UPMC Paris Universitas
Master Informatique - STL

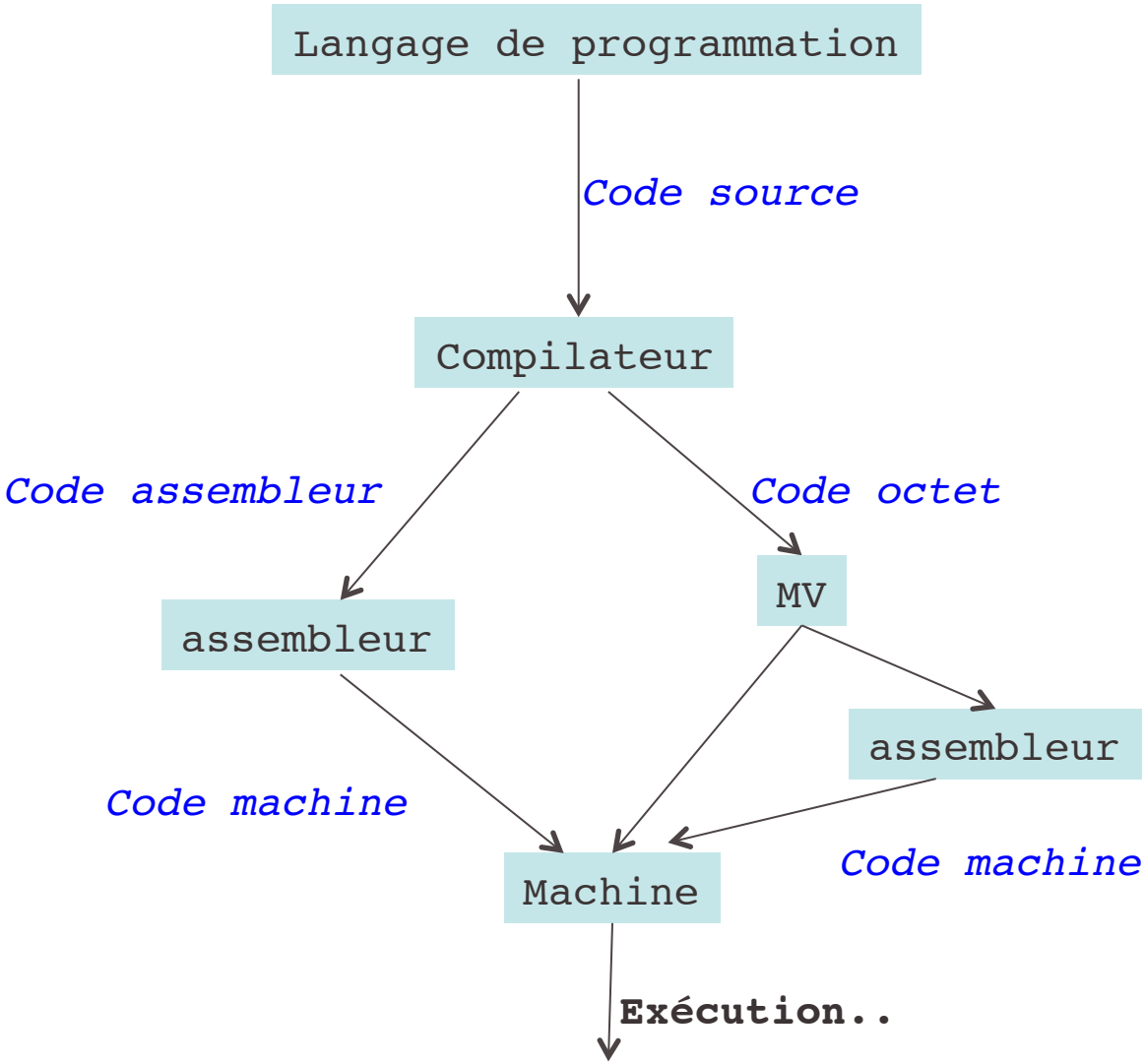
Compilation Avancée (MI190) 2014
Cours 2-3 : Machines Abstraites

Carlos Agon
agonc@ircam.fr

UPMC PARIS UNIVERSITAS

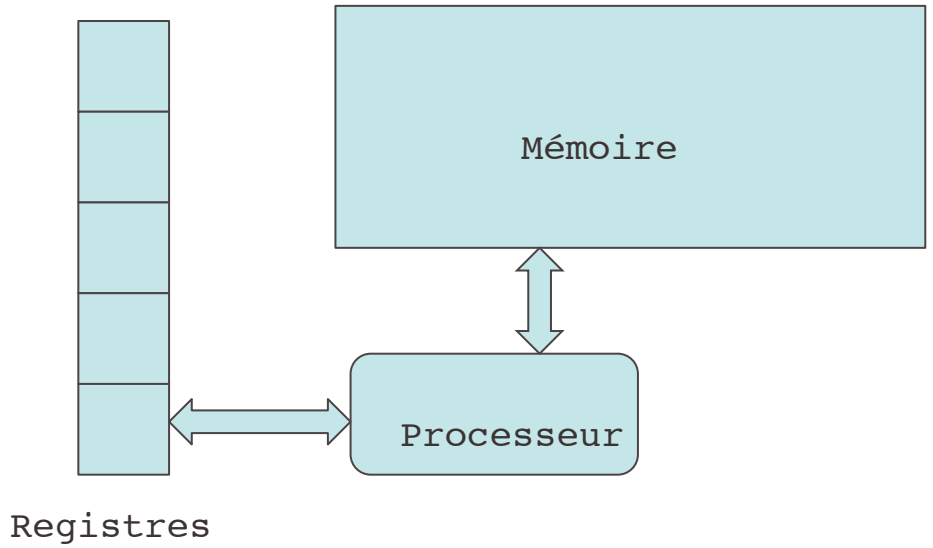
Du langage au code machine

Introduction



Code machine

Modèle de von Newman



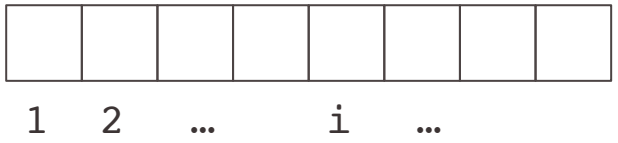
Le programme réside dans la mémoire et le processeur l'interprète, on parle de Machine Universelle.

Le programme est un donnée.

La différence principale entre les processeurs sont les registres et le jeu d'instructions

Registres

Compile (exp , i)



Compile (cte, i) = ***mov*** R_i, cte

Compile (variable, i) = ***load*** R_i, variable

Compile (e1 + e2, i) = **Compile** (e1 , i); **Compile** (e2 , i+1); ***add*** Ri, Ri, Ri+1

Registres

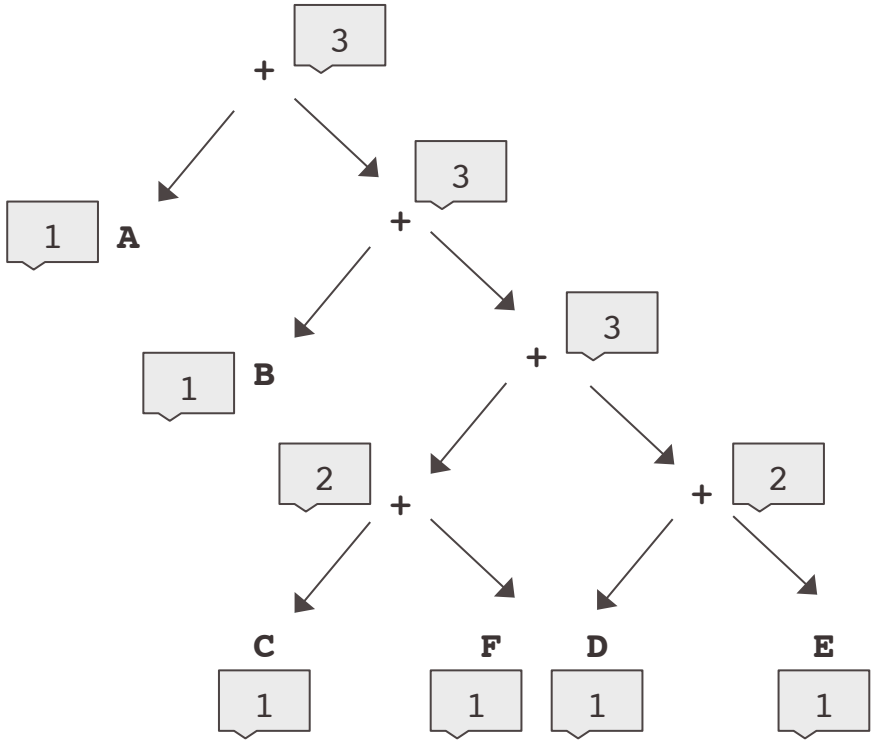
Compile (A+B+C+D+E, 0)

```
load R0, A
load R1, B
load R2, C
load R3, D
load R4, E
add R3, R3, R4
add R2, R2, R3
add R1, R1, R2
add R0, R0, R1
```

Registres

e1 + e2 utilise N registres avec

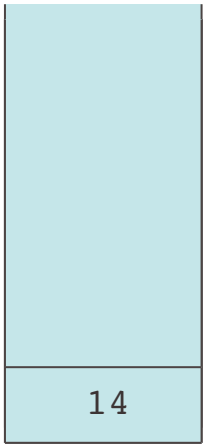
$$N = \begin{cases} \max(N1,N2) & \text{si } N1 \neq N2 \\ 1 + N1 & \text{si } N1 = N2 \end{cases}$$



Piles

Mini-FORTH

2 + (3*4) s'écrit **2 3 4 * +**



operateur := opPile | opArith | opBool

opPile := DUP | DROP | SWAP | ROT
opArith := * | +| / | - | mod
opPile := = | > | < | <>

Mini-FORTH

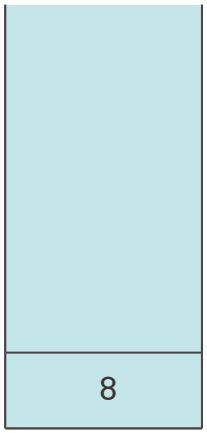
Definition := : NOM **liste_de_mots**

liste_de_mots := vide | mot **liste_de_mots**

: CARRE DUP *

: CUBE DUP CARRE *

2 CUBE



Factorielle

Cond := IF **liste_de_mots** THEN | IF **liste_de_mots** THEN **liste_de_mots** ELSE

: FACT DUP 1 > IF DUP 1 – FACT * THEN

3 FACT

Processeurs CISC/RISC

Complex Instruction Set
8086 ou Motorola 68000

Instructions complexes :
- Taille variable
- Mélangent mémoire et registres
- Grande cardinalité

Les instructions ne sont pas primitives. Sucre pour le programmeur qui veux générer de code à la main.

Peu de registres

Reduced Instruction Set
G4 Alpha Sparc

Instructions simples :
- Homogènes
- 2 instructions entre registres et mémoire (r/w) sinon elles sont entre registres
- Petite cardinalité

Plus facile pour générer de code automatiquement

32 registres

L'écart entre les deux n'est pas si grand.
On fait des mélanges (Intel)

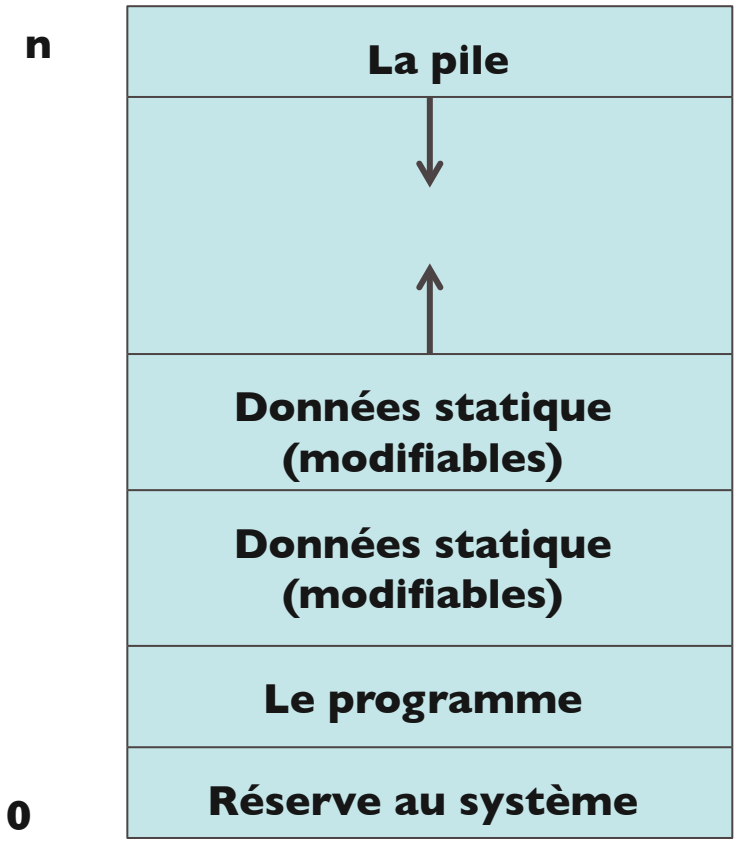
MIPS (les registres)

Nom	Numéro	Usage
zero	0	Zéro (toujours)
at	1	Réservé par l'assembleur
v0 .. v1	2 .. 3	Retour de valeurs
a0 .. a3	4 .. 7	Passage d'arguments
t0 .. t7	8 .. 15	Temporaires non sauvegardés
s0.. s7	16 .. 23	Temporaires sauvegardés
t8.. t9	24 .. 25	Temporaires non sauvegardés
k0.. k1	26 .. 27	Réservés par le système
gp	28	Global Pointer
sp	29	Stack Pointer
fp	30	Frame Pointeur
ra	31	Return Address

Il y a des registres spécifiques au processeur e.g. gp, sp

MIPS (*microprocessor without interlocked pipeline stages*)

De type RISC utilisé surtout par les systèmes SGI, mais aussi dans les consoles Nintendo, Play Station, etc.



Le MIPS (le jeu d'instructions)

add r1, r2, o qui place r2 + o dans r1

4 modes d'adressage (les arguments) :

① Immédiat : un entier
② Direct : le contenu du registre
③ Indirect : le contenu de l'adresse contenue dans un registre
④ Indirecte Indexé : le contenu de l'adresse contenue dans un registre augmenté d'un déplacement

Uniquement 2 instruction interagissent avec la mémoire :

① **lw** r1, n(r2): place dans r1 le mot contenu à l'adresse (r2+o)
② **sw** r1, n(r2): place le mot contenu dans r1 à l'adresse (r2+o)

Uniquement 2 instruction interagissent avec la mémoire :

① **jne** r,a,l : saute à l'adresse l si r et a sont différents
② **jal** o : saute à l'étiquette o et mets pc + 1 dans ra

Le MIPS (le jeu d'instructions)

Syntaxe	Effet
move r_1, r_2	$r_1 \leftarrow r_2$
add r_1, r_2, o	$r_1 \leftarrow o + r_2$
sub r_1, r_2, o	$r_1 \leftarrow r_2 - o$
mul r_1, r_2, o	$r_1 \leftarrow r_2 \times o$
div r_1, r_2, o	$r_1 \leftarrow r_2 \div o$
and r_1, r_2, o	$r_1 \leftarrow r_2$ `land` o
or r_1, r_2, o	$r_1 \leftarrow r_2$ `lor` o
xor r_1, r_2, o	$r_1 \leftarrow r_2$ `lxor` o
sll r_1, r_2, o	$r_1 \leftarrow r_2$ `lsl` o
srl r_1, r_2, o	$r_1 \leftarrow r_2$ `lsr` o
li r_1, n	$r_1 \leftarrow n$
`la` r_1, a	$r_1 \leftarrow a$

Syntaxe	Effet
lw $r_1, o(r_2)$	$r_1 \leftarrow tas.(r_2 + o)$
sw $r_1, o(r_2)$	$r_1 \rightarrow tas.(r_2 + o)$
slt r_1, r_2, o	$r_1 \leftarrow r_2 < o$
sle r_1, r_2, o	$r_1 \leftarrow r_2 \leq o$
seq r_1, r_2, o	$r_1 \leftarrow r_2 = o$
sne r_1, r_2, o	$r_1 \leftarrow r_2 \neq o$
j o	$pc \leftarrow o$
jal o	$ra \leftarrow pc + 1 \wedge pc \leftarrow o$
beq r, o, a	$pc \leftarrow a$ si $r = o$
bne r, o, a	$pc \leftarrow a$ si $r \neq o$
syscall	appel système
nop	ne fait rien

Le langage assembleur

Conversion vers le code machine. La traduction du langage machine en langage assembleur est facile. Elle permet de présenter les instructions machine (mots de 32 bits) sous une forme plus lisible.

Assembleur	Langage machine	Commentaire
`blt` r, o, a	`slt $1,` r, o `bne $1, $0,` a	Justifie le registre `at ($1)` réservé par l'assembleur.
`li $t0, 400020`	`lui $1, 6` `ori $8, $1, 6804`	charge les 16 bits de poids fort puis les 16 bits de poids faible
`add $t0, $t1, 1`	`addi $8, $9, 1`	addition avec une constante
`move $t0, $t1`	`addu $8, $0, $9`	addition "unsigned" avec zéro

Assembleur (exemples)

La fonction minimum en Pascal :

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

```
        blt $t1, $t2, Then   # si t1 < t2 saut à Then
        move $t3, $t2        # t3 := t2
        j End                # saut à End
Then:   move $t3, $t1        # t3 := t1
End:                         # fin du programme
```

Assembleur (exemples)

Pascal : calcule dans t2 = 0 la somme des entiers de 1 à t1 :

```
while t1 > 0  do
begin t2 := t2 + t1; t1 := t1 -1  end
```

```
While:
 if t1 <= 0  then goto
End
 else begin
       t2 := t2 + t1;
       t1 := t1 - 1;
       goto While
      end;
End:
```

```
While:
 ble $t1, $0, End
 add $t2, $t2, $t1
 sub $t1, $t1, 1
 j While
End:
```

```
 j test
:Loop
 add $t2, $t2, $t1
 sub $t1, $t1, 1
:Test
 bgt $t1, $0, Loop
```

Assembleur (allocation dynamique des données)

```
.data
.align  2           # aligner sur un mot (2^2 octets)
globaux :           # début de la zone des globaux
tableau :           # adresse symbolique de tableau
.space  4000        # taille en octets
c :
.space  1           # 1 octet
.align  2
i :
.space  4           # 4 octets
```

```
            # allouer a0 octets de mémoire
brk:        # procédure d'allocation dynamique
li $v0, 9   # appel système 9
syscall     # alloue une taille a0 et
j $ra       # retourne le pointeur dans v0
```

Assembleur (Procédures simples)

```
        .data                   # de la donnée
nl:
        .asciiz "\n"            # la chaîne "\n"
        .text                   # du code
writeln:                        # l'argument est dans a0
        li $v0, 1               # le numéro de print_int
        syscall                 # appel système
        li $v0, 4               # la primitive print_string
        la $a0, nl              # la chaîne "\n"
        syscall
        j $ra                   # retour par saut à l'adresse ra


        __start :
        li $a0 , 1              # a0 <- 1
        jal writeln             # ra <- pc+1; saut à writeln
        li $a0 , 2              # on recommence avec 2
        jal writeln
```

Assembleur (récursivité)

Pour sauver un registre r sur la pile :

```
sub $sp, $sp, 4    # alloue un mot sur la pile
sw r, 0($sp)       # écrit r sur le sommet de la pile
```

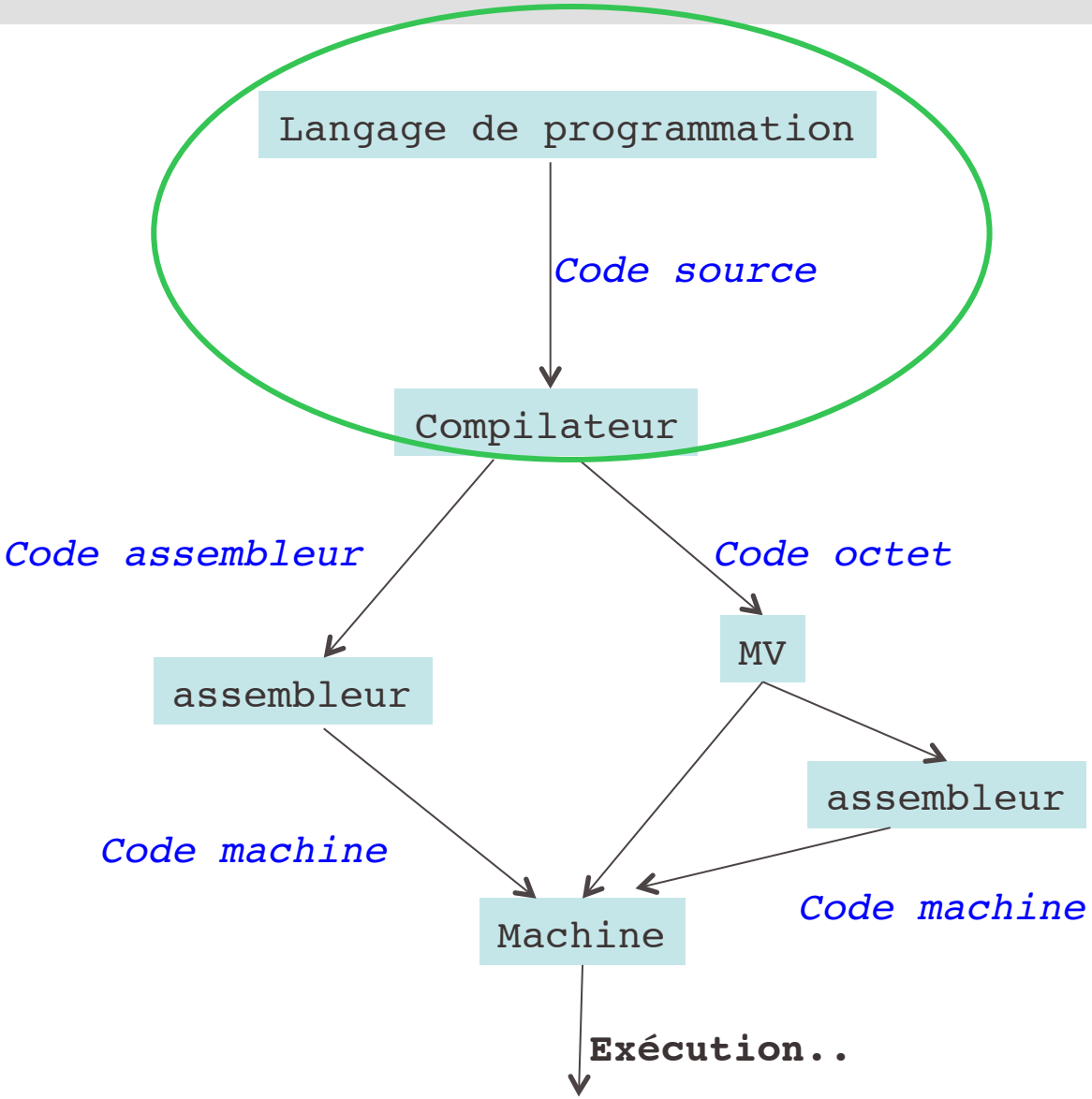
Pour restaurer un mot de la pile dans un registre r :

```
lw r, 0($sp)       # lit le sommet de la pile dans r
add $sp, $sp, 4    # désalloue un mot sur la pile
```

Assembleur (récursivité)

```
fact :
       blez $a0 , fact_0    # si a0 <= 0 saut à fact 0
       sub $sp , $sp , 8    # réserve deux mots en pile
       sw $ra , 0($sp )     # sauve l'adresse de retour
       sw $a0 , 4($sp )     # et la valeur de a0
       sub $a0 , $a0 , 1    # décrémente a0
       jal fact             # v0 <- appel récursif (a0-1)
       lw $a0 , 4($sp )     # récupère a0
       mul $v0 , $v0 , $a0  # v0 <- a0 * v0
       lw $ra , 0($sp )     # récupère l'adresse de retour
       add $sp , $sp , 8    # libère la pile
       j $ra                # retour à l'appelant
fact_0 :
       li $v0 , 1           # v0 <- 1
       j $ra                # retour à l'appelant
```

Introduction



Modeles de calcul

Calculabilité

Le but est de déterminer si un problème donné a une solution algorithmique ou non.

Peut-on définir arret?

```
(defun boucle () (boucle))

(defun test (F) (if (arret? F) (boucle) 1)))
```

-(test test) se termine alors (arret? (test test)) est vraie, mais (test test) boucle

-(test test) ne termine pas alors (arret? (test test)) est faux, mais (test test) = 1

La thèse de Church-Turing

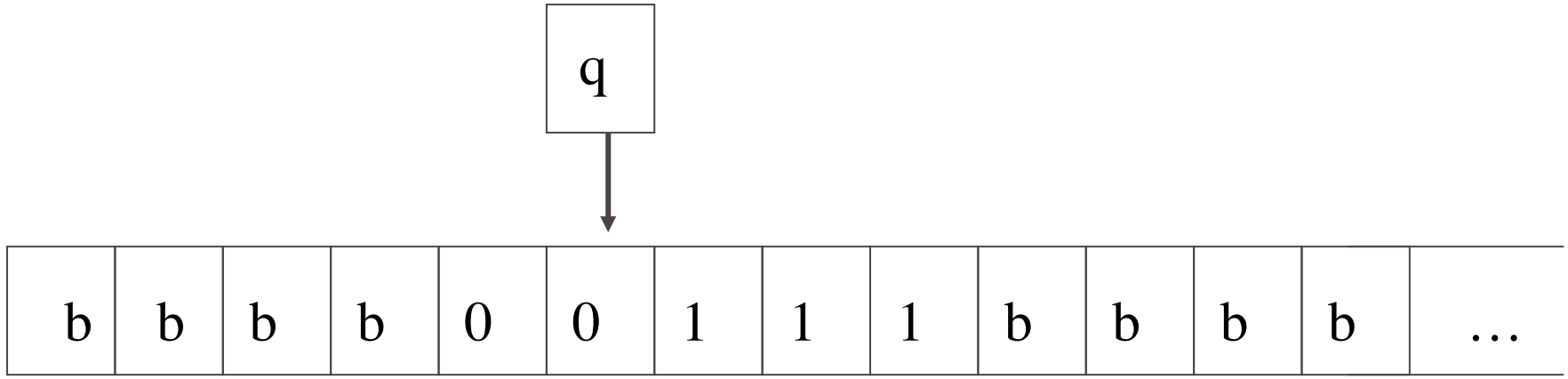
① Machine de Turing

② Lambda Calcul

③ Fonctions récursives (partielles)

④ Logique de premier ordre

La machine de Turing



$$M = (Q, \Sigma, q_0, \delta, F)$$

La machine de Turing

$Q = \{ q_0, \ldots, q_k \}$

$q_0 \in Q$ état initial

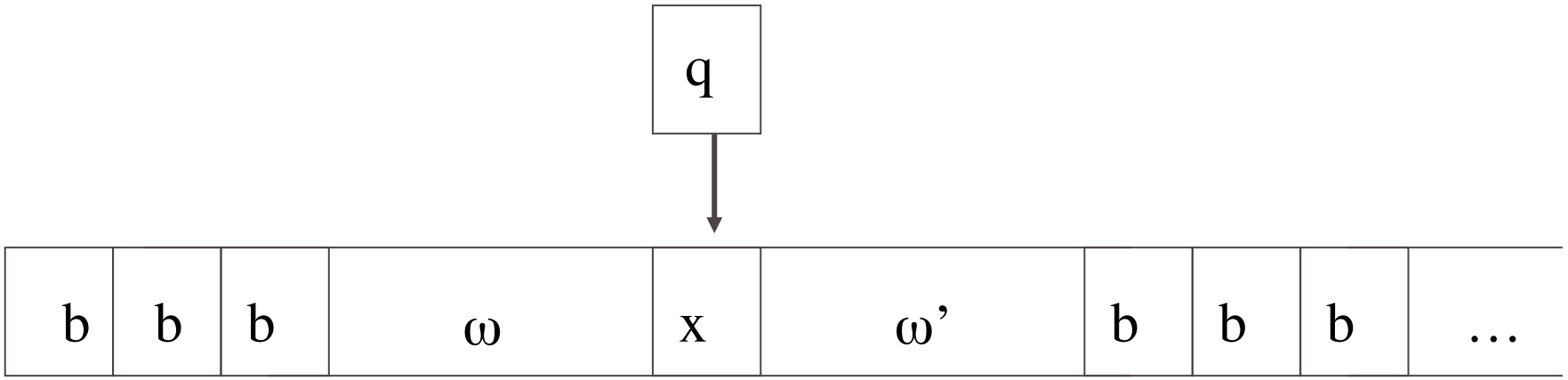
$F \subseteq Q$ états finaux

Σ = alphabet $\{0,1,b\}$

$\delta : Q \times \Sigma \rightarrow Q \times \Sigma \times \{G,D,S\}$

$\delta (q_1,1) = (q_2,0,D)$

Description instantanée



$$\alpha = (q, \omega, x, \omega')$$

où $q \in Q$

$x \in \Sigma$

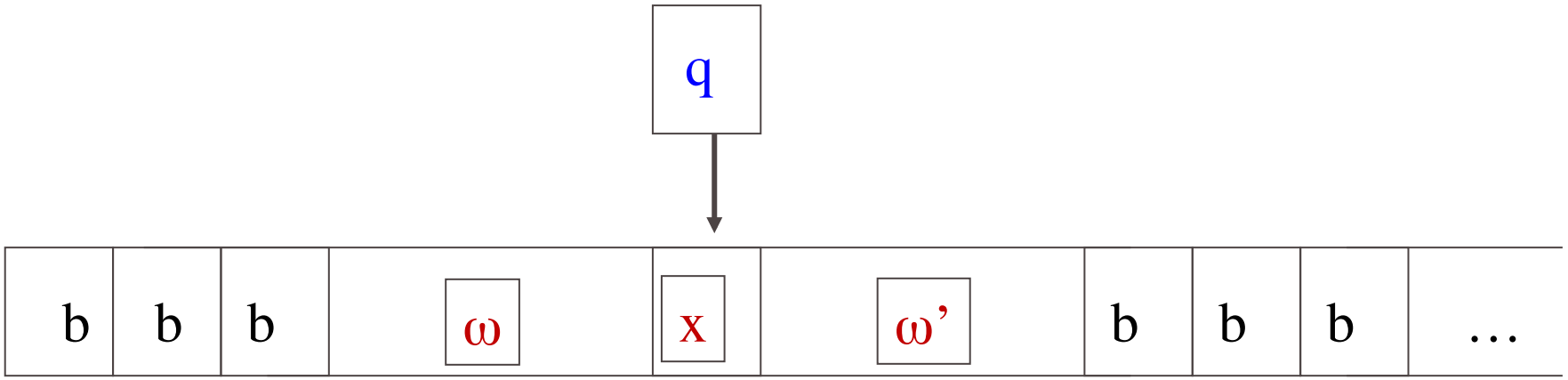
$\omega, \omega' \in \Sigma^*$

MT non-déterministes

Tout est pareil sauf que δ est une relation

$$\delta \subseteq Q \times \Sigma \times Q \times \Sigma \times \{G,D,S\}$$

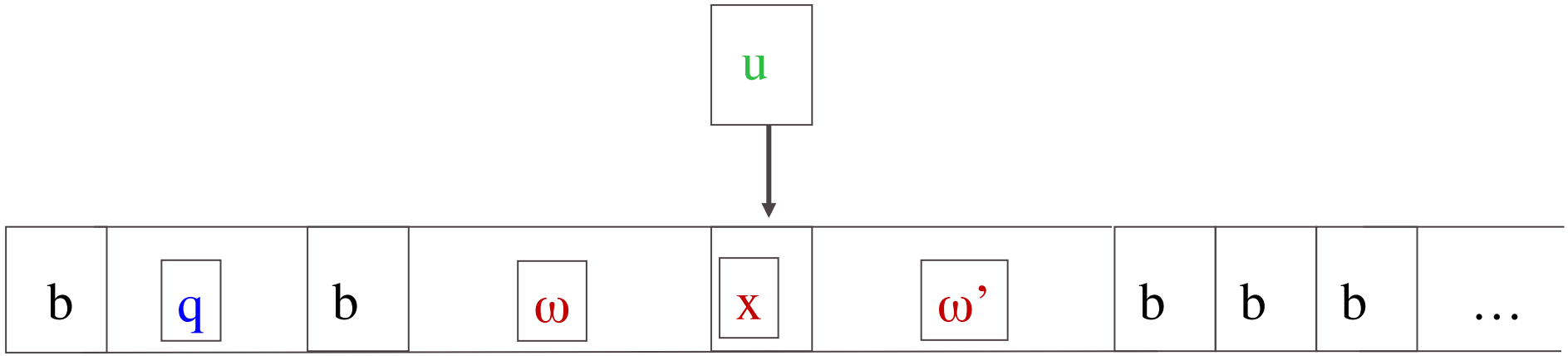
Machine universelle



L'entrée est ω, x, ω'

Le programme est donnée par q

Machine universelle



L'entrée est ω, x, ω'

Le programme est donnée par q

u définit la machine universelle

Syntaxe

E::= x

E::= E E (Application)

E::= λxE (Abstraction)

La β-réduction

Redex $(\lambda x.M)\ N$

Forme normale E tq. E n'a pas de redex

$$(\lambda x.M)\ N \rightarrow_\beta M[N/x]$$

$$((\lambda x.\ \lambda y.(x)y)b)c \rightarrow_\beta (\lambda y.(b)y)c$$

$$\rightarrow_\beta (b)c$$

Stratégies d'évaluation

$(\lambda y.v)((\lambda x.(x)x)\ \lambda x.(x)x)$

$\rightarrow_\beta\ (\lambda y.v)(\lambda x.(x)x)\ \lambda x.(x)x)$

$\rightarrow_\beta\ (\lambda y.v)(\lambda x.(x)x)\ \lambda x.(x)x)$

...

$(\lambda y.v)((\lambda x.(x)x)\ \lambda x.(x)x)$

$\rightarrow_\beta\ v$

Appel par nom

Réduire tjrs le *redex* le plus à gauche

$\lambda v.(\lambda z.z)((\lambda w.w)(x\ (\lambda y.y)))$

✔ $\lambda x.z\ (\text{fact}\ 10) \rightarrow_\beta z$

✖ $\lambda x.x + x\ (\text{fact}\ 10) \rightarrow_\beta (\text{fact}\ 10) + (\text{fact}\ 10)$

Appel par valeur

Réduire tjrs le *redex* le plus a gauche, mais si l'argument du *redex* est une valeur

$$\lambda v.(\lambda z.z)((\lambda w.w)(x\ (\ \lambda y.y)))$$

$$\lambda x.x + x\ (\text{fact}\ 10) \quad \rightarrow_\beta \quad \lambda x.x + x\ 3628800$$

$$\rightarrow_\beta \quad 7257600$$

Evaluation paresseuse

Réduire tjrs le *redex* le plus à gauche, mais si il n'est pas contenu dans une abstraction

$$\lambda v.(\lambda z.z)((\lambda w.w)(x\ (\ \lambda y.y)))$$

Récursivité

$MULT = \lambda x.\ \lambda y.\ \lambda z.(x)(y)z$ $\quad$ $IF = \lambda x.\ \lambda y.\ \lambda z.((x)y)z$

…

$FACT = \lambda n.(((IF)(ZERO?)n)\mathbf{1})((MULT)\ n)(FACT)\ (PRED)n$

$H = \lambda \mathbf{f}.\lambda n.(((IF)(ZERO?)n)\mathbf{1})((MULT)\ n)(\mathbf{f})\ (PRED)n$

$FACT \rightarrow_\beta (H)FACT$

J'ai besoin d'un **Y** tq.

$$\mathbf{Y}(H) \rightarrow_\beta FACT \rightarrow_\beta (H)FACT$$

Récursivité

$$\mathbf{Y}(E) \rightarrow_\beta \mathbf{G} \rightarrow_\beta \mathbf{(E)G}$$

$\mathbf{Y} = \lambda h.\ (\lambda x.(h)(x)\ x))\ \lambda x.(h)(x)\ x$

$\rightarrow_\beta \quad (\lambda h.\ (\lambda x.(h)(x)\ x))\ \lambda x.(h)(x)\ x)\ E$

$\rightarrow_\beta \quad (\lambda x.(E)(x)\ x))\ \lambda x.(E)(x)\ x \qquad = G$

$\rightarrow_\beta \quad (E)\ (\lambda x.(E)(x)\ x)\ \lambda x.(E)(x)\ x)$

$= \quad (E)\ G$

Machines abstraites pour les langages fonctionnels

Modèles d'exécution

- Interprétation :

Parcours de l'AST.

- Compilation en code natif :

Séquence d'instructions machine.

- Compilation en code d'une machine abstraite :

Séquence d'instructions abstraites. Ces instructions sont celles d'une machine abstraite, proches des opérations du langage source.

Une machine abstraite (super simple)

Le langage :

e:= N | e + e | e - e

Le jeu d'instructions de la machine :

CONST(N)	empiler l'entier N
ADD	dépiler deux entiers, empiler leur somme
SUB	dépiler deux entiers, empiler leur différence

Schéma de compilation :

C[N] = **CONST(N)**

C[a1 + a2] = C(a1); C[a2]; **ADD**

C[a1 - a2] = C(a1); C[a2]; **SUB**

Exemple :

C[5 – 1 + 2] = **CONST(5); CONST(1) ; CONST(2);ADD ; SUB**

Une machine abstraite pour les additions

Composants de la machine :

① Un pointeur de code

② Une pile

Transactions de la machine :

Etat avant		Etat après	
Code	Pile	Code	Pile
CONST(n);c	s	c	n.s
ADD;c	n2.n1;s	c	(n1 + n2).s
SUB;c	n2.n1;s	c	(n1 - n2).s

Evaluation

Etat initial code = C[exp] et pile = ε

Etat final code = ε et pile = v. ε v le résultat

Code	Pile
`CONST(3) ; CONST(1) ; CONST(2) ; ADD ; SUB`	ε
`CONST(1) ; CONST(2) ; ADD ; SUB`	3.ε
`CONST(2) ; ADD ; SUB`	1.3.ε
`ADD ; SUB`	2.1.3ε
`SUB`	3.3.ε
ε	0.ε

Exécution du code par interprétation

Interprète écrit en C ou assembler.

```
int interpreter(int * code)
{
int * s = bottom_of_stack;
while (1) {
 switch (*code++) {
  case CONST: *s++ = *code++; break;
  case ADD: s[-2] = s[-2] + s[-1]; s--; break;
  case SUB: s[-2] = s[-2] - s[-1]; s--; break;
  case EPSILON: return s[-1];
  }
 }
}
```

Exécution du code par expansion

Plus vite encore, convertir les instructions abstraites en séquences de code machine.

```
CONST(i)        --->    pushl $i
ADD             --->    popl %eax
                        addl 0(%esp), %eax
SUB             --->    popl %eax
                        subl 0(%esp), %eax
EPSILON         --->    popl %eax
                        ret
```

La SECD machine (Landin 64)

FERMETURES

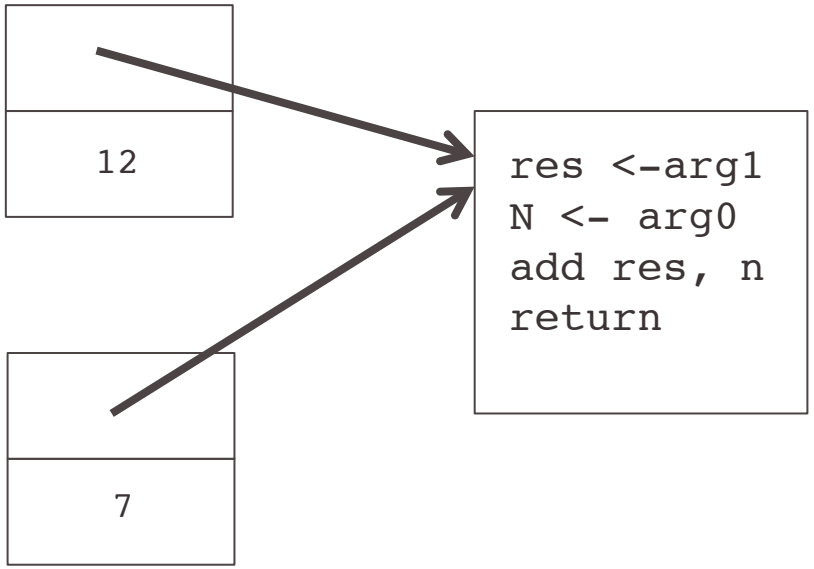
let transpose n = fun x -> x + n

```
transpose 12

res <- arg
add res, 12
return
```

```
transpose 7

res <- arg
add res, 7
return
```

La solution est de compiler comme une fermeture :

① Un pointer vers le code
② Environnement pour les variables libres de la fonction



Une machine abstraite pour λ-calcul en appel par valeur

Le langage :

e:= n | λ a | a b

Composants de la machine :

① Un pointeur de code c

② Un environnement e associant des valeurs aux variables libres)

Le jeu d'instructions de la machine :

ACCESS(n) empiler la n-ième entrée de l'environnement.

CLOSURE(c) empiler une fermeture du code c avec l'environnement courant.

APPLY dépiler une fermeture et un argument, faire l'application.

RETURN Terminer la fonction en cours, retourner à l'appelant.

Une machine abstraite pour λ-calcul en appel par valeur

Schéma de compilation :

C[n] = **ACCESS**(n)

C[λa] = **CLOSURE** (C[a]; **RETURN**)

C[a b] = C[a]; C[b]; **APPLY**

Transactions de la machine :

	Etat avant			Etat après	
Code	Env	Pile	Code	Env	Pile
ACCESS(n);c	e	s	c	e	e(n).s
CLOSURE(c');c	e	s	c	e	c'[e].s
APPLY;c	e	v.c'[e'].s	c'	v.e'	c.e.s
RETURN;c	e	v.c'.e'.s	c'	e'	v.s

C[e] est la fermeture du code c par l'environnement e

Evaluation

Expression : (λx x + 1)2

Code compilé :`CLOSURE(c); CONST(2); APPLY`

`avec c = ACCESS(1); CONST(1); ADD; RETURN`

Code	Env	Pile
`CLOSURE(c); CONST(2); APPLY`	e	s
`CONST(2); APPLY`	e	c[e].s
`APPLY`	e	2.c[e].s
`C`		
`ACCESS(1); CONST(1); ADD; RETURN`	2.e	ε.e.s
`CONST(1); ADD; RETURN`	2.e	2.ε.e.s
`ADD; RETURN`	2.e	1.2.ε.e.s
`RETURN`	2.e	3.ε.e.s
ε	e	3.s

Notation de Bruijn

$\lambda x.x$	$\lambda 1$
$\lambda x.x\lambda y.xy$	$\lambda 1\lambda 2\ 1$
$\lambda x\lambda y\lambda z.x$	$\lambda\ \lambda\ \lambda 3$
$\lambda y.(\lambda xy.x)y$	$\lambda(\lambda 2\ 1)\ 1$
$\lambda x.\lambda y.xy$	$\lambda\ \lambda 2\ 1$

La machine de Krivine pour λ-calcul en appel par valeur

Pareil que avant, mais cette fois la pile et l'environnement ne contiennent pas de valeurs mais de suspensions c-à-d des fermetures c[e] représentant des expressions dont l'évaluation est retardée jusqu'à ce qu'on ait besoin de leur valeur.

Le jeu d'instructions de la machine :

ACCESS(n) évalue la n-ième suspension de l'environnement.

PUSH(c) empile un suspension pour le code c

GRAB dépiler un argument et l'ajoute à l'environnement.

Schéma de compilation :

C[n] = **ACCESS**(n)

C[λa] = **GRAB** ; C[a]

C[a b] = **PUSH** (C[b]) ; C[a]

La machine de Krivine

Transactions de la machine :

Etat avant			Etat après		
Code	Env	Pile	Code	Env	Pile
ACCESS(n);c	e	s	c’	e’	s si e(n)=[c’,e’]
GRAB;c	e	c’[e’].s	c	c’[e’].s	s
PUSH(c’);c	e	s	c	e	c’[e].s

Références

Xavier Leroy
The zinc experiment: an economical implementation of the Ml langauge

Jean-Pierre FOURNIER
http://www.infeig.unige.ch/support/cpil/lect/mvp/web.html

Luc Maranget
Cours de compilation