Projet 2

présentation du cours

daniel.hirschkoff@ens-lyon.fr

Projet 2

- intervenants
  Henning Basold Daniel Hirschkoff Bertrand Simon
- 2h par semaine sur machine (+2h parfois en amphi)
- programmation en **Caml**
  - en binôme
  - exigences adaptées

    quantité de travail $\simeq$ équivalente suivant le niveau

    `http://perso.ens-lyon.fr/daniel.hirschkoff/P2`

- vendredi (02/02) : TP
  salle E001, n'hésitez pas à venir avec votre ordinateur

Contenu

- cours projet : rôle essentiel joué par la *pratique*

- programmer en Caml
- programmer modulairement

  génie logiciel

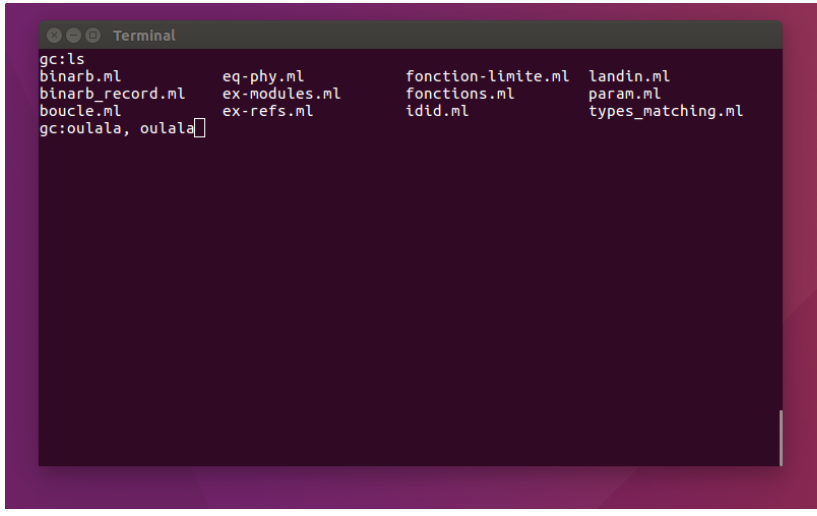
- organiser le code, organiser le travail
- “faire vivre” du code

- 4 rendus au cours du semestre
  - un premier programme
  - un projet, qui “vit” sur plusieurs rendus

    programmes qui manipulent des programmes
    interprètes, un peu de compilation

Au jour le jour

- interagir dans un terminal
- programmer, debugger
  tester, debugger
- utiliser `git`
  pour partager des fichiers
- respecter des échéances



Éléments de Caml

Survol de Caml

- fonctions
- types
- programmation impérative
- modules (2 mots)

Fonctions

- DÉMO `fonctions.ml`
- `fun x -> BLA`

  `let f x y = BLI`, c'est de la convivialité

  `let f = fun x -> (fun y -> BLI)`
- la coquetterie `f x` plutôt que `f(x)`
- l'"équilibre"

  `(f x) y`

  t_1 `->` `(` t_2 `->` t_3 `)`

  et donc, comprendre `f (x y)` et $(t_1 \texttt{->} t_2) \texttt{->} t_3$

`let .. in`

```
g (f 3) + h (f 3)
```

```
let x = f 3 in
  g x + h x
```

LA forme de convivialité qui irrigue tout Caml

↘ *en première approche du moins*

en Caml, on passe son temps

- à définir et utiliser des fonctions
- à passer par des `let`

Types

il n'y a "que" ->

```
int -> bool
                    int -> string -> string
```

- ouais bon : `int char string bool unit float`
- ouais bon : $t_1 * t_2$ (produit cartésien) `{n:int; s:string}`

  en passant, $(t_1 * t_2) \text{->} t_3$ c'est comme $t_1 \text{->} t_2 \text{->} t_3$
- ouais bon : et `type t = Leaf of int | Node of t*t` ?

  cf. plus loin

Se poser des questions : typage, terminaison

notez la couleur de fond, typique des moments de digression ↗

- peut-on écrire des fonctions qui ne terminent pas ?
  - dans le *cœur fonctionnel* de Caml, sans `letrec`,
    toutes les fonctions terminent
  - sans `letrec` mais avec
    - des références DÉMO `landin.ml`
    - des types somme, des exceptions DÉMO `boucle.ml`
- peut-on écrire des fonctions qui ne sont pas typables?

  sans les entiers, les booléens, les listes, les chaînes de caractères, . . .

  `let f x = x x` **Démo** `idid.ml`

- bilan: dans le cœur fonctionnel de Caml,
  - on peut écrire des fonctions que Caml n'aime pas
  - on ne peut pas écrire des fonctions qui se comportent mal (boucle infinie)

Types somme et filtrage

Types somme et filtrage

- structures de données habituelles en informatique

  DÉMO `binarb.ml`

- correspondance entre une définition de type somme et les *grammaires* (cf. FDI)

```
type ar  = Const of int
         | Plus of ar*ar
         | Mult of ar*ar
         | IfEqZero of ar*ar*ar
```

$$
\begin{aligned}
A &\to C_{(k)} \\
A &\to A{+}A \\
A &\to A{*}A \\
A &\to \text{if A then A else A}
\end{aligned}
$$

Types somme et filtrage, version bas niveau

sans types sommes et sans `match...with`

DÉMO `binarb_record.ml`

- les enregistrements ne sont en principe que des produits (n-uples) avec des champs *nommés*
- on a le droit aussi de rendre les champs `mutables`

  (cf. les `refs`)
- presque ce que l'on ferait en C, presque ce qui tourne vraiment "en dessous"

```
struct arb = {int valeur;
              struct arb *gauche;
              struct arb *droite};
```

en C:

- `noeud_singleton` *alloue de la mémoire* (`malloc`)
- une feuille est représentée par un pointeur nul

Formes du filtrage

- le filtrage, c'est revenir à la définition du type somme: *un cas par constructeur*

  `| x::y::l -> ...` c'est

  "`| x::xs -> (match xs with | y::l -> ...)`"

  `| Div(a,b) when b>0 ->` c'est

  "`| Div(a,b) -> if b>0 then...`"

  `| _ -> ...`

- le "vrai truc", c'est

  ```
  | [] -> ...
  | x::xs -> ...
  ```

Types somme et filtrage — à retenir

- constructeurs : MAJUSCULE au début
- un cas de filtrage par constructeur

```
match e with
  | C (x,y) -> ···
```

```
let f = function
   | C (x,y) -> ···

let (e1,e2) = c in ···

 |_ -> ···
```

c'est de la convivialité

- typage, un peu moins évident :

  chaque constructeur (pour le type ***t***)
  a un type de la forme $(t_1 * \cdots * t_k)$ -> ***t***
  - ***t*** : le type auquel est rattaché le constructeur
  - pas de t_1 -> t_2 -> ***t***

Questionnaire

se binômer au plus tard vendredi !

Références, programmation impérative

Programmation impérative en Caml

- références : `ref` `!` `:=` `;` DÉMO `ex-refs.ml`

  dans `a := !a+1`, il y a `a :=` et `!a`
  - `a :=` .. aller écrire en `a`
  - `!a` lire ce qui est stocké en `a`

- `;` s'apparente à de la convivialité

```
print_string "hop"; f x
let _ = print_string "hop" in f x

let _ = run_main_loop()
```

Programmation impérative en Caml, suite

- en Caml, l'exécution d'un programme renvoie toujours quelque chose ... ou diverge

  `a := !a +1` renvoie `() : unit` *(quelque chose par défaut)*

  de même pour `print_int : int -> unit`

- `unit` pour retarder le calcul :

```
let r = ref 3
let affiche_r = print_int !r      affiche_r : unit

let affiche_r = fun () -> print_int !r
                                  affiche_r : unit->unit
```

et aussi

```
for i = 1 to 3 do print_int i done
123- :  unit = ()

let inf = fun () -> while true do print_string "ha" done
inf :  unit -> unit
```

Types et types sommes, digression

on voit volontiers les types comme des *ensembles* de valeurs

- qui a le type t_1->t_2 ?
  - des fonctions
  - intuition venant des maths : espaces fonctionnels

    le produit cartésien pour représenter les fonctions
  - qu'en pense Caml ?
    - comparer les fonctions
    - fonctions non totales
    - fonctions qui "modifient le monde" (effets de bord)
- types sommes
  - des ensembles de *valeurs* (listes, arbres, etc.)
  - quel est le statut de ces "ensembles" ?
    - solutions d'équations récursives
    - sans `->` : équations récursives exprimées par des polynômes

Modules

Modules en deux mots

- des conventions
  - le code Caml est écrit dans des fichiers `machin.ml` (minuscule)

    par exemple

    `machin.ml`
    ```
    let f x y = x*3+y
    ```

- dans un autre fichier, `truc.ml`, on peut
  - écrire `Machin.f 3 5` (notez la majuscule)
  - faire `open Machin` (en tête de fichier)
    et écrire `f 3 5`
- automatisation de tout ça, dépendances : `Makefile`

  on aborde cela par la pratique `make`
- enjeux importants dans le développement d'un programme
  - structuration en fichiers/modules
  - dépendances entre modules
  - NB: les modules, c'est ce que propose Caml

    autres langages, autres approches

Analyses lexicale et syntaxique

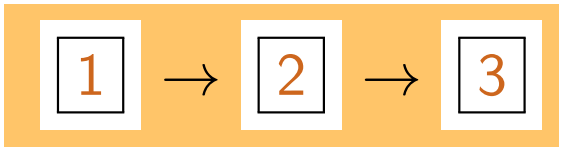
et quelques éléments de compilation

Intepréter / compiler

- **interprète:** implémentation de la sémantique opérationnelle
  exécuter le programme
- **compilateur:** traduction
  traduire (en préservant le sens)
  (p.ex. IMP$\rightsquigarrow$ *assembleur)*

interprètes et compilateurs sont des programmes manipulant des programmes

Un compilateur

- traducteur de code à code (de *fichier source* à *fichier objet*)
- anatomie sommaire 

  1. **front end**
     du fichier de texte à une représentation arborescente
     `"let x = 3 in (f x)+2"`
     ou plutôt
     `['l';'e';'t';' ';'x';' ';'=';' ';'3';' ';'i';'n';' ';' ';'(';'f';' ';'x';')';'+';'2';'\n']`

     `Let( Var "x", Cst 3, Add(App(Var "f", Var "x"), Cst 2) )`
  2. des tas de **transformations** (*représentations intermédiaires*)
  3. **back end**
     génération de code: d'une représentation arborescente à un fichier de texte

     ```
     [Push(rx);Set(rj,f_addr);Call;Pop;Set(r0,2);Add]
     ```
     ```
     start:   push(rx);
              set(rj,f_addr);
              call;
              pop;
              set(r0,2);
              add;
     ```

- "tout" est dans l'étape **2**: analyses, transformations, réécritures, optimisations, ... *cf. cours de M1*

Les deux étapes dans le front end

- analyse lexicale

  flot de caractères (source) → flot de *lexèmes*

  - lexème (*token*): "atome" du langage
  - typiquement:
    - mots-clefs (`let`, `begin`, `while`,...)
    - symboles réservés (`(`, `+`, `;;`, `;`,...)
    - identificateurs (`f`, `toto`, ...)

  ainsi `32*52+(let x = 5 in x*x)`

  → `INT(32), MULT, INT(52), ADD, LPAREN, LET, ID("x"), EGAL, INT(5), IN, ID("x"), MULT, ID("x"), RPAREN`

  (`INT` et `ID` ont un *attribut*, entier et chaîne de caractères respectivement)

- analyse syntaxique

  flot de lexèmes → *arbre de syntaxe abstraite*

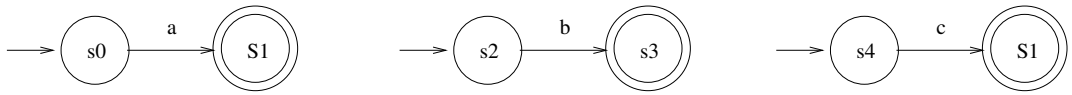
  → `Add( Mult(Int(32), Int(52)),`
  `Let("x", Int(5), Mult(Var("x"), Var("x"))) )`

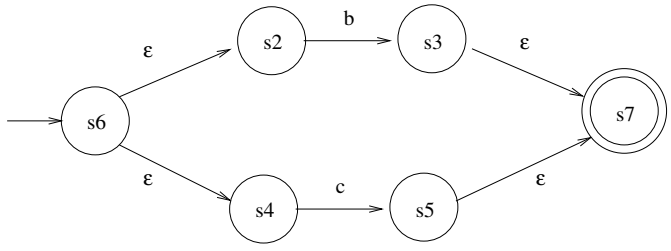
  - étape intermédiaire : arbre d'analyse syntaxique *(parse tree)*

Analyse lexicale : les lexèmes

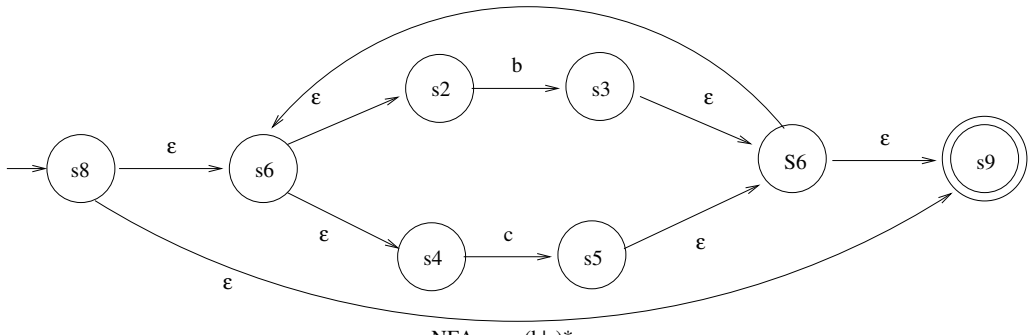
- chaque lexème est décrit par une *expression régulière*
- principaux éléments (syntaxe de `ocamllex`) :
  - caractère `'e'`, `'$'`, chaîne de caractères `"else"`
  - intervalle `['0'-'9']` *(← un chiffre)*
  - disjonction (de caractères)
    `['\t' ' ']` *(tabulation ou espace)*
  - juxtaposition `['A'-'Z']['a'-'z' 'A'-'Z']`
    (mot de 2 lettres commençant par une majuscule)
  - répétitions: `+` signifie au moins 1, `*` zéro ou plus
    `['a'-'z']+['a'-'z' '0'-'9']*`
    (ça commence par une lettre puis des lettres ou des chiffres)
  - disjonction `a* | b*`
- en sortie de l'analyse lexicale : des lexèmes (*mots*)

Expression régulière $\leftrightarrow$ automate non déterministe

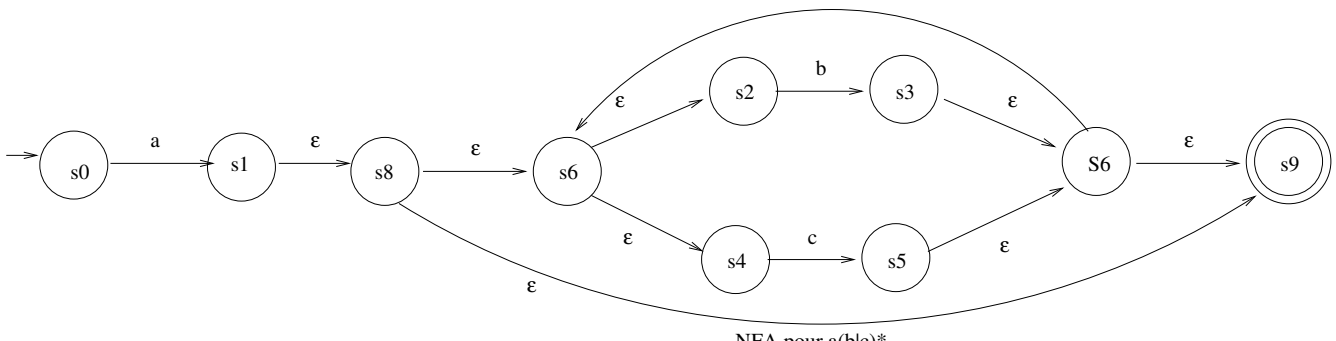




NFA pour b|c



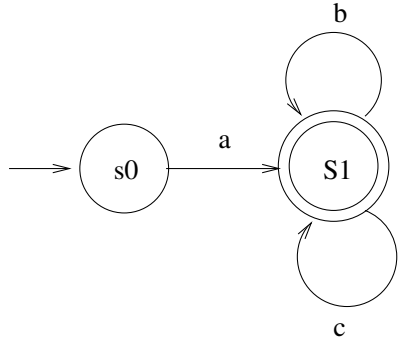
NFA pour (b|c)*



NFA pour a(b|c)*

Déterminisation, minimisation

- à partir de l'automate du transparent précédent, on dispose de procédures pour *déterminiser* l'automate (explosion du nombre d'états), puis le *minimiser*
- on aboutit à



- comment implémenter l'automate résultant?
  - une table (très creuse)

état	a	b	c	d	e
e1	-	e2	e3	-	-
e2	e4	-	-	-	-
e3	-	-	e3	-	-

  - éliminer les états: un plat de spaghetti, fait de `if` et de `goto`

Analyse lexicale : concrètement, `ocamllex`

- on décrit le **dictionnaire**

  ensemble d'expressions régulières,
  auxquelles on associe un nom *(avec éventuellement un attribut)*
  un lexème

```
| "let"              { LET }
| "in"               { IN }
| ['0'-'9']+ as s    { INT (int_of_string s) }
```

- "magiquement", on obtient un programme qui reconnaît les mots du dictionnaire (et proteste sinon)
- règles à savoir
  - le mot le plus long qui peut être reconnu l'est

    `THEN` n'est pas reconnu comme la concaténation de `THE` et de `N`
  - la première règle qui s'applique est appliquée

Démo

Analyse syntaxique

- l'analyse syntaxique se fonde sur une approche plus puissante : **règles de grammaire**
  - les règles de grammaire font intervenir les lexèmes et des "variables" (les non terminaux)
  - exemple de grammaire:

    $E \quad ::= \quad K \mid E + E \mid E * E \mid (E) \mid \texttt{let}\ Id = E\ \texttt{in}\ E$

    - E non terminal (il peut y en avoir plusieurs)
    - $K, \texttt{let}, Id, +, *, (,), \texttt{in}, =$ lexèmes

présentation alternative:

$E \rightarrow K \qquad E \rightarrow E + E \qquad E \rightarrow E * E \qquad E \rightarrow (E) \qquad E \rightarrow \texttt{let}\ Id = E\ \texttt{in}\ + E$

- analyse lexicale : du flot de caractères au flot de lexèmes
- analyseur syntaxique (ou *parser*) : applique les règles de grammaire pour reconnaître une séquence de lexèmes
  - on change la structure : un flot (de lexèmes) devient un arbre
  - on construit des *phrases* à partir de *mots*

Ce que fait le parser

$E \quad ::= \quad E + E \;\mid\; E * E \;\mid\; (E) \;\mid\; a \;\mid\; b \;\mid\; c$ `a+b*c`

pile	entrée	action
$\$$	$a + b * c\$$	shift
$\$a$	$+b * c\$$	reduce : $E \to a$
$\$E$	$+b * c\$$	shift
$\$E+$	$b * c\$$	shift
$\$E + b$	$*c\$$	reduce : $E \to b$
$\$E + E$	$*c\$$	shift **(remarquablement malin)**
$\$E + E*$	$c\$$	shift
$\$E + E * c$	$\$$	reduce : $E \to c$
$\$E + E * E$	$\$$	reduce : $E \to E * E$
$\$E + E$	$\$$	reduce : $E \to E + E$
$\$E$	$\$$	accept

- à la fin, on a un arbre `add(id(a),mul(id(b),id(c)))`

Concrètement, `ocamlyacc`

- analyse syntaxique : on écrit la grammaire, et on associe à chaque règle une *action sémantique* *(construction de l'arbre)*
- DÉMO
- en pratique, avec `ocamlyacc`, on ôte les ambiguïtés
  - en réécrivant la grammaire (comme en FDI)
  - ou en "bricolant"

    cf. exemples sur la page du cours
- on trouve *"partout"* les outils pour les analyses lexicale et syntaxique

  Lex & Yacc, Flex & Bison, ...

```
%{
(* ---- preambule: ici du code Caml ---- *)

open Expr   (* rappel: dans expr.ml:
               type expr = Const of int | Add of expr*expr | Mull of expr*expr *)

%}
/* description des lexemes */

%token <int> INT        /* le lexeme INT a un attribut entier */
%token PLUS TIMES
%token LPAREN RPAREN
%token EOL              /* retour a la ligne */

%left PLUS
%left TIMES


%start main             /* "start" signale le point d'entree: c'est ici main */
%type <Expr.expr> main     /* on _doit_ donner le type du point d'entree */

%%
    /* ---- debut des regles de grammaire ---- */
                           /* a droite, les valeurs associees */
main:                      /* le point d'entree */
    expr EOL               { $1 }  /* on veut reconnaitre un "expr" */
;
expr:                      /* regles de grammaire pour les expressions */
  | INT                    { Const $1 }
  | LPAREN expr RPAREN     { $2 } /* on recupere le deuxieme element */
  | expr PLUS expr         { Add($1,$3) }
  | expr TIMES expr        { Mul($1,$3) }
;
```

```
%{ // useful functions.
#include <stdlib.h>
#include <stdio.h>
#include <iostream>
#include <fstream> //for dag output
#include "ast.h"

using namespace std;

int     line_number     = 1;    /* number of current source line */
extern int      yylex();        /* lexical analyzer generated from lex.l */
extern char     *yytext;        /* last token, defined in lex.l
*/

void yyerror(char *s){
        fprintf(stderr, "line %d: syntax error. Last token was \"%s\"\n", line_number, yytext);
        exit(1);
}

void error(char *s){
        fprintf(stderr, "line %d: error: %s\n", line_number, s);
        exit(1);
}

 struct expr *parsing_result = NULL;
%}

//type of non terminals
%union {
  double number;
  char* id_string;
  struct expr *expr;
}

//token declaration for minic input

%token TK_PLUS TK_MINUS TK_MUL TK_DIV
%token TK_NUM TK_VAR
%token TK_LPAR TK_RPAR

/* Associativity */
%left TK_PLUS TK_MINUS
%left TK_MUL TK_DIV

%type<number> TK_NUM
%type<id_string> TK_VAR
%type<expr> e_expr t_expr f_expr

/* Axiom */
%start expr

%%

//ETF (sub-)grammar

e_expr:
e_expr TK_PLUS t_expr { $$ = expr_binop($1, $3, PLUS) ;}
| e_expr TK_MINUS t_expr { $$ = expr_binop($1, $3, MINUS) ;}
| t_expr
;

t_expr:
  t_expr TK_MUL f_expr { $$ = expr_binop($1, $3, MULT) ;}
| t_expr TK_DIV f_expr { $$ = expr_binop($1, $3, DIV) ;}
| f_expr
;

f_expr:
TK_NUM { $$ = expr_number($1); }
| TK_VAR { $$ = expr_var($1); }
| TK_LPAR e_expr TK_RPAR { $$ = $2; }
;

expr:
e_expr { parsing_result = $1; }
;
```

`ocamllex`, `ocamlyacc`, quelques mots

- éditer
  - un fichier `.ml` où est décrit le type des arbres que l'on construit in fine
  - `lexer.mll` : analyse lexicale
  - `parser.mly` : analyse syntaxique
- compiler : la moulinette fabrique `lexer.ml` `parser.ml`

peu lisibles

- corriger `shift/reduce conflict, reduce/reduce conflict`

. . .

Corriger analyses lexicale et syntaxique

voir les exemples sur la page du cours, et les copieux README

↑
important !!

pour construire votre analyseur syntaxique

- procédez par étapes !!
  - étagez votre syntaxe pour ne pas avoir à saisir 1000 règles de grammaire avant de compiler
- en cas de conflits (shift/reduce, reduce/reduce)
  - regardez le fichier `_build/parser.output`
    - les endroits où se trouvent les conflits peuvent vous renseigner
  - suivez l'exécution pas à pas de l'automate
    - plutôt en dernier recours

Démo

Affaires courantes

- le sujet du prochain rendu sera mis en ligne durant la semaine prochaine

- **vous êtes sensés le regarder dès lundi 26**
  (ne pas arriver en cours vendredi 2 sans avoir commencé à travailler sur le rendu !)

Interpréter, compiler

(3 rendus)

Rendus à venir : un sous-ensemble de Caml

- `let x = ..  in ..`
- `fun x -> ..`
- `f x y`
- `if (g y)>3 then ..  else ..`
- `let rec f x = ...  f ..`
- `let r = ref 3 in ..`
- `t := !f x`
- `try ..  with | E n -> ...`
- `raise (E 15)`

- c'est l'entrée de votre programme (lex, yacc)
- avec la syntaxe de Caml (pour pouvoir tester)
- plusieurs manières d'exécuter les programmes

Écrire un interprète

environnements, portée, clôtures

Environnements : point de départ

pour exécuter `let x = 1+2 in x*4`

- on calcule `1+2 = 3`
- on *enrichit l'environnement* avec l'association `(x,3)`
- on *exécute/évalue* `x*4` dans l'environnement ainsi obtenu
  - quand on tombe sur la variable `x`, on lit `3` dans l'environnement
- on renvoie `12`, *en retirant l'association* `(x,3)` *de l'environnement*

Liaison

pour exécuter `let x = 3 in x*y`

- ça plante dans l'environnement vide

pouf pouf.

pour exécuter `let x = 3 in x*y` dans l'environnement `(y,5)`

- on procède comme avant
- on renvoie `15`
- et on retire `(x,3)` de l'environnement (mais pas `(y,5)`)

la variable `y` est libre dans `let x = 3 in x*y`

alors que `x` est *liée*

L'interprète

l'interprète que vous devrez tous écrire est donc une fonction

```
eval :  expr -> env -> value
```

où

- `eval` est une fonction récursive
- `expr` est le type des expressions
- `env` est le type des environnements
- `value` est le type des *valeurs*

  (valeur = résultat d'un calcul)

comprenez ce que sont `env` et `value` avant de coder

Ce que l'on sait sur les environnements — portée

- une structure de données pour stocker des associations `(x,3)`

  entre une variable `x` et une valeur `3`

- discipline de pile (last in first out)

la discipline de pile dicte ce qui se passe dans

```
let x = 5 in
let n = 3 in
(let n = 4 in
  x*n) + n
```

on parle de la portée
. de la liaison `let n = 4 in..`
. de la liaison `let n = 3 in..`

retour sur `eval : expr -> env -> value` ← comprenez également comment se font les appels récursifs à `eval`

Interlude

remarque au passage :

on exécute

```
let x = 3;;
let y = x*2;;
y+x
```

comme

```
let x = 3 in
let y = x*2 in
y+x
```

Fin de l'échauffement : les fonctions

pour exécuter `let f = fun x -> x+2 in f 3`
dans l'environnement vide

- on ajoute `(f, fun x -> x+2)` à l'environnement
- on exécute `f 3`
  - **on ajoute `(x,3)` à l'environnement**
  - **on exécute `x+2`**
    ⋮
- on renvoie `5` dans l'environnement vide

Portée et fonctions

```
let toto = 3
let inzero = fun toto -> (toto 0)
let succ = fun k -> k+1
toto + (inzero succ)
```

faisons tourner le calcul **au tableau**

Portée et fonctions, clôtures

```
let h = fun t -> t+t
let g = fun y -> 30 + (h y)
let h = 12
g 5
```

pour calculer `g 5`,

- on ajoute `(y,5)` à l'environnement
- on calcule `30 + (h y)`

lorsque l'on récupère la valeur associée à `g` dans l'environnement, il faut que celle-ci "réinstalle" la valeur associée à `h` lors de la définition de `g`

on doit fabriquer une clôture, qui est un couple :

- la fonction, et
- une *copie* de l'environnement dans lequel elle a été définie

Ce que l'on sait sur les environnements — valeurs

- discipline de pile (last in first out)
- une structure de données pour stocker des associations entre variables et valeurs
- une valeur peut être
  - un entier
  - une clôture : une fonction et un environnement

    (un fragment d'environnement)

Exécuter le reste

- `if then else` pas difficile
- `let rec` moins immédiat
- références, exceptions : à voir

Exceptions

Exception = catastrophe

```
# let f x = failwith "certainement pas";;
val f :  'a -> 'b = <fun>
# f 2;;
Exception:  Failure "certainement pas".
```

en fouine,

```
| Fun(_) -> failwith "on ne peut pas sommer une fonction"
```

Exception = saut, retour en arrière

```
exception Zero
let rec parcourtliste l = match l with
    | [] -> 1
    | x::xs -> if x=0 then raise Zero else x*(parcourtliste xs)
let multlist l =
    try parcourtliste l
    with Zero -> 0
```

multlist [2;4;6;0;5;7]

```
exception E of int
let rec f m l = match l with
    | x::xs -> if x<0 then raise (E m) else
      let m' = if x>m then x else m in
      f m' xs
    | [] -> m
let k =
    try f 0 [1;3;2;8;-1;5;10]
    with E n ->
      let n' = n*n in  n'+1
```

$\longrightarrow$ 65

Exceptions et recherche

```
type btree = Leaf | Node of (int*btree*btree)
let rec search a t = match t with
    | Leaf -> false
    | Node(n,t1,t2) ->
      if a=n then true
      else search a t1 || search a t2let b1 = search a t1 in
           if b1 then true
           else search a t2

exception Found
let rec search a t = match t with
    | Leaf -> false
    | Node(n,t1,t2) ->
      if a=n then raise Found
      else let b1 = search a t1 in
           if b1 then raise Foundsearch a t2
           else search a t2
let find a t = try search a t with | Found -> true
```

Exceptions et environnement

```
exception E of int
```

```
try e1
with E x -> e2
```

- c'est `e1` qui contient des `raise (E 19)`
- `e2` parle de `x`

  on a un *lieur*, comme dans `let x =...` ou `fun x ->...`
- fouine : comprendre ce qui se passe du point de vue des environnements lorsqu'on saute

Transformations de programmes

Aspects impératifs : transformateurs d'état

Éliminer les références

- manipulation des références :

```
let e = ref 27    !e    e:=e'    ;
```

- existe-t-il un encodage (une traduction)

$$\text{(Caml pur)}+\texttt{ref} \longrightarrow \text{(Caml pur) ?}$$
$$\text{(fouine pur)}+\texttt{ref} \longrightarrow \text{(fouine pur) ?}$$
$$\texttt{e} \longrightarrow [\![\texttt{e}]\!]$$

- on exécute les programmes avec références à l'aide de l'interprète pour fouine "pur"

- l'idée:

```
[[!e]] = fun s ->
           let v = [[e]] in
           read (s,v)
```

ou plutôt

```
[[!e]] = fun s ->
           let v,s' = [[e]] s in
           (read (s',v)), s'
```

```
read:  (memory*address) -> value
```

Définition de l'encodage

- systématiser la chose: la traduction de `e`, notée ⟦e⟧, est de la forme ⟦e⟧ = `fun s -> ...(v,s')`

  (⟦e⟧ *est un transformateur d'état*)

  - tout le monde devient une fonction prenant `s` en argument
  - un programme décrit un calcul qui est susceptible de modifier l'état de la mémoire

- définitions

⟦52⟧ = ? ⟦e1 + e2⟧ = ? ⟦fun x -> e⟧ = ?

⟦e1 e2⟧ = ? ⟦let x=e1 in e2⟧ = ? ⟦x⟧ = ?

⟦!e⟧ = ? ⟦e1 := e2⟧ =? ⟦ref e⟧ = ? ⟦e1;e2⟧ = ?

Éliminer les constructions impératives

de

```
let f x =
  let a = x + !y in
( z := !z+1; a + !z )
```

à

```
let f x s =
  let vy = read s y in
  let a = x + vy in
  let vz = read s z in
  let s' = modify s (z,vz+1) in
  let vz' = read s' z in
(a + vz', s')
```

- NB : à droite, une version *optimisée* de l'encodage
- ce qu'on a fait

  modulo une implémentation de la mémoire *(dont il n'est pas difficile de se convaincre qu'elle peut être écrite de manière purement fonctionnelle)*, on a montré que les constructions pour la partie impérative de Caml redondent *(en un certain sens)*

Définition de la traduction – cœur de Caml

```
⟦52⟧ = fun s -> (52,s)
⟦fun x -> e⟧ = fun s -> ((fun x -> ⟦e⟧), s)
⟦e1 e2⟧ = fun s ->
          let (f1,s1) = ⟦e1⟧ s in
          let (v2,s2) = ⟦e2⟧ s1 in
          (f1 v2 s2)
⟦let x = e1 in e2⟧ = fun s ->
          let (x,s1) = ⟦e1⟧ s in
          ⟦e2⟧ s1
⟦x⟧ = fun s -> (x,s)
```

Définition de la traduction – constructions impératives

```
        ⟦!e⟧   =  fun s ->
                  let (l,s1) = ⟦e⟧ s in
                  let v = read l s1 in (v,s1)
⟦e1 := e2⟧   =  fun s ->
                  let (l1,s1) = ⟦e1⟧ s in
                  let (v2,s2) = ⟦e2⟧ s2 in
                  let s3 = modify s2 (l1,v2) in
                  ( (), s3)
    ⟦ref e⟧   =  fun s ->
                  let (v,s1) = ⟦e⟧ s in
                  let (l,s2) = allocate v s1 in (l,s2)
    ⟦e1;e2⟧   =  ⟦let _ = e1 in e2⟧
```

exercice : traduire et exécuter

```
let r = ref 32 in (r := !r *52; !r)
```

Programmation par continuations

Passer le futur

style de programmation *par continuations*:

les fonctions ont un argument supplémentaire,
qui est le *"futur"* du calcul

soit la fonction `let f x = x*(x+1)`

sa version par continuations est `let f x k = k (x*(x+1))`

↪ on passe le résultat à `k`

`k` : la **continuation** du calcul effectué par `f`

Continuations – du côté de l'appelant

```
let toto x y =
  (titi x (x+1))*y
```

```
let toto x y k =
  let k' = fun v -> k (v*y) in
  titi x (x+1) k'
```

ou encore

```
let toto x y k =
  titi x (x+1) (fun v -> k (v*y))
```

on passe la main à `titi`, dont le résultat sera multiplié par `y`

exemple d'appel:

```
toto 3 5 (fun i -> (print_int i;print_newline()))
```

Continuations : chercher

```
type btree = Leaf | Node of (int*btree*btree)
let rec search a t = match t with
   | Leaf -> false
   | Node(n,t1,t2) ->
     if a=n then true
     else let b1 = search a t1 in
          if b1 then true
          else search a t2


let rec search a t k = match t with
   | Leaf -> k false
   | Node(n,t1,t2) ->
     if a=n then k true
     else search a t1
       (fun res ->
         if res then k true
         else search a t2 k)
```

Contrôle dans les langages de programmation

- les exceptions sont *le* mécanisme offert en Caml pour manipuler le contrôle dans les programmes
  - dans un langage purement fonctionnel, le contrôle est la plupart du temps implicite, le programmeur n'y a pas directement accès
- le pendant dans des langages impératifs traditionnels sont les instructions telles que

  `return` `while` `break` `continue`

  en C

```
while (p--){
  ..   if (x==0) break;   ..
}
```

  - but: dévier le flot du calcul, qui s'exprime naturellement dans les langages impératifs comme une séquence de commandes
- idée des **continuations**: *expliciter le contrôle* en manipulant le **futur** du calcul
  - cela permet en particulier de "programmer" les exceptions

Exceptions et continuations

Transformations à l'aide de continuations

- de (Caml+exceptions) vers Caml

```
exception E of int
raise (E 32)
try..  with
```

  deux futurs : futur normal, futur "exceptionnel"

$[\![$`32`$]\!]$ = `fun k kE -> ?`

$[\![$`e1+e2`$]\!]$ = `?` $[\![$`fun x -> e`$]\!]$ = `?` $[\![$`x`$]\!]$ = `?`

$[\![$`e1 e2`$]\!]$ = `?` $[\![$`let x = e1 in e2`$]\!]$ = `?`

$[\![$`raise (E e)`$]\!]$ = `?` $[\![$`try e1 with (E x) -> e2`$]\!]$ = `?`

- "exercice de style" : du style *direct* au style par continuations

  $[\![$`e`$]\!]$ = `fun k -> ...`

  - de Caml (sans exceptions) vers Caml
  - NB : $[\![$`fun x y -> x*y`$]\!] \neq$ `fun x y k -> k (x*y)`

    tout du moins si l'on applique la transformation de façon systématique

  - technique de compilation

Compilation vers une machine à pile

Compilation et pile : le point de départ

la compilation :

```
type ex = Cst of int | Add of ex*ex

type instr = C of int | A
type code = instr list

let rec compile :  ex -> code = function
  | Cst k -> [C k]
  | Add (e1,e2) -> (compile e2)@(compile e1)@[A]
```

```
# let c = compile (Add(Cst 3, Add(Cst 5, Cst 7)));;
val c :  code = [C 7; C 5; A; C 3; A]
```

la machine :

. on peut ainsi *exécuter le programme résultant de la traduction*

. *le résultat se lit en haut de la pile à la fin* *(convention)*

Compilation — environnement

`x` est compilé en `ACCESS(x)` (noté $[\![\texttt{x}]\!] = \texttt{ACCESS(x)}$)

`let x = e1 in e2` est compilé en $[\![\texttt{e1}]\!]\texttt{;LET(x);}[\![\texttt{e2}]\!]\texttt{;ENDLET}$

(NB : ";" n'est pas extrêmement bien typé ici)

machine :

un état de la machine est donné par
le code à exécuter (c), l'environnement (e), la pile (s)

ACCESS(x);c	e	s	c	e	e(x)·s
LET(x);c	e	v·s	c	(x,v)·e	s
ENDLET;c	(x,v)·e	s	c	e	s

l'environnement dans la machine rappelle l'environnement pour l'interprète

exemple au tableau

Compilation — fonctions et applications

`fun x -> e` est compilé en `CLOS(x,(⟦e⟧;RET))`

`e1 e2` est compilé en `⟦e2⟧;⟦e1⟧;APPLY`

machine :

ce qu'est une clôture :

(x,c)[e] représente la fonction
- dont l'argument est x
- le corps est c
- l'environnement est e

CLOS(x,c');c	e	s	c	e	(x,c')[e]·s
APPLY;c	e	(x,c')[e']·v·s	c'	(x,v)·e'	c·e·s
RET	e	v·c'·e'·s	c'	e'	v·s

NB : si on tombait sur "RET;c", on jetterait le c à la poubelle

exemple au tableau

Compilation — remarques

- fabriquer une clôture a pour effet de dupliquer l'environnement
  - quand on est naïf
- la machine peut se casser la figure
- la machine s'appelle d'ailleurs SECD dans la littérature

f i n