QUEEN
QA
76.9
.N38
F47614
1989

Gouvernement du Canada
Ministère des Communications

Government of Canada
Department of Communications

Le Centre canadien de recherche sur l'informatisation du travail
Canadian Workplace Automation Research Centre

LE PROJET TULIP

par

Ron Ferguson

QA
76.9
N38
F47614
1989

Canada

QA
76.9
N38
F476f
1989

Industry Canada
LIBRARY
AOUT AUG 04 1998
BIBLIOTHÈQUE
Industrie Canada

LE PROJET TULIP

par

Ron Ferguson

Ce rapport est le résultat de travaux de recherche effectués au Centre canadien de recherche sur l'informatisation du travail sous la direction de Joël Muzard, Groupe systèmes experts.

Ministère des Communications

Laval

Novembre 1989

COMMUNICATIONS CANADA
JUIL JULY 11 1991
LIBRARY — BIBLIOTHÈQUE

Nº de cat. Co 28-1/40-1989F
ISBN 0-662-95734-2

Les opinions émises dans ce rapport n'engagent que l'auteur.

* This report is also available in English

QA
76.9
A38
1989
c.3

DD 9352594
DL 10644195

Table des matières

Introduction.....3

Implantation du système Tulip.....3

Construction du système Tulip.....5

Les fichiers sources du système Tulip.....7

Heuristiques d'analyse de cartes professionnelles.....13

L'architecture du système Tulip.....13

La représentation des connaissances dans le système Tulip.....15

Fonctions de Skolem.....16

Prédicats binaires.....17

Faits connus et inférés.....19

Représentation des verbes.....20

Introduction

Ce rapport discute de l'implantation du système Tulip dans les postes de travail Sun. Les possibilités offertes par le système Tulip, ainsi que ses limites, sont décrites dans le guide de l'utilisateur du système Sun-Tulip (Ferguson 89b).

Implantation du système Tulip

La version Sun du système Tulip est écrite en langage Quintus Prolog et comporte quelques sous-programmes C. Elle fonctionne d'après le système d'exploitation UNIX. Le code source est mis à jour à l'aide du système de contrôle de codes sources (SCCS) UNIX. Le code source du système Tulip figure sous le répertoire **/home/condor/ferguson/tulip.** Tous les sous-répertoires mentionnés dans ce rapport sont des sous-répertoires de **/home/condor/ferguson/tulip.** Le répertoire **tulip** contient les sous-répertoires suivants :

main	-	contient le code source principal pour l'analyse des phrases.
business-card	-	contient le code pour l'analyse des cartes professionnelles et des noms.
expert-system	-	contient le code pour les connaissances intégrées de la date et de l'heure, les opérations arithmétiques et les réponses aux questions.
extraposition-grammar	-	code pour la traduction de grammaires d'extraposition en clauses Prolog.
foreign-files	-	code de commande pour les interfaces avec les sous-programmes C.

help	-	fonction d'assistance du système Tulip.
knowledge	-	fichiers de connaissances qui peuvent être lus par le système Tulip.
lexicon	-	dictionnaire du système Tulip.
transform	-	interpréteur de commandes d'écriture de préprocesseur Prolog.
tulip	-	images exécutables.
documentation	-	documentation du système Tulip.

Construction du système Tulip

Dans le sous-répertoire **main**, les fichiers *parse.pl* et *date-time-parse.pl* contiennent des grammaires d'extraposition [Pereira 83] qui doivent être traduites en clauses Prolog. Les fichiers Prolog correspondants sont *translation-of-parse.pl* et *translation-of-date-time-parse.pl* respectivement. Si l'un des fichiers contenant une grammaire d'extraposition est modifié, il doit être retraduit. Pour retraduire un fichier, chargez Tulip en l'extrayant du sous-répertoire **main** et compilez *xg-fix.pl.* :

compile(.../extraposition-grammar/xg-fix).

Faites ensuite évaluer *gxg* (défini dans *xg-fix.pl*) :

gxg.

Le prédicat *gxg* vous demandera de spécifier des fichiers d'entrée et de sortie. Introduisez *parse.pl* (ou *date-time-parse.pl*) comme fichier d'entrée et *translation-of-parse.pl* (ou *translation-of-date-time-parse.pl*) comme fichier de sortie.

Dans le sous-répertoire **business-card**, le fichier *address.pl* contient une grammaire étendue à clauses définies qui sert à l'analyse des cartes professionnelles [Ferguson 89]. Ce fichier doit être prétraité lorsqu'il est modifié. Pour utiliser le préprocesseur, chargez Tulip comme ci-dessus et faites exécuter *gopt* :

gopt.

Le prédicat gopt vous demandera de spécifier des fichiers d'entrée et de sortie. Introduisez *.../business-card/address.pl* comme fichier d'entrée et *.../business-card/tr-address.pl* comme fichier de sortie.

Le fichier *tulip-system.pl* du sous-répertoire **main** contient les commandes de compilation du système Tulip. Si l'on compile *tulip-system.pl*, le système Tulip est recompilé et un fichier image est sauvegardé dans le sous-répertoire **tulip**. Le sous-répertoire **main** contient un fichier exécutable appelé *tulip* qui charge cette image.

Le système Tulip utilise un lexique très étendu. Celui-ci est constitué de faits Prolog contenus dans une série de fichiers. Lorsque Tulip a été implanté pour la première fois dans des postes de travail Sun, ces fichiers ont été chargés dans le système. Toutefois, le temps de réponse de Tulip sur les machines Sun 3/50 s'est avéré trop long. Il semblerait que cette faiblesse soit en partie causée par la trop grande quantité de mémoire nécessaire pour la pagination, en raison de la taille du programme Tulip. En vue d'accroître la vitesse d'exécution de ce dernier, le lexique a été stocké dans des fichiers de gestion de base de données (dbm) et seules les entrées du lexique qui correspondent aux mots apparaissant dans les phrases d'entrée sont enregistrées en mémoire. En déchargeant ainsi le lexique, on a pu réduire l'espace de mémoire virtuelle nécessaire au programme Tulip, lequel fonctionne maintenant 3,4 fois plus rapidement.

Le lexique de la base de données est constitué d'enregistrements. Chaque enregistrement comprend un champ clé qui est un mot anglais et un champ de contenu qui est un nombre entier. Ce dernier est une représentation vectorielle binaire d'un ensemble qui est un sous-ensemble de «lex_set», lequel est défini dans *lexical-dictionary-interface.pl*. Les éléments de cet ensemble sont les noms des prédicats du lexique Prolog qui ne peuvent avoir pour argument qu'un mot anglais. Par exemple, si *noun(dog))* est un fait dans le lexique Prolog, *noun* est un élément de l'ensemble associé au mot *dog*.

Le lexique de la base de données se trouve dans les fichiers *lexicon.dir* du répertoire **lexicon/dbm/**. Lorsque ces fichiers doivent être reconstitués, vous pouvez suivre la procédure suivante :

Effacez les fichiers *lexicon.pag* et *lexicon.dir*.
Créez des fichiers *lexicon.pag* et *lexicon.dir* vides.
Passez au sous-répertoire **main**.
Compilez *lexicon-system.pl* (ce fichier est un fichier de compilation pour le lexique Prolog).
Chargez le système Tulip sans toutefois entrer (restez dans le système Prolog).
Entrez dans **create_lexicon**.
Prenez note qu'il faut plusieurs heures pour recréer le lexique de la base de données. Si vous ne désirez y apporter que quelques modifications

mineures, il peut s'avérer plus pratique de modifier le lexique existant en utilisant les prédicats des sous-répertoires **dbm.pl** et **fixed-set.pl** qui se trouvent respectivement dans les répertoires **lexicon/dbm** et **main.** Évidemment, en procédant de la sorte, il faudra également effectuer les mêmes modifications dans les fichiers du lexique Prolog. Une liste de ces fichiers est disponible dans le fichier *main/lexicon-system.pl* et une description de ces fichiers est disponible dans [Robb 88].

Certains des fichiers du lexique contiennent des prédicats ayant une arité supérieure à 1. Ces prédicats ne figurent pas dans le lexique de la base de données. Afin de réduire la charge de pagination, le fichier *tulip-system.pl* n'inclut ces fichiers dans le système Tulip que si le fait *lots_of_memory* figure dans la base de données Prolog. Si tel n'est pas le cas, ces fichiers ne sont pas inclus lors de la compilation du système Tulip.

Les fichiers sources du système Tulip

Vous trouverez ci-après une liste des fichiers sources non inclus dans le lexique du système Tulip, ainsi qu'une description de leur contenu. Les noms d'accès de ces fichiers se rapportent au sous-répertoire **main**. Chacun d'eux porte une extension *.pl* qui n'apparaît cependant pas dans la liste.

parse
Grammaire anglaise principale.

date-time-parse
Analyse syntaxique de la date et de l'heure.

.../business-card/address
Analyseur syntaxique de cartes professionnelles.

utilities
Prédicats utilitaires généraux.

collections
Opérations sur les ensembles.

...help/help
Système de menus d'assistance.

tulip-date
Date avec jour de la semaine.

accept_sentence
Lecture d'une phrase.

.../transform/transform
Interpréteur pour transformation de clauses Prolog (préprocesseurs).

tokenize
Analyseur lexical : chaîne __ liste d'éléments lexicaux (réversible).

lexical_structure
Définition des types de données des structures lexicales.

guess-lexical-type
Catégorisation lexicale approximative des mots qui n'apparaissent pas dans le lexique.

.../extraposition-grammar/xg-fix-aux
Non-terminaux de base pour grammaires d'extraposition et enregistrement de mots analysés.

fp
Opérations sur les tableaux (listes de listes).

abbrev
Abréviations.

synonym
Synonymes.

french-english
Traduction de mots du français à l'anglais.

new-lexicon
Lexique de base.

attachment-control
Masques qui contrôlent le lien modificateur.
Voir F. Pereira, «Logic for Natural Language Analysis», p. 64.

top
Routines Prolog de haut niveau pour Tulip.

execute_command
Éxécution des commandes intégrées de Tulip.

chat-lexicon-interface
Recherche de mots dans le lexique.

morphology
Analyse morphologique.

slot-filler
Remplacement des mots par leurs «significations» dans un arbre d'analyse.

simplify
: Simplification des arbres *quant* par élimination des feuilles «vrai».

scope-determination
: Détermination de l'étendue des quantificateurs anglais.

template
: Interface de liaison avec les modèles qui définissent la «signication» des mots.

predicate-calculus
: Conversion en forme clausale des formules de calcul des prédicats.

modify-determiner
: Modification des déterminants des arbres d'analyse d'assertions pour faire en sorte que la quantification soit plus explicitement représentée.

knowledge-file-access
: Lecture et écriture dans les fichiers de connaissances.

transform-answer
: Addition de faits dans la base de connaissances ou consultation de cette dernière.

parse-names
: Analyse d'un nom, d'une adresse ou d'une autre entité non grammaticale.

anaphora
: Traitement des références anaphoriques.

triggers
: Déclenchement d'interventions basées sur les assertions transmises à la base de connaissances.

parse-mods

Modification de la grammaire pour traiter les conjonctions des syntagmes verbaux.

special-nps

Analyseurs spéciaux pour identifier les entités non-grammaticales (par exemple, les noms, adresses et numéros de téléphone de personnes et de sociétés).

.../business-card/gr-transform

Traduction d'une grammaire étendue à clauses définies (pour les cartes professionnelles) en clauses Prolog.

.../business-card/address-util

Analyse des éléments lexicaux de carte professionnelle.

.../business-card/heuristics

Heuristiques pour étiqueter les parties d'une carte professionnelle que la grammaire de cartes professionnelles n'était pas en mesure d'analyser.

.../business-card/semantics

Traitement de la spécification des relations entre les objets de carte professionnelle.

infer

Inférences de base.

loop-checking

Vérification des règles pour détecter les boucles récursives qui peuvent s'y trouver.

database-access

Assertion de clauses révocables.

significant-words

Extraction d'assertions par association de mots.

date-time-facts
: Faits relatifs à la date et à l'heure.

mrl_temp
: Interface de liaison aux *modèles de représentation de sens* des verbes.

type-hierarchy
: Spécification d'une hiérarchie intégrée pour l'appariement des modèles.

.../expert-system/date-time-expert
: Moteur du système expert s'occcupant de la date et de l'heure, c'est-à-dire, le calculateur.

.../expert-system/talkr
: Évaluation d'une interrogation.

.../expert-system/aggreg
: Éxécution d'opérations arithmétiques.

quintus_kludges
: Prédicats qui ne peuvent être définis correctement (ou qui ne l'ont pas été) dans la version Quintus du système Tulip.

La table précédente a été créée par un programme Prolog du fichier *documentation/file-comments.pl* qui reçoit le commentaire d'en-tête provenant de chaque fichier. Le fichier *main/tulip-system.pl* contient les instructions (commentées) pour faire exécuter le programme de commentaires d'en-tête. Le fichier *main/header-comment.pl* contient un modèle pour les commentaires d'en-tête.

Heuristiques d'analyse de cartes professionnelles

Tel qu'il est exposé dans [Ferguson 89], on a eu recours à des heuristiques pour classifier les parties de cartes professionnelles qui ne pouvaient pas être classifiées par la grammaire de cartes professionnelles de Tulip. Toutefois, cette grammaire peut être appliquée à l'analyse de l'information pour laquelle les heuristiques de cartes professionnelles ne sont pas appropriées (c'est-à-dire les organismes qui oeuvrent dans l'intelligence artificielle). Par conséquent, Tulip a été modifié de façon que les heuristiques ne soient appliqués que lorsque le fait *apply_heuristics* est défini. Le drapeau *apply_heuristics* est vérifié dans le fichier *business-card/address.pl*.

L'architecture du système Tulip

L'architecture du système Tulip est basée sur celle du système Chat-80 [Pereira 83]. Les différences entre les deux systèmes sont discutées dans [Ferguson 88a].

Le système Tulip utilise une architecture phasée pour analyser les phrases. L'analyse de la phrase entière est donc faite suivant plusieurs phases successives. Pour une phrase en anglais, ces phases sont :

Processus	Prédicat
- lecture de l'information d'entrée	read_lines/6
- analyse lexicale	tokenize_parse/3
- consultation du lexique syntaxique	lexical_lookup/2
- localisation des mots significatifs	significant_words/4
- analyse syntaxique	sentence/6
- consultation du lexique sémantique	i_sentence/2
- détermination de l'étendue des poids	clausify/2
- simplication d'un arbre de quantification	simplify/2

- identification des entités non-grammaticales	transform_special_nps/2
- localisation des références pronominales	anaphoric_references/3
SI requête	
évaluation de la requête	answer/2
génération de la réponse	surface_descriptions/2
affichage de la réponse	respond/1
SINON % assertion ou règle	
conversion en clauses Horn	translate_to_clausal_form/4
mise à jour de la base de connaissances	update_knowledgebase/2
FIN	

L'approche phasée est avantageuse du point de vue du génie logiciel, car elle divise l'ensemble de la tâche en plusieurs tâches plus petites et largement indépendantes. Chacune de ces dernières est plus facile à comprendre et à mettre au point que ne le serait une seule et unique tâche résultant de leur fusion.

S'il y a de petits retours en arrière (backtracking) entre les différentes phases, on peut détecter et corriger les erreurs avec une plus grande précision que dans une approche intégrée. S'il n'y a pas de retour en arrière entre les phases et qu'il y a une défaillance dans l'une des phases, on sait que l'erreur est survenue dans cette phase. Par contre, s'il peut y avoir des retours en arrière entre les phases, on ne peut être sûr d'avoir détecté l'erreur lorsqu'une défaillance survient dans l'une des phases. Par un retour en arrière à un point choisi dans une phase précédente, on pourrait obtenir une nouvelle alternative pouvant passer par toutes les phases. Par conséquent, si l'on autorise les retours en arrière entre les phases, il est difficile de déterminer si une défaillance dans l'une des phases est due à une erreur d'entrée ou au fait qu'une phase précédente n'a pas encore engendré l'alternative correcte.

La méthode phasée a ses inconvénients. S'il n'y a pas de retour en arrière entre les phases, il faut s'assurer que chaque phase engendre une structure composite qui représente toutes les possibilités pour cette phase. De ce fait, l'approche intégrée peut parfois éliminer les possibilités plus rapidement que l'approche phasée. Dans l'approche intégrée, les contraintes de chaque phase peuvent être appliquées à mesure qu'elles deviennent pertinentes. Toutefois, dans l'approche phasée, les contraintes imposées par les phases ultérieures ne peuvent pas être appliquées tant que l'analyse n'a pas atteint ces phases.

Par exemple, dans une approche intégrée, l'information sémantique pourrait être utilisée pour éviter de construire des arbres d'analyse syntaxique pour les analyses syntaxiquement valides, mais sémantiquement anormales.

De plus, l'approche intégrée est plus plausible que l'approche phasée du point de vue psychologique. L'homme n'attend pas d'avoir entendu tous les mots d'une phrase avant de commencer à en analyser les premiers mots. On pourrait améliorer le temps de réponse du système si l'analyse des premiers mots d'une phrase commençait avant que la phrase entière n'ait été introduite dans le système.

La représentation des connaissances dans le système Tulip

Tulip utilise le langage Prolog comme langage de représentation des connaissances. En fait, à l'exception des fonctions de Skolem, la représentation des connaissances dans le système Tulip est limitée au sous-ensemble Datalog de Prolog (Datalog est la langage Prolog sans les symboles de fonction).

Le langage Prolog est parfois plus efficace qu'un langage logique de premier ordre (langage LPO) comme langage de représentation des connaissances. Prolog utilise l'hypothèse de l'unicité des noms. Par conséquent, en Prolog, les atomes différents désignent des objets différents, Dans un langage LPO, il faut introduire des formules explicites pour indiquer que des atomes différents désignent des objets différents, par exemple :

tom = jack

Prolog utilise également l'assomption du monde fermé. Un fait est considéré faux si sa véracité n'a pas été démontrée.
Le langage Prolog n'est pas aussi expressif qu'un langage LPO, mais il constitue un sous-ensemble d'un langage LPO qui peut être traité de façon efficace.

Fonctions de Skolem

Tulip utilise des fonctions et des constantes de Skolem pour éliminer les quantificateurs existentiels. Par exemple, l'une des interprétations logiques de la phrase :

every man loves some woman

est :

Pour chaque homme X, il existe une femme Y telle que X aime Y.

Les variables existentielles ne peuvent être représentées directement en langage Prolog. Par conséquent, une fonction de Skolem, par exemple *skol (45,X)*, est introduite à la place de la variable existentielle Y. Le premier argument de *skol* est un nombre utilisé pour identifier uniquement cette fonction particulière de Skolem. Par exemple, en langage Prolog la phrase ci-dessus peut être représentée par :

```
loves(X,skol(45,X)) :-man(X).
woman(skol(45,X)).
```

À noter : le fait que les fonctions de Skolem dépendent de X signifie que chaque homme aime une femme différente. La femme aimée par Tom est différente de celle aimée par Jack car skol(45,'Tom') ne peut être unifié avec skol(45,'Jack').

La phrase :

every man loves a woman

peut également être interprétée dans le sens suivant :

il existe une femme Y telle que, pour chaque homme X, X aime Y, ou en d'autres termes, chaque homme aime la même femme.

Dans ce cas, les variables existentielles seraient remplacées par une constante de Skolem et la représentation Prolog serait :

```
loves(X,skol(46)) :-man(X).
woman(skol(46)).
```

Ici la fonction de Skolem ne dépend pas de X et est en fait une constante.

Prédicats binaires

Dans le système Tulip, à l'exception des événements, la plupart des faits sont représentés conceptuellement sous forme de prédicats binaires, c'est-à-dire de prédicats à deux arguments (prédicats Prolog). On trouvera dans [Ferguson 88b] des exemples de prédicats binaires utilisés par Tulip. En général, le nom du prédicat est celui d'une relation et les deux arguments représentent les deux objets en relation.

Il est pratique d'exprimer les faits sous forme de relations binaires, car une relation n-aire peut toujours être représentée par n-l relations binaires. Dans une base de connaissances dynamique à l'intérieur de laquelle le nombre d'attributs associés à une classe d'objets peut augmenter en tout temps, une représentation binaire offre plus de souplesse qu'une relation n-aire. Pour définir un nouvel attribut à un objet dans une représentation binaire, il suffit d'ajouter un nouveau prédicat binaire. Dans une représentation n-aire, c'est-à-dire une représentation où chaque argument du n-uplet représente une valeur d'attribut particulière, il faut remplacer le n-uplet par un (n+1)-uplet lorsqu'on ajoute un nouvel attribut.

Comparée à une représentation n-aire, la représentation binaire a l'inconvénient de prendre généralement plus d'espace.

L'utilisation de prédicats binaires équivaut essentiellement à utiliser un réseau sémantique. Les deux arguments du prédicat correspondent à deux noeuds dans un réseau sémantique et la relation binaire correspond à un arc orienté entre ces deux noeuds, l'arc portant le nom de la relation.

Les prédicats binaires peuvent également être conceptualisés par des représentations de la forme objet-attribut-valeur où le nom du prédicat est l'attribut, le premier argument, l'objet, et le troisième argument, la valeur.

La représentation utilisée dans Tulip est un peu plus complexe que celle décrite ci-dessus. Une relation binaire y est représentée par un prédicat Prolog à trois arguments, le premier étant le nom de la relation binaire et les deux autres, les objets en relation. Le nom du prédicat précise le type de la relation exprimée. Par exemple, la relation binaire exprimée par la phrase :

Joe is the father of Tom

serait représentée par

prop0(father,Tom,Joe).

Ici *father* est la relation et *Tom* et *Joe* sont les objets en relation. Le nom du prédicat, *prop0* indique qu'il s'agit d'une relation de propriété, c'est-à-dire une représentation qui peut être exprimée sous la forme d'une relation possessive, par exemple, le père de Tom est Joe. Tulip doit connaître les types de relations pour répondre de façon appropriée aux interrogations. Par exemple, Tulip sait que toute relation de propriété de la forme :

prop0(Rel,X,Y)

peut être exprimée comme suit :

X's Rel is Y.

Grâce au codage des types de relations, Tulip peut répondre aux interrogations de méta-niveau.

Par exemple, l'interrogation :

What does Hank have?

devient :

answer([X]) :-prop(Rel,'Hank', X)

où Rel représente une relation variable. Ceci permet de trouver les valeurs de toutes les relations de propriété de Hank. Si les relations binaires étaient exprimées directement, cette interrogation prendrait la forme :

answer([X]) :-Rel('Hank', X)

qui serait interdite dans la plupart des versions de Prolog parce que les noms de prédicats variables ne sont pas autorisés.

Faits connus et inférés

Le système Tulip fait la distinction entre les faits connus explicitement et les faits inférés. Les faits explicites sont exprimés par des prédicats dont le nom porte l'extension *0*. Les faits inférés, c'est-à-dire les règles, sont représentés par des prédicats dont le nom ne porte PAS l'extension *0*. Pour relier ces deux prédicats, on utilise une clause de liaison, par exemple :

prop(Rel,X,Y) :-prop0(Rel,X,Y).

En plus de ces clauses de liaison, il peut également y avoir des règles Prolog établissant des relations par inférence. Par exemple, le fait que

Y is the wife of X if X is the husband of Y

peut être exprimé par la règle Prolog : prop(wife,X,Y) :

-prop(husband,Y,X).

Il est utile de pouvoir distinguer les relations directes (celles dont le nom de prédicat porte l'extension *0*) des relations inférées.

Par exemple, lorsque Tulip décrit un objet, il n'affiche que les relations directes de ce dernier.

Représentation des verbes

Dans le système Tulip, les verbes sont généralement représentés par un prédicat *event*.

Par exemple, la phrase :

Jack married Jill on Thursday

est traduite (en partie) par :

event0(marry,skol(53),'Jack','Jill'),adjuct0(on,skol(53),date(89,3,30)).

Le premier argument de *event0* est la forme citée du verbe, le deuxième, une constante de Skolem représentant l'événement en question, le troisième, le sujet du verbe, et le quatrième l'objet du verbe. La constante de Skolem de l'événement, skol(53) (le deuxième argument de event0), est utilisée par les prédicats qui donnent des informations supplémentaires relatives à l'événement, telles que l'instant et le lieu.

Dans le cas présent,

adjunct0(on,skol(53),date(89,3,30))

indique que l'événement est survenu jeudi.

RÉFÉRENCES

[Ferguson 88a]
FERGUSON, Ronald. Enhancements to TULIP from Jan. 1/88 to mar. 4/88, 1988.

[Ferguson 88b]
FERGUSON, Ronald. Translating Words to Their *Meanings* in Tulip, 1988.

[Ferguson 89]
FERGUSON, Ronald. Parsing Business Cards with an Extented Logic Grammar, 1989.

[Ferguson 89]
FERGUSON, Ronald. Guide du l'utilisateur du système Sun-Tulip, 1989.

[Pereira 83]
PEREIRA, Fernado. Logic for Natural Language Analysis, Technical Note 275, SIR International, 1983.

[Robb 88]
ROBB, Madeleine. Index of Work Completed by Madeleine Robb on Tulip Lexicons, Aoust 1988.

CACC / CCAC
95330

QUEEN QA 76.9 .N38 F47614 19
Ferguson, Ronald John, 1951-
-Le projet Tulip

QA
76.9
N38
F476f
1989
c.3

DATE DUE

Pour plus de détails, veuillez communiquer avec :

Le Centre canadien de recherche sur l'informatisation du travail
1575, boulevard Chomedey
Laval (Québec)
H7V 2X2
(514) 682-3400

For more information, please contact:

Canadian Workplace Automation Research Centre
1575 Chomedey Blvd.
Laval, Quebec
H7V 2X2
(514) 682-3400