Génération de Code

CS410 - Langages et Compilation

Julien Henry
Catherine Oriat

Grenoble-INP Esisar

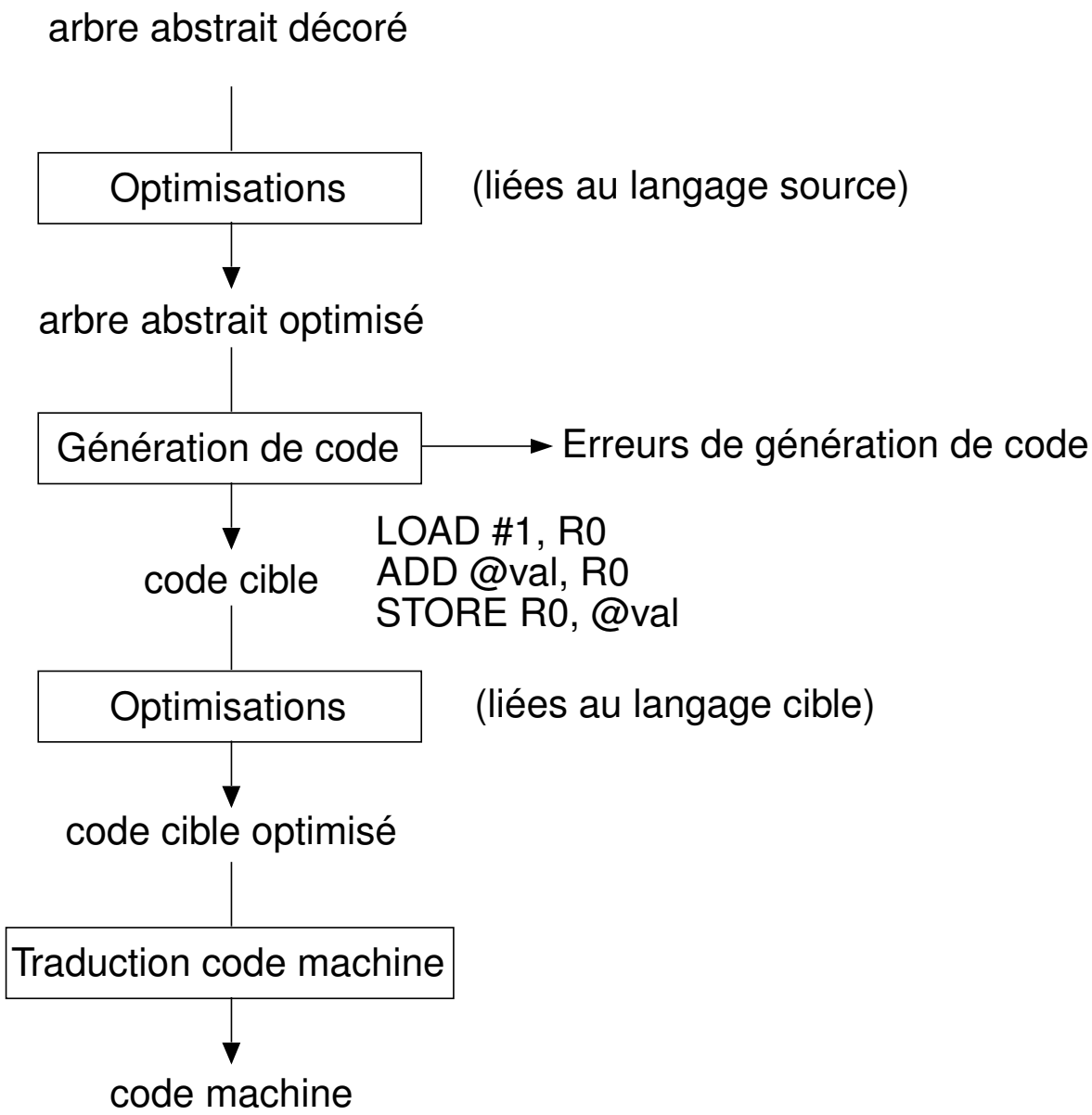
2013-2014

Objectif

La passe de génération de code sert à produire à partir de l'AST décoré obtenu après l'analyse contextuelle un code dans le langage cible.
Le langage cible peut être :

- du code machine ou assembleur pour une architecture donnée.
- un langage de plus haut niveau (par exemple C).
- du code pour machine abstraite (JVM, machine abstraite du TP Jcas)

Vue d'ensemble



Exemple de traduction

Le code assembleur x86 généré par clang à partir du programme C suivant :

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

Avec gcc :

```
gcc -S -masm=... f.c
```

```
    .file   "f.bc"
    .text
    .globl  f
    .align  16, 0x90
    .type   f,@function
f:
# BB#0:
    subl    $8, %esp
    movl    12(%esp), %eax
    movl    %eax, 4(%esp)
    movl    $42, (%esp)
    movl    4(%esp), %eax
    addl    $42, %eax
    addl    $8, %esp
    ret
.Ltmp0:
    .size   f, .Ltmp0-f
```

Exemple de traduction

Pour générer du code dans le langage d'assemblage, on a donc besoin d'utiliser :

- des registres du processeur (`%esp`,`%eax`, etc)
- des adresses de la mémoire pour stocker les variables / le code du programme
  - le langage d'assemblage contient des instructions pour lire / écrire dans la mémoire (`movl`)
  - les adresses doivent être calculées (`4(%esp)`) : il faut savoir à quelle adresse se trouve chacuns des éléments de notre programme.
  - Les adresses de chacun sont relatives : ici, dépendantes de la valeur du registre `%esp`.

Registres, mémoire

Registres : mémoire rattachée au processeur, très rapide d'accès, mais en quantité très limitée.

Mémoire : mémoire dont l'accès coûte plus cher, mais de capacité beaucoup plus importante. Des instructions assembleur spéciales permettent de lire et d'écrire dans cette mémoire.

Conclusion

Il faut savoir comment est organisée la **mémoire** pour pouvoir générer du code dans un langage d'assemblage !

- Les instructions assembleur travaillent sur des registres : certaines opérandes sont nécessairement des registres, et les résultats sont stockés dans des registres.
- Le code assembleur doit si possible utiliser les mémoires de façon optimisée.

Summary

Mémoire

- La mémoire est composée de mots. chaque mot a une adresse unique
- Selon l'architecture, les mots ont une certaine longueur (ex : 32 bits :
  - Un élément dont le type peut être représenté en $\leq$ 32 bits occupe donc 1 mot de la mémoire.
  - Si le type est plus compliqué (tableau, structure, etc), un élément occupera alors plusieurs mots d'adresses consécutives de la mémoire.

Instructions du programme

Une partie de la mémoire est réservée au code du programme.

- Chaque instruction occupe une adresse de la mémoire.
- Deux instructions consécutives sont stockées à deux adresses consécutives.
- Le code s'exécute de manière séquentielle : sauf instruction explicite de saut, les instructions sont exécutées les unes après les autres.
- Si il y a un branchement, il faut savoir à quelle instruction il faut sauter.

Program counter

On utilise un registre spécial du processeur, dont la valeur est l'adresse de l'instruction à exécuter : **pc** (*program counter*)

- Si il y a un saut, on spécifie l'adresse *addr* de l'instruction vers laquelle on saute. `pc ← addr`.
- Si il n'y a pas de saut, l'adresse de l'instruction suivante est simple à calculer. `pc ← pc+decalage`.

La valeur de `decalage` dépend du nombre de ligne de la mémoire utilisée pour encoder l'instruction.
Exemple : Si l'instruction est codée sur 32 bits, et que la mémoire a des lignes d'un octet, alors l'instruction est codée sur 4 lignes. L'instruction suivante est donc à `pc+4`

Allocation mémoire

Lors de la compilation, il faut associer aux identificateurs du programme un espace de la mémoire qu'ils vont occuper pendant l'exécution.

- les identificateurs sont donc associés à des adresses : il faut choisir l'emplacement mémoire de l'objet associé à cet identificateur.
- selon le type de l'identificateur, il faut choisir la taille de l'espace mémoire alloué à cet identificateur.

Allocation mémoire

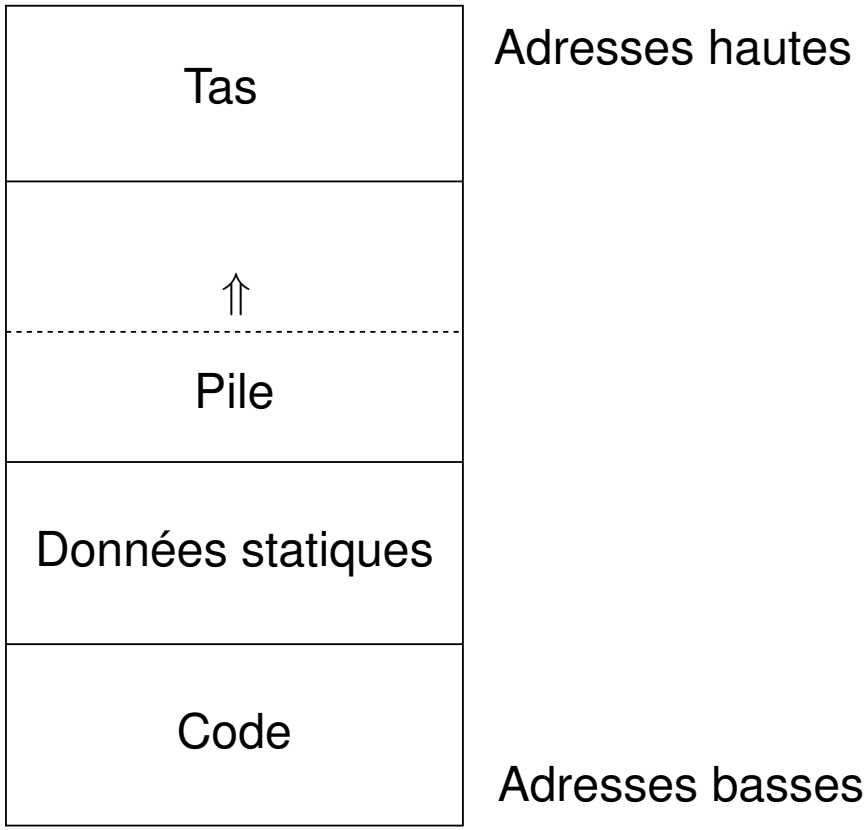
Il existe 3 sortes d'allocation mémoire :

- allocation **statique** : elle fixe l'emplacement mémoire de l'identificateur pour toute l'exécution du programme.
- allocation sur la **pile** : permet de définir l'emplacement mémoire des variables locales et les paramètres des fonctions.
- allocation dans le **tas** : permet d'allouer de l'espace mémoire à un moment quelconque du programme. Cet espace mémoire est alloué jusqu'à ce qu'il soit libéré explicitement. (en C, l'allocation et la libération dans le tas se fait avec les fonctions **malloc** et **free**).

Etat de la mémoire pendant l'exécution

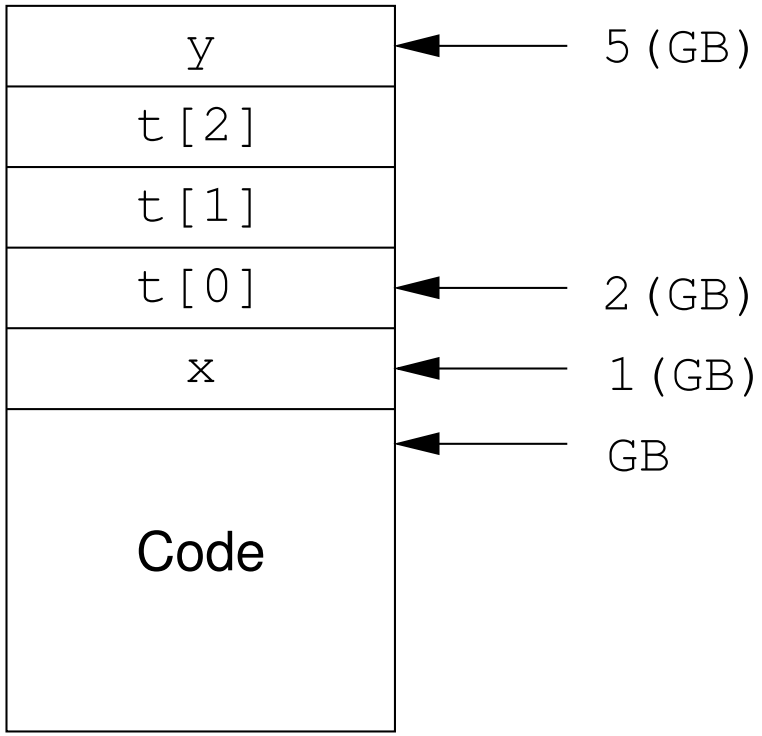
Code contient les instructions du programme.

Le positionnement de chacunes de ces zones dépend de l'architecture.



Données statiques

Les données statiques sont relatives à une certaine adresse appelée **base globale**. La valeur de la base globale est contenue dans un registre spécial du processeur (Dans le projet, il s'agit de GB).



déclaration de 3 variables globales en C :

```
int x;
int t[3];
int y;
```

Pile

La pile est une zone mémoire dans laquelle on peut allouer ou désallouer en sommet de pile.
Dans quels cas utilise-t-on la pile ?

- pour définir des variables locales à une fonction
- pour faire des appels de fonction : il faut alors savoir :
  - quels sont ses paramètres
  - quel est l'endroit du code ayant appelé la fonction
  - ...

Pour connaître la position des éléments de la pile, on utilise deux registres spéciaux :

- Stack pointer (SP) : le sommet de la pile.
- Local base (LB) : pour savoir où commence la zone mémoire utilisée par la fonction courante.

Allocation dans le tas

Allocation dans le tas = Allocation **dynamique**

- En C, on utilise `malloc` et `free`
- En C++, on utilise `new` et `delete`
- En Java, on utilise `new`, la désallocation est automatique à l'aide d'un ramasse-miette (**garbage collector**)

Le choix de l'emplacement dans la mémoire lors d'une allocation est codée dans la bibliothèque standard du langage, qui est liée au programme lorqu'on crée l'exécutable.
La bibliothèque standard du langage peut faire des appels à l'OS pour demander d'agrandir son tas.

Fonctionnement général d'un ramasse-miette

Principe :

1. On parcourt le tas et on marque tous les objets comme étant inaccessibles.
2. On parcourt le tas une seconde fois à partir des variables locales et globales du programme, et on marque *accessible* tous les objets que l'on a pu atteindre.
3. On libère tous les emplacements correspondant aux éléments qui sont *inaccessibles*

Summary

Un exemple de langage d'assemblage : MIPS

Le MIPS comporte 32 registres généraux interchangeables, sauf

- le registre `zero` qui vaut toujours 0, même après une écriture.
- Le registre `ra`, utilisé implicitement par certaines instructions pour sauver l'adresse de retour avant un saut.

Les autres registres ont des utilisations préférentielles, mais cela n'est strict que pour communiquer avec d'autres programmes (par exemple fournir ou utiliser des programmes en librairies) :

- passage (`a0, ... a3`) et retour (`v0, v1`) des arguments ;
- registres temporaires sauvegardés (`s0, .. s7`) ou non (`t0, ... t9`) par les appels de fonction.
- pointeurs de pile `sp`, `fp` ou de donnée `gp` ;

MIPS : Instructions

Les instructions ont des opérandes. Il y a plusieurs types d'opérandes :

- constantes entières (`n`) :
- étiquette (`l`) : `loop_end`
- registre (`r`) : `$v0, $sp`
- absolu (`a`) : constante entière ou étiquette
- opérande : `a` ou `r`

MIPS : Instructions

La plupart des instructions suivent le modèle :

- `add r1, r2, o` qui place dans `r1` la valeur `r2+o`.

Les instructions qui interagissent avec la mémoire sont uniquement les instructions load et store.

- `lw r1, n(r2)` place dans `r1` le mot contenu à l'adresse `r2+n`.
- `sw r1, n(r2)` place `r1` dans le mot contenu à l'adresse `r2+n`.

Les instructions de contrôle conditionnel ou inconditionnel :

- `bne r, a, l` saute à l'adresse `l` si `r` et `a` sont différents,
- `jal o` qui sauve `pc+1` dans `ra` et saute à l'étiquette `o`.

MIPS : Instructions

Un certain nombre d'instructions MIPS :

```
move r1, r2
add r1, r2, o
sub r1, r2, o
mul r1, r2, o
div r1, r2, o
and r1, r2, o
or r1, r2, o
xor r1, r2, o
sll r1, r2, o
srl r1, r2, o
li r1, n
la r1, a
nop
syscall
```

```
lw r1, o(r2)
   r1 <- contenu de (r2 + o)
sw r1, o(r2)
   r1 <- contenu de (r2 + o)
slt r1, r2, o
sle r1, r2, o
seq r1, r2, o
sne r1, r2, o
j o
jal o   ra <- pc+1 et pc <- o
beq r, o, a
bne r, o, a
blt r, o, a
ble r, o, a
```

Exemples de code MIPS

```
if t1 < t2 then t3 := t1 else t3 := t2
```

on considère que toutes les variables sont déjà dans les registres du même nom :

```
        blt   $t1, $t2, Then      # si t1 >= t2 saut a Then
        move  $t3, $t2            # t3 := t1
        j     End                 # saut a Fi
Then:   move  $t3, $t1            # t3 := t2
End:                              # suite du programme
```

Exemples de code MIPS

calcule et affiche la longueur de la chaine str = "Hello world"

```
        .text
        .globl __start
__start:                    # debut
        la $t2,str          # t2 pointe vers la chaine
        li $t1,0            # t1 contient la longueur
nextCh: lb $t0,($t2)        # prendre un bit de la chaine
        beqz $t0,strEnd     # zero -> fin de la chaine
        add $t1,$t1,1       # incremente longueur
        add $t2,1           # incremente le pointeur
        j nextCh            # recommence...
```

Exemples de code MIPS : suite

suite...

```
strEnd: la $a0,ans      # appel systeme a print
        li $v0,4
        syscall
        move $a0,$t1    # appel systeme a print
        li $v0,1
        syscall
        la $a0,endl     # appel systeme a print
        li $v0,4        # newline
        syscall

        li $v0,10
        syscall         # au revoir...

        .data
str:    .asciiz "hello world"
ans:    .asciiz "Length is "
endl:   .asciiz "\n"
```

Summary

Objectifs

Entrée : un arbre de syntaxe abstraite représentant un programme
Sortie : le code cible associé à l'AST

Exemple : pour un sous-arbre de l'AST définissant le calcul d'une expression arithmétique, on peut générer une portion du programme cible effectuant ce calcul.

```
x = x + 42;        ⟶
```

```
LOAD  @x, R1
ADD   #42, R1
STORE R1, @x
```

Instructions du langage cible

Les instructions assembleur accèdent aux données grâce à différents modes d'adressage :

- Adressage **immédiat** : `#42, #10, #1.2E-4...`
- Adressage **immédiat par registres** : `R0, R1, ..., LB, GB`
- Adressage **indirect avec déplacement** : `4(LB)` pour une variable locale, `1(GB)` pour une variable globale
- Adressage **indirect indexé avec déplacement** : `2(LB,4)` pour accéder à la 4ème valeur du tableau stocké à l'adresse `2(LB)`.

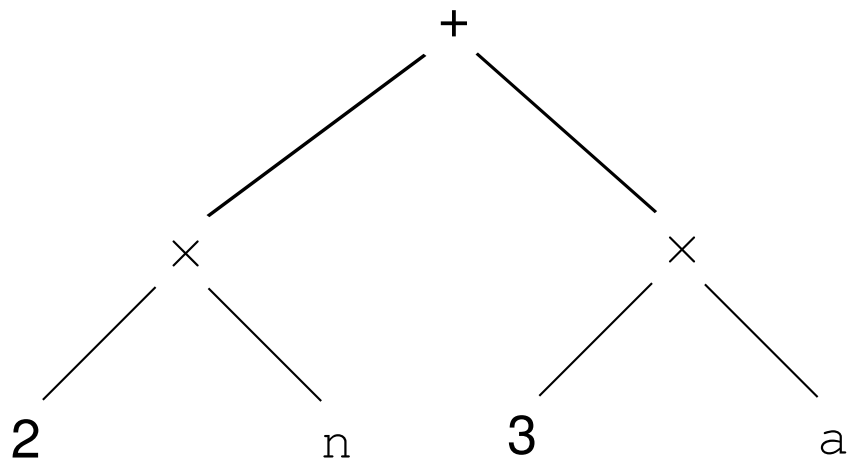
Instructions du langage cible

Le langage cible dispose d'instructions diverses. Exemple :

- `LOAD dval, Ri` : dval peut correspondre à tous les modes d'adressage, mais Ri doit nécessairement être un registre.
  - `LOAD R0, R1`
  - `LOAD #42, R1`
  - `LOAD 1(LB), R1`
- `STORE Ri, dadr` : stocke la valeur stockée dans Ri à l'adresse dadr.
  - `STORE R1, 1(LB)`
  - `STORE R1, #10` n'a pas de sens
- `ADD dval, Ri` : ajoute la valeur stockée à l'adresse `dval` au registre `Ri` (`Ri` $\leftarrow$ `Ri + dval`).
- `SUB dval, Ri`, `MUL dval, Ri`, `DIV dval, Ri`
- `PUSH Ri` : stocke `Ri` en sommet de pile
- `POP Ri` : dépile et stocke dans `Ri`

Exemple

On suppose que `n` est une variable globale stockée à l'adresse `1(GB)` et `a` est une variable locale stockée à l'adresse `4(LB)`. On veut générer le code associé au sous arbre suivant :



Compilation des expressions binaires

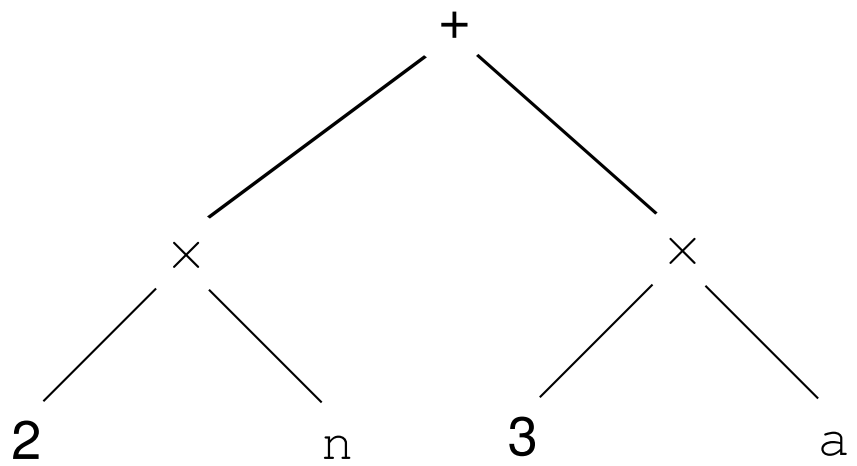
Une méthode automatique pour générer des expressions binaires serait :

- `coder(idf) = generer(LOAD` *addr_idf*`, R0)`
- `coder(num) = generer(LOAD #num, R0)`
- `coder(exp1 + exp2) =`

```
coder(exp1)
generer(PUSH R0)
coder(exp2)
generer(POP R1)
generer(ADD R1, R0)
```

- idem pour $*$, $/$, $-$

Exemple

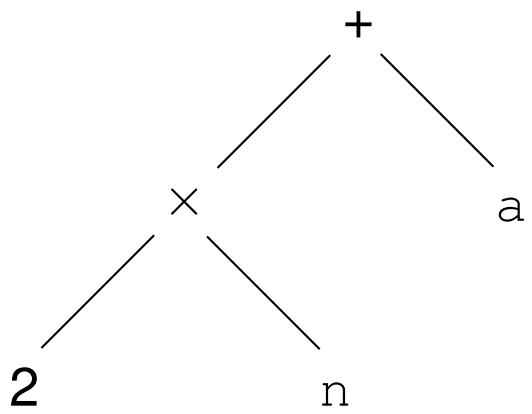



```
LOAD #2, R0
PUSH R0
LOAD 1(GB), R0
POP R1
MUL R1, R0
PUSH R0
LOAD #3, R0
push R0
LOAD 4(LB), R0
POP R1
MUL R1, R0
POP R1
ADD R1, R0
```

Code très inefficace...

Exemple

On suppose encore que `n` est une variable globale stockée à l'adresse `1(GB)` et `a` est une variable locale stockée à l'adresse `4(LB)`. On veut générer le code associé au sous arbre suivant :

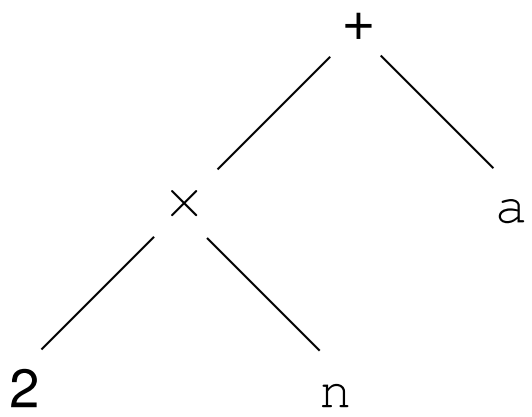


Technique : On charge la partie droite de l'arbre dans un registre pour faire chaque opération :

```
LOAD 1(GB), R1
MUL #2, R1
LOAD 4(LB), R2
ADD R1, R2
```

Utilisation des registres

Selon la technique qu'on utilise pour générer les expressions, le nombre de registres nécessaires pour effectuer une opération peut varier :



Ici, on peut utiliser un seul registre.

```
LOAD 1(GB), R1
MUL #2, R1
ADD 4(LB), R1
```

Utilisation des registres

Pour évaluer une expression arithmétique, on a besoin de registres : chaque opération binaire s'effectue avec un registre.

- A chaque fois, l'un des deux sous arbres doit être évalué et stocké dans un registre.
- L'autre sous arbre doit lui aussi être évalué :
  - sans utiliser le registre dans lequel on a le résultat de l'autre sous-arbre.
  - son résultat ne doit pas nécessairement être contenu dans un registre.
  - Si il n'y a plus assez de registres disponibles, on crée des variables temporaires dans la pile. . .

Si on a un seul registre...

Il faut modifier l'algorithme pour qu'il n'utilise plus qu'un seul registre.

- `coder(idf) = generer(LOAD` *addr_idf*`, R0)`
- `coder(num) = generer(LOAD #num, R0)`
- `coder(exp1 + exp2) =`

  Si `exp2` est une feuille `idf` :

  ```
  coder(exp1)
  generer(ADD addr_idf, R0)
  ```

  Si `exp2` est une feuille `num` :

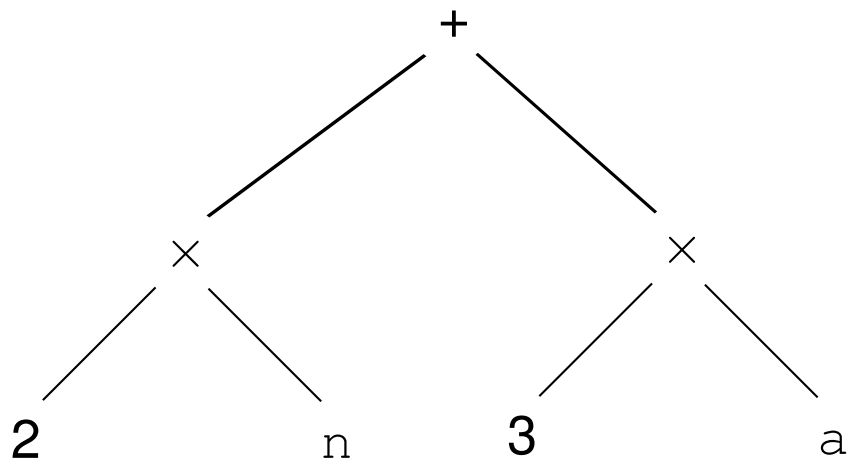
  ```
  coder(exp1)
  generer(ADD #num, R0)
  ```

  Sinon :

  ```
  coder(exp2)
  generer(PUSH R0)
  coder(exp1)
  generer(ADD 0(SP), R0)
  generer(SUB #1, SP)
  ```

Exemple

Quel est le code généré par cet algorithme pour l'arbre suivant ?



Si on a *k* registres disponibles

On essaie d'utiliser au maximum les registres disponibles pour avoir du code généré performant.

Il faut donc savoir pendant la génération de code :

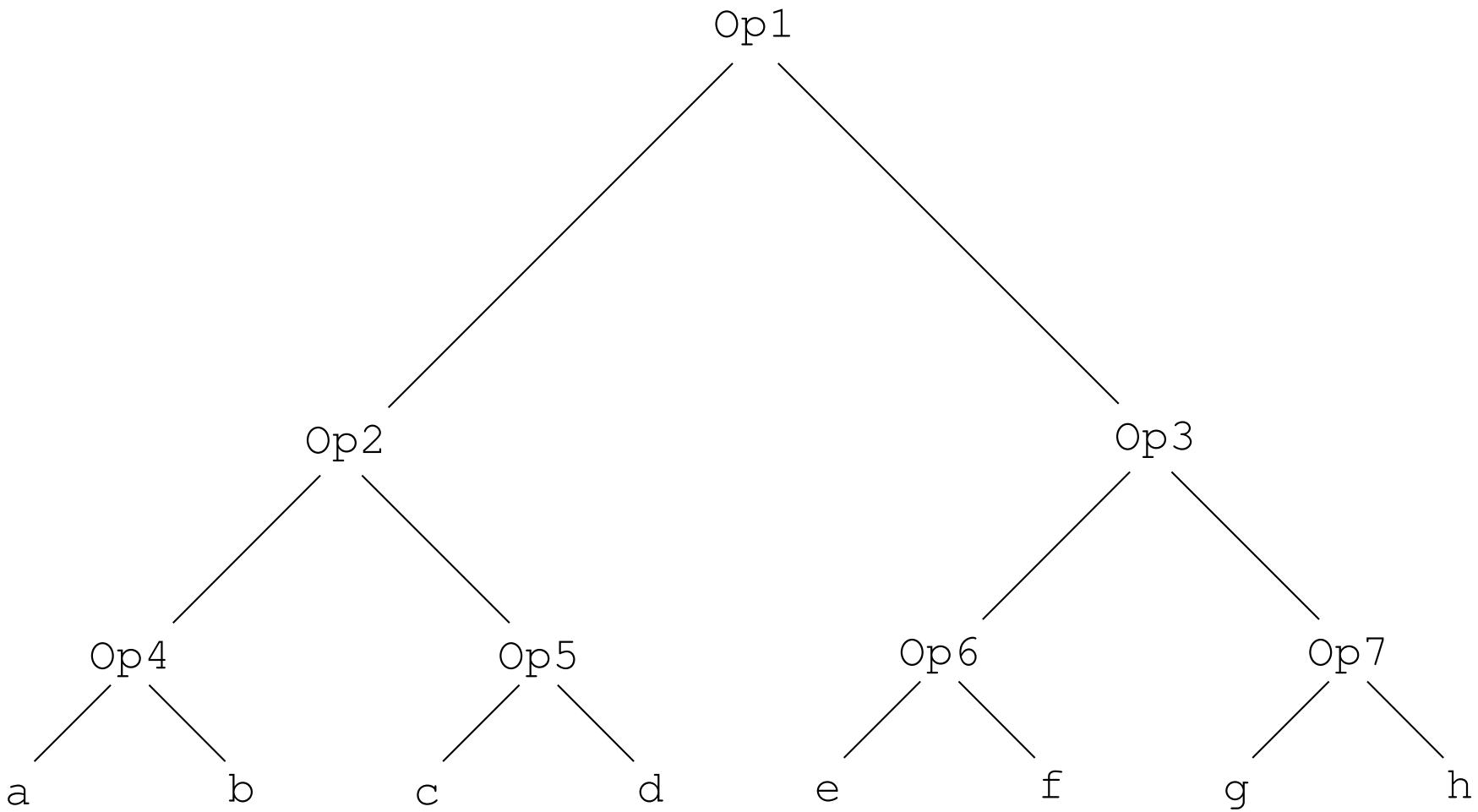
- quels sont les registres libres
- quels sont les registres utilisés (qui contiennent un résultat intermédiaire qu'il ne faut pas écraser)

On implémente donc une classe qui gère l'attribution des registres :

```
public enum Registre {R0, R1, ... }
public class Reg_alloc {
        public Registre get_registre() {...}
        public void liberer_registre(Registre r) {...}
        public Boolean reste_registre() {...}
}
```

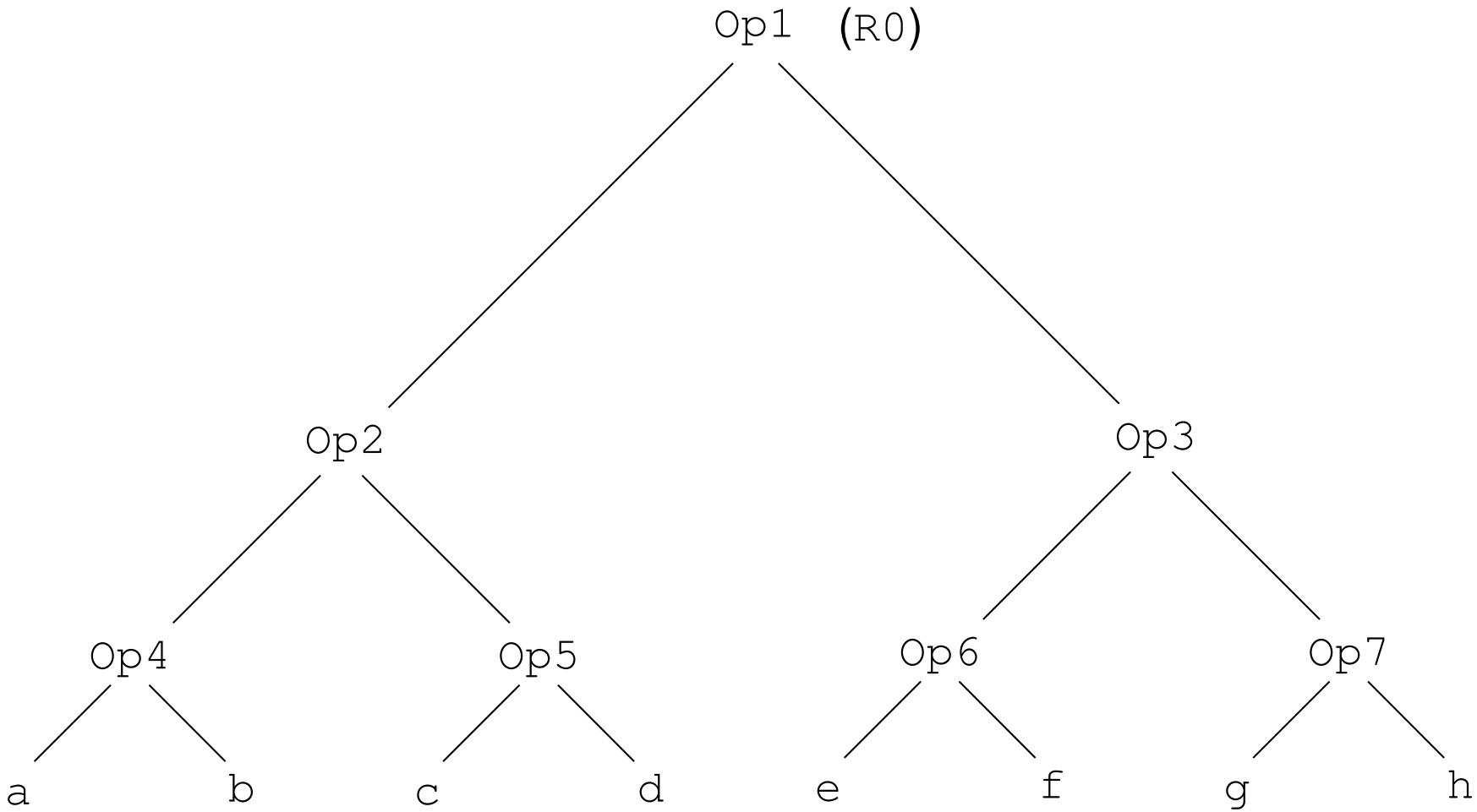
Exemple

Choix des registres pour effectuer les opérations :



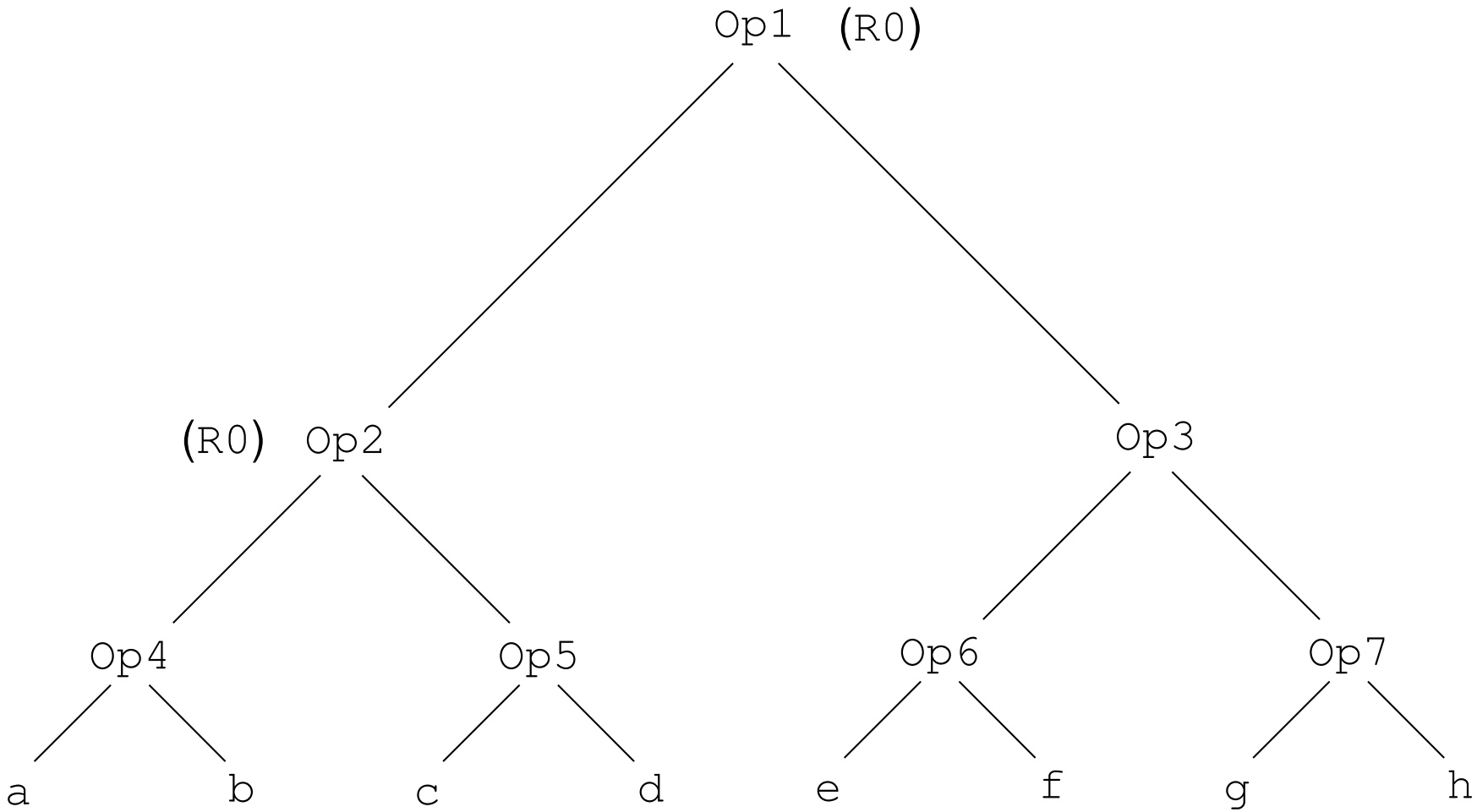
Exemple

Choix des registres pour effectuer les opérations :



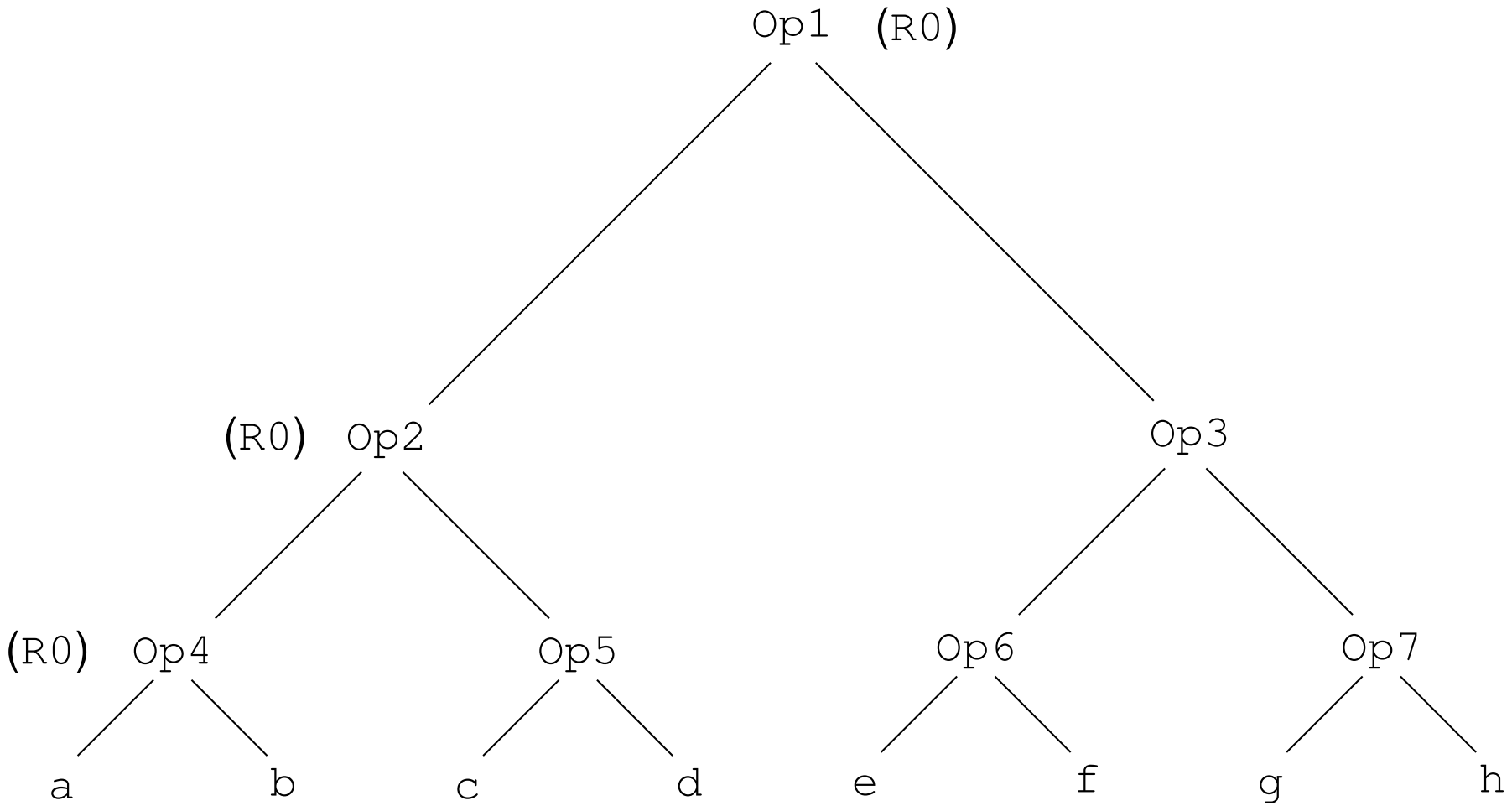
Exemple

Choix des registres pour effectuer les opérations :



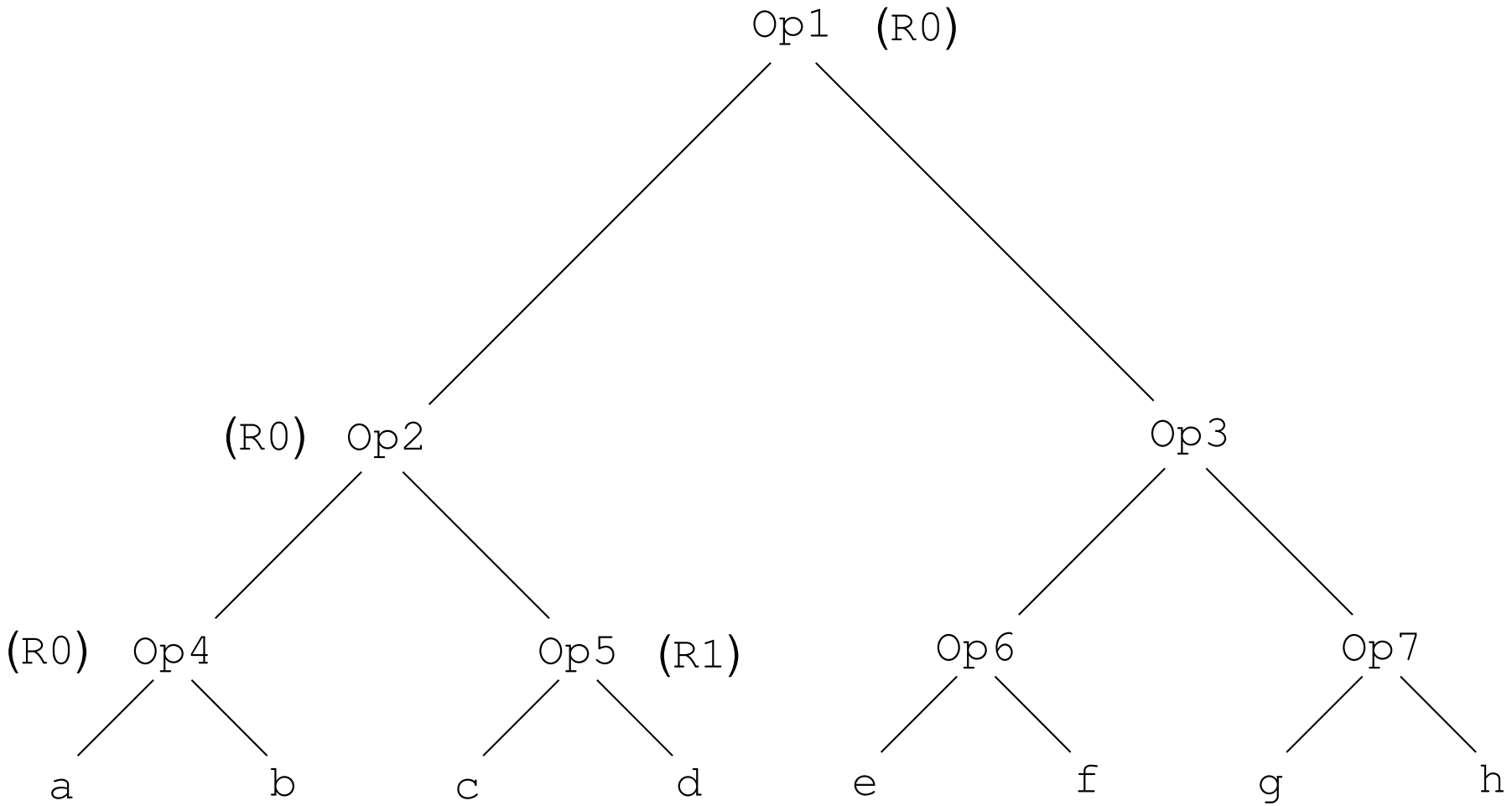
Exemple

Choix des registres pour effectuer les opérations :



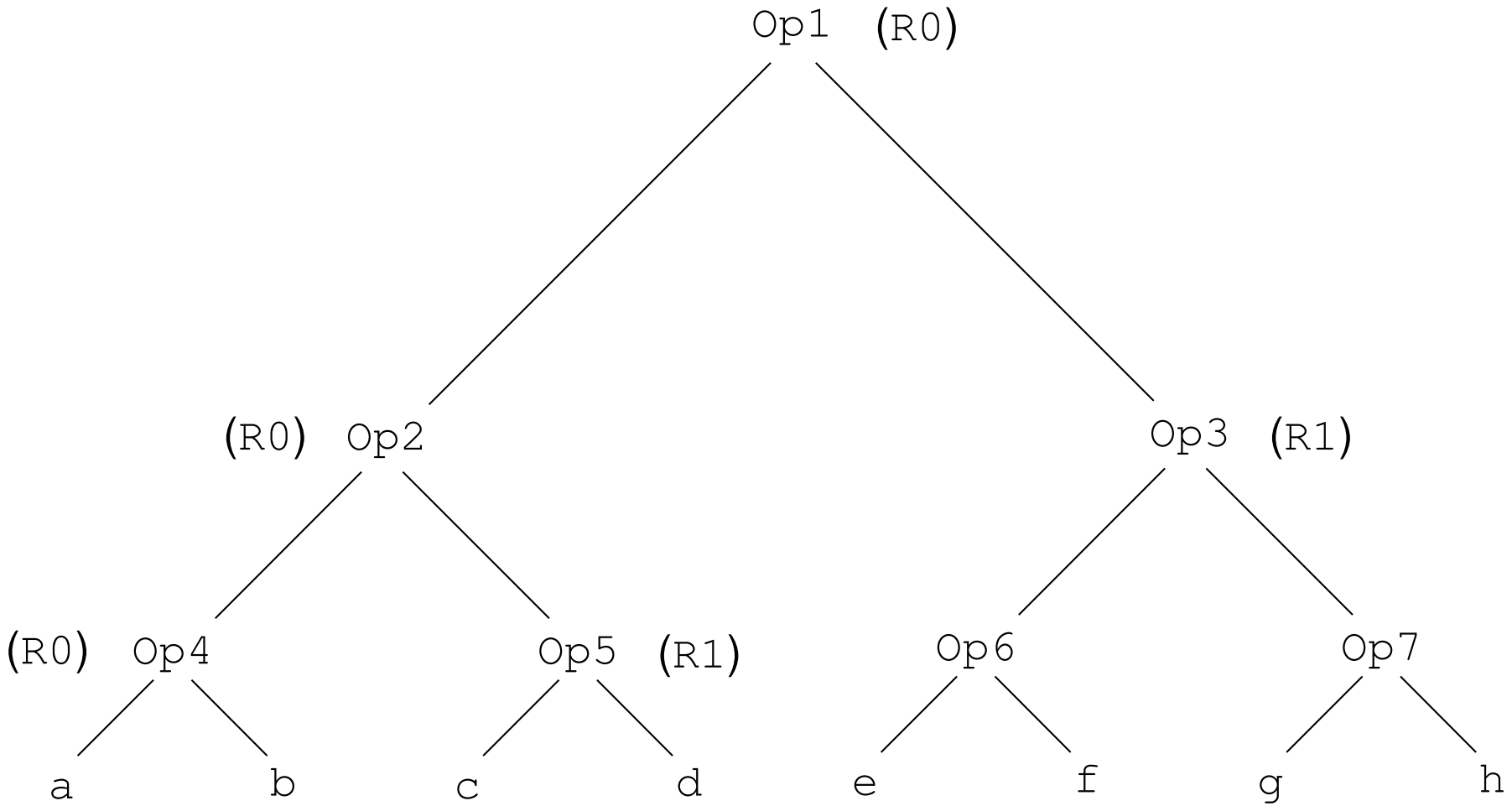
Exemple

Choix des registres pour effectuer les opérations :



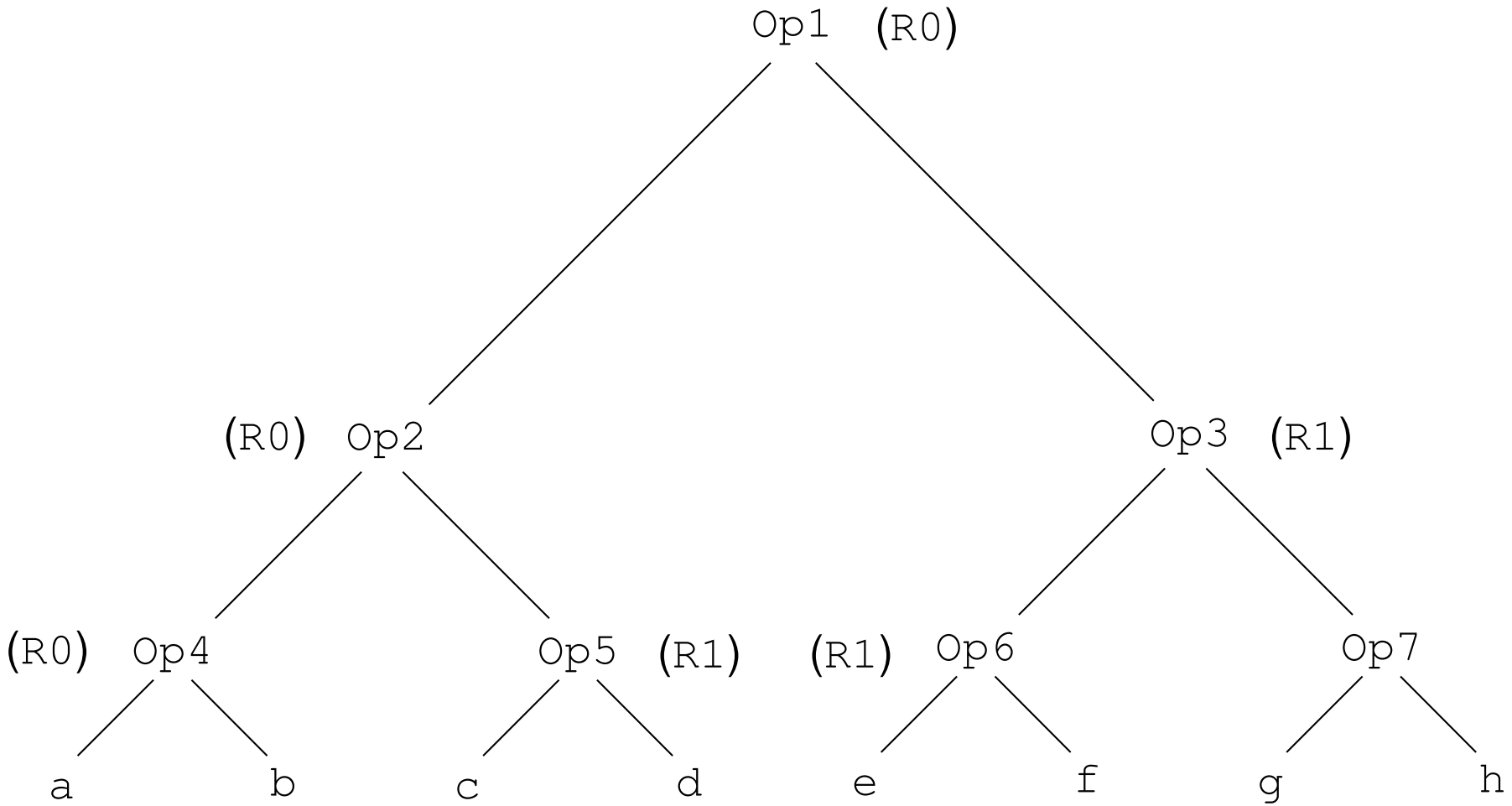
Exemple

Choix des registres pour effectuer les opérations :



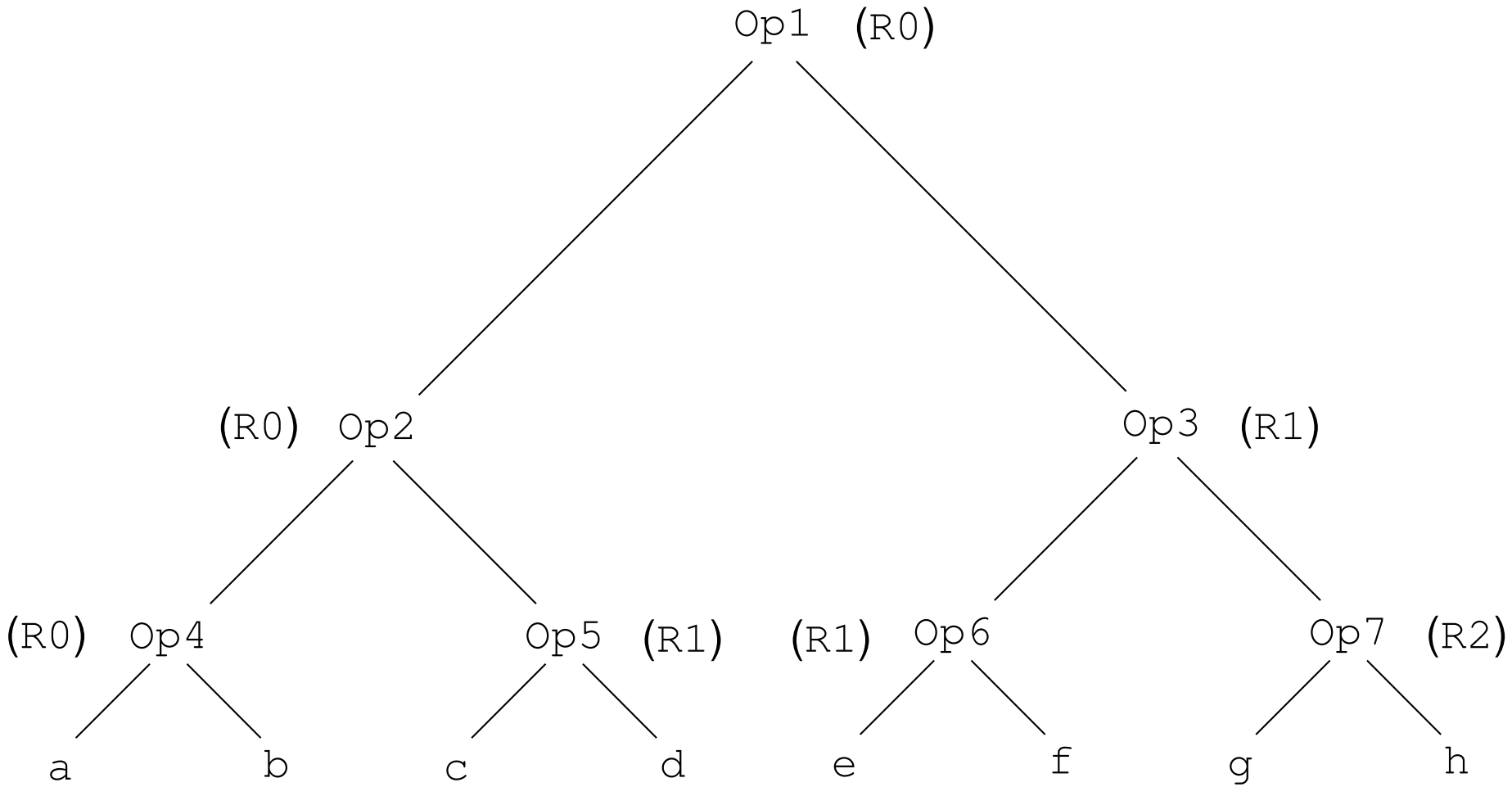
Exemple

Choix des registres pour effectuer les opérations :



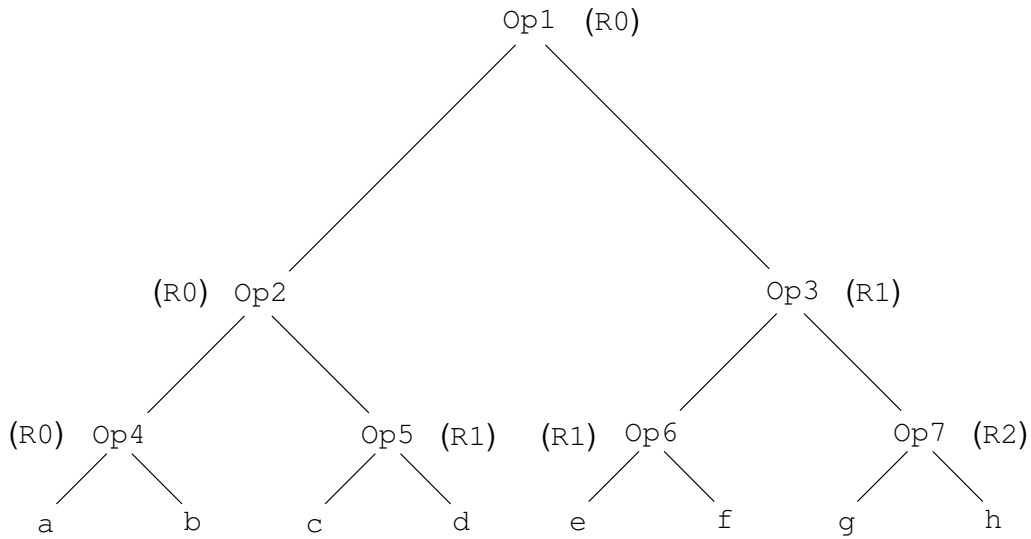
Exemple

Choix des registres pour effectuer les opérations :



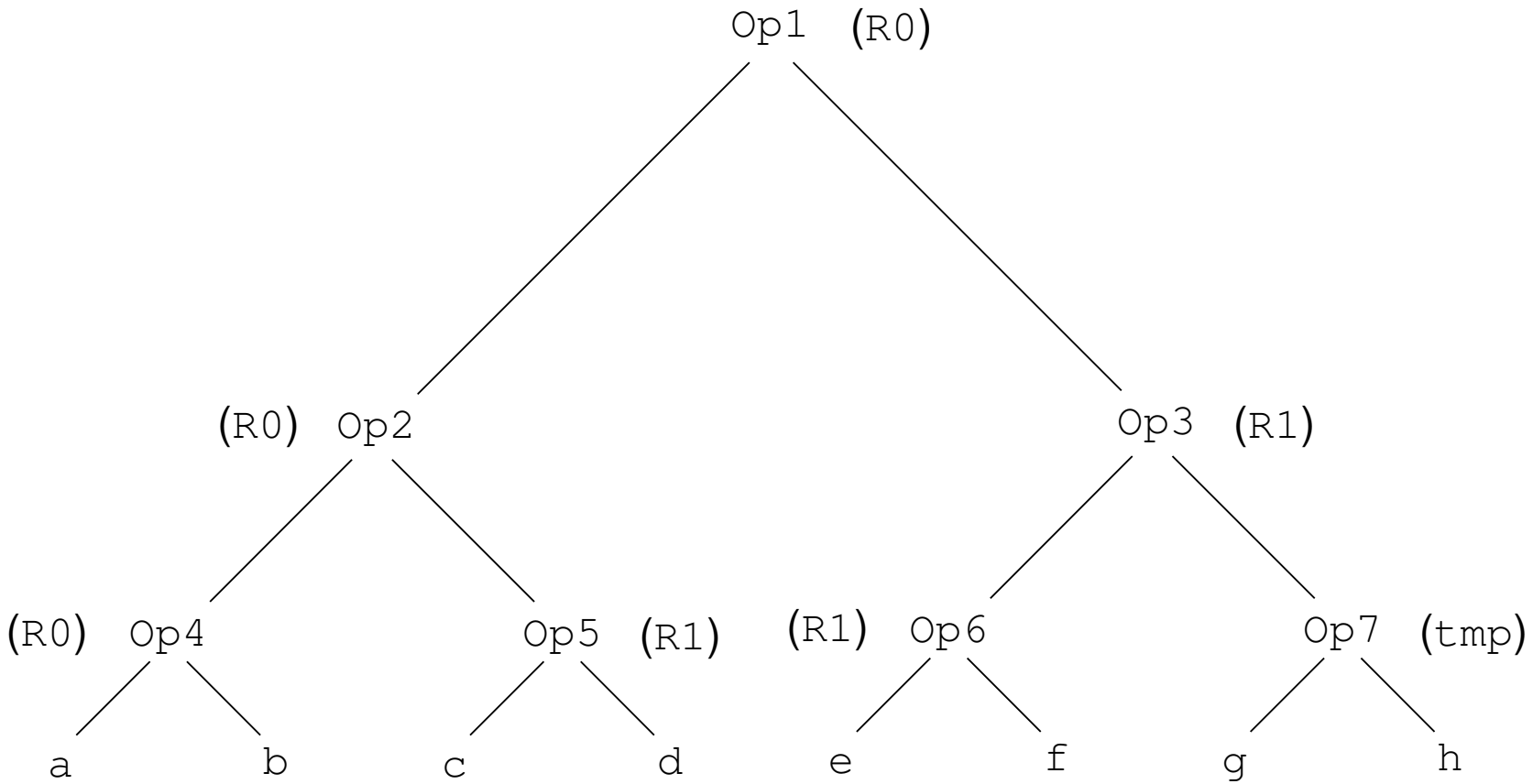
Exemple

```
;; partie gauche
LOAD @a, R0
Op4 @b, R0
LOAD @c, R1
Op5 @d, R1
Op2 R1, R0
;; partie droite
LOAD @e, R1
Op6 @f, R1
LOAD @g, R2
Op7 @h, R2
Op3 R2, R1
;; enfin :
Op1 R1, R0
```



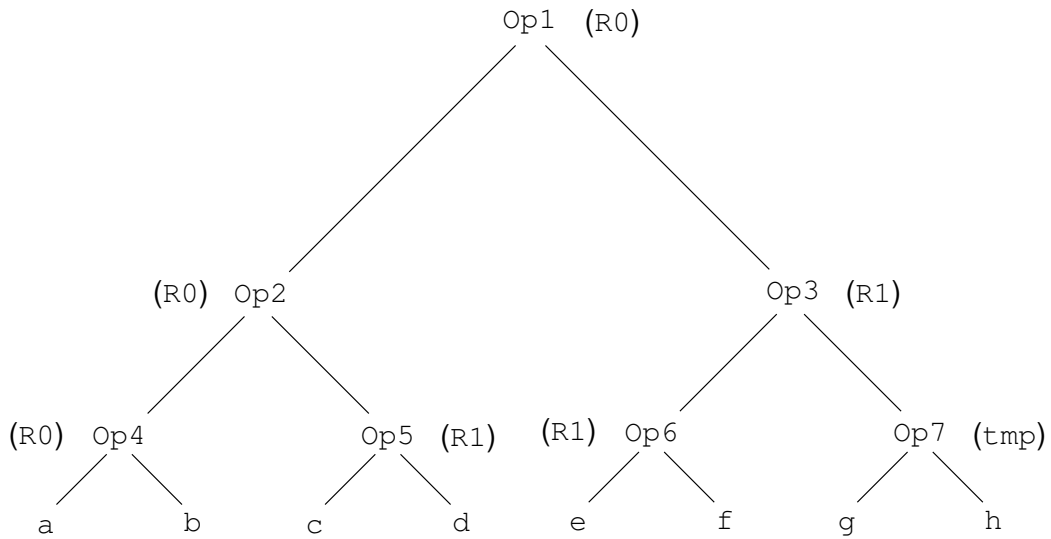
Variables temporaires

Si on n'a que les registres `R0` et `R1`, on doit utiliser une variable temporaire :



Exemple

```
;; partie gauche
LOAD @a, R0
Op4 @b, R0
LOAD @c, R1
Op5 @d, R1
Op2 R1, R0
;; partie droite
LOAD @g, R1
Op7 @h, R1
STORE R1, @tmp
LOAD @e, R1
Op6 @f, R1
Op3 @tmp, R1
;; enfin :
Op1 R1, R0
```



Algorithme

On définit une fonction dont les paramètres sont l'arbre à générer, et le registre dans lequel on veut le résultat :

- `coder(idf, Ri) = generer(LOAD @idf, Ri)`
- `coder(num, Ri) = generer(LOAD #num, Ri)`
- ```
  coder(E binop idf, Ri) =
        coder(E,Ri)
        generer(binop @idf , Ri)
  ```
- ```
  coder(E binop num, Ri) =
        coder(E,Ri)
        generer(binop #num , Ri)
  ```

Algorithme

- `coder(E binop E', Ri) =`

  Si `reste_registre()` alors

  // évaluation gauche-droite

  ```
  coder(E1,Ri)
  Rj = get_registre()
  coder(E2,Rj)
  generer(binop Rj, Ri)
  liberer_registre(Rj)
  ```

  Sinon

  // évaluation droite-gauche

  ```
  coder(E2,Ri)
  generer(PUSH Ri)
  coder(E1,Ri)
  generer(binop 0(SP), Ri)
  generer(SUB #1, SP)
  ```

Calcul du nombre de registres nécessaires

Supposons qu'on veut évaluer l'expression

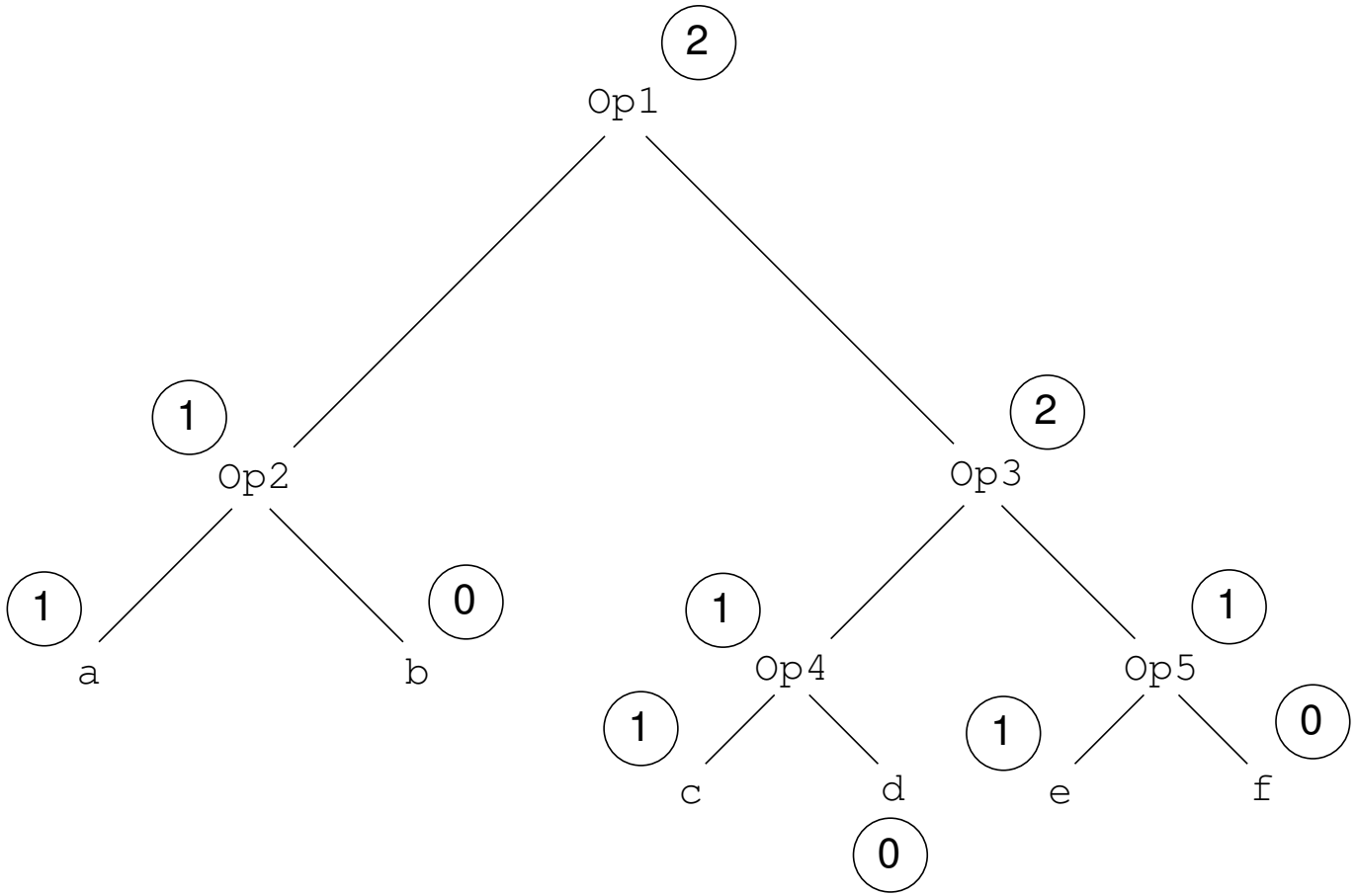
```
binop Op1, Op2
```

Combien faut-il de registres au minimum pour évaluer cette expression ?
Soit n_1 le nombre de registres nécessaires à l'évaluation de `Op1`
Soit n_2 le nombre de registres nécessaires à l'évaluation de `Op2`

- Si on commence par évaluer `Op1`, on a besoin de $max(n_1, n_2 + 1)$
- Si on commence par évaluer `Op2`, on a besoin de $max(n_1 + 1, n_2)$

Il faut donc commencer par évaluer l'expression qui nécessite le plus de registres.

Exemple



- Les feuilles gauches nécessitent 1 registre
- Les feuilles droites ne nécessitent aucun registre

Algorithme en 2 passes

Un meilleur algorithme pour évaluer une expression consiste donc à faire 2 passes sur l'AST :

- 1ère passe : on évalue le nombre de registres dont on a besoin pour évaluer chaque noeud de l'arbre.
  - On décore chaque noeud de l'expression du nombre minimal de registres qu'il nécessite
- 2ème passe : on génère le code des noeuds en commençant par la partie droite ou par la partie gauche, selon le nombre de registres nécessaires à chacunes des deux parties.

Algorithmes d'allocation de registres

Les algorithmes d'allocation de registres essaient d'optimiser l'utilisation des registres :

- minimiser le nombre de registres pour le calcul d'une expression
- essayer de garder dans le même registre une variable tant qu'elle est vivante (**live**).
- ...

Variables vivantes (Live)

Pour allouer les registres de façon efficace, il est bon de garder constamment dans des registres les variables du programme qui vont encore être utilisées (on les qualifie de vivantes).

```
function f() {
  int x = 0;
  int y = 2;
  x = x + y;
  x = x + 5;
}
```

```
STORE #0, @x
STORE #2, @y
LOAD @x, R0
ADD @y, R0
STORE R0, @x
LOAD @x, R0
ADD #5, R0
STORE R0, @x
```

On garde `x` dans un registre :

```
STORE #0, @x
STORE #2, @y
LOAD @x, R0
ADD @y, R0
ADD #5, R0
STORE R0, @x
```

Ce problème est lié au problème de coloriage de graphes