

École Normale Supérieure

Langages de programmation et compilation

Jean-Christophe Filliâtre

- **cours** le vendredi, 8h30–10h30 en salle E. Noether
 - photocopié début janvier
- **TD** en salle info 3 (NIR)
 - groupe 1 le vendredi 10h45–12h15
 - groupe 2 le jeudi 10h45–12h15
 - avec Jérôme Boillot (jerome.boillot@ens.fr)

toutes les infos sur le site web du cours (accessible depuis Moodle)

<http://www.lri.fr/~filliatr/ens/compil/>

questions ⇒ Jean-Christophe.Filliatre@cnrs.fr

- un **examen**, en janvier
 - anciens sujets+corrigés sur le site
- un **projet** = un mini compilateur
 - réalisé en dehors des TD, seul ou en binôme
 - rendu en deux fois (fin novembre, début janvier)

$$note\ finale = \frac{examen + projet}{2}$$

maîtriser les mécanismes de la **compilation**,
c'est-à-dire de la transformation d'un langage dans un autre

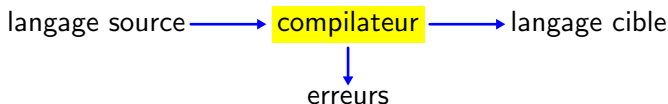
comprendre les différents aspects des **langages de programmation**
par le biais de la compilation

ici on programme

- en cours
- en TD
- pour réaliser le projet
- à l'examen

on programme en **OCaml**

schématiquement, un compilateur est un programme qui traduit un « programme » d'un langage **source** vers un langage **cible**, en signalant d'éventuelles erreurs



compilation vers le langage machine

quand on parle de compilation, on pense typiquement à la traduction d'un langage de haut niveau (C, Java, OCaml, ...) vers le langage machine d'un processeur

```
% gcc -o sum sum.c
```

source `sum.c` → **compilateur C (gcc)** → exécutable `sum`

```
int main(int argc, char **argv) {  
    int i, s = 0;  
    for (i = 0; i <= 100; i++) s += i*i;  
    printf("0*0+...+100*100 = %d\n", s);  
}
```

```
0010011111011111011111111111111100000  
10101111110111111100000000000010100  
1010111111010010000000000000100000  
1010111111010010100000000000100100  
101011111101000000000000000011000  
101011111101000000000000000011100  
100011111101011100000000000011100  
...
```

dans ce cours, nous allons effectivement nous intéresser à la compilation vers de **l'assembleur**, mais ce n'est qu'un aspect de la compilation

un certain nombre de techniques mises en œuvre dans la compilation ne sont pas liées à la production de code assembleur

certains langages sont d'ailleurs

- interprétés (BASIC, COBOL, Ruby, Python, etc.)
- compilés dans un langage intermédiaire qui est ensuite interprété (Java, OCaml, Scala, etc.)
- compilés à la volée (Julia, etc.)
- compilés vers un autre langage de haut niveau

différence entre compilateur et interprète

un **compilateur** traduit un programme P en un programme Q tel que pour toute entrée x , la sortie de $Q(x)$ soit la même que celle de $P(x)$

$$\forall P \exists Q \forall x \dots$$

un **interprète** est un programme qui, étant donné un programme P et une entrée x , calcule la sortie s de $P(x)$

$$\forall P \forall x \exists s \dots$$

dit autrement,

le compilateur fait un travail complexe **une seule fois**, pour produire un code fonctionnant pour n'importe quelle entrée

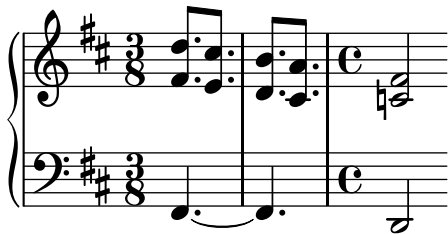
l'interprète effectue un travail plus simple, mais le refait sur chaque entrée

autre différence : le code compilé est généralement bien plus efficace que le code interprété

exemple de compilation et d'interprétation

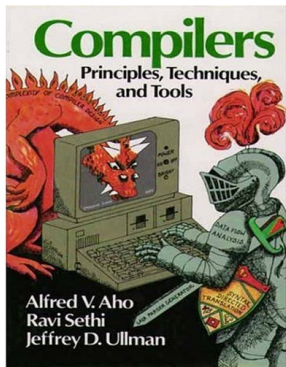
source → **lilypond** → fichier PDF → **evince** → image

```
\new PianoStaff <<  
  \new Staff { \clef "treble" \key d \major \time 3/8  
    <<d8. fis,8.>> <<cis'8. e,8.>> | ... }  
  \new Staff { \clef "bass" \key d \major  
    fis,,4. ~ | fis4. | \time 4/4 d2 }  
>>
```



à quoi juge-t-on la qualité d'un compilateur ?

- à sa correction
- à l'efficacité du code qu'il produit
- à sa propre efficacité



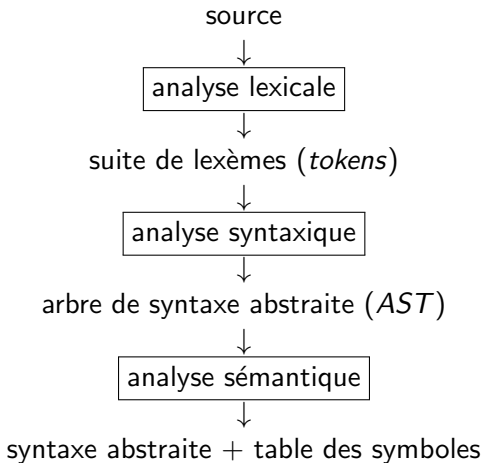
"Optimizing compilers are so difficult to get right that we dare say that no optimizing compiler is completely error-free! Thus, the most important objective in writing a compiler is that it is correct."

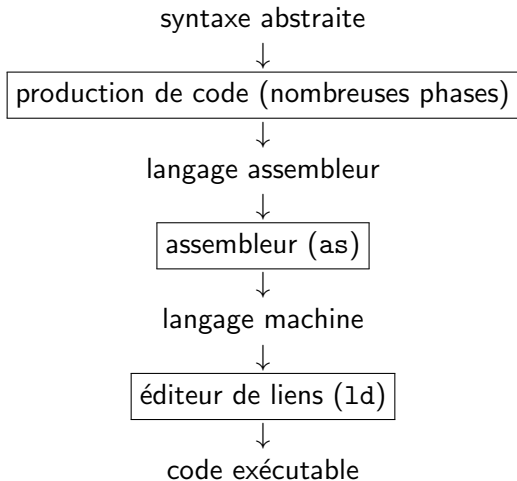
(Dragon Book, 2006)

typiquement, le travail d'un compilateur se compose

- d'une phase d'**analyse**
 - reconnaît le programme à traduire et sa signification
 - signale les erreurs et peut donc échouer (erreurs de syntaxe, de portée, de typage, etc.)

- puis d'une phase de **synthèse**
 - production du langage cible
 - utilise de nombreux langages intermédiaires
 - n'échoue pas

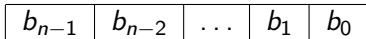




aujourd'hui

assembleur

un entier est représenté par n bits,
conventionnellement numérotés de droite à gauche



typiquement, n vaut 8, 16, 32, ou 64

les bits b_{n-1} , b_{n-2} , etc. sont dits de **poids fort**
les bits b_0 , b_1 , etc. sont dits de **poids faible**

$$\text{bits} = b_{n-1}b_{n-2} \dots b_1b_0$$

$$\text{valeur} = \sum_{i=0}^{n-1} b_i 2^i$$

bits	valeur
000...000	0
000...001	1
000...010	2
⋮	⋮
111...110	$2^n - 2$
111...111	$2^n - 1$

exemple : $00101010_2 = 42$

le bit de poids fort b_{n-1} est le **bit de signe**

$$\text{bits} = b_{n-1}b_{n-2}\dots b_1b_0$$

$$\text{valeur} = -b_{n-1}2^{n-1} + \sum_{i=0}^{n-2} b_i2^i$$

exemple :

$$\begin{aligned} 11010110_2 &= -128 + 86 \\ &= -42 \end{aligned}$$

bits	valeur
100...000	-2^{n-1}
100...001	$-2^{n-1} + 1$
⋮	⋮
111...110	-2
111...111	-1
000...000	0
000...001	1
000...010	2
⋮	⋮
011...110	$2^{n-1} - 2$
011...111	$2^{n-1} - 1$

selon le contexte, on interprète ou non le bit b_{n-1} comme un bit de signe

exemple :

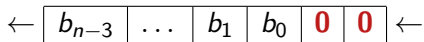
- $11010110_2 = -42$ (8 bits signés)
- $11010110_2 = 214$ (8 bits non signés)

la machine fournit des opérations

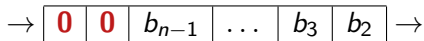
- opérations logiques, encore appelées bit à bit (AND, OR, XOR, NOT)
- de décalage
- arithmétiques (addition, soustraction, multiplication, etc.)

opération		exemple
négation	x	00101001
	NOT x	<u>11010110</u>
ET	x	00101001
	y	01101100
	x AND y	<u>00101000</u>
OU	x	00101001
	y	01101100
	x OR y	<u>01101101</u>
OU exclusif	x	00101001
	y	01101100
	x XOR y	<u>01000101</u>

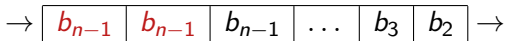
- décalage logique à gauche (insère des 0 de poids faible)



- décalage logique à droite (insère des 0 de poids fort)

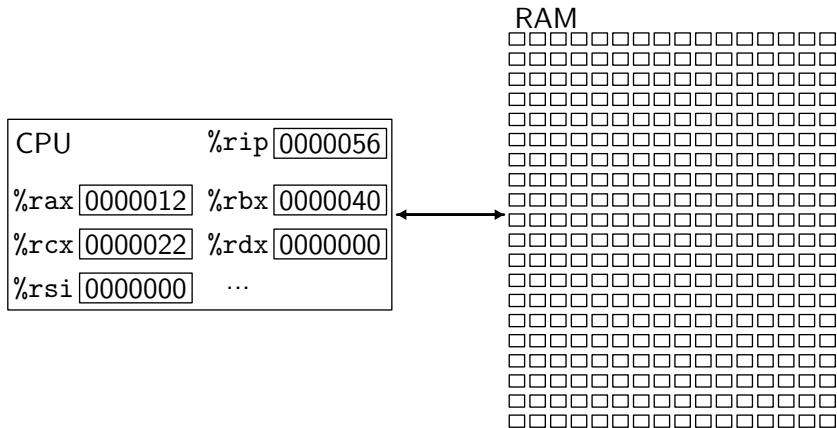


- décalage arithmétique à droite (réplique le bit de signe)



très schématiquement, un ordinateur est composé

- d'une unité de calcul (CPU), contenant
 - un petit nombre de registres entiers ou flottants
 - des capacités de calcul
- d'une mémoire vive (RAM)
 - composée d'un très grand nombre d'octets (8 bits)
par exemple, 1 Gio = 2^{30} octets = 2^{33} bits, soit $2^{2^{33}}$ états possibles
 - contient des données et des instructions



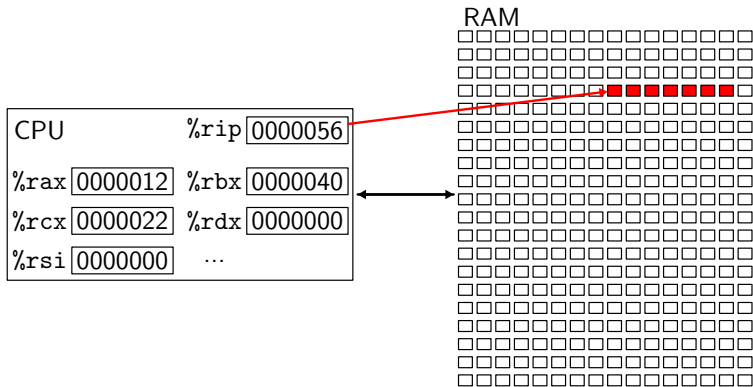
l'accès à la mémoire coûte cher (à un milliard d'instructions par seconde, la lumière ne parcourt que 30 centimètres entre deux instructions !)

la réalité est bien plus complexe

- plusieurs (co)processeurs, dont certains dédiés aux flottants
- une ou plusieurs mémoires cache
- une virtualisation de la mémoire (MMU)
- etc.

schématiquement, l'exécution d'un programme se déroule ainsi

- un registre (`%rip`) contient l'adresse de l'instruction à exécuter
- on lit un ou plusieurs octets à cette adresse (*fetch*)
- on interprète ces bits comme une instruction (*decode*)
- on exécute l'instruction (*execute*)
- on modifie le registre `%rip` pour passer à l'instruction suivante (typiquement celle se trouvant juste après, sauf en cas de saut)



instruction :

48	c7	c0	2a	00	00	00
----	----	----	----	----	----	----

décodage :

	}	}	}			
	movq	%rax	42			

i.e. mettre 42 dans le registre %rax

là encore la réalité est bien plus complexe

- pipelines
 - plusieurs instructions sont exécutées en parallèle
- prédiction de branchement
 - pour optimiser le pipeline, on tente de prédire les sauts conditionnels

deux grandes familles de microprocesseurs

- CISC (*Complex Instruction Set*)
 - beaucoup d'instructions
 - beaucoup de modes d'adressage
 - beaucoup d'instructions lisent / écrivent en mémoire
 - peu de registres
 - exemples : VAX, PDP-11, Motorola 68xxx, AMD/Intel x86
- RISC (*Reduced Instruction Set*)
 - peu d'instructions, régulières
 - très peu d'instructions lisent / écrivent en mémoire
 - beaucoup de registres, uniformes
 - exemples : Alpha, Sparc, MIPS, ARM

on choisit **x86-64** pour ce cours (les TD et le projet)

l'architecture x86-64

x86 une famille d'architectures compatibles

1974 Intel 8080 (8 bits)

1978 Intel 8086 (16 bits)

1985 Intel 80386 (32 bits)

x86-64 une extension 64-bits

2000 introduite par AMD

2004 adoptée par Intel

- 64 bits
 - opérations arithmétiques, logique et de transfert sur 64 bits
- 16 registres
 - `%rax, %rbx, %rcx, %rdx, %rbp, %rsp, %rsi, %rdi, %r8, %r9, %r10, %r11, %r12, %r13, %r14, %r15`
- adressage de la mémoire sur 48 bits au moins (≥ 256 To)
- nombreux modes d'adressage

on ne programme pas en langage machine mais en assembleur

l'assembleur fourni un certain nombre de facilités :

- étiquettes symboliques
- allocation de données globales

le langage assembleur est transformé en langage machine par un programme appelé également **assembleur** (c'est un compilateur)

on utilise ici Linux et des outils GNU

en particulier, on utilise l'assembleur GNU, avec la **syntaxe AT&T**

sous d'autres systèmes, les outils peuvent être différents

en particulier, l'assembleur peut utiliser la **syntaxe Intel**, différente

```
.text                # des instructions suivent
.globl main          # rend main visible pour ld
main:
pushq    %rbp
movq     %rsp, %rbp
movq     $message, %rdi    # argument de puts
call     puts
movq     $0, %rax          # code de retour 0
popq     %rbp
ret

.data                # des données suivent
message:
.string  "hello, world!"  # terminée par 0
```

assemblage

```
> as hello.s -o hello.o
```

édition de liens (gcc appelle ld)

```
> gcc -no-pie hello.o -o hello
```

exécution

```
> ./hello  
hello, world!
```

on peut **désassembler** avec l'outil objdump

```
> objdump -d hello.o
0000000000000000 <main>:
   0: 55                push   %rbp
   1: 48 89 e5          mov    %rsp,%rbp
   4: 48 c7 c7 00 00 00 00 mov    $0x0,%rdi
   b: e8 00 00 00 00    call   10 <main+0x10>
  10: 48 c7 c0 00 00 00 00 mov    $0x0,%rax
  17: 5d                pop    %rbp
  18: c3                ret
```

on note

- que les adresses de la chaîne et de puts ne sont pas encore connues
- que le programme commence à l'adresse 0

on peut aussi désassembler l'exécutable

```
> objdump -d hello
0000000000401126 <main>:
 401126: 55                push   %rbp
 401127: 48 89 e5          mov    %rsp,%rbp
 40112a: 48 c7 c7 30 40 40 00 mov    $0x404030,%rdi
 401131: e8 fa fe ff ff   call   401030 <puts@plt>
 401136: 48 c7 c0 00 00 00 00 mov    $0x0,%rax
 40113d: 5d                pop    %rbp
 40113e: c3                ret
```

on observe maintenant

- une adresse effective pour la chaîne (\$0x404030)
- une adresse effective pour la fonction puts (\$0x401030)
- que le programme commence à l'adresse \$0x401126

on observe aussi que les octets de l'entier 0x00404030 sont rangés en mémoire dans l'ordre 30, 40, 40, 00

on dit que la machine est **petit-boutiste** (en anglais **little-endian**)

d'autres architectures sont au contraire **gros-boutistes** (**big-endian**) ou encore **biboutistes** (**bi-endian**)

(référence : *Les voyages de Gulliver* de Jonathan Swift)

une exécution pas à pas est possible avec gdb (*the GNU debugger*)

```
> gcc -g -no-pie hello.s -o hello
> gdb hello
GNU gdb (GDB) 7.1-ubuntu
...
(gdb) break main
Breakpoint 1 at 0x401126: file hello.s, line 4.
(gdb) run
Starting program: ../hello

Breakpoint 1, main () at hello.s:4
4          pushq   %rbp
(gdb) step
5          movq   %rsp, %rbp
(gdb) info registers
...
```

on peut aussi utiliser Nemiver (installé en salles infos)

> `nemiver hello`

The screenshot shows the Nemiver debugger window with the following components:

- File Edit View Debug Help** menu bar.
- Control buttons: Continue, Run or Restart, Stop.
- hws** window showing assembly code:


```

1 .text
2 .globl main
3 main:
4 pushq %rbp
5 movq %rsp, %rbp
6 movq $message, %rdi
7 call puts
8 movq $0, %rax # return
9 popq %rbp
10 ret
11 .data
12 message:
13 .string "hello, world"
14
```
- Registers** table:

ID	Name	Value
0	rax	0x401126
1	rbx	0x0
2	rcx	0x403e18
3	rdx	0x7ffffffe238
4	rsi	0x7ffffffe228
5	rdi	0x1
6	rbp	0x7ffffffe110
7	rsp	0x7ffffffe110
8	r8	0x7fff7bf2f10
9	r9	0x7fff7fc9040
10	r10	0x7fff7fc3908
11	r11	0x7fff7fd6680
12	r12	0x7ffffffe228
13	r13	0x401126
14	r14	0x403e18
15	r15	0x7fff7ff040
- Line: 6, Column: 1
- Variable table with columns: Variable, Value, Type.
- Context, Breakpoints, Expression Monitor tabs.

jeu d'instructions

63	31	15	8	7	0
%rax	%eax	%ax	%ah	%al	
%rbx	%ebx	%bx	%bh	%bl	
%rcx	%ecx	%cx	%ch	%cl	
%rdx	%edx	%dx	%dh	%dl	
%rsi	%esi	%si		%sil	
%rdi	%edi	%di		%dil	
%rbp	%ebp	%bp		%bpl	
%rsp	%esp	%sp		%spl	

63	31	15	8	7	0
%r8	%r8d	%r8w		%r8b	
%r9	%r9d	%r9w		%r9b	
%r10	%r10d	%r10w		%r10b	
%r11	%r11d	%r11w		%r11b	
%r12	%r12d	%r12w		%r12b	
%r13	%r13d	%r13w		%r13b	
%r14	%r14d	%r14w		%r14b	
%r15	%r15d	%r15w		%r15b	

- chargement d'une constante dans un registre

```
movq    $0x2a, %rax    # rax <- 42
movq    $-12, %rdi
```

- chargement de l'adresse d'une étiquette dans un registre

```
movq    $label, %rdi
```

- copie d'un registre dans un autre

```
movq    %rax, %rbx    # rbx <- rax
```

- addition de deux registres

```
addq    %rax, %rbx    # rbx <- rbx + rax
```

(de même, subq, imulq)

- addition d'un registre et d'une constante

```
addq    $2, %rcx     # rcx <- rcx + 2
```

- incrémentation

```
incq    %rbx         # rbx <- rbx+1
```

(de même, decq)

- négation

```
negq    %rbx         # rbx <- -rbx
```

- non logique

```
notq    %rax                # rax <- not(rax)
```

- et, ou, ou exclusif

```
orq     %rbx, %rcx         # rcx <- or(rcx, rbx)  
andq    $0xff, %rcx       # efface les bits >= 8  
xorq    %rax, %rax        # met à zéro
```

- décalage à gauche (insertion de zéros)

```
salq    $3, %rax    # 3 fois
salq    %cl, %rbx   # cl fois
```

- décalage à droite arithmétique (copie du bit de signe)

```
sarq    $2, %rcx
```

- décalage à droite logique (insertion de zéros)

```
shrq    $4, %rdx
```

- rotation

```
rolq    $2, %rdi
rorq    $3, %rsi
```


le suffixe **q** dans les instructions précédentes signifie une opération sur 64 bits (*quad words*)

d'autres suffixes sont acceptés

suffixe	#octets	
b	1	(<i>byte</i>)
w	2	(<i>word</i>)
l	4	(<i>long</i>)
q	8	(<i>quad</i>)

```
movb    $42, %ah
```

quand les tailles des deux opérandes diffèrent,
il peut être nécessaire de préciser le mode d'**extension**

```
movzbq %al, %rdi    # avec extension de zéros  
movswl %ax, %edi    # avec extension de signe
```

une opérande entre parenthèses désigne un **adressage indirect**
i.e. l'emplacement mémoire à cette adresse

```
movq    $42, (%rax)    # mem[rax] <- 42
incq    (%rbx)         # mem[rbx] <- mem[rbx] + 1
```

note : l'adresse peut être une étiquette

```
movq    %rbx, (x)
```

la plupart des opérations n'acceptent pas plusieurs opérandes indirectes

```
addq    (%rax), (%rbx)
```

```
Error: too many memory references for 'add'
```

il faut donc passer par des registres

```
movq    (%rax), %rcx  
addq    %rcx, (%rbx)
```

plus généralement, une opérande

$$A(B, I, S)$$

désigne l'adresse $A + B + I \times S$ où

- A est une constante sur 32 bits signés
- I est un registre (vaut 0 si omis)
- $S \in \{1, 2, 4, 8\}$ (vaut 1 si omis)

```
movq    -8(%rax,%rdi,4), %rbx # rbx <- mem[-8+rax+4*rdi]
```

l'opération `leaq` calcule l'adresse effective correspondant à l'opérande

$$A(B, I, S)$$

```
leaq -8(%rax,%rdi,4), %rbx # rbx <- -8+rax+4*rdi
```

note : on peut s'en servir pour faire seulement de l'arithmétique

```
leaq (%rax,%rax,2), %rbx # rbx <- 3*%rax
```

la plupart des opérations positionnent des **drapeaux** (*flags*) du processeur selon leur résultat

drapeau	signification
ZF	le résultat est 0
CF	une retenue au-delà du bit de poids fort
SF	le résultat est négatif
OF	débordement de capacité (arith. signée)
etc.	

(exception notable : [lea](#))

des instructions permettent de tester les drapeaux

- saut conditionnel (`jcc`)

```
jne label
```

- positionne à 1 (vrai) ou 0 (faux) (`setcc`)

```
setge %bl
```

- mov conditionnel (`cmovcc`)

```
cmovl %rax, %rbx
```

suffixe	signification
e z	= 0
ne nz	≠ 0
s	< 0
ns	≥ 0
g	> signé
ge	≥ signé
l	< signé
le	≤ signé
a	> non signé
ae	≥ non signé
b	< non signé
be	≤ non signé

on peut positionner les drapeaux sans écrire le résultat quelque part,
pour la soustraction et le ET logique

```
cmpq    %rbx, %rax    # drapeaux de rax - rbx
```

(attention au sens !)

```
testq   %rbx, %rax    # drapeaux de and(rax, rbx)
```

- à une étiquette

```
jmp    label
```

- à une adresse calculée

```
jmp    *%rax
```

beaucoup, beaucoup d'autres instructions

[*Enumerating x86-64 — It's Not as Easy as Counting*]

dont notamment des instructions vectorielles (SSE) opérant sur des registres vectoriels contenant des entiers ou des flottants

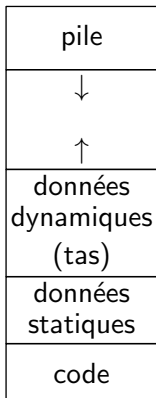
c'est de traduire un programme d'un langage de haut niveau vers ce jeu d'instructions

en particulier, il faut

- traduire les structures de contrôle (tests, boucles, exceptions, etc.)
- traduire les appels de fonctions
- traduire les structures de données complexes (tableaux, enregistrements, objets, clôtures, etc.)
- allouer de la mémoire dynamiquement

- constat** : les appels de fonctions peuvent être arbitrairement imbriqués
- ⇒ les registres peuvent ne pas suffire pour toutes les variables
 - ⇒ 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 ; `%rsp` 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

(note : chaque programme a l'illusion d'avoir toute la mémoire pour lui tout seul ; c'est l'OS qui crée cette illusion)

- on empile avec pushq

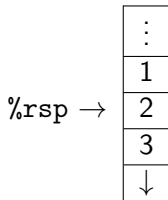
```
pushq    $42
pushq    %rax
```

- on dépile avec popq

```
popq     %rdi
popq     (%rbx)
```

exemple :

```
pushq    $1
pushq    $2
pushq    $3
popq     %rax
```



lorsqu'une fonction f (l'appelant ou *caller*)
souhaite appeler une fonction g (l'appelé ou *callee*),
on ne peut pas se contenter de faire

```
jmp g
```

car il faudra revenir dans le code de f quand g aura terminé

la solution consiste à se servir de la pile

deux instructions sont là pour ça

l'instruction

```
call    g
```

1. empile l'adresse de l'instruction située juste après le `call`
2. transfère le contrôle à l'adresse `g`

et l'instruction

```
ret
```

1. dépile une adresse
2. y transfère le contrôle

problème : tout registre utilisé par g sera perdu pour f

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

- jusqu'à six arguments sont passés dans les registres `%rdi`, `%rsi`, `%rdx`, `%rcx`, `%r8`, `%r9`
- les autres sont passés sur la pile, le cas échéant
- la valeur de retour est passée dans `%rax`

- les registres `%rbx`, `%rbp`, `%r12`, `%r13`, `%r14` et `%r15` sont **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
- les autres registres sont **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

- `%rsp` est le pointeur de pile, `%rbp` le pointeur de *frame*

en entrée de fonction, `%rsp + 8` doit être un multiple de 16

en particulier, des fonctions de bibliothèque (comme par ex. `scanf`) peuvent planter si cela n'est pas respecté

(pour `hello.s`, plus haut, on n'a pas pris la peine de le faire ; on a eu de la chance !)

aligner la pile peut être fait explicitement

```
f:  subq $8, %rsp # aligner la pile
    ...
    ... # car on fait des appels à des fonctions externes
    ...
    addq $8, %rsp
    ret
```

ou être obtenu gratuitement

```
f:  pushq %rbx # on sauvegarde %rbx
    ...
    ... # car on s'en sert ici
    ...
    popq  %rbx # et on le restaure
    ret
```

... ne sont que des conventions

en particulier, on est libre de ne pas les respecter
tant qu'on reste dans le périmètre de notre propre code

si on se lie à du code externe, en revanche,
on se doit de respecter les conventions d'appel

l'appel de fonction, en quatre temps

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

s'organisent autour d'un segment situé au sommet de la pile appelé le **tableau d'activation** (en anglais **stack frame**) situé entre `%rsp` et `%rbp`

1. passe les arguments dans `%rdi, ..., %r9`, les autres sur la pile s'il y en a plus de 6
2. sauvegarde les registres *caller-saved* qu'il compte utiliser après l'appel (dans son propre tableau d'activation)
3. exécute

```
call appelé
```


l'appelé, au début de l'appel

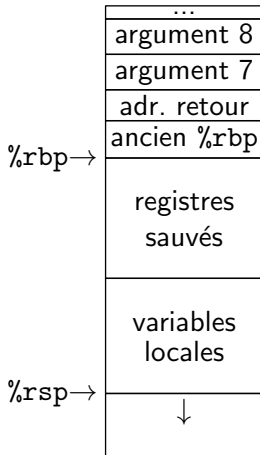
1. sauvegarde `%rbp` puis le positionne, par exemple

```
pushq    %rbp
movq     %rsp, %rbp
```

2. alloue son tableau d'activation, par exemple

```
subq     $48, %rsp
```

3. sauvegarde les registres *callee-saved* dont il aura besoin



`%rbp` 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 %rax
2. restaure les registres sauvegardés
3. dépile son tableau d'activation et restaure %rbp avec

```
leave
```

qui équivaut à

```
movq    %rbp, %rsp  
popq    %rbp
```

4. exécute

```
ret
```

1. dépile les éventuels arguments 7, 8, ...
2. restaure les registres *caller-saved*, si besoin

exercice : programmer la fonction suivante

```
isqrt( $n$ )  $\equiv$   
   $c \leftarrow 0$   
   $s \leftarrow 1$   
  while  $s \leq n$   
     $c \leftarrow c + 1$   
     $s \leftarrow s + 2c + 1$   
  return  $c$ 
```

afficher la valeur de `isqrt(17)`

exercice : programmer la fonction factorielle

- avec une boucle
- avec une fonction récursive

- une machine fournit
 - un jeu limité d'instructions, très primitives
 - des registres efficaces, un accès coûteux à la mémoire
- la mémoire est découpée en
 - code / données statiques / tas (données dynamiques) / pile
- les appels de fonctions s'articulent autour
 - d'une notion de tableau d'activation
 - 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));
}
```



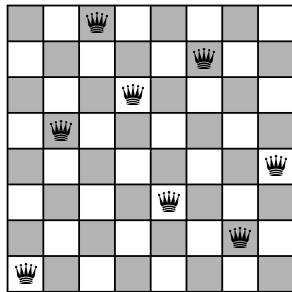
```

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));
}

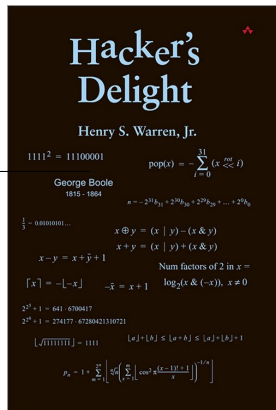
```

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



- recherche par rebroussement (*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 $b \subseteq a$
~a	$\complement a$
a&-a	$\{\min(a)\}$, quand $a \neq \emptyset$
~(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 : $-a = \sim a + 1$

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

exemple :

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

```

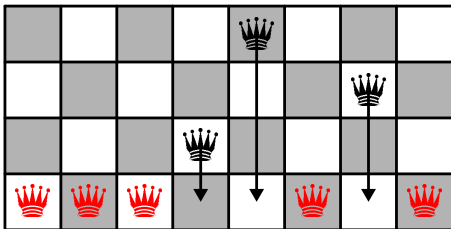
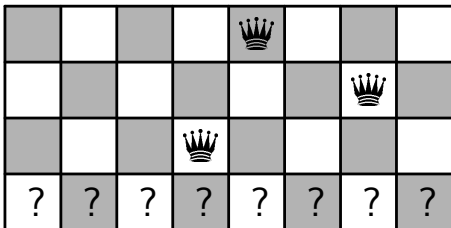
int t(a, b, c)
  f ← 1
  if a ≠ ∅
    e ← (a \ b) \ c
    f ← 0
    while e ≠ ∅
      d ← min(e)
      f ← f + t(a \ {d}, S(b ∪ {d}), P(c ∪ {d}))
      e ← e \ {d}
  return f

```

```

int queens(n)
  return t({0, 1, ..., n - 1}, ∅, ∅)

```



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

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

commençons par la fonction récursive `t` ; il faut

- allouer les registres
- compiler
 - le test
 - la boucle
 - l'appel récursif
 - les différents calculs

- a, b et c sont passés dans %rdi, %rsi et %rdx
- le résultat est renvoyé dans %rax
- les variables locales d, e et f seront stockées dans %r8, %rcx et %rax

en cas d'appel récursif, a, b, c, d, e et f auront besoin d'être sauvegardés, car ils sont tous utilisés après l'appel \Rightarrow sauvés sur la pile

	⋮
	adr. retour
%rbp \rightarrow	ancien %rbp
	%rax (f)
	%rcx (e)
	%r8 (d)
	%rdx (c)
	%rsi (b)
%rsp \rightarrow	%rdi (a)


```
int t(int a, int b, int c) {  
    int f=1;  
    if (a) {  
        ...  
    }  
    return f;  
}
```

```
t:      movq    $1, %rax  
        testq  %rdi, %rdi  
        jz    t_return  
        ...  
t_return:  
        ret
```

création/destruction du tableau d'activation

```
t:    ...
      pushq   %rbp
      movq    %rsp, %rbp
      subq    $48, %rsp          # allouer 6 mots sur la pile
      ...
      addq    $48, %rsp
      popq    %rbp
t_return:
      ret
```

```

if (a) {
    int d, e=a&~b&~c;
    f = 0;
    while ...
}

```

```

xorq %rax, %rax # f <- 0
movq %rdi, %rcx # e <- a & ~b & ~c
movq %rsi, %r9
notq %r9
andq %r9, %rcx
movq %rdx, %r9
notq %r9
andq %r9, %rcx

```

noter l'utilisation d'un registre temporaire %r9 (non sauvegardé)

```
while (expr) {  
    body  
}
```

```
...  
L1:  ...  
      calcul de expr dans %rcx  
      ...  
      testq  %rcx, %rcx  
      jz     L2  
      ...  
      body  
      ...  
      jmp   L1  
L2:  ...
```

il existe cependant une meilleure solution

```
while (expr) {  
    body  
}
```

```
...  
    jmp      L2  
L1:  ...  
        body  
    ...  
L2:  ...  
        expr  
    ...  
    testq   %rcx, %rcx  
    jnz     %rcx, L1
```

ainsi on fait un seul branchement par tour de boucle
(mis à part la toute première fois)

```
while (d=e&-e) {  
    ...  
}
```

```
                jmp      loop_test  
loop_body:  
    ...  
loop_test:  
    movq        %rcx, %r8  
    movq        %rcx, %r9  
    negq        %r9  
    andq        %r9, %r8  
    jnz         loop_body  
t_return:  
    ...
```

compilation de la boucle (suite)

```
while (...) {  
    f += t(a-d,  
          (b+d)*2,  
          (c+d)/2);  
    e -= d;  
}
```

loop_body:

```
movq    %rdi, 0(%rsp)  # a  
movq    %rsi, 8(%rsp)  # b  
movq    %rdx, 16(%rsp) # c  
movq    %r8, 24(%rsp)  # d  
movq    %rcx, 32(%rsp) # e  
movq    %rax, 40(%rsp) # f  
subq    %r8, %rdi  
addq    %r8, %rsi  
salq    $1, %rsi  
addq    %r8, %rdx  
shrq    $1, %rdx  
call    t  
addq    40(%rsp), %rax  # f  
movq    32(%rsp), %rcx  # e  
subq    24(%rsp), %rcx  # -= d  
movq    16(%rsp), %rdx  # c  
movq    8(%rsp), %rsi   # b  
movq    0(%rsp), %rdi   # a
```

```
int main() {  
    int q;  
    scanf("%d", &q);  
    ...  
}
```

```
main:  
    pushq    %rbp  
    movq    %rsp, %rbp  
    movq    $input, %rdi  
    movq    $q, %rsi  
    xorq    %rax, %rax  
    call   scanf  
    movq    (q), %rcx  
    ...  
  
    .data  
input:  
    .string "%d"  
q:  
    .quad   0
```



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

```
main:  
    ...  
    xorq    %rdi, %rdi  
    notq    %rdi  
    salq    %cl, %rdi  
    notq    %rdi  
    xorq    %rsi, %rsi  
    xorq    %rdx, %rdx  
    call   t  
    movq    $msg, %rdi  
    movq    %rax, %rsi  
    xorq    %rax, %rax  
    call   printf  
    xorq    %rax, %rax  
    popq    %rbp  
    ret
```

ce code n'est pas optimal

(par exemple, on pourrait ne sauvegarder que 5 registres)

mais il est meilleur que celui produit par `gcc -O2` ou `clang -O2`

aucun mérite, cependant : on a écrit un code assembleur **spécifique** à ce programme, pas un compilateur !

- produire du code assembleur efficace n'est pas chose aisée (observer le code produit par votre compilateur, avec `gcc -S -fverbose-asm`, `ocamlc -S`, etc., ou plus simplement sur <https://godbolt.org/>)
- maintenant il va falloir automatiser tout ce processus

lire

- *Computer Systems : A Programmer's Perspective*
(R. E. Bryant, D. R. O'Hallaron)
- son supplément PDF *x86-64 Machine-Level Programming*

- TD 1
 - petits exercices d'assembleur
 - génération de code pour un mini-langage d'expressions arithmétiques
- prochain cours
 - syntaxe abstraite
 - sémantique
 - interprète