

Institut Supérieur d'Électronique de Paris
Projet de Fin d'Études

Reponsable: M. Hugueney

Finite State Transducers Just-In-Time Compiling

Do you hear the bytecode ?

Émilien Boulben
Victor Delepine
Corentin Peuvrel

24 juin 2015
Paris

Table des matières

Introduction	1
1 Analyse préalable	2
1.1 Le projet en détail	2
1.2 Premières études	2
1.3 Commencer quelque part	2
2 Les premiers essais	4
2.1 Générer du C en bash	4
2.1.1 Le jeu de données	4
2.1.2 Fonctionnement	4
2.1.3 Les avantages	5
2.2 Générer de l'asmX86 en bash	5
2.3 Adapter le travail précédent en Java	7
2.4 Générer une structure de switch imbriqués en Java	7
3 Générer une FST	8
3.1 Le besoin	8
3.2 Fonctionnement de l'algorithme	8
3.3 Difficultés	8
3.4 Résultat	9
4 Idées d'amélioration et difficultés insurmontables	10
4.1 Idées originelles	10
4.1.1 L'encodage	10
4.1.2 Le traitement d'un texte en multithreading	10
4.2 Les difficultés les plus grandes	10
4.2.1 Compilation à la volée	10
4.2.2 Méthodes Java et la limitation 64Ko	10
4.2.3 Le bytecode et la librairie asm	11
5 Analyse	14
A Annexe : tests avec un script shell	15
A.1 Dictionnaires	15
A.2 FST	16
A.3 Générer à la volée du code C	17
A.3.1 Script shell	17
A.3.2 Code C généré pour la FST définie dans le Tableau 5	19
A.3.3 Code C généré pour la FST définie dans le Tableau 6	21
A.4 Générer à la volée du code assembleur x86	23
A.4.1 Script shell	23
A.4.2 Code assembleur x86 généré pour la FST défini dans le Tableau 5	27
A.4.3 Code assembleur x86 généré pour la FST défini dans le Tableau 6	31
B Annexe : générer du code java dynamiquement	34
B.1 Dans une seule méthode	34
B.1.1 Avec différents états représentés par des méthodes	34
B.1.2 Uniquement avec des switch	37
B.2 Dans différentes méthodes	39

Listings

1	Un exemple de code généré pour un état	5
2	Test sur la taille du token	12
3	Script pour générer un code C à la volée d'une FST	17
4	Code C généré pour la FST définie dans Tableau 5	19
5	Code C généré pour la FST définie dans Tableau 6	21
6	Script pour générer un code assembleur x86 à la colée d'une FST	23
7	Code assembleur x86 généré pour la FST définie dans Tableau 5	27
8	Code assembleur x86 généré pour la FST définie dans Tableau 6	31
9	Code Java générant une structure avec des nœuds	34
10	Code Java générant une structure switch	37
11	Code Java générant une structure switch avec fonctions	39

Liste des tableaux

1	Exemple de dictionnaire	8
2	Dictionnaire utilisé pour tester l'algorithme	9
3	Dictionnaire à utiliser avec la FST dans le Tableau 5	15
4	Dictionnaire à utiliser avec la FST dans le Tableau 6	15
5	FST utilisée avec le dictionnaire Tableau 3, voir Figure 2 page 16	16
6	FST utilisée avec le dictionnaire Tableau 4, voir Figure 3 page 16	16

Table des figures

1	La FST associée avec le Tableau 2 générée par l'algorithme	9
2	La FST associée avec le Tableau 5 page 16	16
3	La FST associée avec le Tableau 6 page 16	16

Introduction

Les FST – Finite State Transducers ou en français Transducteur fini – sont des automates finis particuliers puisque possédant une sortie. Ils sont énormément utilisés par les moteurs de recherche puisqu'ils permettent d'associer à un mot une valeur numérique unique. C'est l'indexation.

Nous savons donc déjà transformer le texte en valeur exploitable ensuite (par comparaison avec un dictionnaire par exemple). Mais de part les quantités non négligeable de texte à analyser, et ce le plus rapidement possible, il est essentiel de continuer les recherches afin de réussir à optimiser cette transformation.

Seulement en ingénierie l'optimisation n'est pas tout, il faut aussi pouvoir simplement déployer les solutions sur différents serveurs : une solution trop complexe même performante sera très coûteuse à long termes et donc probablement pas choisie. C'est alors qu'intervient ce projet.

Il s'agit de faire une étude sur la faisabilité et la pertinence d'une nouvelle manière d'indexer le texte en java, L'objectif n'est pas de faire mieux que la référence qu'est Lucène, mais de déterminer s'il est possible de faire mieux en utilisant cette méthode.

Nous allons avec ce projet chercher à déterminer si un interpréteur de FST reposant sur la compilation à la volée est viable en java.

1 Analyse préalable

1.1 Le projet en détail

Pour pallier à des problèmes de performance et de distribution de la solution lors de l'indexation d'un texte en suivant une FST il est important de penser à de nouvelles solutions qui pourront éventuellement challenger la librairie de référence sur ce sujet : Lucène. L'objectif n'est pas d'y parvenir, mais de déterminer si cela est possible avec une solution qui a été imaginée par M. Hugueney et M. Marty.

Dans Lucène lors de la création d'une FST une structure de données est stockée en ram et le texte la parcourt pour connaître le résultat. Afin de gagner en performance en termes de temps d'exécution lors de cette étape, ne serait-il pas mieux de pouvoir avoir un code simple mais conséquent en taille qui soit parfaitement adapté à la FST désirée ? Finalement, plutôt que créer une structure générique, générer un code dédié à la FST en étude et l'optimiser au mieux en bafouant tous les principes de développement afin de gagner en temps d'exécution. La lisibilité est sacrifiée mais ce n'est pas très important compte tenu que ce code n'a pas destination à être lu, seulement compilé puis parcouru.

L'objectif du projet est de réussir à créer un programme Java qui génèrerait le code correspondant à une FST donnée pour pouvoir parser du texte efficacement, d'abord en Java puis en bytecode. Ensuite, faire une étude des performances et si possible les comparer avec les outils déjà existant. Il est très important de documenter les difficultés rencontrées puisque l'objectif reste de pouvoir se prononcer sur la faisabilité ou l'utilité d'un tel produit.

1.2 Premières études

Il nous a été très compliqué de comprendre ce qu'était une FST. Nous avons beaucoup investi de temps à combler ce manque avec des résultats très mitigés. Il était très difficile avec toute la documentation disponible de savoir par quoi commencer, surtout que souvent pour comprendre certains concepts il nous fallut assimiler ce que les explications considéraient acquis.

Cette incompréhension du sujet fût à l'origine de bien des découragements, et il n'a pas toujours été facile de nous remotiver les uns et les autres. Finalement c'est la pratique qui nous a apporté le plus de réponse.

Une FST qui permette l'indexation de texte, qu'est-ce ? Une simple structure de donnée. Un tableau (voir Tableau 5 et Tableau 6) ou un graphe (voir Figure 2 et Figure 3 peut la représenter). Elle est construite par un algorithme à partir d'un dictionnaire qui associe à des mots des valeurs. Enfin cela permet de parser du texte lettre à lettre (voir mots à mots) rapidement et en parallèle.

Une fois cette base essentielle partiellement comprise nous avons pris le courage de nous lancer dans le code.

1.3 Commencer quelque part

Commencer à coder paraît simple et pourtant nous y avons rencontré de trop nombreuses difficultés. Nous nous sommes heurtés à des nombreuses reprises à notre méconnaissance des FST, et n'arrivions pas à dégager un cas simple sur lequel travailler et monter en compétence.

Nous avons aussi perdu du temps à partir sur du code inutile à ce moment du développement : algorithmes de création d'une FST, tests en bytecode... Alors que ce n'était pas la priorité à ce moment.

C'est après une pause dans le projet que nous avons pu le reprendre d'un regard nouveau et l'aborder avec un outil que nous maîtrisons mieux pour générer du texte : bash. Par un découragement général nous avons presque involontairement trouvé ce dont nous avions besoin pour nous lancer avec efficacité dans le projet : un appui solide mais rapide à construire.

2 Les premiers essais

2.1 Générer du C en bash

L'objectif ici n'était pas de réussir à faire quelque chose de fonctionnel, mais de comprendre ce qu'il nous fallait faire. Pour ce faire nous avons finalement décidé de le faire avec les langages que nous maîtrisons le plus et que nous jugions les plus adaptés pour la situation : le bash et le C.

2.1.1 Le jeu de données

Nous avons créé manuellement des FST très basique, reliées à un dictionnaire, afin de disposer d'un jeu de test. Ils se trouvent en Appendice A.

Nous avons légèrement adapter le format défini par AT&T pour décrire des FST dans un fichier texte. Ce jeu de données servira pendant tout le projet en étant réadapté en Java par la suite.

2.1.2 Fonctionnement

Dans ce premier test nous prenons en entrée du script le fichier texte décrivant la FST, puis générons un code C qui permette de parcourir cette FST. Dans ce nouveau code point d'algorithme complexe : simplicité et naïveté sont ici ce que nous cherchons. Nous espérons alors que la pratique nous permettra de mieux comprendre le sujet. De plus nous faisons confiance à gcc pour optimiser le code à la compilation.

La première étape pour construire ce script est bien sûr de prévoir la forme qu'aura le code C généré : nous avons donc concentré nos premiers efforts à la génération d'une fonction contenant de multiples labels correspondant chacun à un état (ou nœud) et un switch qui, suivant la lettre courante se déplace à la lettre suivante grâce à un goto qui pointe sur le label du state/node suivant. Nous returnons un code d'erreur si le token d'entrée n'est pas compatible avec la FST (le mot n'est pas pris en compte par celle-ci et n'a pas de code associé).

Cette fonction prend comme seul paramètre d'entrée une chaîne (tableau de char) – qui sera le token d'entrée dont on veut connaître la valeur – et retourne le poids cumulé de tous les arcs traversés, ou -1 en cas d'erreur. Il faut remarquer qu'avec cette méthode on ne peut pas supporter de poids négatifs, au risque d'avoir une collision entre le code d'erreur et un poids cumulé effectivement négatif. Pour gérer ce cas, le plus simple serait d'utiliser `errno`.

Le code a été un peu moins simple que prévu pour pouvoir générer du C valide, en effet, il y a un certain nombre de cas particuliers à prendre en compte afin de gérer correctement les erreurs ou de multiples états de fins.

Au final, un état générera un code semblable à celui présent dans le Listing 1 (pour un état appelé "7", qui possède un arc pour le caractère 'X' avec un poids de 6 et qui va au node "21", et un arc pour le caractère 'Y' avec un poids nulle et allant au node "42").

On remarque la présence d'un compteur "pos" incrémenté de manière inconditionnel, vu que l'on se déplace toujours un caractère par un caractère.

Pour pouvoir facilement lancer le programme généré, nous avons rajouté une fonction main qui appelle juste notre fonction `compute_fst` sur le premier argument de la command line, rendant ainsi le programme autonome.

```

1 NODE_7 :
2     switch(token[pos++i]) {
3         case 'X' :
4             total += 6;
5             goto NODE_21;
6         case 'Y' :
7             goto NODE_42;
8         default :
9             return -1;
10    }

```

Listing 1 – Un exemple de code généré pour un état

Il nous suffit donc, pour générer quelque chose d'utilisable de faire ceci :

```
./gen.sh file.fst | gcc -x c -o fst -
```

Puis :

```
./fst LE_TOKEN
```

Les options sur gcc (on peut aussi rajouter un -O3 pour optimiser au maximum la compilation) servant seulement de prendre l'entrée standard comme "fichier" source, puisque gen.sh affiche le code généré sur la sortie standard.

Les différents codes se situent en sous-section A.3, et plus précisément :

- code bash : Listing 3
- Premier exemple de code C obtenu : Listing 4
- Second exemple de code C obtenu : Listing 5

2.1.3 Les avantages

Si le résultat n'avait que peu d'importance ici, ce début à une importance capitale dans ce projet puisque c'est ce petit code qui nous a permis de mieux comprendre ce qui était attendu de nous, comment le faire, et comment utiliser une FST.

2.2 Générer de l'asmX86 en bash

Sachant qu'on devrait sûrement au final généré du bytecode, nous avons décidé que, quitte à avoir du C, autant aller jusqu'à générer directement de l'assembleur. Pas spécialement pour être plus performant que le C (en effet, l'assembleur généré par gcc sera toujours plus efficace que celui que l'on peut faire à la main), mais pour avoir des idées des problèmes que nous rencontrerons en bytecode.

Quand nous parlons d'assembleur, nous entendons "assembleur x86_64" bien sûr, soit l'assembleur qui est généré par gcc sur nos machines.

En donnant l'option "-S" à gcc, on obtient non pas un binaire exécutable mais un fichier ".s" qui contient le code assembleur généré (avant l'assemblage effectif en binaire). En le générant pour nos sources C, nous avons pu faire du rétro-engineering dessus et comprendre la marche à suivre pour notre deuxième script.

La première partie, pour adapter toutes les parties générées de manière statique, a été relativement aisée. Par exemple, la déclaration de la fonction main, bien que plus longue et moins lisible qu'en C pouvait être plus ou moins copié/collé par rapport à ce que générait gcc, et même si quelques lignes restaient un peu mystérieuses au moment de la mise en place de "l'environnement" de la fonction, ce n'était absolument pas bloquant.

À l'opposé, lorsqu'il a fallu adapter les parties générées dynamiquement, ce fut beaucoup moins simple. Nous avons du, l'espace d'un instant, changer notre façon de programmer. En effet, l'assembleur est tellement bas niveau que l'on ne dispose pas de toutes les "syntactic sugar" dont on a tant l'habitude, notamment pour le contrôle de flux. Par exemple, ce n'est pas si simple que ce à quoi nous pourrions nous attendre de faire un "if (...) single_instruction;". Il faut gérer deux sauts, les labels associés, potentiellement préparer un ou deux registres, etc...

Ce fût très amusant à faire, et moins complexe qu'imaginé grâce à l'exhaustivité de la documentation. Néanmoins le temps passé dessus ne s'est révélé être aussi utile qu'escompté car nous le découvrirons plus tard les problèmes rencontrés pour le bytecode sont d'un ordre totalement différents.

Les différents codes se situent en annexe, annexe sous-section A.4, et plus précisément :

- code bash : Listing 6
- Premier exemple de code C obtenu : Listing 7
- Second exemple de code C obtenu : Listing 8

2.3 Adapter le travail précédent en Java

Ici le travail fait précédemment avec la génération d'un fichier C s'est montré être très utile. En effet cette étape a été traversée en quelques heures sans difficulté particulière.

Il nous suffit de séparer en trois cas l'étude d'un nœud et de créer une méthode particulière pour gérer intelligemment chaque cas. Le code est ajouté progressivement dans un StringBuffer avant d'être retourné (pour être imprimé en console ou dans un fichier).

Néanmoins cette étape ne fût absolument pas inutile puisqu'elle permet avec un cas simple de s'essayer à la génération d'une classe en Java. Seulement elle ne convenait pas tout à fait puisque nous avons recréé une manière de recréer des états et ce n'est pas ce que nous voulions.

Finalement nous avons obtenu le code présent en annexe dans le Listing 9.

2.4 Générer une structure de switch imbriqués en Java

Le code précédent nous permet de prendre confiance à propos de la génération de code en Java. Mais il s'agissait à présent de générer une structure en switch afin de s'approcher du sujet.

Cette fois nous avons rencontré de nombreuses difficultés à cause d'une problématique qui allait nous poursuivre pendant tout le projet : les switch ne sont pas voisins mais imbriqués les uns dans les autres. Ceci complexifia le problème mais nous avons finalement réussi grâce à un peu d'astuce et des souvenirs de nos cours de programmation en A1 à écrire une méthode récursive qui remplissait le StringBuffer au fur et à mesure et fermait en remontant dans la stack les blocs ouverts.

Il nous fallu aussi prendre en compte le cas supplémentaire suivant : un état final n'étant pas au bout de l'arbre de la FST (exemple : pour 'arbre' et 'arbres' le parcours du dit arbre est le même excepté que pour 'arbre' l'état final est sur le e). Nous n'y avons pas pensé auparavant et ce fut sources de bugs difficiles à identifier jusqu'à ce que nous comprenions notre erreur.

Le code final utilisé pour la génération se trouve dans le Listing 10.

3 Générer une FST

3.1 Le besoin

Avant de compiler nos FST, il nous faut bien sûr les construire à partir de données d'entrée et de données de sortie. Seulement pour faire des tests d'envergure nous ne pouvions plus les construire à la main. Il nous fallait les construire automatiquement.

Pour ce faire, nous avons décidé d'utiliser un algorithme de construction directe de "Minimal acyclic subsequential transducers". Il s'agit de l'algorithme utilisé dans le projet Apache Lucene, pour la construction de leurs FST. Il permet de construire des transducteurs minimums pour des entrées et sorties données. Nous pouvons donc avoir des données de la forme :

In	Out
John	1
Moth	2
Mob	3

TABLEAU 1 – Exemple de dictionnaire

Note : Les mots en entrées doivent être triés dans l'ordre lexicographique pour que l'algorithme fonctionne.

Cet algorithme nous permet d'être certain que le FST construit associera de manière unique la sortie 1 au mot John, la sortie 2 au mot Moth et la sortie 3 au mot Mob. Et ce même pour des millions d'entrées de cette forme.

De plus, étant utilisé par Lucene, cela nous permettra d'avoir une équivalence entre les FST que nous construisons et ceux construits par Lucene, afin qu'il soit possible de comparer les performances de notre méthode et celle de Lucene.

3.2 Fonctionnement de l'algorithme

L'algorithme est issu d'un papier de recherche qui date de 2001, écrit par Denis Maurel et Stoyan Mihov. Il fonctionne pour tout type de données d'entrée/sortie, mais nous l'avons adapté en fixant pour sorties sont des nombres entiers par souci de simplicité (et nous le supposons de performance mais n'en avons pas de preuve). L'algorithme parcourt les mots en entrée un par un, et cherche un préfixe commun entre le mot actuel et le mot du tour de boucle précédent. En déterminant ce préfixe commun, il est ensuite possible de réutiliser des états qui auraient déjà été créés lors du traitement d'un mot précédent (un peu comme dans un jeu de Scrabble). Petit à petit, on arrive donc à construire un Trie correct et à s'assurer que la valeur de sortie de chaque chaîne est unique.

[Direct Construction of Minimal Acyclic Subsequential Transducers \(2001\)](#)

3.3 Difficultés

Il nous a été très difficile de comprendre le pseudo code et de l'implémenter correctement. Nous avons du lors des premiers essais traverser de très nombreuses erreurs qui ne nous permettaient pas d'avancer suffisamment. Finalement nous n'avons pas eu d'autre choix que de l'implémenter en tout trois fois, en essayant à chaque fois d'avoir plus de recul. Nous avons vraiment travaillé tous les trois

ensemble sur le compréhension (et non l'implémentation) par nécessité afin de pouvoir profiter des interprétations de tous pour pouvoir avancer.

- Aujourd'hui malgré tous nos efforts l'algorithme n'est pas entièrement fonctionnel :
- Les états ne sont pas restitués dans le bon ordre (bug mineur)
 - Les poids ne sont pas correctement distribués sur tous les arcs, particulièrement lorsque les états finaux ne sont pas en bout de branche (bug majeur).
- Nous n'avons pas réussi à déterminer l'origine de ces bugs.

3.4 Résultat

Pour le dictionnaire suivant :

In	Out
A	1
Aani	2
Aaron	3
Aaronic	4
Aaronical	5
Aaronite	6
Aaronitic	7
Aaru	8
Ab	9
Ababdeh	10

TABLEAU 2 – Dictionnaire utilisé pour tester l'algorithme

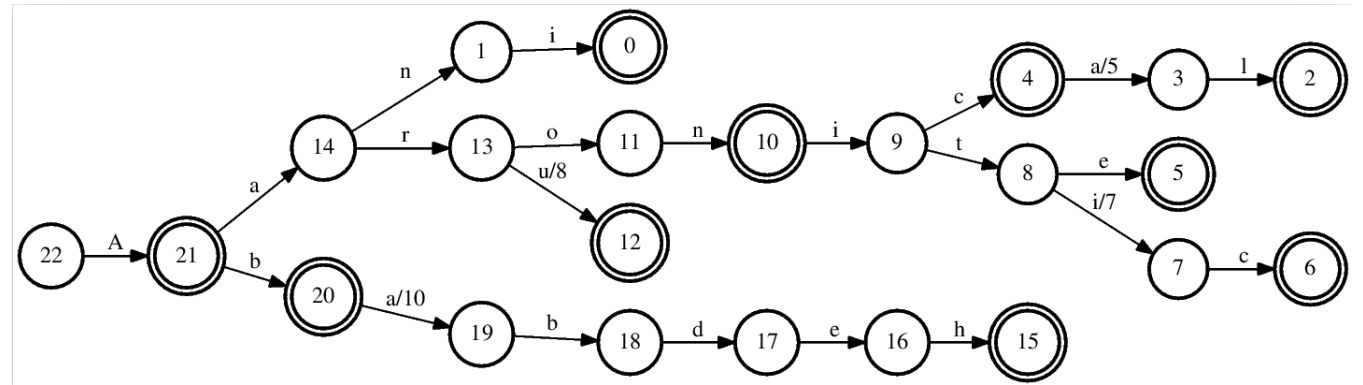


FIGURE 1 – La FST associée avec le Tableau 2 générée par l'algorithme

4 Idées d'amélioration et difficultés insurmontables

Nous disposions à présent d'un algorithme perfectible mais utilisable qui permettait de créer de grands FST. Nous avons aussi un moyen de générer du code Java pour parcourir la FST donnée en argument. Le gros du travail commençait : utiliser ces outils pour tester dans des conditions plus proches de celles réelles.

4.1 Idées originelles

4.1.1 L'encodage

Nous pensions au début du projet qu'il serait malin de changer l'encodage des caractères afin de contourner les mauvaises performances liées à l'UTF16. Seulement pour des besoins d'implémentation algorithmique nous avons très tôt pris la décision de travailler avec des entiers plutôt que les caractères unicodes. Cela nécessite une transformation de l'entrée avant de l'utiliser, mais cela aurait été nécessaire de toute manière.

Nous supposons en ayant fait ce choix avoir gagné en optimisation. Seulement il est regrettable que nous n'ayons pas réussi à implémenter l'algorithme avec des caractères ou des string, car cela nous aurait permis de faire un véritable comparatif.

4.1.2 Le traitement d'un texte en multithreading

Nous avons très vite pensé à la manière par laquelle nous pourrions utiliser notre classe générée avec une grande quantité de texte en entrée. Nous avons comme idée de parcourir l'entrée afin d'y distinguer les mots (séparés par des espaces) et de lancer le parcourt de la FST sur différentes mots en même temps, dans différents thread. En effet, en divisant l'entrée et en prenant en compte les performances du Java en multi-threading il est imaginable que les performances soient améliorées. Hélas nous n'avons pas arrivé à cette étape à cause des difficultés à venir.

4.2 Les difficultés les plus grandes

4.2.1 Compilation à la volée

Afin d'automatiser le process de génération puis utilisation de classe, il est nécessaire de la compiler. Seulement cela n'est pas simple en Java. Nous avons essayé d'utiliser une des bibliothèques fournies par Google pour le faire mais nous n'avons pas réussi à la faire fonctionner correctement.

De plus, même compilée il nous fallait ensuite utiliser la réflexivité pour utiliser la classe. Si bien que nous avons décidé de nous diriger vers le bytecode le plus vite possible afin de contourner ce problème de compilation.

4.2.2 Méthodes Java et la limitation 64Ko

Lorsque nous commençons à tester le code généré avec des entrées beaucoup plus grandes (un vrai dictionnaire avec quelques dizaines de millions d'entrée) notre première surprise fût la taille de la classe générée : 129Mo. Plutôt gros pour du fichier texte. Avant même d'aller plus loin nous étions très déçus de cette découverte sachant que dans le projet Lucène 8 millions de mots occupaient 90Mo.

Mais notre plus mauvaise surprise eu lieu lors de la compilation. Nous nous attendions à ce qu'il y ait une limite de la taille d'une classe ou méthode en Java et appréhendions le résultat de cette compilation. Mais lorsque le compilateur nous dit explicitement qu'une méthode ne pouvait pas dépasser 64Ko (ce que sera confirmé plus tard par de la lecture de documentations diverses) l'information était difficile à croire pour nous. Une grande désillusion. Et pourtant nous n'étions pas au bout de nos peines.

En effet, de par la très grande variété de caractères présents dans les textes aujourd'hui nous pouvions nous attendre à nous approcher de cette limite dès le premier switch, voir la dépasser avant même d'avoir écrit la première imbrication !

Nous avons décidé de continuer malgré cette problématique qui semblait sans solution viable dans tous les cas de figure afin de pouvoir étudier d'autres aspects du projet. Nous avons donc changé la génération afin qu'une méthode soit créée à chaque 'case' d'un switch. L'objectif est évident : réduire la taille des méthodes. Cette méthode utilisée est inspirée de celle du "trampolining" avec une différence très importante : nous n'avons pas un seul switch qui comporte beaucoup de cas différents. Nous avons une multitude de switch imbriqués les uns dans les autres comportant de très nombreux cas différents.

Le code est présent dans le Listing 11. Il fonctionne très bien pour des cas allant jusqu'à 100K mots : 1s pour 10K, 20s pour 30K, 1mn pour 50K ; puis à partir de 100K mots, le temps peut varier de 2 minutes à trop long, considéré infini. Nous n'avons pas trouvé l'origine de ce comportement et pire : nous n'avons pas la moindre idée de ce qui pourrait en être à l'origine sinon un problème de taille mémoire...

Mais nous étions malgré tout cela satisfait d'avoir su contourner ce problème. Jusqu'à ce qu'on étudie le bytecode de très près et comprenne le prix en Java de la création d'une méthode.

4.2.3 Le bytecode et la librairie asm

Écriture manuelle du bytecode

L'objectif ici est de garder notre algorithme existant pour créer du java, mais en l'adaptant pour créer du bytecode. Par ailleurs nous ne nous intéressons qu'au contenu de la méthode `compute()`, le reste étant juste l'initialisation de la classe.

Notes

La première lettre d'une mnémonique d'instruction bytecode est (dans la grande majorité des cas), le type sur laquelle l'instruction courante se base (i pour integer, f pour float, a pour reference, ...).

Dans la suite, on aura toujours 3 variables locales :

- 0 : int[] token, le token donné en argument de la fonction
- 1 : int pos, la position courante dans le token
- 2 : float result, le résultat final incrémenté lorsqu'on traverse un arc avec un poids non nul

Initialisation des variables

On avait, au début de la méthode :

- int pos=0;
- float result=0f;

Il faut donc faire :

- `iconst_0` // pousse 0 (integer) sur la stack
- `istore_1` // pop la stack et enregistre la valeur dans la variable local 1 (pos)
- `fconst_0` // pousse 0 (float) sur la stack
- `fstore_2` // pop la stack et enregistre la valeur dans la variable local 2 (result)

Test d'overflow

Pour chaque state, on avait :

```

1 || if( pos >= token.length ) {
2 ||     return -1;
3 || }
```

Listing 2 – Test sur la taille du token

On remplace cela par :

- `iload_1` // push la valeur de "pos"
- `aload_0` // push la reference de "token"
- `arraylength` // Pop la reference de "token" et en donne sa longueur
- `if_cmpge (ERR)` // Si (pos>=token.length) alors goto (ERR) (cf ci-dessous)

Erreurs

Lorsque l'on rencontre un cas d'erreur, on veut faire un : `return -1` ;

On veut donc avoir quelque part :

- `ldc #2` // push la valeur numéro 2 de la "constant pool"
- `freturn` // retourne la valeur au sommet de la stack

On a donc besoin que la seconde valeur de la constant pool soit "-1" (float). Si ce n'est pas la seconde valeur, il faut changer le #2

Notons qu'utiliser toujours la même adresse pour les cas d'erreurs est plus optimisé en terme de taille de la classe par rapport à ce qui est produit par javac

Switch On a de nombreux switch dans le java.

On a donc d'une part le switch, et d'autre part le `pos++` :

- `aload_0` // push la reference de "token"
- `iload_1` // push la valeur de "pos"
- `iinc 1 1` // incrémente de "1" la variable locale "1" (pos)
- `iaload` // pop les 2 dernière valeur de la stack pour push la i-ème (n-1) case du tableau (n-2), en l'occurrence `token[pos]`

```

lookupswitch {
    (val_X) : (pos_label_X) // correspond aux "case"
    (val_Y) : (pos_label_Y) // exemple : "65 : 42" si le goto pour la lettre 'A' (65) est l'instruction
    numéro 42
    ...
    default : (ERR) // default : return -1
```

```
}
```

== Incrément du total == Lorsque le poids d'un arc est != 0, on fait en java : `result+=21.0f`;

On va remplacer ça par :

```
fload_2 // push "result"
(
  fconst_1 // si le poids est 1
  ## OR ##
  ldc #N // sinon, et il faut ajouter cela dans la constant pool si ce n'est pas déjà présent
)
fadd // additionne les deux dernières valeurs de la stack et push le result
fstore_2 // save le résultat
```

State finaux Lorsque l'on est sur un state final, on fait en java : `return (pos != token.length) ? -1 : result`;

On va utiliser notre (ERR) pour factoriser le code :

```
— iload_1 // push la valeur de "pop"
— aload_0 // push la référence de "token"
— arraylength // pop et push la taille de "token"
— if_cmpne (ERR) // si (pos != token.length) goto (ERR)
— fload_2 // push la valeur de "result"
— freturn // retourne la valeur au sommet de la stack
```

5 Analyse

A Annexe : tests avec un script shell

A.1 Dictionnaires

Value	Word
0	mop
1	moth
2	pop
3	star
4	stop
5	top

TABLEAU 3 – Dictionnaire à utiliser avec la FST dans le Tableau 5

Value	Word
0	mop
1	moth
2	pop
3	slop
4	sloth
5	stop
6	top

TABLEAU 4 – Dictionnaire à utiliser avec la FST dans le Tableau 6

A.2 FST

Nœu	0	0	0	0	1	2	3	2	4	6	7	5	7	8	
Nœu suivant	1	4	4	6	2	3	9	9	5	7	5	9	8	9	
Nœu final															9
Caractère	M	P	T	S	O	T	H	P	O	T	O	P	A	R	
Poids			2	5	3			1				1			

TABLEAU 5 – FST utilisée avec le dictionnaire Tableau 3, voir Figure 2 page 16

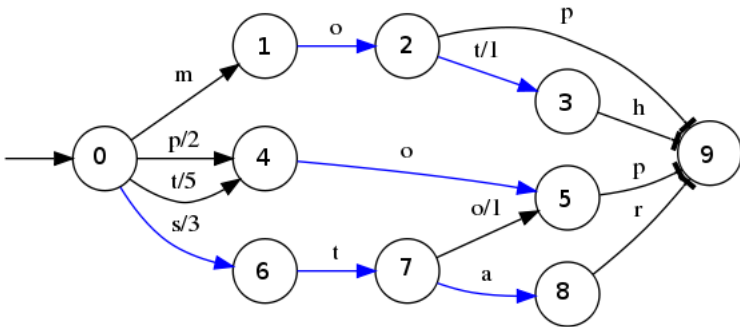


FIGURE 2 – La FST associée avec le Tableau 5 page 16

Nœu	0	0	0	0	3	3	1	2	4	5	6	5	
Nœu suivant	1	1	3	4	1	4	2	7	5	6	7	7	
Nœu final													7
Caractère	P	T	S	M	T	L	O	P	O	T	H	P	
Poids		2	6	3		2					1		

TABLEAU 6 – FST utilisée avec le dictionnaire Tableau 4, voir Figure 3 page 16

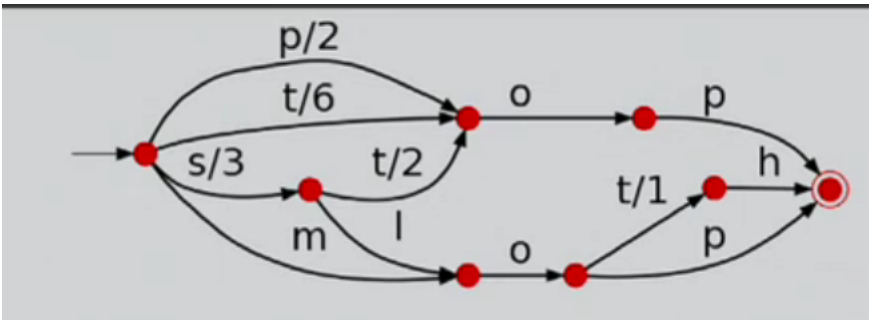


FIGURE 3 – La FST associée avec le Tableau 6 page 16

Les

A.3 Générer à la volée du code C

A.3.1 Script shell

```

1  #!/bin/bash
2
3  FST="$1"
4
5  sort "$FST" > "$FST.sort"
6
7  FIRST_CALL=1
8
9  cat <<EOF
10 #include <stdio.h>
11
12 int compute_fst(const char* token)
13 {
14     int pos=0;
15     int total=0;
16
17 EOF
18
19 while read DEP ARR CHAR WEIGHT ; do
20     # If it's a final node
21     if [[ ! "$CHAR" ]]; then
22         WEIGHT=${ARR:-0}
23         cat <<EOF
24         default:
25             return -1;
26     }
27
28 NODE_$DEP :
29 EOF
30     (( WEIGHT != 0 )) &&
31     echo "        total += $WEIGHT;"
32     echo "        if (token[pos] != '\\0') return -1;"
33     echo "        goto END;"
34
35     continue
36 fi
37
38 : ${WEIGHT:=0}
39
40 if [[ $DEP != $PREV_DEP ]]; then
41     if [[ ! "$FIRST_CALL" ]]; then
42         cat <<EOF
43         default:
44             return -1;
45     }
46
47 EOF
48     fi
49
50     cat <<EOF
51 NODE_$DEP :
52     switch (token[pos++]) {
53 EOF
54     fi
55
56     echo "        case '$CHAR':"
```

```
57      (( WEIGHT != 0 )) &&
58          echo "          total += $WEIGHT;"
59      echo "          goto NODE_$ARR;"
60
61      PREV_DEP=$DEP
62      FIRST_CALL=
63  done < "$FST.sort"
64
65  cat <<EOF
66
67  END :
68      return total;
69  }
70
71  int main(int argc, const char *argv[])
72  {
73      if (argc < 2)
74          return 1;
75
76      printf("%d\n", compute_fst(argv[1]));
77      return 0;
78  }
79  EOF
80
81  rm "$FST.sort"
```

Listing 3 – Script pour générer un code C à la volée d'une FST

A.3.2 Code C généré pour la FST définie dans le Tableau 5

```
1  #include <stdio.h>
2
3  int compute_fst(const char* token)
4  {
5      int pos=0;
6      int total=0;
7
8  NODE_0 :
9      switch (token[pos++]) {
10         case 'M':
11             goto NODE_1;
12         case 'P':
13             total += 2;
14             goto NODE_4;
15         case 'T':
16             total += 5;
17             goto NODE_4;
18         case 'S':
19             total += 3;
20             goto NODE_6;
21         default:
22             return -1;
23     }
24
25  NODE_1 :
26      switch (token[pos++]) {
27         case 'O':
28             goto NODE_2;
29         default:
30             return -1;
31     }
32
33  NODE_2 :
34      switch (token[pos++]) {
35         case 'T':
36             total += 1;
37             goto NODE_3;
38         case 'P':
39             goto NODE_9;
40         default:
41             return -1;
42     }
43
44  NODE_3 :
45      switch (token[pos++]) {
46         case 'H':
47             goto NODE_9;
48         default:
49             return -1;
50     }
51
52  NODE_4 :
53      switch (token[pos++]) {
54         case 'O':
55             goto NODE_5;
56         default:
57             return -1;
58     }
```



```
59 |
60 | NODE_5 :
61 |     switch (token[pos++]) {
62 |     case 'P':
63 |         goto NODE_9;
64 |     default:
65 |         return -1;
66 |     }
67 |
68 | NODE_6 :
69 |     switch (token[pos++]) {
70 |     case 'T':
71 |         goto NODE_7;
72 |     default:
73 |         return -1;
74 |     }
75 |
76 | NODE_7 :
77 |     switch (token[pos++]) {
78 |     case 'O':
79 |         total += 1;
80 |         goto NODE_5;
81 |     case 'A':
82 |         goto NODE_8;
83 |     default:
84 |         return -1;
85 |     }
86 |
87 | NODE_8 :
88 |     switch (token[pos++]) {
89 |     case 'R':
90 |         goto NODE_9;
91 |     default:
92 |         return -1;
93 |     }
94 |
95 | NODE_9 :
96 |     goto END;
97 |
98 | END :
99 |     return total;
100 | }
101 |
102 | int main(int argc, const char *argv[])
103 | {
104 |     if (argc < 2)
105 |         return 1;
106 |
107 |     printf("%d\n", compute_fst(argv[1]));
108 |     return 0;
109 | }
```

Listing 4 – Code C généré pour la FST définie dans Tableau 5

A.3.3 Code C généré pour la FST définie dans le Tableau 6

```

1  #include <stdio.h>
2
3  int compute_fst(const char* token)
4  {
5      int pos=0;
6      int total=0;
7
8  NODE_0 :
9      pos++;
10     switch (token[pos-1]) {
11     case 'P':
12         total += 2;
13         goto NODE_1;
14     case 'T':
15         total += 6;
16         goto NODE_1;
17     case 'S':
18         total += 3;
19         goto NODE_3;
20     case 'M':
21         goto NODE_4;
22     default:
23         return -1;
24     }
25
26  NODE_1 :
27     pos++;
28     switch (token[pos-1]) {
29     case 'O':
30         goto NODE_2;
31     default:
32         return -1;
33     }
34
35  NODE_2 :
36     pos++;
37     switch (token[pos-1]) {
38     case 'P':
39         goto NODE_7;
40     default:
41         return -1;
42     }
43
44  NODE_3 :
45     pos++;
46     switch (token[pos-1]) {
47     case 'T':
48         total += 2;
49         goto NODE_1;
50     case 'L':
51         goto NODE_4;
52     default:
53         return -1;
54     }
55
56  NODE_4 :
57     pos++;
58     switch (token[pos-1]) {

```

```
59     case '0':
60         goto NODE_5;
61     default:
62         return -1;
63     }
64
65 NODE_5 :
66     pos++;
67     switch (token[pos-1]) {
68     case 'T':
69         total += 1;
70         goto NODE_6;
71     case 'P':
72         goto NODE_7;
73     default:
74         return -1;
75     }
76
77 NODE_6 :
78     pos++;
79     switch (token[pos-1]) {
80     case 'H':
81         goto NODE_7;
82     default:
83         return -1;
84     }
85
86 NODE_7 :
87     goto END;
88
89 END :
90     return total;
91 }
92
93 int main(int argc, const char *argv[])
94 {
95     if (argc < 2)
96         return 1;
97
98     printf("%d\n", compute_fst(argv[1]));
99     return 0;
100 }
```

Listing 5 – Code C généré pour la FST définie dans Tableau 6

A.4 Générer à la volée du code assembleur x86

A.4.1 Script shell

```

1  #!/bin/bash
2
3  FST="$1"
4  oIFS=$IFS
5  : ${ASM_SWITCH:=1}
6  (( ASM_SWITCH == 0 )) && unset ASM_SWITCH
7
8  DO_SWITCH=
9  : ${SWITCH_LIMIT:=2}
10
11 sort "$FST" > "$FST.sort"
12
13 FIRST_CALL=1
14
15 compute_switch() {
16     ARR_ASCII=( $(awk 'BEGIN{for(n=0;n<256;n++)ord[sprintf("%c",n)]=n}
17 /~'$DEP'/{print ord[$3]}' "$FST.sort" | sort) )
18 }
19
20 next() {
21     PREV_DEP=$DEP
22     FIRST_CALL=
23
24     continue
25 }
26
27 cat <<EOF
28     .file      "${FST%.*}.c"
29     .text
30     .globl    compute_fst
31     .type     compute_fst, @function
32
33 compute_fst:
34     pushq    %rbp                # remember old base pointer
35     movq     %rsp, %rbp          # set new base pointer
36     movq     %rdi, -24(%rbp)      # put content of rdi (token) in -24(%rbp)
37     movl     \0, -8(%rbp)         # pos (-8(%rbp)) = 0
38     movl     \0, -4(%rbp)         # total (-4(%rbp)) = 0
39
40 EOF
41
42 while read DEP ARR CHAR WEIGHT ; do
43     # If it's a final node
44     if [[ ! "$CHAR" ]]; then
45         if [[ ${#tmp[@]} != 0 ]] ; then
46             IFS=$'\n'
47             echo "${tmp[*]}"
48             IFS=$oIFS
49             tmp=()
50         fi
51
52         WEIGHT=${ARR:-0}
53
54         [[ ! "$DO_SWITCH" ]] &&
55             cat <<EOF
56     movl \0-1, %eax                # default : return -1

```

```

57         jmp .RET
58 EOF
59         echo -e ".NODE_$DEP:"
60
61         (( WEIGHT != 0 )) &&
62         echo "    addl \$$WEIGHT, -4(%rbp)      # total += Weight"
63         echo "    jmp .END                      # goto END"
64
65         continue
66     fi
67
68     : ${WEIGHT:=0}
69
70     # Change of node
71     if [[ $DEP != $PREV_DEP ]]; then
72         if [[ ! "$FIRST_CALL" && ! "$DO_SWITCH" ]]; then
73             cat <<EOF
74             movl \$_-1, %eax      # default : return -1
75             jmp .RET
76         EOF
77     EOF
78     fi
79
80     if [[ ${#tmp[@]} != 0 ]] ; then
81         IFS=$'\n'
82         echo "${tmp[*]}"
83         IFS=$oIFS
84         tmp=()
85     fi
86
87     cat <<EOF
88 .NODE_$DEP:
89     addl    \$_1, -8(%rbp)      # pos++
90     movl    -8(%rbp), %eax     # eax = pos
91     leaq    -1(%rax), %rdx     # rdx = pos - 1
92     movq    -24(%rbp), %rax    # load token (address) in rax
93     addq    %rdx, %rax        # rax = &(token[pos-1])
94     movzbl  (%rax), %eax      # eax = token[pos-1]
95
96 EOF
97     if [[ "$ASM_SWITCH" ]] ; then
98         DO_SWITCH=
99         compute_switch
100         (( ${#ARR_ASCII[@]} > SWITCH_LIMIT )) &&
101         DO_SWITCH=1
102     fi
103 fi
104
105 printf -v CHAR_INT '%d' "\"$CHAR"
106
107 [[ ! "$DO_SWITCH" ]] &&
108     echo "    cmpl \$$CHAR_INT, %eax      # case '$CHAR'"
109
110 if (( WEIGHT != 0 || DO_SWITCH == 1 )) ; then
111     tmp+=(
112         ".NODE_${DEP}_$CHAR:"
113         "    addl \$$WEIGHT, -4(%rbp)      # total += $WEIGHT"
114         "    jmp .NODE_$ARR"
115         ""
116

```

```

117         )
118
119     if [[ "$DO_SWITCH" ]]; then
120         [[ $DEP == $PREV_DEP ]] && next
121
122         echo "        subl    \${ARR_ASCII[0]}, %eax        # eax -= '$(
123             printf '%c' "$CHAR")'"
124         # MAX - MIN
125         CHAR_RANGE=$(( ARR_ASCII[${#ARR_ASCII[@]}-1] - ARR_ASCII[0] ))
126
127         cat <<EOF
128     cmpl \${CHAR_RANGE}, %eax        # eax = '$(printf '%c' "$(printf "\
129         x$(printf "%x" ${ARR_ASCII[${#ARR_ASCII[@]}-1}]" )"' - '$(printf
130         "\x$(printf "%x" ${ARR_ASCII[0]})" )' (max - min)
131     ja .END
132     movq .NODE_${DEP}_SW(,%rax,8), %rax
133     jmp *%rax
134
135     .section .rodata
136     .NODE_${DEP}_SW:
137 EOF
138     j=0
139     for (( i = 0; i <= (ARR_ASCII[${#ARR_ASCII[@]}-1] - ARR_ASCII[0])
140         ; i++ )); do
141         if (( ARR_ASCII[0] + i == ARR_ASCII[j] )) ; then
142             echo "        .quad .NODE_${DEP}_$(printf "\x$(printf "%x" $
143                 {ARR_ASCII[j]})" )"        # '$(printf "\x$(printf "%x"
144                 ${ARR_ASCII[j]})" )'"
145             (( j++ ))
146         else
147             echo "        .quad .ERR        # '$(printf "\x$(
148                 printf "%x" $(( ARR_ASCII[0] + i )))'"
149         fi
150     done
151
152     echo -e "        .text\n"
153 else
154     echo "        je .NODE_${DEP}_$CHAR"
155 fi
156 else
157     echo "        je .NODE_$ARR"
158 fi
159
160 next
161 done < "$FST.sort"
162
163 cat <<EOF
164
165 .ERR:
166     movl \${-1}, %eax        # return -1
167     jmp .RET
168
169 .END:
170     movl    -4(%rbp), %eax    # Put return value in eax
171
172 .RET:
173     popq    %rbp
174     ret
175
176 .size      compute_fst, .-compute_fst
177 .section   .rodata
178
179 .PRINTF_FMT:

```

```

170     .string "%d\n"
171     .text
172     .globl main
173     .type main, @function
174 main:
175     pushq    %rbp
176     movq     %rsp, %rbp
177     subq     \$16, %rsp
178     movl     %edi, -4(%rbp)
179     movq     %rsi, -16(%rbp)
180     cmpl     \$1, -4(%rbp)           # if (argc < 2)
181     jg       .DO_MAIN
182     movl     \$1, %eax              # return 1;
183     jmp      .END_MAIN
184 .DO_MAIN:
185     movq     -16(%rbp), %rax        # rax = &argv[0]
186     addq     \$8, %rax              # rax = &argv[1]
187     movq     (%rax), %rax          # rax = argv[1]
188     movq     %rax, %rdi            # rdi = argv[1]
189     call     compute_fst
190     movl     %eax, %esi            # esi = compute_fst()
191     movl     \$.PRINTF_FMT, %edi
192     movl     \$0, %eax
193     call     printf
194     movl     \$0, %eax
195 .END_MAIN:
196     leave
197     ret
198     .size    main, .-main
199     .ident   "GCC: (GNU) 4.9.2 20150212 (Red Hat 4.9.2-6)"
200     .section .note.GNU-stack,"",@progbits
201 EOF
202
203 rm "$FST.sort"

```

Listing 6 – Script pour générer un code assembleur x86 à la colée d'une FST

A.4.2 Code assembleur x86 généré pour la FST défini dans le Tableau 5

```

1      .file      "3.c"
2      .text
3      .globl    compute_fst
4      .type     compute_fst, @function
5
6  compute_fst:
7      pushq     %rbp                # remember old base pointer
8      movq     %rsp, %rbp          # set new base pointer
9      movq     %rdi, -24(%rbp)      # put content of rdi (token) in -24(%rbp)
10     movl     $0, -8(%rbp)         # pos (-8(%rbp)) = 0
11     movl     $0, -4(%rbp)         # total (-4(%rbp)) = 0
12
13  .NODE_0:
14     addl     $1, -8(%rbp)         # pos++
15     movl     -8(%rbp), %eax        # eax = pos
16     leaq     -1(%rax), %rdx       # rdx = pos - 1
17     movq     -24(%rbp), %rax      # load token (address) in rax
18     addq     %rdx, %rax           # rax = &(token[pos-1])
19     movzbl   (%rax), %eax         # eax = token[pos-1]
20
21     cmpl     $77, %eax            # case 'M'
22     je       .NODE_1
23     cmpl     $80, %eax            # case 'P'
24     je       .NODE_0_P
25     cmpl     $84, %eax            # case 'T'
26     je       .NODE_0_T
27     cmpl     $83, %eax            # case 'S'
28     je       .NODE_0_S
29     movl     $-1, %eax            # default : return -1
30     jmp      .RET
31
32  .NODE_0_P:
33     addl     $2, -4(%rbp)         # total += 2
34     jmp      .NODE_4
35
36  .NODE_0_T:
37     addl     $5, -4(%rbp)         # total += 5
38     jmp      .NODE_4
39
40  .NODE_0_S:
41     addl     $3, -4(%rbp)         # total += 3
42     jmp      .NODE_6
43
44  .NODE_1:
45     addl     $1, -8(%rbp)         # pos++
46     movl     -8(%rbp), %eax        # eax = pos
47     leaq     -1(%rax), %rdx       # rdx = pos - 1
48     movq     -24(%rbp), %rax      # load token (address) in rax
49     addq     %rdx, %rax           # rax = &(token[pos-1])
50     movzbl   (%rax), %eax         # eax = token[pos-1]
51
52     cmpl     $79, %eax            # case 'O'
53     je       .NODE_2
54     movl     $-1, %eax            # default : return -1
55     jmp      .RET
56
57  .NODE_2:
58     addl     $1, -8(%rbp)         # pos++

```



```

59      movl    -8(%rbp), %eax    # eax = pos
60      leaq    -1(%rax), %rdx    # rdx = pos - 1
61      movq    -24(%rbp), %rax   # load token (address) in rax
62      addq    %rdx, %rax        # rax = &(token[pos-1])
63      movzbl  (%rax), %eax      # eax = token[pos-1]
64
65      cmpl    $84, %eax         # case 'T'
66      je      .NODE_2_T
67      cmpl    $80, %eax         # case 'P'
68      je      .NODE_9
69      movl    $-1, %eax         # default : return -1
70      jmp     .RET
71
72 .NODE_2_T:
73      addl    $1, -4(%rbp)      # total += 1
74      jmp     .NODE_3
75
76 .NODE_3:
77      addl    $1, -8(%rbp)      # pos++
78      movl    -8(%rbp), %eax    # eax = pos
79      leaq    -1(%rax), %rdx    # rdx = pos - 1
80      movq    -24(%rbp), %rax   # load token (address) in rax
81      addq    %rdx, %rax        # rax = &(token[pos-1])
82      movzbl  (%rax), %eax      # eax = token[pos-1]
83
84      cmpl    $72, %eax         # case 'H'
85      je      .NODE_9
86      movl    $-1, %eax         # default : return -1
87      jmp     .RET
88
89 .NODE_4:
90      addl    $1, -8(%rbp)      # pos++
91      movl    -8(%rbp), %eax    # eax = pos
92      leaq    -1(%rax), %rdx    # rdx = pos - 1
93      movq    -24(%rbp), %rax   # load token (address) in rax
94      addq    %rdx, %rax        # rax = &(token[pos-1])
95      movzbl  (%rax), %eax      # eax = token[pos-1]
96
97      cmpl    $79, %eax         # case 'O'
98      je      .NODE_5
99      movl    $-1, %eax         # default : return -1
100     jmp     .RET
101
102 .NODE_5:
103     addl    $1, -8(%rbp)      # pos++
104     movl    -8(%rbp), %eax    # eax = pos
105     leaq    -1(%rax), %rdx    # rdx = pos - 1
106     movq    -24(%rbp), %rax   # load token (address) in rax
107     addq    %rdx, %rax        # rax = &(token[pos-1])
108     movzbl  (%rax), %eax      # eax = token[pos-1]
109
110     cmpl    $80, %eax         # case 'P'
111     je      .NODE_9
112     movl    $-1, %eax         # default : return -1
113     jmp     .RET
114
115 .NODE_6:
116     addl    $1, -8(%rbp)      # pos++
117     movl    -8(%rbp), %eax    # eax = pos
118     leaq    -1(%rax), %rdx    # rdx = pos - 1

```

```

119      movq    -24(%rbp), %rax # load token (address) in rax
120      addq    %rdx, %rax     # rax = &(token[pos-1])
121      movzbl  (%rax), %eax    # eax = token[pos-1]
122
123      cmpl    $84, %eax      # case 'T'
124      je      .NODE_7
125      movl    $-1, %eax      # default : return -1
126      jmp     .RET
127
128 .NODE_7:
129      addl    $1, -8(%rbp)    # pos++
130      movl    -8(%rbp), %eax  # eax = pos
131      leaq    -1(%rax), %rdx  # rdx = pos - 1
132      movq    -24(%rbp), %rax # load token (address) in rax
133      addq    %rdx, %rax     # rax = &(token[pos-1])
134      movzbl  (%rax), %eax    # eax = token[pos-1]
135
136      cmpl    $79, %eax      # case 'O'
137      je      .NODE_7_0
138      cmpl    $65, %eax      # case 'A'
139      je      .NODE_8
140      movl    $-1, %eax      # default : return -1
141      jmp     .RET
142
143 .NODE_7_0:
144      addl    $1, -4(%rbp)    # total += 1
145      jmp     .NODE_5
146
147 .NODE_8:
148      addl    $1, -8(%rbp)    # pos++
149      movl    -8(%rbp), %eax  # eax = pos
150      leaq    -1(%rax), %rdx  # rdx = pos - 1
151      movq    -24(%rbp), %rax # load token (address) in rax
152      addq    %rdx, %rax     # rax = &(token[pos-1])
153      movzbl  (%rax), %eax    # eax = token[pos-1]
154
155      cmpl    $82, %eax      # case 'R'
156      je      .NODE_9
157      movl    $-1, %eax      # default : return -1
158      jmp     .RET
159
160 .NODE_9:
161      jmp     .END           # goto END
162
163 .END:
164      movl    -4(%rbp), %eax  # Put return value in eax
165 .RET:
166      popq    %rbp
167      ret
168      .size    compute_fst, .-compute_fst
169      .section        .rodata
170
171 .PRINTF_FMT:
172      .string  "%d\n"
173      .text
174      .globl   main
175      .type    main, @function
176 main:
177      pushq    %rbp
178      movq     %rsp, %rbp

```

```

179     subq    $16, %rsp
180     movl    %edi, -4(%rbp)
181     movq    %rsi, -16(%rbp)
182     cmpl    $1, -4(%rbp)           # if (argc < 2)
183     jg      .DO_MAIN
184     movl    $1, %eax               # return 1;
185     jmp     .END_MAIN
186 .DO_MAIN:
187     movq    -16(%rbp), %rax        # rax = &argv[0]
188     addq    $8, %rax               # rax = &argv[1]
189     movq    (%rax), %rax           # rax = argv[1]
190     movq    %rax, %rdi             # rdi = argv[1]
191     call    compute_fst
192     movl    %eax, %esi             # esi = compute_fst()
193     movl    $.PRINTF_FMT, %edi
194     movl    $0, %eax
195     call    printf
196     movl    $0, %eax
197 .END_MAIN:
198     leave
199     ret
200     .size   main, .-main
201     .ident  "GCC: (GNU) 4.9.2 20150212 (Red Hat 4.9.2-6)"
202     .section .note.GNU-stack,"",@progbits

```

Listing 7 – Code assembleur x86 généré pour la FST définie dans Tableau 5

A.4.3 Code assembleur x86 généré pour la FST défini dans le Tableau 6

```

1      .file      "3.c"
2      .text
3      .globl    compute_fst
4      .type     compute_fst, @function
5
6  compute_fst:
7      pushq     %rbp                # remember old base pointer
8      movq      %rsp, %rbp          # set new base pointer
9      movq      %rdi, -24(%rbp)     # put content of rdi (token) in -24(%rbp)
10     movl      $0, -8(%rbp)        # pos (-8(%rbp)) = 0
11     movl      $0, -4(%rbp)        # total (-4(%rbp)) = 0
12
13  .NODE_0:
14     addl      $1, -8(%rbp)        # pos++
15     movl      -8(%rbp), %eax       # eax = pos
16     leaq      -1(%rax), %rdx      # rdx = pos - 1
17     movq      -24(%rbp), %rax     # load token (address) in rax
18     addq      %rdx, %rax          # rax = &(token[pos-1])
19     movzbl    (%rax), %eax        # eax = token[pos-1]
20
21     cmpl      $80, %eax           # case 'P'
22     je        .NODE_0_P
23     cmpl      $84, %eax           # case 'T'
24     je        .NODE_0_T
25     cmpl      $83, %eax           # case 'S'
26     je        .NODE_0_S
27     cmpl      $77, %eax           # case 'M'
28     je        .NODE_4
29     movl      $-1, %eax           # default : return -1
30     jmp       .RET
31
32  .NODE_0_P:
33     addl      $2, -4(%rbp)        # total += 2
34     jmp       .NODE_1
35
36  .NODE_0_T:
37     addl      $6, -4(%rbp)        # total += 6
38     jmp       .NODE_1
39
40  .NODE_0_S:
41     addl      $3, -4(%rbp)        # total += 3
42     jmp       .NODE_3
43
44  .NODE_1:
45     addl      $1, -8(%rbp)        # pos++
46     movl      -8(%rbp), %eax       # eax = pos
47     leaq      -1(%rax), %rdx      # rdx = pos - 1
48     movq      -24(%rbp), %rax     # load token (address) in rax
49     addq      %rdx, %rax          # rax = &(token[pos-1])
50     movzbl    (%rax), %eax        # eax = token[pos-1]
51
52     cmpl      $79, %eax           # case 'O'
53     je        .NODE_2
54     movl      $-1, %eax           # default : return -1
55     jmp       .RET
56
57  .NODE_2:
58     addl      $1, -8(%rbp)        # pos++

```

```

59      movl    -8(%rbp), %eax    # eax = pos
60      leaq    -1(%rax), %rdx    # rdx = pos - 1
61      movq    -24(%rbp), %rax   # load token (address) in rax
62      addq    %rdx, %rax       # rax = &(token[pos-1])
63      movzbl  (%rax), %eax      # eax = token[pos-1]
64
65      cmpl    $80, %eax         # case 'P'
66      je      .NODE_7
67      movl    $-1, %eax         # default : return -1
68      jmp     .RET
69
70 .NODE_3:
71      addl    $1, -8(%rbp)      # pos++
72      movl    -8(%rbp), %eax    # eax = pos
73      leaq    -1(%rax), %rdx    # rdx = pos - 1
74      movq    -24(%rbp), %rax   # load token (address) in rax
75      addq    %rdx, %rax       # rax = &(token[pos-1])
76      movzbl  (%rax), %eax      # eax = token[pos-1]
77
78      cmpl    $84, %eax         # case 'T'
79      je      .NODE_3_T
80      cmpl    $76, %eax         # case 'L'
81      je      .NODE_4
82      movl    $-1, %eax         # default : return -1
83      jmp     .RET
84
85 .NODE_3_T:
86      addl    $2, -4(%rbp)      # total += 2
87      jmp     .NODE_1
88
89 .NODE_4:
90      addl    $1, -8(%rbp)      # pos++
91      movl    -8(%rbp), %eax    # eax = pos
92      leaq    -1(%rax), %rdx    # rdx = pos - 1
93      movq    -24(%rbp), %rax   # load token (address) in rax
94      addq    %rdx, %rax       # rax = &(token[pos-1])
95      movzbl  (%rax), %eax      # eax = token[pos-1]
96
97      cmpl    $79, %eax         # case 'O'
98      je      .NODE_5
99      movl    $-1, %eax         # default : return -1
100     jmp     .RET
101
102 .NODE_5:
103     addl    $1, -8(%rbp)      # pos++
104     movl    -8(%rbp), %eax    # eax = pos
105     leaq    -1(%rax), %rdx    # rdx = pos - 1
106     movq    -24(%rbp), %rax   # load token (address) in rax
107     addq    %rdx, %rax       # rax = &(token[pos-1])
108     movzbl  (%rax), %eax      # eax = token[pos-1]
109
110     cmpl    $84, %eax         # case 'T'
111     je      .NODE_5_T
112     cmpl    $80, %eax         # case 'P'
113     je      .NODE_7
114     movl    $-1, %eax         # default : return -1
115     jmp     .RET
116
117 .NODE_5_T:
118     addl    $1, -4(%rbp)      # total += 1

```

```

119     jmp .NODE_6
120
121 .NODE_6:
122     addl    $1, -8(%rbp)    # pos++
123     movl    -8(%rbp), %eax  # eax = pos
124     leaq    -1(%rax), %rdx  # rdx = pos - 1
125     movq    -24(%rbp), %rax # load token (address) in rax
126     addq    %rdx, %rax      # rax = &(token[pos-1])
127     movzbl  (%rax), %eax    # eax = token[pos-1]
128
129     cmpl    $72, %eax       # case 'H'
130     je      .NODE_7
131     movl    $-1, %eax       # default : return -1
132     jmp     .RET
133
134 .NODE_7:
135     jmp     .END            # goto END
136
137 .END:
138     movl    -4(%rbp), %eax  # Put return value in eax
139 .RET:
140     popq    %rbp
141     ret
142     .size   compute_fst, .-compute_fst
143     .section      .rodata
144
145 .PRINTF_FMT:
146     .string  "%d\n"
147     .text
148     .globl   main
149     .type    main, @function
150 main:
151     pushq   %rbp
152     movq    %rsp, %rbp
153     subq    $16, %rsp
154     movl    %edi, -4(%rbp)
155     movq    %rsi, -16(%rbp)
156     cmpl    $1, -4(%rbp)    # if (argc < 2)
157     jg      .DO_MAIN
158     movl    $1, %eax        # return 1;
159     jmp     .END_MAIN
160 .DO_MAIN:
161     movq    -16(%rbp), %rax  # rax = &(argv[0])
162     addq    $8, %rax         # rax = &(argv[1])
163     movq    (%rax), %rax     # rax = argv[1]
164     movq    %rax, %rdi      # rdi = argv[1]
165     call    compute_fst
166     movl    %eax, %esi      # esi = compute_fst()
167     movl    $.PRINTF_FMT, %edi
168     movl    $0, %eax
169     call    printf
170     movl    $0, %eax
171 .END_MAIN:
172     leave
173     ret
174     .size   main, .-main
175     .ident  "GCC: (GNU) 4.9.2 20150212 (Red Hat 4.9.2-6)"
176     .section      .note.GNU-stack,"",@progbits

```

Listing 8 – Code assembleur x86 généré pour la FST définie dans Tableau 6

B Annexe : générer du code java dynamiquement

B.1 Dans une seule méthode

B.1.1 Avec différents états représentés par des méthodes

```

1  import java.util.List;
2
3  class FstGenerator {
4
5      private StringBuffer strBuff;
6      private List<State> fstStates;
7
8      public FstGenerator(List<State> fstStates) {
9          this.fstStates = fstStates;
10         this.strBuff = new StringBuffer();
11     }
12
13     public StringBuffer compute() {
14         append("class FstCompute {");
15         generateMain();
16
17         for ( State state : fstStates ) {
18             generateStateCase(state);
19         }
20
21         append("}");
22
23         return strBuff;
24     }
25
26     private void generateMain() {
27         appendWithTab("public static float compute(int[] token) {"
28             , 1);
29
30         appendWithTab("return node_0(token, 0, 0f);", 2);
31
32         appendWithTab("}", 1);
33     }
34
35     private void generateStateCase(State state) {
36         switch (state.getNumArcs()) {
37             case 0:
38                 generateLastStateCase(state);
39                 break;
40             case 1:
41                 generateStateWithOneArc(state);
42                 break;
43             default:
44                 generateGeneralState(state);

```

```

44         break;
45     }
46 }
47
48 private void generateLastStateCase(State state) {
49     appendWithTab("private static float node_" + state.getId()
50         + "(int[] token, int pos, float result) {", 1);
51     appendWithTab("return result;", 2);
52     appendWithTab("}", 1);
53 }
54
55 private void generateStateWithOneArc(State state) {
56     appendWithTab("private static float node_" + state.getId()
57         + "(int[] token, int pos, float result) {", 1);
58
59     appendWithTab("if(pos>=token.length || token[pos]!=" +
60         state.getArc(0).getIlabel() + ")", 2);
61     appendWithTab("return -1;", 3);
62     appendWithTab("return node_" + state.getArc(0).
63         getNextState().getId()
64         + "(token, pos+1, result+" +
65         state.getArc(0).getWeight() + "f);", 2);
66
67     appendWithTab("}", 1);
68 }
69
70 private void generateGeneralState(State state) {
71     appendWithTab("private static float node_" + state.getId()
72         + "(int[] token, int pos, float result) {", 1);
73     appendWithTab("if(pos>=token.length) {return -1;}", 2);
74     appendWithTab("switch(token[pos]) {", 2);
75     for (int i = 0; i < state.getNumArcs(); i++) {
76         appendWithTab("case " + state.getArc(i).getIlabel() +
77             ":", 3);
78         appendWithTab("return node_" + state.getArc(i).
79             getNextState().getId()
80             + "(token, pos+1, result+" +
81             state.getArc(i).getWeight() + "f);", 4);
82     }
83     appendWithTab("default:", 3);
84     appendWithTab("return -1;", 4);
85     appendWithTab("}", 2);
86     appendWithTab("}", 1);
87 }
88
89 private void append(String strToAppend) {
90     strBuff.append(strToAppend);
91     strBuff.append("\n");
92 }

```



```
91 | private void appendWithTab(String strToAppend, int numberOfTab
    | ) {
92 |     for (int i = 0; i < numberOfTab; i++) {
93 |         strBuff.append("\t");
94 |     }
95 |     strBuff.append(strToAppend);
96 |     strBuff.append("\n");
97 | }
98 |
99 | }
```

Listing 9 – Code Java générant une structure avec des nœuds

B.1.2 Uniquement avec des switch

```

1 package generator;
2
3 import util.State;
4
5 public class FstGenerator {
6
7     private StringBuffer strBuff;
8
9     public FstGenerator() {
10    }
11
12    public StringBuffer compute(State initState, String className)
13    {
14        strBuff = new StringBuffer();
15        System.out.println("[FstGenerator] Transforming Fst to a
16            java class");
17        append("package generated;");
18
19        append("public class " + className + " {");
20
21        appendWithTab("public static float compute(int[] token) {"
22            , 1);
23
24        appendWithTab("int pos=0;", 2);
25        appendWithTab("float result=0f;", 2);
26
27        generateCases(initState, 2);
28
29        appendWithTab("}", 1);
30
31        append("}");
32        System.out.println("[FstGenerator] Successfully
33            transformed fst to " + className + ".java");
34        return strBuff;
35    }
36
37    private void generateCases(State currentState, int tab) {
38
39        if( currentState.getNumArcs() > 0) {
40            generateTokenLengthTest(tab);
41            appendWithTab("switch(token[pos++]) {" , tab);
42            for (int i = 0; i < currentState.getNumArcs(); i++) {
43                appendWithTab("case " + currentState.getArc(i).
44                    getIlabel() + ":", tab+1);
45                if (currentState.getArc(i).getOlabel() != 0) {
46                    appendWithTab("result+=" + currentState.getArc
47                        (i).getOlabel() + "f;", tab+2);

```

```

43         }
44         if (currentState.getArc(i).getNextState().
45             isFinalState()) {
46             appendWithTab("if(pos==token.length) {return
47                 result;}", tab+2);
48         }
49         generateCases(currentState.getArc(i).getNextState
50             (), tab+2);
51     }
52     appendWithTab("default:", tab+1);
53     appendWithTab("return -1;", tab+2);
54     appendWithTab("}", tab);
55 } else {
56     appendWithTab("return (pos!=token.length) ? -1 :
57         result;", tab);
58 }
59 }
60
61 private void generateTokenLengthTest(int tab) {
62     appendWithTab("if(pos>=token.length) {return -1;}", tab);
63 }
64
65 private void append(String strToAppend) {
66     strBuff.append(strToAppend);
67     strBuff.append("\n");
68 }
69
70 private void appendWithTab(String strToAppend, int numberOfTab
71     ) {
72     for (int i = 0; i < numberOfTab; i++) {
73         strBuff.append("\t");
74     }
75     strBuff.append(strToAppend);
76     strBuff.append("\n");
77 }
78 }

```

Listing 10 – Code Java générant une structure switch

B.2 Dans différentes méthodes

```

1  package generator;
2
3  import util.State;
4  import java.util.List;
5  import java.util.ArrayList;
6
7  public class BigFstGenerator {
8
9      private StringBuffer strBuff;
10
11     public BigFstGenerator() {
12     }
13
14     public StringBuffer compute(State initState, String className)
15     {
16         strBuff = new StringBuffer();
17         System.out.println("[FstGenerator] Transforming Fst to a
18             java class");
19         append("package generated;");
20
21         append("public class " + className + " {");
22
23         appendWithTab("public static float compute(int[] token) {",
24             1);
25
26         appendWithTab("int pos=0;", 2);
27         appendWithTab("float result=0f;", 2);
28         appendWithTab("return state_" + initState.getId() + "(
29             token, pos, result);", 2);
30
31         appendWithTab("}", 1);
32
33         List<State> doneStates = new ArrayList<>();
34         generateCases(initState, doneStates);
35
36         append("}");
37         System.out.println("[FstGenerator] Successfully
38             transformed fst to " + className + ".java");
39         return strBuff;
40     }
41
42     private void generateCases(State currentState, List<State>
43         doneStates) {
44         List<State> nextStates = new ArrayList<>();
45
46         append("\n\tprivate static float state_" + currentState.
47             getId() +

```

```

42         "(int[] token, int pos, float result) {");
43
44     generateTokenLengthTest(2);
45     appendWithTab("switch(token[pos++]) {", 2);
46     for (int i = 0; i < currentState.getNumArcs(); i++) {
47         appendWithTab("case " + currentState.getArc(i).
48             getIlabel() + ":", 3);
49
50         if (currentState.getArc(i).getOlabel() != 0) {
51             appendWithTab("result+=" + currentState.getArc(i).
52                 getOlabel() + "f;", 4);
53         }
54
55         if (currentState.getArc(i).getNextState().isFinalState
56             ()) {
57             appendWithTab("if(pos==token.length) {return
58                 result;}", 4);
59         }
60
61         if( currentState.getArc(i).getNextState().getNumArcs()
62             <= 0) {
63             appendWithTab("return (pos!=token.length) ? -1 :
64                 result;", 4);
65             continue;
66         }
67
68         appendWithTab("return state_" + currentState.getArc(i)
69             .getNextState().getId() +
70             "(token, pos, result);", 4);
71
72         nextStates.add(currentState.getArc(i).getNextState());
73     }
74     appendWithTab("default:", 3);
75     appendWithTab("return -1;", 4);
76     appendWithTab("}", 2);
77     appendWithTab("}", 1);
78
79     for (State next: nextStates) {
80         if (doneStates.contains(next)) {
81             continue;
82         }
83         generateCases(next, doneStates);
84         doneStates.add(next);
85     }
86
87     private void generateTokenLengthTest(int tab) {
88         appendWithTab("if(pos>=token.length) {return -1;}", tab);
89     }

```

```
85 |     private void append(String strToAppend) {
86 |         strBuff.append(strToAppend);
87 |         strBuff.append("\n");
88 |     }
89 |
90 |     private void appendWithTab(String strToAppend, int numberOfTab
91 |         ) {
92 |         for (int i = 0; i < numberOfTab; i++) {
93 |             strBuff.append("\t");
94 |         }
95 |         strBuff.append(strToAppend);
96 |         strBuff.append("\n");
97 |     }
98 | }
```

Listing 11 – Code Java générant une structure switch avec fonctions