Du langage à l'action: compilation et typage

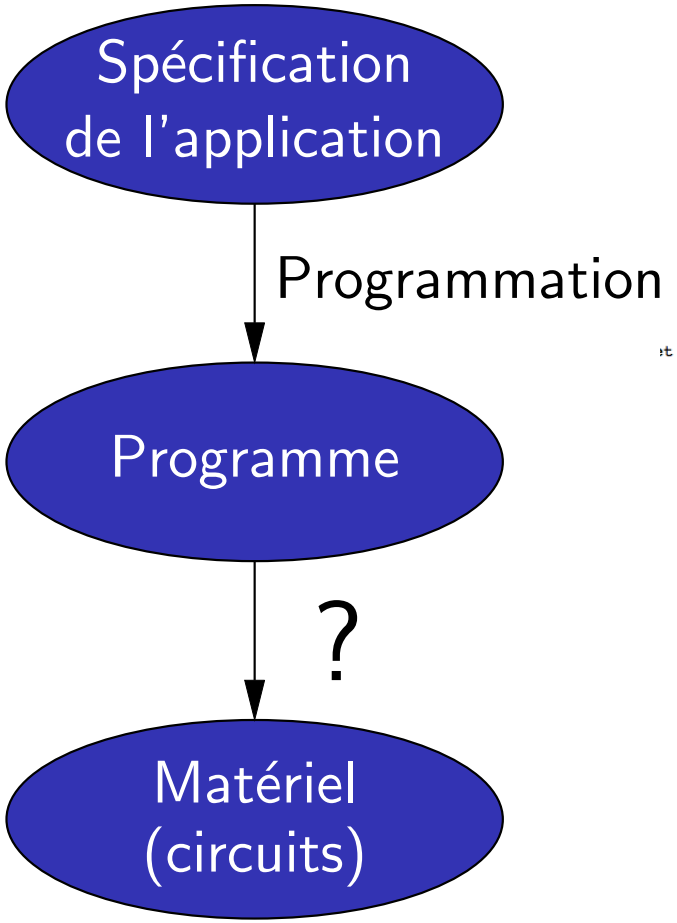
Xavier Leroy

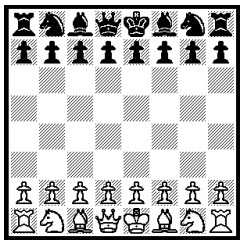
INRIA Paris-Rocquencourt

Collège de France, 8/2/2008

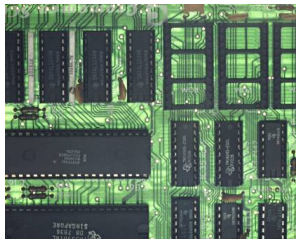
Du langage à l'action

Une fois un programme écrit, comment le faire exécuter par l'ordinateur ?





```
t rec alpha_beta pos depth achievable
  check_timer ();
  let pm = List.map (fun m -> delta po
  let pm = List.sort (fun (x, _) (y, _
  (match pm with
   | (d,m) :: _ when d >= win -> raise
   | _ -> ());
  let rec loop ach cut lst best =
      if ach >= cut then (ach, best) e
      (match lst with
       | [] -> (ach, best)
       | (d,m) :: tl ->
          (try
              let p_new = make_move po
```



Le langage de la machine

Le microprocesseur et ses circuits savent exécuter directement un seul langage : le langage machine.

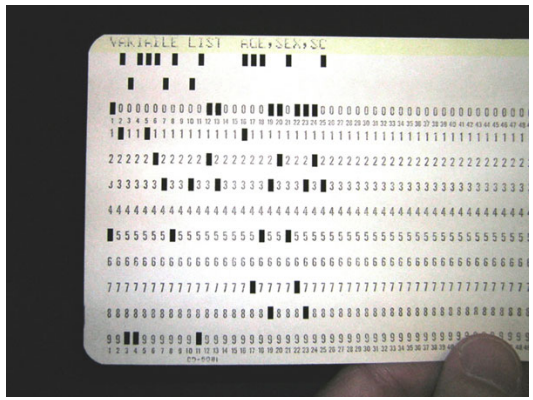
Ce langage est très pauvre et inadapté à la programmation.

Exemple : Le calcul de la factorielle, en code machine Intel.

```
10111000 00000001 00000000 00000000 00000000
10111010 00000010 00000000 00000000 00000000
00111001 11011010
01111111 00000110
00001111 10101111 11000010
01000010
11101011 11110110
11000011
```

La préhistoire (années 40) : traduction manuelle en langage machine

(Grace Hopper)

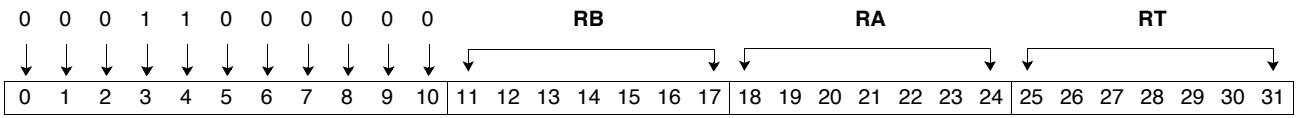


Instruction Set Architecture

Synergistic Processor Unit

Add Word **Required** **v 1.0**

a **rt,ra,rb**



For each of four word slots:

- The operand from register RA is added to the operand from register RB.
- The 32-bit result is placed in register RT.
- Overflows and carries are not detected.

Le réflexe de l'informaticien

Lorsqu'une tâche répétitive et ennuyeuse se présente,
écrire un programme pour la faire à notre place.

Première partie

La compilation

S'aider de la puissance de la machine

Le compilateur : un programme qui traduit automatiquement («compilation») un langage de haut niveau compréhensible par le programmeur en langage machine directement exécutable par le processeur.

L'Antiquité (1950) : le langage assembleur

Une représentation textuelle du langage de la machine :

- Utilisation de noms «mnémoniques» pour les instructions, les registres, etc.
- Utilisation de noms «symboliques» pour désigner les points dans le programme, les cases de la mémoire, etc.
- Possibilité de mettre des commentaires sur le code.

Exemple de programme en assembleur

Le calcul de la factorielle, en langage assembleur Intel.

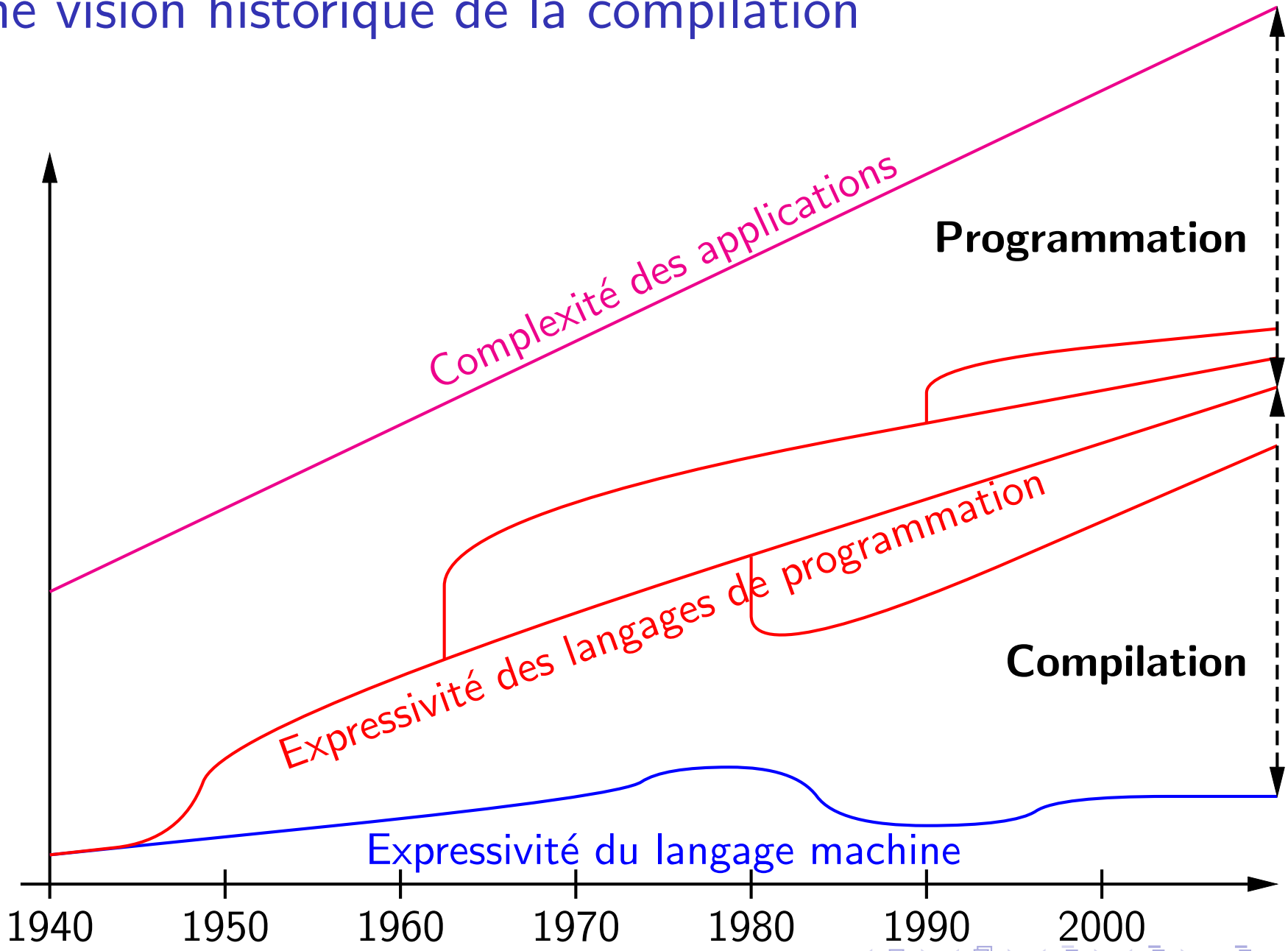
```
;; Calcul de la fonction factorielle.
;; En entrée: l'argument N est dans le registre EBX
;; En sortie: la factorielle de N est dans le registre EAX

Factorielle:
        mov eax, 1      ;; résultat initial = 1
        mov edx, 2      ;; indice de boucle = 2
L1:     cmp edx, ebx    ;; tant que indice <= N ...
        jg L2
        imul eax, edx   ;; multiplier le résultat par l'indice
        inc edx         ;; incrémenter l'indice
        jmp L1          ;; fin tant que
L2:     ret             ;; fin de la fonction Factorielle
```

Compilation du langage assembleur

- Calcul des adresses mémoire correspondant aux noms symboliques.
- Encodage sous forme de nombres des mnémoniques d'instructions et de registres.
- Production d'un listing, d'un index de références croisées, etc, pour faciliter la relecture du code.

Degré d'automatisation modeste, mais qui a cependant permis un saut qualitatif dans la complexité des programmes que l'on peut écrire.

Une vision historique de la compilation



La Renaissance (1957) : les expressions arithmétiques

Un problème récurrent en programmation : calculer la valeur de formules mathématiques compliquées.

$$x_1, x_2 = \frac{-b \pm \sqrt{b^2 - 4ac}}{2a}$$

$\rightarrow$ Le langage FORTRAN (1957) (FORmula TRANslator).

En assembleur :

```
mul t1, b, b      sub x1, d, b
mul t2, a, c      div x1, x1, t3
mul t2, t2, 4     neg x2, b
sub t1, t1, t2    sub x2, x2, d
sqrt d, t1        div x2, x2, t3
mul t3, a, 2
```

En FORTRAN :

```
D = SQRT(B*B - 4*A*C)
X1 = (-B + D) / (2*A)
X2 = (-B - D) / (2*A)
```

Comment compiler les expressions arithmétiques ?

Algorithme $\mathcal{C}(e, i)$:

Entrées :

- e : une expression arithmétique, représentée par son arbre de syntaxe abstraite.
- i : le numéro du registre de destination.

Résultat :

- Une séquence d'instructions assembleur
- ... dont l'exécution calcule la valeur de e
- ... et la dépose dans le registre R_i
- ... tout en préservant la valeur des registres $R_0, \ldots, R_{i-1}$.

Compilation des expressions arithmétiques

Par récursion et par cas sur la forme de l'expression :

- $\mathcal{C}(\mathit{constante}, i) = \texttt{mov}\ R_i, \mathit{constante}$
  (charge la constante dans le registre R_i)
- $\mathcal{C}(\mathit{variable}, i) = \texttt{load}\ R_i, \mathit{variable}$
  (charge le contenu de la case mémoire dans le registre R_i)
- $\mathcal{C}(e_1 + e_2, i) = \mathcal{C}(e_1, i);\ \mathcal{C}(e_2, i+1);\ \texttt{add}\ R_i, R_i, R_{i+1}$

Compilation des expressions arithmétiques

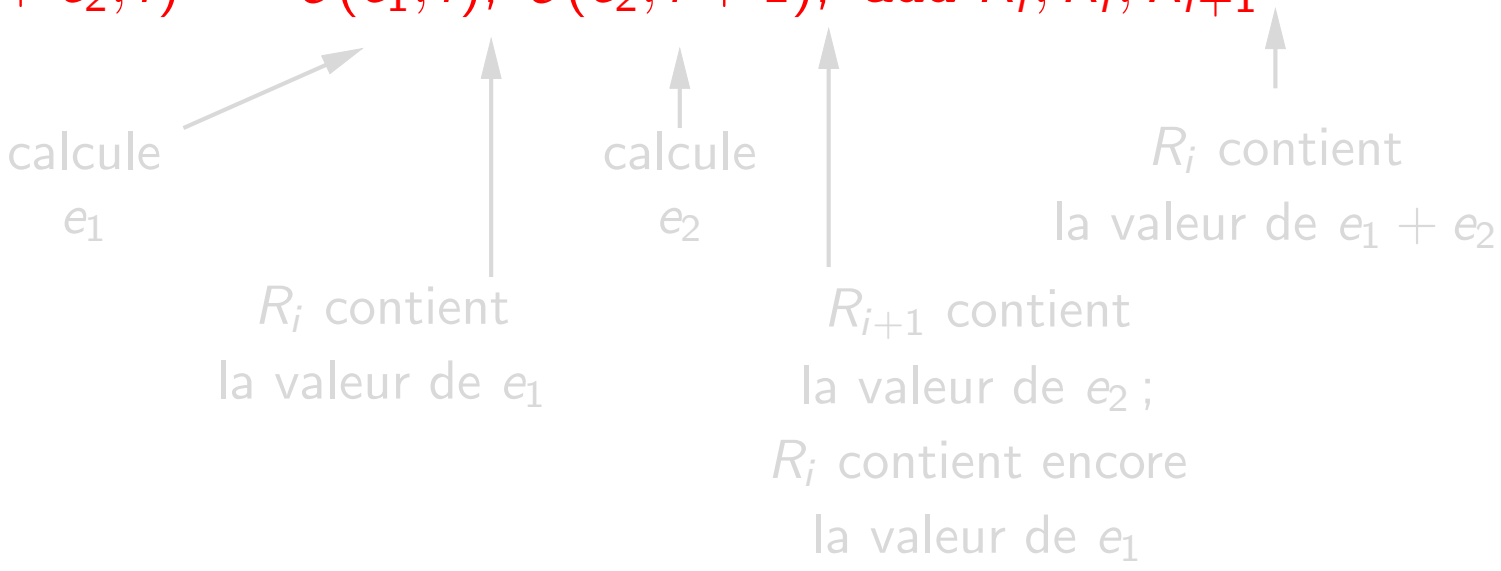
Par récursion et par cas sur la forme de l'expression :

- $\mathcal{C}(\mathit{constante}, i) = \texttt{mov}\ R_i, \mathit{constante}$
  (charge la constante dans le registre R_i)
- $\mathcal{C}(\mathit{variable}, i) = \texttt{load}\ R_i, \mathit{variable}$
  (charge le contenu de la case mémoire dans le registre R_i)
- $\mathcal{C}(e_1 + e_2, i) = \mathcal{C}(e_1, i);\ \mathcal{C}(e_2, i+1);\ \texttt{add}\ R_i, R_i, R_{i+1}$

Compilation des expressions arithmétiques

Par récursion et par cas sur la forme de l'expression :

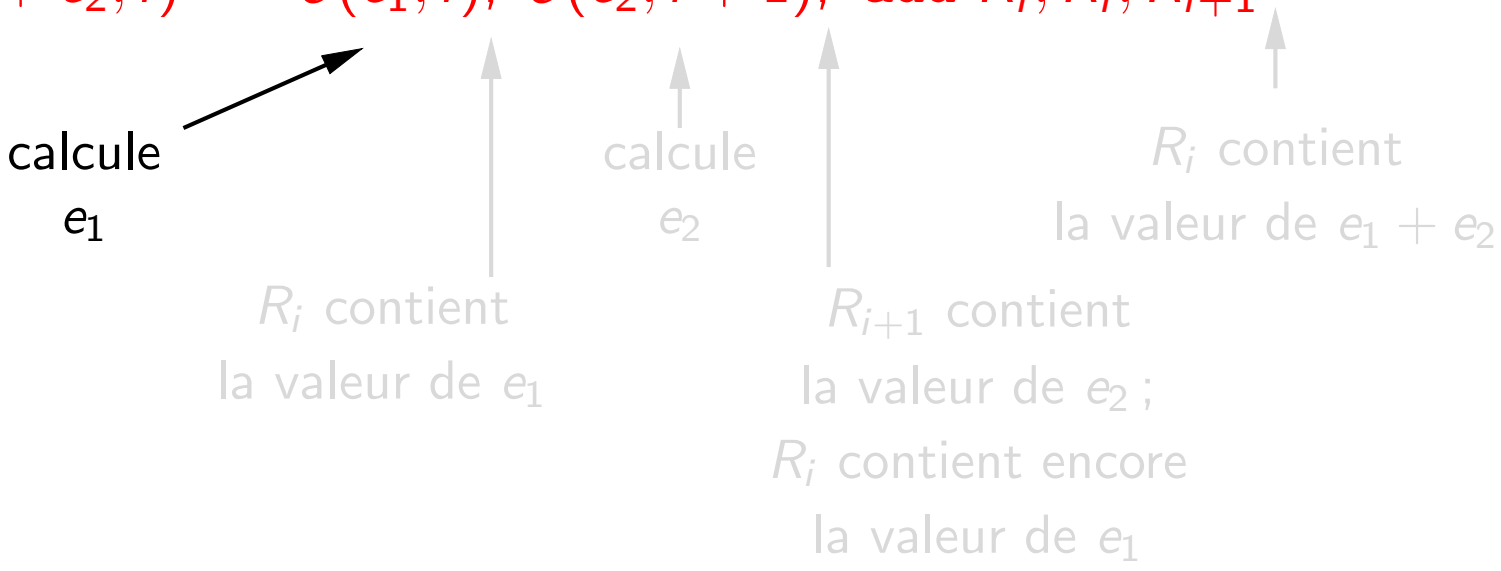
- $\mathcal{C}(\mathit{constante}, i) = \texttt{mov}\ R_i, \mathit{constante}$
  (charge la constante dans le registre R_i)
- $\mathcal{C}(\mathit{variable}, i) = \texttt{load}\ R_i, \mathit{variable}$
  (charge le contenu de la case mémoire dans le registre R_i)
- $\mathcal{C}(e_1 + e_2, i) = \mathcal{C}(e_1, i);\ \mathcal{C}(e_2, i+1);\ \texttt{add}\ R_i, R_i, R_{i+1}$



Compilation des expressions arithmétiques

Par récursion et par cas sur la forme de l'expression :

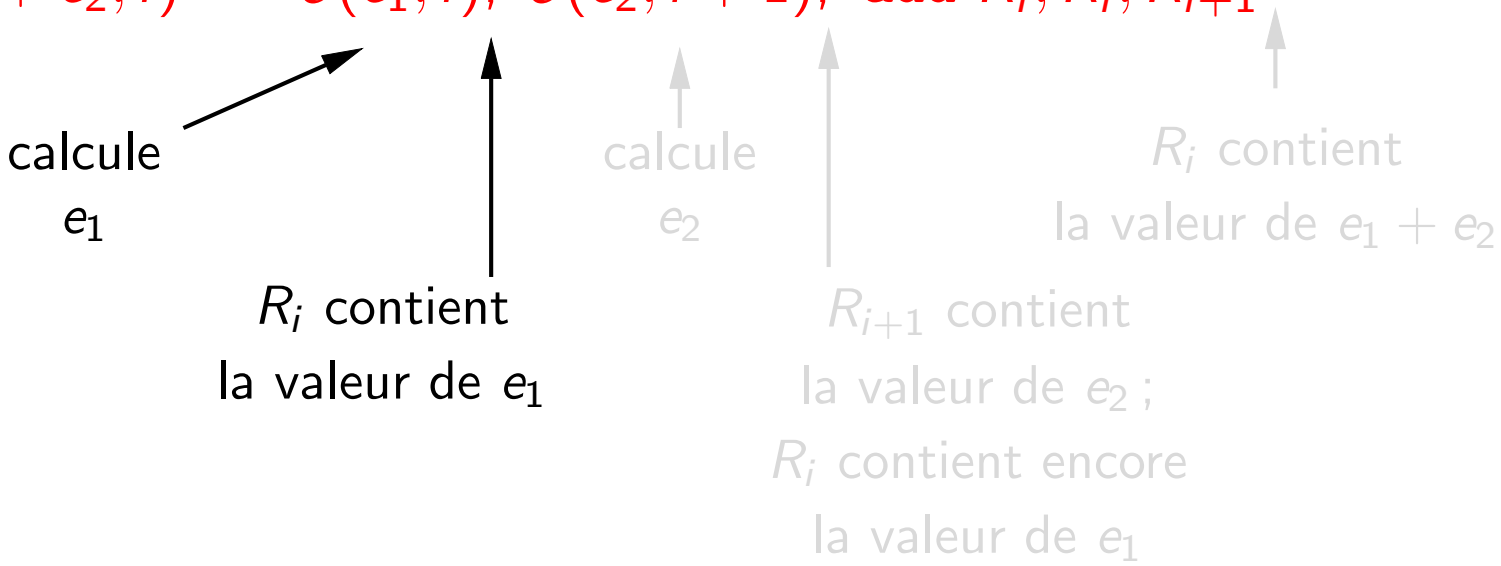
- $\mathcal{C}(constante, i) = \texttt{mov}\ R_i, constante$
  (charge la constante dans le registre R_i)
- $\mathcal{C}(variable, i) = \texttt{load}\ R_i, variable$
  (charge le contenu de la case mémoire dans le registre R_i)
- $\mathcal{C}(e_1 + e_2, i) = \mathcal{C}(e_1, i);\ \mathcal{C}(e_2, i+1);\ \texttt{add}\ R_i, R_i, R_{i+1}$



Compilation des expressions arithmétiques

Par récursion et par cas sur la forme de l'expression :

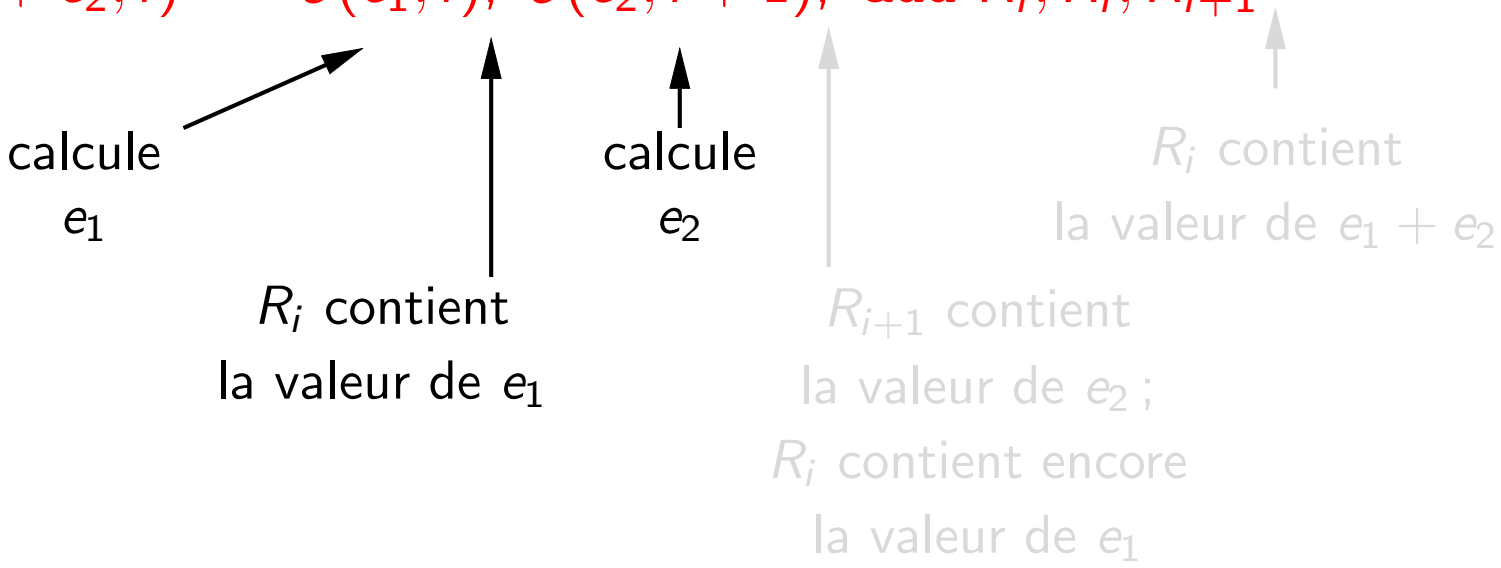
- $\mathcal{C}(\mathit{constante}, i) = \texttt{mov}\ R_i, \mathit{constante}$
  (charge la constante dans le registre R_i)
- $\mathcal{C}(\mathit{variable}, i) = \texttt{load}\ R_i, \mathit{variable}$
  (charge le contenu de la case mémoire dans le registre R_i)
- $\mathcal{C}(e_1 + e_2, i) = \mathcal{C}(e_1, i);\ \mathcal{C}(e_2, i+1);\ \texttt{add}\ R_i, R_i, R_{i+1}$



Compilation des expressions arithmétiques

Par récursion et par cas sur la forme de l'expression :

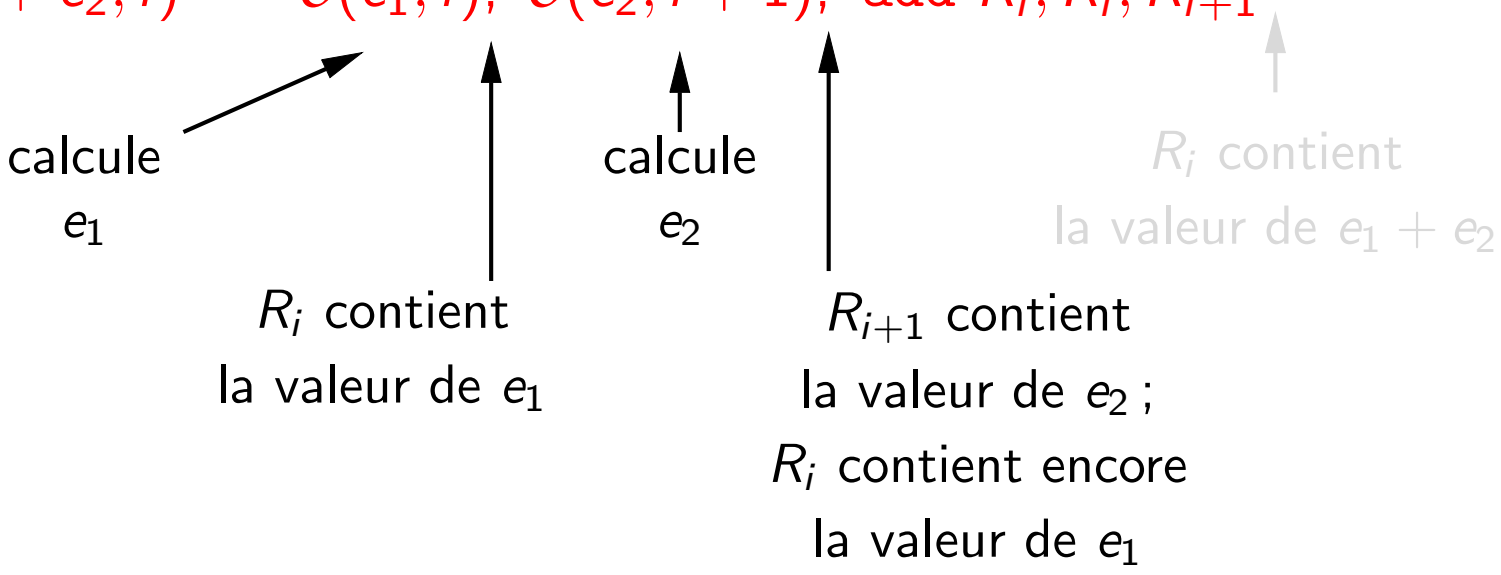
- $\mathcal{C}(\mathit{constante}, i) = \texttt{mov}\ R_i, \mathit{constante}$
  (charge la constante dans le registre R_i)
- $\mathcal{C}(\mathit{variable}, i) = \texttt{load}\ R_i, \mathit{variable}$
  (charge le contenu de la case mémoire dans le registre R_i)
- $\mathcal{C}(e_1 + e_2, i) = \mathcal{C}(e_1, i);\ \mathcal{C}(e_2, i+1);\ \texttt{add}\ R_i, R_i, R_{i+1}$



Compilation des expressions arithmétiques

Par récursion et par cas sur la forme de l'expression :

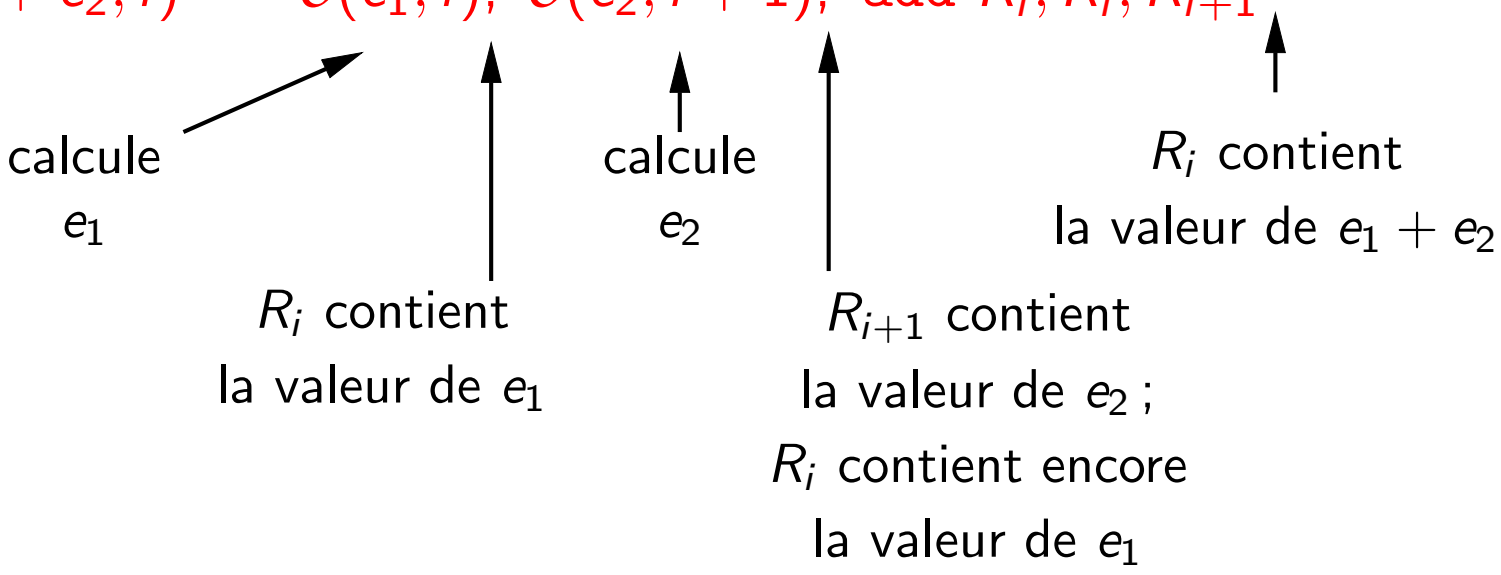
- $\mathcal{C}(\mathit{constante}, i) = \texttt{mov}\ R_i, \mathit{constante}$
  (charge la constante dans le registre R_i)
- $\mathcal{C}(\mathit{variable}, i) = \texttt{load}\ R_i, \mathit{variable}$
  (charge le contenu de la case mémoire dans le registre R_i)
- $\mathcal{C}(e_1 + e_2, i) = \mathcal{C}(e_1, i);\ \mathcal{C}(e_2, i+1);\ \texttt{add}\ R_i, R_i, R_{i+1}$



Compilation des expressions arithmétiques

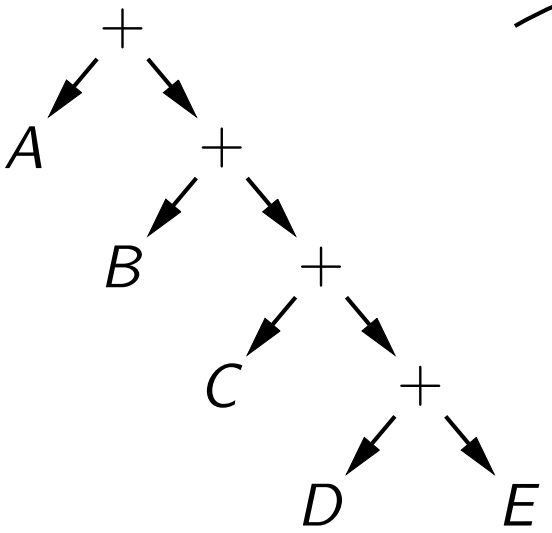
Par récursion et par cas sur la forme de l'expression :

- $\mathcal{C}(\mathit{constante}, i) = \texttt{mov}\ R_i, \mathit{constante}$
  (charge la constante dans le registre R_i)
- $\mathcal{C}(\mathit{variable}, i) = \texttt{load}\ R_i, \mathit{variable}$
  (charge le contenu de la case mémoire dans le registre R_i)
- $\mathcal{C}(e_1 + e_2, i) = \mathcal{C}(e_1, i);\ \mathcal{C}(e_2, i+1);\ \texttt{add}\ R_i, R_i, R_{i+1}$



Exemple, et problème

Expression e



$\mathcal{C}(e, 0)$

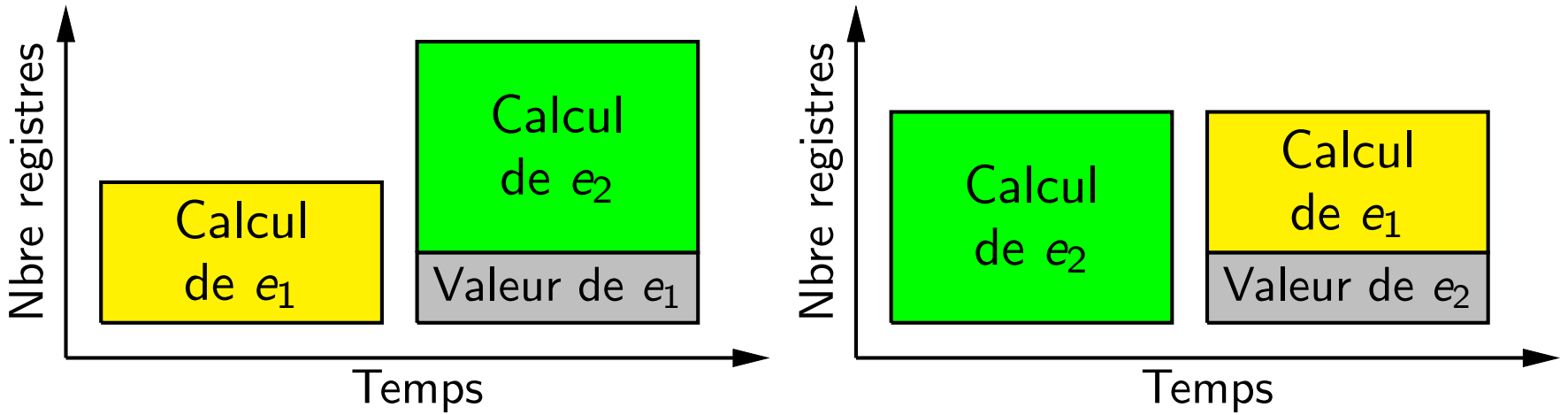
```
load R0, A
load R1, B
load R2, C
load R3, D
load R4, E
add R3, R3, R4
add R2, R2, R3
add R1, R1, R2
add R0, R0, R1
```

Ce code utilise beaucoup de registres temporaires !

Comment économiser les registres ?

Supposons qu'on sache calculer e_1 en utilisant N_1 registres, et e_2 en utilisant N_2 registres.

Pour calculer $e_1 + e_2$, on peut commencer ou bien par e_1 ou par e_2 :

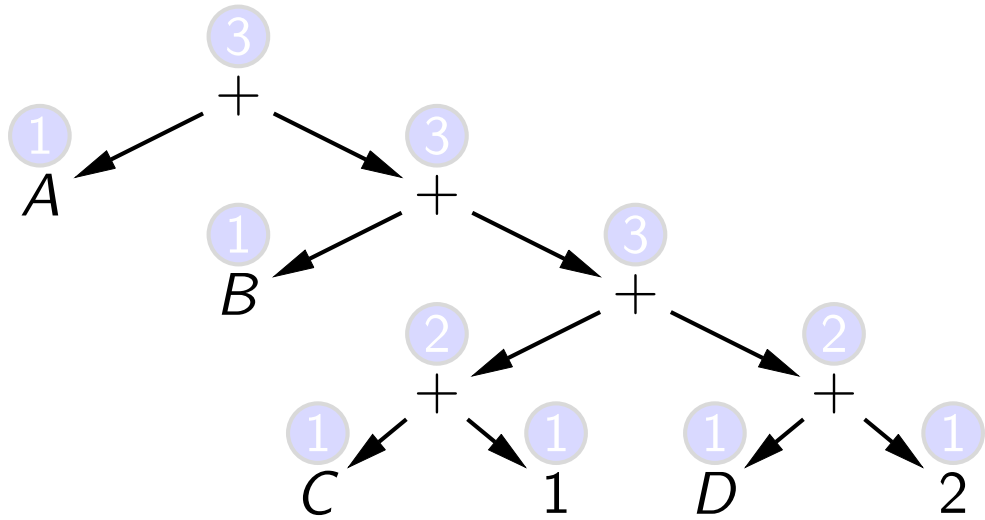


Le scénario le plus économe en registres est de commencer par e_1 si $N_1 > N_2$ et par e_2 si $N_1 < N_2$. Le calcul de $e_1 + e_2$ utilise alors N registres :

$$N = \begin{cases} \max(N_1, N_2) & \text{si } N_1 \neq N_2 \\ 1 + N_1 & \text{si } N_1 = N_2 \end{cases}$$

Comment économiser les registres ?

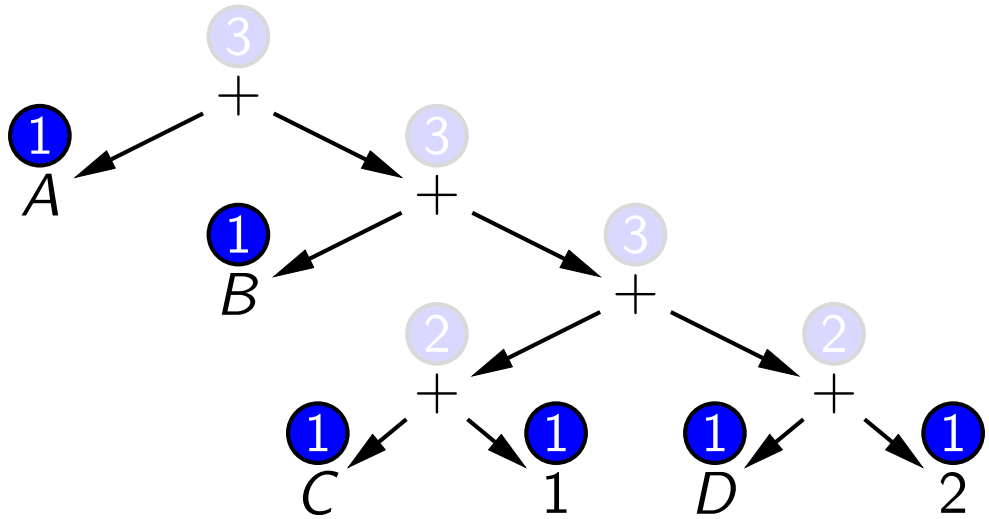
Phase 1 : en utilisant la formule précédente, calculer le nombre minimal de registres nécessaires pour chaque sous-expression de l'expression à compiler.



Phase 2 : produire le code de manière récursive comme précédemment, mais en commençant toujours par la sous-expression nécessitant le plus grand nombre de registres.

Comment économiser les registres ?

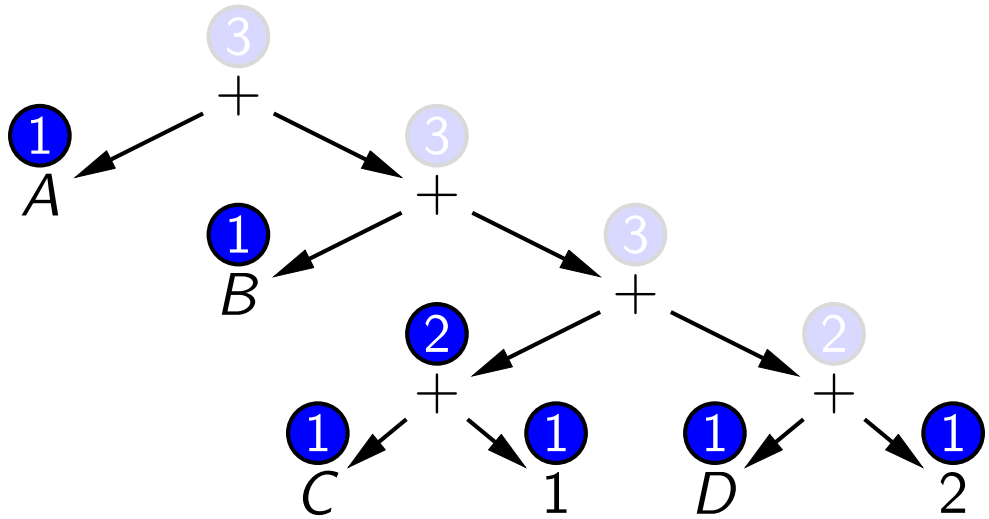
Phase 1 : en utilisant la formule précédente, calculer le nombre minimal de registres nécessaires pour chaque sous-expression de l'expression à compiler.



Phase 2 : produire le code de manière récursive comme précédemment, mais en commençant toujours par la sous-expression nécessitant le plus grand nombre de registres.

Comment économiser les registres ?

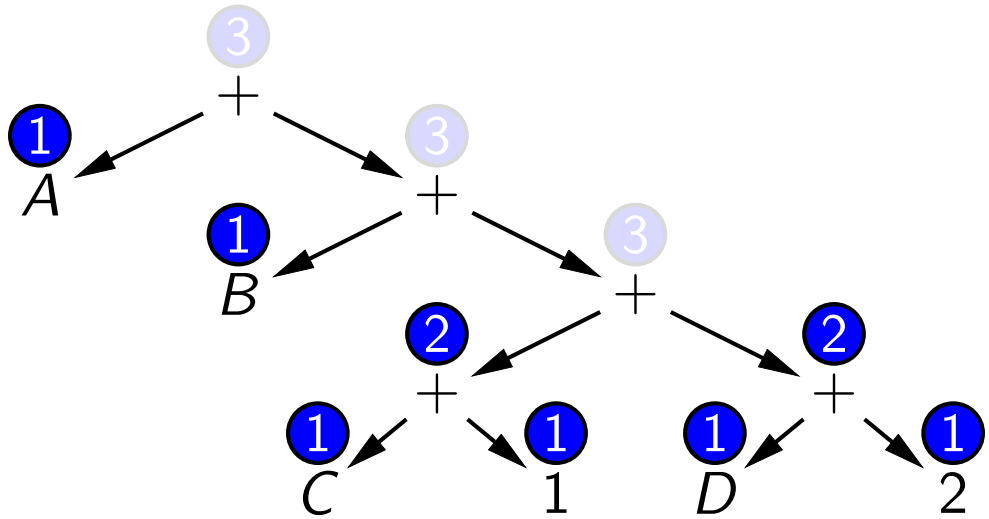
Phase 1 : en utilisant la formule précédente, calculer le nombre minimal de registres nécessaires pour chaque sous-expression de l'expression à compiler.



Phase 2 : produire le code de manière récursive comme précédemment, mais en commençant toujours par la sous-expression nécessitant le plus grand nombre de registres.

Comment économiser les registres ?

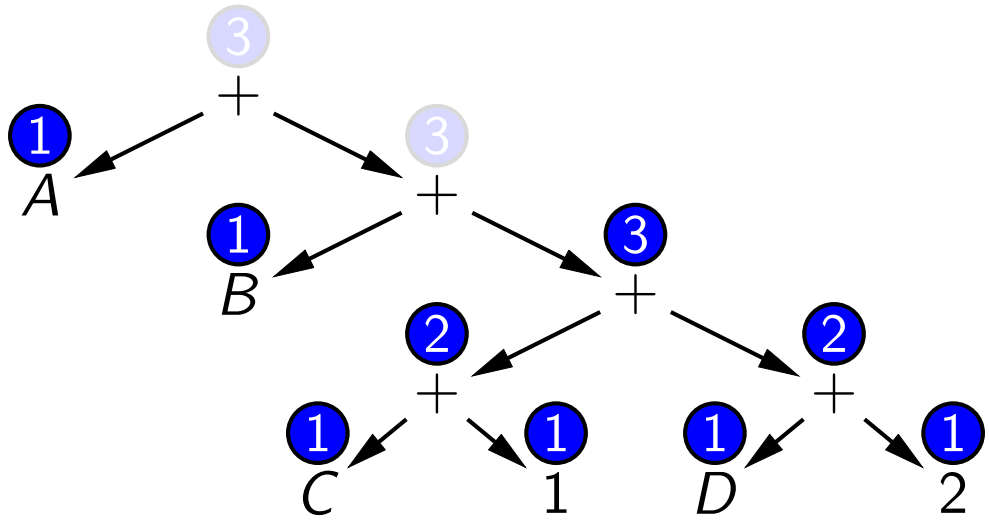
Phase 1 : en utilisant la formule précédente, calculer le nombre minimal de registres nécessaires pour chaque sous-expression de l'expression à compiler.



Phase 2 : produire le code de manière récursive comme précédemment, mais en commençant toujours par la sous-expression nécessitant le plus grand nombre de registres.

Comment économiser les registres ?

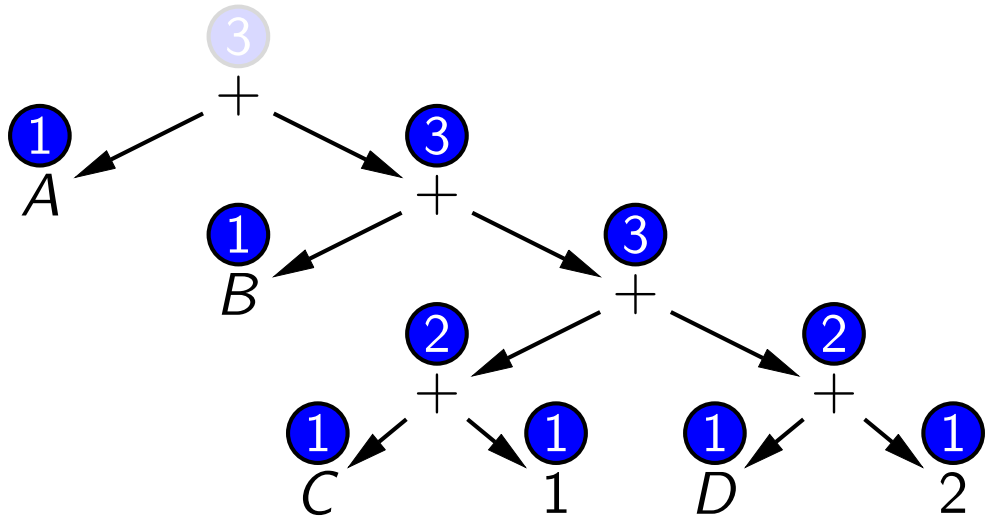
Phase 1 : en utilisant la formule précédente, calculer le nombre minimal de registres nécessaires pour chaque sous-expression de l'expression à compiler.



Phase 2 : produire le code de manière récursive comme précédemment, mais en commençant toujours par la sous-expression nécessitant le plus grand nombre de registres.

Comment économiser les registres ?

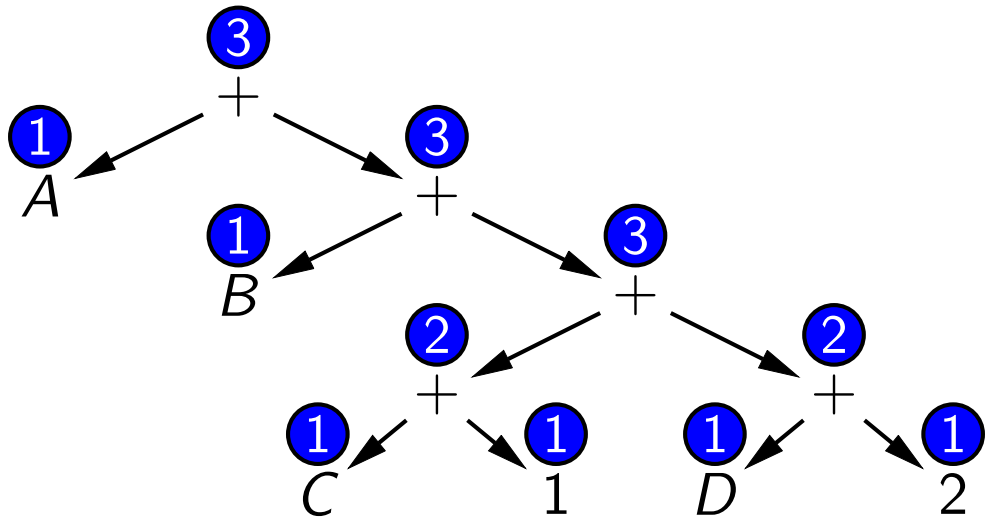
Phase 1 : en utilisant la formule précédente, calculer le nombre minimal de registres nécessaires pour chaque sous-expression de l'expression à compiler.



Phase 2 : produire le code de manière récursive comme précédemment, mais en commençant toujours par la sous-expression nécessitant le plus grand nombre de registres.

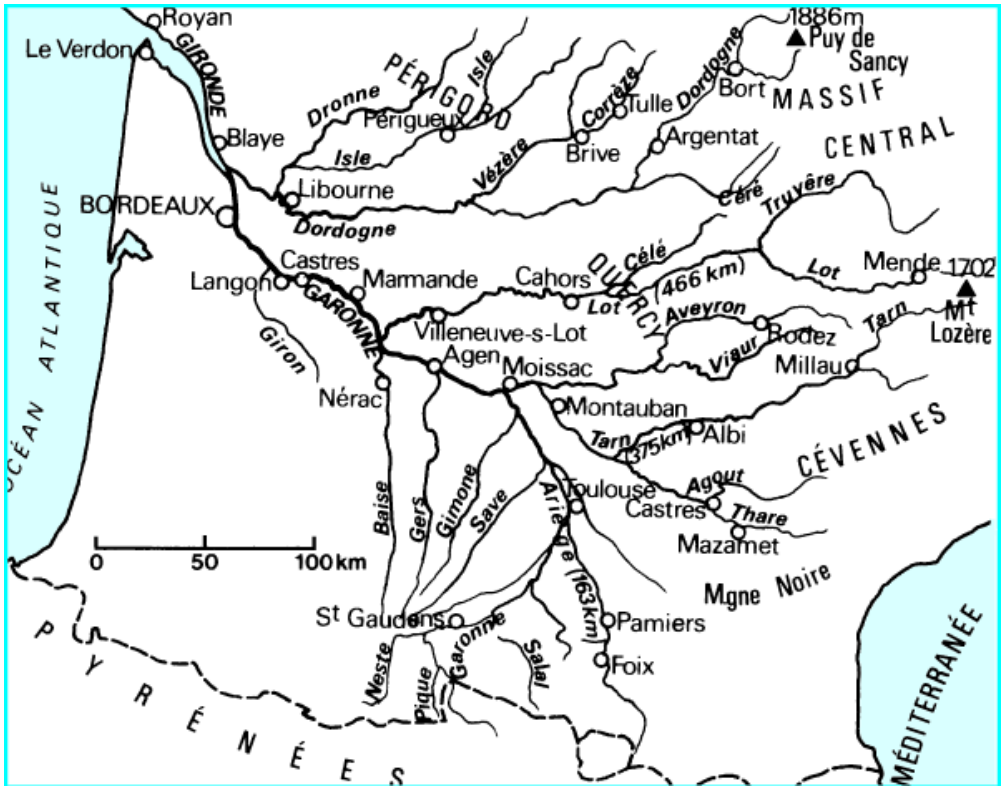
Comment économiser les registres ?

Phase 1 : en utilisant la formule précédente, calculer le nombre minimal de registres nécessaires pour chaque sous-expression de l'expression à compiler.



Phase 2 : produire le code de manière récursive comme précédemment, mais en commençant toujours par la sous-expression nécessitant le plus grand nombre de registres.

Analogie avec un schéma de nommage des rivières



Si une rivière rencontre une plus grosse rivière, la plus grosse conserve son nom. Si deux rivières de même taille se rencontrent, on invente un nouveau nom pour leur confluent.

Compilation et analyse statique

Avec cet algorithme modifié, 4 registres temporaires suffisent à compiler toutes les expressions rencontrées en pratiques.

Comme le montre cet exemple, la production de code de qualité dans un compilateur nécessite très souvent une phase préalable d'analyse statique du programme.

Une analyse statique calcule certaines caractéristiques du programme (p.ex. le nombre de registres nécessaires), qui guident ensuite la production du code.

($\rightarrow$ voir aussi le cours «À la chasse aux bugs»)

Les Lumières (1965) : la récursion

Itération (boucles):

```
r = 1;
for (i = 2; i <= n; i++) {
  r = r * i;
}
```

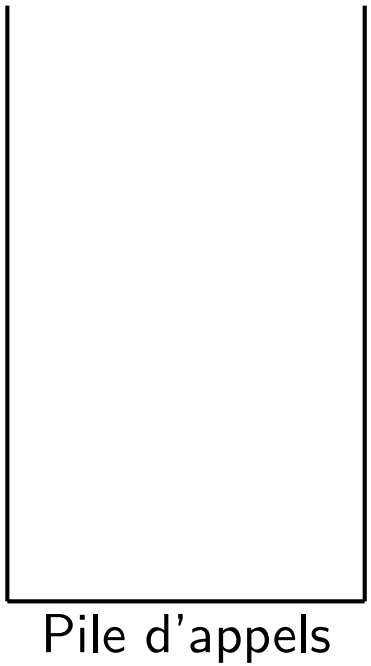
Compilation : facile, à l'aide d'instructions de branchement conditionnels.

Récursion:

```
let rec fact n =
  if n <= 1
  then 1
  else fact(n-1) * n
```

Compilation : plus délicate, utilise une pile d'appels (une structure de données gérée par le compilateur).

La pile d'appels




```
let rec fact n =
  if n <= 1
  then 1
  else fact(n-1) * n
```

Appel de `fact` avec `n=3`

Appel de `fact` avec `n=2`

Appel de `fact` avec `n=1`

Retour avec résultat=1

Calcul de `1 * n` avec `n=2`

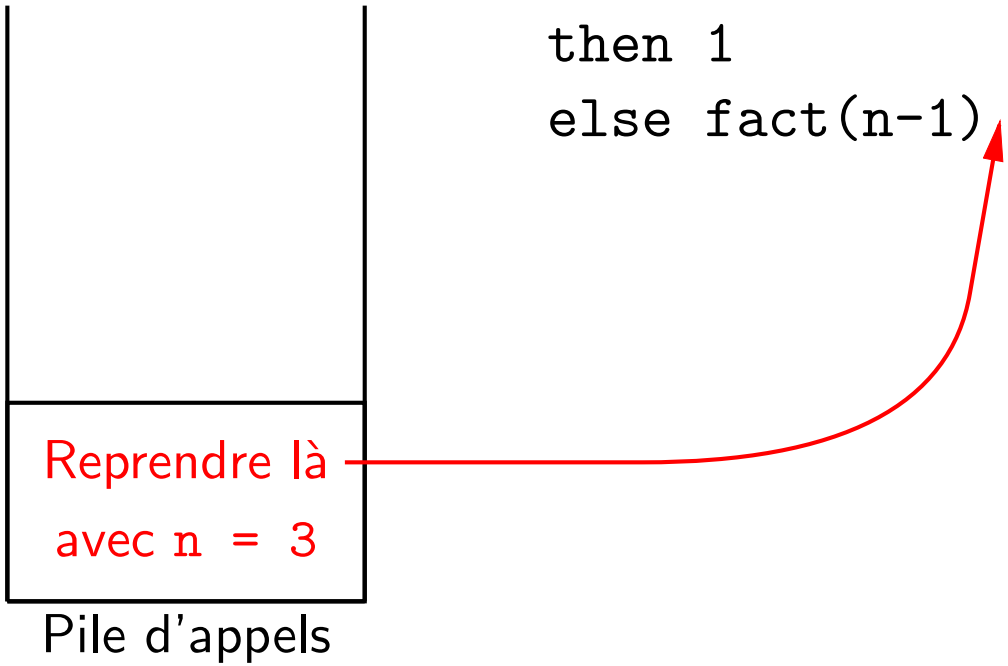
Retour avec résultat=2

Calcul de `2 * n` avec `n=3`

Arrêt avec résultat=6

La pile d'appels

```
let rec fact n =
  if n <= 1
  then 1
  else fact(n-1) * n
```



Appel de `fact` avec `n=3`

Appel de `fact` avec `n=2`

Appel de `fact` avec `n=1`

Retour avec résultat=1

Calcul de `1 * n` avec `n=2`

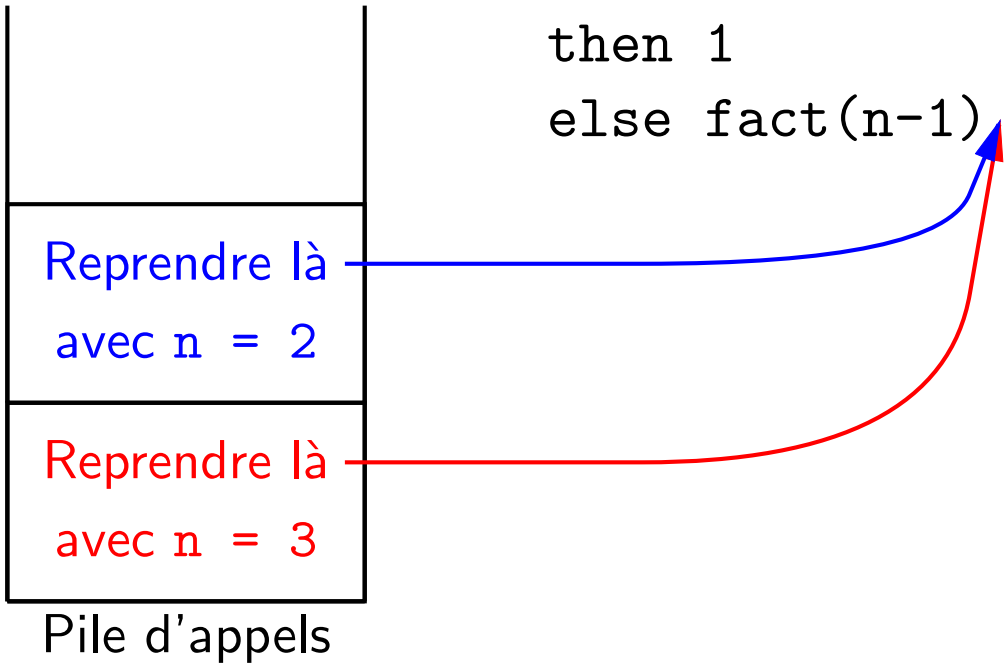
Retour avec résultat=2

Calcul de `2 * n` avec `n=3`

Arrêt avec résultat=6

La pile d'appels

```
let rec fact n =
  if n <= 1
  then 1
  else fact(n-1) * n
```



Appel de `fact` avec `n=3`

Appel de `fact` avec `n=2`

Appel de `fact` avec `n=1`

Retour avec résultat=1

Calcul de `1 * n` avec `n=2`

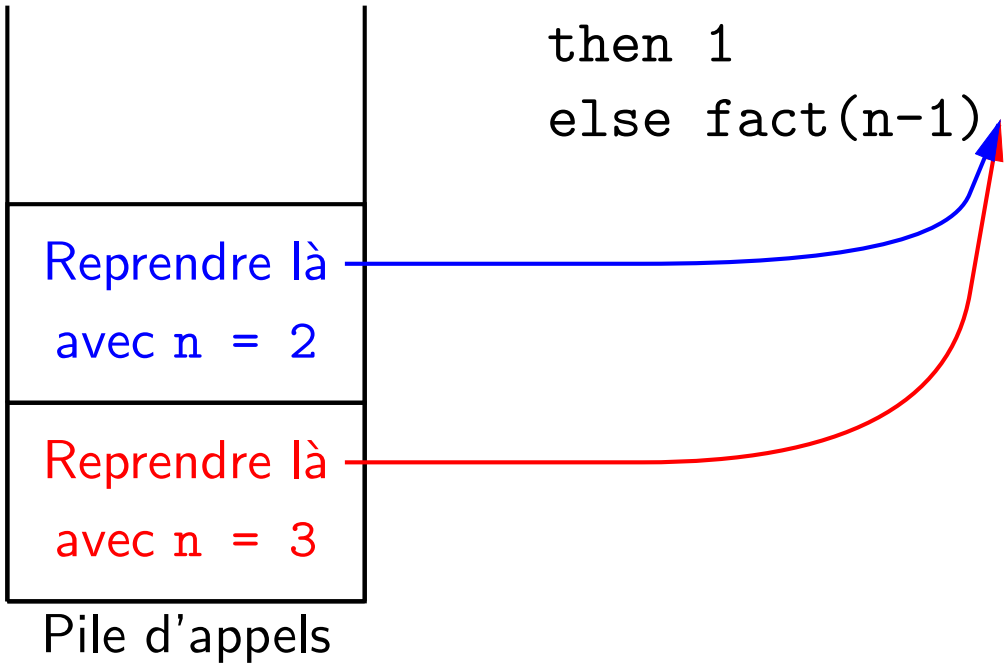
Retour avec résultat=2

Calcul de `2 * n` avec `n=3`

Arrêt avec résultat=6

La pile d'appels

```
let rec fact n =
  if n <= 1
  then 1
  else fact(n-1) * n
```



Appel de `fact` avec `n=3`

Appel de `fact` avec `n=2`

Appel de `fact` avec `n=1`

Retour avec résultat=1

Calcul de `1 * n` avec `n=2`

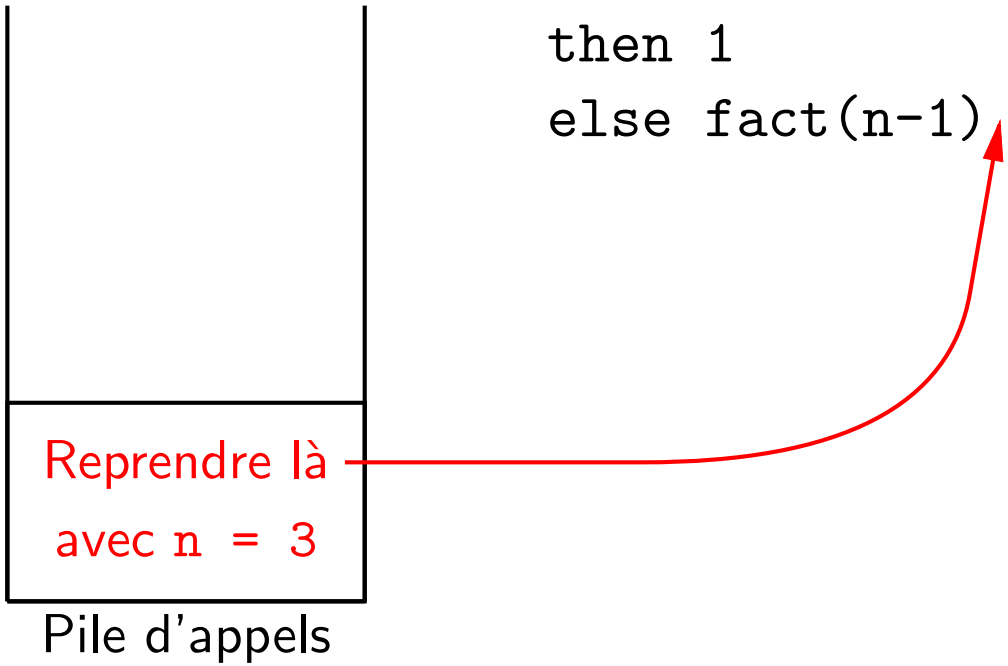
Retour avec résultat=2

Calcul de `2 * n` avec `n=3`

Arrêt avec résultat=6

La pile d'appels

```
let rec fact n =
  if n <= 1
  then 1
  else fact(n-1) * n
```



Appel de `fact` avec `n=3`

Appel de `fact` avec `n=2`

Appel de `fact` avec `n=1`

Retour avec résultat=1

Calcul de `1 * n` avec `n=2`

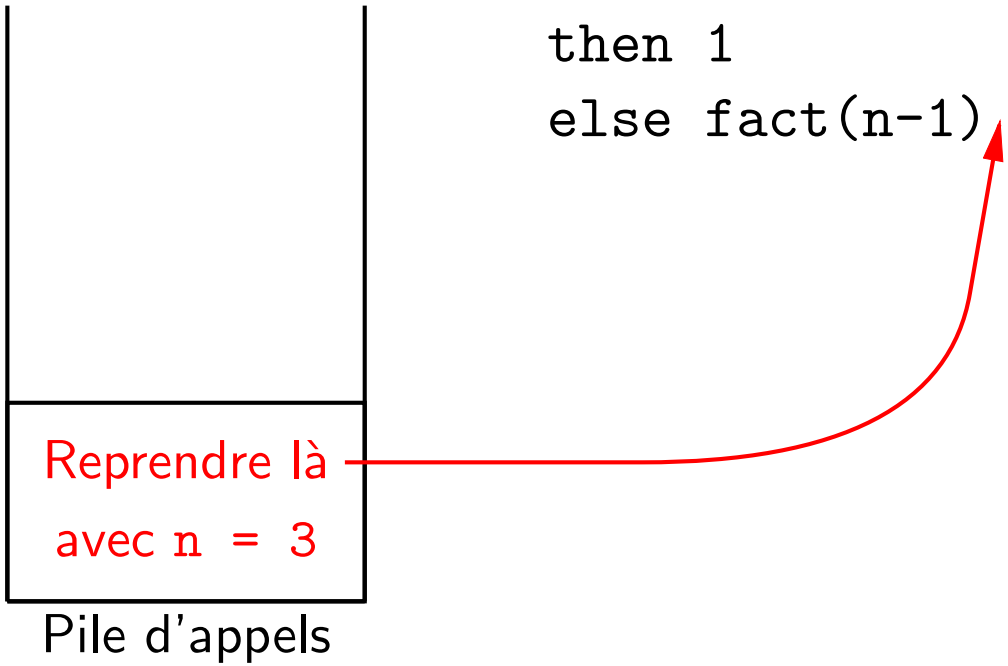
Retour avec résultat=2

Calcul de `2 * n` avec `n=3`

Arrêt avec résultat=6

La pile d'appels

```
let rec fact n =
  if n <= 1
  then 1
  else fact(n-1) * n
```



Appel de `fact` avec `n=3`

Appel de `fact` avec `n=2`

Appel de `fact` avec `n=1`

Retour avec résultat=1

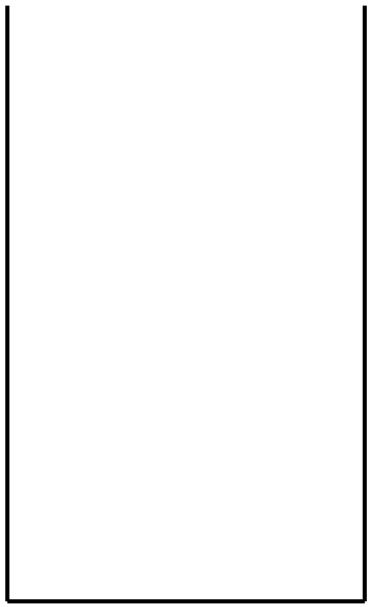
Calcul de `1 * n` avec `n=2`

Retour avec résultat=2

Calcul de `2 * n` avec `n=3`

Arrêt avec résultat=6

La pile d'appels




```
let rec fact n =
  if n <= 1
  then 1
  else fact(n-1) * n
```

Appel de `fact` avec `n=3`

  Appel de `fact` avec `n=2`

    Appel de `fact` avec `n=1`

    Retour avec résultat=1

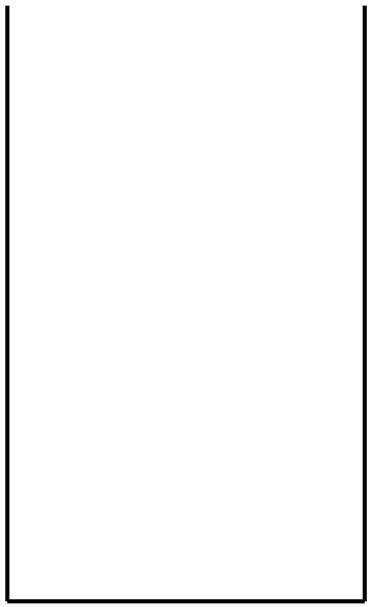
  Calcul de `1 * n` avec `n=2`

  Retour avec résultat=2

Calcul de `2 * n` avec `n=3`

Arrêt avec résultat=6

La pile d'appels




```
let rec fact n =
  if n <= 1
  then 1
  else fact(n-1) * n
```

Appel de `fact` avec `n=3`

  Appel de `fact` avec `n=2`

    Appel de `fact` avec `n=1`

    Retour avec résultat=1

  Calcul de `1 * n` avec `n=2`

  Retour avec résultat=2

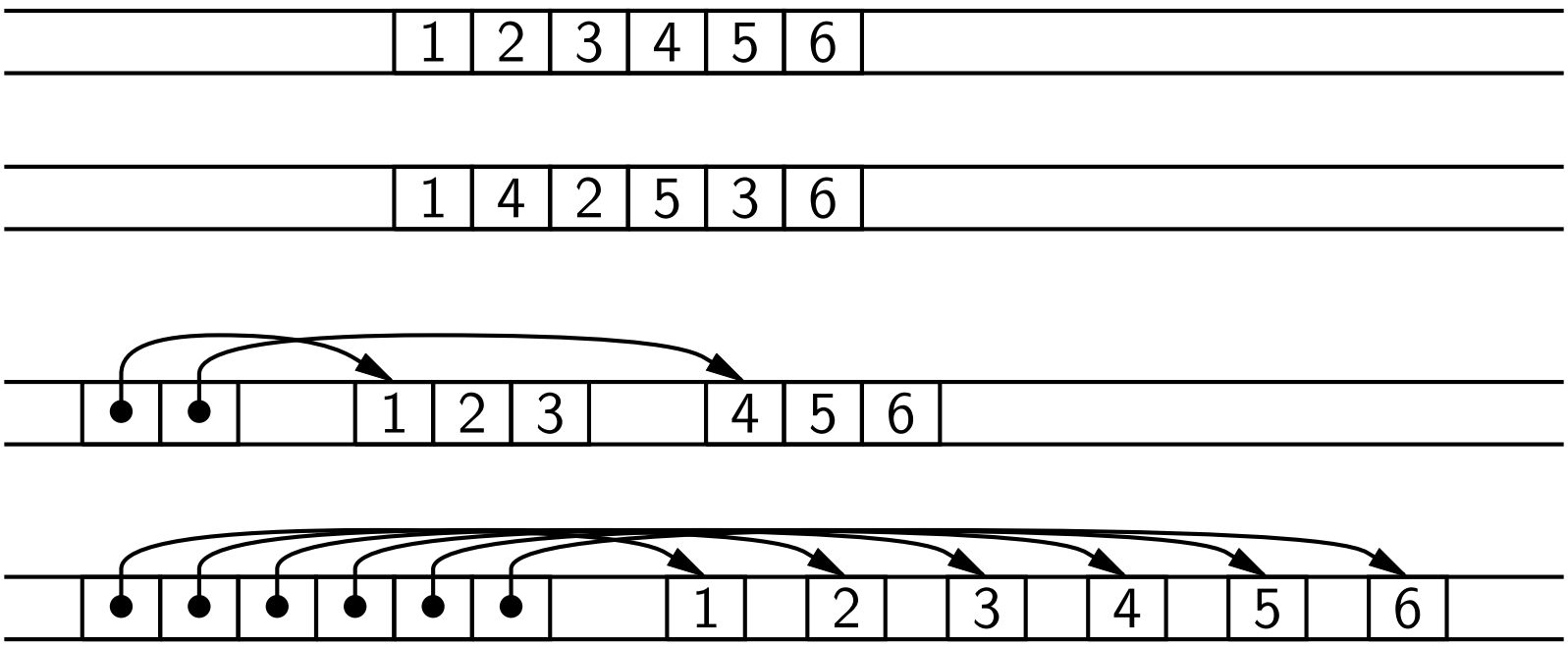
Calcul de `2 * n` avec `n=3`

Arrêt avec résultat=6

Les Temps Modernes (1985) : représentations automatiques des données

Il y a souvent différentes manières de représenter en mémoire une structure de données.

Exemple : la matrice $\begin{pmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \end{pmatrix}$



Les Temps Modernes (1985) : représentations automatiques des données

Langages de bas niveau (assembleur, C, C++, . . .) :
le programmeur choisit les représentations en mémoire, gère les allocations dynamiques, les déréférencements de pointeurs, etc.

```
struct list * x1 = malloc(sizeof(struct list));
struct list * x2 = malloc(sizeof(struct list));
if (x1 == NULL || x2 == NULL) return NULL;
x1->hd = 1; x1->tl = x2; x2->hd = 2; x2->tl = x;
return x1;
```

Langages de haut niveau (langages fonctionnels, langages de scripts) :
le compilateur choisit les représentations en mémoire, gère les allocations et les pointeurs.

```
1 :: 2 :: x
```

Deuxième partie

L'optimisation de code

S'aider de la puissance de la machine (2)

Ce que nous avons vu jusqu'ici est la phase de génération de code : traduire un langage de haut niveau en code machine.

Les compilateurs effectuent également des phases d'optimisation : améliorer la qualité du code produit.

Optimiser le code, pour quoi faire ?

Le plus souvent : pour accélérer les calculs.

- En éliminant des inefficacités laissées par le programmeur.
- En tirant meilleur parti des possibilités du processeur (pipeline, parallélisme, registres, caches sur la mémoire, . . .)

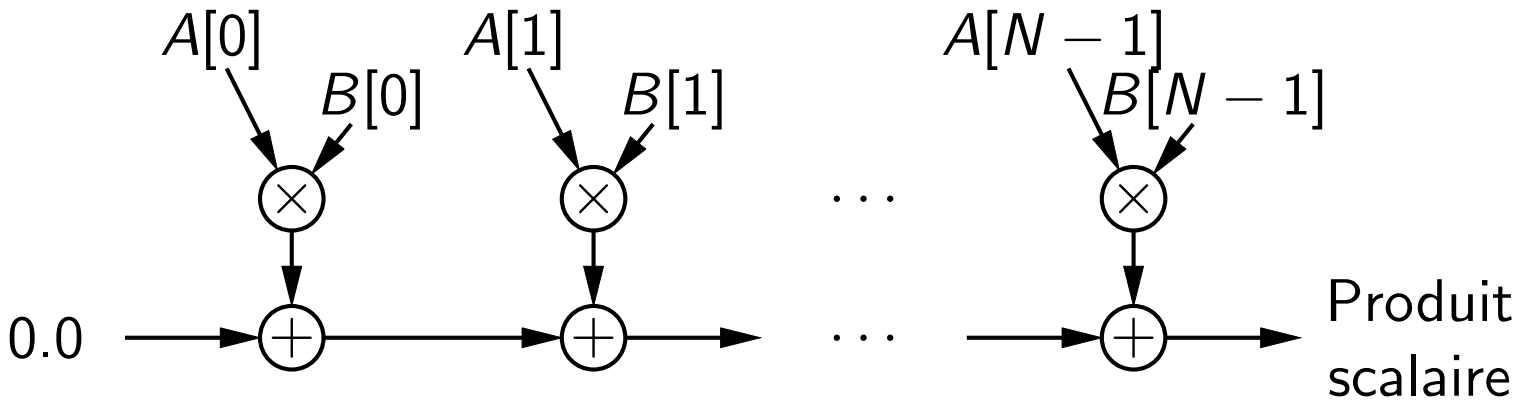
Mais aussi : pour réduire la taille du code machine.
(Cartes à puces et autres systèmes embarqués bon marché.)

Ou encore : pour réduire la consommation électrique.
(Tous les appareils à batteries.)

Un exemple d'optimisation de code

Le calcul du produit scalaire de deux vecteurs `A` et `B`, de taille `N`.

```
double produit_scalaire(int N, double A[], double B[])
{
    int i;
    double p = 0.0;
    for (i = 0; i < N, i++) p = p + A[i] * B[i];
    return p;
}
```



Passage en code intermédiaire

Ne change rien aux performances du code, mais rend explicite les calculs intermédiaires.

```
...
p = 0.0;
for (i = 0; i < N, i++) {
    a = load(A + i * 8);
    b = load(B + i * 8);
    c = a * b;
    p = p + c;
}
...
```

Progression arithmétique

Remplacer le calcul de la suite

$$A + i \times 8, \quad i = 0, 1, \ldots, N - 1$$

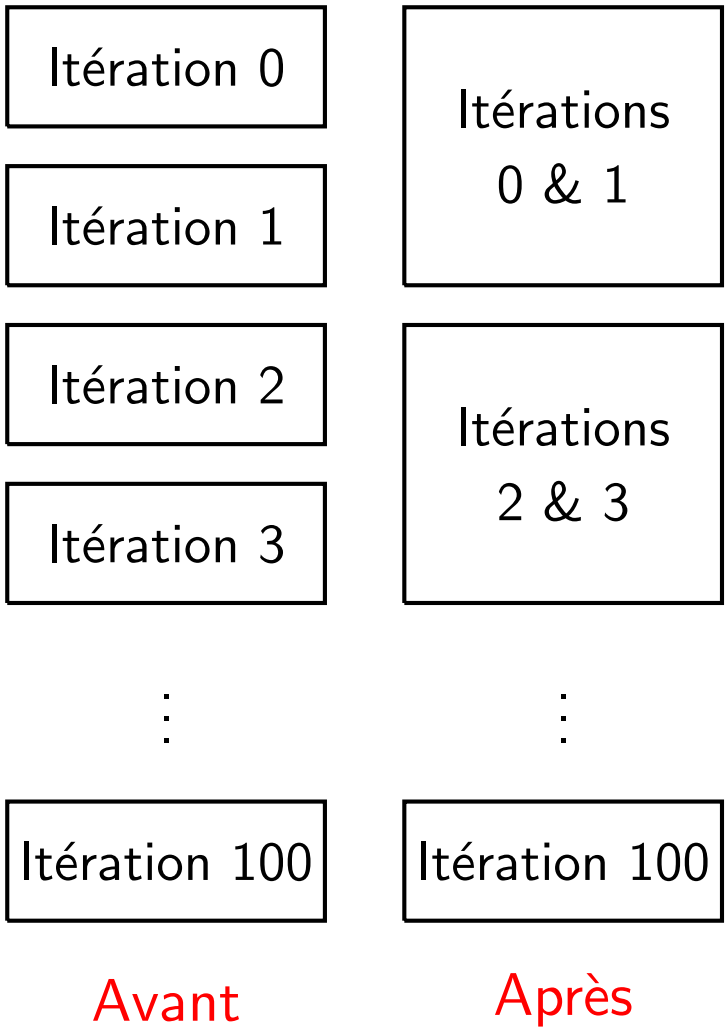
par le calcul de la suite

$$A, A + 8, A + 16, \ldots$$

et de même pour B.

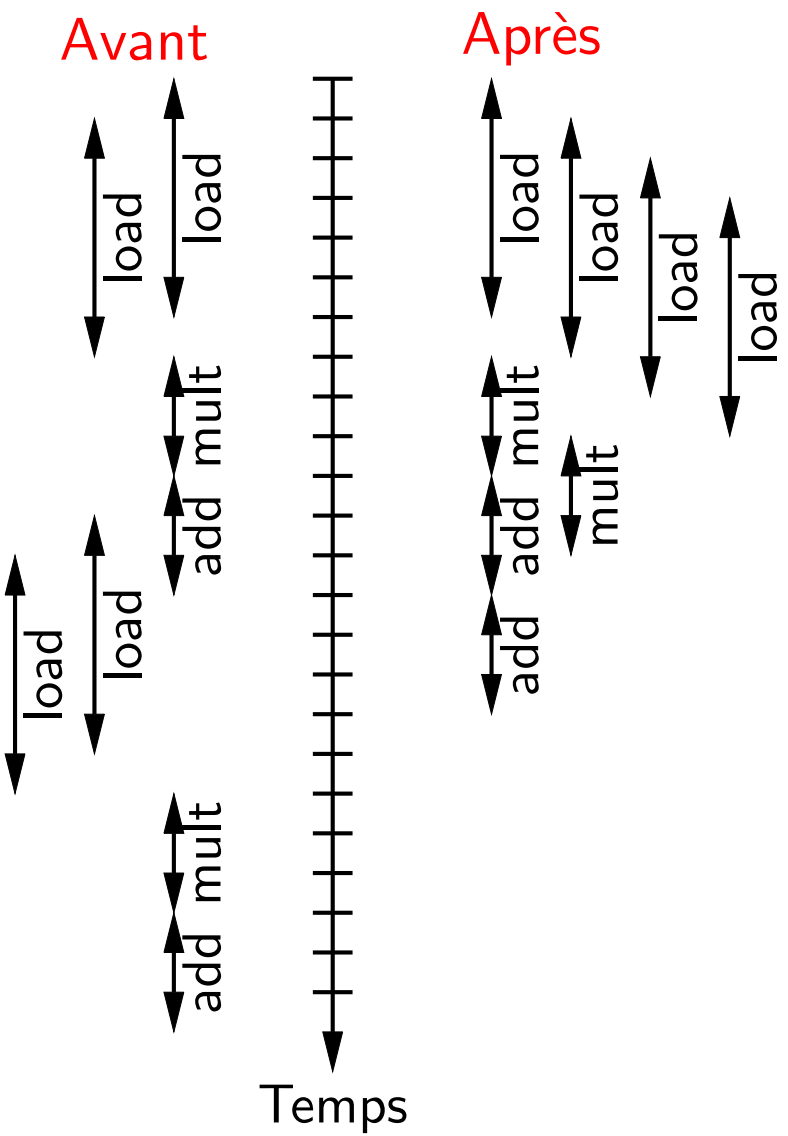
```
...
p = 0.0;
for (i = 0; i < N, i++) {
    a = load(A);
    b = load(B);
    c = a * b;
    p = p + c;
    A = A + 8;
    B = B + 8;
}
...
```

Déroulage de boucle



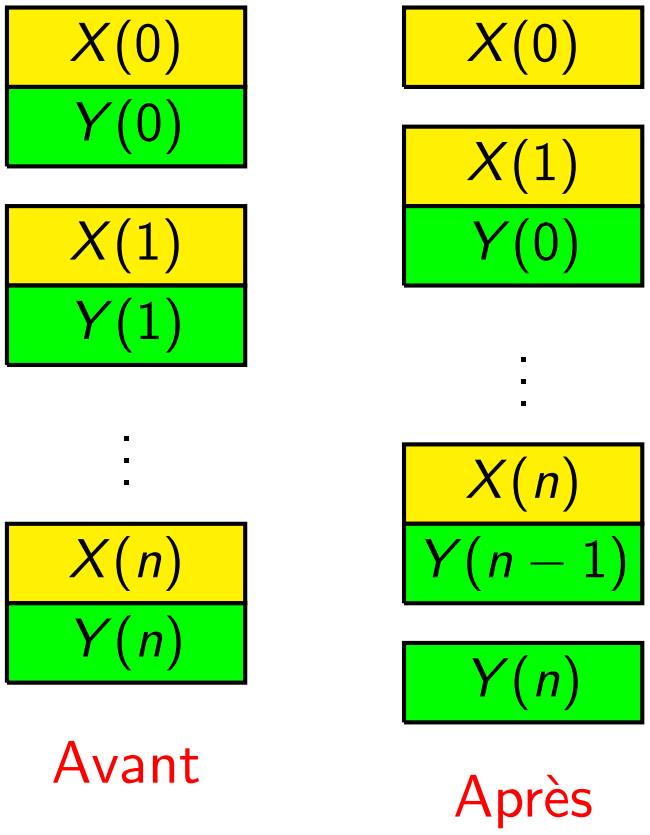

```
for (i=0; i < N-1; i += 2) {
    a1 = load(A);
    b1 = load(B);
    c1 = a1 * b1;
    p = p + c1;
    a2 = load(A + 8);
    b2 = load(B + 8);
    c2 = a2 * b2;
    p = p + c2;
    A = A + 16;
    B = B + 16;
}
if (i < N) {
    a = load(A);
    b = load(B);
    c = a * b;
    p = p + c;
}
```

Réordonnancement des instructions




```
for (i=0; i < N-1; i += 2) {
    a1 = load(A);
    b1 = load(B);
    a2 = load(A + 8);
    b2 = load(B + 8);
    c1 = a1 * b1;
    A = A + 16;
    c2 = a2 * b2;
    p = p + c1;
    B = B + 16;
    p = p + c2;
}
...
```

Pipeline logiciel




```
a3 = load(A); a4 = load(A + 8);
b3 = load(B); b4 = load(B + 8);
for (i=0; i < N-3; i += 2) {
    a1 = a3; a3 = load(A + 16);
    b1 = b3; b3 = load(B + 16);
    c1 = a1 * b1;
    a2 = a4; a4 = load(A + 24);
    b2 = b4; b4 = load(B + 24);
    c2 = a2 * b2;
    p = p + c1;
    A = A + 16;
    B = B + 16;
    p = p + c2;
}
c1 = a3 * b3;
c2 = a4 * b4;
p = p + c1;
p = p + c2;
```

Bilan des optimisations

Un code source tout simple

```
for (i = 0; i < N, i++) p = p + A[i] * B[i];
```

a été transformé en un code intermédiaire fort compliqué

```
i = 0;
if (N >= 4) {
   a3 = load(A); a4 = load(A + 8);
   b3 = load(B); b4 = load(B + 8);
   for (; i < N-3; i += 2) {
       a1 = a3; a3 = load(A + 16);
       b1 = b3; b3 = load(B + 16);
       c1 = a1 * b1;
       a2 = a4; a4 = load(A + 24);
       b2 = b4; b4 = load(B + 24);
       c2 = a2 * b2;
       p = p + c1; A = A + 16;
       B = B + 16; p = p + c2;
   }
       c1 = a3 * b3;
       c2 = a4 * b4;
       p = p + c1; p = p + c2;
       A = A + 16; B = B + 16;
}
for(; i < N; i++) {
  a = load(A); A = A + 8;
  b = load(B); B = B + 8;
  c = a * b;
  p = p + c;
}
```

. . . mais qui s'exécute 2 à 3 fois plus rapidement.

Troisième partie

Le typage

S'aider de la puissance de la machine (3)

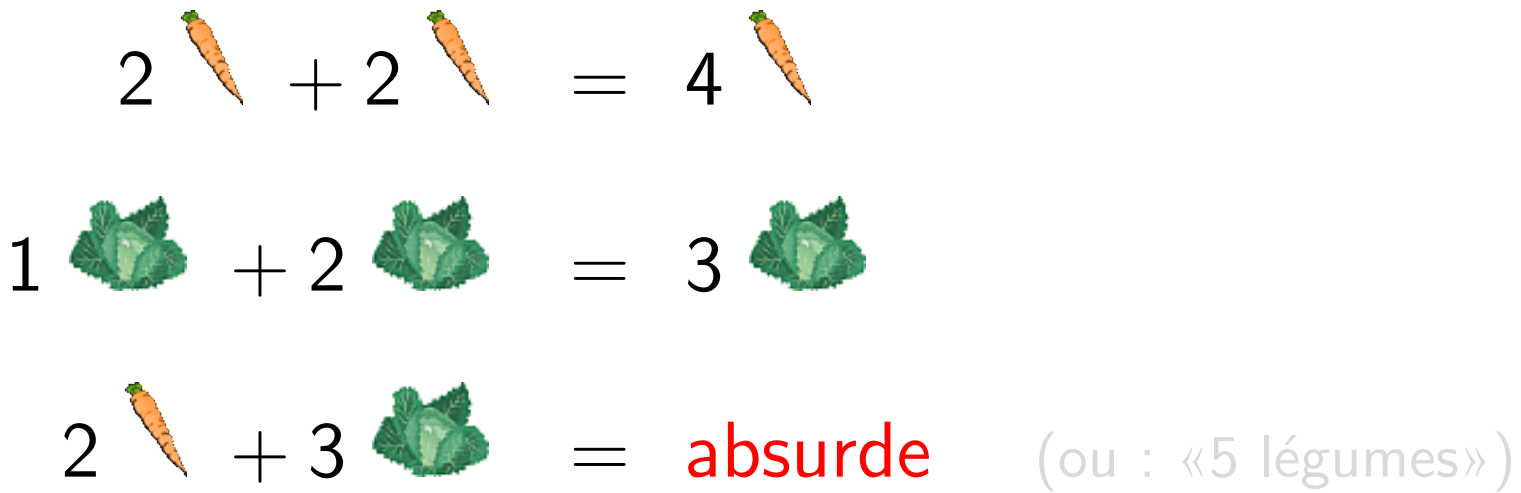
Pour produire du code machine exécutable et en améliorer les performances.

Pour améliorer la fiabilité des programmes en détectant des erreurs de programmation.

($\rightarrow$ le cours «À la chasse aux bugs»).

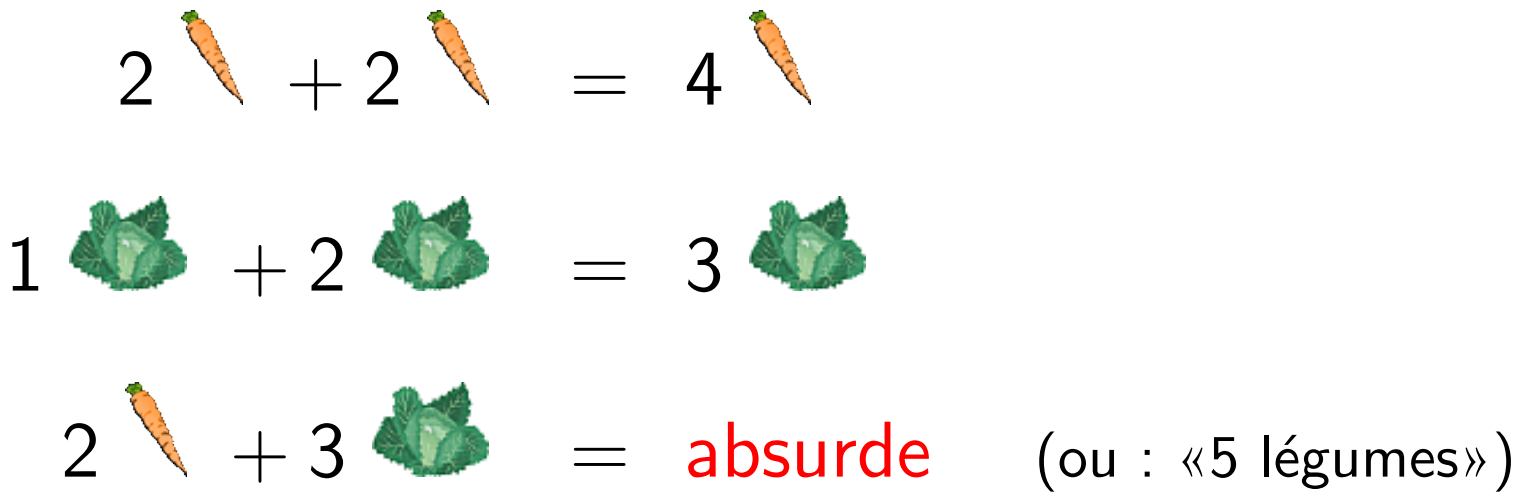
Le typage : une technique pour détecter automatiquement une classe d'erreurs de programmation.

Le typage, à l'école primaire



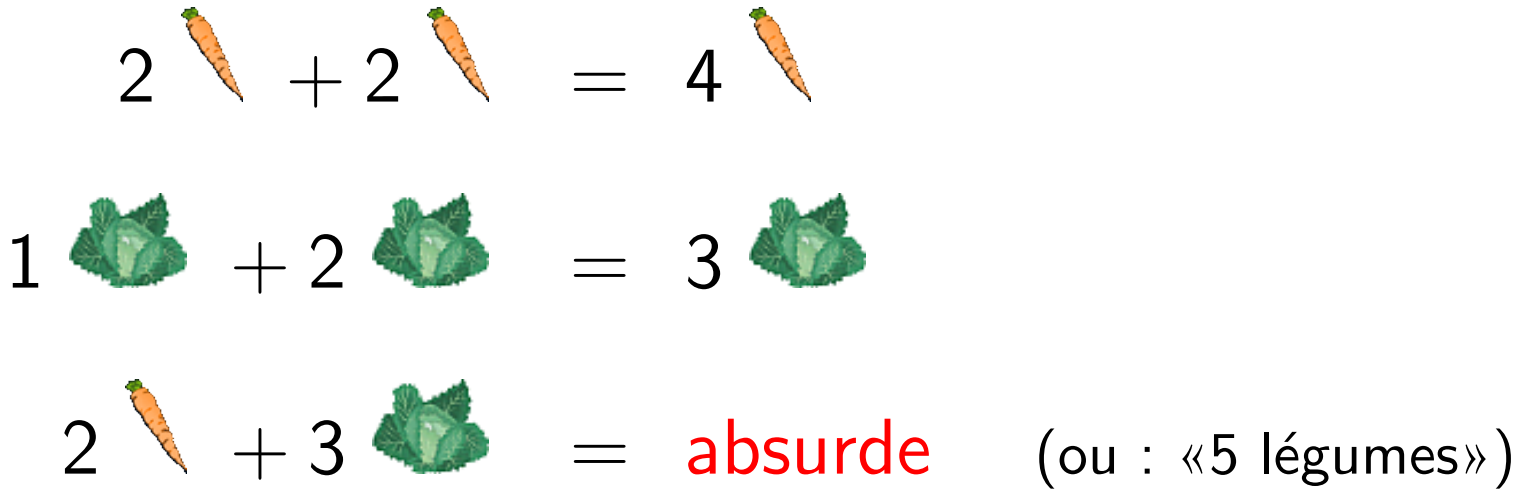
L'addition n'est définie qu'entre deux quantités homogènes (de la même sorte).

Le typage, à l'école primaire



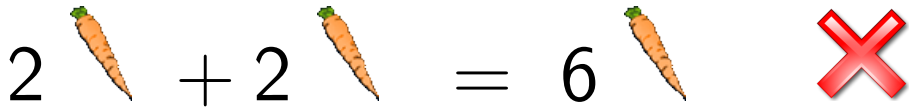
L'addition n'est définie qu'entre deux quantités homogènes (de la même sorte).

Le typage, à l'école primaire

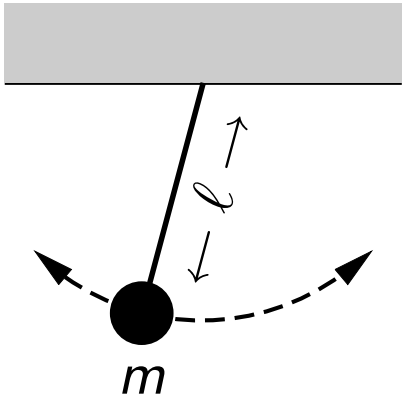


L'addition n'est définie qu'entre deux quantités homogènes (de la même sorte).

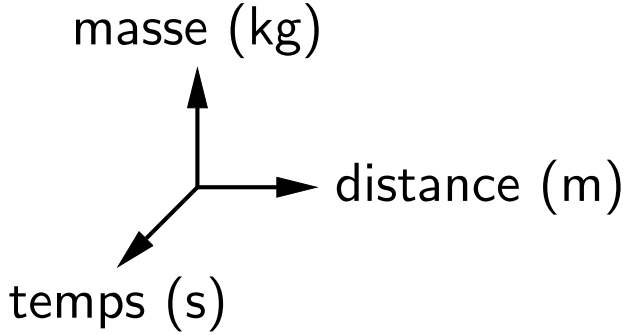
Ce critère n'exclut pas toutes les erreurs de calcul :



Le typage, en physique (équations aux dimensions)



Période d'oscillation T



g : $dist.temps^{-2}$

✗ $T = 2g\ell$

Équation non homogène
donc physiquement absurde

$temps \neq (dist.temps^{-2}).dist$

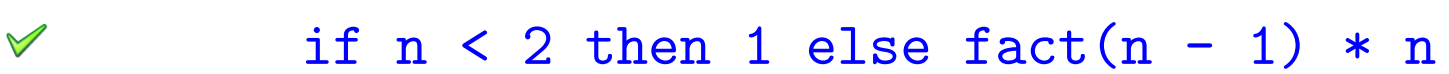
$$T = 2\pi\sqrt{\frac{\ell}{g}}$$

Équation homogène

$$temps = \sqrt{\frac{dist}{dist.temps^{-2}}}$$

Le typage, en programmation

Programme correct et bien typé :



```
let rec fact n =
    if n < 2 then 1 else fact(n - 1) * n
```

Programme mal typé :

```
let rec fact n =
    if n < 2 then 1 else fact(n - "toto") * n
```

(Soustraire la chaîne `"toto"` de l'entier `n` est absurde.)

Programme bien typé mais cependant incorrect :

```
let rec fact n =
    if n < 2 then 1 else fact(n + 1) * n
```

L'algèbre des types

Un type $\approx$ un ensemble de données ayant la même forme.

Types de base :

`int`	les nombres entiers $0, 1, -12, \ldots$
`bool`	les valeurs de vérité (`true` et `false`)
`string`	les chaînes de caractères (textes)

Types composés :

$A \times B$	les paires d'un élément de type A et d'un de type B
$A + B$	ou bien un élément de type A, ou bien un élément de type B
$A \to B$	les fonctions prenant un argument de type A et rendant un résultat de type B
$\texttt{list}(A)$	les listes homogènes dont tous les éléments sont de type A

Exemple : $\texttt{list}(\texttt{int} \times \texttt{string}) \to \texttt{list}(\texttt{int}) \times \texttt{list}(\texttt{string})$.

Les règles de typage

Les constantes $0, 1, 2, \ldots$ *ont le type* `int`.

L'expression $x < y$ *est bien typée seulement si* x *et* y *sont de type* `int` *; le résultat a alors le type* `bool`.

L'expression `if` c `then` x `else` y *est bien typée seulement si* c *est de type* `bool`, *et* x *et* y *sont d'un même type* A *; le résultat a alors le type* A.

Exemples :

- ✔ `if 1 < 2 then 3 else 4`
- ✘ `true < false`
- ✘ `if 1 < 2 then 3 else false`

La notation des règles de typage

Le jugement de typage :

$$\underbrace{x_1 : A_1, \ldots, x_n : A_n}_{\Gamma} \vdash expr : B$$

«En supposant que les variables $x_1, \ldots, x_n$ ont les types $A_1, \ldots, A_n$ respectivement, l'expression *expr* est bien typée et son type est B.»

Les règles de typage :

$$\frac{P_1 \qquad \ldots \qquad P_n}{C}$$

«Si les prémisses $P_1, \ldots P_n$ sont toutes vraies, alors la conclusion C est vraie.»

Un exemple de système de types

$$\frac{}{\Gamma \vdash 0 : \texttt{int}} \qquad \frac{\Gamma(x) = A}{\Gamma \vdash x : A} \qquad \frac{\Gamma \vdash e_1 : \texttt{int} \quad \Gamma \vdash e_2 : \texttt{int}}{\Gamma \vdash (e_1 + e_2) : \texttt{int}}$$

$$\frac{\Gamma \vdash e_1 : \texttt{bool} \quad \Gamma \vdash e_2 : A \quad \Gamma \vdash e_3 : A}{\Gamma \vdash (\texttt{if } e_1 \texttt{ then } e_2 \texttt{ else } e_3) : A}$$

$$\frac{\Gamma \vdash f : A \to B \quad \Gamma \vdash e : A}{\Gamma \vdash f(e) : B} \qquad \frac{x : A,\ \Gamma \vdash e : B}{\Gamma \vdash (\texttt{function } x \to e) : A \to B}$$

Que faire avec un système de types ?

Typage dynamique : pendant l'exécution, chaque opération vérifie les types de ses arguments, et arrête le programme si une erreur de type est détectée.

Avantage : le programme ne va pas continuer avec des données mal formées.
Inconvénient : les erreurs de types sont détectées trop tard.

Typage statique : les types sont vérifiés lors de la compilation du programme, avant toute exécution. Toute erreur de type arrête la compilation.

Avantage : détecte les erreurs très tôt dans le cycle de développement.
Inconvénient : le typage statique peut rejeter à tort des programmes corrects.

Vérification versus inférence de types

Vérification de types :

Le programmeur déclare les types des variables et des fonctions.
Le typeur vérifie la cohérence de ces déclarations.

```
let rec fact (n : int) : int =
    if n < 2 then 1 else fact(n-1) * n
```

Inférence de types :

Le typeur «devine» les types des variables et des fonctions à partir de leurs utilisations dans le programme.

```
let rec fact n =
    if n < 2 then 1 else fact(n-1) * n
```

Infère `n` : `int` et `fact` : `int` $\longrightarrow$ `int`.

Le polymorphisme

Exemple : une fonction de tri, de type $\texttt{list}(A) \to \texttt{list}(A)$ pour un certain type A.

Typage simple (monomorphe) : la fonction est spécialisée à un seul type de liste. Si on veut trier différents types de listes, il faut dupliquer le code de la fonction.

```
let tri_entier (l : list(int)) : list(int) = ...
let tri_chaîne (l : list(string)) : list(string) = ...
```

Typage polymorphe : la fonction peut recevoir un schéma de types de la forme

$$\texttt{tri} : \forall A.\ \texttt{list}(A) \to \texttt{list}(A)$$

puis être utilisée avec plusieurs choix différents du paramètre A :

$\texttt{tri} : \texttt{list}(\texttt{int}) \to \texttt{list}(\texttt{int})$	pour trier une liste d'entiers
$\texttt{tri} : \texttt{list}(\texttt{string}) \to \texttt{list}(\texttt{string})$	pour trier une liste de chaînes

L'abstraction de types

Le dual du polymorphisme $\forall X.T$ est l'abstraction de types $\exists X.T$.

Exemple : un composant logiciel fournissant la structure de données «ensembles finis d'entiers» peut recevoir le type

$$\begin{array}{lll} \exists\, \texttt{ensemble}. & \texttt{ensemble} & \text{l'ensemble vide} \\ & \times\ (\texttt{int} \times \texttt{ensemble} \to \texttt{bool}) & \text{test d'appartenance} \\ & \times\ (\texttt{int} \times \texttt{ensemble} \to \texttt{ensemble}) & \text{ajout} \end{array}$$

La représentation concrète des ensembles (liste triée, arbre binaire de recherche, ...) est cachée au reste du programme : le type `ensemble` se comporte comme un nouveau type de base.

Permet plus facilement de changer *a posteriori* la représentation des ensembles sans «casser» le reste du programme.

Quatrième partie

Liens avec la logique mathématique

Systèmes de types et logique constructive

Langages fonctionnels typés		Logique constructive
Types	$\leftrightarrow$	**Propositions**
$A \times B$		A et B
$A + B$		A ou B
$A \rightarrow B$		A implique B
$\forall A \ldots$		pour tout A, ...
$\exists A \ldots$		il existe A tel que ...
Règles de typage	$\leftrightarrow$	**Règles de déduction logique**
$\dfrac{\Gamma \vdash f : A \rightarrow B \qquad \Gamma \vdash e : A}{\Gamma \vdash f(e) : B}$		$\dfrac{A \text{ implique } B \qquad A}{B}$
Programmes	$\leftrightarrow$	**Preuves**

(Isomorphisme de Curry-Howard.)

Des interactions fructueuses

Langages fonctionnels typés		Logique constructive
Opérateurs de contrôle (`call/cc` en Scheme)	$\rightarrow$	Contenu calculatoire des preuves en logique classique (p.ex. $\forall P.\ P \vee \neg P$)
Combiner typage statique et preuves de programmes	$\leftarrow$	Types dépendants (p.ex. $\{n : \texttt{int} \mid n > 0\}$)

Compilation et preuves

Compilateur certifiant : étant donné un programme et une preuve qu'il est correct, produire du code machine équivalent et une preuve que ce code est correct.

Compilateur vérifié : prouver que le compilateur préserve la correction du programme : toute propriété vraie du programme source est également vraie du code machine produit.

Un peu de programmation éloigne de la logique mathématique ; beaucoup de programmation y ramène.