SI2 - FIP : Bonus
Analyse de Séquences ADN

Organisation

— Ce projet a pour objectif pédagogique de mettre en pratique l'ensemble des concepts étudiés dans ce module.

— Le projet est à réaliser en binôme, constitué de deux étudiants du **même** groupe de TP.

— Vous devez soumettre votre projet sur Moodle, avant la date limite indiquée sur Moodle. Ne soumettez qu'un projet par binôme, sur un seul compte utilisateur. Tout retard sera pénalisé.

Conseils

1. Réaliser le projet requiert d'avoir un environnement de programmation bien configuré : on demande donc ScalaIDE + Scalastyle (voir Notice ScalaIDE).
2. Il faut tester suffisamment ses fonctions, selon la méthodologie indiquée dans l'énoncé. Choisissez vos données de tests pour couvrir tous les cas représentatifs d'après la spécification des fonctions.
3. Les fonctions principales, auxiliaires, et les tests doivent être documentés
4. Programmez de manière élégante, concise et dans le respect du style fonctionnel : nommage des val, pas de code redondant, pas de code mort, pas de boucles `while` ou `for` lorsque les fonctions doivent être récursives, pas de variables globales mutables.

1 Méthodologie de travail

Fonctions à programmer L'essentiel de votre travail consiste à implémenter les fonctions indiquées dans les fichiers sources fournis par des tâches `TODO`, à visualiser dans la vue Task de ScalaIDE. Quand vous avez fini de traiter un `TODO`, supprimez-le. **Vous devrez suivre la feuille de route indiquée en Partie 5.**

Vous aurez besoin de définir des fonctions auxiliaires, à appeler dans les fonctions Scala demandées. Aussi, il est normal d'obtenir, à la fin de votre travail, davantage de fonctions Scala qu'à l'origine.

Test de vos fonctions Il est nécessaire de **tester** son code au fur et à mesure. Définir des tests demande du temps : vous pouvez vous répartir les tâches entre les deux membres du binôme pour gagner du temps. Les deux activités de programmation et de test peuvent être réalisées indépendamment, une fois que les spécifications des fonctions sont fixées. Deux stratégies de test sont possibles et complémentaires :

— tester manuellement, en affichant des résultats calculés, dans l'application principale.

— tester avec des tests unitaires. C'est la méthode de test la plus utile et la plus pérenne. Pour ce faire, utilisez les fichiers fournis. C'est à vous de programmer ces tests au fur et à mesure. Référez-vous à la Notice ScalaIDE. Attention, dans vos tests unitaires, veillez à n'utiliser que des données petites, simples, et que vous maîtrisez parfaitement : construisez-les à la main.

2 Introduction

L'objectif global de ce projet est de réaliser une application Scala, avec une interface graphique, qui permette d'analyser des (courtes) séquences d'ADN pour y identifier des motifs caractéristiques de certains marqueurs génétiques, ou bien des zones de l'ADN ciblées par certaines enzymes.

L'ADN, ou *Acide DésoxyriboNucléique* est une molécule biologique présente dans les cellules des êtres vivants. Elle contient toute l'information nécessaire à leur fonctionnement. Elle est structurée en une double hélice : deux brins d'ADN enroulés l'un autour de l'autre. De manière simplifiée [1], on peut considérer que chaque brin est une suite, ou séquence, de *bases azotées*. Les quatre bases azotées possibles sont : l'adénine (A), la cytosine (C), la guanine (G) et la thymine (T).

On souhaite que l'application Scala permette de :

— charger une séquence d'ADN contenue dans un fichier

— exprimer des motifs de bases azotées, au moyen d'expressions régulières (voir Partie 3)

— rechercher dans la séquence d'ADN les différentes occurrences d'un motif

— résumer le résultat de la recherche par un message simple

La Figure 1 ci-contre donne un exemple d'utilisation de l'application. La séquence ADN analysée est affichée dans la zone grisée. Le motif recherché dans la séquence est indiqué sur fond noir. Dans cet exemple, le motif recherché correspond à : deux bases T, suivies d'au moins une répétition des bases G ou C, suivie d'une base A. L'application indique combien de fois ce motif est présent dans la séquence chargée : ici, 81 fois. Chacun des motifs trouvés est également repéré graphiquement dans la séquence, par des couleurs qui alternent entre bleu et rouge. Des exemples de sous-séquences qui correspondent au motif recherché sont : TTGCGGA, TTCA, TTGA...

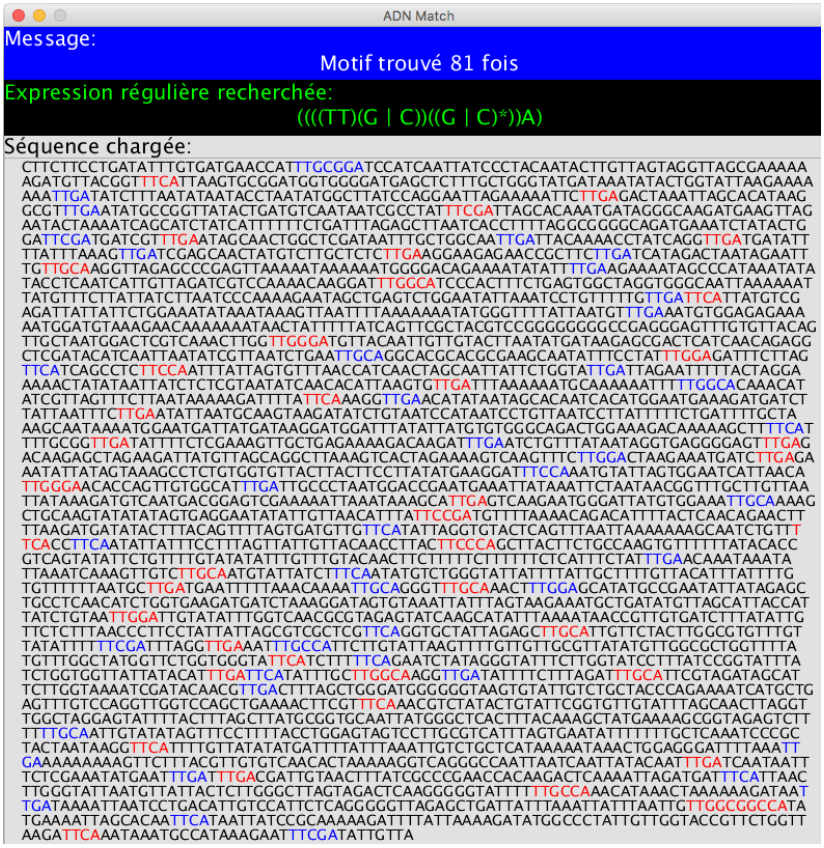

FIGURE 1 – Capture d'écran pour l'application finale à réaliser

Pour réaliser cette application, on utilisera la bibliothèque Scribble, disponible dans le cadre du module SI2. L'interface utilisateur de l'application est rudimentaire : les actions de chargement de fichier, de saisie de l'expression régulière, et de lancement de la recherche se feront en pressant différentes touches du clavier. Plus de détails sont donnés en Partie 5.3. La programmation de l'interface graphique n'est pas du tout prioritaire, pédagogiquement parlant. On vous en fournit donc une version, que vous pourrez adapter selon vos envies, si vous terminez d'abord toutes les fonctionalités demandées.

1. Plus d'information ici : `https://fr.wikipedia.org/wiki/Acide_desoxyribonucleique`

3 Expressions régulières

Les expressions régulières, issues de la théorie des langages formels, permettent de décrire de manière concise et finie des ensembles potentiellement infinis de séquences de caractères d'un alphabet $\mathcal{A}$ donné. Dans notre contexte, l'alphabet comprend les quatre bases azotées $\mathcal{A}$ = { A, T, G, C }. Dans la suite, une séquence de bases azotées sera notée entre crochets. Ex : [ATTGTAAAGC], [A], [ATGC], [TTG], et la séquence vide [].

Le Tableau 1 décrit les expressions régulières considérées dans ce projet, et les illustre avec quelques exemples et contre-exemples. On utilise des parenthèses dans les exemples pour délimiter les sous-expressions.

			Exemples		
Notation	Nom	Description	Expression Régulière	Séquences décrites	Séquences non décrites
$b \in \mathcal{A}$	base simple	séquence d'une seule base parmi A, T, G, ou C.	T	[T]	[C] , [A], [G], [AT], [GCC]...
$e_1 \mid e_2$	choix	les séquences décrites soit par e_1 soit par e_2.	A|T	[A],[T]	[G], [C]...
$\cdot$	base quelconque	séquence d'une seule base non précisée.	C·T	[CAT],[CTT], [CCT],[CGT]	[CT], [CCA]...
e_1e_2	concaténation	les séquences constituées d'une séquence décrite par e_1, puis d'une séquence décrite par e_2.	TT(A|C)	[TTA],[TTC]	[ATG], [A]...
e^*	répétition	les séquences constituées d'un nombre quelconque (potentiellement nul) de séquences décrites par e. Les séquences ne sont pas toujours identiques.	A(TG)*A	[AA],[ATGA], [ATGTGA], [ATGTGTGA], ...	[ACA], [ATA], [ATGCTG], ...
			(TG|A)*	[],[TG],[A], [TGAATG], ...	[C],[T], [TGCTG],...
$e\{n\}$	n-répétition	les séquences constituées d'exactement n séquences, chacune décrite par e.	A((TG){2})	[ATGTG]	[A],[ATG], [ATGTGTG],...
@	impossibilité	aucune séquence	@	aucune	toutes les séquences
%	vide	la séquence vide	%	[]	les séquences non vides

Table 1 – Expressions régulières considérées dans le projet

Les expressions régulières @ (impossibilité) et % (vide) peuvent paraître inutiles ou surprenantes. Leur utilité sera clarifiée en Partie 5.2.

Exercice 1 (Compréhension)**.** Lire le Tableau 1 attentivement et assurez-vous de comprendre chaque exemple et chaque contre-exemple.

Exercice 2 (Compréhension)**.** Pour les trois expressions régulières suivantes, donnez quelques exemples de séquences décrites, et de séquences non décrites : A(T|A), T*(·*)G, et T*(·{3})G.

4 Préliminaires

Import de projet En vous aidant de la Notice ScalaIDE, lancez l'éditeur ScalaIDE (Eclipse 4.7 Scala) et importez le projet ScalaIDE `ADNMatch.zip` disponible dans Moodle.

Auto-évaluation du projet Pour vous aider à auto-évaluer votre travail, il est nécessaire de tester votre code. Pour ce projet, nous ne fournissons pas de tests unitaires. **C'est à vous de définir des tests unitaires pour vérifier l'implémentation de vos fonctions.** Consultez la Notice ScalaIDE.

4.1 Architecture du projet fourni

Le projet `ADNMatch.zip` contient les répertoires et bibliothèque suivants :

- `src` contient tous les fichiers sources dans le package `fr.istic.si2.adnmatch` :
  - `app_{v1,v2,v3}.scala` : versions progressives des applications principales.
  - `rexp_basique.scala` : définition des types algébriques pour les expressions régulières, et fonctions de base sur les expressions régulières.
  - `rexp_match.scala` : analyse des séquences de bases azotées, calcul des résultats de recherche.
  - `sequences_images.scala` : représentation graphique des séquences de base azotées, avec ou sans marqueurs de résultat de recherche.
  - `rexp_universe.scala` : définition de l'interface graphique.
- `tests` contient tous les fichiers de test, dans le package `fr.istic.si2.test.adnmatch` (voir Partie 1).
- des fichiers `.jar`, contenant le nécessaire pour utiliser Scribble, saisir des expressions régulières au clavier, et lire des séquences ADN depuis un fichier.
- `fichiers` contient des fichiers `.txt` contenant des exemples de séquences ADN, plus ou moins longues, sur lesquelles tester vos fonctions et l'interface graphique. Vous pouvez en ajouter d'autres.

4.2 Modélisation des séquences et expressions régulières

Le fichier `rexp_basique.scala` contient la définition des deux types algébriques suivants.

Bases azotées, et séquences de bases azotées On modélise les bases azotées avec le type algébrique `Base` défini ci-contre. De plus, une séquence de bases azotées est modélisée par le type Scala `List[Base]`. Ainsi, la séquence [AAGC] est représentée par la liste `A::A::G::C::Nil`.

```scala
/** Type algébrique des bases azotées */
sealed trait Base
case object A extends Base
case object T extends Base
case object G extends Base
case object C extends Base
```

Expressions régulières Par définition des expressions régulières (Tableau 1), on a besoin d'un type algébrique récursif, pour pouvoir modéliser des expressions régulières arbitrairement complexes. Le type de données correspondant est `RExp`, défini ci-contre. La valeur `Nqb` correspond à l'expression régulière · décrivant une base quelconque.

```scala
/** Type algébrique des expressions régulières */
sealed trait RExp
case object Impossible extends RExp
case object Vide extends RExp
case object Nqb extends RExp
case class UneBase(b: Base) extends RExp
case class Choix(e1: RExp, e2: RExp) extends RExp
case class Concat(e1: RExp, e2: RExp) extends RExp
case class Repete(e: RExp) extends RExp
case class NFois(e: RExp, n: Int) extends RExp
```

5 Feuille de route

Test

5.1 Application Version 1 : prise en main, briques de bases

Exercice 3 (Définition de séquences)**.** Dans `app_v1.scala`, définissez à la main quelques `val` de type `List[Base]` . En complément, vous pourrez utiliser la fonction suivante fournie dans `adnmatchlib.jar` :

```
/** Lit au clavier une chaîne de caractères, terminée par un retour à la ligne
 *  @return la liste de bases azotées correspondantes saisies au clavier,
 *          si la chaîne entrée au clavier ne comporte que les caractères A, T, G, ou C. */
def lireSequence(): Option[List[Base]]
```

Exercice 4 (Représentation des `List[Base]`)**.** Les `List[Base]` sont adaptées à tous les calculs au coeur de la recherche de motifs. Dans l'application principale, en revanche, on souhaite offrir à l'utilisateur un affichage plus naturel que les listes de `Base` . Pour cela, définir la fonction suivante dans `rexp_basique.scala`.

```
/** @param lb une liste de bases azotées
 *  @return une chaîne de caractères représentant les bases de lb, dans l'ordre */
def listeBasesToString(lb: List[Base]): String = ???
```

Exercice 5 (Définition d'expressions)**.** Définissez dans `app_v1.scala` plusieurs `val` de type `RExp` . En complément, vous pourrez utiliser la fonction suivante fournie dans `adnmatchlib.jar`. La syntaxe à utiliser est celle du Tableau 1. Utilisez des parenthèses en tant que de besoin.

```
/** Construction d'une RExp à partir d'une chaîne de caractères.
 * Les espaces sont ignorés. Les parenthèses délimitent les sous-expressions.
 * Affiche en console un message d'avertissement si s est ambigüe.
 * @param s une chaîne de caractères
 * @return une expression régulière correspondant à s, si elle existe  */
def litRExp(s: String): Option[RExp]
```

Exercice 6 (Représentation des `RExp`)**.** On souhaite afficher une expression régulière de façon lisible à l'utilisateur. Pour cela, définir la fonction suivante dans `rexp_basique.scala`. Pour tester votre fonction : créer des `RExp` et contrôler que `litRExp` produit bien la même expression à partir de la représentation produite par `rExpToString`.

```
/** @param e une expression régulière
  * @return la représentation textuelle de e, avec toutes les parenthèses nécessaires */
def rExpToString(e: RExp): String = ???
```

Exercice 7 (Dérouler une expression)**.** Dérouler une expression régulière consiste à produire une séquence de bases décrite par cette expression. En général, plusieurs déroulages possibles existent. On se contente ici d'un déroulage où tous les choix sont fixés à l'avance. Définir la fonction suivante dans `rexp_basique.scala`.

```
/** @param e une expression régulière
 *  @return une liste de bases obtenue en déroulant e tout le temps de la même manière.
 *  @note Indiquez ici vos choix réalisés pour les répétitions, les choix, les Nqb. */
def deroule(e: RExp): Option[List[Base]] = ???
```

Cette fonction vous permettra de constituer des jeux de test pour le reste du projet.

Exercice 8 (Application principale V1)**.** Proposez dans `app_v1.scala` une application principale qui démontre le bon fonctionnement des fonctions précédentes : par exemple, une boucle d'interaction avec l'utilisateur qui lit une expression régulière au clavier, l'affiche de manière lisible, puis déroule cette expression, et affiche ce déroulage de manière lisible en console. Attention à la gestion des cas d'erreur.

5.2 Application Version 2 : analyse de séquences ADN

On s'intéresse maintenant à l'analyse de séquences ADN. On considère deux analyses. D'abord, on souhaite déterminer si une séquence entière est décrite par une expression régulière donnée : la séquence est un déroulage possible pour l'expression. Ensuite, on souhaite déterminer si une séquence ADN possède des sous-séquences décrites par une expression régulière donnée. Les fonctions seront définies dans `rexp_match.scala`.

Dérivée de Brzozowski Au coeur des deux analyses, on utilise la notion de dérivée de Brzozowski.

Soit e une expression régulière décrivant un ensemble de séquences S_e. La dérivée de Brzozowski de e par rapport à une base azotée b, notée $d_b(e)$ est une autre expression régulière e', qui décrit l'ensemble $S_{e'} = \{[s'] \mid [bs'] \in S_e\}$, c'est-à-dire l'ensemble des séquences de S_e dont on a enlevé la première base, si elles commençaient bien par b. Les séquences de S_e ne commençant pas par b sont ignorées.

Une autre façon de voir $d_b(e)$ est la suivante. L'expression e est le motif recherché dans une séquence. Si la séquence analysée commence par la base b, alors $d_b(e)$ est le motif qu'il reste à rechercher dans la séquence après sa première base. Selon ce principe, on définit $d_b(e)$ récursivement sur la définition de e.

$$d_b(@) = @ \qquad d_b(e_1|e_2) = d_b(e_1) \mid d_b(e_2) \qquad d_b(b') = \begin{cases} \% & \text{si } b = b' \\ @ & \text{sinon} \end{cases}$$

$$d_b(\%) = @ \qquad d_b(e^*) = d_b(e)(e^*) \qquad d_b(e_1e_2) = \begin{cases} d_b(e_1)e_2 & \text{si } e_1 \text{ ne décrit pas } [] \\ d_b(e_1)e_2 \mid d_b(e_2) & \text{sinon} \end{cases}$$

Exercice 9 (Compréhension)**.** Calculez à la main (papier crayon), en appliquant la définition ci-dessus, les dérivées suivantes : $d_A(GTA)$, $d_A(ATGC)$, $d_G(AT|GC)$, $d_C((A^*)CT)$.

Exercice 10 (Compréhension)**.** En vous inspirant de la définition ci-dessus, proposez une définition de la dérivée de Brzozowski pour la base quelconque $\cdot$, et la n-répétition $e\{n\}$.

Exercice 11 (Dérivée par rapport à une base)**.** Programmez la fonction de calcul de la dérivée de Brzozowski par rapport à une base azotée donnée. Vous aurez besoin de programmer une fonction auxiliaire déterminant si une expression régulière décrit la séquence vide []. Rédigez également des tests unitaires pour ces fonctions.

```
/** @param e une expression régulière
 *  @param b une base azotée
 *  @return la dérivée de Brzozowski de e par rapport à b  */
def derivee(e: RExp, b: Base): RExp = ???
```

Correspondance complète Pour déterminer si une séquence complète est décrite par une expression régulière donnée, le mécanisme de dérivée peut être ré-appliqué en suivant une à une toutes les bases de la séquence à analyser.

Exercice 12 (Correspondance complète)**.** Programmez la fonction d'analyse suivante.

```
/** @param e une expression régulière
  * @param lb une liste de bases azotées
  * @return vrai ssi la liste lb entière est décrite par e  */
def matchComplet(e: RExp, lb: List[Base]): Boolean = ???
```

Exercice 13 (Application principale V2)**.** Proposez dans `app_v2.scala` une application principale qui démontre le bon fonctionnement des fonctions précédentes : par exemple, une boucle d'interaction qui permet de saisir une expression régulière au clavier, l'affiche de manière lisible, lit au clavier une séquence, puis réalise l'analyse de correspondance complète, et affiche le verdict de l'analyse. Attention à la gestion des erreurs.

Recherche de sous-séquences On cherche maintenant à affiner la recherche : on souhaite déterminer si la séquence ADN à analyser possède des sous-séquences décrites par une expression régulière donnée. On ne tiendra pas compte des recouvrement possibles entre sous-séquences décrites.

La recherche de sous-séquences utilise aussi la dérivée de Brzozowski : on parcourt la séquence jusqu'à trouver un préfixe décrit par l'expression. On sait qu'un préfixe est décrit par l'expression dès que la dérivée par rapport à ce préfixe décrit la séquence vide. Le reste de la séquence, après le préfixe, est ré-analysé de la même manière. Par exemple, pour S = [ATGCTCCTGGCTGTTTTGTACTTTTTA], et l'expression T(T*)(A|C), les sous-séquences sans recouvrement de S ainsi identifiées seraient [TC], [TA], et [TTTTTA].

Pour cette analyse, le résultat n'est plus un simple booléen. On doit introduire une notion de marqueurs, utilisés pour indiquer, base après base, le résultat de la recherche. On annotera directement la séquence elle-même, base après base. Exemple : informellement, pour une séquence [ATCGATCGGGACCCTT], si on cherche les sous-séquences décrites par l'expression C(·*)A, et si on utilise les marqueurs ✓ et × en indice des bases, le résultat de recherche peut s'illustrer par $[A_{\times}T_{\times}C_{\checkmark}G_{\checkmark}A_{\checkmark}T_{\times}C_{\checkmark}G_{\checkmark}G_{\checkmark}G_{\checkmark}A_{\checkmark}C_{\times}C_{\times}C_{\times}T_{\times}T_{\times}]$.

En Scala, on s'appuiera sur un type algébrique dédié : le type `Marqueur` , et les résultats d'analyse seront des séquences de bases annotées, c'est-à-dire des `List[(Marqueur,Base)]` .

Exercice 14 (Définition des marqueurs)**.** Définir le type algébrique simple `Marqueur` en haut du fichier `rexp_match.scala`. Dans un premier temps, on souhaite juste indiquer si oui ou non, une base fait partie d'une sous-séquence décrite par une expression.

Exercice 15 (Marquer toute une séquence)**.** Lors de l'analyse de la séquence, vous aurez besoin de marquer des sous-séquences comme étant décrites ou non décrites. A cette fin, programmer les fonctions suivantes :

```
/** @param lb une liste de bases azotées
 *  @return la liste des bases de lb, marquées pour indiquer que la totalité de lb est décrite  */
def sequenceDecrite(lb: List[Base]): List[(Marqueur,Base)] = ???

/** @param lb une liste de bases azotées
 *  @return la liste des bases de lb, marquées pour indiquer que la totalité de lb n'est pas décrite  */
def sequenceNonDecrite(lb: List[Base]): List[(Marqueur,Base)] = ???
```

Exercice 16 (Difficile – Recherche de sous-séquences)**.** On suggère de programmer cette recherche avec deux fonctions auxiliaires. La première détermine le plus petit préfixe d'une séquence décrit par une expression donnée. La deuxième supprime ce préfixe trouvé de la séquence à analyser. Programmer ces deux fonctions :

```
/** @param e une expression régulière
 *  @param lb une liste de bases azotées
 *  @return s'il existe, le plus petit prefixe de lb qui est décrit par e  */
def prefixeMatch(e: RExp, lb: List[Base]): Option[List[Base]] = ???

/** @param pref une liste de bases azotées *préfixe* de lb
 *  @param lb une liste de bases azotées
 *  @return la sous-liste de lb située après le préfixe pref  */
def suppPrefixe(pref: List[Base], lb: List[Base]): List[Base] = ???
```

Puis, à l'aide de ces deux fonctions, programmer la recherche des sous-séquences décrites, sans recouvrement :

```
/** @param e une expression régulière
 *  @param lb une liste de bases
 *  @return une liste  (m1, base1)::...::(mN,baseN)::Nil, qui marque, base après base,
 *          les sous-listes de lb décrites par e. Les basei sont les bases de lb dans l'ordre. */
def tousLesMatchs(e: RExp, lb: List[Base]): List[(Marqueur, Base)] = ???
```

Exercice 17 (Message de résultat)**.** On souhaite indiquer par un message simple, le résultat global de la recherche. Pour ce faire, programmer la fonction `messageResultat` dans `rexp_match.scala`. **Attention**, le message doit être court. Il sera aussi affiché dans la fenêtre "Message" de l'interface graphique.

Exercice 18 (Application principale V2)**.** Dans `app_v2.scala`, adaptez votre application principale pour réaliser, **en plus**, la recherche des sous-séquences, et afficher le résultat global de l'analyse.

5.3 Application Version 3 : interface graphique

On s'intéresse maintenant à l'interface graphique de l'application. L'application principale est dans `app_v3.scala`, et utilise les fonctionalités programmées dans `rexp_universe.scala`. Vous devrez définir les fonctions restantes dans `rexp_match.scala` et `sequences_images.scala`.

Mode d'emploi de l'interface graphique L'application est réactive : elle interagit avec l'utilisateur via les touches du clavier, et la console ScalaIDE. Les touches gérées sont les suivantes (voir `rexp_universe.scala`) :

- **f** Lire une séquence depuis un fichier, en indiquant son nom dans la console.
- **e** Lire une expression régulière au clavier, saisie en console.
- **c** Déterminer si la séquence entière est décrite l'expression régulière courante, et afficher le résultat.
- **m** Rechercher les sous-séquences décrites par l'expression régulière courante, et afficher le résultat.
- **a** Annuler le résultat de la dernière recherche effectuée.
- **q** Terminer l'application.
- **esc** Fermer la fenêtre graphique de l'application réactive.

Exercice 19 (Gestion des marqueurs)**.** Pour maintenir à jour les résultats de la recherche, et pouvoir déclencher des nouvelles recherches, l'application requiert de pouvoir ré-initialiser le résultat d'une recherche, et d'extraire une séquence "nue" depuis un résultat de recherche. Pour cela, programmez les fonctions `annulerResultat` et `sansMarqueurs` dans `rexp_match.scala`.

Exercice 20 (Retours à la ligne)**.** Pour visualiser convenablement la séquence et le résultat de la recherche, on découpe la séquence en plusieurs morceaux, chacun à afficher sur une ligne. Pour cela, définir la fonction `lignes` suivante dans `sequences_images.scala`. On suggère de définir deux fonctions auxiliaires : la première calcule le contenu de la première ligne, la deuxième détermine le reste de la séquence à découper.

```
/** @param lmb une liste de bases marquées
 *  @param tligne entier strictement positif, taille de ligne en nombre de bases marquées
 *  @return la liste contenant les sous-listes de lmb, toutes de taille tligne, sauf p.-être la dernière. */
def lignes(lmb: List[(Marqueur, Base)], tligne: Int): List[List[(Marqueur, Base)]] = ???
```

Exercice 21 (Images pour séquences de bases marquées)**.** Il ne reste plus qu'à traduire la liste des lignes de bases marquées en une image pour l'afficher dans la fenêtre graphique. Pour cela, programmez les fonctions `marqueurBaseToImage`, `imageUneLigne`, et `imagePlusieursLignes` dans `sequences_images.scala`.

Pour les dimensions de la fenêtre proposée, on vous conseille d'utiliser des images contenant du texte de la bonne taille. Utiliser pour cela la `val fontSizeBase` définie dans `sequences_images.scala`.

Exercice 22 (Bonus)**.** Etendre l'application pour que la recherche des sous-séquences distingue les sous-séquences décrites, de façon à les dénombrer. Pour cela, étendre les marqueurs possibles, et programmer une nouvelle fonction de recherche. Enfin, il faudra étendre la fonction `react` définie dans `rexp_universe.scala` pour déclencher cette nouvelle recherche, en pressant la touche de votre choix.