Majeure 2

Informatique

Jean-Jacques Lévy

INRIA-Rocquencourt

tel:	01 39 63 56 89 inria
	01 45 67 68 54 maison
email:	`jean-jacques.levy@inria.fr`
web:	http://pauillac.inria.fr/~levy/

Les supports de cours sont en

http://www.polytechnique.fr/poly/edu/informatique/

Introduction

- Caml-light 0.74 est un dialecte de ML = Lisp + types
- Gestion automatique de la mémoire
- Appel de procédures avec filtrage
- Fonctionalité arbitraire
- Modularité, types de données abstraits
- Types polymorphes
- OCaml = Caml orienté objets + compilateur code natif
- Caml-light tourne en peu de mémoire, système à *byte-code*
- Développé à l'INRIA Rocquecourt.
  Meilleure implémentation existante.

Références

- Pierre Weis and Xavier Leroy. Le langage Caml. InterÉditions, 1993.
- Guy Cousineau and Michel Mauny. Approche fonctionnelle de la programmation. Ediscience international, 1995.
- Thérèse Accart Hardin and Véronique Donzeau-Gouge Viguié. Concepts et outils de programmation. InterÉditions, 1992.
- Jeffrey D. Ullman. Elements of ML Programming. Prentice Hall, 1994.

URLs

Des notes de cours se trouvent en

- http://www.polytechnique.fr/poly/ levy/
- http://www.polytechnique.fr/poly/ weis/
- http://www-spi.lip6.fr/ queinneçWWW/cours/tdcaml.html
- http://pauillac.inria.fr/caml/

Plan

1. Rattrapage (3×2h) en C et en ML
   Queinnec, Lévy — Paris 6, INRIA

2. Lundi 12: mise en place des systèmes,
   Premiers pas en Caml

3. Mercredi 14: Caml théorique

4. Lundi 19 10:15 –12:15: TD de Caml

5. (les forts): xxxx

Structure d'un programme

- programme = suite de phrases
- phrase = directive ou définition ou expression suivie de ";;"
- exemple

```
#open "printf" ;;
let x = 2 ;;
let y = x + 3 ;;
let f(x) = x*x ;;
let z = f(2) ;;
printf "%d, %d, %d\n" x y z ;;
```

Exécuter un programme

- En interprété, sous Unix, Mac, PC winXX[/NT]:

```
% camllight
>       Caml Light version 0.74
##open "printf" ;;
let x = 2 ;;
let y = x + 3 ;;
let f(x) = x*x ;;
let z = f(2) ;;
printf "%d, %d, %d\n" x y z ;;#x : int = 2
#y : int = 5
#f : int -> int = <fun>
#z : int = 4
...
#quit ();;      ou ^D
%
```

- On peut éditer le texte en Emacs, BBedit, vi, ed, Word, Office 97. Caml-mode sous emacs (avec mots clé en couleur). Le texte peut être coupé, collé en entrée de l'interpréteur. On peut aussi le sauver dans une fichier, et utiliser l'instruction `include`

Exécuter un programme

- En interprété, sous Unix, Mac, PC winXX[/NT]:

```
cha% camllight
>       Caml Light version 0.74

#include "p1.ml" ;;
x : int = 2
y : int = 5
f : int -> int = <fun>
z : int = 4
2, 5, 4
- : unit = ()
- : unit = ()
...
#quit () ;;   ou ^D
```

- En compilé, sous Unix:

```
% camlc p1.ml
% ./a.out
2, 5, 4
```

En fin du programme on peut aussi mettre:

```
flush (stdout);;
```

Quelques exemples de programmes

```
let rec fact (x) =
   if x = 0 then 1 else x * fact (x-1) ;;

let rec fib (x) =
   if x = 0 || x = 1 then 1 else fib (x-2) + fib (x-1);;

let l = 1 :: 2 :: 3 :: [];;

let rec sort = function
  | [] -> []
  | x :: l -> insert x (sort l)

and insert elem = function
 | [] -> [elem]
 | x :: l ->
     if elem < x then
         elem :: x :: l
     else
         x :: insert elem l;;

sort [2; 1; 0] ;;
sort ["yes"; "ok"; "sure"; "ya"; "yep"] ;;

let rec sigma f = function
 | [] -> 0
 | x :: l -> f x + sigma f l ;;

sigma (function x -> x * x) [1; 2; 3];;
```

Du typique Tronc Commun

```
let tri_selection a =
 let min = ref 0 in
 for i = 0 to vect_length a - 2 do
  min := i;
  for j = i + 1 to vect_length a - 1 do
   if a.(j) < a.(!min) then min := j
  done;
  let t = a.(!min) in
  a.(!min) <- a.(i); a.(i) <- t
 done;;
```

(Remarque syntaxique: if partiel marche ici car la seule alternative min := j est de type unit, qui est aussi le type par défaut du else)

```
let tri_selection a plus_petit_que =
 let min = ref 0 in
 for i = 0 to vect_length a - 2 do
  min := i;
  for j = i + 1 to vect_length a - 1 do
   if plus_petit_que(a.(j),  a.(!min)) then min := j
  done;
  let t = a.(!min) in
  a.(!min) <- a.(i); a.(i) <- t
 done;;
```

Du typique Tronc Commun – bis

```
let tri_insertion a =
 let j = ref 0 in
 for i = 1 to vect_length a - 1 do
   let v = ref a.(i) in
      begin
      j := i;
      while !j > 0 && a.(!j - 1) > !v do
        a.(!j) <- a.(!j - 1);
        decr j
      done;
      a.(!j) <- !v;
      end
 done;;
```

où decr est définir pas

```
let decr (x) =
    x := !x -1 ;;
```

(c'est en fait une fonction de bibliothèque). Itou pur incr.

Du typique Tronc Commun – ter

```
let initialisation (n) =
  let a = make_vect n 0 in
     begin
     for i = 0 to vect_length a - 1 do
        a.(i) <- random__int 100
     done;
     a
     end ;;
```

Majeure 2
Rattrapage Caml
bis

Jean-Jacques Lévy
jean-jacques.levy@inria.fr

21 janvier 1998

Slide 1

Les modules de bibliothéques – int.mli

```
(* Operations on integers *)

(* Integers are 31 bits wide (or 63 bits on 64-bit processors).
   All operations are taken modulo $2^{31}$ (or $2^{63}$).
   They do not fail on overflow. *)

exception Division_by_zero;;

value minus : int -> int = 1 "~int"
  and minus_int : int -> int = 1 "~int"
        (* Unary negation. You can write [-e] instead of [minus e]. *)
  and succ : int -> int = 1 "succ"
        (* [succ x] is [x+1]. *)
  and pred : int -> int = 1 "pred"
        (* [pred x] is [x-1]. *)
  and prefix + : int -> int -> int = 2 "+int"
...;;
```

Slide 2

Les modules de bibliothéques – int.mli (2)

```
value string_of_int : int -> string
        (* Convert the given integer to its decimal
           representation. *)
  and int_of_string : string -> int = 1 "int_of_string"
        (* Convert the given string to an integer,
           in decimal (by default) or in hexadecimal,
           octal or binary if the string begins with
           [0x], [0o] or [0b].
           Raise [Failure "int_of_string"] if the given
           string is not a valid representation of an
           integer. *)
```

Slide 3

Les modules de bibliothéques – string.mli

`s.[n]` est le $n+1$ème caractère de `s`

`s.[n] <- c` positionne ce caractère

Ce qui peut être trouvé dans l'interface `string.mli` .

```
value string_length : string -> int;;
value nth_char : string -> int -> char
  (* [nth_char s n] returns character number [n] in string [s].
     The first character is character number 0.
     The last character is character number [string_length s - 1].
     Raise [Invalid_argument "nth_char"] if [n] is ouside the range
     0 to [(string_length s - 1)].
     You can also write [s.[n]] instead of [nth_char s n]. *)
  and set_nth_char : string -> int -> char -> unit
  (* [set_nth_char s n c] modifies string [s] in place,
     replacing the character number [n] by [c].
     Raise [Invalid_argument "set_nth_char"] if [n] is ouside
     the range 0 to [(string_length s - 1)].
     You can also write [s.[n] <- c] instead of [set_nth_char s n c]. *)
;;
```

Slide 4

Les modules de bibliothéques – string.mli (2)

Et aussi plus succinctement:

```
value prefix ^ : string -> string -> string
  and sub_string : string -> int -> int -> string;;
value create_string : int -> string
  and make_string : int -> char -> string;;
value fill_string : string -> int -> int -> char -> unit
  and blit_string : string -> int -> string -> int -> int -> unit
  and replace_string : string -> string -> int -> unit;;
value eq_string : string -> string -> bool = 2 "=string"
  and neq_string : string -> string -> bool = 2 "<>string"
  and le_string : string -> string -> bool = 2 "<=string"
  and lt_string : string -> string -> bool = 2 "<string"
  and ge_string : string -> string -> bool = 2 ">=string"
  and gt_string : string -> string -> bool = 2 ">string";;
value compare_strings : string -> string -> int = 2 "compare_strings";;
value string_for_read : string -> string;;
```

Slide 5

Entrées/Sorties tamponnées – io.mli

```
type in_channel  and  out_channel;;
exception End_of_file;;
value stdin: in_channel  and  stdout: out_channel;;

value exit : int -> 'a;;

value print_char : char -> unit
  and print_string : string -> unit
  and print_int : int -> unit
  and print_float : float -> unit
  and print_endline : string -> unit
  and print_newline : unit -> unit;;

value read_line : unit -> string
  and read_int : unit -> int
  and read_float : unit -> float;;
```

Slide 6

Entrées/Sorties tamponnées – io.mli (2)

```
value open_out : string -> out_channel
  and flush : out_channel -> unit
  and output_char : out_channel -> char -> unit
  and output_string : out_channel -> string -> unit
  and output : out_channel -> string -> int -> int -> unit
  and seek_out : out_channel -> int -> unit
  and pos_out : out_channel -> int
  and close_out : out_channel -> unit;;

value open_in : string -> in_channel
  and input_char : in_channel -> char
  and input_line : in_channel -> string
  and input : in_channel -> string -> int -> int -> int
  and seek_in : in_channel -> int -> unit
  and pos_in : in_channel -> int
  and close_in : in_channel -> unit;;
```

Slide 7

Où trouver les modules de bibliothèques

En Unix, on peut lire sous Emacs tous les `.mli` en

`/usr/local/lib/caml-light`

Sur Mac ou Pc, dans le répertoire `lib` de la distribution Caml-light.

Sur le Web, on les trouve en:

`http://pauillac.inria.fr/caml/man-caml/`

aussi à l'X en:

Slide 8

Compilation séparée

2 types de noms d'identificateur:

nommage absolu: `graphics__moveto`, `graphics__lineto`, etc

nommage relatif: `moveto`, `lineto` après avoir fait `#open "graphics"`

Le top-level interactif est dans un module `top` implicite.

Pour charger un module, on fait `load "x.ml"`, ou `load_object "x.zo"`.

Les interfaces de module doivent être compilés (format `.zi`). L'interface d'un module est implicitement importé par le nommage absolu (contenant le nom du module) ou par la directive `#open`.

Slide 9

Compilation séparée (2)

Les suffixes `.zo` désignent des fichiers compilés d'implémentation.

La compilation génère des fichiers `.zo` ou `.zi` à partir de `.ml` ou `.mli`. Si pas de `.mli`, un `.zi` est automatiquement généré en faisant un interface standard. La compilation de `.mli`, puis de `.ml` permet de vérifier l'adéquation de l'interface par rapport à l'implémentation.

Majeure 2
Rattrapage Caml
bis

Jean-Jacques Lévy
jean-jacques.levy@inria.fr

14 janvier 1998

Slide 1

Les supports de cours sont en

http://www.polytechnique.fr/poly/edu/informatique/
http://www.polytechnique.fr/poly/~gonthier/
http://www.polytechnique.fr/poly/~levy/
http://www.polytechnique.fr/poly/~weis/
http://pauillac.inria.fr/~levy/x/annexe-caml/node39.html

Slide 2

Introduction

- Caml-light 0.74 est un dialecte de ML $=$ Lisp $+$ types
- Gestion automatique de la mémoire
- Appel de procédures avec filtrage
- Fonctionalité arbitraire
- Modularité, types de données abstraits
- Types polymorphes
- OCaml $=$ Caml orienté objets $+$ compilateur code natif
- Caml-light tourne en peu de mémoire, système à *byte-code*

Slide 3

Les langages

	Pascal	C	Java	Caml
Typage	$\sim$sûr **record case** manuel	indicatif coercions manuel	sûr dynamique manuel	sûr paramétré auto
Pointeurs	explicites	centraux	auto	auto
Objets	non	non	centraux	O'Caml

Slide 4

Syntaxe du langage

- Directives: ouvrir un module, pour désigner les variables sans mettre le nom de leur module.

```
#open "printf" ;;
#open "graphics" ;;
#open "mon_module" ;;
```

- Définitions: une variable est un nom donné à une valeur ou fonction. Définir une variable fixe à la fois sa valeur et son type.

```
let quatre = 2+2;;
let carre n = n*n;;
let triangle n =
   let rectangle = n * (n+1) in
   rectangle/2;;
```

Slide 5

La date de Pâques

Lilius et Clavius en 16xx (Knuth vol1, p.155); soit $Y < 100\ 000$:

1. [Nombre d'or] $G = (Y \bmod 19) + 1$
2. [Siècle] $C = \lfloor Y/100 \rfloor + 1$
3. [Corrections] $X = \lfloor 3C/4 \rfloor - 12$
   $Z = \lfloor (8C + 5)/25 \rfloor - 5$
4. [Un dimanche] $D = \lfloor 5Y/4 \rfloor - X - 10$
5. [Epact.] $E = (11G + 20 + Z - X) \bmod 30,$
   $E \leftarrow E + 1$ si $E = 24$, ou $E = 25$ et $G > 11$
6. [Pleine lune] $N = 44 - E$, mais $N \leftarrow N + 30$ si N ¡ 21
7. [Le jour] $J = N + 7 - ((D + N) \bmod 7)$
8. [Le mois] la date est le J Mars si $J \leq 31$,
   le $J - 31$ Avril sinon.

Slide 6

La date de Pâques en Caml

```
let paques y =
  let g = y mod 19 +1 in
  let c = y / 100 + 1 in
  let x = (3*c) / 4 - 12 and z = (8*c+5) / 25 - 5 in
  let d = (5*y) / 4 - x - 10 in
  let e = let e' = (11*g + 20 + z - x) mod 30 in
          if e'=24 || e'=25 && g>11 then e'+1 else e' in
  let n = let n' = 44-e in if n'<21 then n'+30 else n' in
  n + 7 - (d+n) mod 7;;
let mois j = if j<=31 then "Mars" else "Avril";;
let jour j = if j<=31 then j else j-31;;
#jour (paques 1998);; mois (paques 1998);;
- : int = 12
- : string = "Avril"
```

Slide 7

Les types, valeurs, et primitives

type	valeurs	opérations
`int`	`0,1,2,-2`	`+ - * / mod = <> < > <= >=`
`bool`	`true,false`	`not && || if..then..else`
`float`	`1.0,-3.5,1E3`	`+. -. *. /. ** sin = <> < > <= >=`
`string`	`"la tete a toto"`	`string_length ^ = <> < > <= >=`

Conversions explicites `float_of_int`, `int_of_float`, `string_of_int`, ...

Slide 8

Les fonctions

En Caml, on ne met *pas* de parenthèses autour des arguments.

```
#let norme2 dx dy = dx**2.0 +. dy**2.0;;
norme2 : float -> float -> float = <fun>
```

Mais, il faut parenthéser chaque argument qui est une expression.

```
#let dist x y x' y' = sqrt (norme2 (x-.x') (y-.y'));;
dist : float -> float -> float -> float -> float = <fun>
```

Messages étranges quand on oublie des arguments.

```
#dist 0.0 0.0 1.0;;
- : float -> float = <fun>
```

Slide 9

Pour imprimer

D'abord, ouvrir le module.

```
#open "printf";;
```

L'affichage est contrôlé par un *format*=chaîne à trous.

```
let imprPaques y =
  let j = paques y in
  printf "Le %2d %s %d\n" (jour j) (mois j) y;;
imprPaques 1998;;
Le 12 Avril 1998
- : unit = ()
```

Les trous : `%d` entiers (décimaux), `%s` chaînes, `%g` réels, `%b` booléens, ...

Slide 10

Le contrôle

Séquence, tests, filtrage, boucles, échappements.

```
begin
  instr1;
  instr2;
  ...
  instrN
end
```

```
if cond then instr

if cond then
  instr
else
  instr
```

```
match val with
  0 -> instr1
| 1 -> instr2
...
| n -> instr(n)
```

```
for i = 1 to n do
  instr1;
  instr2;
  ...
done
```

```
while cond do
  instr1;
  instr2;
  ...
done
```

```
try
  while true do
    ... raise Exit
    ...
  done
with Exit -> ...
```

Slide 11

Références

Une référence est un objet qui peut *contenir* une valeur. On les crée en leur donnant un contenu initial. On utilise l'opérateur `!` pour récupérer leur contenu, et `:=` pour le modifier.

```
#let somme = ref 0;;
somme : int ref = ref 0
#for i = 1 to 5 do somme := !somme + i done; somme;;
- : int ref = ref 15
#!somme;;
- : int = 15
```

Par rapport à Pascal ou C, il suffit d'indiquer les références par des `!` :

```
let fact n =
  let prod = ref 1 in
  for i = 1 to n do prod := !prod * i done;
  !prod;;
```

Slide 12

Les vecteurs

Comme les références, les vecteurs sont des *contenants*. Comme pour les chaînes, leur longueur est implicite; on peut l'obtenir avec `vect_length`. Les vecteurs sont toujours indicés à partir de 0.

```
#let produitScalaire v1 v2 =
  let n = min (vect_length v1) (vect_length v2) in
  let p = ref 0.0 in
  for i = 0 to n-1 do p := !p +. v1.(i) *. v2.(i) done;
  !p;;
produitScalaire : float vect -> float vect -> float = <fun>
#produitScalaire [| 1.0; -2.0; 3.0|] [| -1.0; 1.0; 1.0|];;
```

On peut créer des vecteurs de longueur donnée.

```
let iota n =
  let v = make_vect n 0 in
  for i = 1 to n-1 do v.(i) <- i done; v;;
```

Slide 13

Les matrices

On utilise des vecteurs de vecteurs; *Attention!* Utiliser `make_matrix` pour créer une matrice.

```
#let sommeMat m1 m2 =
  let h = vect_length m1 in
  let w = if h > 0 then vect_length m1.(0) else 0 in
  if h <> vect_length m2 || h>0 && h <> vect_length m2.(0) then
    failwith "sommeMat: tailles incompatibles";
  let s = make_matrix h w 0.0 in
  for i = 0 to h-1 do
    for j = 0 to w-1 do
      s.(i).(j) <- m1.(i).(j) +. m2.(i).(j)
    done
  done;
  s;;
sommeMat : float vect vect -> float vect vect -> float vect vect
```

Slide 14

Aliasing

Que donne `make_vect 2 (make_vect 2 0)` ?

```
#let m = make_vect 2 (make_vect 2 0);;
m : int vect vect = [|[|0; 0|]; [|0; 0|]|]
#m.(0).(0) <- 1; m;;
- : int vect vect = [|[|1; 0|]; [|1; 0|]|]
```

Slide 15

Récursivité

En Caml, on peut définir des fonctions récursives

```
#let rec f n = if n<2 then 1 else f(n-1) + f(n-2);;
```

mais

```
#let f x = x + 3 ;;
#let f n = if n<2 then 1 else f(n-1) + f(n-2);;
```

donnera n'importe quoi!

Attention à `rec` !!

Slide 16

Boucles vs. récursion

On peut toujours remplacer une boucle par une récursion

```
let rec f () = if not Cond then begin Instr; f() end;;
```

équivaut à

```
let f () = while Cond do Instr done;;
```

Le système Caml reconnaît cet usage (*récursion terminale*), et l'exécute tout aussi efficacement.

Slide 17

Récursion vs. références

La récursion permet de remplacer les références par un passage explicite de paramètre.

```
let rec total v n = if n=0 then 0.0
                    else v.(n-1) +. total v (n-1);;
```

Les fonctions auxiliaires peuvent être définies localement, comme les autres variables.

```
let somme v =
  let rec ajout i s = if i=0 then s
                      else ajout (i-1) (s +. v.(i-1)) in
  ajout (vect_length v) 0.0;;
```

Règle du jeu : boucles pour les itérations simples, récursion dès que les choses se compliquent.

Slide 18

Tri sélection

```
let triSel t = let n = vect_length t in
  for i = vect_length t - 1 downto 1 do
    let j = ref i in
    for k = 0 to i-1 do
      if t.(k) > t.(!j) then j := k
    done;
    let x = t.(i) in
    t.(i) <- t.(!j); t.(!j) <- x
  done;
  t;;
```

Slide 19

Tri insertion

```
let triIns t =
  for i = 1 to vect_length t - 1 do
    let x = t.(i) and j = ref i in
    while !j>0 && t.(!j-1) > x do
      t.(!j) <- t.(!j-1);
      decr j
    done;
    t.(!j) <- x
  done;
  t;;
```

Slide 20

Tri par fusion

```
let rec triFusion t =
  let n = vect_length t in
  if n > 1 then begin
    let n1 = n/2 in let t1 = sub_vect t 0 n1 in
    let n2 = n-n1 in let t2 = sub_vect t n1 n2 in
    let rec fusion i j =
      if i<n1 && (j>=n2 || t1.(i) < t2.(j)) then
        begin t.(i+j) <- t1.(i); fusion (i+1) j end
      else if j<n2 then
        begin t.(i+j) <- t2.(j); fusion i (j+1) end in
    triFusion t1; triFusion t2; fusion 0 0
  end; t;;
```

Slide 21

Nombres complexes

```
type complex = Complex of float * float;;
let re (Complex(x,_)) = x;;
let im (Complex(_,y)) = y;;
let prefix +: (Complex(x1,y1)) (Complex(x2,y2)) =
  Complex(x1 +. x2, y1 +. y2);;
let prefix -: (Complex(x1,y1)) (Complex(x2,y2)) =
  Complex(x1 -. x2, y1 -. y2);;
let prefix *: (Complex(x1,y1)) (Complex(x2,y2)) =
  Complex(x1 *. x2 -. y1 *. y2,  x1 *.y2 +. x2 *. y1);;
let prefix /: (Complex(x1,y1)) (Complex(x2,y2)) =
  let n2 = x2*.x2 +. y2*.y2 in
  Complex((x1*.x2 +. y1*.y2) /. n2, (x2*.y1 -. x1*.y2) /. n2);;
let omega n =
  let a = 8.0 *. atan 1.0 /. float_of_int n in
  Complex(cos a, sin a);;
```

Slide 22

Types récursifs

Récurrences pour définir l'organisation des données $=$ *structures de données*.

Par exemple les suites d'entiers $[a_1 \cdot a_2 \cdots a_n]$ peuvent être décrites ainsi; une telle suite σ est

- soit la suite vide $[]$
- soit constituée d'un premier élément a_1, et d'une sous-suite $[a_2 \cdots a_n]$.

En Caml, on peut exprimer cette récurrence directement

```
type suiteInt = Vide | Cons of int * suiteInt;;
```

On utilise le *filtrage* pour discriminer les cas et accéder aux éléments.

Slide 23

Listes Caml

Les listes sont tellement utiles qu'en Caml elles ont un type prédéfini générique `'a list`, comme les tableaux ont `'a vect`.

Elles ont une syntaxe spécifique, utilisée pour la création et le filtrage :

- `[]` est la liste vide.
- a`::`l est la liste dont le premier élément est a, et le reste est l.
- `[`a_1`;` a_2`;` `...;` a_n`]` est la liste $[a_1 \cdot a_2 \cdots a_n]$
- On a `[a;b;c]` $\equiv$ `a::[b; c]`
  $\equiv$ `a::b::[c]`
  $\equiv$ `a::b::c::[]`

Slide 24

Récurrence sur les listes

A type récursif, fonction récursives!

$$\text{longueur } \sigma = |\sigma| = \begin{cases} 0 & \text{si } \sigma = [] \\ 1 + |\sigma'| & \text{si } \sigma = a_1 \cdot \sigma' \end{cases}$$

Ce qui se traduit *littéralement* en Caml :

```
let rec longueur l = match l with
    []    -> 0
| _::l' -> 1 + longueur l';;
```

Slide 25

Quelques exemples

La bibliothèque de base de Caml comprend un large assortiment de fonctions sur les listes, et particulièrement des *fonctionnelles*.

```
let rec prefix @ l1 l2 = match l1 with
  [] -> l2
| a::l1' -> a::(l1' @ l2);;
let rev l =
  let rec revapp l1 l2 = match l1 with
    [] -> l2 | a::l1' -> revapp l1' (a::l2) in
  revapp l [];;
let rec map f = function
  [] -> []
| a::l -> f a :: map f l;;
let rec exists p =
   function [] -> false | a::l -> p a || exists p l;;
let mem x l = exists (fun y -> x=y) l;;
```

Slide 26

Polynômes

On représente un polynôme une liste triée de monômes :

```
type 'a monome = Mono of 'a * int;;
type 'a polynome = Poly of 'a monome list;;
```

Par exemple

```
#let poly_of_list l = Poly (map Mono l);;
poly_of_list : ('a * int) list -> 'a polynome = <fun>
#let p = poly_of_list [1.0,5; -2.0,3; 4.0,2; 3.0,0];;
p : float polynome =
 Poly [Mono (1.0, 5); Mono (-2.0, 3);
       Mono (4.0, 2); Mono (3.0, 0)]
```

Slide 27

Addition de polynômes

On suit le modèle de la procédure de fusion.

```
let prefix +$ (Poly p1) (Poly p2) =
  let rec add = function
    [], l2 -> l2
  | l1, [] -> l1
  | (Mono(a1,n1)::l1' as l1),
    (Mono(a2,n2)::l2' as l2) ->
           if n1 > n2 then hd l1 :: add (l1', l2)
      else if n1 < n2 then hd l2 :: add (l1, l2')
      else Mono(a1+.a2, n1) :: add (l1', l2') in
  Poly (add (p1, p2));;
+$ : float polynome -> float polynome -> float polynome
```

Slide 28

Listes associatives

Ce sont des listes dont les éléments servent à ranger une valeur sous une *clé* permettant de les retrouver.

```
let rec assoc k = function
  [] -> raise Not_found
| (k',e)::l -> if k'=k then e else assoc k l;;
let mem_assoc k l = exists (fun (k',_) -> k=k') l;;
```

Ces deux fonctions font partie de la bibliothèque Caml.

```
let pages_zoom nom annuaire =
  try
    assoc nom annuaire
  with Not_found ->
    "Il n'y a pas d'abonne " ^ nom;;
```

Slide 29

Types enregistrement

C'est le pendant en Caml des `record` de Pascal ou des `struct` de C.

- Déclaration, utilisation :

```
type point = {x:int; y:int};;
let translate p dx dy = {x=p.x+dx; y=p.y+dy};;
```

  Attention!! les noms de champs (`x`, `y` ici) ne peuvent être partagés entre plusieurs types enregistrement différents.

- On peut aussi *modifier* certains champs:

```
type pixel = {x:int; y:int;
              mutable couleur: graphics__color};;
let noircir p = p.couleur <- graphics__black;;
  (* p.x <- 0 n'est pas permis *)
```

Slide 30

Listes de collision

On hache dans un tableau de a-listes; complexité moyenne $O(\alpha)$, où $\alpha = n/m$, avec un cas pire moyen en $O(\log n / \log \log n)$.

```
type ('a,'b) hashtable = {
  mutable hash_buckets: ('a * 'b) list vect;
  mutable hash_n: int;
  hash_fun: int -> 'a -> int};;
let make_hash f m = {
  hash_buckets = make_vect m [];
  hash_n = 0; hash_fun = f};;
let slot k t =
  t.hash_fun (vect_length t.hash_buckets) k;;
let lookup k t =
  assoc k t.hash_buckets.(slot k t);;
```

Il y a des tables de hachage génériques dans les bibliothèques Caml.

Slide 31

Insertion

```
let hash_put t (k,_ as p) =
  let i = slot k t in
  t.hash_buckets.(i) <- p::t.hash_buckets.(i);;
let overfull t =
  t.hash_n > 10 * vect_length t.hash_buckets;;
let rehash t =
  let b = t.hash_buckets in
  t.hash_buckets <- make_vect (next_prime (2*t.hash_n)) [];
  do_vect (do_list (hash_put t)) b;;
let insert k e t =
  t.hash_n <- t.hash_n + 1;
  if overfull t then rehash t;
  hash_put t (k,e);;
```

Slide 32

Listes modifiables

L'utilisation des a-listes gaspille de la mémoire :

- Le système utilise *deux* blocs de mémoire pour chaque élément (un pour la cellule, un pour la paire).
- Lorsque l'on change une entrée, l'ancienne valeur reste!

Solution : gérer directement les cellules.

```
type annu = {nom: string; mutable tel: string;
              mutable next: annu};;
```

On utilise une valeur particulière pour la liste vide.

```
let rec null = {nom = ""; tel = ""; next = null};;
```

Attention aux effets de bord!!

Slide 33

Listes modifiables

Modification (gare aux effets de bord!) :

```
let rec modif nom tel a =
  if a == null then {nom = nom; tel = tel; next = a}
  else begin
    if a.nom = nom then a.tel <- tel
    else a.next <- modif nom tel a.next;
  a end;;
```

Suppression :

```
let rec retire nom a =
  if a == null then a
  else if a.nom = nom then a.next
  else begin
    a.next <- retire nom a.next;
  a end;;
```

Slide 34

Versions itératives

```
let modif nom tel a =
  let a' = ref a in
  while !a' != null && !a'.nom <> nom do
    a' := !a'.next
  done;
  if !a' == null then {nom = nom; tel = tel; next = a}
  else begin !a'.tel <- tel; a end;;

let retire nom a =
  let a' = ref null and a'' = ref a in
  while !a'' != null && !a''.nom <> nom do
    a' := !a''; a'' := !a''.next
  done;
  if !a'' == null then a
  else if !a' == null then a.next
  else begin !a'.next <- !a''.next; a end;;
```

Slide 35

Retournement

```
let retourne a =
  let rec ret_app a1 a2 =
    if a1==null then a2 else begin
      let a1' = a1.next in
      a1.next <- a2;
      ret_app a1' a1
    end in
  ret_app a null;;
```

En itératif

```
let retourne a =
  let a' = ref a and r = ref null in
  while !a' != null do
    let a'' = !a'.next in
    !a'.next <- !r; r := !a'; a' := a''
  done; !r;;
```

Slide 36

Listes doublement chaînées

Pointer sur la cellule précédente permet de supprimer une cellule en temps constant. Il est aussi commode d'avoir un chaînage circulaire et une cellule d'entête.

```
type 'a list2c = {val: 'a;
  mutable prev: 'a list2c; mutable next: 'a list2c};;
let make_void v =
  let rec r = {val=v; prev=r; next=r} in r;;
let insert v l =
  let c = {val=v; prev=l; next=l.next} in
  l.next <- c; c.next.prev <- c;;
let delete c =
  c.next.prev <- c.prev; c.prev.next <- c.next;;
let is_single c = c.prev==c.next;;
let make_single v = insert v (make_void v);;
```

Slide 37

Arbres binaires

Comme pour les listes:

```
type 'a arbre = ArbreVide
              | Noeud of 'a arbre * 'a * 'a arbre;;

let rec flat = function
  ArbreVide -> []
| Noeud(g,x,d) -> flat g @ [x] @ flat d;;
```

Slide 38

Recherche dans les arbres

La dichotomie se fait toute seule!

```
let rec cherche y = function
  ArbreVide -> false
| Noeud (g, x, d) ->
    x=y || cherche y (if y<x then g else d);;
let rec ajout y a = match a with
  ArbreVide -> Noeud (ArbreVide, y, ArbreVide)
| Noeud (g, x, d) ->
    if y<x      then Noeud (ajout y g, x, d)
    else if x<y then Noeud (g, x, ajout y d)
    else a;;
```

Slide 39

Arbres modifiables

Comme pour les listes, on peut utiliser des enregistrements pour représenter les nœuds; on peut panacher avec le type concret pour simplifier la représentation de l'arbre vide. Les mutables permettent de faire l'ajout “en place” :

```
type 'a arbre =  ArbreVide | Noeud of 'a noeud
and  'a noeud =
  {val: 'a; mutable f_g: 'a arbre; mutable f_d: 'a arbre;};;
let ajout y a =
  let a_y = Noeud {val=y; f_g=ArbreVide; f_d=ArbreVide} in
  let rec ins_y n =
    if y < n.val then match n.f_g with
      Noeud n' -> ins_y n' | _ -> n.f_g <- a_y
    else if y > n.val then match n.f_d with
      Noeud n' -> ins_y n' | _ -> n.f_d <- a_y in
  match a with Noeud n -> ins_y n; a | _ -> a_y;;
```

Slide 40

Rotations

```
let rot y = function
  Noeud(Noeud(g, x1, m), x2, d) when y < x2 ->
    Noeud(g, x1, Noeud(m, x2, d))
| Noeud(g, x1, Noeud(m, x2, d)) when x1 < y ->
    Noeud(Noeud(g, x1, m), x2, d)
| a -> a;;
```

Slide 41

Quelques erreurs. . .

- On ne peut utiliser que des *constantes* dans un filtre de `match`. Pour comparer avec des paramètres, il faut utiliser '`=`' et `if` ou `when`.
- Evitez de mélanger fonctions récursives et références. En général, la récursion est plus simple. Il est facile de retourner plusieurs valeurs en Caml.
- Evitez les booléens arbitraires. Utiliser `raise Exit` pour sortir d'une boucle, et des fonctions différentes (éventuellement mutuellement récursives).

Slide 42

Modules

- Pour limiter la taille d'un programme ou en isoler des parties critiques, on le divise en *modules*.
- En Caml, on écrit un fichier d'interface, `mod.mli`, en plus de l'implémentation `mod.ml`.
- On utilise les "objets" `x` (fonctions, types, étiquettes) d'un module `mod` soit comme `mod__x`, soit après avoir fait `#open "mod"`.
- La bibliothèque Caml est organisée en modules.
- Il faut "charger" vos modules persos en appelant `load "mod"`

Slide 43

Fichier d'interface `.mli`

On y met des déclarations de types, exceptions, et de *valeurs* (des fonctions le plus souvent).

```
type color == int;;
value black : color;;
value point_color : int -> int -> color;;
value set_color : color -> unit;;
value lineto : int -> int -> unit;;
```

Les déclarations de types peuvent être *abstraites* (pas de représentation). Dans ce cas on ne pourra opérer sur les valeurs de ce type qu'avec les fonctions du module.

Slide 44

Piles

C'est le plus simple des “contenants”; on ne peut lire et retirer que la dernière valeur insérée.

```
type 'a stack;;
exception Empty;;
value new : int -> 'a stack;;
value empty : 'a stack -> bool;;
value push : 'a -> 'a stack -> unit;;
value top : 'a stack -> 'a;;
value pop : 'a stack -> 'a;;
```

Peu utilisées en Caml, à cause de la récursion. Mais très utile pour la conception d'algorithme.

Slide 45

Implémentation des piles (1)

Le plus efficace : des tableaux.

```
type 'a stack =
  {mutable i: int; m: int; mutable t:'a vect};;
let new m = {i = -1; m = m; t = [||]};;
let empty s = s.i < 0;;
let push x s =
   s.i <- s.i+1;
   if s.i < vect_length s.t then s.t.(s.i) <- x
   else s.t <- concat_vect s.t (make_vect s.m x);;
let top s = if s.i >= 0 then s.t.(s.i) else raise Empty;;
let pop s = let x = top s in s.i <- s.i-1; x;;
```

Slide 46

Implémentation des piles (2)

Le plus simple : des listes.

```
type 'a stack == 'a list ref;;
let new m = ref [];;
let empty s = !s = [];;
let push x s = s := x :: !s;;
let top s = match !s with x::_ -> x
                        | [] -> raise Empty;;
let pop s = match !s with x::r -> s := r; x
                        | [] -> raise Empty;;
```

La différence entre piles et listes relève de l'usage.

Slide 47

Files

Semblable aux piles, mais on lit et on récupère le *premier* élément inséré.

```
type 'a queue;;
exception Empty;;
value new : int -> 'a stack;;
value empty : 'a queue -> bool;;
value add : 'a -> 'a queue -> unit;;
value peek : 'a queue -> 'a;;
value take : 'a queue -> 'a;;
```

Permettent par exemple le parcours d'un arbre par niveau.

Slide 48

Implémentation des files (1)

Tableaux circulaires.

```
type 'a queue = {mutable i: int; mutable j: int;
                 m: int; mutable t:'a vect};;
let new m = {i = 0; j = 0; m = m; t = [||]};;
let empty q = q.i = q.j;;
let add x ({i=i; t=t} as q) = let n = vect_length t in
  q.j <- (q.j+1) mod n;
  if q.j <> i then t.(q.j) <- x else begin
    q.t <- concat_vect (sub_vect t i (n-i)) (sub_vect t 0 i);
    q.t <- concat_vect q.t (make_vect q.m x);
    q.i <- 0; q.j <- n
  end;;
let take q = if q.i = q.j then raise Empty;
             let n = vect_length q.t and x = q.t.(q.i) in
             q.i <- (q.i+1) mod n; x;;
```

Slide 49

Implémentation des files (2)

```
type 'a queue =
  {mutable l1: 'a list; mutable l2: 'a list};;
let new m = {l1 = []; l2 = []};;
let empty q = q.l1 = [] && q.l2 = [];;
let add x q = q.l1 := x::q.l1;;
let take q = if q.l2 = [] then begin
                q.l2 <- rev q.l1; q.l1 <- []
              end;
              match q.l2 with
                x::l2' -> q.l2 <- l2'; x
              | [] -> raise Empty
```

On peut aussi utiliser des listes modifiables circulaires.

Majeure 2
Langages et Compilation
Cours 1

Grammaires formelles

`jean-jacques.levy@inria.fr`

21 janvier 1998

Slide 1

Les supports de cours sont en

`http://www.polytechnique.fr/poly/edu/informatique/`
`http://www.polytechnique.fr/poly/~levy/`

Livres:

- Kozen, Automata and Computability, Springer 1997.
- Hopcroft-Ullman, Formal languages and their Relation to Automata, Addison Wesley, 1969
- Aho-Ullman. The Theory of Parsing, Translation, and Compiling, volume 1 et 2: Parsing. Prentice-Hall, 1972.

Slide 2

Introduction

- Langages
- Grammaires
- Classification de Chomsky

Slide 3

Langages

Soit Σ un alphabet (fini) donné,

un mot est une chaîne de caractères de Σ,

le mot vide ϵ a une longueur nulle,

Σ^* est l'ensemble des mots sur Σ,

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

On appelle *langage* tout sous-ensemble L de Σ^*.

uv est le mot obtenu en concaténant u et v,

$u\epsilon = \epsilon u = u$ et $(uv)w = u(vw)$.

Slide 4

Exemples de langages

Prenons $\Sigma = \{a, b\}$ et notons w^R l'image miroir de w:

$$L_1 = \{a^n b^n \mid n \geq 0\}$$
$$L_2 = \{ww^R\}$$
$$L_3 = \{w \mid w = w^R\}$$

Slide 5

Algorithmes et semi-algorithmes

- Un algorithme est un programme qui termine toujours
- Un semi-algorithme peut ne pas terminer
- Un langage L est récursif s'il existe un algorithme qui re'pond a' la question $u \in L$ par oui ou par non.
- Un langage L est récursivement énumérable s'il existe un semi-algorithme qui répond par oui à la question $u \in L$ si $u \in L$, mais qui répond non ou ne termine pas si $u \notin L$.

Attention: ici récursif est dans le sens de Kleene, donc différent de celui de la programmation. On dit aussi que l'appartenance à un langage récursif est décidable.

Slide 6

Grammaire

On cherche une description finie d'un langage (éventuellement infini). Chomsky a inventé la notion de grammaire formelle.

Une grammaire G est un quadruplet $(\Sigma, V_N, S, \mathcal{P})$ où:

Σ est l'alphabet des terminaux (ou *tokens*),
V_N est l'alphabet des non-terminaux, $V_N \cap \Sigma = \emptyset$,
$S \in V_N$ est l'axiome,
$\mathcal{P}$ est un ensemble fini de règles de production de la forme $\alpha \to \beta$ avec $\alpha \in V^+$, $\beta \in V^*$, $V = \Sigma \cup V_N$

Slide 7

Dérivations

Une grammaire génère les mots d'un langage en dérivant l'axiome.

Si $\alpha \to \beta$ est une règle de production, on peut dériver le mot $x\alpha y$ en $x\beta y$. On écrit $x\alpha y \Rightarrow x\beta y$.

On pose $\alpha \Rightarrow^* \beta$ si $\alpha = \alpha_0 \Rightarrow \alpha_1 \cdots \Rightarrow \alpha_n = \beta$, où $n \geq 0$ (fermeture transitive de $\Rightarrow$).

Le langage généré par G est défini par
$L(G) = \{w \mid w \in \Sigma^*,\ S \Rightarrow^* w\}$

C'est donc l'ensemble des mots de l'alphabet terminal que l'on peut dériver depuis l'axiome.

Slide 8

Exemples de grammaire

$\Sigma = \{a, b\}$, $V_N = \{S\}$

$S \to aSb$

$S \to ab$

$L(G_1) = \ldots$ context-free

$S \to aSb$

$S \to ab$

$S \to SS$

$L(G_2) = \ldots$ context-free

$\Sigma = \{a, b, c\}$, $V_N = \{S, B, C\}$

$S \to aSBC$

$S \to aBC$

$CB \to BC$

$aB \to ab$

$bB \to bb$

$bC \to bc$

$cC \to cc$

$L(G_3) = \ldots$ context-sensitive

Slide 9

Classification de Chomsky

4 types de grammaires selon la forme des règles de production.

type 0: quelconque

type 1: $\alpha \to \beta$ avec $|\alpha| \leq |\beta|$ (context sensitive)

type 2: $A \to \alpha, \quad \alpha \neq \epsilon$ (context free – algébrique)

type 3: $A \to aB$ (régulière – rationnelle)

$A \to a$

avec $A, B \in V_N$, $a \in \Sigma$, $\alpha, \beta \in V$

Un langage est de type i s'il existe une grammaire de type i qui le génère: langages réguliers, context-free, ou context-sensitifs.

Slide 10

Autres exemples de grammaires

$\Sigma = \{a, b\}, V_N = \{S, A, B\}$	$\Sigma = \{a, b\}, V_N = \{S, A, B\}$
$S \to aB$	$S \to aA$
$S \to bA$	$S \to bB$
$A \to a$	$A \to aA$
$A \to aS$	$A \to aS$
$A \to bAA$	$A \to bB$
$B \to b$	$B \to bB$
$B \to bS$	$B \to b$
$B \to aBB$	$B \to a$
$L(G_4) = \ldots$ context-free	$S \to a$
	$L(G_5) = \ldots$ régulière

Slide 11

La chaîne vide

Seuls les langages de type 0 contiennent le mot vide ϵ. Pourtant, on voudrait que $L \cup \{\epsilon\}$ soit de type i, si L est de type i.

On change la classification de Chomsky en autorisant la règle $S \to \epsilon$ à chaque type de grammaire, pourvu que S ne soit pas utilisé dans les parties droites des règles de production. La règle $S \to \epsilon$ ne peut donc être utilisée que pour ajouter ϵ dans le langage généré.

Lemme 1 Si G est de type i, il existe G_1 de type i telle que $L(G) = L(G_1)$ et G_1 n'utilise pas son axiome dans les parties droites de ses règles.

La démonstration est laissée en exercice.

Slide 12

La chaîne vide (2) — Exercices

Théorème 1 Si L est de type i, alors $L \cup \{\epsilon\}$ et $L - \{\epsilon\}$ sont de type i.

Exercices

On note $\#_a(x)$ le nombre de a que contient le mot x. Trouver des grammaires générants les langages suivants:

$L_6 = \{w \mid \#_a(w) = \#_b(w) = \#_c(w)\}$
$L_7 = \{w \mid \#_a(w) = 2 \times \#_b(w)\}$
$L_8 = \{w \mid w$ n'a jamais $2b$ consécutifs $\}$
$L_9 = \{ww \mid w \in \Sigma^*\}$

Slide 13

Exercices (2)

Montrer que les langages réguliers sont aussi engendrés par des grammaires dont les productions sont de la forme:

$A \to xB$

$A \to x$

où $x \in \Sigma^*$

Montrer qu'il existe un algorithme pour reconnaître tout langage context-sensitif.

Montrer qu'il existe un semi-algorithme pour reconnaître tout langage de type 0.

Majeure 2
Langages et Compilation
Cours 2

Automates finis

`jean-jacques.levy@inria.fr`

26 janvier 1998

Slide 1

Les supports de cours sont en

`http://www.polytechnique.fr/poly/edu/informatique/`

`http://www.polytechnique.fr/poly/~levy/`

Livres:

- Kozen, Automata and Computability, Springer 1997.
- Hopcroft-Ullman, Formal languages and their Relation to Automata, Addison Wesley, 1969
- Aho-Ullman. The Theory of Parsing, Translation, and Compiling, volume 1 et 2: Parsing. Prentice-Hall, 1972.
- Aho, Sethi, Ullman, Compilers: Principle, Techniques, and Tools, Addison Wesley, 1986

Slide 2

Introduction

- Quelques exemples
- Automates finis déterministes
- Automates non-déterministes
- Automates minimaux
- Automates finis et langages réguliers
- Propriétés de fermeture
- Problèmes décidables
- Expressions régulières
- Automates en ML

Slide 3

Quelques exemples

- Mécanisme d'un distributeur automatique
- Protocoles réseau (bit alterné, etc)
- Peloton d'exécution (*Firing squad problem*)
- Systèmes réactifs
- Contrôle dans les circuits
- Analyseurs lexicaux

Slide 4

Distributeur à boissons

Il y a une fente pour les pièces de 1F, 2F, 5F, 10F. Le café sort si on met 5F. On peut aussi faire une machine avec buffer de longueur 1 ou 2, qui permet de mettre directement 10F pour avoir 2 cafés.

Slide 5

Protocoles réseau

Slide 6

Peloton d'exécution

Il y a une forte brûme sur le plateau. Le général convoque 400 soldats, les aligne, et veut les faire tirer, tous en même temps. A tout moment, le général et les soldats changent d'état de manière synchrone en ne tenant compte que de leurs 2 voisins immédiats. Il y a donc 3 types de machines: le général, les soldats et le soldat du bout de la chaîne.

Montrer qu'on peut trouver une solution en moins de 8 états pour les 3 types de machines quelquesoit le nombre de soldats.

Slide 7

Autre exemple

Automate fini acceptant tous les mots contenant un nombre pair de a et de b.

Slide 8

Automate fini

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

Σ est un alphabet (de terminaux ou *tokens*),
Q est un ensemble *fini* d'états,
$q_0 \in Q$ est l'état initial,
$F \subset Q$ est l'ensemble des états finaux,
$\delta : Q \times \Sigma \to Q$ est la fonction de transition

On peut étendre δ sur $Q \times \Sigma^* \to Q$ comme suit:

$\delta(q, \epsilon) = q$
$\delta(q, aw) = \delta(\delta(q, a), w)$

Le langage reconnu par l'automate M est:

$T(M) = \{w \mid \delta(q_0, w) \in F\}$

Slide 9

Graphiquement

Les mots de Σ à analyser sont sur une bande (non modifiable). L'automate est un lecteur de bande. Initialement la tête de lecture est sur la première lettre du mot, l'automate est dans l'état q_0. Puis la tête de bande de déplace d'un cran vers la droite en changeant d'état en fonction du caractère lu, et ainsi de suite jusqu'à la fin du mot. Le mot est accepté si sa fin est obtenue dans un état de F.

Slide 10

Représentation des automates finis

Quatre méthodes:

- le contrôle fini d'un programme. Autant de points d'entrée (étiquettes ou procédures) que d'états:

```
let c = ref( input_char() ) in
let rec
    q() = if !c = 'a' then begin c := get(); r() end else
          if !c = 'b' then begin c := get(); t() end
and r() = if !c = 'a' then begin c := get(); q() end else
          if !c = 'b' then begin c := get(); s() end
and s() = if !c = 'a' then begin c := get(); t() end else
          if !c = 'b' then begin c := get(); r() end
```

Slide 11

```
and t() = if !c = `a` then begin c := get(); s() end else
          if !c = `b` then begin c := get(); q() end ;;
```

ou

Slide 12

Représentation des automates finis (2)

```
let m() =
  let rec m1 q c = match q with
        0 -> if c = `a` then let c = get() in m1 1 c else
             if c = `b` then let c = get() in m1 3 c else
             if c = eof then exit(0)
      | 1 -> if c = `a` then let c = get() in m1 0 c else
             if c = `b` then let c = get() in m1 2 c
      | 2 -> if c = `a` then let c = get() in m1 3 c else
             if c = `b` then let c = get() in m1 1 c
      | 3 -> if c = `a` then let c = get() in m1 2 c else
             if c = `b` then let c = get() in m1 0 c
      | _ -> ()
```

Slide 13

```
  in let c = get() in  m1 0 c ;;
```

ou encore:

Slide 14

Représentation des automates finis (3)

```
let m() =
  let rec m1 q c = if c = eof then exit (q) else
      let c' = get() in match q with
        0 -> if c = 'a' then m1 1 c' else if c = 'b' then m1 3 c'
      | 1 -> if c = 'a' then m1 0 c' else if c = 'b' then m1 2 c'
      | 2 -> if c = 'a' then m1 3 c' else if c = 'b' then m1 1 c'
      | 3 -> if c = 'a' then m1 2 c' else if c = 'b' then m1 0 c'
      | _ -> ()
  in let c = get() in  m1 0 c ;;
```

ou encore encore:

Slide 15

Représentation des automates finis (3)

```
let m() =
  let rec m1 q c = if c = eof then exit (q) else
      let c' = get() in match q, c  with
        0, 'a' -> m1 1 c'
      | 0, 'b' -> m1 3 c'
      | 1, 'a' -> m1 0 c'
      | 1, 'b' -> m1 2 c'
      | 2, 'a' -> m1 3 c'
      | 2, 'b' -> m1 1 c'
      | 3, 'a' -> m1 2 c'
      | 3, 'b' -> m1 0 c'
      | _ -> ()
```

Slide 16

```
  in let c = get() in  m1 0 c ;;
```

Slide 17

Représentation des automates finis (4)

- matrice $|Q| \times |\Sigma|$ de transition (éventuellement compressée).

```
let mat =  make_matrix 10 10 0;;


let m() =
  let rec m1 q c =
      if c = eof then exit (q) else
      let c' = get() in
      let q' = mat.(q).(int_of_char c) in
      m1 q' c'
  in  let c = get() in  m1 0 c ;;
```

- listes d'association, ie listes de paires (q, a), pour chaque état.

Slide 18

Automates non déterministes

A chaque état, l'automate peut faire un choix non déterministe entre plusieurs transitions. Formellement, $\delta : Q \times \Sigma \to 2^Q$ et on étend δ comme suit

$$\delta(q, \epsilon) = \{q\} \qquad \delta(q, aw) = \bigcup_{q' \in \delta(q,a)} \delta(q', w)$$

Le langage reconnu par l'automate non déterminite M est:

$T(M) = \{w \mid \delta(q_0, w) \cap F \neq \emptyset\}$

Un mot est accepté si une série de choix peut amener l'automate dans un des états finaux de F. Les mauvais choix ne comptent donc pas.

Slide 19

Exemples non déterministes

- $L = \{xaay \mid x, y \in \Sigma^*\} \cup \{xbby \mid x, y \in \Sigma^*\}$
- $L = L_1 \cup L_2$ où L_1 et L_2 sont reconnus par des automates finis

Slide 20

Déterminisation des automates finis

Théorème 1 [Rabin-Scott] Tout langage reconnu par un automate fini non déterministe peut aussi être reconnu par un automate fini déterministe.

Démonstration: on considère l'automate déterministe défini sur 2^Q, en prenant $\{q_0\}$ comme état initial et F comme ensemble de fin.

Remarque L'automate déterminisé peut avoir 2^n états, si n est le nombre d'états de l'automate non-déterministe.

Slide 21

Automates finis et langages réguliers

Théorème 2 Les langages réguliers sont les langages reconnus par les automates finis.

Démonstration: Soit $G = (\Sigma, V_N, S, \mathcal{P})$ une grammaire régulière. Construisons M tel que $L(G) = T(M)$. Posons
$M = (Q, \Sigma, \delta, S, \{T\})$, où $T \notin V_N$ et $Q = V_N \cup T$ avec
$\delta(A, a) = B$ ssi $A \to aB$
$\delta(A, a) = T$ ssi $A \to a$

La réciproque est facile.

Slide 22

Automate minimal

Tout automate $M = (Q, \Sigma, \delta, S, \{T\})$ définit sur Σ^* la relation d'équivalence: $x\mathcal{R}y$ ssi $\delta(q_0, x) = \delta(q_0, y)$.

Appelons index de $\mathcal{R}$ son nombre de classes d'équivalences. L'index de $\mathcal{R}$ n'est donc pas plus grand que le nombre d'états de Q.

En outre, $x\mathcal{R}y$ implique $xz\mathcal{R}yz$ pour tout z. La relation est fermée par facteur droit.

Lemme 3 Les 3 clauses suivantes sont équivalentes:

1. L est reconnu par un automate fini.
2. L est l'union de classes d'équivalence d'une relation d'équivalence fermée par facteur droit d'index fini.
3. La relation $\mathcal{R}$, définie par $x\mathcal{R}y$ ssi pour tout z, on a $xz \in L \Leftrightarrow yz \in L$, est d'index fini.

Slide 23

Automate minimal (2)

Théorème 4 [Myhill - Nerode] L'automate minimal reconnaissant un langage L est unique à un isomorphisme près sur le nom des états.

De manière non constructive, nous avons démontré qu'il existe un automate minimal. Reste à trouver un algorithme efficace pour le trouver.

De la même manière, on peut déterminiser un automate et construire son automate minimal, ou essayer d'obtenir plus directement l'automate fini correspondant.

Slide 24

Construction de l'automate minimal

Soit $M = (Q, \Sigma, \delta, q_0, F)$

On considère la partition $\Pi = \{F,\ Q - F\}$

On itère la procédure suivante tant que Π n'est pas stationnaire.

Pour tout $E \in \Pi$, on casse E en E_1, E_2, etc tels que, pour tous $q, q' \in E_i$, pour tout $E' \in \Pi$, et pour tout $a \in \Sigma$, on a $\delta(q, a) \in E'$ ssi $\delta(q', a) \in E'$.

A la fin, on élimine les états inaccessible depuis l'état initial.

Slide 25

Propriété de fermetures des langages réguliers

Les langages réguliers forment une algèbre de Boole (fermés par union, intersection et complément).

Si L est régulier, L^* est régulier.

On peut définir le produit de 2 langages L et L', par extension naturelle sur la concaténation des mots:
$LL' = \{uu' \mid u \in L, u' \in L'\}$.

Si L et L' sont réguliers, LL' est régulier.

Théorème 5 Les langages réguliers sont la plus petite classe de langages contenant les ensembles finis et fermée par produit, étoile et union.

Slide 26

Expressions régulières

Une expression régulière e représente un langage $[\![e]\!]$

$[\![a]\!] = \{a\}$
$[\![e + e']\!] = [\![e]\!] \cup [\![e']\!]$
$[\![ee']\!] = [\![e]\!][\![e']\!]$
$[\![e^*]\!] = [\![e]\!]^*$
$[\![(e)]\!] = [\![e]\!]$

Parfois, on écrit $e \mid e'$ au lieu de $e + e'$.

Exemples

$(a + b)^*abb$ est l'ensemble des mots sur a et b finissant par abb
$(a + b)^*w(a + b)^*$ est l'ensemble des mots contenant w.

Slide 27

Un peu d'algèbre

Ajoutons 0 et 1 en posant $[\![0]\!] = \emptyset$ et $[\![1]\!] = \{\epsilon\}$.

Les lois suivantes sont vérifiées:

1	$e + e' = e' + e$
2	$e + (e' + e'') = (e + e') + e''$
3	$e + 0 = e$
4	$e + e = e$
5	$e(e'e'') = (ee')e''$
6	$1e = e1 = e$
7	$e(e' + e'') = ee' + ee''$
8	$(e + e')e'' = ee'' + e'e''$
9	$0e = e0 = 0$
10	$1 + ee^* = e^*$
11	$1 + e^*e = e^*$

Slide 28

Un peu d'algèbre (2)

Et en notant $e \leq e'$ quand $e + e' = e'$, ie $[\![e]\!] \subset [\![e']\!]$, on a aussi:

12	$e' + ee'' \leq e'' \Rightarrow e^*e' \leq e''$
13	$e' + e''e \leq e'' \Rightarrow e'e^* \leq e''$

On peut montrer que les 13 axiomes ou règles précédentes permettent de déduire toutes les équations sur les expressions régulières (théorie équationnelle complète, cf DEA SPP pour plus de détails).

Montrer, par exemple, que $(ee')^*e = e(e'e)^*$, ou $ee^* = e^*e$, ou $(e^*e')^*e^* = (e + e')^*$, etc.

Slide 29

Expressions régulières et langages réguliers

Proposition 6 Expressions régulières et langages réguliers coincident.

Décidabilité sur les automates finis Il existe des algorithmes pour décider de pratiquement toutes les propriétés sur les automates finis. Par exemple:

$T(M)$ est-il vide?

$T(M)$ est-il infini?

$T(M) = T(M')$?

Slide 30

Autres formes d'automates

On ne change pas la force de reconnaissance des automates finis si on considère les automates avec des:

- transitions incomplètes. Si un état q ne propose pas de transitions pour toutes les lettres de l'alphabet, l'automate peut se bloquer dans ce cas, et le mot est refusé.
- transitions vides. Si une transition vide relie q à q', l'automate peut faire les transitions issues de q et de q', car il a le droit d'aller spontanément vers q'.
- transitions vides par défaut. Si transition vide par défaut relie q à q', l'automate (incomplet) peut faire les transitions issues de q, mais aussi celles de q' ssi une lettre ne figure pas dans les transitions possibles à partir de q.

Slide 31

Expressions régulières et Unix

Les commandes *shell* d'Unix autorisent les expressions régulières. D'autres commandes comme

`/bin/sh` (Bourne) pour les noms de fichiers,

`sed, ed` (Thomson) Stream editor, Editor

`grep` Get Regular ExPression

Emacs `ESC-x re-search-forward`

`awk` Interpréteur C (Aho, Weinberger, Kernighan)

`lex` (Lesk) Méta-compilateur d'expressions régulières pour C

recherchent des expressions régulières dans les fichiers. Tous ont leur syntaxe propre. En général, ils essaient d'avoir les expressions régulières les plus déterministes possible, sauf `awk`.

Slide 32

Expressions régulières et Unix (2)

On considèrera Awk, Lex et Camllex la prochaine fois.

Regardez le manuel sous Unix entre-temps

```
% man awk
% man lex
% man sed
```

Slide 33

Exercices

- Débutants: on exécute un automate non déterministe donné par une expression régulière donné sous forme d'un type ML.
- Moyens: On considère un automate non de´terministe donné par des listes de transitions, ou par une expression régulière. On veut le déterminiser et minimiser l'automate obtenu.
- Forts: Filtrage: KMP (Knuth, Morris, Pratt), BM (Boyer, Moore).

Majeure 2
Langages et Compilation
Cours 3

Grammaires *contex-free*

`jean-jacques.levy@inria.fr`

2 février 1998

Slide 1

Les supports de cours sont en

`http://www.polytechnique.fr/poly/edu/informatique/`
`http://www.polytechnique.fr/poly/~levy/`

Livres:

- Kozen, Automata and Computability, Springer 1997.
- Hopcroft-Ullman, Formal languages and their Relation to Automata, Addison Wesley, 1969
- Appel, Modern Compiler Implementation in Java, Cambridge University Press, 1997
- Aho, Sethi, Ullman, Compilers: Principle, Techniques, and Tools, Addison Wesley, 1986

Slide 2

Introduction

- Expressions régulières dans Unix
- Analyseurs lexicaux
- Lex, Camllex
- Lemme de l'étoile
- Grammaires context-free
- Formes normales
- Lemme de la pompe
- Grammaires avec parties droites vides
- Grammaires en Yacc, Camlyacc

Slide 3

Expressions régulières dans Unix

`awk`, `sed`, `ed`, `grep`, `sh`

ont des expressions régulières dans leurs arguments. Pour le shell,

`*` répétition quelconque de caractères

`?` tout caractère

`[az]` union des caractères `a` ou `b`

`[a-p]` tous les caractères entre `a` et `p`

Exemples:

```
% ls *.c
% ls *[ab][0-2]*.ml
% ls ?emacs?
```

Slide 4

Expressions régulières dans sed

En `sed`, une expression e régulière est:

c	caractère quelconque à matcher
`^`	au début pour le caractère nul début de ligne
`$`	à la fin pour le caractère nul fin de ligne
`.`	pour tout caractère sauf le newline du bout
e'`*`	pour répéter l'expression e'
`[s]`	pour matcher tout caractère de la chaîne s
`[^s]`	pour matcher tout caractère pas dans la chaîne s
$e'e''$	

Exemples:

```
% sed -e 's/aa*/a/g'
% sed -e '/Ca[^m]*ml/d'
```

Slide 5

Expression régulières dans awk

Interpréteur C guidé par des expressions régulières. Un programme est une suite de paires (expression régulière, programme C):

```
pattern1 {action1}
pattern2 {action2}
etc
```

Deux motifs spéciaux: BEGIN et END pour désigner le début et la fin du fichier.

En outre, la ligne est délimitée en plusieurs champs par `\t` (ou autre caractère donné en option), et les actions peuvent manipuler les champs par `$1`, `$2`, etc. Il y a aussi des tableaux associatifs accessibles par le contenu.

Awk $>>$ Excel.

Slide 6

Exemple de programme Awk

```
awk -F: 'BEGIN {
         code["Prog. L"] = "BD";
         code["BdD"] = "CP";
         code["Cal. Par."] = "PL";
         code["Algo NP"] = "AP";
      } {if (NF >= 13){
    num = $1; cie = $2; unknown = $8;
    nom = $3; prenom = $4; nom = nom "(" prenom ")";
    cp = $5; voie = $6; svoie = $7;
    maj1 = $9; maj2 = "info";
    choix1 = code[$11];
    choix2 = code[$12];
    choix3 = code[$13];
    printf ( "%-23s: %-11s:    %s: %s: %s: \n", \
              nom, \
                         maj1, \
                            choix1, \
                                choix2, \
                                   choix3);
     };}' maj2.txt
```

Slide 7

Expressions régulières en Perl

Perl est un shell avec des sous-programmes. Il a des tableaux associatifs et des expressions régulières (comme awk) qui peuvent intervenir dans les conditions booléennes, grâce à 2 opérateurs: *m//* pour matcher une expression, ou *s//* idem avec remplacement.

Perl n'est pas typé.

Perl est très utilisé dans les `cgi-bin` qui sont les programmes appelés par le Web pour les formulaires, etc.

Slide 8

Analyseurs lexicaux

Un compilateur commence par transformer le flux de caractères d'entrée en une suite d'entités plus abstraites: les lexèmes. Typiquement, il saute les blancs, les tabulations, les newline, et les commentaires. Les lexèmes sont par exemple les

identificateurs,
nombres,
chaînes de caractères,
opérateurs

Slide 9

Analyseurs lexicaux (2)

Chaque lexème a sa syntaxe donnée par une expression régulière:

identificateur = lettre (lettre ou chiffre)*

nombre = chiffre (chiffre)*

chaîne = ”(caractère autre que ”)*”

opérateur = symbole non alphanumérique

Il suffit donc d’un automate fini pour faire l’*Analyse lexicale*.

Comment construire cet automate?

Slide 10

Lex

Lex (Lesk et Schmidt) est un utilitaire Unix qui prend des expressions régulières en entrée et qui produit les tables d’un automate fini déterministe qui reconnait le langage correspondant.

Un programme Lex a la forme suivante:

```
définitions
%%
pattern1 action1
pattern2 action2
...
%%
programmes de l’utilisateur
```

Lexer pour bc levy/m2/bc/scan.l

```
%{
#include "bcdefs.h"
#include "y.tab.h"
#include "global.h"
#include "proto.h"
%}

DIGIT [0-9A-F]
LETTER [a-z]
%%
scale  return(Scale);
ibase  return(Ibase);
obase  return(Obase);
"+"|"-"|";"|"("|")"|"{"|"}"|"["|"]"|","|"^" {
        yylval.c_value = yytext[0]; return((int)yytext[0]); }
&&    { return(AND); }
\|\|  { return(OR); }
"!"   { return(NOT); }
"*"|"/"|"%" { yylval.c_value = yytext[0]; return(MUL_OP); }
"="|\+=|-=|\*=|\/=|%=|\^=  {
        yylval.c_value = yytext[0]; return(ASSIGN_OP); }
=\+|=-|=\*|=\/|=%|=\^  { yylval.c_value = '='; yyless (1);
        return(ASSIGN_OP); }
[ \t]+ { /* ignore spaces and tabs */ }
[a-z][a-z0-9_]* {
        yylval.s_value = strcopyof(yytext); return(NAME); }
\"[^\"]*\"  {
        char *look;
        yylval.s_value = strcopyof(yytext);
        for (look = yytext; *look != 0; look++)
            if (*look == '\n') line_no++;
        return(STRING);
        }
{DIGIT}({DIGIT}|\\\n)*
   ("."({DIGIT}|\\\n)*)?|"."(\\\n)*{DIGIT}({DIGIT}|\\\n)* {
...
            }
.       yyerror ("illegal character: %s",yytext);
%%
```

Slide 11

Slide 12

Camllex

La structure d'un lexer est identique à celle demandée par Lex.

```
{ header }
rule entrypoint =
  parse pattern_1 { action_1 }
      | ...
      | pattern_n { action_n }
and entrypoint =
  parse ...
and ...
;;
```

Slide 13

Quelques exemples de grammaires context-free

- Grammaires d'un langage de programmation
- Expressions bien parenthèsées
- Expressions arithmétiques
- L-systèmes servent pour la synthèse d'images
  (arbres penchés, génération de musac)

Slide 14

Grammaires contex-free

Ce sont les grammaires de type 2, dont les règles sont toutes de la forme $A \to \alpha$ où $\alpha \in V^*$, $V = \Sigma \cup V_N$.

Slide 15

Arbre syntaxique

Arbre dont la racine est étiqueté par l'axiome de la grammaire, les noeuds par des non-terminales et les feuilles par des terminales. Si $A \to \alpha$ est une règle de la grammaire, alors les lettres de α peuvent étiqueter les fils (dans l'ordre) des noeuds de A.

Tout mot w est accepté par la grammaire ssi il existe un arbre syntaxique dont les feuilles forment w. On vérifie facilement que cette condition équivaut à dire qu'il existe une dérivation à partir de l'axiome.

Plusieurs dérivations peuvent correspondre à un même arbre (qui représente en fait une classe de permutation de dérivations produisant une chaîne donnée).

Slide 16

Grammaires ambigues

Plusieurs arbres syntaxiques peuvent aboutir à la même chaîne. On dit alors que la grammaire est *ambigue*. Par exemple:

$S \rightarrow if \;\; C \;\; then \;\; S$
$S \rightarrow if \;\; C \;\; then \;\; S \;\; else \;\; S$

Si on considère le mot

$if \;\; C \;\; then \;\; S \;\; if \;\; C' \;\; then \;\; S' \;\; else \;\; S''$

a deux arbres syntaxiques différents.

Slide 17

L'exemple des expressions arithmétiques

$$\begin{array}{lll} E \rightarrow E + P & E \rightarrow E - P & E \rightarrow P \\ P \rightarrow P * T & E \rightarrow E/T & P \rightarrow T \\ T \rightarrow id & T \rightarrow nb & T \rightarrow (E) \end{array}$$

L'arbre syntaxique de $a * (b + 1)$ est:

Slide 18

Syntaxe abstraite

L'arbre de syntaxe abstraite est un arbre d'opérateurs qui ne contient plus que la sémantique de l'expression. Pour $a * (b + 1)$ c'est:

Pratiquement tous les compilateurs construisent cet arbre avant de générer du code. Il permet de construire des interpréteurs et de faire les différentes analyses.

Slide 19

Yacc

Yacc (Johnson) est un utilitaire Unix qui prend des grammaires en entrée et qui produit les tables d'un analyseur syntaxique qui reconnait le langage correspondant.

Un programme Yacc a la forme suivante:

```
déclarations
%%
règle1 règle2 ...
%%
programmes de l'utilisateur
```

Parser pour bc levy/m2/bc/scan.y

```
%{
#include "bcdefs.h"
#include "global.h"
#include "proto.h"
%}

%start program

%union {
        char      *s_value;
        char       c_value;
        int        i_value;
        arg_list *a_value;
       }
%token <i_value> NEWLINE AND OR NOT
%token <s_value> STRING NAME NUMBER
...
/* Types of all other things. */
%type <i_value> expression return_expression ...
%type <c_value> '+' '-'
...
/* precedence */
%left OR
%left AND
%nonassoc NOT
...
%%
program      : ...
expression   :
               expression '+' expression
                 {
                   generate ("+");
                   $$ = $1 | $3;
                 }
               NUMBER
                 {
                   int len = strlen($1);
                   $$ = 1;
                   if (len == 1 && *$1 == '0')
                     generate ("0");
                   else if (len == 1 && *$1 == '1')
                     generate ("1");
                   else
                     {
                       generate ("K");
                       generate ($1);
                       generate (":");
                     }
                   free ($1);
                 }
               '(' expression ')'
                 { $$ = $2 | 1; }
...
```

Slide 20

Slide 21

Camlyacc

```
/* File parser.mly */
%token <int> INT
%token PLUS MINUS TIMES DIV
%token LPAREN RPAREN
%token EOL
%left PLUS MINUS        /* lowest precedence */
%left TIMES DIV         /* medium precedence */
%nonassoc UMINUS        /* highest precedence */
%start Main             /* the entry point */
%type <int> Main
%%
Main:
    Expr EOL                { $1 }
;
Expr:
    INT                     { $1 }
  | LPAREN Expr RPAREN      { $2 }
  | Expr PLUS Expr          { $1 + $3 }
  | Expr MINUS Expr         { $1 - $3 }
  | Expr TIMES Expr         { $1 * $3 }
  | Expr DIV Expr           { $1 / $3 }
  | MINUS Expr %prec UMINUS { - $2 }
;
```

Slide 22

Lexer correspondant

```
(* File lexer.mll *)
{
#open "parser";;(* The type token is defined in parser.mli *)
exception Eof;;
}
rule Token = parse
    [` ` `\t`]     { Token lexbuf }     (* skip blanks *)
  | [`\n` ]        { EOL }
  | [`0`-`9`]+     { INT(int_of_string (get_lexeme lexbuf)) }
  | `+`            { PLUS }
  | `-`            { MINUS }
  | `*`            { TIMES }
  | `/`            { DIV }
  | `(`            { LPAREN }
  | `)`            { RPAREN }
  | eof            { raise Eof }
;;
```

Slide 23

Programme principal

```
(* File calc.ml *)
try
  let lexbuf = lexing__create_lexer_channel std_in in
  while true do
    let result = parser__Main lexer__Token lexbuf in
      print_int result; print_newline(); flush std_out
  done
with Eof ->
  ()
;;
```

On compile le tout en tapant les ordres suivants au niveau shell.

```
camllex lexer.mll        # generates lexer.ml
camlyacc parser.mly      # generates parser.ml and parser.mli
camlc -c parser.mli
camlc -c lexer.ml
camlc -c parser.ml
camlc -c calc.ml
camlc -o calc lexer.zo parser.zo calc.zo
```

Slide 24

Les modules Camllex et Camlyacc

Dans l'exemple précédent:

`lexing__create_lexer_channel` crée un canal d'entrée attaché à stdin, sur le lexer `lexer__Token` décrit en camllex pourra fonctionner.

Le lexer retourne les valeurs `EOL`, `INT`, `PLUS`, etc qui sont les constructeurs d'un type token généré par camlyacc dans l'interface parser.mli. `INT` est un constructeur unaire qui prend un entier en argument.

En yacc, des expressions Caml sont attachées à chaque règle, et retournées comme valeur de la non-terminale membre gauche de la règle. Les valeurs associés aux lettres des parties droites sont notées \$1, \$2, etc en fonction de leur ordre d'apparition.

Slide 25

Formes normales de grammaires

Prop 1 Soit $G = (\Sigma, V_N, S, \mathcal{P})$ générant un langage non vide. Alors on peut trouver une grammaire équivalente G_1 telle que pour toute non-terminale A de G_1, il existe $w \in \Sigma^*$ tel que $A \Rightarrow^* w$.

Prop 2 Soit L un langage context-free. Alors il existe une grammaire G générant L telle que pour toute non-terminale A de G, il existe une dérivation $S \Rightarrow^* w_1 A w_2 \Rightarrow^* w_1 w_2 w_3$ où w_1, w_2, w_3 sont des chaines terminales.

Prop 3 Soit G une grammaire context-free. Si $G \Rightarrow^* w$, alors il existe une dérivation gauche (ie dérivant la non-terminale toujours la plus à gauche) donnant w.

Slide 26

Formes normales de grammaires (2)

Prop 4 Soit G une grammaire context-free, on peut trouver une grammaire équivalente sans aucune règle de la forme $A \rightarrow B$, où A et B sont deux non-terminales.

Forme normale de Chomsky Tout langage context-free peut être généré par une grammaire dont toutes les règles sont de la forme $A \rightarrow BC$ ou $A \rightarrow a$ où A, B sont non-terminales, et a est terminale.

Prop 5 Appelons A-règle toute règle dont A est le membre gauche. Si $A \rightarrow \alpha_1 B \alpha_2$ est une A-règle et $\{B \rightarrow \beta_1, \cdots, B \rightarrow \beta_r\}$ est l'ensemble des B-règles. Alors la grammaire G_1 obtenue en remplaçant la règle $A \rightarrow \alpha_1 B \alpha_2$ par $A \rightarrow \alpha_1 \beta_1 \alpha_2, \cdots A \rightarrow \alpha_1 \beta_r \alpha_2$ est équivalente.

Slide 27

Formes normales de grammaires (3)

Prop 4 Soit G une grammaire context-free Soit $\{A \to A\alpha_1, \cdots A\alpha_r\}$ l'ensemble des A-règles dont A est le symbole le plus à gauche. Soit $A \to \beta_1, \cdots A \to \beta_s$ l'ensemble des A-règles restantes. Alors on peut trouver une grammaire équivalente, dont toutes les A-règles sont de la forme $A \to \beta_i$, $A \to \beta_i Z$, $Z \to \alpha_i$, $Z \to \alpha_i Z$ où Z est un nouveau symbole non terminal.

Forme normale de Greibach Toute langage context-free peut être généré par une grammaire dont toute les règles sont de la forme $A \to a\alpha$ où a est une terminale et $\alpha \in V^*$, $V = \Sigma \cup V_N$.

Slide 28

Lemme de l'étoile pour les langages réguliers

Théorème Soit L un langage régulier. Il existe p ne dépendant que de L tel que pour tout mot z de L dont la longueur $|z|$ excède p, il existe une décomposition $z = uvw$ avec $|v| \leq p$, $|v| \neq 0$, et pour tout n, uv^nw est dans L.

Exercice Montrer que le langage suivants n'est pas régulier

$L_0 = \{a^nb^n\}$

Slide 29

Lemme de la pompe pour les langages context-free

Théorème Soit L un langage context-free. Il existe p et q dépendant de L tel que pour tout mot z de L dont la longueur $|z|$ excède p, il existe une décomposition $z = uvwxy$ avec $|vwx| \leq q$, $|vx| \neq 0$, et pour tout n, uv^nwx^ny est dans L.

Exercice Montrer que les langages suivants ne sont pas context-free

$L_1 = \{a^n b^n c^n\}$
$L_2 = \{ww\}$
$L_2 = \{a^n \mid n \text{ premier}\}$

Montrer que tout langage context-free sur un alphabet à une lettre est régulier.

Slide 30

Exercices

- Quelques-uns: refaire `word_count` avec lex.
- Tous: on fait une calculette qui peut avoir des variables globales et une instruction *plot*

  Exemple:

```
set x = 25 ;;
set y = 228 ;;
set z = x ** 4 - y / (1 - x) ;;
2*z ;;

plot f(x) = cos (x ** 2);;
```

Majeure 2
Langages et Compilation
Cours 4

Analyse syntaxique,
Portée lexicale

jean-jacques.levy@inria.fr

9 février 1998

Slide 1

Les supports de cours sont en

http://www.polytechnique.fr/poly/edu/informatique/
http://www.polytechnique.fr/poly/~levy/

Livres:

- Hopcroft-Ullman, Formal languages and their Relation to Automata, Addison Wesley, 1969
- Appel, Modern Compiler Implementation in Java, Cambridge University Press, 1997
- Aho, Sethi, Ullman, Compilers: Principle, Techniques, and Tools, Addison Wesley, 1986
- Wilhelm, Maurer, Compiler Design, Addison Wesley, 1995
- Peyton Jones, The implementation of Functional Programming Languages, Prentice-Hall, 1987

Slide 2

Introduction

- Automates à pile
- Analyse syntaxique
- Tables de Yacc
- Erreurs syntaxiques
- Portées des déclarations
- Interpréteur et portée

Slide 3

Automates à pile

Un automate à pile (*pushdown automata*) est un n-uplet $(Q, \Sigma, \Gamma, \delta, q_0, Z_0, F)$ où

Q est un ensemble *fini* d'états,
Σ est un alphabet (de terminaux ou *tokens*),
Γ est un alphabet fini de symboles de pile,
$q_0 \in Q$ est l'état initial,
$Z_0 \in \Gamma$ est le symbole de pile initial,
$F \subset Q$ est l'ensemble des états finaux,
$\delta : Q \times (\Sigma \cup \{\epsilon\}) \times \Gamma \to 2^{Q \times \Gamma^*}$ est la fonction de transition

Slide 4

Déplacements élémentaires

$\delta(q, a, Z) = \{(q_1, \gamma_1), \cdots (q_n, \gamma_n)\}$ veut dire que, dans l'état q, si le sommet de pile est Z et si la tête de lecture voit la lettre a, alors l'automate déplace sa tête de lecture vers la droite et peut aller non déterministiquement dans un des états q_i en remplaçant la lettre du sommet de pile Z par le mot γ_i. (Par convention, la pile est lue de la gauche vers la droite en allant du sommet à sa base).

$\delta(q, \epsilon, Z) = \{(q_1, \gamma_1), \cdots (q_n, \gamma_n)\}$ veut dire qu'on fait de même sans regarder le symbole sous la tête de lecture. Alors la tête de lecture ne se déplace pas vers la droite.

(M.P.Schutzenberger, 1962)

Slide 5

Configurations et acceptation

Une configuration est une paire (q, γ) où q est l'état courant et γ est le contenu de la pile.

$(q, Z\gamma) \xrightarrow{a} (p, \beta\gamma)$ si $(p, \beta) \in \delta(q, a, Z)$ où $a \in \Sigma \cup \epsilon$,

$(q, \gamma) \xrightarrow{w} (p, \beta)$ si $w = a_1 a_2 \dots a_n$ et $(q, \gamma) = (q_0, \gamma_0)$, $(p, \beta) = (q_n, \gamma_n)$ et $(q_i, \gamma_i) \xrightarrow{a_i} (q_{i+1}, \gamma_{i+1})$ pour $0 \leq i < n$, $n \geq 0$.

L'automate M reconnait le langage $T(M)$

$T(M) = \{w \mid (q_0, Z_0) \xrightarrow{w} (q, \gamma),\ q \in F\}$

Slide 6

Acceptation

On peut aussi reconnaitre par pile vide, sans se servir de F.

$T'(M) = \{w \mid (q_0, Z_0) \xrightarrow{w} (q, \epsilon)\}$

Cela ne change pas la catégorie des langages reconnus.

Un automate à pile est déterministe si une seule transition est possible dans chaque configuration.

Théorème Les langages context-free sont les langages reconnus par les automates à pile.

Démonstration: évidente dans un sens. Légèrement plus compliquée dans l'autre sens. $\square$

Slide 7

Propriété de fermetures des langages context-free

Les langages context-free sont fermés par union, mais pas par intersection. Par exemple, $L_1 = \{a^n b^n c^p\}$ et $L_2 = \{a^p b^n c^n\}$ sont context-free, mais pas $L_1 \cap L_2$.

L'intersection d'un langage régulier et d'un context-free est un langage context-free.

Appelons homomorphisme toute application $\Sigma_1 \to \Sigma_2^*$ que l'on étend trivialement sur Σ_1^*. Alors les langages context-free sont clos par homomorphisme. Mieux:

Théorème[Chomsky-Schutzenberger] Tout langage context-free est l'image par homomorphisme de l'intersection d'un langage de parenthèses (langage de Dyck) et d'un langage régulier.

Slide 8

Automates déterministes

Pour analyser les langages de programmation en temps linéaire, on veut des automates déterministes (DPDA, *deterministic pushdown automata*).

Les langages context-free déterministes sont les langages reconnaissables par un automate déterministe. Par exemple, le langage $\{a^n b^n\} \cup \{a^n b^{2n}\}$ est context-free non déterministe.

[La conjecture sur l'équivalence des DPDA ne vient d'être résolue qu'en 1997 par Sénizergues de Bordeaux]

La programmation moderne utilise la récursion plutôt que les piles. Mais Yacc génère des analyseurs avec une pile. Le problème du déterminisme se retrouve aussi si on programme récursivement.

Slide 9

Analyse syntaxique

cf. le livre d'Andrew Appel

Slide 10

Récupération d'erreurs

En fait, on analyse syntaxiquement un langage et ses erreurs. Il est difficile de faire de bonnes récupérations d'erreurs. Typiquement on veut sauter au `;;` suivant. Il faut donc savoir récupérer l'erreur soit grace à un token spécial `error` en yacc, soit en récupérant l'execption `Parse_error` en Caml, et en sautant manuellement les lexèmes jusqu'au `;;` suivant, par exemple.

Slide 11

PCF: un petit langage fonctionnel

e, e', e''	$::=$		expressions
	$\mid$	nombre	constante
	$\mid$	$\oplus\, e$	opération unaire
	$\mid$	$e \otimes e'$	opération binaire
	$\mid$	$\texttt{ifz}\, e\ \texttt{then}\ e'\ \texttt{else}\ e''$	conditionnelle
	$\mid$	x	variable
	$\mid$	$\texttt{let}\, x = e\ \texttt{in}\ e'$	définition locale
	$\mid$	$e(e')$	application
	$\mid$	$\texttt{fun}\, x \to e$	fonction
	$\mid$	$\texttt{letrec}\, x = e\ \texttt{in}\ e'$	déf. récursive

Slide 12

Un interpréteur

On construit une fonction d'évaluation $eval$

$$eval(\underline{n}) = n$$
$$eval(-e) = -eval(e)$$
$$eval(e + e') = eval(e) + eval(e')$$
$$eval(e - e') = eval(e) - eval(e')$$
$$eval(e + e') = eval(e) + eval(e')$$

...

$$eval(\texttt{ifz}\, e\ \texttt{then}\ e'\ \texttt{else}\ e'') = \begin{cases} eval(e') & \text{si } eval(e) = 0 \\ eval(e'') & \text{sinon} \end{cases}$$

Slide 13

Déclarations locales et portées

Pour avoir des variables locales et des définitions locales, il faut donner un deuxième argument à la fonction d'évaluation, l'environnement ρ qui décrit la liaison des variables locales.

ρ est une liste d'association entre les noms des variables locales et leur valeur.

On redéfinit la fonction $eval$.

$eval(\underline{n}, \rho) = n$
$eval(-e, \rho) = -eval(e, \rho)$
$eval(e + e', \rho) = eval(e, \rho) + eval(e', \rho)$
...
$eval(\mathtt{let}\, x = e \,\mathtt{in}\, e', \rho) = eval(e', (x, eval(e, \rho)) :: \rho)$
$eval(x, (x, v) :: \rho) = v$
$eval(x, (y, v) :: \rho) = eval(x, \rho) \qquad x \neq y$

Slide 14

Valeurs fonctionnelles et application

Comment déclarer des fonctions et les appliquer? C'est simple dans PCF, il suffit de rajouter l'expression fonctionnelle $\mathtt{fun}\, x \rightarrow e$ et l'application $e(e')$.

Les expressions pourront donner non seulement des valeurs scalaires, mais aussi des *fermetures*, i.e. un couple formé d'une expression et d'un environnement.

On ajoute simplement à l'évaluateur les règles:

$eval(\mathtt{fun}\, x \rightarrow e, \rho) = (\mathtt{fun}\, x \rightarrow e, \rho)$
$eval(ee', \rho) = eval(e'', (x, eval(e', \rho)) :: \rho'')$
où $eval(e, \rho) = (\mathtt{fun}\, x \rightarrow e'', \rho'')$

Slide 15

Récursivité

Comme attendu, la récursivité introduite par `letrec` est une simple variation sur l'évaluation de `let`.

$eval(\texttt{letrec}\, x = e\ \texttt{in}\ e', \rho) = eval(e', (x, v) :: \rho)$
où $v = eval(e, (x, v) :: \rho)$

Slide 16

Exemples et exercices

On peut maintenant analyser et évaluer les petits programmes suivants:

```
letrec fact = fun x -> ifz x then 1 else x * fact (x-1) in
fact(10);;
```

```
letrec fib = fun x -> ifz x then 0 else ifz x-1 then 0 else
                      fib (x-1) + fib(x-2) in
fib(20);;
```

```
let x = let y = 20 in y+3*y in
let y = let y = 20 in x-y in
x + y ;;
```

```
let twice = fun f -> fun x -> f (f x) in
let succ = fun x -> x+1 in
twice succ 0;;
```

Majeure 2
Langages et Compilation
Cours 5

Tables des symboles, Frames
Code intermédiaire

`jean-jacques.levy@inria.fr`

2 mars 1998

Slide 1

Les supports de cours sont en

`http://www.polytechnique.fr/poly/edu/informatique/`

`http://www.polytechnique.fr/poly/˜levy/`

Livres:

- Appel, Modern Compiler Implementation in Java, Cambridge University Press, 1997
- Aho, Sethi, Ullman, Compilers: Principle, Techniques, and Tools, Addison Wesley, 1986
- Wilhelm, Maurer, Compiler Design, Addison Wesley, 1995
- Peyton Jones, The implementation of Functional Programming Languages, Prentice-Hall, 1987

Slide 2

Introduction

- La calculette fonctionnelle
- Sémantique opérationnelle
- Appel par nom
- Blocs d'activation et piles
- CTigre: un langage impératif à compiler
- Langage intermédiaire

Slide 3

La calculette fonctionnelle

Au cours précédent, l'interpréteur de la calculette posait un léger problème pour la programmation du `letrec`, où il fallait construire un environnement cyclique. En Caml, on peut construire des données récursives non-fonctionnelles, pourvu qu'un constructeur se trouve entre la définition et les occurences récursives. Ainsi:

```
let rec x = 1 :: x in ...
type enregistrement = {hd: int; tl: enregistrement};;
let rec x = {hd = 1; tl = y}  and  y = {hd = 2; tl = x} ;;
```

sont autorisés.

En `http://pauillac.inria.fr/~levy/x/m2/`, on trouve les solutions en Caml et en C. (On peut comparer leur longueur respective).

Slide 4

Sémantique opérationnelle – *bigstep SOS*

$$\rho \vdash n = n \qquad \frac{\rho \vdash e = n}{\rho \vdash -e = -n}$$

$$\frac{\rho \vdash e = n \quad \rho \vdash e' = n'}{\rho \vdash e + e' = n + n'} \qquad \frac{\rho \vdash e = n \quad \rho \vdash e' = n'}{\rho \vdash e * e' = n \times n'}$$

$$\frac{\rho \vdash e = 0 \quad \rho \vdash e' = v}{\rho \vdash \mathtt{ifz}\, e \;\mathtt{then}\; e' \;\mathtt{else}\; e'' = v}$$

$$\frac{\rho \vdash e = n \quad \rho \vdash e'' = v \quad (n \neq 0)}{\rho \vdash \mathtt{ifz}\, e \;\mathtt{then}\; e' \;\mathtt{else}\; e'' = v}$$

$$\frac{\rho \vdash e = v \quad \rho, x = v \vdash e' = v'}{\rho \vdash \mathtt{let}\, x = e \;\mathtt{in}\; e' = v'}$$

$$\rho, x = v \vdash x = v \qquad \frac{\rho \vdash x = v}{\rho, y = v \vdash x = v}$$

Slide 5

Sémantique opérationnelle – bis

$$\rho \vdash \mathtt{fun}\, x \rightarrow e = (\mathtt{fun}\, x \rightarrow e)[\rho]$$

$$\frac{\rho \vdash e = (\mathtt{fun}\, x \rightarrow e'')[\rho''] \qquad \rho \vdash e' = v' \qquad \rho'', x = v' \vdash e'' = v}{\rho \vdash e(e') = v}$$

$$\frac{\rho, x = v \vdash e = v \quad \rho, x = v \vdash e' = v'}{\rho \vdash \mathtt{letrec}\, x = e \;\mathtt{in}\; e' = v'}$$

Adaptant la technique de Barendregt(72) pour le λ-calcul de Church(30), Plotkin(78) a été le promoteur de cette technique de description, aussi appelée sémantique naturelle. Il existe d'autres techniques pour donner un sens aux programmes, notamment la sémantique dénotationnelle de Scott(70) et Strachey(68).

Slide 6

Appel par valeur, appel par nom

On calcule la valeur v d'un paramètre avant de le passer à une fonction

$$v \quad ::= \quad n \mid (\mathtt{fun}\, x \rightarrow e)[\rho]$$

comme en C, Caml, Java, Pascal.

On peut changer la sémantique pour faire une évaluation paresseuse, qui n'évalue pas l'argument avant de le passer à une fonction ou procédure. C'est l'appel-par-nom d'Algol, Miranda, Haskell, LML, Gaml. On change alors la SOS. En paresseux, on fabrique un nouveau type d'expressions qu'on évaluera plus tard: les *thunk* ou glaçons $e[\rho]$

Slide 7

Sémantique opérationnelle paresseuse– *bigstep*

$$\rho \vdash n = n \qquad \frac{\rho \vdash e = n}{\rho \vdash -e = -n}$$

$$\frac{\rho \vdash e = n \quad \rho \vdash e' = n'}{\rho \vdash e + e' = n + n'} \qquad \frac{\rho \vdash e = n \quad \rho \vdash e' = n'}{\rho \vdash e * e' = n \times n'}$$

$$\frac{\rho \vdash e = 0 \quad \rho \vdash e' = v}{\rho \vdash \texttt{ifz}\, e \;\texttt{then}\; e' \;\texttt{else}\; e'' = v}$$

$$\frac{\rho \vdash e = n \quad \rho \vdash e'' = v \quad (n \neq 0)}{\rho \vdash \texttt{ifz}\, e \;\texttt{then}\; e' \;\texttt{else}\; e'' = v}$$

$$\frac{\rho \vdash e = v \quad \rho, x = v \vdash e' = v'}{\rho \vdash \texttt{let}\, x = e \;\texttt{in}\; e' = v'}$$

$$\frac{\rho' \vdash e' = v}{\rho, x = e'[\rho'] \vdash x = v} \qquad \frac{\rho \vdash x = v}{\rho, y = v \vdash x = v}$$

Slide 8

Sémantique opérationnelle paresseuse – bis

$$\rho \vdash \texttt{fun}\, x \to e = (\texttt{fun}\, x \to e)[\rho]$$

$$\frac{\rho \vdash e = (\texttt{fun}\, x \to e'')[\rho''] \qquad \rho'', x = e'[\rho] \vdash e'' = v}{\rho \vdash e(e') = v}$$

$$\frac{\rho, x = v \vdash e = v \quad \rho, x = v \vdash e' = v'}{\rho \vdash \texttt{letrec}\, x = e \;\texttt{in}\; e' = v'}$$

En appel paresseux, le résultat est le même si on remplace les arguments dans le corps des fonctions

$(\texttt{fun}\, x \to e)(e') \equiv e\{x := e'\}$

(*copy rule*). Autrement dit, une expression a le même sens quelle que soit sa place dans le programme (*referential transparency*). C’est un argument pour la parallélisation automatique.

Slide 9

Portée des noms, bloc d'activations et piles

L'environnement ρ est gérée comme une liste d'association dans l'exemple de la calculette fonctionnelle. Les valeurs des expressions sont des nombres ou des fermetures, ie paires (fonction, environnement). Si on ne retourne jamais de fonctions, on peut se contenter d'une pile pour gérer les environnements, même si on passe des fonctions en arguments, comme en Pascal, Algol, ou Modula.

Cette remarque due à Randell et Russel(60) a permis d'interpréter ces langages avec des piles comme seul espace d'allocation mémoire, et donc de les compiler simplement.

Slide 10

SOS pour blocs d'activations

Modifions la SOS (en appel par valeur) pour faire apparaître la notion de pile pour gérer l'environnement. Les valeurs sont maintenant

$$v \quad ::= \quad n \mid (\mathtt{fun}\, x \to e)[s]$$

où s est un nombre permettant d'accéder au s-ième élément de l'environnement ρ courant.

Posons
$|\rho|$ pour la longueur de ρ,
$\rho[1..s]$ pour la suite des s premiers éléments de ρ.

Slide 11

SOS pour blocs d'activations

$$\rho \vdash n = n \qquad \frac{\rho \vdash e = n}{\rho \vdash -e = -n}$$

$$\frac{\rho \vdash e = n \quad \rho \vdash e' = n'}{\rho \vdash e + e' = n + n'} \qquad \frac{\rho \vdash e = n \quad \rho \vdash e' = n'}{\rho \vdash e * e' = n \times n'}$$

$$\frac{\rho \vdash e = 0 \quad \rho \vdash e' = v}{\rho \vdash \texttt{ifz}\, e \texttt{ then } e' \texttt{ else } e'' = v}$$

$$\frac{\rho \vdash e = n \quad \rho \vdash e'' = v \quad (n \neq 0)}{\rho \vdash \texttt{ifz}\, e \texttt{ then } e' \texttt{ else } e'' = v}$$

$$\frac{\rho \vdash e = v \quad \rho, (|\rho|, x = v) \vdash e' = v'}{\rho \vdash \texttt{let}\, x = e \texttt{ in } e' = v'}$$

$$\rho, (s, x = v) \vdash x = v \qquad \frac{\rho[1..s] \vdash x = v}{\rho, (s, y = v') \vdash x = v}$$

Slide 12

SOS pour blocs d'activations – bis

$$\rho \vdash \texttt{fun}\, x \rightarrow e = (\texttt{fun}\, x \rightarrow e)[\,|\rho|\,]$$

$$\frac{\begin{array}{c}\rho \vdash e = (\texttt{fun}\, x \rightarrow e'')[s] \\ \rho \vdash e' = v' \qquad \rho, (s, x = v') \vdash e'' = v\end{array}}{\rho \vdash e(e') = v}$$

$$\frac{\rho, (|\rho|, x = v) \vdash e = v \quad \rho, (|\rho|, x = v) \vdash e' = v'}{\rho \vdash \texttt{letrec}\, x = e \texttt{ in } e' = v'}$$

ρ a le comportement d'une pile, dont le bas se lit à gauche et le sommet à droite.

Slide 13

Blocs d'activations et liaison statique

A chaque appel de fonction (ou pour chaque `let`, on empile une paire (valeur, *lien statique*). Nous n'avons considéré que les fonctions unaires. Mais une variation sur la sémantique précédente permet de traiter le cas n-aire. Alors à chaque application de n arguments à une fonction, on créera un $n+1$-uplet $(s, x_1 = v_1, \cdots x_n = v_n)$, ie un bloc d'activation (*frame*) contenant au moins ce n-uplet. Ce bloc sera libéré quand l'appel de fonction sera terminé et rendra sa valeur.

Tous les langages ne peuvent s'évaluer ainsi. C'est le cas pour notre calculette fonctionnelle (PCF). Par exemple:

$\rho \vdash (\mathtt{fun}\, x \rightarrow \mathtt{fun}\, y \rightarrow x)(1) = (\mathtt{fun}\, y \rightarrow x)[|\rho| + 1]$!!

Slide 14

Blocs d'activations et liaison statique – bis

Il faut faire une analyse d'échappements (*escape analysis*) pour savoir si on peut le faire. (cf Blanchet(98) pour ML)

Tout Pascal marche, car les valeurs fonctionnelles passées uniquement en argument font toujours référence à des environnements plus bas dans la pile. C marche aussi, puisque le lien statique dans C vaut toujours 1, puisqu'on ne peut pas déclarer de fonctions dans des fonctions. On peut donc retourner des fonctions sans problèmes d'allocation de pile.

Slide 15

Blocs d'activations et variables locales

Dans les blocs d'activations, on met aussi les variables locales engendrées par les `let`. On accède donc à toutes les variables par une indirection et un déplacement par rapport au début du bloc d'activation dont l'adresse est contenue dans le pointeur d'activation (*frame pointer*). Un autre pointeur intéressant est le pointeur de pile, (*stack pointer*). L'entrée et la sortie des procédures / fonctions consistera à leur fixer une bonne valeur.

En Pascal, on a vu qu'un programme (manipulant des listes ou arbres) n'a pas toutes ses données allouées dans les différents blocs d'activation de la pile d'exécution. Il y a aussi besoin d'un tas (*heap*) où on met les données créées dynamiquement (cf plus tard).

Slide 16

Un petit langage impératif à compiler

On utilisera le langage Tigre d'Appel, en Caml-isant la syntaxe. La BNF se trouve en `http://pauillac.inria.fr/~levy/x/m2/ctig/`

Voici un exemple de programme en CTigre.

```
let N := 100 in
type Tableau = array of int in
let TriInsertion (a:Tableau) =
    let j := 0 and v := 0 in
        for i := 2 to N do
            v := a[i]; j := i;
            while a[j-1] > v do
                a[j] := a[j-1]; j:= j-1
                done;
            a[j] := v
        done in
let a := array [N] of 0 in

a[1] := 3; a[2] := 2; a[3] := 1;
TriInsertion (a)
```

Slide 17

Un petit langage impératif à compiler – bis

Un autre exemple avec des structures de données dynamiques.

```
type ListeInt = {hd: int, tl: ListeInt} in
let TriFusion (x: ListeInt, y: ListeInt) =
    if x = nil then y else
    if y = nil then x else
    if hd(x) <= hd(y) then
        ListeInt{hd = hd(x), TriFusion (tl(x), y)
    else
        ListeInt{hd = hd(y), TriFusion (x, tl(y))
in
let x = ListeInt{hd = 3, tl =  ListeInt{hd = 2, tl = nil}}
and y = ListeInt{hd = 1, tl =  ListeInt{hd = 4, tl = nil}}
in TriFusion (x, y)
```

Slide 18

La BNF de CTigre

```
programme ::=  expression

expression  ::=
        primary-expression
      | construction-expression
      | nary-expression
      | sequencing-expression

primary-expression  ::=
        ident
     |  constant
     |  ( expression )
     |  begin expression end

construction-expression  ::=
     |  type-id [ expression ] of expression
     |  type-id { label = expression {; label = expression } }

nary-expression  ::=
         ident  ( expression {, expression } )
      |  - expression   |  expression bin-op expression
      |  l-value        |  l-value := expression

sequencing-expression  ::=
         expression ; expression
      |  if expression then expression [  else expression ]
      |  while expression do expression done  |  break
      |  for ident = expression (  to |  downto ) expression do expression done
      |  type  type-binding  { and type-binding }  in expression
      |  let  let-binding  { and let-binding }  in expression

l-value ::= ident  |  l-value.label  |  l-value [ expression ]

type-binding  ::=  id = type-expression

type-expression   ::=
         type-id
   |  { ty-fields }
   |  array of type-id

let-binding ::=
         ident [: type-id ] := expression
      |  ident ( ty-fields ) [: type-id ] = expression

ty-fields  ::=  [  ident : type-id  {, ident : type-id } ]

bin-op  ::=  + | -  | * | /  | = | <> | < | <= | > | >=

constant ::= integer-literal | string-literal | nil

type-id  ::=  ident

label  ::=  ident
```

Slide 19

Plan pour la suite

CTigre ressemble à Pascal. Il n'a pas de pointeurs. Ses structures de données sont les entiers, chaînes, tableaux et enregistrements, qui peuvent être récursifs (comme en Caml ou Java). Les variables sont toutes modifiables.

Le plan est donc de construire l'AST (arbres de syntaxe abstraite) de CTigre, et d'avoir très rapidement un interpréteur, qui ne sera qu'une légère modification de celui de la calculette PCF.

Puis, on construira un générateur de code pour CTigre, l'optimum étant de pouvoir générer du code machine pour R-2000, dont nous avons l'interpréteur `spim`, `xspim` de Larus (Wisconsin 97) sur les machines de l'X. Une autre solution sera de générer du code pour le Pico-Risc de Weis et Leroy (chapitre 13 du livre *Le langage Caml*).

Slide 20

AST de CTigre

$$
\begin{array}{lcl}
e, e' & ::= & \textsf{NilExp} \mid \textsf{IntExp}(n) \mid \textsf{StringExp}(s) \mid lv \\
 & \mid & \textsf{RecordExp}(x, l_f) \mid \textsf{ArrayExp}(x, e, e') \\
 & \mid & \textsf{CallExp}(x, l) \mid \textsf{OpExp}(op, e, e') \\
 & \mid & \textsf{AssignExp}(lv, e) \mid \textsf{SeqExp}(e, e') \\
 & \mid & \textsf{IfExp}(e, e', e'') \mid \textsf{IfExp1}(e, e') \\
 & \mid & \textsf{WhileExp}(e, e') \mid \textsf{ForExp}(d_v, e, e') \\
 & \mid & \textsf{BreakExp}() \\
 & \mid & \textsf{LetExp}(l_{dv}) \mid \textsf{TypeExp}(l_{dt}) \\
lv & ::= & \textsf{SimpleVar}(x) \mid \textsf{FieldVar}(lv, x) \mid \textsf{SubscriptVar}(lv, e) \\
d_v & ::= & \textsf{VarDec}(x, t, e) \mid \textsf{FunDec}(x, l_f, t, e) \\
d_t & ::= & \textsf{TypeDec}(x, \tau, e) \\
\tau & ::= & \textsf{NameType}(a) \mid \textsf{RecordType}(a) \mid \textsf{ArrayType}(a) \\
l & ::= & e_1, e_2, \cdots e_n \\
l_{dv} & ::= & d_{v_1}, d_{v_2}, \cdots d_{v_n} \\
l_{dt} & ::= & d_{t_1}, d_{t_2}, \cdots d_{t_n} \\
l_f & ::= & (x_1, t_1), (x_2, t_2), \cdots (x_n, t_n)
\end{array}
$$

où x,f, t sont des noms de variables, champs ou type.

Slide 21

Sémantique de CTigre

On ne donne pas la SOS. C'est faisable, en remarquant qu'alors les environnements ρ sont des fonctions des noms dans les locations, et qu'il faut donc manipuler un état mémoire s, fonction des locations dans les valeurs. $\rho(x)$ est aussi appelé *left-value* de x. Les règles sont de la forme $\rho \vdash \langle e, s \rangle = \langle v, s' \rangle$, où s et s' sont les états mémoire de début et de fin d'évaluation. Exercice?!

Qques remarques:

- deux types d'expressions avec résultat ou sans, ce qui fait la différence entre les 2 cas de déclaration de fonctions,

- les chaînes ne servent que pour les fonctions de librairies (impression, concaténation, etc). On peut tester leur égalité.

Slide 22

IC: un langage intermédiaire

La première idéalisation de la machine cible contient 2 types d'objets:

$$\begin{array}{lll} e, e' & ::= & \mathsf{CONST}(n) \mid \mathsf{NAME}(l) \\ & \mid & \mathsf{TEMP}(t) \mid \mathsf{MEM}(e) \mid \mathsf{MEM}'(e, k) \\ & \mid & \mathsf{BINOP}(op, e, e') \mid \mathsf{CALL}(e, args) \mid \mathsf{ESEQ}(s, e) \end{array}$$

$$\begin{array}{lll} s, s' & ::= & \mathsf{MOVE}(e, e') \mid \mathsf{EXP}(e) \\ & \mid & \mathsf{JUMP}(e, targets) \mid \mathsf{CJUMP}(op, e, e', l_t, l_f) \\ & \mid & \mathsf{SEQ}(s, s') \mid \mathsf{LABEL}(l) \end{array}$$

$$\begin{array}{lll} args & ::= & e_1, e_2, \cdots e_n \\ targets & ::= & l_1, l_2, \cdots l_n \end{array}$$

Slide 23

IC: un langage intermédiaire – bis

Un programme sera une liste d'association (l, e) ou (l, s). On pourra utilisé les tables avec hachage de Caml pour aller plus vite.

Les noms l sont des étiquettes (comme en assembleur). TEMP(t) est le registre t. On suppose un nombre infini de registres. MEM(e) est la mémoire d'adresse e. ESEQ(s, e) donne la valeur de e après avoir fait s.

Dans les instructions, on fait surtout MOVE(TEMP$(t), e)$ et MOVE(MEM$(e), e')$. Dans JUMP$(e, targets)$, la valeur de e doit être incluse dans $targets$. (C'est pour faciliter les analyses ultérieures). EXP(e) évalue e et oublie sa valeur. SEQ(s, s') fait s puis s'. Et LABEL(l) définit l'étiquette l.

Slide 24

IC: un langage intermédiaire – ter

CALL$(e, args)$ appelle la procédure à l'adresse e. CALL calcule son corps en créant un bloc MEM$'$(TEMP$(fp), nW)$ où figurent les valeurs des n arguments (W est la taille d'un mot mémoire). Le résultat de CALL est la valeur de l'expression, corps de la fonction (valeur folklo si c'est une procédure). CALL a donc une interaction avec le mécanisme d'adressage de toutes les variables, via le registre TEMP(fp).

CALL évalue les arguments args (dans un sens non précisé) et donne la main à la procédure / fonction qui se trouve en e avec les arguments en MEM(TEMP(fp)), MEM(TEMP$(fp) + W)$, ...MEM(TEMP$(fp) + (n - 1)W)$ où W est la taille du mot machine.

Le retour de CALL remet TEMP(fp) à son ancienne valeur.

Slide 25

Syntaxe concrète pour IC

Pour arriver à lire la traduction, on va écrire le code IC sous forme humaine (sans arbres en espérant que la traduction en arbre est évidente).

`if e then e1 else e2` devient

```
if e then goto l1 else goto l2 {l1,l2}; r; done:
l1: r := e1; goto done {done}
l2: r := e2; goto done {done}
```

`while e do e1 done` devient

```
loop: if e then goto l1 else goto done {l1,done}
l1: e1; goto loop  {loop}
done: r
```

Slide 26

Quelques autres traductions

`for i := e1 to e2 do body done` devient

```
let i := e1
and limite := e2
in  while i <= limite do
      body;
      i := i + 1
    done
```

Slide 27

Les variables simples

La location d'une variable est dans un bloc d'activation. On suppose que le frame pointer est toujours disponible dans le registre TEMP(fp). L'accès à une variable du bloc courant est donc

MEM(BINOP$(+$, CONST(k), TEMP$(fp)))$ ou encore `mem[fp-k]`

Si la variable est défini dans un bloc n fois englobant, on suit le lien statique pour arriver à sa valeur

```
mem[...[mem[mem[fp-k1]-k2]...-kn]]
```

où $k_1 \cdots k_{n-1}$ sont les adresses des liens statiques dans les blocs et k_n est l'adresse de la variable recherchée.

Slide 28

Let et letrec

`let x1 = v1 and x2 = v2 and ... xn = vn` se traduit par la création d'un nouveau bloc MEM$'$(TEMP$(fp), (n+1)W)$ contenant les valeurs des variables et le lien statique pour accéder aux variables des blocs englobant. Alors ce lien vaut la valeur de l'ancien fp.

La taille réservée peut être plus grande si des tableaux ou des enregistrements sont déclarés dans `let`.

`letrec` a le même comportement.

Slide 29

Calcul du lien statique

Les définitions de procédures peuvent être emboitées, on passe donc aussi à l'appel le lien statique du corps de la procédure appelée. Celui-ci est le lien statique du bloc contenant la définition de la fonction appelée. Donc si la procédure est définie dans un bloc n-fois englobant, on ira chercher ce lien statique comme pour accéder une variable simple.

```
call(f, e1, e2, ... en, mem[...[mem[mem[fp]]...]+k]])
```

si on suppose le liens statique toujours rangé au premier mot de chaque bloc. C'est cette valeur qu'on passe en argument supplémentaire.

Slide 30

Données non scalaires

Pour accéder à `a[i]`, on doit faire `mem[fp+i*t+a]` où t est la taille d'un élément du tableau. On pourra générer un test pour le débordement des bornes du tableau. (Si on est savant, on peut même faire de l'interprétation abstraite des indices, pour ne générer ces tests que s'ils sont utiles. Par exemple, dans la séquence: `i:=1; .. a[i] ...`, ce n'est pas la peine.

Les accès aux enregistrements ressemblent à ceux des tableaux. Il faut calculer le déplacement correspondant à chaque champ, `mem[fp+k_i]`.

Slide 31

Annexes

- l’annexe Tiger d’Appel
- le manuel SPim de Larus
- le pico-R2000 de Leroy et Weis

Majeure 2
Langages et Compilation
Cours 6

Linéarisation du code
Pile, tas
Code 3 adresses

jean-jacques.levy@inria.fr

9 mars 1998

Slide 1

Les supports de cours sont en

```
http://www.polytechnique.fr/poly/edu/informatique/
http://www.polytechnique.fr/poly/~levy/
http://http.cs.berkeley.edu/~aiken/cs164temp/
http://www.cs.princeton.edu/~appel/modern/ml/
```

Livres:

- Appel, Modern Compiler Implementation in Java, Cambridge University Press, 1997
- Aho, Sethi, Ullman, Compilers: Principle, Techniques, and Tools, Addison Wesley, 1986
- Wilhelm, Maurer, Compiler Design, Addison Wesley, 1995

Slide 2

Introduction

- Linéarisation du code
- Un nouveau code intermédiaire: code à 3 adresses
- Sélection des instructions
- Piles et tas

Slide 3

Linéarisation du code

Dans une machine, en général CJUMP ne saute qu'à une adresse, l'adresse par défaut est l'instruction suivante.

Les noeuds ESEQ et CALL créent des effets de bords dans le calcul des expressions. On va les sortir des expressions. Idem pour un CALL dans un autre CALL.

Le résultat de ces transformations nous donnera une liste d'instructions suivie d'une expression pour chaque fonction.

Slide 4

Montée des noeuds ESEQ

On applique les réécritures suivantes:

ESEQ$(s_1,$ ESEQ$(s_2, e)) \rightarrow$ ESEQ(SEQ$(s_1, s_2), e)$

BINOP$(op,$ ESEQ$(s, e_1), e_2) \rightarrow$ ESEQ$(s,$ BINOP$(op, e_1, e_2))$
MEM(ESEQ$(s, e_1)) \rightarrow$ ESEQ$(s,$ MEM$(e_1))$
JUMP(ESEQ$(s, e1), t) \rightarrow$ SEQ$(s,$ JUMP$(e1, t))$
CJUMP$(op,$ ESEQ$(s, e1), e2, l1, l2) \rightarrow$
SEQ$(s,$ CJUMP$(op, e1, l1, l2))$

Si s et e_1 commutent,

BINOP$(op, e_1,$ ESEQ$(s, e_2)) \rightarrow$ ESEQ$(s,$ BINOP$(op, e_1, e_2))$
CJUMP$(op,$ ESEQ$(s, e1), e2, l1, l2) \rightarrow$
SEQ$(s,$ CJUMP$(op, e_1, e_2, l1, l2))$

Slide 5

Montée des noeuds ESEQ – bis

Si s et e_1 ne commutent pas, on prent t nouveau et:

BINOP$(op, e_1,$ ESEQ$(s, e_2)) \to$
ESEQ(MOVE(TEMPt, e_1),
ESEQ(s, BINOP(op, TEMPt, e_2)))

CJUMP$(op,$ ESEQ$(s, e1), e2, l1, l2) \to$
SEQ(MOVE(TEMPt, e_1),
SEQ(s, CJUMP(op, TEMP$t, l1, l2$))

De même, on remonte les CALL en se servant de nouveaux temporaires si nécessaire.

Dorénavant, on n'a plus que des formes canoniques d'expressions d'IC qui n'ont plus de ESEQ ou CALL à l'intérieur de leurs expressions.

Slide 6

Traces et blocs de base

Les expressions sont donc des listes d'instructions suivies d'une expression ne contenant plus d'instruction de séquencement. Les instructions sont aussi des suites d'instructions.

Un programme est une liste d'association entre étiquettes et instructions ou expressions.

Un bloc de base (*basic block*) est toute suite d'instructions telle que:

- la première instruction est LABEL
- la dernière est CJUMP ou JUMP
- il n'y a pas d'autres LABEL, CJUMP ou JUMP

On ne peut qu'entrer par le début et sortir par la fin d'un bloc de base.

Slide 7

Traces

On peut donc diviser tout le code des fonctions en suite de blocs de base. On part du haut et on s'arrête dès qu'on voit un LABEL ou un saut (conditionnel ou pas). Si on se retrouve avec un bloc ne commençant pas par une étiquette, on en rajoute une. Si on se trouve avec un bloc qui ne finit pas par un saut, on rajoute un saut à l'instruction suivante.

Une trace est une suite de blocs de base qui peuvent s'exécuter en séquence. En partant de la première instruction d'une fonction, on peut construire une trace en essayant de mettre en séquence les JUMP et leurs étiquettes correspondantes (on pourra ainsi les détruire). Si certains blocs de base ne sont pas couverts, on recommence une trace à partir d'eux.

Slide 8

Sauts conditionnels

Les traces permettent d'implanter le code linéairement en mémoire. Pour les sauts conditionnels, pour beaucoup d'entre eux, la trace aura mis une des 2 étiquettes après le saut. On peut donc changer le sens de la condition pour que ce soit le cas *false*.

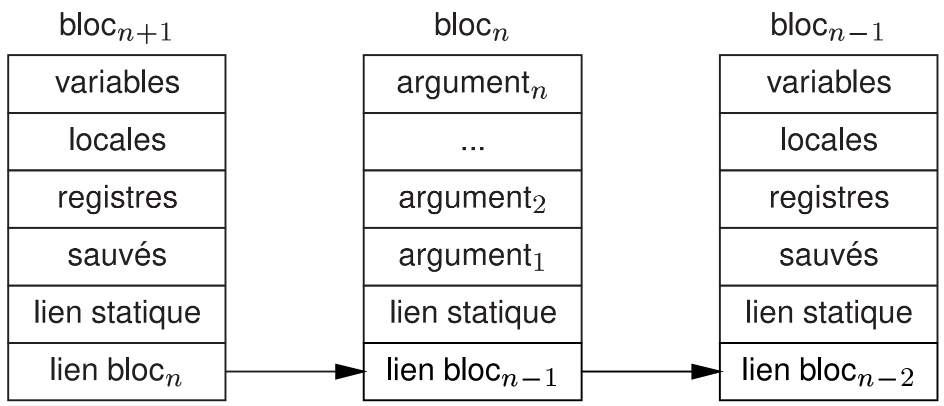
Si ce n'est pas le cas, on peut toujours introduire une nouvelle étiquette intermédiare et faire un saut dessus.

CJUMP(op, a, b, l_t, l'_f)
LABELl'_f
JUMP(NAMEl_f)

Slide 9

Pile

Les blocs sont empilés à l'entrée d'une instruction `let` ou quand on passe les arguments à une fonction. (Dans la réalité, on limite la création des blocs; par exemple les arguments sont souvents passés dans un bloc semi-ouvert).



Slide 10

Registres globaux

Toutes les machines modernes se servent de registres globaux (ie dont la durée de vie dépasse le cadre d'une fonction) pour stocker les valeurs suivantes:

`sp` sommet de pile

`fp` bloc d'activation courant

`rv` valeur de retour (d'une procédure)

Slide 11

Le RISC Mips R2000

Les machines *Reduced Instruction Set Computer* partent du principe que le temps d'une tâche est le produit $C \times T \times I$ où:

C = nombre de cycles par instruction,
T = temps par cycle (vitesse de l'horloge),
I = nombre d'intructions dans la tâche.

Le temps mis par une intruction dépend donc de C et T. Dans un Risc, on essaie de faire $C = 1$. Pour y arriver, on utilise:

le pipeline des instructions
architecture simple des load et store
load retardés
branchements retardés .

Slide 12

Le RISC Mips R2000 – bis

Les seules opérations ALU n'ont que 2 formes:

```
rd = rs op rt
rd = rs op CONSTANTE(16 bits)
```

où `rs`, `rt`, `rd` sont 32 registres. Toute instruction tient sur 32 bits; 6 bits sont réservés pour désigner l'opération (code-opération).

Les seules instructions load et store font communiquer la mémoire et les registres. (On a toujours `r0` = 0).

Slide 13

Le RISC Mips R2000 – pipeline

. .

Slide 14

Le RISC Mips R2000 – bis

```
# 238                    argc--, argv++;
   lw   $15, 72($sp)
   addu $24, $15, -1
   sw   $24, 72($sp)
   lw   $25, 76($sp)
   addu $8, $25, 4
   sw   $8, 76($sp)

# 275                    if ((tptr = index(mytty,'y')) != 0)
   la   $4, mytty
   li   $5, 121
   jal  index
   sw   $2, tptr
   lw   $10, tptr
   beq  $10, 0, $44
```

en mémoire les instructions sont réordonnancées pour permettre les `load`, `store` et sauts retardés:

Slide 15

```
# 238                  argc--, argv++;
0x40020c <main+44>: lw $t7,72($sp)
0x400210 <main+48>: lw $t9,76($sp)
0x400214 <main+52>: addiu $t8,$t7,-1
0x400218 <main+56>: addiu $t0,$t9,4
0x40021c <main+60>: sw $t0,76($sp)
0x400220 <main+64>: sw $t8,72($sp)
0x400224 <main+68>: lw $t1,72($sp)

# 275                  if ((tptr = index(mytty,'y')) != 0)
0x40037c <main+412>: lui $a0,4096
0x400380 <main+416>: addiu $a0,$a0,32272
0x400384 <main+420>: jal 0x405bf0 <index>
0x400388 <main+424>: li $a1,121
0x40038c <main+428>: sw $v0,-30568($gp)
0x400390 <main+432>: lw $t4,-30568($gp)
0x400394 <main+436>: nop
0x400398 <main+440>: beq $t4,$zero,0x4003b0 <main+464>
```

Slide 16

Le pico-RISC WL9311-pro

Le code WL9311-pro (Weis-Leroy 93, batiment 11, professionnel) est un code à 3 opérandes: destination, source, source2. La 2ème source peut aussi être une constante immédiate. On note o quand non précisé.

$$d \leftarrow s \oplus o$$

où $\oplus$ est une opération arithmétique, de test logique ou de décalage. Le registre `r0` contient toujours la valeur 0. Le nombre de registres est supposé infini pour l'instant.

Donc $d \leftarrow s + \texttt{r0}$ équivaut à faire $d \leftarrow s$.

Slide 17

Le tableau des instructions WL9311-pro

$\oplus(s, o, d)$	met dans d le résultat de l'opération entre o et s
`load`(s, o, d)	charge le registre d avec le mot à l'adresse $s + o$
`store`(s, o, d)	range le registre d dans le mot à l'adresse $s + o$
`jmp`(o, r)	saute à l'adresse o en mettant dans r l'adresse suivant l'instruction `jmp`. (*Jump and Link* du R2000)
`braz`(r, a)	saute à l'adresse a si $r = 0$,
`branz`(r, a)	saute à l'adresse a si $r \neq 0$,
`scall`(n)	appelle la fonction système n,
`stop`	arrête le programme.

Slide 18

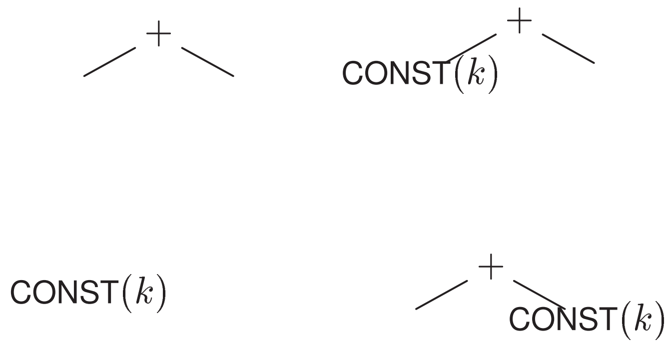
L'interface code.mli du pico-RISC

```
type registre == int;;

type opérande =
     Reg of registre
   | Imm of int;;

type instruction =
     Op of opération * registre * opérande * registre
   | Jmp of opérande * registre
   | Braz of registre * int
   | Branz of registre * int
   | Scall of int
   | Stop

and opération =
    Load | Store | Add | Mult | Sub | Div
  | And | Or | Xor | Shl | Shr
  | Slt | Sle | Seq;;

value nombre_de_registres: int
  and sp: int
  and ra: int
  and taille_du_mot: int;;
```

Slide 19

Les instructions sous-forme arborescente

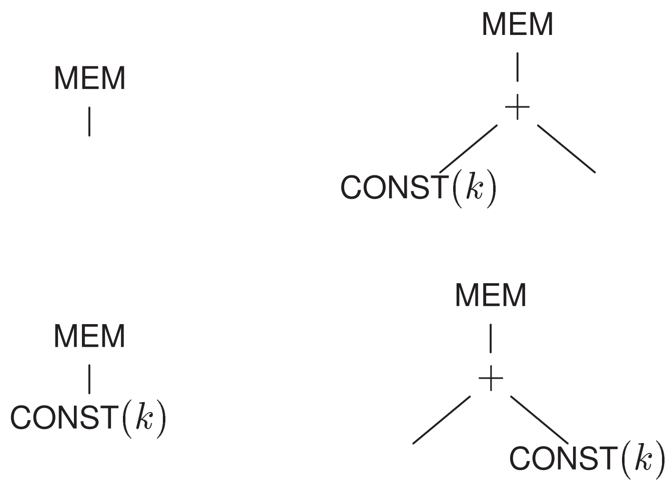
`Add` correspond aux arbres suivants d'IC:



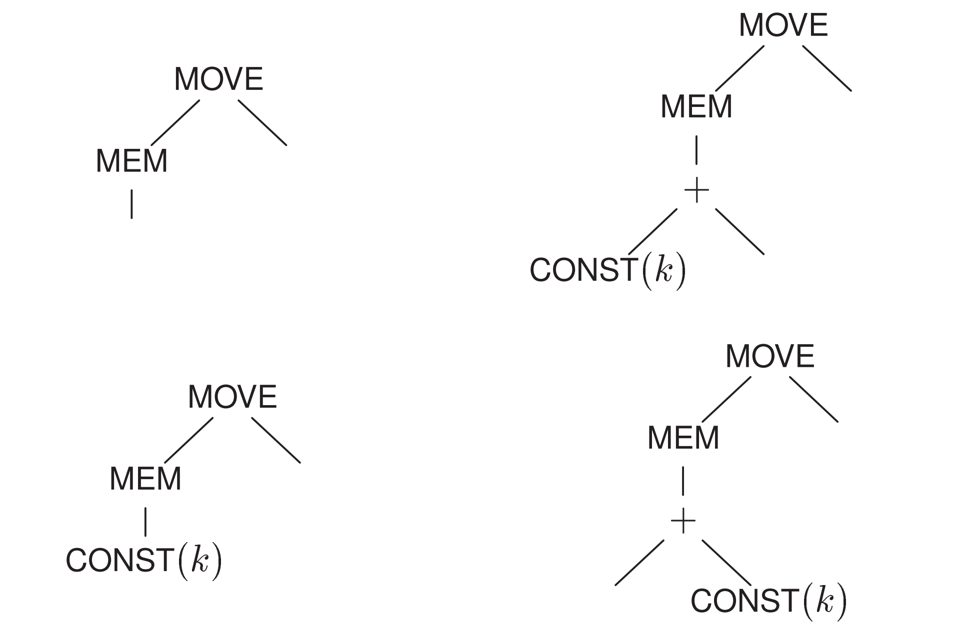
Slide 20

Les instructions sous-forme arborescente

`load` et `store` correspondent aux arbres suivants d'IC:



Slide 21



Slide 22

Sélection des instructions

On doit recouvrir toute expression de IC par des motifs correspondants aux instructions WL9311-pro. On part du sommet de l'expression en trouvant un motif correspondant à une instruction du pico-Risc, puis on descent récursivement dans les sous-arbres non explorés. Ensuite, on part des feuilles pour générer le code en remontant vers la racine.

Remarquons que la mise à plat des instructions par rapport aux expressions fait que les noeuds MEM ne peuvent se trouver qu'au dessus des expressions.

Pour les sauts, on génère le code de l'expression logique, comme pour toute expression. Le saut proprement dit se fait par les instructions `braz`, `branz`.

Slide 23

Sélection des instructions – suite

Plusieurs recouvrements sont possibles pour une même expression. Certains donnent des résultats meilleurs. Si on veut être optimal, il faut une notion de coût attachée à chaque instruction et se lancer dans un peu de programmation dynamique pour tabuler les coûts intermédiaires et arriver au meilleur coût.

Plus important, la sélection des instructions peut prendre du temps, car il faut filtrer toute expression de IC par tous les motifs (correspondant à toutes les instructions machines). Si on sait factoriser le filtrage, on peut gagner beaucoup de temps de compilation (cf Huet-Lévy (79), O'Donnell-Hoffmann (80))

En pratique, prendre le plus gros motif à chaque fois est une bonne heuristique.

Slide 24

Création de blocs d'activation

On crée les nouveaux blocs au sommet de la pile en faisant:

```
MEM[sp] := fp
fp := sp
sp := sp +/- k
```

et on met les valeurs dans chacune des entrées du nouveau bloc. Le lien sur le premier mot du bloc permet de libérer le bloc quand on le quitte.

`+/-` est un décision relevant de l'architecture. Les piles croissent dans le sens décroissant des adresses mémoire (Mips, Vax), mais dans le sens croissant sur HP9000. Attention donc à ce phénomène, car il faut changer + en – dans tous les calculs d'adresse du code intermédiaire pour accéder aux objets dans la pile.

Slide 25

Début et fin des fonctions

A l'appel, on crée un bloc dans la pile, et on y met les arguments, et on fait `jmp`$(proc,$ `ra`$)$. Le registre `ra` contient l'adresse de retour.

Le code à l'entrée et à la sortie des fonctions dépend de la machine. A l'entrée, nous sauvons quelques registres, (`ra` pour l'instant). A la sortie, on restaure `ra` si nécessaire, on met la valeur dans le registre `rv` où on met les résultats et on fait un `jmp`$(ra, 0)$.

Slide 26

Annexes

- l'annexe Tiger d'Appel
- le manuel SPim de Larus
- le pico-R2000 de Leroy et Weis

Majeure 2
Langages et Compilation
Cours 7

Durée de vie
Allocation de registres
Allocation mémoire

jean-jacques.levy@inria.fr

16 mars 1998

Slide 1

Les supports de cours sont en

http://www.polytechnique.fr/poly/edu/informatique/

http://www.polytechnique.fr/poly/ levy/

http://http.cs.berkeley.edu/ aiken/cs164temp/

http://www.cs.princeton.edu/ appel/modern/ml/

Livres:

- Appel, Modern Compiler Implementation in Java, Cambridge University Press, 1997
- Aho, Sethi, Ullman, Compilers: Principle, Techniques, and Tools, Addison Wesley, 1986
- Wilhelm, Maurer, Compiler Design, Addison Wesley, 1995

Slide 2

Introduction

- Durée de vie des variables
- Allocation de registres
- Allocation mémoire
- Recherche en informatique

Slide 3

Rappel sur les cours précédents

1. CTigre (mini Pascal avec données dynamiques)
   (a) Lexeur et table des symboles
   (b) Analyseur syntaxique
2. AST de CTigre
   (a) Résolution des noms
   (b) Allocation des variables
3. IC
   (a) Séparation des instructions et des expressions
   (b) Élimination des branchements inutiles
4. IC canonisé et linéarisé
   (a) Gestion de la pile des blocs d'activation
   (b) Protocole d'entrée et de sortie des procédures
   (c) Sélection des instructions
5. Code 3 adresses (pico-Risc)

Slide 4

Aujourd'hui

1. Code 3 adresses (pico-Risc)
   (a) Allocation des registres
2. Code R2000
   (a) Gestion de la mémoire dynamique

Slide 5

Durée de vie des variables

Pour pouvoir utiliser un même registre pour y ranger les valeurs de deux variables différentes, il faut analyser la durée de vie des variables. Exemple:

```
1       x := 0
2   l1: y := x+1
3       s := s + y
4       x := 2 * y
5       if x < N then goto l1
6       return s
```

x vit de la fin de 1 au début de 2, et de la fin de 4 à l'entrée de 2,
y vit de la sortie de 2 à l'entrée dans 4,
s vit pendant tout le programme.

Quel est le sens de la vie d'une variable?

Slide 6

Définition de la durée de vie des variables

Une variable est vivante entre sa définition et son utilisation.

Autrement dit, en considérant le graphe de contrôle d'un programme (dont les nœuds sont les instructions), une variable est vivante sur un arc, ssi cet arc est sur le chemin entre un nœud qui la définit et un nœud qui l'utilise.

Les ensembles in(n) et out(n) des variables vivantes à l'entrée et à la sortie d'un nœud n sont définis récursivement par:

$$\text{in}(n) = \text{use}(n) \cup (\text{out}(n) - \text{def}(n))$$
$$\text{out}(n) = \cup\{\text{in}(s) \mid s \in \text{succ}(n)\}$$

Slide 7

Calcul de la durée de vie des variables

Il s'agit d'une définition arrière (*backward analysis*). Pour calculer ces ensembles, on itère la définition à partir des ensembles vides jusqu'à trouver un état stationnaire. (On applique un théorème de point fixe de Tarsky; on peut aussi le justifier par la théorie des domaines de Scott et des points fixes; c'est aussi la base de la théorie de l'interprétation abstraite).

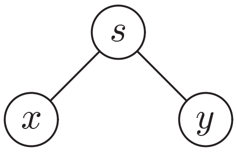
Dans notre exemple, voici les itérations:

	def	use	in	out	in	out	in	out	in	out	in
1	x					x		x,s	s	x,s	s
2	y	x			x	s,y	x,s	s,y	x,s	s,y	x,s
3	s	s,y			s,y	y	s,y	y	s,y	y,s	s,y
4	x	y			y	x	y	s,x	y,s	x,s	y,s
5		x			x	s,x	s,x	x,s	x,s	x,s	x,s
6		s			s		s		s		s

Slide 8

Graphe d'interférence

C'est un graphe non dirigé dont les noeuds sont les variables et les arcs relient les variables vivantes simultanément. Dans l'exemple précédent, ce graphe est



Intuitivement, deux variables qui interfèrent ne peuvent être allouées au même endroit.

Slide 9

Durée de vie des variables et affectations simples

```
x := 5
y := x
z := x + 2
y := 3
```

En fait, x et y peuvent être alloués au même endroit. Dans ce cas, on ne met pas d'interférence entre les deux qui contiennent la même valeur. Mais cet arc sera rajouté plus tard, si

```
x := 5
y := x
y := 3
z := x + 2
```

Slide 10

Allocation de registres

On a supposé un nombre infini de registres dans le code 3 adresses, mais dans la réalité, on n'a que k registres ($k = 16$, $k = 32$, etc). Comment donc allouer les variables dans cet ensemble fini de ressources? En 1980, Chaitin fait une percée en réduisant cette question à un problème de coloriage de graphes.

En effet, il s'agit de colorier avec k couleurs les noeuds du graphe d'interférence avec la seule règle: ne pas mettre la même couleur à deux noeuds reliés par un arc.

Slide 11

Exemple 1 – cf le livre d'Appel

.

Slide 12

Exemple 2 – cf le livre d'Appel

.

Slide 13

Heuristique simple pour le coloriage

On choisit un noeud n qui n'a pas plus que $k - 1$ voisins. On l'empile en l'enlevant du graphe. On colorie le graphe restant. Puis on donne à n une couleur différente de celle de ses $k - 1$ voisins, ce qui est possible puisqu'on dispose de k couleurs.

Cette procédure est récursive. On peut remarquer qu'un noeud avec plus k voisins peut se retrouver plus tard avec moins que $k - 1$ voisins. Mais il se peut qu'on arrive à une situation où il n'y a plus que des noeuds avec plus de k voisins.

On empile un tel noeud marqué *spill* (débordement) en l'enlevant du graphe, et on continue pour explorer tous les possibles *spill*. Il se peut qu'au retour on sache colorier ce noeud si deux voisins ont même couleur.

Slide 14

Heuristique simple pour le coloriage – bis

Si on n'arrive pas à colorier les noeuds *spill*, on les alloue dans la pile, en changeant le programme pour mettre les instructions d'accès à la pile.

Et on relance la procédure de coloriage. En principe, ca converge alors.

Slide 15

Elimination des affectations inutiles

Deux variables x et y qui n'interfèrent pas mais qui sont reliées par `x:= y` peuvent être allouées au même endroit. Pour éliminer cette instruction inutile, on peut confondre les 2 noeuds dans le graphe d'interférence, mais alors il y a plus de contraintes à ce noeud, ce qui n'est pas grave si l'ensemble de ses voisins n'a lui-même pas plus que k voisins de degrés plus grands que k.

On peut mélanger cette stratégie au coloriage en traitant d'abord les noeuds non reliés par un `move` élémentaire. Puis en essayant de confondre les autres.

Slide 16

Noeuds précolorés et procédures

Certaines variables sont pré-colorées: `fp`, `sp`, `rv`. Attention qu'il n'y en ait pas trop, sinon on risque de ne pas savoir colorier le graphe.

Certaines variables doivent rester vivantes pendant un appel de fonctions. D'habitude, on les sauve dans la pile à l'entrée de la procédure et on les restitue à la sortie. Ce qui peut être obtenu automatiquement par l'allocateur de registres en déclarant que la procédure redéfinit toutes les variables à sauver de l'appelant.

Slide 17

Allocation des registres dans les expressions

Si on n'a que des expressions (et leur AST en arbre), une technique simple d'allocation est due à Ershov.

Pour évaluer $e \otimes e'$, si on évalue e avec n registres et e' avec n' registres, on évalue $e + e'$ en $\max(m, 1 + n)$ ou $\max(1 + m, n)$ registres selon qu'on calcule e ou e' d'abord.

On a donc intérêt à évaluer d'abord l'expression qui demande le plus de registres.

Slide 18

Allocation des données dynamiques

Les variables déclarées par des `let` ou arguments de fonctions ont une durée de vie qui n'excède pas celle du bloc où elles sont définies. On peut donc les allouer dans la pile.

D'autres peuvent être crées dynamiquement à tout moment (enregistrements et tableaux en CTigre, `new` en Pascal ou Java, `malloc` en C), et on peut continuer à les référencer en dehors du bloc où elles sont apparues. On les alloue donc dans le tas (*heap*).

A quel moment peut-on les détruire?

Slide 19

Glaneur de cellules – *garbage collector*

On peut laisser la déallocation des données dynamiques à la charge du programmeur (`dispose` en Pascal ou `free` en C).

Mieux et plus moderne: le système d'exécution du langage le fait automatiquement (Lisp, ML, Mesa, Modula-3, Java).

Il y a un *garbage collector* ou glaneur de cellule, GC pour simplifier, qui fait la déallocation implicite.

Tous les langages modernes ont un GC.

Slide 20

Mark and Sweep

Le GC est souvent appelé quand il n'y a plus de place dans le tas. Il existe alors un ensemble de *racines* dans le programme en cours d'exécution, ie toutes les variables de la pile qui sont des références vers le tas.

Dans la méthode Mark and Sweep, on marque tous les noeuds accessibles à partir de la racine par une exploration en profondeur d'abord. Puis on scanne tout le tas, en rangeant les noeuds non marqués dans une liste des cellules libres, *free-list*.

Pour marquer, on peut utiliser un vecteur de bits. Pour l'exploration, si on n'a pas de place pour mettre dans la pile, on retourne les pointeurs pendant l'exploration.

Slide 21

Stop and copy

Mark and Sweep ne compactifie pas la mémoire utilisée. Avec Stop and Copy, on coupe la mémoire en 2 parties. Et on alloue dans une des deux moitiés. Quand cette moitié `from-space` est pleine, en partant des racines, on recopie les cellules utilisées dans l'autre moitié `to-space`. Pour faire la copie, on laisse un renvoi dans `from-space` à chaque cellule recopiée vers `to-space`.

Il y a bien d'autres méthodes: incrémentales, concurrentes, etc. Depuis 5-10 ans, on ajoute des générations aux GC traitées différemment, car souvent il n'y a que 2 types de données dynamiques: les très éphémères et les permanentes.

CAML a deux générations et le GC majeur est en Mark en Sweep.

Slide 22

Ce qu'on n'a pas traité

Types (statiques, dynamiques)

Modules

Données persistantes

Les 50 méthodes d'optimisation:

propagation des constantes

constant folding

peephole optimisation

dépliage des boucles

ordonnancement des instructions

analyse de localité et caches

élimination du code inaccessible,

parallélisme, etc

Slide 23

Comment simplifier le projet

Ne pas faire de définitions de fonctions imbriquées (comme en C). Plus besoin du lien statique des blocs, 2 espaces de variables: globales et locales.

Générer directement le code 3 adresses sans passer par IC. Alors on n'optimise pas les branchements et on crée des étiquettes au fur et à mesure qu'on compile le code. Le choix des instructions n'est pas non plus optimisés.

Allouer les registres de manière triviale, ie dans l'ordre de leur apparition, en mettant en pile dès qu'il y a débordement (*spill*).

C'est ce qu'y est fait dans le livre «le Langage Caml», Weis-Leroy 93.

Slide 24

Comment améliorer le projet

Tracer les erreurs de syntaxe, en gardant le numéro de caractère dans les lexèmes, pour pouvoir localiser les messages d'erreur du compilateur.

Mettre un GC.

Faire les objets (cf cours 8)

Faire l'allocation de registres qui supprime les affectations inutiles.

Générer de l'assembleur MIPS et l'exécuter soit sur un IRIS, decstation (non alpha station), où sur le simulateur Spim.

Slide 25

Conclusions sur langages et compilation

Un langage s'impose parce qu'il comble un vide. (Aiken)

Un mauvais langage peut s'imposer. Fortran a été le langage de l'analyse numérique, remplacé par C++. Java est celui de la programmation du Web. Les deux derniers ont été crées pour faire autre chose.

Beaucoup de confusion dans la terminologie des langages de programmation. Objets veut souvent dire modulaire ou abstraction, mais il existe aussi d'autres mécanismes pour y arriver: modules, fonctions.

Les méthodes d'analyse statique peuvent aussi servir à la certification du logiciel, au delà de l'optimisation. De manière générale, peu a encore été fait comme aides à la mise au point, souvent arrétée par des problèmes d'efficacité.

Encore un bel avenir pour les langages spécialisés: réactif, agents, circuits.

Slide 26

2 exemples de DEA: DEA Algorithmique

1. Algorithmique et combinatoire des mots, [J. Berstel]
2. Géométrie computationnelle, [J.-D. Boissonnat]
3. Introduction au calcul parallèle et distribué, [M. Morvan]
4. Algorithmes probabilistes, [J.-M. Steyaert]

. Pratique du calcul formel.

Filières, responsables et cours

1. Analyse d'algorithmes [J.-M. Steyaert]:
2. Automates et mots [J.-E. Pin]:
3. Calcul formel [D. Lazard]:
4. Combinatoire [R. Cori]:
5. Complexité, codage et cryptographie [J. Stern]:
6. Géométrie, images et robotique [O. Faugeras]:
7. Parallélisme et concurrence [A. Petit]:

. Conception de circuits, [P. Bertin, J. Vuillemin].

`http://www.polytechnique.fr/Ens/prog/3cycle/DEAP/algo.html`

Slide 27

Slide 28

Autre exemple 1997-98: DEA Sémantique, Preuves et Programmation

1. Les termes en logique du premier ordre, [J.P. Jouannaud]
2. Lambda calcul, [T. Hardin]
3. Preuves constructives, [G. Dowek]
4. Sémantique dénotationnelle, [R. Di Cosmo]
5. Typage et programmation, [M. Mauny - D. Rémy - X. Leroy]

Filières, responsables et cours

1. Langages, [G. Cousineau]
Langages distribués, [C. Queinnec];
Sous-typage et langages à objets, [G. Castagna - F. Rouaix - D. Rémy]
Compilation des langages fonctionnels et impératifs, [X. Leroy]
Programmation logique avec contraintes et systèmes concurrents, [F. Fages - L.Fribourg]

2. Modèles Sémantiques, [P.-L. Curien]
Types, catégories et domaines, [P.-L. Curien-G. Longo]
Logique linéaire, [R. Di Cosmo]
Concurrence et Communication, [G. Gonthier - J.-J. Lévy]
Modèles algébriques des processus et méthodes de vérification, [Ph. Schnoebelen]

3. Preuves et spécifications, [J.-P. Jouannaud]
Calcul des constructions inductives, [C. Paulin - B. Werner]
La méthode B, [V. Donzeau-Gouge, M. Simonot]
Preuve par des techniques d'automates, [H. Comon]
Réécriture et preuves, [J.-P. Jouannaud - C. Marche - M. Rusinowitch]

4. Sémantique et Interprétation abstraite, [R. Cousot]
Fondements de l'interprétation abstraite, [P. Cousot]
Interprétation abstraite: applications, [A. Deutsch - P. Granger - R. Cridlig]
Langages objets, contraintes et typage, [F. Bourdoncle - B. Monsuez]
Parallélisme : sémantique et preuve, [R. Cousot - E. Goubault - I. Mackie]

`http://www.ens.fr/ dicosmo/DEA/SPP/`

Slide 29

Laboratoires de recherche

CNET Issy, Lannion	Labri Bordeaux
CNRS	Imag Grenoble
INRIA, 5 Unités de recherche	Marseille Lumigny
ENS	Paris 6
ENS Cachan	Paris 7, (Logique)
ENS Lyon	Marne la Vallée
ENPC	ORSAY
ENSMP	Saint Denis
ENST	Léonard de Vinci
LIX	CNAM
Thomson LCR	Lucent, AT&T Bell labs
Alcatel Marcoussis	Bellcore
GENSET	Digital SRC, WRL
ILOG	IBM Almaden, Yorktown
O2 Technology	Microsoft Research Redmond, Cambridge
Chorus Systèmes	Xerox PARC
Xerox Grenoble	HPlabs
Web Consortium	SUN

Slide 30

Domaines de recherche (exemple de l'INRIA)

Thèmes

1. Réseaux et systèmes
2. Génie logiciel et calcul symbolique
3. Interaction homme-machine, images, données, connaissances
4. Simulation et optimisation de systèmes complexes

Action de développements

DYADE : conception de systèmes d'information avancés

GÉNIE : science de l'information et ingénierie concourante

MÉDIACULTURE : système multimédia distribué pour la documentation culturelle
PRAXITELE : transport urbain public individuel

PRÉVISIA : interopérabilité des systèmes d'information
SYNCHRONE : environnement de développement de systèmes temps-réel
W3C : les services d'information sur Internet
WEBTOOLS : outils logiciels pour le World Wide Web

`http://www.inria.fr/Recherche/activites-fra.html`

Slide 31

Thème 1

Parallélisme et architecture :

APACHE - Algorithmique parallèle et partage de charge
API - Architectures parallèles intégrées
CAPS - Compilation, architectures parallèles et systèmes
PAMPA - Environnement de programmation des architectures massivement parall èles
ReMaP - Régularité et parallélisme massif
SLOOP - Simulation, langages orientés objets et parallélisme

Réseaux, systèmes, évaluation de performances :

MÉVAL - Modélisation et évaluation des systèmes informatiques
MISTRAL - Modélisation en informatique et systèmes de télécommuni cation : recherche et applications logicielles
MODEL - Modélisation de systèmes aléatoires
RESEDAS - Outils logiciels pour les télécommunications et les systèmes d istribués
RODEO - Réseaux a‘ haut débit, réseaux ouverts
SATURNE - Systèmes répartis tolérant les fautes et les intrusions
SIRAC - Systèmes informatiques répartis pour applications coopératives
SOLIDOR - Architectures et systèmes distribués extensibles, tolérance au x fautes, programmation objet
SOR - Systèmes d'objets répartis

Programmation distribuée et temps-réel :

ADP - Algorithmes répartis et protocoles
EPM-PATR - Environnement de programmation pour applications temps réel
MEIJE - Parallélisme, synchronisation et temps réel
PARA - Parallélisme, mobilité
REFLECS - Systèmes informatiques répartis temps réel tolérant les fautes
SPECTRE - Spécification et programmation des systèmes communicants et tem ps réel

Slide 32

Thème 2

Sémantique et programmation :

CALLIGRAMME - Logique linéaire, réseaux de démonstration et grammaires cat égorielles
COQ - Spécifications et preuves de programmes
CRISTAL - Programmation typée, modularité et compilation
CROAP - Conception et réalisation d'outils d'aide a' la programmation
EURECA - Preuve, calcul symbolique et logique
LANDE - Langages déclaratifs
LOCO - Programmation logique avec contraintes
OSCAR - Outils syntaxiques pour la construction et l'analyse de programmes
PROTHEO - Contraintes, déduction automatique et preuves de propriétés de logiciels

Algorithmique et calcul formel :

ALGO - Algorithmes
CODES - Codes et protection de l'information
PRISME - Géométrie, algorithmes et robotique
SAFIR - Systèmes algébriques formels pour l'industrie et la recherche

Slide 33

Thème 3

Bases de données, bases de connaissances, systèmes cognitifs :

ACACIA - Acquisition des connaissances pour l'assistance a' la conception par interaction entre agents
AIRELLE - Représentations et langages
ATOLL - Atelier d'outils logiciels pour le langage naturel
DIALOGUE - Dialogue homme-machine a' forte composante langagière
OPERA - Outils pour les documents électroniques : recherche et applications
ORION - Modélisation des connaissances pour l'automatisation des tâches
PSYCHO-ERGO - Psychologie ergonomique pour l'informatique
REPCO - Représentation des connaissances
RODIN - Systèmes de bases de données
SHERPA - Bases de connaissances a' objets
SHOOD - Méthodes et outils pour l'intégration de systèmes industriels
SYCO - Systèmes de compréhension et bases de connaissances
VERSO - Bases de données

Vision, analyse et synthèse d'images :

AIR - Traitement d'image et données satellites dynamiques
EPIDAURE - Imagerie et robotique médicale
iMAGIS - Modèles, algorithmes, géométrie pour le graphique et l'image de synthèse
ISA - Image, synthèse, analyse
MOVI - Modélisation, localisation, reconnaissance et interprétation en vis ion parordinateur
PASTIS - Analyse de scènes et traitement d'images symboliques
ROBOTVIS - Vision par ordinateur et robotique
SHARP - Programmation automatique et systèmes décisionnels en robotique
SIAMES - Synthèse d'image, animation, modélisation et simulation
SYNTIM - Analyse et synthèse d'images
TEMIS - Traitement, exploitation et modélisation d'images séquentielles

Slide 34

Thème 4

Automatique, robotique, signal :

AS - Automatique et signal
ATGC - Action transversale génome et calcul
BIP - Conception et contrôle de robots marcheurs et applications
COMORE - Contrôle et modélisation de ressources renouvelables
CONGÉ - Contrôle géométrique des systèmes non linéaires
FRACTALES - Approche fractale pour l'analyse et la modélisation de signaux complexes
ICARE - Instrumentation, commande et architecture des robots évolués
MÉTA 2 - Méta automatique et méthodes de l'automatique
MIAOU - Mathématiques et informatique de l'automatique et de l'optimisation pour l'utilisateur
PROMATH - Programmation mathématique
SAGEP - Simulation, analyse et gestion des systèmes de production
SOSSO - Applications et outils de l'automatique
SYSTOL - Modélisation statistique et applications biomédicales

Modélisation et calcul scientifique :

ALADIN - Algorithmes adaptés au calcul numérique intensif
CAIMAN - Calcul scientifique, modélisation et analyse numérique
ESTIME - Estimation de paramètres
IDOPT - Identification et optimisation de systèmes en physique et en environnement
M3N - Multi-modèles et méthodes numériques
MODULEF - Méthodes et outils pour le calcul scientifique
NUMATH - Analyse mathématique et traitement numérique de modèles non linéaires
OMEGA - Méthodes numériques probabilistes
ONDES - Modélisation, analyse, simulation des équations des ondes
SINUS - Simulation numérique dans les sciences de l'ingénieur
SYSDYS - Systèmes dynamiques stochastiques

Slide 35

Annexes

- l’annexe Tiger d’Appel
- le manuel SPim de Larus
- le pico-R2000 de Leroy et Weis