Projet 2

présentation du cours

daniel.hirschkoff@ens-lyon.fr

Projet 2

- intervenants

  Aurore Alcolei   Daniel Hirschkoff   Bertrand Simon
- 2h par semaine sur machine   (+2h parfois en amphi)
- programmation en **Caml**
  - seul(e), puis en binôme
  - exigences adaptées

    quantité de travail $\simeq$ équivalente suivant le niveau

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

- demain (24/01) : TP
  - de quoi nourrir tout le monde
  - issu de la fiche 11 de Caml en Projet 1

    P. Oechsel, H. Valentin, N. Champseix
  - salle E001, n'hésitez pas à venir avec votre ordinateur
- ensuite : 1 DM seul(e), 3 ou 4 rendus en binôme

Contenu

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

- programmer en Caml
- programmer modulairement

  génie logiciel

- organiser le travail
- “faire vivre” du code

- rendus
  - librairies, structures de données
  - programmes qui manipulent des programmes

    interprètes, un peu de compilation

Éléments de Caml

Survol de Caml

- fonctions
- types
- programmation impérative
- modules
- représentation mémoire, égalité physique

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, $(t_1 * t_2)$`->`t_3 c'est t_1`->`t_2`->`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
```

$$\begin{aligned} \mathsf{A} &\rightarrow \mathsf{C}_{(k)} \\ \mathsf{A} &\rightarrow \mathsf{A{+}A} \\ \mathsf{A} &\rightarrow \mathsf{A{*}A} \\ \mathsf{A} &\rightarrow \textsf{if A then A else A} \end{aligned}$$

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...`"

  `| _ -> ...` *Agnostos Theos*

- le "vrai truc", c'est
  `| [] -> ...`
  `| x::xs -> ...`

En supplément des douze dieux principaux et d'innombrables divinités mineures, les grecs anciens adoraient une divinité nommée Agnostos Theos, ce qui signifie le dieu inconnu.

À Athènes, il y avait un temple dédié spécialement à ce dieu et de nombreux athéniens juraient "par le nom du dieu inconnu". Apollodore, Philostrate et Pausanias écrivirent à propos de ce Agnostos Theos.

Ce dieu inconnu n'était pas une divinité spécifique, mais une marque substitutive, pour n'importe quel dieu ou divinité qui existait à l'époque mais dont le nom et la nature n'avaient pas été révélés aux athéniens et au monde grec en général.

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

- 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

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

Questionnaire

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
  ```

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
```

Modules et signatures

Paramétrer

- paramétrer par un type, *généricité* DÉMO `param.ml`

  ```
  # let bis = fun f -> fun x -> f (f x);;
  val bis :  ('a -> 'a) -> 'a -> 'a = <fun>
  ```

  - `'a` : *n'importe quel type, appelons-le 'a*
    polymorphisme
  - ça se fait implicitement (le code de bis ne parle pas de 'a)
  - signification comportementale/bas niveau : paramétricité, on ne déréférence pas le pointeur
- au-delà du polymorphisme :
  - `'a` peut désigner une structure de donnée quelconque
    (entier, triplet, liste, ...)
  - il est souvent utile de regrouper une structure de données et des opérations qui s'en servent
    - *génie logiciel*
      définir des "entités" pour la programmation
      modules, objets, composants, librairies, ...
    - cf. aussi structure algébrique (groupe, monoïde, ordre partiel, ..)
- paramétrer par un type muni d'opérations : modules

Modules, types de modules

- exemple : du code pour les matrices

```
type mat = (float list) list                    type ope = mat->mat->mat

                  let affiche = ..                                   let mult m1 m2 = ..
```

- typer le code ci-dessus :

```
type mat = (float list) list   val affiche: mat->unit   type ope = mat->mat->mat   val mult: ope
```

- en Caml on peut définir
  - des modules   Démo `ex-modules.ml`

  `module M =`

  `struct` *(des définitions de types, des valeurs)* `end`
  - des types de modules (signatures)

  `module type T =`

  `sig` *(des définitions de types, des types de valeurs)* `end`

- les modules servent à structurer le code en Caml

  développement séparé, réutilisation du code

Types abstraits

rendre le code indépendant de l'implémentation sous-jacente

- *cacher* un type

```
module type T = sig
  type mat = (float list) listtype mat
  val init: float -> mat
  val affiche: mat -> unit
  type ope = mat->mat->mat
  val mult: ope
end
```

```
module M : T = struct
  type mat = (float list) list
  let init x = ..
  let affiche = fun m -> ..
  type ope = mat->mat->mat
  let mult m1 m2 = ..
end
```

- l'abstraction favorise . impose la modularité

Modules paramétrés (foncteurs)

"fonctions sur les modules" :

```
module type T = sig
  type mat
  val accède:mat->(int*int)->float
end
```

définition d'un module paramétré:

```
module AffMat (M:T) = struct
  let affiche = ...  M.accède (x,y) ..
end
```

`M.accède` désigne `accède` tel que défini dans `M`

- plusieurs implémentations de `T` possibles
- qui a priori ne définissent pas que `mat` et `accède`
  `M:T` est validé si "`M` fournit au moins `T`"

Modules paramétrés — exemple

allez lire `set.mli`   `/usr/lib/ocaml/set.mli`

un foncteur pour faire les ensembles à partir d'un type ordonné

```
module type OrderedType =
  sig
    type t
    (** The type of the set elements.*)
    val compare :  t -> t -> int
  end
```

```
module type S =
  sig
    type elt
    (** The type of the set elements.  *)

    type t
    (** The type of sets.  *)

    val empty:  t
    (** The empty set.  *)

    val add:  elt -> t -> t
    (** [add x s] returns a set containing all elements of [s],
      plus [x].  If [x] was already in [s], [s] is returned unchanged.  *)
   ⋮
module Make (Ord : OrderedType) :  S with type elt = Ord.t
```

Compilation séparée en Caml

- on peut répartir un développement Caml sur plusieurs fichiers

  `toto.ml` `titi.ml` `tutu.ml`
- convention un peu scabreuse : un fichier `.ml` $\leftrightarrow$ un module
  - cela s'apparente à de la convivialité
  - `Toto` `Titi` `Tutu` (`Toto.f`, `Titi.g`, `List.fold_left`)
- types de modules: fichiers `.mli`
  1. on compile `toto.mli` $\rightsquigarrow$ `toto.cmi`
  2. on compile `toto.ml` $\rightsquigarrow$ `toto.cmo` si les types sont OK
- c'est comme s'il y avait des `struct.. end` implicites dans un `.ml` (et des `sig.. end` dans un `.mli`)
- automatisation de tout ça, dépendances: `Makefile`

  sera abordé par la pratique DÉMO `DM-Squelette`

- NB: les modules, c'est ce que propose Caml

  autres langages, autres approches

Les valeurs en mémoire

Égalité structurelle, égalité physique

Valeurs

- ce que Caml stocke en mémoire, ce sont des valeurs

  le résultat d'un calcul
- $2+5$ n'est pas une valeur, 3 est une valeur

  `"toto"` aussi
- `f 12` n'est pas une valeur

  `fun x -> 3*x` est une valeur

  `fun x -> BLA` **est une valeur** $\forall$ `BLA`
- `[1;1;2;3;5;8]` est une valeur

Comparer les valeurs : égalité structurelle

- `fun t -> if` $\boxed{\texttt{t = "mickey"}}$ `then 5 else 12`
- quelques (dis)égalites

  `5=5   6+5=11`

  `[1;2;3] = [1;2;3]   [|1;2|] = [|1;2|]`

  `fun x->x+0` $\neq$ `fun x->x-0   fun x->x` $\neq$ `fun t->t`

  `let cmp = fun e ->  if e=e then 1 else 0`

- égalité entre valeurs, et pas toutes
- l'égalité = est dite structurelle, elle parle des valeurs en tant qu'entité abstraite/mathématique
  - en estimant à la limite qu'il y a une infinité d'entiers, listes, etc.
  - …entre valeurs *comparables en Caml*

Stocker les valeurs

- autre notion d'égalité : l'égalité physique, notée `==`
  cela a trait à comment sont stockées les valeurs en mémoire
  Démo `eq-phy.ml`
- quand on peut, on stocke la valeur directement `53` `'f'` `true`
- sinon, une valeur est représentée par une adresse en mémoire
  - liste, arbre, type somme
    pointeur vers la tête de la liste, la racine de l'arbre, etc.
  - fonction

  on n'a pas besoin de savoir comment les choses sont représentées en mémoire
- pour `==`, on compare *directement* deux mots en mémoire
  - deux valeurs pour les valeurs dites "immédiates"
  - deux pointeurs sinon
  - `==` est donc plus efficace que `=` en général
  - quid de `154 == [1;4;2]` ? pas de risque : typage
    (i.e., si la liste `[1;4;2]` est stockée à l'adresse `154`)

Stocker les fonctions en mémoire

- comment est représentée une fonction en mémoire ?

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

```
let a = g 5                    let b = g h
```

- pour stocker `g`, il faut lui associer un moyen d'accéder à `h`
  (la fonction `h`, telle qu'elle existe au moment où `g` est définie)
- même chose pour

```
let k = ref 12
let f = fun x -> x * !k
```

- clôture : le code
  \+ un *environnement*
  permettant de donner du sens au code

Pensez à vous binômer !! *(avant le 3 février)*

À propos du premier rendu

1. *BDDs*
2. *Analyses lexicale et syntaxique*
3. *Transformation de Tseitin*

Rendu 1

fichier en entrée

$\downarrow$

formule logique $\longrightarrow$ BDD

$\downarrow$

formule SAT

$\downarrow$

`minisat`

Binary Decision Diagrams (BDD)

Fonctions booléennes

- on s'intéresse à des fonctions $f : \{0,1\}^n \to \{0,1\}$
- applications typiques :

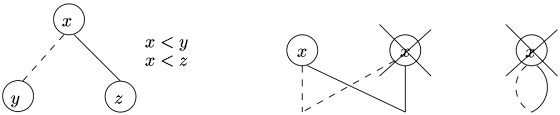
  raisonnement sur des circuits, vérification
- représentation naïve :
  - un arbre binaire complet avec $2^n - 1$ nœuds,
    les feuilles étant 0 ou 1
  - pour calculer $f(x_1, \ldots, x_n)$,
    on descend dans l'arbre suivant la valeur des x_i

    (les x_i étiquettent les nœuds, les arcs sont étiquetés par 0 et 1)
- BDDs : *replier* un tel arbre, en faisant du partage et en éliminant les nœuds non informatifs

Binary Decision Diagrams

on s'intéresse en réalité aux ROBDD
(reduced ordered binary decision diagram)

- trois types de noeuds: 0, 1 feuilles
  var deux fils, une variable comme étiquette
- trois conditions pour un ROBDD :



© H.R. Andersen

- variables Ordonnées
  de la racine à une feuille, on croît dans l'ordre
- pas de redondance (Réduit)

un DAG (*directed acyclic graph*) plutôt qu'un arbre

ROBDDs : unicité et construction

- **Théorème:** étant donné une formule ϕ et un ordre sur les variables de ϕ, il existe un unique ROBDD représentant ϕ.

  Preuve: par induction sur le nombre de variables, on écrit la formule $\phi(x_1, \ldots, x_k)$ comme
  `if` x_1 `then` $\phi(1, x_2, \ldots, x_k)$ `else` $\phi(0, x_2, \ldots, x_k)$

- construction, opération élémentaire : calculer $\alpha \diamond \alpha'$, où α et α' sont deux BDDs, et $\diamond$ est une opération logique ($\wedge, \vee, \ldots$).

$$\alpha \diamond \alpha' = \begin{cases} (v, \ell \diamond \ell', h \diamond h') & \text{si } v = v' \\ (v, \ell \diamond \alpha', h \diamond \alpha') & \text{si } v < v' \\ (v', \alpha \diamond \ell', \alpha \diamond h') & \text{si } v > v' \end{cases}$$

α est représenté par (v, ℓ, h), idem pour α'

cf. *melding*, D. Knuth

NB : dépend de l'ordre $<$ sur les variables

exemples au tableau

ROBDDs : implémenter le partage

on construit un BDD qui satisfait “RO” *directement*

- implémentation du partage :
  chaque sous-arbre (y compris les feuilles) a un *identificateur*
  - une table associe à chaque identificateur i de sous-arbre ce qu'il représente — triplet (v, h, b), sauf pour les feuilles
  - une table associe à chaque triplet (v, h, b) l'identificateur i d'un nœud existant si le BDD correspondant a déjà été construit
  - partage maximal : $\boxed{= \quad \Leftrightarrow \quad ==}$
  - en C, on peut utiliser l'adresse physique pour l'identificateur
- calcul des opérations : encore une table
  programmation dynamique / mémorisation

  ne pas recalculer $\alpha \diamond \alpha'$ si cela a déjà été fait

- par ailleurs, la place occupée en mémoire peut dépendre fortement de l'ordre sur les variables
  - intuitivement, certaines variables jouent un rôle crucial, d'autres pas
  - ce phénomène se retrouve dans d'autres approches pour SAT

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 $\leadsto$ *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 $\boxed{1} \rightarrow \boxed{2} \rightarrow \boxed{3}$
  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, . . .

Les deux étapes dans le front end

- analyse lexicale

  flot de caractères (source) $\rightarrow$ 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)`

  $\rightarrow$ `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 $\rightarrow$ *arbre de syntaxe abstraite*

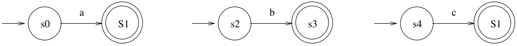
  $\rightarrow$
  ```
  Add( Mult(Int(32), Int(52)),
       Let("x", Int(5), Mult(Var("x"), Var("x"))) )
  ```

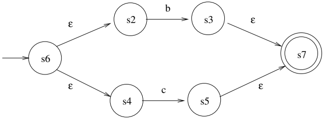
  - étape intermédiaire: arbre d'analyse syntaxique *(parse tree)*

Analyse lexicale

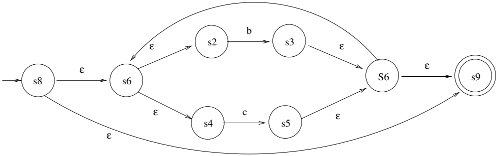
- chaque lexème est décrit par une *expression régulière*
- principaux éléments (syntaxe de `ocamllex`):
  - caractère `'$'`, chaîne de caractères `"else"`
  - intervalle `['0'-'9']` *(un chiffre)*
  - disjonction (de caractères)
    `['\t' ' ']` *(tabulation ou espace)*
  - juxtaposition `['A'-'Z']['a'-'z' '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 *mots*

Expression régulière ↔ automate non déterministe

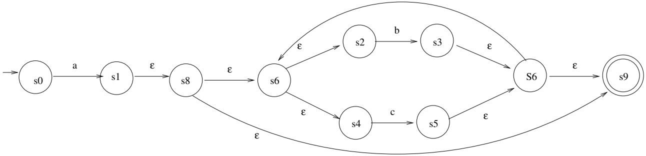




NFA pour b|c



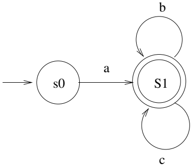
NFA pour (b|c)*



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 : au total

- on décrit le **dictionnaire**

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

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

- 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
- "magiquement", on obtient un programme qui reconnaît les mots du dictionnaire (et proteste sinon)

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:

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

- 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 : $E \rightarrow a$
$\$E$	$+b * c\$$	shift
$\$E+$	$b * c\$$	shift
$\$E + b$	$*c\$$	reduce : $E \rightarrow b$
$\$E + E$	$*c\$$	shift **(remarquablement malin)**
$\$E + E*$	$c\$$	shift
$\$E + E * c$	$\$$	reduce : $E \rightarrow c$
$\$E + E * E$	$\$$	reduce : $E \rightarrow E * E$
$\$E + E$	$\$$	reduce : $E \rightarrow E + E$
$\$E$	$\$$	accept

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

Lex & Yacc, Flex & Bison, . . .

- analyse syntaxique : on écrit la grammaire, et on associe à chaque règle une *action sémantique* *(construction de l'arbre)*
- DÉMO avec ocamllex ocamlyacc
- *remarque :* avec yacc, on ôte les ambiguïtés en "bricolant", pas en réécrivant la grammaire (comme en FDI)
- on trouve *"partout"* les outils pour les analyses lexicale et syntaxique

```
%{
(* —— 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` PARENTAL ADVISORY *peu lisibles*
- corriger `shift/reduce conflict, ...`
  - vous pouvez regarder le fichier `_build/parser.output`
  - assez empirique, de manière inhérente

Transformation de Tseitin

Formules quelconques, lois de de Morgan

- on prend en entrée une formule logique avec les connecteurs usuels

$$\big(\neg p \wedge (q \Rightarrow r)\big) \;\Rightarrow (q \vee \neg p)$$

- on veut la transformer en une formule en *forme normale conjonctive* (CNF) $(\overline{x_2} \; : \neg x_2)$

$$(x_1 \vee \overline{x_3} \vee \overline{x_4}) \;\wedge\; (x_1 \vee x_4 \vee x_5) \;\wedge\; (\overline{x_2} \vee \overline{x_3})$$

Lois de de Morgan :

$$[(p \wedge q) \vee r] = (p \vee r) \wedge (q \vee r) \qquad [\neg(p \wedge q)] = \neg p \vee \neg q$$

$$[\neg(p \vee q)] = \neg p \wedge \neg q \qquad [p \Rightarrow q] = \neg p \vee q \qquad [\neg\neg p] = p$$

- on obtient une formule en CNF en itérant ces lois...
  ...mais la distributivité fait exploser la taille
- inévitable si on veut préserver le sens de la formule
- on se contente de vouloir préserver la **satisfiabilitté**
  et on veut aussi pouvoir engendrer, le cas échéant, un contre-exemple de la formule de départ

Transformation de Tseitin, définition

- pour chaque sous-formule p de la formule de départ, on introduit une nouvelle variable ξ_p
- on y va inductivement, pour associer à chaque sous-formule p une formule $[p]$, *directement en forme normale conjonctive*:

$$[p = p_1 \vee p_2] = (\neg\xi_p \vee \xi_{p_1} \vee \xi_{p_2}) \wedge (\xi_p \vee \neg\xi_{p_1}) \wedge (\xi_p \vee \neg\xi_{p_2})$$

$$[p = p_1 \wedge p_2] = (\neg\xi_p \vee \xi_{p_1}) \wedge (\neg\xi_p \vee \xi_{p_2}) \wedge (\xi_p \vee \neg\xi_{p_1} \vee \neg\xi_{p_2})$$

$$[p = \neg p_1] = (\neg\xi_p \vee \neg\xi_{p_1}) \wedge (\xi_p \vee \xi_{p_1})$$

$$[p = x] = (\neg\xi_p \vee x) \wedge (\xi_p \vee \neg x)$$

ainsi, par exemple, lorsque $p = p_1 \vee p_2$, $[p]$ exprime que p est satisfaite si et seulement si $p_1 \vee p_2$ l'est

- une formule p est transformée en la conjonction

$$\xi_p \wedge \bigwedge_{p' \text{ sous-formule de } p} [p']$$

le ξ_p impose que la formule initiale (à la racine) soit satisfaite

Tseitin, propriétés

on se donne une formule p, on en calcule la transformée de Tseitin, notée (par abus) $[p]$

- $[p]$ se calcule en temps **linéaire** par rapport à la taille de p
- p et $[p]$ sont **équi-satisfiables**
  - toute valuation qui satisfait $[p]$ satisfait p (et satisfait toujours p en modifiant la valeur de variables ne se trouvant pas dans p)
  - une valuation qui satisfait p peut être étendue en une valuation qui satisfait $[p]$
- (on peut optimiser la représentation en mémoire en partageant des sous-expressions)

SAT pour `minisat`

il vous est demandé dans le rendu de

- transformer une formule quelconque en une formule SAT via Tseitin
- écrire la formule dans un fichier, au format DIMACS

```
p cnf nvar nclau
-4 2 5 0
2 -1 0
```

`nvar` : nombre de variables (ici 5)
`nclau` : nombre de clauses (ici 2)
$(\overline{x_4} \vee x_2 \vee x_5) \wedge (x_2 \vee \overline{x_1})$

- faire manger le résultat à `minisat`

Affaires en cours

- DM non encore rendus
- binômes à constituer
- commencez aujourd'hui à répartir le travail pour le premier rendu

  la semaine prochaine : *il doit y avoir quelque chose de fait*
- commencez à travailler avec `git`

Second demi-projet

interpréter, compiler

(3 rendus)

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

Interpréter

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*

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 = 4 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..`

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, avec une copie de l'environnement dans lequel celle-ci a été créée

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

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

Compilation — environnement

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

`let x = e1 in e2` est compilé en
$[\![\mathtt{e1}]\!]$`;LET(x);`$[\![\mathtt{e2}]\!]$`;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 $\texttt{CLOS(x,(}[\![\texttt{e}]\!]\texttt{;RET))}$

`e1 e2` est compilé en $[\![\texttt{e2}]\!]\texttt{;}[\![\texttt{e1}]\!]\texttt{;APPLY}$

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

f i n