MIMot
Rapport de projet

Sylvain Périfel Nicolas Padoy Mathieu Hoyrup
Sébastien Hinderer Pierre Guillon Thomas Fernique
Binh-Minh Bui-Xuan

17 Décembre 2002

Projet MIM 2.1
ENS Lyon
http://www.ens-lyon.fr/~pguillon/mimot

1 Présentation

De nombreuses opérations sur les langages formels sont singulièrement fastidieuses à réaliser à la main, comme le calcul de la plus longue sous-séquence commune à deux mots, la vérification de l'appartenance d'un mot à un langage (du moins dans certains cas), la vérification de l'équivalence de deux expressions rationnelles, *etc.* Pourtant, bon nombre de ces opérations sont décidables, et une machine aurait bien moins de mal à les effectuer qu'un être humain.

Il semblerait par conséquent pratique et utile de disposer d'un *logiciel de langages formels* afin d'automatiser les tâches susdites. Plus concrètement, il s'agirait d'une interface simple à l'image de Maple, destinée à faciliter les manipulations de mots, des langages et des automates en regroupant un certain nombre d'opérations les concernant. S'il existe beaucoup de calculateurs formels, ce type de programme demeure en revanche assez rare.

Le projet MIMot est né pour répondre à ce besoin.

1.1 Avant MIMot

Le domaine des langages formels, bien que pauvre au niveau logiciel, disposait tout de même de quelques outils antérieurs à MIMot. On recense notamment le projet Finite State Automata Utilities de l'Université de Groningen aux Pays-Bas, qui se spécialisait dans les langages rationnels et les automates finis.

La distribution standard de Caml contient déjà une librairie de manipulation des expressions rationnelles, qui était convertie par Xavier Leroy de la librairie C Str, orientée pour le shell. Une autre librairie, Regexp, a été développée au LRI par Claude Marché, qui se rapprochait plus que de notre conception des expressions rationnelles, mais présentait quelques limites, notamment la borne du nombre de lettres aux 256 caractères ASCII (et encore, sans les mots-clés du langage). Enfin, une librairie permettant les opérations de base sur les automates, AutomatX, a été développée il y a cinq ans par Michel Quercia. Mais dans ces trois cas, le domaine concerné était restreint; ce qu'apporte MIMot est un nouvel ensemble d'outil, vaste et facile à utiliser.

1.2 Développement

Le programme a été écrit en langage Objective Caml, qui se prêtait tout à fait à la manipulation de types abstraits qu'il nécessitait (mots, expressions rationnelles, grammaires, automates...) et à la récursivité qu'induisait les

algorithmes les manipulant. Cela facilitait également les analyse lexicale et syntaxique, les outils OCamlLex et OCamlYacc étant sûrs et performants.

Le projet MIMot propose donc à l'utilisateur une bibliothèque de fonctions de manipulation des langages sous Caml, mais, pour les pauvres néophytes qui méconnaissent ce langage, il dispose également d'une interface texte et d'une interface graphique.

1.3 Organisation

Les algorithmes ont été divisés en plusieurs sous-projets, selon les structures abstraites définies et manipulées, d'abord indépendemment, puis en interaction les uns avec les autres:

- Mots et langages finis
- Expressions rationnelles
- Grammaires
- Automates

D'autre sous-projets ont eu pour but la mise en place de l'interfaçage entre l'homme et la machine.

- Langage de l'interface, liaison avec les parties algorithmiques]
- Interface texte
- Interface graphique

2 Mots et langages finis

2.1 Description du sous-projet

Ce sous-projet a pour objectif l'implantation des fonctions qui travaillent sur des mots ou des ensembles finis de mots. Il s'agit aussi bien de l'écriture des fonctions de base telles que l'énumération des préfixes d'un mot ou le calcul de son mirroir, que de la mise en oeuvre d'algorithmes plus complexes comme par exemple la recherche de motifs ou de carrés.

2.2 Types de base

Les mots constituent les premiers objets non triviaux de MIMot. Ils ont été définis comme une liste de lettres où les lettres sont codées par des entiers:

```
type letter_t = int
type word_t = letter_t list
```

On dispose ainsi de plusieurs milliards de lettres. Même si cette quantité de lettres parait amplement suffisante dans la pratique, il sera toujours possible, si le besoin s'en fait sentir par la suite, de remplacer le type **int** par le type **bigint** en apportant quelques modifications mineures.

La structure de tableau faisait elle aussi un bon candidat pour incarner le type des mots. Mais comme pour chacune de ces deux structures il existe des algorithmes dont l'écriture est simplifiée, c'est afin de privilégier la clarté et la lisibilité du code que la structure de liste a été retenue. Pour les algorithmes qui utilisent intrinsèquement la notion de tableau, afin d'éviter des accés multiples aux éléments en temps linéaire, les listes sont converties au préalable en tableaux.

2.3 Fonctions sur les mots

Il s'agit des fonctions figurant dans le module Words.

Les fonctions de base sur les mots reprennent, lorsque c'est possible, les fonctions sur les listes déjà implémentées en CAML. Il s'agit des fonctions suivantes:

Fonction	Description
word_empty	Renvoie le mot vide
word_is_empty u	Teste si le mot u est vide
word_of_letter l	Renvoie le mot composé de l'unique lettre l
word_add l u	Concatène la lettre l au début du mot u
word_add_end l u	Concatène la lettre l à la fin du mot u
word_concat u v	Renvoie la concaténation des mots u et v
word_power u i	Renvoie le mot u élevé à la puissance i
word_reverse u	Renvoie le mirroir du mot u
word_length u	Renvoie la longueur du mot u
word_head u	Renvoie la première lettre du mot u
word_last u	Renvoie la dernière lettre du mot u
word_tail u	Renvoie le mot u privé de sa première lettre
word_liat u	Renvoie le mot u privé de sa dernière lettre
nth u i	Renvoie la lettre en position i du mot u
is_palindrom u	Vérifie si le mot u est un palindrome

Une nouvelle exception s'ajoute à celles qui peuvent être levées par le module List:

```
exception Too_short_word
```

On dispose ensuite de fonctions aux noms relativement explicites destinées à un usage interne et s'intéressant plus particulièrement à l'ensemble des lettres d'un mot:

```
is_word_in_alphabet, is_word_in_alpha_bar, add_word_to_alphabet,
add_word_to_alphabet_disjoint
```

Les fonctions suivantes sont destinées à manipuler les sous-mots:

prefix u i	Renvoie le préfixe de u de longueur i
suffix u i	Renvoie le suffixe de u de longueur i
factor u i n	Renvoie le facteur de longueur n débutant en position i
lcpref u v	Renvoie le plus long prefixe commun à u et v
lcsuff u v	Renvoie le plus long suffixe commun à u et v
is_factor u v	Vérifie si u est un facteur de v
is_prefix u v	Vérifie si u est un préfixe de v
is_suffix u v	Vérifie si u est un suffixe de v

S'ajoutent à ces fonctions la fonction **longest_common_subsequence** qui calcule par programmation dynamique avec une complexité en $\mathcal{O}(n^3)$ la

plus longue sous-séquence commune à deux mots, et la fonction **is_subsequence** qui vérifie si le premier mot passé en argument est une sous-séquence du second. La fonction **find_matchings** prend en argument un motif et un mot et renvoit en temps linéaire les positions où apparait le motif dans le mot. La fonction **is_factor** en est dérivée. Enfin la fonction **find_squares** retourne en temps quadratique la liste des positions où apparaissent des carrés dans le mot passé en argument. Pour implémenter les deux dernières fonctions, on se sert d'une fonction f qui calcule en temps linéaire à partir d'un mot u un tableau t tel que $t(i)$ soit la longueur du plus grand facteur de u débutant en position i et qui soit aussi un préfixe de u. La fonction **find_matchings** applique f à la concaténation du motif et du texte, tandis que **find_squares** utilise la caractérisation suivante: u admet un carré en position k si et seulement si

$$\exists i \geqslant 0 \quad f\left(u[k \dots n]\right)_i \geqslant i$$

Provisoirement, ces deux dernières fonctions ne renvoient dans le logiciel mimot que la première position où l'objet recherché est trouvé, en l'absence de type dans le langage permettant de renvoyer l'ensemble des résultats.

Les morphismes sont implémentés comme des listes de couples, chaque couple assurant la correspondance entre une lettre et son image. Le calcul de l'image d'un mot par un morphisme prend un temps $\mathcal{O}(nm)$ où n est la taille du mot et m la taille de l'ensemble source du morphisme. Si la partie morphisme venait à être développée plus en profondeur, sans doute serait-il préférable d'utiliser une structure plus complexe pour stocker les morphismes comme par exemple la structure de tas, afin d'abaisser cette complexité à $\mathcal{O}(n \times log(m))$.

En raison de l'actuelle absence des listes et des couples dans le langage de mimot, ou dans l'attente du codage des morphismes comme objets de base du langage, ces deux fonctions ne sont pas encore utilisables dans l'interface.

La plupart des fonctions précédentes sont utilisées dans l'interface par des raccourcis, en voici deux exemples: le premier récupére un facteur, le second est une recherche de motif:

```
# abbccbba[4,2];
word : "cc"
# ee < abeeg;
bool : true
#
```

2.4 Fonctions sur les langages finis

Les langages finis sont définis (dans le module Finite) à l'aide du module Set. On dispose ainsi des opérations de base sur les langages finis:

Fonction	Description
is_finite_empty l	Teste si le langage l est vide
finite_cardinal l	Renvoie le cardinal du langage
is_word_in_finite u l	Teste l'appartenance de u à l
finite_add u l	Renvoie le langage l auquel on adjoint u
finite_remove u l	Renvoie le langage l privé du mot u
finite_union l1 l2	Renvoie l'union des deux langages
disjoint_finite l1 l2	Teste si les deux langages sont disjoints
finite_of_list s	Crée un langage fini à partir d'une liste de mots

Finalement, les fonctions suivantes créent les langages finis constitués respectivement des préfixes, des suffixes, des facteurs, et des sous-mots (au sens large) du mot passé comme argument: **prefixes, suffixes, factors, subwords**.

2.5 Perspectives

Voici un ensemble non exhaustif de fonctions qui pourraient étendre les fonctionnalités de mimot:

- la recherche de cubes, et plus généralement la recherche de la présence d'un motif de forme quelconque (par exemple xxyy pour deux carrés consécutifs). [?] présente d'autres singularités qu'il est possible de rechercher dans des mots.
- la résolution d'équations aux mots particulières
- la vérification de propriétés sur les morphismes
- dans un langage fini, la recherche des facteurs, sous-mots (...) communs à tous les éléments
- des fonctions sur les codes
- ...

3 Expressions rationnelles

Les expressions rationnelles sont définies par induction à partir de l'expression vide et des mots, grâce aux opérateurs unaire étoile et binaires plus et concaténation. Nous utilisons lors de leur construction des constructeurs particuliers qui les simplifient et les factorisent à la volée grâce à un petit système de réécriture (bien sûr non complet et par conséquent toutes les simplifications ne seront pas effectuées), avec . associatif et + associatif et commutatif:

$$\begin{aligned}
\emptyset.E &\to \emptyset \\
\varepsilon.E &\to E \\
E^*.E^* &\to E^* \\
E^*.E &\to E.E^* \\
E + F &\to F \text{ si } E \subseteq F \\
\emptyset + E &\to E \\
E^* + E^n &\to E^* \\
E.F_1 + E.F_2 &\to E.(F_1 + F_2) \\
E_1.F + E_2.F &\to (E_1 + E_2).F \\
\emptyset^* &\to \varepsilon \\
\varepsilon^* &\to \varepsilon \\
E^{**} &\to E^* \\
(E + E^n)^* &\to E^*
\end{aligned}$$

Cela permet un affichage plus lisible des expressions et facilite la manipulation, notamment les algorithmes qui ont une complexité coûteuse en la taille de l'expression.

Nous expliquons ici quelques algorithmes sur les expressions rationnelles implantés dans MIMot.

3.1 Appartenance d'un mot à un langage rationnel

Si Σ est un alphabet, e une expression rationnelle sur Σ, et w un mot (fini) sur Σ, `word_in_reg` (w, Σ) teste l'appartenance de w au langage $\lambda(e)$ dénoté par e. L'algorithme travaille par induction sur la structure de e, puis sur la longueur de w. J'écrirai, pour simplifier, $w \in e$ plutôt que $w \in \lambda(e)$.

– si e est un mot,

$$w \in e \iff w = e$$

– si $e = e_1 + e_2$,

$$w \in e_1 + e_2 \iff w \in e1 \text{ ou } w \in e_2$$

– si $e = e_1.e_2$, c'est moins simple. On définit un prédicat binaire R sur l'ensemble des mots :

$$R(u, r) \iff \exists s, t \quad | \quad (u = st) \wedge (rs \in e_1) \wedge (t \in e_2)$$

Alors $w \in e_1.e_2$ si et seulement si $\exists s, t \quad | \quad (w = st) \wedge (s \in e_1) \wedge (t \in e_2)$, c'est-à-dire si et seulement si $R(w, \varepsilon)$ est vrai. Pour calculer $R(u, r)$, on utilise une induction sur la longueur de u :

– $u = \varepsilon$:

$$R(\varepsilon, r) \iff (r \in e_1) \wedge (\varepsilon \in e_2)$$

– $u = av$ avec a une lettre de Σ :

$$\begin{aligned} R(av, r) &\iff \exists s, t \quad | \quad (av = st) \wedge (rs \in e_1) \wedge (t \in e_2) \\ &\iff \text{ou} \begin{cases} \{\text{cas } s = \varepsilon\} & (r \in e_1) \wedge (av \in e_2) \\ \{\text{cas } s = as'\} & \exists s', t | (v = s't) \wedge (ras' \in e_1) \wedge (t \in e2) \end{cases} \\ &\iff \text{ou} \begin{cases} (r \in e_1) \wedge (av \in e_2) \\ R(v, ra) \end{cases} \end{aligned}$$

– si $e = (e')^*$. On définit encore un prédicat binaire R sur les mots :

$$R(u, r) \iff \exists s \neq \varepsilon, t \quad | \quad (u = st) \wedge (rs \in e') \wedge (t \in (e')^*)$$

Alors $w \in (e')^*$ si et seulement si $w = \varepsilon$ ou $\exists s \neq \varepsilon, t \quad | \quad (u = st) \wedge (s \in e') \wedge (t \in (e')^*)$, c'est-à-dire si et seulement si $w = \varepsilon$ ou $R(w, \varepsilon)$. On utilise encore une induction sur la longueur de u pour calculer $R(u, r)$:

– $u = \varepsilon$:

$$R(\varepsilon, r) = \texttt{false}$$

– $u = av$ avec $a \in \Sigma$:

$$\begin{aligned} R(av, r) &\iff \exists s \neq \varepsilon, t \quad | \quad (av = st) \wedge (rs \in e') \wedge (t \in (e')^*) \\ &\iff \text{ou} \begin{cases} \{\text{cas } s = a\} & (ra \in e') \wedge (v \in (e')^*) \\ \{\text{cas } s = as', s' \neq \varepsilon\} & \exists s' \neq \varepsilon, t | (v = s't) \wedge (ras' \in e') \wedge (t \in (e')^*) \end{cases} \\ &\iff \text{ou} \begin{cases} (ra \in e') \wedge ((v = \varepsilon) \vee R(v, \varepsilon)) \\ R(v, ra) \end{cases} \end{aligned}$$

Le calcul peut devenir exponentiel. Une autre méthode consiste à calculer un automate déterministe associé à e et à le simuler sur le mot w. Mais la déterminisation peut être dans certains cas exponentielle. L'utilisateur a le choix. Une bonne manière de faire est d'utiliser la fonction `word_in_reg` pour tester une appartenance, et de construire l'automate déterministe si l'on veut tester l'appartenance de plusieurs mots au langages : l'automate est créé une fois pour toute, et les simulations sont linéaires en la taille des mots.

3.2 Inclusion d'expressions rationnelles

Etant données deux expressions rationnelles e_1 et e_2, on peut se demander si $\lambda(e_1) \subseteq \lambda(e_2)$. La fonction `reg_inclusion` permet exécute ce test. Le principe de l'algorithme est de transformer e_2 en automate déterministe, puis de calculer l'ensemble des états atteints par simulation de l'automate sur les mots de $\lambda(e_1)$, puis de tester si cet ensemble d'états est inclus dans l'ensemble des états finaux de l'automate. On note

$$Q(q, e) = \{\delta(q, u) | u \in \lambda(e)\}$$

où q est un état de l'automate et δ sa fonction de transition. On veut donc tester si $Q(q_0, e_1) \subseteq F$ où q_0 est l'état initial et F est l'ensemble des états finaux de l'automate. Pour calculer $Q(q, e)$, on utilise une induction sur la structure de e.

– si w est un mot,
$$Q(q, w) = \delta^*(q, w)$$

–
$$Q(q, e_1 + e_2) = Q(q, e_1) \cup Q(q, e_2)$$

–
$$Q(q, e_1.e_2) = \bigcup_{q' \in Q(q, e_1)} Q(q', e_2)$$

–
$$Q(q, e^*) = \bigcup_{q' \in Q(q, e)} Q(q', e^*)$$

Pour calculer $Q(q, e^*)$ on effectue en fait un parcours en largeur de l'automate réduit à $Q(e, e^*)$, et où l'on ne calcule qu'une fois les $Q(q', e^*)$. On marque en quelque sorte les sommets lorsqu'on les a vu une fois, et on n'y touche plus par la suite. Ainsi le parcours est borné par le nombre d'états de l'automate.

3.3 Résolution de systèmes

Nous expliquons ici la résolution de certains types de systèmes d'équations en langages. L'algorithme a été implanté en particulier pour être utilisé dans la conversion des automates et des grammaires en expressions rationnelles.

Ici, une équation est de la forme

$$X = B + \sum_{i=1}^{n} A_i X_i \text{ (appelée équation orienté à droite)}$$

or

$$X = B + \sum_{i=1}^{n} X_i A_i \text{ (appelée équation orientée à gauche)}$$

où A_i, B sont des expressions rationnelles sur un alphabet Σ, et X, X_i des variables, c'est à dire des lettres qui ne sont pas dans Σ.

Un system d'équations est un ensemble fini de telles équations ayant la même orientation, et où chaque variable apparaît au plus une fois dans un membre gauche.

Nous montrons ici le fonctionnement de l'algorithme sur des équations orientées à droite. Pour des équations orientées à gauche, c'est symétrique.

3.3.1 Résolution d'une équation

Si l'on a une équation

$$X = AX + B$$

alors une solution minimale (un langage L tel que $L = AL + B$ et qui est contenu dans toute autre solution) est A^*B. En effet, c'est une solution, et si L est une solution, alors

$$\begin{aligned} L &= AL + B \\ &= A(AL + B) + B \\ &= A^2L + AB + B \\ &\vdots \\ &= A^{n+1}L + A^nB + A^{n-1}B + \cdots + B \quad \forall n \geqslant 0 \end{aligned}$$

Donc pour tout $n \geqslant 0$, $A^nB \subseteq L$ et donc $A^*B \subseteq L$. A^*B est donc une solution minimale (c'est en fait LA solution minimale).

Si $\varepsilon \notin A$, l'ensemble des solutions est réduit à $\{A^*B\}$.

3.3.2 Résolution d'un système

Supposons que l'on veuille résoudre le système

$$\begin{cases} X_1 &= A_{1,1}X_1 + A_{1,2}X_2 + \ldots + A_{1,n}X_n + B_1 \\ X_2 &= A_{2,1}X_1 + A_{2,2}X_2 + \ldots + A_{2,n}X_n + B_2 \\ & \vdots \\ X_n &= A_{n,1}X_1 + A_{n,2}X_2 + \ldots + A_{n,n}X_n + B_n \end{cases}$$

Une solution est un $n-$uplet $L_1, \ldots, L_n$ tel que pour tout i,

$$L_i = A_{i,1}L_1 + A_{i,2}L_2 + \ldots + A_{i,n}L_n + B_i$$

L'algorithme est récursif :

- if $n = 1$, on a une équation à résoudre, on utilise la méthode décrite ci-dessus.
- if $n > 1$, on remplace X_n par

  $$A_{nn}^* \left(A_{n,1}X_1 + \ldots + A_{n,n-1}X_{n-1} + B_n\right)$$

  dans les $(n-1)$ premières équations, on les résout et on remplace les valeurs de $X_1, \ldots, X_{n-1}$ dans l' expression ci-dessus pour trouver la valeur X_n.

Ici encore, si ε n'est dans aucun A_{ij}, le système admet une unique solution. Cette unicité est importante pour les conversions d'automates et de grammaires en expressions rationnelles.

3.3.3 Types internes

Les variables sont simplement des lettres (type `letter_t`) qui ne sont pas dans l'alphabet utilisé pour construire les expressions $A_{i,j}$ et B_i. Alors les variables peuvent apparaître dans une expression rationnelle, comme si elles étaient des lettres quelconques.

La fonction `syst_solve` reçoit en argument une liste de `(letter_t * regexp_t)`. Chaque élément (x, e) de cette liste correspond à une équation $x = e$. L'algorithme infère l'alphabet des variables : c'est l'ensemble des lettres x telles que (x, e) est dans la liste pour un certain e. `syst_solve` refuse les arguments dans lesquels une variable est associée à plusieurs expressions rationnelles : quand (x, e) et (x, e') apparaissent dans la liste.

Alors les expressions rationnelles sont converties en un type `monome_sum_t`. Une exception est levée s'il y a des concaténations de variables (par exemple, dans $AXBYC$ où X et Y sont des variables), des étoiles au-dessus de variables ($(AX)^*$), des unions d'expressions rationnelles qui n'ont pas la même orientation ($AX + YB$).

Pour connaître l' orientation d'un `monome_sum_t`, ce type est un enregistrement avec un champ `side` et un liste de `monome_t`. Le champ `side` est un est trois constructeurs `Right`, `Left`, `Noside`. Les expressions A, X où A ne contient aucune variable ne sont pas orientées, c'est pourquoi `Noside` est nécessaire.

Alors la résolution est lancée. Elle consiste en deux passes. Durant la première passe, la première équation (x, e) est résolue (on calcule la valeur de x en fonction des autres variables) puis les occurrences de x sont remplacées par e dans les équations suivantes, puis le processus est répété à partir de la deuxième équation. On obtient alors un système triangulaire où les variables du membre droit d'une équation ne sont les membres gauches que des équations suivantes. Ceci implique que la dernière équation est résolue. Alors la seconde passe est lancée : on remplace les occurrences de la dernière variable par sa valeur dans les premières équations, puis on recommence à partir de l'avant-dernière équation.

3.4 Conversion des automates en expressions rationnelles

L'algorithme de résolution de système est utilisé pour calculer une expression rationnelle dénotant le langage reconnu par un automate déterministe. Soit donc un automate $M = (Q, \Sigma, \delta, q_0, F)$. On crée d'abord un système. Les expressions rationnelles sont construites sur Σ, une variable X_q est associée à chaque état $q \in Q$. Les équations sont

$$\begin{cases} X_q = \sum_{a\in\Sigma} aX_{\delta(q,a)} & \text{if } q \notin F \\ X_q = \sum_{a\in\Sigma} aX_{\delta(q,a)} + \varepsilon & \text{si } q \in F \end{cases}$$

X_q dénote le langage

$$\{u \in \Sigma^* | \delta(q, u) \in F\}$$

Alors, le langage reconnu par l'automate est X_{q_0}. Comme les facteurs associés au variables (les $A_{i,j}$) sont des lettres de Σ, ε n'est pas dans les langages associés, donc la solution est unique et c'est le langage reconnu par l'automate.

Le principe est le même pour la conversion des grammaires en expressions rationnelles.

Les conversions entre automates et expressions rationnelles sont utiles pour résoudre certains problèmes :

- équivalence d'expressions rationnelles : les 2 expressions sont converties en automates, qui sont déterminisés et minimisés. Alors on teste si les 2 automates sont isomorphes.
- complémentaire : pour trouver une expression rationnelle dénotant le complémentaire du langage dénoté par une expression donnée, on convertit cette dernière en automate déterministe complet, on change F en $Q \setminus F$ et on convertit ce nouvel automate en expression rationnelle.
- appartenance d'un mot à un langage.

4 Grammaires

4.1 Description du module

Ce sous-projet fournit la partie relative aux grammaires dans MIMot. Une grammaire sert de générateur de langage et a donc toute sa place dans un logiciel de manipulation de langages formels. Il y a différents types de grammaires, plus ou moins puissants. MIMot est capable de manipuler tout type de grammaire, mais la plupart des fonctions ne sont implantées que pour certains types, car les problèmes qu'elles résolvent sont indécidables pour les types plus généraux.

4.2 Différents types de grammaires

Les grammaires sont des ensembles de règles qui permettent d'engendrer des langages. Ce système est beaucoup plus général que les automates, par exemple, puisqu'il permet d'engendrer n'importe quel langage récursivement énumérable. Cependant, on limite leur puissance en distinguant plusieurs types de grammaires correspondant à plusieurs classes de langages. Dans cette partie, nous présentons ces différents types, ainsi que les fonctions de MIMot qui permettent de les reconnaître.

4.2.1 Grammaire RE

Les grammaires récursivement énumérables (recursively enumerable, RE) constituent le type le plus général : les langages reconnus par les grammaires RE sont les langages récursivement énumérables.

DÉFINITION 4.1 (GRAMMAIRE RE)
Une grammaire RE est un quadruplet $G = (N, T, S, P)$ où N et T sont deux alphabets disjoints, $S \in N$ et $P \subseteq V^*NV^* \times V^*$ avec $V = N \cup T$. N est appelé alphabet des variables (ou non-terminaux), et T alphabet des terminaux. S est appelé axiome et P est l'ensemble des productions de G.

On écrit $u \longrightarrow v$ si $(u, v) \in P$. Lorsque $x = x_1 u x_2$ et $y = x_1 v x_2$ avec $x_1, x_2, u, v \in V^*$, on note $x \vdash y$ si $u \longrightarrow v$, et $\vdash^*$ désigne la fermeture transitive et réflexive de $\vdash$: $u \vdash^* v \Longleftrightarrow u = v$ ou $u \vdash u_1 \vdash \cdots \vdash u_n \vdash v$.

Si G est une grammaire RE on définit le langage qu'elle reconnaît comme suit :

DÉFINITION 4.2 ($L(G)$)
Soit G une grammaire RE. $L(G) = \{v \in T^* | S \vdash^* v\}$ est le langage reconnu par G.

Deux grammaires G_1 et G_2 sont dites équivalentes si $L(G_1) = L(G_2)$.

On définit des grammaires plus restrictives que les grammaires RE en imposant des conditions sur les productions.

4.2.2 Grammaires CS

Les grammaires sensibles au contexte (context-sensitive, CS) sont définies comme suit :

Définition 4.3 (Grammaire CS)
$G = (N, T, S, P)$ est une grammaire CS si c'est une grammaire RE vérifiant (toujours en notant $V = N \cup T$)

$u \longrightarrow v \Longrightarrow \exists u_1, u_2 \in V^*, A \in N$ et $x \in V^+$ tel que $u = u_1 A u_2$ et $v = u_1 x u_2$.

Ainsi, les productions ne peuvent être appliquées que si le mot a une certaine forme $u_1 A u_2$.

Définition 4.4 (Grammaire monotone)
Une grammaire est monotone si $u \longrightarrow v \Longrightarrow |u| \leqslant |v|$.

Remarque Pour les grammaires CS et monotones, on autorise une production $S \longrightarrow \varepsilon$ si S n'apparaît dans le membre droit d'aucune production.

Proposition 4.1 *La classe des langages reconnus par grammaires CS est égale à celle des langages reconnus par grammaires monotones (i.e. une grammaire monotone est équivalente à une grammaire CS, et réciproquement une grammaire CS est équivalente à une grammaire monotone).*

4.2.3 Grammaires CF

Une grammaire est algébrique (context-free, CF) lorsque le membre gauche des productions est seulement une variable :

Définition 4.5 (Grammaire CF)
Une grammaire CF est une grammaire RE pour laquelle $u \longrightarrow v \Longrightarrow u \in N$.

4.2.4 Grammaires LIN

Définition 4.6 (Grammaire LIN)
Une grammaire linéaire (LIN) est une grammaire RE telle que $u \longrightarrow v \Longrightarrow$ $[u \in N$ et $v \in T^* \cup T^* N T^*]$.

4.2.5 Grammaires REG

Les grammaires rationnelles (regular, REG) reconnaissent les langages rationnels.

DÉFINITION 4.7 (GRAMMAIRE REG)
Une grammaire REG est une grammaire RE telle que $u \longrightarrow v \Longrightarrow [u \in N$ et $v \in T \cup TN \cup \{\varepsilon\}]$.

Une autre façon de caractériser les langages rationnels sont les grammaires linéaires à gauche (LLIN) et à droite (RLIN).

DÉFINITION 4.8 (GRAMMAIRE LLIN)
Une grammaire LLIN est une grammaire RE telle que $u \longrightarrow v \Longrightarrow [u \in N$ et $v \in T^* \cup NT^*]$.

DÉFINITION 4.9 (GRAMMAIRE RLIN)
Une grammaire RLIN est une grammaire RE telle que $u \longrightarrow v \Longrightarrow [u \in N$ et $v \in T^* \cup T^*N]$.

Proposition 4.2 *Les classes de langages reconnus par grammaires REG, LLIN et RLIN sont les mêmes (i.e. une grammaire REG est équivalente à une grammaire LLIN et à une grammaire RLIN, et de même pour tous les autres sens).*

On note RE, CS, CF, LIN et REG les classes des langages reconnus par grammaires RE, CS, CF, LIN et REG respectivement.

4.2.6 Propriétés

Il suit des définitions que REG $\subset$ LIN $\subset$ CF $\subset$ CS $\subset$ RE. Les inclusions sont strictes.

Bien entendu, on sait faire plus ou moins de chose sur les grammaires selon leur type, comme on peut le voir dans le tableau suivant : on se pose 6 questions sur deux grammaires G_1 et G_2 et un mot x (sur un alphabet quelconque).

- Équivalence : G_1 et G_2 sont-elles équivalentes ?
- Inclusion : $L(G_1) \subseteq L(G_2)$?
- Appartenance : $x \in L(G_1)$?
- Vide : $L(G_1) = \emptyset$?
- Finitude : $L(G_1)$ est fini ?

– Rationnel : $L(G_1)$ est un langage rationnel ?

La décidabilité de ces 6 problèmes dépend de la classe des grammaires que l'on considère. Pour la classe CS par exemple, on se donne G_1 et G_2 deux grammaires CS, et on veut savoir parmi les 6 questions lesquelles sont décidables. Le tableau ci-dessous résume les résultats : D signifie décidable, alors que N signifie non-décidable. MIMot est capable de résoudre toutes les questions décidable indiquées.

	RE	CS	CF	LIN	REG
Équivalence	N	N	N	N	D
Inclusion	N	N	N	N	D
Appartenance	N	D	D	D	D
Vide	N	N	D	D	D
Finitude	N	N	D	D	D
Rationnel	N	N	N	N	trivial

Ainsi, nous voyons que certaines fonctions ne devront être appeler que sur certains types de grammaires. Or en OCaml, on ne peut pas créer de type (informatique) `grammar_t` dont les éléments dépendent d'une propriété d'autres éléments du type ; par exemple, l'axiome de la grammaire doit être une lettre de l'alphabet des variables, ce qu'il n'est pas possible de tester directement à la création du type `grammar_t`. C'est pourquoi il nous faut un type `grammar_t` communs à toutes les grammaires, et cela nécessite des fonctions de reconnaissance du type d'une grammaire.

Nous avons décider d'utiliser les enregistrements pour le type `grammar_t`, d'où la définition

```
type grammar_t =
  {n : alphabet_t; t : alphabet_t; s : letter_t; p : prod_list_t}
```

puis nous avons implémenté les fonctions suivantes qui testent l'appartenance d'une grammaire à une certaine classe :

```
val is_a_grammar_re : grammar_t -> bool
val is_a_grammar_cs : grammar_t -> bool
val is_a_grammar_monotonous : grammar_t -> bool
val is_a_grammar_cf : grammar_t -> bool
val is_a_grammar_lin : grammar_t -> bool
val is_a_grammar_reg : grammar_t -> bool
val is_a_grammar_llin : grammar_t -> bool
val is_a_grammar_rlin : grammar_t -> bool
```

L'implémentation de ces fonctions suit la définition des grammaires ; cependant il n'est pas toujours évident de tester certaines propriétés, notamment celles des grammaires CS. Nous avons eu besoin de fonctions intermédiaires, par exemple pour voir si une production est bien de la forme $uAv \longrightarrow uwv$ avec $A \in N$ et u, v, w trois mots où $|w| > 0$. Toutes les fonctions précédentes ont une complexité linéaire en la taille de la grammaire (nombre de symboles utilisé pour la coder).

Avant d'appliquer une fonction à une grammaire, il s'agit d'avoir d'abord testé son type grâce aux fonctions précédentes. Par la suite, certaines fonctions ne retourneront pas le bon résultat (voire un message d'erreur) si on leur fournit une grammaire quelconque, mais nous avons décider de ne pas restreindre l'usage des fonctions, en ne testant pas systématiquement le type des grammaires au début. Il sera simple, si on le décide, d'appeler ces fonctions uniquement si les arguments sont de bon type, mais pour l'instant, il faut s'assurer par soi-même du type de grammaire que l'on fournit. Cela pour ne pas tester plusieurs fois le type d'une grammaire (lorsque l'on appelle deux fonctions sur la même grammaire CF, par exemple, on ne souhaite pas vérifier deux fois son type), pour ne pas alourdir les fonctions et surtout pour laisser plus de liberté (au risque d'avoir une erreur) à l'utilisateur. Nous pourrons également stocker le type d'une grammaire pour n'avoir à le tester qu'une fois.

4.3 Manipulation de grammaires

Voyons maintenant quelques fonctions permettant de manipuler les grammaires.

4.3.1 De l'interface à la grammaire

Nous avons vu qu'une grammaire est stockée avec ses deux alphabet et la liste des productions. Pour qu'écrire une grammaire ne soit pas insupportable pour l'utilisateur, les alphabets sont facultatifs, ainsi que l'axiome lorsqu'il n'y a pas ambigüité, *i.e.* dans le cas des grammaires CF. En effet, on peut alors connaître toutes les variables (membre gauche des productions), puis en déduire les terminaux (si toutefois toutes les variables apparaissant à droite sont utiles). Si l'axiome n'est pas donné, on suppose que c'est le membre gauche de la première production.

Ainsi, le module Grammars contient la fonction

```
val infere_grammar :
  alphabet_t option * alphabet_t option * letter_t option
  * prod_list_t -> grammar_t
```

4.3.2 Union, concaténation, étoile

L'union de deux grammaires $G = (N, T, S, P)$ et $G' = (N', T', S', P')$ est très simple : l'union $G'' = (N'', T'', S'', P'')$ de G et G' s'obtient en gardant les productions de G et G' et en ajoutant $S'' \longrightarrow S|S'$. De même pour la concaténation, on ajoute $S'' \longrightarrow SS'$. Cependant, on oublie quelques détails techniques : il faut d'abord renommer les variables de G' pour que $N \cap N' = \emptyset$. Cette opération anodine nécessite en fait de modifier chaque production de P' pour qu'elle utilise les nouvelles variables. Pour l'étoile de Kleene, en revanche, nous n'avons pas besoin de renommage puisqu'on travaille sur une seule grammaire. Il suffit d'ajouter les règles $S \longrightarrow \varepsilon$ et $S \longrightarrow SS$. Nous avons finalement trois fonctions :

```
val grammar_union : grammar_t -> grammar_t -> grammar_t
val grammar_concat : grammar_t -> grammar_t -> grammar_t
val grammar_star : grammar_t -> grammar_t
```

Cependant, un autre problème apparaît. Lorsque nous avons des grammaires rationnelles (REG), nous aimerions que l'union et la concaténation restent dans REG, alors que les grammaires que nous avons obtenues précédemment ne le sont plus (elles sont seulement CF). Il faut alors compliquer un peu le principe. Considérons les alphabets déjà disjoints et construisons une grammaire REG pour l'union, la concaténation et l'étoile.

Pour l'union, nous avons : $N'' = N \cup N' \cup \{S''\}$ où S'' est une nouvelle variable et est l'axiome de G'', $T'' = T \cup T'$, $P'' = P \cup P' \cup X$ avec $X = \{S'' \longrightarrow x | S \longrightarrow x \in P \text{ ou } S' \longrightarrow x \in P'\}$. L'idée est qu'avec S'' on peut choisir quelle grammaire on va utiliser.

Pour la concaténation, c'est un peu plus difficile : $N'' = N \cup N'$, $T'' = T \cup T'$, $S'' = S$ et $P'' = \tilde{P} \cup P'$ avec $\tilde{P} = \{u \longrightarrow v | u \longrightarrow \tilde{v} \in P, \tilde{v} \in T \text{ et } v = \tilde{v}S'\} \cup \{u \longrightarrow v | u \longrightarrow \varepsilon \in P \text{ et } S' \longrightarrow v \in P'\} \cup \{u \longrightarrow v | u \longrightarrow v \in P, v \notin T \cup \{\varepsilon\}\}$. Les règles terminales de P sont complétées par S' pour commencer un nouveau mot de G'.

Puis pour l'étoile de Kleene, c'est un peu le même principe que pour la concaténation, mais pour une seule grammaire : $N'' = N$, $T'' = T$, $S'' = S$ et $P'' = P \cup \tilde{P} \cup \{S \longrightarrow \varepsilon\}$ avec $\tilde{P} = \{u \longrightarrow v | u \longrightarrow \tilde{v} \in P, \tilde{v} \in T \text{ et } v = \tilde{v}S\} \cup \{u \longrightarrow v | u \longrightarrow \varepsilon \in P \text{ et } S \longrightarrow v \in P\}$.

On obtient ainsi trois fonctions pour les grammaires rationnelles :

```
val reg_gram_union : grammar_t -> grammar_t -> grammar_t
val reg_gram_concat : grammar_t -> grammar_t -> grammar_t
val reg_gram_star : grammar_t -> grammar_t
```

La complexité de toutes les fonctions précédentes est linéaire ; souvent ce qui prend le plus de temps est de renommer les variables.

4.3.3 Dérivations

Une grammaire est avant tout un générateur de langages : en partant de l'axiome on suit des productions qui nous mènent à certains mots sur les terminaux. Il se devait donc de faire des fonctions permettant de dériver des mots suivant les productions. Là encore, le principe est simple, l'implantation plus délicate. Nous avons pour cela créé plusieurs fonctions de dérivations :

```
val derivation : word_t -> production_t -> finite_t
val leftmost_derivation : word_t -> production_t -> word_t
val language_n : int -> grammar_t -> finite_t
val language_n_from_word : int -> word_t -> grammar_t -> finite_t
val language_word_size : int -> grammar_t -> finite_t
```

La première fonction `derivation` permet de dériver un mot suivant une production, elle retourne l'ensemble des mots que l'on peut atteindre. Par exemple, si la production est $aAba \longrightarrow ac$ (dans une grammaire RE), et le mot est $aaAbaAbab$, alors le résultat est l'ensemble $\{aacAbab, aaAbacb\}$. Cette fonction utilise la fonction du module Words de recherche d'un motif dans un mot :

```
val find_matchings : word_t -> word_t -> int list
```

Cette fonction retourne la liste des positions du motif dans le mot. Il suffit alors de remplacer l'occurence de $aAba$ dans $aaAbaAbab$ par ac.

La deuxième fonction `leftmost_derivation` dérive un mot selon une production en suivant la règle "le plus à gauche", et retourne le mot produit. La troisième fonction `language_n k G` retourne le langage reconnu par `G` en moins de `k` dérivation à partir de l'axiome. La dernière fonction fait de même mais en partant d'un mot spécifié.

À partir de ces fonctions, on peut tester l'appartenance d'un mot dans une grammaire CS : sachant qu'une telle grammaire est monotone, *i.e.* que les mots dérivés sont de taille plus grande que le mot de départ (ce qui n'est plus le cas pour une grammaire RE), il suffit de regarder l'ensemble des mots de $L(G)$ qui sont de taille inférieure ou égale au mot à chercher. Pour cela, on itère `derivation` à partir de l'axiome, et on enlève les mots dont la taille est trop grande. On obtient la fonction

```
val is_a_word_in_cs : word_t -> grammar_t -> bool
```

Attention, cette fonction, ainsi que les deux précédentes et la suivante, est exponentielle en la taille du mot, car on peut avoir à tester un nombre exponentiel de mots (les dérivations peuvent mener à plus d'un mot et engendrer

une croissance exponentielle du nombre de mots à vérifier). Pour les grammaires CF, nous verrons qu'il y a un algorithme polynômial pour tester l'appartenance d'un mot.

On fait de même pour la fonction `language_word_size k G` qui retourne l'ensemble des mots de $L(G)$ de moins de `k` lettres.

4.3.4 Simplification et formes normales

Lorsque nous avons une grammaire, il peut être avantageux de la "simplifier" pour l'avoir sous une forme canonique "propre".

DÉFINITION 4.10 (FORME PROPRE)
Une grammaire G est "propre" si

i. $\forall u \longrightarrow v \in P, v = \varepsilon \Longrightarrow u = S$ ($S \longrightarrow \varepsilon$ est la seule ε-transition) ;

ii. $P \cap (N \times N) = \emptyset$ (pas de production de la forme $X \longrightarrow Y$ avec X et Y deux variables) ;

iii. $\forall A \in N$, $\exists u, v \in (N \cup T)^*$ tel que $S \vdash^* uXv$ (toutes les variables sont atteintes) ;

iv. $\forall A \in N$, $L_A(G) = \{v \in T^* | A \vdash^* v\} \neq \emptyset$ (toutes les variables sont utiles).

Proposition 4.3 *Toute grammaire CF est équivalente à une grammaire propre effectivement constructible.*

Dans MIMot, la fonction qui transforme une grammaire CF en grammaire propre a été appelée

```
val simplify_cf : grammar_t -> grammar_t
```

L'algorithme n'est pas simple, il fait intervenir plusieurs fonctions intermédiaires. Par exemple, celle de savoir si le langage est vide (qui intervient notamment dans le (**??**)), qui a donné lieu à la fonction

```
val is_cf_empty : grammar_t -> bool
```

Nous présentons ici cet algorithme car le même principe est utilisé plusieurs fois (par exemple pour savoir si le langage contient ε).

Soit G une grammaire CF, $G = (N, T, S, P)$. Comme précédemment, pour $A \in N$ on note L_A le langage définit par $L_A = \{v \in T^* | A \vdash^* v\}$. Nous cherchons à savoir si $L(G) = L_S$ est vide. Notons que $L(G) \neq \emptyset \Longrightarrow \exists u \longrightarrow v \in P$ tel que $v \in T^*$, sinon on aurait toujours au moins une variable dans nos dérivations. La première étape (simple) de l'algorithme est de trouver

toutes les variables $A \in N$ telles que $\exists v \in T^*$ avec $A \longrightarrow v$. S'il n'y en a pas, alors $L(G) = \emptyset$. Sinon, on appelle L_0 l'ensemble de toutes ces variables.

Puis nous dérivons à l'envers : il s'agit de trouver toutes les variables $B \in N$ telles que $\exists v \in (L_0 \cup T)^*$ et $B \longrightarrow v \in P$. On appelle L'_1 l'ensemble de tous ces B, et on définit $L_1 = L_0 \cup L'_1$. Et ainsi de suite : de L_n on construit L_{n+1}. Dès que $L_{n_0+1} = L_{n_0}$, alors $\forall n \geqslant n_0, L_n = L_{n_0}$ et L_{n_0} contient exactement les variables A telles que $L_A \neq \emptyset$. Il suffit alors de vérifier si $S \in L_{n_0}$: en effet, $S \in L_{n_0} \Longleftrightarrow L(G) \neq \emptyset$.

Voici donc l'algorithme implémentant ce que nous venons de voir :

```
Is_empty(G) =
  L <- [] ; c <- 0 ;
  Pour tout (u,v) dans P faire
    Si v ne contient que des terminaux alors
      L <- [L,u] ;
  Si |L|=0 alors
    Retourner True
  Tant que (|L| <> c) faire
    c <- |L| ;
    Pour tout (u,v) dans P faire
      b <- True ;
      Si u n'est pas dans L alors
        Pour toute variable B de v faire
          b <- b & (B in L) ;
        Si b alors
          L <- [L,u] ;
  Retourner (S in L).
```

La complexité de cet algorithme est $\mathcal{O}(|N||P|n) = \mathcal{O}(n^3)$, où n est la taille de G (nombre de symboles nécessaires pour la coder).

Une autre étape de l'algorithme de simplification consiste à déterminer les variables atteintes en partant de S. Pour cela, il suffit de mettre à jour une liste de variables atteintes (initialisée à $[S]$) lorsque l'on suit les productions en partant de S.

Pour éliminer les productions $A \longrightarrow B \in (N \times N)$, il "suffit" de remplacer B par l'ensemble de ses successeurs. Comme d'habitude, l'implantation est bien plus délicate que le principe de l'algorithme. L'implémentation complète de l'algorithme de simplification requiert 6 étapes :

```
Simplify_CF(G) =
  Si Is_empty(G) alors
    Retourner G'=([S],[],S,[]) ;
```

```
1. G' <- Reduce_alphabet(G) ;
2. G' <- No_simple_prod(G') ;
3. G' <- No_epsilon(G') ;
4. G' <- Non_redundant(G') ;
5. G' <- Reached(G') ;
6. G' <- Terminates(G') ;
Retourner G'.
```

Rendre une grammaire propre simplifie la grammaire, et constitue une étape vers la construction de la forme normale de Chomsky d'une grammaire. La complexité de la simplification est $\mathcal{O}(n^3)$ où n est la taille de la grammaire (nombre de symboles qui sont utilisés).

DÉFINITION 4.11 (FORME NORMALE DE CHOMSKY)
Une grammaire $G = (N, T, S, P)$ est sous forme normale de Chomsky si

$$P \subseteq (N \times T) \cup (N \times N^2) \cup \{S \longrightarrow \varepsilon\}$$

c'est-à-dire que les productions sont de la forme $S \longrightarrow \varepsilon$, $A \longrightarrow a$ ou $A \longrightarrow BC$ avec $a \in T$, et $A, B, C \in N$.

Proposition 4.4 *Toute grammaire CF est équivalente à une grammaire sous forme normale de Chomsky, effectivement constructible.*

À partir de la forme propre d'une grammaire CF, il est assez facile de construire une grammaire équivalente sous forme normale de Chomsky. En effet, on garde les productions convenables $S \longrightarrow \varepsilon$, $A \longrightarrow a$ et $A \longrightarrow BC$; les autres productions sont de la forme $A \longrightarrow u$ avec $|u| \geqslant 2$ puisque la grammaire de départ est propre. Si $|u| = 2$:

- si $u = ab \in T^2$ alors on crée trois nouvelles productions $X \longrightarrow a$, $Y \longrightarrow b$ et $A \longrightarrow XY$, où X et Y sont de nouvelles variables ;
- si $u = Ba \in NT$ alors on crée deux nouvelles productions $X \longrightarrow a$ et $A \longrightarrow BX$;
- si $u = aB \in TN$ alors on crée deux nouvelles productions $X \longrightarrow a$ et $A \longrightarrow XB$.

Si $|u| > 2$, on note $u = u_1 u_2 \dots u_n$ avec $n > 2$:

- si $u_1 \in N$, alors on crée une nouvelle variable X et deux nouvelles productions $A \longrightarrow u_1 X$ et $X \longrightarrow u_2 \dots u_n$, puis on recommence récursivement à "chomskyfier" cette dernière production, dont le membre droit a pour taille $n - 1$;

– si $u_1 \in T$, alors on crée deux nouvelles variables X et Y, et trois nouvelles productions $A \longrightarrow XY$, $X \longrightarrow u_1$ et $Y \longrightarrow u_2 \dots u_n$, puis on recommence récursivement à "chomskyfier" cette dernière production, dont le membre droit a pour taille $n-1$.

Cet algorithme a fournit dans MIMot la fonction

```
val chomsky : grammar_t -> grammar_t
```

La complexité est linéaire en le nombre de symboles de la grammaire.

Le grand intérêt de la forme normale de Chomsky est de permettre de résoudre l'appartenance d'un mot à une grammaire en temps polynômial. Pour cela, on utilise l'algorithme CYK, implémenté par la fonction suivante (CNF signifie Chomsky's Normal Form) :

```
val is_word_in_CNF : word_t -> grammar_t -> bool
```

L'algorithme CYK[1] permet de vérifier l'appartenance d'un mot à une grammaire algébrique mise sous forme normale de Chomsky. Sa complexité est en $\mathcal{O}(n^3)$ où n est la longueur du mot en entrée, et la constante intervenant dans le $\mathcal{O}$ est elle-même une fonction polynômiale du nombre de lettres et du nombre de productions de la grammaire considérée. Son principe repose sur la notion de programmation dynamique : on détermine pour chaque sous-mot u du mot d'entrée (en s'attaquant d'abord aux sous-mots de longueur un, puis de longueur deux...) quelles sont les variables de la grammaire qui mènent à u par une série de dérivations. On conclue ensuite en vérifiant si la variable axiome apparaît dans l'ensemble des variables qui permettent d'engendrer le mot d'entrée.

4.3.5 Finitude du langage engendré par une grammaire CF

Maintenant que nous disposons d'une forme propre d'une grammaire CF, nous pouvons facilement savoir si le langage engendré par une grammaire CF est fini : il suffit de repérer les éventuels "cycles" dans les productions. Par cycle, nous entendons qu'il existe deux mots u et v de $(T \cup N)^*$ non tous les deux vides, et une variable $A \in N$ tels que $A \vdash^* uAv$. Le langage est infini si et seulement si la grammaire CF sous forme propre ne contient pas de cycle. En effet, toutes les variables sont utiles puisque la grammaire est propre ; ainsi, si nous avons un cycle, le langage engendré par A est infini (puisque l'on peut engendrer des mots de longueur non bornée). Réciproquement, s'il

1. du nom de ses auteurs Cocke, Younger et Kasami

n'y a pas de cycles, alors le nombre de mots est borné et le langage est fini. Ce raisonnement donne lieu à la fonction suivante dans MIMot :

```
val is_cf_finite : grammar_t -> bool
```

Cette fonction teste pour chaque variable A s'il existe un cycle $A \vdash^* uAv$.

4.4 Conversions

4.4.1 Expressions rationnelles

Dans MIMot, on aimerait pouvoir convertir des grammaires rationnelles en expressions rationnelles, et inversement, car elles reconnaissent les mêmes langages. Nous avons maintenant tous les outils à notre disposition pour le faire simplement.

Lorsque nous disposons d'une expression rationnelle r, nous aimerions créer une grammaire reconnaissant le même langage. Or nous avons des fonctions `gram_reg_union`, `gram_reg_concat` et `gram_reg_star` que nous avons vues à la section **??** : la fonction

```
val regexp_to_grammar : regexp_t -> grammar_t
```

fonctionne récursivement sur la structure de l'expression rationnelle r.

- Si $r = \varepsilon$ alors `regexp_to_grammar`$(r) = (\{S\}, \emptyset, S, \{S \longrightarrow \varepsilon\})$.
- Si $r = a$ où a est une lettre, alors `regexp_to_grammar`$(r) = (\{S\}, \{a\}, S, \{S \longrightarrow a\})$.
- Si $r = \alpha + \beta$ alors
  `regexp_to_grammar`$(r) =$
  `reg_gram_union`(`regexp_to_grammar`(α), `regexp_to_grammar`(β)).
- Si $r = \alpha\beta$ alors
  `regexp_to_grammar`$(r) =$
  `reg_gram_concat`(`regexp_to_grammar`(α), `regexp_to_grammar`(β)).
- Si $r = \alpha^*$ alors `regexp_to_grammar`$(r) =$ `reg_gram_star`(`regexp_to_grammar`(α)).

Voyons maintenant comment convertir une grammaire REG en une expression rationnelle. La fonction de MIMot s'appelle

```
val grammar_to_regexp : grammar_t -> regexp_t
```

Le principe en est simple : soit $G = (N, T, S, P)$ une grammaire rationnelle. Pour tout $A \in N$, on appelle $L_A = \{u \in T^* | A \vdash^* u\}$, donc $L_S = L(G)$ est le langage que l'on cherche à exprimer par une expression rationnelle. Nous obtenons un système d'équations "linéaires à gauche"

$$\{L_A = (M_A)L_A + N_A\}_{A \in N} \text{ où } M_A = \sum_{A \longrightarrow aA} a \text{ et } N_A = \sum_{A \longrightarrow b} b + \sum_{A \longrightarrow bB} bL_B$$

Nous utilisons alors la fonction `syst_solve` du module Reg pour résoudre le système. Il suffit alors de renvoyer l'expression rationnelle correspondant à L_S. Notre fonction permet également de transformer une grammaire RLIN ou LLIN en expression rationnelle, car la fonction `syst_solve` résout aussi les équations correspondantes.

4.4.2 Conversions dans la classe REG

Nous avons vu que les classes REG, LLIN et RLIN sont les mêmes. C'est pourquoi nous voulons des fonctions de conversions entre ces différents types de grammaires. Le seul intérêt est de transformer une grammaire LLIN ou RLIN en grammaire REG, puisque les grammaires REG sont les plus utilisées et les plus simples. MIMot fournit ainsi les fonctions suivantes :

```
val rlin_to_reg : grammar_t -> grammar_t
val llin_to_reg : grammar_t -> grammar_t
```

Pour cela, le moyen le plus simple est de passer par les expressions rationnelles : en effet, il suffit de déclarer

```
let rlin_to_reg g = regexp_to_grammar (grammar_to_regexp g)
let llin_to_reg g = regexp_to_grammar (grammar_to_regexp g)
```

4.4.3 Conversion dans la classe CS

Une grammaire CS est toujours monotone, mais la réciproque est fausse. C'est pourquoi nous avons besoin d'une fonction dans MIMot effectuant la conversion :

```
val monotonous_to_cs : grammar_t -> grammar_t
```

Le principe est le suivant : soit $G = (N, T, S, P)$ une grammaire monotone, construisons une grammaire CS équivalente.

- Pour tout terminal $t \in T$, on ajoute une nouvelle variable t', et la production CS $t' \longrightarrow t$. Toutes les productions sont remplacées pour ne travailler que sur les variables, donc les seules productions contenant des terminaux sont maintenant de la forme $t' \longrightarrow t$.

– Si nous avons une production de la forme $X \longrightarrow Y_1 \dots Y_n$, elle est déjà CS et on la garde.

– Sinon, la production est de la forme $X_1 X_2 \dots X_m \longrightarrow Y_1 Y_2 \dots Y_n$ avec $2 \leqslant m \leqslant n$. On la remplace alors par l'ensemble des productions CS suivantes, où les Z_i, pour $1 \leqslant i \leqslant m$ sont de nouvelles variables :

$$
\begin{array}{c}
X_1 X_2 \dots X_m \longrightarrow Z_1 X_2 \dots X_m \\
Z_1 X_2 \dots X_m \longrightarrow Z_1 Z_2 X_3 \dots X_m \\
\vdots \\
Z_1 Z_2 \dots Z_{m-2} X_{m-1} X_m \longrightarrow Z_1 Z_2 \dots Z_{m-1} X_m \\
Z_1 Z_2 \dots Z_{m-1} X_m \longrightarrow Z_1 \dots Z_m Y_{m+1} \dots Y_n \\
Z_1 \dots Z_m Y_{m+1} \dots Y_n \longrightarrow Y_1 Z_2 \dots Z_m Y_{m+1} \dots Y_n \\
Y_1 Z_2 \dots Z_m Y_{m+1} \dots Y_n \longrightarrow Y_1 Y_2 Z_3 \dots Z_m Y_{m+1} \dots Y_n \\
\vdots \\
Y_1 \dots Y_{m-1} Z_m Y_{m+1} \dots Y_n \longrightarrow Y_1 \dots Y_n
\end{array}
$$

4.5 Récapitulatif des commandes

Dans ce qui précède, le nom des fonctions correspondait à celui dans le code source. Dans le tableau qui suit en revanche, c'est le nom des fonctions dans l'interface de MIMot qui est mentionné. Un bref commentaire explique la fonction, et la dernière colonne indique le type de grammaire sur lequel la fonction agit correctement.

`is_grammar_re`	teste si une grammaire est RE	RE
`is_grammar_cs`	teste si une grammaire est CS	RE
`is_grammar_monotonous`	teste si une grammaire est monotone	RE
`is_grammar_cf`	teste si une grammaire est CF	RE
`is_grammar_lin`	teste si une grammaire est LIN	RE
`is_grammar_llin`	teste si une grammaire est LLIN	RE
`is_grammar_rlin`	teste si une grammaire est RLIN	RE
`is_grammar_reg`	teste si une grammaire est REG	RE
`derivation`	ensemble des dérivés d'un mot selon une production	RE
`leftmost_derivation`	mot obtenu par dérivation la plus à gauche	RE
`language_n`	langage obtenu en moins de k dérivations	RE
`language_n_from_word`	langage en moins de k dérivations en partant d'un mot donné	RE
`language_word_size`	ensemble des mots de taille $\leqslant k$ reconnus par une grammaire	CS
`is_word_in_cs`	teste si un mot reconnu par une grammaire	CS
`is_cf_empty`	teste si le langage d'une grammaire est vide	CF
`is_cf_finite`	teste si le langage d'une grammaire est fini	CF
`simplify_cf`	forme propre d'une grammaire	CF
`chomsky`	forme normale de chomsky (FNC) d'une grammaire propre	CF
`is_word_in_CNF`	teste si un mot est dans le langage d'une grammaire sous FNC	CF
`grammar_union`	union de deux grammaires	RE
`grammar_concat`	concaténation de deux grammaires	RE
`grammar_star`	étoile de Kleene d'une grammaire	RE
`reg_gram_union`	union de deux grammaires (dans REG)	REG
`reg_gram_concat`	concaténation de deux grammaires (dans REG)	REG
`reg_gram_star`	étoile de Kleene d'une grammaire (dans REG)	REG
`regexp_to_grammar`	converti une expression rationnelle en grammaire REG	REG
`grammar_to_regexp`	converti une grammaire en expression rationnelle	REG
`rlin_to_reg`	converti une grammaire RLIN en grammaire REG	REG
`llin_to_reg`	converti une grammaire LLIN en grammaire REG	REG
`monotonous_to_cs`	converti une grammaire monotone en grammaire CS	CS

4.6 Perspectives

Tout ce que nous avions prévu de faire a été implémenté : les principales fonctions décidables sur les grammaires et d'autres commodités. Cependant, il reste de nombreuses perspectives pour contribuer au développement de MIMot. Par exemple, certains algorithmes implémentés peuvent peut-être être améliorées ; de plus, certains problèmes indécidables sur les grammaires le sont peut-être sur certaines instances : on pourrait généraliser les fonctions de MIMot à un plus grand nombre de grammaires. Il est également possible qu'au fil de l'utilisation d'un tel logiciel on se rende compte de fonctions non implantées et qui seraient utiles. Mais comme on le voit, l'implantation de certaines fonctions supplémentaires conduirait vraisemblablement à plus de recherche.

5 Automates

5.1 Automates finis

Il y a différents types d'automates finis : déterministes ou non, avec ou sans ε-transitions. Ceci peut donc naturellement conduire à définir autant de types distincts d'automates. Pourtant, le type retenu est un type plus général englobant toutes ces variantes d'automates finis. Cette solution a été retenue parce que d'une part il n'en résulte pas de complexité plus grande des fonctions manipulant les automates, d'autre part le code est plus léger car on évite de coder pour chaque type les fonctions pour lesquelles ce type n'est pas crucial.

Par ailleurs, nous avons choisi la structure d'*enregistrement* du langage *Objective Caml*, et non la structure de n-uplet de la définition formelle d'un automate fini. Ce choix se justifie par un soucis d'uniformité du code (type enregistrement des grammaires...) et de lisibilité (noms des champs), ainsi que pour des aspects pratiques (accès direct à un champ sans filtrage).

Le type `automaton_t` retenu est donc :

```
type automaton_t =
    {
      nb_states : int;
      sigma : alphabet_t;
      transitions : transition_t list;
      initial : state_t;
      finals : state_t list;
    }
```

Où le type `transition_t` est défini par :

```
type transition_t =
  | D   of state_t * letter_t * state_t
  | ND  of state_t * letter_t * state_t list
  | Eps of state_t * state_t list
```

Listons maintenant les fonctionnalités implantées, avec leur type et une courte description.

Reconnaissance d'un mot :

```
val recognized : automaton_t -> word_t -> bool
```

`recognized A u` renvoie *vrai* si le mot `u` appartient au langage défini par l'automate `A`

Déterminisation :

```
val determinize : automaton_t -> automaton_t
```

`determinize A` renvoie un automate déterministe équivalent (*i.e.* reconnaissant le même langage) à l'automate `A`.
Le nombre d'états de l'automate déterminisé est *a priori* exponentiel en le nombre d'états de l'automate passé en argument. La fonction construit un automate dont les états sont des ensembles d'états de l'automate en entrée et où il y a une transition (déterministe) entre deux états $\{q_{i_1}, \ldots, q_{i_k}\}$ et $\{q_{j_1}, \ldots, q_{j_l}\}$ étiquetée par une lettre a si et seulement si dans l'automate d'entrée, chaque état q_{j_*} est accessible à partir d'un état $q_{i_.}$.
Les états de l'automate ainsi construit sont ensuite renommés (pour être du même type que les états de l'automate d'entrée) et les autres champs sont modifiés en conséquence.

Minimisation :

```
val minimize : automaton_t -> automaton_t
```

`minimize A` renvoie un automate déterministe complet, unique au nom des états près, équivalent à `A` et possédant le plus petit nombre d'états possibles.
L'algorithme de minimisation employé est l'algorithme de *Moore* : on détermine les paires (p, q) d'états distingables (*i.e.* tels qu'il existe un mot u étiquetant un chemin partant de p conduisant à un état final et un chemin partant de q ne conduisant pas à un état final) puis on en déduit les ensembles d'états indistingables et on les fusionne (en modifiant en conséquence les autres champs).
L'implémentation de cet algorithme est en kn^2, où k est le cardinal de l'alphabet de l'automate, et n son nombre d'états. On peut obtenir une complexité en $kn \log n$ de cet algorithme en utilisant une structure de données plus complexe pour les transitions (algorithme d'*Hopcroft*).

Complétion :

```
val complete : automaton_t -> automaton_t
```

`complete A` renvoie un automate équivalent à `A` mais complet, *i.e.* dans lequel pour chaque état p et lettre α, il existe une transition partant de p étiquetée par α. Il suffit pour cela de rajouter un état "puits" à `a`.

Négation

```
val aut_Neg : automaton_t -> automaton_t
```

`aut_Neg A` renvoie un automate reconnaissant le langage constitué des mots qui ne sont pas dans le langage reconnu par `A`

Conversion entre automates finis et expressions rationnelles

```
val aut_Empty : automaton_t
val aut_Word : word_t -> automaton_t
val aut_Concat : automaton_t -> automaton_t -> automaton_t
val aut_Plus : automaton_t -> automaton_t -> automaton_t
val aut_Star : automaton_t -> automaton_t
```

Ces fonctions permettent de construire récursivement un automate fini reconnaissant le même langage qu'une expression rationnelle donnée. `aut_Empty` reconnaît le langage vide, et `aut_Word w` le langage réduit au mot `w`.
Si l'automate `A1` (*resp.* `A2`) reconnaît le langage $L(e_1)$ de l'expression rationnelle e_1 (*resp.* $L(e_2)$) alors :

- `aut_Concat A1 A2` reconnaît $L(e_1.e_2)$
- `aut_Plus A1 A2` reconnaît $L(e_1 + e_2)$
- `aut_Star A1` reconnaît $L(e_1^*)$

Ces fonctions utilisent les ε-transitions et l'indéterminisme pour construire facilement un automate adéquat, que l'on peut alors déterminiser et minimiser si l'on veut (on en verra l'intérêt avec la fonction `équivalent`).

Équivalence :

```
val equivalent : automaton_t -> automaton_t -> bool
```

`equivalent A B` renvoie *vrai* si les automates A et B reconnaissent le même langage, *faux* sinon.

Elle procède en déterminisant, complétant puis minimisant les automates A et B. L'unicité de l'automate minimal, aux noms des états près, permet alors de vérifier facilement l'équivalence (matching en temps linéaire en la taille des automates minimaux).
Notamment, cette fonction s'avère utile pour déterminer si deux expressions rationnelles définissent le même langage : on construit des automates associés aux expressions rationnelles (*cf* les fonctions précédentes) puis on détermine s'ils sont équivalents.

Fonctions diverses :

```
val check_automaton : automaton_t -> automaton_t
```

Lorsque l'utilisateur entre un automate au clavier, il ne le fait pas forcément de la manière la plus "adaptée" au type `automaton_t`. Par exemple, pour créer un automate dans lequel un état p peut conduire par la lettre a aux états q_1 et q_2, il peut créer les deux transitions $D(p, a, q_1)$ et $D(p, a, q_2)$. Cependant un tel automate ne sera pas déterministe, et il aurait été plus naturel (ceci étant certes subjectif) d'entrer une seule transition $ND(p, q, [q_1; q_2])$.
Dans le même ordre d'idées, il est plus naturel d'avoir une transition $D(p, a, q)$ que $ND(p, a, [q])$, ou $ND(p, a, [q_1, q_2, q_3])$ plutôt que $ND(p, a, [q_1, q_2])$ et $D(p, a, q_3)$...
Les transformations de ce type ne sont pas exhaustivement énumérés ici, mais dans le code source de la fonction `check_automaton`, qui les effectue sur un automate qu'on lui passe en argument.

```
val is_determinist : automaton_t -> bool
```

`is_determinist A` renvoie *vrai* si `A` est déterministe, *faux* sinon.

5.2 Automates à pile

Un automate à pile (*pushdown automaton* en anglais) est un automate fini travaillant avec une pile sur laquelle il peut mettre ou enlever des lettres, ce qui lui apporte des capacités de "comptage", et ainsi une puissance de reconnaissance supérieure à celle des simples automates finis précédemment décrits. Les langages reconnus par de tels automates indéterministes sont dits *algébrique* : c'est une sur-classe stricte des langages rationnels. Cette classe de

langages est aussi celle reconnue par grammaires *context-free* (ou *Chomsky-2*).

Nous avons implémenté les automates à pile *indéterministes* et non les *déterministes* car, contrairement au cas des automates finis, ils ne reconnaissent pas la même classe de langages. Il en résulte aussi qu'il n'existe pas de fonction déterminisant un automate à pile indéterministe.

Le type `pd_automaton` retenu est :

```
type pd_automaton_t =
    {
      pd_nb_states : int ;
      pd_alphabet : alphabet_t ;
      pd_stack_alphabet : alphabet_t ;
      pd_transitions : pd_transition_t list ;
      pd_stack : letter_t Stack.t ;
      pd_initial : state_t
    }
```

Où le type `pd_transition_t` est défini par :

```
type pd_transition_t =
    state_t * letter_t * letter_t * (state_t * letter_t) list
```

Listons maintenant les fonctionnalités implantées, avec leur type et une courte description.

Reconnaissance d'un mot :

```
val pd_recognized : pd_automaton_t -> word_t -> bool
```

`pd_recognized A u` renvoie *vrai* si le mot `u` appartient au langage défini par l'automate `A`

Conversion entre automate à pile et grammaire *context-free* :

```
val pd_automaton_to_grammar : pd_automaton -> grammar_t
val grammar_to_pd_automaton : grammar -> pd_automaton
```

Ces deux fonctions permettent de passer d'un automate à pile reconnaissant le langage algébrique L à une grammaire *context-free* reconnaissant le même langage, et réciproquement.

5.3 Perspectives

Les automates finis reconnaissant des langages de mots infinis (ω-langages) n'ont pas été implémentés par manque de temps essentiellement. De plus, leur étude a à peine commencée dans le cours d'automates et langages formels... Les fonctionnalités *a priori* implantables sont celles de conversion entre différents automates de ce genre (*Büchi*, *Müller* et *Rabin*, algorithme de Safra...) et un lien avec un prolongement de la partie sur les expressions rationnelles (expressions décrivant des ω-langages).

Les automates cellulaires n'ont pas été implantés non plus, la principale raison étant que rien (sauf cas triviaux) n'est décidable pour de tels automates. La simple question de *reconnaissance* d'un mot ne peut être tranchée. On peut à la rigueur faire tourner un automate cellulaire sur un mot en entrée : il ne s'arrêtera que si le mot est dans le langage qu'il reconnaît ; on ne peut donc pas répondre "non" si le mot n'est pas dans le langage.

6 Langage de l'interface, liaison avec les parties algorithmiques

Certes l'ensemble des fonctionnalités proposées par MIMot peut aisément s'utiliser sous forme d'une petite bibliothèque Caml, éventuellement intégrable à un *toplevel* grâce à l'outil OCamlMkTop. Cependant, le risque aurait été de limiter son utilisation aux seuls utilisateurs de ce langage particulier. Il nous a donc paru important de créer une interface particulière pour ce logiciel, qui disposerait alors d'un langage particulier facile à apprendre par n'importe quelle personne méconnaissant la programmation, de sorte qu'elle puisse travailler sur les langages formels à partir d'une interface graphique, ou d'un *toplevel* texte.

Notre langage se réduit à la base à des appels de fonction de MIMot et à des assignations de variables, pour pouvoir réutiliser des résultats précédemment trouvés. On y ajoute les demandes d'aide sur une fonction donnée, et surtout, ce qui alourdit terriblement le *parser*, les raccourcis pour des fonctions triviales, notamment les fonctions arithmétiques, booléennes, l'égalité...

6.1 Analyse lexicale

L'analyseur lexical, écrit en OCamlLex (fichier `lexer.mll`) va découper la chaîne de caractères provenant de l'interface (texte ou graphique) en jetons de différents types:

- le signe de fin de commande: point-virgule
- les parenthèses
- le séparateur d'expressions (arguments de fonctions): virgule
- les délimiteurs spécifiques: accolades, guillemets, barres obliques, crochets, chevrons (double inférieur ou double supérieur)
- les opérateurs: plus, moins, fraction, étoile, accent circonflexe, point, et, ou, non, tests d'égalité ou d'inégalité, flèche vers la droite
- le symbole d'assignation: flèche vers la gauche
- le point d'interrogation pour l'aide
- les identificateurs (de fonctions ou de mots)

– lettres entre apostrophes, nombres, variables débutant par $ directives débutant par # sont directement différenciés

6.2 Analyse syntaxique

L'analyseur syntaxique définit en OCamlYacc la grammaire de notre langage (fichier `parser.mly`). Les règles sont très simples, de manière à être facilement assimilables par n'importe qui. La définition directe de valeurs se fait ainsi:

– booléens, entiers et lettres sont reconnus dès l'analyse lexicale

– les alphabets sont une liste de lettres séparées par des virgules, entre accolades

– les mots sont constitués d'une séquence de lettres (entre apostrophes) et de caractères (considérés comme étant une lettre chacun, pour ne pas alourdir) entre guillemets

– les expressions rationnelles sont définies entre chevrons, en utilisant les notations habituelles: point, plus et étoile

– les grammaires sont définies entre barres obliques, en spécifiant obligatoirement comme dernier argument la liste des productions (mot, flèche, mot), séparées par des virgules, et éventuellement (les arguments absents sont inférés grâce aux productions) dans l'ordre, l'alphabet des variables, l'alphabet des terminaux et l'axiome

– les automates sont définies entre < | et | >, en spécifiant obligatoirement comme derniers arguments la liste d'états finaux entre crochets et la liste des transitions (état virgule lettre flèche entier), séparées par des virgules, et éventuellement (les arguments absents sont inférés grâce aux transitions) dans l'ordre, le nombre d'états, l'alphabet et l'état initial

Les autres expressions autorisées sont:

– l'assignation de variables

– l'appel de variables, de fonctions

– les opérations unaires et binaires arithmétiques et booléennes classiques

– la manipulation intuitive des lettres, mots et expressions régulières: concaténation, puissance, extraction de facteurs

– une demande d'aide sur une fonction

6.3 Typage

Il va de soit que les attributs des règles de la grammaire LALR correspondante, synthétisés directement pour permettre une évaluation au vol, posent des problèmes concernant le typage fort de Caml, puisqu'il gère des objets fondamentalement différents. Il a donc fallu créer dans un module particulier (fichier `values.ml`) un type *valeur*, somme de tous les types gérés (entiers, booléens, lettres, alphabets, mots, expressions rationnelles, grammaires, langages finis, règles de production, automates), qui est utilisé dans l'analyseur.

Il faut maintenant faire des tests de types presque à toutes les règles de notre analyseur, afin de s'assurer par exemple que l'on n'additionne pas des alphabets avec des grammaires[2]... Pour cela, on définit des destructeurs du type *valeur*, afin d'extraire le contenu même des objets et lever, si le type de valeur n'est pas celui recherché, une exception, qui sera, comme les autres, rattrapée dans l'interface.

6.4 Variables

Les variables sont stockées (fichier `values.mli` sous leur type *valeur* dans une table de hachage, dont la fonction est optimisée pour les chaînes de caractère (fichier `stringtbl.mli`), de manière à trouver rapidement le contenu, de type *valeur*, en fonction du nom. L'assignation d'une variable déjà assignée écrase bien sûr la précédente définition. Quelques variables, comme l'alphabet ou l'expression régulière vide, sont prédéfinies dans cette table.

6.5 Fonctions

Les fonctions définies dans les autres modules et destinées à être intégrées à notre mini-langage sont stockées dans une table de symboles (fichier `functions.mli`), également optimisée pour les chaînes de caractères. On y intègre, outre le *wrapper* permettant d'appeler la fonction avec des arguments de type *valeur* et en retournant un type *valeur*, la liste des types des arguments (il a donc fallu définir un type *type*), qui permet de vérifier le nombre des arguments et leurs types, levant éventuellement une erreur de type. La table contient aussi une courte description (en anglais) de chaque fonction, qui est affichée lors de la demande d'aide.

2. On ne fait pas du C.

6.6 Lettres

Les lettres sont représentées de manière interne à notre programme par un type particulier, correspondant à des entiers (ce de manière à ne pas se limiter à 26 ou 256 lettres). Néanmoins, l'utilisateur, quant à lui, écrit et souhaite lire à l'écran de "vraies" lettres, représentées par des chaînes de caractères. Il a donc fallu mémoriser les associations des lettres entre les deux représentations dans deux tables, une pour les chaînes, l'autre pour les entiers (fichier `inttbl.mli`).

L'interface du module pour les lettres (fichier `letters.mli`) fournit, outre les fonctions de conversion dans les deux sens (une pour l'analyse lexicale, l'autre pour l'affichage), des fonctions permettant de créer une lettre sans l'associer à une chaîne de caractère particulière, comme certains algorithmes sur les mots ou sur les expressions régulières en ont besoin, ou de créer une lettre dans les tables, en l'associant à une chaîne de caractère non encore assignée, utilisée dans les grammaires, par exemple pour créer des variables nouvelles, qui ne seront pas utilisées ailleurs, mais qu'on a besoin de pouvoir afficher à l'écran.

6.7 Joli-affichage

Le résultat de l'évaluation d'une expression doit ensuite être retourné à l'interface de manière lisible pour l'utilisateur (fichier `pretty_printer.mli`). Nous affichons le type de l'objet renvoyé (à la manière d'un *toplevel* Caml), suivi de sa valeur, en essayant de simplifier les notations, notamment au niveau des parenthésages (par exemple dans une concaténation de trois expressions rationnelles). La syntaxe de l'affichage est la même que pour l'analyse, ce qui permet éventuellement un *copier-coller* de résultat afin de l'intégrer dans une expression ultérieure.

6.8 Erreurs

Les exceptions levées par Caml depuis le *parsing* ou les différents modules fonctionnels (qui définissent des exceptions spécifiques pour faciliter leur utilisation particulière), sont rattrapées par un *pattern matching*, associant une chaîne de caractère à chaque erreur, qui sera affichée par l'interface au même titre qu'un résultat valide (fichier `eval.ml`). Mais l'interface garde quand même la connaissance de l'invalidité du résultat, ce qui lui permet éventuellement de ne pas mettre l'instruction fautive dans l'historique.

6.9 Gestionnaire des sessions

Des directives particulières ont été ajoutées à l'analyseur syntaxique pour sauvegarder le travail effectué dans un fichier, et le récupérer. En plus de permettre le rechargement rapide de lignes précédemment saisies sous MIMot, Cette fonctionnalité permet de saisir des commandes MIMot depuis un éditeur de texte quelconque. Les directives utilisées sont `#save nomfichier` et `#load nomfichier`. En outre, deux autres directives sont fournies: `#quit` pour quitter MIMot, et `#clear`, pour démarrer une nouvelle session, c'est-à-dire signaler au programme que toutes les lignes saisies avant cette directive ne doivent pas être sauvegardées par `#save`.

6.10 Aide à l'utilisateur

Compte tenu de la diversité des fonctions proposées par MIMot, il nous a semblé indispensable de fournir à l'utilisateur un système d'aide lui permettant de connaître les fonctions à sa disposition, ainsi que la façon de les utiliser. A cet effet, les commandes de la forme `? mot-clé` sont acceptées, où `mot-clé` est soit `list` qui demande d'afficher la liste des fonctions disponibles, soit le nom d'une de ces fonctions, auquel cas MIMot donne le typage de la fonction, ainsi qu'un brève description de ce qu'elle calcule. Il est à noter que la génération de l'aide est automatique. Elle utilise les informations de typage de la table des symboles. L'ajout de nouvelles fonctionnalités au projet n'induit donc pas de surcoût quant à la maintenance du système d'aide.

6.11 Perspectives

Il eut été extrêmement intéressant d'élargir le langage de notre logiciel en y intégrant une structure complexe telle que le produit cartésien ou les ensembles (ou listes) quelconques, voire même les fonctions définies par l'utilisateur. Le premier permettrait entre autres de définir les morphismes sur les mots. Le second faciliterait la récupération des solutions de systèmes d'équations sur les langages rationnels et rendrait plus aisée l'analyse syntaxique des alphabets, langages finis, des ensembles d'états dans les automates, des listes de transition ou de règles de production... Cela représente une amélioration considérable, qui demanderait une révision complète du *parser* et du système de typage, les rendant incomparablement plus complexes.

Une autre amélioration envisageable serait un rattrapage plus précis des erreurs commises par l'utilisateur. Le *pretty-printer* renvoie pour l'instant des messages non localisés, si l'on excepte, en ce qui concerne les appels de

fonctions, la place dans la liste d'arguments. Il faudrait que l'analyseur syntaxique donne à l'interface une information de situation de l'exception, afin de rendre le langage plus facile à utiliser et à maîtriser. Une interface moins austère permettrait de rendre accessible ce logiciel au plus grand nombre, but premier de MIMot.

7 Interface texte

La nécessité de doter MIMot d'une interface permettant d'entrer des données et de récupérer des résultats en mode texte est apparue au cours du développement. Il nous a en effet semblé qu'une interface texte présentait plusieurs avantages dont il aurait été dommage de se priver :

- l'existence de bibliothèques standard, tant pour la saisie de texte que pour son affichage, permettait d'envisager un développement rapide et peu onéreux ;
- la compilation d'une telle interface nécessite très peu de ressources (temps, bibliothèques), et donc la présence d'une interface en mode texte permet à un utilisateur désireux de découvrir MIMot de le faire assez rapidement, et de pouvoir s'en faire une bonne idée sans avoir à installer les bibliothèques requises pour faire fonctionner l'interface graphique.

C'est pour ces raisons que nous avons choisi, un peu tardivement il est vrai, de nous consacrer au développement d'une interface texte s'appuyant sur deux bibliothèques extrêmement répandues sous Unix : `readline` qui permet de saisir du texte tout en s'y déplaçant, et `history` qui permet de gérer une liste des lignes déjà entrées.

7.1 Travail réalisé

Les bibliothèques citées ci-dessus ont été écrites en C. Par conséquent, comme MIMot est écrit en Caml, la liaison avec ces bibliothèques a nécessité l'écriture d'une interface entre Caml et C, pour que les modules Caml puissent appeler les fonctions fournies par les bibliothèques C. Bien que le code de cette interface se révèle finalement de taille assez modeste, un certain temps a été nécessaire à son élaboration. Il a en effet été nécessaire d'étudier une certaine quantité de documentation, d'une part les documentations relatives aux bibliothèques elles-mêmes, d'autre part la documentation relative à l'interfaçage entre C et OCaml. Comme de coutume, la liaison entre des modules écrits dans des langages de programmation différents est un point assez délicat.

Le sous-projet consiste pour l'instant en quatre fichiers source :

- `readline_low.c` : il s'agit de la partie en C de l'interface. On y définit des fonctions chargées de récupérer des arguments stockés au format Caml, de les convertir en arguments acceptables par des fonctions C,

d'appeler les fonctions C correspondantes, de convertir les résultats de ces appels au format Caml, et de les retourner.

- `readline.ml` et `readline.mli` : ces fichiers contiennent les déclarations Caml des fonctions évoquées précédemment. On y donne leur type et on déclare les exception qui peuvent être levées par les fonctions C en cas d'erreur.

- `toplevel.ml` : Ce fichier représente le module principal du projet MIMot (du moins dans sa version texte). Il contient uniquement une boucle d'interaction, qui lit une chaîne de caractères sur l'entrée standard, la transmet à l'analyseur syntaxique et affiche le résultat de cette analyse, sous forme d'une chaîne de caractères.

7.2 Aspect de l'interface

Le comportement de l'interface texte est le suivant. Lorsque MIMot est lancé, un message indiquant que MIMot attend des données de l'utilisateur s'affiche. L'utilisateur peut alors entrer des données de manière confortable, puisque toutes les facilités offertes par la bibliothèque readline sont à sa disposition : utilisation des flèches pour se déplacer dans le texte en cours de saisie, utilisation des raccourcis habituels pour effacer des mots...

Lorsque l'utilisateur a terminé la saisie, il le signale en la validant par un appui sur la touche entrée. La ligne est alors analysée, et le résultat de cette analyse est affiché.

L'utilisateur peut ensuite entrer une nouvelle ligne à évaluer. Cependant, cette fois, il dispose d'une possibilité supplémentaire : il peut utiliser la flèche vers le haut pour sélectionner une ligne précédemment écrite, qu'il peut modifier à sa guise avant de la valider.

7.3 Perspectives

Tout d'abord, un aspect un peu regrettable de notre toplevel est le fait que le retour à la ligne valide la commande, la bibliothèque readline gérant cet événement spécifiquement. Cela empêche ainsi l'utilisateur de la présenter joliment sur plusieurs lignes, ce qui pourrait faciliter la lisibilité des grammaires et des automates notamment. Donner la possibilité de taper des commandes sur plusieurs lignes, comme le parser le permet, représenterait une amélioration intéressante.

L'interface, telle qu'elle a été décrite ci-dessus, est déjà relativement fonctionnelle. Cependant, comme les fonctions proposées par MIMot ont des noms

relativement longs, l'interface gagnerait en confort et en efficacité si elle était en mesure de fournir des mécanismes de complétion, sur les noms de fonctions par exemple, mais aussi pour les noms de variables. Ces mécanismes existent dans la bibliothèque `readline`. Celle-ci permet en effet aux applications d'enregistrer des fonctions qui spécifient comment compléter les chaînes de caractères. Il serait intéressant de mener une étude pour déterminer comment exploiter au mieux une telle fonctionnalité. On pourrait par exemple imaginer que l'interface soit capable de compléter les noms de fonctions ou de variables, ce qui éviterait à l'utilisateur d'avoir à les mémoriser, et diminuerait le nombre de lignes erronées. La mise en œuvre d'une telle fonctionnalité se heurte cependant à une petite difficulté technique : il faut fournir à la bibliothèque readline des fonctions qui lui disent comment compléter le texte. Or, ces fonctions devraient certainement être écrites en Caml, pour leur permettre d'accéder plus facilement à la table des symboles. Donc, il faudrait que du code C (celui de readline) puisse appeler du code Caml, c'est-à-dire le contraire de ce qui était fait jusqu'à présent. Or ceci est difficile à réaliser et demande une bonne connaissance à la fois de C et de Caml. D'ailleurs, ce point particulier est considéré comme avancé dans la documentation OCaml.

8 Interface graphique

L'interface graphique semble être la partie la moins difficile et la plus rébarbative de ce projet. Elle reste cependant indispensable à la concrétisation d'un logiciel informatique, quel qu'il soit ! D'autant plus que *MIMot* se veut un outil pratique qui permet, entre autre, une manipulation facile. En quelque sorte, il veut apporter aux théoriciens des langages formels ce qu'apporte la plus vulgaire des calculatrices à Monsieur Tout-le-monde ou, à une échelle différente, *Maple* aux mathématiciens. De cette motivation est né le sous-projet Interface Homme-Machine (IHM), baptisé *MotMI*

8.1 Description générale

La société chrono-maniaque moderne exige une efficacité optimale de chaque individu. Plus particulièrement, elle ne laisse aux chercheurs de haut niveau, tels que ceux des langages formels, guère de temps à consacrer à un outil aussi basique que *MIMot*. (Surtout si son utilisation équivaut à l'apprentissage d'un nouveau langage de programmation, même lorsque ce dernier s'appelle *OCaml* !)

Ce contexte a motivé le développement d'une interface graphique facile à maîtriser. Ainsi nous avons adopté le concept *MotMI*, les **MOT**s **M**is en **I**mages.

MotMI consiste en une réalisation d'une interface *Maple*-like permettant d'utiliser directement la librairie *MIMot*. Concrètement, *MotMI* se greffera à la partie lexeur/parseur pour interpréter l'entrée des commandes, puis, à toutes les parties restantes du projet pour afficher les résultats renvoyés.

L'objectif de *MotMI* est donc simple :

- Savoir prendre la commande de l'utilisateur et la faire passer à l'analyse lexicale et syntaxique.
- Savoir afficher les résultats renvoyés par les parties fonctionnelles de *MIMot*.
- Savoir de plus effectuer les deux tâches précédentes d'une manière digne de l'ambition d'excellence que nous poursuivons.

8.2 Utilisation de *MIMot*

MotMI dispose de tout le nécessaire pour utiliser simplement *MIMot*.

- La fenêtre principale du logiciel est composée essentiellement d'une ligne de commande `"Command line"` et d'une sortie texte, `"Result"`,

affichant les résultats renvoyés par la bibliothèque $MIMot$ quand on lui passe en argument la dite ligne de commande.

- La barre de menu de $MotMI$ contient la très utile fonctionnalité `"Help"` permettant à un novice de $MIMot$ d'avoir une prompte familiarisation à cette bibliothèque.

- La barre de menu offre également la possibilité d'ouvrir une fenêtre `"History"` permettant d'avoir un aperçu rapide des commandes passées.

En résumé, il est indispensable de disposer de $MotMI$ pour une utilisation simple et rapide de $MIMot$.

8.3 Gestion des fichiers de sauvegarde

De nombreux travaux d'arrache-pied ont été bêtement perdus sans une sauvegarde intelligente. En effet, votre ligne électrique pourrait toujours être interrompue soudainement, votre carte graphique favorite pourrait toujours vous lâcher sans raison, votre ordinateur pourrait toujours s'éteindre brusquement suite à la réception d'une météorite venant de $Mars$, etc. Bref, les sauvegardes pour le travail, c'est aussi important que la 5e roue pour la route.

De cette excellente réflexion, nous avons implémenté les fonctionnalités basiques `"New"`, `"Open"`, `"Save"`, `"Save As"` au sein de $MotMI$.

8.4 Ergonomie dans $MotMI$

Qu'il est parfois ennuyeux de devoir quitter son clavier et prendre la souris pour continuer son travail ! Comprenant cette obligation dérangeante, $MotMI$ a implémenté les raccourcis- clavier pour la plupart des opérations de la barre de menu.

Une autre implémentation que l'on pourrait mentionner est la gestion des raccourcis `"Page-Up"`, `"Page-Down"` pour reprendre une ancienne ligne de commande à la mode de shell. Ceci requiert entre autres l'implantation des listes doublement chaînées, que notre analyste-développeur-chef de sous-projet a décidé d'englober dans une structure d'objet afin de permettre une manipulation plus transparente du code.

Une fenêtre d'accueil conviviale à souhait et apte à répondre à toute exigence de la part de nos sponsors sera présentée à chaque exécution du logiciel. Une fois cette fenêtre fermée, la fenêtre principale de $MotMI$ se présente. De beaux décors minutieusement soignés sont présents dans cette dernière. Par souci de réalisme, nous avons choisi de simuler la sponsorisation de la

société SHADOKO. Enfin, la fenêtre About MIMot a été également travaillée avec amour afin de permettre aux générations futures de garder le contact avec leur aïeux.

8.5 Bref aperçu sur l'implantation de *MotMI*

Cette partie est une très courte introduction au code de motMI.ml. Nous allons commenter brièvement le fichier *motMI.ml*

8.5.1 LablGtk

L'interface entière a été implémentée sous *OCaml*. Plus particulièrement, il s'agit de la bibliothèque *LablGtk* de *OCaml*. Ainsi, un grand merci à Messieurs Garrigue, Fauque, Furuse et Kagawa pour ce considérable apport.

Notez que les parties fonctionnelles de *MIMot* requiert la version 3.06 de *OCaml*. Par conséquent, *MotMI* requiert une version de *LablGtk* qui supporte ladite version de *OCaml*, par exemple, la version $1.2.5 - 4$.

8.5.2 La classe commandList

L'interface est complètement orientée objet. La classe commandList contient une liste doublement chaînée et offre toutes les manipulations nécessaires qui l'accompagnent. Elle sert uniquement à la gestion des raccourcis "Page-Up", "Page-Down" et celle de l'historique.

8.5.3 La classe motMI

Cette classe définit les fenêtres qui seront exécutées lors d'un appel de leur fonction win#show (). Elle n'a que deux méthodes :

```
class motMI () =
object
        method pubWin = GWindow.window
        method mainWin = GWindow.window
end
```

pubWin est la fenêtre de publicité (vive le franc-lais). mainWin est la fenêtre principale de *MotMI*.

Ainsi, la classe momi, présente à la fin du fichier, qui *hérite* motMI n'est là que pour afficher les fenêtres adéquates via les appels de pubWin#show () et mainWin#show () :

```
class momi () =
```

```
object
        inherit motMI ()

        initializer
                ignore (pubWin#connect#destroy mainWin#show);
                ignore (mainWin#connect#destroy GMain.Main.quit);

                pubWin#show ()
end
```

8.6 Perspectives

Parmi les fonctionnalités prévues au niveau, seul l'affichage graphique des automates n'a pas abouti. Une telle réalisation s'est avérée dépasser les ressources matérielles et temporelles dont nous disposions, mais il serait en fait intéressant de se pencher sur cette question, qui donnerait à l'utilisateur un moyen plus concret de se représenter les automates, la représentation syntaxique sous forme de quintuplet (même si elle est allégée par les arguments optionnels lors de la définition) se révélant parfois un peu lourde... Pour cela, on pourrait peut-être se baser sur la librairie (non standard) GraphX de Caml, qui dessine des graphes, éventuellement orientés ou étiquetés, avec manipulation interactive, mais cela représente un travail conséquent.

9 Utilisation

Si quelques fonctionnalités pourraient encore être implantées, MIMot présente déjà un système relativement puissant permettant de manipuler différentes structures formelles tournant autour de la théorie des langages.

Comme c'est le cas pour Maple, cet outil peut être utilisé dans l'enseignement pour donner des exemples de la décidabilité de certains problèmes, ou pour alléger les vérifications rébarbatives, qui, malheureusement, sont souvent trop présentes dans ce domaine de recherche. Le seul frein à sa diffusion est la peur des machines dont trop de théoriciens font encore part...

10 Déploiement

Le programme peut se trouver par archive CVS depuis le *repository* `goldorak.ens-lyon.fr:/home/shindere/CVS`.

Pour la manipulation des structures et fonctions de la bibliothèque constituée par MIMot, il suffit de détenir une version de OCaml; cette utilisation sera facilitée par le mode Tuareg d'Emacs, le gestionnaire de bibliothèques OCamlBrowser ou l'outil de liaison des objets en un *toplevel* OCamlMkTop, utilisable directement par la commande `make top`.

L'utilisateur qui veut utiliser le logiciel proprement dit doit posséder un compilateur OCamlC d'une version supérieure à la 3.05[3]. En effet, dans la distribution 3.04 subsiste un bogue au niveau des tables de hachage, qui sont beaucoup utilisées par le *parser*. Il faut également disposer de la librairie C ReadLine pour l'interface texte, et de LablGTK[4] pour l'interface graphique.

La compilation est très simple: dans le répertoire racine de l'archive, taper `make` crée l'interface texte `mimot` et taper `make gui` crée l'interface graphique `mimot.x`. Un fichier `INSTALL` est disponible pour plus de précisions.

3. disponible sur `http://caml.inria.fr/ocaml/distrib.html`
4. disponible sur `http://wwwfun.kurims.kyoto-u.ac.jp/soft/olabl/lablgtk.html`