RAPPORT DE PROJET - PROJET S4

Djibril Traore (Chef de projet)
Amine Mike EL MAALOUF
Attilio LEVIEILS
Classe: B1

Mai 2023

Contents

1 Introduction

2 Reprise du cahier des charges

2.1 Rappel du projet

Underperl est donc le fruit d'échanges approfondis à l'intérieur du groupe sur ce qu'est un logiciel utile ainsi que des discussions sur quel types d'algorithmes nous trouvons intéressants et voulons implémenter une version qui nous est propre.

Un point qui nous a tous rapproché a été notre intérêt commun pour les cours de THLR suivis au semestre dernier. Nous avons donc évoqué la piste de faire un projet en rapport avec cette matière et donc logiquement quelque chose qui utilise les expressions régulières.

Un second point d'entente a été notre appréciation du projet Abacus réalisé comme TP de programmation au deuxième semestre. Plus particulièrement, pouvoir construire quelque chose de A à Z sans se reposer sur des modules préconçus nous avait tous plu.

Ainsi, de ces deux observations le projet semblait dorénavant évident, un interpréteur d'expressions régulières qui prend en input une entrée de l'utilisateur. De cette input, nous devons en récupérer les token formant l'expression régulière (lettres ou symbole de concaténation ou autre...) puis les passer dans le "pipeline" vu en THLR qui, d'une expression régulière forme un automate qui l'accepte.

Ce programme sera mis à la disposition de l'utilisateur à travers une interface où il pourra l'utiliser pour effectuer des traitements sur des fichiers texte tel que de la recherche, de la suppression et du remplacement de paternes.

2.2 Répartition des tâches

Tâches	**Djibril**	**Amine**	**Attilio**
Site web	-	X	X
Logiciel: design/UI	X	-	-
Utilisation des RegEx	X	-	X
Implémentation algorithmique	-	X	X

Table 1: Répartition des tâches

Nous avons décidé de scinder le projets en quatre sous tâches :

- la création du site web : il faudra prendre en compte son style et les fonctionnalités présentées.

- le design de notre logiciel : ce qui comporte l'interface utilisateur avec notamment la partie inscription de l'expression régulière ainsi qu'une zone pour y ajouter des fichiers par exemple et/ou une autre pour choisir l'action qui va être exécutée (recherche, tri, remplacement,...).

- l'implémentation des programmes effectuant les actions citées précédemment, c'est-à-dire l'interprétation de l'entrée donnée par l'utilisateur en tant qu'expression régulière et l'application dans un cadre concret de ces expressions.

- l'implémentation des algorithmes permettant de créer un automate qui reconnaît une expression régulière donnée. Nous aurons besoin, comme vu dans la partie précédente, de plusieurs algorithmes qui sont interdépendants et font office de "pipeline" puisqu'avec notre implémentation, chaque output d'un algorithme sera utilisé comme input du suivant.

Cette répartition nous semble logique, proposant des tâches ayant besoin de qualités différentes:
de la création et un esprit artistique pour la partie design du site web et du logiciel ainsi qu'un mode de pensée plus abstrait lors de l'implémentation des algorithme en comparaison avec celui nécessaire à la mise en place des programmes utilisant ces algorithmes. Nous trouvons ceci adapté aux points forts dont chaque membre de l'équipe dispose.

De plus, aucunes de ces tâches ne dépendent les unes des autres, assurant une autonomie de travail pour chaque personne dans le groupe à tout moment de la réalisation du projet. Cette indépendance est primordiale à la bonne réalisation du projet. Une mise en commun sera cependant obligatoire lors de la connexion des différentes parties entre elles.

2.3 Rappel des objectifs

Taches	Sout 1		Sout 2		Sout 3	
	projection	constaté	projection	constaté	projection	constaté
Site web	30%	25%	70%	65%	100%	100%
Logiciel: design/UI	30%	20%	70%	75%	100%	100%
Utilisation des RegEx	20%	20%	70%	70%	100%	100%
Implémentation algorithmique	50%	55%	90%	95%	100%	100%

Table 2: Avancement constaté

3 Remarques sur l'avancement

3.1 Soutenance 1

Nous avions prévu une avancée homogène dans les différentes parties du projet, avec cependant une nette avance de l'implémentation algorithmique par rapport au reste.

Ce choix s'est avéré réaliste puisque nous avons vite remarqué l'étendu du travail de fond qu'il fallait réaliser.
En effet, nous nous sommes rendu compte du nombre d'outil, avec en tête de proue les structures de données, dont nous avons besoin et allons utiliser tout au long du projet et donc devoir implémenter en tout premier lieu.

3.2 Soutenance 2

Dans un second temps, il a fallu implémenter la pipeline permettant de passer d'une expression régulière à un automate déterministe en priorité.
En effet, c'est seulement après s'être assuré que cette partie était opérationnelle que l'on a pu témoigner du bon fonctionnement de la partie utilisation des expression régulière.

Notre seconde priorité pour la deuxième soutenance était de régler la partie lexer et parser de notre projet.
Après les différentes difficultés rencontrées lors de la première soutenance, nous devions nous assurer de ne pas devoir perdre du temps dessus après cette soutenance et ainsi pouvoir nous concentrer sur l'exploitation des expressions régulières.

Le site internet ainsi que l'interface restent pour des tâches moins importantes avec un intérêt qui reste principalement esthétique.
Il faut cependant noter que nous avions comme objectif de pouvoir utiliser l'interface pour s'assurer du bon fonctionnement de

chaque partie et que ces parties soient bien liées entre elles.

3.3 Soutenance 3

Comme nous avons respecté nos projections, et même dans certains cas été en avance, il n'y avait pas d'urgence pour cette dernière soutenance. Nous savions ce qui devait être fait, principalement la manipulation de fichiers ainsi que peaufiner les interfaces et nous avons pu nous organiser en conséquence pour s'assurer que cela serait fait en temps et en heure.

Pour la manipulation de fichier il était question de pouvoir modifier des fichiers en utilisant des expressions régulières. Les opérations de modifications que nous avons retenues sont la recherche de paterne dans un fichier ainsi que le remplacement de tous les mots respectant un paterne par un mot donné.

Pour l'interface utilisateur notre objectif était d'implementer les fonctionalités de manipulation de fichier décrit plus haut sur le logiciel pour les rendre tout bonnement utilisable.

Le site web devait s'enrichir de quelques ajouts pour être aux normes voulues pour cette soutenance finale.

4 Les structures

Les structures de données ont été le coeur de nos efforts lors des deux premières soutenances et nous avons dû les modifier tout au long du projet pour qu'elles répondent au mieux à nos exigences qui ont évoluées au cours du projet.

4.1 Les ensembles et dictionnaires

Les différents algorithmes qui manipulent des automates que nous voulons implémenter ont recours de nombreuses fois aux ensembles et dictionnaires, ils ont donc été les premiers à être implémenter.
En effet, il sera pratique d'avoir des structures pouvant représenter l'ensemble des sommets de départ ou l'ensemble des arcs qui composent un chemin par exemple...

4.1.1 Le hashage

L'implémentation que nous avons privilégiée pour les ensembles et les dictionnaires se repose sur celle vu lors d'un TP de programmation utilisant un table de hashage.

Il faut donc générer une clé de hashage à chaque valeur de données que nous voulons ajouter à notre ensemble.
Pour se faire nous utilisons l'algorithme de Jenkins ("Jenkins one at a time hash") que nous calculons modulo la capacité de la table de hashage pour s'assurer que le résultat peut être un index de cette table.

De plus, nous avons décidé de gérer d'éventuelles collisions primaires par liste chaînée. Ce qui nous a motivé à faire ce choix est la facilité de mise en place de ce système.

4.1.2 Les ensembles

Les ensembles sont donc représentés par une table de hashage de capacité donnée, qui double à chaque fois qu'elle atteint un seuil de remplissage supérieur ou égal à 75%, comportant un tableau de données. Ces données sont définies par leur clé de hashage, une autre clé contenant leur valeur et possède un pointeur qui, en cas de collision primaire, pointe vers une éventuelle autre donnée ayant le même index dans la table.

```
typedef struct data
{
    uint32_t hkey;
    void *key;
    struct data *next;
}data;

typedef struct set
{
    size_t len;
    size_t size;
    size_t capacity;
    struct data **elements;
}set;
```

Avec cette structure, s'accompagnent les diverses fonctions effectuant des opérations basiques sur les ensembles tel que la création d'un nouvel ensemble vide, la recherche d'une clé, l'insertion d'une donnée, sa suppression ainsi que la suppression de l'ensemble dans son intégralité mais aussi l'union entre deux ensembles.

Nous avons vite remarqué un problème quant à l'utilisation de la clé pour localiser un élément dans l'ensemble. Nous avons noté que l'élément est bien inséré à la place correspondante au hashage de sa clé, cependant, lorsque l'ensemble est agrandi, l'élément n'est plus à sa place logique correspondante au nouveau hashage de sa clé.

Cette nouvelle a été problématique, les ensembles sont très souvent agrandi (lorsqu'ils sont occupés à 75%) et donc cette relation

d'ordre si pratique et faisant la force de cette structure de données ne pouvait pas être exploité.

Ce problème était à résoudre impérativement pour les raisons cité au dessus et d'autant plus que la solution provisoire était loin d'être optimale. Nous avions choisi de parcourir séquentiellement l'ensemble lorsqu'il fallait trouver un élément, passant donc d'une complexité constante (ou linéaire en fonction du nombre d'élément ayant le même hash en cas de collision primaire) à une complexité linéaire en fonction du nombre d'éléments dans l'ensemble...

Le temps pris pour détecter où dans le code fut particulièrement long, ce bug n'apparaissait que dans des cas précis, d'autant plus que sa résolution a été facile : il fallait intervertir deux bouts de codes...

4.1.3 Les dictionnaires

La structure des dictionnaires s'apparente grandement à celle des ensembles.
En effet, cette structure de données se repose sur les mêmes principes : une table de hashage et des données disposant d'une clé de hashage qui s'insèrent dedans. Nous utilisons également les listes chainées pour résoudre les collisions primaires.
Leur implémentation a été automatique et plus faite "au cas où" que pour répondre à un réel besoin prévu.

La seule différence importante est dans la structure des données qui composent la table de hashage. Ils disposent en plus de leur clé de hashage et d'un pointeur vers une autre données s'il y a eu collision d'un champ clé et d'un champ valeur de tout type.
Ces deux champs sont ceux qui font de cette structure un dictionnaire, puisqu'ils permettent à l'utilisateur d'accéder à la valeur grâce à la clé.

```c
typedef struct pair
{
    uint32_t hkey;
    char* key;
    void* value;
    struct pair* next;
}pair;
```

Des fonctions évidentes viennent s'ajouter : la création d'un nouveau dictionnaire, la recherche par rapport à la clé, l'insertion, la suppression et la possibilité d'accéder à la valeur de la donnée grâce à la clé.

Nous avons réalisé que nous avions pas de raison d'utiliser de dictionnaire, rendant toute cette implémentation futile...

4.2 Structures de données linéaires

4.2.1 File

En informatique, une file *en anglais* ***queue***) est un type abstrait basé sur le principe " premier entré, premier sorti " ou PEPS, désigné en anglais par l'acronyme FIFO (" first in, first out ") : les premiers éléments ajoutés à la file seront les premiers à en être retirés.

Nous avons revu notre implémentation des files qui se reposaient sur un tableau et donc d'une taille qui devait être connue à l'avance. Les files nous sont utiles dans le lexer puisque les tokens obtenus après lecture de la string sont mis dans une file. Avec le sucre syntaxique, la taille de la file de sortie n'est pas forcement la même que celle de la chaîne de caractère d'entrée, nous devons donc utiliser une liste chaînée et non plus un tableau.

Pour se faire nous nous sommes appuyés de l'implémentation proposée lors de nos cours magistraux de programmation.

```
typedef struct queue
{
    struct queue* next;
    Token* data;
} Queue;
```

4.2.2 Pile

En informatique, une pile (*en anglais* ***stack***) est une structure de données fondée sur le principe " dernier arrivé, premier sorti " (en anglais LIFO pour last in, first out), ce qui veut dire qu'en général, le dernier élément ajouté à la pile est le premier à en sortir.

Les mêmes modifications ont été appliquées aux piles lors de l'avancement de notre projet.

4.2.3 Listes chaînée

Une liste chaînée ou liste liée (en anglais linked list) désigne en informatique une structure de données représentant une collection ordonnée et de taille arbitraire d'éléments de même type, dont la représentation en mémoire de l'ordinateur est une succession de cellules faites d'un contenu et d'un pointeur vers une autre cellule. De façon imagée, l'ensemble des cellules ressemble à une chaîne dont les maillons seraient les cellules.

L'accès aux éléments d'une liste se fait de manière séquentielle : chaque élément permet l'accès au suivant (contrairement au tableau dans lequel l'accès se fait de manière directe, par adressage de chaque cellule dudit tableau).

Le principe de la liste chaînée est que chaque élément possède, en plus de la donnée, un pointeur vers un élément qui lui est contigu dans la liste. L'usage d'une liste est souvent préconisé pour des raisons de rapidité de traitement, lorsque l'ordre des éléments est important et que les insertions et les suppressions d'éléments quelconques sont plus fréquentes que les accès.

En effet, les insertions en début ou fin de liste et les suppressions se font en temps constant car elles ne demandent au maximum que deux écritures. En revanche, l'accès à un élément quelconque nécessite le parcours de la liste depuis le début jusqu'à l'index de l'élément choisi.

Ces listes nous serons utiles lors de l'implémentation de notre structure d'automate.

4.3 Les arbres binaires

Dans la pipeline permettant de passer d'une chaîne de caractères représentant une expression régulière à la création d'un automate à epsilon transition, nous devons transformer cette expression en un AST.

Les AST sont, dans notre cas, des arbres binaires de syntaxes où les feuilles sont les opérandes (les lettres de notre alphabet) et les noeuds internes les opérateurs (. + * par exemple).

Nous devons donc implémenter des arbres binaires qui seront définis par leur clé, c'est-à-dire, la valeur que contient sa racine et ses deux fils respectivement child1 et child2 éventuellement nuls. Nous avons donc choisi l'implémentation la plus naïve, s'appuyant sur une définition récursive des arbres. Un arbre est :

- sois un noeud vide
- sois un neud avec deux fils (potentielment vide)

4.4 Les automates

Un automate est un dispositif reproduisant en autonomie une séquence d'actions prédéterminées sans l'intervention humaine, le système fait toujours la même chose.

Dans le domaine de l'informatique, on nomme automate une machine à traiter de l'information. Par opposition à la notion de fonction continue, cette information est de nature discrète : nombres entiers, par exemple 0 ou 1, caractères " a, b, c... "

La notion d'automate a émergé des besoins de programmation relatifs à l'analyse syntaxique : elle permettait de remplacer par des données — faciles à modifier — et un programme de cheminement unique ce qui aurait demandé un programme bien plus complexe et surtout bien plus délicat à maintenir par la suite (ce

principe a été ensuite celui des systèmes experts).

Dans le cadre de notre projet, un automate est un graphe orienté valué par les mots d'un langages. Nous avons décider d'utiliser l'implémentaion par listes d'adjacences pour notre automate, la jugeant étant la plus adaptée à notre projet. Notre Structure de donnée possède alors comme variables:

- **order**: qui représente le nombre de sommets du graphe.
- **initial_states**: qui est un ensemble contenant les états initiaux de l'automate.
- **final_states**: qui est un ensemble contenant les états finaux de l'automate.
- **alphabet**: qui est un ensemble contenant l'alphabet de l'automate.
- **adjlists**: qui contient les listes d'adjacence de l'automate.

5 Interpreteur

5.1 Lexer

En informatique, l'analyse lexicale, lexing, segmentation ou tokenization est la conversion d'une chaîne de caractères (un texte) en une liste de symboles (tokens en anglais).
Elle fait partie de la première phase de la chaîne de compilation. Ces symboles sont ensuite consommés lors de l'analyse syntaxique.
Un programme réalisant une analyse lexicale est appelé un analyseur lexical, tokenizer ou lexer. Un analyseur lexical est généralement combiné à un analyseur syntaxique pour analyser la syntaxe d'un texte.

Le lexer qui est la première étape dans l'***analyse lexicale*** convertis une chaîne de caractères, dans notre cas une expression régulière, en une liste de symboles (*tokens en anglais).*

Dans le cadre de underpearl, l'implementation ci-dessous a été retenue.

L'objectif étant de reconnaître les différents caractères importants à l'aide de l'**enum tokentype** afin de les transformer en **Token**. Le Lexer retourne finalement un **Array** contenant une file de Token et la taille de cette dernière.

```
enum tokentype
{
    backslash,
    dot,
    interogation_mark,
    star,
    open_bracket,
    close_bracket,
    pipe,
    open_parentheses,
    close_parentheses,
    add,
```

```
    space,
    other
};

typedef struct token
{
    size_t tokentype;
    size_t prioroty;
    size_t parity;
    char   symbole;
} Token;

typedef struct array
{
    size_t len;
    Token** start;
} Array;
```

5.1.1 Le sucre syntaxique

Le lexer, dans sa forme la plus simple a donc été implémenté dès la première soutenance.
La partie plus complexe, qui a été elle mise en place pour la deuxième soutenance, était d'implémenter le sucre syntaxique, c'est-à-dire des simplifications de notations permettant d'écrire des expressions régulières plus courte, facile à lire et à comprendre ainsi qu'exprimer des cas plus complexes.

Concrètement, deux améliorations étaient attendues, la concaténation implicite ainsi que l'implémentation des "brackets".

- Concaténation implicite :

Alors qu'il fallait précédemment noter toutes les concaténations de caractères à l'aide du point '.', nous voulions faire en sorte de nous en passer et la rendre implicite dans plusieurs cas :

- entre deux caractères :

$$"a.b" = "ab"$$

- entre deux parenthèses ou brackets :

$$"(ab).[cd]" = "(ab)[cd]"$$

- entre des parenthèses/brackets et des caractères :

$$"(ab).c.[de]" = "(ab)c[de]"$$

Comme vous pouvez le constater cette petite modification simplifie énormément l'écriture et la rend plus lisible.

Pour l'implémenter, nous faisons attention au token précédemment lu dans et dans le cas où celui courant est de même type que ceux dans l'un des trois cas énuméré au-dessus un token '.' point est inséré entre le précédent et le courant.

- Les brackets :

Un autre outil très puissant des expression régulières est l'utilisation des brackets '[]'. Ces dernières permettent de définir un ensemble de caractère à l'intérieur et l'expression est validée si l'un de ces caractère est présent.
Ainsi, par exemple [abc] veut dire soit a soit b soit c.

La deuxième propriété des brackets qui les rend très intéressantes à utiliser est la possibilité d'exprimer des intervalles de caractères. Ainsi, par exemple [a-z] veut dire une lettre minuscule (de a à z).

Pour l'implémenter nous parcourons le contenu des brackets et nous ajoutons des '+' entre chaque caractères dans le premier cas ou des '+' ainsi que les caractères compris dans l'interval dans le second cas.

$$[0-5] = 0 + 1 + 2 + 3 + 4 + 5.$$

5.2 Parser

Le parser, qui la seconde étape dans l'***analyse lexicale*** est un processus permettant de structurer une représentation linéaire en accord avec une grammaire donnée. Il s'applique à des domaines aussi variés que le langage naturel, le langage informatique ou aux structures de données.

Dans le cas d'underpearl, le parser va prendre en entrée la sortie du lexer pour retourner un arbre de syntaxe abstraire ou plus communément appelé **AST**.
Pour une meilleur visibilité du code mais également pour faciliter le debuggage nous avons divisé le parser en deux parties:

- Algorithme de Shunting-Yard
- Formation de l'AST

5.2.1 Shunting-Yard

L'algorithme Shunting-yard est une méthode d'analyse syntaxique (*parser*) d'une expression mathématique, logique ou les deux exprimée en notation algébrique parenthésée inventé par Edsger Dijkstra. Il peut être utilisé pour traduire l'expression en notation polonaise inverse ou **RPN**, ou en arbre syntaxique abstrait. Pour les raisons citées précédemment nous préférons passer par RPN avant d'obtenir l'AST. La RPN permet d'écrire de façon non ambiguë les formules arithmétiques sans utiliser de parenthèses.

Input	Output
3 + 4 × 2 ÷(1 - 5) ^2 ^3	3 4 2 × 1 5 - 2 3 ^^÷+

Table 3: expression arithmétique en notation polonaise inversée

Notre version initiale de l'algorithme de Shunting-Yard lors de la première soutenance ne traitait pas les parenthèses, limitant

donc les expressions régulières que nous pouvions manipuler.

Cette modification facile mais essentielle a été faite pour la deuxième soutenance avec également la détection d'erreur en cas de mauvais parenthésage de l'expression régulière.

5.2.2 AST

Du résultat obtenu par l'algorithme de Shunting-Yard, nous voulons avoir un arbre de syntaxe, comme dis précédement nous voulons avoir uns structure de donnée qui est un arbre binaire avec comme feuilles les opérandes et dans les noeuds internes les opérateurs.

Nous pouvons être sûrs qu'il s'agira d'un arbre binaire puisque tous nos opérateurs nécessites au plus deux opérandes (seul l'étoile de Kleene n'en nécessite que d'une), ce qui nous facilite grandement le travail.

Ce passage de la pile de résultat de Shunting-Yard à un arbre de syntaxe est crucial puisque l'algorithme de Thompson, dans la forme retenue pour le projet qui est celles s'apparentant le plus à celle implémentée en Python lors de la THLR au S3, peut construire un automate qu'à partir d'un AST

L'AST étant opérationel à 100% dès la première soutenance, ce fût l'un des rares morceaux du projet qui ne s'est pas vu être modifié au cours de l'avancement du projet.

6 Les algorithmes sur les automates

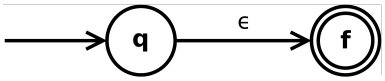
6.1 l'algorithme de Thompson

6.1.1 Principe

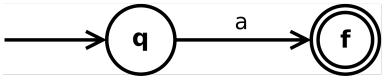
Le théorème de Kleene affirme que l'ensemble des langages rationnels sur un alphabet **A** est exactement l'ensemble des langages sur **A** reconnaissables par automate fini. Il existe des algorithmes pour passer de l'un à l'autre. L'algorithme de Thompson permet d'aller de l'expression à l'automate.

L'algorithme consiste à construire l'automate petit à petit, en utilisant des constructions pour l'union, l'étoile et la concaténation. Ces constructions font apparaître des epsilon transitions qui sont ensuite éliminées. À chaque expression rationnelle **s** est associé un automate fini **N(s)**. Cet automate est construit par induction sur la structure de l'expression.
Il existe deux cas de base:

Expression ϵ

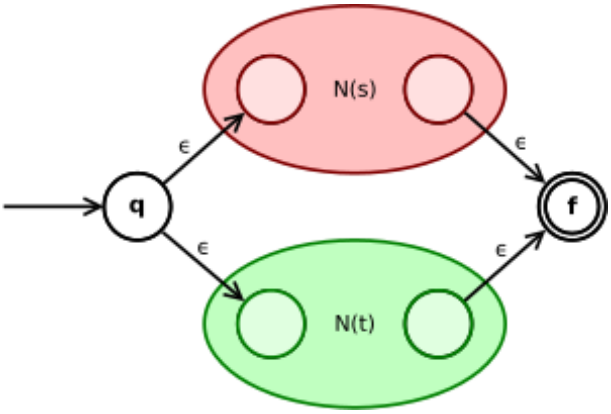


Expression a où a est une lettre

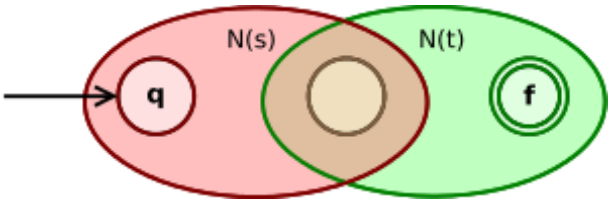


Il existe trois cas inductif:

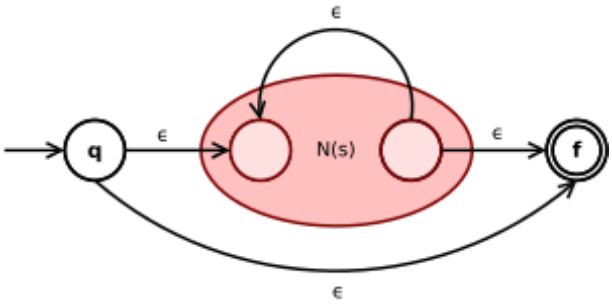
L'union



La concaténation



L'étoile de Kleene



6.1.2 Taille de l'automate résultant

On peut donner une majoration de la taille de l'automate en fonction de la taille de l'expression. La taille —e— d'une l'expression e est mesurée par le nombre de symboles qui y figurent, à l'exception des parenthèses. Donc

$$|\emptyset| = |\varepsilon| = |a| = 1,\ |s+t| = |s \cdot t| = |s| + |t| + 1,\ |s^*| = 1 + |s|.$$

Notons n(s) le nombre d'états de l'automate N(s). Alors

$$n(\emptyset) = n(\varepsilon) = n(a) = 2,$$

$$n(s+t) = n(s) + n(t) + 2,\ n(s \cdot t) = n(s) + n(t) - 1,$$

$$n(s^*) = 2 + n(s).$$

Dans tous les cas, on a donc

$$n(s) \leq 2|s|,$$

en d'autres termes, le nombre d'états est au plus deux fois la taille de l'expression.

Pour le nombre de transitions, un argument encore plus simple s'applique : de chaque état sortent au plus deux flèches, donc le nombre de transitions est au plus le double du nombre d'états.

6.1.3 Complexité en temps

En supposant connus les automates finis N(s) et N(t) associés à deux expressions régulières s et t, la construction de l'automate associé à l'union (s+t) ou la concaténation (s.t) s'effectue en temps constant (c'est-à-dire qui ne dépend pas de la taille des expressions régulières, ni des automates). De même pour la construction de l'automate associé à l'étoile de Kleene (s*).

Par conséquent, la construction de l'automate associé à une expression de taille n s'effectue en $\Theta(n)$.

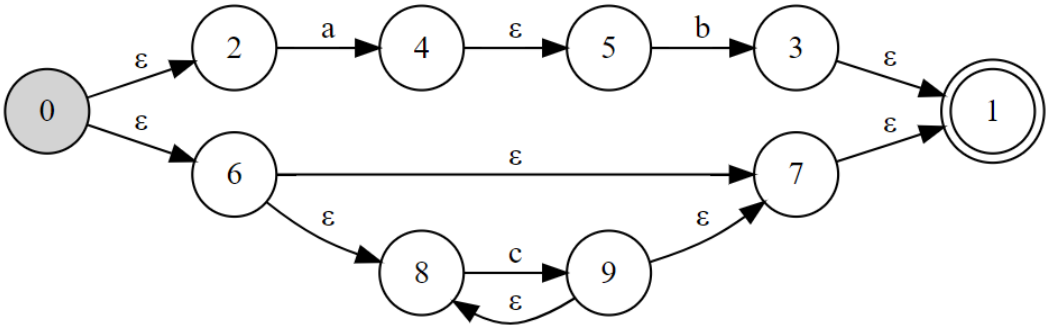


Figure 1: Automate générer par notre programme pour l'expression régulière $a.b + c^*$

6.2 Suppression des ϵ-transition

Bien que l'algorithme de Thompson soit très efficace, il possède un point négatif, il crée des ϵ-transitions: un court mot pourrait emprunter un long chemin, sans ϵ-transition le chemin ferra exactement la taille de l'entrée.

Afin de résoudre ce problème, nous allons procéder à l'élimination arrière des ϵ-transition de l'automate:
Soit trois états de l'automate q1, q2 et q3, s'il existe un chemin de q1 à q2 en empruntant uniquement des ϵ-transitions, et qu'il existe un arcs de q2 à q3 de valeur a, alors nous ajoutons une arrête de q1 à q3 de valeur a.
De plus, si à partir d'un état q nous pouvons atteindre un état final en empruntant uniquement des ϵ-transition, alors l'état q devient un état final.
Une fois cela fait, nous retirons toutes les ϵ-transition de l'automate et nous obtenons alors un automate fini non derministe (NFA).

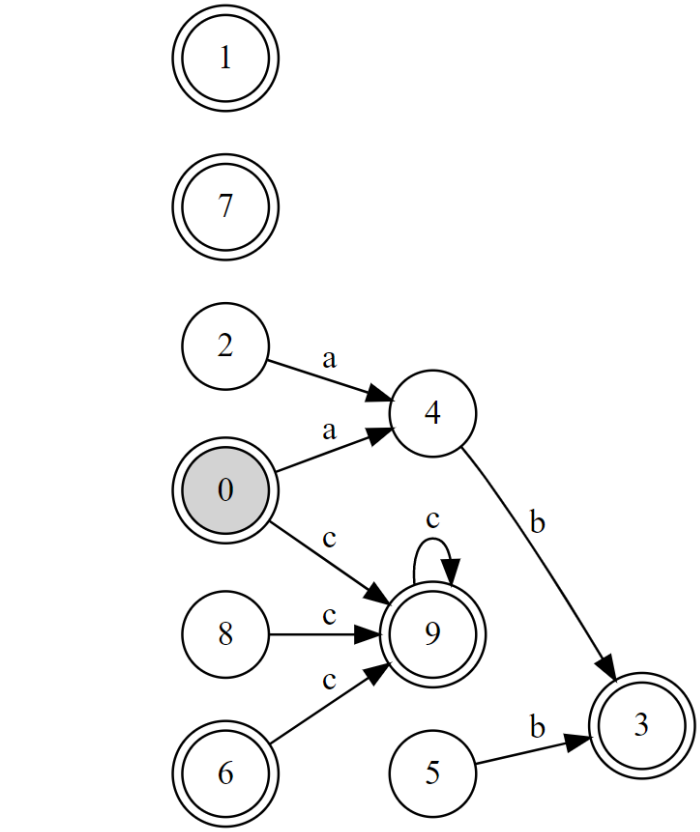


Figure 2: NFA de la regex a.b+c*

6.3 Suppression des états inutiles: Élagage

Un état est dit utile s'il est à la fois accessible et co-accessible.
Un état est dit accessible s'il existe un chemin depuis un état initial vers lui.
Un état est dit co-accessible s'il existe un chemin depuis cet état vers un état final.
Pour chaque état de l'automate, nous allons vérifier s'il est utile, s'il ne l'est pas, nous le supprimons de l'automate.

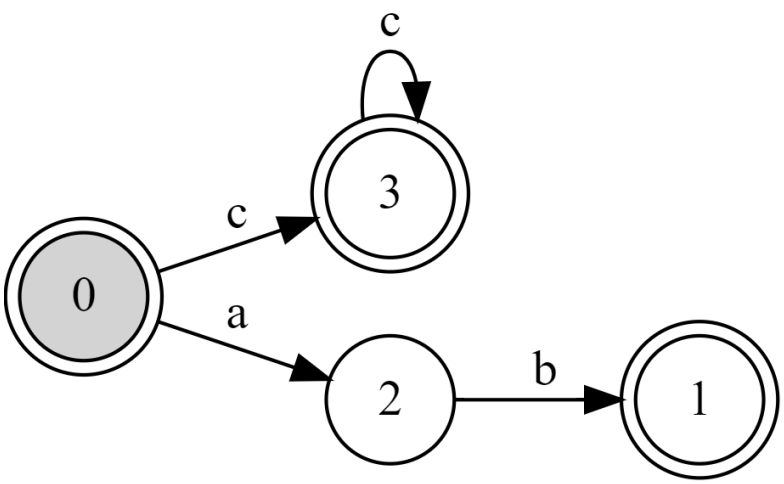


Figure 3: NFA de la regex a.b+c* après élagage

6.4 Déterminisation de l'automate

Notre automate ayant déjà vécu un long voyage, il ne lui reste plus qu'un arrêt avant le terminus: il faut le rendre déterministe.

Le problème que pose notre automate actuellement est qu'il peut exister plusieurs arcs avec la même lettre qui sortent depuis un même état. Le réel problème que cela pose est que le temps d'exécution ne sera pas linéaire: au cas où n arcs ont la même lettres, il faudra au pire parcourir les n états afin de savoir si le mot appartient au langage.

Afin de rendre le temps d'exécution linéaire, nous allons prendre le risque de potentiellement avoir une explosion exponentielle de la taille de l'automate: nous échangeons de la mémoire pour de la vitesse.

Pour l'algorithme nous allons travailler avec des ensembles d'états.

-Premièrement créons une file et nous enfilons un ensemble contenant les états initiaux de l'automate et nous créons un automate de taille 1.

-Tant que la file n'est pas vide nous la défilons.

-Pour chaque lettre de l'alphabet de l'automate, nous allons cherchez les états accessible depuis chacun des états de l'ensemble défilé, nous allons ensuite mettre ces états accessibles dans un ensemble qu'on enfilera dans la file, si l'ensemble ne possède pas déjà un état dans l'automate, nous lui en créons un.

Enfin nous ajoutons un arcs de valeur la lettre de la recherche depuis l'état enfiler en début de boucle vers l'état que nous venons d'enfiler.

Après l'exécution de cet algorithme, nous obtenons enfin un automate fini déterministe (DFA).

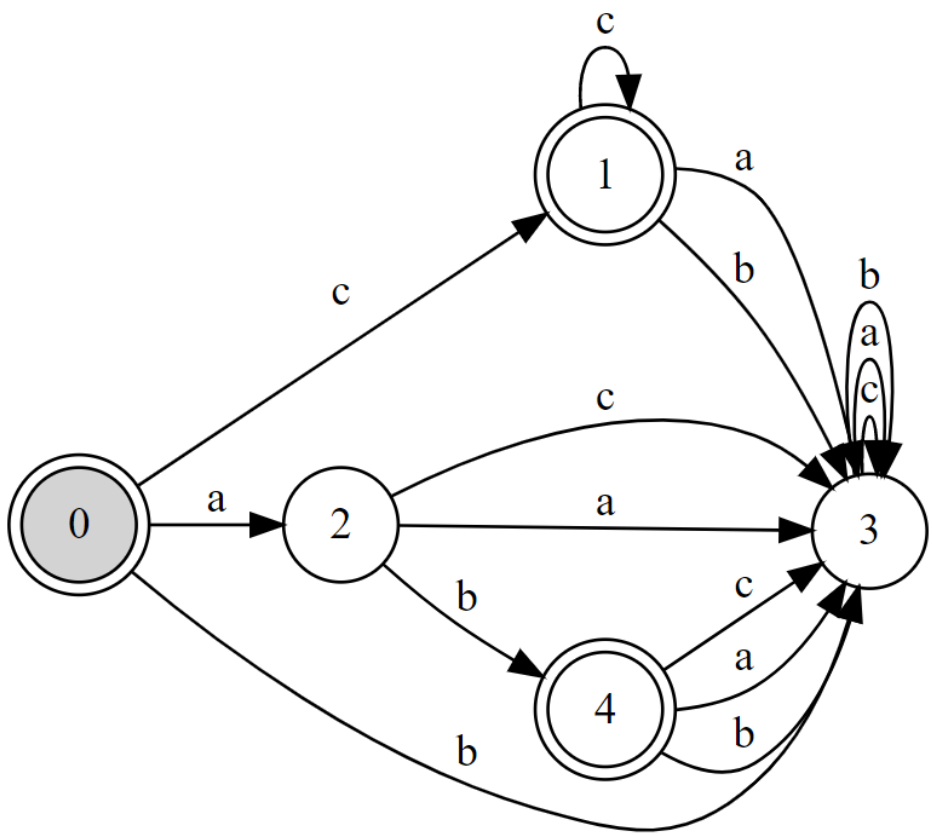


Figure 4: DFA de la regex a.b+c*

7 Vérification de l'appartenance d'un mot au langage de la regex

Arrive enfin l'étape tant attendue: comment savoir si un mot appartient au langage de l'expression régulière. Pour commencer, nous allons créer un AST à partir de l'expression régulière et nous allons appliquer notre pipeline pour construire un automate fini déterministe (DFA): Construction de l'automate grâce à Thompson –> Suppression des ϵ-transition –> Élagage -> Determinisation.

Une fois cela fait, nous allons parcourir le mot lettre par lettre. En commençant depuis l'état initial de l'automate, nous allons pour chaque lettre se diriger vers l'état où nous mène l'arc valué de cette lettre ainsi que passer à la lettre suivante du mot.
S'il n'existe pas d'arc sortant avec une lettre du mot, cela signifie que la lettre n'appartient pas au langage de l'expression régulière, donc le mot non plus.
Une fois le mot terminé si nous nous situons dans l'automate sur un état final, alors le mot appartient au langage de l'expression régulière, sinon, le mot n'appartient pas au langage de l'expression régulière.

8 Manipulation de fichiers

Cette partie est la quintessence de notre projet. En effet, les différentes structures implémentées ainsi les programmes écrits jusqu'alors, ont été créé dans l'optique de pouvoir être réutilisé et fusionné à ce moment là.

C'est cette section du projet qui se charge de réaliser l'idée de départ de UnderPerl, c'est-à-dire la manipulation de fichiers à l'aide d'expressions régulières.
Nous avions établi deux actions de modifications impératives à implémenter qui allaient de sois.

Dans un premier temps l'implémentation d'une fonction de recherche qui reconnaîtra un paterne dans un fichier texte quelconque et retournera l'indice des ligne ainsi que les mots repérés qui correspondent au paterne.
Nous avons déterminé au cours de la réalisation de la fonction que la façon la plus simple pour nous d'exploiter les résultats de cette fonction était de les sauvegarder dans un ficher.

Puis dans un second temps, celle de l'implémentation d'une fonction de remplacement qui reconnaîtra elle aussi un paterne dans un fichier texte quelconque mais qui remplacera le mot correspondant au paterne par un autre, donné par l'utilisateur du logiciel.
Le fichier sera directement modifié et l'utilisateur pourra constater des modifications apportées à son fichier.

Nous allons maintenant vous expliquer en détail par quel moyen sommes nous arrivés à implémenter ces deux fonctions pour notre projet et quelles ont été les éventuelles difficultés rencontrées lors de la réalisation de ces fonctions.

8.1 Recherche

Comme décrit plus haut, et comme le sous entend le nom de la fonction, ce que la fonction devait faire était clair.

Cependant, son implémentation en C n'a pas été aussi évidente qu'elle aurait pu paraître dû à une difficulté rencontrée : réussir à extraire les mots du fichier texte. Pour se faire nous avons établit une définition, sans doute pas parfaite, de ce que l'on considérait comme un mot.
Nous avons donc décider de qualifier de mot toute suite de caractères (ou simple caractère) se situant entre deux séparateurs. Cette définition s'appuie sur celle d'un séparateur, que nous avons défini simplement comme l'un de ces caractères suivants :

”.,;:!?/’([{}])\n”

Cela limite un peu nos expressions régulières puisqu'elles ne peuvent pas donc contenir ces symboles à part si l'expression se résume à l'un de ces séparateurs.

Une autre difficulté qui nous a fait face a été la manipulation du résultat obtenu à l'appel de la fonction de recherche.
Renvoyant le mot et le numéro de ligne de ce mot pour toutes les occurrences de mots vérifiant l'expression régulière notre fonction renvoie un nombre indéterminé de mots et de numéro de ligne.
C'est pour ce genre de problème que nous avons implémenté une liste chaînée. Cependant, il a fallu modifier légèrement la structure de la liste puisqu'elle doit contenir deux éléments de type différents : le numéro de ligne qui est un *size_t* et le mot validant l'expression régulière qui est une string *char**.

Dans un premier temps nous pensions que nous allions opter pour un affichage sur le canal de sortie du résultat dans un format choisi qui est le suivant :

line x : word

Avec x le numéro de ligne et word le mot reconnu par l'expression régulière.

Nous avions donc écrit une fonction qui parcours la liste chaînée et affiche sous ce format les éléments présent dans cette liste.
Puis, après échange avec l'équipe se chargeant de l'interface utilisateur, il a été jugé plus pratique de remplir un fichier résultat (nommé res) de ces informations pour pouvoir mieux les afficher à l'utilisateur

Nous avons donc modifié la fonction d'affichage pour qu'elle écrive dans un fichier res que nous créons. Il est évident, mais, à l'heure où ce rapport est écrit, trop tard, qu'à la lumière de ces nouvelles contraintes nous pouvions nous abstenir d'utiliser les listes chaînées puisque nous pouvions juste écrire les résultats dans le fichier res, au fur et à mesure du parcours du fichier.

8.2 Remplacement

Ici aussi la fonction est claire et simple à expliquer et imaginer son déroulement, son implémentation ne doit donc ne pas être si compliquée.

Cette assertion est vraie pour la version finale de la fonction, néanmoins la première inspiration de l'équipe chargée de l'implémenter ne s'est pas montrée judicieuse.
En effet, dans un premier temps, il a été envisagé de modifier le fichier en même temps qu'il est parcourus ligne par ligne. Chaque ligne est scannée mot par mot de la même façon que pour la fonction de recherche, cependant, lorsqu'un mot est accepté par le paterne, alors il est remplacé par le mot de substitution déterminé par l'utilisateur, la ligne est modifiée pour faire de la place à ce nouveau mot puis cette ligne est réinsérée dans le fichier texte à sa place.

Le problème dans cette implémentation est dû à une mauvaise compréhension de ce qu'est une ligne dans un fichier texte. Nous pensions que c'était une instance particulière alors que ce n'est juste qu'un ensemble de caractère qui se termine par un retour à la ligne, \n ou autre.

Le décalage effectué sur la ligne pour "faire place" au nouveau mot est donc insuffisant et doit être répété sur le fichier dans son intégralité pour chaque mot devant être remplacé.
Cette opération devient très coûteuse en complexité (d'ordre polynômial) ce qui ne nous convient pas.

Une solution trouvée a été de créer un fichier temporaire tmp et de parcourir mot par mot le fichier à modifier.
A chaque mot l'un des deux cas peut se présenter :

- le mot ne valide pas le paterne, il est donc copié tel quel dans le fichier tmp

- le mot valide le paterne, il est donc remplacé par le mot de substitution dans le fichier tmp

Lorsque le fichier est parcouru en entier il est supprimé et le fichier tmp est renommé au nom du fichier d'origine.

9 Les interfaces

9.1 L'interface utilisateur

L'interface utilisateur est un dispositif matériel ou logiciel qui permet à un usager d'interagir avec un produit informatique. C'est une interface informatique qui coordonne les interactions homme-machine, en permettant à l'usager humain de contrôler le produit et d'échanger des informations avec le produit.

Parmi les exemples d'interface utilisateur figurent les aspects interactifs des systèmes d'exploitation informatiques, des logiciels informatiques, des smartphones et, dans le domaine du design industriel, les commandes des opérateurs de machines lourdes et les commandes de processus.

9.1.1 Outils utilisés

Afin de présenter notre projet avec une interface graphique intuitive et facile d'utilisation, nous avons décidé d'utiliser **GTK** qui est un ensemble de bibliothèques logicielles pour interfaces graphiques.

De plus, dans le but de nous faciliter le travail mais aussi de mieux maitriser le rendu final, nous avons utilisé **Glade Interface Designer** qui est un outil interactif de conception d'interface graphique GTK+ générant un fichier *XML* qui servira de base dans l'utilisation de GTK.

9.1.2 Les étapes

La production de l'interface utilisateur aka UI, peut être divisée en 3 majeures:

1. Création d'un visuel comme dit ci-dessus, intuitif et facile d'utilisation ainsi que la liste des des fonctionnalites nécessaires en prenant en compte notre avancement.

2. Implementation d'une version basique fonctionnelle.
3. Amélioration du produit et fusion de toutes les parties dans l'interface graphique.

Comme précisé dans le second rapport de soutenance, l'ensemble des étapes avait été réalisé pour cette dernière et l'objectif pour cette dernière soutenance était de rendre l'UI plus facile d'utilisation tout en évidemment ajoutant les fonctionnalités supplémentaires.

Afin de rendre un produit facile d'accès et facile à prendre en main nous avons décidé dans un premier temps de diviser la zone d'entrée de texte en deux afin de pouvoir différentier l'expression régulière du mot qu'on souhaite étudier.

Dans un second temps nous avons fait en sorte de pouvoir récupérer les précédentes saisies de chacunes des zones de textes pour ne pas avoir à tout réécrire à chaque fois.

Enfin nous avons fait en sorte qu'un bouton soit utilisable seulement si toute les donnés nécessaires à son utilisation sont saisies. En ce qui concerne les ajouts de fonctionnalités il est maintenant possible d'importer un fichier et de faire une recherche ou une modification à partir d'une expression régulière.

Figure 5: Page d'acceuil

Figure 6: Page principale

9.1.3 Implémentation

"L'implémentation à évoluer au fil du projet tout en gardant la même base. L'objectif étant " d'avoir un code pas trop complexe à comprendre et surtout afin de ne pas se perdre en le lisant, nous avons instantanément pensé à créer des struct et c'est principalement ces derniers qui risque d'évoluer.

Etant donnée que nous avons deux pages diffrentes nous avons donc creér 2 struct. Le premier struct Page qui contient les elements d'une page, ces derniers varient d'une page à l'autre, et le second correspond à l'interface utilisateur dans l'ensemble et contient notement les struct de chaques pages.
L'objectif étant d'avoir accés à toute données utile facilement.

```
typedef struct Page
{
    GtkWindow *window;
    GtkButton *start_button;
    GtkButton *close_button;
    GtkImage *image_area;
    GtkTextView *fileView_area;
    GtkButton *execute_button;
    GtkEntry *command_entry;
    GtkEntry *word_entry;
    GtkButton *last_word;
    GtkButton *last_command;
    GtkButton *next_word;
    GtkToggleButton *find_button;
    GtkToggleButton *replace_button;
    GtkButton *next_command;
    GtkFileChooserButton *file_input;
} Page;


typedef struct UserInterface
{
    GtkWindow *current_window;
    Page *first_page;
    Page *second_page;
    Page *third_page;
    GtkImage *image_area;
    GdkPixbuf *pixbuf;
    GtkEntry *command_entry;
    GtkEntry *word_entry;
    GtkButton *execute_button;
    GtkTextView *fileView_area;
    GtkFileChooserButton *file_input;
```

```
    vector *command_histo;
    vector *word_histo;
    size_t command_i;
    size_t word_i;
    GtkButton *last_word;
    GtkButton *last_command;
    GtkButton *next_word;
    GtkButton *next_command;
    GtkToggleButton *find_button;
    GtkToggleButton *replace_button;
    gchar *filename;
    int f;
    int r;


} UserInterface;
```

Un point positif et que nous avons fortement apprécié avec l'utilisation de Glade est que nous n'avions pas gérer la création de boutton ou de ***GtkWidget*** plus généralement mais uniquement les signaux associés avec la fonction ***g_signal_connect()***.

En effet, nous pouvons récupérer les *GtkWidget* via un ***GtkBuilder*** qui est initalisé à partir du fichier *XML* généré par Glade. Ceci a grandement simplifié le code aussi bien pour nous que pour les potentiels personnes qui le liront.

Dans la même optique, nous avons décidé de travailler sur plusieurs fichiers. Les trois principaux sont les fichiers *utils.c*, *utils.h* et le *main.c* qui contiennent respectivement:

- La majorité des fonctions utilisées, notement celles executées par l'activation d'un signal avec ***G_CALLBACK()***.
- La déclaration des struct et des fonctions utilisées.
- La fonction *main* qui contient l'initialisation du *GtkBuilder* énoncé précédememt et par conséquent des *GtkWidget* et des signaux.

9.2 Le site web

Un site web, site Web ou simplement site, est un ensemble de pages web et de ressources reliées par des hyperliens, défini et accessible par une adresse web. Un site est développé à l'aide de langages de programmation web, puis hébergé sur un serveur web accessible via le réseau mondial Internet, un intranet local, ou n'importe quel autre réseau, tel que le réseau Tor.

Comme nous l'avions préciser dans notre cahier des charges, le développement d'un site web est tâche inédite pour les membres de notre groupe et la tâche s'est avérée être plus dure que prevue...

Une première période de projet fut une période expérimentale qui s'est avéré fructueuse, nous avons pu découvrir des outils tels que VS Code et son extension Live Server qui nous a facilité la tâche tout au long du développement du site web en nous rendant alors encore plus productif. Live Server permet de voir en temps réel les changements apportés à notre site web grâce à n'importe quel navigateur. Cet outil nous permet de réaliser un gain de temps non négligeable.

Une première avancée du site web a été de créer les premières maquettes des pages de notre site.
Avoir notre index.html sur Github nous permet d'utiliser leur service d'hébergement qui est gratuit, et c'est comme ça que nous avons pu mettre en ligne notre site web.

Alors que nous avions plutôt bien commencé nos avancées sur le site web en arrivant à le mettre en ligne rapidement et en trouvant des outils comme Live Server qui s'annonçaient comme être très utiles pour la suite, nous avons pris du retard lors de la deuxième soutenance par rapport aux projections de début de projet.

Ce qui est d'autant plus frustrant est que ce retard ne peut pas

s'expliquer par un manque de travail. En effet, l'équipe chargé du site a passé un nombre non négligeable d'heures à effectuer différentes maquettes pour le rendu esthétique du site web mais aucunes ne nous a convaincues.
Nous sommes arrivés à la réalisation que CSS et HTML sont plus durs à prendre en main et la complexité de ces langages a peut-être été sous estimé au début du projet.

Des changements notables ont cependant été apportés lors de la deuxième soutenance : la mise en place d'un bandeau pour cacher le reste du temps les boutons qui n'étaient pas au goût de l'équipe design ainsi que l'apparition de contenus sur les différentes pages du site.
Les utilisateurs pourront désormais s'informer sur ce qu'est UnderPerl, connaître un peu plus l'équipe derrière le projet ainsi qu'apprendre ce qu'est une expression régulière et comment la manipuler.

Le développement de l'application Underperl ayant été un succès, notre équipe ne pouvait pas se permettre de ne pas avoir un site web dans les règles de l'art. C'est pour cela que nous avons consacrés une grande partie de notre travail à l'amélioration de notre site web.

Notre bandeau de sélection des pages étant très rudimentaire, nous avons complètement réimaginer sa conception et à présent nous possédons un menu des pages beaucoup plus moderne, ce changement à lui tout seul rendit notre site beaucoup plus attirant et épuré, nous donnant alors plus de crédibilité aux yeux des éventuels utilisateurs.

Souhaitant faciliter la communication entre les utilisateurs et notre équipe, nous avons ajouter à notre site web un espace contact où les utilisateurs ou les éventuels utilisateurs pourront partager leurs problèmes et leurs questions avec les membres de

l'équipe.

Afin de partager notre expérience et notre parcours au monde, nous avons ajouter à notre site la possibilité de télécharger notre cahier des charges ainsi que nos rapports de soutenances. Ayant travailler durement sur une période de cinq mois, nous espérons que notre travail et nos recherches pourrons venir en aide aux autres.

10 Point de vu de l'équipe

10.1 Nos impressions

10.1.1 TRAORE Djibril

Lors de ce semestre j'ai été avec mon équipe impliqué dans ce projet passionnant qui a suscité en moi un sentiment extrêmement positif même si à l'origine je n'étais pas spécialement emballé par ce dernier.

Travailler en équipe avec mes camarades de classe motivés et talentueux a été une expérience enrichissante. Nous avons pu mettre en pratique nos compétences techniques, renforcer notre collaboration et découvrir de nouvelles solutions innovantes.

Malgré les défis rencontrés tout au long du projet, la satisfaction de voir nos efforts porter leurs fruits et de constater les résultats concrets de notre travail a été extrêmement gratifiante.

Ce projet m'a permis de développer mes compétences en résolution de problèmes, en communication et étant chef de groupe en leadership. Ce projet m'a également permis de mieux visualiser et surtout de concrétiser nos cours théoriques de Théorie des Languages Rationnel du S3 en faisant d'underpearl une application directe de ce cours.

Je suis d'autant plus satisfait de ce projet que c'est personnellement le premier que je finis entièrement et dont je suis complètement satisfait du résultat. J'espère continuer dans cette lancée lors de mes prochaines années à l'EPITA.

10.1.2 Attilio Levieils

Cette période de travail s'est avérée intense mais je réalise déjà les compétences qu'elle m'a permis d'améliorer.

Elle a été intense et a eu des moments assez frustrants lorsqu'il a été question d'essayer de régler les différents bugs hérités des soutenances précédentes .

Le travail sur le lexer/parser m'a particulièrement plus puisqu'il s'apparente au projet Abacus de l'année dernière et qui a été une motivation au commencement du projet.

Puis j'ai dû m'occuper du développement Web pour le site du projet ce qui demande un tout autre arsenal de qualité, que je dois assimiler et que je réalise plus dur à maîtriser que ce que je pensais.

L'implémentation des fonction de recherche et de remplacement a été un challenge différent puisqu'il était désormais question de manipulation de fichiers, ce qui a été déroutant au début. J'ai aimé travailler dessus car cela m'a permis d'enlever une grosse zone d'ombre que j'avais quant à la programmation.

Je suis content de la qualité de notre rendu final, nous avons bien avancé, de manière régulière et sans prendre (trop) de retard et je pense que cela s'est ressenti sur la qualité finale du projet. De plus cela m'a permis une bonne fois pour toute de bien maitriser les pointeurs ou les structs et, grâce à la création du site web, m'apprend de nouveaux langages et outils.

10.1.3 Amine Mike El Maalouf

Ce projet fut une opportunité exceptionnelle pour moi, j'ai eus la chance de travailler dans une excellente équipe qui se serrait les coudes. La belle ambiance dans le groupe à causer une entre aide

mutuelle, poussant chacun des membres du groupes à s'améliorer.

Ayant apprécier les cours de THLR, l'opportunité d'implémenter mes acquis de moi même sans aide extérieure m'a vraiment motiver, de plus, le fait que le projet soit rédiger en C m'a permis d'attaquer le problème d'un angle différent que celui vue en cours, poussant alors ma connaissance sur les graphes et les automates à grandement s'enrichir.

Je suis extrêmement heureux des progrès que se soit en C, en Makefile, ou même en développement web que j'ai pu constater, un bagage qui je suis sur me sera très utile à l'avenir que se soit dans la suite de mes études ou même dans ma carrière professionnelle.

Pour conclure, je peut fièrement dire que ce projet fut l'une des meilleures expériences lors de mes études, ayant était capable de grandir main dans la main avec les autres membres du groupes.

11 Conclusion

L'équipe de MatchMakers est heureuse et fière de vous présenter la version finale de Underperl !
Chaque membre du groupe s'est prouvé être impliqué, motivé et a cru en la qualité de ce projet tout au long de sa réalisation. Nous avons l'intime conviction que Underperl pourra être un outil réellement utile.

Nous avons tous aimé travailler sur ce projet de par notre intérêt propre et personnel et nous avons tous réalisé les progrès en programmation que ce projet nous a apporté.

De plus, le recours à différents outils que nous connaissions pas ou peu, tel que GTK, HTML, CSS et tant d'autres, nous a permis de nous challenger et de nous apporter des compétences qui nous seront utiles tout au long de notre cursus dans l'école ainsi que dans nos futur métiers.

Enfin, le travail de groupe s'est particulièrement bien passé et nous a permis de créer de vrais liens entre les différents membres du groupe. Cette bonne entente a été motrice et est responsable de la réussite qu'est Underperl.
Pour conclure, l'équipe de MatchMakers espère que vous profiterez de Underperl et reconnaîtrait que cela a été un projet cohérent, de qualité et répondant aux exigences de son cahier des charges fixés en début d'année calendaire.

12 Annexe

L'algorithme de Thompson
https://en.wikipedia.org/wiki/Thompson%27s_construction

Élimination des ϵ-transition
https://en.wikipedia.org/wiki/Nondeterministic_finite_automaton

Élagage de l'automate
https://en.wikipedia.org/wiki/DFA_minimization

Déterminisation de l'automate
https://en.wikipedia.org/wiki/Deterministic_finite_automaton

Minimisation avec l'algorithme de Moore
https://en.wikipedia.org/wiki/DFA_minimization