




### Compilateur Numérix

#### Préambule

Ce sujet d'oral d'informatique est à traiter, sauf mention contraire, en respectant l'ordre du document. Votre examinatrice ou votre examinateur peut vous proposer en cours d'épreuve de traiter une autre partie, afin d'évaluer au mieux vos compétences.

Le sujet comporte plusieurs types de questions. Les questions sont différenciées par une icône au début de leur intitulé :

- les questions marquées avec  nécessitent d'écrire un programme dans le langage demandé. Le jury sera attentif à la clarté du style de programmation, à la qualité du code produit et au fait qu'il compile et s'exécute correctement ;
- les questions marquées avec  sont des questions à préparer pour présenter la réponse à l'oral lors d'un passage de l'examinatrice ou l'examinateur. Sauf indication contraire, elles ne nécessitent pas d'appeler immédiatement l'examinatrice ou l'examinateur. Une fois la réponse préparée, vous pouvez aborder les questions suivantes ;
- les questions marquées avec  sont à rédiger sur une feuille, qui sera remise au jury en fin d'épreuve.

Votre examinatrice ou votre examinateur effectuera au cours de l'épreuve des passages fréquents pour suivre votre avancement. En cas de besoin, vous pouvez signaler que vous sollicitez explicitement son passage. Cette demande sera satisfaite en tenant également compte des contraintes d'évaluation des autres candidates et candidats.

#### 1 Introduction

Dans ce sujet on souhaite créer un compilateur pour un langage fictif appelé le Numérix. Ce langage ne permet pas de définir de fonctions, juste un code en impératif (donc avec conditionnelles et boucles). Il ne fait qu'effectuer des calculs numériques sur les entiers (les flottants, les pointeurs et les tableaux n'existent pas). Le programme s'exécute et affiche dans la console, à la fin de son exécution, la valeur de chacune de ses variables.

Par exemple le code suivant calcule la somme des nombres de 1 à 100 et stocke le résultat dans *s*. La syntaxe est détaillée dans la section suivante.

```
s := 0;
i := 1;
TantQue < i 101 Faire
  s := + s i;
  i := + i 1
FinTq
```

Après exécution on souhaite avoir la sortie suivante dans la console.

```
s = 5050
i = 101
```

On définit inductivement les différents éléments du langage.

- Premièrement on définit ce qu'est une expression arithmétique: les cas de bases sont les variables (uniquement composées de lettres minuscules) et les constantes (en base décimal), les cas inductifs sont les opérations binaires (addition, soustraction, multiplication et division entière). Dans le langage Numérix, on commence toujours l'écriture d'une opération binaire par le symbole puis on ajoute les deux arguments. Par exemple pour réaliser l'opération  $18/3$  on écrira en Numérix `/ 18 3`. Grâce à cette notation, les parenthèses sont inutiles : l'expression  $(1 + 2) \times (3 - 4)$  s'écrit sans ambiguïté `x + 1 2 * 3 4`.
- Ensuite on définit les expressions logiques : on retrouve les constantes Vrai et Faux dans ce langage, ainsi que la comparaison d'expressions arithmétiques et les opérateurs logiques classiques en programmation (conjonction, disjonction et négation). Pour simplifier la compilation, Numérix n'a que deux opérateurs de comparaison d'expressions arithmétiques: l'égalité (=) et strictement inférieur (<). Comme pour les expressions arithmétiques, on commence l'expression logique par l'opérateur, par exemple la comparaison  $x + 1 > y$  s'écrit `> + x 1 y`. Pour les opérateurs logiques, on procède de même avec comme mot clés et, ou et non.
- Enfin, un programme est aussi défini inductivement:
  - Le programme Rien n'effectue aucune action
  - Le programme `x := e` avec `x` le nom d'une variable et `e` une expression arithmétique calcule la valeur de l'expression arithmétique et stocke le résultat dans la variable `x`.
  - Le programme `Si b Alors p1 Sinon p2 FinSi` est une conditionnelle avec `p1` et `p2` des programmes et `b` une expression logique.
  - Le programme `TantQue b Faire p FinTq` est une boucle exécutant le programme `p` tant que l'expression logique `b` est vraie.
  - Le programme `p1 ; p2` est un programme exécutant le programme `p1` puis `p2`.

Le langage n'est pas sensible au retour à la ligne ni à l'indentation qui sont interprétés comme des espaces.

**Q 1.** Dans un fichier `ex1.num` écrire un programme dans la syntaxe Numérix qui calcule  $25!$  et stocke le résultat dans une variable `s`.

## 2 Découpage en lexèmes

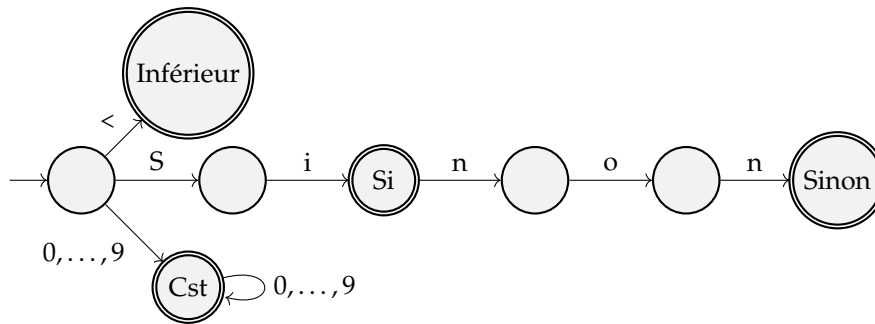
On appelle *lexèmes* l'ensemble des mots-clés et symboles du langage. Le nom des variables sont aussi des lexèmes. Par exemple, sur Numérix on va retrouver parmi les lexèmes: `23`, `x`, `+`, `;` et `TantQue`.

Le but de cette section est de construire l'*analyseur lexical* qui prend en entrée le texte du programme Numérix et renvoie une liste de lexèmes. Pour l'exemple présenté en section précédente, la liste commencera ainsi:

```
[ LVar "s" ; LAffectation ; LCst 0 ; LPointVirgule ; LVar "i" ; LAffectation ; LCst 1;
LPointVirgule ; LTantQue ; LInferieur ; LVar "i" ; LCst 101; LFaire; LVar "s";
LAffectation ; LPlus ; LVar "s"; LCst 1 ; ... ; LFinTq ]
```

### 2.1 Automates

On utilisera pour cela un automate fini déterministe. Son but est de commencer la lecture du code et d'accepter le plus long mot possible. Le dernier état acceptant trouvé détermine le lexème créé. On reprend alors la lecture du code jusqu'à avoir consommé tous les caractères (ou une erreur lexicale). L'automate ci-dessous n'est qu'une sous-partie de l'automate pour Numérix, il permet de reconnaître les constantes numériques, le symbole inférieur ainsi que les mots-clés Si et Sinon.



**Q 2.** Compléter cet automate pour reconnaître le symbole de l'affectation (`:=`) et les mots clés `Faire` et `FinSi`.

On souhaite, à partir d'un automate ne contenant qu'un état initial non-acceptant, ajouter successivement les états nécessaires pour reconnaître chacun des lexèmes. On utilisera une implémentation d'automates déterministes grâce à une table de hachage afin de rajouter facilement des états dans l'automate.

Récupérer les sources sur Cahier-De-Prépas. Dans le fichier `lexeur.ml`, un type pour l'automate déterministe est fourni, ainsi qu'une liste des lexèmes et leur syntaxe dans `Numérix`. Les constantes numériques et les noms des variables ne sont pas listés car il en existe une infinité.

Pour le type `automate`, le champ `nb`, qui précise le nombre d'états dans l'automate est mutable car celui-ci va évoluer au fur et à mesure que l'on ajoute des lexèmes reconnus par l'automate. On suppose que les états sont numérotés en partant de 0. L'automate a pour unique état initial l'état 0. La table de hachage `final` associe à chaque état une option sur un booléen: si l'état est associé à `None` c'est qu'il n'est pas final et ne reconnaît pas un lexème, sinon il reconnaît le lexème associé. La table de hachage `delta` associe à chaque couple d'état et de caractère l'état atteint par une transition depuis le premier état avec le caractère fourni.

**Q 3.** Créer l'automate `lexeur` initial. Il sera une variable globale qui ne sera pas passé en argument des fonctions. On l'initialise avec un unique état initial (non-acceptant).

**Q 4.** Écrire une fonction `ajouter_transition : int -> char -> int` telle que l'appel `ajouter_transition q a`:

- Renvoie l'état  $q'$  s'il existe une transition de  $q$  à  $q'$  avec la lettre  $a$  dans l'automate `lexeur`.
- Crée un nouvel état  $q'$  non acceptant dans l'automate et ajoute la transition de  $q$  à  $q'$  avec la lettre  $a$  et renvoie le nouvel état  $q'$  si la transition n'existait pas.

Pour ajouter un lexème à l'automate on va donc appeler cette fonction sur chacun des caractères du lexèmes, en partant de l'état 0. Le dernier état obtenu est rendu acceptant et on lui associe le lexème alors reconnu.

**Q 5.** Écrire une fonction `ajouter_lexeme : string -> lexem -> unit` qui ajoute à l'automate la reconnaissance d'un lexème dont on donne la syntaxe.

**Q 6.** Via une fonction récursive sur la liste des mots-clés fournies, ajouter tous les lexèmes de `Numérix` à l'automate `lexeur`.

On va maintenant rajouter la reconnaissance des constantes et des noms de variables.

**Q 7.** Créer un nouvel état dans l'automate pour reconnaître les constantes. Sur le même principe que l'automate présenté plus haut, ajouter les transitions appropriées pour reconnaître les nombres en notation décimale.

Les noms de variables sont composés uniquement de lettres minuscules, de  $a$  à  $z$ . On peut donc comme pour les constantes numériques partir de l'état initial pour faire la reconnaissance des noms des variables: tous les mots-clés du langage doivent commencer par une majuscule.

**Q 8.** Créer un nouvel état dans l'automate pour reconnaître les noms de variables. Sur le même principe que l'automate présenté plus haut, ajouter les transitions appropriées. On pourra s'aider de la chaîne `minuscules` fournies pour parcourir l'ensemble des lettres.

**Q 9.** De combien d'états l'automate est-il composé ?

## 2.2 Découpage

Le programme est supposé donnée comme une chaîne de caractère que l'on va découper en lexèmes. Supposons que le lexème précédent est été reconnu jusqu'au  $i$ -ième caractère (inclus). On cherche alors à reconnaître le plus long lexèmes à partir du caractère ( $i + 1$ ). On avance alors dans l'automate tant qu'il existe des transitions pour les lettres lues. Si on tombe sur un état acceptant on teste procède par backtracking: si continuer de lire le programme permet de reconnaître un lexème alors on retourne ce lexème long, sinon, si on a donc échoué à reconnaître un lexème plus long, alors on retourne celui de notre état acceptant courant.

**Q 10.** ✎ Donner le pseudo-code de cette méthode de backtracking pour reconnaître le lexème le plus long. La fonction prendra en argument le texte du programme, l'indice  $i$  de départ et l'automate `lexeur`. Elle donnera en sortie le lexème reconnu ainsi que l'indice de fin de lecture du lexème, ou `None` si aucun n'a été reconnu.

**Q 11.** ✎ Faites valider votre pseudo-code.

**Q 12.** 📖 Implémenter ce pseudo code dans une fonction de signature:  
`plus_long_lexeme : string -> int -> (lexem * int) option` .

On va maintenant découper l'ensemble du texte du programme. Si le texte ne peut pas être entièrement lu (la recherche d'un plus long lexème échoue) alors on termine l'analyse lexicale par un message d'erreur: "Erreur lexicale". Si le prochain caractère est un blanc (espace, tabulation  
t ou saut à la ligne  
n) alors on passe au caractère suivant.

**Q 13.** 📖 Implémenter la fonction `analyse : string -> lexem list` . On pourra construire la liste des lexèmes à l'envers et retourner cette liste en fin d'analyse.

On remarque cependant que l'analyse lexicale ne permet pas de savoir quelle variable ni quelle constante numérique a été reconnue. On va alors modifier le type de ces deux lexèmes comme ci-dessous. À la création de l'automate on laisse ces deux chaînes de caractères vides. En revanche, à chaque lecture réussie par un appel à `plus_long_lexeme` dans `analyse`, on peut préciser l

```
type lexem =  
2 | LCst of string | LVar of string  
3 | LPlus | LMoins | LFoix | LDiv  
4 ...
```

**Q 14.** 📖 Modifier le code comme décrit au-dessus pour prendre en compte ce stockage des constantes et variables dans les lexèmes.

**Q 15.** ✎ Faites valider vos changements.

## 2.3 Lecture

On va maintenant ouvrir le fichier d'exemple précédemment créé et effectuer l'analyse lexicale dessus.

**Q 16.** 📖 Écrire un fonction `analyse_fichier : string -> lexem list` qui prend en argument le nom d'un fichier et renvoie le résultat de son analyse lexicale.

## 3 Analyse syntaxique

On veut maintenant vérifier si la liste de lexème forme un programme correcte syntaxiquement.

**Q 17.** ✎ Proposer une grammaire pour ce langage. On prendra comme alphabet l'ensemble des lexèmes et pour symbole initiale  $P$ . On aura les symboles non-terminaux suivants:

- le symbole  $P$  permet d'obtenir une dérivation d'un programme;
- le symbole  $A$  permet d'obtenir une dérivation d'une expression arithmétique;
- le symbole  $B$  permet d'obtenir une dérivation d'une expression logique.

On peut introduire d'autres symboles non-terminaux.

Le fichier parseur.ml propose une définition inductive du langage. Le but de cette section est de créer un objet de type `program` à partir de la liste de lexème obtenues en section précédente. Pour chaque symbole non-terminal  $S$  et son type `type_S` d'objet dans `pareur.ml`, on construit une fonction `parse_S : lexem list -> (type_s * lexem list)`. Par exemple on construit `parse_A : lexem list -> (expr_arith * lexem list)` pour les expressions arithmétiques. Ces fonctions tentent de construire un élément de type `type_s` à partir du début de la liste de lexèmes. Elle renvoie l'élément construit ainsi que les lexèmes restants. Par exemple:

```
# parse_A [LPlus ; LVar "s"; LCst "1" ; LPointVirgule ; ...]
(ABin (Plus, Var "s", Cst 1), [LPointVirgule ; ...])
```

On pourra donc demander le programme via `parse_P`. Le résultat doit être un couple  $(p, l)$  où  $p$  est un objet de type `program` et  $l$  est la liste vide.

Dans le cas où une règle est de la forme  $S \rightarrow aXb$  avec  $X$  non-terminal et  $a$  et  $b$  terminaux, on va par exemple vérifier si la liste commence bien par le lexème  $a$ , puis appeler le parseur pour  $X$ , puis vérifier si la liste renvoyée par ce parseur commence bien par le lexème  $b$ .

Dans le cas d'une règle  $S \rightarrow X|XaS$  où  $S$  et  $X$  sont non-terminaux et  $a$  est terminal, on pourra remplacer par:  $S \rightarrow XS'$  et  $S' \rightarrow \epsilon|aS'$  avec  $S'$  un nouveau symbole non-terminal.

**Q 18.** Écrire l'ensemble des fonctions `parse_S` de votre grammaire. Certaines devront être mutuellement récursives. Si une fonction ne peut pas construire d'élément, elle renvoie un message d'erreur: "Erreur parsing:  $S$  était attendu" (en adaptant  $S$  à votre type d'objet à reconnaître.)

**Q 19.** Écrire une fonction `parse: string -> program` qui prend en argument le nom d'un fichier et renvoie le programme décrit par ce fichier, elle appellera la fonction d'analyse lexicale de la section précédente. Elle renverra un message d'erreur si la liste obtenue après construction du programme par `parse_P` n'est pas vide: "Erreur parsing: Fin de fichier attendue."

## 4 Traduction

On souhaite traduire le Numérix en langage C. Le premier exemple doit être traduit en:

```
1 #include <stdlib.h>
2 #include <stdio.h>
3
4 int main() {
5     // Déclaration des variables
6     int s;
7     int i;
8     // Exécution
9     s = 0;
10    i = 1;
11    while (i < 101) {
12        s = s + i;
13        i = i + 1;
14    }
15    // Affichage
16    printf("s=_%d\n", s);
17    printf("i=_%d\n", i);
18    return 0;
19 }
```

Dans un nouveau fichier `compilo.ml`

**Q 20.** Écrire une fonction `vars : program -> string list` qui renvoie l'ensemble des noms de variables utilisées ou affectées dans le programme donné. Cette liste doit être sans doublon.

**Q 21.** 📖 Écrire deux fonctions `arith_to_c : exp_arith -> string` et `log_to_c : exp_log -> string` qui renvoie une chaîne de caractère correspondant au code C calculant respectivement l'expression arithmétique ou logique donnée. Par exemple:

```
>> arith_to_c (ABin (Plus, Var "s", Cst 1))
"s + 1"
>> log_to_c (Non (Comparaison (Egal, Var "s", Cst 0))))
"!(s == 0)"
```

On décidera d'une stratégie pour placer les parenthèses, quitte à être trop prudent.

**Q 22.** 📖 Sur le même principe écrire une fonction `prog_to_c : program -> string` qui donne le code C correspondant au calcul du programme donné en argument (section "Exécution" dans l'exemple). On ne se préoccupe pas de déclarer les variables.

**Q 23.** 📖 Écrire une fonction `compile_numerix : program -> string` qui renvoie le code C (avec include, déclaration des variables et affichage de la sortie) correspondant au programme Numérix donné en argument.

**Q 24.** 📖 Enfin, écrire une fonction `compilation : string -> string -> ()` qui prend en argument deux noms de fichier. La fonction effectue une analyse lexicale et syntaxique du programme écrit dans le premier fichier et écrit sa traduction en langage C dans le second fichier.