Projet 2: un compilateur

cours: Daniel Hirschkoff
TPs: Paul Renaud-Goud

Contenu et objectifs

- comprendre le fonctionnement d'un compilateur

  et en écrire un

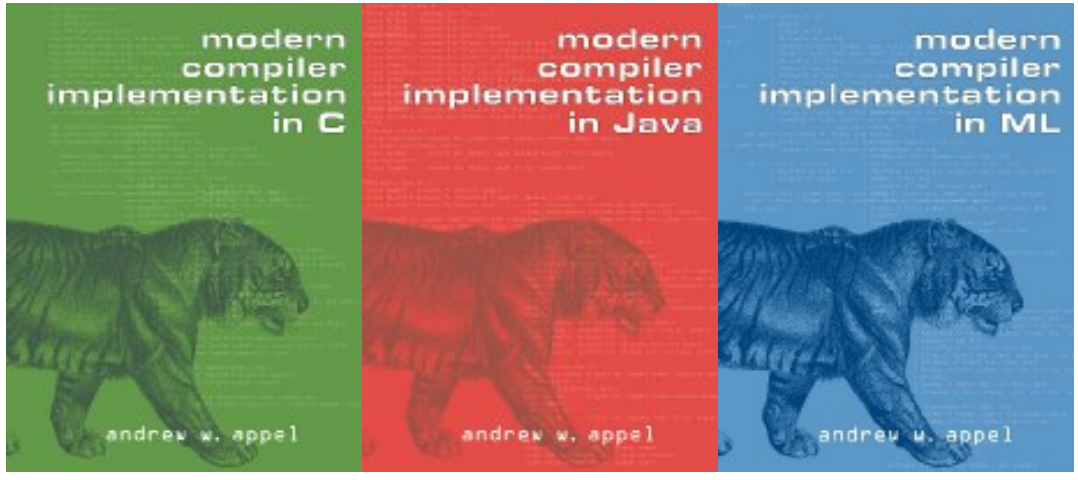
- mener un projet de programmation de taille conséquente
  - améliorer ses capacités de programmation
    - organisation du code, modularité
    - efficacité du code *(de manière moins prépondérante)*

Contenu et objectifs

- comprendre le fonctionnement d'un compilateur

  et en écrire un

- mener un projet de programmation de taille conséquente
  - améliorer ses capacités de programmation
    - organisation du code, modularité
    - efficacité du code *(de manière moins prépondérante)*
- référence pour le cours: **le triptyque d'Andrew Appel**



disponibles à la bibliothèque

Organisation

- aujourd'hui: mise en route
- ensuite: essentiellement des séances sur machine, quelques cours
  - projet 2 est orienté "pratique"

Organisation

- aujourd'hui: mise en route
- ensuite: essentiellement des séances sur machine, quelques cours
  - projet 2 est orienté "pratique"

- notation: **rendus de programmes**
  - un premier DM, à rendre le 16/2
  - diverses tranches du compilateur, jusqu'à la version finale
    - en binôme (au plus un monôme)
    - exigences *"adaptées"*
      . niveaux fort différents en programmation
      . **questionnaire**

Organisation

- aujourd'hui: mise en route
- ensuite: essentiellement des séances sur machine, quelques cours
  - projet 2 est orienté "pratique"

- notation: **rendus de programmes**
  - un premier DM, à rendre le 16/2
  - diverses tranches du compilateur, jusqu'à la version finale
    - en binôme (au plus un monôme)
    - exigences *"adaptées"*
      . niveaux fort différents en programmation
      . **questionnaire**
  - important: coder *entre les séances*

Projet(s)

- par défaut:

un mini-Pascal, compilé vers SPIM, en OCaml

- autres langages de programmation: C, Java
  - si vous êtes plus à l'aise dans l'un de ces langages
  - *(si vous maîtrisez bien Caml et voulez profiter de l'occasion pour apprendre un autre langage)*

- autres langages source à compiler
  - si vous "pouvez vous le permettre"
  - à discuter

Planning indicatif du semestre

- vacances
  - hiver : 20-26 février
  - printemps : 9-15 avril
  - semaine d'examens: semaine du 28 mai
- constitution des binômes: 10 février
- premier devoir: à envoyer pour le 16/2 à 23h59
- tranches de compilateur:
  - analyses lexicale et syntaxique 8 mars
  - analyse statique, table des symboles 22 mars
  - génération de code, version 1 19 avril
  - génération de code, version 2 3 mai
  - génération de code, version 3 23 mai *(mercredi)*

déplaçables en discutant (un peu à l'avance)

Machines à l'ENS, machine personnelle

pour faire votre projet, il vous faudra utiliser

- un compilateur (Caml, C, Java)
- `fpc`, compilateur Pascal, pour faire des tests
- `qtspim`, pour le code généré

tout cela est disponible[1] sur les salles machines de l'ENS: soit vous travaillez dessus, soit vous installez tout sur votre machine personnelle

[1] sous peu

DM: le compilateur des débutants

- objectifs
  - faire une “maquette de compilateur”, pour voir où l'on va
  - se familiariser avec SPIM
- ce qui est demandé
  - un point de départ est fourni sur la page web du cours `(+ *)`
  - ajouter `- / fun(x,y)`
- c'est un compilateur minimal:
  - on a déjà traité (en Théorie de la Programmation) l'entrée: analyses lexicale et syntaxique
  - on passe directement à la génération de code

Un exemple de langage de bas niveau

SPIM

Présentation de SPIM

- SPIM simule une machine MIPS
  - n'est pas fidèle à la machine simulée en termes de performances
  - propose des *pseudo-instructions* (qui sont gérées de manière astucieuse par la vraie architecture, pas par le simulateur)
- on voit l'assembleur de SPIM comme un langage de programmation, avec son modèle d'exécution
  - on traduit d'un modèle à l'autre
- le "moteur"
  - unité de calcul (CPU)
  - 32 registres, avec des conventions de nommage et d'utilisation: les respecter (si on était en vrai, il faudrait les respecter pour la compatibilité avec d'autres codes)

SPIM: ingrédients (aperçu)

- instructions en SPIM: un langage impératif

```
            bla
            bli     # commentaire captivant
machin:     blu
            blo
```

SPIM: ingrédients (aperçu)

- instructions en SPIM: un langage impératif

```
            bla
            bli     # commentaire captivant
machin:     blu
            blo
```

- opérations: elles se font uniquement sur les registres

`add` ri, rj, rk	la somme de rj et rk dans ri
`add` ri, rj, n	rj+n dans ri
`addu`	même chose en traitant ses arguments comme des entiers non signés

SPIM: ingrédients (aperçu)

- instructions en SPIM: un langage impératif

```
            bla
            bli     # commentaire captivant
machin:     blu
            blo
```

- opérations: elles se font uniquement sur les registres

`add` ri, rj, rk	la somme de rj et rk dans ri
`add` ri, rj, n	rj+n dans ri
`addu`	même chose en traitant ses arguments comme des entiers non signés
`div` ri, rj	divise ri par rj; quotient dans lo, reste dans hi
`seq` ri, rj, rk	ri=1 si rj=rk, 0 sinon
`seq` ri, rj, n	ri=1 si rj=n, 0 sinon

SPIM: ingrédients (aperçu)

- instructions en SPIM: un langage impératif

```
          bla
          bli     # commentaire captivant
machin:   blu
          blo
```

- opérations: elles se font uniquement sur les registres

`add` ri, rj, rk	la somme de rj et rk dans ri
`add` ri, rj, n	rj+n dans ri
`addu`	même chose en traitant ses arguments comme des entiers non signés
`div` ri, rj	divise ri par rj; quotient dans lo, reste dans hi
`seq` ri, rj, rk	ri=1 si rj=rk, 0 sinon
`seq` ri, rj, n	ri=1 si rj=n, 0 sinon

- sauter

`j` *lab*	saut à l'instruction labélisée par *lab*
`jal` *lab*	saute en *lab* après avoir noté en *ra* l'adresse de l'instruction suivante
`beq` ri, rj, *lab*	va en *lab* si ri=rj
`beq` ri, n, *lab*	va en *lab* si ri=n

Ingrédients de SPIM, suite – mémoire

- *tout* réside quelque part en mémoire
- interactions avec la mémoire

`lw` ri, add	met dans ri le mot situé en add
`la` ri, add	met dans ri l'*adresse* add
`li` ri, k	met k dans ri

formes possibles des adresses

`(rj)`	adresse contenue dans rj
`k(rj)`	k + adresse contenue dans rj
`lab`	adresse du label lab
`k`	l'entier k

`sw` ri, add	recopie le contenu de ri en add

SPIM – autres

- primitives prédéfinies: appels système

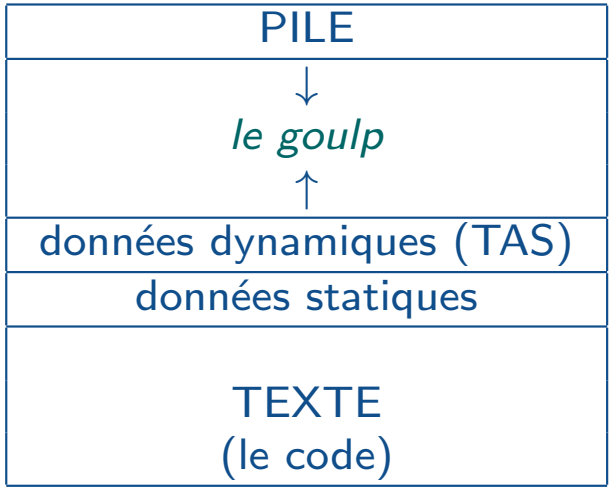
```
li $v0, 1     1 signifie print_int
li $a0, 32    32 est la valeur à imprimer
syscall
```

SPIM – autres

- primitives prédéfinies: appels système

```
li $v0, 1     1 signifie print_int
li $a0, 32    32 est la valeur à imprimer
syscall
```

- organisation de la mémoire

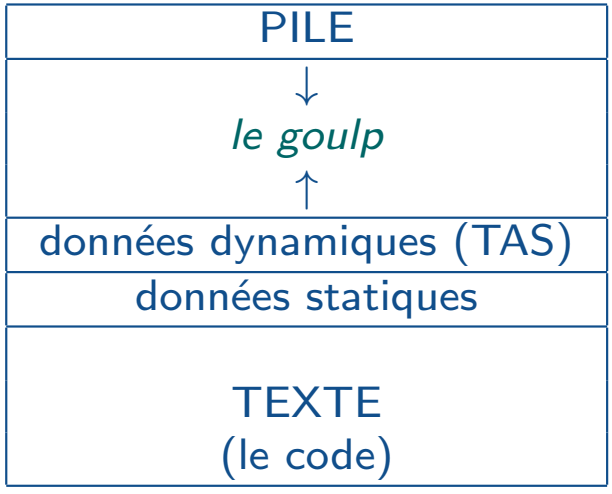


SPIM – autres

- primitives prédéfinies: appels système

```
li $v0, 1     1 signifie print_int
li $a0, 32    32 est la valeur à imprimer
syscall
```

- organisation de la mémoire



- directives
  - `.text` BLA mettre BLA dans la zone de texte
  - `.data` BLI mettre BLI dans la zone de données (statiques)

```
        .data                                          des données
str:    .asciiz "le plus difficile, ce n'est pas\n"    un label, des données
        .text                                          du code
        li $v0, 4                                      4 signifie print_string
        li $a0, str
        syscall
```

Génération de code

Redescendre de l'arbre

- analyses lexicale et syntaxique: le *front end* d'un compilateur
- *back end*: génération de code

  à partir d'un arbre (pas trop l'arbre de syntaxe abstraite, mais le résultat de potentiellement beaucoup de transformations et d'analyses) on engendre du code *linéaire*

1. expressions arithmétiques
2. contrôle (if, boucles)
3. fonctions

La pile

- retour aux expressions arithmétiques

```
type expr =
    Const of int
  | Add of expr*expr
  | Mul of expr*expr
```

Évaluer les expressions arithmétiques sur la pile

une **machine abstraite à pile**: modèle d'exécution intermédiaire

(le modèle, pas l'exécution)

- pour évaluer `Add(Const 5, Const 7)`, on engendre

```
push 7
push 5
add
```

  - l'appel à `add`
    - dépile les arguments
    - fait le calcul
    - met le résultat sur le sommet de la pile

Évaluer les expressions arithmétiques sur la pile

une **machine abstraite à pile**: modèle d'exécution intermédiaire

(le modèle, pas l'exécution)

- pour évaluer `Add(Const 5, Const 7)`, on engendre

```
push 7
push 5
add
```

  - l'appel à `add`
    - dépile les arguments
    - fait le calcul
    - met le résultat sur le sommet de la pile
- l'expression `Add(Const 5, Const 7)` est traduite en une suite d'instructions de la *machine abstraite*
  - la pile + un jeu simple d'instructions
  - abstraction: on ne rentre pas dans les détails sur l'architecture, sur comment est implémentée la pile, etc. *(portabilité, p.ex. Java)*
- le sommet de la pile est accédé très fréquemment

  p.ex. trois accès lors de l'exécution de `add`

Un peu plus de détails

- raffinement de la machine abstraite
  - on dispose d'un *registre*, l'accumulateur, nommé `acc`
    - registre: accès rapide (il y en a généralement plus d'un!)
  - les calculs se font sur l'accumulateur
  - la pile n'est utilisée que pour stocker les arguments

Un peu plus de détails

- raffinement de la machine abstraite
  - on dispose d'un *registre*, l'accumulateur, nommé `acc`
    - registre: accès rapide (il y en a généralement plus d'un!)
  - les calculs se font sur l'accumulateur
  - la pile n'est utilisée que pour stocker les arguments
- cela donne (avec un nouveau jeu d'instructions)

```
acc <- 7
push acc
acc <- 5
acc <- acc + top    (* opération atomique *)
pop
```

Exemple un peu plus grand: 3 + (7 + 5)

```
acc <- 3
push acc
 acc <- 7
 push acc
 acc <- 5
 acc <- acc + top
 pop
acc <- acc + top
pop
```

Exemple un peu plus grand: 3 + (7 + 5)

```
acc <- 3
push acc
 acc <- 7
 push acc
 acc <- 5
 acc <- acc + top
 pop
acc <- acc + top
pop
```

- *leitmotive*
  - on rend la pile dans l'état de propreté dans laquelle on l'a trouvée
  - le résultat est en `acc` à la fin
- but du jeu
  - jeu d'instructions simples
  - traitement uniforme (ici: évaluer une addition)
  - les instructions correspondent à des opérations élémentaires en machine

Raffinement encore: 'programmer' la pile soi-même

on atteint le niveau de SPIM

```
acc <- 7              li $a0, 7
push acc              sw $a0, 0($sp)
                      add $sp, $sp, -4
acc <- 5              li $a0, 5
acc <- acc + top      lw $t1, 4($sp)
                      add $a0, $a0, $t1
pop                   add $sp, $sp, 4
```

- `$sp` registre contenant l'adresse de la fin de la pile
- `4`: 4 octets, de quoi stocker un mot
- `li reg, imm` *met le registre `reg` à `imm`*
- `sw reg1, decalage(reg2)` *stocke le contenu de `reg1` a l'adresse (contenu de `reg2`)+`decalage`*
- `add reg1, reg2, imm` *ajoute `imm` au contenu de `reg2` et stocke dans `reg1`*
- `add reg1, reg2, reg3` *ajoute les contenus de `reg2` et `reg3` et stocke cela en `reg1`*
- `lw reg1, decalage(reg2)` *charge ce qui est stocké en l'adresse en `reg2` + `decalage` dans `reg1`*

Raffinement encore: 'programmer' la pile soi-même

on atteint le niveau de SPIM

```
acc <- 7              li $a0, 7
push acc              sw $a0, 0($sp)
                      add $sp, $sp, -4
acc <- 5              li $a0, 5
acc <- acc + top      lw $t1, 4($sp)
                      add $a0, $a0, $t1
pop                   add $sp, $sp, 4
```

- `$sp` registre contenant l'adresse de la fin de la pile
- `4`: 4 octets, de quoi stocker un mot
- `li reg, imm` *met le registre `reg` à `imm`*
- `sw reg1, decalage(reg2)` *stocke le contenu de `reg1` a l'adresse (contenu de `reg2`)+`decalage`*
- `add reg1, reg2, imm` *ajoute `imm` au contenu de `reg2` et stocke dans `reg1`*
- `add reg1, reg2, reg3` *ajoute les contenus de `reg2` et `reg3` et stocke cela en `reg1`*
- `lw reg1, decalage(reg2)` *charge ce qui est stocké en l'adresse en `reg2` + `decalage` dans `reg1`*
- le résultat est en `$a0` à la fin

Le backend: de l'arbre au fichier objet

```
let p = print_string
```

- `Const i` $\rightarrow$

```
p "li $a0 "; print_int i;
print_newline();
```

- `Add (e1,e2)` $\rightarrow$

```
engendre(e1);
  (* push *)
p "sw $a0, 0($sp)\n";
p "add $sp, $sp, -4\n";
engendre(e2);
  (* pop *)
p "lw $t1, 4($sp)\n";
p "add $sp, $sp, 4\n";
p "add $a0, $a0, $t1\n";
```

Branchements (in)conditionnels

```
‘‘if e1=e2 then e3 else e4’’
```

```
type expr = ...    | IfEq of expr*expr*expr*expr
```

```
                  ‘‘engendre(e1)’’
                  sw $a0, 0($sp)
                  add $sp, $sp, -4
                  ‘‘engendre(e2)’’
                  lw $t1, 4($sp)
                  add $sp, $sp, 4
                  beq $a0, $t1, branche_vrai
branche_faux:     ‘‘engendre(e4)’’
                  b fin_if
branche_vrai:     ‘‘engendre(e3)’’
fin_if:           ...
```

Branchements (in)conditionnels

```
‘‘if e1=e2 then e3 else e4’’
```

```
type expr = ...    | IfEq of expr*expr*expr*expr
```

```
                  ‘‘engendre(e1)’’
                  sw $a0, 0($sp)
                  add $sp, $sp, -4
                  ‘‘engendre(e2)’’
                  lw $t1, 4($sp)
                  add $sp, $sp, 4
                  beq $a0, $t1, branche_vrai
branche_faux:     ‘‘engendre(e4)’’
                  b fin_if
branche_vrai:     ‘‘engendre(e3)’’
fin_if:           ...
```

exercice: boucles

Un premier compilateur

un front end, un back end

- Démo

Un premier compilateur

un front end, un back end

- DÉMO

- *NB: on aurait pu tout faire en une passe...*
  - au lieu de `| expr PLUS expr { Add($1,$3) }`,
    mettre `| expr PLUS expr { $1 + $3 }`

  ... *seulement tant que le langage est banal (`let f x = ...`)*

Les fonctions

- la structuration de programmes en fonctions qui s'appellent entre elles rend (encore plus) naturelle l'utilisation d'une **pile**

Les fonctions

- la structuration de programmes en fonctions qui s'appellent entre elles rend (encore plus) naturelle l'utilisation d'une **pile**
- sur la pile: *enregistrements d'activation*
  - ce sont des enregistrements *(records)*
  - un enregistrement sur la pile correspond à la période durant laquelle *un* appel d'une fonction est actif
    - il contient l'information "propre" à *cet* appel de *cette* fonction

Un langage fort simple

- juste des fonctions “de type `int->int`”

  un programme:

  des fonctions,

  une expression

```
def f(x) = x*x
def g(y) = 5+y
def h(z) = g(z+3)*z

h(4)
```

- du point de vue de la compilation, il faut
  - engendrer le code correspondant au corps d'une fonction

```
corps_f:  inst1
          inst2
          ⋮
```

  - prévoir la gestion de l'entrée et de la sortie dans un appel de fonction (manipulations de la pile)

Compilation des fonctions, éléments de base

- appel de la fonction: `f(e)`

```
‘‘engendre(e)’’
jal code_f
blibli
```

  - `jal`: saute à l'adresse donnée, et stocke l'adresse qui suit le `jal` dans un registre spécial, `$ra`

Compilation des fonctions, éléments de base

- appel de la fonction: `f(e)`

```
‘‘engendre(e)’’
jal code_f
blibli
```

  - `jal`: saute à l'adresse donnée, et stocke l'adresse qui suit le `jal` dans un registre spécial, `$ra`
  - lorsqu'on saute,
    - `$a0` contient la valeur du paramètre de `f`
    - `$ra` contient l'adresse de `blibli`, l'instruction a exécuter en retournant de l'appel à `f`

Compilation des fonctions, éléments de base

- appel de la fonction: `f(e)`

```
''engendre(e)''
jal code_f
blibli
```

  - `jal`: saute à l'adresse donnée, et stocke l'adresse qui suit le `jal` dans un registre spécial, `$ra`
  - lorsqu'on saute,
    - `$a0` contient la valeur du paramètre de `f`
    - `$ra` contient l'adresse de `blibli`, l'instruction a exécuter en retournant de l'appel à `f`

- au sein de la fonction: `def f(x)=corps`

```
code_f:   ''engendrer(corps)''
          jr $ra      (a pour effet de retourner en blibli)
```

Compilation des fonctions, éléments de base

- appel de la fonction: `f(e)`

```
''engendre(e)''
jal code_f
blibli
```

  - `jal`: saute à l'adresse donnée, et stocke l'adresse qui suit le `jal` dans un registre spécial, `$ra`
  - lorsqu'on saute,
    - `$a0` contient la valeur du paramètre de `f`
    - `$ra` contient l'adresse de `blibli`, l'instruction a exécuter en retournant de l'appel à `f`

- au sein de la fonction: `def f(x)=corps`

```
code_f:   ''engendrer(corps)''
          jr $ra      (a pour effet de retourner en blibli)
```

  - oui mais dans `corps`
    - on veut accéder au paramètre `x` de la fonction
    - on veut utiliser `$a0` pour faire des calculs

Compilation des fonctions, éléments de base

- appel de la fonction: `f(e)`

```
''engendre(e)''
jal code_f
blibli
```

  - `jal`: saute à l'adresse donnée, et stocke l'adresse qui suit le `jal` dans un registre spécial, `$ra`
  - lorsqu'on saute,
    - `$a0` contient la valeur du paramètre de `f`
    - `$ra` contient l'adresse de `blibli`, l'instruction a exécuter en retournant de l'appel à `f`

- au sein de la fonction: `def f(x)=corps`

```
code_f:   ''engendrer(corps)''
          jr $ra      (a pour effet de retourner en blibli)
```

  - oui mais dans `corps`
    - on veut accéder au paramètre `x` de la fonction
    - on veut utiliser `$a0` pour faire des calculs
  - convention: le paramètre est accédé dans `$a1`
    recopie `$a0`→`$a1`: avant de sauter (ou juste après)

Compilation des fonctions, éléments de base

- appel de la fonction: `f(e)`

```
''engendre(e)''
jal code_f
blibli
```

  - `jal`: saute à l'adresse donnée, et stocke l'adresse qui suit le `jal` dans un registre spécial, `$ra`
  - lorsqu'on saute,
    - `$a0` contient la valeur du paramètre de `f`
    - `$ra` contient l'adresse de `blibli`, l'instruction a exécuter en retournant de l'appel à `f`

- au sein de la fonction: `def f(x)=corps`

```
code_f:   ''engendrer(corps)''
          jr $ra      (a pour effet de retourner en blibli)
```

  - oui mais dans `corps`
    - on veut accéder au paramètre `x` de la fonction
    - on veut utiliser `$a0` pour faire des calculs
  - convention: le paramètre est accédé dans `$a1`
    recopie `$a0`→`$a1`: avant de sauter (ou juste après)
  - fonctions qui appellent des fonctions:

```
def h(z) = g(z+3)*z
```

il faut *sauvegarder* `$a1` et `$ra`

Une approche possible

- à chaque appel à une fonction, on empile un *enregistrement d'activation*, pour stocker deux informations:
  - la valeur de `$ra` au moment où on entre dans la fonction
  - la valeur du paramètre
- pour gérer un appel `f(e)`:
  - on évalue l'argument `e`
  - on sauvegarde la valeur de `$a1` sur la pile, puis on recopie `$a0` en `$a1`
  - on saute
  - on sauvegarde `$ra` sur la pile
  - on exécute le corps de la fonction, le résultat est dans `$a0`
  - on restaure les valeurs originelles de `$a1` et `$ra`
  - on sort de la fonction

- optimisation possible: si `corps` ne fait pas d'appel de fonction, ne pas toucher à `$ra`

Une approche possible

- à chaque appel à une fonction, on empile un *enregistrement d'activation*, pour stocker deux informations:
  - la valeur de `$ra` au moment où on entre dans la fonction
  - la valeur du paramètre
- pour gérer un appel `f(e)`:
  - on évalue l'argument `e`
  - on sauvegarde la valeur de `$a1` sur la pile, puis on recopie `$a0` en `$a1`
  - on saute
  - on sauvegarde `$ra` sur la pile
  - on exécute le corps de la fonction, le résultat est dans `$a0`
  - on restaure les valeurs originelles de `$a1` et `$ra`
  - on sort de la fonction

- optimisation possible: si `corps` ne fait pas d'appel de fonction, ne pas toucher à `$ra`
- **tout ceci est donné à titre d'exemple**

Le compilateur: de Pascal à SPIM

Un exemple de programme Pascal

```
program exemple;

var z : integer;

function f (var n : integer) : integer;
var a:integer;
begin
    a := n*n;
    n := n-1;
    f := a+n;
end;

function g (k : integer) : integer;
    function h (p : integer) : integer;
    begin
        h := p*k;
    end;
begin
    g := h(2*k) + h(3*k);
end;

begin
    z := 5;
    writeln (f(z) + g(z));
end.
```

- fonctions,
  fonctions récursives

  `f := 12` pour renvoyer le résultat
- variables locales
- fonctions à l'intérieur de fonctions
- passage par référence
- votre "corrigé" : `fpc`

Rendus 2 et 3

- vous pouvez vous attaquer rapidement au rendu numéro 2: analyses lexicale et syntaxique
  - lex, yacc, shift/reduce conflicts, etc.
  - en sortie: AST (arbre de syntaxe abstraite) ← *à définir*

Rendus 2 et 3

- vous pouvez vous attaquer rapidement au rendu numéro 2: analyses lexicale et syntaxique
  - lex, yacc, shift/reduce conflicts, etc.
  - en sortie: AST (arbre de syntaxe abstraite) ← *à définir*
- rendu numéro 3: analyse statique, table des symboles
  - analyse statique: détecter des problèmes dans l'AST
    - problèmes de portée des variables
      `f := x+g(y);` `y` non défini
      `a := h(2);` `h` non définie
    - problèmes de "typage": nombre d'arguments des fonctions
      `x := f(y,y,z);`
    - autres (pas de "`f :=...`" dans la définition de `f`, ...)
  - table des symboles: parcourir l'AST pour
    - décorer les identificateurs avec leur "carte d'identité"
    - renommer éventuellement les identificateurs (`x1, x2, x3,...`)

    ↪ structure de données à concevoir, qui sera fort probablement reprise/étendue par la suite