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

23 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 Adopter le travail précédent en Java . . . 6
2.4 Générer une structure de switch imbriqués en Java . . . 6

3 Générer une FST **7**

4 Générer une structure switch en bytecode **8**

5 Analyse **9**

Conclusion **10**

A Annexe : tests avec un script shell **11**
A.1 Dictionnaires . . . 11
A.2 FST . . . 12
A.3 Générer à la volée du code C . . . 13
A.3.1 Script shell . . . 13
A.3.2 Code C généré pour la FST définie dans le Tableau 3 . . . 15
A.3.3 Code C généré pour la FST définie dans le Tableau 4 . . . 18
A.4 Générer à la volée du code assembleur x86 . . . 20
A.4.1 Script shell . . . 20
A.4.2 Code assembleur x86 généré pour la FST définié dans le Tableau 3 . . . 24
A.4.3 Code assembleur x86 généré pour la FST définié dans le Tableau 4 . . . 28

B Annexe : générer du code java dynamiquement **31**
B.1 Dans une seule méthode . . . 31
B.2 Dans différentes méthodes . . . 31

Listings

1 Un example de code généré pour un état 5
2 Script pour générer un code C à la volée d'une FST 13
3 Code C généré pour la FST définie dans Tableau 3 15
4 Code C généré pour la FST définie dans Tableau 4 18
5 Script pour générer un code assembleur x86 à la colée d'une FST 20
6 Code assembleur x86 généré pour la FST définie dans Tableau 3 24
7 Code assembleur x86 généré pour la FST définie dans Tableau 4 28

Liste des tableaux

1 Dictionnaire à utiliser avec la FST dans le Tableau 3 11
2 Dictionnaire à utiliser avec la FST dans le Tableau 4 11
3 FST utilisée avec le dictionnaire Tableau 1, voir Figure 1 page 12 12
4 FST utilisée avec le dictionnaire Tableau 2, voir Figure 2 page 12 12

Table des figures

1 La FST associée avec le Tableau 3 page 12 12
2 La FST associée avec le Tableau 4 page 12 12

Introduction

Les FST – Finite State Transducers ou en français Transducteur fini – sont des automates finis particuliers puique possédant une sortie. Ils sont enormément 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 disctionnaire 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 ingénérie 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 acquit.

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 3 et Tableau 4) ou un graphe (voir Figure 1 et Figure 2 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 méconnaissances 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 maitrisons mieux pour générer du text : bash. Par un découragement général nous avons presqu'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 maitrisions 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 opur 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'alorithme complexe : simplicité et naïvité 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 généation d'une fonction contenant de multiples labels correspondant chacun à un état (ou nœud) et un switch qui, suivant la lettre courrante 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 chaine (tableau de char) – qui sera le token d'entrée dont on veut connaitre 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 appellé "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.

```
NODE_7 :
    switch(token[pos++i]) {
        case 'X' :
            total += 6;
            goto NODE_21;
        case 'Y' :
            goto NODE_42;
        default :
            return -1;
    }
```

Listing 1 – Un example de code généré pour un état

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

$$./gen.sh\ file.fst \mid gcc\ -x\ c\ -o\ fst -$$

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 2
— Premier example de code C obtenu : Listing 3
— Second example de code C obtenu : Listing 4

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 éxécutable 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, nottamment 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 5
— Premier example de code C obtenu : Listing 6
— Second example de code C obtenu : Listing 7

2.3 Adopter le travail précédent en Java

2.4 Générer une structure de switch imbriqués en Java

3 Générer une FST

4 Générer une structure switch en bytecode

5 Analyse

Conclusion

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 1 – Dictionnaire à utiliser avec la FST dans le Tableau 3

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

Tableau 2 – Dictionnaire à utiliser avec la FST dans le Tableau 4

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 3 – FST utilisée avec le dictionnaire Tableau 1, voir Figure 1 page 12

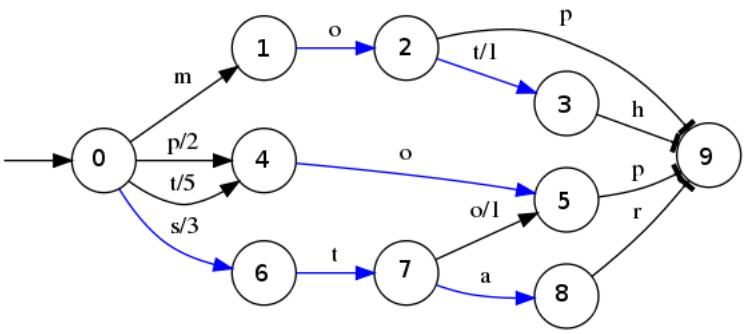

FIGURE 1 – La FST associée avec le Tableau 3 page 12

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 4 – FST utilisée avec le dictionnaire Tableau 2, voir Figure 2 page 12

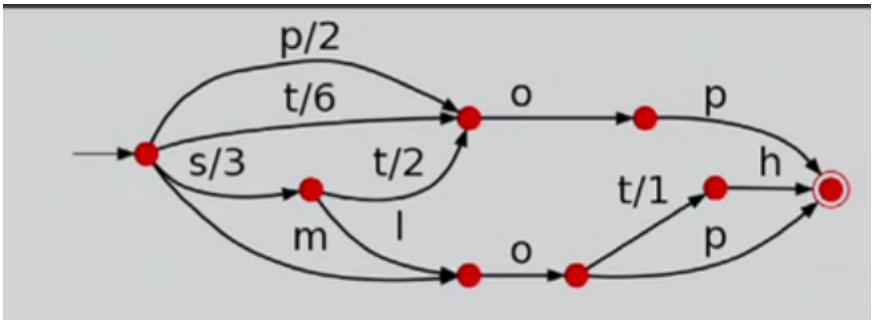

FIGURE 2 – La FST associée avec le Tableau 4 page 12

Les

A.3 Générer à la volée du code C

A.3.1 Script shell

```
#!/bin/bash

FST="$1"

sort "$FST" > "$FST.sort"

FIRST_CALL=1

cat <<EOF
#include <stdio.h>

int compute_fst(const char* token)
{
    int pos=0;
    int total=0;

EOF

while read DEP ARR CHAR WEIGHT ; do
    # If it's a final node
    if [[ ! "$CHAR" ]]; then
        WEIGHT=${ARR:-0}
        cat <<EOF
    default:
        return -1;
    }

NODE_$DEP :
EOF
    (( WEIGHT != 0 )) &&
        echo "    total += $WEIGHT;"
    echo "    if (token[pos] != '\\0') return -1;"
    echo "    goto END;"

        continue
    fi

    : ${WEIGHT:=0}

    if [[ $DEP != $PREV_DEP ]]; then
        if [[ ! "$FIRST_CALL" ]]; then
            cat <<EOF
    default:
        return -1;
    }

EOF
        fi

        cat <<EOF
NODE_$DEP :
    switch (token[pos++]) {
EOF
    fi

    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 2 – 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 3

```c
#include <stdio.h>

int compute_fst(const char* token)
{
    int pos=0;
    int total=0;

NODE_0 :
    pos++;
    switch (token[pos-1]) {
    case 'M':
        goto NODE_1;
    case 'P':
        total += 2;
        goto NODE_4;
    case 'T':
        total += 5;
        goto NODE_4;
    case 'S':
        total += 3;
        goto NODE_6;
    default:
        return -1;
    }

NODE_1 :
    pos++;
    switch (token[pos-1]) {
    case 'O':
        goto NODE_2;
    default:
        return -1;
    }

NODE_2 :
    pos++;
    switch (token[pos-1]) {
    case 'T':
        total += 1;
        goto NODE_3;
    case 'P':
        goto NODE_9;
    default:
        return -1;
    }

NODE_3 :
    pos++;
    switch (token[pos-1]) {
    case 'H':
        goto NODE_9;
    default:
        return -1;
    }

NODE_4 :
    pos++;
    switch (token[pos-1]) {
```

```
59      case 'O':
60          goto NODE_5;
61      default:
62          return -1;
63      }
64
65  NODE_5 :
66      pos++;
67      switch (token[pos-1]) {
68      case 'P':
69          goto NODE_9;
70      default:
71          return -1;
72      }
73
74  NODE_6 :
75      pos++;
76      switch (token[pos-1]) {
77      case 'T':
78          goto NODE_7;
79      default:
80          return -1;
81      }
82
83  NODE_7 :
84      pos++;
85      switch (token[pos-1]) {
86      case 'O':
87          total += 1;
88          goto NODE_5;
89      case 'A':
90          goto NODE_8;
91      default:
92          return -1;
93      }
94
95  NODE_8 :
96      pos++;
97      switch (token[pos-1]) {
98      case 'R':
99          goto NODE_9;
100     default:
101         return -1;
102     }
103
104 NODE_9 :
105     goto END;
106
107 END :
108     return total;
109 }
110
111 int main(int argc, const char *argv[])
112 {
113     if (argc < 2)
114         return 1;
115
116     printf("%d\n", compute_fst(argv[1]));
117     return 0;
```

```
118 || }
```

Listing 3 – Code C généré pour la FST définie dans Tableau 3

A.3.3 Code C généré pour la FST définie dans le Tableau 4

```c
#include <stdio.h>

int compute_fst(const char* token)
{
    int pos=0;
    int total=0;

NODE_0 :
    pos++;
    switch (token[pos-1]) {
    case 'P':
        total += 2;
        goto NODE_1;
    case 'T':
        total += 6;
        goto NODE_1;
    case 'S':
        total += 3;
        goto NODE_3;
    case 'M':
        goto NODE_4;
    default:
        return -1;
    }

NODE_1 :
    pos++;
    switch (token[pos-1]) {
    case 'O':
        goto NODE_2;
    default:
        return -1;
    }

NODE_2 :
    pos++;
    switch (token[pos-1]) {
    case 'P':
        goto NODE_7;
    default:
        return -1;
    }

NODE_3 :
    pos++;
    switch (token[pos-1]) {
    case 'T':
        total += 2;
        goto NODE_1;
    case 'L':
        goto NODE_4;
    default:
        return -1;
    }

NODE_4 :
    pos++;
    switch (token[pos-1]) {
```

```
59      case 'O':
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 4 – Code C généré pour la FST définie dans Tableau 4

A.4 Générer à la volée du code assembleur x86

A.4.1 Script shell

```
#!/bin/bash

FST="$1"
oIFS=$IFS
: ${ASM_SWITCH:=1}
(( ASM_SWITCH == 0 )) && unset ASM_SWITCH

DO_SWITCH=
: ${SWITCH_LIMIT:=2}

sort "$FST" > "$FST.sort"

FIRST_CALL=1

compute_switch() {
    ARR_ASCII=( $(awk 'BEGIN{for(n=0;n<256;n++)ord[sprintf("%c",n)]=n}
    /^'$DEP'/{print ord[$3]}' "$FST.sort" | sort) )
}

next() {
    PREV_DEP=$DEP
    FIRST_CALL=

    continue
}

cat <<EOF
        .file   "${FST%.*}.c"
        .text
        .globl  compute_fst
        .type   compute_fst, @function

compute_fst:
        pushq   %rbp            # remember old base pointer
        movq    %rsp, %rbp      # set new base pointer
        movq    %rdi, -24(%rbp) # put content of rdi (token) in -24(%rbp)
        movl    \$0, -8(%rbp)    # pos   (-8(%rbp)) = 0
        movl    \$0, -4(%rbp)    # total (-4(%rbp)) = 0

EOF

while read DEP ARR CHAR WEIGHT ; do
    # If it's a final node
    if [[ ! "$CHAR" ]]; then
        if [[ ${#tmp[@]} != 0 ]] ; then
            IFS=$'\n'
            echo "${tmp[*]}"
            IFS=$oIFS
            tmp=()
        fi

        WEIGHT=${ARR:-0}

        [[ ! "$DO_SWITCH" ]] &&
            cat <<EOF
        movl \$-1, %eax          # default : return -1
```

```
        jmp .RET
EOF
        echo -e ".NODE_$DEP:"

    (( WEIGHT != 0 )) &&
        echo "  addl  \$$WEIGHT, -4(%rbp)          # total += Weight"
    echo "      jmp .END                # goto END"

        continue
    fi

    : ${WEIGHT:=0}

    # Change of node
    if [[ $DEP != $PREV_DEP ]]; then
        if [[ ! "$FIRST_CALL" && ! "$DO_SWITCH" ]]; then
            cat <<EOF
        movl \$-1, %eax           # default : return -1
        jmp .RET

EOF
        fi

        if [[ ${#tmp[@]} != 0 ]] ; then
            IFS=$'\n'
            echo "${tmp[*]}"
            IFS=$oIFS
            tmp=()
        fi

        cat <<EOF
.NODE_$DEP:
        addl    \$1, -8(%rbp)     # pos++
        movl    -8(%rbp), %eax    # eax = pos
        leaq    -1(%rax), %rdx    # rdx = pos - 1
        movq    -24(%rbp), %rax   # load token (address) in rax
        addq    %rdx, %rax        # rax = &(token[pos-1])
        movzbl  (%rax), %eax      # eax = token[pos-1]

EOF
        if [[ "$ASM_SWITCH" ]] ; then
            DO_SWITCH=
            compute_switch
            (( ${#ARR_ASCII[@]} > SWITCH_LIMIT )) &&
                DO_SWITCH=1
        fi
    fi

    printf -v CHAR_INT '%d' "\"$CHAR"


    [[ ! "$DO_SWITCH" ]] &&
        echo "  cmpl \$$CHAR_INT, %eax           # case '$CHAR'"

    if (( WEIGHT != 0 || DO_SWITCH == 1 )) ; then
        tmp+=(
            ".NODE_${DEP}_$CHAR:"
            "   addl  \$$WEIGHT, -4(%rbp)       # total += $WEIGHT"
            "   jmp .NODE_$ARR"
            ""
```

```
117             )
118
119         if [[ "$DO_SWITCH" ]]; then
120             [[ $DEP == $PREV_DEP ]] && next
121
122             echo "       subl    \$${ARR_ASCII[0]}, %eax        # eax -= '$(
                    printf '%c' "$CHAR")'"
123             # MAX - MIN
124             CHAR_RANGE=$(( ARR_ASCII[${#ARR_ASCII[@]}-1] - ARR_ASCII[0] ))
125
126             cat <<EOF
127         cmpl \$$CHAR_RANGE, %eax            # eax = '$(printf '%c' "$(printf "\
                x$(printf "%x" ${ARR_ASCII[${#ARR_ASCII[@]}-1]})")")' - '$(printf
                "\x$(printf "%x" ${ARR_ASCII[0]})")' (max - min)
128         ja .END
129         movq .NODE_${DEP}_SW(,%rax,8), %rax
130         jmp *%rax
131
132     .section .rodata
133 .NODE_${DEP}_SW:
134 EOF
135             j=0
136             for (( i = 0; i <= (ARR_ASCII[${#ARR_ASCII[@]}-1] - ARR_ASCII[0])
                    ; i++ )); do
137                 if (( ARR_ASCII[0] + i == ARR_ASCII[j] )) ; then
138                     echo "       .quad .NODE_${DEP}_$(printf "\x$(printf "%x" $
                        {ARR_ASCII[j]})")          # '$(printf "\x$(printf "%x"
                        ${ARR_ASCII[j]})")'"
139                     (( j++ ))
140                 else
141                     echo "       .quad .ERR               # '$(printf "\x$(
                        printf "%x" $(( ARR_ASCII[0] + i )))")'"
142                 fi
143             done
144
145             echo -e "    .text\n"
146         else
147             echo "       je .NODE_${DEP}_$CHAR"
148         fi
149     else
150         echo "  je .NODE_$ARR"
151     fi
152
153     next
154 done < "$FST.sort"
155
156 cat <<EOF
157
158 .ERR:
159         movl \$-1, %eax           # return -1
160         jmp .RET
161 .END:
162         movl    -4(%rbp), %eax   # Put return value in eax
163 .RET:
164         popq    %rbp
165         ret
166         .size   compute_fst, .-compute_fst
167         .section        .rodata
168
169 .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 5 – Script pour générer un code assembleur x86 à la colée d'une FST

A.4.2 Code assembleur x86 généré pour la FST définié dans le Tableau 3

```
        .file   "3.c"
        .text
        .globl  compute_fst
        .type   compute_fst, @function

compute_fst:
        pushq   %rbp            # remember old base pointer
        movq    %rsp, %rbp      # set new base pointer
        movq    %rdi, -24(%rbp) # put content of rdi (token) in -24(%rbp)
        movl    $0, -8(%rbp)    # pos   (-8(%rbp)) = 0
        movl    $0, -4(%rbp)    # total (-4(%rbp)) = 0

.NODE_0:
        addl    $1, -8(%rbp)    # pos++
        movl    -8(%rbp), %eax  # eax = pos
        leaq    -1(%rax), %rdx  # rdx = pos - 1
        movq    -24(%rbp), %rax # load token (address) in rax
        addq    %rdx, %rax      # rax = &(token[pos-1])
        movzbl  (%rax), %eax    # eax = token[pos-1]

        cmpl $77, %eax          # case 'M'
        je .NODE_1
        cmpl $80, %eax          # case 'P'
        je .NODE_0_P
        cmpl $84, %eax          # case 'T'
        je .NODE_0_T
        cmpl $83, %eax          # case 'S'
        je .NODE_0_S
        movl $-1, %eax          # default : return -1
        jmp .RET

.NODE_0_P:
        addl  $2, -4(%rbp)      # total += 2
        jmp .NODE_4

.NODE_0_T:
        addl  $5, -4(%rbp)      # total += 5
        jmp .NODE_4

.NODE_0_S:
        addl  $3, -4(%rbp)      # total += 3
        jmp .NODE_6

.NODE_1:
        addl    $1, -8(%rbp)    # pos++
        movl    -8(%rbp), %eax  # eax = pos
        leaq    -1(%rax), %rdx  # rdx = pos - 1
        movq    -24(%rbp), %rax # load token (address) in rax
        addq    %rdx, %rax      # rax = &(token[pos-1])
        movzbl  (%rax), %eax    # eax = token[pos-1]

        cmpl $79, %eax          # case 'O'
        je .NODE_2
        movl $-1, %eax          # default : return -1
        jmp .RET

.NODE_2:
        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_O
138         cmpl $65, %eax          # case 'A'
139         je .NODE_8
140         movl $-1, %eax          # default : return -1
141         jmp .RET
142
143 .NODE_7_O:
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 6 – Code assembleur x86 généré pour la FST définie dans Tableau 3

A.4.3 Code assembleur x86 généré pour la FST définié dans le Tableau 4

```
        .file   "3.c"
        .text
        .globl  compute_fst
        .type   compute_fst, @function

compute_fst:
        pushq   %rbp            # remember old base pointer
        movq    %rsp, %rbp      # set new base pointer
        movq    %rdi, -24(%rbp) # put content of rdi (token) in -24(%rbp)
        movl    $0, -8(%rbp)    # pos   (-8(%rbp)) = 0
        movl    $0, -4(%rbp)    # total (-4(%rbp)) = 0

.NODE_0:
        addl    $1, -8(%rbp)    # pos++
        movl    -8(%rbp), %eax  # eax = pos
        leaq    -1(%rax), %rdx  # rdx = pos - 1
        movq    -24(%rbp), %rax # load token (address) in rax
        addq    %rdx, %rax      # rax = &(token[pos-1])
        movzbl  (%rax), %eax    # eax = token[pos-1]

        cmpl $80, %eax          # case 'P'
        je .NODE_0_P
        cmpl $84, %eax          # case 'T'
        je .NODE_0_T
        cmpl $83, %eax          # case 'S'
        je .NODE_0_S
        cmpl $77, %eax          # case 'M'
        je .NODE_4
        movl $-1, %eax          # default : return -1
        jmp .RET

.NODE_0_P:
        addl  $2, -4(%rbp)      # total += 2
        jmp .NODE_1

.NODE_0_T:
        addl  $6, -4(%rbp)      # total += 6
        jmp .NODE_1

.NODE_0_S:
        addl  $3, -4(%rbp)      # total += 3
        jmp .NODE_3

.NODE_1:
        addl    $1, -8(%rbp)    # pos++
        movl    -8(%rbp), %eax  # eax = pos
        leaq    -1(%rax), %rdx  # rdx = pos - 1
        movq    -24(%rbp), %rax # load token (address) in rax
        addq    %rdx, %rax      # rax = &(token[pos-1])
        movzbl  (%rax), %eax    # eax = token[pos-1]

        cmpl $79, %eax          # case 'O'
        je .NODE_2
        movl $-1, %eax          # default : return -1
        jmp .RET

.NODE_2:
        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 7 – Code assembleur x86 généré pour la FST définie dans Tableau 4

B Annexe : générer du code java dynamiquement

B.1 Dans une seule méthode

B.2 Dans différentes méthodes