TP 4

AUTOMATE DE THOMPSON

Ce sujet est assez directement inspiré d'une série d'articles écrits par Russ Cox et disponibles en ligne : https://swtch.com/~rsc/regexp/. Russ Cox est l'un des créateurs du langage Go, et l'auteur de la bibliothèque d'expressions régulières RE2.

Introduction

L'objectif de cette séance est de programmer une version rudimentaire (mais utilisable) de `grep` en C, utilisant l'automate de Thompson associé à une expression. Les limitations que nous allons accepter :

- le programme prendra en entrée (en plus de l'expression régulière) un éventuel fichier passé en argument, ou l'entrée standard sinon ;
- le programme traitera le flux d'entrée ligne par ligne, et décidera simplement pour chaque ligne si elle (dans son ensemble) acceptée par l'expression ou non : les lignes acceptées seront envoyées sur la sortie standard ;
- la seule « classe de caractères » valide sera `.` (qui accepte un caractère quelconque, autre que `\n`), il n'y aura donc pas de `\w`, de `[aeiou]`...
- les seuls opérateurs seront la concaténation (que nous noterons explicitement `@`), l'alternative `|`, l'étoile `*` et le « zéro ou une fois » `?` : pas de `{5}` par exemple ;
- les caractères `*`, `@`, `|`, `.` et `?` seront réservés pour les opérateurs, et ne pourront donc jamais être *matchés* (pas de `\?` pour accepter un caractère `?`) ;
- le plus important, et de loin : l'expression sera donnée en notation postfixe, ce qui simplifiera énormément son analyse syntaxique.

Exemple 4.1

En supposant que notre programme a été compilé vers un exécutable `mygrep`, on aurait les équivalences suivantes :

- ```
  grep -E '^ab*$' entree.txt
  mygrep 'ab*@' entree.txt
  ```
- ```
  grep -E '^(ab)*$' entree.txt
  mygrep 'ab@*' entree.txt
  ```
- ```
  grep -E '(ab)*c$'
  mygrep '.*ab@*@c@'
  ```
- ```
  grep -E '([ab].)*'
  mygrep '.*ab|.@*.*@@
  ```

1 Construction de l'automate de Thompson

Exercice 4.2 — Retour sur l'automate de Thompson p. 8

1. Rappeler, sous forme de schémas, les constructions de Thompson pour des expressions de la forme :
   a. a, où $a \in \Sigma$;
   b. ef, où e et f sont des expressions régulières ;
   c. $e|f$, où e et f sont des expressions régulières ;

d. e^*, où e est une expression régulière.

2. Proposer une construction pour e?, où e est une expression régulière et le ? signifie « zéro ou une fois ».

   Remarques

   - Cette construction devra posséder un unique état initial, sans transition entrante, et un unique état final, sans transition sortante.
   - Cette construction est inutile en théorie puisque l'on dispose d'un automate pour ε et donc pour e? $\equiv e|\varepsilon$. Elle nous évite cependant de traiter le cas ε, et utilise moins d'états.

3. Combien de transitions sortantes étiquetées par une lettre un état de l'automate de Thompson peut-il posséder ? et combien de transitions sortantes étiquetées par ε ? Peut-il posséder à la fois les deux types de transition sortante ?

4. Justifier que si la représentation postfixe de e est de longueur n (en tant que chaîne de caractères), alors l'automate de Thompson associé possède au plus $2n$ états.

Nous allons représenter l'automate de Thompson d'une manière très différente de ce que nous avons fait en OCaml lors du TP précédent. Ce n'est pas du tout lié au langage, et pas tellement lié au fait qu'on a un automate non déterministe avec transitions spontanées : c'est tout simplement plus naturel si l'on considère le processus de construction de l'automate.

Chaque état de l'automate sera représenté par une structure de ce type :

```
struct state {
    int c;
    struct state *out1;
    struct state *out2;
    int last_set;
  };

typedef struct state state_t;
```

- Si `c` a valeur entre 0 et 255, l'état possède une unique transition sortante, étiquetée par le caractère `c`.
- Dans ce cas, `out1` pointe vers l'état d'arrivée de cette transition, et `out2` doit valoir `NULL`.
- Si `c` a la valeur particulière `MATCH` (définie globalement comme étant égale à 256), alors il n'a aucune transition sortante, et les pointeurs `out1` et `out2` sont nuls.
- Si `c` a la valeur `EPS` (définie globalement comme égale à 257), alors il possède deux transitions sortantes étiquetées par ε, et `out1` et `out2` pointent vers les états d'arrivée de ces transitions.
- Si `c` a la valeur `ALL` (définie globalement comme égale à 258), alors l'état possède une transition sortante pour chaque caractère de l'alphabet, et ces transitions pointent toutes vers l'état désigné par `out1`. Le pointeur `out2` doit être nul dans ce cas.
- Le champ `last_set` sera expliqué plus tard : pour l'instant, il faut juste savoir qu'il devra être initialisé à `-1` à la création de l'état.

L'automate lui-même sera représenté par la structure suivante :

```
struct nfa {
    state_t *start;
    state_t *final;
    int n;
};

typedef struct nfa nfa_t;
```

- Le champ `start` pointe vers l'état initial de l'automate.
- Le champ `final` pointe vers l'état final.
- Le champ `n` indique le nombre total d'états de l'automate.

Exercice 4.3

p. 8

1. Écrire la fonction `new_state` renvoyant un pointeur vers un nouvel état (alloué sur le tas). Comme dit plus haut, on initialisera `last_set` à `-1`.
2. Écrire la fonction `character` qui renvoie un `nfa_t` reconnaissant le caractère donné. Attention, on renvoie bien un `nfa_t`, par valeur, et pas un `nfa_t*`.
3. Écrire une fonction `all` qui renvoie un `nfa_t` reconnaissant n'importe quel mot de longueur 1. On utilisera le même automate que pour `character`, sauf que le champ `c` de l'état initial sera mis à la valeur `ALL`.
4. Écrire les fonctions `concat`, `alternative`, `star` et `maybe` qui correspondent aux différentes constructions du dernier exercice. Autrement dit, dans l'appel `concat(a, b)`, on suppose que `a` et `b` sont deux automates de Thompson, d'ensembles d'états disjoints, reconnaissant deux expressions régulières e et `f`, et l'on demande de renvoyer l'automate de Thompson pour ef.

```
state_t *new_state(int c, state_t *out1, state_t *out2);

nfa_t character(char c);

nfa_t all(void);

nfa_t concat(nfa_t a, nfa_t b);
nfa_t alternative(nfa_t a, nfa_t b);
nfa_t star(nfa_t a);
nfa_t maybe(nfa_t a);
```

La construction de l'automate de Thompson suit directement la structure de l'expression (en tant qu'arbre binaire/unaire), ce qui permet de la réaliser très naturellement à partir de la version postfixe de l'expression, à l'aide d'une pile.

Exercice 4.4

p. 10

1. Rappeler le principe de l'évaluation d'une expression arithmétique en notation postfixe en traitant l'exemple suivant : `12 4 + 3 2 ! * -` (où `+` et `-` sont binaires et `!` binaire).
2. À l'aide du type `stack_t` et des fonctions associées fournis, écrire une fonction `build` qui prend en entrée une *regex* en notation postfixe (sous forme d'une chaîne de caractères) et renvoie l'automate de Thompson correspondant.
3. Déterminer la complexité temporelle de `build` en fonction de la longueur m de la chaîne donnant l'écriture postfixe de l'expression régulière.

```
struct stack {
    int length;
    int capacity;
    nfa_t *data;
};
typedef struct stack stack_t;

stack_t *stack_new(int capacity);
void stack_free(stack_t *s);
nfa_t pop(stack_t *s);
void push(stack_t *s, nfa_t a);

nfa_t build(char *regex);
```

2 Exécution de l'automate

L'automate de Thompson que nous venons de construire est un ε-AFND, et il n'est donc pas possible de tester l'appartenance d'un mot à son langage en se déplaçant simplement d'état en état à chaque caractère. Deux options se présentent à nous :

- éliminer les ε-transitions puis déterminiser l'automate, pour finalement effectuer la reconnaissance sur l'automate déterministe ;
- décider directement l'appartenance en exécutant l'automate de Thompson, en gérant les ε-transitions et le non déterminisme.

L'élimination des ε-transitions n'est pas complètement immédiate à programmer, mais elle se ramène essentiellement à un parcours de graphe, et peut être effectuée de manière efficace. Le calcul de l'automate des parties pour déterminiser, en revanche, peut provoquer une explosion combinatoire.

Pour l'éviter, nous allons choisir la deuxième approche. Deux variantes sont possibles :

- procéder par *backtracking*, en essayant une par une les transitions possibles à partir d'un état jusqu'à les épuiser ou en trouver une permettant d'accepter le mot ;
- garder trace de l'ensemble des états dans lesquels on *peut* se trouver à un moment donné de la lecture du mot, ce qui revient essentiellement à explorer un chemin dans l'automate des parties, sans calculer explicitement cet automate.

Exercice 4.5 – Exécution avec retour sur trace

p. 10

On se place ici dans le cas particulier des automates non-déterministes du type de l'automate de Thompson :

- un seul état initial ;
- un seul état final ;
- entre une et deux transitions sortantes par état, avec le cas à deux transitions sortantes réservé aux ε-transitions.

1. Proposer une fonction très simple `backtrack` ayant le prototype suivant :

```
bool backtrack(state_t *state, char *s);
```

Cette fonction renverra `true` si la lecture du mot `s` depuis l'état pointé par `state` nous amène dans un état final, `false` sinon. On considérera que le mot s'arrête au premier caractère nul ou `'\n'` rencontré, exclu.

On se donne ensuite les deux fonctions suivantes :

```
bool accept_backtrack(nfa_t a, char *s){
  return backtrack(a.start, s);
}

void match_stream_backtrack(nfa_t a, FILE *in){
  char *line = malloc((MAX_LINE_LENGTH + 1) * sizeof(char));
  while (true) {
    if (fgets(line, MAX_LINE_LENGTH, in) == NULL) break;
    if (accept_backtrack(a, line)) {
      printf("%s", line);
    }
  }
  free(line);
}
```

Remarque
L'appel `fgets(line, MAX_LINE_LENGTH, in)` lit un maximum de `MAX_LINE_LENGTH` caractères depuis le flux `in` jusqu'à tomber sur un caractère `'\n'` ou sur la fin du flux. Ces caractères sont recopiés sur `line`, y compris le retour à la ligne s'il y en avait un, et un caractère nul est

placé ensuite (on peut donc écrire jusqu'à `MAX_LINE_LENGTH + 1` caractères). L'appel renvoie un pointeur nul si aucun caractère n'a été lu (si l'on était déjà à la fin du flux, donc).
On suppose la constante `MAX_LINE_LENGTH` préalablement définie, et suffisamment grande pour traiter toutes les lignes du flux.

2. Écrire un programme ayant le comportement suivant :
   - le premier argument en ligne de commande (obligatoire) est la *regex* sur laquelle on travaille, en notation postfixe ;
   - s'il y a un deuxième argument, il fournit le fichier d'entrée – sinon, l'entrée est l'entrée standard ;
   - le programme traite les lignes de l'entrée une par une, dans l'ordre, en affichant celles qui sont acceptées par l'expression régulière sur la sortie standard.

   Remarque
   On ne se préoccupera pas pour l'instant de libérer proprement la mémoire.
3. Utiliser ce programme pour chercher tous les mots de la langue français contenant à la fois un q et un w. Comparer le temps d'exécution de cette requête avec une requête équivalente traitée par `grep` (on utilisera l'utilitaire `time` pour ce faire : `time grep regex file`).
4. Les fichiers `aXXb.txt` fournis contiennent chacun une unique ligne, constituée de `XX` caractères a suivis d'un unique caractère b. Mesurer le temps pris par votre programme pour chercher dans ces fichiers une occurrence de l'expression $(aa?)*$. Que constate-t-on, et comment l'expliquer ?
5. Comparer ce temps avec celui pris pour la même tâche par `grep` et par `grep_py` (script Python qui utilise le moteur d'expressions régulières de Python).
6. Que se passe-t-il si on exécute notre programme avec la *regex* suivante : $(a?)^*$ (et une entrée non vide quelconque) ?
   Remarque
   C'est évidemment un problème qu'il faudrait régler si on voulait réellement utiliser cette solution, mais nous allons en choisir une autre. . .

L'autre idée possible pour exécuter un automate non déterministe (avec ou sans ε-transitions) est de maintenir à jour l'ensemble des états dans lesquels on peut se trouver à un moment donné de la lecture de l'entrée. Un tel ensemble correspond exactement à un état de l'automate des parties : essentiellement, on explore et construit uniquement le chemin de l'automate des parties qui correspond à la lecture du mot.

Pour représenter un ensemble d'états, on utilise la structure suivante :

```c
struct set {
    int length;
    int id;
    state_t **states;
};

typedef struct set set_t;
```

On fournit une fonction pour créer un ensemble vide de capacité et `id` données, et une pour libérer la mémoire associée :

```c
set_t *empty_set(int capacity, int id){
    state_t **arr = malloc(capacity * sizeof(state_t*));
    set_t *s = malloc(sizeof(set_t));
    s->length = 0;
    s->id = id;
    s->states = arr;
    return s;
}
```

- Le champ `length` indique le cardinal de l'ensemble.
- Le champ `states` est un tableau de pointeurs vers des états. Sa longueur sera au moins égale à `length` mais peut être supérieure : dans ce cas, seules les valeurs des cases d'indice `0` à `length - 1` ont un sens, les autres peuvent être ignorées.
- Le champ `id` permet d'identifier l'ensemble de manière unique. Ce champ sera utilisé en combinaison avec le champ `last_set` de la structure `state` : pour un état `s`, la valeur de `s.last_set` sera égale au champ `id` de l'ensemble construit le plus récemment qui contient `s`. Si `s` n'appartient à aucun des ensembles construits jusqu'à maintenant, son champ `last_set` sera égal à `-1` (ce sera donc le cas initialement, en particulier).

Exercice 4.6 p. 12

1. Écrire une fonction `add_state` ayant la spécification suivante :

   Prototype :

   ```
   void add_state(set_t *set, state_t *s);
   ```

   Préconditions :

   - `set` est un pointeur valide vers une structure de type `set` ;
   - `set.length >= 0;`
   - `set.states` est suffisamment grand pour contenir tous les états ;
   - `s` est soit le pointeur nul, soit un pointeur vers un état de l'automate de Thompson ;
   - les états présents dans `set` sont exactement les états `x` dont le champ `last_set` est égal au champ `id` de `set` (ce qui peut inclure l'état `s`).

   Postconditions :

   - l'état `s` a été ajouté à `set` (s'il n'y était pas déjà) ;
   - tous les états accessibles depuis `s` en n'utilisant que des ε-transitions l'ont également été (à nouveau, s'ils n'y étaient pas déjà) ;
   - les champs des différents objets ont été mis à jour pour conserver les invariants.

2. Écrire une fonction `step` ayant la spécification suivante :

   Prototype :

   ```
   void step(set_t *old_set, char c, set_t *new_set);
   ```

   Préconditions :

   - `old_set` et `new_set` sont deux pointeurs valides (et non aliasés) ;
   - les tableaux `states` des deux ensembles sont suffisamment grands pour recevoir tous les états nécessaires.

   Postconditions :

   - `new_set` contient l'ensemble des états accessibles depuis les états de `old_set` en effectuant une transition étiquetée par `c`, plus éventuellement des ε-transitions ;
   - l'identifiant de `new_set` est incrémenté de une unité par rapport à celui de `old_set` (et le champ `last_set` a été mis à jour en conséquence dans les états concernés).

3. Déterminer la complexité en temps de la fonction `step`.

4. Écrire une fonction `accept` qui prend en entrée un automate et une chaîne, et renvoie `true` ou `false` suivant que le mot (la chaîne) est reconnu ou pas. À nouveau, on considérera que le mot se termine juste avant le premier caractère `'\n'` ou `'\0'` rencontré.

   ```
   bool accept(nfa_t a, char *s, set_t *s1, set_t *s2);
   ```

Remarques

- Les deux arguments supplémentaires `s1` et `s2` sont fournis pour éviter d'avoir à allouer des ensembles.
- On pourra supposer que tous les états de l'automate ont une valeur de `last_set` strictement inférieure à `s1->id` au début de l'appel, et il faudra garantir que c'est toujours le cas à la fin de l'appel.

5. Écrire une fonction `match_stream` ayant le même comportement que `match_stream_backtrack` mais utilisant la nouvelle méthode d'exécution de l'automate. On ne créera que deux `set_t` au total.

```
void match_stream(nfa_t a, FILE *in);
```

Il reste une question à régler : jusqu'à présent, nous avons toujours fait en sorte, lorsque nous écrivions du C, de respecter une politique « zéro déchet », c'est-à-dire de libérer toute la mémoire allouée avant de quitter le programme. Ici, ce n'est pas le cas : nous avons alloué chaque état de l'automate individuellement à l'aide d'un `malloc`, mais nous n'avons jamais fait les `free` qui y correspondent. D'un certain côté, ce n'est pas grave du tout : nous ne voulons libérer les états qu'immédiatement avant de quitter le programme. Dans ce cas précis, en production, on laisserait le système d'exploitation s'en charger (de manière automatique en sortie de programme), ce qui serait plus simple et marginalement plus efficace. Cependant, il y a quelque chose de problématique : si nous *voulions* libérer proprement la mémoire (par exemple parce qu'on souhaitait construire plusieurs automates successivement), nous aurions beaucoup de mal à le faire !

Exercice 4.7 p. 14

1. On propose le code suivant pour libérer les états associés à un automate fini :

```
void free_accessible_states(state_t *q){
  if (q == NULL) return;
  free_accessible(q->out1);
  free_accessible(q->out2);
  free(q);
}

void free_automaton(nfa_t a){
  free_accessible(a.start);
}
```

   Expliquer pourquoi cela ne fonctionne pas.

2. Proposer un algorithme permettant de réaliser cette opération, sans changer notre manière d'allouer les états (on ne demande pas de le programmer).

3. Une solution beaucoup plus simple est de ne pas utiliser du tout de `malloc`. Proposer une solution utilisant deux variables globales :

```
state_t* State_array;
int Nb_states;
```

   Les seules modifications à apporter sont dans `new_state` et `main`.

Solutions

Correction de l'exercice 4.2 page I

1. Se référer au cours : théorème **??**.
2. On peut économiser un état par rapport à l'automate pour e^* puisqu'on n'a pas besoin de rajouter une transition sortante depuis l'état final.

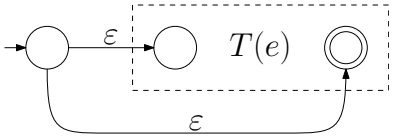

FIGURE 4.3 – $T(e?)$

3. Un état de l'automate de Thompson possède :
   - soit aucune transition sortante (uniquement pour l'état final) ;
   - soit une unique transition sortante étiquetée par une lettre ;
   - soit une unique transition sortante étiquetée par ε ;
   - soit deux transitions sortantes étiqueteés toutes deux par ε.
4. On procède par induction sur la structure arborescente de l'expression, en notant $|e|$ la longueur de sa représentation postfixe et $|T(e)|$ le nombre d'états de l'automate de Thompson.
   - L'unique cas de base est $a \in \Sigma$ (on ne considère ici ni ε ni $\emptyset$), et l'on a $|a| = 1$ et $|T(a)| = 2$.
   - On traite très rapidement les cas inductifs :
     — $|ef| = |e| + |f| + 1$ (puisque $\mathtt{postfixe}(ef) = \mathtt{postfixe}(e)\mathtt{postfixe}(f)@$) et $|T(ef)| = |T(e)| + |T(f)|$. Par hypothèse d'induction, $|T(e)| \leqslant 2|e|$ et $|T(f)| \leqslant 2|f|$, donc $|T(ef)| \leqslant 2 + 2|e| + 2|f| < 2|ef|$;
     — $\big|e|f\big| = |e| + |f| + 1$ et $\big|T(e|f)\big| = 2 + |T(e)| + |T(f)|$. L'hypothèse d'induction donne alors $\big|T(e|f)\big| \leqslant 2 + 2|e| + 2|f| = 2\big|e|f\big|$;
     — les autres cas se traitent de manière similaire, avec $|T(e^*)| = 2 + |T(e)|$ et $|T(e?)| = 1 + |T(e)|$.

Correction de l'exercice 4.3 page 3

1. Pas de difficulté.

```c
state_t *new_state(int c, state_t *out1, state_t *out2){
    state_t *state = malloc(sizeof(state_t));
    state->c = c;
    state->out1 = out1;
    state->out2 = out2;
    state->last_set = -1;
    return state;
}
```

2. Attention, on manipule des pointeurs vers des `state_t` mais directement des `nfa_t` (pas de pointeurs).

```c
nfa_t character(char c){
    state_t *final = new_state(MATCH, NULL, NULL);
    state_t *start = new_state(c, final, NULL);
    nfa_t a = {.start = start, .final = final, .n = 2};
    return a;
}
```

3.

```c
nfa_t all(void){
    state_t *final = new_state(MATCH, NULL, NULL);
    state_t *start = new_state(ALL, final, NULL);
    nfa_t a = {.start = start, .final = final, .n = 2};
    return a;
}
```

4. On crée entre zéro et deux nouveaux états suivant les cas :

```c
nfa_t concat(nfa_t a, nfa_t b){
    a.final->c = EPS;
    a.final->out1 = b.start;
    a.final = b.final;
    a.n += b.n;
    return a;
}

nfa_t alternative(nfa_t a, nfa_t b){
    state_t *start = new_state(EPS, a.start, b.start);
    state_t *final = new_state(MATCH, NULL, NULL);
    a.final->c = EPS;
    a.final->out1 = final;
    b.final->c = EPS;
    b.final->out1 = final;
    nfa_t c = {.start = start, .final = final, .n = a.n + b.n + 2};
    return c;
}

nfa_t star(nfa_t a){
    state_t *start = new_state(EPS, a.start, NULL);
    state_t *final = new_state(MATCH, NULL, NULL);
    start->out2 = final;
    a.final->c = EPS;
    a.final->out1 = final;
    a.final->out2 = a.start;
    nfa_t astar =  {.start = start, .final = final, .n = a.n + 2};
    return astar;
}

nfa_t maybe(nfa_t a){
    state_t *start = new_state(EPS, a.start, a.final);
    nfa_t amaybe =  {.start = start, .final = a.final, .n = a.n + 1};
    return amaybe;
}
```

Correction de l'exercice 4.4 page 3

1. En mettant le sommet de la pile à droite :
   - `12 4 + 3 2 ! * -`, pile vide ;
   - `4 + 3 2 ! * -`, pile `12` ;
   - `+ 3 2 ! * -`, pile `12 4` ;
   - `3 2 ! * -`, pile `16` ;
   - `2 ! * -`, pile `16 3` ;
   - `! * -`, pile `16 3 2` ;
   - `* -`, pile `16 3 4` ;
   - `-`, pile `16 12` ;
   - fin de traitement, pile `4`.
2. D'après la borne prouvée plus haut, une pile de capacité $2n$ suffit.

```c
nfa_t build(char *regex){
    int n = strlen(regex);
    stack_t *s = stack_new(2 * n);
    for (int i = 0; i < n; i++) {
        char c = regex[i];
        if (c == '@') {
            nfa_t b = pop(s);
            nfa_t a = pop(s);
            push(s, concat(a, b));
        } else if (c == '*') {
            nfa_t a = pop(s);
            push(s, star(a));
        } else if (c == '|') {
            nfa_t b = pop(s);
            nfa_t a = pop(s);
            push(s, alternative(a, b));
        } else if (c == '?') {
            nfa_t a = pop(s);
            push(s, maybe(a));
        } else if (c == '.') {
            push(s, all());
        } else {
            push(s, character(c));
        }
    }
    assert(s->length == 1);
    nfa_t result = pop(s);
    stack_free(s);
    return result;
}
```

3. La fonction `strlen` et en $O(n)$, et l'initialisation de la pile en $O(1)$ (mais $O(n)$ ne changerait rien). Ensuite, les fonctions `pop`, `append` ainsi que `concat` et autres sont en temps constant, donc la boucle principale est en $O(n)$. On obtient donc du $O(n)$ au total.

Correction de l'exercice 4.5 page 4

1. Traiter correctement le cas où `state` vaut `NULL` permet de limiter les cas particuliers, mais il faut être très attentif à l'ordre dans lequel on fait les tests.
   - On teste d'abord si l'état est nul (il ne faut pas déréférencer le pointeur dans ce cas).
   - Si on est dans un état à ε-transitions, on les suit **même si le mot est « terminé »**.
   - Sinon, si le mot est terminé, on regarde si l'état est final.
   - Sinon, on suit la transition si c'est possible (en n'oubliant pas le cas `ALL`).

```c
bool backtrack(state_t *state, char *s){
    if (state == NULL) return false;
    if (state->c == EPS) {
        return backtrack(state->out1, s) || backtrack(state->out2, s);
    }
    if (s[0] == '\0' || s[0] == '\n') return  state->c == MATCH;
    if (s[0] == state->c || state->c == ALL) {
        return backtrack(state->out1, &s[1]);
    }
    return false;
}
```

2. Le comportement souhaité est celui de `match_stream_backtrack`, il y a juste à gérer les arguments en ligne de commande.

```c
int main(int argc, char* argv[]){
    assert(argc >= 2);
    FILE *in = stdin;
    if (argc >= 3) in = fopen(argv[2], "r");
    nfa_t a = build(argv[1]);
    match_stream_backtrack(a, in);
    if (argc >= 3) fclose(in);
    return 0;
}
```

3. En supposant qu'on a compilé en un exécutable `backtrack` :

```
$ ./backtrack '.*w.*q@@q.*w@@|.*@@' francais.txt
clownesque
clownesques
squaw
squaws
wisigothique
wisigothiques
```

En compilant avec les optimisations, j'obtiens un temps d'exécution de 5 centièmes de seconde ; `grep` est environ 10 fois plus rapide sur cet exemple. On peut remarquer que la version Python est en revanche quatre fois plus lente.

4. On obtient :

```
$ time ./backtrack 'aa?@*' a25b.txt
./backtrack 'aa?@*' a25b.txt  0,00s user 0,00s system 56% cpu 0,007 total
$ time ./backtrack 'aa?@*' a30b.txt
./backtrack 'aa?@*' a30b.txt  0,03s user 0,00s system 92% cpu 0,037 total
$ time ./backtrack 'aa?@*' a35b.txt
./backtrack 'aa?@*' a35b.txt  0,35s user 0,00s system 99% cpu 0,359 total
$ time ./backtrack 'aa?@*' a40b.txt
./backtrack 'aa?@*' a40b.txt  3,82s user 0,00s system 99% cpu 3,825 total
```

Il n'y a pas énormément de points pour extrapoler, mais il semble que rajouter 5 caractères multiplie le temps de calcul par 10 environ – en tout cas, la complexité paraît exponentielle en la taille de l'entrée. On peut facilement expliquer ce phénomène :

- il y a deux états dans l'automate produit où il y a un choix à faire ;

- celui qui correspond à l'étoile ne pose pas problème, puisque sortir de l'étoile nous amène directement à un échec (on essaie de *matcher* un `b` qui n'est pas là) ;
- celui qui correspond au point d'interrogation est en revanche très problématique. Les branches de l'arbre se terminent toutes par un échec, mais elles sont de longueur comprises entre $n/2$ et n (suivant que l'on prend `aa` ou `a`), et donnent donc au moins $2^{n/2}$ feuilles !

5. `grep` semble bien se comporter :

```
$ time grep '^(aa?)*$' a40b.txt
grep '^(aa?)*$  0,00s user 0,00s system 90% cpu 0,003 total
```

Un test sur un fichier constitué de 100 000 caractères *a* suivi d'un *b* se termine aussi instantanément. En revanche, Python ne fait pas mieux que notre version :

```
$ time ./grep_py.py '^(aa?)*$' a30b.txt
./grep_py.py '^(aa?)*$' a30b.txt
0,16s user 0,00s system 98% cpu 0,166 total
$ time ./grep_py.py '^(aa?)*$' a35b.txt
./grep_py.py '^(aa?)*$' a35b.txt
1,52s user 0,01s system 99% cpu 1,537 total
$ time ./grep_py.py '^(aa?)*$' a40b.txt
./grep_py.py '^(aa?)*$' a40b.txt
16,65s user 0,01s system 99% cpu 16,668 total
```

À nouveau, l'ajout de 5 caractères semble multiplier le temps de calcul par environ 10 : Python utilise un moteur de *regex* avec *backtracking*.

6. Le programme ne termine pas sur cette entrée. C'est normal, puisque l'automate correspondant possède un cycle étiqueté par ε :

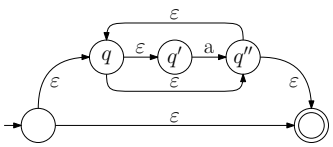

Figure 4.4 – Automate pour $(a?)^*$: il y a un cycle $q \xrightarrow{\varepsilon} q'' \xrightarrow{\varepsilon} q$.

Le problème se posera dès qu'on a une sous-expression de la forme e^*, où $\varepsilon \in \mathcal{L}(e)$.

Correction de l'exercice 4.6 page 6

1. On parcourt l'ε-fermeture avec des appels récursifs :

```
void add_state(set_t *set, state_t *s){
    if (s == NULL || s->last_set == set->id) return;
    s->last_set = set->id;
    set->states[set->length] = s;
    set->length++;
    if (s->c == EPS) {
        add_state(set, s->out1);
        add_state(set, s->out2); // s->out2 may be NULL
    }
}
```

2. Pour vider `new_set` (sur lequel on ne connaît rien), il suffit de mettre le champ `length` à

zéro ; on met aussi à jour son champ `id`, à une valeur dont on sait qu'elle est strictement supérieure à tous les `last_set`.

```c
void step(set_t *old_set, char c, set_t *new_set){
    new_set->id = old_set->id + 1;
    new_set->length = 0;
    for (int i = 0; i < old_set->length; i++){
        state_t *s = old_set->states[i];
        if (s->c == c || s->c == ALL) {
            add_state(new_set, s->out1);
        }
    }
}
```

3. Au pire, `step` appelle `add_state` sur tous les états de l'automate (et fait quelques opérations en temps constant). Le coût total est donc de $m = |Q|$, plus la somme des coûts des `add_state` sur des états non encore visités (les appels ultérieurs pouvant être « facturés » à la fonction appelante). Il y a au plus m tels appels, la complexité temporelle de `step` est donc en $O(|Q|)$.
4. Il faut juste faire attention à échanger `s1` et `s2` après chaque caractère : on respecte ainsi bien l'invariant que l'appel `step(s1, s[i], s2)` se fait avec des champs `last_set` qui sont tous inférieurs ou égaux à `s1->id`.

```c
bool accept(nfa_t a, char *s, set_t *s1, set_t *s2){
    s1->length = 0;
    add_state(s1, a.start);
    int i = 0;
    while (s[i] != '\0' && s[i] != '\n') {
        step(s1, s[i], s2);
        set_t *tmp = s1;
        s1 = s2;
        s2 = tmp;
        i++;
    }
    return a.final->last_set == s1->id;
}
```

5. Pas de difficulté particulière, il faut juste penser à incrémenter `s1->id` après chaque ligne. On aura un dépassement (qui est *UB* sur un entier signé) après $2^{31} - 1$ caractères, donc pour un fichier de plus de deux giga-octets il faudra procéder un peu différemment (ou passer `id` et `last_set` en 64 bits).

```c
void match_stream(nfa_t a, FILE *in){
    char line[MAX_LINE_LENGTH + 1];
    set_t *s1 = empty_set(a.n, 0);
    set_t *s2 = empty_set(a.n, 1);
    while (fgets(line, MAX_LINE_LENGTH, in) != NULL) {
        if (accept(a, line, s1, s2)) printf("%s", line);
        s1->id++;
    }
    set_free(s1);
    set_free(s2);
}
```

Correction de l'exercice 4.7 page 7

1. On aura un *double free* dès que deux chemins permettent d'accéder à un même état.
2. On peut garder en mémoire l'ensemble des pointeurs déjà libérés, pour ne parcourir qu'une fois chaque nœud du graphe. En supposant que l'on dispose d'une structure `pointer_set_t` (qui peut par exemple être une table de hachage) et des opérations associées, on aurait alors :

```
void free_accessible_states(state_t *q, pointer_set_t *freed){
  if (q == NULL || member(freed, q)) return;
  add(freed, q);
  free_accessible(q->out1, freed);
  free_accessible(q->out2, freed);
  free(q);
}

void free_automaton(nfa_t a){
  pointer_set_t *freed = make_empty_set();
  free_accessible(a.start, freed);
  free_set(freed);
}
```

3. On peut modifier `new_state` ainsi :

```
state_t *new_state(int c, state_t *out1, state_t *out2){
    state_t *state = &State_array[Nb_states];
    Nb_states++;
    state->c = c;
    state->out1 = out1;
    state->out2 = out2;
    state->last_set = -1;
    return state;
}
```

Et pour `main` :

```
int main(int argc, char* argv[]){
    Nb_states = 0;
    // Compute an upper bound on the number of states
    // (twice the length of the regex will do)
    State_array = malloc(upper_bound * sizeof(state_t));
    // ...
    // ...
    free(State_array);
}
```