

# Langages : interprétation et compilation

Pablo Rauzy

pr@up8.edu

[pablo.rauzy.name/teaching/liec](https://pablo.rauzy.name/teaching/liec)



UFR MITSIC / L3 informatique

Séance d

Compilation de langages impératifs

# Compilation de langages impératifs

---

- ▶ On qualifie souvent le langage C d'*assembleur portable*.
- ▶ C'est un peu exagéré, mais c'est vrai qu'on peut considérer C, parmi les langages de haut niveau, comme celui de plus bas niveau.
- ▶ Mais alors, quel est le saut qualitatif qui le distingue de l'assembleur ?

- ▶ Essentiellement, deux choses permettent de qualifier le C de langage de plus haut niveau que l'assembleur :
  - sa syntaxe, plus proche des mathématiques et donc plus naturelle ;
  - la présence de *structures de contrôle*.

- ▶ Une *structure de contrôle* est une instruction qui influe sur le *flot de contrôle* du programme.
- ▶ Le flot de contrôle est l'ordre dans lequel les instructions sont exécutées.
  - Il s'agit donc d'une propriété *dynamique*.
  - Le graphe de flot de contrôle d'un programme, représentant tous les flots de contrôles possibles de ce programme, est lui une propriété a priori *statique*.

- ▶ Les principales structures de contrôles qu'on retrouve dans les langages impératifs sont :
  - les alternatives,
  - les boucles,
  - les fonctions.

- ▶ À variation de syntaxe prêt, une *alternative* a généralement une des formes suivantes :
  - `if (cond) block-then`
  - `if (cond) block-then else block-else`
- ▶ Si on note  $\llbracket p \rrbracket$  la sémantique du programme `p` et `skip` le programme qui ne fait rien :
  - $\llbracket \text{if } (cond) \text{ block-then} \rrbracket = \llbracket \text{if } (cond) \text{ block-then else skip} \rrbracket$
  - $\llbracket \text{if } (cond) \text{ block-then else block-else} \rrbracket = \begin{cases} \llbracket \text{block-else} \rrbracket & \text{si } \llbracket cond \rrbracket = 0 \\ \llbracket \text{block-then} \rrbracket & \text{sinon} \end{cases}$

- ▶ À variation de syntaxe prêt, une *boucle* a généralement une des formes suivantes :
  - `while (cond) block-do`
  - `for (init ; cond ; next) block-do`
- ▶ Toujours avec la même notation pour la sémantique :
  - $\llbracket a ; b \rrbracket = \llbracket a \rrbracket \circ \llbracket b \rrbracket$
  - $\llbracket \{ p \} \rrbracket = \llbracket p \rrbracket$
  - $\llbracket \text{for } (init ; cond ; next) \text{ block-do} \rrbracket = \llbracket init ; \text{while } (cond) \{ \text{block-do} ; next \} \rrbracket$
  - $\llbracket \text{while } (cond) \text{ block-do} \rrbracket = \begin{cases} \llbracket \text{skip} \rrbracket & \text{si } \llbracket cond \rrbracket = 0 \\ \llbracket \text{block-do} ; \text{while } (cond) \text{ block-do} \rrbracket & \text{sinon} \end{cases}$



- ▶ Une fonction est *définie* à un endroit et peut être *appelée* à plusieurs.
- ▶ Pour donner la sémantique des fonctions on va devoir être plus précis :
  - On va avoir besoin d'un environnement, que l'on mettra en indice de notre fonction :  $\llbracket \_ \rrbracket_E$ .
  - $E$  est un ensemble qui contient des associations *nom* : *valeur*.
  - Si  $nom : valeur \in E$  alors  $E[nom] = valeur$ .
- ▶ On va également préciser notre sémantique en lui ajoutant les règles suivantes :
  - $\llbracket v = expr \rrbracket_E \circ \llbracket p \rrbracket_E = \llbracket p \rrbracket_{E \cup \{v: \llbracket expr \rrbracket_E\}}$
  - $\llbracket v \rrbracket_E = E[v]$

- ▶ À variation de syntaxe prêt, une *définition de fonction* est généralement de la forme :
  - `function f (arg1, arg2, ..., argn) block-body`
- ▶ Sémantiquement :
  - $\llbracket \text{function } f (arg_1, arg_2, \dots, arg_n) \text{ block-body} \rrbracket_E = \llbracket f = \text{block-body} \rrbracket_E$

- ▶ À variation de syntaxe prêt, un *appel de fonction* est généralement de la forme :
  - $f(\text{exp}_1, \text{exp}_2, \dots, \text{exp}_n)$
- ▶ Sémantiquement :
  - $\llbracket f(\text{exp}_1, \text{exp}_2, \dots, \text{exp}_n) \rrbracket_E = \llbracket \text{arg}_1 = \text{exp}_1 ; \text{arg}_2 = \text{exp}_2 ; \dots ; \text{arg}_n = \text{exp}_n ; f \rrbracket_E$

- ▶ En assembleur les seules structures de contrôles dont on dispose sont des branchements et des sauts.
- ▶ Sémantiquement, au niveau de l'assembleur, on ne distingue pas les branchements des sauts.
- ▶ Tout ce qui compte c'est que le seul type d'instruction qui influe sur le flot de contrôle permet d'aller directement à une instruction donnée, et que sinon par défaut après toutes les autres instructions on passe à l'instruction suivante dans la mémoire / le code.

- ▶ Les branchements en assembleur sont de deux types :
  - inconditionnels, qui sautent de toutes façons à l'instruction donnée ;
  - conditionnels, qui sautent seulement quand une condition est remplie.
- ▶ En MIPS :
  - inconditionnels : `b`, `j`, `jal`, `jr`, `jalr`, ...
  - conditionnels : `beq`, `beqz`, `bne`, `bgt`, `bge`, ...

- ▶ Compiler, c'est traduire un programme en langage haut niveau vers un programme *équivalent* en langage bas niveau.
- ▶ Cela suppose entre autre d'être capable d'encoder chacune des structures de contrôle haut niveau en assembleur, tout comme on l'avait fait pour le  $\lambda$ -calcul.
- ▶ Si on veut prouver la correction de notre compilateur, il faudrait aussi donner une sémantique à notre langage cible (par exemple l'assembleur), et montrer que la sémantique du programme traduit est la même que celle du programme original.
  - On pourrait par exemple faire en sorte que  $\llbracket \_ \rrbracket_E$  renvoie une fonction qui prend la mémoire dans un certain état et renvoie un nouvel état de la mémoire.
  - Il faudrait alors vérifier que la sémantique de notre programme en assembleur a bien le même comportement (i.e., provoque les mêmes changements en mémoire) que notre programme d'origine.

- ▶ Pour compiler une alternative, il faut :
  - compiler la condition pour obtenir une valeur booléenne,
  - faire un branchement vers le bon bloc d'instructions.

- ▶ Dans un langage haut niveau, une condition est une expression arbitraire qui s'évalue vers une valeur booléenne.
- ▶ Selon les langages les valeurs booléennes ont leur propre type ou sont définies par convention.
  - En C la valeur 0 représente "faux", et tout le reste, "vrai".
  - Dans les langages avec un vrai type booléen, on peut décider de représenter "faux" par 0 en mémoire et "vrai" par 1, pour garder la compatibilité avec le C et se simplifier la tâche.
- ▶ Compiler la condition devrait donc être assez direct.
- ▶ L'idée est simplement de mettre le résultat de l'évaluation toujours au même endroit :
  - soit dans un registre spécifique,
  - soit sur le haut de la pile.



- ▶ Une fois la condition compilée, il faut émettre la bonne instruction de branchement.
- ▶ Soit **C[]** ma fonction de compilation.
- ▶ Si on décide que la compilation des conditions produit toujours leur valeur dans **\$t0** :
  - **C[if (cond) block-then else block-else]**  
 = **C[cond]**  
   beqz \$t0, elseXXX  
   **C[block-then]**  
   b endifXXX  
   elseXXX:  
   **C[block-else]**  
   endifXXX:
- ▶ Il faut bien sûr veiller à ce que **\$t0** ne soit pas utilisé par autre chose en même temps.
  - En MIPS vous avez plein de registre donc il peut-être raisonnable d'en avoir un dédié à ce rôle.
  - Sinon, en règle générale, il faudrait sauvegarder le registre et le restaurer après.

► Les boucles fonctionnent sur le même principe que les alternatives.

- **C[while (cond) block-do]**

```
= loopXXX:
```

```
  C[cond]
```

```
  beqz $t0, endloopXXX
```

```
  C[block-do]
```

```
  b loopXXX
```

```
endloopXXX:
```

- ▶ Pour compiler une fonction, il faut :
  - compiler le code de la définition de la fonction,
  - compiler les appels de la fonction.

- ▶ Quand on écrit une fonction, on ne sait pas où elle va être appelée.
  - Il n'est par exemple pas possible de savoir quels registres seront disponibles au moment de l'exécution de la fonction.
- ▶ Par ailleurs, on ne peut pas deviner comment est implémentée une fonction qu'on appelle.
  - Il est pourtant nécessaire de lui passer ses arguments de la manière dont elle les attend.
- ▶ Il est donc nécessaire de mettre en place des *conventions d'appels* :
  - pour le passage des arguments,
  - pour la sauvegarde des registres.

- ▶ En MIPS, des conventions existent déjà.
- ▶ Pour les arguments :
  - on passe les arguments dans les registres `$a0` à `$a4`,
  - on passe les arguments suivants (si besoin) sur la *pile*,
  - les valeurs de retours sont dans les registres `$v0` et `$v1`.
- ▶ Pour les registres, il y en a deux sortes :
  - ceux sous la responsabilité de l'appelant,
  - ceux sous la responsabilité de l'appelé.

- ▶ Les registres  $\$v0$  et  $\$v1$  ainsi que  $\$a0$  à  $\$a4$ , et  $\$t0$  à  $\$t9$  peuvent être modifiés par les fonctions appelées.
- ▶ Si une fonction a besoin de conserver leur valeur au travers de l'appel d'une autre fonction :
  - elle doit donc les sauvegarder avant l'appel,
  - et les rétablir après pour pouvoir continuer de les utiliser.

- ▶ Les registres `$s0` à `$s7` ainsi que `$gp`, `$sp`, `$fp`, et `$ra` ne doivent pas être modifiés par une fonction appelée.
- ▶ Si une fonction a besoin de les utiliser :
  - elle doit donc les sauvegarder avant,
  - ne pas oublier de les rétablir avant de terminer.

- ▶ La *pile*, c'est la zone de la mémoire où on va pouvoir faire ces sauvegardes.
- ▶ On l'appelle ainsi car on la gère comme une pile :
  - le registre `$sp` (stack pointer) est un pointeur sur le haut de la pile,
  - c'est lui qu'on va incrémenter et décrémenter pour savoir où on en est dans la pile,
  - on accédera aux variables sur la pile via ce pointeur.



- ▶ Attention, la pile grandit vers le bas !
- ▶ Pour empiler la valeur de `$t0`, on fait donc :
  - `addi $sp, $sp, -4`  
`sw $t0, 0($sp)`
- ▶ Si on devait empiler `$t0`, `$t1`, `$t2` :
  - `addi $sp, $sp, -12`  
`sw $t0, 0($sp)`  
`sw $t1, 4($sp)`  
`sw $t2, 8($sp)`

► Pour dépiler, c'est pareil dans l'autre sens :

- `lw $t0, 0($sp)`  
`addi $sp, $sp, 4`

► Ou encore :

- `lw $t0, 0($sp)`  
`lw $t1, 4($sp)`  
`lw $t2, 8($sp)`  
`addi $sp, $sp, 12`

- ▶ Pour appeler une fonction, on doit donc :
  - empiler l'état des registres sous la responsabilité de l'appelant (seulement si on en a l'utilité),
  - appeler la fonction (avec `jal`),
  - rétablir l'état de ces mêmes registres en les dépilant.
  
- ▶ À la définition d'une fonction, on doit donc :
  - empiler l'état des registres sous la responsabilité de l'appelé (seulement ceux qu'on va modifier),
  - implémenter le corps de la fonction,
  - rétablir l'état de ces mêmes registres en les dépilant,
  - retourner à la fonction appelée (avec `jr $ra`).

## ► Compilons à la main le code suivant :

```
1 #include <stdio.h>
2
3 int fact (int n)
4 {
5     if (n == 0 || n == 1) return 1;
6     return n * fact(n - 1);
7 }
8
9 int factloop (int n)
10 {
11     int r = 1;
12     while (n > 0) {
13         r = n * r;
14         n--;
15     }
16     return r;
17 }
18
19 int main ()
20 {
21     int num;
22     printf("n? ");
23     scanf("%d", &num);
24     printf("n! = ");
25     printf("%d", fact(num));
26     printf("\n");
27     return 0;
28 }
```