Partiel - 20 octobre 2025

Exercice 1 *Langages réguliers (4 points)*

On cherche à reconnaître des expressions qui représentent des types d'opérateurs simples comme `bool` (objet booléen), `int → int` (fonction unaire sur les entiers), `int*int → bool` (fonction binaire qui prend en argument 2 entiers et renvoie un booléen).

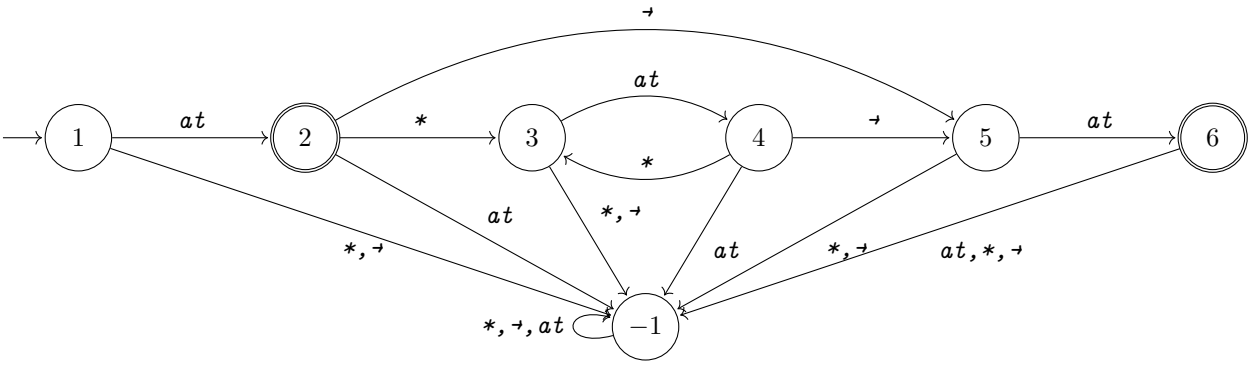
Pour cela on suppose que l'on a un ensemble de trois symboles terminaux $\{\texttt{at}, \rightarrow, *\}$ dans lequel `at` représente n'importe quelle expression de type atomique comme `int`, `bool`.

Un type est soit un type atomique (terminal `at`), soit de la forme α `→ at` avec α une suite non vide de types atomiques séparés par le caractère $*$. On appelle T le langage des types ainsi décrit.

1. Donner une expression régulière dont le langage associé est exactement T.
2. Donner un automate déterministe et complet qui reconnaît exactement les mots de T.
3. Donner le chemin dans votre automate pour la reconnaissance de la suite de terminaux `at * at → at`.
4. Donner une grammaire non-ambigüe qui reconnaît exactement les mots de T.
5. Donner l'arbre de dérivation syntaxique dans votre grammaire pour la suite de terminaux `at * at → at`.

Correction : *D'après l'énoncé, les mots* at, $at{\rightarrow}at$ *et* $at{*}at{\rightarrow}at$ *sont dans le langage, par contre* $at{*}at$ $at{\rightarrow}at{\rightarrow}at$ *ne le sont pas. Ces exemples devront être validés sur chaque description du langage (expression régulière, automate, grammaire).*

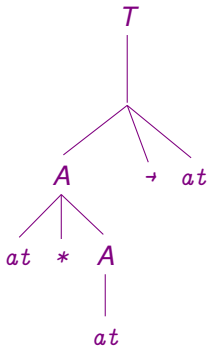
1. *Une expression régulière qui reconnaît le langage est :* $(at\ (*at)^{*}{\rightarrow})?at$
2. *Voici un automate déterministe et complet reconnaissant le langage. Il y a pour chaque état et chaque symbole terminal une et une seule transition. L'état puits* -1 *(dont on ne peut sortir) sert d'état destination pour tous les cas d'erreur.*



3. *Le chemin est* $1at2{*}3at4{\rightarrow}5at6$
4. *On introduit deux non-terminaux,* A *pour reconnaître des suites de types atomiques séparés par le caractère* $*$ *et* T *pour les types eux-mêmes. On prend soin de bien préciser le symbole initial qui est* T*. Les règles sont :*

$$T ::= at \mid A{\rightarrow}at \qquad A ::= at \mid at\ *\ A$$

5. *L'arbre de dérivation pour cette grammaire et l'entrée* $at * at \to at$ *est*



Exercice 2 *Expressions régulières (2 points)*

1. Dans le langage Python, une constante de chaîne de caractères simple commence par le symbole `"` et se termine à la première apparition d'un symbole `"`. A l'intérieur de la chaîne de caractères on peut utiliser le caractère d'échappement `\`. Le caractère d'échappement est toujours suivi d'un caractère (par exemple un guillemet ou le caractère d'échappement lui-même). La suite des deux caractères `\x` est interprétée comme un seul caractère significatif. Ainsi la suite `\"` est interprétée comme le caractère `"` à l'intérieur de la chaîne. La suite `\\"` est interprétée comme le caractère `\` à l'intérieur de la chaîne, suivie du caractère `"` de fin de chaîne. Une chaîne simple ne peut pas contenir de caractère de retour à la ligne.

   En utilisant la syntaxe ocamllex, donner une expression régulière qui reconnaît exactement les chaînes simples de Python.

2. Un utilisateur veut représenter le texte suivant comme une chaîne de caractères Python.

   ```
   Toto dit :
   "Vivement les vacances!"
   ```

   Donner une chaîne simple Python dont la valeur est ce texte.

Correction :

1. *Une chaîne simple démarre par le caractère guillemet, ensuite il va y avoir une suite quelconque formée soit d'un caractère qui n'est ni un retour à la ligne (*`\n`*) ni un guillemet ni un backslash soit d'une suite de deux caractères formés d'un backslash suivi d'un caractère quelconque qui n'est pas un retour à la ligne l'expression se termine par une occurrence du caractère guillemet.*

   On obtient au final, l'expression régulière `'"'([^'\n' '"' '\\'] | '\\' [^'\n'])*'"'`

   On utilise la syntaxe ocamllex dans laquelle les caractères terminaux s'écrivent entre guillemets simples comme `'a'`*.*

2. *L'écriture d'un chaîne simple ne peut pas contenir un saut à la ligne mais par contre on peut utiliser le suite de caractères* $\backslash n$ *pour qu'un retour à la ligne soit inséré dans la valeur de la chaîne. Il ne faut pas confondre la manière dont on écrit un objet dans le langage de programmation (règles de syntaxe, avec l'usage d'un caractère d'échappement pour éviter les ambiguïtés) et la valeur que cette écriture représente (règles de sémantique).*

   ```
   "Toto dit :\n \"Vivement les vacances! \""
   ```

Exercice 3 *Grammaires (7 points)*

On se donne un ensemble de symboles terminaux : AT, RA, LP, RP, EOI.

On définit 3 grammaires, elles ont toutes pour symbole initial S et un symbole non-terminal T ainsi que la règle "initiale" $S ::= T\ \texttt{EOI}$.

On indique ci-dessous les non-terminaux et règles supplémentaires.

— La grammaire G_1 ajoute les règles

$$T ::= \texttt{AT} \mid T\ \texttt{RA}\ T \mid T\ \texttt{AT} \mid \texttt{LP}\ T\ \texttt{RP}$$

— La grammaire G_2 a comme symbole non-terminal supplémentaire P et A avec les règles

$$T ::= P\ \texttt{RA}\ T \mid P \qquad P ::= P\ \texttt{AT} \mid A \qquad A ::= \texttt{AT} \mid \texttt{LP}\ T\ \texttt{RP}$$

— La grammaire G_3 a comme symbole non-terminal supplémentaire A avec les règles :

$$T ::= A\ \texttt{RA}\ T \mid T\ \texttt{AT} \mid A \qquad A ::= \texttt{AT} \mid \texttt{LP}\ T\ \texttt{RP}$$

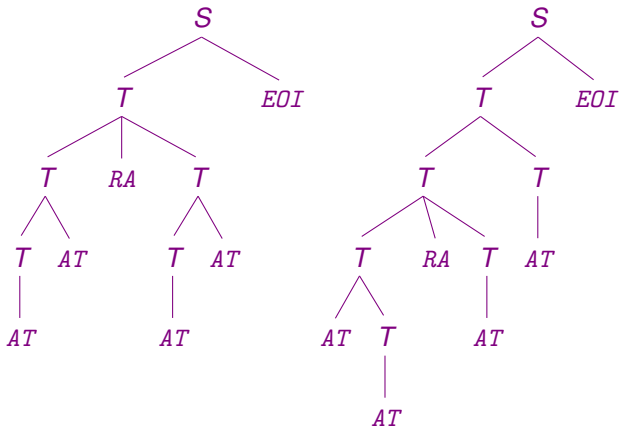
1. L'entrée AT AT RA AT AT EOI est-elle reconnue par les grammaires G_1 et G_3 ?
   Si oui donner l'arbre de dérivation. Sinon justifier.
2. On considère ici la grammaire G_2 :
   (a) Construire l'automate LR(0) de G_2. L'état initial est l'état de l'item $[S \prec \bullet T\ \texttt{EOI}]$ et on considèrera un état avec $[S \prec T\ \bullet \texttt{EOI}]$ comme un état de succès lorsque le prochain caractère est EOI.
   (b) Donner les équations récursives qui définissent les suivants des non-terminaux T, P et A.
   (c) Construire la table d'analyse SLR(1) pour cette grammaire (on rappelle qu'il s'agit de ne faire l'opération de réduction d'une règle $X \prec \alpha$ dans un état qui contient l'item $X \prec \alpha\bullet$ que lorsque le caractère de l'entrée appartient aux suivants de X).
3. Dire si chacune des grammaires ci-dessus est ambigüe en justifiant.
4. L'analyse par menhir de la grammaire G_1 a donné un état avec deux conflits.

```
State 6:
## Known stack suffix:
## T RA T
## LR(1) items:
T -> T . AT [ RP RA EOI AT ]
T -> T . RA T [ RP RA EOI AT ]
T -> T RA T . [ RP RA EOI AT ]
## Transitions:
-- On RA shift to state 5
-- On AT shift to state 7
## Reductions:
-- On RP RA EOI AT
--   reduce production T -> T RA T
** Conflict on RA AT
```

   (a) Pour chaque conflit, donner un exemple d'entrée sur laquelle le conflit se produit.
   (b) Donner le parenthésage de chaque entrée qui aurait été obtenu en utilisant la grammaire G_2.
   (c) Quelles précédences et règles d'associativité faut-il ajouter à la grammaire G_1 pour obtenir le même comportement que dans la grammaire G_2 ?

Correction :

1. *L'entrée* $AT\ AT\ RA\ AT\ AT\ EOI$ *est reconnue par la grammaire* G_1*. Il y a plusieurs arbres de dérivation possibles :*



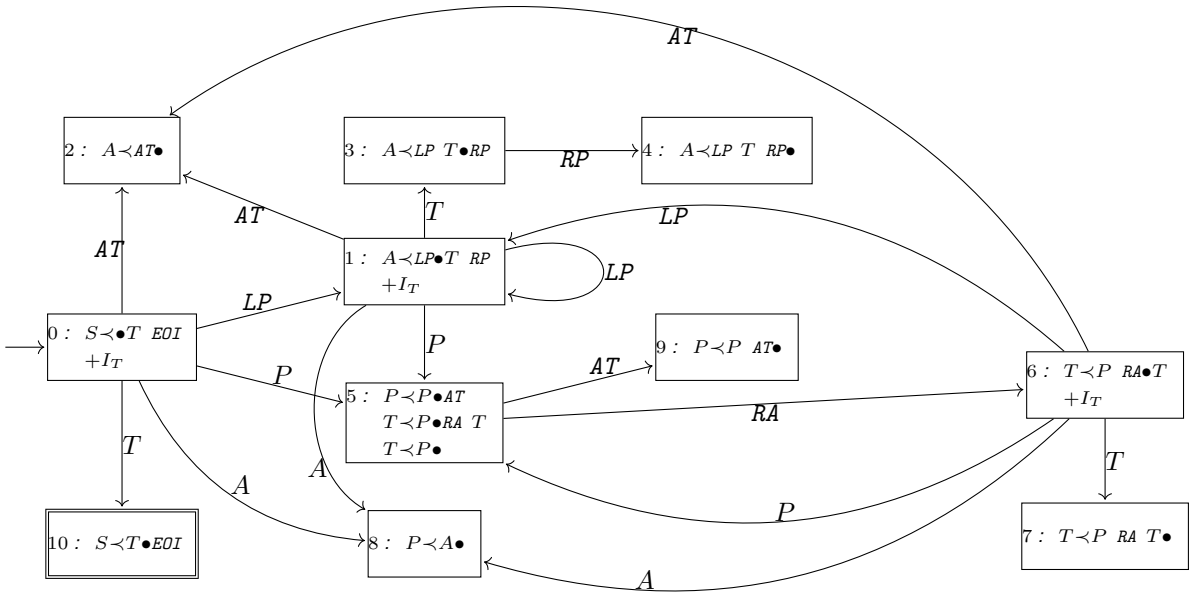
Par contre elle n'est pas reconnue par la grammaire G_3 *en effet après la première dérivation* $S \to TEOI$*, la seule règle qui s'applique est* $T ::= TAT$ *(l'entrée n'est pas atomique et pour appliquer la règle* $T ::= A\ RA\ T$*, il faudrait dériver* $A \to^* AT\ AT$ *ce qui n'est pas possible). On en revient à devoir dériver* $T \to^* AT\ AT\ RA\ AT$*, la seule règle qui s'applique est* $T ::= A\ RA\ T$ *mais on en revient encore à trouver une dérivation de* $A \to^* AT\ AT$ *qui n'existe pas.*

2. (a) *Lorsqu'un état contient un item avec* $\bullet T$*, il contient aussi les items suivants que nous notons* I_T

I_T :
$T \prec \bullet P\ RA\ T$
$T \prec \bullet P$
$P \prec \bullet P\ AT$
$P \prec \bullet A$
$A \prec \bullet AT$
$A \prec \bullet LP\ T\ RP$

.

L'automate LR(0) déterministe contient 11 états.



(b) *On considère à chaque fois les règles dans lesquelles le terminal apparaît à droite. Les équations pour calculer les suivants sont :*

— *Suivants*$(T) = \{$`EOI`, `RP`$\}$ *à cause des règles* $S ::= T$ `EOI` *et* $A ::=$ `LP` T `RP` *(la règle* $T ::= P$ `RA` T *induit une contrainte* *Suivants*$(T) \subseteq$ *Suivants*(T) *qui n'apporte aucun élément nouveau et est donc ignorée).*

— *Suivants*$(P) = \{$`RA`, `AT`$\} \cup$ *Suivants*(T) *à cause des règles* $T ::= P$`RA` T $P ::= P$ `AT` *et* $T ::= P$

— *Suivants*$(A) =$ *Suivants*(P) *à cause de la règle* $S ::= A$

On a donc :

$$\textit{Suivants}(T) = \{\texttt{EOI}, \texttt{RP}\} \quad \textit{Suivants}(P) = \textit{Suivants}(A) = \{\texttt{EOI}, \texttt{RP}, \texttt{RA}, \texttt{AT}\}$$

(c) *La table d'analyse SLR(1) est la suivante*

	`LP`	`AT`	`RA`	`RP`	`EOI`	T	P	A
0	1	2				10	5	8
1	1	2				3	5	8
2		$A \prec$ `AT`						
3				4				
4		$A \prec$ `LP` T `RP`						
5		9	6	$T \prec P$		3	5	8
6	1	2				7	5	8
7				$T \prec P$ `RA` T				
8		$P \prec A$						
9		$P \prec P$ `AT`						
10					*succes*			

Le seul état qui pourrait présenter un conflit entre lecture et réduction est l'état 5, mais on voit que la limitation de la réduction aux suivant de T *est suffisante pour choisir l'action à réaliser. La grammaire est donc SLR(1).*

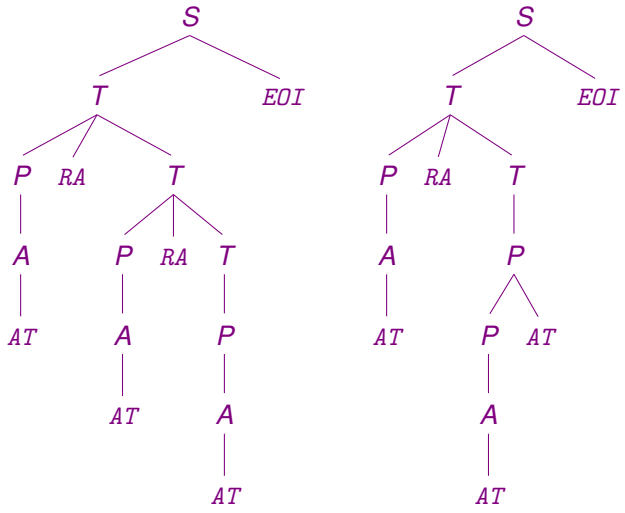
3. *La grammaire* G_1 *est ambigüe, nous avons exhibé 2 arbres pour l'entrée donnée, la grammaire* G_2 *est SLR(1) donc non-ambigüe. Attention, le seul lien entre conflit et ambiguïté est qu'une grammaire ambigüe aura toujours un conflit dans une analyse LL(k) ou LR(k). Par contre la réciproque est fausse, l'existence d'un conflit ne signifie pas que la grammaire est ambigüe, juste que l'analyse n'est pas assez puissante. C'est le fait d'avoir 2 arbres de dérivation différents pour la même entrée qui démontre l'ambigüité. Bien sûr l'analyse du conflit peut aider à trouver ces deux arbres.*

   Pour la grammaire G_3 *on a également une ambiguïté sur l'entrée* `AT RA AT AT` *qui peut se parenthéser en* `AT RA (AT AT)` *ou en* `(AT RA AT) AT`

4. (a) *Une entrée qui produit le conflit entre réduction et lecture de* `RA` *est* `AT RA AT RA AT EOI` *qui peut se parenthéser en* `AT RA (AT RA AT) EOI` *si lecture et* `(AT RA AT) RA AT EOI` *si réduction*

   Une entrée qui produit le conflit entre réduction et lecture de `AT` *est* `AT RA AT AT EOI` *qui peut se parenthéser en* `AT RA (AT AT) EOI` *si lecture et* `(AT RA AT) AT EOI` *si réduction*

(b) *La grammaire G_2 donne les arbres*



qui correspondent aux parenthésages `AT RA (AT RA AT) EOI` *et* `AT RA (AT AT) EOI`

(c) *Pour obtenir le comportement ci-dessus il faut déclarer* `RA` *comme un symbole associatif à droite et que le symbole* `AT` *soit plus prioritaire que* `RA`*. Cela se fait avec les deux lignes suivantes en respectant l'ordre.*

```
%right RA
%nonassoc AT
```

Mettre une associativité sur `AT` *n'aurait de sens que s'il y avait un conflit entre une règle de priorité* `AT` *à savoir* $T ::= $ `AT` *et* $T ::= T$ `AT` *et une lecture de* `AT`*. Or ce n'est pas le cas, dans ces deux situations, il n'y a pas de conflit, seule la réduction mène à une solution.*

Exercice 4 *Analyse de programmes, récurrence, compilation (7 points)*

Dans cet exercice, lorsqu'on demande de définir une fonction, on pourra écrire du code caml ou bien définir la fonction par des équations récursives par cas sur la structure des expressions.

On se donne un type d'arbre de syntaxe abstraite pour des expressions arithmétiques avec constantes entières, variables globales et l'opération d'addition. On se donne en plus une opération spécialisée d'ajout d'une constante à une expression en vue d'optimiser le code assembleur généré. On donne ci-dessous le type des arbres de syntaxe abstraite de notre langage. L'expression AddI(e,n) représente l'expression e à laquelle on a ajouté la constante n.

```
type expr =
    Cst of int
  | Add of expr * expr
  | Var of string
  | AddI of expr * int
```

1. La fonction d'évaluation des expressions dans un environnement qui associe des valeurs entières aux noms de variable est :

```
let eval env e =
  let rec erec = function
    | Cst n -> n
    | Var x -> List.assoc x env
    | Add(e1,e2) -> erec e1 + erec e2
    | AddI(e,n) -> (* completer *)
  in erec e
```

Compléter le cas du constructeur d'addition constante.

2. La compilation classique des expressions arithmétiques autres que les constantes dans un langage assembleur comme le MIPS consiste
   — pour une variable : accéder à la mémoire à partir de l'étiquette de la variable (nécessite un registre)
   — pour une addition constante : engendrer le code pour calculer e puis effectuer l'opération `addi`. Cela nécessite les registres pour calculer e, et si ce nombre est 0 (cas des constantes), il faut au moins un registre pour l'addition
   — pour une addition binaire, on engendre le code de la première opérande qu'on stocke dans un registre, puis on engendre le code de la seconde opérande avant de faire l'opération d'addition.

   On peut donc définir une fonction nbreg qui calcule le nombre maximum de registres nécessaires au calcul d'une expression non constante suivant cette méthode naïve.

```
let rec nbreg = function
    | Cst n -> 0
    | Var x -> 1
    | Add(e1,e2) -> max (nbreg e1) (1 + nbreg e2)
    | AddI(e,n) -> max 1 (nbreg e)
```

   (a) Donner une expression ne contenant que des additions binaires et des variables qui nécessite 8 registres pour être calculée suivant cette méthode.
   (b) Proposer une expression qui a la même valeur que l'expression précédente mais qui ne nécessite que 2 registres.

3. On appelle hauteur de l'arbre le nombre maximal de nœuds binaires `Add` (pas `AddI`) sur une branche.
   (a) Définir une fonction `ht` qui calcule la hauteur d'une expression.
   (b) Donner une borne supérieure du nombre de registres nécessaires en fonction de la hauteur de l'expression.
   (c) Démontrer par récurrence sur la structure de l'expression que le nombre de registres est toujours inférieur à votre borne supérieure.
   (d) Justifier que votre borne supérieure est atteinte en donnant une suite d'expressions, de hauteur non bornée, pour laquelle la borne est atteinte.

4. Afin de minimiser le nombre de registres nécessaires et de faire certains calculs dès la compilation, on introduit la notion d'expression optimisée.

   Une expression est optimisée si
   — soit c'est une constante
   — soit c'est une expression qui ne contient que des additions binaires et des variables

— soit c'est une expression de la forme AddI(e,n) avec e qui ne comporte que des additions binaires et des variables
— de plus, dans toute addition binaire Add(e1,e2), on demande que le nombre de registres pour le calcul de e1 soit supérieur ou égal au nombre de registres pour le calcul de e2.

Ecrire une fonction `transf` qui prend en argument une expression et la transforme en une expression optimisée de même valeur.

On conseille (fortement) d'utiliser une fonction annexe add qui prend en argument deux expressions optimisées e1 et e2 telles que le nombre de registres pour le calcul de e1 est supérieur ou égal au nombre de registres pour le calcul de e2 et qui renvoie une expression optimisée de même valeur que Add(e1,e2).

On ne demande pas de preuve complète par récurrence de la correction de la fonction, on s'attend par contre à des justifications sur des cas clés de pourquoi le résultat de la fonction transf est bien une expression optimisée de même valeur.

5. Commenter la qualité de votre solution : est-ce qu'elle permet d'obtenir la solution minimale en terme de nombre de registres nécessaires ? quelle est l'ordre de grandeur de complexité des fonctions proposées ? la méthode s'étend-elle simplement si on ajoute d'autres opérations arithmétiques (multiplication, soustraction) ? ou à l'ajout d'appels de fonctions dans les expressions ?

   Il est demandé des réponses courtes, claires et argumentées. Il n'est pas nécessaire de répondre à toutes les questions pour avoir le maximum (1 point), privilégier la qualité sur la quantité !

Correction :

1. *On complète par AddI(e,n) –> erec e + n. La variable n est un entier donc la fonction erec en peut pas lui être appliquée.*
2. (a) *Il est demandé une expression avec des variables, pas des constantes. Il faut prendre un arbre dans lequel la branche droite demande toujours un registre de plus. Par exemple :*

   Add(Var "x0",Add(Var "x1",Add(Var "x2",Add(Var "x3",Add(Var "x4",Add(Var "x5",Add(Var "x6", Var "x7")))))))

   (b) *Il suffit de rebalancer l'expression dans l'autre sens :*

   Add(Add(Add(Add(Add(Add(Add(Var "x0",Var "x1"),Var "x2"),Var "x3"),Var "x4"),Var "x5"),Var "x6"),Var "x7")
3. (a) *La hauteur d'un nœud binaire Add(e1,e2) est un de plus que le maximum des hauteurs des deux sous-arbres.*

```
let rec ht = function
    | Cst n -> 0
    | Var x -> 0
    | Add(e1,e2) -> 1 + max (ht e1) (ht e2)
    | AddI(e,n) -> ht e
```

   (b) *On a* $\texttt{nbreg}\, p \leq (\texttt{ht}\, p) + 1$

   (c) *On pose donc* $P(e)$ *la propriété* $\texttt{nbreg}\, e \leq (\texttt{ht}\, e) + 1$ *à montrer par récurrence sur la structure de l'arbre* e*.*

   — *Si* e *est une constante ou une variable, le nombre de registres est* 0 *ou* 1 *et la hauteur est* 0 *on a donc bien* $\texttt{nbreg}\, e \leq (\texttt{ht}\, e) + 1$

   — *Si* e *est de la forme* $\texttt{AddI}(e', n)$ *on a :*
   $\texttt{nbreg}\,(\texttt{AddI}(e', n)) = \texttt{max}\, 1\, (\texttt{nbreg}\, e')$ *et* $\texttt{ht}\,(\texttt{AddI}(e', n)) = \texttt{ht}\, e'$*.*

Par hypothèse de récurrence on a $\mathtt{nbreg}\, e' \leq (\mathtt{ht}\, e') + 1$ *et donc*
$\mathtt{nbreg}\,(\mathtt{AddI}(e', n)) \leq \mathtt{max}\, 1\, ((\mathtt{ht}\, e') + 1) = \mathtt{ht}(e') + 1$ *(puisque* $(\mathtt{ht}\, e') + 1 \geq 1$*) et donc*
$\mathtt{nbreg}\,(\mathtt{AddI}(e', n)) \leq \mathtt{ht}\,(\mathtt{AddI}(e', n))$.

— *Si* e *est de la forme* $\mathtt{Add}(e_1, e_2)$ *on a* $\mathtt{nbreg}\,(\mathtt{Add}(e_1, e_2)) = \mathtt{max}(\mathtt{nbreg} e_1)\,(1 + (\mathtt{nbreg}\, e_2))$ *et* $\mathtt{ht}\,(\mathtt{Add}(e_1, e_2)) = 1 + (\mathtt{max}\,(\mathtt{ht} e_1)\,(\mathtt{ht} e_2))$.

Par hypothèse de récurrence on a $\mathtt{nbreg}\, e_i \leq (\mathtt{ht}\, e_i) + 1$ *et donc*

$$\begin{aligned}\mathtt{nbreg}\,(\mathtt{Add}(e_1, e_2)) &\leq \mathtt{max}\,((\mathtt{ht}\, e_1) + 1)\,((\mathtt{ht}\, e_2) + 2)\\ &\leq \mathtt{max}\,((\mathtt{ht}\, e_1) + 2)\,((\mathtt{ht}\, e_2) + 2)\\ &= (\mathtt{max}\,(\mathtt{ht}\, e_1)(\mathtt{ht}\, e_1)) + 2\\ &= \mathtt{ht}\,(\mathtt{Add}(e_1, e_2)) + 1\end{aligned}$$

(d) *On reprend l'exemple de l'expression sous-forme de peigne d'addition binaire, généralisé à une hauteur* n *arbitraire. La hauteur est le nombre d'additions sur la branche droite et le nombre de registres est cette hauteur + 1*

4. *La fonction d'addition prend en argument deux expressions optimisées, s'il y a un* `AddI` *ou une constante alors c'est le symbole de tête. De plus on sait que le premier argument a besoin d'un nombre de registres au moins aussi grand que le second. On regarde tous les cas où il y a soit une constante soit un* `AddI` *à gauche et/ou à droite et on réalise l'addition de manière à respecter les conditions d'optimisation.*

   On remarque que comme le nombre de registres de $\mathtt{AddI}(e, n)$ *est celui de* e*, la condition sur les nombres de registres des arguments est bien préservée lorsqu'on introduit le constructeur* `Add`*. De même la condition sur les registres nous assure que si le premier argument est une constante alors forcément le second aussi.*

```
let rec add e1 e2 = match e1,e2 with
    Cst n1, Cst n2 -> Cst (n1+n2)
  | AddI(a1,m1), Cst n2 -> AddI(a1, m1+n2)
  | x, Cst n2 -> AddI(x,n2)
  | AddI(a1,m1), AddI(a2,m2) -> AddI(Add(a1,a2), m1+m2)
  | AddI(a1,m1), x -> AddI(Add(a1,x), m1)
  | x, AddI(a2,m2) -> AddI(Add(x,a2), m2)
  | _,_ -> Add(e1,e2)
```

   Concernant la fonction de transformation, on optimise les arguments puis on utilise notre addition optimisée en prenant soin de réordonner si besoin les arguments de manière à exécuter en premier celui qui demande le plus de registres.

```
let rec transf e = match e with
    | Add(e1,e2) ->
      let a1 = transf e1 and a2 = transf e2 in
      if nbreg a1 > nbreg a2 then add a1 a2 else add a2 a1
    | AddI(e,n) -> add (transf e) (Cst n)
    | _ -> e
```

5. — *Cette fonction ne donne pas la solution optimale car elle ne va pas réarranger un arbre équilibré sous forme de peigne. Ainsi une expression comme Add (Add (Var "x0", Var "x1"), Add (Var "x2", Var "x3")) va toujours nécessiter 3 registres après optimisation.*
   On pourrait aussi penser à optimiser en supprimant l'opération `AddI` *lorsque la constante est nulle.*

— *Du point de vue de la complexité, la fonction de transformation fait appel au calcul du nombre de registres afin de décider s'il faut réordonner. C'est une mauvaise idée d'un point de vue complexité qui devient alors quadratique, il serait plus judicieux de faire remonter le nombre de registres comme résultat de la fonction de transformation.*

— *En terme d'extension, réordonner n'est correct que parce que l'addition est commutative, en présence d'une opération comme la soustraction, il faudrait introduire un opérateur de soustraction alternatif auxilliaire qui commencerait par évaluer son second argument. La forme très simplifiée de nos expressions optimisées (avec au plus un seul `AddI` ou `Cst`) en tête est également très liée au fait qu'il n'y ait que l'addition. En présence de multiplication, on pourrait chercher des formes "canoniques", par exemple des polynômes.*

— *S'il y a des appels de fonction dans les expressions, les réarrangements ne peuvent se faire que si ces fonctions ne font pas d'effet de bord. Sinon le comportement du programme peut être altéré.*