La Compilation

Cours de Majeure 2
de l'École Polytechnique

Didier Rémy

Janvier 2001

Résumé

Ce cours est très court pour un cours de compilation ! Le premier objectif est de démystifier la compilation en décortiquant le processus en une composition de transformations plus élémentaires, permettant simultanément d'en aborder les aspects essentiels.

Au delà de la compilation pour elle-même c'est aussi l'introduction concepts de méthodes et d'outils que l'on retrouve dans de nombreux domaines et applications de l'informatique.

Malgré tout, le cours comporte également une composante pratique importante, et devrait amener à l'écriture d'un compilateur complet pour un mini-Pascal. La restriction a un petit langage permet simplement une simplification en largeur, mais les composants essentiels d'un compilateur sont présentés de façon générale et rigoureuse. Le compilateur résultant, bien que celui d'un langage restreint a une architecture et une efficacité très réalistes.

Note

Ce recueil est composé des transparents du cours de compilation de l'année 2000, qui sont également disponibles électroniquement à l'adresse

http://cristal.inria.fr/~remy/poly/compil/

Table des matières

Présentation du Cours.

Avertissement

La version distribuée des notes de cours est la compilation des transparents du cours de l'année 2000. Elles seront mises à jour et parfois détaillées au fur et à mesure du cours. Ces notes de cours sont également disponibles sur le WEB à l'adresse

http://cristal.inria.fr/~remy/poly/compil/

Slide 1

dans leur version la plus récente. Une version HTML portable (pour être installée sur une machine personnelle et lue sans connection réseau) peut aussi être trouvée à cette adresse et installée sur votre machine.

Emploi du temps
Introduction
La chaîne de compilation.

Informations utiles

Votre Chef de Travaux pratiques Luc Maranget

Le calendrier

Slide 2

Les mardi
12/01, 16/01, 30/01, 06/02, 13/02, 27/03, 06/03, 13/03, 20/03
- Cours en salle ? ? de 8h30 à 10h
- Travaux dirigés en salle 33 de 10h15 à 12h15

Examens
- écrit le lundi 23/03 de 10h à 12h.
- oraux, entre le 26/03 et le 29/03.

Quelques buts du cours de compilation

Slide 3

- Démystifier et comprendre comment les programmes sont compilés puis exécutés par un ordinateur.
- Apprendre des techniques, des méthodes et l'utilisation d'outils classiques utiles : expressions régulières, analyseurs lexicaux et grammaticaux, interprètes, transformations par découpage en passes successives.
- Réaliser un mini-compilateur :
  - La compilation est un domaine difficile ; L'erreur ne pardonne pas car elle peut se manifester bien plus tard...
  - L'ensemble est compliqué : il faut bien le découper pour le simplifier.
  + Des méthodes et des techniques bien comprises.

Qu'est-ce que la compilation ?

Slide 4

C'est une traduction, en général d'un langage évolué vers un langage moins expressif.

Quelques exemples :
- Compilation des expressions régulières (utilisées dans les outils `sed`, `awk`, l'éditeur `Emacs` et les bibliothèques des langages `C`, `Perl` et `Ocaml` ;
- Minimisation des automates (compilation d'un automate vers un automate plus efficace) ;
- De Tex vers Postscript ou HTML (Hevea) ; *etc.*

La *décompilation* : on va dans le sens inverse, d'un langage moins structuré vers un langage évolué. Ex : retrouver le source C à partir d'un code compilé (d'un programme C). Applications : piratage, récupération de vieux logiciels.

Compilation des langages de programmation

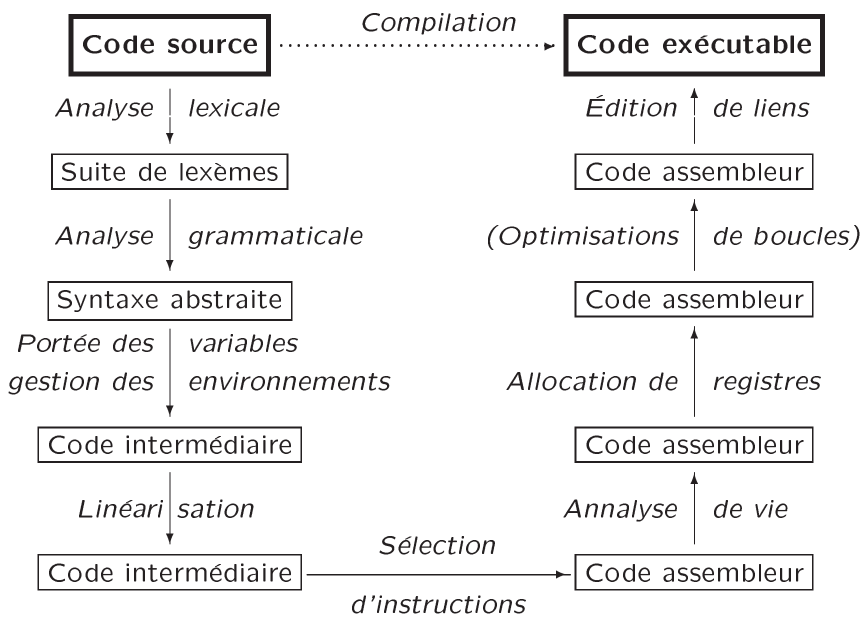
Slide 5

Il s'agit de transformer un fichier (suite de caractères) décrivant un programme en un fichier (suite d'instructions) directement exécutable par une machine.

Principe :
- Découper la compilation en une succession de transformations simples.
- Introduire des représentations intermédiaires à des points importants.

Différence avec la compilation des expressions régulières où l'on peut passer (presque) directement de la chaîne de caractères décrivant l'expression régulière à l'automate.

Slide 6


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

1

Code Machine
Code Assembleur
Processeurs RISC
(Byte code).

Informations utiles

- La pico-machine décrite dans le livre *Le langage Caml* de Pierre Weis et Xavier Leroy.
- Le simulateur SPIM du processeur MIPS R2000 est présenté succinctement ci-dessous. Voir aussi son manuel de référence en ligne et en Postscript.

Slide 1

Les processeurs

Slide 2

Les processeurs se ressemblent tous du point de vue de l'utilisateur (*i.e.* de la partie visible de leur architecture). Ils comportent principalement une mémoire, un ensemble de registres, et un jeu d'instructions.

Les différences proviennent surtout du jeu d'instructions :
- Les (vieux) processeurs CISC (*Complex Instruction Set*)
  - Leurs instructions sont de taille variable et beaucoup réalisent des transferts avec la mémoire ; ils possèdent en général peu de registres (et pas uniformes)
  - Conçus avant 1985. Typiquement : Intel 8086 et Motorola 68000.
- Les (nouveaux) processeurs RISC (*Reduced Instruction Set*)
  - Instructions de taille fixe, régulières (*trois adresses*) dont peu font des transfert avec la mémoire. Ils possèdent en général beaucoup de registres (uniformes).
  - Conçus après 1985. Typiquement : Alpha, Sparc et Mips. G4.

La mémoire

Slide 3

Tous les processeurs modernes comportent une unité mémoire (MMU) qui permet de manipuler des adresses virtuelles, *i.e.* de faire un renommage, transparent pour l'utilisateur, entre les adresses virtuelles du programme et les adresses réelles en mémoire.

Cela permet à chaque programme de choisir ses adresses indépendemment des autres programmes (qui peuvent être exécutés en même temps sur la même machine avec les mêmes adresses virtuelles mais des adresses réelles différentes).

Du point de vue de l'utilisateur, la mémoire est un (grand) tableau dont les indices sont les adresses.

Les zones mémoire

Slide 4

Adresses hautes	Stack
	↓ ↑
	Données dynamiques Allouée dynamiquement
	Données statiques Modifiable
	Texte (programme) Non écrivable
Adresses basses	Réservé au système

Les registres du MIPS

Slide 5

Le MIPS comporte 32 registres généraux interchangeables, sauf

- le registre `zero` qui vaut toujours `0`, même après une écriture.
- Le registre `ra`, utilisé implicitement par certaines instructions pour sauver l'adresse de retour avant un saut.

Les autres registres ont des utilisations préférentielles, mais cela n'est strict que pour communiquer avec d'autres programmes (par exemple fournir ou utiliser des programmes en librairies) :

- passage (`a0`, ... `a3`) et retour (`v0`, `v1`) des arguments ;
- registres temporaires sauvegardés (`s0`, .. `s7`) ou non (`t0`, ... `t9`) par les appels de fonction.
- pointeurs de pile `sp`, `fp` ou de donnée `gp` ;
- réservés par l'assembleur `at` et le système `k0`, `k1`.

Le jeu d'instructions

n	une constante entière
ℓ	une étiquette (adresse)

r	nom de registre
a	absolu (n ou ℓ)
o	opérande (r ou a)

Slide 6

La plupart des instructions suivent le modèle

– **add** r_1, r_2, o qui place dans r_1 la valeur $r_2 + o$.

Les instructions qui interagissent avec la mémoire sont uniquement les instructions `load` et `store`.

– **lw** r_1, $n(r_2)$ place dans r_1 le mot contenu à l'adresse $r_2 + n$.

– **sw** r_1, $n(r_2)$ place r_1 dans le mot contenu à l'adresse $r_2 + n$.

Les instructions de contrôle conditionnel ou inconditionnel :

– **bne** r, a, ℓ saute à l'adresse ℓ si r et a sont différents,

– **jal** o qui sauve $pc + 1$ dans `ra` et saute à l'étiquette o.

Slide 7

Syntaxe	Effet
move r_1, r_2	$r_1 \leftarrow r_2$
add r_1, r_2, o	$r_1 \leftarrow o + r_2$
sub r_1, r_2, o	$r_1 \leftarrow r_2 - o$
mul r_1, r_2, o	$r_1 \leftarrow r_2 \times o$
div r_1, r_2, o	$r_1 \leftarrow r_2 \div o$
and r_1, r_2, o	$r_1 \leftarrow r_2$ land o
or r_1, r_2, o	$r_1 \leftarrow r_2$ lor o
xor r_1, r_2, o	$r_1 \leftarrow r_2$ lxor o
sll r_1, r_2, o	$r_1 \leftarrow r_2$ lsl o
srl r_1, r_2, o	$r_1 \leftarrow r_2$ lsr o
li r_1, n	$r_1 \leftarrow n$
la r_1, a	$r_1 \leftarrow a$

Syntaxe	Effet
lw $r_1, o(r_2)$	$r_1 \leftarrow tas.(r_2 + o)$
sw $r_1, o(r_2)$	$r_1 \rightarrow tas.(r_2 + o)$
slt r_1, r_2, o	$r_1 \leftarrow r_2 < o$
sle r_1, r_2, o	$r_1 \leftarrow r_2 \leq o$
seq r_1, r_2, o	$r_1 \leftarrow r_2 = o$
sne r_1, r_2, o	$r_1 \leftarrow r_2 \neq o$
j o	$pc \leftarrow o$
jal o	$ra \leftarrow pc + 1 \wedge pc \leftarrow o$
beq r, o, a	$pc \leftarrow a$ si $r = o$
bne r, o, a	$pc \leftarrow a$ si $r \neq o$
syscall	appel système
nop	ne fait rien

Les appels systèmes

Ils permettent l'interaction avec le système d'exploitation, et en dépendent. Le numéro de l'appel système est lu dans `v0` (attention, ce n'est pas la convention standard). Selon l'appel, un argument supplémentaire peut être passé dans `a0`.

Le simulateur SPIM implémente les appels suivants :

Slide 8

Nom	N^o	Effet
`print_int`	1	imprime l'entier contenu dans `a0`
`print_string`	4	imprime la chaîne en `a0` jusqu'à '\000'
`read_int`	5	lit un entier et le place dans `v0`
sbrk	9	alloue `a0` bytes dans le tas,
		retourne l'adresse du début dans `v0`.
exit	10	arrêt du programme en cours d'exécution

Le jeu d'appel système dépend du système d'exploitation.

Langage d'assembleur et langage machine

Le langage d'assembleur est un langage symbolique qui donne des noms aux instructions (plus lisibles que des suites de bits). Il permet aussi l'utilisation d'étiquettes symboliques et de pseudo-instructions et de modes d'adressage surchargés.

Slide 9

Le langage machine est une suite d'instructions codées sur des mots (de 32 bits pour le MIPS). Les instructions de l'assembleur sont expansées en instructions de la machine à l'édition de lien. Les étiquettes sont donc résolues et les pseudo-instructions remplacées par une ou plusieurs instructions machine.

L'assemblage est la traduction du langage d'assembleur en langage machine. Le résultat est un fichier object qui contient, en plus du code, des informations de relocation qui permettent de lier *(linker)* le code de plusieurs programmes ensemble.

Pseudo-instructions

La décompilation du langage machine en langage assembleur est facile. Elle permet de présenter les instructions machine (mots de 32 bits) sous une forme plus lisible.

Exemple d'expansion (présentée sous une forme décompilée).

Slide 10

Assembleur	Langage machine	Commentaire
blt r, o, a	**slt** $1, r, o **bne** $1, $0, a	Justifie le registre `at` (`$1`) réservé par l'assembleur.
li $t0, 400020	**lui** $1, 6 **ori** $8, $1, 6804	charge les 16 bits de poids fort puis les 16 bits de poids faible
add $t0, $t1, 1	**addi** $8, $9, 1	addition avec une constante
move $t0, $t1	**addu** $8, $0, $9	addition "unsigned" avec zéro

La zône `Text Segment` se présente sous la forme tabulée :

```
Adresse         Instruction Forme          Adresse              Instruction
machine         machine     décompilée     symbolique           assembleur

[0x00400000]    0x0109082a  slt $1, $8, $9                  ; 2: blt $t0, $t1, trois
[0x00400004]    0x14200003  bne $1, $0, 12 [trois-0x00400004]
[0x00400008]    0x3c01003d  lui $1, 61                      ; 4: lui $t0, 4000020
[0x0040000c]    0x34280914  ori $8, $1, 2324
[0x00400010]    0x21280001  addi $8, $9, 1                  ; 6: add $t0, $t1, 1
[0x00400014]    0x00094021  addu $8, $0, $9                 ; 8: move $t0, $t1
```

On peut donc voir qu'à l'instruction assembleur **blt** $t0, $t1, trois correspond deux instruction machine 0x0109082a et 0x14200003, qui se désassemble en **slt** $1, $8, $9 et **bne** $1, $0, 12 [trois −0x00400004].

La pico-machine

C'est une machine inventée (Pierre Weis et Xavier Leroy) pour des raisons pédagogiques.

C'est un modèle simplifié de machine RISC, très proche du MIPS.

Son jeu d'instructions est (pas tout à fait) un sous-ensemble du MIPS.

Slide 11

Une grosse différence de syntaxe... Dans le code 3 adresses, source et destination sont inversés :

Code MIPS	Pico code	signification
nom r_1, o, r_2	nom r_2, o, r_1	$r_1 \leftarrow o$ nom r_2

Exemple de programme assembleur

hello.spi

Slide 12

```
        .data
hello:  .asciiz "hello\n"     # hello pointe vers "hello\n\0"
        .text
        .globl __start
__start :
        li      $v0, 4        # la primitive print_string
        la      $a0, hello    # a0 l'adresse de hello
        syscall
```

Le programme est assemblé, chargé puis exécuté par :

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

Par convention le programme commence à l'étiquette __start.

Si on retire l'option −notrap, le chargeur ajoute un prélude qui se branche à l'étiquette main (remplacer alors __start par main).

Slide 13

Slide 14

if-then-else

On utilise des sauts conditionnels et inconditionnels :

Pascal la fonction minimun

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

Assembleur Mips

```
        blt   $t1, $t2, Then     # si t1 >= t2 saut à Then
        move  $t3, $t2           # t3 := t1
        j     End                # saut à Fi
Then:   move  $t3, $t1           # t3 := t2
End:                             # suite du programme
```

Boucles

Pascal : calcule dans t2 = 0 la somme des entiers de 1 à t1

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

Slide 15

Programme équivalent

```
While:
    if t1 <= 0 then goto End
    else
      begin
        t2 := t2 + t1;
        t1 := t1 − 1;
        goto While
      end;
End:
```

Code Mips

```
While:
    ble  $t1, $0, End


    add  $t2, $t2, $t1
    sub  $t1, $t1, 1
    j    While

End:
```

Appel de procédure

Slide 16

Pour appeler une procédure, il faut sauver l'adresse de retour à l'appel, en général dans $ra, pour revenir après l'appel.

À la fin d'une procédure, on retourne à l'appelant en sautant à l'adresse contenu dans $ra.

Par exemple on définit une procédure writeln qui imprime un entier puis un retour à la ligne.

```
writeln :                # l'argument est dans a0
        li   $v0, 1      # le numéro de print_int
        syscall          # appel système
        li   $v0, 4      # la primitive  print_string
        la   $a0, nl     # la chaîne "\n"
        syscall
        j    $ra         # saut à l'adresse ra
```

Le programme complet

Slide 17

```
        .data                # le tas
nl:     .asciiz  "\n"        # la chaîne "\n"
        .text                # la zône code
        .globl  __start
__start :
        li    $a0, 1         # a0 <- 1
        jal   writeln        # ra <- pc+1; saut à writeln
        li    $a0, 2         # on recommence avec 2
        jal   writeln
        j     Exit           # saut à la fin  du programme
writeln :
        ...
Exit:                        # fin du programme
```

Noter la différence entre les instructions j et **jal**.

Procédures récursives

Slide 18

Lorsqu'une procédure est récursive plusieurs appels imbriqués peuvent être actifs simultanément :

$$f_n \left\{ \begin{array}{l} a_0 \leftarrow v_0 - 1 \\ f_{n-1} \left\{ \begin{array}{l} a_0 \leftarrow v_0 - 1 \\ \ldots \\ v_0 \leftarrow v_0 \times a_0 \end{array} \right. \\ v_0 \leftarrow v_0 \times a_0 \end{array} \right.$$

C'est bien sûr le même code qui est utilisé pour f_n et f_{n-1}. Il faut donc sauver avant l'appel récursif la valeur des registres qui seront (ou risque d'être) modifiés par l'appel récursif et dont les valeurs seront encore nécessaires (lues) au retour de l'appel.

On utilise la pile.

La pile

Par convention, la pile grossit vers les adresses décroissantes. Le registre sp pointe vers le dernier mot utilisé.

Pour sauver un registre r sur la pile

```
sub  $sp, $sp, 4    # alloue un mot sur la pile
sw   r, 0($sp)      # écrit r sur le sommet de la pile
```

Slide 19

Pour restaurer un mot de la pile dans un registre r

```
lw   r, 0($sp)      # lit le sommet de la pile dans r
add  $sp, $sp, 4    # désalloue un mot sur la pile
```

En général, on alloue et désalloue l'espace en pile par blocs pour plusieurs temporaires à la fois.

On peut aussi utiliser la pile pour allouer des tableaux si leur durée de vie le permet.

Conventions d'appel

En général :

- les arguments sont passés dans les registres `a0` à `a3` puis dans la pile.
- la ou les valeurs de retour dans `v0` et `v1`.

De même :

Slide 20

- les registres t_i sont sauvés par l'appelant (l'appelé peut les écraser)
- les registres s_i sont sauvés par l'appelé (qui doit les remettre dans l'état où il les a pris)

Mais ce n'est qu'une convention ! (celle proposé par le fabriquant)

La respecter permet de comminiquer avec d'autres programmes (qui la respectent également).

Choisir une autre convention est possible tant que l'on n'intéragit pas avec le monde extérieur.

Exemple : calcul de la factorielle

Slide 21

L'argument est dans `a0`, le résultat dans `v0`.

```
fact:    blez $a0, fact_0       # si a0 <= 0 saut à fact_0
         sub  $sp, $sp, 8       # réserve deux mots en pile
         sw   $ra, 0($sp)       # sauve l'adresse de retour
         sw   $a0, 4($sp)       # et la valeur de a0
         sub  $a0, $a0, 1       # décrément a0
         jal  fact              # v0 <- appel récursif (a0-1)
         lw   $a0, 4($sp)       # récupére a0
         mul  $v0, $v0, $a0     # v0 <- a0 * v0
         lw   $ra, 0($sp)       # récupére l'adresse de retour
         add  $sp, $sp, 8       # libère la pile
         j    $ra               # retour à l'appelant

fact_0:  li   $v0, 1            # v0 <- 1
         j    $ra               # retour à l'appelant
```

Allocation dans le tas

Slide 22

Allocation statique Un espace est réservé au chargement.

```
Tableau:                  # adresse symbolique sur le début
        .align 2          # aligner sur un mot (2^2 bytes)
        .space 4000       # taille en bytes
```

Allocation dynamique En cours de calcul on peut demander au système d'agrandir le tas.

En SPIM, l'appel système numéro 9 prend la taille dans `v0` et retourne le pointeur vers le début de bloc dans `a0`.

```
malloc:                   # procédure d'allocation dynamique
        li $v0, 9         # appel système n. 9
        syscall           # alloue une taille a0 et
        j  $ra            # retourne le pointeur dans v0
```

Gestion de l'allocation

En pratique, on alloue un gros tableau `Tas` (statiquement ou dynamiquement) dans lequel on s'alloue des petits blocs.

Par exemple on réserve un registre pour pointer sur le premier emplacement libre du tas.

Slide 23

```
__start :
        la   $t8, Tas          # on réserve le registre t8
        ...
array :
        sw   $a0, ($t8)        # écrit la taille dans l'entête
        add  $v0, $t8, 4       # v0 <- adresse de la case 0
        add  $t8, $v0, $a0     # augmente t8 de la taille +1
        j    $ra
```

(On ignore ici les problèmes de débordement —qu'il conviendrait de traiter.)

Le byte-code

C'est un jeu d'intructions inventé pour une machine virtuelle (ou abstraite) qui est alors interprété par un programme, lui même compilé dans l'assembleur de la machine réelle.

Avantages : le byte-code est indépendant de l'architecture, il est portable (fonctionne sur différentes machines).

Slide 24

Le jeu d'instructions peut être mieux adpapté au langage compilé.

Inconvénient : l'interprétation du byte-code ajoute au moins un facteur 5 en temps d'exécution (pour de très bons byte-codes).

Exemples : Java, Ocaml.

Micro-code : Dans une machine réelle, les instructions sont en fait interprétées par du micro-code (qui lui est vraiment câblé).

Exercices

Utilisation de SPIM en td Spim est installé sur les machines de la salle 33. Dans les autres salles, il faut éventuellement se loger à distance sur des machines de cette salle linux. (Voir les détails dans le version HTML.)

Slide 25

La fonction de Fibonnaci

Écrire la fonction fib en assembleur MIPS. On pourra commencer par une version simple, inefficace.

Écrire également une version optimisée qui mémorise les deux appels précédents dans des registres. On pourra lire l'entier au clavier et l'imprimer à l'écran.

Slide 26

Compilation des expressions arithmétiques

On se propose de compiler les expressions arithmétiques en langage MIPS. On se limitera à des expressions avec les 4 opérations de bases, les constantes entières et une seule variable (qui sera lue au début du programme dans le flux d'entrée).

Pour simplifier, une expression arithmétique sera donnée par sa syntaxe abstraite, et on prendra la définition de type suivante (**imposée**) :

expression.mli

```
type expression =
  | Binexp of binop * expression * expression
  | Int of int
  | X (* variable *)
and binop = Plus | Minus | Times | Div;;
```

Pour éviter les problèmes d'entrée sortie, on écrira le programme à compiler sous la forme d'un fichier source. Par exemple, pour

Slide 27

compiler l'expression ((x * x) + (1 - x * 2)), on écrira le source dans un fichier

arith.ml

```
open Expression;;
open Arithmetique;;
let e =
  Binexp (Plus,
          Binexp (Times, X, X),
          Binexp (Minus,
                  Int 1,
                  Binexp (Times, X, Int 2)))
in compile e;;
```

On fabriquera un exécutable `arith.out` par la commande :

```
ocamlc -o arith.out expression.mli arithmetique.ml arith.ml
```

puis le fichier de code Spim compilé en lançant cette commande

```
./arith.out > arith.spi
```

Slide 28

Le programme compilé `arith.spi` devra être exécutable sur la machine SPIM par la commande :

```
spim -notrap -file arith.spi
```

standard, évalue l'expression, et ainsi de suite tant que l'entier lu est non nul.

L'exercice consiste donc a écire un fichier source `arithmetique.ml` qui implémente la fonction `compile : epxressions -> unit` qui imprime dans la sortie standard le code spim correspondant à l'évaluattion de l'expression reçue en argument.

Dans un premier temps, on ne recherchera pas l'efficacité, mais la simplicité. Par exemple :

- on effectue toutes les opérations dans les registres a_1 et a_2.
- on utilise le registre a_3 pour la valeur de la variable, et
- on sauve les arguments dans la pile lors d'un calcul auxiliaire.
- Commencer par une version de la fonction compile qui ne traire que les cas simples, *i.e.* où l'expression arithmétique et une

constante, une variable, ou ne contient qu'une seule opération.

Extensions

- Cummuler les résultats auxilliaires, et afficher la somme à la fin.
- utiliser les registres au mieux et la pile seulement si nécessaire.

Slide 29

2

Langages
Analyse lexicale
Expressions régulières
Automates.

En amont de la chaîne de compilation

$$\mathrm{SC}\ \boxed{\overset{\text{Analyses}}{\xrightarrow{\text{Lex.}}\ \mathrm{SL}\ \xrightarrow{\text{Gram.}}}}\ \mathrm{SA}\ \overbrace{\xrightarrow{\text{Génér. Canon.}}}^{(1)}\ \mathrm{CI}\ \overbrace{\xrightarrow{\text{Sélect. instr.}}\ \mathrm{CA}\ \xrightarrow{\text{Alloc. regist.}}}^{(2)}\ \mathrm{CA}$$

Slide 1

Analyse en deux passes :

1. Analyse lexicale : transforme une suite de caractères en une suite de lexèmes (mots).
2. Analyse grammaticale : transforme une suite de lexèmes en une représentation arborescente (syntaxe abstraite).

Enjeux

Slide 2

Les analyses lexicales et grammaticales ont un domaine d'application bien plus large que celui de la compilation. On les retrouve comme première passe dans de nombreuses applications (analyses des commandes, des requêtes, etc,).

Ces deux analyses utilisent de façon essentielle les automates, mais on retrouve aussi les automates dans de nombreux domaines de l'informatique.

Les expressions régulières sont un langage de description d'automates ; elles sont utilisées dans de nombreux outils Unix, et fournies en bibliothèque dans la plupart des langages de programmation.

Note

Slide 3

L'étude détaillée des automates et des grammaires formelles pourrait constituer un cours à part entière.

Nous nous contentons ici de la présentation formelle minimale, avec comme but :
- d'expliquer le fonctionnement des analyseurs de façon à pouvoir écrire soi-même des analyseurs lexicaux ou grammaticaux.
- de se familiariser aussi avec les expressions régulières et les automates, car on les retrouve ensuite fréquemment en informatique.

Le but du cours n'est pas d'écrire le moteur d'un analyseur, ni de répertorier toutes les techniques d'analyses.

Les langages formels

Slide 4

On se donne un ensemble Σ appelé alphabet, dont les éléments sont appelés caractères.

Un mot (sur Σ) est une séquence de caractères (de Σ).

On note ϵ le mot vide, uv la concaténation des mots u et v (la concaténation est associative avec ϵ pour élément neutre).

Σ^* est l'ensemble des mots sur Σ

Σ^+ est l'ensemble des mots non vides.

Un langage sur Σ est un sous-ensemble L de Σ^*.

Si U et V sont des langages sur Σ, on note UV l'ensemble des mots obtenus par la concaténation d'un mot de U et d'un mot de V ; U^* (resp. U^+), l'ensemble des mots obtenus par la concaténation d'un nombre arbitraire, éventuellement nul (resp. non nul) de mots de U.

Exemples

Slide 5

1. Σ_1 est l'alphabet français et L_1 l'ensemble des mots du dictionnaire français avec toutes leurs déclinaisons.
2. Σ_2 est L_1 et L_2 est l'ensemble des phrases grammaticalement correctes de la langue française.

   Ou bien L'_2 le sous-ensemble des palindromes de L_2.
3. Σ_3 est l'ensemble des caractères ASCII, et L_3 est composé de tous les mots clés de pseudo pascal, de l'ensemble des symboles, de l'ensemble des identificateurs et de l'ensemble des entiers.
4. Σ_4 est L_3 et L_4 est l'ensemble des programmes pseudo pascal.
5. Σ est $\{a, b\}$ et L est l'ensemble $\{a^n b^n \mid n \in \mathbb{N}\}$ (sous ensemble des expressions bien parenthésées).

Expressions régulières

Slide 6

Les expressions régulières sont des opérateurs permettant de décrire certains langages (les langages réguliers) définis sur un même alphabet Σ.

On note a, b, *etc.* des lettres de Σ, M et N des expressions régulières, $[M]$ le langage associé à M.

- Une lettre de l'alphabet a désigne le langage $\{a\}$.
- Epsilon : ϵ désigne le langage $\{\epsilon\}$.
- Concaténation : $M \cdot N$ désigne le langage $[M][N]$.
- Alternative : $M \mid N$ désigne le langage $[M] \cup [N]$.
- Répétition : M^* désigne le langage $[M]^*$.

On ajoute également du sucre syntaxique (*i.e.* sans changer l'expressivité) :

- $[abc]$ pour $(a \mid b \mid c)$ et $[a_1 - a_2]$ pour $\{c \in \Sigma, a_1 \leq c \wedge c \leq a_2\}$, (on suppose ici que l'alphabet est ordonné)
- $M?$ pour $M \mid \epsilon$ et M^+ pour MM^*.

Analyse lexicale

Slide 7

On veut couper une phrase (suite de caractères) en lexèmes.

On décrit les lexèmes par des expressions régulières. Exemple :

1. Les mots clés : `"let"`, `"in"`
2. Les variables : `['a'-'z']+ ['0'-'9']*`
3. Les entiers : `['0'-'9']+`
4. Les symboles : `'('`, `')'`, `'+'`, `'*'`, `'='`
5. Le lexème vide : `(' ' | '\n')`

On peut représenter les lexèmes par un type concret :

```
type token =
    LET | IN | ..                        (* mots clés *)
  | VAR of string                        (* variables *)
  | INT of int                           (* entiers *)
  | LPAREN | RPAREN | PLUS | TIMES | EQUAL (* symboles *)
```

Analyse lexicale (suite)

Slide 8

Principe C'est un algorithme qui
- prend une suite de caractère
- la transforme en une suite de lexème, et retourne les caractères non reconnus.

Problème Il y a des ambiguités. Par exemple :
- `let` pourrait être reconnu comme une variable.
- `lettre` pourrait aussi être reconnu comme la séquence `LET; VAR "tre"` ou encore `VAR "let"; VAR "tre"`

Règles de priorité

Slide 9

On choisit par priorité :

1. Le lexème le plus long,
2. L'ordre de définition.

Ainsi la phrase `let lettre = 3 in 1 + fin` produit la suite de lexèmes :

```
LET; VAR "lettre"; EQUAL; IN; INT 1; PLUS; VAR "fin"
```

ocamllex

On écrit un fichier `lexeur.mll` qui est compilé en un fichier source `lexeur.ml` par le programme `ocamllex`.

Slide 10

```
{ (* prelude : code Ocaml reproduit verbatim *) }
rule entree_1 = parse
  regexp_1  { (* code ocaml qui retourne un lexème *) }
| regexp_2  { (* code ocaml qui retourne un lexème *) }
| ...
and entree_2
...
{ (* postlude : code Ocaml reproduit verbatim *) }
```

Pour construire les lexèmes, on récupére la chaîne reconnue par `Lexing.lexeme lexbuf`. Attention : le nom de la variable `lexbuf` est fixé par convention.

Exemple

Fichier essai.mll

Slide 11

```
{ open Lexing
  type token = IDENT of string | INT of int |  LET | IN
       | PLUS | TIMES | EQUAL | LPAREN | RPAREN
  exception Eof  exception Illégal  }
rule token = parse
   [' ' '\ t' '\ n']          { token lexbuf }  (* skip *)
|  "let"                      { LET }
|  "in"                       { IN }
|  [' a'-'z'] + ['0'-'9'] *   { IDENT (lexeme lexbuf)}
|  ['0'-'9']+                 { INT(int_of_string
                                      (lexeme lexbuf)) }
|  '=' { EQUAL } | '+' { PLUS } | '*'  { TIMES }
|  eof { raise  Eof }  | _ { raise  Illégal  }
```

Slide 12

Exemple (suite)

Fichier essai.mll

```
{ let lex buf =
    let rec lex () =
      try let h = token buf in h :: lex () with Eof -> [] in
    lex ()
  let lex_string s = lex (Lexing.from_string s)
  let lex_chan c = lex (Lexing.from_channel c) }
```

Compilation en deux étapes :

1. Fabrication du fichier essai.ml
   ```
   ocamllex essai.mll
   ```
2. Compilation normale du fichier essai.ml
   ```
   ocamlc -c essai.ml
   ```

Slide 13

Remarques

Utilisation de la récursion `token lexbuf` pour une production vide.

L'utilisation de plusieurs règles revient à combiner plusieurs lexeurs indépendants mais qui travaillent sur la même entrée. Application à l'analyse des chaînes de caractères (inefficace) :

```
rule token = parse
    [' ' '\t' '\n']  { token lexbuf }
  | '"'             { STRING(String.concat "" (string lexbuf)) }
  | ['0'-'9']+      { INT(int_of_string (lexeme lexbuf)) }
  | eof             { raise EOF }
and string = parse
  | '"'             { [] }
  | '\\' '"'        { "\"" :: string lexbuf }
  | eof             { raise Unterminated_string }
  | _               { let c = lexeme_char lexbuf 0 in
                      String.make 1 c :: string lexbuf }
```

Récupération des erreurs

Slide 14

En cas d'erreur, il est nécessaire d'indiquer un minimum d'informations, telles que la position de l'erreur, et/ou la dernière séquence non reconnue :

– La position du dernier lexème dans le flux d'entrée est déterminée par `Lexing.lexeme_start lexbuf` et `Lexing.lexeme_end lexbuf`.
– La sous-chaîne correspondante est (`Lexing.lexeme lexbuf`).

Comment fonctionnne Ocamllex ?

Slide 15

Un exemple de compilation :

1. Chaque expression régulière est compilée en un automate,
2. L'ensemble des automates sont fusionnés en un seul,
3. L'automate résultant est determinisé.
4. L'automate est minimisé.

Automates finis déterministes

Slide 16

Un automate fini déterministe M est un quintuple $(Q, \Sigma, \delta, q_0, F)$ où

- Σ est un alphabet ;
- Q est un ensemble fini d'états ;
- $\delta : Q \times \Sigma \to Q$ est la fonction (partielle) de transition ;
- q_0 est l'état initial ;
- F est un ensemble d'états finaux.

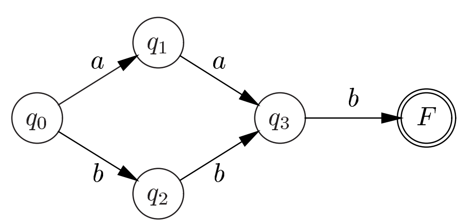
On peut étendre δ sur $Q \times \Sigma^* \to Q$ par $\begin{cases} \delta(q, \epsilon) = q \\ \delta(q, aw) = \delta(\delta(q, a), w) \end{cases}$

Le langage $L(M)$ reconnu par l'automate M est l'ensemble $\{w \mid \delta(q_0, w) \in F\}$ des mots permettant d'atteindre un état final à partir de l'état initial.

Exemple

Slide 17

Automate déterministe :



- $\Sigma = \{a, b\}$
- $Q = \{q_0, q_1, q_2, q_3, F\}$
- $\delta = \{(q_0, a) \mapsto q_1, (q_0, b) \mapsto q_2, (q_1, a) \mapsto q_3, (q_2, b) \mapsto q_3, (q_3, b) \mapsto F\}$
- État initial q_0 et un seul état final F.

Quel est le langage reconnu ? $L = \{aab, bbb\}$

Automates finis non-déterministes

Slide 18

Il existe plusieurs transitions avec la même étiquette, mais vers des états différents. On autorise aussi des ϵ-transitions.

Formellement, comme ci-dessus, excepté $\delta : Q \times (\Sigma \cup \{\epsilon\}) \to 2^Q$.

On étend δ sur $Q \times \Sigma^* \to Q$ par $\begin{cases} q \in \delta(q, \epsilon) & \text{(en plus)} \\ \delta(q, aw) = \bigcup_{q' \in \delta(q,a)} \in \delta(q', w) \end{cases}$

Le langage $L(M)$ reconnu par un automate non déterministe est $\{w \mid \delta(q_0, w) \cap F \neq \emptyset\}$

Automate d'une expression régulière

Slide 19

L'alphabet Σ est fixé.

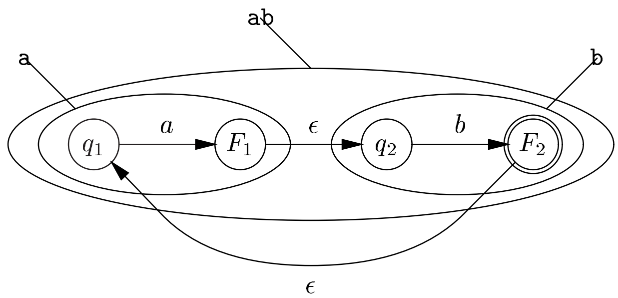
On associe à une expression régulière M un automate non déterministe (Q, δ, s, F) défini récursivement par :

- $[\![a]\!] = (\{s, f\}, \{s \overset{a}{\mapsto} f\}, s, \{f\})$
- $[\![\epsilon]\!] = (\{s, f\}, \{s \overset{\epsilon}{\mapsto} f\}, s, \{f\})$
- $[\![M \mid M']\!] = (Q \cup Q' \cup \{s''\}, \delta \cup \delta' \cup \{s'' \overset{\epsilon}{\mapsto} s, s'' \overset{\epsilon}{\mapsto} s'\}, s'', F \cup F')$
- $[\![M \cdot M']\!] = (Q \cup Q', \delta \cup \delta' \cup \{f \overset{\epsilon}{\mapsto} s', f \in F\}, s, F')$
- $[\![M^*]\!] = (Q, \delta \cup \{f \overset{\epsilon}{\mapsto} s, f \in F\}, s, \{s\})$

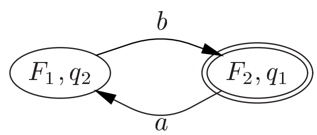
Exemple

Automate non-déterministe associé à l'expression régulière (`ab`)$*$.

Slide 20



Déterminisation de l'automate (egale à l'ϵ-fermeture, ici) :



Déterminisation d'un automate

Slide 21

Pour tout automate non déterministe $M = (Q, \Sigma, \delta, q_0, F)$, il existe un automate déterministe M' qui reconnaît le même langage.

Une solution est $M' = (2^Q, \Sigma, \delta', \{q_0\}, \{F\})$ où

$$\delta'(q', a) = \texttt{fermeture}\left(\bigcup_{q \in q'} \delta(q, a)\right)$$

où $\texttt{fermeture}(q')$ est le plus petit ensemble contenant q' fermé par ϵ-transition :

$$q \in \texttt{fermeture}(q') \Longrightarrow \delta(q, \epsilon) \subset \texttt{fermeture}(q')$$

Se calcule par point fixe !

L'automate déterministe associé à Q a $2^{|Q|}$ états, mais en général, seulement les états atteignables depuis $\{q_0\}$ nous intéressent. Heureusement, en pratique[a], cet ensemble est

[a] avec quelques précautions

relativement petit, souvent de l'ordre de $|Q|$.

Slide 22

Minimisation de l'automate

Tout automate admet un automate équivalent (qui reconnaît le même langage) avec un nombre d'états minimum.

Principe On peut le calculer à partir d'un automate déterministe par superposition d'états. Deux états sont équivalents si les suffixes de L reconnus par l'automate à partir de ces états sont égaux.

Slide 23

En particulier, $\forall q \in Q, \forall w \in \Sigma^*, \begin{cases} q \in F \wedge q' \notin F \Rightarrow q \not\mathcal{R}\, q' \\ \delta(q,w) \not\mathcal{R}\, \delta(q',w) \Rightarrow q \not\mathcal{R}\, q' \end{cases}$

L'automate minimal est obtenu en prenant l'équivalence la plus large. C'est le plus grossier raffinement de la partition initiale $\mathcal{P}_0 \triangleq \{F, Q \setminus F\}$ qui soit stable par image inverse $A \mapsto \delta^{-1}(A,a)$ pour tout $A \in \mathcal{P}$ et $a \in \Sigma$.

Calcul par une variation sur *divide and conquer*

Quelques indications

Slide 24

Étant donné un langage L sur Σ, on définit la relation d'équivalence $\mathcal{R}$ dans Σ^* par $x \mathcal{R} y$ ssi $\forall z \in \Sigma^*, xz \in L \Leftrightarrow yz \in L$. Cette relation est fermée par suffixe, *i.e.* si $x \mathcal{R} y$ alors $\forall z \in \Sigma^*, xz \mathcal{R} yz$.

On montre que les trois propriétés suivantes sont équivalentes (par exemple, dans l'ordre $(1) \Rightarrow (2) \Rightarrow (3) \Rightarrow (1)$).

(1) Le langage L défini sur Σ est reconnaissable par un automate déterministe.

(2) L est une partition finie d'une relation d'équivalence fermée par suffixe.

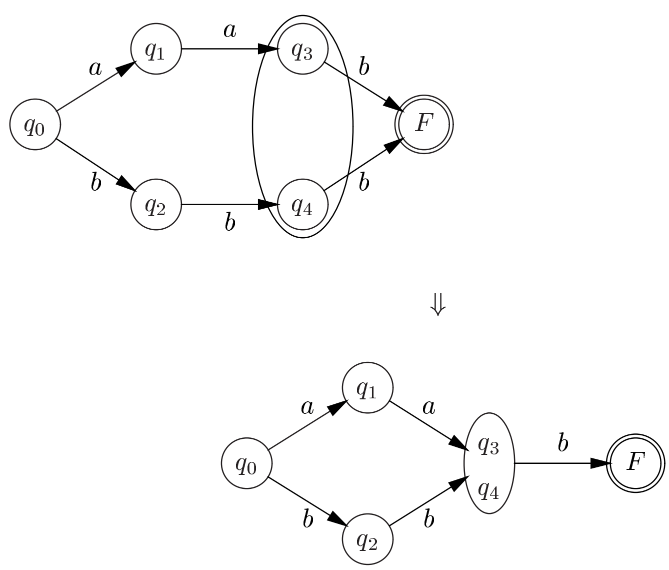
(3) La partition de la relation $\mathcal{R}$ ci-dessus est finie.

Pour montrer $(3) \Rightarrow (1)$ on considère l'automate dont les états sont les classes d'équivalence pour $\mathcal{R}$, la fonction de transition est définie par $\delta(\bar{x}, a) = \overline{xa}$, l'état initial est le mot vide et les états finaux sont $\{\bar{x}, x \in L\}$.

Cet automate est minimal, car la relation $\mathcal{R}$ ne peut pas avoir plus de classes qu'un automate reconnaissant L.

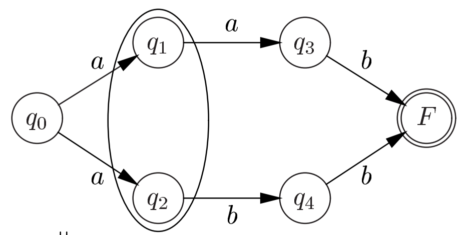
Exemple de minimisation

Slide 25

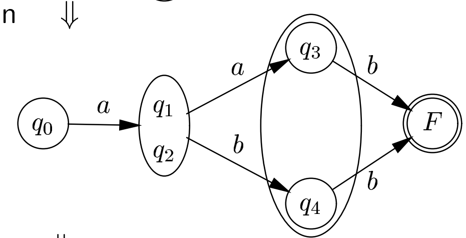


Déterminisation + minimisation

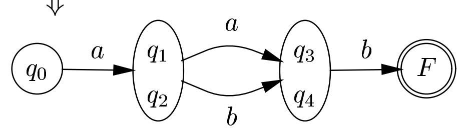
Slide 26



Déterminisation $\Downarrow$



Minimisation $\Downarrow$



Calcul de l'automate minimal

Slide 27

1. On considère un raffinement $\mathcal{P}$ de la partition $\mathcal{P}_0$. Un élément A de $\mathcal{P}$ est dit stabilisé si $\mathcal{P}$ est stable par les images inverses $\delta^{-1}(A, a)$ pour tout a.
2. On choisit un élément A non stabilisé de $\mathcal{P}$.
3. Pour chaque $a \in \Sigma$, on raffine la partition $\mathcal{P}$ afin qu'elle soit stable par les $\delta^{-1}(A, a)$.

   Pour cela, il faut diviser chaque composante $B \in \mathcal{P}$ en $B_1 = B \cap \delta^{-1}(A, a)$ et $B_2 = B \setminus \delta^{-1}(A, a)$.
4. Si B est non stabilisé, alors il faut stabiliser B_1 et B_2. Sinon, il suffit de stabiliser B_1 pour que B_2 le soit, ou inversement. On choisit de stabiliser la composante de plus petit cardinal.

Avec de (très) bonnes structures de donnée, la complexité est en $m \log(n)$ où m et n sont les nombres d'arcs et de nœuds de l'automate initial.

Représentation d'un automate

Slide 28

La fonction δ de domaine fini $Q \times \Sigma$ peut être représentée par une matrice de dimension 2 dont les éléments sont les états (pour un automate déterministe) ou ensembles d'états (pour un automate non-déterministe) définissant δ.

On peut choisir une représentation pleine (tableau de tableaux) ou creuse (tableau de listes) selon la situation. La première est bien sûr plus efficace en temps mais plus gourmande en espace.

Dans le cas d'une matrice creuse, on peut aussi économiser de l'espace en superposant les tableaux de domaines disjoints (astuce souvent utilisée en pratique qui a l'efficacité en temps de la représentation pleine et souvent l'efficacité en espace de la représentation creuse).

Plutôt que d'interpréter l'automate, on peut le représenter directement par une définition récursive avec filtrage.

Exemple

Slide 29

Automate minimal représentant l'expression régulière $(\mathtt{ab})^*$:

Les états sont des entiers. Les étiquettes sont aussi des entiers 0 et 1 représentant les transitions `a` et `b`. Par convention, l'état initial est 0. Dans la table -1 représente une transition interdite.

```
let sigma = function
    'a' -> 0 | 'b' -> 1 | _ -> failwith "sigma"
type automat= {final : bool array; delta : int array array}
let automat =
  { final = [| true; false ; |];
    delta = [| (* étiquettes :    a    b  *)
               (* état 0 *)  [|   1;  -1  |];
               (* état 1 *)  [|  -1;   0  |]; |] }
exception Exit;;
```

Interprétation des tables

Slide 30

```
let run automat input =
  let i = ref 0 in
  let matched = ref (-1) in
  if automat.final.(0) then matched := 0;
  let q = ref 0 in
  (try while !i < String.length input do
    let a = sigma (input.[! i]) in
    let q' = automat.delta.(!q).(a) in
    if q' < 0 then raise Exit;
    q := q';
    incr i;
    if automat.final.(q') then matched := !i;
  done with Exit -> ());
  !matched;; (* retourne le dernier caractère reconnu *)
run automat "ababaabb";; (* retourne 4 *)
```

Automate pour lex

Slide 31

La question n'est plus *est-ce la chaîne est reconnaissable* en entier *par une expression régulière*, mais *trouver la chaîne la plus longue reconnue et la première règle (dans l'ordre de définition) la reconnaissant*.

On veut construire le lexeur défini par les deux règles $(e_i\{a_i\})_{i\in I}$ (lire reconnaître l'expression régulière la plus longue parmi les e_i et exécuter l'action a_i en privilégiant l'ordre.)

Pour cela on enrichit on enrichie à la fois la construction de l'automate et son interprétation.

Fabrication de l'automate pour lex

On construit les automates non déterministes $(Q_i, \Sigma, \delta_i, q_i, F_i)_{i \in I}$.

L'automate initial non déterministe pour lex $(Q, \Sigma, \delta, q_0, \{q_F\})$ est défini par :

$$Q \triangleq \bigcup_{i\in I} Q_i \cup q_0 \cup \{p_i, i \in I\}$$
$$\delta \triangleq \bigcup_{i\in I} \delta_i \cup \{q_0 \overset{\epsilon}{\mapsto} q_i \mid i \in I\} \cup \{q \overset{p_i}{\mapsto} q_F \mid q \in F_i, i \in I\}$$

Slide 32

On déterminise cet automate, puis on retire, sur l'automate déterminisé, les transitions *de sortie* $q \mapsto p_i q_F$ lorsqu'il existe une transition $q \mapsto p_j q_F$ avec $i > j$.

En pratique, on représente les transitions de sortie à la place des états finaux, et on construit une structure de la forme :

```
type état = int (* on numérote les règles *)
type rule = int  (* on numérote les états  *)
type automat=
  { final : rule option array; delta : état array array; };;
```

Interprétation de l'automate pour lex

On inteprète l'automate se souvenant de la dernière sortie possible

À chaque pas, on regarde s'il existe une transition possible vers la sortie, auquel cas, on indique la règle p_i permettant cette transition, ainsi que le nombre de lettres lus depuis le début.

Slide 33

Lorsqu'on échoue (plus de transitions possibles) on retourne exécute l'action de la dernière sortie possible (en lui passant le nombre de caractères lus), sinon on échoue.

En Ocaml

Slide 34

Les états finaux sont les transitions

```
let run automat input =
  let i = ref 0 in
  let matched = ref None in
  let q = ref 0 in
  (try while !i < String.length input do
    let a = sigma (input.[! i ]) in
    let q' = automat.delta.(!q).( a) in
    if q' < 0 then raise Exit;
    q := q';
    incr i ;
    begin match automat.final.(q') with
    | Some r -> matched := Some (r, !i)
    | None -> ()
    end;
  done with Exit -> ());
  match !matched with
    Some (r, k) ->
         Some (r, String.sub input 0 k),
         String.sub input k (String.length input - k)
  | None -> None, input;;
```

Langage reconnu par un automate

Slide 35

Le langage reconnu par un automate est rationnel (nombre fini de suffixes, voir aussi le cours suivant), en particulier, un automate ne peut pas reconnaître le langage composé des expressions bien parenthésées.

En effet un automate a un nombre fini d'états. Or la reconnaissance des expressions bien parenthésées demande de se souvenir du nombre de parenthèses déjà reconnues qui peut être arbitraire.

Pour définir un langage de programmation, il nous faut donc des outils plus puissants que les expressions régulières.

Ocamllex

Les actions sont des expressions arbitraires, donc Ocamllex est un langage complet, et la question du langage reconnu par Ocamllex n'as pas de sens si les actions peuvent interagir avec le flux d'entrée.

Slide 36

Toutefois avec une seule règle et sans interaction entre les actifs et le flux d'entrée, on reste à l'intérieur du formalisme précédent.

Reconnaissance des parenthèses On utilise simplement un compteur pour le nombre de parenthèses ouvertes.

```
{ let c = ref 0 }
rule token = parse
    '('  { incr c; token lexbuf }
  | ')'  { if !c > 0 then (decr c; token lexbuf) else false }
  | eof  { (!c = 0) }
{ let test s = token (Lexing.from_string s) }
```

Élimination des commentaires

Slide 37

Une variante toute simple devient un programme intéressant...

```
{ let depth = ref 0 }
rule uncomment = parse
  | "(*<*)"   { incr depth; uncomment lexbuf }
  | "(*>*)"   { if !depth > 0 then decr depth else exit 1;
                uncomment lexbuf }
  | eof       { if !depth > 0 then exit 1 }
  | _         { if !depth = 0 then
                    print_string (Lexing.lexeme lexbuf);
                uncomment lexbuf; }
{ let lexbuf = Lexing.from_channel stdin in
  uncomment lexbuf }
```

Exercice 1 *Parser aussi les chaînes de caractères.* □

Exercices

Slide 38

Exercice 2 *Écrire un lexeur pour le langage Pseudo-Pascal. Les lexèmes sont*

- *Les blancs (blanc, tabulation, retour à la ligne)*
- *Les mots clés :* **var**, *alloc*, **false**, **true**, **read**, **write** , **writeln**, **array**, **of**, **do**, **begin** , **end**, **if**, **then**, **else**, **while**, **type** , **function**, **procedure**, **integer**, **boolean**, **program**.
- *les entiers*
- *les identificateurs (composés de caractères alphabétiques (mininuscule ou majuscules et terminant éventellement par des chiffres.*
- *les lexèmes :* ;; := <> <= >= < > ; ˜ : = − + ∗ / ( ) [ ].

Exercices (suite)

Slide 39

La liste de mots clés, qui peut être assez longue, au risque de conduire à une explostion du nombre d'états de l'automate minimal. Pour éviter cela, on confond les mots clés avec les identificateurs au niveau du lexeur, et on les différencie dans le langage hôte. Écrire une nouvelle version du lexeur, équivalente à la précédente, qui procède ainsi. □

En travaux dirigé, suivre l'énoncé détaillé de Luc Maranget.

Extraire les URL d'un document HTML

Il s'agit d'écrire un lexeur qui extrait d'un document HTML une liste de chaîne de caractères, représentant l'ensemble des liens vers d'autres documents.

Un document HTML contient un ensemble de balises séparées par du texte. Les balises sont entre les caractères < et >). On pourra considérer que ces caractères n'apparaissent nulle part ailleurs.

1. En s'inspirant du lexeur qui élimine les commentaires, écrire un lexeur qui ne retient que les balises HTML.
2. Raffiner le lexeur pour n'imprimer que les URL des balises de type `A`.

Slide 40

3. Pour simplifier on pourra d'abord considérer que
   – Les chaînes ne contiennent pas la clé `href`.
   – Les chaînes ne contiennent pas le caractère `'"'`.
4. Relacher les contraintes précédentes.
5. On pourra aussi considérer les liens "src" dans les ancres de type `IMG`.

Analyse des URL

Une URL est elle-même structurée : elle commmence par le nom d'un protocole (optionnel), l'adresse d'un serveur (optionnel), un chemin absolu ou relatif, et des suffixes (arguments ou étiquettes).

Déstructurer une URL en ses composantes de base.
– en utilisant un lexeur,
– en utilisant la librairie des str des expressions régulières.

Slide 41

Grammaires
Analyse grammaticale.

Grammaires algébriques (context-free)

Slide 1

Une grammaire permet une représentation finie d'un langage (éventuellement infini). Noam Chomsky *(c.f. author of Manufacturing Conscent)* a inventé la notion de grammaire formelle.

Une grammaire algébrique (ou *context free*) G est un quadruplet (Σ, V, S, P) où :

– Σ est l'alphabet des terminaux (lexèmes ou *tokens*),
– V est l'alphabet des non-terminaux (disjoint de Σ),
– $S \in V$ est l'unique axiome.
– P est un ensemble fini de *règles de production*,
  de la forme $\alpha \to \beta$ avec $\alpha \in V$, et $\beta \in (\Sigma \cup V)^*$

Le langage $L(G)$ associé à la grammaire G est la fermeture transitive du singleton $\{S\}$ par la règle d'induction :

$$u\alpha v \in L \wedge \alpha \to \beta \in P \Longrightarrow u\beta v \in L$$

Dérivations

Pour montrer qu'un mot w est dans le langage L il faut donc exhiber une suite finie d'applications de règles de production partant de S et aboutissant à w, appelée une dérivation.

On peut représenter une dérivation par

Slide 2

$$S = w_0 \xrightarrow{P_1} w_1 \xrightarrow{P_2} \ldots \xrightarrow{P_n} w_n = w$$

où $w_k = u_k \beta_k v_k = u_{k+1} \alpha_{k+1} v_{k+1}$ et $P_k = \alpha_k \longrightarrow \beta_k$.

On laisse P_i implicite lorsqu'il n'y a pas d'ambiguité, ou on écrit :

$$\begin{array}{ll} S = & u_1 \underline{\alpha_1} v_1 \longrightarrow u_1 \overline{\beta_1} v_1 = u_2 \underline{\alpha_2} v_2 \longrightarrow u_2 \overline{\beta_2} v_2 = \ \ldots \\ & u_n \underline{\alpha_n} v_n \longrightarrow u_n \overline{\beta_n} v_n \end{array}$$

Exemples : expressions arithmétiques

Les terminaux (lexèmes) sont `NUM`, `ID`, `PLUS` et `MINUS`.

$S \to E \quad E \to \mathtt{NUM} \quad E \to \mathtt{ID} \qquad E \to E\ \mathtt{PLUS}\ E \qquad E \to E\ \mathtt{MINUS}\ E$

Pour reconnaître l'expression `1 - 1 + x`, *i.e.* après analyse lexicale, `NUM MINUS NUM PLUS ID`, il existe de nombreuses dérivations possibles. L'ensemble de ces dérivations peut être représenté par un ensemble d'arbres syntaxiques (ici deux) :

Slide 3

$$S \to E \to \left\{ \begin{array}{l} E \to \left\{ \begin{array}{l} E \to \mathtt{NUM} \\ \mathtt{MINUS} \\ E \to \mathtt{NUM} \end{array} \right. \\ \mathtt{PLUS} \\ E \to \mathtt{ID} \end{array} \right. \qquad S \to E \to \left\{ \begin{array}{l} E \to \mathtt{NUM} \\ \mathtt{MINUS} \\ E \to \left\{ \begin{array}{l} E \to \mathtt{NUM} \\ \mathtt{PLUS} \\ E \to \mathtt{ID} \end{array} \right. \end{array} \right.$$

Chaque arbre syntaxique représente lui même un ensemble de dérivations obtenues en fixant un ordre d'application des règles.

Dérivations

Slide 4

Par exemple, la dérivation la plus à gauche (resp. à droite) de l'arbre syntaxique de gauche est obtenue en réduisant toujours le non-terminal le plus à gauche (resp. à droite).

S		S
E		E
E PLUS E		E PLUS E
E MINUS E PLUS E	resp.	E PLUS ID
NUM MINUS E PLUS E		E MINUS E PLUS ID
NUM MINUS NUM PLUS E		E MINUS NUM PLUS ID
NUM MINUS NUM PLUS ID		NUM MINUS NUM PLUS ID

Parmi l'ensemble des dérivations d'un arbre syntaxique, il en existe toujours une la plus à gauche (resp. la plus à droite).

Grammaires régulières

Slide 5

Les règles de production sont de la forme $\alpha \rightarrow \beta$ où $\beta \in \{\epsilon\} \cup \Sigma V$.

Langage régulier Le langage décrit par une grammaire régulière peut être reconnu par un automate fini, et réciproquement.

- À (Σ, V, S, P), on associe $(V \cup \{F\}, \Sigma, \delta, S, \{F\})$ où $q' \in \delta(q, a)$ si $q \rightarrow aq'$ et $q \in F$ si $q \rightarrow \epsilon$.
- À un automate fini déterministe $(Q, \Sigma, \delta, q_0, F)$ on associe la grammaire (Σ, Q, q_0, P) où P contient $q \rightarrow aq'$ si $q' \in \delta(q, a)$ et $q \rightarrow \epsilon$ si $q' \in F$.

Exemple à l'expression régulière (ab)$*$ correspond la grammaire composée des lexèmes a et b et des règles :

$$S \longrightarrow \qquad S \longrightarrow \mathtt{a}I \qquad I \longrightarrow \mathtt{b}S$$

Un exemple de dérivation :

$$\underline{S} \longrightarrow \overline{\mathtt{a}\underline{I}} \longrightarrow \mathtt{a}\overline{\mathtt{b}\underline{S}} \longrightarrow \mathtt{ab}\overline{\mathtt{a}\underline{I}} \longrightarrow \mathtt{aba}\overline{\mathtt{b}\underline{S}} \longrightarrow \mathtt{abab}$$

Grammaires ambigües

Dérivations et arbres de dérivations

- En général, il existe de nombreuses dérivations d'un même arbre syntaxique, mais ce n'est pas gênant.
- Lorsqu'il existe plusieurs arbres syntaxiques dérivables pour une même expression, la grammaire est dite *ambigüe*.

Slide 6

Élimination des ambiguités

L'existence de plusieurs arbres syntaxiques pose un problème, car il faudra faire un choix dans la construction de l'arbre de syntaxe abstraite qui va en général influer sur la sémantique (par exemple, la soustraction des expressions arithmétiques n'est pas associative).

Heureusement, il est souvent possible d'éliminer les ambiguités, ce qui consiste à ré-écrire certaines règle de façon à obtenir une grammaire non-ambigüe reconnaissant le même langage.

Élimination des ambiguïtés

Typiquement, on peut séparer un terminal E en deux terminaux E et T.

$S \to E$	$E \to T$
$E \to E$ `PLUS` T	$T \to$ `NUM`
$E \to E$ `MINUS` T	$T \to$ `ID`

Slide 7

Fin de fichier/phrase on ajoute un lexème `EOF` et

- on change les règles $S \to \alpha$ en $S \to \alpha$ `EOF`, ou bien
- on introduit un symbole S' et on prend $S' \to S$ `EOF` pour axiome (S devient un non-terminal).

Grammaires prédictives LL(1)

Slide 8

Simples, elles permettent l'écriture manuelle d'analyseurs.

Pour $\alpha \in \Sigma$ et $\beta \in (\Sigma \cup V)^*$, on définit

- l'ensemble $follow(\alpha) \stackrel{\triangle}{=} \{a \in \Sigma \mid \exists \gamma, \gamma' \in (\Sigma \cup V)^*, S \longrightarrow^* \gamma\alpha a\gamma'\}$ des lexèmes qui peuvent succéder à α dans une dérivation de S.
- l'ensemble $first(\alpha) \stackrel{\triangle}{=} \{a \in \Sigma \mid \exists \gamma \in (\Sigma \cup V)^*, \beta \rightarrow^* a\gamma\}$ des lexèmes qui peuvent commencer une dérivation de β.

Nous construisons une relation $\mathcal{T}$ partiellement définie de $V \times \Sigma$ dans 2^P en ajoutant pour chaque règle $p \stackrel{\triangle}{=} \alpha \longrightarrow \beta \in P$ les sous-relations :

- $(\alpha, a) \mapsto p$ pour $a \in first(\gamma)$
- $(\alpha, a) \mapsto p$ pour $a \in follow(\alpha)$ lorsque $\beta \longrightarrow^* \epsilon$.

Une grammaire est *prédictive*, i.e. LL(1), si la relation $\mathcal{T}$ est une fonction.

Grammaires prédictives LL(k)

Slide 9

Généralisation possible à LL(k)

mais peu utilisée (inefficacité) On remplace $a \in \Sigma$ par $a \in \Sigma^k$ dans les définitions de $first(\beta)$ et de $follow(\alpha)$.

Exemples

La grammaire des expressions arithmétiques donnée ci-dessus n'est pas prédictive, car :

$$first(E \ \mathtt{PLUS} \ T) = \{\mathtt{NUM}, \mathtt{ID}\} = first(E \ \mathtt{MINUS} \ T) = \{\mathtt{NUM}, \mathtt{ID}\}$$

La grammaire définie par les deux règles $E \longrightarrow aEa$ et $E \longrightarrow \epsilon$. n'est pas prédictive, car $\mathcal{T}(E, a)$ contient

- La règle $E \longrightarrow aEa$ car $first(E) = \{a\}$,
- La règle $E \longrightarrow \epsilon$ car $follow(E) = \{a\}$ et $\epsilon \longrightarrow^* \epsilon$.

Élimination de la récursion gauche

La récursion gauche est une source d'ambiguité (conflit). On peut l'éliminer par tansformation (réécriture) en des règles équivalentes, sans récursion gauche.

Slide 10

Exemple $\begin{pmatrix} E \rightarrow E \ \texttt{PLUS} \ T \\ E \rightarrow E \ \texttt{MINUS} \ T \\ E \rightarrow T \end{pmatrix} \Longrightarrow \begin{pmatrix} E \rightarrow T \ E' \\ E' \rightarrow \texttt{PLUS} \ T \ E' \\ E' \rightarrow \texttt{MINUS} \ T \ E' \\ E' \rightarrow \epsilon \end{pmatrix}$

Cas général $\begin{pmatrix} X \rightarrow X \ \beta_1 \\ X \rightarrow X \ \beta_2 \\ X \rightarrow \gamma_1 \\ X \rightarrow \gamma_2 \end{pmatrix} \Longrightarrow \begin{pmatrix} X \rightarrow \gamma_1 \ X' \\ X \rightarrow \gamma_2 \ X' \\ X' \rightarrow \beta_1 \ X' \\ X' \rightarrow \beta_2 \ X' \\ X' \rightarrow \epsilon \end{pmatrix}$

Factorisation gauche

Elle se traite de façon similaire.

Exemple du if-then-else

Slide 11

$$\begin{pmatrix} E \rightarrow \texttt{IF} \ B \ \texttt{THEN} \ E \ \texttt{ELSE} \ E \\ E \rightarrow \texttt{IF} \ B \ \texttt{THEN} \ E \end{pmatrix} \Longrightarrow \begin{pmatrix} E \rightarrow \texttt{IF} \ B \ \texttt{THEN} \ E \ X \\ X \rightarrow \texttt{ELSE} \ E \\ X \rightarrow \epsilon \end{pmatrix}$$

Analyseur pour les grammaires prédictives

Pour chaque non-terminal $\alpha \in V$, on peut décider de la production à appliquer en fonction du prochain lexème $a \in \Sigma$: c'est l'unique règle $\alpha \to \beta \in \mathcal{T}(V, a)$ si elle existe, une erreur sinon (la phrase n'est pas reconnaissable).

Slide 12

Exemple pour les expressions arithmétiques (après élimination de la récursion gauche).

	PLUS	MINUS	ID	NUM	EOF
S			E EOF	E EOF	
E			$T\ E'$	$T\ E'$	
E'	PLUS $T\ E'$	MINUS $T\ E'$			ϵ
T			ID	NUM	

Programme d'analyse

La table ci-dessus peut être implémentée directement en interprétant le tableau ou bien par filtrage. (Il faudrait aussi ajouter un rattrapage d'erreur dans les cases vides du tableau) :

Slide 13

Fichier expr.ml

```
type lexème = PLUS | MINUS | ID | NUM | EOF;;
let buf = ref [];;
let lit_lexème () =
  match !buf with h::t -> buf:= t; h | [] -> EOF;;
let lexème = ref (EOF : lexème);;
let avance() = lexème := lit_lexème();;
let accepte a =
  if ! lexème = a then avance() else failwith "accepte"
```

Slide 14

```
let rec _S() =
  match !lexème with
  | ID | NUM -> _E(); accepte EOF
and _E() =
  match !lexème with
  | ID | NUM -> _T(); _E'()
and _E'() =
  match !lexème with
  | PLUS -> accepte PLUS; _T(); _E'()
  | MINUS -> accepte MINUS; _T(); _E'()
  | EOF -> ()
and _T() =
  match !lexème with
  | ID -> accepte ID;
  | NUM -> accepte NUM

let parse l = buf := l; avance(); _S();;
```

Slide 15

Programme d'analyse fonctionnel

```
let ( ! )  a = function b::v when a = b -> v
let ( * )  f g = function x -> g (f x)
let epsilon  x = x

let rec _S t = (match t with
| ( ID | NUM)::_ -> _E * !EOF
      ) t
and _E t = (match t with
| ( ID | NUM)::_ -> _T * _E'
      ) t
and _E' t = (match t with
| PLUS :: _  -> !PLUS * _T * _E'
| MINUS :: _ -> !MINUS * _T * _E'
| EOF :: _   -> epsilon
      ) t
and _T t = (match t with
| ID :: _  -> !ID
| NUM :: _ -> !NUM
      ) t
```

S'étend facilement au cas LL(k).

Langages LR(1)

***L**eft-to-right parse, **R**ight-most derivation (**1**-token lookahead)*

Les grammaires LL(1) permettent l'écriture facile donc manuelle d'analyseurs, mais l'écriture d'une grammaire LL(1) pour un langage est contraignante et revient à une forme de "compilation manuelle". De plus, celle-ci n'est pas toujours possible.

Slide 16

Les grammaires LR sont plus générales : une grammaire LR(1) est aussi LL(1) donc la classe des langages reconnus par une grammaire LR(1) contient celle des langages LL(1).

Les grammaires LR(1) sont également plus souples (moins contraignantes) dans l'écriture des règles.

Principe

Leur moteur utilise une pile auxiliaire. Le flux de lexème est lu en consultant au plus 1 lexème non empilé. Il faut alors pouvoir décider d'une action à effectuer (en fonction de la pile et du prochain lexème) :

– $shift$, *i.e.* consommer et empiler un lexème, ou
– $reduce$, *i.e.* réduire une règle. Cela revient à appliquer une règle de production sur le sommet (partie droite) de la pile.

Slide 17

Si pour une même configuration de pile et du prochain lexème, plusieurs axiomes sont possibles, alors la grammaire n'est pas LR(1) et dite ambigüe.

Généralisation possible en regardant les k prochains lexèmes (grammaires LR(k)) mais peu utilisée (inefficacité).

Slide 18 Trace d'exécution

Pile	Flux de lexèmes	Action
	NUM MINUS NUM PLUS ID EOF	*shift*
NUM	MINUS NUM PLUS ID EOF	*reduce*
T	MINUS NUM PLUS ID EOF	*reduce*
E	MINUS NUM PLUS ID EOF	*shift*
E MINUS	NUM PLUS ID EOF	*shift*
E MINUS NUM	PLUS ID EOF	*reduce*
E MINUS T	PLUS ID EOF	*reduce*
E	PLUS ID EOF	*shift*
E PLUS	ID EOF	*shift*
E PLUS ID	EOF	*reduce*
E PLUS T	EOF	*reduce*
E	EOF	*shift*
E EOF		*reduce*
S		

Reconnaissance par un automate à pile

Slide 19

Une grammaire LR(1) est reconnaissable par un automate à pile qui mécanise le calcul précédent.

Description

- Les éléments sur la pile sont des paires (X_i, s_i) d'un état s_i et d'un lexème $X \in \Sigma \cup V$ sauf le fond de pile qui est l'état initial s_0.
- L'automate est déterminé par deux fonctions partiellement définies :
  - $goto(s, X)$ pour $X \in \Sigma \cup V$.
  - $action(s, a)$ qui peut prendre les valeurs $shift$ ou $reduce(P)$ où P est une règle de production

Transitions

Slide 20

Dans une configuration de pile $s_0(X_1, s_1)...(X_m, s_m)$ (sommet à droite), l'automate consulte le lexème suivant a et par cas sur la valeur de $action(s_m, a)$ effectue l'une des deux actions :

- $shift$: le prochain lexème est consommé et $(a, goto(s_m, a))$ est empilée
- $reduce(X \to \alpha)$: k éléments où k est la longueur de α sont dépilés et $(X, goto(s_{m-k}, X))$ est empilé.

Lorsque $action(s_m, a_i)$ est indéfinie, l'automate s'arrête, avec succès si la pile est réduite à l'axiome, avec échec sinon.

Construction des états

Slide 21

Un état I est composé d'un ensemble de configurations C.

Une configuration C est une paire composée :

1. d'une règle de production pointée $\alpha \to \beta.\gamma$. *i.e.* $\alpha \to \beta\gamma$ est une règle de production et β est le sommet de pile.
2. du prochain lexème possible $a \in \Sigma$.

La fermeture d'un ensemble de configurations I est le plus petit ensemble contenant I et satisfaisant

$$((X \to \alpha \cdot Y\beta, a) \in I \wedge Y \to \gamma \in G \wedge b \in first(\beta a)) \Longrightarrow (Y \to \cdot\gamma, b) \in I$$

La transition d'une configuration I par un lexème $X \in \Sigma \cup V$ est

$$goto(I, Y) = fermeture(\{(X \to \alpha Y \cdot \beta, a) \mid (X \to \alpha \cdot Y\beta, a) \in I\})$$

Construction des tables

L'état initial est la fermeture de l'axiome. Les états sont les ensembles de configurations fermées non vides atteignables par une suite de transitions arbitraires à partir de l'état initial.

Les actions sont de 4 types :

Slide 22

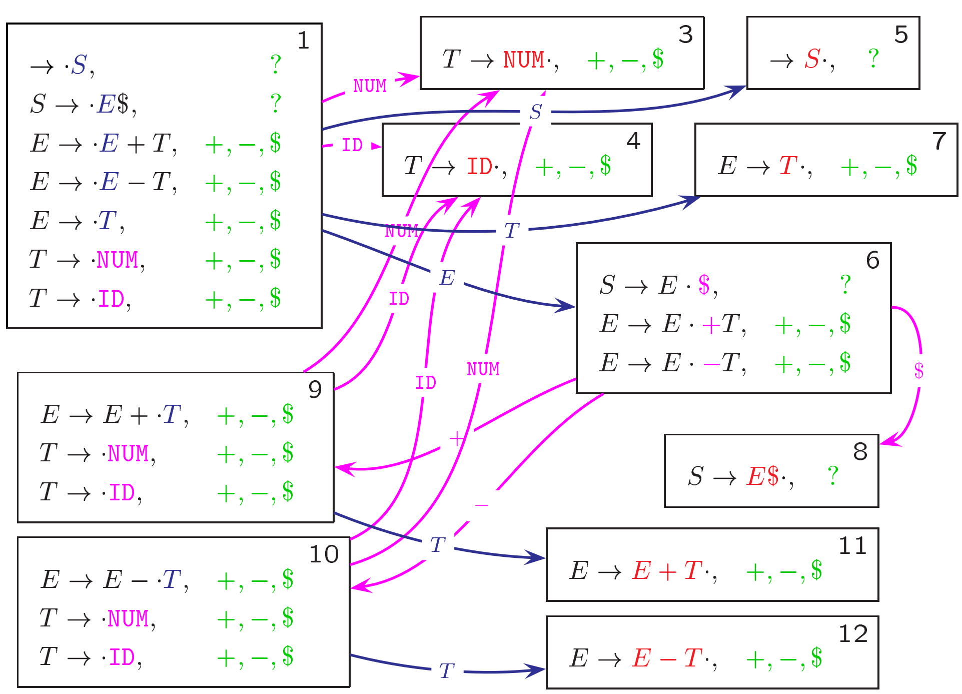
1. Si $(X \to \alpha \cdot a\beta, b) \in I$ et $goto(I, a) = J$, alors $action(I, a) = shift$ (on peut représenter simultanément $goto(I, a)$ en écrivant $shift(J)$)
2. Si $(X \to \alpha\cdot, a) \in I$, alors $action(I, a) = reduce(X \to \alpha)$. Sauf, si $X \to \alpha$ est l'axiome, alors $action(I, a) = accept$.

(Il reste à représenter la table $goto(I, Y)$ pour $Y \in V$.)

Enfin, la grammaire n'est pas LR(1) s'il y a des conflits, *i.e.* plusieurs valeurs possibles pour $action(I, a)$.

Pour les autres cases, $action(I, a) = echec$.

Slide 23



LR(0) v.s. LR(1)

Slide 24

Dans la construction précédente, l'ensemble des lexèmes prossibles après le déclanchement d'une règle est presque toujours `EOF`, `PLUS`, `MINUS`, et ne sert jamais à déambigüer deux états.

En effet, l'exemple considérer est aussi LALR(0).

Exercice 3 *On considère la grammaire suivante*

$$S \to E \ \texttt{EOF} \qquad E \to T \ \texttt{PLUS} \ E \qquad E \to T \qquad T \to \texttt{ID}$$

Vérifier que la grammaire n'est pas LR(0). (On construit les états comme pour une grammaire LR(1), mais dans lesquelles les configurations ne comporte que les premières composante (une règle pointée. Il suffit alors de vérifier qu'il existe deux transition possibles avec une même étiquette).

Vérifier que la grammaire est LR(1). □

Construction de la table de l'automate

Slide 25

On construit un tableau de dimension 2 avec les états en ordonnées et les éléments de lettre de l'alphabet $\Sigma \cup V$ en abscisse.

On remplit le tableau ligne par ligne. Pour chaque état I,

- Pour chaque flèche étiquettée par un terminal $I \to^{a} J$, on associe une action $shift(J)$ dans la colonne a.
- Pour chaque flèche étiquettée par un non-terminal $I \to^{a} J$, on associe on mémorise la valeur J de $goto(I, a)$ dans la case a. (pseudo-action $shift(J)$).

Enfin, pour chaque règle pointée à la fin (en rouge) on associe une action réduce dans la case associé à chacun des terminaux suivant possibles (en vert).

Tables

Slide 26

I	$+$	$-$	NUM	ID	\$	S	E	T
1			$s3$	$s4$		5	6	7
3	$r5$	$r5$			$r5$			
4	$r6$	$r6$			$r6$			
5								
6	$s9$	$s10$			$s8$			
7	$r4$	$r4$			$r4$			
8	$r1$	$r1$			$r1$			
9			$s3$	$s4$				11
10			$s3$	$s4$				12
11	$r2$	$r2$			$r2$			
12	$r3$	$r3$			$r3$			

r	règle
1	$S \to E\$$
2	$E \to E + T$
3	$E \to E - T$
4	$E \to T$
5	$T \to \mathtt{NUM}$
6	$T \to \mathtt{ID}$

Slide 27

Pile	Flux de lexèmes	Action
$_1$	$\mathtt{NUM} - \mathtt{NUM} + \mathtt{ID}\$$	*shift*
$_1\mathtt{NUM}_3$	$- \mathtt{NUM} + \mathtt{ID}\$$	*reduce*
$_1 T_7$	$- \mathtt{NUM} + \mathtt{ID}\$$	*reduce*
$_1 E_6$	$- \mathtt{NUM} + \mathtt{ID}\$$	*shift*
$_1 E_6\ -$	$\mathtt{NUM} + \mathtt{ID}\$$	*shift*
$_1 E_6\ -_{10}\ \mathtt{NUM}_3$	$+ \mathtt{ID}\$$	*reduce*
$_1 E_6\ -_{10}\ T_{12}$	$+ \mathtt{ID}\$$	*reduce*
$_1 E_6$	$+ \mathtt{ID}\$$	*shift*
$_1 E_6\ +_9$	$\mathtt{ID}\$$	*shift*
$_1 E_6\ +_9\ \mathtt{ID}_4$	$\$$	*reduce*
$_1 E_6\ +_9\ T_{11}$	$\$$	*reduce*
$_1 E_6$	$\$$	*shift*
$_1 E\$_8$		*reduce*
$_1 S_5$		

Correction

Exercice 4 (Correction) *Vérifier que lorsque l'algorithme ci-dessus trouve une solution, alors il est possible de construire une dérivation.* □

Exercice 5 (Complétude) *Vérifier que lorsque l'algorithme ne trouve pas de solution, alors il n'existe pas de dérivation.* □

Slide 28

Conflits

Un conflit se produit lorsqu'il existe plusieurs une configuration (I, a) pour laquelle $action(I, a)$ admet plusieurs valeurs possibles.

On parle de conflit :

– $reduce - reduce$ lorsque $(X \to \alpha\cdot, a) \in I$ et $(Y \to \beta\cdot, a) \in I$.
  Ces conflits sont graves et correspondent généralement à une forte ambiguïté dans la grammaire. Il faut réécrire la grammaire.
– $shift - reduce$ lorsque $(X \to \alpha \cdot a\gamma, b) \in I$ et $(Y \to \beta\cdot, a) \in I$.
  Ces conflits sont moins graves, et correspondent souvent à des priorités qu'il faut rendre explicites, soit par réécriture, soit par ajout de règles de priorité.

(Par construction, les conflits $shift - shift$ ne sont pas possibles)

Slide 29

Règles d'associativité

La grammaire suivante contient un conflit $shift/reduce$

$$1 : E \to E + E \qquad\qquad 2 : E \to \mathtt{NUM}$$

En effet, la séquence NUM PLUS NUM PLUS NUM est reconnue par

Slide 30

$$shift, reduce\,2, shift, shift, reduce\,2, \begin{cases} \underline{reduce\,1}, shift, reduce\,2, reduce\,1 \\ \underline{shift}, reduce\,2, reduce\,1, reduce\,1 \end{cases}$$

La première solution reconnaît $(E + E) + E$, l'autre $E + (E + E)$.

On peut réécrire la grammaire pour forcer l'une ou l'autre règle, mais il est aussi possible et préférable (bien que sortant du formalisme LR(1)) de choisir entre les deux actions possibles par des règles d'associativité et de priorité.

(Les règles de priorité s'appliquent localement pour ordonner les actions possibles à partir d'un état ambigü.)

Règles de priorité

Selon que + est déclaré associative à gauche ou à droite, l'action retenue sera $reduce$ ou $shift$.

On peut également déclarer des lexèmes plus prioritaires que d'autres, ou des règles plus prioritaires que d'autres.

Slide 31

Grammaires LALR(1)

Slide 32

L*ook* **A***head* **LR(1)**

Les grammaires LALR(1) sont une restriction des grammaires LR(1) afin d'obtenir des tables plus petites. Le langage reconnu est un peu plus restrictif, mais ce n'est pas une gêne en pratique.

Dans une grammaire LALR(1) les états qui ne sont distingués que par le lexème suivant (mais qui ont exactement le même ensemble de règles de production) sont identifiés (avant construction des tables).

Yacc, Ocamlyacc, Bison, etc., sont des compilateurs de grammaires LALR(1).

Ocamlyacc

Slide 33

Fichier foo.mly

```
%{
  (* prelude : code Ocaml reproduit verbatim *)
%}
  /* declarations yacc */
%%
  /* règles yacc */
%%
  (* postlude : code Ocaml reproduit verbatim *)
```

Pour compiler la grammaire

```
ocamlyacc -v foo.mly          # produit foo.ml et foo.output
ocamlc foo.ml
```

L'option `-v` copie les tables dans un fichier `foo.output` (utile pour la mise au point)

Slide 34

Exemple

/exp.mly

Slide 35

```
/exp.mly
 /exp.mly
 %%
 %% This is file ‘. tex’,
 %% generated with the docstrip utility .
 %%
 %% The original source files were:
 %%
 %% fileerr.dtx  (with options: ‘ return’)
 %%
 %% This is a generated file.
 %%
 %% Copyright 1993 1994 1995 1996 1997 1998 1999
 %% The LaTeX3 Project and any individual authors listed elsewhere
 %% in this file .
 %%
 %% This file is part  of  the Standard LaTeX ‘Tools Bundle’.
 %% −−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−
 %%
 %% This file,  and all  the other files   in  this  bundle as listed   below,
 %% may be distributed under the terms of the LaTeX Project Public
 %% License, as described in  lppl . txt  in  the base LaTeX distribution.
 %% Either version 1.0 or,  at  your option,  any later  version .
 %%
  \message{File ignored}
 \endinput
 %%
 %% End of file ‘.tex’./exp.mly
```

Indication des priorités

Dans la partie déclaration :

```
%noassoc lower
%left PLUS MINUS
%left TIMES DIV
%right FOO
%noassoc BAR
%noassoc higher
```

Slide 36

Par défaut, la priorité d'une règle est celle de son dernier lexème.

La directive %**prec** peut être utilisée à la fin d'une règle pour indiquer la priorité à donner à cette règle.

Mise en œuvre

Il faut combiner le parseur avec un lexeur. voir `lexp.mll`.

Le parseur doit être compilé en premier, car il produit simultanément la définition du type des `token` utilisé par le lexeur (le nom de l'identificateur token est fixé par convention).

Le programme principal appelle le parseur qui appelle le lexeur comme ci-dessus.

Slide 37

Report d'erreur

Il existe un mécanisme de report d'erreur dans yacc, mais qui n'est pas facile à mettre en œuvre. En cas d'erreur, il est simple et souvent suffisant de signaler le dernier lexème lu et son emplacement (voir le compilateur Pseudo-Pascal main.ml)

Exercices

Exercice 6 (Langage de parenthèses) *Écrire un parseur (et un lexeur) pour le langage de parenthèses.*

Comparer avec la reconnaissance des parenthèses directement dans le lexeur. Quelle approche est-elle préférable ? Réponse □

Slide 38

Exercice 7 (Expressions arithmétiques) *Écrire un parseur (et un lexeur) pour reconnaître les expressions arithmétiques sans variables. Retourner l'arbre de syntaxe abstraite.*

En déduire une calculette en remplaçant la génération de l'arbre par le calcul.

Pour obtenir une Super calculette, *on pourra*
- *Ajouter des mémoires (set n et get n).*
- *Ajouter la comparaison et les expressions conditionnelles.*

□

Travaux dirigés

Exercice 8 (Pseudo Pascal) *Écrire un lexeur pour le langage Pseudo Pascal. (On réutilisera le lexeur de cet exercice).*

Un exemple de programme Pseudo Pascal est fact.p.

Slide 39

La syntaxe programmes est donnée dans une forme BNF (Backus Normal Form) que l'on peut lire comme des règles de production, une par ligne (le symbole | remplace un saut à la ligne).

La notation $u\, s \ldots u$ *(resp.* $u\, s \ldots u\, s$*) représente une séquence* possiblement vide *de* u *séparés par des* s *(resp. et terminée par un* s*).*

Programmes

p	::= `program` $v; \dots v;\ d; \dots d;\ i\ ;;$	*Programme*

Déclarations

v	::= `var` $x : t$	*Décl. de variable*
d	::= `function` $x($ `var` $x : t, \dots$ `var` $x : t\ ) : t;\ v; \dots v;\ i$	*Décl. de fonction*
	`procedure` $x($ `var` $x : t, \dots$ `var` $x : t\ );\ v; \dots v;\ i$	*Décl. de procédure*

Slide 40

Instructions

i	::= $x := e$	*Affectation*
	`begin` $i; \dots i$ `end`	*Séquence*
	`if` e `then` i `else` i	*Conditionnelle*
	$x(\ e, \dots e\ )$	*Appel de procédure*
	`read` (x)	*Lecture*
	`write` (e)	*Écriture*
	`writeln` (e)	*Écriture avec newline*
	$e[e] := e$	*Affectation de tableau*

Exercice (Suite)

Slide 41

Expressions

e	::=	(e)	*Expression parenthèsée*
		x	*Variables*
		n	*Entiers*
		`true` | `false`	*Booléens*
		$x(\ e, \dots e\ )$	*Appel de fonction*
		`alloc` $(e : t)$	*Allocation*
		$e\ bin\ e$	*Opération binaire*
		$e[e]$	*Accès dans un tableau*

Opérations

bin	::=	$+ \mid - \mid \times \mid /$	*Arithmétiques*
		$< \mid > \mid <= \mid >= \mid = \mid <>$	*Comparaison*

Types

t	::=	`integer`
		`boolean`
		`array of` t

Exercice (Suite)

Dans un premier temps, on pourra laisser les actions vides (on retour unit). Ensuite, on produira un arbre de syntaxe abstraite défini par le type concret program *du fichier d'interface pp.mli.* Réponse

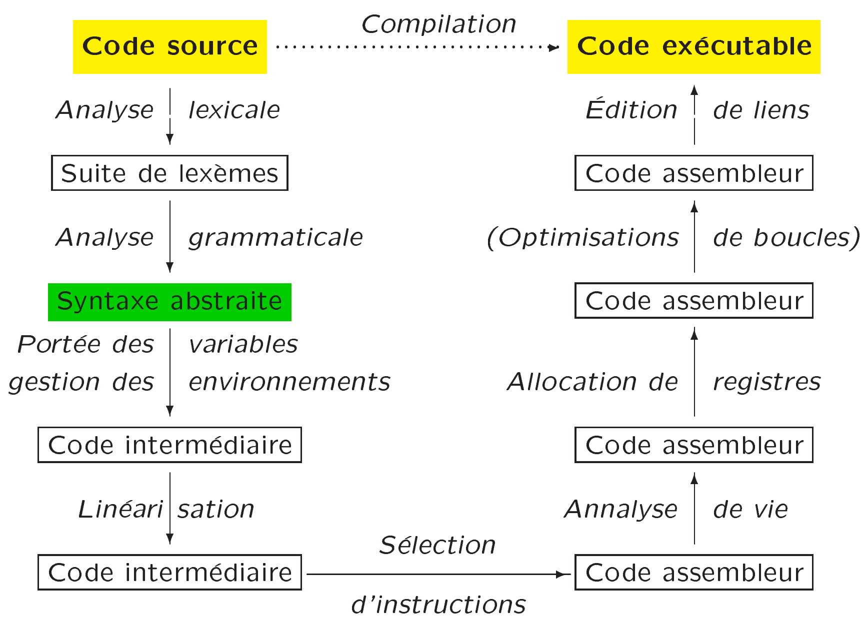
Slide 42

Enfin, on pourra compléter le programme avec un imprimeur qui imprime le programme (arbre de syntaxe abstraire reconnu) dans la sortie standard. En particulier, cela permet de vérifier que le programme a été compris correctememt. Réponse □

4

Langages de programmation
Sémantiques opérationnelles
Interprètes
Le langage pseudo-pascal.

Slide 1



Langages de programmation

Slide 2

Langages généraux

Ils doivent être complets, *i.e.* permettre d'exprimer tous les algorithmes calculables. Au minimum, il faut une mémoire infinie (ou grande) et la possibilité d'exprimer la récursion (construction primitive ou boucle `while`). Ex : Fortran, Pascal, C, Ocaml, etc.

Langages spécialisés (DSL pour *Domain Specific Languages*)

Langages pour le graphisme, pour commander des robots, pour faire des annimations graphiques ; la calculette. Ils peuvent ne pas être complets.

Expressivité

Les langages généraux ne sont pas tous équivalents. L'expressivité est la capacité d'exprimer des algorithmes succinctement (et directement).

Exemples de constructions expressives

Slide 3

Les fonctions
- Est-ce que les fonctions peuvent être locales ?
- Est-ce que les fonctions sont des valeurs ?

Les structures de données (et filtrage)

Les modules, les objets

Le typage restreint l'expressivité au profit de la sécurité. (Comparer Scheme et ML).

L'expressivité du système de typage est donc aussi importante (comparer ML et Pascal ou Java).

Note : Pour comparer l'expressivité formellement, il faut en général le faire point à point en exhibant un codage d'un langage dans un autre qui respecte certaines règles (compositionalité, localité, *etc.*)

Syntaxe concrète – syntaxe abstraite

La syntaxe concrète décrit les mots du langage et la façon d'assembler les mots en phrases pour constituer des programmes.

Slide 4

Elle ne donne aucun sens aux phrases :
- Plusieurs notations sont possibles pour une même construction. Exemple : les caractères `'a'` et `'\097'` ou en la boucle `for` qui s'exprime en fonction de la boucle `while`
- La syntaxe concrète est linéaire et utilise des parenthèses.

La syntaxe abstraite est une représentation arborescente qui fait abstraction de la notation (on parle d'arbre de syntaxe abstraite). Elle définit les constructions du langage.

L'analyse *lexicale* traduit une suite de caractères en suite de mots. L'analyse *grammaticale* lit une suite de mots, reconnaît les phrases du langages et retourne un arbre de syntaxe abstraite.

Exemple simple : La calculette

Slide 5

Syntaxe concrète (dans le style BNF)

```
expression ::= ENTIER
             | IDENTIFICATEUR
             | expression  binop expression
             | '(' expression ')'
binop      ::= '+' | '-' | '*' | '/'
```

Syntaxe abstraite

```
type expression =
  | Const of int
  | Variable  of string
  | Bin of opérateur * expression * expression
and opérateur = Plus | Moins | Mult | Div;;
```

Exemple l'expression `(1 - x) * 3` a pour représentation

```
Bin (Mult, Bin (Moins, Const 1, Variable "x"), Const 3);;
```

Sémantique des programmes

Il s'agit de donner un sens aux programmes.

Sémantique dénotationnelle

On associe un objet mathématique (abstrait) à chaque expression. Par exemple, on construit un modèle mathématique des entiers, muni des opérations sur les entiers.

Slide 6

C'est beaucoup plus dur pour les fonctions (calculables).

Sémantique opérationnelle

On définit un ensemble de valeurs (ou résultats) puis une relation d'évaluation qui relie des programmes avec des résultats.

Cette sémantique est plus proche de la syntaxe (moins abstraite), mais elle est plus facile à manipuler.

C'est aussi celle qui nous intéresse le plus souvent (pour calculer).

Sémantique opérationnelle de la calculette

Les valeurs sont les entiers.

```
type valeur = int
```

On peut définir l'évaluation par un programme Ocaml qui prend un environnement initial associant des valeurs à certaines

Slide 7

variables, une expression à évaluer et retourne un entier :

Slide 8

```
let cherche x env = List.assoc x env
let rec évalue env = function
  | Const n -> n
  | Variable x -> cherche x env
  | Bin (op, e1, e2) ->
      let v1 = évalue env e1 and v2 = évalue env e2 in
      begin match op with
       | Plus -> v1 + v2 | Moins -> v1 - v2
       | Mult -> v1 * v2 | Div -> v1 / v2
      end ;;
```

Slide 9

Une présentation plus neutre (S.O.S.)

(**S**émantique **O**pérationnelle **S**tructurelle)

Une autre présentation équivalente, mais plus mathématique et plus modulaire modulaire consiste à définir une relation $\rho \vdash e \Rightarrow v$ qui se lit *"Dans l'environnement ρ, l'expression e s'évalue en la valeur v"* par des règles d'inférence, c'est-à-dire comme le plus petit ensemble vérifiant une certains nombre de règles (implications).

Une règle d'inférence est une implication $P_1 \wedge \ldots P_k \Longrightarrow C$ présentée sous la forme

$$\frac{P_1 \wedge \ldots \wedge P_k}{C}$$

que l'on peut lire pour réaliser (évaluer) C il faut réaliser à la fois P_1 et ... P_k.

Sémantique des expressions arithmétiques

Slide 10

Les jugements de la forme $\rho \vdash e \Rightarrow v$

- ρ lie des variables x à des valeurs v, i.e. c'est un ensemble de paires notées $x \mapsto v$.
- $e \in$ *expressions* et $v \in I\!N$. On note $\bar{n}$ l'entier naturel associé à sa représentation n.

$$\rho \vdash \texttt{Const}\ n \Rightarrow \bar{n} \qquad \frac{x \in dom\,(\rho)}{\rho \vdash \texttt{Variable}\ x \Rightarrow \rho(x)}$$

$$\frac{\rho \vdash e_1 \Rightarrow v_1 \qquad \rho \vdash e_2 \Rightarrow v_2}{\rho \vdash \texttt{Bin}\,(\texttt{Plus}\,, e_1, e_2) \Rightarrow v_1 + v_2} \qquad \frac{\rho \vdash e_1 \Rightarrow v_1 \qquad \rho \vdash e_2 \Rightarrow v_2}{\rho \vdash \texttt{Bin}\,(\texttt{Times}\,, e_1, e_2) \Rightarrow v_1 * v_2}$$

Une construction de liaison

Slide 11

On étend la calculette avec des liaisons. On ajoute un noeud de syntaxe abstraite

```
| Let of string * expression * expression
```

avec, par exemple, la syntaxe concrète

```
"let" IDENTIFICATEUR "=" expression "in" expression
```

L'expression $\texttt{Let}\,(x, e_1, e_2)$ lie la variable x à l'expression e_1 dans l'évaluation de l'expression e_2.

Formellement :

$$\frac{\rho \vdash e_1 \Rightarrow v_1 \qquad \rho, x_1 \mapsto v_1 \vdash e_2 \Rightarrow v_2}{\rho \vdash \texttt{Let}\,(x, e_1, e_2) \Rightarrow v_2}$$

où $\rho, x \mapsto v$ ajoute la liaison de x à v dans l'environnement ρ en cachant une ancienne liaison éventuelle de x.

Modification de l'interprète

```
let ajoute x v env = (x,v):: env
```

Noter le codage de $\rho, x \mapsto v$ par $(x, v) :: \rho$

Slide 12

```
let rec évalue env =
  ...
| Let (x, e1, e2) ->
    let v1 = évalue env e1 in
    évalue (ajoute x v1 env) e2 ;;
```

Exercice 9
Pour l'expression **let** *x = 1* **in** *(***let** *x = 2* **in** *x) + x, donner :*
- *Sa syntaxe abstraite,*
- *Le résultat de son évaluation,*
- *La dérivation de l'évaluation de l'expression la plus interne.*

□

Arbre de dérivation

Slide 13

L'ensemble des réponses (de l'exercice précédent) est contenu dans l'arbre de dérivation, qui constitue une preuve de l'évaluation de l'expression en la valeur 3

$$\dfrac{\emptyset \vdash \texttt{Const}\ 1 \Rightarrow 1 \qquad \dfrac{\dfrac{x \mapsto 1 \vdash \texttt{Const}\ 2 \Rightarrow 2 \qquad x \mapsto 1, x \mapsto 2 \vdash x \Rightarrow 2}{x \mapsto 1 \vdash \texttt{Let}\,(x, \texttt{Const}\ 2, x) \Rightarrow 2} \qquad x \mapsto 1 \vdash x \Rightarrow 1}{x \mapsto 1 \vdash \texttt{Bin}\,(\texttt{Plus}\,, \texttt{Let}\,(x, 2, x), x) \Rightarrow 3}}{\emptyset \vdash \texttt{Let}\,(x, \texttt{Const}\ 1, \texttt{Bin}\,(\texttt{Plus}\,, \texttt{Let}\,(x, 2, x), x)) \Rightarrow 3}$$

Formalisation des erreurs

Slide 14

L'évaluation peut mal se passer, même dans la calculette, par exemple, lors d'une division par 0 ou de l'accès à une variable non liée.

La sémantique opérationnelle doit aussi formaliser les erreurs.

On remplace la relation $\rho \vdash e \Rightarrow v$ par une relation $\rho \vdash e \Rightarrow r$ où r est une réponse. Les réponses sont l'union des valeurs v ou des erreurs z.

$$\frac{\rho \vdash e_1 \Rightarrow v_1 \quad \rho \vdash e_2 \Rightarrow v_2 \quad v_2 \neq 0}{\rho \vdash \mathtt{Bin}\,(\mathtt{Div}\,, e_1, e_2) \Rightarrow v_1 / v_2} \qquad \frac{\rho \vdash e_2 \Rightarrow 0}{\rho \vdash \mathtt{Bin}\,(\mathtt{Div}\,, e_1, e_2) \Rightarrow \mathtt{Division}}$$

Le type des erreurs (par exemple) :

```
type erreur = Division_par_zéro | Variable_libre of string
```

Implémentation des erreurs

Slide 15

Formellement, on utilise un type somme

```
type résultat = Valeur of valeur | Erreur of erreur
```

En pratique, on utilise les exceptions du langage hôte

```
exception Erreur of erreur
let erreur x = raise (Erreur x)
let cherche x l =
  try List.assoc x l
  with Not_found -> erreur (Variable_libre x)
let rec évalue environnement = function
    ...
  | Bin (Div, e1, e2) ->
      let v1 = évalue env e1 and v2 = évalue env e2 in
      if v2 = 0 then erreur (Division_par_zéro )
      else v1 / v2 ;;
```

Terminaison

L'évaluation peut ne pas terminer.

La S.O.S (à grands pas) ne permet ne modélise pas la terminaison. En effet, elle met directement en relation un programme avec la réponse finale. Aussi, un programme qui ne termine pas ne peut pas être mis en relation avec une réponse.

Slide 16

Sémantique à réduction à petits pas

Dans le cours *Langage et programmation* on définit une sémantique opérationnelle à réduction "à petits pas" qui permet quand même de décrire le calcul des programmes qui ne terminent pas : le calcul est modélisé par une relation de réduction interne, i.e. les programmes se réduisent sur eux mêmes (chaque étape élémentaire du calcul est modélisé par un petit pas de réduction) jusqu'à l'obtention d'une valeur ou d'un programme bloqué (erreur).

La calculette fonctionnelle

Ajout des fonctions

- $\texttt{Fun}(x, e)$ désigne une fonction qui à x associe l'expression e.
- $\texttt{App}(e_1, e_2)$ désigne l'application d'une expression fonctionnelle e_1 à une expression e_2.

Difficultés engendrées

Slide 17

1. Les fonctions sont définies dans un environnement ρ mais elles sont appelées dans un environnement ρ' en général différent ;

   La *liaison statique* exige que la valeur des variables soit prise au moment de la définition.

2. Si les fonctions sont des valeurs (retournées en résultat), alors leur durée de vie peut excéder leur portée lexicale.

Exemple

Liaison statique

Slide 18

```
let x = 3                  (* env1 : x est lié  à une valeur v   *)
let rec mem =              (* la valeur de x est celle de env1 *)
  function [ ] -> false  |  h:: t -> x = h && mem t
let x = w                  (* env2 : x est lié  à la valeur w    *)
mem l                      (* doit être évalue; dans env1        *)
```

Valeur fonctionnelles

```
let incr x = fun y -> x + y
                        (* incr x retourne une fonction        *)
let f = incr 3          (* f doit mémoriser la valeur 3 de x   *)
f 4                     (* pour la restaurer lors de l'appel   *)
```

Fermetures

Slide 19

Une solution générale consiste à évaluer les fonctions en des fermetures.

Une fermeture est une paire $\langle \texttt{Fun}\,(x, e), \rho\rangle$ formée d'une expression fonctionnelle $\texttt{Fun}\,(x, e)$ et d'un environnement ρ.

$$\rho \vdash \texttt{Fun}\,(x, e) \Rightarrow \langle \texttt{Fun}\,(x, e), \rho\rangle$$

On peut éventuellement restreindre ρ à l'ensemble des variables libres dans $\texttt{Fun}\,(x, e)$. Elle permet d'*enfermer* une expression avec son environnement statique. Une fermeture est une valeur qui peut être passée en argument.

L'application restore l'environnement statique.

$$\frac{\rho \vdash e_1 \Rightarrow \langle \texttt{Fun}\,(x, e_0), \rho_0\rangle \qquad \rho \vdash e_2 \Rightarrow v_2 \qquad \rho_0, x \mapsto v_2 \vdash e_0 \Rightarrow v}{\rho \vdash \texttt{App}\,(e_1, e_2) \Rightarrow v}$$

Variables libres

Un variables libre d'une expression est une variable utilisée dans cette expression dans un contexte où elles ne sont pas liée :

$$vl\,(\texttt{Const}\ n) = \emptyset \qquad vl\,(\texttt{App}\,(e_1, e_2)) = vl\ e_1 \cup vl\ e_2$$
$$vl\,(\texttt{Variable}\ x) = \{x\} \qquad vl\,(\texttt{Plus}\,(e_1, e_2)) = vl\ e_1 \cup vl\ e_2$$
$$vl\,(\texttt{Fun}\,(x, e)) = vl\ e_1 \setminus \{x\} \qquad \ldots$$

Slide 20

Exemple La seul variable libre de `(fun y -> x + (fun x x) y)` est `x`. En effet `y` apparaît dans une contexte dans lequel elle est liée. Par contre, `x` apparaît au moins une fois dans un contexte dans lequel elle n'est pas liée.

Fermetures

Il suffit de mettre dans une fermeture les variables libres. On peut remplacer $\langle e, \rho\rangle$ par $\langle e, \rho \upharpoonright vl\,(e)\rangle$ car ρ ne sera utilisé que pour évaluer l'expression e.

Évaluation paresseuse

La sémantique précédente est dite *en appel par valeur* parce que les arguments sont évalués avant d'être passés aux fonctions.

Pour donner une sémantique *en appel par nom* (utilisée dans les langages Algol, Miranda, Haskell, Gaml), on gèle les arguments des fonctions, jusqu'à ce que leur évaluation devienne indispensable.

Slide 21

Comme pour les fonctions, il faut sauver le contexte d'évaluation dans des fermetures, aussi appelées glaçons (expressions gelées).

$$\frac{\rho \vdash e_1 \Rightarrow \langle \texttt{Fun}\,(x, e_0), \rho_0\rangle \qquad \rho_0, x \mapsto \langle e_2, \rho\rangle \vdash e_0 \Rightarrow v}{\rho \vdash \texttt{App}\,(e_1, e_2) \Rightarrow v}$$

$$\frac{\rho' \vdash e \Rightarrow v}{\rho, x \mapsto \langle e, \rho'\rangle \vdash x \Rightarrow v}$$

Évaluation paresseuse

L'appel par nom conduit à réévaluer plusieurs fois la même expression (chaque fois qu'un glaçon est dégelé).

L'*évaluation paresseuse* est une évaluation en appel par nom, avec en plus la propriété qu'un glaçon n'est évalué qu'un seule fois. On utilise l'affectation (voir plus loin) pour transformer un glaçon en une valeur après sa première évaluation.

Slide 22

L'évaluation paresseuse parce qu'elle ne force pas l'évaluation des arguments à de meilleures propriétés $(\mathtt{App}\,(\mathtt{Fun}\,(x, e_1), e_2))$ est équivalent à $e_1\{x \leftarrow e_2\}$, *i.e.* e_1 où x est remplacé par e_2.

En pratique toutefois :

+ L'évaluation paresseuse ne se combine pas bien avec des effets de bords (affectation, exception).
- L'évaluation paresseuse peut être simulée (manuellement) en appel par valeur en codant les glaçons comme des fonctions.

La conditionnelle

La construction $\mathtt{Ifz}\,(e_1, e_2, e_3)$ est toujours une construction paresseuse (l'un des deux derniers arguments seulement est évalué).

$$\frac{\rho \vdash e_1 \Rightarrow 0 \qquad \rho \vdash e_2 \Rightarrow v}{\rho \vdash \mathtt{Ifz}\,(e_1, e_2, e_3) \Rightarrow v} \qquad \frac{\rho \vdash e_1 \Rightarrow n \qquad n \neq 0 \qquad \rho \vdash e_3 \Rightarrow v}{\rho \vdash \mathtt{Ifz}\,(e_1, e_2, e_3) \Rightarrow v}$$

Slide 23

Le AND et le OR

En Pascal ces opérateurs sont stricts : ils évaluent leurs arguments. En C et dans la plupart des langages, ils sont paresseux comme la construction Ifz. Le plus simple est de lire e_1 **and** e_2 comme **if** e_1 **then** e_2 **else false**.

Fonctions globales non retournées (en appel par valeur)

Slide 24

Les fermetures peuvent être évitées lorsque les fonctions sont globales et jamais retournées en résultat.

Les seules variables libres d'une fonction sont alors les variables globales. L'environnement de définition est l'environnement global, que l'on peut réinstaller à toute application.

Pour cela, on sépare l'environnement global ρ_0 de l'environnement local ρ et on écrit : $\rho_0; \rho \vdash e \Rightarrow v$.

$$\rho_0; \emptyset \vdash \mathtt{Fun}\,(x, e) \Rightarrow \mathtt{Fun}\,(x, e)$$

$$\frac{\rho_0; \rho \vdash e_1 \Rightarrow \mathtt{Fun}\,(x, e_0) \qquad \rho_0; \rho \vdash e_2 \Rightarrow v_2 \qquad \rho_0; x \mapsto v_2 \vdash e_0 \Rightarrow v}{\rho_0; \rho \vdash \mathtt{App}\,(e_1, e_2) \Rightarrow v}$$

Fonctions locales non retournées

Slide 25

Les fonctions locales non retournées sont un cas intermédiaire.

On peut éviter les fermetures par remontée des variables.

Principe : une fonction $f \stackrel{\triangle}{=} \mathtt{Fun}\,(x, e)$ avec une variable libre y locale à une fonction globale g peut être transformée en une fonction globale $f' \stackrel{\triangle}{=} \mathtt{Fun}\,((x, y), e)$. Les appels à f de la forme $\mathtt{App}\,(f, e)$ sont transformés en $\mathtt{App}\,(f, e, y)$.

Il faut si besoin renommer d'autres liaisons locale de la variable y qui cacheraient la liaison globale y.

Note : Comme les deux programmes ont la même sémantique, on peut utiliser la transformation inverse si le langage le permet, et écrire des programmes où les variables sont "descendues".

Exemple

Deux styles de programmation différents.

Slide 26

Variables remontées

```
let member x l =
  let rec mem = function
    | [] -> false
    | h:: t ->
        x = h || mem t
  in mem l
```

Variables descendues

```
let rec member x =
  function
  | [] -> false
  | h:: t ->
      x = h || member x t
```

En pratique les deux écritures sont utiles (selon le contexte, et le nombre d'arguments).

Un compilateur peut effectuer la transformation de gauche à droite pour éliminer des fonctions locales, ou dans le sens inverse pour sortir des invariants de boucles, par exemple s'il y avait un calcul (sans effet de bord) avant l'appel à la fonction interne.

L'affectation des variables

Slide 27

On peut affecter les variables en Pascal ou C.

Il faut introduire une notion de mémoire σ qui lie des adresses-mémoire ℓ à des valeurs. L'environnement lie maintenant les variables modifiables (ou les références) à des positions mémoire (*left-value*) dont la vraie valeur (*right-value*) est définie dans σ.

On écrit $\rho/\sigma \vdash e \Rightarrow v/\sigma'$ qui se lit *dans l'état-mémoire σ et l'environnement ρ, l'évaluation de l'expression e produit une valeur v et un nouvel état mémoire σ'.*

$$\frac{\rho/\sigma \vdash e_1 \Rightarrow v_1/\sigma_1 \qquad \ell \notin \sigma_1 \qquad \rho, x \mapsto \ell/\sigma_1, \ell \mapsto v_1 \vdash e_2 \Rightarrow v_2/\sigma_2}{\rho/\sigma \vdash \texttt{Let}\,(x, e_1, e_2) \Rightarrow v_2/\sigma_2}$$

Lecture et écriture

$$\frac{x \in dom\ \rho \qquad \rho(x) \in dom\ \sigma}{\rho/\sigma \vdash x \Rightarrow \sigma(\rho(x))/\sigma} \qquad \frac{x \in dom\ \rho \qquad \rho/\sigma \vdash e \Rightarrow v/\sigma'}{\rho/\sigma \vdash \texttt{Affecte}\,(x, e) \Rightarrow v/\sigma', \rho(x) \mapsto v}$$

Implémentation de l'affectation

Slide 28

On peut profiter des strutures mutables du langage hôte et se contenter d'un environnement qui lie les variables à des valeurs mutables (cela revient à utiliser la propre mémoire du langage hôte pour représenter σ).

```
type environnement = (string * valeur ref) list
...
let rec évalue env = function
    ...
  | Variable x -> !(chercher x env)
  | Affecte (x, e) ->
      let v = évalue env e in (trouve x env) := v
```

(Pour simplifier, on a considérer toutes les variables mutables ici.)

Tableaux

Slide 29

Les tableaux alloués dynamiquement sont des structures de données mutables un peu comme les variables. Mais, à la différences des variables, les tableaux peuvent être passés à d'autres fonctions. L'accès à un tableau est explicite (il est implicite pour une variable).

On ajoute trois constructions

- $\texttt{Tableau}(e_1, e_2)$ alloue un nouveau tableau de taille la valeur de e_1 initialisé avec la valeur de e_2.
- $\texttt{Lire}(e_1, e_2)$ lit la case (correspondant à la valeur de) e_2 du tableaux (la valeur de) e_1.
- $\texttt{Ecrire}(e_1, e_2, e_3)$ écrit la case e_2 du tableaux e_1 avec la valeur de e_3.

(Les références de Ocaml sont des tableaux à une seule case.)

Sémantique des tableaux

On introduit de nouvelles valeurs t qui sont des fonctions d'un ensemble d'indices $[0, k-1]$ vers des adresses-mémoire.

Création

Slide 30

$$\frac{\rho/\sigma \vdash e_1 \Rightarrow k/\sigma_1 \qquad \rho/\sigma_1 \vdash e_2 \Rightarrow v/\sigma_2 \qquad \ell_i \notin dom\,(\sigma_2)^{i\in[0,k-1]}}{\rho/\sigma \vdash \texttt{Tableau}\,(e_1, e_2) \Rightarrow (i \mapsto \ell_i)^{i\in[0,k-1]}/\sigma_2, (\ell_i \mapsto v)^{i\in[0,k-1]}}$$

Lecture

$$\frac{\begin{array}{c}\rho/\sigma \vdash e_1 \Rightarrow t/\sigma_1 \\ \rho/\sigma_1 \vdash e_2 \Rightarrow k/\sigma_2 \qquad k \in dom\,(t) \qquad t(k) \in dom\ \sigma_2\end{array}}{\rho/\sigma \vdash \texttt{Lire}\,(e_1, e_2) \Rightarrow \sigma_2(t(k))/\sigma_2}$$

Écriture similaire

Interprétation des tableaux

- On profite des tableaux mutables du langage hôte.
- Il y a plusieurs formes de valeurs qu'il faut distinguer par un type somme.
- L'accès à une valeur du mauvais type produit une erreur (que le typage devra empêcher).
- L'accès en dehors des bornes produit également une erreur (que le typage ne détecte pas en général).

Slide 31

Distinguer les valeurs (par un type somme)

```
(* les différents types de valeur *)
type valeur = Entier of int | Tableau of valeur array
type erreur = ... | Type | Indice
(* pour retirer une valeur de la bonne forme *)
let entier = function Entier n -> n | _ -> erreur Type
let tableau = function Tableau t -> t | _ -> erreur Type
```

Tableaux (suite)

Slide 32

```
let rec évalue env =
    ...
  | Tableau (e_1, e_2) ->
      let k = entier (évalue env e_1) in
      let v = entier (évalue env e_2) in
      Tableau (Array.create k v)
  | Lire (e_1, e_2) ->
      let t = tableau (évalue env e_1) in
      let k = entier (évalue env e_2) in
      if 0 <= k && k < Array.length t
      then Tableau (Array.create k v)
      else erreur Indice
  | Ecrire (e_1, e_2) ->
      ...
```

Variables et tableaux non initialisées

Slide 33

En Pascal ou en C, les variables et les tableaux ne sont pas toujours initialisés.

Formellement, on introduit une valeur indéfinie et l'accès à une variable ou à une case non définie produit une erreur.

Ordre d'évaluation

La présence d'effets de bords (affectation) fixe l'ordre d'évaluation !

$$\frac{\rho/\sigma \vdash e_1 \Rightarrow \langle \mathtt{Fun}\,(x, e_0), \rho_0\rangle/\sigma_1 \qquad \rho/\sigma_1 \vdash e_2 \Rightarrow v_2/\sigma_2 \qquad \rho_0, x \mapsto v_2/\sigma_2 \vdash e_0 \Rightarrow v/\sigma'}{\rho/\sigma \vdash \mathtt{App}\,(e_1, e_2) \Rightarrow v/\sigma'}$$

Le langage Pseudo-Pascal (PP)

C'est un pascal simplifié, pour limiter le nombre de constructions.

Description informelle

Slide 34

- Syntaxe Pascal.
- Valeur entières et booléens ; ni chaînes ni flottants
- Des tableaux dynamiques (durée de vie infinie). Pas d'enregistrements.
- Les fonctions sont globales, mutuellement récursives et jamais retournées.
  Le résultat d'une fonction est passé en affectant la variable de même nom.
- Le passage d'argument est toujours en appel par valeur.
- Les variables sont toutes mutables.

Arbre de syntaxe abstraite

Slide 35

On conserve les informations de types (pour permettre une analyse de type ultérieure)

```
type type_expr = Integer | Boolean | Array of type_expr;;
```

Un programme est composé d'un ensemble de déclarations de variables et de fonctions et d'un corps (une instruction).

```
type var_list = (string * type_expr) list
type program = {
   global_vars : var_list ;
    definitions : ( string * definition ) list ;
    main : instruction ;   (* corps du programme *) }
and definition = {
    arguments : var_list ; result : type_expr option;
    local_vars : var_list ;
    body : instruction ;   (* corps de la fonction *) }
```

Suite (expressions)

Slide 36

```
and expression =
    (* constantes *)
  | Int of int | Bool of bool
    (* opérations binaires *)
  | Bin of binop * expression * expression
    (* accès à une variable *)
  | Get of string
    (* appel de fonction *)
  | Function_call of string * expression list
    (* accès dans un tableau à une position *)
  | Geti of expression * expression
    (* Création d'un tableau d'une certaine taille *)
  | Alloc of expression * type_expr
and binop = Plus | Minus | Times | Div
  | Lt | Le | Gt | Ge | Eq | Ne
```

Suite (expressions)

Slide 37

```
  and instruction =
      (* Affectation d'une variable *)
    | Set of string * expression
      (* Suite d'instructions *)
    | Sequence of instruction list
    | If of expression * instruction * instruction
    | While of expression * instruction
      (* Appel de procédure *)
    | Procedure_call of string * expression list
      (* Ecriture d'un entier *)
    | Write_int of expression
      (* Lecture d'un entier dans une variable *)
    | Read_int of string
      (* Affectation dans un tableau *)
    | Seti of expression * expression * expression
```

Exercices

Interprète pour le langage PP

La syntaxe abstraite est donnée par `pp.mli`.

Les interfaces sont imposées `interpret.mli`

Quelques programmes tests : `fact.ml`, `fib.ml`, et `memo_fib.ml`

Slide 38 Écrivez-en quelques autres.

Extensions du langage langage PP

Considérer un langage PP^+ plus évolué et le traduire dans le langage PP. On pourra

- Autoriser une construction de liaison locale `Let` (x, e_1, e_n) et la tradiore en des déclarations de variables au niveau la définition de fonction ou au niveau global.
- Autoriser les fonctions locales et les traduire en fonctions globales par remontée des variables.

Projets de compilation

But :

- Mieux maîtriser l'ensemble de la chaîne d'un compilateur.
- Apporter votre solution personnelle à un problème concret.

Points communs (pour de nombreux projets) :

- Modification légère de l'ensemble.
- Écriture, ré-écriture, ou modification importante d'un maillon. **Slide 39**
- Quelques sujets plus indépendants demande d'écrire une autre chaîne de compilation.

Choisissez

- Un sujet qui vous plaît
- En amont ou en aval de la chaîne de compilation.
- À votre mesure (les difficultés sont **très** variées).

Pleins de sujets

Slide 40

1. Implémenter (un mini) Lex.
2. Implémenter (un mini) Yacc.
3. Transformer les fonctions locales en fonctions globales au niveau du langage source.
4. Ajouter des enregistrements, des chaînes de caractères, des caractères (alloués dans le tas ou en pile).
5. Compiler un Pseudo Pascal Fonctionnel ou un mini ML (fonctions locales retournés avec fermetures ou environnements chaînés en pile)
6. Optimisations du code source (évaluation partielle, ...)
7. Remplacer une solution approchée d'un des maillons par une solution plus efficace (produisant un meilleur résultat).
8. Écrire un vérificateur de types pour Pseudo Pascal.

Slide 41

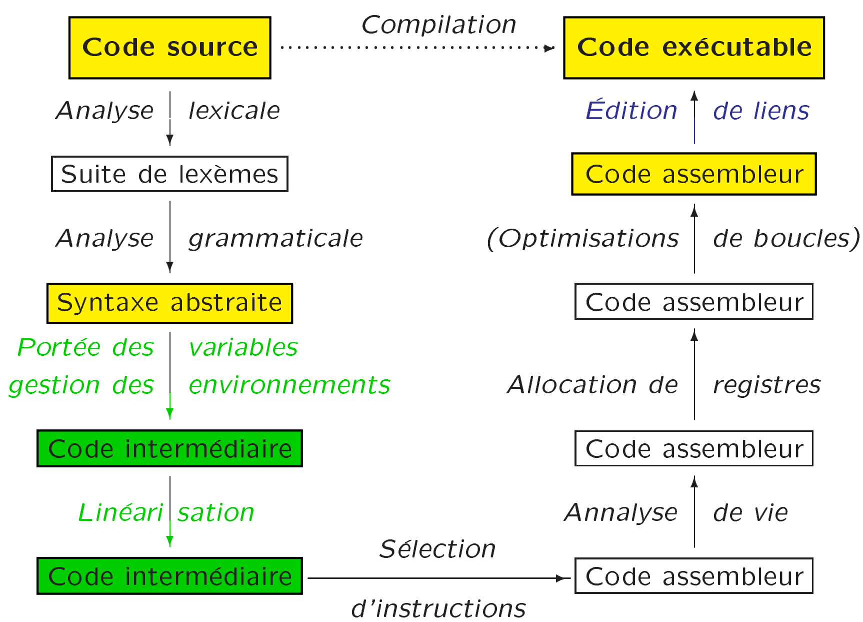
9. Gestionnaire de mémoire (par compteur de référence ou stop and copy)
10. Compiler vers une autre architecture, ou vers une machine réelle (par exemple x86)

∞. **Proposez votre projet.**

5

Code Intermédiaire Génération de Code Linéarisation Canonisation.

Slide 1



Difficulté de la génération de code

- Gestion des environnements : conventions d'appel des fonctions, sauvegarde en pile, etc.
- Gestion des registres.
- Mécanisation des optimisations.

Slide 2

Code Intermédiaire

Découpage en petites étapes

$$\text{SA}\ \overset{(1)}{\underbrace{\xrightarrow{\text{Générer du code}} \text{CI} \xrightarrow{\text{Canoniser Linéariser}}}_{\text{Code intermédiaire linéaire, inefficace}}}\ \text{CI}\ \overset{(2)}{\underbrace{\xrightarrow{\text{Sélection d'instruct.}} \text{CA} \xrightarrow{\text{Allocation de registres}}}_{\text{Code machine Efficace}}}\ \text{CA}$$

Slide 3

Approche systématique : robuste, modulable, réutilisable.

Approche globale :

- La phase (1) introduit des sources d'inefficacités
- éliminées par la phase (2).

Pourquoi un code intermédiaire ?

Slide 4

Partage du travail

– Entre plusieurs langages, et plusieurs machines

$$\left.\begin{array}{r}\text{Java}\\ \text{C}\\ \dots\\ \text{Caml}\\ \text{Ocaml}\end{array}\right\}\ \boxed{\text{CI}}\ \left\{\begin{array}{l}\text{MIPS}\\ \text{xx86}\\ \dots\\ \text{Sparc}\\ \text{Byte-code}\end{array}\right.$$

– Entre les différentes constructions d’une même langage ou d’une même machine
  – La syntaxe abstraite comporte trop de constructions voisines
  – Le code assembleur comporte trop d’instructions voisines

  La représentation intermédiaire est plus concise.

Le principe du code intermédiaire

Slide 5

Le but du code intermédiaire est de passer d’une structure arborescente à une (bonne) structure linéaire et de préparer la sélection d’instructions.

Les détails dépendant de l’architecture sont relégués à une phase ultérieure de sélection d’instructions.

Quelques caractéristiques du code intermédiaire :
– Les branchements sont explicites.
– Code arborescent (expressions) ou linéaire (instructions)
– Utilise une infinité de registres (temporaires), dont l’utilisation est privilégiée (réversible) et le coût négligé.
– L’adressage en mémoire est une forme séparée qui n’est retenue que lorsque c’est indispensable (irréversible).
– L’appel de fonction est implicite, et sera résolu dans une phase ultérieure.
  (Différentes formes dans différents langages d’assemblage)

Les expressions (Code.exp)

Slide 6

```
type exp =
     Const of int                  (* Entiers et Booléens *)
   | Name of label                 (* Globaux *)
   | Temp of temp                  (* Lecture d'un temporaire *)
   | Mem of exp                    (* Lecture mémoire *)
   | Bin of binop * exp * exp      (* Opération binaire *)
   | Call of frame * exp list      (* Appel de fonction *)
```

Pour certains langages, il est utile d'ajouter un nœud

```
   | Eseq of stm * exp
```

pour représenter une séquence retournant un résultat (comme `e1; e2` *en ML). Ce n'est pas nécessaire pour Pseudo-Pascal.*

Les instructions (Code.stm)

Slide 7

```
and stm =
   | Label of label                (* Étiquette *)
   | Move_temp of temp * exp       (* Écriture de exp dans un
                                      temporaire *)
   | Move_mem of exp * exp         (* Écriture en mémoire à
                                      une adresse calculée *)
   | Seq of stm list               (* Séquence d'instructions *)
   | Exp of exp                    (* Expression avec effet *)
   | Jump of label
   | Cjump of relop * exp * exp * label * label

and relop = Req | Rne | Rle | Rge | Rlt | Rgt
and binop = Plus | Minus | ...  | Eq | Ne
```

Utilisation des temporaires

Les étiquettes et les temporaires sont créés à la demande. Par sécurité on les manipulera de façon abstraite. Penser

temporaire = registre virtuel

Les temporaires seront associés à des registres réels, et éventuellement à des emplacements en pile pour les sauvegarder.

Slide 8

La génération de code crée beaucoup de temporaires, mais de durée de vie très courte. Les temporaires avec une durée de vie très courte n'auront pas besoin d'être sauvegardés en pile.

Plus on en crée, plus leur durée de vie sera courte et plus facilement ils pourront partager le même registre : ne jamais chercher à réutiliser le même temporaire, au contraire utiliser des temporaires différents dès que possible !

Première partie
Génération de code.

Génération de code

On traduit récursivement expressions et instructions :

$[\![\_]\!]^e$: $\texttt{Pp.expression} \to \texttt{Code.exp}$

$[\![\_]\!]^s$: $\texttt{Pp.instruction} \to \texttt{Code.stm}$

de façon la plus simple possible.

Slide 9

$[\![\texttt{Int } n]\!]^e = \texttt{Const } n \quad [\![\texttt{Bool } \mathit{true}]\!]^e = \texttt{Const } 1 \quad [\![\texttt{Bool } \mathit{false}]\!]^e = \texttt{Const } 0$

$$\begin{array}{rcl} [\![\texttt{Bin}\,(op, e_1, e_2)]\!]^e & = & \texttt{Bin}\,(op, [\![e_1]\!]^e, [\![e_2]\!]^e) \\ [\![\texttt{Sequence}\,(s_1; \ldots; s_n)]\!]^s & = & \texttt{Seq}\,[[\![s_1]\!]^s; \ldots; [\![s_n]\!]^s] \end{array}$$

Opérations sur les tableaux (alloués dans le tas) :

$$\begin{array}{rcl} [\![\texttt{Geti}\,(e_1, e_2)]\!]^e & = & \texttt{Mem}\,(\texttt{Bin}\,(\texttt{Plus}\,, [\![e_1]\!]^e, [\![e_2]\!]^e)) \\ [\![\texttt{Seti}\,(e_1, e_2, e_3)]\!]^s & = & \texttt{Move\_mem}\,(\texttt{Bin}\,(\texttt{Plus}\,, [\![e_1]\!]^e, [\![e_2]\!]^e), [\![e_3]\!]^e) \end{array}$$

Expressions conditionnelles

Test simple :

$$[\![\texttt{If}\,(\texttt{Bin}\,(op, e_1, e_2), s_t, s_f)]\!]^s = \texttt{Seq}\left[\begin{array}{l} \texttt{Cjump}\,(op, [\![e_1]\!]^e, [\![e_2]\!]^e, l_t, l_f); \\ \texttt{Label } l_t; [\![s_t]\!]^s; \texttt{Jump } fi; \\ \texttt{Label } l_f; [\![s_f]\!]^s; \texttt{Jump } fi; \\ \texttt{Label } fi; \end{array}\right]$$

Slide 10

Test complexe :

$$[\![\texttt{If}\,(e_1, s_t, s_f)]\!]^s = [\![\texttt{If}\,(\texttt{Bin}\,(\texttt{Ne}\,, e_1, \texttt{Const } 0), s_t, s_f)]\!]^s$$

La boucle while se traite de façon similaire.

Gestion des environnements

Pour comprendre la traduction des variables, il faut comprendre comment sont traités les arguments et les variables locales des fonctions.

Cela dépend des constructions du langage.

Par exemple :

Slide 11

- L'appel par référence (ou `&x` en C) permet de passer l'adresse plutôt que la valeur d'une variable. Ainsi, une variable contient tantôt la valeur cherchée tantôt l'adresse à laquelle chercher cette valeur.
- Les tableaux en pile sont (en général) passés par référence.
- Les fonctions locales obligent à passer par référence soit :
  - les variables qui s'échappent dans la fonction locale (remontée des variables),
  - le lien statique (fermeture en pile ≈ tableau en pile).

Cas simplifié de Pseudo-Pascal

Slide 12

En Pseudo-Pascal, il n'y a pas de fonctions locales,
- les arguments sont passés par valeurs, (*i.e.* le contenu des variables et non la boîte).
- les tableaux sont alloués dans le tas et passés par référence (*e.g.* passe leur adresse mais on ne les recopie pas).

En particulier, une variable ne peut pas être directement affectée par un appel de fonction (mais la partie mutable d'une valeur associée à une variable peut être modifiée).

On place les variables dans des temporaires (qui en général seront des registres).

On accède à (on écrit dans) à une variable en lisant (en écrivant dans) le temporaire associé.

Compilation des accès

L'accès à une variable locale est donc toujours la lecture d'un temporaire. Son numéro est trouvé dans l'environnement de compilation (cf. analogue à l'environnement d'évaluation) qu'il faut passer comme argument à la fonction de compilation.

$$[\![\texttt{Get}\ x]\!]^e{}_\rho = \texttt{Temp}\ t \qquad \text{si } \rho(x) = t$$

Slide 13

(On remplace partout ailleurs $[\![e]\!]^e$ et $[\![e]\!]^s$ par $[\![e]\!]^e{}_\rho$ et $[\![e]\!]^s{}_\rho$.)

Les globaux sont mémorisés dans un tableau statique alloué au début du programme à l'adresse symbolique G. Dans ρ les globaux sont définis par leur position dans le tableau (w est la taille du mot).

$$[\![\texttt{Get}\ x]\!]^e{}_\rho = \texttt{Mem}\,(\texttt{Bin}\,(\texttt{Plus}\,,\texttt{Name}\ G, \texttt{Const}\,(n * w)) \qquad \text{si } \rho(x) = n$$

On pourrait aussi placer $\texttt{Name}\ G$ dans un temporaire spécial réservé $\texttt{Temp}\ t_G$.

Mise en œuvre

Dans l'environnement de compilation, il faut associer aux variables (chaînes de caractères) des emplacements.

Ce dont sont des temporaires pour les variables locales, soit un indice dans la table globale. On utilise un type somme :

Slide 14

```
type access = Local of Gen.temp | Global of int
let rec translate_expr  env = function
   ...
 | Get x ->
    begin match Env.find_var env x with
     | Local t -> Temp t
     | Global n ->
       let g = Frame.global_space and w = Frame.word_size
       in Mem (Bin (Name g , Const (n * w)))
    end
```

Compilation des affectations

Slide 15

De façon analogue, c'est l'écriture :
- dans le temporaire associé à la variable considérée, pour les accès locaux ;
- dans la case du tableau des globaux associé à la variable considérée.

Formellement :

$$[\![\texttt{Set}\ x\ e]\!]^s{}_\rho =$$

$$\begin{array}{ll} \texttt{Move\_temp}\,(t, [\![e]\!]^e{}_\rho) & \text{si } \rho(x) = t \\ \texttt{Move\_mem}\,(\texttt{Bin}\,(\texttt{Plus}\,, \texttt{Name}\ G, \texttt{Const}\,(n * w)), [\![e]\!]^e{}_\rho) & \text{si } \rho(x) = n \end{array}$$

Utilisation la pile en Pseudo-Pascal

Slide 16

La pile ne sert plus qu'à passer les arguments (> 4) et à sauvegarder les registres pendant les appels récursifs, ou lorsqu'il y a trop de temporaires.

Ces utilisations de la pile seront traités *ultérieurement* par la compilation des appels de fonction et par l'allocateur de registres.

En effet, dans le code intermédiaire,
- On utilise un nombre arbitraire de temporaires, que l'on suppose sauvegardés pas les appels de fonctions.
- On utilise une instruction de haut niveaux pour les appels de fonctions ; elle devra être remplacée ultérieurement par du code qui place les arguments à des positions conventionnelles et sauvergarde, si nécessaire, les valeurs des temporaires pendant les appels (*e.g.* récursifs).

Compilation des fonctions

Lors de l'appel de fonction, il faut communiquer entre l'appelant et l'appelé pour passer les arguments et recevoir les résultats. Par exemple les premiers arguments sont passés dans les registres a_0 à a_3, les suivants sur la pile.

Slide 17

- Avant l'appel les arguments sont dans des temporaires de l'appelant.
- Après, ils sont dans des temporaires de l'appelé, a priori différents (sauf avec allocation de registres inter-procédurale).

L'interface entre la vue externe et la vue interne, *i.e.* le (code de) transfert des arguments entre l'appelant et l'appelé, sera inséré ultérieurement en fonction de la machine cible, lors de la traduction des instructions `Call` et en ajoutant un prologue et un épilogue à chaque fonction.

Slide 18

L'unité de traduction en code intermédiaire est la fonction ou procédure.

Slide 19

Exemple

Fonction succ en Pseudo-Pascal (syntaxe abstraite) :

```
{ arguments = [ "n", Integer ];    result  = Some Integer;
  local_vars  = [  ];
  body = Set ("succ", Bin (Plus, Get "n", Int 1)) }
```

Son code intermédiaire :

```
[ Label succ;
  Move_temp
    (t104, Code.Bin (Code.Plus, Temp t105, Const 1));
  Jump succ_end ]
```

où

t104	emplacement de l'argument `n`
t105	emplacement du résultat de `succ`
succ	adresse du prélude (utilisée pour l'appel)
succ_end	adresse du prologue (utilisée pour le retour)

Slide 20

Exemple (suite)

Appel de la fonction `succ` ;

```
  Function_call  ("succ", Const 2)
```

Code de l'appel :

```
  Call  (<succ>, [Const 3])]
```

C'est à la charge de l'instruction Call de placer les arguments dans le temporaire t104 et de récupérer le résultat dans le temporaire t105.

Cela sera réalisé dans une phase ultérieure de la compilation, après l'allocation des registres, par le prélude et le prologue.

Slide 21

Compilation des appels

L'appel de fonction est compilé par une expression `Call`

$$[\![\texttt{Function\_call}\,(g, e_1, \ldots, e_n)]\!]^e = \texttt{Call}\,(F_g, [\![e_1]\!]^e, \ldots, [\![e_n]\!]^e)$$

où F_g est une structure décrivant g (appelée *frame*) comprenant :

- L'étiquette pour entrer dans la fonction (celle placée en tout début du code de g). Plus tard, on y insérera un prélude (avec la sauvegarde éventuelle de certains registres).
- Une étiquette pour sortir de la fonction. Plus tard on y insérera le prologue (restauration des registres)
  Pour sortir d'une fonction il faut toujours faire un saut au prologue. (c'est convention que nous prenons à ce stade.)
- L'emplacement exact des arguments dans la vue interne : position en pile, ou numéro de temporaire (il s'agit toujours d'un temporaire en Pseudo-Pascal).
- Position du résultat (si c'est une fonction).

Slide 22

Mise en œuvre

Le type frame est implémenté dans frame.ml

```
type frame =
    { name : Gen.label;
      return_label : Gen.label;
      args : address list ;
      result : address option;
      mutable locals : int ; }
```

Mais, on les manipule de façon abstraite (interface frame.mli) pour permettre une modification ultérieure si nécessaire.

Le frame est recherché dans l'environnement de compilation :

```
| Function_call (f, args) ->
    let frame_f = Env.find_definition env f in
    let args_compilés = List.map (translate_expr env) args
    in Call (g, args_compilés)
```

Compilation du corps des fonctions

Pour chaque fonction :

1. Créer un nouveau frame, avec des emplacements internes pour les arguments, et le résultat (optionnel). (Ce sont simplement des temporaires –fraîchement alloués– en pseudo-pascal)
2. Attribuer des emplacements aux variables locales (idem)
3. Augmenter l'environnement de compilation.
4. Compiler le corps dans cet environnement.

Slide 23

Récursion

Comme les fonctions sont récursives en PP, l'étape 1 (allocation des frames) est faite en premier pour toutes les définitions, avant d'effectuer les autres phases pour chaque définition.

Compilation dans le cas général

Pseudo-pascal est un cas simplifié :

- Les arguments sont passés par valeurs.
- Les tableaux sont passés par références.

Variables passées par référence : Il faut distinguer l'adressage direct et indirect (*i.e. Left-* et *Right-value*).

Slide 24

Tableaux alloués dans la pile : Idem. De plus, toutes les variables allouées n'ont plus la même taille (ce qu'il faut gérer).

Fonctions locales (non retournées comme valeur) Au choix :

- chaîner les blocs d'activation en pile (cf liens statiques), ou
- remonter les variables pour rendre toutes les fonctions globales.

Les variables qui s'échappent sont toujours placées en pile.

Fonctions comme valeur (ML) : fabriquer des fermetures.

Liens statiques

Slide 25

Pour les fonctions locales non retournées, la fermeture peut être allouée en pile. Pour simplifier, supposons que toutes les variables de f s'échappent, par exemple, elles sont utilisées par une fonction g locale à f, et sont placées en pile (x_1, ... x_n) :

	$\vdots$	
pointeur de bloc $\rightarrow$	ℓ	Lien statique
	x_1	
	$\vdots$	Variables de f
	x_n	
pointeur de pile $\rightarrow$	$\vdots$	Registres sauvegardés
$\downarrow\uparrow$	$\vdots$	autres appels
		appel à g

Explication par remontée des variables

Slide 26

Par remontée des variables il faut passer à g en plus de ses arguments $y_1, \ldots y_q$, les adresses $\&x_1, \ldots \&x_q$ des variables de f.

À condition de figer statiquement leur position en pile, il suffit de passer à g l'adresse du bloc d'activation de f (l'environnement dynamique dans lequel est défini g) qui marque le début des variables de f. Vu de g, on l'appelle le **lien statique de** g.

La fonction g retrouve les x_i par décalage (calculer statiquement) par rapport au lien statique (passé dynamiquement).

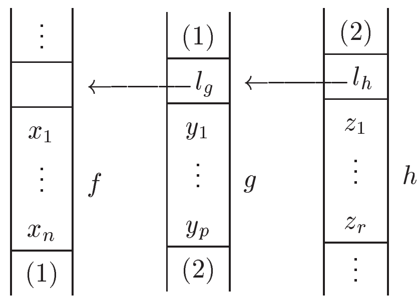
- elle est compilée comme si elle recevait en plus de ses arguments un enregistrement ℓ_g dont les champs sont x_1, .. x_n.
- elle est appelée en passant en plus des arguments y_1, ... y_q le lien statique ℓ_g.

Vu comme une fermeture en pile

Au lieu d'être représentés à plat les environnements sont vus comme une chaîne de petits environnements à plat.

Il peuvent être directement implémentés dans la pile tant que les fonctions ne sont pas retournées en résultat (leur durée de vie dynamique n'excède par leur portée lexicale).

Slide 27



L'unité de compilation

Par simplicité, l'unité de compilation est la procédure : nous ne ferons pas ici d'optimisations ni d'allocation de registres inter-procédurale.

Chaque unité de compilation comporte un frame (bloc d'activation).

Slide 28

On traite le corps principal de la même façon, en le transformant en une procédure qui ne reçoit pas d'arguments et sans variables locales.

Les optimisations inter-procédurales sont possibles, mais plus complexes et plus coûteuses.

Le programme principal

Slide 29

Les étapes

- Décider des emplacements pour les globaux,
- Allouer les frames des fonctions globales,
- Créer l'environnement global,
- Compiler le corps des procédures (y compris le corps du programme)

Le résultat de la compilation : une structure décrivant :

- le nombre de globaux.
- la procédure principale (distinguée) et les sous-procédures, chacune comportant un frame et son code (intermédiaire).

```
type 'a procedure = Frame.frame * 'a
type 'a program =
    { number_of_globals : int;
      main : 'a procedure; procedures : 'a procedure list }
val program : Pp.program -> Code.code program
```

Les primitives

Slide 30

Dans le code intermédiaire, on traite les primitives comme des appels à des fonctions externes.

Il suffit donc de se donner une étiquette réservée par primitive (en fait un frame dont seule l'étiquette importe).

$$[\![\texttt{Alloc}\ e)]\!]^e = \texttt{Call}\,(alloc, [\![e]\!]^e)$$

où $alloc$ est le frame de la primitive `alloc` (créé une seule fois).

Le fichier frame.ml contient les déclarations :

```
let write_int   = Frame.new_primitive "write_int"   1 false
let writeln_int = Frame.new_primitive "writeln_int" 1 false
let read_int    = Frame.new_primitive "read_int"    0 true
let alloc       = Frame.new_primitive "alloc"       1 true
```

(le premier argument indique l'arité de la primitive, le second si elle retourne ou non un résultat.)

Linéarisation.

Linéarisation (but)

Slide 31

- Les expressions permettent des structures emboîtées telles que $\texttt{Call}\,(f, e_1) + \texttt{Call}\,(g, (\texttt{Call}\,(h, e_2)))$. Or, un appel de fonction est une opération complexe, qui produit des effets de bords (écriture dans des registres).
- Dans la plupart des langages (mais pas en Pseudo-Pascal) les expressions peuvent contenir des séquences. Par exemple $e_1 + (s; e_2)$ en ML. À nouveau s peut modifier la valeur des registres utilisés pour évaluer e_1.

La linéarisation est une phase postérieure à la génération de code qui :

- extrait les appels de fonctions et les séquences des expressions.
- aplatit les séquences d'instructions.

Exemple

Le code suivant est non linéaire (il contient des instructions Call dans les expressions).

$$\texttt{Bin}\,(\texttt{Plus}\,,\texttt{Call}\,(f,e_1),\texttt{Call}\,(g,(\texttt{Call}\,(h,e_2))))$$

La linéarisation le transforme en :

Slide 32

$$\begin{array}{l}
\texttt{Move}\,(\texttt{Temp}\ t_1,e_1);\\
\texttt{Move}\,(\texttt{Temp}\ t_2,\texttt{Call}\,(f,t_1));\\
\texttt{Move}\,(\texttt{Temp}\ t_3,e_2);\\
\texttt{Move}\,(\texttt{Temp}\ t_4,\texttt{Call}\,(h,t_3));\\
\texttt{Move}\,(\texttt{Temp}\ t_5,\texttt{Call}\,(g,t_4)),\\
\texttt{Bin}\,(\texttt{Plus}\,,\texttt{Temp}\ t_1,\texttt{Temp}\ t_5)
\end{array}$$

Linéarisation (méthode)

Par réécriture du code. La règle principale d'extraction

$$\texttt{Call}\,(f,e_1,e_2) \longrightarrow \texttt{Move\_temp}\,(t,\texttt{Call}\,(f,e_1,e_2)),\texttt{Temp}\ t$$

Plus généralement l'extraction retire d'une expression e une séquence s en fournissant une expression résiduelle e'. Elle procède récursivement pour atteindre les expressions les plus internes :

Slide 33

$$\frac{e_1 \longrightarrow s, e_1'}{e_1 + e_2 \longrightarrow s, (e_1' + e_2)} \qquad \frac{e_2 \longrightarrow s, e_2' \qquad \boxed{(e_1, s)\ \text{commutent}}}{e_1 + e_2 \longrightarrow s, (e_1 + e_2')}$$

Pour préserver l'ordre d'évaluation, il faut aussi extraire les expressions qui se trouve sur le chemin et qui ne commutent pas :

$$\frac{e_2 \longrightarrow s, e_2'}{e_1 + e_2 \longrightarrow \texttt{Move\_temp}\,(t, e_1); s, (\texttt{Temp}\ t + e_2')}$$

(Règle à n'appliquer qu'en dernier recours.)

Commutation

Slide 34

Il s'agit de ne pas permuter l'évaluation d'une instruction et d'une expression lorsque cela risque de changer le sens du programme.
- typiquement, par le changement de l'ordre lecture-écriture,
- éventuellement, par la levée d'une exception.

Une assez bonne approximation est qu'une expression constante qui ne lit ni n'écrit commute avec toute instruction.

(A priori une instruction extraite écrit quelque chose, donc le cas sans écriture qui commute évidemment n'est pas intéressant.)

Pour être plus fin, on peut autoriser des lectures et écritures dans des emplacements (temporaire ou mémoire) disjoints.

Linéarisation (suite)

Slide 35

On procède de même pour toutes les expressions pouvant contenir des expressions complexes (call) aux feuilles, et pour les instructions qui peuvent contenir des expressions...

$$\frac{e_1 \longrightarrow s, e_1'}{\mathtt{Exp}\,(e_1) \longrightarrow s; \mathtt{Exp}\ e_1'} \qquad \frac{i \longrightarrow s'; i'}{s; i \longrightarrow s; s'; i'} \qquad \textit{etc.}$$

On en profite pour aplatir les séquences

$$\frac{s \longrightarrow s'}{s_0; \mathtt{Seq}\,(s) \longrightarrow s; s'}$$

Attention à ne pas extraire les expressions déjà extraites, *i.e.* les `Call` qui apparaissent déjà dans l'une des deux configurations :

$$\mathtt{Move}\,(\mathtt{Temp}\ t, \mathtt{Call}\,(f, \mathtt{Temp}\ t_i)) \qquad \text{ou} \qquad \mathtt{Exp}\,(\mathtt{Call}\,(f, \mathtt{Temp}\ t_i))$$

Mise en œuvre de la linéarisation

Slide 36

Les fonctions principales rewrite_stm et rewrite_exp décrivent les règles de réécriture en utilisant le pattern matching.

```
let rec rewrite_exp : exp -> stm list * exp =
  function
    | Call (f, el) ->
        let s, el' = rewrite_args el in
        let t = G.new_temp() in
        s @ [Move_temp (t, Call (f, el'))], Temp t
    | Bin (binop, e1, e2) ->
        let s, [e1';e2'] = rewrite_args args in
        s, Bin (binop, e1', e2')
    | .......
```

Mise en œuvre (suite)

Slide 37

```
and rewrite_stm : stm -> stm list =
  function
    | Move_temp (t, Call (f, el)) ->
        let s, el' = rewrite_args el in
        s @ [Move_temp (t, Call (f, el'))]
    | Move_temp (t, e1) ->
        let s, e1' = rewrite_exp e1 in
        s @ [Move_temp (t, e1')]
              ......
```

La procédure auxiliaire `rewrite_args` réécrit une liste d'expressions (arguments) :

```
and rewrite_args : exp list -> stn list * exp list = ...
```

(Elle doit gérer les problèmes de commutation)

Blocs élémentaires
Calcul des traces .

Les blocs de base

Slide 38

Un bloc de base est une séquence d'instructions linéaire (une seule entrée, une seule sortie) la plus longue possible. Il commence par une étiquette, fini par un saut et ne comporte ni étiquette (point entrée) ni saut (point de sortie) au milieu.

Le calcul des blocs de base est immédiat : il suffit de parcourir le code en commençant par le début, et chaque instruction de saut ou étiquette termine le bloc courant et en commence un nouveau.

On ajoute si nécessaire des instructions Jump lorsque le bloc courant est fermé par la rencontre d'une étiquette.

Les traces

Slide 39

Une trace d'exécution est un parcours le plus long possible en enchaînant les blocs de bases les uns à la suite des autres sans jamais repasser par un bloc déjà vu.

Cela revient à un parcours de graphe orienté où les sommets sont les blocs de base et les arcs relient deux blocs lorsqu'il existe un saut du premier vers le second.

On visite alors tous les noeuds du graphe, en allant le plus loin possible sans jamais repasser par un nœud déjà vu.

Nettoyage des traces

Slide 40

Les traces mettent bout à bout des sauts et le bloc suivant. On trouve donc fréquemment la configuration suivante que l'on simplifie (économiser des sauts étant un des buts recherchés) :

$$\texttt{Jump}\ l; \texttt{Name}\ l \longrightarrow \texttt{Name}\ l$$

Les instructions de saut conditionnel de tous les assembleurs prennent l'instruction qui suit immédiatement pour branche "false". On réarrange les sauts conditionnels tels qu'ils soient toujours suivis par leur branche false, soit en inversant les tests (suffit en général) soit en introduisant une étiquette intermédiaire.

$$\texttt{Cjump}\,(op, e_1, e_2, l_1, l_2); \texttt{Name}\ l_1 \longrightarrow \texttt{Cjump}\,(neg(op), e_1, e_2, l_2, l_1); \texttt{Name}\ l_1$$

$$\texttt{Cjump}\,(op, e_1, e_2, l_1, l_2) \longrightarrow \texttt{Cjump}\,(op, e_1, e_2, l_1, l_3); \texttt{Name}\ l_3; \texttt{Jump}\ l_2$$

Traces optimales

Certaines traces sont meilleures que d'autres.

En effet, il est plus important de minimiser les sauts dans le code fréquemment parcouru, par exemple dans les boucles les plus internes.

Slide 41

Une telle boucle devra de préférence être toute entière dans sa trace. (Un seul saut pour le retour dans le cas normal).

Une meilleure approximation consisterait à rechercher les boucles les plus internes par un tri topologique sur les blocs de base. Puis à commencer par linéariser dans l'ordre de *domination*.

Calcul incrémental des traces

Dans un langage structuré (sans instruction goto), il n'est pas possible d'entrer dans un bloc "par le milieu". On peut se passer du du calcul des traces, et généré dès le départ du code linéaire canonique...

Slide 42

Le résultat est comparable, voir meilleur qu'un calcul des traces automatique mais naïf, mais moins bon qu'un calcul qui utilise les dominateurs.

De plus, l'approche est moins modulaire, car le calcul des traces est est délocalisé dans la compilation de chaque construction de branchement, alors que dans l'approche générale il ne dépend que du code intermédiaire et est indépendant de la compilation de la syntaxe abstraite en code intermédiaire.

Exercices

Exercice 10 (Génération du code intermédiaire) *Écire un générateur de code intermédiare pour Pseudo-Pascal. Le code intermédiaire est donné par l'interface code.mli.*

La gestion des étiquettes, des environnements et des frames sont données par les fichiers env.mli, gen.mli et frame.mli. □

Slide 43

Exercice 11 (Linéarisation et canonisation) *Implémenter la linéarisation, le calcul des blocs de base et la mise sous forme canonique.*

On impose l'interface canon.mli. □

Exercice 12 (Interprète du code intermédiaire) *Écire un interprète pour le code intermédiaire. Celui-ci devra fonctionner aussi bien sur du code arborescent que sur du code linéaire. On pourra également écrire un imprimeur du code intermédiaire.* □

Génération de code intermédiaire

Librairies fournies

- La syntaxe abstraite de Pseudo-Pascal pp.mli
- La gestion des environnements env.mli et env.ml
  (Reprend les fonctions écrites au td2 pour l'interprète.)
- La gestions des étiquettes et temporaires gen.mli et gen.ml.

Slide 44

```
type temp
val new_temp : unit -> temp

type label
val new_label: unit -> label
val named_label : string -> label
```

– La gestion des blocs d'activation frame.ml et frame.mli

```
type frame
val new_frame : var_list   -> type_expr option -> frame
val named_frame
      : string  -> var_list  -> type_expr option -> frame
val frame_name : frame -> label
val frame_result  :  frame -> temp option
val frame_return :  frame -> label
```

Slide 45

– La définition du code intermédiaire code.mli

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

```
cp ~remy/compil/td3/* ./
```

On vous donne une implémentation très incomplète trans.ml et son interface trans.mli.

Installation de l'ensemble

Vérifier d'abord que l'ensemble est bien installé. La version incomplète de `trans.ml` est compilable.

On pourra utiliser ce Makefile et exécuter

```
make
ocaml un.mlx
```

Slide 46

Pour voir le code produit

```
ocaml < un.mlx
```

(et voici le fichier `run.ml`)

Compléter le fichier trans.ml

On traitera d'abord les expressions et les instructions simples, puis les variables globales, afin de traiter le programme pascal `deux.p`, *i.e.* :

```
ocaml deux.mlx
```

Compilation des fonctions

Écrire la compilation des fonctions.

(On pourra reprendre la compilation du corps d'un programme pour le traiter comme celui d'une procédure sans arguments et sans variables locales).

On pourra tester avec les programmes `fact.mlx` et `fib.mlx` (voir le code source et les entrée à fournir dans le cours précédent

Slide 47

Compilation des tableaux

On pourra enfin tester avec les programmes "tri.mlx" "quick.mlx" et "bignum.mlx".

Solution : trans.ml

Note : Les fichiers d'interfaces `*.mli` utilisés par `trans.ml` comportent maintenant une spécification plus détaillée des opérations implémentées.

Interprète du code intermédiaire

Le simulateur fourni est un interprète le code intermédiaire, qui fonctionne avant ou après la linéarisation du code.

Cet interprète est donné comme un module `simul.ml` d'interface `simul.mli`, compatible avec les conventions de `frame.mli` :

Slide 48

- L'appel d'une fonction ou d'une procédure, ainsi que du programme principal s'effectue en sautant à l'étiquette du frame correspondant.
- Le retour des même appels est détecté par un saut au prologue du frame.
- Les sauts aux adresses des primitives déclarées dans `frame.mli` sont reconnues les primitives correspondantes sont exécutées.
- Les globaux commencent à l'adresse `global_space`.

Une différence importante par rapport à une machine réelle, est que l'interprète sauvegarde tous les temporaires avant un appel les restaure au retour.

Canonisation du code

Pour ceux qui le souhaite, vous pouvez implémenter la canonisation et la linéarisation du code obtenu par la première étape de la compilation.

Interpafe imposée canon.mli

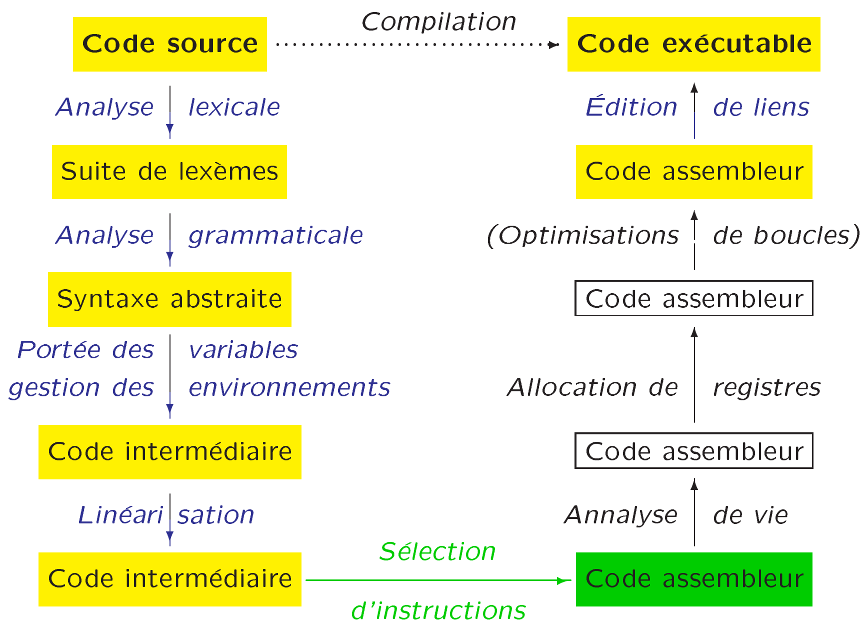
Slide 49

Solution canon.ml

(Utilise de façon optionnel un module scc.ml d'interface scc.mli qui énumère les composantes connextes dans un ordre de non dépendance).

Sélection d'instructions
Projets de compilation.

Slide 1



Slide 2

Slide 3

Une étape dans la chaîne de compilation

$$\text{SA} \overbrace{\xrightarrow{\text{Générat. Canon.}}}^{(1)} \text{CI} \overbrace{\boxed{\xrightarrow{\text{Sélection d'instruct.}}} \text{CA} \underbrace{\xrightarrow{\text{Durée de vie}} \text{CA} \xrightarrow{\text{Coloriage}}}_{\text{Allocation de registres}}}^{(2)} \text{CA}$$

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

Slide 4

But

Traduire le code intermédiaire en code assembleur.

- Générer 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

Slide 5

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 (à deux arguments) :

```
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.

Slide 6

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.

Note : ℓ est nulle pour un saut extérieur à la procédure compilée.

Slide 7

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).

Slide 8

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[a]. 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) :

Slide 9

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

Souvent, plusieurs choix sont possibles.

[a]Par analogie, ce n'est pas l'alphabet mais le nombre de mots (combinaisons de lettres bien formées) qui fait la richesse d'une langue.

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) :

$$\overbrace{\boxed{\begin{array}{l}\texttt{addu}\ t', t_1, 3\\ \texttt{lw}\ t', (t')\end{array}} \qquad\qquad \boxed{\texttt{lw}\ t, 3(t_1)}}^{\texttt{Mem(Bin(Plus,Temp}\ t_1\texttt{,Const}\ 3\texttt{))}}$$

Slide 10

Autres motifs similaires :

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

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

Plus un motif par défaut $\texttt{Mem}\ e$ considérer en dernier.

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...

Slide 11

$$\overbrace{\boxed{\texttt{mul}\ \texttt{t}_1, \texttt{t}_2, 4} \qquad\qquad \boxed{\texttt{sll}\ \texttt{t}_1, \texttt{t}_2, 2}}^{\texttt{Move\_Temp}(t_1\texttt{,Bin(Times,Temp}\ t_2\texttt{,Const}\ 4\texttt{))}}$$

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 :

Slide 12

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

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 :

Slide 13

```
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

Slide 14

```
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

Slide 15

```
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 = Gen.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.

Slide 16

```
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;;
```

On leur donne éventuellement des noms :

```
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), ...
```

Slide 17

Les registres conventionels du MIPS

Slide 18

```
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 seront déclarés survivre 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

Slide 19

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

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

(a) ne logeant pas dans des registres passés à une fonction appelée depuis f (ici 3)

Appel de fonction ou de 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.

Slide 20

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 = Gen.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).

Slide 21

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 :

Slide 22

```
  | 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.

Slide 23

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

Slide 24

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

Slide 25

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.

Slide 26

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*).*

Slide 27

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 :

Slide 28

```
    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 []))
```

Slide 29

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.

Slide 30

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

Slide 31

```
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...

Slide 32

```
__start :                main:                        fact :
  la    $fp, Mem           subu  $sp, $sp, main_f      subu  $sp, $sp,fact_f
  la    $gp, Glob          move  $116, $s0             move  $110, $s0
  jal   main               move  $117, $ra             move  $111, $ra
  li    $v0 10             jal   read_int              move  $105, $a0
  syscall                  move  $108, $v0             li    $112, 1
                           sw    $108, 0($gp)          bgt   $105, $112, L9
                           lw    $118, 0($gp)        L8:
L9:                        move  $a0, $118             li    $113, 1
  move $107, $105          jal   fact                  move  $104, $113
  sub  $114, $105, 1       move  $109, $v0           L10:
  move $a0, $114           move  $a0, $109             move  $v0, $104
  jal  fact                jal   println_int           move  $s0, $110
  move $106, $v0           move  $s0, $116             move  $ra, $111
  mul  $115, $107, $106    move  $ra, $117             addu  $sp, $sp,fact_f
  move $104, $115          addu  $sp, $sp, main_f      j     $ra
  j    L10                 j     $ra
```

(avec un seul registre callee-saved s0)

Exercices

Slide 33

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.

Slide 34

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
```

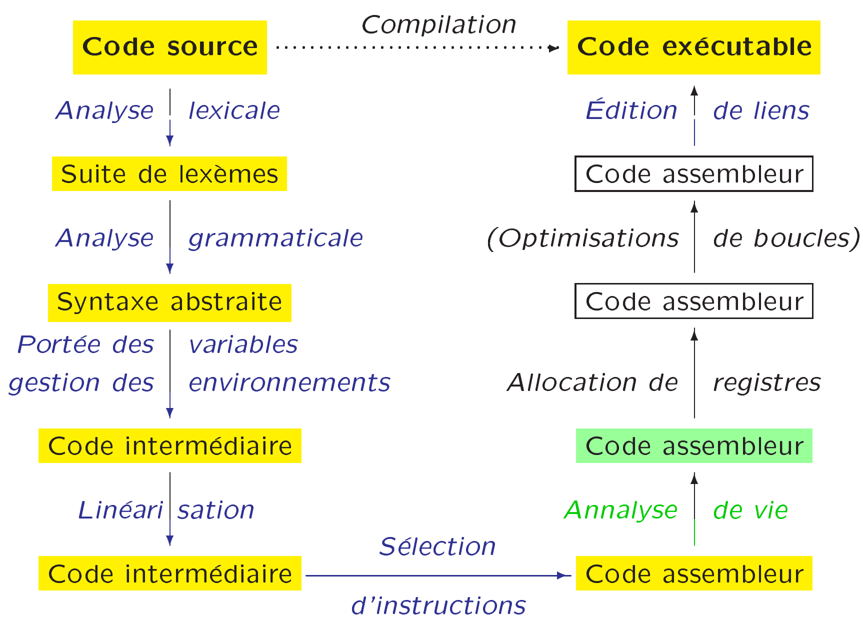
(le prélude n'est pas affiché) et comparer le code que vous générez.

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

Slide 35

Analyse de durée de vie.

Slide 1



Sa place dans la chaîne de compilation

$$\mathrm{SA}\ \overbrace{\xrightarrow{\text{Générat. Canon.}}}^{(1)}\ \mathrm{CI}\ \overbrace{\xrightarrow{\text{Sélection d'instruct.}}\ \mathrm{CA}\ \underbrace{\boxed{\xrightarrow{\text{Durée de vie}}}\ \mathrm{CA}\ \boxed{\xrightarrow{\text{Coloriage}}}}_{\text{Allocation de registres}}}^{(2)}\ \mathrm{CA}$$

Slide 2

Calculer pour chaque instruction la liste des temporaires vivants à la fin de l'instruction :

⇒ Les temporaires vivant simultanément ne peuvent pas être identifiés. En particulier, ils ne peuvent pas utiliser le même registre.

⇒ Les autres peuvent partager le même registre.

L'analyse de la durée se retrouve sous d'autres formes (analyse de flow) en aval pour faire des optimisations plus poussées.

Les analyses de flow sont encore aujourd'hui, un domaine en pleine effervescence et utilisées aussi bien pour des analyses de programme, dans le but d'une meilleure compilation, ou de plus en plus pour des analyse de sécurité.

Slide 3

Le problème

Slide 4

Chaque instruction
– **utilise** (lit) un ensemble de temporaires, calcule et
– **définit** (écrit) un ensemble de temporaires.
Les deux ensembles ne sont pas nécessairement disjoints.

Un temporaire est **vivant** à la sortie d'une instruction i si sa valeur est utilisée dans une instruction suivante (au sens large) avant d'être redéfinie. La **durée de vie** d'un temporaire est l'ensemble des instructions où il est vivant.

Lorsqu'un temporaire est mort, sa valeur n'a pas d'importance. Deux temporaires qui ont des durées de vie disjointes peuvent être superposés et utiliser le même registre.

Le calcul de la durée de vie est un préliminaire à l'allocation de registres.

Exemples

Slide 5

$\boxed{z \leftarrow x + y}$ utilise les variables x, y et définit z.

Analogie temporaire $\longleftrightarrow$ variable globale

Groupements On peut traiter un groupe d'instructions linéaires (sans saut ni entrée), en particulier un bloc élémentaire, comme une "macro-instuction" :

Bloc		
$z \leftarrow x + y$ $t \leftarrow z$	utilise x, y définit z, t	**Important** : la valeur locale de z est utilisée dans le bloc, mais pas sa valeur à l'entrée du bloc ! Donc z n'est pas lu par le bloc.

L'instruction call se comporte comme une instruction qui

– lit $a0$, $a1$, etc. (selon le nombre d'arguments).
– écrit ra, les registres spéciaux, les registres t, v, et a.

Le graphe d'un programme

Slide 6

Un programme peut être vu comme un graphe dont les nœuds sont les instructions et les arcs décrivent la possibilité de passer d'une instruction à une autre et sont étiquetés par des conditions (mutuellement exclusives —déterminées à l'évaluation).

L'exécution du programme évalue une instruction puis passe à l'instruction suivante autorisée (satisfaisant la condition de saut).

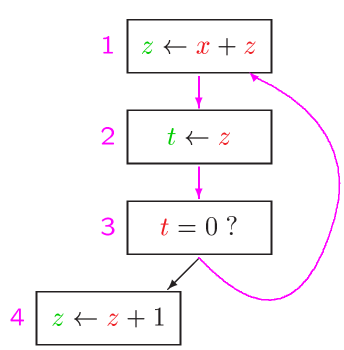
Les conditions de branchement sont des conditions dynamiques. Elles peuvent dépendent d'un calcul arbitrairement complexe. Elles ne sont donc pas décidables.

Approximation On ne peut calculer qu'une approximation de la durée de vie. (On la choisira par excès pour être correct.)

On majore grossièrement les conditions de saut en les supposant toujours vraies (elles ne sont plus disjointes).

Un (sous) programme

Slide 7



	Def	Use	Succ	live in	live out
1	z	x, z	2	x, z	x, z
2	t	z	3	x, z	x, z, t
3	.	t	4, 1	x, z, t	x, z
4	z	z	.	z	.

Définition de la durée de vie

Pour chaque instruction i, on définit
- $\texttt{Use}(i)$ l'ensemble des temporaires utilisés par i
- $\texttt{Def}(i)$ l'ensemble des temporaires définis par i
- $\texttt{Out}(i)$ l'ensemble des temporaires vivants à la sortie de i
- $\texttt{Succ}(i)$ l'ensemble des successeurs immédiats de i

Slide 8

Par définition, $t \in \texttt{Out}(i)$ si et seulement si il existe un chemin à partir de i pour lequel t sera lu avant d'être écrit :

$$\exists i_1 \in \texttt{Succ}(i), \ldots i_{n+1} \in \texttt{Succ}(i_n), \bigwedge \begin{cases} t \in \texttt{Use}(i_{n+1}) \\ \forall k \in [1, n], t \notin \texttt{Def}(i_k) \end{cases}$$

Calcul de la durée de vie

On définit
- $\texttt{Succ}^k(i)$ comme l'ensemble des séquences de k-instructions consécutives à i.
- $\overleftarrow{p}$ la séquence p privée de la dernière instruction,
- $\hat{p}$ la dernière instruction de p.

Slide 9

On peut réécrire $\texttt{Out}(i)$ comme

$$\bigcup_{n \in \mathbb{N}^*} \bigcup_{p \in \texttt{Succ}^n(i)} \bigwedge \begin{cases} t \in \texttt{Use}(\hat{p}) \\ \forall j \in \overleftarrow{p}, t \notin \texttt{Def}(j) \end{cases}$$

(En fait on peut ajouter le cas $n = 0$ en considérant que $\texttt{Succ}^0(i)$ est vide.)

Temporaires vivant en entrée

$$\mathtt{Out}^{\,n}(i) \triangleq \bigcup_{k=1}^{n} \bigcup_{p\in \mathtt{Succ}^{\,k}(i)} \bigwedge \begin{cases} t \in \mathtt{Use}\,(\hat{p}) \\ \forall j \in \overleftarrow{p}, t \notin \mathtt{Def}\,(j) \end{cases}$$

peut s'écrire

Slide 10

$$\bigcup_{i' \in \mathtt{Succ}\ i} \underbrace{\left(\bigcup_{k=0}^{n-1} \bigcup_{p\in \mathtt{Succ}^{\,k}(i')} \bigwedge \begin{cases} t \in \mathtt{Use}\,(\hat{p}) \\ \forall j \in \overleftarrow{i'p}, t \notin \mathtt{Def}\,(j) \end{cases} \right)}_{\triangleq\ \mathtt{In}^{\,n-1}(i')}$$

Soit

$$\mathtt{In}^{\,n}(i) = \underbrace{\mathtt{Use}\,(i)}_{k=0} \cup \underbrace{\bigcup_{k=1}^{n} \bigcup_{p\in \mathtt{Succ}^{\,k}(i)} \bigwedge \begin{cases} t \in \mathtt{Use}\,(\hat{p}) \\ \forall j \in \overleftarrow{p}, t \notin \mathtt{Def}\,(j) \\ t \notin \mathtt{Def}\,(i) \end{cases}}_{\mathtt{Out}^{\,n}(i) \setminus \mathtt{Def}\,(i)}$$

Algorithme

On a $\mathtt{Out}\,(i) = \bigcup_{n\in I\!N} \mathtt{Out}^{\,n}(i)$ où

$$\begin{cases} \mathtt{Out}^{\,n}(i) = \displaystyle\bigcup_{i' \in \mathtt{Succ}\,(i)} \mathtt{In}^{\,n-1}(i) \\ \mathtt{In}^{\,n}(i) = \mathtt{Use}\,(i) \cup (\mathtt{Out}^{\,n}(i) \setminus \mathtt{Def}\,(i)) \end{cases} \qquad \begin{cases} \mathtt{In}^{\,0}(i) = \emptyset \\ \mathtt{Out}^{\,0}(i) = \emptyset \end{cases}$$

Slide 11

Les suites $\mathtt{In}^{\,k}$ et $\mathtt{Out}^{\,k}$ sont croissantes et bornées (par l'ensemble de tous les temporaires), donc elles convergent. Ainsi, elles sont constantes à partir d'un certain rang.

Algorithme On calcule les fonctions $(\mathtt{Out}^{\,n}, \mathtt{In}^{\,n})$ pour n croissant tant que $\mathtt{Out}^{\,n}$ est différent de $\mathtt{Out}^{\,n-1}$ (*i.e.* tant qu'il existe i tel que $\mathtt{Out}^{\,n}(i)$ est différent de $\mathtt{Out}^{\,n-1}(i)$)

Complexité en $O(n^4)$: au plus n^2 itérations sur $O(n)$ instructions coûtant $O(n)$ par instruction.

Équations au point fixe

Lorsque le point fixe est atteint :

$$\begin{cases} \texttt{Out}\,(i) = \displaystyle\bigcup_{i' \in \texttt{Succ}\ i} \texttt{In}\,(i') \\ \texttt{In}\,(i) = \texttt{Use}\,(i) \cup (\texttt{Out}\,(i) \setminus \texttt{Def}\,(i)) \end{cases}$$

Slide 12

Résultat : *Un temporaire est vivant à l'entrée de i s'il est lu par i ou s'il est vivant en sortie de i et n'est pas écrit par i.*

L'algorithme précédent calcule le plus petit point fixe de ces équations.

Il existe aussi un plus grand point fixe, que l'on peut calculer en partant des ensembles pleins. Plus grand et plus petit point fixes ne sont pas égaux (considérer un temporaire jamais utilisé).

Le plus petit point fixe est bien celui que nous cherchions...

Accélération de la convergence

On peut remplacer $\texttt{In}^{\,n-1}$ par $\texttt{In}^{\,n}$ dans le calcul $\texttt{Out}^{\,n}$, lorsque que $\texttt{In}^{\,n}$ est déjà connu.

On a donc intérêt à calculer l'instruction $\texttt{Succ}\ i$ avant l'instruction i.

Un tri topologique permettrait de traiter les composantes connexes les plus profondes en premier.

Slide 13

Comme la plupart des instructions sont simplement des sauts à l'instruction suivante, i.e. $\texttt{Succ}\,(i) = \{i+1\}$, on obtient un bon comportement par un parcours du code en sens inverse.

En pratique, quelques itérations suffisent, et on obtient un comportement entre linéaire et quadratique en la taille du code.

On peut représenter les ensembles de temporaires par des listes ordonnées (union et différence en temps linéaire) ou par des vecteurs de bits.

Calcul sur les blocs de base

Slide 14

On peut faire une analyse en deux étapes :

1. On calcule $\texttt{Def}\,(b)$ et $\texttt{Use}\,(b)$ pour les blocs de base : Un bloc de base se comporte comme un “grosse” instruction pour laquelle
$$\begin{cases} \texttt{Def}\,(b) = \bigcup_{i \in b} \texttt{Def}\,(i) \\ \texttt{Use}\,(b) = \bigcup_{i \in b} \left(\texttt{Use}\,(i) \setminus \bigcup_{i' \in \texttt{Pred}^*(i) \cap b} \texttt{Def}\,(i')\right) \end{cases}$$
Puis les variables vivantes $\texttt{Out}\,(b)$ à la sortie du bloc b comme précédemment.
2. On retrouve alors les $\texttt{Out}\,(i)$ pour les instructions du bloc b par une simple passe linéaire en sens inverse sur le bloc.

La convergence est aussi rapide (même nombre d'itérations) à condition de faire un parcours arrière dans les deux cas, mais à chaque fois on considère un plus petit nombre de nœuds (donc d'opérations).

Calcul sur l'exemple

Slide 15

Données				Calcul normal						-accéléré	
i	Def	Use	*Succ*	in_0 / out_0	in_1 / out_1	in_2 / out_2	in_3 / out_3	in_4 / out_4	in_5 / out_5	in'_0 / out'_0	in'_1 / out'_1
1	z	x,z	2	x,z / .	x,z / z	x,z / z	x,z / z	x,z / z	x,z / x,z	x,z / z	x,z / x,z
2	t	z	3	z / .	z / t	z / t	z / x,z,t	x,z / x,z,t	x,z / x,z,t	z / z,t	x,z / x,z,t
3	.	t	$4,1$	t / .	t / x,z	x,z,t / x,z	x,z,t / x,z	x,z,t / x,z	x,z,t / x,z	z,t / z	x,z,t / x,z
4	z	z	.	z / .	z / .	z / .	z / .	z / .	z / .	z / .	z / .

$$\texttt{Out}\,(i) = \bigcup_{i' \in \texttt{Succ}\ i} \texttt{In}\,(i') \qquad et \qquad \texttt{In}\,(i) = \texttt{Use}\,(i) \cup (\texttt{Out}\,(i) \setminus \texttt{Def}\,(i))$$

Mise en œuvre

Un nœud est une instruction annotée par les informations `Def`, `Use` et `Succ` ainsi que les champs mutables `In` et `Out`.

Slide 16

```
type flowinfo = {
    instr : Ass.instr ;
    def : temp set; use : temp set;
    mutable live_in : temp set; mutable live_out : temp set;
    mutable succ : flowinfo list ;
  }
```

On construit le graphe :

```
flowgraph : code -> flowinfo list ;;
```

On itère le calcul jusqu'à ce que plus rien n'ai changé :

```
fixpoint : flowinfo list -> flowinfo list ;;
```

(On peut facilement afficher la liste des temporaires vivants dans le code, en commentaire de chaque instruction.)

L'exemple du cours

Slide 17

```
  [ Label ("L1", l1);
    Oper ("add ^d0, ^s0, ^s1", [ x; z], [ z], None);
    Move ("move ^d0, ^s0", z, t);
    Oper ("beq ^s0, $zero", [ t], [], Some [l1; l4]);
    Label ("L4", l4);
    Oper ("add ^d0, ^s0, 1", [ z], [ z], None);
    Oper ("jal $ra", [], [], Some []); ] ;;
```

Résultat :

```
                          # Def <= Use      # Out
L1:
    add  $z, $x, $z       # z <= z x        # z x
    move $t, $z           # t <= z          # z x t
    beq  $t, $zero, L1    # <= t            # z x
L4:
    add  $z, $z, 1        # z <= z          # .
    jump Exit
```

Slide 18

```
fact:
    sub  $sp, $sp, fact_f       # sp <= sp             # a0 s0 ra
    move $112, $ra              # 112 <= ra            # a0 s0 112
    move $113, $s0              # 113 <= s0            # a0 112 113
    move $108, $a0              # 108 <= a0            # 108 112 113
    li   $114, 1                # 114 <=               # 108 112 113 114
    bgt  $108, $114, L9         # <= 108 114           # 108 112 113
    li   $115, 1                # 115 <=               # 112 113 115
    move $107, $115             # 107 <= 115           # 107 112 113
L10:
    move $v0, $107              # v0 <= 107            # v0 112 113
    move $s0, $113              # s0 <= 113            # v0 s0 112
    move $ra, $112              # ra <= 112            # v0 s0 ra
    add  $sp, $sp, fact_f       # sp <= sp             # v0 s0 ra
    j    $ra                    # <= v0 s0 ra          #
L9:
    sub  $116, $108, 1          # 116 <= 108           # 108 112 113 116
    move $a0, $116              # a0 <= 116            # a0 108 112 113
    jal  fact                   # v0 a0 ra <= a0       # v0 108 112 113
    move $109, $v0              # 109 <= v0            # 108 109 112 113
    mul  $117, $108, $109       # 117 <= 108 109       # 112 113 117
    move $107, $117             # 107 <= 117           # 107 112 113
    j    L10                    # <=                   # 107 112 113
```

Importance des annotations Use et Def

Slide 19

Noter l'effet des Use et Def dans
– un appel de primitive,
– un appel de fonction/procédure
– au retour d'un programme.

En particulier, en rendant les registres spéciaux vivants à la fin d'une procédure, ils resteront vivants pendant toute la procédure puisqu'ils ne sont jamais écrits.

(Pour simplifier, les registres spéciaux n'ont pas été affichés dans l'exemple ci-dessus.)

Corriger, si nécessaire, ces annotations dans la génération de code

Autres analyses de flux

L'analyse de durée de vie est une analyse arrière : on a intérêt à aller en arrière (dans le sens du calcul) pour accélérer la convergence, parce que l'inforation se propage vers l'arrière.

La plupart des analyses de flux sont à l'inverse des analyses avant.

Slide 20

Analyse d'atteignabilité

On veut calculer l'ensemble des *définitions* qui sont vivantes à chaque point du programme (instruction ou bloc).

Application à la propagation de constantes

En amont (sur le code source) ou en aval (sur le code machine) :

Si l'affectation d'une constante c à une variable globale (amont) ou à un temporaire t (aval) atteint un point p, alors on sait qu'à ce point on peut remplacer t par la constante c.

Analyse avant

L'information est en général déterminée comme le plus petit point fixe d'une équation de la forme :

$$\begin{cases} \texttt{In}\,(i) = \bigcup_{i' \in \texttt{Pred}\ i} \texttt{Out}\,(i') \\ \texttt{Out}\,(i) = \texttt{Gen}\,(i) \cup (\texttt{In}\,(i) \setminus \texttt{Kill}\,(i)) \end{cases}$$

Slide 21

Les fonctions `Gen` et `Kill` sont analogues à `Use` et `Def` :
- $\texttt{Gen}\,(i)$: l'ensemble des défintions générées par l'instruction i
- $\texttt{Kill}\,(i)$: l'ensemble des définitions détruites par i

Noter aussi l'inversion des ensembles $\texttt{In}\,(i)$ et $\texttt{Out}\,(i)$ dans les équations.

Graphe d'interférence

Les nœuds sont les temporaires et les arcs non orientés relient deux temporaires qui ne peuvent pas être superposés.

Cas général : Lorsqu'une instruction écrit dans un temporaire t, il n'est pas possible d'identifier t avec un autre temporaire qui survit après l'instruction.

Slide 22 Formellememt, on ajoute, pour chaque instruction i les arcs $\mathtt{Def}\,(i) \times (\mathtt{Out}\,(i) \setminus \mathtt{Def}\,(i))$.

Cas particulier des instructions Move : Comme à la sortie, la source et la destination ont le même contenu, il n'y a pas jamais interférence entre la source et la destination, même si la source survit à l'instruction. Au contraire, il est possible (et souhaitable) de pouvoir les identifier.

Formellement, on ajoute, pour chaque instruction move les arcs $\mathtt{Def}\,(i) \times ((\mathtt{Out}\,(i) \setminus \mathtt{Def}\,(i)) \setminus \mathtt{Use}\,(i))$.

Élimination des instructions Move

Ces instructions ne calculent pas, et pourront être éliminées, si les temporaires source et destination sont superposés, ce que l'on cherche à faire. Donc, ces deux temporaires n'interfèrent pas ! au contraire ils *s'attirent*. On les représente dans le graphe d'interférence par des arcs d'attraction.

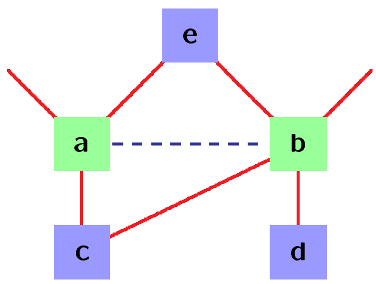
Slide 23 Formellement, on ajoute pour chaque instruction move d s un arc d'attraction entre s et d.

Exemple

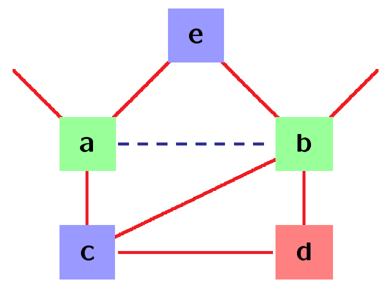
Les arcs d'interférence sont en rouge.

Les arcs qui s'attirent (liés par une instruction move) en blue pointillés). Ce seront des candidats privilégiés pour la superposition de nœuds.

Slide 24



2 couleurs



3 couleurs

Mise en œuvre

Un nœud du graphe est un temporaire annoté par la liste des nœuds adjacents.

Slide 25

```
type interference =
  { temp : temp;
    mutable color : temp option;
    mutable adj : interference list ; }
type move =
  { move : Ass.instr ;
    left : interference ; right : interference ; }
```

Le graphe d'interférence est créé en parcourant le code annoté par l'ensemble des registres vivants après chaque instruction.

```
interference :
      temp list (* précoloriés *) ->
      flowinfo list -> interference list * move list
```

Allocation de registres triviale

Slide 26

Le graphe d'interférence permet d'effectuer l'allocation de registres. Nous proposons ici une méthode triviale (une méthode plus performante sera présentée dans le cours suivant).

Les temporaires représentant des registres machines sont pré-coloriés, tous de couleur différente (on pourrait identifier les couleurs avec les registres.) Le but est de colorier les temporaires de telle façon que deux temporaires qui interfèrent aient deux couleurs différentes.

1. On choisit un temporaire t non colorié au hasard.
2. Les couleurs interdites sont l'ensemble des couleurs des temporaires déjà coloriés qui interfèrent avec t.
3. On choisit une couleur non interdite au hasard.
4. En cas d'échec, on choisit de placer t en pile.

Allocation en pile

Slide 27

Lorsqu'un temporaire t doit être alloué en pile, il faut
- lui attribuer un emplacement en pile (position dans le bloc d'activation) à une distance n du sommet de pile.
- pour chaque utilisation de t
  - charger la valeur à la position n du sommet de pile dans un nouveau temporaire t' (ajouter une instruction load)
  - remplacer le temporaire t par t' dans l'instruction.
- pour chaque définition de t
  - remplacer le temporaire t par un nouveau temporaire t' dans l'instruction.
  - sauver la valeur de t' à la position n du sommet pile. (ajouter une instruction store)

Note d'implémentation : pour allouer un emplacement en pile, on utilise module frame (voir frame.mli).

Exemple d'allocation en pile de `s0` et `ra`

Slide 28

```
fact:
    sub   $sp,  $sp, fact_f
    move $117, $s0              move $141, $s0
                                sw    $141, 0($sp)
    move $125, $ra              move $142, $ra
                                sw    $142, 4($sp)
    move $112, $a0
    li    $126, 1
    bgt   $112, $126, L10
    li    $127, 1
    move $111, $127
L11:
    move $v0, $111
                                lw    $143, 0($sp)
    move $s0, $117              move $s0,  $143
                                lw    $144, 4($sp)
    move $ra, $125              move $ra,  $144
    add   $sp, $sp, fact_f
    j     $ra
```

Attention...

Slide 29

Lors de la réécriture du code, si une instruction lit deux fois le même registre, il faut recharger la valeur une seule fois :

```
                                lw    $201, 8($sp)
    add   $100, $200, $200      add  $100, $201, $201
```

```
 et pas                              lw    $201, 8($sp)
                                lw    $202, 8($sp)
    add   $100, $200, $200      add  $100, $201, $202
```

de façon à conserver le partage existant.

Par contre, on n'essayera pas d'optimiser entre deux instructions.

On pourra donc retrouver dans certains cas

move \$200, \$100	**move** \$201, \$100
	sw \$201, 8(\$sp)
	lw \$202, 8(\$sp)
add \$100, \$200, \$200	**add** \$100, \$201, \$202

Slide 30

Élimination triviale des moves inutiles

Après coup, on élimine les moves inutiles.

En fait, une amélioration triviale consiste à choisir parmi les couleurs possibles d'un temporaire t la couleur d'un temporaire t' attiré par t (directement ou indirectement).

Cela permet d'éliminer de nombreux moves. Une approche plus systématique sera vue dans le chapitre suivant.

Slide 31

Allocation de registres minimale

Au minimum, il s'agit de produire du code qui tourne... Si on sauve systématiquement les registres $s0, s1, ...$ en pile dans chaque procédure, on peut éviter le spilling dans les petits exemples. On peut donc se contenter d'un coloriage le plus simple possible.

Amélioration facile

Si le coloriage tire une couleur au hasard parmi les couleurs possibles, on trouvera peu de move inutiles.

Coloriage biaisé : on choisit, parmi les couleurs possibles, par ordre de préférence :

Slide 32

1. une couleur désirable :

   celle d'un temporaire déjà colorié relié par un move;
2. qui n'est pas déconseillée :

   celle qui est désirable pour un temporaire relié par un move;

Cela permet d'éliminer de nombreux moves, malgré une allocation des couleurs triviale (donc malgré un nombre de placements en pile relativement important).

Exercices

Ce td est très indépendant des précédents

Implémenter l'analyse de durée de vie

Slide 33

On fournit (dans `~remy/compil/td5/`)
– une librairie (smallset.ml : smallset.mli) pour calculer sur les ensembles de registres (représentés par des listes ordonnées).
– un squelette (dépouillé) liveness.ml et son interface liveness.mli.

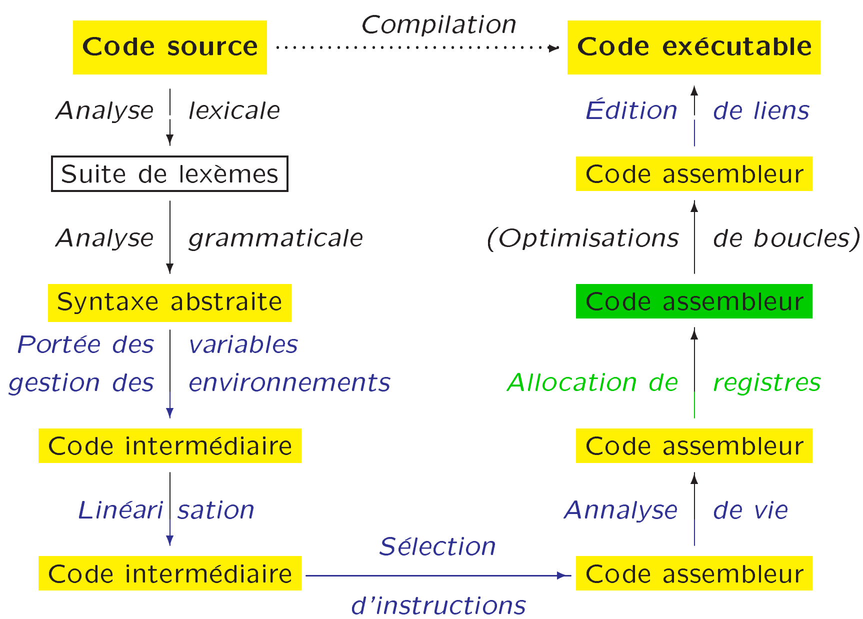
Pour la mise au point, Afficher la dureé de vie en commentaire des instructions. *(On pourra utiliser l'exemple du cours, lui ajouter un appel de fonction ou de primitive. Pour de gros exemples, insérer le module liveness à la fin de la chaîne de compilation.)*

Voir `run.ml` et fact.mlx

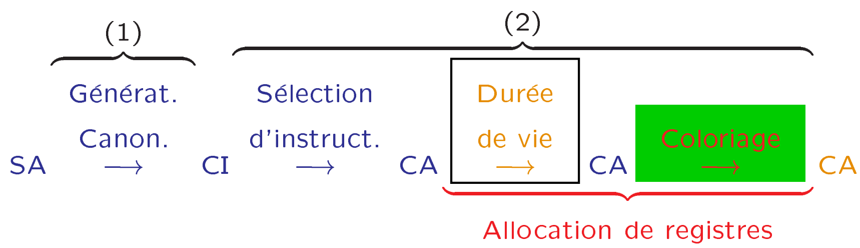
Construire le graphe d'interférence.

Allocation de registres.

Slide 1



Sa place dans la chaîne de compilation



Slide 2

On veut allouer les registres au mieux afin
– d'effectuer le minimum de sauvegardes en pile et
– d'éliminer le maximum d'instructions MOVE.
C'est un des maillons les plus connus de la chaîne de compilation.

Enjeux

Slide 3

L'allocation triviale semble avoir déjà des performances raisonnables. Pourquoi aller plus loin ?

1. La pression en registres, faible sur des petits exemples, peut devenir importante sur des exemples de taille réaliste, et en particulier si le compilateur expanse en ligne des appels à de petites procédures.
2. La pression en registres augmente aussi considérablement sur des architectures qui ont peu de registres.
3. Enfin, les performances d'une machine RISC sont très bonnes parce qu'il y a beaucoup de registres et la plupart des calculs intermédiaires peuvent être faits dans des registres, mais elles se dégradent rapidement si des sauvegardes superflues doivent être faites en pile.

Une allocation de très bonne qualité n'est donc pas du superflue.

Rappel de l'allocation triviale

Slide 4

Allocation

1. On choisit un temporaire t non colorié au hasard.
2. Les couleurs interdites sont l'ensemble des couleurs des temporaires déjà coloriés qui interfèrent avec t.
3. On choisit au hasard une couleur non interdite.
4. En cas d'échec, on décide de placer t en pile.

Élimination des MOVE

On choisit par préférence la couleur d'un temporaire déjà colorié en relation avec t (par une ou plusieurs instructions MOVE) ... tout en évitant la couleur préférée des temporaires qui interfèrent avec t qui ne sont pas encore coloriés.

Critiques de l'allocation triviale

Slide 5

Superposition des temporaires

Les temporaires sont traités dans un ordre arbitraire, sans tenir compte de la topologie.

Certains nœuds peu contraints sont servis en premier et reçoivent une couleur arbitraire, alors qu'il aurait été plus judicieux de les servir plus tard et leur attribuer une couleur restant disponible.

Élimination des MOVE

L'élimination des MOVE est traitée de façon ad-hoc, en aval, et sans collaborer avec l'allocation.

Un temporaire impliqué dans un MOVE devrait être privilégié pour lui donner plus de chance de choisir sa couleur égale à celle d'un temporaire avec lequel il est en relation.

Allocation fine

Principe

- Allocation des registres par un algorithme de coloriage de graphe,
- Prise en compte simultanée des instructions MOVE par une variation sur l'algorithme de coloriage.

Slide 6

Heuristiques

La méthode est générale, mais approchée.

Les choix sont guidés par des heuristiques *modulaires* plutôt que par des solutions *ad-hoc*.

Coloriage de graphe

Nous nous intéressons d'abord à l'algorithme de coloriage sans nous soucier des instructions MOVE.

Problème Étant donné un graphe et un ensemble de K couleurs, il s'agit d'attribuer une couleur à chaque nœud du graphe de telle façon qu'un arc relie toujours des nœuds de couleurs différentes.

Slide 7

Complexité Le problème est NP-complet.

Histoire Le problème de l'allocation de registres a été résolu par un mathématicien Russe au 19e siècle par coloriage de graphe (en fait, il a résolu le problème analogue de la minimisation du nombre de variables globales dans un programme).

Une solution approchée du coloriage

Principe

Si un nœud n est de degré (*i.e.* nombre de voisins) plus petit que K et si le graphe $G - \{n\}$ est K-coloriable, alors le graphe G est K-coloriable. En effet, une fois $G - \{n\}$ K-colorié il reste au moins une couleur qui ne soit pas celle d'un voisin de n.

Slide 8

Procédure récursive

1. Retirer les nœuds de faible degré (plus petit que K). Cela diminue le degré des nœuds restant et permet de continuer au mieux jusqu'à ce que le graphe soit vide.
2. Dans ce cas le graphe est coloriable, et on est certain de pouvoir attribuer correctement les couleurs au retour de la procédure récursive.
3. Sinon, le graphe peut ne pas être coloriable.

Une stratégie optimiste

À l'aller Lorsque tous les nœuds sont de fort degré le graphe peut ne pas être coloriable. On retire un nœud qui sera éventuellement placer en pile, et on poursuit comme précedemment.

Slide 9

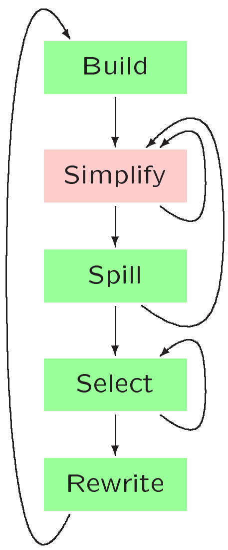
Au retour, on essaye de colorier ce nœud : en effet la solution précédente étant approchée, il se peut que le graphe soit malgré tout coloriable.

Si ce n'est pas possible, alors on décide de placer définitivement le nœud en pile, sans lui attribuer de couleur, et on poursuit.

À la fin S'il y a eu des placements en pile, il faut réécrire le code, et recommencer : en effet, le placement en pile introduit de nouveaux temporaires...

Procédure itérative

Slide 10



En empilant les nœuds retirés, on obtient :

– Construire le graphe d'interférence

– Retirer un nœud de faible degré du graphe et l'empiler

– Retirer un nœud de fort degré du graphe et l'empiler

– Dépiler et colorier un nœud ou décider de le placer en pile

– Réécrire le code

Heuristique

Slide 11

Lorsqu'un nœud de fort degré est placé en pile, le choix de ce nœud parmi les candidats possibles n'est pas anodin. On veut :

– que ce choix ait un effet maximal, débloquant d'autres nœuds : il faut choisir un nœud de degré élevé.
– que la sauvegarde en pile ne soit pas trop coûteuse : choisir un nœud (temporaire) peu utilisé (dont le nombre d'occurrences dans `Def` et `Use` est faible).

Il faut trouver un compromis entre l'efficacité du spill et son coût. On paramètre ce choix par une fonction de priorité p, par exemple, $p(n) = d(n)/u(n)$ où $u(n)$ et $d(n)$ sont le nombre d'utilisations du nœud n et son degré. (Il faudrait pondérer chaque utilisation par une estimation de sa fréquence d'utilisation : une instruction à l'intérieur d'une boucle interne sera exécutée plus souvent qu'une instruction plus externe.)

Itération

Slide 12

Lorsqu'il y a un placement en pile. La réécriture du code dans la phase en aval va introduire de nouveaux temporaires (dit *auxiliaires*).

Les temporaires auxiliaires ont une durée de vie très courte et en général cela ne pose pas de problème. Ils trouveront leur place dans des interstices. Mais s'il y a une forte demande de registres, il peut être nécessaire d'allouer d'autres temporaires en pile pour faire de la place pour les temporaires auxiliaires.

Ainsi, il faut recommencer l'allocation après la réécriture jusqu'à ce qu'il n'y ait plus de spill. En général, une ou deux itérations suffisent.

On aurait pu aussi pré-réserver des registres pour les temporaires auxiliaires, mais c'est en général un gaspillage de registres (une recette ad-hoc).

Terminaison

Slide 13

Le programme peut donc boucler... Mais cela ne peut se produire que si un temporaire auxiliaire est à nouveau placé en pile, ce qui n'a pas de sens (son propre auxiliaire lui sera isomorphe).

Normalement, cela peut être évité par un réglage de la priorité : en effet, si tous les temporaires sauf les auxiliaires sont placés en pile, il suffit de trois registres machine pour compiler du code 3-adresses (`a0`, `v0` et `ra`).

On peut détecter le risque de non terminaison et lever une exception si il n'y a plus d'autres solutions que celle de placer un temporaire auxiliaire en pile. Il s'agit alors d'une erreur de conception ou de réglage : trop peu de registres t ou bien d'une mauvaise fonction de priorité (les temporaires utilisés pour un spill dit de deuxième génération ont un coût de spill le plus élevé : à ne spiller qu'en dernier recours).

Prise en compte des instructions MOVE

Slide 14

On peut internaliser le traitement des instructions MOVE dans la procédure d'allocation. Ce qui permet de mieux choisir l'ordre de coloriage et de favoriser l'élimination des MOVE.

Plutôt qu'un choix passif en aval *en remarquant qu'on peut donner la même couleur à deux temporaires*, il est préférable de prévoir en amont de donner ultérieurement la même couleur à deux temporaires.

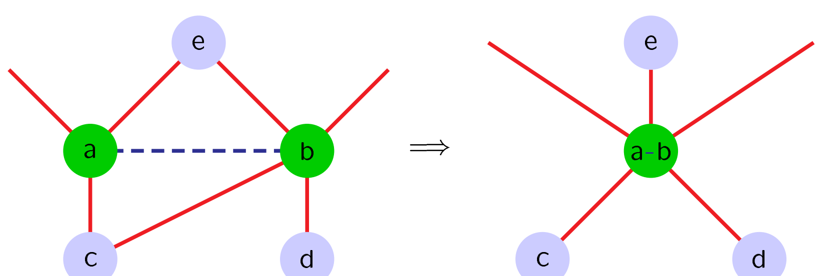
Cela revient à supposer que leurs nœuds seront fusionnés. Il faut alors mettre à jour le graphe, ce qui
- augmente (ou laisse inchangé) le degré des nœuds fusionnés,
- diminue (ou laisse inchangé) le degré des autres nœuds (voisins).

L'allocateur prendra alors automatiquement ces choix en compte pour déterminer un meilleur ordre de coloriage.

Fusion (coalescence) et son effet

Slide 15

On représente les MOVE par des arcs en pointillés.



Le degré des nœuds a et b augmente.

Le degré des nœud c et e diminue.

Le degré du nœud d est inchangé.

Une stratégie sure

Ne fusionner les nœuds (a) et (b) dans le graphe G que si cela préserve sa coloriabilité.

Deux critères sûr

Slide 16

1. Après fusion le nœud (a-b) a moins de K voisins de fort degré.
2. Le nœud (a) est tel que tous ses voisins de fort degré interfèrent avec (b). (*typiquement (b) est un registre*)

Preuve : après avoir éliminé tous les nœuds de faible degré dans le graphe résultant

1. le nœud (a-b) a moins de K voisins et peut aussi être retiré.
2. le nœud (a-b) peut être identifié avec le nœud (b) de G.

Dans les deux cas, il reste un sous-graphe de G, donc coloriable.

Allocation combinée

On représente dans le graphe les MOVE par des arcs en pointillés. Les nœuds qui n'ont pas d'arc MOVE sont dits *simplifiables*, les autres sont dits *complexes*.

La fusion augmente le degré des nœuds fusionnés : on ne peut donc pas simplifier un nœud tant qu'il est complexe.

Slide 17

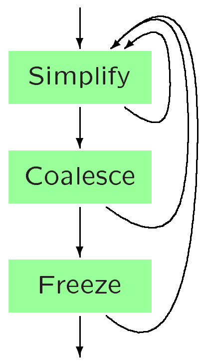
Par priorité :

1. on retire du graphe un nœud simplifiable de faible degré (comme précédemment) ;
2. on effectue une fusion sure (qui préserve la coloriabilité) ;
3. on retire tous les arcs MOVE d'un nœud complexe de faible degré (abandonnant tout espoir de le fusionner) ce qui le rend simplifiable ;
4. on retire un nœud de fort degré qui sera a priori placé en pile (spill).

Allocation combinée, graphiquement

Slide 18

Remplacer la boîte Simplify de l'algorithme précédent par :



- Retirer du graphe un nœud simplifiable de faible degré
- Effectuer une fusion sure
- Retirer un arc MOVE

Au pire, aucune fusion n'est possible : les arcs MOVE sont retirés un à un et on retombe sur (une trace de) l'algorithme précédent.

Mise en œuvre

Slide 19

Au cours du calcul, on maintient des ensembles de nœuds (et d'arcs). Les nœuds sont dans différents états (simple/complexe, faible/fort degré, etc.). Chaque étape revient à choisir un nœud dans un certain état pour le placer dans un autre état (et ajuster le graphe).

Le module `partition` avec l'interface ci-dessous permet de maintenir à jour l'ensemble des nœuds (et d'arcs) dans chaque état de façon efficace (paresseuse).

```
type 'a partition          type 'a elem
val make : int -> 'a partition  array
val create :  'a partition  -> 'a -> 'a elem
val info :  'a elem -> 'a
val belong :  'a elem -> 'a partition  -> bool
val move : 'a elem -> 'a partition  -> unit
val pick :  'a partition  -> 'a elem
```

Représentation des nœuds et des arcs

Slide 20

```
type node_info = {
  temp : temp;                    (* temporaire associé *)
  mutable moves : move list;      (* arcs moves *)
  mutable adj : node list ;       (* arcs d'interference *)
  mutable degree : int;
  mutable alias : node option;    (* noeud fusionné *)
  mutable color : int ;
  mutable occurs : int;           (* nombre d'utilisations *)
}
and move_info =
   { instr : Ass.instr ; left : node; right : node; }
and node = node_info elem
and move = move_info elem;;
```

Les partitions de nœuds

Slide 21

On distingue 9 états pour les nœuds :

- pré-coloriés : ne changeront pas d'état
- initiaux : état temporaire de tous les nœuds non coloriés juste après la construction du graphe d'interférence ;
- simplifiables de faible degré : candidats à la simplification ;
- complexes de faible degré : candidats pour être gelés (i.e. pour geler leurs arcs MOVE) ;
- de fort degré : candidats pour être placés en pile ;
- spilled : définitivement programmés pour être placés en pile ;
- fusionés : leur champ alias contient le nœud résultant ;
- empilés : en attente d'être coloriés (l'ordre est essentiel) ;
- coloriés : état final de tous les nœuds non pré-coloriés.

Les partitions d'arcs MOVE

Slide 22

On distingue 5 états pour les arcs MOVE :

- fusionnés : ne seront plus considérés.
- contraints : ne peuvent être fusionnés car source et destination interfèrent. Ne seront plus considérés.
- gelés : ceux pour lesquels la fusion a été définitivement abandonnée ; ils ont été retirés du graphe, et ne seront plus considérés.
- candidats à la fusion.
- candidats à la fusion temporairement bloqués tant que leur fusion n'est pas sure. Ils seront reconsidérés (candidats à la fusion) dès que le degré d'un de leurs voisins aura diminué.

Implémentation

Slide 23

L'implémentation suit l'algorithme pas à pas : sélectionner un nœud ou un arc à traiter, le traiter, puis mettre le graphe (i.e. l'état des nœuds et des arcs voisins) à jour.

Ré-écriture du code

Dans le cas où il y a des registres à placer en pile, il faut réécrire le code et recommencer.

On peut prendre en compte les fusions qui ont eu lieu avant le premier spill, à l'exclusion de celles vers un registre mis en pile[a]. En effet, celles-ci seront forcément reproductibles.

Par contre, il faut ignorer toutes les fusions qui ont eu lieu après un spill. Pour cela, on sauvegarde l'état des temporaires et des MOVE déjà fusionnés au moment du premier spill.

[a] Une fusion conservatrice préserve mais n'impliquement pas la coloriabilité ; un registre peut donc être fusionné vers un registre qui se retrouve spillé.

Exemple : code de fact

Slide 24

```
    fact_f =8
fact:
    sub  $sp, $sp, fact_f
    sw   $s0, 0($sp)
    sw   $ra, 4($sp)
    li   $t9, 1
    bgt  $a0, $t9, L8
L7:
    li   $v0, 1
L9:
    lw   $s0, 0($sp)
    lw   $ra, 4($sp)
    add  $sp, $sp, fact_f
    j    $ra
L8:
    move $s0, $a0
    sub  $a0, $a0, 1
    jal  fact
    mul  $v0, $s0, $v0
    j    L9
```

Comparer avec le code juste après son émission...

Ajustements

Slide 25

On peut assembler l'ensemble de la chaîne de compilation.

Ajustement de la pile Une fois l'allocation de registres effectuée, la taille de la pile est connue. Il reste à déclarer celle-ci comme une constante en tête du prélude par une instruction de la forme :

```
Oper ("fact_f=8", [], [],    None)
```

(Dans le cas où aucun spill n'est nécessaire, on pourra supprimer les instructions d'ajustement de la pile)

Procédure principale En principe, les temporaires pourraient être placés dans des globaux. Pas soucis de simplicité, on peut revoir la compilation de la procédure principale et la traduire en un appel à une procédure `main`. On place donc le code principal dans un appel de procédure.

Exercices

Slide 26

Allocation de registres triviale

Dans un premier temps, on cherche simplement à générer du code exécutable. On peut choisir la sauvegarde systématique des registres $s0, s1, \ldots$ en pile. Il suffit alors d'implanter le coloriage trivial sans prendre en compte les MOVE.

Spilling

Implémenter le spill d'un registre. On peut maintenant sauver les registres $s0, s1, \ldots$ dans des temporaires et l'allocateur de registre les sauvera en pile seulement si nécessaire.

Élimination des MOVE

Éliminer les MOVE au moment de la sélection des couleurs. Pour espérer éliminer beaucoup de move, il faut dans le coloriage, privilégier les arcs move.

Mini projet

Slide 27

Algorithme de coloriage

Implémenter l'algorithme de coloriage (sans optimiser les MOVE).

Utilisera une heuristique simple pour le choix des registres proposés pour l'allocation en pile.

Prise en compte simultanée des MOVE

L'implantation de l'algorithme complet est un projet. On pourra utiliser un module implémentant des partitions d'ensembles (partition.ml : partition.mli).

Exhiber des exemples montrant les limites d'une allocation de registres trop élémentaire et l'efficacité de l'allocation fine.

Amélioration de la procédure de Spill

Slide 28

L'implémentation du Spill n'est pas forcément efficace, car toutes les occurrences sont spillés et traitées indépendemment, ce qui peut conduire à des mouvements fréquent avec la pile, dont la plupart peuvent être inutiles.

Pour une implémentation du spill plus efficace, voir l'examen de l'an 2000 et son prolongement.

A Examen de l'année 1999 (durée 2h)

On attachera une importance particulière à la clarté, à la précision et la la concision de la rédaction.

1 Expansion en ligne

L'expansion en ligne (*inlining*) consiste à remplacer dans le source des appels à des fonctions par le corps de ces fonctions. On considère la fonction `carré` suivante :

```
function carré (n : integer) : integer;
begin
   carré := n * n
end;
```

1.1

Donner un code SPIM possible pour la fonction `carré` en prenant comme registres conventionnels `a0`, `a1`, `a2`, et `a3` pour les arguments et `v0` pour le résultat.

```
carre:
    mul $v0, $a0, $a0        # v0 <- n n
    j   $ra                  # Retour à l'appelant
```

Réponse :

```
carre:
    mul $v0, $a0, $a0        # v0 <- n n
    j   $ra                  # Retour à l'appelant
```

1.2

On considère la fonction `cube` suivante :

```
function cube (n : integer) : integer;
begin
   cube := carré(n) * n
end;
```

a) Donner un code SPIM possible pour cette fonction (en utilisant les mêmes conventions d'appel), sans faire d'hypothèse sur le code de la fonction `carré` (on indiquera brièvement en commentaires les opérations effectuées par chaque instruction).

Réponse :

```
    cube_f=8
cube:
    subu $sp, $sp, cube_f    # ajustement de la pile
    sw   $ra, 0($sp)         # sauvegarde de l'adresse de retour
    sw   $a0, 4($sp)         # sauvegarde de n
    jal  carre               # v0 <- carré(n)
    lw   $a0, 4($sp)         # a0 <- n
    mul  $v0, $a0, $v0       # v0 <- carré(n) * n
    lw   $ra, 0($sp)         # rappel de l'adresse de retour
    addu $sp, $sp, cube_f    # ajustement de la pile
    j    $ra
```

b) Donner le code source résultant de l'expansion en ligne de la fonction `carré` dans la fonction `cube`.

Réponse :

function cube (n : integer) : integer;
begin
cube := (n ∗ n) ∗ n
end;

c) Donner un code SPIM pour le code source résultant de l'expansion en ligne.

Réponse :

```
cube:
    mul $v0, $a0, $a0        # v0 <- n * n
    mul $v0, $v0, $a0        # v0 <- (n * n) * n
    j   $ra                  # retour à l'appelant
```

1.3

On s'appuyera sur l'exemple ci-dessus.

a) Quel sont les différentes sources de gain de l'expansion en ligne ?

Réponse : *Le gain immédiat est l'économie des appels de fonction, donc du passage des arguments et la récupération du résultat aux emplacements conventionnels, la sauvegarde de l'adresse de retour, et deux sauts (aller et retour).*

Le gain indirect est la connaissance exacte du code du corps de la fonction carré ce qui permet de savoir qu'elle n'utilise pas les registres `ra`, `t0`, `t1`, *etc. et d'éviter ainsi des sauvegardes intermédiaires en pile. (Un meilleur placement des temporaires dans les registres peut également résulter du regoupement des ces deux fonctions et de leur compilation simultanée.)*

b) Expliquer comment une partie des gains de l'expansion en ligne, dits indirects, peuvent se retrouver simplement sans effectuer l'expansion en ligne.

Réponse : *Les gains indirects sont décrits à la question précédente.*

On peut retrouver ces gains si l'on connaît effectivement les registres utilisés par la fonction `carré` *pendant la compilation de la fonction* `cube` *(il suffit de compiler* `carré` *avant* `cube`*). On saura alors, par exemple, que les registres temporaires* `t0, t1, ...` *ne sont pas utilisés par la fonction* `carré` *et peuvent l'être par la fonction* `cube` *pour sauvegarder les valeurs intermédiaires et l'adresse de retour, ce qui évitera tous les mouvements de pile dans la fonction* `cube`.

Plus généralement, une analyse inter-procédurale permettrait de passer les arguments dans des registres non conventionnels.

c) Quel est le coût de l'expansion en ligne ? Quel compromis proposer ?

Réponse : *L'expansion en ligne duplique le code du corps de la fonction. De façon générale, il faut limiter l'expansion en ligne en tenant compte du rapport entre la taille du code source et celle du code résultant de l'expansion en ligne et éventuellement du gain en rapidité d'exécution.*

Pour de petites fonctions auxiliaires, comme ci-dessus, le code résultant de l'expansion est plus petit que le code d'un appel de fonction (qui inclue le code protocolaire de placement des arguments, et de récupération du résultat). On a donc simultanément un gain en espace et en rapidité d'exécution. De tels appels peuvent être systématiquement mis en ligne.

À l'opposé de grosses fonctions ne seront jamais mises en ligne, sauf peut-être si elles ne sont appelées qu'une seule fois, ou si elles sont dans à l'interieurs de boucles critiques les plus internes.

Les fonctions récursives sont un cas particulier pour lesquelles l'expansion en ligne n'est pas possible. (On peut en revanche dérouler la récursion un petit nombre de fois.)

1.4

L'expansion en ligne naïve ne fonctionne pas correctement. Considérons la fonction :

function quatre (n : integer) : integer;
begin
quatre := carré (carré (n∗n))
end;

a) Expliquer sur cet exemple un problème lié à l'expansion naïve.

Réponse : *La traduction naïve donne :*

```
function quatre (n : integer) : integer;
begin
    quatre := ((n * n) * (n * n)) * ((n * n) * (n * n))
end;
```

L'argument de `carré` *(par exemple* `(n*n)` *mais aussi* `carré (n*n)`*) est évalué deux fois (au toral* `(n*n)` *est évalué quatre fois) ce qui est inefficace.*

b) En déduire un exemple montrant que l'expansion naïve peut aussi être incorrecte.

Réponse : *Il suffit d'inclure un effet de bord dans le passage d'argument.*

```
function trace (n : integer) : integer;
begin
   write (x);
   trace := x
end;
function erronée () : integer;
begin
    erronée := carré (trace x)
end;
```

La fonction `erronée` *évalue* `trace x` *une seule fois. Après l'expansion en ligne naïve,* `trace x` *sera évalué deux fois et imprimera deux fois la valeur de* `x`*, changeant ainsi la sémantique du programme.*

c) Donner une meilleure version de l'expansion de `carré` dans la fonction `quatre` qui suggère un schéma général et correct.

Réponse : *Pour n'évaluer les arguments qu'une seule fois, on place leur valeur dans des variables auxillaires. Pour pouvoir emboîter les appels imbriqués de façon systématique, on place également le résultat de chaque appel dans des variables auxillaires.*

Pour cela, on introduit pour chaque fonction expansée en ligne autant de variables nouvelles que d'arguments, plus une variable nouvelle pour le résultat. On prend soin de renommer les variables du corps de la fonction expansée en ligne de façon cohérente.

```
function quatre (n : integer) : integer;
var n1 : integer; var carré1 : integer;
var n2 : integer; var carré2 : integer;
begin
    n1 := n;                (* argument de l'appel le plus interne *)
    carré1 := n1 * n1;      (* résultat du premier appel *)
    n2 := carré1;           (* argument du deuxième appel *)
    carré2 := n2 * n2;      (* argument de l'appel externe *)
    quatre := carré2
end;
```

Précisons que les variables auxilliaires ne pénalisent pas (en général) le programme, car elles seront associées à des temporaires qui pourront si besoin partager le même registre. Au pire, dans des cas pathologiques, la séquentialisation pourrait éliminer quelques opportunités de simplification.

2 Langage de parenthèses

Sur l'alphabet $\Sigma = \{g, d\}$ on considère la grammaire G_1 (resp. G_2) définie par l'axiome S_1 (resp. S_2) et les règles de production ci-dessous à gauche (resp. à droite) :

$S_1 \to E$	$S_2 \to E$
$E \to gEd$	$E \to gEdE$
$E \to EE$	$E \to \epsilon$
$E \to \epsilon$	

2.1

a) Donner deux dérivations de $ggdgdd$ l'une dans G_1 et l'autre dans G_2.

Réponse : *Dérivation de ggdgdd.*

$$\underline{S_1} \to \overline{\underline{E}} \to \overline{g\underline{E}d} \to g\overline{\underline{E}E}d \to g\overline{g\underline{E}d}Ed \to ggd\underline{E}d \to ggd\overline{g\underline{E}d}d \to ggdgdd$$

$$\underline{S_2} \to \overline{\underline{E}} \to \overline{gEd\underline{E}} \to g\underline{E}d \to g\overline{g\underline{E}dE}d \to ggd\underline{E}d \to ggd\overline{g\underline{E}dE}d \to ggdgd\underline{E}d \to ggdgdd$$

b) Les grammaires G_1 et G_2 sont-elles ambigües ? LL(1) ? LR(1) ?

Réponse : *La grammaire G_1 est ambigüe car, par exemple, on peut reconnaître le mot vide par des dérivations appartenant à des arbres syntaxiques (ou arbres de dérivations) différents :*

$$S_1 \to E \to \left\{ \begin{array}{l} E \to \epsilon \\ E \to \epsilon \end{array} \right. \qquad\qquad S_1 \to E \to \epsilon$$

Donc elle n'est pas LR(1) ni a fortiori LL(1).

La grammaire G_2 est LL(1). Pour le vérifier, nous calculons pour chaque règle $\alpha \longrightarrow \beta$ les ensembles $first(\beta)$, $follow(\alpha)$, et nous vérifions si $\alpha \longrightarrow^* \epsilon$; finalement nous construisons la relation de transition de domain $V \times \Sigma$. Les calculs sont résumés dans la table suivante :

α	β	$first(\beta)$	$follow(\alpha)$	$\alpha \longrightarrow^* \epsilon$	g	d
S	E	g	g	yes	$\surd$	
E	$gEdE$	g	d	yes	$\surd$	
	ϵ					$\surd$

La fonction de transition donnée par les $\surd$ dans la sous-matrice en bas à droite ne comporte qu'un élément par case ; elle définie donc bien une fonction. La grammaire est donc LL(1). Par conséquence, elle est aussi LR(1) et n'est pas ambigüe.

2.2

La grammaire G_1 définit le langage des parenthèses L. Peut-on reconnaître L par une expression régulière ? par un lexeur ? (Justifier très brièvement chaque réponse).

Réponse : *Le langage des parenthèses L ne peut pas être reconnu par une expression régulière. Sinon, il serait reconnaissable par un automate fini. Or, les ensembles $L_n = \{w \mid g^n w \in L\}$ des suffixes des mots de L commençants par g^n sont deux à deux distincts (au moins un mot est dans l'un mais pas dans l'autre). Il faudrait donc un nombre d'états arbitraires à un automate pour disntinguer l'ensemble des configurations possibles au cours de la reconnaissance d'une expression.*

L peut être reconnu par un lexeur mais seulement en utilisant la puissance du langage hôte (qui est un complet en général) dans les actions du lexeur pour compter les parenthèses ouvertes.

2.3

Montrer que le langage reconnu par G_2 est un sous ensemble de L.

Réponse : *La règle $E \to gEdE$ est une règle dérivée dans G_1 par $E \to \underline{E}E \to \overline{gEd}E$. On ne change pas le langage reconnu par G_1 en lui ajoutant cette règle dérivée (on appelle G_1' la grammaire résultante). La grammaire G_2 peut être plongée de façon homomorhe dans la grammaire G_1'. En effet, au renommage des non-terminaux près, ses règles sont incluses dans celles de G_1'. Une dérivation de G_2 est donc aussi une dérivation de G_1' au renommage des non-terminaux près. Ainsi tout mot reconnu par G_2 est aussi reconnu par G_1' et appartient donc à L.*

2.4

On peut représenter les arbres syntaxiques (ou arbres de dérivations) de la grammaire G_1 par un type concret `Ocaml` :

```
type g1 = G1_S of e1
and e1 =
    G1_gEd of e1
  | G1_EE of e1 * e1
  | G1_0;;
```

a) Donner en `Ocaml` une définition de type `g2` représentant les arbres syntaxiques de la grammaire G_2.

Réponse :

```
type g2 = G2_S of e2
and e2 =
  | G2_gEdE of e2 * e2
  | G2_0;;
```

b) Donner en `Ocaml` l'arbre syntaxique `a2` correspondant à la dérivation de *ggdgdd* dans G_2 que vous avez donnée ci-dessus.

Réponse :

```
let a2 = G2_S (G2_gEdE (G2_gEdE (G2_0, G2_gEdE (G2_0, G2_0)), G2_0));;
```

c) Écrire en `Ocaml` une fonction `mot1` qui prend un arbre syntaxique de type `g1` et qui retourne le mot (chaîne de caractères) du langage L lui correspondant.

Réponse :

```
let rec lemot1 = function
  | G1_gEd e -> "g" ^ (lemot1 e) ^ "d"
  | G1_EE (e, e') -> (lemot1 e) ^ (lemot1 e')
  | G1_0 -> ""
let mot1 (G1_S e) = lemot1 e;;
```

d) Écrire en `Ocaml` une fonction totale `réécrit` qui transforme un arbre syntaxique `a1` de type `g1` en un arbre syntaxique `a2` de type `g2` tel que `a2` représente le même mot que `a1` (la preuve de correction n'est pas demandée dans cette question).

Réponse :

```
let rec réécrit_le  = function
    G1_EE (G1_EE (e, e'), e") -> réécrit_le (G1_EE (e, G1_EE (e', e")))
  | G1_EE (G1_gEd e, e') -> G2_gEdE (réécrit_le e, réécrit_le e')
  | G1_EE (G1_0, e) -> réécrit_le e
  | G1_gEd e -> G2_gEdE (réécrit_le e, G2_0)
  | G1_0 -> G2_0;;
let réécrit  (G1_S e) = G2_S (réécrit_le e);;
```

e) Justifier la terminaison de ce programme.

Réponse : *La fonction* `réécrit` *est récursive. Cependant l'argument de chaque appel récursif est un sous-arbre strict de l'argument de l'appel principal, sauf dans le premier cas du filtrage : l'argument de l'appel principal est alors un nœud de la forme* `G1_EE(u,v)` *et l'argument de l'appel récursif est un nœud de même forme* `G1_EE(u',v')` *tel que* `u'` *soit un sous-arbre strict de* `u`*. On ne peut donc passer par le premier cas de filtrage qu'un nombre fini de fois ; ensuite l'argument de l'appel récursif ne peut que décroître strictement.*

f) On admet la correction de la fonction `réécrit`. En déduire que G_2 reconnaît tout L.

Réponse : *La correction assure que la fonction* `mot1` *est égale à la composée de* `mot2` *(définie de façon analogue à* `mot1`*) et de* `réécrit`*. Tout mot w de L peut être reconnu par G_1 (par définition de L). Il existe donc un arbre syntaxique a_1 dont w est l'image par* `mot1`*. L'image de a_1 par la fonction* `réécrit` *est un arbre a_2 de la grammaire G_2 dont l'image par* `mot2` *est le même mot w. Le mot w est donc aussi reconnu par G_2.*

```
let a1 = G1_S (G1_gEd (G1_EE (G1_gEd G1_0, G1_gEd G1_0)));;
```

```
let rec lemot2 = function
  | G2_gEdE (e, e') -> "g" ^ (lemot2 e) ^ "d" ^ (lemot2 e')
  | G2_0 -> ""
let mot2 (G2_S e) = lemot2 e;;

mot1 a1;;
mot2 a2;;

réécrit a1 = a2;;


type binop = unit;;
type type_expr = unit;;
```

B Examen de l'année 2000 (durée 2h)

On attachera une importance particulière à la clarté, à la précision, mais aussi à la ***concision*** *de la rédaction. Les questions 1 et 4 sont indépendantes, ainsi que certaines sous-questions d'autres parties.*

Mise en pile

Dans le cours, nous avons apporté un soin particulier à l'allocation de registres de façon à éviter au maximum les mises en pile et à éliminer le plus possible d'instructions `move`. Cependant, dans le cas d'une mise en pile, nous avons fait le choix délibéré de la simplicité, parfois au détriment de la qualité (rapidité d'exécution) du code produit. On cherche ici à améliorer le résultat de la mise en pile.

1 De bons exemples

On considère la fonction suivante d'un programme Pseudo-Pascal :

```
function sum (var n : integer) : integer;
begin
  if n > 0 then sum := n + sum (n−1) else sum := 0
end;
```

1.1 Donner un code Spim possible, tel que pourrait le retourner le compilateur du cours (en particulier, on n'effectuera pas de transformation de programme avancée, telle que l'élimination de la récursion).

Réponse :

```
      sum_f=8
sum:
      sub   $sp, $sp, sum_f
      sw    $ra, 4($sp)
      sw    $s0, 0($sp)
      move  $s0, $a0
      ble   $s0, $zero, L2
      sub   $a0, $s0, 1
      jal   sum
      add   $v0, $s0, $v0
L1:
      lw    $s0, 0($sp)
      lw    $ra, 4($sp)
      add   $sp, $sp, sum_f
      j     $ra
L2:
      move  $v0, $zero
      j     L10
```

1.2 Donner un argument prouvant que la mise en pile est inévitable dans la compilation du programme ci-dessus (on s'interdit toujours des transformations de programme avancées), quelque soit la qualité de l'allocateur de registres et le nombre de registres disponibles.

Réponse : *La fonction* `sum` *est récursive, non terminale. Il faut donc revenir à la fonction après l'appel récursif. Pour cela l'adresse de retour (la valeur du registre* **$ra***) doit être sauvegardée pendant l'appel récursif (qui utilise lui-même* **$ra***). Aucun registre n'est candidat à la sauvegarde de* **$ra***, car il serait lui-même écrasé : c'est le même code qui effectue l'appel récursif donc il utilise potentiellement (*i.e. *c'est une approximation statique) les mêmes registres que l'appel lui-même.*

(On pourrait sauvegarder **$ra** *dans la tas, mais cela revient à implémenter une pile dans le tas.)*

1.3 La mise en pile est-elle satisfaisante dans l'exemple ci-dessus ?

Réponse : *Oui, on ne peut guère faire mieux à la main (sans transformation de programme avancée) : pour chacun des registres `ra` et `s0`, il n'y a qu'une seule sauvegarde et restauration en pile qui est inévitable.*

1.4 En induire un cas, assez fréquent, de mise en pile bien traité par la solution simple du cours.

Réponse : *Les registres sauvés par l'appelant sont placés dans un temporaire au début du programme et restaurés à la fin. Si le temporaire utilisé pour cette sauvegarde doit être mise en pile, cette mise en pile sera efficace car le temporaire n'a qu'une seule occurrence en écriture et en lecture.*

2 Des insuffisances

Dans toute la suite du problème, on utilise les notations et restrictions suivantes. On suppose qu'il n'existe que trois registres machine **$a0**, **$v0** et **$t0** qui sont aussi des temporaires particuliers, pré-coloriés (on réserve le préfixe $ aux registres machine). Pour simplifier, on traite les registres spéciaux (**$sp**, **$ra** et **$zero**) à part : on ne les considérera pas comme des temporaires et on les exclut de toutes les phases d'analyse (durée de vie, interférence, allocation de registres).

On note $\mathcal{G}(A)$ le graphe d'interférence associé au code assembleur A et $\varphi(A)$ le code produit par la mise en pile d'un seul temporaire t dans le code A en utilisant la procédure du cours (le temporaire t est toujours le même et laissé implicite).

2.1 On considère comme exemple le code C suivant :

```
1:    add  t,   $a0, 1
2:    add  t,   t, $a0
3:    add  $a0, t, t
4:    jal  foo
5:    add  $v0, t, $v0
```

où la fonction `foo` lit **$a0**, écrase les trois registres machines et retourne son résultat dans `v0`. On suppose que le registre **$v0** est vivant à la fin du code C. Donner le code $\varphi(C)$.

Réponse :

```
1:    add  p1,  $a0, 1
      sw   p1,  0($sp)
2:    lw   q1,  0($sp)
      add  p2,  q1, $a0
      sw   p2,  0($sp)
3:    lw   q2,  0($sp)
      add  $a0, q2, q2
4:    jal  foo
5:    lw   q3,  0($sp)
      add  $v0, q3, $v0
```

2.2 Donner au moins une raison pour laquelle le code $\varphi(C)$ n'est pas bon.

Réponse : *L'instruction **lw** `q1`, 0(**$sp**) rappelle le contenu du sommet de pile dans le temporaire auxiliaire `q1`. C'est absurde, car ce contenu est aussi celui du temporaire `p1` : on pourrait certainement remplacer **lw** `p1`, 0(**$sp**) par l'instruction **move** `q1`, `p1` qui évite une lecture en pile inutile (de plus, cette instruction sera ensuite éliminée par l'allocation de registres). En général, certaines occurences de t n'ont pas besoin d'être traitées en pile (qu'il s'agisse d'une lecture ou d'une écriture).*

2.3 Donner un code de mise en pile meilleure que $\varphi(C)$, obtenu manuellement, en utilisant au mieux les registres disponibles (on donnera le code final, après allocation des registres).

Réponse :

```
1:    add  $t0, $a0, 1
2:    add  $t0, $t0, $a0
      sw   $t0, 0($sp)
```

```
3:    add  $a0, $t0, $t0
4:    jal  foo
5:    lw   $t0, 0($sp)
      add  $v0, $t0, $v0
```

(On pourrait inclure les instructions d'ajustement de pile ***sub $sp, $sp****, 4 au début, et* ***add $sp, $sp*** *4 à la fin, ici comme à la question précédente, bien qu'elles ne font pas partie du code de mise en pile, mais de la phase finale postérieure à l'allocation de registres.)*

3 Approche mécanique

On note $\varphi^\dagger(A)$ le programme A transformé par la procédure suivante :

1. On donne un numéro $i \in I$ à chaque instruction qui lit le temporaire t et un numéro $j \in J$ à chaque instruction qui écrit le temporaire t (une instruction peut avoir à la fois un numéro $i \in I$ et un numéro $j \in J$ si t y est utilisé à la fois en lecture et en écriture.)
2. On se donne des nouveaux temporaires $q_i^{i\in I}$ et $p_j^{j\in J}$.
3. On remplace dans chaque instruction $i \in I$ toutes les occurrences de t en lecture par q_i.
4. On remplace chaque instruction $j \in J$ par la séquence d'instructions composée, dans l'ordre :
   (a) de l'instruction j dans laquelle toutes les occurrences en écriture de t sont remplacées par p_j,
   (b) des instructions $(\textbf{move}\ q_i,\ p_j)^{i\in I}$ (on rappelle que **move** u, v copie v dans u).

3.1 Donner, très soigneusement, le code $\varphi^\dagger(C)$.

Réponse :

```
                                # Temporaires vivants en sortie

1:    add  p1,  $a0, 1          # $a0, p1
      move q1,  p1              # $a0, p1, q1
      move q2,  p1              # $a0, p1, q1
      move q3,  p1              # $a0, q1
2:    add  p2,  q1, $a0         # p2
      move q1,  p2              # p2
      move q2,  p2              # p2, q2
      move q3,  p2              # q2, q3
3:    add  $a0, q2, q2          # $a0, q3
4:    jal  foo                  # $v0, q3
5:    add  $v0, q3, $v0         # $v0
```

3.2 Montrer, dans le cas général, que la transformation de A en $\varphi^\dagger(A)$ est correcte, c'est-à-dire que les deux codes assembleurs sont équivalents (pour une machine qui préserverait la valeur des temporaires à chaque appel de fonction).

Réponse : *On considère comme une macro-instruction l'image $\varphi^\dagger(k)$ par réécriture d'une instruction k de A. On exécute en parallèle les codes A et $\varphi^\dagger(A)$. On donne au départ des valeurs indéfinies à t et à tous les ${q_i}^{i\in I}$. On montre facilement, par récurrence, qu'à chaque pas $s \in I\!N$ (un pas correspond à l'exécution d'une instruction ou d'une macro-instruction)*

- *si le programme A est à l'instruction k, alors le programme $\varphi^\dagger(A)$ est à l'instruction $\varphi^\dagger(k)$;*
- *les valeurs des ${q_i}^{i\in I}$ sont toutes identiques dans le programme $\varphi^\dagger(A)$ et égales à la valeur de t dans le programme A ;*
- *pour tous les autres temporaires de A leur valeur dans $\varphi^\dagger(A)$ est égale à leur valeur dans A.*

  (Formellement, comme c'est évidemment vrai au début du programme, il suffit de vérifier que si la propriété est vérifiée pour s, elle l'est aussi pour $s+1$. Soit $k \in A$ la position du programme dans A à l'instant s. Si l'instruction k effectue un saut dans A à l'instruction l, alors compte tenu de la correspondance entre les valeurs des registres, l'instruction $\varphi^\dagger(k)$ effectue un saut à

l'instruction l de $\varphi^\dagger(A)$, donc le programme est à l'instruction $l \in A$ et $\varphi^\dagger(l) \in \varphi^\dagger(A)$ à l'instant $s+1$.

- *Si l'instruction l lit une valeur v dans t (la valeur de t à l'instant s), la macro-instruction $\varphi^\dagger(l)$ lit à la place la valeur de q_i, donc v, pour un certain i de I.*
- *Si l'instruction l écrit v dans t , alors l'instruction $\varphi^\dagger(l)$ écrit v dans p_j puis écrit la valeur de p_j donc v dans tous les ${q_i}^{i\in I}$.)*

Ainsi, à une trace d'exécution de A correspond une trace d'exécution de $\varphi^\dagger(A)$ ayant un comportement identique vis à vis de l'extérieur. Comme l'exécution est déterministe, la trace observée pour $\varphi^\dagger(A)$ est la seule possible.

3.3 Donner, soigneusement, pour chaque instruction de $\varphi^\dagger(C)$ la liste des temporaires vivants en sortie (on ne demande pas le détail du calcul, mais seulement le résultat).

Réponse : *Voir réponse à la question 3.1.*

3.4 En déduire immédiatement (donc sans faire d'allocation de registres) que l'on peut supprimer certaines instructions **move**.

Réponse : *On peut supprimer les instructions **move** dont le registre de destination n'est pas vivant en sortie.*

3.5 Donner le code résultant que l'on notera C'.

Réponse :

```
1:   add  p1,  $a0, 1    # $a0, p1
     move q1,  p1        # $a0, p1, q1
2:   add  p2,  q1, $a0   # p2
     move q2,  p2        # p2, q2
     move q3,  p2        # q2, q3
3:   add  $a0, q2, q2    # $a0, q3
4:   jal  foo            # $v0, q3
5:   add  $v0, q3, $v0   # $v0
```

4 Interférences parasites

On s'intéresse maintenant au bloc de code B suivant :

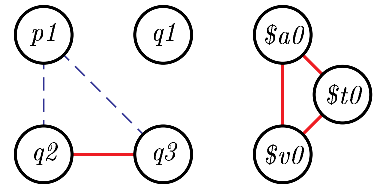
```
add  p2,  q1, $a0
move q2,  p2
move q3,  p2
```

On suppose que les registres q2 et q3 sont les seuls vivants en sortie du bloc. Donner pour chaque instruction les registres vivants en sortie. Donner le graphe d'interférence (sans justification).

Réponse :

```
                      # vivants    # Interférences  engendrées
add  p2,  q1, $a0     # p2         #
move q2,  p2          # p2 q2      #
move q3,  p2          # q2 q3      # q2−q3
```

- *Le graphe d'interférence comporte les trois registres et quatre autres nœuds p2, q1, q2. q3. Il comporte un seul arc* (q2, q3) *d'interférence en plus du sous-graphe plein des interférences entre registres (que l'on pourrait omettre).*
- *Les arcs d'attirance qui ne sont pas demandés.*



4.1 Exhiber un arc superflu. En quoi est-ce gênant ?

Réponse : *Les temporaires q2 et q3 interfèrent. Comme ces temporaires ont toujours la même valeur (lorsque celle-ci est utile), on pourrait en fait leur associer le même registre machine, ce qui est ici interdit par leur interférence, au risque de ne pas pouvoir les colorier du tout.*

4.2 On regroupe les deux instructions **move** q2, p2 et **move** q3, p2 ensemble en une macro-instruction **mmv** q2 q3, p2 (lire "multi-move") qui lit p1 et écrit q2 et q3.

Que devient le graphe d'interférence si le multi-move est traité comme une instruction de calcul ordinaire (nœud Oper dans le cours) ?

Réponse : *Les arcs d'interférence sont inchangés (on ignore les arcs d'attirance) : les 2 temporaires q2 et q3 qui survivent à l'instruction sont écrits par celle-ci donc ils interfèrent entre eux.*

4.3 On traite le multi-move comme une forme généralisée de **move** : l'instruction **mmv** $d_1 \dots d_n,\ s$ copie le temporaire s dans les temporaires $(d_k)^{k \in 1..n}$.

Décrire les interférences minimales crées par une instruction **mmv** $d_1 \dots d_n,\ s$ en fonction de l'ensemble *dest* des temporaires $(d_k)^{k \in 1..n}$, de s et de l'ensemble *out* des temporaires vivants en sortie de cette instruction.

Réponse : *Chaque temporaire d_k interfère avec les temporaires vivants autres que s et ceux de dest. Au total, le sous-graphe induit par une telle instruction est dest* $\times$ (*out* $\setminus$ ($\{s\} \cup$ *dest*)).

4.4 Donner le graphe d'interférence $\mathcal{G}(B)$ en considérant les deux moves successifs comme un multi-move.

Réponse : *Les seuls registres vivants en sortie de $\mathcal{G}(B)$ sont dans la destination du multi-move, donc n'interfèrent pas. Ainsi $\mathcal{G}(B)$ est le graphe vide (*i.e. *sans arcs, il comporte bien sûr des nœuds).*

5 Exemple à suivre

Dans la suite, on note $\varphi^{\ddagger}(A)$ le code assembleur obtenu à partir de $\varphi^{\dagger}(A)$ en remplaçant les séquences de **move** introduites par des instructions **mmv**, puis en simplifiant ces instructions après l'analyse de durée de vie (en retirant certaines destinations, donc certains move) comme indiqué à la question 3.4.

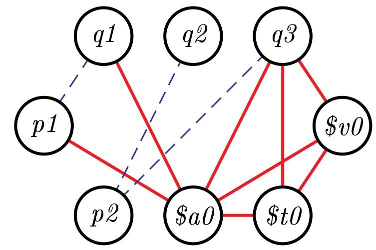
5.1 Retrouver les temporaires vivants à la sortie de chaque instruction du code $\varphi^{\ddagger}(C)$ (en réutilisant simplement les réponses aux questions 3.3 et 3.5). En déduire $\mathcal{G}(\varphi^{\ddagger}(C))$.

Réponse :

```
                              # vivants       # Interférences engendrées

1:    add   p1,  $a0, 1       # $a0, p1       # $a0−p1
      mmv   q1,  p1           # $a0, q1       # $a0−q1
2:    add   p2,  $a0, q1      # p2            #
      mmv   q2 q3,  p2        # q2, q3        #
3:    add   $a0,  q2, q2      # $a0, q3       # $a0−q3
4:    jal   foo               # $v0, q3       # $a0−q3, $v0−q3, $t0−q3
5:    add   $v0, q3, $v0      # $v0
```

- *On pourrait omettre les arcs entre registres qui forment un sous-graphe plein.*
- *On a mis les arcs d'attirance qui ne sont pas demandés mais qui aident au choix des registres demandé dans la question suivante.*



5.2 Donner un résultat possible pour l'allocateur de registres, en tenant compte des contraintes d'interférence, et si possible des préférences (arcs "move") : on indiquera dans une table pour chaque temporaire le registre-machine associé ou bien le temporaire lui-même s'il doit être mis en pile.

Réponse :

Temporaire	*\$a0*	*\$v0*	*\$t0*	*p1*	*p2*	*q1*	*q2*	*q3*
Registre	*\$a0*	*\$v0*	*\$t0*	*\$t0*	*\$t0*	*\$t0*	*\$t0*	*q3*

5.3 Donner le code résultant après expansion des instructions **mmv** et simplification des instructions **move** devenues inutiles par la fusion de leur source et de leur destination.

Réponse :

```
1:   add   $t0,  $a0, 1
2:   add   $t0,  $t0, $a0
     move  q3,   $t0
3:   add   $a0,  $t0, $t0
4:   jal   foo
5:   add   $v0,  q3, $v0
```

6 Retour au cas général

On reprend le cas général d'un code A, coloriable à l'exception d'un seul temporaire t qu'il faut placer en pile.

6.1 Montrer que le graph $\mathcal{G}(\varphi^{\ddagger}(A))$ est coloriable à l'exception peut-être de certains nœuds parmi ${q_i}^{i\in I}$ et ${p_j}^{j\in J}$.

Réponse : *Il suffit de vérifier qu'une fois retirés tous les nœuds* ${q_i}^{i\in I}$ *and* ${p_j}^{j\in J}$ *de* $\mathcal{G}(\varphi^{\ddagger}(A))$*, on obtient un sous-graphe de* $\mathcal{G}(A)$*, donc coloriable. En l'occurence, il est exactement égal au graphe* $\mathcal{G}(A)$ *privé du nœud* t*.*

*En effet, pour les autres noeuds (*i.e. *temporaires), les variables lus et écrites dans chaque instruction de* A *et de* $\varphi^{\ddagger}(A)$ *sont identiques. Les temporaires restants vivants en sortie de chaque instruction sont donc identiques (les instructions ajoutées dans* $\varphi^{\ddagger}(A)$ *ne concerne pas ces temporaires). Par conséquent les contraintes d'interférence générées pour les temporaires restant sont aussi identiques.*

6.2 Peut-on être plus précis si l'on sait que $\mathcal{G}(\varphi(A))$ est coloriable (on ne fera pas de preuve) ?

Réponse : *Les nœuds* ${t_j}^{j\in J}$ *sont alors aussi coloriables dans* $\mathcal{G}(\varphi^{\ddagger}(A))$ *(on inclut les nœuds* ${t_j}^{j\in J}$ *et on remplace* A *par* $\varphi^{\dagger}(A)$ *dans le raisonnement précédent).*

6.3 On définit $\varphi^{\#}(A)$ à partir de $\varphi^{\ddagger}(A)$ en remplaçant chaque instruction i par l'instruction **move** $r_i,\ q_i$ suivie de l'instruction i dans laquelle q_i est remplacé par r_i où r_i est un nouveau temporaire. Quel est l'intérêt de colorier $\mathcal{G}(\varphi^{\#}(A))$ plutôt que $\mathcal{G}(\varphi^{\ddagger}(A))$? En déduire, dans certains cas, une forme définitive, $\varphi^{\infty}(A)$, du code.

Réponse : *Dans ce cas, les nœuds* ${r_i}^{i\in I}$ *sont aussi coloriables dans* $\mathcal{G}(\varphi^{\#}(A))$. *Par conséquent, il existe un coloriage pour lequel l'ensemble* S *des nœuds non coloriables est un sous ensemble de* Q *égal à* ${q_i}^{i\in I}$. *Comme les temporaires* S *n'apparaissent que dans des instructions de la forme* **mmv** $Q_j,\ p_j$ *(où* $Q_j \subset Q$*) ou bien* **move** $r_i,\ q_i$*, il suffit de remplacer les premières par* ***sw*** p_j*, 0(****\$sp****) suivi de* **mmv** $Q_j \setminus S,\ p_j$ *et les secondes par* ***lw*** r_i*, 0(****\$sp****) pour obtenir le code* $\varphi^{\infty}(A)$. *Le graphe d'interférence résultant est un sous-graphe du précédent, sans les nœuds mis en pile, donc il est coloriable, avec les mêmes couleurs. Aussi on n'a pas besoin de refaire un nouveau coloriage.*

6.4 *(facultative)* Quand le graphe n'est pas coloriable, il existe, en général plusieurs coloriages partiels possibles. Comment peut-on guider l'algorithme de coloriage du cours pour qu'il aboutisse, autant que possible, vers une forme permettant d'obtenir le code définitif $\varphi^{\infty}(A)$ à partir du coloriage de $\mathcal{G}(\varphi^{\#}(A))$?

Réponse : *Lors de l'allocation de registres, lorsqu'il n'existe plus de nœuds simplifiables (*i.e. *on n'est alors pas sûr de pouvoir colorier le graphe), on choisit de façon optimiste un nœud complexe que l'on simplifie. On peut guider ce choix en prenant par priorité :*

- *les registres* ${q_i}^{i\in I}$ *(que l'on s'attend à mettre en pile)*
- *les registres anciens (autres que* ${q_i}^{i\in I}$, ${r_i}^{i\in I}$ *et* ${p_j}^{j\in J}$*) que l'on espère ne pas avoir à mettre en pile.*
- *les registres éphémères* ${p_j}^{j\in J}$ *et* ${r_i}^{i\in I}$*, que l'on ne veut surtout pas mettre en pile.*

C Suite de l'examen de l'année 2000

Si vous êtes curieux

Ces questions ne faisaient pas partie de l'examen, mais en sont une suite naturelle. On reprend les notations et les définitions du sujet de l'examen.

7 Lectures superflues

On considère le code D suivant :

```
add t, $a0, 1
jal foo
add $v0, t, $v0
add $v0, t, $v0
```

où le registre $v0$ est vivant en sortie.

7.1 Donner le code après mise en pile $\varphi^{\infty}(D)$ (i.e., déduit du coloriage de $\varphi^{\#}(D)$) : on donnera d'abord le code avant application du coloriage puis le code final colorié.

Réponse : *La transformation automatique introduit ici deux temporaire `q1` et `q2` écrit avant l'appel à la fonction `foo` et lus tous deux après : ils devront donc être mise en pile. On obtient, après mise en pile :*

add p1, $a0, 1	add $t0, $a0, 1
sw p1, 0($sp)	sw $t0, 0($sp)
jal foo	jal foo
lw r1, 0($sp)	lw $t0, 0($sp)
add $v0, r1, $v0	add $v0, $t0, $v0
lw r2, 0($sp)	lw $t0, 0($sp)
add $v0, r2, $v0	add $v0, $t0, $v0

7.2 Peut-on, manuellement, obtenir un code meilleur ? (On décrira seulement le code final après mise en pile et application du coloriage).

Réponse : *Il suffit de retirer la dernière instruction sur dans le code précédent.*

```
add $t0, $a0, 1
sw $t0, 0($sp)
jal foo
lw $t0, 0($sp)
add $v0, $t0, $v0

add $v0, $t0, $v0
```

Évidemment, le move résultant sera à son tour éliminé par l'allocateur de registres.

7.3 On définit la transformation $\varphi^{*}(A)$ comme la transformation $\varphi^{\#}(A)$ suivie du remplacement de chaque instruction **move** $r_i,\ q_i$ par un multi-move (**mmv** $(q_k)k \in I, k \neq i\ r_i,\ q_i$).

Décrire l'effet général de cette transformation. Vérifiez qu'elle traite correctement le cas ci-dessus.

Réponse : *Cette transformation met à jour à chaque lecture de q_i tous les autres q_k avec la dernière valeur lue. L'effet est de recourcir les durées de vie des temporaores q_i : la lecture de q_i ne dépend plus de la dernière écriture de t, mais de la dernière lecture ou écriture de t.*

*Dans l'exemple précédent, on obtient avant allocation de registres (à droite après élimination des **move** inutiles) :*

```
1:   add p1, $a0, 1          add p1, $a0, 1
2:   mmv q1 q2, p1           mmv q1, p1
3:   jal  foo                jal  foo
4:   mmv q2 r1, q1           mmv q2 r1, q1
5:   add $v0, r1, $v0        add $v0, r1, $v0
6:   mmv q1 r2, q2           mmv r2, q2
7:   add $v0, r2, $v0        add $v0, r2, $v0
```

Ainsi, il n'y a plus besoin de placer `q2` *en pile car sa durée de vie ne contient plus l'appel à* `foo`*. En appliquant le coloriage sur la partie droite, on retrouve le code obtenu manuellement ci-dessus.*

8 Effet pervers

La transformation précédente permet de réduire le nombre de sites mis en pile, en raccourcissant la longueur des interférences. Cependant, en retardant la mise en pile le plus tard possible, elle a aussi des effets secondaires.

8.1 Donner un exemple où la transformation entraîne une sauvegarde en pile alors que l'instruction correspondante du code d'origine ne faisait que lire t. (On donnera le code d'origine et le code final après la mise en pile et application du coloriage).

Réponse :

```
1:   add t, $a0, 1           add $t0, $a0, 1
2:   add $s0, t, 1           add $s0, $t0, 1
                             sw   $t0, 0($sp)
3:   jal  foo                jal  foo
                             lw   $t0, 0($sp)
4:   add $v0, t, $v0         add $v0, $t0, $v0
```

8.2 En déduire (en utilisant une boucle) un cas, où la transformation conduit finalement à un résultat moins bon que $\varphi^{\#}(A)$.

Réponse : *Il suffit d'ajouter une itération du bloc de code 2-4. On donner, de gauche à droite, le code d'origine, le code $\varphi^*(A)$ et le code $\varphi^{\#}(A)$, ces deux derniers après mise en pile.*

```
1:   add t, $a0, 1      add $t0, $a0, 1       add $t0, $a0, 1
                                              sw   $t0, 0($sp)
2:   add $s0, t, 1      add $s0, $t0, 1       add $s0, $t0, 1
                        sw   $t0, 0($sp)
3:   jal  foo           jal  foo              jal  foo
                        lw   $t0, 0($sp)      lw   $t0, 0($sp)
4:   add $v0, t, $v0    add $v0, $t0, $v0     add $v0, $t0, $v0
5:   bltz $v0, 3        bltz $v0, 3           bltz $v0, 3
```

La sauvegarde est effectuée à chaque boucle dans le code $\varphi^*(A)$, alors qu'elle n'est effectuée qu'une seule fois en dehors de la boucle dans $\varphi^{\ddagger}(A)$.

9 Une solution conservatrice

On effectue toujours l'allocation de registres pour le code $\varphi^*(A)$, ce qui fournit un coloriage θ et une partition des temporaires $Q = \{q_i, i \in I\}$ en deux ensembles R et S, les premiers étant coloriables alors que les seconds ne le sont pas et devront être mis en pile. Afin de corriger l'effet pervers précédent, on change la procédure de mise en pile : on décide de sauvegarder p_j en pile après change instruction j, maintenant ainsi la pile à la valeur de t avant tansformation, ce qui permet de ne plus jamais sauver r_i en pile.

Donner un exemple où cette transformation est moins bonne que $\varphi^{\#}(A)$.

Réponse : *C'est l'exemple C de l'examen. La nouvelle transformation va effectuer deux sauvergardes en pile (comme la procédure de mise en pile du cours), autant que d'instruction $j \in J$. Ici, le code $\varphi^{\#}(A)$ permettait de ne faire qu'une seule sauvegarde en pile.*

10 La solution, enfin

10.1 Montrer que les sauvegardes en pile superflues peuvent facilement être éliminées par une analyse de durée de vie...sur l'emplacement en pile.

Réponse : *On peut trace les lectures et les écritures en pile (dans l'emplacement réservé pour t) on le fait pour un temporaire. Une écriture en pile est inutile si dans toute trace d'exécution elle n'est jamais suivie directement d'une lecture en pile. Cela revient à l'élimination des moves superflues.*

10.2 Comment la mettre en œuvre simplement ?

Réponse : *Dans un premier temps on remplace les opérations en pile par des moves de ou vers un temporaire s représentant l'emplacement en pile. On fait l'analyse de durée de vie, puis on élimine parmi ces moves, ceux qui sont inutiles,* i.e. *les moves vers s lorsque s n'est pas vivant en sortie.*

10.3 Décrire comment à partir de $\varphi^*(A)$, du coloriage θ et de la partition $R \uplus S$ de Q obtenu par l'allocation de registres de $\varphi^*(A)$, on obtient le code final (que l'on note $\varphi^\infty(A)$).

Réponse : *Si S est non vide, on introduit un temporaire s pour représenter l'emplacement en pile et on appelle A_1 le code obtenu en remplaçant dans $\varphi^*(A)$:*

1. *chaque multimove* **move** $K_j,\ p_j$ *issu d'une lecture par* **move** $s,\ p_i$ *suivi de* **move** $K_j \cap R,\ p_i$
2. *chaque multimove* **move** $r_i\ K_i,\ q_i$ *issu d'une écriture par*
   - **move** $r_i,\ s$ *suivi de* **move** $K_i \cap R,\ r_i$ *si* $q_i \in S$ *et*
   - **move** $r_i\ K_i \cap R,\ q_i$ *sinon.*
3. *On effectue l'analyse de durée de vie (du registre s) sur le code A_1. On appelle A_2 le code obtenu en éliminant dans A_1 les moves superflus (de ou vers s).*
4. *On appelle A_3 le code obtenu en remplaçant dans A_2 les move de ou vers s restant par des lecture ou des écriture en pile.*

On applique le coloriage et la coalescence définis par le coloriage θ (sans refaire de calcul) au code A_3, ou à $\varphi^*(A)$ si S est vide. On obtient le code final $\varphi^\infty(A)$.

D Pour en savoir plus...

- Andrew Appel *Modern Compiler Implementation in ML*, Cambridge University Press, 1998.
- Aho, Sethi, Ullman, *Compilers : Principle, Techniques, and Tools,* Addison Wesley, 1986. *("Dragon Book").*
- Gerry Kane, *mips RISC Architechture*, 1989, Prentice Hall.
- Pierre Weis et Xavier Leroy *Le Langage Caml*, Chapitre 13, *Le Pico-processeur*, InterEditions, Paris, 1993. *(Sources disponibles dans la distribution Caml-Light)*
- James Larus *SPIM (*`http://www.cs.wisc.edu/~larus/spim.html`) (Un simulateur MIPS, R2000)
- Le cours *Programming Languages and Compilers (*`http://www-inst.EECS.Berkeley.EDU/ cs164/`) d'Alex Aiken à Berkeley.
- Le cours *Langages et Compilation (*`http://pauillac.inria.fr/~levy/x/m2/lc.html`) de Jean-Jacques Lévy de 1998.