2 := tmp + i
tmp := t2
t3 := i + 1
i := t3
aller à ldebutBoucle1
lfinBoucle1:

```

Tableaux:

```

E → E1[E2, E3] { t1 = nouvTemp()
                  t2 = nouvTemp
                  t3 = nouvTemp
                  E.code =
                    E1.code
                    E2.code
                    E3.code
                  /* on suppose que les indices de 0 à N-1 et placement par ligne en memoire*/
                  t1 := E1.tailleLigne '*' E2.place
                  t2 := t1 + E2.place
                  t3 := E1.place[t2]
                }

```

```

I → E1[E2,E3] := E4 { I.code =
                    E1.code
                    E2.code
                    E3.code
                    E4.code
                    t1 := E1.tailleLigne '*' E2.place
                    t2 := t1 + E2.place
                    E1.place [t2] ':=' E4.place
                }

```

Sur exemple:

Pointeurs:

```

E → E1^ { E.code =
          E2.code
          E.place ':=' * E2.place
        }

```

```

I → E1^ { E.code =

```

Sur l'exemple:

```

* p := 4
x := p
u := *p

```

EXERCICE 4 Génération de code pour les fonctions/procédures

Générer le code 3 adresses pour les codes suivants :

1. Fonction retournant un entier :

```
int mafonction (a:int, b:int)
{
    return (a+b);
}
```

2. Procédure modifiant un paramètre booléen

```
void maproc (var b:bool)
{
    b := true
}
```

CORRECTION :

Fonction :

```
DeclF → enteteF corpsF { TS [ f ].code =
    enter n // nb arguments
    res m // taille en memoire des variables locales
    corpsF.code
}
```

Sur l'exemple:

```
t1 := a +b
return t1
```

Procédure

```
DeclP → enteteP corpsP {TS [ p ].code =
    enter n // nb arguments
    res m // taille en memoire des variables locales
    corpsP.code
    return
}
```

Sur l'exemple

```
b := 1 /* si je suppose que true est un entier != 0 dans le code 3 adresses*/
return
```

EXERCICE 5 Optimisation du nombre de registres

Soit le code suivant :

```
int x,y,z,a,b,c ;
x := a+b+c;
y := a*b + 2 *c ;
```

1. Générer du code 3 adresses et de l'assembleur en supposant un nombre non borné de registres.
2. Générer du code 3 adresses et de l'assembleur en supposant que l'on dispose uniquement de 4 registres : R1, R2, R3 et R4

CORRECTION :

Nombre de registre illimité:

```
t1 := a + b
t2 := t1 + c
x := t2
t3 := a * b
t4 := 2 * c
t5 := t3 + t4
y := t5
```

Nombre de registre limité:

```
t1 := a + b
x := t1 + c
t2 := a * b
t3 := 2 * c
y := t2 + t3
```

Optimisation indépendante de la machine

Il reste ensuite à faire la production du code assembleur :

- Sélection des instructions
- Allocation des registres
- Optimisations ...

Problèmes spécifiques à la machine cible Pour en savoir plus : **Compilateurs (dragon book) Aho, Sethi, Ullman**

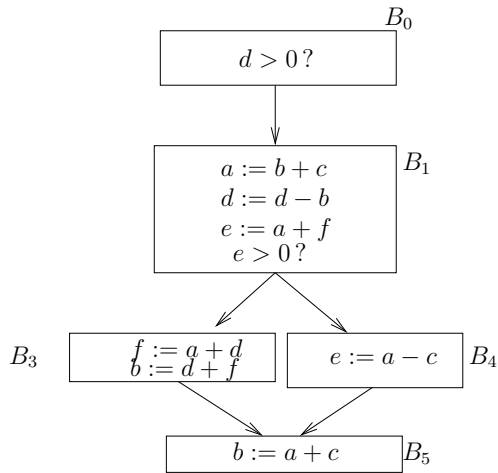
Dans la suite nous regardons les optimisations indépendantes de machine cible. Elles sont en général réalisées sur le graphe de flot de contrôle (cf document joint).

EXERCICE 6 Graphe de flot

Générer le graphe de flot de contrôle pour le code suivant :

```
while d>0 do{
  a:=b+c;
  d:=d-b;
  e:=a+f;
  if e>0
    {f:=a-d;b:=d+f}
  else
    {e:=a-c}
  b:=a+c}
```

CORRECTION : On obtient pour graphe de flot de contrôle :



On met le test dans le bloc B_1 , oui oui !

EXERCICE 7 Variables actives

Sur le graphe de flot de l'exercice précédent :

1. Écrire et résoudre le système d'équations relatif aux ensembles Gen , $Kill$, In et Out , en prenant bien soin d'initialiser les ensembles correctement.
2. Supprimer les instructions inutiles ("code mort").

CORRECTION :

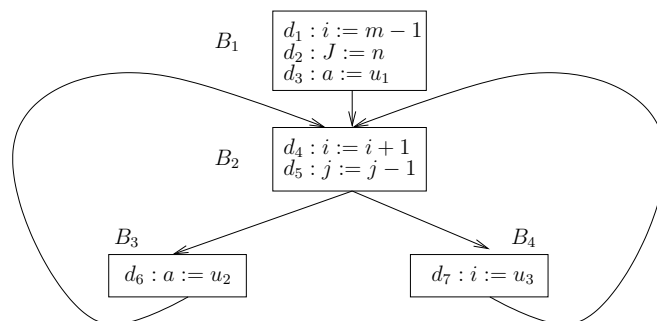
B_i	Gen	$Kill$	In	Out	In	Out	In	Out	In	Out	In	Out	In	Out
B_0	d	\emptyset	d	$bcdf$	$bcdf$	$bcdf$	$bcdf$	$bcdf$	$bcdf$	$bcdf$	$bcdf$	$bcdf$	$bcdf$	$bcdf$
B_1	$bcdf$	ade	$bcdf$	e	$bcdf$	$acde$	$bcdf$	$acde$	$bcdf$	$acde$	$bcdf$	$acde$	$bcdf$	$acde$
B_2	e	\emptyset	e	acd	$acde$	acd	$acde$	acd	$acde$	$acdf$	$acdef$	$acdf$	$acdef$	$acdef$
B_3	ac	e	ad	ac	acd	acd	acd	$acdf$	acd	$acdf$	acd	$acdf$	acd	$acdf$
B_4	ad	bf	ac	ac	ac	acd	acd	$acdf$	acd	$acdf$	acd	$acdf$	acd	$acdf$
B_5	ac	b	ac	d	acd	$bcdf$	acd	$bcdf$	acd	$bcdf$	acd	$bcdf$	acd	$bcdf$

On a donc : $Out(B_3) = \{a, c, d, f\}$. b est affectée en B_3 mais n'est pas vivante en sortie de B_3 , on peut donc enlever cette affectation.

De même, dans le bloc 4, e n'est pas vivante en sortie, donc on peut supprimer l'affectation $e := \dots$

EXERCICE 8 Variables Actives

Après génération de code, on obtient le programme suivant :



Supprimer le code mort.

CORRECTION :

Initialisation :

B_i	Gen	$Kill$
B_1	m, n, u_1	i, J, a
B_2	i, j	i, j
B_3	u_2	a
B_4	u_3	i

Itérations

B_i	In	Out	In	Out	In	Out	In	Out
B_1	m, n, u_1	i, j	m, n, j, u_1	u_2, u_3, i, j	m, n, u_1, u_2, u_3, j	u_2, u_3, i, j	u_1, u_2, u_3, m, n, j	<i>idem</i>
B_2	i, j	u_2, u_3	u_2, u_3, i, j	i, j, u_2, u_3	u_2, u_3, i, j	u_2, u_3, i, j	u_2, u_3, i, j	<i>idem</i>
B_3	u_2	i, j	i, j, u_2	u_2, u_3, i, j	u_2, u_3, i, j	u_2, u_3, i, j	u_2, u_3, i, j	<i>idem</i>
B_4	u_3	i, j	u_3, j	u_2, u_3, i, j	u_2, u_3, j	u_2, u_3, i, j	u_2, u_3, j	<i>idem</i>

Conclusion : on peut supprimer d_2, d_3 et d_6 car $J, a \notin Out(B_1)$ et $a \notin Out(B_3)$.