Algorithmique avancée et Programmation C

Durée : *3h00*
Documents autorisés : **AUCUN**

Remarques :

- Veuillez lire attentivement les questions avant de répondre ;
- Rendez une copie propre ;
- En annexe vous trouverez les signatures de fonctions et procédures qui peuvent être utilisées pour écrire vos algorithmes.

Vocabulaire

Dans cet examen nous allons nous intéresser, entre autres, à quelques types de données et algorithmes du domaine de l'informatique théorique. Voici le vocabulaire associé :

un alphabet est un ensemble fini de lettres. En pratique les lettres sont représentées à l'aide de caractères ;

un mot est une suite finie de lettres. En pratique les mots sont représentés à l'aide de chaines de caractères ;

un langage est un ensemble (potentiellement infini) de mots ;

ϵ représente la lettre et le mot « vide ».

1 QCM

Répondez au qcm ci joint (à rendre avec votre copie). La note de chaque question peut varier de -1 à $+1$.

Soit B le nombre de bonnes réponses à une question et soit M le nombre de mauvaises réponses à cette même question ($B + M$ est égale au nombre de réponses à la question). Chaque bonne réponse cochée rapporte $1/B$ points et chaque mauvaise réponse $-1/M$ points pour la compétence évaluée par la question.

Solution proposée:

Compétences évaluées

- AN301 : Savoir lister les collections usuelles
- CD102 : Savoir calculer une complexité dans le pire et le meilleur des cas
- CD403 : Savoir concevoir et utiliser des arbres (binaires, n-aires)
- CP006 : Connaître le rôle de la conception préliminaire

2 La distance de Levenshtein

Pour rappel, « la distance de Levenshtein est une distance mathématique donnant une mesure de la similarité entre deux mots. Elle est égale au nombre minimal de lettres qu'il faut supprimer, insérer ou remplacer pour passer d'un mot à l'autre.

On appelle distance de Levenshtein entre deux mots M et P le coût minimal pour aller de M à P en effectuant les opérations élémentaires suivantes :

— substitution d'une lettre de M en une lettre de P ;

— ajout dans M d'une lettre de P ;

— suppression d'une lettre de M.

On associe ainsi à chacune de ces opérations un coût. Le coût est toujours égal à 1, sauf dans le cas d'une substitution de lettres identiques, il vaut alors 0. » (inspiré de Wikipédia).

Pour calculer cette distance on utilise un matrice m de taille $|P|+1 \times |M|+1$ (tel $|s|$ représente la longueur d'un mot s) indicée à partir de 0, tel que :

$$m_{0,j} = j, j \in 0..|M|$$

$$m_{i,0} = i, i \in 0..|P|$$

$$m_{i,j} = min(m_{i,j-1}+1, m_{i-1,j}+1, m_{i-1,j-1}+1_{P_i,M_j}), i \in 0..|P|, j \in 0..|M|$$

tel que $1_{P_i,M_j}$ vaut 0 si $P_i = M_j$ (la ième lettre de P est égale à la jème lettre de M), 1 sinon.

La distance de Levenshtein est alors égale à $m_{|P|,|M|}$.

1. Remplissez (sur votre copie) la matrice suivante pour calculer la distance de Levenshtein entre les deux mots `"voiture"` et `"toile"`.

$$m = \begin{array}{c} \\ \text{␣} \\ t \\ o \\ i \\ l \\ e \end{array} \begin{array}{c} \begin{array}{cccccccc} \text{␣} & v & o & i & t & u & r & e \end{array} \\ \begin{pmatrix} & & & & & & & \\ & & & & & & & \\ & & & & & & & \\ & & & & & & & \\ & & & & & & & \\ & & & & & & & \end{pmatrix} \end{array}$$

Solution proposée:

Compétences évaluées
— AN004 : Comprendre et savoir appliquer des consignes algorithmiques sur un exemple

$$m = \begin{array}{c} \\ \text{␣} \\ t \\ o \\ i \\ l \\ e \end{array} \begin{array}{c} \begin{array}{cccccccc} \text{␣} & v & o & i & t & u & r & e \end{array} \\ \begin{pmatrix} 0 & 1 & 2 & 3 & 4 & 5 & 6 & 7 \\ 1 & 1 & 2 & 3 & 3 & 4 & 5 & 6 \\ 2 & 2 & 1 & 2 & 3 & 4 & 5 & 6 \\ 3 & 3 & 2 & 1 & 2 & 3 & 4 & 5 \\ 4 & 4 & 3 & 2 & 2 & 3 & 4 & 5 \\ 5 & 5 & 4 & 3 & 3 & 3 & 4 & 4 \end{pmatrix} \end{array}$$

2. À quel paradigme de conception appartient cet algorithme ? Justifiez.
Solution proposée:

Compétences évaluées

— CD701 : Savoir définir la programmation dynamique

C'est un algorithme de programmation dynamique car :

— c'est un algorithme du type "diviser pour régner" qui calcule tout d'abord les résultats de base pour les assembler et ainsi calculer le résultat recherché : la valeur $m_{i,j}$ est la distance de Levenshtein entre les i premières lettres du mot M et les j premières lettre du mot P.

— il utilise un tableau pour stocker des valeurs qui sont susceptibles d'être calculées plusieurs fois.

3. Donnez l'algorithme de la fonction qui permet de calculer la distance de Levenshtein entre deux mots.
Solution proposée:

Compétences évaluées

— CD002 : En tant qu'utilisateur, savoir respecter une signature

— CD004 : Savoir écrire des algos avec le pseudo code utilisé à l'INSA

— CD005 : Savoir écrire un pseudo code lisible (indentation, identifiant significatif)

— CD006 : Savoir choisir la bonne itération

— CD009 : Savoir écrire un algorithme qui résout le problème

— CD702 : Savoir appliquer la programmation dynamique pour des cas simples

```
fonction cout (c1, c2 : Caractere) : Naturel
debut
   si c1=c2 alors
      retourner 0
   finsi
   retourner 1
fin
fonction distanceLevenhstein (mot1, mot2 : Chaine de caracteres) : Naturel
   ⌊précondition(s)  longeur(mot1)≤MAX et longueur(mot2)≤MAX
   Déclaration  m : Tableau[0..MAX][0..MAX] de Naturel
debut
   pour i ←0 à longueur(mot1) faire
      m[i,0] ← i
   finpour
   pour j ←0 à longueur(mot2) faire
      m[0,j] ← j
   finpour
   pour i ←1 à longueur(mot1 faire
      pour j ←1 à longueur(mot2) faire
         m[i,j] ← min3(m[i,j-1]+1, m[i-1,j]+1, m[i-1,j-1] + cout(iemeCaractere(mot1,i), iemeCaractere(mot2,j)))
      finpour
   finpour
   retourner m[longueur(mot1), longueur(mot2)]
fin
```

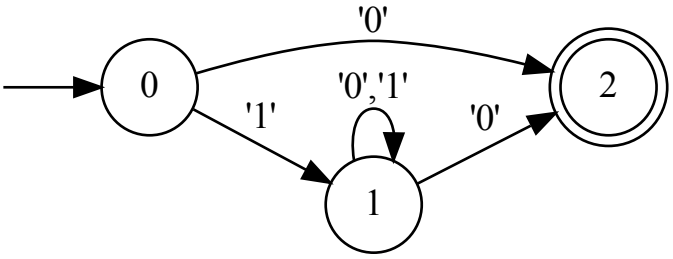


FIGURE 1 – Un exemple d'automate fini

3 Les automates finis

En informatique, un automate fini (appelé aussi simplement automate) est un graphe orienté valué par une lettre ou ϵ. Dans ce contexte, les sommets du graphe sont nommés les états de l'automate fini et les arcs du graphe sont nommés les transitions de l'automate. Il y a trois types états : l'état initial (unique dans un automate fini, identifié par une flèche entrante sans source), le ou les états intermédiaires et les états finaux (représentés graphiquement par un double cercle). Pour pouvoir désigner les états, on les numérote.

Lorsqu'une transition est valuée par ϵ, on dit que c'est une ϵ-transition, et l'automate en question est dit « avec ϵ-transition ».

Lorsque pour tous les états d'un automate il n'y a aucune ϵ-transition et qu'il n'y a pas plus d'une transition sortante qui soit étiquetée avec une même lettre, l'automate est dit « déterministe » sinon il est dit « indéterministe ».

La figure 1 représente un automate fini avec l'état 0 comme état initial et l'état 2 comme seul état final. Cet automate est dit indéterministe car depuis l'état 1, il y plusieurs transitions sortantes valuées par le caractère `'0'`.

Un automate fini permet de représenter un langage, nommé langage engendré. Par exemple l'automate fini de la figure 1 engendre le langage des représentations des nombres pairs en binaire.

Un automate permet aussi de savoir si un mot appartient au langage engendré. Il suffit pour cela de partir de l'état initial et pour chaque lettre du mot d'essayer de passer de l'état courant à un autre état en choisissant la (« bonne » ou l'unique dans le cas d'un automate déterministe) transition étiquetée par la lettre courante. Si après avoir traité la dernière lettre du mot, on est sur un état final, alors le mot appartient au langage engendré. Par exemple le mot `"110"` est bien la représentation d'un nombre pair en binaire car :

1. en partant de l'état initial 0, à l'aide du premier `'1'`, on passe à l'état 1 ;
2. puis à l'aide du deuxième `'1'` on reste dans l'état 1 ;
3. enfin à l'aide de la dernière lettre `'0'` on passe à l'état 2, qui est un état final.

Une ϵ-transition permet de passer d'un état à un autre sans consommer une lettre du mot à reconnaître.

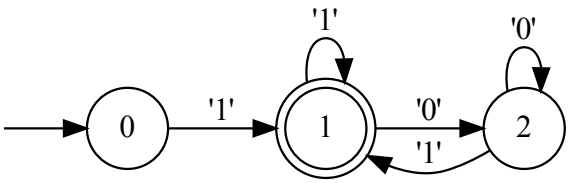
3.1 Compréhension

Dessinez un automate qui permet d'engendrer le langage des représentations des nombres impairs en binaire.

Solution proposée:

Compétences évaluées

— AN004 : Comprendre et savoir appliquer des consignes algorithmiques sur un exemple



3.2 Analyse

Spécifiez les TAD `Etat` et `Automate` (avec les axiomes) ayant les opérations suivantes :

— Etat :
  — création d'un état pour un automate donné ;
  — obtention de l'automate auquel appartient l'état ;
  — obtention du numéro de l'état.

— Automate :
  — obtention d'un automate avec uniquement un état initial ;
  — obtention de l'état initial ;
  — ajouter une transition entre deux états à partir d'un caractère ;
  — ajouter une ϵ-transition entre deux états ;
  — obtenir l'ensemble des états accessibles par ϵ-transition sortante depuis un état ;
  — obtenir l'ensemble des caractères de toutes les transitions sortantes depuis un état ;
  — obtenir l'ensemble des états atteignables par transition depuis un état et un caractère ;
  — obtenir l'ensemble des états ;
  — indiquer qu'un état est final ;
  — savoir si un état est final.

Solution proposée:

Compétences évaluées

— AN201 : Savoir identifier les dépendances d'un TAD
— AN203 : Savoir si une opération identifiée fait partie du TAD à spécifier
— AN204 : Savoir formaliser des opérations d'un TAD
— AN205 : Savoir formaliser les préconditions d'une opération d'un TAD
— AN206 : Savoir formaliser des axiomes ou savoir définir la sémantique d'une opération d'un TAD

Nom: Etat
Utilise: Automate, **Naturel**
Opérations: etat: Automate → Etat
automate: Etat → Automate
obtenirNumero: Etat → **Naturel**
Axiomes: - automate(etat(a))=a
- obtenirNumero(e1)≠obtenirNumero(e2)⇒e1≠e2

Nom: Automate
Utilise: Etat, **Caractere**, **Booleen**, Ensemble
Opérations:

automate:	→ Automate
etatInitial:	Automate → Etat
ajouterTransition:	Automate × Etat × Etat × **Caractere** ↛ Automate
ajouterEpsilonTransition:	Automate × Etat × Etat ↛ Automate
etatsAccessiblesParEpsilonTransition:	Automate × Etat ↛ Ensemble<Etat>
caracteresDesTransitions:	Automate × Etat ↛ Ensemble<**Caractere**>
etatsAccessiblesParTransitions:	Automate × Etat × **Caractere** ↛ Ensemble<Etat>
etats:	Automate → Ensemble<Etat>
fixerCommeFinal:	Automate × Etat ↛ Automate
estFinal:	Automate × Etat ↛ **Booleen**

Préconditions: ajouterTransition(a,e1,e2,c): estPresent(etats(a),e1) et estPresent(etats(a),e2)
ajouterEpsilonTransition(a,e1,e2): estPresent(etats(a),e1) et estPresent(etats(a),e2)
etatsAccessiblesParEpsilonTransition(a,e): estPresent(etats(a),e)
caracteresDesTransitions(a,e): estPresent(etats(a),e)
etatsAccessiblesParTransition(a,e,c): estPresent(etats(a),e)
fixerCommeFinal(a,e): estPresent(etats(a),e)
estFinal(a,e): estPresent(etats(a),e)
Axiomes: - a←automate(), etats(a)=ajouter(ensemble(), etatInitial(a))
- e←etat(a),estMembre(etats(a),e)
- estMembre(etatsAccessiblesParTransition(ajouterEpsilonTransition(a,e1,e2,c),c),e2)
- estMembre(caracteresDesTransitions(ajouterTransition(a,e1,e2,c),e1),c)
- estMembre(etatsAccessiblesParTransition(ajouterTransition(a,e1,e2,c),c),e2)
- estFinal(fixerCommeFinal(a,e),e)

3.3 Conception préliminaire

Donnez les signatures des fonctions et/ou procédures des opérations des deux TAD précédents.

Solution proposée:

Compétences évaluées

- CP003 : Savoir choisir entre une fonction et une procédure
- CP004 : Savoir concevoir une signature (préconditions incluses)
- CP005 : Savoir choisir un passage de paramètre (E, S, E/S)

— Etat
fonction etat (a : Automate) **:** Etat
fonction automate (e : Etat) **:** Automate
fonction numero (e : Etat) **: Naturel**

— Automate
fonction automate () **:** Automate
fonction etatInitial (a : Automate) **:** Etat
procédure ajouterTransition (**E/S** a : Automate, **E** e1,e2 : Etat, c : **Caractere**)
⌊**précondition(s)** estPresent(etats(a),e1) et estPresent(etats(a),e2)
procédure ajouterEpsilonTransition (**E/S** a : Automate, **E** e1,e2 : Etat)
⌊**précondition(s)** estPresent(etats(a),e1) et estPresent(etats(a),e2)
fonction etatsAccessiblesParEpsilonTransition (a : Automate, e : Etat) **:** Ensemble<Etat>
⌊**précondition(s)** estPresent(etats(a),e)
fonction caracteresDesTransitions (a : Automate, e : Etat) **:** Ensemble<**Caractere**>
⌊**précondition(s)** estPresent(etats(a),e)
fonction etatsAccessiblesParTransition (a : Automate, e : Etat, c : **Caractere**) **:** Ensemble<Etat>
⌊**précondition(s)** estPresent(etats(a),e)
fonction etats (a : Automate) **:** Ensemble<Etat>
procédure fixerCommeFinal (**E/S** a : Automate, **E** e : Etat)
⌊**précondition(s)** estPresent(etats(a),e)
fonction estFinal (a : Automate, e : Etat) **: Booleen**
⌊**précondition(s)** estPresent(etats(a),e)

3.4 Utilisation

3.4.1 Déterministe, ou pas ?

Donnez l'algorithme d'une fonction qui permet de savoir si un automate fini est déterministe.

Solution proposée:

Compétences évaluées

— CP004 : Savoir concevoir une signature (préconditions incluses)
— CD002 : En tant qu'utilisateur, savoir respecter une signature
— CD004 : Savoir écrire des algos avec le pseudo code utilisé à l'INSA
— CD005 : Savoir écrire un pseudo code lisible (indentation, identifiant significatif)
— CD006 : Savoir choisir la bonne itération
— CD009 : Savoir écrire un algorithme qui résout le problème
— CD011 : Savoir utiliser les bons types de données (paramètres formels, variables locales)

fonction estDeterministe (a : Automate) **: Booleen**

```
Déclaration  lEtats : Liste<Etat>
             etatCourant : Etat
             lCar : Liste<Caractere>
             i,j : NaturelNonNul
             resultat : Booleen
debut
  lEtats ← ensembleEnListe(etats(a))
  i ← 1
  resultat ← VRAI
  tant que i≤longueur(lEtats) et resultat faire
    etatCourant ← obtenirElement(lEtat,i)
    j ← 1
    lCar ← caracteresDesTransitions(a,etatCourant)
    tant que j≤longueur(lCar) et resultat faire
      resultat ← cardinalite(etatsAccessiblesParTransition(a,etatCourant,obtenirElement(lCar,j)))=1
      j ← j+1
    fintantque
    resultat ← resultat et cardinalite(etatsAccessiblesParEpsilonTransition(a,etatCourant))=0
    i ← i+1
  fintantque
  retourner resultat
fin
```

3.4.2 Appartient au langage engendré ?

Donnez l'algorithme itératif d'une fonction qui permet de savoir si un mot appartient ou pas au langage engendré par un automate fini que l'on sait déterministe.

Solution proposée:

Compétences évaluées

- CP004 : Savoir concevoir une signature (préconditions incluses)
- CD002 : En tant qu'utilisateur, savoir respecter une signature
- CD004 : Savoir écrire des algos avec le pseudo code utilisé à l'INSA
- CD005 : Savoir écrire un pseudo code lisible (indentation, identifiant significatif)
- CD006 : Savoir choisir la bonne itération
- CD009 : Savoir écrire un algorithme qui résout le problème
- CD011 : Savoir utiliser les bons types de données (paramètres formels, variables locales)

```
fonction appartientAuLangageEngendre (a : Automate, m : Chaine de caracteres) : Booleen
  ⌊précondition(s)  estDeterministe(a)
debut
  etatCourant ← etatInitial(a)
  i ← 1
  yAUneTransition ← VRAI
  tant que i≤longueur(m) et yAUneTransition faire
```

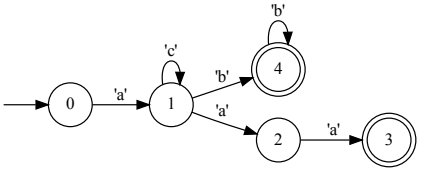
```
    prochainsEtats ← etatsAccessiblesParTransition(a,etatCourant,iemeCaractere(m,i))
    si cardinalite(prochainsEtats)=1 alors
      etatCourant ← obtenirElement(ensembleEnListe(prochainsEtats),1)
      i ← i+1
    sinon
      yAUneTransition ← FAUX
    finsi
  fintantque
  retourner i=longueur(m)+1 et estFinal(a,etatCourant)
fin
```

3.4.3 Concaténation

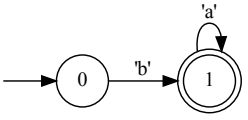
La concaténation de deux automates est un nouvel automate dont les états et les transitions sont les mêmes que les états et les transitions des deux automates concaténés, sauf que

- les états de l'automate calculé correspondants aux état finaux du premier automate ne sont pas finaux ;
- l'état de l'automate calculé correspondant à l'état initial du deuxième automate n'est pas initial ;
- il y a des ϵ-transitions entre les états correspondants aux état finaux du premier automate et l'état correspondant à l'état initial du deuxième automate.

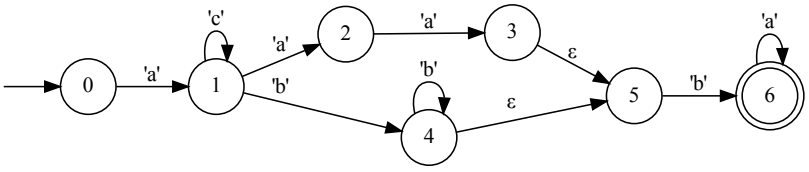
La figure 2 montre un exemple de concaténation de deux automates.



(a) 1er automate



(b) 2ème automate



(c) Résultat de la concaténation

FIGURE 2 – Exemple de concaténation de deux automates

Donnez l'algorithme d'une fonction directement ou indirectement récursive (dans ce dernier cas, c'est le sous-programme appelé qui est récursif) qui calcule la concaténation de deux automates finis.

Cet algorithme parcourt en profondeur (de par la récursion) les deux automates en commençant par l'état initial du premier automate à concaténer, tel que :

- un dictionnaire permet d'associer les états des deux automates à concaténer aux états de l'automate calculée (sur l'exemple de la figure 2 ce dictionnaire associe l'état 0 de l'automate 2a à l'état 0 de l'automate 2c ou encore l'état 0 de l'automate 2b à l'état 5 de l'automate 2c) ;

— pour l'état courant d'un des automates à concaténer :
  — il ajoute les transitions et ϵ-transitions sortante à l'état associé de l'automate calculé ;
  — si l'état courant est un état final du premier automate de la concaténation, il ajoute une ϵ-transition entre l'état associé à l'état courant et l'état associé à l'état initial du deuxième automates de la concaténation ;
  — si l'état courant est un état final du deuxième automate à concaténer, son état associé est aussi un état final.

Solution proposée:

Compétences évaluées

— CP004 : Savoir concevoir une signature (préconditions incluses)
— CD002 : En tant qu'utilisateur, savoir respecter une signature
— CD004 : Savoir écrire des algos avec le pseudo code utilisé à l'INSA
— CD005 : Savoir écrire un pseudo code lisible (indentation, identifiant significatif)
— CD006 : Savoir choisir la bonne itération
— CD009 : Savoir écrire un algorithme qui résout le problème
— CD011 : Savoir utiliser les bons types de données (paramètres formels, variables locales)
— CD201 : Savoir identifier et résoudre le problème des cas non récursifs
— CD202 : Savoir identifier et résoudre le problème des cas récursifs
— CD802 : Savoir écrire des algorithmes de parcours en largeur ou en profondeur

```
fonction concatenation (a1,a2 : Automate) : Automate
  Déclaration  resultat : Automate
               etatsCrees : Dictionnaire<Etat,Etat>
debut
  resultat ← automate()
  etatsCrees ← dictionnaire()
  ajouter(etatsCrees,etatInitial(a1),etatInitial(resultat)
  concatener(a1,a2,resultat,a1,etatInitial(a1),etatsCrees)
  retourner resultat
fin
procédure concatener (E a1,a2 : Automate, E/S resultat : Automate, E automateCourant : Automate,
etatCourant : Etat)
  Déclaration  etatCourantDuNouvelAutomate,e : Etat
               c : Caractere
debut
  etatCourantDuNouvelAutomate ← obtenirValeur(etatsCrees,etatCourant)
  pour chaque c de caracteresDesTransitions(automateCourant, etatCourant)
    pour chaque e de etatsAccessiblesParTransition(automateCourant, etatCourant, c)
      si non estPresent(etatsCrees,e) alors
        ajouter(etatsCrees, e, etat(resultat))
        concatener(a1,a2,resultat,automateCourant,e,etatsCrees)
      finsi
      ajouterTransition(resultat, etatCourantDuNouvelAutomate, obtenirValeur(etatsCrees,e), c)
```

```
    finpour
  finpour
  pour chaque e de etatsAccessiblesParEpsilonTransition(automateCourant, etatCourant)
    si non estPresent(etatsCrees,e) alors
      ajouter(etatsCrees, e, etat(resultat))
      concatener(a1,a2,resultat,automateCourant,e,etatsCrees)
    finsi
    ajouterEpsilonTransition(resultat, etatCourantDuNouvelAutomate, obtenirValeur(etatsCrees,e))
  finpour
  si estFinal(automateCourant,etatCourant) alors
    si automateCourant=a1 alors
      si non estPresent(etatsCrees(etatInitial(a2))) alors
        ajouter(etatsCrees, etatInitial(a2), etat(resultat))
        concatener(a1,a2,resultat,a2,etatInitial(a2),etatsCrees)
      finsi
      ajouterEpsilonTransition(obtenirValeur(etatsCrees, etatCourant), obtenirValeur(etatsCrees, etatInitial(a2)))
    sinon
      fixerCommeFinal(resultat, etatCourantDuNouvelAutomate)
    finsi
  finsi
fin
```

Annexe

Voici les signatures de fonctions et procédures dont vous avez peut être besoin dans cet examen.

Chaine de caractères

- **fonction** longueur (uneChaine : **Chaine de caracteres) : Naturel**
- **fonction** iemeCaractere (uneChaine : **Chaine de caracteres**, iemePlace : **Naturel) : Caractere**
  ⌊**précondition(s)** $0 < iemePlace$ et $iemePlace \leq longueur(uneChaine)$

Liste

- **fonction** liste **() :** Liste
- **fonction** estVide (uneListe : Liste**) : Booleen**
- **procédure** insérer **(E/S** uneListe : Liste, **E** position : **Naturel**, element : Element**)**
  ⌊**précondition(s)** $1 \leq position \leq longueur(uneListe) + 1$
- **procédure** supprimer **(E/S** uneListe : Liste, **E** position : **Naturel)**
  ⌊**précondition(s)** $1 \leq position \leq longueur(uneListe)$
- **fonction** obtenirElement (uneListe : Liste, position : **Naturel) :** Element
  ⌊**précondition(s)** $1 \leq position \leq longueur(uneListe)$
- **fonction** longueur (uneListe : Liste**) : Naturel**

Ensemble

- **fonction** ensemble () **:** Ensemble
- **procédure** ajouter (**E/S** unEnsemble : Ensemble, **E** element : Element)
- **procédure** retirer (**E/S** unEnsemble : Ensemble, **E** element : Element)
- **fonction** estPresent (unEnsemble : Ensemble, element : Element) **: Booleen**
- **fonction** cardinalite (unEnsemble : Ensemble) **: Naturel**
- **fonction** union (e1, e2 : Ensemble) **:** Ensemble
- **fonction** intersection (e1, e2 : Ensemble) **:** Ensemble
- **fonction** soustraction (e1, e2 : Ensemble) **:** Ensemble

Dictionnaire

- **fonction** dictionnaire () **:** Dictionnaire
- **procédure** ajouter (**E/S** unDictionnaire : Dictionnaire, **E** clef : Clef, element : Valeur)
- **procédure** retirer (**E/S** unDictionnaire : Dictionnaire, **E** clef : Clef)
- **fonction** estPresent (unDictionnaire : Dictionnaire, clef : Clef) **: Booleen**
- **fonction** obtenirValeur (unDictionnaire : Dictionnaire, clef : Clef) **:** Valeur
  ⌊**précondition(s)** estPresent(unDictionnaire, clef)
- **fonction** obtenirClefs (unDictionnaire : Dictionnaire) **:** Liste<Clef>
- **fonction** obtenirValeurs (unDictionnaire : Dictionnaire) **:** Liste<Valeur>

Fonction de transtypage

- **fonction** ensembleEnListe (e : Ensemble) **:** Liste