Université Paris 7

UFR d'Informatique Fondamentale

Centre d'Etude et de Recherche en Informatique Linguistique
Laboratoire d'Automatique Documentaire et Linguistique

ANALYSE SYNTAXIQUE TRANSFORMATIONNELLE DU FRANCAIS PAR TRANSDUCTEURS ET LEXIQUE-GRAMMAIRE

(ANNEXE)

Emmanuel ROCHE

Thèse de Doctorat d'Informatique Fondamentale

Directeur: Maurice Gross

Janvier 1993

Jury

M. Crochemore
J. Désarménien
M.Gross
F. Günthner
D. Perrin
J-M. Staeyert

PUPITRE

Manuel de l'utilisateur

PUPITRE

Manuel de l'utilisateur

1. INTRODUCTION

Le système *PUPITRE* est destiné à la manipulation et la construction d'automates à états finis et de transducteurs de très grande taille pour la manipulation et le traitement des textes en langage naturel. Nous décrirons dans le tutorial l'application de PUPITRE à trois types de tâches :

1. La compilation de dictionnaires sous des formats permettant des manipulations très rapides,

2. L'utilisation de ces dictionnaires pour faire une analyse morphologique des textes,

3. L'analyse syntaxique du langage naturel.

Nous nous plaçons ici du point de vue de l'utilisateur qui veut compiler lui-même ses dictionnaires et tester sur des textes les analyses produites. On suppose que cet utilisateur est capable de se servir du système d'exploitation du système sur lequel il travail (UNIX ou OS2) pour les commandes simples de manipulation de fichiers et de directories. On suppose aussi qu'il sait ce qu'est un automate à états finis, si tel n'est pas le cas il pourra se référer à l'annexe 1.
La présentation qui va suivre est divisée en trois parties :

1. Un exemple de session de travail typique,

2. La liste des commandes *PUPITRE* avec leurs options,

3. Une série d'annexes spécifiant quelques problèmes standards.

2. TUTORIAL

2.0 LES PRINCIPALES FONCTIONNALITES ET LEURS COMMANDES

Un moyen simple de se familiariser avec le système est de lire la description et les exemples des commandes d'une même famille, c'est à dire des commandes servant au même type d'usage. Nous donnons ici un ensemble de regroupements naturels qui permet de mieux naviguer dans l'ensemble des commandes.

Le système *PUPITRE* est un système dont le but est double :

1. permettre la création, la manipulation et l'utilisation la plus efficace possible d'automates et de transducteurs de très grande taille et

2. utiliser ces fonctionnalités pour le traitement automatique des langues naturelles.

2.0.1 Fonctionnalités générales

On peut créer des automates soit en les décrivant directement à partir d'expressions rationnelles (commande **_rat_aut**) soit en dessinant des graphes qui les représentent (commandes **_graph_aut144** et **pauto**) soit en partant des listes de chemins (commandes **_lst_aut** et **_mnm**).

Nous décrivons précisément les différentes structures de représentation d'automates dans l'annexe 1 *Notions de bases sur les automates à états finis et leurs représentations.*

Les opérations classiques (et certaines moins classiques) se font soit en donnant des expressions rationnelles dont certains termes sont des automates et en les transformant en automate, soit directement par des commandes. Les opération unaires sont la réduction (**_reduc**), la minimisation (**_mnm**), la déterminisation (**_det**), l'indexation (**_cal_crd**) et la mise sous forme d'accès direct compacté (**_compac** , voir annexe 1). Les opérations binaires, en plus de **_rat_aut**, sont l'union (**_union**), l'intersection et l'intersection par facteurs (**_inter**), la différence et la différence par facteurs (**_diff**).

L'utilisation de ces automates se fait par **_ds_alph**, **_morpho**.

Les transducteurs se construisent à partir des automates équivalents (c'est à dire de ceux dont les étiquettes sont les couples étiquette gauche-étiquette droite des transducteurs). Cela permet donc d'utiliser l'ensemble des commandes précédentes. La transformation des automates en transducteurs et des manipulation spécifiques aux transducteurs se font par la commande **_aut_trans**. La composition de transducteurs se fait par **_compose** et l'utilisation des transducteurs, c'est à dire leur application à des séquences ou à des automates se fait par la commande **_transduc**.

2.0.2 Fonctionnalités spécifiques aux langues naturelles

Pour la construction de dictionnaires de mots simples la commande principale est **_fdic**, cette commande utilise les commandes **_mnm** et **_union**. Pour la construction des dictionnaires de mots composés la commande principale est **_fnc** qui utilise également **_mnm** et **_union**. Le résultat de ces fonctions peut être testé par la commande **_morpho** de consultation de dictionnaires.

Pour convertir un dictionnaire morphologique en un dictionnaire d'automates et permettre ainsi l'analyse morphologique des phrases, la commande principale est **_conscot**.

Une fois les fichiers *DELAF.(eic+lic+aic+dic)* et *DELAF.cot* constitués grâce aux commandes **_fdic** et **_concot** on peut faire l'analyse morphologiques et mots ou de phrases. Les commandes principales sont **_mot_aut** et **_ph_aut** qui convertissent respectivement un mot et une phrase en une représentation morphologique sous forme d'automate acyclique.

Cette représentation morphologique peut être utilisée pour rechercher des séquences morphologiques particulières (du type "un nom suivit d'un verbe conjugué à la troisième personne) par la commande **_pgrep**. Le résultat peut être affiné par l'utilisation d'une

grammaire de contraintes locales et de la commande **_plgrep**. Le repérage des mots composés se fait par la commande **_agrep**. Ces commandes doivent permettre la création d'index spécifiques pour la manipulation de corpus.

La constitution d'un dictionnaire syntaxique se fait en deux étapes. La première consiste à transformer les données de départ que sont les tables et les automates de références en une structure intermédiaire de type *DELSYN* qui regroupe toutes les tables et tous les automates de référence et permet de générer ainsi l'automate de chaque verbe (ou chaque entrée) de manière indépendante. La commande principale est **_fdelsyn** et son utilisation est détaillée dans l'annexe 4 *Construction d'un dictionnaire syntaxique de type DELSYN*, la commande **_consult** permet la consultation de cette structure La seconde étape consiste à transformer ce dictionnaire en un automate (ou transducteur) unique, appelé *AUTSYN* (ou *f_DELSYN*), qui regroupe toutes les structures de toutes les entrées du dictionnaire syntaxique en un seul automate ou transducteur. Le script de cette construction est donnée dans l'annexe 5 *Construction de l'automate syntaxique AUTSYN*. Les commandes utilisées pour cette construction sont principalement **_consult** (qui permet de manipuler la structure intermédiaire), **_union**, **_mnm**, **_reduc**, **_desabr**, **_transduc**, **_graph_autl44**.

Les dictionnaires morphologiques et syntaxique permettent l'analyse syntaxique des phrases par les commandes **_analyse** et **_parse**.

Le traitement de la morphologie dérivationnelle par transducteur consiste à la constitution d'un transducteur unique par les commandes **_aut_trans**, ... et à son utilisation par la commande **_transduc**. On a donné dans l'annexe 3 Construction d'un transducteur morphologique le script exacte de la construction d'un transducteur pour l'analyse morphologique du français.

2.1 LA CONSTRUCTION ET LA MANIPULATION DES DICTIONNAIRES

2.1.1 Transformer une liste mots en automates

Supposons que l'on ait une liste de mots décrites dans le fichier ess1.txt :

```
$ cat ess1.txt
il
elle
parti
$
```

et que l'on veuille pouvoir répondre rapidement à la question "ce mot fait-il partie de la liste ?". Commençons par transformer cette liste en automate. Il faut pour cela utiliser la commande **_lst_aut** (transforme une liste en automate) qui prend en entrée (sur l'entrée standard) le fichier qui représente la liste et qui écrit sur la sortie standard l'automate. Ainsi :

```
$ _lst_aut <ess1.txt
qs      lcO
$
```

On constate que le résultat n'est pas directement lisible. En effet toutes les structures manipulées sont sauvegardées sous des formats qui minimisent la taille de stockage mais qui ne peuvent être lus directement.

Ainsi, il faut faire:

```
$ _lst_aut <ess1.txt >aut1
```

On peut cependant regarder le résultat par la commande **_aff** (affiche). Cette commande prend en entrée n'importe laquelle des structures que nous allons manipuler et affiche le résultat.

Ainsi

```
$ _aff <aut1
automate de type AUT_L13
nombre d'états :
.
.
$
```

Cette commande donne systématiquement le type de la structure manipulée. Ici il s'agit d'un automate de type *AUT_L13*. C'est le type standard d'automates qui représente les listes de mots. Nous verrons que pour les listes nous avons quatre type d'automates possible: *AUT_L13*, *AUT_D13*, *AUT_E_L13* et *AUT_T_D13* que nous expliquerons au fur et à mesure qu'ils seront rencontrés.

Si on avait voulu regarder directement le résultat de la transformation liste automate, il aurait suffit de faire la commande :

```
$ _lst_aut <ess1.txt / _aff
```

2.1.2 Consulter l'automate

Une fois l'automate construit on peut tester les mots qui appartiennent et ceux qui n'appartiennent pas par la commande ***_ds_alph*** (*dans alphabet*). Cette commande prend en paramètre l'automate (spécifié après -aut) et en entrée un fichier de mots. En sortie, on obtient mot code où code vaut 0 si le mot n'appartient pas à l'automate et 1 sinon.

Ainsi :

```
$ _ds_alph -aut aut1
il
il 5
elle
elle 0
parti
parti 5
^D
$
```

Ici on a testé l'automate sur l'entrée standard. *Le control D* spécifie la fin du fichier. On aurait pu tester si tous les mots de ess1.txt étaient bien représentés par l'automate en faisant

```
$ _ds_alph -aut aut1 <ess1.txt
il 5
est 5
parti 5
$
```

2.1.3 Représentation de la liste de manière ordonnée

Jusqu'à maintenant la représentation *AUT_L13* (standard) ne permet de poser qu'un seul type de question: "tel mot appartient-il à la liste que décrit l'automate acyclique ?". On peut vouloir une réponse plus précise si la liste était classé par ordre alphabétique, on peut vouloir connaître le numéro d'ordre du mot qu'on demande dans l'ordre alphabétique. Par exemple ou voudrait savoir que *est* est le mot numéro 1 et *parti* le mot numéro 3. Cette question n'est pas possible directement avec la structure *AUT_L13*, il faut la transformer en un automate un peu plus complet (de type *AUT_E_L13*) où on a rajouté des informations pour chaque état. Cet enrichissement est possible avec la fonction **_cal_crd** (calcul le "cardinal" de chaque état, c'était l'information manquante). Cette commande prend l'automate de type AUT_L13 en entrée et renvoie l'automate de type AUT_E_L13 correspondant. Ainsi, dans notre exemple :

```
$ _cal_crd <aut1 >aut3
```

Sur ce type d'automate, on peut en plus de la consultation **_ds_alph** que nous faisions, donner l'option -num à **_ds_alph** qui renverra alors le numéro d'ordre du mot. Ainsi:

```
$ _ds_alph -aut aut2 <ess1.txt
il 5
est 5
parti 5
$ _ds_alph -aut aut2 -num <ess1.txt
il 2
est 1
parti 3
$
```

2.1.4 Réduction du temps d'accès et de la taille de stockage

Les deux structures que nous manipulons jusqu'à présent ne sont pas optimales du point de vue du temps d'accès et de la place nécessaire pour les stocker en mémoire (voir annexe 1). Cet inconvénient n'est pas perceptible pour les petits lexiques (quelques milliers de mots) mais peut être légèrement gênant pour les dictionnaires de taille normale (quelques centaines de milliers de mots). La structure *AUT_L13* (respectivement *AUT_E_L13*) peut donc être traduite en une structure plus appropriées : *AUT_D13* (respectivement *AUT_T_D13*). Leur construction se fait par la commande ***_compac*** de la manière suivante:

```
$ _compac <aut1 >aut3
$ _compac <aut2 >aut4
```

On pourra faire sur *aut3* et *aut4* les mêmes opérations, avec les même commande, (2.1.1 à 2.1.3) que sur *aut1* et *aut2*.

les structures AUT_D13 et AUT_T_D13 sont celles qui sont couramment utilisées pour représenter les dictionnaires.

2.1.5 Faire l'union de deux automate représentant deux listes

Supposons que l'on ait deux automates de type AUT_L13 qui représentent chacun une liste de mots (par exemple un dictionnaire et des ajouts à celui-ci). Par exemple, dans le cas suivant:

```
$ cat ess2.txt
et
cela
aussi
$ _lst_aut <ess2.txt >aut5
```

on peut réunir aut5 et aut1 en un seul automate aut6 qui représente l'union des deux ensembles ess1.txt et ess2.txt. La commande qui effectue cette union est **_union**.

```
$ _union aut1 aut5 >aut5
$ _ds_alph -aut aut5 <ess1.txt
il 5
est 5
parti 5
$ _ds_alph -aut aut5 <ess2.txt
et 5
cela 5
aussi 5
$
```

2.1.6 Minimiser un automate

Cependant l'automate obtenu par l'union n'a pas un nombre d'états minimal. C'est à dire qu'on peut trouver un autre automate qui lui est strictement équivalent et qui a un nombre plus réduit d'états et occupe donc moins de place mémoire. Cette opération s'appelle minimisation d'automate et se fait avec la commande **_mnm**.

```
$ _mnm <aut5 >aut6
$ _ds_alph -aut aut6 <ess1.txt
il 5
est 5
parti 5
$
```

2.1.7 Représenter des entrées plus leur information

Pour l'instant on n'a manipulé que des listes de mots. On ne dit rien sur les mots représentés. Il est cependant possible, avec les mêmes types d'automates, d'associer un nombre à chaque mot représenté. Cela permet de représenter des dictionnaires quelconques. En effet, si on a le dictionnaire dic1 :

```
$ cat dic1
il,pro
est,verbe
parti,verbe participe passe
$
```

Si on a une table de type *dic1.cod*

```
$ cat dic1.cod
pro
verbe
verbe participe passe
$
```

et si on associe à *il*, *est* et *parti* respectivement 1 2 et 3, il est clair que l'on peut retrouver l'information complète du mot. Ainsi le problème devient celui de représenter un dictionnaire pour lequel on aura associé à chaque mot un nombre. On utilise pour cela la commande **_lst_aut** avec l'option *-1*:

```
$ cat dic2
il,1
est,2
parti,3
$ _lst_aut -1 -cod <dic2 >aut7
```

Cette fois, si on applique la commande *_ds_alph*, on obtient le code du mot, ainsi:

```
$ _ds_alph -aut aut7 <ess1.txt
il,1
est,2
parti,3
$
```

2.1.8 Construction complète d'un dictionnaire

En utilisant en série les commandes précédentes, il est possible de construire une représentation par automate d'un dictionnaire complet. Cependant, pour simplifier ce travail, il existe une commande qui permet de construire directement une telle représentation. Il s'agit de la commande **_fdic** qui prend en argument le fichier représentant le dictionnaire et qui construit l'automate et une table de correspondance.

Voir annexe.

2.1.9 Consultation du dictionnaire

Le dictionnaire construit par la commande **_fdic** peut être consulté par la commande **_morpho**. Cette commande prend en paramètre le path qui permet d'accéder au dictionnaire. Ainsi si le dictionnaire est stocké sous */users/eroche/donnee*, on peut effectuer la manipulation suivante :

```
$ _morpho D/users/eroche/donnee/dico <ess1.txt
il,Pro
est,être.V34:P3s,est.Adj:ms:mp:fp:fp,est.N:ms
parti,partir.V5:P3s
$
```

Ce paramètre peut être omis si la variable d'environement DELAF est définie par *set DELAF=/users/eroche/donnee/dico*.

2.1.10 Déterminisation d'un automate

Il est possible d'effectuer un certain nombre d'autres opérations qui ne sont pas directement utiles à la construction de dictionnaires. On peut par exemple déterminiser un automate de type AUT_L13 par la commande **_det** :

```
$ _det <aut7 >aut8
```

Ici aut7 et aut8 sont rigoureusement identiques car les automates construits jusqu'à maintenant sont tous déterministes.

2.2 LES AUTOMATES A ALPHABET DE MOTS

Les listes que nous avons représentées jusqu'à présent étaient des listes de mots simples et il était naturel de les décrire par des automates dont les transitions sont des lettres. Cependant, dans certains cas, il peut sembler plus naturel de décrire des listes avec des automates dont les transitions, et donc dont l'alphabet, sont des mots (c'est à dire une suite de caractères).

Prenons l'exemple d'un dictionnaire de mots composés dans lequel on a *pomme de terre* et *pommes de terre*, on peut bien sûr représenter ces entrées par un automate de type AUT_L13 comme dans l'exemple pris au début mais il semble plus naturel de procéder autrement.

2.2.1 Construction de l'automate

Tout peut se faire par la commande **_fnc** mais il est possible de décomposer les étapes en utilisant des commandes plus générales comme nous allons le voir ici. Supposons que l'on ait une liste de mots composés dans le fichier ess2.txt:

```
$ cat ess3.txt
pomme de terre
pommes de terre
$
```

La première étape de construction de ce dictionnaire consiste à extraire la liste des mots simples et à la mettre sous forme d'automate. La commande **_ext_mot_s** extrait les mots simples d'un lexiques ou d'un dictionnaire de mots composés. On peut extraire cette liste et la constituer en automate en une seule étape:

```
$ _ext_mot_s <ess3.txt / _lst_aut /_cal_crd /_compac >aut9
$
```

Cet automate aut5 est un automate qui établit un lien entre chaque mot simple et son numéro d'ordre, il constitue un alphabet. Cependant les alphabets doivent en plus avoir la propriété qu'on puisse leur ajouter des mots nouveaux. On peut transformer cette liste figée ordonnée en un alphabet par la commande _aut_alph de la manière suivante :

```
$ _aut_alph <aut9 >alph1
```

On peut alors construire l'automate qui représente la liste des mots composés en utilisant la commande *_lst_aut* avec l'option -mc (mots composés) et en précisant l'alphabet employer avec -alph :

```
$ _lst_aut -mc -alph alph1 <ess3.txt >aut10
$ _aff <aut10
automate de type AUT_L44
nombre d'états :
. .
$
```

2.2.2 Réduction du temps d'accès

Comme pour les automates de type AUT_L13 il est possible de réduire sensiblement le temps d'accès pour les gros dictionnaires par la commande _compac

```
$ _compac <aut10 >aut11
$ _ds_alph
```

Dans ce cas l'automate obtenu est de type AUT_D44.

2.2.3 Modification de l'alphabet de l'automate.

Il est possible de modifier l'alphabet de l'automate, on peut par exemple vouloir lui rajouter un mot simple, par exemple *poire*. Cela ne modifie pas l'ensemble représenté mais cela permet de préparer des opérations ultérieures. Par exemple si l'on veut comparer deux automates on peut vouloir qu'ils aient le même alphabet. Il faudra donc rajouter à l'un tous les mots présents dans l'autre qui lui manquent.

Pour cela il faut procéder en trois étapes: (i) on extrait l'alphabet de l'automate sous la forme ALPH_T_D13, (ii) on modifie l'alphabet ainsi obtenu et (iii) on met l'automate d'origine à jour pour ce nouvel alphabet:

```
$ _ext_alph <aut11 >alph2
$ _aj_mot -alph alph2 >alph3
poire
^D
$ _mod_alph aut11 alph3
$
```

2.2.4 Représentation des ambiguïtés morphologiques dans une phrase.

Un exemple typique d'automate de type AUT_L44 que nous venons de présenter est le cas où on veut représenter l'ensemble des ambiguïtés morphologiques d'un mot ou d'une phrase. Ainsi, dans le cas suivant:

```
$ _mot_aut
le
.
..
$
```

Cette commande transforme donc un mot en un automate qui représente exactement ses ambiguïtés morphologiques. Il est clair que cette fonction utilise un dictionnaire dont le chemin, comme pour la commande _morpho, aura été donné par la variable d'environnement DELAF ou par le paramètre Dpath. Une commande existe pour des phrases entières et permet d'obtenir un automate de structure AUT_L44 qui pourra être manipulé par la suite:

```
$ cat ess4.txt
le président
$ _ph_aut <ess4.txt >aut12
$ _aff <aut12
..
..
$
```

Cette commande, comme _morpho et _mot_aut utilise le dictionnaire. Il faut donc préciser où le trouver de la même manière.

2.2.5 Construction de tous les automates d'un texte.

La commande précédent permet pratiquement directement de construire tous les automates d'un texte. Cependant il faut quand même découper les textes en phrases et séparer les mots des séparateurs par exemple. La commande ***_dec_txt***(pour *découpe textes*) effectue cela. Si on l'appelle en série avec la construction des automates on obtient la liste des automates:

```
$ _dec_txt <roman / _ph_aut /_aff
```

va donc afficher la liste de ces automates.

2.2.6 Recherche d'une séquence dans un texte par intersection

Si on recherche dans un texte des séquences particulières tels que : un nom au singulier, il suffit de rechercher dans les automates des phrases les séquences *lex n ?? ?? ?? ?? s ??* et d'extraire les données pertinentes. Le symbole *??* signifie que l'on accepte n'importe quel mot à cet emplacement. Si l'automate aut14 représente la liste des patterns recherchés, il faut faire l'intersection de cet automate avec l'automate du texte. Le résultat de cette opération est une série d'automates de type RES_INTER, ce sont des automates de type AUT_L44 auxquels ont donne les références des états des automates qui sont à son origine.

La commande ***_inter*** permet de réaliser cette opération. Il faut l'utiliser dans ce cas avec l'option -f (pour signifier que l'on autorise les séquences décrites par le second automate à débuter n'importe où : f est mis pour facteur).

```
$ _aff <aut14
automate de type AUT_L44
nombre d'états :
..
$ _inter -f autph aut14 >res
$ _chemin <res
$ _chemin -clair <res
```

Notons que l'on peut utiliser ces fonctions en série. La commande :

```
$ _ph_aut /_inter -f autl14 /_aff
```

donne l'ensemble des automates trouvés.

3. LES COMMANDES

Nous donnons ici la listes des commandes du système

_AFF

Introduction

Cette commande affiche la structure représentée sur le fichier donné en entrée.

Présentation

_aff <FILE

Affiche les données à l'écran (sortie standard) de manière lisible. Pour les structures

AUT_L13

AUT_E_L13

AUT_D13

AUT_T_D13

ALP_T_D13

_DELCOT

DELSYN

AUT_L44

AUT_D44

_aff -lst <aut

Affiche la liste des chemins de l'automate pour

AUT_L13

AUT_E_L13

AUT_T_D13

AUT_L44

_aff -mem <FILE

Affiche la taille de la donnée écrite sur FILE.

Exemple

```
$ _aff <aut1
automate de type AUT_L44
Nb d'états: 2
0,0: - aaa ->1, - bbb ->1
1,5:
$ _aff -lst <aut1
aaa
bbb
$ _aff -mem <aut1
Nb etats: 2
Nb Transitions:2
Taille mémoire: 2*16+2*16=64
$
```

[_ACCESS]

Introduction

Accède dans des positions de textes d'après leur index.

Présentation

[_access] TXT_FILE IND_FILE [-lc NUM] <LST_FILE

Pour les séquences de caractères du fichier LST_FILE la commande affiche les contextes des mots trouvés dans le texte TXT_FILE d'après l'index décrit dans IND_FILE.

[_access] TXT_FILE IND_FILE -inline

Donne l'accès à un interpréteur shell qui permet de consulter et le texte et son index.

Présentation du shell

ac$ pomme/de/terre cont

Affiche les *maxoc* premiers contextes de *pomme/de/terre* si cette séquence est un élément de l'index.

ac$ 52 lc

Fixe la longueur du contexte à 52 caractères à gauche et à droite.

ac$ 30 maxoc

Fixe la longueur des nombres d'occurrences à afficher

ac$ FILE_AUTL44 load

Charge et met au sommet de la pile l'automate de type AUT_L44 décrit dans le fichier FILE_AUTL44. Cet automate peut être utiliser par la commande cont (donne les contextes de chacune des séquences de l'automate) ou ind int (intersection avec l'indexe, i.e. donne les séquences de l'automate qui sont également décrites dans l'indexe.

Exemple

Commandes associées

Les index se construisent avec les commandes *_agrep*, *_pgrep*, *_plgrep* et *_index.*

_AGREP

Introduction

Cherche dans un texte les séquences de mots décrites dans un automate. Cette commande diffère de *_pgrep* et *_plgrep* en ce qu'il n'y a pas d'analyse morphologique préalable du texte. C'est à dire qu'on peut chercher manger, mangions mais on ne peut spécifier un pattern du type *[<manger>v.v-t]* qui est l'ensemble des formes conjuguées du verbe manger. Si la liste des séquences est un dictionnaire (de mots composés par exemple) cela permet de chercher tous les éléments de ce dictionnaire dans les textes. Utilisé avec un dictionnaire de mots composés cette commande permet de construire un index du texte (conjointement avec les commandes *_index* et *_access*: voir l'exemple).

Présentation

_agrep AUT_FILE (-ind IND_FILE) <TXT_FILE

Affiche a l'écran le texte du fichier *TXT_FILE* dans lequel les séquences de *AUT_FILE* sont mises entre crochets. si un fichier IND_FILE est spécifier les mots détectés seront mis dans ce fichier avec leur position (distance en caractère à l'origine). Cette commande fonctionne pour des automates de type

AUT_D44

[AUT_L44]

[AUT_L13]

[AUT_D13]

[ALP_T_D13]

[_agrep AUT_FILE -1 <TXT_FILE]

Fait la même chose mais uniquement pour les mots qui ne sont pas des éléments de l'automate.

_agrep aut -lst (-1) <TXT_FILE

Donne la liste des séquences détectés avec leur position dans le texte. Cette commande est utilisée pour construire l'indexe d'un texte (voir exemple).

[_agrep AUT_FILE -context NUM <TXT_FILE]

Donne la liste des séquences détectées mise dans leur contexte (contexte de taille NUM caractères).

[_agrep AUT_FILE -lstph <TXT_FILE]

Donne la liste des mots avec le numéro de la pseudo-phrase dans laquelle ils apparaissent.

Exemple

$ _agrep delacf.lic -lst <TXT_FILE /awk -F, '{print $1}' / _index -count /_aff -lst

Affiche la liste des mots composés détectés dans le texte avec en regard leur nombre d'apparition dans le texte.

$ _agrep delacf.lic -lst <TXT_FILE /_index >IND_FILE

Construit l'index des mots composés du texte TXT_FILE.

_ANALYSE

Introduction

Analyse syntaxiquement des phrases d'après les dictionnaire morphologiques delaf, delacf et une grammaire sous forme de transducteur

Présentation

_analyse [D/dlaf] [G\gram] <TXT_FILE >LST_AUT

Prend un texte découpé avec une phrase par ligne et renvoie la liste des automates correspondant à leur analyse.

_analyse -simple1 <TXT_FILE

Renvoie l'expression parenthèsée des analyses

_analyse -proj2 <TXT_FILE

Renvoie l'expression parenthèsée avec les labels

_analyse [D/dlaf] [G\gram] -sh

Lance un shell d'analyse: voir paragraphe suivant

Présentation du shell

string ph

charge la phrase dans une pile de phrase

affph

affiche la pile de phrases

morpho

analyse morphologiquement la phrase du sommet de la pile

step

applique la grammaire une fois

simple1

projette le dernier automate pour voir les structures parenthésées

ana

fait l'analyse syntaxique de la phrase du sommet de la pile

_AUT_ALPH

Introduction

Construit l'alphabet de type ALP_T_D13 à partir d'un automate compacté de type AUT_D_13. La différence principale entre l'automate de type AUT_T_D13 et l'alphabet de type ALP_T_D13 est que l'alphabet peut être augmenter dynamiquement. (soit par la commande _aj_alph ou par la fonction ordaj_alptd13).

Présentation

_aut_alph <aut >alph

Prend l'automate de type AUT_T_D13 et en fait un alphabet de type ALP_T_D13.

Exemple

```
$ _aff -lst <aut
max
gaston
$ _aut_alph <aut >alph
$ _aj_mot -alph alph
luc
$ _aff -lst <alph
max
gaston
luc
$
```

_AUT_TRANS

Introduction

Manipulation de transducteurs et conversion d'automates en transducteurs.

Présentation

L'automate d'entrée représentant un transducteur a des transitions de type "*a:b*" où *a:b* est une lettre de l'alphabet. Il faut convertir cela en un transducteur dont l'alphabet contiendra "a" et "b" séparément. On se reportera à l'annexe 3 pour voir le détail de la construction d'un transducteur à travers un exemple complet. Il faut donc d'abord convertir l'alphabet:

_aut_trans <alph1 >alph2

passe d'un alphabet contenant *a:b* en un autre contenant *a* et *b*

_aut_trans -alph alph2 <aut >trans

Si alph2 est l'alphabet qui vient d'être calculé, l'automate (AUT_L44) est converti en un transducteur (TRANS_L44).

_aut_trans -gauche_atout <aut >trans

Construit le transducteur qui fait la transduction *L(aut) <-> A**.

_aut_trans -etend_suffixe <trans1 >trans2

Le résultat vérifie *L(trans2)=L(trans1).A**, c'est à dire que si a:a est élément de trans1, *a:a, a:a/c:c/ a:a/b:b/b:b..* est également élément de trans2.

_aut_trans -etend_prefixe <trans1 >trans2.

Rajoute la boucle @:@ sur l'état initial. Donc si m<->n par trans1, A*m<->n par trans2.

_aut_trans -complete alph <aut1 >aut2

Traite les transitions @:a, a:@ et @:@ en les remplaçant par toutes les étiquettes décrites dans alph. Par exemple si @:@ boucle sur l'état initial, pour toutes les lettres de alph (de type a:b) qui ne sont pas des étiquettes d'autres transitions partant de l'état initial, on rajoute la transition ayant cette étiquette et bouclant sur l'état initial. Pour chaque état on ne rajoute pas des transitions pour des lettres qui existent déjà.

_CAL_CRD

Introduction

Cette commande prend en entrée un automate non indexé et construit l'automate indexé.correspondant

Présentation

Les automates de type AUT_E_L13 peuvent être compactés en automate de type AUT_T_D13 avec *_compac* puis servir de base à un alphabet de type ALP_T_D13 avec *_aut_alph.*

_cal_crd <aut1 >aut2

Calcul l'index d'un automate indexé (voir annexe 1 pour une explication sur la structure d'automates indexés).

Exemple

```
$ _aff -mem <aut1
Automate de type AUT_L13
78000 etats
150242 transitions
$ _cal_crd <aut1 >aut2
$ _ds_alph -num -aut aut2
zouaves
zouaves,578443
a
a,25
^D$
```

_CHEMIN

Introduction

Test un chemin dans un automate.

Présentation

_chemin -aut aut <FILE

Donne le chemin suivie par les chaînes de caractères décrites dans FILE.

Exemple

```
$ cat f1
le/plus/petit
$ _chemin -aut aut <f1
[0,0] le [7,0] plus [123,0] petit [14536,5]
```

signifie que le chemin existe dans aut, qu'il suit les états 0,7,123 puis 14536 qui sont tous non terminaux sauf le dernier.

_CHG_ALPH

Introduction

Change l'alphabet d'un automate.

Présentation

_chg_alph alph <aut1 >aut2

aut2 est le même automate que aut1 au sens où L(aut1)=L(aut2), mais la numérotation des transitions peut changer (voir la description des automates à deux niveaux de type AUT_L44 Annexe 1). Les mots étiquetant aut1 qui ne sont pas dans alph sont rajouté à cet alphabet au cours de l'opération.

ATTENTION: le changement d'alphabet implique que les transitions ne sont plus rangées dans l'ordre croissant. Pour les réordonner, il faut employer la fonction *ordonne_l44*.

_COMPAC

Introduction

Cette commande prend en entrée un automate dont les transitions sont des listes et construit la représetation en accès direct compacté correspondante.

Présentation

_compac <aut1 >aut2

Construit l'automate à accès direct compacté (voir annexe 1) correspondant à l'automate en liste aut1. Marche pour les automates de type

AUT_L13

AUT_E_L13

AUT_L44

_COMPOSE

Introduction

Fait la composition de deux transducteurs

Présentation

_compose trans1 trans2 >trans3

```
Fait la composition trans2 o trans1.
```

_CONSCOT

Introduction

SPECIFIQUE A L'ANALYSE MORPHOLOGIQUE. Cette commande permet de transformé un delaf.cod, qui contient donc des codes morphologiques en extension du type 3er,V3.P1p en un delaf.cot qui est la liste de tous les automates possibles représentant les ambiguïtés morphologiques des entrées. Il n'est pas nécessaire de comprendre la forme du résultat mais seulement que delaf.cot est une entrée nécessaire des commande *_ph_aut* ou *_mot_aut.*

Présentation

_conscot -2pr delaf.cod tmp.cod

enlève les deux premières lignes de l'input

_conscot -tp 1 -alph alph.tmp -f delaf.cod -dep 0 -l 500

let dans res.tmp les automates des entrées sous une forme non compacté.L'option -tp précise la langue et la version du dictionnaire utilisé:

-tp 1 signifie "français première version"

_conscot -folker <in >out

met les codes dans le formalisme cat2.

_CONSULT

Introduction

SPECIFIQUE A LA CONSULTATION DE DICTIONNAIRES SYNTAXIQUES. Cette commande permet de consulter des dictionnaires syntaxiques de type _DELSYN.

Présentation

_consult [-l44] <*mange-1* >aut

Renvoie l'automate de type AUT_LT44 ou AUT_L44 correspondant à cette entrée.

_consult <*mange* >les_auts

Renvoie la liste des automates correspondants à toutes les entrées de ce verbe.

_consult -table num

donne la liste des entrées de la table

_consult -code <*mange-1*

donne la ligne de plus et de moins correspondant à cette entrée.

_consult -code <*mange*

fait la même chose pour chacune des entrées de ce verbe

_consult -test_finter aut<*mange-1*
mange-1 0 (ou 1)

test si l'automate de l'entrée a une intersection non vide avec l'automate.

_consult -finter aut <*manger-1* >aut2

Donne l'union des intersections de l'intersection par facteur de l'automate de l'entrée avec l'automate aut. Dans cet automate V0, Vpp, etc. ont été remplacés par leur valeur.

_consult -finter_add aut aut2 manger-1 >aut3

Ajoute à aut2 le résultat de *_consult -finter aut manger-1.*

_consult -union_verbe <liste.txt >aut

Calcul l'union des automates des entrées donnés dans le fichier liste.txt.

_DEB_ANA

Introduction

SPECIFIQUE A L'ANALYSE SYNTAXIQUE. Cette commande prend l'automate morphologique d'une phrase et pose les marqueurs qui permettent au parseur de l'analyser.

Présentation

_deb_ana <autph >autph2

rajoute [P] au début et à la fin des phrases.

_DECOMP

Introduction

Cette commande prend un automate à accès direct compacté (AUT_D13, AUT_T_D13) et le retransforme en un automate à listes (AUT_L13 ou AUT_E_L13). C'est la commande inverse de *_compac*.

Présentation

_decomp <AUT1 >AUT2

Calcul l'automate en liste correspondant à l'automate à accès direct compacté.

Exemple

_decomp <aut1 / _compac >aut2

Donne le même automate

_DEC_TXT

Introduction

Cette commande prend un fichier texte et le décompose en une succession de phrases

Présentation

_dec_txt <TXT_FILE

Affiche à l'écran la succession de phrases.

_dec_txt -nb n

Affiche la phrase numéro n.

_dec_txt -l n1 n2

Affiche les phrases n1 à n2.

_Exemple

$ _dec_txt <texte
[1] Il est parti.
[2] Ceci est la deuxième phrases.
[3] Et voici la dernière du fichier

Affiche à l'écran la succession de phrases.

dec_txt <texte / _ph_aut /_aff

Affiche les automates morphologiques des phrases du texte.

_DEFACT

Introduction

Cette commande défactorise un fichier texte.

Présentation

_defact SET <TXT_FILE

Défactorise le fichier TXT_FILE d'après le caractère SEP.

Exemple

```
$ cat txt
président,V,N
a,b,c:d:e
$ _defact "," <txt /_defact ":"
président,V
président,N
a,b
a,c:d
a,c:e
$
```

Considère que le premier élément précédant le séparateur est en facteur par rapport aux autres. Cette commande est l'inverse de la commande *_fact*.

_DESABR

Introduction

Cette commande permet d'utiliser certaines abréviation à l'intérieur d'un automate..

Présentation

_desabr -devmorpho [D/dicodir/delaf] <aut1 aut>2

Transforme les transitions suivant leur signification morphologique. par exemple *[v]* sera transformé en la série */unit/lex/v/??/??/??/??/??/??/??/??*. La convention est que les traits qui sont des lettres simples sont écrites dans un ordre quelconque jusqu'au premier point. Les traits qui sont des séquences continues de lettres sont séparés par des point :(.v-t. ou .Pro.). Ainsi, on peut utiliser les abréviations suivantes:

[v], *[n]* etc pour verbe, nom, etc.

[vp] pour verbe à une personne du pluriel

[v.v-t] pour un verbe conjugué

[.pre] *[.pro]* pour préposition ou pronom,

[<manger>vp] pour manger conjugué au pluriel,

le[.pro] pour le mot *le* considéré comme pronom

président[<présider>v],

mange pour le mot *mange* lui-même.

Cette commande utilise le dictionnaire des mots simples *delaf.eic delaf.cot*. Si ces fichiers se trouvent sur */dicodir*, le programme ira les chercher là si la variable *DELAF* est définie par */dicodir/delaf* et sinon il faut utiliser l'option *D*.

_desabr -auttr_simple auttr <aut1 >aut2

Si *auttr* définie une transduction de type 1-n (voir la commande _aut_trans ou l'annexe sur les transducteurs), cette commande effectue la transduction sur chacune des transitions.

_desabr -syn1 <aut1 >aut2

Les transitions de type *V0-<abasourdir-0>* rencontrée dans les automates syntaxiques sont transformées en *V0-/0/--/abasourdir/-V0*.

_desabr -supp_crochet <aut1 >aut2

Supprime les *[@* et *@]* pour l'analyse syntaxique.

Exemple

$ _aff -lst <aut1
[.det] [nms]
$ _desabr -devmorpho <aut1 >aut2
$ _aff -lst <aut2.

```
unit lex det ?? unit lex n ?? ?? ?? m s ??
$ cat texte
J'ai vu le garçon
$ _pgrep -aut aut2 <texte
J'ai vu [le garçon].
$
```

_DEFACT

Introduction

Cette commande défactorise un fichier texte.

Exemple

```
$ cat txt
président,V,N
a,b,c:d:e
$ _defact "," <txt /_defact ":"
président,V
président,N
a,b
a,c:d
a,c:e
$
```

Considére que le premier élément précédent le séparateur est en facteur par rapport aux autres. Cette commande est à mettre en relation avec la commande _fact.

_DELAF_TRANS

Introduction

Cette commande traduit une transduction définie par une suite de relations binaires et le transforme en transducteur

Présentation

_delaf_trans <in >trans

Transforme la suite des relations binaires décrites sur le fichier in en un transducteur de type TRANS_L44.

_delaf_trans -del1

Transforme les données de type delaf en données de transduction:

.N21:ms -> *,N+ms* en plus de la transformation précédente.

Exemple

```
$ cat txt
mangions,manger.V3:P1p
abc,def
abc,ddd
abc,eee
$ _delaf_trans <txt >trans
$ cat test
mangions
qsdf
abc
$_transduc trans -1 <test
manger.V3:P1p
def
ddd
eee
$ cat test2
manger.V3:P1p
eee
$ _transduc trans -2 <test2
mangions
abc
$
```

_DET

Introduction

Cette commande déterminise les automates. Elle accepte les automates de type AUT_L44 ou AUT_L13.

Présentation

_det <aut1 >aut2

Déterminise l'automate aut1.

_det -supp_epsi -epsi "<E>" <aut1 >aut2

Supprime les epsilon transitions sans déterminiser complètement. Par défaut, pour un automate de type AUT_L44, epsilon="<E>", pour un automate de type AUT_L13, epsilon='0'.

_det -trace <aut1 >aut2

Permet de suivre l'évolution de la fonction

_det -newmax NUM<aut1 >aut2

Le nombre maximum d'état de l'automate aut2 est fixé par défaut à 10.000, il est possible de modifier ce paramètre avec l'option newmax.

Exemple

_det -trace -newmax 150000 <aut1 >aut2

_DIFF

Introduction

Cette commande fait la différence facteur de deux automates de type AUT_L44

Présentation

_diff aut1 aut2

Si L1=L(aut1), L2=L(aut2) et L3=l(aut3) alors L3=L1 -L2

_diff -f aut2 <aut1 >aut3

Si L1=L(aut1), L2=L(aut2) et L3=l(aut3) alors L3=L1 - A*.L2.A*

Exemple

$ _aff -lst <aut1
le[.pro] [n]
ne [n]
$ _aff -lst <aut2
[n]
$ _desabr -devmorpho <aut2 >pattern
$ _desabr -devmorpho <aut1 >gram.aut
$ _dec_txt <texte / _ph_aut /_diff gram.aut /_pgrep pattern

Cette commande recherche dans le texte tous les noms en tenant compte de la grammaire locale gram.aut. Cette grammaire spécifie que les particules préverbales le et n ne peuvent être suivit d'un nom.

_DIRECT

Introduction

Cette commande permet de faire des manipulations directes sur des automates ou des transducteurs comme ajouter un état, une transition ou supprimer un état ou une transition ou changer le type (sorte) d'un état.

Présentation

_direct -sorte NUM1 NUM2 <AUT1 >AUT2

Met le type de l'état NUM1 à NUM2 de l'automate AUT1 et met le résultat dans AUT2.

_direct -add_trans NUM1 STRING NUM2 <AUT1 >AUT2

Ajoute la transition STRING entre NUM1 et NUM2

_direct -add_state <AUT1 >AUT2

Rajoute un état

_direct -supp_trans NUM1 STRING NUM2 <AUT1 >AUT2

Supprime la transition STRING entre NUM1 et NUM2

_direct -supp_state NUM1 <AUT1 >AUT2

Supprime l'état NUM1 avec toutes ses transitions

_direct -netpoubelle <AUT1 >AUT2

Supprime toutes les transition *poubelle* et *trash*. Il faut a priori composer cette commande avec la réduction **_reduc**.

_direct -ordonne_transi <AUT1 >AUT2

Range les transitions de chaque état dans l'ordre croissant.

_direct -is_oder <AUT1 >AUT2

Teste si toutes les transitions sont bien ordonnées

_DS_ALPH

Introduction

Cette commande teste l'appartenance d'un mot à un automate.

Présentation

_ds_alph -aut aut1 <*unmot*

Teste l'appartenace de unmot à aut1.

Exemple

```
$ cat texte
il
et
elle
$ _lst_aut <texte >aut
$ cat test
il
elles
$ _ds_alph -aut aut <test
il 5
elles  10000000
$
```

_EXT_ALPH

Introduction

Pour un automate cette commande extrait l'alphabet.

Présentation

_ext_alph <aut >alph

```
Extrait l'alphabet alph de l'automate.
```

Exemple

```
$ _aff -lst <aut
il est parti
$ _ext_alph <aut >alph
$ cat txt
il
elle
est
$ _ds_alph -aut alph <txt
il,2
est,1
$
```

_EXT_AUT

Introduction

Extrait des automates de structures plus complexes.

Présentation

_ext_aut <resinter >aut

Extrait l'automtate d'un structure RES_INTER qui représente le résultat d'une intersection.

_FACT

Introduction

Factorise un fichier non factorisé d'après la virgule.

Présentation

_fact <in >out

Factorise le fichier in. Les entrées identiques sont supposées contigues. Il faut donc trier préalablement le fichier.

Exemple

```
$ cat in
il,gaston
il,ernest
$ _fact <in
il,gaston,ernest
$
```

_FDELSYN

Introduction

SPECIFIQUE A LA CONSTRUCTION DE DICTIONNAIRES SYNTAXIQUES. Ce programme permet de passer d'un ensemble de tables et d'automates de référence (voir annexe Traitement des tables).

Présentation

_fdelsyn -tabdic 35s

Transforme les fichiers 35s.car et 35s.trs en un fichier 35s.dic.

_fdelsyn -motdeb <text >out

Extrait les mots précédents le point dans la transition.

_fdelsyn -aut_vref alph <aut >vref

Transforme l'automate de départ en une structure d'automate de référence en tenant compte de l'alphabet alph.

_fdelsyn -splt_entre_code <in >out

Prend le fichier des tables avec entrée unique et met les codes formatés dans *cod.lst*.

_fdelsyn -assdic

Prend les fichiers codes.lst, dic.cut et *.vref pour en faire un seul fichier: delsyn.

_FDIC

Introduction

Cette commande prend en entrée un dictionnaire écrit sur un fichier texte et construit l'automate correspondant

Présentation

La construction du dictionnaire se fait en deux étapes. La première consiste à transformer le fichier texte en un fichier ou chaque ligne contient un mot suivit d'une virgule et d'un numéro de code. La seconde consiste à prendre cette deuxième représentation et à la transformer en un automate.
Si le fichier entrée est exactement du type suivant:
manger,mangions.qmsdkfjmq
c'est à dire du type entrée-virgule-forme canonique extensive-point-texte la commande

_fdic -1 dicotexte dicoarr

construit directement l'automate final, elle fait donc les deux étapes automatiquement. On ne suppose pas que le fichier est trié. Les entrées ne sont pas factorisées. Le résultat est mis dans *dicoarr.aic* (AUT_D13), *dicoarr.lic* (AUT_L13) et dicoarr.cod (fichier texte).

_fdic -2 texte

Si la forme canonique est déjà représentée de manière différentielle, la commande texte effectue la même opération.

_fdic -nettdic texte dicoarr

prend le texte du type _fdic -1 et construit la représentation où chaque mot est suivit de son code.

_fdic -constdic dicotexte dicoarr

appliqué à ce résultat construit l'automate final. ATTENTION, les codes doivent être >0. La donnée initiales est dicotexte et dicotexte.cod.

Pour construire le dictionnaire _fdic construit un fichier de commande qui appel _fdic lui-même (avec d'autres options) et les commandes _lst_aut, _union et _mnm.

_fdic -constlex dic1 dic1.aut

Construit l'automate d'un lexique décrit dans dic1.

_fdic -facteur FACT <AUT

Donne la fréquence d'apparition du facteur FACT (une chaine de caractères), c'est à dire le nombre d'état où apparaît le facteur.

_fdic -facteurs NUM <AUT

Donne la liste des facteurs de longueur NUM. Si on utilise cette commande en combinaison avec *_index -count* on obtient la liste des facteurs avec la fréquence de chacun.

Exemples

$ cat dic

```
mange
mangions
il
elles
$ _fdic -constlex dic dic.aut
$ _cal_crd <dic.aut /_aff -lst
elles
il
mange
mangions
$
```

```
$ _fdic -facteurs 2 <AUT /_index -count /_aff -lst
```

Affiche la liste des facteurs de longueur 2 avec la fréquence de chacun.

_FNC

Introduction

Cette commande prend en un dictionnaire de mots composés et construit en sortie l'automate correspondant.

Présentation

Le principe de la construction des dictionnaires de mots compoés est le même que celui suivit pour les dictionnaires de mots simples (voire commande _fdic). La différence principale réside dans le fait qu'il faut préalablement construire le dictionnaire des mots simples qui composes les mots composés.

_fnc -constlex lex aut

Transforme le lexique de mots composés en un automate.

_fnc -1 dico aut

Prend le dictionnaire de mots composés au format

a/majuscules,a/majuscule.NA:mp/-+;un

sur le fichier dico et construit l'automate correspondant ainsi que le fichier des codes sur *aut.lic* et *aut.cod*. Ces deux fichiers sont ensuite utilisables par la commande *_morpho -nc* (consultation du dictionnaire compacté).

_fnc -diff <dic1 >dic2

Ecrit la forme canonique de manière différentielle: *a/majuscules,a/majuscule* devient *a/majuscules,/1*.

_fnc -netdic dico1 dico2

Met le dictionnaire sous la forme entree+code. Met le résultat dans dico2 et dico2.cod.

_fnc -constdic dico2 dico3

Prend le résultat précédent (*dico2* et *dico2.cod*) et construit l'automate finit en liste non compacté (dans *dico3.lic* et *dico3.cod*).

Exemple

```
$ cat dic
pomme/de/terre
pommes/de/terre
$ _fnc -constlex dic aut
$ _aff -lst <aut
pomme de terre
pommes de terre
$
```

_GRAPH_AUTL44

Introduction

Cette commande transforme un automate déssiné sur le programme Editor (Max Silberztein) en un automate de type AUT_L44

Présentation

_graph_autl44 <aut.graph >aut

Transforme la forme graphique issue de l'éditeur en un automate de type AUT_L44.

Exemple

_graph_autl44 <aut.graph /_det/_mnm>aut

_INDEX

Introduction

Construit des index à partir de listes de mots d'un texte et de leur position dans ce texte. Ces listes de mots sont en général le résultat de l'utilisation de commandes comme *_agrep*, *_pgrep* ou *_plgrep* et ces index sont en général à utiliser avec la commande *_access*.

Présentation

_index -count <TXT_FILE >AUT

Construit l'arbre (de type AUT_L13) qui liste les mots qui apparaisent dans TXT_FILE et qui met à chaque noeud le nombre d'appartions du mot correspondant dans la liste TXT_FILE.

[_index <TXT_FILE >IND_FILE]

Si TXT_FILE contient une liste de mots avec leur position :

il,356

elle,23

il,1234

le commande construit l'index (i.e. l'arbre) équivalent. Ceci diffère de la commande _lst_aut en ce que le nombre en regard de chaque mot n'a plus à être unique: dans notre exemple deux nombres sont associés à il.

Exemple

$ _agrep delacf.lic -lst <TXT_FILE /_index >IND_FILE

Construit l'index des mots composés du texte de TXT_FILE.

_INIT

Introduction

Cette commande initialise certains type de structure, c'est à dire elle crée des ensembles vides de certaines structures.

Présentation

_init -alptd13 >alph

```
alph est alors un alphabet de type ALP_T_D13 avec aucun mot.
```

_INTER

Introduction

Cette commande effectue des intersection entre deux automates. ATTENTION, le résultat de l'intersection est un automate NON REDUIT. Il faut donc éventuellement utiliser *_reduc.*

Présentation

_inter aut1 aut2 >aut3

Fait l'intersection des deux automates aut1 et aut2 si ils sont de type AUT_L44, TRANS_L44 ou AUT_L13. Dans le cas d'un automate de type AUT_L13 la commande tient compte de la sémantique de @ qui est l'abréviation de "à un état donné, toutes les lettres qui ne sont pas représentées à cet état".

_inter aut1 aut2 -trace -newmax NUM >aut3

L'option -trace permet de suivre l'opération et l'option newmax permet de changer la borne pour le nombre d'états de aut3 fixée par défaut à 50.000.

_inter -f aut1 aut2 >res

Fait l'intersection par facteurs de *aut1* avec *aut2*, c'est à dire on recherche les facteurs de *aut1* qui sont dans *aut2*. Le résultat est une suite de RES_INTER.

_inter -f aut2 <aut1 >res

Fait la même chose que la commande précédente.

_inter -syn2 aut1 aut2 >res

Fait l'intersection syntaxique de l'automate aut1 qui représente la phrase avec l'automate aut2 qui représente la grammaire.

_LST_AUT

Introduction

Cette fonction prend en entrée une liste de mots et donne en sortie un automate minimal. Attention, cette commande n'est pas prévue pour des lexiques ou des dictionnaires de beaucoup plus de 10.000 lignes. Pour des fichiers plus important il est préférable d'utiliser les commandes *_fdic* et *_fnc*.

Présentation

Cette commande prend en entrée un fichier texte qui contient une liste de mots séparés par des retours à la ligne et renvoie un automate déterministe et minimal qui représente ce lexique. Si les lignes du fichier d'entrée contiennent chacune une virgule, l'emploi de l'option -code permet de considérer ce qui suit comme le numéro de code du mot. Il est impératif que si un mot apparaît deux fois, ce soit avec le même code. On peut construire un automate qui représente un automate de mots composés de la même façon.

_lst_aut <lexique >aut

Transforme un lexique de mots simple en un automate.

_lst_aut -code <dico >aut

Transforme un dictionnaire de mots simple en un automate multiterminal. Le séparateur entre l'entrée et les informations étant la virgule.

_lst_aut -nc aut1 <lexique >aut2

Etant donné l'automate *aut1* de type AUT_T_D13 qui représente les mots simples constituant le lexique de mots composés *lexique*, cette commande construit l'automate acyclique minimal de type AUT_L44 qui le représente.

_lst_aut -nc aut1 -code <dico >aut

La même chose mais avec un code pour chaque entrée.

_lst_aut -nc aut1 (-code) -2 <dico >aut

Fait la même chose mais utilise une fonction moins efficace mais qui ne limite pas le nombre de transitions par états (borne fixée par défaut à 10000). Il faut donc en principe utiliser cette option pour compacté des dictionnaires de mots composés de taille importante.

_MNM

Introduction

Cette commande prend en entrée un automate acyclique et renvoie l'automate minimal correspondant

Présentation

_mnm <aut1 >aut2

Minimise l'automate déterministe acyclique aut1 en l'automate aut2. L'automate aut1 peut être de type AUT_L13, AUT_L44.

_mnm -1 <aut1 >aut2

Dans le cas où aut1 est de type AUT_L13 cette option rend l'opération plus rapide.

_MORPHO

Introduction

Cette commande prend en entrée une liste de mots et donne en sortie un automate

Présentation

_morpho [D/dicodirectory/dicoprefixe]

Etant donné le dictionnaire dont on précise le path, cette commande consulte le dictionnaire. Si la variable d'environnement système DELAF vaut /dicodir/dicopref, le programme ira automatiquement rechercher les fichiers dicopref.eic dicopref.cod dans le directory précisé.

Exemples

```
$_morpho
il
il,Pro
mangions
mangions,manger.V3:P1p
$
```

NEXTPS_ALPTD13

Introduction

Transforme l'alphabet de type ALPTD13 pour qu'il corresponde au code ASCII IBM.

Présentation

nextps_alptd13 <alph1 >alph2

NEXTPS_AUTL13

Introduction

Change le code ASCII d'un automate de type AUTL13

Présentation

nextps_autl13 <aut1 >aut2

NEXTPS_DELCOT

Introduction

Change le code ASCII d'une structure _DELCOT.

Présentation

nextps_delcot <delcot1 >delcot2

_PARSE

Introduction

Fait l'analyse syntaxique d'une phrase dont on a l'automate de représentation morphologique.

Présentation

_parse gram.aut <autph1 >autph2

Si autph1 est l'automate morphologique de la phrase, autph2 est le résultat de l'analyse syntaxque d'après la grammaire gram.aut. gram.aut, autph1 et autph2 sont de type AUT_L44. Le résultat peut être visualisé par la commande _proj.

_parse gram.aut -prof 2 <autph1 >autph2

Cette option limite la profondeur de l'analyse à 2

Exemple

```
$ cat texte
que Luc ait dit cela étonne Jean
$ _ph_aut <texte >autph
$ _parse gram.aut <autph >autph2
$ _proj <autph2
(que ((Luc) ait dit (cela))) étonne (Jean)
```

PAUTO

Introduction

Ce programme est un logiciel d'édition graphique et de visualisation d'automates et de transducteur. Ce programme est actuellement implémenté sur système NextStep.

Présentation de l'édition

L'utilisation de la commande new crée un nouvel automate avec un seul état (initial) et sans transition.

sélectionner un état:

cliquer dessus

créer un état

command clic à l'emplacement voulu. Construit en même temps des transitions entre chacun des états préalablement sélectionnés et ce nouvel état.

sélectionner une transition

cliquer dessus. On peut alors modifier l'étiquette dans la fenêtre de contrôl de la fenêtre principale où la déplacer (courbe de bézier en déplacant un des deux points de contrôl

effacer une transition

Control-cliquer sur la transition

effacer un état

Control-cliquer sur l'état, cela efface du même coup toutes ses transitions (entrantes ou sortantes).

Exemple d'édition

Après appel du programme l'appel de New crée un automate avec uniquement l'état initial (figure 1 et 2). Command clic ailleurs crée un nouvel état et une transition entre les états préalablement sélectionnés (état 0) et ce nouvel état. L'étiquette est provisoirement tmp (figure 3). Pour changer cette transition on clique dessus et on obtient la configuration de la figure 4.En écrivant une nouvelle étiquette *un exemple d'étiquette* l'étiquette de la transition est modifiée et donne le résultat de la figure 5. Pour modifier la tranjectoire d'une transition qui est une courbe de bézier on la sélectionne et on déplace un des points de contrôle par control-drag (figure 6). Cela donne par exemple une transition comme sur la figure 7. Jusqu'à ce point chaque lettre de la transition est tangente à la courbes mais il peut être plus lisible d'écrire toute l'étiquette horizontalement comme dans la figure 8.

Présentation de la visualisation

La commande *View AUT_L* du menu *File* permet de charger un automate (calculé par une des commandes de *PUPITRE*) et affiche l'état initial ainsi que chacune de ses transitions (avec leur état d'arrivée). *Shift-clic* sur un état le développe, c'est à dire affiche ses transitions et les états vers lesquels elles pointent.

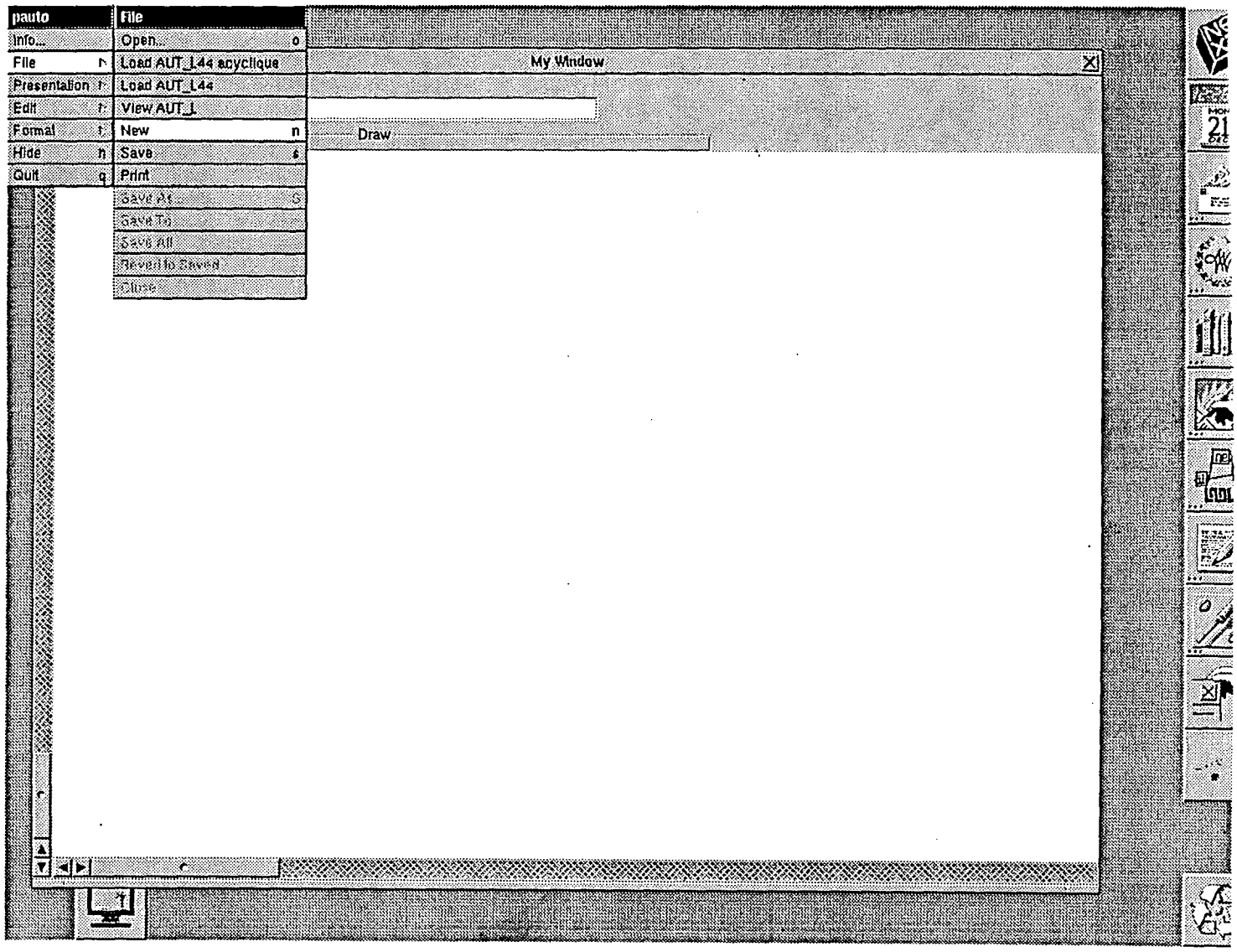

pauto
Info...
File
Presentation
Edit
Formal
Hide
Quit
File
Open...
Load AUT_L44 acyclique
Load AUT_L44
View AUT_L
New
Save
Print
Save As
Save To
Save All
Revert to Saved
Close
My Window
Draw

figure 1

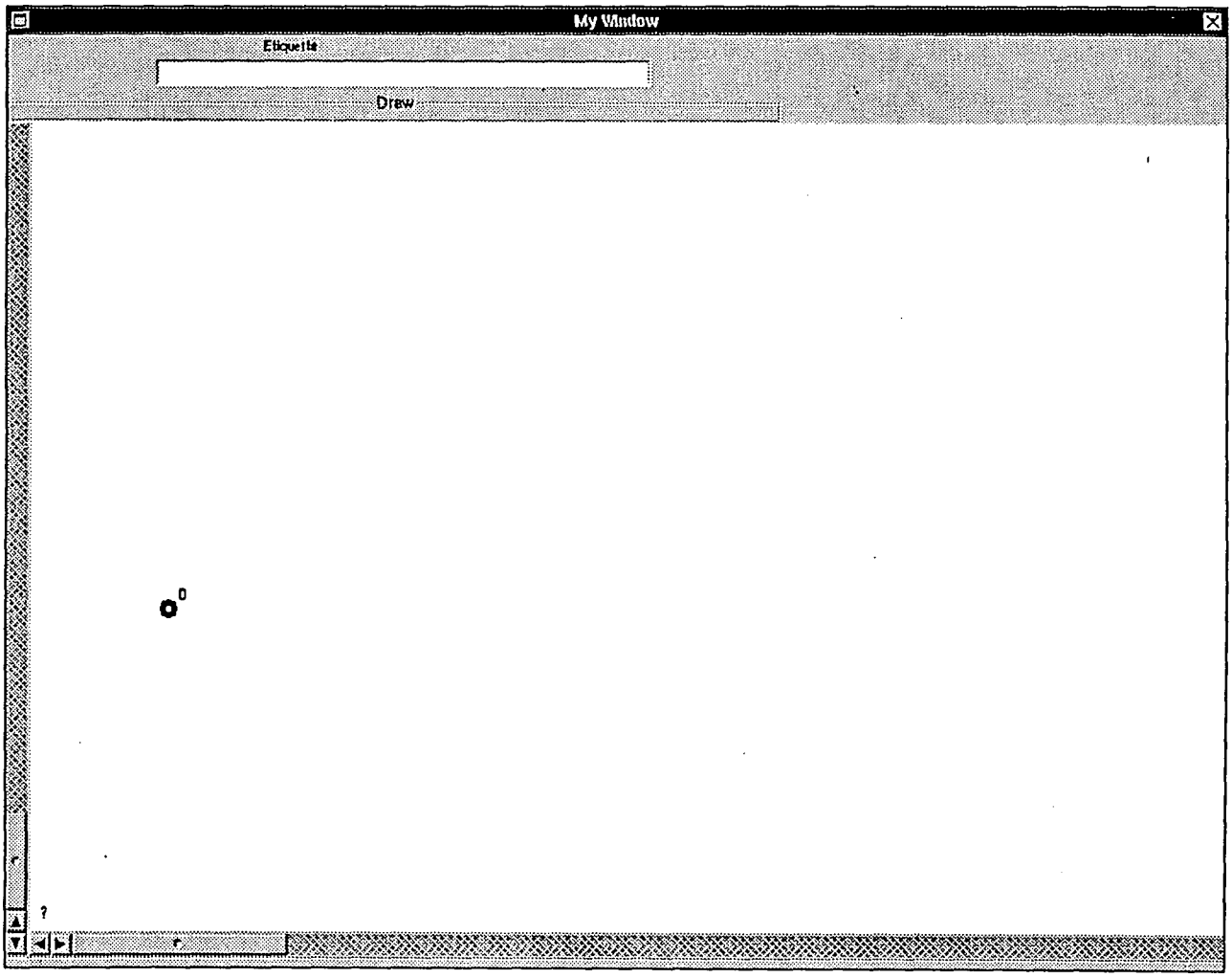


figure 2

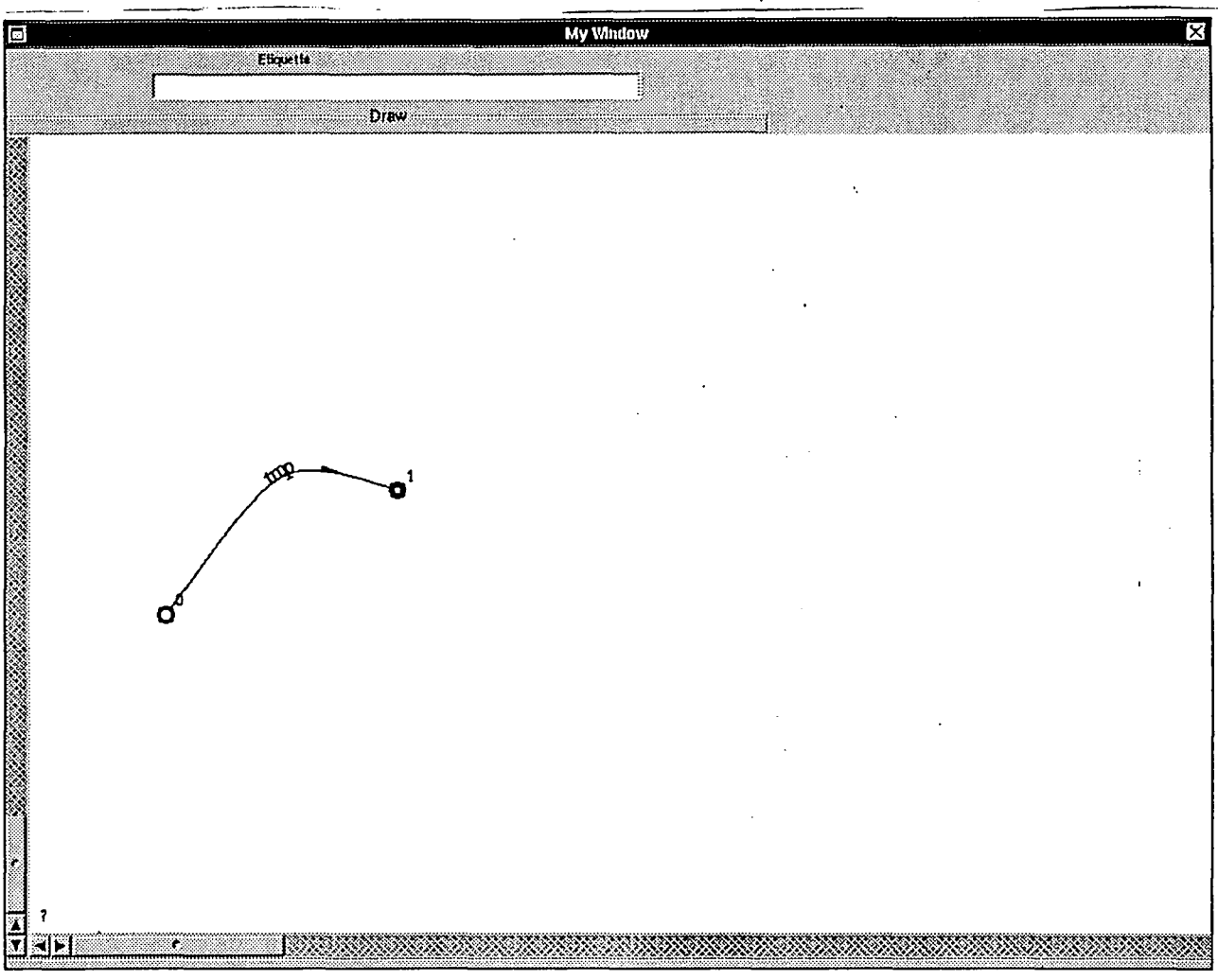


figure 3

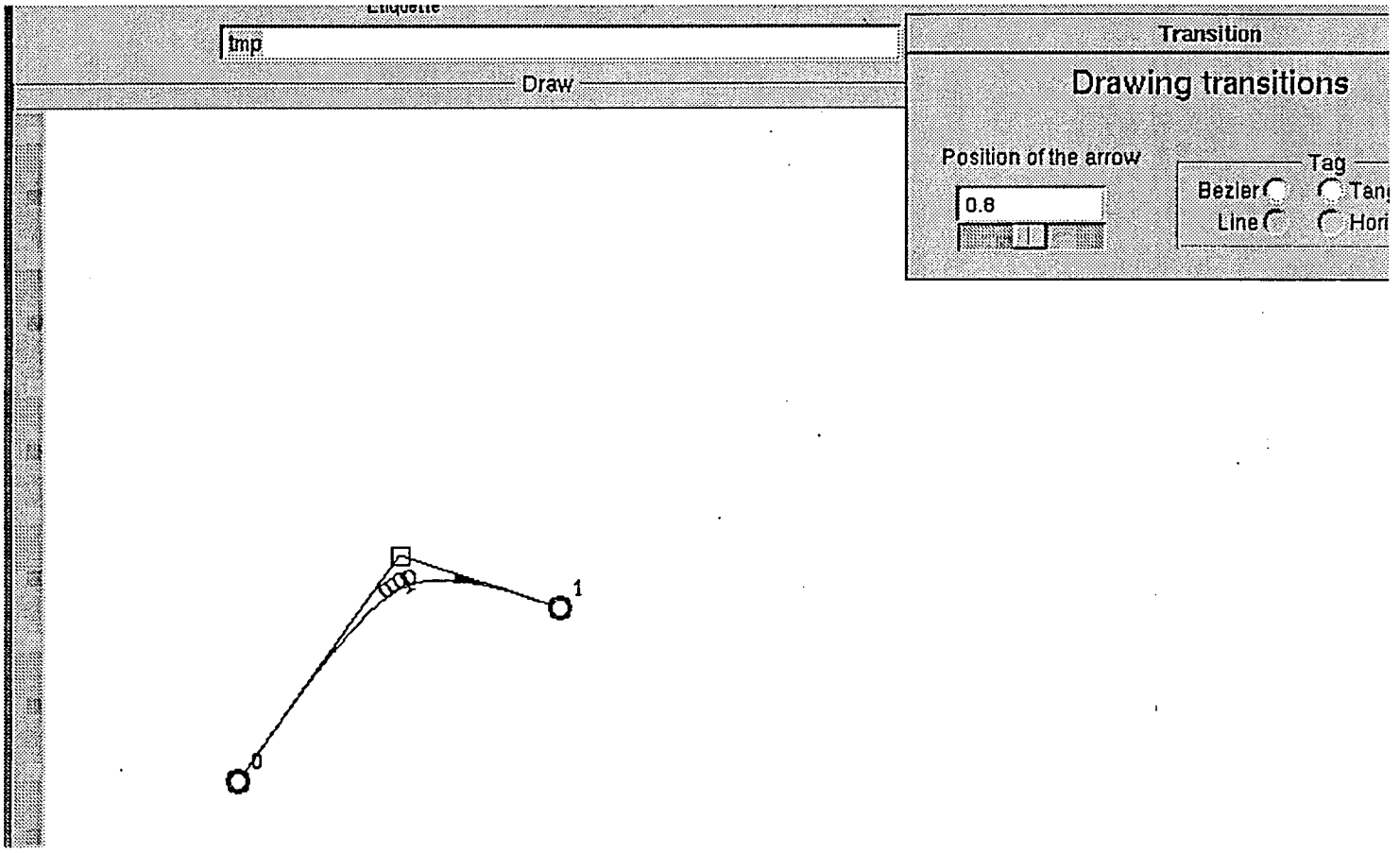

tmp
Draw
Transition
Drawing transitions
Position of the arrow
0.6
Tag
Bezier
Line
0
1

figure 4

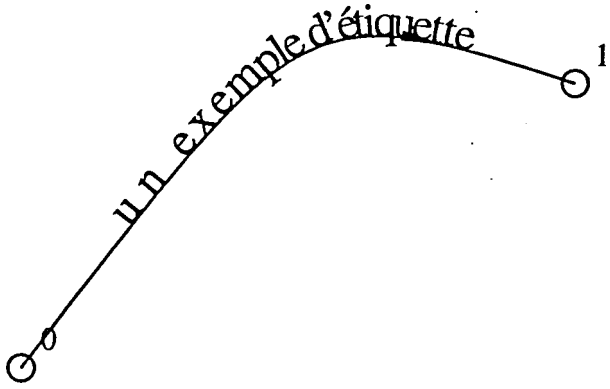


figure 5

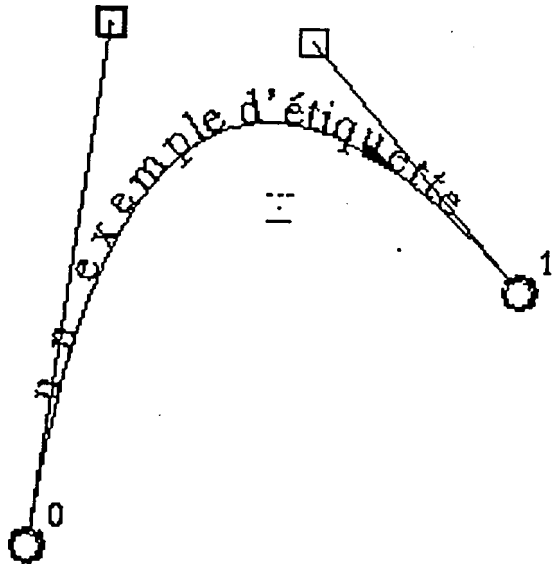


figure 6

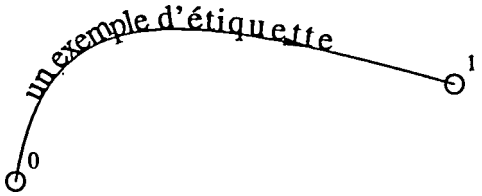
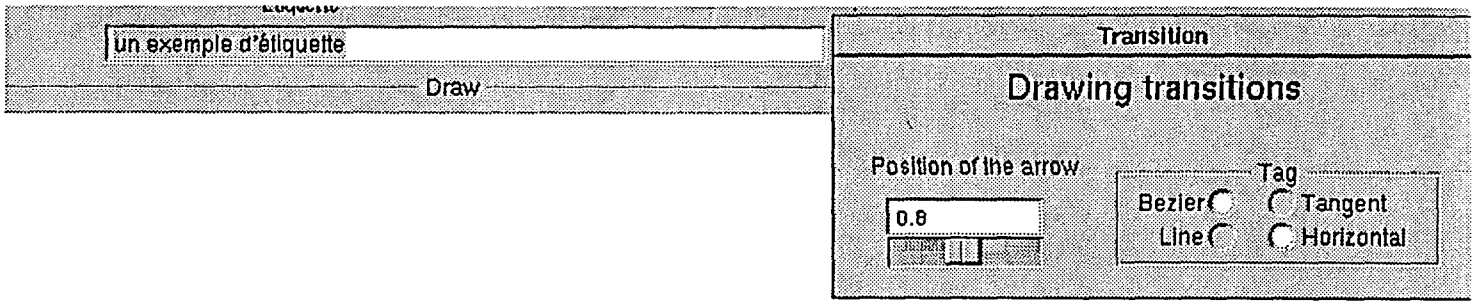


figure 7


un exemple d'étiquette
Draw
Transition
Drawing transitions
Position of the arrow
0.8
Tag
Bezier
Tangent
Line
Horizontal

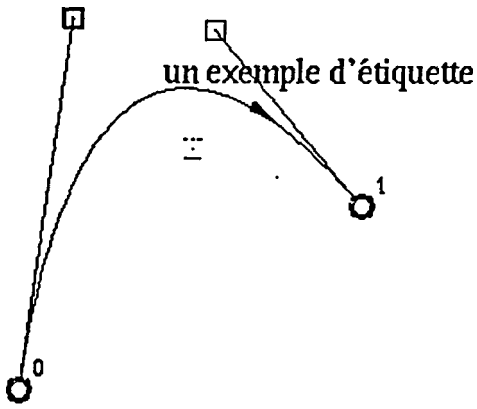

un exemple d'étiquette
0
1

figure 8

_PCK
(David Clemenceau)

Introduction

Transforme un fichier de règles morphologiques 2-level du programme PCKimmo en des structures manipulables par PUPITRE dans le but de construire des transducteurs 2-Level. Voir Annexe 3 pour le détail de la construction de transducteurs de dictionnaires

Présentation

_pck -extalph <fich >LST_FILE

Extrait la liste de toutes les équivalences lettre-lettre.

_pck -alph alph <fich

A partir du même fichier et d'après l'alphabet de type ALP_T_D13 qui contient toutes les équivalences précédentes on construit la liste des automates représentant chacune des transductions du fichier fich.

_PGREP

Introduction

Cette commande prend en entrée un texte et une structure morphologique et donne toutes les occurences de cette structure. La structure peut être représentée par un automate.

Présentation

_pgrep -aut aut [D/dicodir/dicopref] <texte

Marque dans le texte toutes les occurence morphologiques de texte.

Exemple

```
$ _aff -lst <aut1
[.det] [nms]
$ _desabr -devmorpho <aut1 >aut2
$ _aff -lst <aut2
unit lex det ?? unit lex n ?? ?? ?? m s ??
$ cat texte
J'ai vu le garçon
$ _pgrep -aut aut2 <texte
J'ai vu [le garçon].
$
```

_PLGREP

Introduction

Cette commande prend en entrée un texte et une structure morphologique et donne toutes les occurences de cette structure en ayant levée des ambiguïtés grâce à une grammaire locale.

Présentation

_plgrep -aut aut [D/dicodir/dicopref] <texte

Marque dans le texte toutes les occurence morphologiques de texte.

_PROJ

Introduction

Cette commande prend un automate et ne garde que certaines informations (que l'on spécifie par un transducteur) et permet leur affichage. Elle permet par exemple de prendre le résultat d'une analyse morphologique et d'en extraire les notations parenthèsées. La projection est en fait la composition des trois opérations suivantes:

-> application d'une transuction à l'automate

-> déterminisation du résultat

-> Parcours de l'automate et affichage des chemins

Présentation

_proj -trans TRANSDUC <AUT

Affiche la liste des chemins de AUT1 en ne conservant que les informations spécifiées dans le transducteur transduc. On peut omettre TRANSDUC.si la variable globale d'environement TRANS_PROJ est définie et vaut TRANSDUC.

_proj -simple1 <AUT

Première projection par défaut, donne les résultats avec parenthèse

_proj -simple2 <AUT

Deuxième projection par défaut, on conserve les labels des parenthèses fermantes.

PSPLT

Introduction

Découpe un fichier, qui peut être binaire, en plusieurs morceaux plus réduits.

Présentation

pslpt fichin fichout

```
Découpe fichin en fichouta fichout b ..
```

_RAT_AUT

Introduction

Transforme une expression rationnelle en automate.

Présentation

_REDUC

Introduction

Réduit un automate pour les structure AUT_L13, AUT_L44 et AUT_LT44

Présentation

_reduc <aut1 >aut2

Fait la réduction de aut1 et la met dans aut2. La réduction supprime les états non accessibles (pas de chemin d'un état initial à eux) et les états non coaccessibles (pas de chemin partant d'eux et arrivant à un état terminal).

_TRAD

Introduction

Effectue des changements simples de format entre les différents type d'automates manipulés par PUPITRE.

Présentation

_trad -autlt44_autl44 <aut1 >aut2

Passe de AUT_LT44 à AUT_L44.

_trad -autl44_autlt44

Passe de AUT_L44 à AUT_LT44

_trad -auttd13_autd13

Passe de AUT_T_D13 à AUT_D13

_trad -transl44_autl44_1

Met la partie gauche du transducteur de type TRANS_L44 dans un automate de type AUT_L44

_trad -transl44_autl44_2

Met la partie droite du transducteur de type TRANS_L44 dans un automate de type AUT_L44

_trad -autl13_autl44

Passe de AUT_L13 à AUT_L44 en consérant une lettre comme 'a' comme le mot "a".

_trad -transl44_autl44

Passe de TRANS_L44 à AUT_L44. Les transition de type "aze":"qsd" sont transformées en "aze:qsd".

_TRANSDUC

Introduction

Applique un transducteur à un mot ou à un automate.

Présentation

_transduc trans -1 <*mot*

Applique la transduction décrite par trans de type TRANS_L44 au mot *mot* dans le sens 1 (gauche droite).

_transduc trans -2 <*mot*

Fait la même chose dans le sens 2 (droite-gauche).

_transduc trans -1 -trace <*mot*

Laisse la trace des transformations.

Exemples

```
$ cat texte
mangions,manger.V3:P1p
$ _delaf_trans <texte >tmp1
$ _fnc -constlex tmp1 aut1
$ _ext_alph <aut1 /_aut_trans>alph
$ _aut_trans -alph alph <aut1 >trans
$ _transduc trans -1
mangions
manger,V3:P1p
$
```

_UNION

Introduction

Cette commande fait l'union de deux automates

Présentation

_union aut1 aut2 >aut3

Fait l'union de aut1 et aut2 et met le résultat dans aut3. Les automates sont supposés déterministes.

ANNEXE 1 : NOTIONS DE BASE SUR LES AUTOMATES A ETATS FINIS ET LEUR REPRESENTATION

Fondamentalement, on utilise trois types de représentation des automates. Dans cette annexe nous présentons successivement ces trois types: automates en listes, automates en listes dans les tableaux, automate à accès direct compacté. Nous donnerons ensuite la manière dont ces types sont implémentés.

1. AUTOMATES EN LISTE

Prenons comme exemple l'automate suivant :

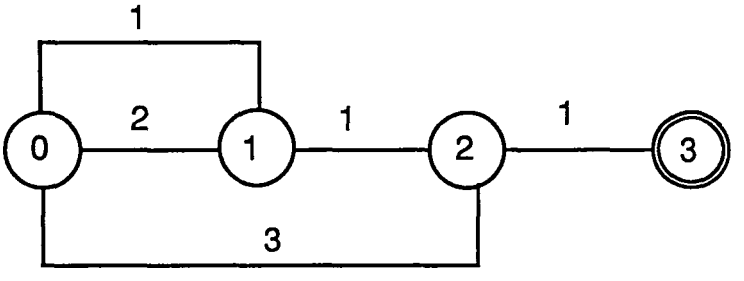


figure A1.1

Cet automate est constitué de quatre états numérotés de 0 à 3 et d'un alphabet de quatre lettres {0,1,2,3} dont la première lettre n'apparaît dans aucune transition.

La manière la plus direct de coder cet automate est de considérer les états comme un tableau pointant vers des listes chaînées. Ainsi l'automate précédent peut-il se coder dans la représentation de la figure A1.2. Le type d'un état (*terminal* vs *non terminal* correspond à 0 vs 5 (arbitrairement)).

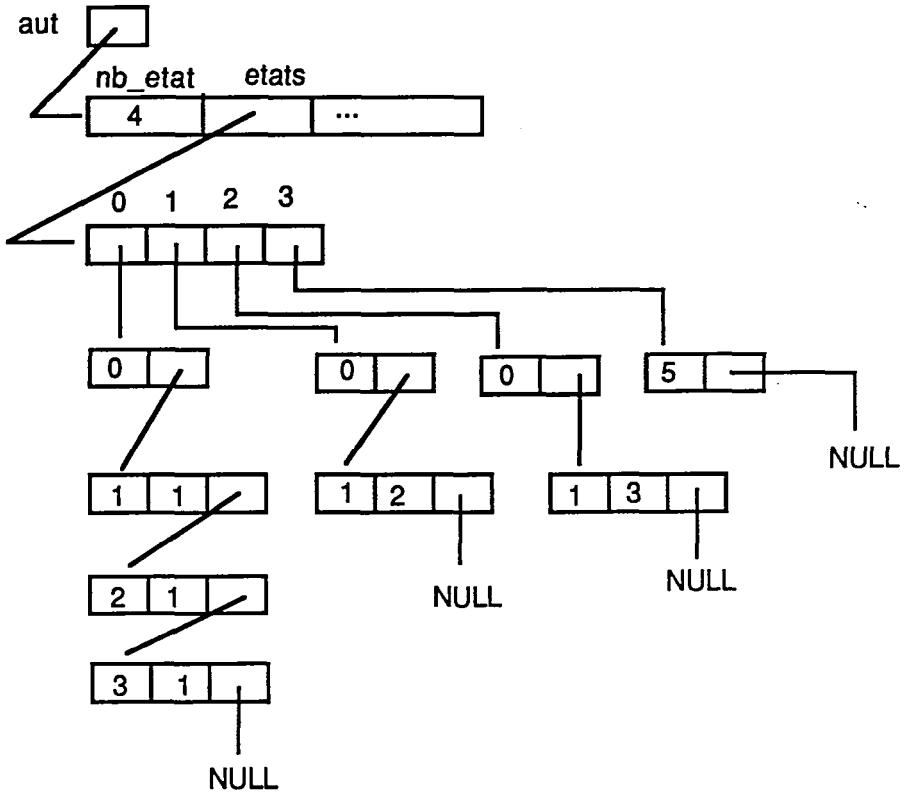


figure A1.2

Si la structure est définie en langage C, on accède aux valeurs de la manière suivante : le type de l'état 2 a pour valeur *aut->etats[2]->sorte=0*, sa première transition a pour étiquette *aut->etats[2]->liste->cel_g* et pour état d'arrivée *aut->etats[2]->liste->cel_d* alors que *aut->etats[2]->liste->suivant=NULL*. Ce type de représentation est utilisé dans les structures notées AUT_L* (AUT_L13, AUT_L44, TRANS_L44, ..).

Cette représentation a pour avantage d'être souple, on peut rajouter et supprimer des transitions à un état dans n'importe quel ordre. En revanche, elle est la plus couteuse en place mémoire car chaque transition contient l'étiquette, l'état d'arrivée et surtout un pointeur vers la transition suivante.

2. AUTOMATES EN LISTE-TABLEAU

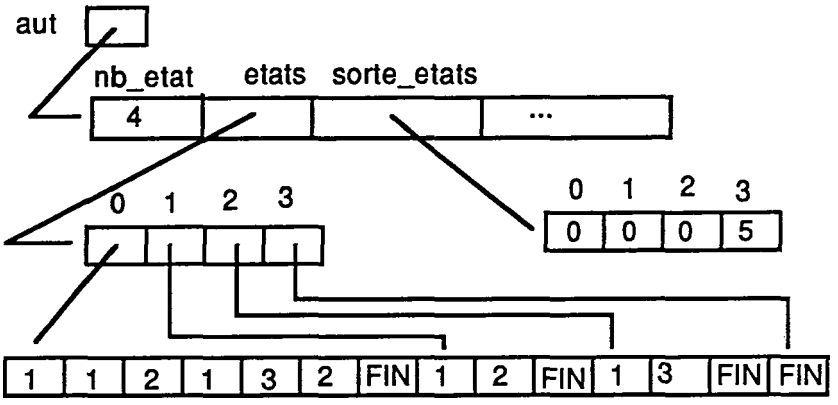


figure A1.3

Le même automate étant donné, il est possible de représenter la liste des transitions dans un tableau. Pour chaque transition, le stockage d'un pointeur vers la transition suivante devient alors inutile. Cependant cette structure est plus contraignante puisqu'il est impossible (ou couteux) d'ajouter des transitions à un état qui n'est pas le dernier.

Ce type de structure est celui utilisé dans les structure de type AUT_LT* (AUT_LT13, AUT_LT44).

3. AUTOMATES A ACCES DIRECT COMPACTE

Rappelons tout d'abord qu'un accès direct classique se présente sous la forme suivant:

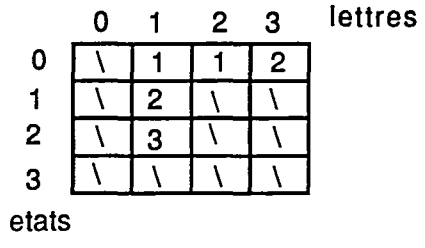


figure A1.4

Il faut remarquer que pour un grand nombre d'automates cette matrice est presque entièrement vide, c'est à dire que chaque état a un nombre de transition beaucoup plus petit que le cardinal de l'alphabet. On va tenir compte de cette propriété pour proposer une structure plus économique mais qui conserve la propriété de l'accès directe (contrairement aux automates en liste-tableau). Pour cela il faut d'abord considérer la représentation intermédiaire suivante :

0	1	2	3	4	5	6	7	SORTE_POS	
\	\	1	1	2	1	3	2	SORTE_CODE	0
\	\	1	2	\	\	\	\	SORTE_CODE	0
\	\	1	3	\	\	\	\	SORTE_CODE	0
\	\	\	\	\	\	\	\	SORTE_CODE	5

figure A1.5

Pour chaque lettre, si il existe une transition (par exemple état 0 lettre 1), on note en (lettre*2) le numéro de la lettre et en (lettre*2)+1 l'état d'arrivée. A ce point l'information est redondante puisque la position permet de déduire le numéro de la lettre. Cependant cette matrice peut être écrite dans une seule ligne en bouchant les trous sans perdre la relation entre les états et leur transition comme dans la figure suivante :

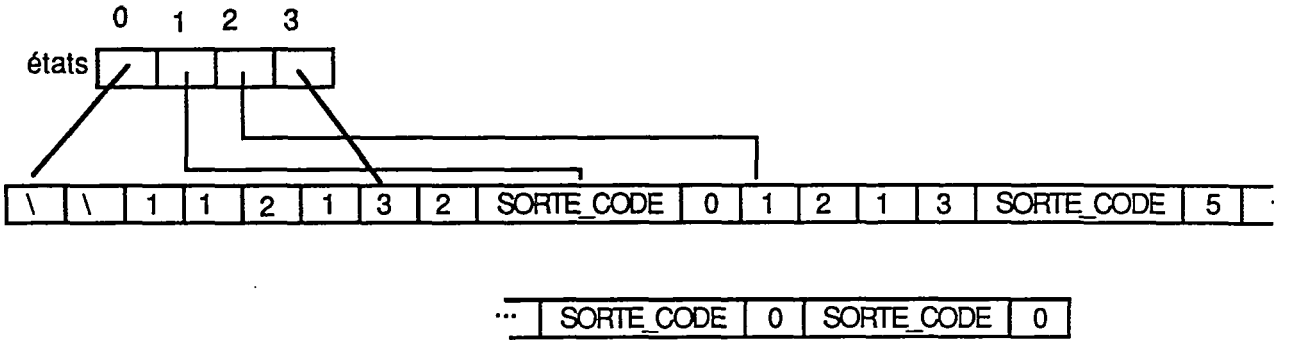


figure A1.6

En fait on peut remarquer que, l'état initial étant unique, il est inutile de garder un tableau d'états pointant dans le tableau des transitions. Les transitions, plutôt que d'indiquer le numéro de l'état d'arrivée, peuvent donner directement la position dans la ligne où les transitions de cet état commence. Cela correspond de fait à une renumérotation des états. On obtient donc enfin la représentation suivante :

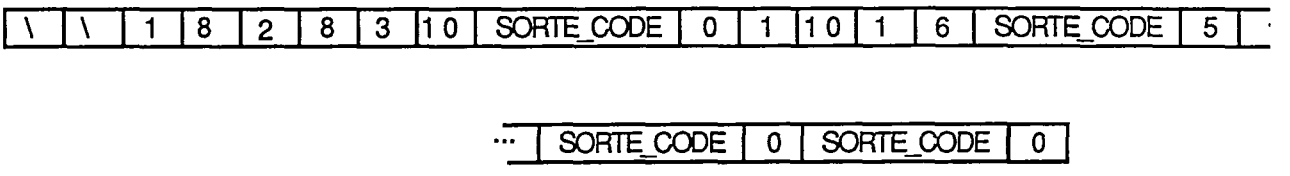


figure A1.7

L'algorithme qui permet de vérifier qu'un mot est reconnu par l'automate est alors le suivant :

```
dep=0
fin=NON
while fin!=OUI
{
  ligne=STRING
  if (*ligne=FIN_CHAINE)
        fin=OUI
  else
  {
        if (etats[dep+(*ligne*2)]==*ligne)
        {
              dep=etats[dep+(*ligne*2)+1]
              ligne++
        }
        else
              fin=OUI
  }
}
trouve=NON
if *ligne==FIN_CHAINE
  if ((etats[dep+POS_CODE]==SORTE_CODE) &&
              (etats[dep+POS_CODE+1]!=NON_TERMINAL))
                    trouve=OUI
```

La commande qui permet d'obtenir des automates de ce type est *_compac.*

4. L'INDEXATION DES AUTOMATES

Les automates permettent de représenter certains ensembles de mots (ensembles rationnels) de manière non ordonné. Il est possible d'enrichir ces structures pour obtenir des ensembles ordonnés suivant un ordre lexicographique (l'ordre du dictionnaire). Pour cela il faut que chaque état ait ses transitions codées suivant l'ordre alphabétique. A chaque état on attache (rectangles de la figure A1.8) le nombre de chemins possibles jusqu'à un état terminal. Ainsi, à partir de l'état initial 0, 3 chemins (un pour chaque mot) sont possibles alors qu'à parti de 5 seul le chemin vide est accepté. Cherchons le mot *est* dans cet automate. On cherche la transition *e* à l'état 0, cette transition existe mais suit une transition vers l'état 1 (pour *b*). Cela veut dire que, étant engagé dans ce chemin, on sait qu'il existe 1 (le rectangle de 1) chemin reconnu partant de 1, cela signifie qu'il existe déjà un mot précédant celui recherché. Dans l'état 6 on cherche une transition pour s, elle existe mais est précédée d'une transition vers 3, un autre mot précède donc celui recherché. Arrivé en 7 il n'y a qu'une transition. Arrivé en 5, qui est terminal, on a reconnu le mot et on sait que deux mots le précède dans l'ordre lexicographique. On a donc reconnu le troisième mot.

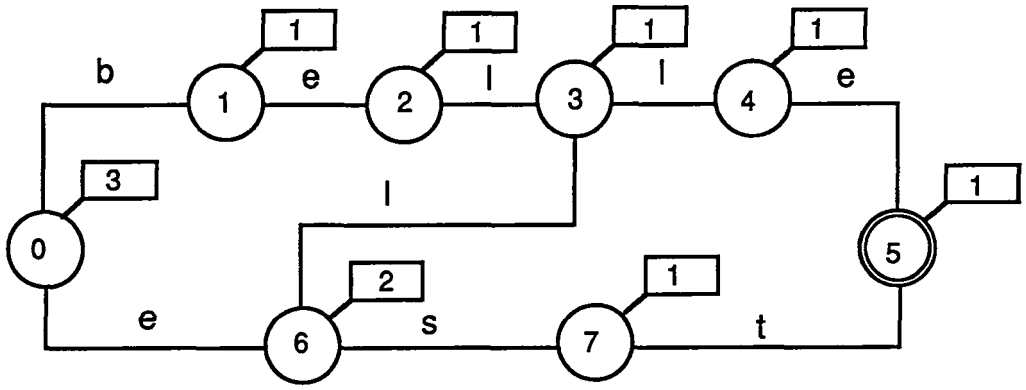


figure A1.8

Ce type d'automate est représenté dans les structures AUT_E_L* (AUT_E_L13).

On peut remarquer que ce type de représentation oblige à parcourir l'ensemble des transitions qui précèdent la transition recherché, cela est parfaitement naturel pour les automates en liste ou en liste tableau mais nous fait perdre l'avantage de l'accès direct pour les automates à accès direct compacté. Pour ces derniers on utilise un transducteur qui émet le nombre de mots qu'on sait être avant celui recherché. Ainsi à la fin du parcours, la somme des émissions donne le nombre de mots qui précède celui recherché. Ce type d'automate est stocké sur des structures de type AUT_T_* (e.g. AUT_T_D13)

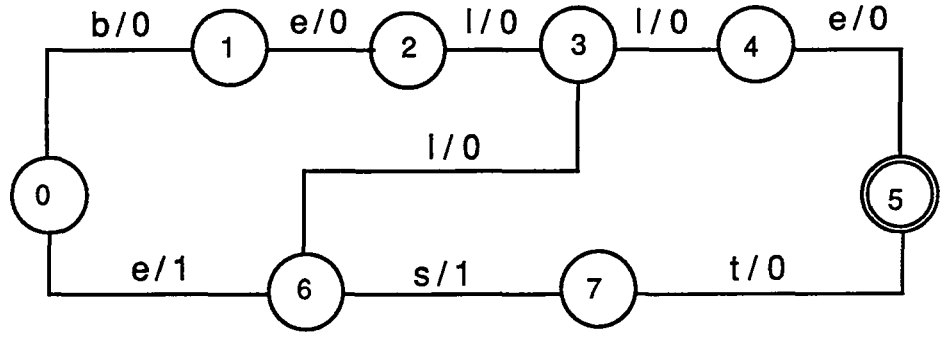


figure A1.9

5. LES STRUCTURES CORRESPONDANT A CHACUNE DE CES REPRESENTATIONS

En plus des différents choix ci-dessus, un autre critère intervient dans le choix des structures manipulés: la taille des alphabets et des automates. En effet, si les transitions de l'automate sont des lettres, le cardinal de l'alphabet est inférieur à 256, l'étiquette de la transition peut donc être stockée sur un octet. En revanche, si l'alphabet est plus grand, on veut pouvoir le stocker sur quatre octets. AUT_L13 signifie que l'automate est en liste et que l'étiquette est stockée sur un octet et l'état d'arrivée sur 3. Au contraire, AUT_L44 est également un automate en liste mais l'étiquette et l'état d'arrivé sont tous deux stockés sur quatre octets.Ainsi les automates on comme structure possible

AUT_L13
AUT_D13
AUT_LT13
AUT_L44
AUT_LT44

Par ailleurs, pour pouvoir manipuler des alphabet auxquels on peut ajouter des mots, on peut utiliser un automate de type AUT_T_D13 qui décrit en ensemble fixe (et peut-être important: plusieurs centaines de milliers de mots) et un arbre qui sera décrit par un automate de type AUT_L13. Ce type d'automate a pour structure **ALP_T_D13**.

Par ailleurs, la structure **_DELCOT** permet de traiter un ensemble d'automates de type *AUT_L44* pour des commandes comme *_mot_aut* ou *_ph_aut*. La structure **DELSYN** est utilisée pour les dictionnaires syntaxiques, elles contient des listes de *VREF_L44* qui sont des *AUT_L44* plus des pointeurs vers les transitions à supprimer.

ANNEXE 2 : LISTE DES STRUCTURES MANIPULEES

On se reportera au source de eroche.h pour voir une description précise de chacune des structures et en particulier de :

AUT_L13
AUT_D13
AUT_LT13
AUT_L44
AUT_LT44

ANNEXE 3 : FABRICATION D'UN TRANSDUCTEUR MORPHOLOGIQUE

La construction d'un transducteur morphologique à partir de
- le dictionnaire DELAF
- les préfixations et suffixations sur le DELAS
- les règle 2-level d'heuristique de raccordement des préfixes et des suffixes

se fait en utilisant en série des commandes PUPITRE. La construction utilise successivement 4 fichiers de commandes qui sont les suivant.

0. Compilation des règles

```
_pck -extalph <RUL_FILE | _lst_aut | _cal_crd | _compac |
_aut_alph >alph.rul
_pck -alph alph.rul <RUL_FILE
echo "on génère les automates 0.aut, 1.aut etc .. "
```

1. Construction du transducteur DELAF-DELAS à partir de l'automate par le fichier de commande FAIT1

```
_aff -lst <dlaf.eli |_morpho |_supp_ent|_defact "," |_defact ":"
>tmp1
_delaf_trans <tmp1 >tmp2
echo "on extrait l'alphabet des couples"
_fnc -constlex tmp2 tmp2.aut
_ext_alph <tmp2.aut|_aut_trans >alph2
_aut_trans -alph alph2 <tmp2.aut >dlaf.trans
```

2. Construction de l'automate *prefixe-deslas-suffixes inter règles*: fichier FAIT2

```
echo "passage du transducteur de delaf a celui du delas delas1"
_trad -transl44_autl44_1 <dlaf.trans >delas0
_aff -lst <delas0 |awk -F+ '{if (NF>1) print $1}' |sed "s/
//g"|sort -u >delas
_fdic -constlex delas delas.aut
_decomp <delas.aut >delas.aut2

echo "calcul de l'automate des prefixes à partir de leur liste"
echo "qui est dans prefixes.lst"
_lst_aut <prefixes.lst|_direct -sorte 0 5 |_direct -sorte 3
0>prefixes.aut1

echo "calcul de l'automate des suffixes"
_graph_autl44<suffixes.graph|_det|netpoubelle>sufff1
_trad -autl44_autl13<sufff1>suffixes.aut

echo "concaténation des préfixes et du delas"
_concat prefixes.aut1 delas.aut2 |_direct -sorte 3 5>tmp1
_det -newmax 50000 <tmp1 >tmp11
_mnm <tmp11 >tmp2

echo "on rajoute le cycle des prefixes"
```

```
echo "c'est à dire que tous les états qui ont une * doivent
pointer vers 0"
echo "avec l'étoile"
_aut_trans -etoile_init <tmp2 >tmp3

echo "on concatène avec A*"
_aut_trans -etend_suffixe <tmp3 >tmp4

echo "on fait l'intersection avec les suffixes"
_inter tmp4 suffixes.aut -newmax 100000>tmp5

echo "on transforme en transducteur et on fait l'intersection
avec les règles"
_trad -autl13_autl44 <tmp5 >tmp55
_reduc <tmp55 >tmp555
_aut_trans -gauche_atout <tmp555 >tmp6

echo "intersection avec chacune des règles successivement"
_ext_alph <regle.trans >alph.trans
_aut_trans -alph alph.trans <0.aut >0.trans
_inter tmp6 0.trans -newmax 100000 >tmp7
_trad -transl44_autl44 <tmp7 >tmp77
_reduc <tmp77 >tmmp

echo "automate 1"
_inter tmmp 1.aut -newmax 100000>tmmp1
_aff -mem <tmmp1
_reduc <tmmp1 >tmmp
_aff -mem <tmmp

echo "automate 2 à 10"
cp 2.aut res0
for i in 3 4 5 6 7 8 9 10
do
_inter res0 $i.aut >res1
_reduc <res1 >res0
done

_inter tmmp res0 -newmax 100000>tmmp1
_aff -mem <tmmp1
_reduc <tmmp1 >tmmp
_aff -mem <tmmp


echo "automate 11 à 20"
cp 11.aut res0
for i in 12 13 14 15 16 17 18 19 20
do
_inter res0 $i.aut >res1
_reduc <res1 >res0
done


ETC.....

_aff -mem <res0
_inter tmmp res0 -newmax 150000>tmmp1
_aff -mem <tmmp1
```

```
_reduc <tmmp1 >tmmp
_aff -mem <tmmp

_aut_trans -alph alph.trans <tmmp >delas6
```

3. Transformation du transducteur dels-delaf et règle en (pref)delaf-(pref)delas: fichier FAIT3:

```
echo "transformation du transducteur delas-delaf et règle en
(pref)delaf-(pref)delas"


echo "******************"
echo "PHASE 1 ************"
echo "fabrication du co_*inculperons<->co_*inculper+V+P1p"

echo "calcul de l'automate des prefixes à partir de leur liste"
echo "qui est dans prefixes.lst"
_lst_aut <prefixes.lst|_direct -sorte 0 5 |_direct -sorte 3
0>prefixes.aut1
_trad -autl13_autl44 <prefixes.aut1 >pref.aut2
_aut_trans -aut_identique <pref.aut2 |_trad -
transl44_autl44>pref.aut3

echo "la concatenation avec delas-delaf"
_trad -transl44_autl44 <dlaf.trans >dlaf.aut1
_concat pref.aut3 dlaf.aut1 |_direct -sorte 3 5>tmp1
_det -newmax 80000 <tmp1 >tmp11

echo "on rajoute le cycle des prefixes"
echo "c'est à dire que tous les états qui ont une *:* doivent
pointer vers 0"
echo "avec l'étoile"
_aut_trans -etoile_init <tmp11 >tmp3

_ext_alph <tmp3|_aut_trans>alph.trans1
_aut_trans -alph alph.trans1 <tmp3 >delaf1.trans


echo "******************"
echo "PHASE 2 *************"
echo "******************"
echo "fabrication de co_*inculperons<->coïnculperons"
_trad -transl44_autl44_2 <dlaf.trans >tmp1
_det -newmax 80000<tmp1 >tmp2
_mnm <tmp2 >delaf.aut0

echo "on concatène avec les préfixes"
_lst_aut <prefixes.lst|_direct -sorte 0 5 |_direct -sorte 3
0>prefixes.aut1
_trad -autl13_autl44 <prefixes.aut1 >pref.aut2
_concat pref.aut2 delaf.aut0 |_direct -sorte 3 5 >tmp1
_det -newmax 80000 <tmp1 >tmp11
_aut_trans -aut_identique<tmp11 |_trad -transl44_autl44 >tmp2
_aut_trans -etoile_init <tmp2 >tmp3
_ext_alph <tmp3|_aut_trans >alph0.trans
_aut_trans -alph alph0.trans<tmp3 |_trad -
transl44_autl44_1>tmp1
```

```
echo "on fait l'intersection avec chacune des règles"
_aut_trans -gauche_atout <tmp1 >tmp2

echo "intersection avec chacune des règles successivement"
_ext_alph <regle.trans >alph.trans
_aut_trans -alph alph.trans <0.aut >0.trans
_inter tmp2 0.trans -newmax 100000 >tmp7
_trad -transl44_autl44 <tmp7 >tmp2
_reduc <tmp2 >tmmp

echo "automate 1 à 10"
cp 1.aut res0
for i in 2 3 4 5 6 7 8 9 10
do
_inter res0 $i.aut >res1
_reduc <res1 >res0
done

_inter tmmp res0 -newmax 100000>tmmp1
_aff -mem <tmmp1
_reduc <tmmp1 >tmmp
_aff -mem <tmmp

echo "automate 11 à 20"
cp 11.aut res0
for i in 12 13 14 15 16 17 18 19 20
do
_inter res0 $i.aut >res1
_reduc <res1 >res0
done

_inter tmmp res0 -newmax 150000>tmmp1
_aff -mem <tmmp1
_reduc <tmmp1 >tmmp
_aff -mem <tmmp

echo "automate 21 à 30"
cp 21.aut res0
for i in 22 23 24 25 26 27 28 29 30
do
_inter res0 $i.aut >res1
_reduc <res1 >res0
done

_inter tmmp res0 -newmax 150000>tmmp1
_aff -mem <tmmp1
_reduc <tmmp1 >tmmp
_aff -mem <tmmp

ETC.....

mv tmmp delaf.aut3
_ext_alph <delaf.aut3 |_aut_trans >alph2.trans
_aut_trans -alph alph2.trans <delaf.aut3 >delaf2.trans
```

```
echo "*****************"
echo "PHASE 3 composition des deux delaf"
echo "*****************"
_compose delaf1.trans delaf2.trans -newmax
150000>delaf3.trans
_trad -transl44_autl44 <delaf3.trans >tmp1
_reduc <tmp1 >delaf3.aut0
```

5. Union finale en le DELAF-DELAS enrichie des préfixes et le DELAS enrichie des préfixations et des suffixations dérivationnelles. FAIT4:

```
set -v
echo "union des deux plus gros transducteurs"
_trad -transl44_autl44 <delas6 >tmp
_union -alph_un delaf3.aut0 tmp -newmax 200000>tmp2

echo "transformation en transducteurs"
_ext_alph <tmp2 |_aut_trans >alph
_aut_trans -alph alph <tmp2 >dlafmorpho.trans
```

ANNEXE 4 : CONSTRUCTION D'UN DICTIONNAIRE SYNTAXIQUE DE TYPE DELSYN

A partir des tables et des automates de références construits avec Editor ou pauto, la construction du dictionnaire de type DELSYN se fait principalement avec la fonction _fdelsyn. Pour faire des modifications: **1.** reconstruire tout :**constout**, **2.** modifier une table: **constab** (**transtot** et **typa** sont à modifier si on ajoute ou supprime une table), **3.** modification d'un automate de référence: **vref**.

Les différents fichiers de commandes valent :

CONSTOUT

```
vref
constab
```

VREF

```
vref1 1
vref1 2
echo "on met en commun les alphabets"
_lst_aut <alph.txt /_cal_crd/_compac/_aut_alph>alph
echo "on transforme chacun en un vref d'après le nouvel alphabet"
_fdelsyn -aut_vref alph <1.aut  >1.vref
```

CONSTAB

```
transtot
typa
sort -t, <dico.dic /_fact/_fdelsyn -slpt_entre_code >dic.dic
_lst_aut -code <dic.dic /_compac>abr.cut
_fdelsyn -assdic
echo "le resultat est dans delsyn"
```

VREF1

```
_graph_autl44 <$1.graph /_det/_mnm>$1.aut
_ext_alph <$1.aut/_aff -lst/_fdelsyn -mot_deb >>alph.txt
```

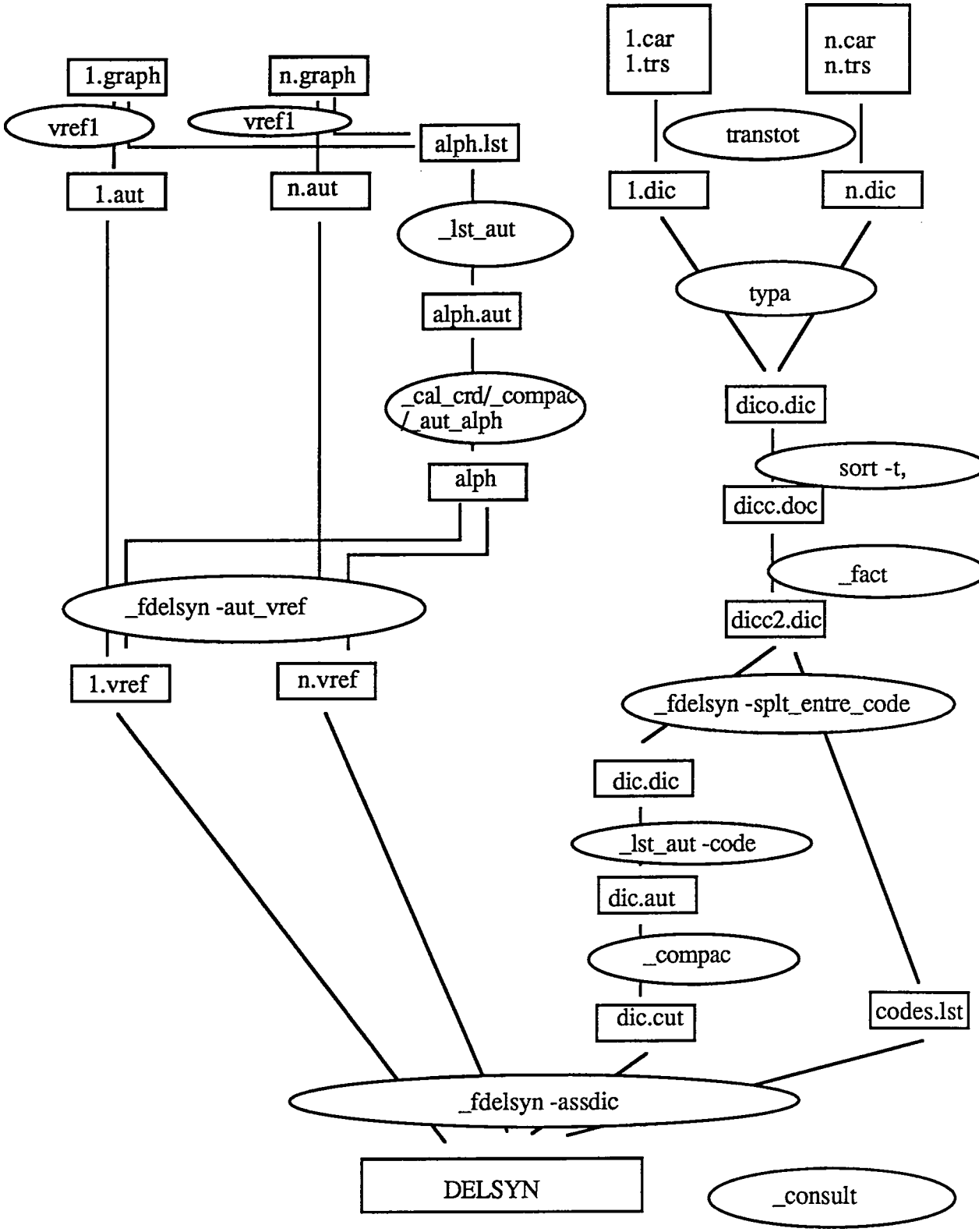
TRANSTOT

```
for i in 1 2 3 4 5 6 7 ..
do
  _fdelsyn -tabdic $i
done
```

TYPA

```
cp 1.dic dico.dic
for i in 2 3 4 5 ..
do
  cat $i.dic >>dico.dic
done
```

Le shéma de construction est le suivant:



ANNEXE 5 : CONSTRUCTION DE L'AUTOMATE SYNTAXIQUE AUTSYN

Nous décrivons ici les opérations nécessaires pour transformer le dictionnaire syntaxique de type DELSYN en un automate unique. On utilise le fichier faitautsyn qui lui même appel le fichier de commande ensyntax

faitautsyn:

```
echo "transforme delsyn en un automate unique de type AUT_L44"
echo "transformations premilinaires des automates de formalisme"
set -v

echo "CONSTRUCTION DE LA TRANSDUCTION DE DESABREVIATION"
_graph_autl44 <trans0.graph |_det|_mnm>trans0.aut
_graph_autl44 <trans1.graph |_det|_mnm>trans1.aut
_graph_autl44 <trans2.graph |_det|_mnm>trans2.aut
_graph_autl44 <deuxcro.graph |_det|_mnm>deuxcro
_graph_autl44 <n.graph |_det|_mnm>n.aut
_graph_autl44 <lesmauvais.graph |_det|_mnm>lesmauvais.aut
_union -alph_un trans1.aut trans2.aut >trans3.aut

echo "CONSTRUCTION DE LA GRAMMAIRE PROPREMENT DITE"
echo "Pour chacune des tables on construit l'automate"
echo "table 4"
_consult -table_trans 4 >table

echo "0 50"
_fdic -splt table table.tmp 0 50
_consult -union_verbe <table.tmp |_reduc |_mnm >resa
ensyntax resa res
ls -l res*
echo "50 100"
_fdic -splt table tmp2 50 100
_consult -union_verbe <tmp2 >resa
ensyntax resa res2
_union -alph_un res res2 >resi
_mnm <resi >res3
ls -l res*
echo "on rajoute les cycles en transformant ??? en (??)+"
_desabr -raj_cycles <res >4.aut1
ls -l res*


echo "on rajoute la syntaxe du groupe nominal"
ensyntax n.aut n.gram
```

```
_union -alph_un res n.gram |_mnm |_desabr -
raj_cycles>gram.aut
```

ensyntax:

```
_desabr -auttr_simple trans0.aut <$1|_desabr -syn1 >at0
_trad -autl44_autlt44 <at0 |_diff lesmauvais.aut |_trad -
autlt44_autl44|_reduc|_det>at1
_desabr -auttr_simple trans3.aut <at1 |_det -newmax
100000|_mnm >at2
echo "supprime les sequences @] [@"
_desabr -supp_crochet <at2 >at3
_trad -autl44_autlt44 <at3 |_diff deuxcro |_trad -
autlt44_autl44|_reduc|_det>$2
```

ANNEXE 6 CONSTRUCTION DU DICTIONNAIRE MORPHOLOGIQUE DES AUTOMATES.

L'opération consiste à prendre les codes du dictionnaire dlaf.cod du type

8393
162254
1.N1:mp
.N1:ms
.N21:fp
2re.V3:T3s

et à le transformer en une structure _DELCOT qui, à chaque ligne de dlaf.cod, associe un DAG représentant ces informations. C'est cette structure qui est utilisée par les commandes **_mot_aut**, **_ph_aut**, **_pgrep**, **_plgrep** et **_analyse**.

Cette opération exige que l'on précise la syntaxe de ces automates, c'est à dire qu'il faut spécifier un langage L1 (décrit par un automate) tel que quelque soit l'automate A d'une entrée L(A) soit inclus dans L1. Concrètement, pour un nom il faut décider dans quel ordre mettre les attributs (forme canonique, genre, nombre).

Pour construire ce type de dictionnaire il faut suivre la procédure de modification de _conscot suivante:

Le programme _conscot est compoé d'un fichier _conscot.c qui contient toutes les fonctions générales, il faut rajouter à ce ernier un fichier _conscN.c qui dépendra du type de dictionnaire. Par exemple le fichier _consc1.c décrit la manière de transformer le dictionnaire du français. Il faut donc

1. Modifier _consc1.c

2. Recompiler

3. Appliquer la commande cons1_delaf suivante

cons1_delaf
_concot -2pr dlaf.cod tmp.cod
_init -alptd13 >alp.tmp
s1 0 500
cp res.tmp cod1
...
_conscot -2 -f cod1 -alp alph.tmp
_graph_autl44 <autsyn.graph /_det/_mnm >autsyn
_conscot -3 autsyn.aut

et **s1** vaut
_conscot -tp 1 -alph alph.tmp -f dlaf.cod -dep $1 -l $2

Annexe 7 : Structures manipulées en C

Nous donnons ici le fichier eroche.h inclus dans toutes les commandes présentées ici, fichier qui donne la définition exacte de toutes les structures manipulées

```
/*****************************
      eroche.h

structures générales
*****************************/

#ifndef EROCHE

#define EROCHE

#include <stdio.h>
#include <string.h>


#include <stdlib.h>

#ifdef OS2



#include <stdlib.h>

#endif


unsigned char *tabcod;


#define OUI -19
#define NON -18
#define C1 26  /* caractère de fin de fichier*/
#define C2 253
#define C3 252
#define FIN_ETAT 65001
#define FIN_AUT 65000
#define SS_TRANS -1 /**** quand pas de transition pour la lettre */
#define ETT 30
#define TERMINAL 5  /** terminal vaut 5 si l'etat est terminal */
#define NON_TERMINAL 0 /* 0 sinon */
#define TROU 64999
#define TRS_FIN '$'
#define ETT2 -1     /*postion du code de l'état dans les AUT_D??*/
#define FIN_LONG 10000000
#define FIN_SHORT 65002
#define LONG_OUT 100000000
#define FIN_CHAR 255
#define SS_T '&'
#define SS_T2 -2
```

```
typedef struct strcel22 {
   unsigned int g;
   unsigned int d;
   } CEL22;
#define TAILLE_CEL22 (2*sizeof(int))


typedef struct st_ilst {
   int entier;
   struct st_ilst *suivant;
       } *I_LISTE;
#define TAILLE_ILISTE (sizeof(int) + sizeof(I_LISTE))

typedef struct st_ilst_l {
   long entier;
   struct st_ilst_l  *suivant;
       } *L_LISTE;
#define TAILLE_LLISTE (sizeof(long) + sizeof(L_LISTE))


typedef union {
   unsigned char str[4];
   unsigned long l;
   } UBLOC;
#define TAILLE_UBLOC (sizeof(long))

typedef union {
   unsigned char str[2];
   unsigned short i;
   } IBLOC;

typedef union {
   unsigned short i;
   struct {
   unsigned int a0:1;
   unsigned int a1:15;
   } s;

   } BBLOC;


typedef struct str2_cell {
      int cel_d;
      int cel_g;
      struct str2_cell *suivant;
        } *CELL2;
#define TAILLE_CELL2 (2*sizeof(int)+sizeof(CELL2))

typedef struct str4_cell {
      unsigned long cel_d;
      unsigned long cel_g;
      struct str4_cell *suivant;
        } *CELL44;
#define TAILLE_CELL44 (2*sizeof(long)+sizeof(CELL44))
```

```
typedef struct str4_cellt {
        unsigned long cel_d;
        unsigned long cel_g1;
        unsigned long cel_g2;
        struct str4_cellt *suivant;
                                } *CELLT44;
#define TAILLE_CELLT44 (3*sizeof(long)+sizeof(CELLT44))

typedef struct str3_cellt {
        UBLOC nombre;
        unsigned long cel_g2;
        struct str3_cellt *suivant;
                                } *CELLT13;
#define TAILLE_CELLT13 (2*sizeof(long)+sizeof(CELLT13))



typedef struct s_cell13{
   UBLOC nombre;
   struct s_cell13 *suivant;
   } *CELL13;
/*#define TAILLE_CELL13 (sizeof(long)+sizeof(CELL13))*/
#define TAILLE_CELL13 16



/*****************************
      structures d'alphabets

*****************************/

/*automates non indexés de représentation*/

typedef struct {
   CELL13 liste;
   unsigned long sorte;
   } *ET_L13;
#define TAILLE_ETL13 (sizeof(CELL13)+sizeof(long))

typedef struct {
   ET_L13 *etats;
   unsigned long nb_etat;
   unsigned long taille;
   } *AUT_L13;
#define TAILLE_AUTL13 (sizeof(ET_L13 *)+sizeof(long)*2)
#define T_AUT_L13 48

typedef struct {
   UBLOC **etats;
   unsigned short *sorte_etats;
   unsigned long nb_etat;
   unsigned long taille;
   } *AUT_LT13;
#define TAILLE_AUTLT13 (sizeof(UBLOC **)+sizeof(short *)+2*sizeof(long))
#define T_AUT_LT13 47

typedef struct {
   UBLOC *bloc;
```

```
   unsigned long nb_etat;
   unsigned long taille;
   } *AUT_D13;
#define TAILLE_AUTD13 (sizeof(UBLOC *)+sizeof(long)*2)
#define T_AUT_D13 49

#define TROU_D13 255
#define SORTE_POS_D13 -1
#define COD_SORTE_D13 254


/*automates indexés de représentation*/

typedef struct {
   unsigned long sorte;
   unsigned long carde;
   CELL13 liste;
   } *ET_E_L13;
#define TAILLE_ETEL13 (2*sizeof(long) + sizeof(CELL13))



typedef struct {
   unsigned long nb_etat;
   ET_E_L13 *etats;
   unsigned long nb_mot;
   } *AUT_E_L13;
#define TAILLE_AUTEL13 (2*sizeof(long)+sizeof(ET_E_L13 *))
#define T_AUT_E_L13 78

typedef struct {
   unsigned short *sorte_etats;
   unsigned long *carde_etats;
   UBLOC **etats;
   unsigned long taille;
   unsigned long nb_etat;
   unsigned long nb_mot;
   } *AUT_E_LT13;
#define TAILLE_AUTELT13 (sizeof(long *)*3)

typedef struct {
   UBLOC *bloc;
   unsigned long *cardt;
   unsigned long taille;
   unsigned long nb_etat;
   unsigned long nb_mot;
   } *AUT_T_D13;
#define TAILLE_AUTTD13 (sizeof(long)*3+sizeof(long *)+sizeof(UBLOC *))
#define T_AUT_T_D13 85


/*les alphabets complets*/

typedef struct {
   AUT_E_LT13 aut1;
   AUT_L13 aut2;
   unsigned long nb_mot1;
   unsigned long nb_mot;
```

```
   unsigned char **mot2s;
   } *ALP_E_LT13;
#define TAILLE_ALPELT13 (sizeof(AUT_E_LT13)+sizeof(AUT_L13)+2*sizeof(long))
#define T_ALPELT13 43

typedef struct {
   AUT_T_D13 aut1;
   unsigned int stat;   /*vaux 1 si l'automate est déjà en mémoire*/
   AUT_L13 aut2;
   unsigned long nb_mot1;
   unsigned long nb_mot;
   unsigned char **mot2s;
   unsigned long taille;
   } *ALP_T_D13;
# d e f i n e                    T A I L L E _ A L P T D 1 3
(sizeof(AUT_T_D13)+sizeof(AUT_L13)+3*sizeof(long)+sizeof(int))
#define T_ALP_T_D13 44

/*************************

     les automates sur des mots

*************************/

typedef struct
{
   ALP_E_LT13 alp_elt13;
   ALP_T_D13 alp_td13;
   CEL22 *bloc;
   unsigned long nb_etat;
   unsigned long taille;
} *AUT_D22;
#define TAILLE_AUTD22
#define T_AUT_D22 45

typedef struct {
   CELL2 liste;
   unsigned long sorte;
   } *ET_L22;
#define TAILLE_ETL22 (sizeof(long)+sizeof(CELL2))

typedef struct {
   ALP_E_LT13 alp_e_lt13;
   ALP_T_D13 alp_t_d13;
   ET_L22 *etats;
   unsigned long nb_etat;
   unsigned long taille;
   } *AUT_L22;
#define TAILLE_AUTL22 (sizeof(ALP_E_LT13)+sizeof(ALP_T_D13)+sizeof(ET_L22
*)+sizeof(long)*2)
#define T_AUT_L22 46

typedef struct {
   ALP_E_LT13 alp_e_lt13;
   ALP_T_D13 alp_t_d13;
   unsigned short **etats;
   unsigned long nb_etat;
   } *AUT_LT22;
```

```
#define TAILLE_AUTLT22 (sizeof(ALP_E_LT13)+sizeof(ALP_T_D13)+sizeof(short
*)+sizeof(long))
#define T_AUT_LT22 47

typedef struct {
   CELL44 liste;
   unsigned long sorte;
   } *ET_L44;
#define TAILLE_ETL44 (sizeof(long)+sizeof(CELL44))


typedef struct {
   ALP_E_LT13 alp1;
   ALP_T_D13 alp2;
   ET_L44 *etats;
   unsigned long nb_etat;
   unsigned long taille;
   } *AUT_L44;
#define TAILLE_AUTL44 (sizeof(ALP_E_LT13)+sizeof(ALP_T_D13)+sizeof(ET_L44
*)+sizeof(long)*2)
#define T_AUT_L44 57

typedef struct {
        unsigned long g;
        unsigned long d;
        } BLOC_D44;
#define TAILLE_BLOCD44 (2*sizeof(long))

typedef struct {
        ALP_T_D13 alp2;
        BLOC_D44 *bloc;
        unsigned long nb_etat;
        unsigned long taille;
        } *AUT_D44;
#define  TAILLE_AUTD44  (sizeof(ALP_T_D13)  +  sizeof(BLOC_D44
*)+2*sizeof(long))
#define T_AUT_D44 166

#define SORTE_POS_D44 -1
#define COD_SORTE_D44 1000001


typedef struct {
        unsigned long nb_tube;
        unsigned long *tube_nom;
        unsigned long **tube_in;
        unsigned long **tube_out;
                        } *TUBE_INTER_SYN;
#define         TAILLE_TUBEINTERSYN (sizeof(long)+sizeof(long *)+sizeof(long
**)*2)

#define MAX_ENTRE_TUBE 20


typedef struct {
  AUT_L44 aut;
  unsigned long **f;
        TUBE_INTER_SYN tubes;
```

```
  } *RES_INTER_L44;
#define TAILLE_RESINTERL44 (sizeof(AUT_L44)+sizeof(long
**)+sizeof(TUBE_INTER_SYN))
#define T_INTER_L44 76


typedef struct {
   ALP_E_LT13 alp_elt13;
   ALP_T_D13 alp_td13;
   unsigned long **etats;
   unsigned long *sorte_etats;
   unsigned long nb_etat;
   } *AUT_LT44;
# d e f i n e T A I L L E _ A U T L T 4 4 3 2
/*(sizeof(ALP_E_LT13)+sizeof(ALP_T_D13)+sizeof(long **)+sizeof(long)+sizeof(long
*))*/
#define T_AUT_LT44 87

typedef struct {
  AUT_LT44 aut;
  unsigned long **f;
        TUBE_INTER_SYN tubes;
  } *RES_INTER_LT44;
#define TAILLE_RESINTERLT44 (sizeof(AUT_LT44)+sizeof(long **))
#define T_INTER_LT44 176


/*structures spécifiques des dictionnaires*/


typedef struct {
   AUT_LT44 *auts;
   ALP_E_LT13 alp1;
   ALP_T_D13 alp2;
   AUT_L44 autsyn;
   unsigned char ***fcan;
   unsigned char ***pos_can;
   unsigned char *nb_can;
   unsigned char **posmot;
   unsigned long nb_aut;
   unsigned long nb_etat_tot;
   unsigned long nb_trans_tot;
   unsigned long nb_fcan_tot;
   unsigned long nb_let_fcan;
   unsigned long nb_poscan_tot;
   unsigned long nb_posmot_tot;
   } *_DELCOT;
#define T_DELCOT 123
#define TAILLE_DELCOT 60

typedef struct {
   AUT_LT22 *auts;
   ALP_E_LT13 alp1;
   ALP_T_D13 alp2;
   AUT_L44 autsyn;
   unsigned char ***fcan;
   unsigned char ***pos_can;
   unsigned char *nb_can;
```

```
   unsigned char **posmot;
   unsigned long nb_aut;
   unsigned long nb_etat_tot;
   unsigned long nb_trans_tot;
   unsigned long nb_fcan_tot;
   unsigned long nb_let_fcan;
   unsigned long nb_poscan_tot;
   unsigned long nb_posmot_tot;
   } *_MINI_DELCOT;
#define T_MINI_DELCOT 123
#define TAILLE_MINI_DELCOT 60


typedef struct {
   AUT_L44 aut;
   int nb_prop;
   CELL44 *props;
   } *VREF_L44;
#define TAILLE_VREFL44 (sizeof(AUT_LT22)+sizeof(int)+sizeof(int *))
#define T_VREF_L44 55

typedef struct {
   AUT_LT44 aut;
   int nb_prop;
   unsigned long **props;
   } *VREF_LT44;
#define TAILLE_VREFLT44 (sizeof(AUT_LT22)+sizeof(int)+sizeof(int *))
#define T_VREF_LT44 55


typedef struct {
   unsigned int nb_ref;
   VREF_LT44 *vrefs;
   ALP_T_D13 alph;
   unsigned char ***codes;
   unsigned int *nb_carac;
   int nb_code;
   int nb_entre;
   AUT_D13 aut;
   } *DELSYN;
#define TAILLE_DELSYN 128
/* (sizeof(int)+sizeof(VREF_LT44 *)+sizeof(ALP_T_D13)+sizeof(char ***)+sizeof(int
*)+sizeof(int)+sizeof(int)+sizeof(AUT_D13)*/
#define T_DELSYN 137

typedef struct {
        unsigned char *commentaire;
        unsigned long *ref;
        unsigned long nb_entree;
        unsigned long nb_char;
                } *TAB_T_D13;
#define TAILLE_TABTD13 16
#define T_TAB_T_D13 243

typedef struct {
   CELLT44 liste;
   unsigned long sorte;
   } *ET_T_L44;
```

```
#define TAILLE_ETTL44 (sizeof(long)+sizeof(CELLT44))

typedef struct {
   ALP_E_LT13 alp1;
   ALP_T_D13 alp2;
   ET_T_L44 *etats;
   unsigned long nb_etat;
   unsigned long taille;
   } *TRANS_L44;
# d e f i n e                    T A I L L E _ T R A N S L 4 4
(sizeof(ALP_E_LT13)+sizeof(ALP_T_D13)+sizeof(ET_T_L44 *)+sizeof(long)*2)
#define T_TRANS_L44 157


typedef struct {
   CELLT13 liste;
   unsigned long sorte;
   } *ET_T_L13;
#define TAILLE_ETTL13 (sizeof(long)+sizeof(CELLT13))

#define EPSI_TRANSL13 253

typedef struct {
   ET_T_L13 *etats;
   unsigned long nb_etat;
   unsigned long taille;
   } *TRANS_L13;
#define TAILLE_TRANSL13 (sizeof(ET_T_L13 *)+sizeof(long)*2)
#define T_TRANS_L13 158

typedef struct {
   unsigned long **etats;
   unsigned long *sorte_etats;
   unsigned long nb_etat;
   unsigned long taille;
   } *TRANS_LT13;
#define TAILLE_TRANSLT13 (sizeof(long **)+sizeof(long *)+2*sizeof(long))
#define T_TRANS_LT13 159


#ifndef MAX_TAS

#define MAX_TAS 40000

#endif



#endif
```