Génération de code

Fabienne Carrier & Laurent Mounier & Catherine Parent

Université Grenoble Alpes

7 janvier 2020

Plan

Langage

Langage impératif sans blocs ni procédures

$p \to d\ ;\ c$
$d \to \mathit{var}\ x \mid d; d$
$c \to x := a \mid c\ ;\ c \mid \mathit{si}\ b\ \mathit{alors}\ c\ \mathit{sinon}\ c \mid \mathit{tantque}\ b\ c$
$a \to n \mid x \mid a + a \mid a * a$
$b \to b\ \mathit{et}\ b \mid a = a \mid \mathit{non}\ b$

- Les termes sont bien typés
- Les expressions arithmétiques et booléennes sont distinctes

Domaines sémantiques

AEXP : expressions arithmétiques

BEXP : expressions booléennes

INST : commandes (ou instructions)

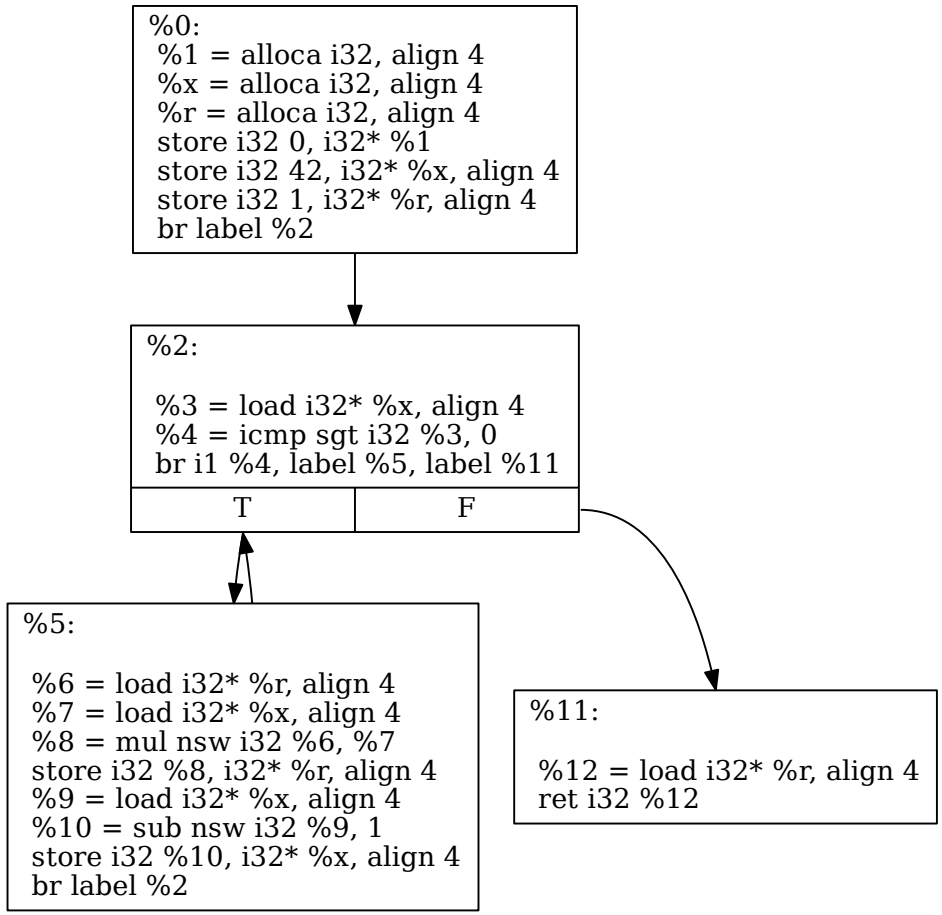
Example 2 : Java ByteCode (machine à pile)

```
public static
    int main() {
int x, r;
x=42 ; r=1 ;
while (x>0) {
   r = r*x;
   x = x-1;
} ;
return r ;
}
```

```
public static int main(java.lang.String[]);
   Code:
      0: bipush        42
      2: istore_1
      3: iconst_1
      4: istore_2
      5: iload_1
      6: ifle          20
      9: iload_2
     10: iload_1
     11: imul
     12: istore_2
     13: iload_1
     14: iconst_1
     15: isub
     16: istore_1
     17: goto          5
     20: iload_2
     21: ireturn
```

Example 2 : LLVM IR (machine à registre)

```
int main() {
int x, r;
x=42 ; r=1 ;
while (x>0) {
    r = r*x;
    x = x-1;
} ;
return r ;
}
```



CFG for 'main' function

Machine de type RISC

Machine virtuelle inspirée du ARM

- Registres notés Ri (nombre illimité)
- Trois registres particuliers : compteur programme (PC), sommet de pile (SP), base de l'environnement courant (FP)
- Adresses, instructions, entiers codés sur 4 octets
- Instructions : opérations arithmétiques et logiques, transfert registre/mémoire, chargement et comparaisons de registres, branchements, appel/retour de procédure.

Les instructions(1/2)

Notations

`val` désigne une valeur entière

`adr` désigne une adresse de la forme :
adr : := Ri + Rj | Ri + val | Ri | val

`OPER` = {`ADD`, `SUB`, `AND`, ...}

`BRANCH cond` = {`BA`, `BEQ`, `BNE`, `BGT`, ...}
Les conditions de branchement `EQ`, `GT`, ... sont associées à des expressions booléennes portant sur les codes de condition arithmétiques `Z, N, C , V`

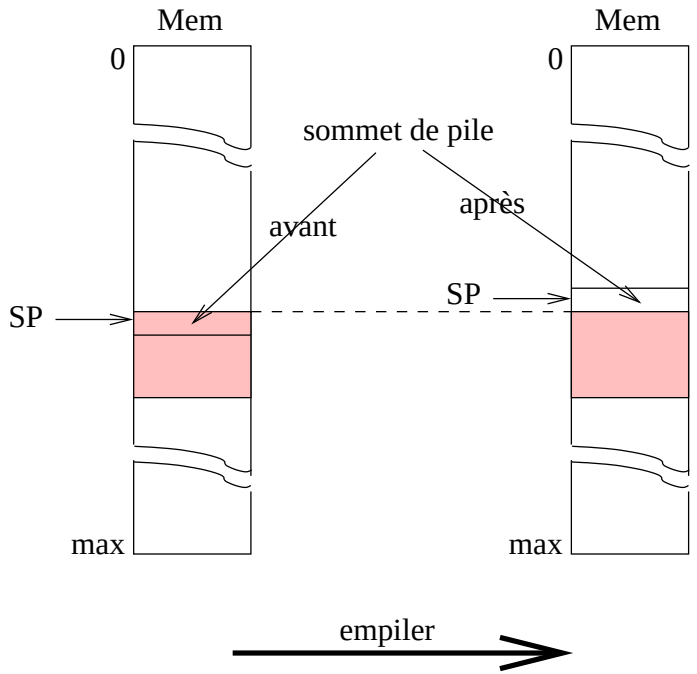
Les instructions(2/2)

instruction	sémantique informelle
`MOV Ri, Rj`	Ri ← Rj
`MOV Ri, val`	Ri ← val
`OPER Ri, Rj, Rk`	Ri ← Rj oper Rk
`OPER Ri, Rk, val`	Ri ← Rj oper val
`CMP Ri, Rj`	Ri-Rj + mise à jour de Z,N,C,V
`LD Ri, [adr]`	Ri ← Mem[adr]
`ST Ri, [adr]`	Mem[adr] ← Ri
`BRANCH cond label`	si cond alors PC ← adresse label
	sinon PC ← PC + 4
`CALL label`	sauvegarde de l'adresse de retour dans la pile
	puis branchement à la procédure label
`RET`	retour de procédure en dépilant dans PC

Fonctionnement de la pile

SP repère le dernier mot empilé

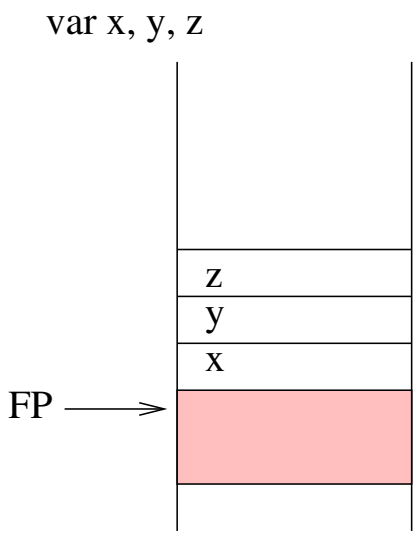
La pile progresse vers les adresses basses



Stockage des variables

Conventions :

- Les variables sont allouées sur la pile
- L'accès aux variables est fait par un adressage relatif au registre FP



Fonctions auxiliaires

`AllouerRegistre` : $\rightarrow \mathbb{N}$

alloue un nouveau registre

`nouvelleEtiq` : $\rightarrow \mathbb{N}$

fournit une nouvelle étiquette

`GetSymbDepl` : Noms $\rightarrow \mathbb{N}$

renvoie le déplacement (`depl`) associé à la variable spécifiée.

Représentation du code

Code dénote une instruction de la machine cible

On note Code* une séquence de codes

L'opérateur de concaténation de séquence est noté : @

Génération de code pour les expressions arithmétiques (1/2)

`GenCodeAExp` : AExp $\rightarrow$ Code* $\times$ $\mathbb{N}$

{`GenCodeAExp(a)` *produit un couple (*`C`*,* `i`*) où* `C` *est le code permettant de calculer la valeur de* `a` *et de la mémoriser dans le registre* `Ri`}

Génération de code pour les expressions arithmétiques (2/2)

`GenCodeAExp(x)`	=	i ← `AllouerRegistre();` k ← `GetSymbDepl(x);` return ((`LD` Ri,[FP-k]),i)
`GenCodeAExp(n)`	=	i←`AllouerRegistre();` return ((`MOV` Ri,n),i)
`GenCodeAExp(`a_1 `+` a_2`)`	=	(C_1,i_1)←`GenCodeAExp(`a_1`);` (C_2,i_2)←`GenCodeAExp(`a_2`);` k←`AllouerRegistre();` return ((C_1 @ C_2 @ `ADD` Rk, Ri_1,Ri_2),k)

Génération de code pour les expressions booléennes (1/2)

`GenCodeBExp` : BExp$\times$ $\mathcal{L}$abel$\times$ $\mathcal{L}$abel $\rightarrow$ Code*

{`GenCodeBExp(b, lvrai, lfaux)` *produit le code* `C` *permettant de calculer la valeur de* `b` *et d'effectuer un branchement sur* `lvrai` *lorsque cette valeur est "vrai" et sur* `lfaux` *sinon*}

Génération de code pour les expressions booléennes (2/2)

`GenCodeBExp`(a_1=a_2,lvrai,lfaux)	=	(C_1,i_1)←`GenCodeAExp`(a_1) ; (C_2,i_2)←`GenCodeAExp`(a_2) ; return (C_1 @ C_2 @ `CMP` Ri_1, Ri_2 @ `BEQ` lvrai @ `BA` lfaux)
`GenCodeBExp`(b_1 et b_2),lvrai,lfaux)	=	l←`nouvelleEtiq()` ; return (`GenCodeBExp`(b_1,l,lfaux) @ l :@ `GenCodeBExp`(b_2,lvrai,lfaux))
`GenCodeBExp(NON b,lvrai,lfaux)`	=	`GenCodeBExp(b,lfaux,lvrai)`

Génération de code pour les instructions (1/4)

`GenCodeInst` : Inst $\rightarrow$ Code*

{`GenCodeInst(c)` *calcule le code* `C` *permettant d'exécuter la commande* `c`}

Génération de code pour les instructions (2/4)

`GenCodeInst (x := a)`	=	(C,i)←`GenCodeAExp`(a) ; k←`GetSymbDepl`(x) ; return (C @ `ST` Ri, [FP-k])
`GenCodeInst` (c_1 ; c_2)	=	C_1 ← `GenCodeInst`(c_1) ; C_2 ← `GenCodeInst`(c_2) ; return (C_1 @ C_2)

Génération de code pour les instructions (3/4)

```
GenCodeInst(tantque b c)  =  ldebut←nouvelleEtiq();
                             lvrai←nouvelleEtiq();
                             lfaux←nouvelleEtiq();
                             return (
                             ldebut :@
                             GenCodeBExp(b,lvrai,lfaux)@
                             lvrai :@
                             GenCodeInst(c)@
                             BA ldebut@
                             lfaux :
                             )
```

Génération de code pour les instructions (4/4)

`GenCodeInst` (si b alors c_1 sinon c_2)=

lsuivant←`nouvelleEtiq()`;
lvrai←`nouvelleEtiq()`;
lfaux←`nouvelleEtiq()`;
return (
`GenCodeBExp`(b,lvrai,lfaux) @
lvrai :
`GenCodeInst`(c_1) @
`BA` lsuivant @
lfaux :@
`GenCodeInst`(c_2) @
lsuivant :
)

Exemple procédure simple

```
procedure P (int a) {
int x;
  x = a;
}

procedure Q (int b) {
int y;
  y = b;
  call P (7);
}
```

en langage d'assemblage

call P (7)

```
        push 7 @
        call P @
        add sp, sp, 4
```

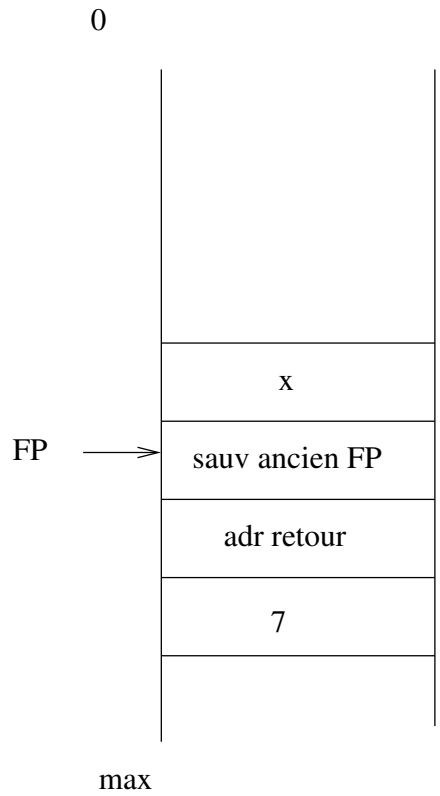
Note : push 7 =

```
sub sp, sp, 4 @
mov R0, 7 @
str R0, [SP]
```

Exemple : pile lors de l'exécution de P

```
procedure P (int a) {
int x;
  x = a;
}

procedure Q (int b) {
int y;
  y = b;
  call P (7);
}
```



accès à `a` : FP + 8

accès à `x` : FP - 4

Code de la procédure P

```
P : push FP          | prologue
    MOV FP, SP       | parametre = taille zone variables
    ADD SP, SP, -4   |             locales

    LDR R0, FP+8     | corps de la procedure
    STR R0, FP-4

    MOV SP, FP       | epilogue
    pull FP          | pull FP = LD FP, [SP] || ADD SP, SP, 4

    RET              | retour
```

Accès aux objets non locaux : exemple

```
// -> niveau imbrication 0
procedure R0 (int a, int b) {
int u, v;       // -> niveau imbrication 1
   procedure P1 (int x, y) {
   int z;          // -> niveau imbrication 2
      z = x+y+u;
   }
   procedure Q1 (int c) {
   int y;          // -> niveau imbrication 2
      y=c+u+v;
      P1(1, y+4);
   }
   u=3; v=9; Q1(u+v);
}
main() {
   R0(4, 2)
}
```

Notion de lien statique

Pour implémenter une sémantique à liaison statique

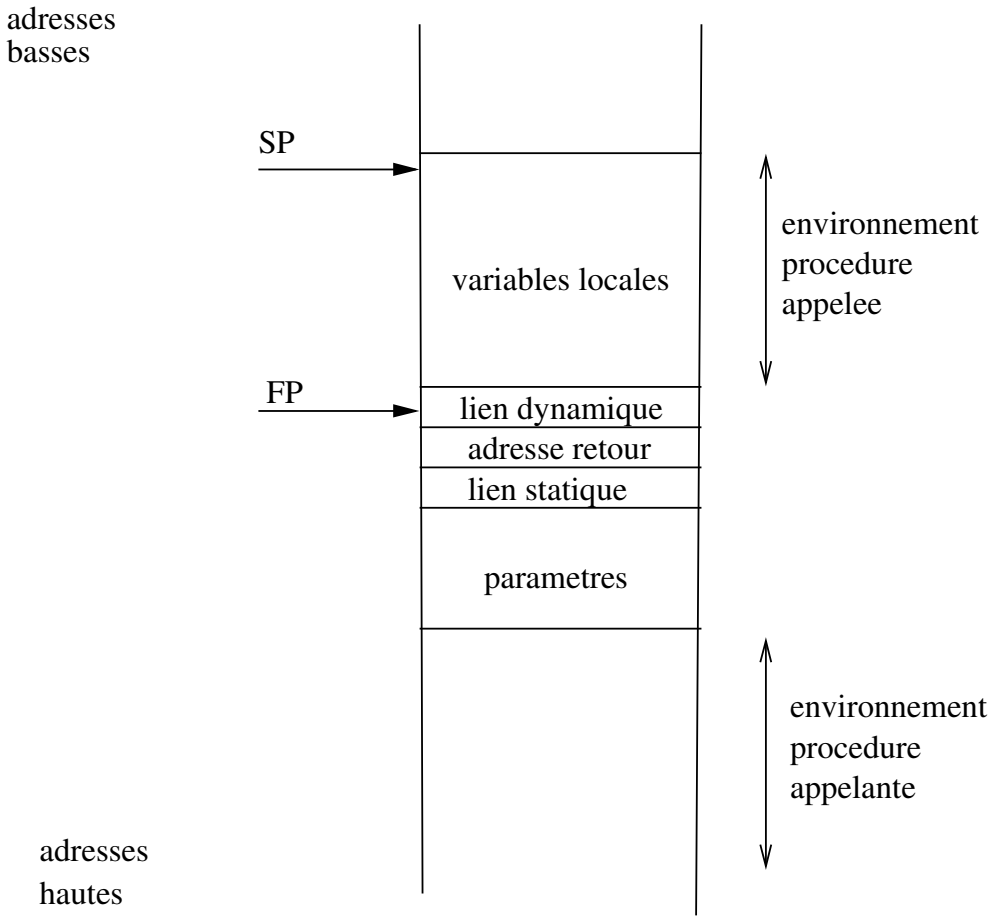
Pour accéder aux objets non locaux

Chaque procédure doit avoir un lien vers l'environnement de la procédure dans laquelle elle est définie $->$ lien statique

Note : il s'agit de l'environnement conservé avec la procédure (Cf. sémantique des procédures en cas de liaison statique)

Note : le lien permettant de rétablir l'environnement au retour s'appelle lien dynamique.

Organisation de la pile



Utilisation du lien statique

Pour chaque variable, chaque procédure (i.e. chaque nom) on garde : le type, le déplacement dans la pile et le niveau d'imbrication

Générer autant d'indirections sur le lien statique que la différence entre le niveau d'imbrication de la procédure courante et le niveau d'imbrication de la définition cherchée

Calcul du lien statique

Pour une procédure appelée ayant un seul niveau d'imbrication de différence avec la procédure courante, le lien statique est égal au FP courant (celui de l'appelant)

Si la différence entre le niveau de la procédure courante et le niveau de la procédure appelée est > 1, il faut calculer autant d'indirections sur le lien statique que cette différence.

Une procédure ayant le même niveau d'imbrication que la procédure appelante a le même lien statique.

Exemple (1/3)

Accès à u dans la procédure P1 :

Niveau d'imbrication dans P1 = 2

Niveau de déclaration de u = 1

```
ld R2, [FP+8]  -- lien statique de P1
ld R2, [R2-4]  -- R1 == u
```

Code de z = x + y + u

```
ld R0, [FP+16]   -- x
ld R1, [FP+12]   -- y
ld R2, [FP+8]    -- lien statique de P1
ld R2, [R2-4]    --  u
add R0, R0, R1
add R0, R0, R2   -- x+y+u
st R0, [FP-4]    -- z=...
```

Exemple (2/3)

Appel à P1(1, y+4) dans Q1 :
Niveau de declaration de P1 = 1
Niveau de declaration de Q1 = 1
Ils ont même lien statique

```
mov R0, 1
push R0          -- empiler 1
ld R0, [FP-4]
add R0, R0, 4
push R0         -- empiler y+4
ld R0, [FP+8]
push R0         -- empiler lien statique
call P1         -- appel
add sp, sp, 12 -- retablir la pile
```

Exemple (3/3)

Appel à Q1(u+v) dans R0 :
Niveau de declaration de Q1 = 1
Niveau R0 = 0
Le lien statique de Q1 est la base de l'environnement de R0 (c.a.d. FP courant)

```
ld R0, [FP-4]
ld R1, [FP-8]
add R0, R0, R1    -- R0 == u+v
push R0           -- empiler u+v
push FP           -- empiler lien statique
call Q1           -- appel
add sp, sp, 8     -- retablir la pile
```