# Compilation — Examen — Automne 2023

Durée 2h. Documents de cours et notes personnelles autorisés.

On s'intéresse à un langage disposant de listes chaînées, manipulées avec les mêmes primitives que nous avons en caml : la liste contenant les éléments `1`, `2` et `3` est par exemple être notée `[1; 2; 3]` ou `[1; 2; 3;]`, et peut également être construite en ajoutant successivement chacun de ses éléments à la liste vide : `1 :: 2 :: 3 :: []`. En outre, deux fonctions `hd` et `tl` permettent respectivement d'accéder au premier élément et à la queue d'une liste. Ces fonctions vérifient en particulier les équations `hd(x :: l) = x` et `tl(x :: l) = l`, tandis que `hd([])` et `tl([])` ne sont pas définies.

Cet examen comporte trois exercices essentiellement indépendants, portant sur trois aspects de ces listes : typage, analyse syntaxique, et traduction en assembleur.

**Exercice 1** (Types et sémantique) On se donne les types $\tau$ suivants :

| | | | |
|---|---|---|---|
| $\tau$ | ::= | `int` | nombre entiers |
| | \| | $\tau$ `list` | listes homogènes dont les éléments ont le type $\tau$ |

et des règles de typage :

$$\frac{}{\Gamma \vdash n \texttt{ : int}} \qquad \frac{\Gamma \vdash e_1 \texttt{ : int} \qquad \Gamma \vdash e_2 \texttt{ : int}}{\Gamma \vdash e_1 \texttt{ + } e_2 \texttt{ : int}}$$

$$\frac{}{\Gamma \vdash \texttt{[]} \texttt{ : } \tau \texttt{ list}} \qquad \frac{\Gamma \vdash e \texttt{ : } \tau \qquad \Gamma \vdash \ell \texttt{ : } \tau \texttt{ list}}{\Gamma \vdash e \texttt{ :: } \ell \texttt{ : } \tau \texttt{ list}} \qquad \frac{\Gamma \vdash e_1 \texttt{ : } \tau \qquad \ldots \qquad \Gamma \vdash e_n \texttt{ : } \tau}{\Gamma \vdash \texttt{[}e_1\texttt{; ...; }e_n\texttt{]} \texttt{ : } \tau \texttt{ list}}$$

1. Les expressions suivantes sont-elles typables ? Si oui, donner une dérivation, et sinon expliquer le problème.
   (a) `(1 + 2) :: (3 :: [])`
   (b) `(1 :: []) :: (2 :: [])`
   (c) `(1 :: []) :: ([] :: [])`
2. Donner deux règles de typage pour les fonctions `hd` et `tl`.
3. Montrer que si le jugement $\Gamma \vdash$ `[`$e_1$`; ...; `$e_n$`]` : $\tau$ `list` est valide, alors $\Gamma \vdash e_1$ `::` ... `::` $e_n$ `:: []` : $\tau$ `list` l'est également. *Indication : par récurrence sur n.*
4. Y a-t-il des termes bien typés dont l'évaluation pose néanmoins problème ?

Pour affiner notre analyse de types, on introduit deux types particuliers $\tau$ `nelist` pour les listes garanties non vides, et `elist` pour des listes garanties vides. On se donne avec un nouveau jugement de typage $\Gamma \vdash_s e : \tau$ associé aux règles d'inférence suivantes.

$$\frac{}{\Gamma \vdash_s n \texttt{ : int}} \qquad \frac{\Gamma \vdash_s e_1 \texttt{ : int} \qquad \Gamma \vdash_s e_2 \texttt{ : int}}{\Gamma \vdash_s e_1 \texttt{ + } e_2 \texttt{ : int}}$$

$$\frac{}{\Gamma \vdash_s \texttt{[]} \texttt{ : elist}} \qquad \frac{\Gamma \vdash_s e \texttt{ : } \tau \qquad \Gamma \vdash_s \ell \texttt{ : } \tau \texttt{ list}}{\Gamma \vdash_s e \texttt{ :: } \ell \texttt{ : } \tau \texttt{ nelist}} \qquad \frac{\Gamma \vdash_s e \texttt{ : elist}}{\Gamma \vdash_s e \texttt{ : } \tau \texttt{ list}} \qquad \frac{\Gamma \vdash_s e \texttt{ : } \tau \texttt{ nelist}}{\Gamma \vdash_s e \texttt{ : } \tau \texttt{ list}}$$

5. Donner des dérivations pour les jugements
   (a) $\Gamma \vdash_s$ `1 :: (2 :: []) : int nelist`
   (b) $\Gamma \vdash_s$ `(1 :: []) :: ([] :: []) : (int list) nelist`
6. Montrer que si le jugement $\Gamma \vdash_s e : \tau$ est valide, alors $\Gamma \vdash e : \tau'$ est valide également pour un certain $\tau'$, et décrire ce type $\tau'$. *Indication : par induction structurelle sur* $\Gamma \vdash_s e : \tau$.
7. Proposer des règles de typage pour `hd` et `tl` dans ce nouveau système, de sorte à accepter moins d'expressions mais éviter les problèmes évoqués à la question 4. Donner une expression $e$ et un type $\tau$ tels que $\Gamma \vdash e : \tau$ est valide mais pas $\Gamma \vdash_s e : \tau$.

□

**Exercice 2** (Analyse syntaxique) On se donne la grammaire `menhir` suivante pour des expressions avec des listes. Le symbole non terminal `main` désigne une phrase complète, `expr` une expression quelconque, et `elts` une séquence d'expressions séparées par ; (point-virgule).

```
%token INT PLUS LPAR RPAR EOF
%token LSQB SEMI RSQB CONS NIL
%start <unit> main
%%

main:
| expr EOF {}
```

```
expr:
| INT {}
| NIL {}
| LPAR expr RPAR {}
| expr PLUS expr {}
| expr CONS expr {}
| LSQB elts RSQB {}
```

```
elts:
| (* empty *)    {}
| expr            {}
| expr SEMI       {}
| expr SEMI elts {}
```

1. Donner une dérivation pour l'expression `[ 1; 2; 3 ]`.
   *Indication : cette expression correspond à la séquence de l'exèmes* `LSQB INT SEMI INT SEMI INT RSQB`.
2. Donner les différents arbres de dérivation possibles pour l'expression `1 + 2 :: 3 :: []`.
   *Indication : cette expression correspond à la séquence de lexèmes* `INT PLUS INT CONS INT CONS NIL`.
3. Sachant que l'expression précédente est bien typée, en déduire la bonne dérivation parmi les possibilités précédentes, et les priorités et associativités à donner aux symboles `INT`, `PLUS`, `CONS` et `NIL`.
   *Indication : tous ne nécessitent pas une annotation.*
4. L'annexe montre un extrait du fichier de diagnostic généré par `menhir` à propos d'un autre conflit.
   (a) Décrire ce conflit. Donner un exemple d'entrée sur laquelle il se manifeste, et les différentes dérivations possibles sur cet exemple.
   (b) Comment pourrait-on ajuster la grammaire pour faire disparaître ce conflit ?

□

**Exercice 3** (Génération de code) On va représenter une liste en mémoire par un ensemble de blocs alloués sur le tas. Chaque bloc a trois champs :

— Le premier champ est un entête contient un booléen indiquant si le bloc est utilisé ou non.
— Le deuxième champ contient un élément de la liste.
— Le troisième champ est un pointeur vers le bloc suivant.

Une liste est représentée par un pointeur vers un tel bloc, et la liste vide par le pointeur nul.

1. Dessiner les structures représentant la liste `[1; 2; 3]`, en précisant la cible de chaque pointeur et les décalages de chaque champ d'un bloc par rapport à son adresse de base.
2. Écrire un code assembleur qui suppose que le registre `$a0` contient une liste non vide, et qui place dans `$v0` le premier élément de cette liste.
3. Écrire un code assembleur qui suppose que les registres `$a0` et `$a1` contiennent des listes d'entiers, et qui teste si ces deux listes contiennent les mêmes éléments.

On va gérer l'allocation de nos nouveaux blocs à la main, de la manière suivante : on stocke en mémoire une liste chaînée de blocs non utilisés (des blocs à trois champs dont le premier vaut faux et le troisième est un pointeur vers un autre tel bloc non utilisé, le deuxième pouvant être n'importe quoi). On se donne une variable globale `free_blocks`, qui pointe vers le premier de ces blocs.

4. Écrire un code assembleur qui suppose que le registre `$a0` contient un élément $e$ et le registre `$a1` un pointeur vers une liste $\ell$, que la liste des blocs libres n'est pas vide, et qui crée un nouveau bloc $e :: \ell$ en utilisant un bloc prélevé dans la liste des blocs libres (au passage, il faut préciser comment vous matérialisez la variable `free_blocks`).
5. Écrire un code assembleur qui suppose que le registre `$a0` contient une liste, et qui détruit le bloc associé (c'est-à-dire, qui le remet dans la liste des blocs libres).
6. Écrire un code assembleur qui suppose que le registre `$a0` contient une liste, et qui détruit toute la liste.
7. Écrire un code assembleur qui ajoute 256 nouveaux blocs à la liste des blocs libres (il faudra pour cela étendre le tas d'un volume suffisant).

□

## Annexe. Aide-mémoire MIPS

Voici quelques instructions MIPS susceptibles de vous être utiles (vous avez le droit d'en utiliser d'autres) :

| | |
|---|---|
| `li` $r$, $n$ | charge l'entier $n$ dans le registre $r$ |
| `move` $r_1$, $r_2$ | copie le registre $r_2$ dans le registre $r_1$ |
| `add` $r_1$, $r_2$, $r_3$ | calcule la somme de $r_2$ et $r_3$ et la place dans $r_1$ |
| `lw` $r_1$, $n(r_2)$ | charge dans $r_1$ la valeur contenue à l'adresse $r_2 + n$ |
| `sw` $r_1$, $n(r_2)$ | écrit en mémoire à l'adresse $r_2 + n$ la valeur contenue dans $r_1$ |
| `b` $L$ | saute à l'adresse désignée par l'étiquette $L$ |
| `beqz` $r$, $L$ | saute à l'adresse désignée par l'étiquette $L$ si le registre $r$ contient 0 |
| `syscall` | effectue un appel système, dont la nature est donnée par le registre `$v0` ; par exemple, si `$v0` contient 9, alors l'appel déclenché est `sbrk`, qui alloue sur le tas un nombre d'octets donné par `$a0`, et qui place dans `$v0` l'adresse de début du bloc ainsi alloué |

## Annexe. Extrait de fichier `.conflicts`

```
** Conflict (reduce/reduce) in state 12.
** Token involved: RSQB
** This state is reached from main after reading:

LSQB expr SEMI

** The derivations that appear below have the following common factor:
** The question mark symbol (?) represents the spot where the derivations begin
** to differ.

main
expr EOF
LSQB elts RSQB // lookahead token appears
     (?)

** In state 12, looking ahead at RSQB, reducing production
** elts ->
** is permitted because of the following sub-derivation:

expr SEMI elts // lookahead token is inherited
          .

** In state 12, looking ahead at RSQB, reducing production
** elts -> expr SEMI
** is permitted because of the following sub-derivation:

expr SEMI .
```