

Université Paris-Sud

Compilation et langages

Thibaut Balabonski
d'après Jean-Christophe Filliâtre @ ENS

Cours 6 / 21 octobre 2016

Les **fonctions** :

- Stratégie d'évaluation
- Mode de passage des paramètres
- Compilation des appels de fonction

Stratégie d'évaluation

Dans la **déclaration** d'une fonction

```
function f(x1, ..., xn) =  
    ...
```

les variables x_1, \dots, x_n sont appelées **paramètres formels** de f

Dans l'**appel** de cette fonction

```
f(e1, ..., en)
```

les expressions e_1, \dots, e_n sont appelées **paramètres effectifs** de f

La stratégie d'évaluation d'un langage spécifie l'ordre dans lequel les calculs sont effectués

On peut la définir à l'aide d'une sémantique formelle (cf cours 4)

Le compilateur se doit de respecter la stratégie d'évaluation

En particulier, la stratégie d'évaluation **peut** spécifier

- à quel moment les paramètres effectifs d'un appel sont évalués
- l'ordre d'évaluation des opérandes et des paramètres effectifs

Certains aspects de l'évaluation peuvent cependant rester **non spécifiés**

Cela laisse alors de la latitude au compilateur, notamment pour effectuer des optimisations (par exemple en ordonnant les calculs comme il le souhaite)

On distingue notamment :

- L'**évaluation stricte** : les opérandes / paramètres effectifs sont évalués avant l'opération / l'appel

Exemples : C, C++, Java, OCaml, le compilateur des TP

- L'**évaluation paresseuse** : les opérandes / paramètres effectifs ne sont évalués que si nécessaire

Exemples : Haskell, Clojure

mais aussi les connectives logiques `&&` et `||` de beaucoup de langages

Un langage impératif adopte systématiquement une évaluation stricte, pour garantir une séquentialité des effets de bord qui coïncide avec le texte source

Par exemple, le programme OCaml

```
let r = ref 0
let id x = r := !r + x; x
let f x y = !r
let () = print_int (f (id 40) (id 2))
```

affiche 42 car les deux arguments de f ont été évalués

La non-terminaison est également un effet

Ainsi, le programme

```
let rec loop () = loop ()  
let f x y = x + 1  
let v = f 41 (loop ())
```

ne termine pas, bien que l'argument `y` n'est pas utilisé

Un langage purement applicatif, en revanche, peut adopter la stratégie d'évaluation de son choix, car une expression aura toujours la même valeur (on parle de **transparence référentielle**)

En particulier, il peut faire le choix d'une évaluation paresseuse

Le programme Haskell

```
loop () = loop ()  
f x y = x  
main = putChar (f 'a' (loop ()))
```

termine (après avoir affiché a)

Modes de passage des paramètres

Dans un langage comprenant des modifications en place, une affectation

```
e1 := e2
```

modifie un emplacement mémoire désigné par l'expression e1

L'expression e1 est limitée à certaines constructions,
car des affectations comme

```
42 := 17  
true := false
```

n'ont en général pas de sens

On parle de **valeur gauche** pour désigner les expressions légales à gauche d'une affectation

La sémantique précise également le **mode de passage** des paramètres lors d'un appel

On distingue notamment

- l'**appel par valeur** (*call by value*)
- l'**appel par référence** (*call by reference*)
- l'**appel par nom** (*call by name*)
- l'**appel par nécessité** (*call by need*)

(on parle aussi parfois de **passage** par valeur, par référence, etc.)

De **nouvelles** variables représentant les paramètres formels reçoivent les **valeurs** des paramètres effectifs

```
function f(x) =  
  x := x + 1  
  
main() =  
  int v := 41  
  f(v)  
  print(v)    // affiche 41
```

Les paramètres formels désignent les **mêmes** valeurs gauches que les paramètres effectifs

```
function f(x) =  
  x := x + 1  
  
main() =  
  int v := 41  
  f(v)  
  print(v)    // affiche 42
```


Les paramètres effectifs sont **substitués** aux paramètres effectifs, textuellement, et donc évalués seulement si nécessaire

```
function f(x, y, z) =  
    return x*x + y*y  
  
main() =  
    print(f(1+2, 2+2, 1/0)) // affiche 25  
    // 1+2 est évalué deux fois  
    // 2+2 est évalué deux fois  
    // 1/0 n'est jamais évalué
```

Les paramètres effectifs ne sont évalués que si nécessaire,
mais **au plus une fois**

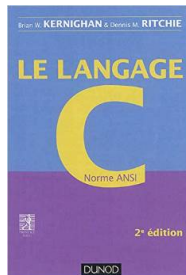
```
function f(x, y, z) =  
    return x*x + y*y  
  
main() =  
    print(f(1+2, 2+2, 1/0)) // affiche 25  
    // 1+2 est évalué une fois  
    // 2+2 est évalué une fois  
    // 1/0 n'est jamais évalué
```

Quelques mots sur le langage C

Le langage C est un langage impératif relativement bas niveau, notamment parce que la notion de pointeur, et d'arithmétique de pointeur, y est explicite

On peut le considérer inversement comme un assembleur de haut niveau

Un ouvrage toujours d'actualité :
Le langage C
de Brian Kernighan et Dennis Ritchie



Le langage C est muni d'une stratégie d'évaluation stricte,
avec appel par valeur

L'ordre d'évaluation n'est pas spécifié

- On trouve des types de base tels que `char`, `int`, `float`, etc. (mais pas de booléens)

- Un type τ^* des pointeurs vers des valeurs de type τ

Si p est un pointeur de type τ^* , alors $*p$ désigne la valeur pointée par p , de type τ

Si e est une valeur gauche de type τ , alors $\&e$ est un pointeur sur l'emplacement mémoire correspondant, de type τ^*

- Des enregistrements, appelés *structures*, tels que

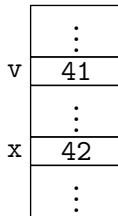
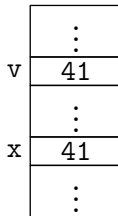
```
struct L { int head; struct L *next; };
```

Si e a le type `struct L`, on note `e.head` l'accès au champ

En C, une valeur gauche est de la forme

- x , une variable
- $*e$, le déréférencement d'un pointeur
- $e.x$, l'accès à un champ de structure,
si e est elle-même une valeur gauche
- $t[e]$, qui n'est autre que $*(t+e)$
- $e \rightarrow x$, qui n'est autre que $(*e).x$

```
void f(int x) {  
    x = x+1;  
}  
  
int main() {  
    int v = 41;  
    f(v);  
    // v vaut toujours 41  
}
```



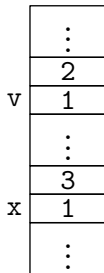
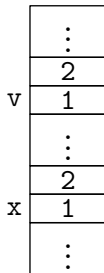
L'appel par valeur implique que les structures sont **copiées** lorsqu'elles sont passées en paramètres ou renvoyées

Les structures sont également copiées lors des affectations de structures, *i.e.* des affectations de la forme $x = y$, où x et y ont le type `struct S`

```
struct S { int a; int b; };
```

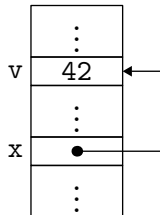
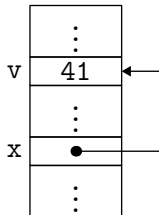
```
void f(struct S x) {  
    x.b = x.b + 1;  
}
```

```
int main() {  
    struct S v = { 1, 2 };  
    f(v);  
    // v.b vaut toujours 2  
}
```



On peut **simuler** un passage par référence en passant un pointeur explicite

```
void incr(int *x) {  
    *x = *x + 1;  
}  
  
int main() {  
    int v = 41;  
    incr(&v);  
    // v vaut maintenant 42  
}
```



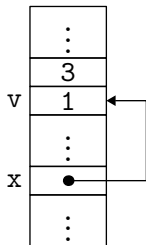
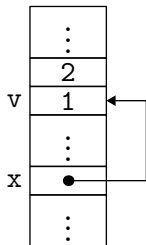
mais ce n'est qu'un passage de pointeur **par valeur**

Pour éviter le coût des copies, on passe des pointeurs sur les structures le plus souvent

```
struct S { int a; int b; };
```

```
void f(struct S *x) {  
    x->b = x->b + 1;  
}
```

```
int main() {  
    struct S v = { 1, 2 };  
    f(&v);  
    // v.b vaut maintenant 3  
}
```



La manipulation explicite de pointeurs peut être dangereuse

Considérons le programme

```
int* p() {  
    int x;  
    return &x;  
}
```

Il renvoie un pointeur qui correspond à un emplacement sur la pile qui vient justement de disparaître (à savoir le tableau d'activation de `p`), et qui sera très probablement réutilisé rapidement par un autre tableau d'activation

On parle de référence fantôme (*dangling reference*)

On peut déclarer un tableau ainsi :

```
int t[10];
```

La notation $t[i]$ n'est que du sucre syntaxique pour $*(t+i)$ où

- t désigne un pointeur sur le début d'une zone contenant 10 entiers
- $+$ désigne une opération d'*arithmétique de pointeur* (qui consiste à ajouter à t la quantité $4i$ pour un tableau d'entiers 32 bits)

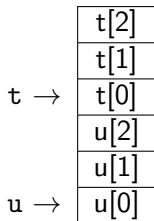
Le premier élément du tableau est donc $t[0]$ c'est-à-dire $*t$

Quand on passe un tableau en paramètre, on ne fait que passer le pointeur (par valeur, toujours)

On ne peut affecter des tableaux, seulement des pointeurs

Ainsi, on ne peut pas écrire

```
void p() {  
    int t[3];  
    int u[3];  
    t = u;  
}
```



car `t` et `u` sont des tableaux (alloués sur la pile) et l'affectation de tableaux n'est pas autorisée

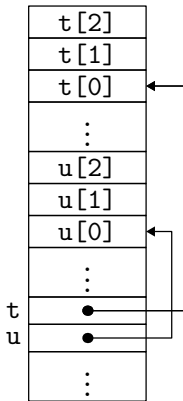
En revanche on peut écrire

```
void q(int t[3], int u[3]) {  
    t = u;  
}
```

car c'est exactement la même chose que

```
void q(int *t, int *u) {  
    t = u;  
}
```

et l'affectation de pointeurs est autorisée



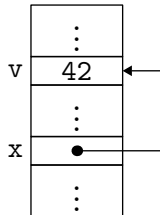
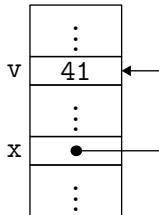
Quelques mots sur le langage C++

En C++, on trouve (entre autres) les types et constructions du C, avec une stratégie d'évaluation stricte

Le mode de passage est par valeur par défaut

mais on trouve aussi un **passage par référence** indiqué par le symbole & au niveau de l'argument formel

```
void f(int &x) {  
    x = x + 1;  
}  
  
int main() {  
    int v = 41;  
    f(v);  
    // v vaut maintenant 42  
}
```



En particulier, c'est le compilateur qui

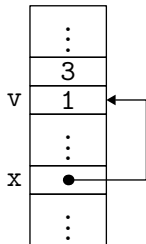
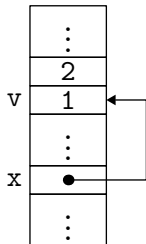
- a pris l'adresse de `v` au moment de l'appel
- a déréférencé l'adresse `x` dans la fonction `f`

On peut passer une structure par référence

```
struct S { int a; int b; };

void f(struct S &x) {
    x.b = x.b + 1;
}

int main() {
    struct S v = { 1, 2 };
    f(v);
    // v.b vaut maintenant 3
}
```



Quelques mots sur le langage OCaml

OCaml est muni d'une stratégie d'évaluation stricte, avec appel par valeur

L'ordre d'évaluation n'est pas spécifié

Une valeur est

- soit d'un type primitif (booléen, caractère, entier machine, etc.)
- soit un pointeur vers un bloc mémoire (tableau, enregistrement, constructeur non constant, etc.) alloué sur le tas en général

Les valeurs gauches sont les éléments de tableaux

```
a.(2) <- true
```

Et les champs mutables d'enregistrements

```
x.age <- 42
```

Rappel : une référence est un enregistrement

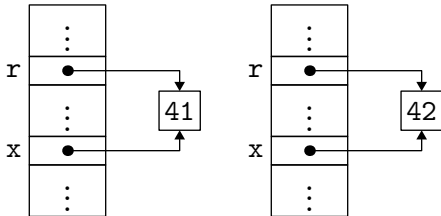
```
type 'a ref = { mutable contents: 'a }
```

et les opérations := et ! sont définies par

```
let (!)  r    = r.contents
let (:=) r v = r.contents <- v
```



```
let f x =  
  x := !x + 1  
  
let main () =  
  let r = ref 41 in  
  f r  
  (* !r vaut maintenant 42 *)
```



C'est toujours un passage par valeur,
d'une valeur qui est un pointeur (vers une valeur mutable)

On peut **simuler l'appel par nom** en OCaml, en remplaçant les arguments par des fonctions

Ainsi, la fonction

```
let f x y =  
  if x = 0 then 42 else y + 1
```

peut être réécrite en

```
let f x y =  
  if x () = 0 then 42 else y () + 1
```

et appelée comme ceci

```
let v = f (fun () -> 0) (fun () -> failwith "oops")
```

Plus subtilement, on peut aussi **simuler l'appel par nécessité** en OCaml

On commence par introduire un type pour représenter les calculs paresseux

```
type 'a value = Value of 'a  
              | Frozen of (unit -> 'a)
```

```
type 'a by_need = 'a value ref
```

et une fonction qui évalue un tel calcul si ce n'est pas déjà fait

```
let force l = match !l with  
  | Value v -> v  
  | Frozen f -> let v = f () in l := Value v; v
```

On définit alors la fonction `f` comme ceci

```
let f x y =  
  if force x = 0 then 42 else force y + 1
```

et on l'utilise ainsi

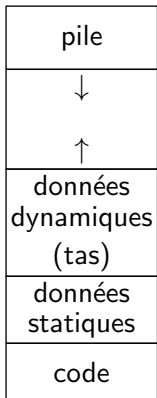
```
let v = f (ref (Frozen (fun () -> 0)))  
          (ref (Frozen (fun () -> failwith "oups")))
```

Note : la construction `lazy` d'OCaml fait quelque chose de semblable (un peu plus subtilement)

Compilation des appels de fonction

Constat : les appels de fonctions peuvent être arbitrairement imbriqués
⇒ les registres ne peuvent suffire pour les paramètres / variables locales
⇒ il faut allouer de la mémoire pour cela

Les fonctions procèdent selon un mode *last-in first-out*, c'est-à-dire de **pile**



La **pile** est stockée tout en haut, et croît dans le sens des adresses décroissantes ; `$sp` pointe sur le sommet de la pile

Les données dynamiques (survivant aux appels de fonctions) sont allouées sur le **tas** (éventuellement par un GC), en bas de la zone de données, juste au dessus des données statiques

Ainsi, on ne se marche pas sur les pieds

Lorsqu'une fonction f (l'appelant ou *caller*) souhaite appeler une fonction g (l'appelé ou *callee*), elle exécute

```
jal g
```

et lorsque l'appelé en a terminé, il lui rend le contrôle avec

```
jr $ra
```

Problème :

- si g appelle elle-même une fonction, $\$ra$ sera écrasé
- de même, tout registre utilisé par g sera perdu pour f

Il existe de multiples manières de s'en sortir,
mais en général on s'accorde sur des **conventions d'appel**

Utilisation des registres

- `$at`, `$k0` et `$k1` sont réservés à l'assembleur et l'OS
- `$a0`–`$a3` sont utilisés pour passer les quatre premiers arguments (les autres sont passés sur la pile) et `$v0`–`$v1` pour renvoyer le résultat
- `$t0`–`$t9` sont des registres **caller-saved** i.e. l'appelant doit les sauvegarder si besoin ; on y met donc typiquement des données qui n'ont pas besoin de survivre aux appels
- `$s0`–`$s7` sont des registres **callee-saved** i.e. l'appelé doit les sauvegarder ; on y met donc des données de durée de vie longue, ayant besoin de survivre aux appels
- `$sp` est le pointeur de pile, `$fp` le pointeur de *frame*
- `$ra` contient l'adresse de retour
- `$gp` pointe au milieu de la zone de données statiques (10008000_{16})

Il y a quatre temps dans un appel de fonction

1. pour l'appelant, juste avant l'appel
2. pour l'appelé, au début de l'appel
3. pour l'appelé, à la fin de l'appel
4. pour l'appelant, juste après l'appel

Ils s'organisent autour d'un segment situé au sommet de la pile appelé le **tableau d'activation**, en anglais **stack frame**, situé entre \$fp et \$sp

1. Passe les arguments dans \$a0–\$a3, les autres sur la pile s'il y en a plus de 4
2. Sauvegarde les registres \$t0–\$t9 qu'il compte utiliser après l'appel (dans son propre tableau d'activation)
3. Exécute

```
jal    appelé
```

L'appelé, au début de l'appel

1. Alloue son tableau d'activation, par exemple

```
addi $sp, $sp, -28
```

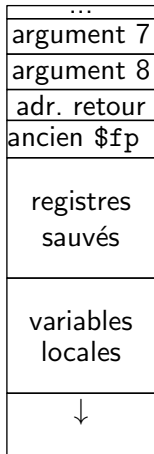
\$fp →

2. Sauvegarde \$fp puis le positionne, par exemple

```
sw    $fp, 24($sp)
addi  $fp, $sp, 24
```

3. Sauvegarde \$s0-\$s7 et \$ra si besoin

\$sp →



\$fp permet d'atteindre facilement les arguments et variables locales, avec un décalage fixe quel que soit l'état de la pile

1. Place le résultat dans \$v0 (voire \$v1)
2. Restaure les registres sauvegardés, dont \$fp, par exemple

```
lw    $fp, 24($sp)
```

3. Dépile son tableau d'activation, par exemple

```
addi $sp, $sp, 28
```

4. Exécute

```
jr    $ra
```

1. Dépile les éventuels arguments 5, 6, ...
2. Restaure les registres caller-saved `$t0–$t9`, si besoin

La sémantique d'un langage décrit

- la manière dont doivent être compris les paramètres d'une fonction,
- et éventuellement l'ordre d'évaluation.

Les appels de fonctions s'articulent autour

- d'une notion de tableau d'activation alloué dans la pile,
- de conventions d'appel.

```
t(a,b,c){int d=0,e=a&~b&~c,f=1;if(a)
for(f=0;d=(e-=d)&-e;f+=t(a-d,(b+d)*2,
(c+d)/2));return f;}main(q){scanf("%d",
&q);printf("%d\n",t(~(~0<<q),0,0));}
```



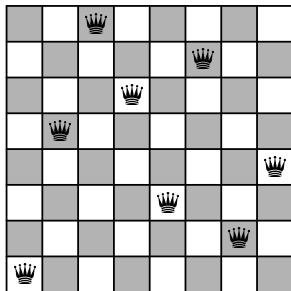
```
int t(int a, int b, int c) {  
    int d=0, e=a&~b&~c, f=1;  
    if (a)  
        for (f=0; d=(e-=d)&-e; f+=t(a-d, (b+d)*2, (c+d)/2));  
    return f;  
}  
  
int main() {  
    int q;  
    scanf("%d", &q);  
    printf("%d\n", t(~(~0<<q), 0, 0));  
}
```

Clarification (suite)

```
int t(int a, int b, int c) {
    int f=1;
    if (a) {
        int d, e=a&~b&~c;
        f = 0;
        while (d=e&-e) {
            f += t(a-d, (b+d)*2, (c+d)/2);
            e -= d;
        }
    }
    return f;
}

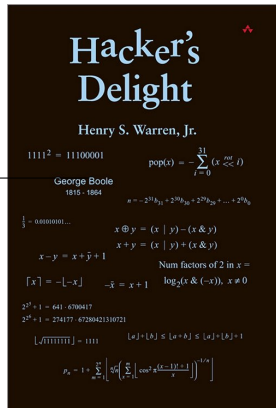
int main() {
    int q;
    scanf("%d", &q);
    printf("%d\n", t(~(~0<<q), 0, 0));
}
```

Ce programme calcule
le nombre de solutions
du problème dit
des n reines



- recherche par force brute (*backtracking*)
- entiers utilisés comme des ensembles :
par ex. $13 = 0 \dots 01101_2 = \{0, 2, 3\}$

| entiers | ensembles |
|--------------------|--|
| 0 | \emptyset |
| $a \& b$ | $a \cap b$ |
| $a + b$ | $a \cup b$, quand $a \cap b = \emptyset$ |
| $a - b$ | $a \setminus b$, quand $b \subseteq a$ |
| $\sim a$ | $\complement a$ |
| $a \& -a$ | $\{ \min(a) \}$, quand $a \neq \emptyset$ |
| $\sim(\sim 0 < n)$ | $\{0, 1, \dots, n-1\}$ |
| $a * 2$ | $\{i+1 \mid i \in a\}$, noté $S(a)$ |
| $a / 2$ | $\{i-1 \mid i \in a \wedge i \neq 0\}$, noté $P(a)$ |



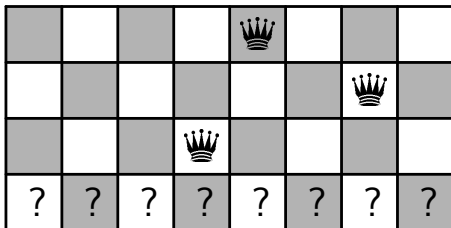
En complément à deux : $-a = \sim a + 1$

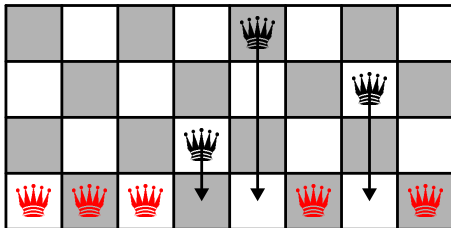
$$\begin{aligned}
 a &= b_{n-1}b_{n-2} \dots b_k 10 \dots 0 \\
 \sim a &= \overline{b_{n-1}b_{n-2} \dots b_k} 01 \dots 1 \\
 -a &= \overline{b_{n-1}b_{n-2} \dots b_k} 10 \dots 0 \\
 a \& -a &= 0 \quad 0 \dots 010 \dots 0
 \end{aligned}$$

Exemple :

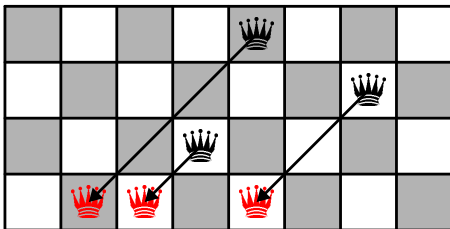
$$\begin{aligned}
 a &= 00001100 = 12 \\
 -a &= 11110100 = -128 + 116 \\
 a \& -a &= 00000100
 \end{aligned}$$

```
int  $t(a, b, c)$   
   $f \leftarrow 1$   
  if  $a \neq \emptyset$   
     $e \leftarrow (a \setminus b) \setminus c$   
     $f \leftarrow 0$   
    while  $e \neq \emptyset$   
       $d \leftarrow \min(e)$   
       $f \leftarrow f + t(a \setminus \{d\}, S(b \cup \{d\}), P(c \cup \{d\}))$   
       $e \leftarrow e \setminus \{d\}$   
  return  $f$   
  
int  $queens(n)$   
  return  $t(\{0, 1, \dots, n-1\}, \emptyset, \emptyset)$ 
```

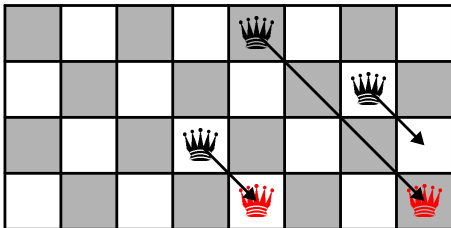




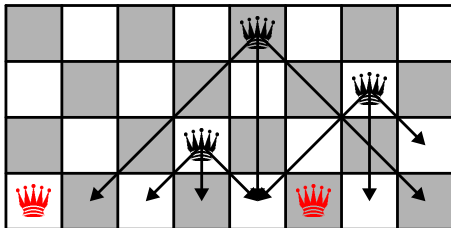
$a = \text{colonnes à remplir} = 11100101_2$



b = positions interdites à cause de diagonales vers la gauche = 01101000_2



c = positions interdites à cause de diagonales vers la droite = 00001001_2



$a \& \sim b \& \sim c$ = positions à essayer = 10000100_2

Intérêt de ce programme pour la compilation

```
int t(int a, int b, int c) {
    int f=1;
    if (a) {
        int d, e=a&~b&~c;
        f = 0;
        while (d=e&-e) {
            f += t(a-d, (b+d)*2, (c+d)/2);
            e -= d;
        }
    }
    return f;
}

int main() {
    int q;
    scanf("%d", &q);
    printf("%d\n", t(~(~0<<q), 0, 0));
}
```

Court, mais contient

- un test (**if**)
- une boucle (**while**)
- une fonction récursive
- quelques calculs

C'est aussi une
excellente solution
au problème des n reines