

# Compilation (INF 564)

Architecture du compilateur  
Sélection d'instructions: de PP à UPP

François Pottier

26 janvier 2011

Architecture d'un compilateur

Architecture de notre petit compilateur

Sélection d'instructions : de PP à UPP

# Un fossé à franchir

Pseudo-Pascal	MIPS
opérateurs <i>ordinaires</i>	opérateurs <i>ad hoc</i> (k+), ( $\ll$ k), ...
expressions <i>structurées</i>	instructions <i>élémentaires</i>
instructions <i>structurées</i>	<i>branchements</i>
pile <i>implicite</i>	pile <i>explicite</i>
variables en nombre <i>illimité</i>	registres en nombre <i>fini</i>

Définir d'un seul jet une traduction de bonne qualité de Pseudo-Pascal vers MIPS est virtuellement *impossible*.

## Organisation en phases

Pour franchir un torrent, on saute de *roche* en *roche*...

De même, le compilateur est découpé en une série de *phases*. Chaque phase traduit le programme d'un *langage intermédiaire* vers un autre.

Chaque langage intermédiaire est *proche* du langage intermédiaire précédent et n'en diffère qu'en un *petit nombre* d'aspects.

# Indépendance des phases

Chaque langage intermédiaire dispose de sa propre *syntaxe abstraite* et (en principe) de sa propre *sémantique*.

La *spécification* de chaque phase est donc limpide : étant donné un programme exprimé dans le langage intermédiaire  $L_k$ , elle produit un programme exprimé dans le langage intermédiaire  $L_{k+1}$  dont la sémantique est *équivalente*.

Les phases du compilateur ne communiquent par aucun autre moyen (pas de tables de symboles globales, ...). Chaque phase est une fonction *pure*.

## Influence des langages intermédiaires

Franchir un torrent devient plus aisé si on sait où est le gué...

Le *choix* des différents langages intermédiaires constitue en fait la clef de la *conception* du compilateur.

Une fois la spécification de chaque phase fixée, *réaliser* celle-ci devient souvent un *exercice* relativement simple. Toutefois, certaines phases complexes seront elles-mêmes subdivisées en plusieurs étapes.

Architecture d'un compilateur

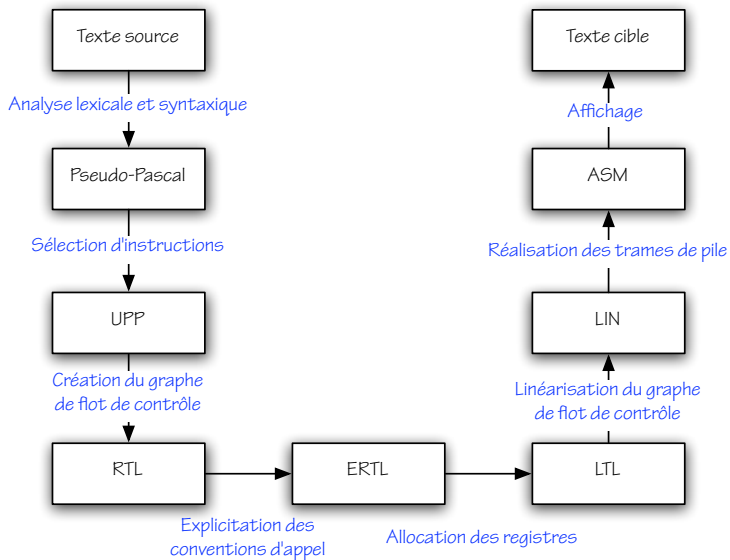
Architecture de notre petit compilateur

Sélection d'instructions : de PP à UPP

# Organisation en phases

Voici à présent l'organisation de notre *petit* compilateur. Celle-ci est inspirée par une réalisation de plus grande ampleur due à Xavier Leroy.

Nous avons *7 langages intermédiaires*, abrégés PP, UPP, RTL, ERTL, LTL, LIN, et ASM, ce dernier étant un sous-ensemble de l'assembleur MIPS.



# Pseudo-Pascal (PP)

Voici le texte Pseudo-Pascal de la fonction factorielle :

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

Nous allons illustrer *une partie* des différences entre langages intermédiaires en étudiant son évolution à travers les phases.

# Untyped Pseudo-Pascal (UPP)

Dans UPP,

- ▶ les *types* sont supprimés, après vérification;
- ▶ variables globales et locales sont *distinguées*; les variables globales sont désignées par leur *adresse*;
- ▶ l'initialisation à zéro des variables locales est *explicitée*;
- ▶ les *opérateurs d'accès aux tableaux* de PP sont remplacés par les opérations **lw** et **sw** du MIPS;
- ▶ les *opérateurs arithmétiques* de PP sont remplacés par ceux du MIPS.

Les deux derniers points constituent la *sélection d'instructions*.

# Untyped Pseudo-Pascal (UPP)

Voici une traduction de la fonction factorielle dans UPP :

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

Le *type* du paramètre n a été oublié. Un opérateur *d'addition unaire* est utilisé pour le décrémenter.

La syntaxe *concrète* employée ici et dans ce qui suit est sans grande importance.

# Register Transfer Language (RTL)

Dans RTL,

- ▶ expressions et instructions structurées sont *décomposées* en *instructions élémentaires* organisées en *graphe de flot de contrôle*;
- ▶ les variables locales sont remplacées par des *pseudo-registres* dont on dispose en nombre illimité.

L'organisation en graphe est destinée à faciliter certaines transformations ultérieures en permettant *l'insertion* ou la *suppression* d'une instruction individuelle.

# Register Transfer Language (RTL)

Voici une traduction de la fonction factorielle dans RTL :

```
function f(%0) : %1
var %0, %1, %2, %3
entry f6
exit f0
f6: li    %1, 0      → f5
f5: blez %0         → f4, f3
f3: addiu %3, %0, -1 → f2
f2: call %2, f(%3) → f1
f1: mul  %1, %0, %2 → f0
f4: li  %1, 1      → f0
```

*Paramètre*, *résultat*, *variables locales* sont des pseudo-registres. Le graphe est donné par ses *labels d'entrée* et de *sortie* et par une table qui à chaque label associe une instruction. Chaque instruction mentionne explicitement le ou les labels de ses *successeurs*.

# Explicit Register Transfer Language (ERTL)

Dans ERTL, la *convention d'appel* est explicitée.

- ▶ *paramètres* et, le cas échéant, *résultat* des procédures et fonctions sont transmis à travers des *registres physiques* et/ou des *emplacements de pile*;
- ▶ *l'adresse de retour* devient un paramètre explicite;
- ▶ l'allocation et la désallocation des *trames de pile* devient explicite;
- ▶ les registres physiques *callee-save* sont *sauvegardés* de façon explicite.

# Explicit Register Transfer Language (ERTL)

Voici une traduction de la fonction factorielle dans ERTL :

```

procedure f(1)
var %0, %1, %2, %3, %4, %5, %6
entry f11
f11: newframe           → f10
f10: move %6, $ra      → f9
f9 : move %5, $s1     → f8
f8 : move %4, $s0     → f7
f7 : move %0, $a0     → f6
f6 : li %1, 0         → f5
f5 : blez %0          → f4, f3
f3 : addiu %3, %0, -1 → f2
f2 : j                → f20
f20: move $a0, %3     → f19
f19: call f(1)        → f18
f18: move %2, $v0     → f1
f1 : mul %1, %0, %2   → f0
f0 : j                → f17
f17: move $v0, %1     → f16
f16: move $ra, %6     → f15
f15: move $s1, %5     → f14
f14: move $s0, %4     → f13
f13: delframe        → f12
f12: jr $ra          (xmits $v0)
f4 : li %1, 1        → f0

```

# Location Transfer Language (LTL)

Dans LTL,

- ▶ la notion de pseudo-registre *disparaît*; seuls subsistent registres physiques et emplacements de pile;
- ▶ les instructions **move** d'un registre ou d'un emplacement de pile vers lui-même sont supprimées;
- ▶ les instructions **move** d'un registre vers un emplacement de pile, ou vice-versa, deviennent des instructions **lw** ou **sw**.

Le passage de ERTL à LTL, connu sous le nom *d'allocation de registres*, est complexe, et nécessite d'abord une *analyse de durée de vie* puis la *construction* et le *coloriage* d'un *graphe d'interférences*.

# Location Transfer Language (LTL)

Voici une traduction de la fonction factorielle dans LTL :

```

procedure f(1)
var Ⓛ
entry f11
f11: newframe           → f10
f10: sets local(0), $ra → f9
f9 : j                  → f8
f8 : sets local(4), $s0 → f7
f7 : move $s0, $a0      → f6
f6 : j                  → f5
f5 : blez $s0           → f4, f3
f3 : addiu $a0, $s0, -1 → f2
f2 : j                  → f20
f20: j                  → f19
f19: call f              → f18
f18: j                  → f1
f1 : mul $v0, $s0, $v0 → f0
f0 : j                  → f17
f17: j                  → f16
f16: gets $ra, local(0) → f15
f15: j                  → f14
f14: gets $s0, local(4) → f13
f13: delframe          → f12
f12: jr $ra
f4 : li $v0, 1          → f0

```

# Code Linéarisé (LIN)

Dans LIN,

- ▶ le graphe de flot de contrôle disparaît au profit d'une *suite linéaire* d'instructions;
- ▶ le *successeur* de chaque instruction redevient implicite, sauf en cas de branchement;
- ▶ les *labels* disparaissent, sauf pour les instructions cibles d'un branchement.

# Code Linéarisé (LIN)

Voici une traduction de la fonction factorielle dans LIN :

```
procedure f(1)
var 8
f11:
newframe
sets local(0), $ra
sets local(4), $s0
move $s0, $a0
blez $s0, f4
addiu $a0, $s0, -1
call f
mul $v0, $s0, $v0
f16:
gets $ra, local(0)
gets $s0, local(4)
delframe
jr $ra
f4:
li $v0, 1
j f16
```

# Assembleur (ASM)

Dans ASM,

- ▶ la gestion des trames de pile se fait par *incrément* et *décrément* explicite du registre *sp*;
- ▶ l'accès à la pile se fait à l'aide d'un *décalage fixe* vis-à-vis de *sp*;
- ▶ la notion de procédure en tant qu'entité indépendante disparaît.

ASM est un *fragment* du langage assembleur MIPS et peut être aisément *affiché* sous forme textuelle, lisible par *spim*.

# Assembleur (ASM)

Voici une traduction de la fonction factorielle dans ASM :

```
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
```

# Interpréter et afficher

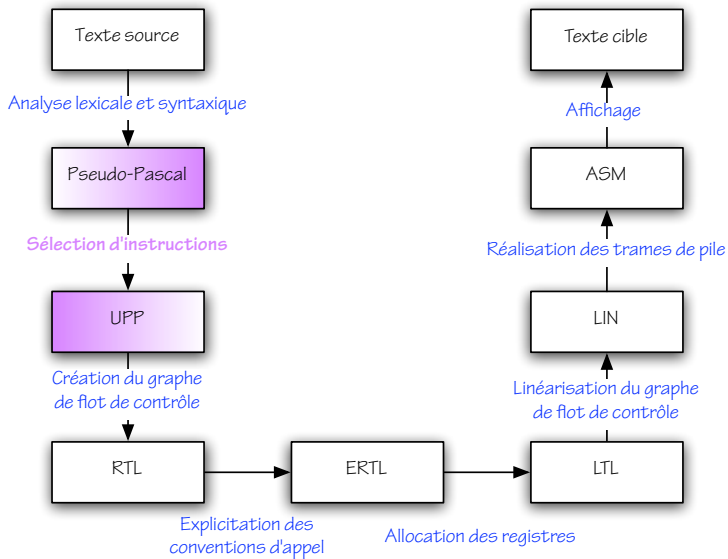
Pour tout langage intermédiaire « foo », vous pouvez interpréter ou afficher le programme traduit dans « foo » à l'aide des options « -ifoo » et « -dfoo ».

```
petit$ ./compilo -iupp test/fact.p < test/fact.in
479001600
petit$ ./compilo -dupp test/fact.p
...
```

Architecture d'un compilateur

Architecture de notre petit compilateur

Sélection d'instructions : de PP à UPP



## Untyped Pseudo-Pascal (UPP)

Dans UPP,

- ▶ les *types* sont supprimés, après vérification;
- ▶ variables globales et locales sont *distinguées*; les variables globales sont désignées par leur *adresse*;
- ▶ l'initialisation à zéro des variables locales est *explicitée*;
- ▶ les *opérateurs d'accès aux tableaux* de PP sont remplacés par les opérations **lw** et **sw** du MIPS;
- ▶ les *opérateurs arithmétiques* de PP sont remplacés par ceux du MIPS.

Les deux derniers points constituent la *sélection d'instructions*.

## Traduction des accès aux variables globales

Dans PP, la construction *EGetVar* concerne les variables locales et globales. Dans UPP, on distingue *EGetVar* et *EGetGlobal*.

Dans UPP, *EGetGlobal* porte non pas un *nom* mais un *décalage* (« *offset* ») entier, lequel désigne un emplacement dans la zone qui contient les variables globales.

Dans UPP, on distingue de même *ISetVar* et *ISetGlobal*.

Dans PP, le champ *globals* contient une *table* qui aux noms de variables associe des types. Dans UPP, ce champ indique simplement la *taille* en octets de la zone dédiée aux variables globales.

Comment traduire?

## Traduction des accès aux tableaux

Dans PP, on dispose des constructions *EArrayGet* et *IArraySet*, où le tableau et l'indice concernés sont *deux expressions* arbitraires. Dans UPP, on a *ELoad* et *IStore*, où l'adresse concernée est donnée par *une expression* arbitraire et *un décalage* constant.

Dans PP, on dispose de la construction *EArrayAlloc*. Dans UPP, on dispose d'une fonction primitive *Alloc*.

Comment traduire intelligemment?

# Traduction des opérations arithmétiques

Les opérateurs arithmétiques proposés par UPP forment un *sur-ensemble* de ceux proposés par PP, à l'exception de la négation :  $-e$  sera traduit par  $0 - e$ .

Une traduction naïve est donc très aisée...

## Traduction des opérations arithmétiques

Cependant, UPP propose également les opérateurs d'addition unaire à une constante ( $k+$ ), de décalage à gauche ( $\ll k$ ), et de comparaison unaire à une constante ( $< k$ ). Pour *bien les exploiter*, il faut remplacer  $x + 1$  par  $(1+)x$ , remplacer  $4 * i$  par  $(\ll 2)i$ , etc.

De plus, on peut souhaiter effectuer *autant d'évaluation* que possible pendant la compilation : par exemple, remplacer  $(256 - 1)$  par  $255$ , remplacer  $(x + 1) + (y + 1)$  par  $x + y + 2$ , etc.

Pour cela, nous construirons d'abord des expressions UPP de façon naïve puis les *réécrivons* en des expressions sémantiquement équivalentes mais supposées préférables.

# Réécriture

Un processus de *réécriture* est donné par un jeu de *règles*. Chaque règle est constituée d'un membre gauche et d'un membre droit — ici, des expressions UPP — pouvant contenir des *méta-variables*.

Par exemple, voici l'introduction d'une addition unaire :

$$e + k \rightarrow (k+)e \quad \text{si } k \in [-2^{15} \dots 2^{15} - 1]$$

Cette règle s'écrira en Objective Caml :

```
match ... with
| EBinOp (OpAdd, e, EConst k) when fits16 k →
  EUnOp (UOpAddi k, e)
```

## Terminaison et confluence

On peut se donner un système de réécriture arbitraire, si :

- ▶ chaque réécriture *préserve la sémantique* ;
- ▶ le système est *fortement normalisant*, c'est-à-dire qu'on ne peut pas réécrire une expression à l'infini ;
- ▶ et, de préférence, le système est *confluent*, c'est-à-dire que le résultat final ne dépend pas de l'ordre d'application des règles.

On prouve la normalisation en exhibant *une* relation d'ordre *bien fondée* telle que, pour *toute* règle, le membre droit est strictement inférieur au membre gauche.

## Exemples de règles de réécriture

$$\begin{aligned}k_1 + k_2 &\rightarrow (k_1 + k_2) \\k_1 + (k_2+)e &\rightarrow (k_1 + k_2) + e \\(k_1+)e + k_2 &\rightarrow (k_1 + k_2) + e \\O + e &\rightarrow e \\e + O &\rightarrow e \\k + e &\rightarrow (k+)e \\e + k &\rightarrow (k+)e\end{aligned}$$

Comment lire la première règle? Cette notation est *ambiguë*.

Chacune de ces règles *diminue* la taille de l'expression.

*L'ordre* d'application de ces règles est-il significatif?

## Exemples de règles de réécriture

$$\begin{aligned}(k_1+) e_1 + (k_2+) e_2 &\rightarrow ((k_1 + k_2+) (e_1 + e_2)) \\ e_1 + (k+) e_2 &\rightarrow (k+) (e_1 + e_2) \\ (k+) e_1 + e_2 &\rightarrow (k+) (e_1 + e_2)\end{aligned}$$

Ces règles diminuent-elles la taille de l'expression?

Il faut une relation d'ordre *plus complexe*.

## Exemples de règles de réécriture

$$(O - e_1) + (O - e_2) \rightarrow O - (e_1 + e_2)$$

$$e_1 + (O - e_2) \rightarrow e_1 - e_2$$

$$(O - e_1) + e_2 \rightarrow e_2 - e_1 \quad \text{si } e_1 \text{ et } e_2 \text{ sont pures}$$

Une expression est *pure* si elle n'effectue aucune écriture (dans une variable, dans un tableau, ...) et ne peut ni boucler ni effectuer d'entrées/sorties.

Pourquoi cette condition de pureté?

Comment *décider* si une expression est pure?

Comment *raffiner* ce critère de commutation?

# Formes canoniques

Une expression est en *forme canonique* si aucune règle de réécriture ne lui est applicable.

Pour une implantation simple et efficace, on ne construira que des expressions en forme canonique.

Pour cela, on écrira des fonctions qui, étant donnés des expressions filles *déjà* en forme canonique, leur *appliquent un constructeur* et *réduisent* aussitôt l'expression ainsi obtenue en forme canonique. On appelle parfois ces fonctions « *smart constructors* ».

## Formes canoniques

Ainsi, on définira un « smart constructor » *mkadd*, permettant de construire des nœuds d'addition binaire, dont le type sera :

```
val mkadd: UPP.expression → UPP.expression → UPP.expression
```

où les deux arguments comme le résultat sont en forme canonique.

De même pour les autres opérateurs.

# Formes canoniques

La règle de réécriture

$$k_1 + (k_2 +) e \rightarrow (k_1 + k_2) + e$$

devient alors l'une des branches qui définissent la fonction `mkadd` :

```
let rec mkadd e1 e2 =  
  match e1, e2 with  
  | ...  
  | EConst i1, EUnOp (UOpAddi i2, e) →  
    mkadd (EConst (i1 + i2)) e  
  | ...
```

Noter comment *l'appel récursif* permet de poursuivre la réécriture.

## En résumé

L'interface `pp2upp.mli` déclare :

```
(* This module translates [PP] into [UPP]. *)
```

```
val translate_program: PP.program → UPP.program
```

## En résumé

L'interface **upp2upp.mli** déclare :

*(\* This module provides functions for applying a [PP] unary or binary operator to [UPP] expressions.*

*These functions build a single [UPP] abstract syntax tree node. Furthermore, they perform optimization by rewriting certain [UPP] expressions into equivalent, more efficient forms. \*)*

**open** UPP

**val** mkunop: PP.unop  $\rightarrow$  expression  $\rightarrow$  expression

**val** mkbinop: MIPSOps.binop  $\rightarrow$  expression  $\rightarrow$  expression  $\rightarrow$  expression