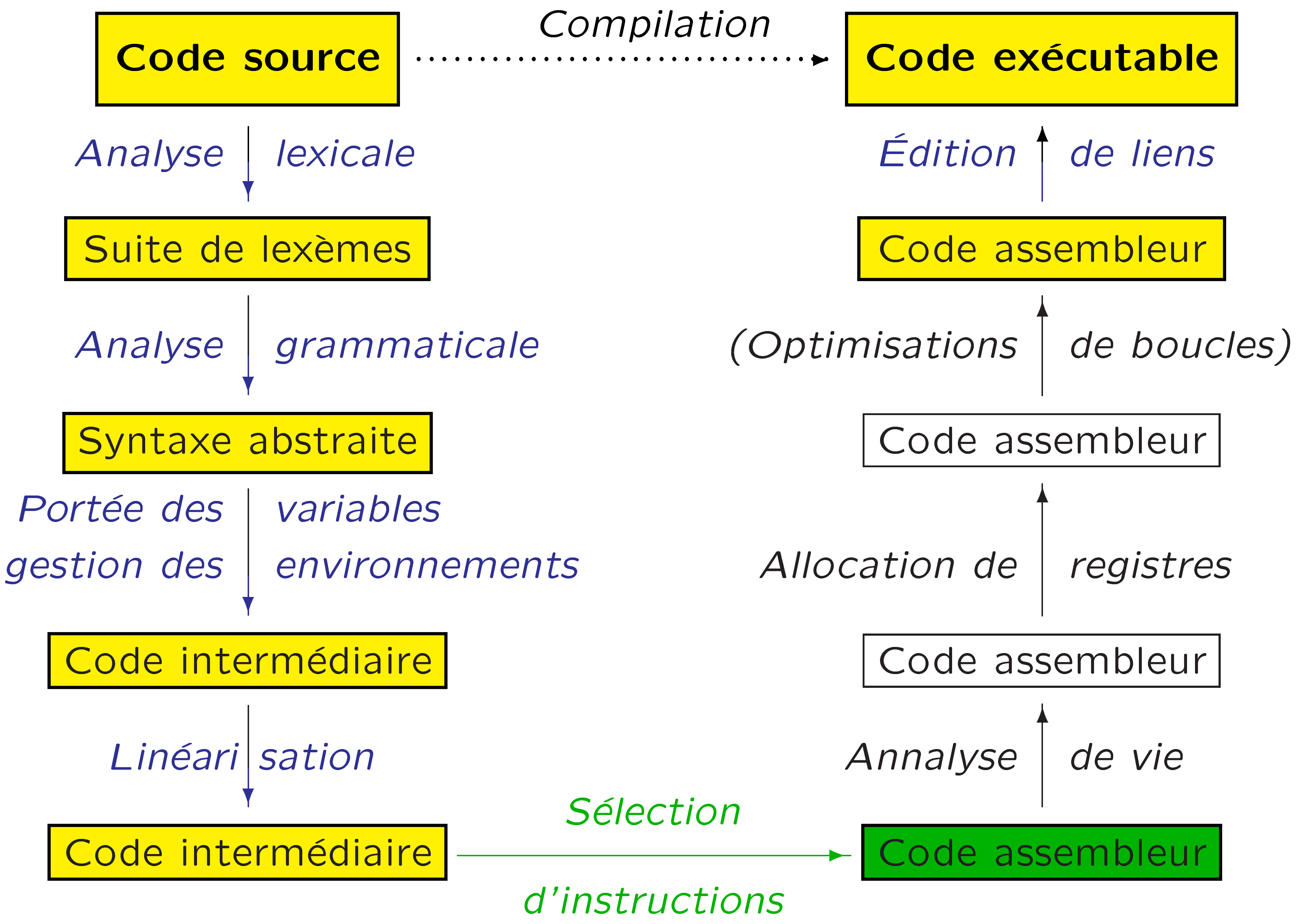
Sélection d'instructions
Projets de compilation

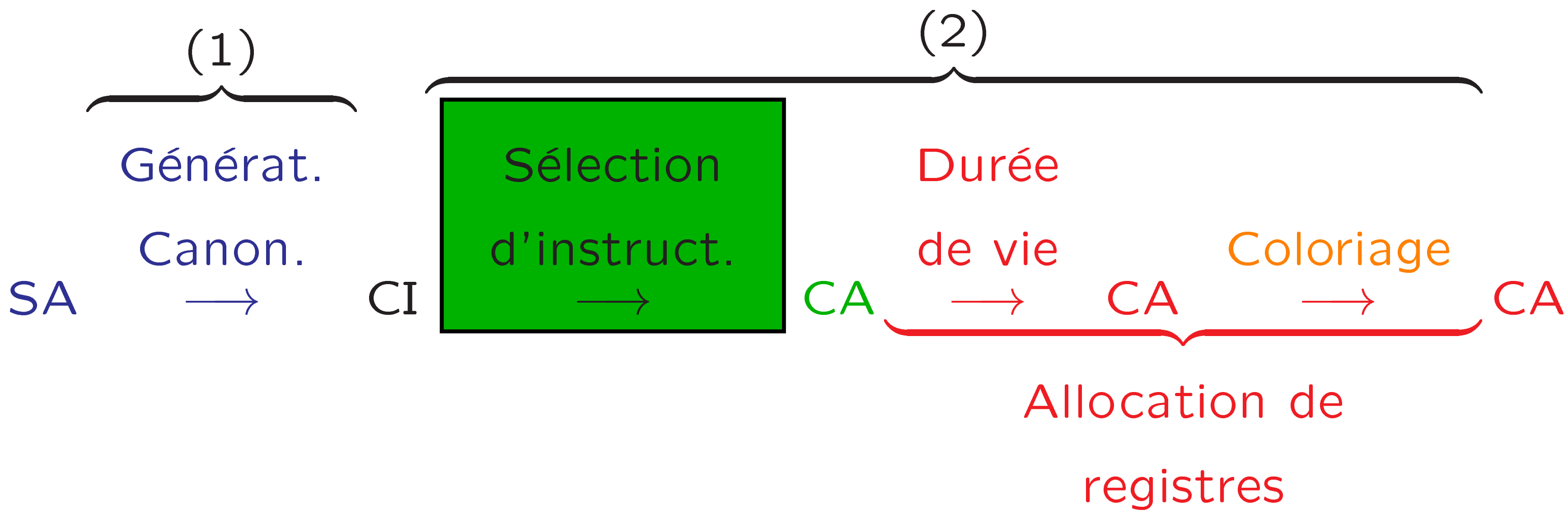
Didier Rémy

Octobre 2000

http://cristal.inria.fr/~remy/poly/compil/4/
http://w3.edu.polytechnique.fr/profs/informatique/Didier.Remy/compil/4/


Code source
Compilation
Code exécutable
Analyse lexicale
Suite de lexèmes
Analyse grammaticale
Syntaxe abstraite
Portée des variables
gestion des environnements
Code intermédiaire
Linéarisation
Code intermédiaire
Sélection
d'instructions
Code assembleur
Annalyse de vie
Code assembleur
Allocation de registres
Code assembleur
(Optimisations de boucles)
Code assembleur
Édition de liens

Une étape dans la chaîne de compilation



Pourquoi choisir les instructions avant d'allouer les registres ?

– La sélection des instructions dépend des conventions d'appel.
– Elle utilise au besoin de nouveaux temporaires.
– L'allocation des registres tiendra compte de ces temporaires et minimisera les mouvements de registres autour des appels.

La sélection d'instructions

But

Traduire le code intermédiaire en code assembleur.

- Génerer les meilleures combinaisons d'instructions possibles
- En fonction des conventions d'appel, générer les prologues et les épilogues des fonctions.

Contrainte

Pour permettre l'allocation de registres, il faut empaquetter le code assembleur dans une structure permettant ensuite l'allocation de registres.

- Choisir définitivement les instructions,
- Sans fixer le choix des registres (laissé à l'allocateur).

Les instructions sont abstraites par rapport au choix final des registres.

La représentation des instructions

On distingue les registres sources (lecture) des registres destination (écriture).

Les instructions sont représentées par des chaînes ; les registres sont codés par le caractère d'échappement `'^'` suivi d'un code `'d'` pour destination et `'s'` pour source et d'un chiffre i indiquant qu'il s'agit du i-ème registre destination ou source. Ainsi, par exemple, `"add ^d0, ^s0, ^s1"` est une représentation compacte de la fonction :

```
fun d s -> "add "^(nth d 0)^", "^(nth s 0)^", "^(nth s 1)
```

On retrouve celle-ci par décodage de la chaîne (cf `printf`) :

```
décode : string -> (temp list -> temp list -> string)
```

`décode inst src dest` retourne une instruction dans laquelle les registres formels destination et sources ont été substitués par des registres réels `src` et `dest`.

L'instruction principale

Oper (instr, src, dest, sauts)

```
type instr =
  | Oper of
      string * temp list * temp list * label list option
```

La chaîne instr code l'instruction comme indiqué précédemment.

Les listes de registres src et dest décrivent les temporaires lus et écrits par l'instruction.

– Cette information est utilisée par l'allocateur de registres ;
– Leurs temporaires seront *in fine* associés à de vrais registres.

La liste d'étiquette sauts vaut `None` si le contrôle passe toujours à l'instruction suivante. Sinon, `Some` ℓ indique explicitement l'ensemble des instructions (étiquettes) de la procédure à laquelle le contrôle peut suivre.

Deux instructions particulières

Instructions de transfert Move (instr, src, dest)

```
| Move of string * temp * temp
```

Ces instructions ne calculent pas et pourront être supprimées lorsqu'il sera possible d'associer les temporaires `src` et `dest` à un même registre (ce qu'essaye de faire l'allocateur).

Les étiquettes Label (nom, lab)

```
| Label of string * label
```

Elles permettent de suivre le flux de contrôle, ce qui est indispensable pour le calcul de durée de vie préalable à l'allocation de registres (et utile aussi pour d'autres optimisations).

Sélection des instructions

Le code intermédiaire comporte peu d'instructions, bien moins que le code machine. N'a-t-on pas perdu de l'information dans la traduction vers le code intermédiaire ?

Cette information est contenue dans la richesse de la combinaison des expressions du code intermédiaire qu'il faut reconstituer. La sélection d'instructions ne s'effecture pas construction par construction, mais traduit souvent des expressions complexes en une seule instruction machine (et quelque fois inversement) :

$$\texttt{Move\_temp}\,(t_0, \texttt{Bin}\,(\texttt{Plus}\,, \texttt{Temp}\; t_1, \texttt{Const}\;\; 3)) \longrightarrow \texttt{addu}\; t_0, t_1, 3$$

Souvent, plusieurs choix sont possibles.

Couvrir des expressions complexes

C'est nécessaire pour profiter au mieux du jeu d'instructions de la machine (à gauche une traduction pas à pas) :

$$\texttt{Mem}\,(\texttt{Bin}\,(\texttt{Plus}\,,\texttt{Temp}\ t_1,\texttt{Const}\ 3))$$

$$\boxed{\begin{array}{l}\texttt{addu}\ t',t_1,3\\ \texttt{lw}\ t',(t')\end{array}} \qquad \boxed{\texttt{lw}\ t,3(t_1)}$$

Autres motifs similaires :

$$\texttt{Mem}\,(\texttt{Bin}\,(\texttt{Plus}\,,\texttt{Const}\ 3,\texttt{Temp}\ t_1))$$

$$\texttt{Mem}\,(\texttt{Bin}\,(\texttt{Minus}\,,\texttt{Temp}\ t_1,\texttt{Const}\ 3))$$

Plus un motif par défaut : $\texttt{Mem}\ e\ldots$

Une solution est de procéder de l'extérieur vers l'intérieur par recouvrement maximal d'une partie préfixe à chaque pas.

Celle solution s'implémente naturellement par filtrage.

Coût réel des instructions

Le recouvrement maximal est une assez bonne stratégie, en particulier avec les architecture RISC, où il n'y a en général pas beaucoup de choix pour les opérations coûteuses en mémoire.

La traduction évidente n'est pas forcément la meilleure...

$$\overbrace{\texttt{Move\_Temp}\,(t_1, \texttt{Bin}\,(\texttt{Times}\,, \texttt{Temp}\ t_2, \texttt{Const}\ \ 4))}$$

$\boxed{\texttt{mul } \texttt{t}_1, \texttt{t}_2, 4}$ $\boxed{\texttt{sll } \texttt{t}_1, \texttt{t}_2, 2}$

Au lieu de se fier à l'expertise du programmeur, on pourrait compter les cycles d'exécution et trouver une approximation de la meilleure solution par programmation dynamique.

(Méthodes automatiques utiles pour les architectures CISC.)

Instructions et expressions

Chaque instruction de la machine réelle produit un effet de bord, i.e. une écriture dans un registre (pas d'expressions-machine).

Beaucoup d'expressions du code intermédiaire sont capturées par la traduction d'une instruction englobante, comme :

$$\texttt{Move\_Temp}\,(t_1, \texttt{Bin}\,(\texttt{Plus}\,, \texttt{Temp}\ t_2, \texttt{Const}\ 4))$$

Lorsqu'une expression est plus complexe,

$$\texttt{Move\_Temp}\,(t_1, \texttt{Bin}\,(\texttt{Plus}\,, \texttt{Temp}\ t_2, \texttt{Bin}\,(\texttt{Times}\ t_3, \texttt{Const}\ 4)))$$

il faut l'évaluer dans un temporaire auxiliaire :

$\texttt{mul}\ t', t_3, 4$

$\texttt{addu}\ t_1, t_2, t'$

L'allocation de registres identifiera si besoin t' et t_1. Il ne faut surtout pas chercher à le faire ici.

Mise en œuvre

Les instructions machines sont placées dans une liste globale les unes à la suite des autres —c'est plus facile.

```
emit : instr -> unit
```

La traduction d'une instruction du code intermédiare émet le code correspondant et ne retourne rien :

```
emit_stm : stm -> unit
```

La traduction d'une expression retourne le temporaire (qu'elle doit en général créé) dans lequel est placé le résultat :

```
emit_exp : exp -> temp
```

Certaines configurations sont impossibles parce que le code est supposé linéaire.

Compilation des expressions

```
let rec emit_exp = function
  | Temp t -> t
  | Const i ->
      let d = Gen.new_temp() and si = string_of_int i in
      emit (Oper ("li ^d0, " ^si , [], [  d],  None)); d
  | Mem (Bin (Plus, e, Const i))  ->
      let d = Gen.new_temp() and si = string_of_int i in
      let s = emit_exp e in
      emit (Oper ("lw ^d0, " ^si^ "(^s0)", [ s], [ d],  None));
      d
  | ...
  | Mem (e) ->
      let d = Gen.new_temp() and s = emit_exp e in
      emit (Oper ("lw ^d0, 0(^s0)", [s], [ d],  None)); d
  | Call  (f ,  args) -> raise Non_linear
```

Compilation des instructions

```
type compilation_env = { frame : Frame.frame; ...   }

let rec emit_stm compilation_env = function
  | Cjump (Rne, e0, e1, l1, l2 ) ->
      let s0 = emit_exp e0 and s1 = emit_exp e1 in
      let l = label_string l1 in
      emit (Oper ("bne ^s0, $zero, "^l, [ s0; s1], [],
                  Some [l1;l2]))
  | Move_temp (d, Bin (Plus, e, Const i)) ->
      ...
  | Move_temp (d, e) ->
      let s = emit_exp e in
      emit (Move ("move ^d0, ^s0", s, d))
  | ... ->
  | Seq q -> raise Non_linear
```

Les registres de la machine

Le passage d'arguments aux fonctions, la sauvegarde en pile, utilisent des registres conventionnels, donc des vrais registres de la machine.

On représente ceux-ci par des temporaires spéciaux. Pour cela, on définit un paquet de registres spéciaux dans le modude `Gen`, suffisamment pour en avoir assez pour toutes les architectures.

```
Gen.registers : Gen.temp array
```

Puis dans le module `spim.ml`, on puise le nombre qui convient en fonction de l'architecture,

```
let r = Array.sub registers 0 32;;
```

```
let zero, at, v0, v1, a0, a1, a2, a3 =
  r.(0), r.(1), r.(2), r.(3), r.(4), r.(5), r.(6), r.(7)
let t0, t1, t2, t3, t4, t5, t6, t7 =
  r.(8), r.(9), ...
```

Les registres conventionels du MIPS

```
let arg_registers = [a0; a1; a2; a3]
  (* Registres pour placer les premiers arguments.
     Les autres arguments seront mis sur la pile . *)
let res_registers = [v0;]
  (* Registres utilisés pour les valeurs retournées *)
let caller_saved_registers = [t0; t1; t2; t3; t4; t5; t6; t7]
let callee_saved_registers = [s0; s1; s2; s3; s4; s5; s6; s7;
                              ra (* à ne pas oublier *)]
let special_registers = [fp; gp; sp; zero]
  (* Ces registres d'usage global, survivent à la procédure,
     ce qui empêchera l'allocateur de les écraser *)
```

Ce n'est que conventionnel, sauf pour `ra` et `zero`. Autre choix :

```
let arg_registers = [a0] and caller_saved_registers = [ ]
and callee_saved_registers = [s0; ra]
```

Passage des arguments

		...	
	16	local 2	
Frame f	12	local 1	
Size 5	8	libre	$\longleftarrow$ nombre maximun d'arguments[a]
	4	arg. 6 de g	arg. 6 à l'addresse $sp + 4$
	0	arg. 5 de g	$\longleftarrow sp$
Frame g		$\vdots$	
Size s		$\vdots$	arg. 6 de g à l'addresse $sp + s + 4$
		$\vdots$	$\longleftarrow sp$

stack pointer dans le frame précédent $= sp + s$

(a) passés à une fonction appelée depuis f

Appel de fonction ou procédure

Cas général

- Placer les premiers arguments dans les registres conventionnels, puis dans la pile.
- Appeler la fonction.
- Récupérer si besoin le résultat dans le registre conventionnel.

Exemple appel d'une fonction (retournant un résultat) :

```
| Move_temp (t, Call (f, args)) ->
    let arg_in_reg = emit_args compilation_env args in
    let lab = label_string (frame_name f) in
    emit (Oper ("jal "^lab, arg_in_reg , trashed_by f, None));
    emit (Move ("move ^d0, ^s0", v0, t))
```

- `arg_in_reg` est la liste des registres conventionnels utilisés.
- `trashed_by f` est la liste des registres écrasés par l'appel à `f`.

Appel de fonction (suite)

Vu de l'appelant,

```
Oper ("jal "^lab, arg_reg, trash, None)
```

se comporte comme une super-instruction qui

1. lit les registres `arg_reg`,
2. écrit les registres `trash_by f` qui incluent les registres spéciaux utilisés dans le prologue et l'épilogue, les registres pour passer les arguments et le résultat, ceux sauvés par l'appelant (donc non sauvés par l'appelé).
3. passe à l'instruction suivante (le code de la procédure n'est pas analysé).

La liste des registres écrasés (2.) est une valeur par excès, sauf pour les primitives qui l'on traitera exactement.

Une analyse globale montrerait l'étiquette (3.) explicitement ce qui permettrait de calculer une meilleure valeur de (2.).

Appel de primitive

Un appel de primitive est comme un appel de fonction, mais

– On le reconnaît “au passage” à son étiquette.

– On connait le comportement exact de la primitive (les registres lus et écrits).

Par exemple :

```
| Move_temp (t, Call (f, args)) when f = alloc ->
    let arg_reg = emit_args args in
    let lab = label_string (frame_name f) in
    emit (Oper ("jal "^lab, [ a0], [ v0], None));
    emit (Move ("move ^d0, ^s0", v0, t))
```

Les procédures

On traduit le corps principal, mais l'une et l'autre doivent être entourées d'un prologue et d'un épilogue, qui sont placés à l'entrée et à la sortie de l'appel.

On insère le prologue juste après l'étiquette d'entrée dans la procodédure.

On reconnaît la sortie par un saut à l'étiquette de sortie. À la fin de faire le saut, on exécute l'épilogue (qui termine par un saut à `$ra`).

On compile le corps dans un contexte qui contient le frame de la procédure, et d'autres informations (voir plus loin).

Prologues et épilogues

Le prologue des procédures doit :

1. Allouer k mots en pile.
2. Sauver les registres callee-saved (s_i).
3. Placer les arguments (reçus à leur position conventionnelle, dans les registres puis dans le frame parent) dans les temporaires réservés à cet effet comme décrit dans le frame.

Symétriquement, l'épilogue doit :

3 Placer le résultat éventuel dans le registre conventionnel `v0`

2 Ressusciter les registres callee-saved (s_i).

1 Libérer les k mots en pile.

0 Sauter à l'adresse de retour.

Les instructions de l'étape 1 pourront être supprimées si le frame est vide.

Mouvement des arguments

Lorsqu'une procédure f appelle une procédure g (avec un seul argument pour simplifier) :

- l'argument de f est calculé dans un temporaire t_f (vue de l'appelant) et transféré dans a_0 avant l'appel à g.
- à l'entrée de g la valeur de a_0 est placée dans un temporaire t_g (vue de l'appelé).

On a donc trois emplacement pour l'argument : t_f (avant), a_0 (pendant) et t_g (après).

Ces mouvements sont nécessaires dans le cas général pour libérer les registres conventionnels pour d'autres appels de fonctions.

Ils seront éliminés, si possible, par l'allocateur de registres, qui choisira alors le même registre a_0 pour t_f et t_g.

(Remplacer a_0 par une position en pile pour le 6-ième argument, qui a deux adressages différents $4(sp)$ et $4 + s(sp)$ avant et après)

Sauver les registres callee-save (s_i)

Principe Ces registres doivent être sauvegardés à l'entrée de la fonction et restaurés à la fin. En général, cette sauvegarde se fera en pile, en particulier pour une fonction récursive. Mais

- ce n'est pas toujours nécessaire.
- il ne faut sauver que les registres réellement utilisés.

Pratique Le plus simple consiste à se donner de nouveaux temporaires, de transférer ces registres dans un temporaire dans le prologue et inversement de replacer les temporaires dans les registres dans l'épilogue.

- Les transferts inutiles seront annulés par l'allocateur de registres (en identifiant le temporaire et le registre).
- Si nécessaire, le temporaire sera placé en pile par l'allocateur de registres.

Solution temporaire

On peut mettre les callee-save (s_i) systématiquement en pile.

C'est pessimiste (et donc mauvais), mais cela permettra à une version simplifiée de l'allocateur de registres de fonctionner dans de nombreux cas sans avoir à faire aucune autre sauvegarde en pile (voir spilling).

Ajustement de la pile

Le nombre de mots à mettre en pile, donc la taille du frame, n'est pas encore connue au moment de la génération de code. Elle le sera après l'allocation de registres.

On peut retarder ou paramètrer la génération de l'ajustement de la pile, ou bien utiliser une constante du langage d'assembleur :

```
fact:
        sub $sp, $sp, fact_frame_size
        sw $t0, 4+fact_frame_size($sp)
```

dont la déclaration sera ajoutée dans le prologue après l'allocation de registres :

```
fact_frame_size =24
fact:
        sub $sp, $sp, fact_frame_size
        ...
```

Le saut final

La toute dernière instruction de l'épilogue est un saut à l'adresse `ra`. L'instruction correspondante est

```
emit (Oper ("j  $ra", ra:: imortal_registers  , [],   Some []))
```

C'est la dernière instruction. Comme on ne suit pas le flux, il est important de donner la liste des registres `imortal_registers` qui survivent à l'appel :

- Les registres `callee-saved` s_i, qui viennent d'être résuscités y compris ra, c'est ce qui forcera leur sauvegarde en cas d'utilisation.
- les registres spéciaux (globaux)

(Pour éviter un traitement particulier, on peut aussi y mettre `sp` et `zero` ce qui les rendra vivant tout le temps).

Procédure principale

Elle est traitée comme un appel de procédure.

Le prélude du programme alloue une table pour les globaux (ou alloue les globaux directement), exécute la procédure principale et arrête l'exécution du programme.

Exemple de la fonction factorielle...

```
program
  var x : integer ;
function fact ( var n : integer ) : integer ;
if n <= 1 then fact := 1 else fact := n * fact (n - 1);
begin
   read (x);
   writeln (fact (x))
end;;
```

Code intermédiaire

```
val cp : Code.stm Trans.program =
{Trans.number_of_globals=1;
 Trans.main=
  <main>,
  [Label main; Move_temp (t108, Call (<read_int>, []));
   Move_mem (Code.Bin (Code.Plus, Name Glob, Const 0), Temp t108);
   Move_temp
    (t109,
     Call (<fact>, [Mem (Code.Bin (Code.Plus, Name Glob, Const 0))]));
   Exp (Call (<println_int >, [Temp t109])); Jump main_end];
 Trans.procedures=
  [<fact>,
   [Label fact; Cjump (Rgt, Temp t105, Const 1, L9, L8); Label L8;
    Move_temp (t104, Const 1); Label L10; Jump fact_end; Label L9;
    Move_temp (t107, Temp t105);
    Move_temp
     (t106, Call (<fact>, [Code.Bin (Code.Minus, Temp t105, Const 1)]));
    Move_temp (t104, Code.Bin (Code.Times, Temp t107, Temp t106));
    Jump L10]]}
```

Code assembleur...

```
__start :
  la      $fp, Mem
  la      $gp, Glob
  jal     main
  li      $v0 10
  syscall

L9:
  move $107, $105
  sub  $114, $105, 1
  move $a0, $114
  jal  fact
  move $106, $v0
  mul  $115, $107, $106
  move $104, $115
  j    L10
```

```
main:
  subu $sp, $sp, main_f
  move $116, $s0
  move $117, $ra
  jal  read_int
  move $108, $v0
  sw   $108, 0($gp)
  lw   $118, 0($gp)
  move $a0, $118
  jal  fact
  move $109, $v0
  move $a0, $109
  jal  println_int
  move $s0, $116
  move $ra, $117
  addu $sp, $sp, main_f
  j    $ra
```

```
fact :
  subu $sp, $sp,fact_f
  move $110, $s0
  move $111, $ra
  move $105, $a0
  li   $112, 1
  bgt  $105, $112, L9
L8:
  li   $113, 1
  move $104, $113
L10:
  move $v0, $104
  move $s0, $110
  move $ra, $111
  addu $sp, $sp,fact_f
  j    $ra
```

(avec un seul registre callee-saved s0)

Exercices

Implémenter la génération de code machine MIPS

On fournit :

- L'ensemble des maillons de la chaîne précédente.
- La descrition du code Machine ass.ml et son interface ass.mli.
- Une implémentation partielle spim.ml d'interface spim.mli.

Pour récupérer le tout dans le répertoire courant (attention à vous placer au bon endroit) :

```
cp ~remy/compil/td4/* ./
```

Vérifier que l'ensemble est bien installé :

```
make
```

On ne pourra tester qu'après avoir complété au moins une partie du programme :

ocaml un.mlx

...

ocaml bignum.mlx

qui utilisent l'entête `run.ml`

Dans un premier temps, la génération de code peut être inefficace, en ne retenant que les cas génériques.

On pourra facilement voir le résultat en imprimant la liste des instructions (avec `Ass.format`, voir `run.ml`)

Solution spim.ml

Pour voir le code produit, vous pouvez utiliser le compilater de démonstration avec l'option `-ass`.

```
cd ~remy/compil/ppc/
ppc -ass test/fact.p
```

(le prélude n'est pas affiché) et comparer le code que vous générez.

Des différences sont possibles, bien entendu, mais elles vous permettront sans doute de trouver des erreurs éventuelles.