Compilation (INF 564)

Introduction & architecture MIPS

François Pottier

2 décembre 2015

Qu'est-ce qu'un compilateur ?

Un compilateur *traduit* un langage de haut niveau (adapté à l'esprit humain) vers un langage de bas niveau (conçu pour être exécuté efficacement par une machine).

Aperçu du cours

- En 9 séances, *écrivons un compilateur* d'un langage source modeste (sous-ensemble de Pascal et C) vers un langage assembleur (MIPS).
- *Chaque cours expose une des phases* du compilateur. Le TP qui suit permet d'aborder la pratique devant la machine.

Pourquoi suivre ce cours ?

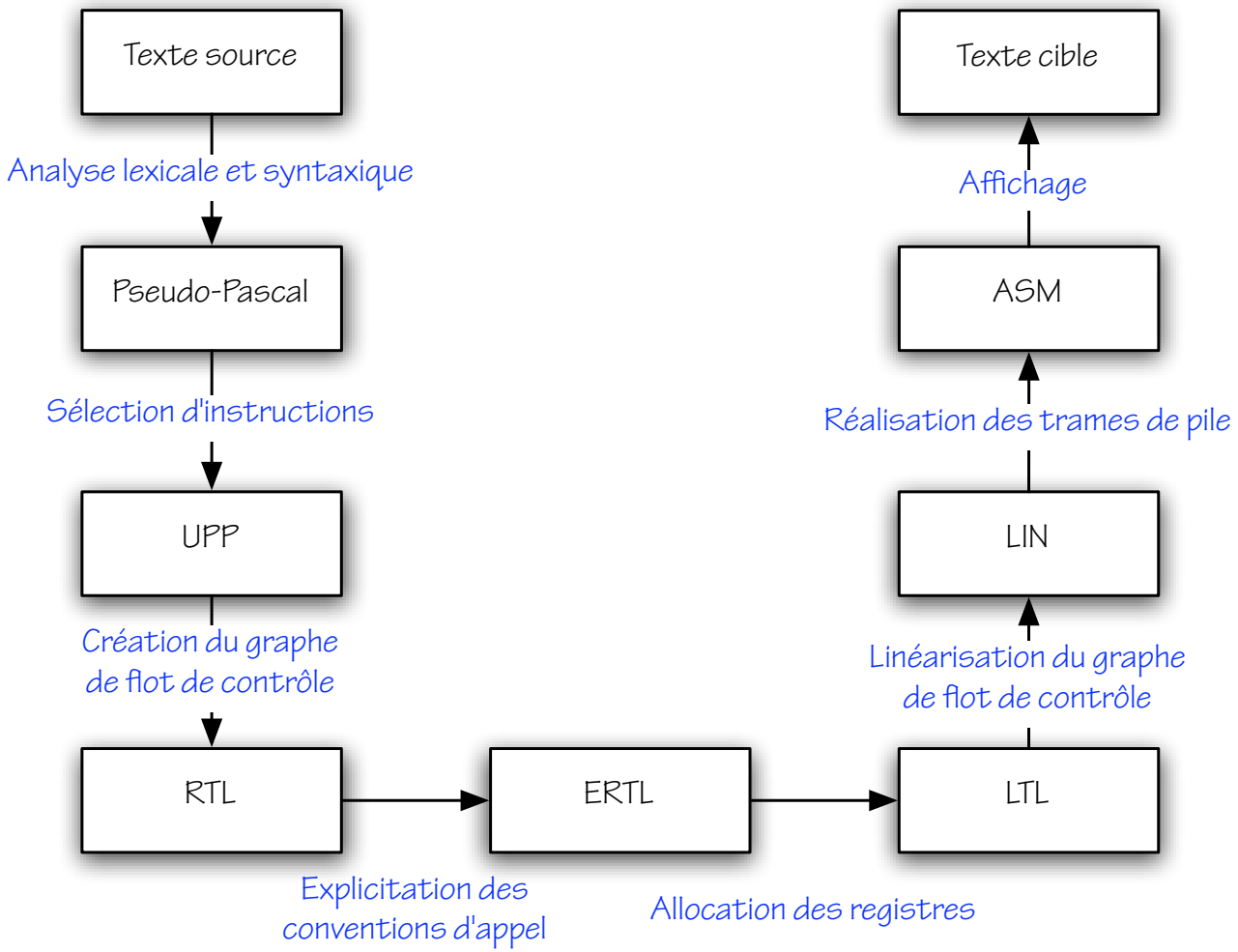
- pour écrire un programme *complexe* et *élégant* dans un langage de très haut niveau (Objective Caml) ;
- pour comprendre le *fossé* entre l'intention humaine et le langage de bas niveau exécuté par le microprocesseur ;
- pour découvrir des *techniques* et *algorithmes* d'usage général : analyse lexicale, analyse syntaxique, transformation d'arbres de syntaxe abstraite, analyse de flot de données, allocation de registres par coloriage de graphe, etc.

Attention !

Un compilateur, ...

- c'est *beau* comme des maths, et en plus ça tourne !
- ça demande du *travail* :
  - on demande de finir chez soi chaque TP ;
  - il n'y a pas de poly : venez en cours !

Un compilateur de Pseudo-Pascal vers MIPS



Plan du cours

1. Architecture *MIPS*.
2. Syntaxe, sémantique et interprétation de *Pseudo-Pascal*.
3. Analyse lexicale et *syntaxique* (du texte source vers PP).
4. Typage. *Sélection d'instructions* (de PP vers UPP).
5. Création du *graphe de flot de contrôle* (de UPP vers RTL).
6. Explicitation de la *convention d'appel* (de RTL vers ERTL).
7. Analyse de *durée de vie* (sur ERTL).
8. *Coloriage de graphe* et allocation des registres (de ERTL vers LTL).
9. *Linéarisation* du code (de LTL vers LIN) puis *réalisation des trames* de pile (de LIN vers ASM).

Présentation du cours

Le processeur MIPS

Programmation du MIPS

Architecture d'un ordinateur

Du point de vue du programmeur ou du compilateur, un ordinateur est constitué principalement d'un *processeur* et d'une *mémoire*.

Le processeur

Le processeur est capable de lire et d'écrire des informations en mémoire. Il dispose également d'un petit nombre d'emplacements mémoire plus rapides, appelés *registres*.

Les *instructions* que doit exécuter le processeur sont elles-mêmes stockées en mémoire. Un registre spécial nommé *pc* contient *l'adresse* de la prochaine instruction à exécuter.

De façon répétée, le processeur lit l'instruction stockée à l'adresse pc (*fetch*), l'analyse (*decode*), puis l'interprète (*execute*), ce qui peut avoir pour effet de modifier certains registres (dont pc) et/ou la mémoire.

CISC et RISC

Les principales différences entre processeurs concernent leur jeu d'instructions.

- Les processeurs CISC (*Complex Instruction Set*)
  - Leurs instructions, de taille variable, sont variées et réalisent souvent des transferts avec la mémoire. Ces processeurs possèdent en général peu de registres, dont certains sont réservés pour des usages spécifiques.
  - Exemples : Intel 8086 et Motorola 68000 (pré-1985).
- Les processeurs RISC (*Reduced Instruction Set*)
  - Leurs instructions, de taille fixe, sont régulières et peu d'entre elles lisent ou écrivent en mémoire. Ces processeurs possèdent en général de nombreux registres, lesquels sont uniformes.
  - Exemples : Alpha, Sparc, Mips, PowerPC, ARM (post-1985).

La mémoire

La mémoire est un grand tableau dont les indices sont les adresses.

Pour des raisons historiques, les adresses sont mesurées en *octets* (8 bits).

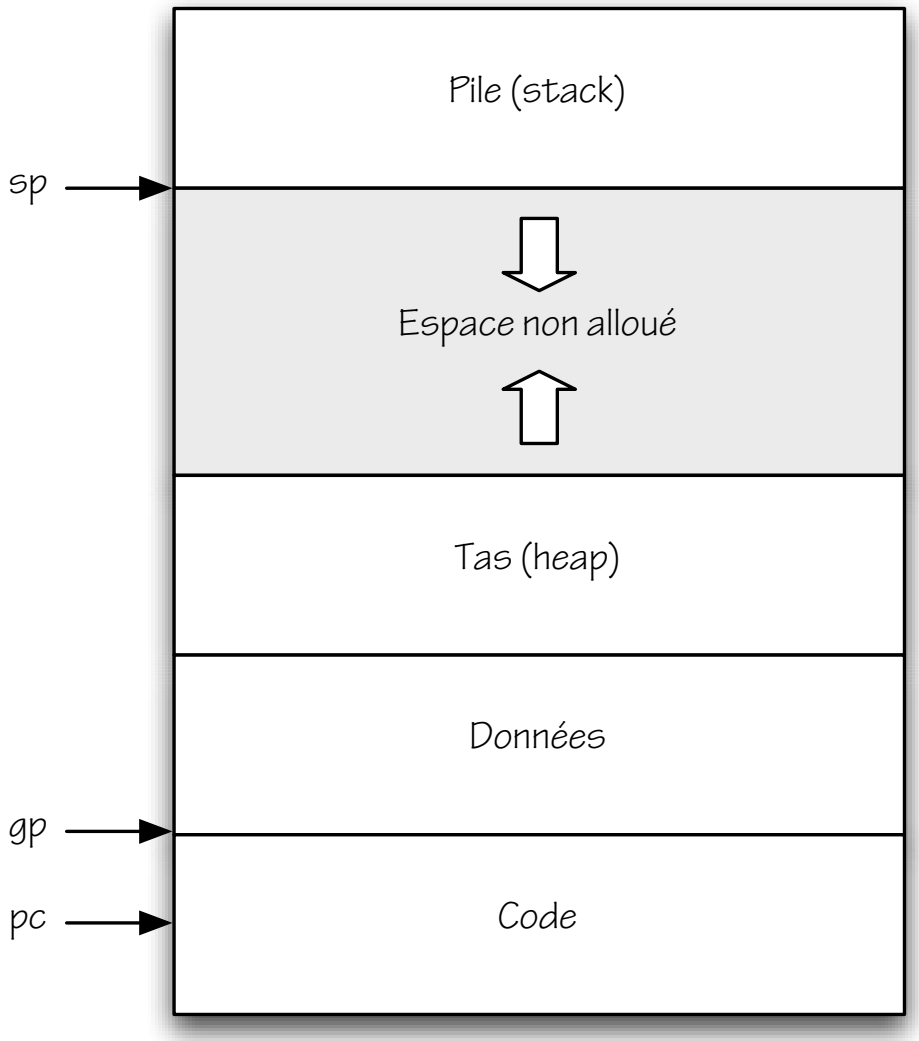
Cependant, les lectures et écritures en mémoire se font sur des quantités de mémoire plus importantes que l'octet. Le *mot* est la taille des données que l'on peut transférer en une instruction : typiquement 32 ou 64 bits, selon les processeurs. Le mot est également la taille des adresses et des registres.

La mémoire virtuelle

Tous les processeurs modernes contiennent une unité de gestion mémoire (MMU) qui distingue adresses *physiques* et adresses *virtuelles*.

La MMU est utilisée par le système d'exploitation pour donner à chaque processus l'illusion qu'il dispose de toute la mémoire.

La mémoire vue depuis un processus



Paramètres de fonctions
Variables locales

Objets alloués dynamiquement

Variables globales

(en lecture seule)

Les registres du MIPS

Le MIPS comporte 32 registres généraux, nommés r0 à r31. Le registre r0, appelé *zero*, contient toujours la valeur 0, même après une écriture. Les 31 autres registres sont *interchangeables*.

Néanmoins, ces 31 registres peuvent être, par *convention* :

- réservés pour le passage d'arguments (*a0-a3*, *ra*) ou le renvoi de résultats (*v0-v1*);
- considérés comme sauvegardés par l'appelé (*s0-s7*) ou non (*t0-t9*) lors des appels de fonctions;
- réservés pour contenir des pointeurs vers la pile (*sp*, *fp*) ou vers les données (*gp*);
- réservés par le noyau (*k0-k1*);
- réservés par l'assembleur (*at*).

Les instructions du MIPS

Le MIPS propose trois types principaux d'instructions :

- les instructions de *transfert* entre registres et mémoire ;
- les instructions de *calcul* ;
- les instructions de *saut*.

Seules les premières permettent d'accéder à la mémoire ; les autres opèrent uniquement sur les registres.

Les instructions de transfert

- *Lecture* (load word) :

```
lw   dest, offset(base)
```

On ajoute la constante (de 16 bits) *offset* à l'adresse contenue dans le registre *base* pour obtenir une nouvelle adresse ; le mot stocké à cette adresse est alors transféré vers le registre *dest*.

Les instructions de transfert

- *Écriture* (store word) :

```
sw   source, offset(base)
```

On ajoute la constante (de 16 bits) *offset* à l'adresse contenue dans le registre *base* pour obtenir une nouvelle adresse ; le mot stocké dans le registre *source* est alors transféré vers cette adresse.

Les instructions de calcul

Ces instructions lisent la valeur de 0, 1 ou 2 registres dits *arguments*, effectuent un calcul, puis écrivent le résultat dans un registre dit *destination*.

Un même registre peut figurer plusieurs fois parmi les arguments et destination.

Les instructions de calcul nullaires

- *Écriture d'une constante* (Load Immediate) :

```
li   dest, constant
```

Produit la constante *constant*.

Les instructions de calcul unaires

- *Addition d'une constante* (Add Immediate) :

```
addi   dest, source, constant
```

Produit la somme de la constante (de 16 bits) *constant* et du contenu du registre *source*.

- *Déplacement* (Move) :

```
move   dest, source
```

Produit le contenu du registre *source*. Cas particulier de **addi** !

- *Négation* (Negate) :

```
neg    dest, source
```

Produit l'opposé du contenu du registre *source*. Cas particulier de **sub** !

Les instructions de calcul binaires

- *Addition* (Add) :

```
add    dest, source1, source2
```

Produit la somme des contenus des registres *source1* et *source2*.

- On a également **sub**, **mul**, **div**.
- *Comparaison* (Set On Less Than) :

```
slt    dest, source1, source2
```

Produit 1 si le contenu du registre *source1* est inférieur à celui du registre *source2* ; produit 0 sinon.

- On a également **sle**, **sgt**, **sge**, **seq**, **sne**.

Les instructions de calcul — variante

Les instructions de calcul précédentes (**add**, **addi**, **sub**, etc.) provoquent une *exception* du processeur (*trap*) en cas de dépassement de capacité (*overflow*).

Une exception provoque un appel de procédure à un *gestionnaire* (*handler*) préalablement installé.

Il existe des variantes dites *non signées* (*unsigned*) de ces instructions (**addu**, **addiu**, **subu**, etc.), qui ne provoquent pas d'exception.

Nous utiliserons ces dernières, aussi bien pour les calculs d'adresses que pour les calculs sur les entiers de Pseudo-Pascal.

Les instructions de saut

On distingue les instructions de saut selon que :

- leurs destinations possibles sont au nombre de 1 (saut *inconditionnel*) ou bien 2 (saut *conditionnel*) ;
- leur adresse de destination est *constante* ou bien *lue* dans un registre ;
- une *adresse de retour* est sauvegardée ou non.

Saut inconditionnel

- *Saut* (Jump) :

```
j       address
```

Saute à l'adresse constante *address*. Celle-ci est en général donnée sous forme symbolique par une *étiquette* que l'assembleur traduira en une constante numérique.

Saut conditionnel

- *Saut conditionnel unaire* (Branch on Greater Than Zero) :

```
bgtz    source, address
```

Si le contenu du registre *source* est supérieur à zéro, saute à l'adresse constante *address*.

- On a également **bgez**, **blez**, **bltz**.
- *Saut conditionnel binaire* (Branch On Equal) :

```
beq     source1, source2, address
```

Si les contenus des registres *source1* et *source2* sont égaux, saute à l'adresse constante *address*.

- On a également **bne**.

Saut avec retour

- *Saut avec retour* (Jump And Link) :

```
jal    address
```

Sauvegarde l'adresse de l'instruction suivante dans le registre *ra*, puis saute à l'adresse constante *address*.

Saut vers adresse variable

- *Saut vers adresse variable* (Jump Register) :

```
jr      target
```

Saute à l'adresse contenue dans le registre *target*.

L'instruction **jr** $ra est typiquement employée pour rendre la main à l'appelant à la fin d'une fonction ou procédure.

Une instruction spéciale

- *Appel système* (System Call) :

  syscall

  Provoque un appel au noyau. Par convention, la nature du service souhaité est spécifiée par un code entier stocké dans le registre *v0*. Des arguments supplémentaires peuvent être passés dans les registres *a0-a3*. Un résultat peut être renvoyé dans le registre *v0*.

spim propose un très petit nombre d'appels système : voir la page A-44 de sa documentation.

Langage machine et langage assembleur

Le *langage assembleur* attribue des noms symboliques aux instructions, aux registres, et aux adresses constantes.

Le *langage machine* est une suite de mots, qui peuvent représenter des données ou des instructions.

L'assembleur (pour nous, spim ou mars) est un programme qui traduit le langage assembleur en langage machine.

Instructions et pseudo-instructions

Certaines des instructions précédentes ne sont en fait pas implantées par le processeur, mais *traduites* par l'assembleur en séquences d'instructions plus simples.

Par exemple,

```
blt  $t0, $t1, address
```

est expansée en

```
slt  $at, $t0, $t1
bne  $at, $zero, address
```

Instructions et pseudo-instructions

Ou encore,

```
li   $t0, 400020
```

est expansée en

```
lui  $at, 6
ori  $t0, $at, 6804
```

Pour nous, la distinction entre instructions et pseudo-instructions n'aura pas d'importance.

Exemple de programme assembleur

```
    .data                       # Des données suivent.
hello:                          # L'adresse de la donnée qui suit.
    .asciiz "hello world\n"     # Chaîne terminée par un zéro.
    .text                       # Des instructions suivent.
main:                           # Adresse de l'instruction qui suit.
    li   $v0, 4                 # Code de l'appel print_string.
    la   $a0, hello             # Adresse de la chaîne.
    syscall                     # Appel au noyau.
    jr   $ra                    # Fin de la procédure main.
```

Exécution d'un programme par spim

Le programme est assemblé puis exécuté à l'aide de la commande :

```
spim -file hello.spi
```

Par convention, l'exécution débute au label *main* et termine lorsque cette procédure rend la main.

spim est donc à la fois un *assembleur* et un *simulateur*.

Exécution d'un programme par spim

En réalité, l'exécution débute au label *__start* et termine lorsque le programme effectue l'appel système numéro 10 ("exit").

Cela ne se voit pas parce que spim charge aussi un morceau de code qui ressemble à ceci :

```
__start:
        jal   main
        li    $v0, 10
        syscall       # Stoppe l'exécution du processus.
```

Si on donne à spim l'option -notrap, il ne charge pas ce code.

Exécution d'un programme par mars

mars se comporte comme spim -notrap. On écrit donc :

```
mars hello.spi
```

et l'exécution débute au label *__start*.

mars est aussi un *éditeur* intelligent. Tapez simplement "mars" et écrivez votre programme. Vous pourrez l'assembler et l'exécuter sans quitter l'interface graphique.

Haut niveau et bas niveau

Des constructions qui peuvent sembler élémentaires dans un langage tel que Pascal (*tests*, *boucles*, *fonctions*) seront traduites par de nombreuses instructions assembleur.

Par ailleurs, des notions d'apparence pourtant simple n'ont pas d'équivalent. Par exemple, les *variables* de Pascal seront traduites en termes d'emplacements mémoire et de registres — une tâche non triviale.

Tests

Le fragment de Pascal suivant :

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

peut être traduit en assembleur MIPS comme ceci :

```
blt  $t1, $t2, then     # si t1 < t2 saut vers l'étiquette then
move $t3, $t2           # t3 := t2
j    done               # saut vers l'étiquette done
then:
move $t3, $t1           # t3 := t1
done:                   # suite du programme
```

Boucles

Le fragment de Pascal suivant :

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

peut être traduit en assembleur MIPS comme ceci :

```
li   $t2, 0          # t2 := 0
while:               # début de la boucle
blez $t1, done       # si t1 <= 0 saut vers l'étiquette done
add  $t2, $t2, $t1   # t2 := t2 + t1
sub  $t1, $t1, 1     # t1 := t1 - 1
j    while           # retour vers l'étiquette while
done:                # suite du programme
```

Fonctions

La fonction Pascal suivante :

```
function succ (x : integer) : integer;
begin
    succ := x + 1
end;
```

peut être traduite en assembleur MIPS comme ceci :

```
succ:                   # $a0 contient l'argument x
addi $v0, $a0, 1        # $v0 contient le résultat
jr   $ra                # retour à l'appelant
```

Le choix des registres est imposé par la *convention d'appel*.

Convention d'appel

La convention d'appel régit la *communication* entre appelant et appelé lors de l'appel de procédure ou de fonction. Pour le MIPS, la convention *proposée* par le fabricant est :

- les arguments sont passés dans *a0-a3* puis (s'il y en a plus de quatre) sur la pile ;
- l'adresse de retour est passée dans *ra* ;
- la valeur de retour est renvoyée dans *v0* ;
- les registres mentionnés ci-dessus, ainsi que *t0-t9*, *peuvent être modifiés* par l'appelé ; l'appelant doit donc les sauvegarder si nécessaire ; on les appelle *caller-save* ;
- les registres *s0-s7 doivent être préservés* par l'appelé ; on les appelle *callee-save*.

Fonctions récursives

Plus difficile : voici une fonction Pascal récursive.

```
function f (n : integer) : integer;
begin
    if n <= 0 then
        f := 1
    else
        f := n * f (n - 1)
end;
```

Fonctions récursives

Voici une traduction naïve et *incorrecte* en assembleur MIPS :

```
fact:
blez $a0, fact0          # si a0 <= 0 saut vers l'étiquette fact0
sub  $a0, $a0, 1         # a0 := a0 - 1
jal  fact                # appel récursif; lit a0 et écrit dans v0
mul  $v0, $a0, $v0       # v0 := a0 * v0
jr   $ra                 # retour à l'appelant
fact0:
li   $v0, 1              # v0 := 1
jr   $ra                 # retour à l'appelant
```

Quelle est l'erreur ?

Fonctions récursives

Parce que *fact* est récursive, plusieurs appels imbriqués peuvent être actifs simultanément :

$$\text{fact} \begin{cases} a_0 \leftarrow a_0 - 1 \\ \text{fact} \begin{cases} a_0 \leftarrow a_0 - 1 \\ \ldots \\ v_0 \leftarrow a_0 \times v_0 \end{cases} \\ v_0 \leftarrow a_0 \times v_0 \end{cases}$$

L'appel récursif *modifie* a0 — en fait, il en détruit le contenu. Or, celui-ci est utilisé *après* l'appel.

Il faut donc *sauvegarder* le contenu de a0 avant l'appel, puis le *restaurer* après l'appel. (Et de même pour *ra*.)

La pile

Par convention, la pile grandit dans le sens des adresses décroissantes. Le registre sp pointe vers le *sommet* de la pile, c'est-à-dire vers le dernier mot alloué.

On *sauvegarde* le contenu de a0 sur la pile comme ceci :

```
subu $sp, $sp, 4        # alloue un mot sur la pile
sw   $a0, 0($sp)        # copie $a0 vers le sommet de la pile
```

On *restaure* le contenu de a0 depuis la pile comme cela :

```
lw   $a0, 0($sp)        # copie le sommet de la pile vers $a0
addu $sp, $sp, 4        # désalloue un mot sur la pile
```

La version assembleur

Voici le code produit par une version de notre compilateur :

```
f17:
addiu $sp, $sp, -8
sw    $ra, 4($sp)
sw    $s0, 0($sp)
move  $s0, $a0
blez  $s0, f4
addiu $a0, $s0, -1
jal   f17
mul   $v0, $s0, $v0
f28:
lw    $ra, 4($sp)
lw    $s0, 0($sp)
addiu $sp, $sp, 8
jr    $ra
f4:
li    $v0, 1
j     f28
```

Exercice recommandé : expliquez chaque instruction.