

QUEEN
QA
76.9
.N38
F476
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

2. THE TULIP PROJECT

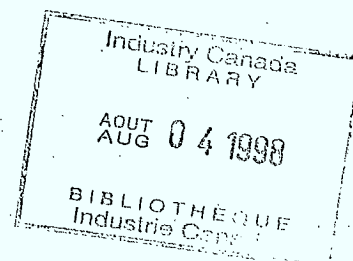
by

4 Ron Ferguson

QA
76.9
N38
F476
1989

Canada

QA
76.9
N38
F476e
1989
C-3



2. THE TULIP PROJECT

by

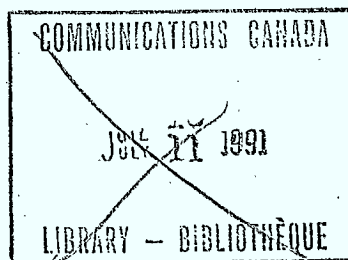
/ Ron Ferguson /

This report is the result of research done at Canadian Workplace
Automation Research Centre under the direction of
Joel Muzard, Experts Systems Group

Department of Communications

Laval

November 1989



N° 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
N38
F416e
1989
0.3

DD938253
DL 1064/209

Table of contents

<u>Introduction</u>	2
<u>Tulip Implementation</u>	2
<u>Building Tulip</u>	4
<u>The Tulip Source Files</u>	6
<u>Heuristics for Business Card Parsing</u>	12
<u>The Architecture of Tulip</u>	12
<u>Tulip's Knowledge Representation</u>	14
<u>Skolem Functions</u>	14
<u>Binary Predicates</u>	16
<u>Know and Inferred Facts</u>	18
<u>Representation of Verbs</u>	19
REFERENCES	21

Introduction

This report discusses the implementation of Tulip on Sun workstations. The capabilities and limitations of Tulip are described in the Sun Tulip User Manual [Ferguson 89b].

Tulip Implementation

The Sun version of Tulip is written in Quintus Prolog plus a few C routines. It runs under the UNIX operating system. The source code is maintained using UNIX SCCS (Source Code Control System). The source code for Tulip is contained underneath the directory '/home/condor/ferguson/tulip'. All (sub)directories mentioned in this report will be sub-directories mentioned in this report will be sub-directories of '/home/condor/ferguson/tulip'. The 'tulip' directory contains the following subdirectories :

- | | |
|------------------------------|--|
| main | - contains the main source code for parsing sentences. |
| business-card | - contains code for parsing business cards and names. |
| expert-system | - contains code for 'built-in' knowledge of dates and times, arithmetic operations, and answering questions. |
| extraposition-grammar | - code for translating extraposition grammars to Prolog clauses. |
| foreign-files | - code for interfacing with C routines. |
| help | - Tulip's help facility. |
| knowledge | - knowledge files which can be read by Tulip. |
| lexicon | - Tulip's dictionary. |

- 4 -

transform

- shell for writing Prolog preprocessors.

tulip

- executable images.

documentation

- documentation of Tulip.

Building Tulip

In subdirectory 'main', the files parse.pl and date-time-parse.pl contain extraposition grammars [Pereira 83] which must be translated into Prolog. The corresponding Prolog files are translation-of-parse.pl and translation-of-date-time-parse.pl, respectively. If one of the files containing an extraposition grammar is changed, it must be retranslated. To retranslate a file, load Tulip from 'main' and compile xg-fix.pl :

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

Then run 'gxxg' (defined in xg-fix.pl) :

```
gxxg.
```

gxxg will prompt for an input file and an output file. Enter parse.pl (or date-time-parse.pl) as the input file and translation-of-parse.pl (or translation-of-date-time-parse.pl) as the output file.

In subdirectory 'business-card', the file address.pl contains an extended Definite Clause Grammar for parsing business card [Ferguson 89]. This file must be preprocessed if it is changed. To run the preprocessor, load Tulip from 'main' and run 'gopt':

```
gopt.
```

gopt will prompt for an input file and an output file. Enter ../business-card/address.pl as the input file and ../business-card/tr-address.pl as the output file.

The file 'tulip-system.pl' in 'main' contains commands to compile the Tulip system. If one compiles 'tulip-system.pl', Tulip is recompiled and an image file is saved in the 'tulip' sub-directory. The 'main' sub-directory contains an executable file called 'tulip' which loads this image.

Tulip uses a large lexicon. The lexicon is expressed as Prolog facts in a series of files. When Tulip was first ported to the Sun, these lexicon files were loaded as part of Tulip. However, the response time Tulip on Sun

3/50 machines was found to be poor. It seemed likely that the poor response time was partly due to the memory paging that was required because of the size of the Tulip program. In an attempt to increase the speed of Tulip, the lexicon was stored in dbm files and only the lexicon entries for words which actually appear in an input sentence were read into memory. Off-loading the lexicon in this manner reduced the amount of virtual memory required by the Tulip program, and as a result Tulip now runs 3.4 times faster.

The database lexicon consists of records. Each record has a key field which is an English word and a content field which is an integer. The integer is a bit vector representation of a set. The set is a subset of `lex_set` which is defined in `lexical-dictionary-interface.pl`. The elements of this set are the names of Prolog lexicon predicates which take an English word as their only argument. For example, if `'noun(dog)'` is a fact in the Prolog lexicon, then `'noun'` will be a member of the set associated with the word `'dog'`.

The database lexicon is contained in the files `'lexicon.pag'` and `'lexicon.dir'` in the directory `'lexicon/dbm/'`. If these files have to be recreated, the following procedure can be followed :

- Delete the files `'lexicon.pag'` and `'lexicon.dir'`.
- Create empty `'lexicon.pag'` and `'lexicon.dir'` files.
- Go to the `'main'` sub-directory.
- Compile `'lexicon-system.pl'` (this is a compile file for the Prolog lexicon).
- Load but do not enter Tulip (remain in Quintus Prolog).
- Enter `'create_lexicon.'`

Note that it takes several hours to make the database lexicon from scratch. If one only wants to make minor changes to the database lexicon, it may be more convenient to modify the existing database lexicon using the predicates in `'dbm.pl'` and `'fixed-set.pl'` which are located in the `'lexicon/dbm'` and `'main'` directories, respectively. Of course, if one does modify the database lexicon directly, one should make corresponding changes to the Prolog lexicon files. See the file `'main/lexicon-system.pl'` for a list of the Prolog lexicon files.

A description of the lexicon files is given in [Robb 88].

Some of the lexicon files contain predicates with an arity greater than one. These predicates are not contained in the database lexicon. In order to reduce the paging load, the 'tulip-system.pl' file will include these files in Tulip only if the fact, 'lots_of_memory' occurs in the Prolog database. Otherwise, these files are not included when Tulip is compiled.

The Tulip Source Files

The following table gives the names of the non-lexicon source files in the Tulip system and a description of the contents of each file. The pathnames given for the files are with respect to the 'main' sub-directory. Each of these files has '.pl' as a file extension. The file extension is not shown in the table.

parse

Main English Grammar

date-time-parse

Date and Time Parsing

../business-card/address

Parser for Business Card

utilities

Generally Useful Predicates

collections

Operations on Sets

../help/help

Help Menu System

tulip-date

Tulip date-includes day of the week

accept-sentence

Read a sentence.

../transform/transform

Shell for transforming Prolog Clauses (Preprocessors)

tokenize

Tokenizer : string = list of tokens (reversible)

lexical-structure

Lexical Structure Data Type Definition

guess-lexical-type

Guess the lexical categories of words which are not in the
lexicon

../extraposition-grammar/xg-fix-aux

Basic Nonterminald for Extraposition Grammars/Record of parsed words

fp

Operations on arrays (lists of lists)

abbrev

Abbreviations

synonym

Synonymous Words

french-english

English Translation of French Words

new-lexicon

Basic Lexicon

attachment-control

Masks which control modifier attachment.

Ref. pg. 64, "Logic for Natural Language Analysis", F. Pereira

top

Top Level Prolog routines for Tulip.

execute command

Process Tulip's built-in commands

chat-lexicon-interface

Lookup words in the lexicon

morphology

Morphological Analysis

slot-filler

Replace words in the parse tree with their 'meanings'.

simplify

Simplify 'quant'trees by removing 'true'leaves

scope-determination

Determine the scope of English quantifiers.

template

Interface to templates which define the 'meaning'of words.

predicate-calculus

Convert Predicate Calculus formulae to clausal form.

modify-determiner

Change the determiners in the parse trees of assertions so that the quantification is more explicitly represented.

knowledge-file-access

Read from and write to knowledge files.

transform-answer

Add facts to or query the knowledgebase

parse-names

Parse a name, address or other non-grammatical entity.

anaphora

Handle anaphoric references

triggers

Trigger actions based on assertions being made to the knowledgebase.

parse-mods

Modification to grammar to handle conjunction of verb phrases

special-nps

Special parsers for identifying 'named'objects (eg. person and company names, addresses, phone numbers).

../business-card/gr-transform

Translate an extended DCG grammar (for business cards) to Prolog clauses.

../business-card/address-util

Token analysis for business cards

../business-card heuristics

Heuristics to label those parts of a business card which the business card grammar was unable to parse.

../business-card/semantics

Process the specification of relationships between business card objects

infer

Basic inferences

loop-checking

Check rules for recursive loops.

database-access

Removable clause assertion

significant-words

Assertion retrieval via word matching

date-time-facts

Facts about dates and times

mrl_temp

Interface to Meaning Representation templates for verbs

type-hierarchy

Specification of built-in type hierarchy for template matching.

../expert-system/date-time-expert

This is the Date-Time Expert System Engine i.e. calculator

../expert-system/talkr
Query evaluation

../expert-system/aggreg
Execute arithmetic operations

quintus-kludges
Predicates which can not (or have not) been properly defined in
the Quintus version of Tulip.

The preceding table was created by a Prolog program in
'documentation/file-comments.pl' which gets the header comment from
each file. The 'main/tulip-system.pl' file contains (commented out)
commands to execute the header comment program. The file 'main/header-
comment.pl' contains a template for header comments.

Heuristics for Business Card Parsing

As described in [Ferguson 89], heuristics were used to classify those parts of a business card grammar. However, Tulip's business card grammar may be applied to the parsing of information for which the business card heuristics are not appropriate (ie. organizations which are engaged in AI). Therefore, Tulip has been modified so that the heuristics are only applied if the fact 'apply_heuristics' is defined. The 'apply_heuristics' flag is checked in 'business-card/address.pl'.

The Architecture of Tulip

The architecture of Tulip is based on that of Chat-80 [Pereira 83]. Differences between Tulip and Chat-80 are discussed in [Ferguson 88a].

Tulip uses a phased architecture to process sentences. The entire sentence is analysed by one phase before it is passed to the next phase. The phases involved in the parsing of an English sentence are :

Process

read input
tokenization
syntactic lexicon lookup
finding significant words
syntactic parsing
semantic lexicon lookup
quantifier scope determination
quant tree simplification
identify named objects
find anaphoric referents

Predicate

read_lines/6
tokenize_parse/3
lexical_lookup/2
significant_words/4
sentence/6
i_sentence/2
clausify/2
simplify/2
transform_special_nps/2
anaphoric_references/3


```
IF query
    query evaluation          answer/2
    answer generation        surface_descriptions/2
    answer display           respond/1
ELSE % assertion or rule
    convert to Horn clauses  translate_to_clausal_form/4
    update knowledgebase     update_knowledgebase/2
END
```

A phased approach is advantageous from a software engineering point of view because it divides the overall task into several smaller, largely independent, tasks. Each of these smaller tasks is easier to comprehend and debug than it would be if the tasks were merged into one large task.

If there is little backtracking between the different phases, one can do more accurate error detection and correction than would be possible in an integrated approach. If there is no backtracking between phases and a particular phase fails, one knows that the error occurred in that phase. On the other hand, if backtracking is allowed between the phases, then failure of a particular phase does not necessarily mean that an error has been found. By backtracking to a choice point in a previous phase, one may obtain a new alternative which is able to pass all the phases. Thus if backtracking is allowed between phases, it is difficult to determine whether failure of a particular phase is due to an input error, or due to a preceding phase not having yet generated the correct alternative.

These are disadvantages to using a phased approach. If one does not allow backtracking between phases, one must ensure that each phase generates a composite structure which represents all possible alternatives of that phase. Related to this is the fact that an integrated approach may be able to eliminate possibilities faster than a phased approach. In an integrated approach, the constraints of each phase can be applied as soon as they become relevant. However, in a phased approach, the constraints imposed by later phases can not be applied until the analysis has reached those phases.

For example, in an integrated approach, semantic information could be used to avoid building syntactic parse trees for parses which are syntactically valid, but semantically anomalous.

An integrated approach is also more psychologically plausible than a phased approach. It does not seem likely that humans wait until they have heard all the words in a sentence, before they begin to analyse the first few words. Also, response time could be improved if analysis of the first words in a sentence began before the entire sentence had been entered.

Tulip's Knowledge Representation

Tulip uses Prolog as its knowledge representation (KR) language. In fact, with the exception of skolem functions, Tulip's knowledge representation is limited to the Datalog subset of Prolog (Datalog is Prolog without function symbols).

Using Prolog as the knowledge representation has efficiency advantages over using full first order logic (FOL). Prolog uses the Unique Names Assumption. This means that in Prolog different atoms are assumed to denote different objects. In FOL one must introduce explicit formulae to specify that different atoms refer to different objects, eg :

tom = jack

Prolog also uses a Closed World Assumption. If a fact is not known to be true, it is assumed to be false.

Prolog is not as expressive as FIL but it implements a subset of FOL which can be efficiently processed.

Skolem Functions

Tulip introduces skolem functions and skolem constants in order to eliminate existential quantifiers. For example, one logical interpretation of the sentence :

every man loves some woman.

is :

for every man, X, there exists a woman, Y, such that X loves Y.

Existential variables can not be represented directly in Prolog. Therefore, a skolem function, eg. skol(45,X), is introduced to take the place of the existential variable Y. The first argument of 'skol' is a number which is used to uniquely identify this particular skolem function. Thus the above statement can be represented in Prolog by:

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

Note that the fact that the skolem function depends on X means that each man loves a different woman. The woman that Tom loves is different from the woman Jack loves since skol(45,'Tom') can not be unified with skol(45,'Jack').

Another possible interpretation of:

every man loves a woman.

is :

there exists a woman Y, such that for every man X, X loves Y. ie.
every man loves the same woman.

In this case the existential variable would be replaced by a skolem constant, and the Prolog representation would be .:

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

Note that in this case the skolem 'function' does not depend on X and is, in fact, a constant.

Binary Predicates

Most facts in Tulip, with the exception of events, are conceptually represented as binary predicates - that is Prolog predicates which take two arguments. [Ferguson 88b] has examples of the binary predicates used by Tulip. In general, the predicate name is the name of a relationship, and the two arguments represent the two objects that have the relationship. It is convenient to express facts in terms of binary relationships because a more complex n-ary relationship can always be expressed as n-1 binary relationships. In a dynamic knowledge base in which the number of attributes associated with a class of object can increase at any time, a binary representation offers more flexibility than an n-ary relationship. In a binary representation, to define a new attribute for an object one just adds a new binary predicate. On the other hand, if an object is represented by an n-tuple, where each argument of the n-tuple represents a particular attribute value, one must replace the n-tuple with an (n+1)-tuple when one adds a new attribute.

A disadvantage of a binary representation with respect to an n-tuple representation is that the binary representation will generally take up more space.

Using binary predicates is essentially equivalent to a semantic net representation. The two arguments of the predicate correspond to two nodes in a semantic net and the binary relationship corresponds to a directed arc between the two nodes. The arc is labelled with the name of the relationship.

Binary predicates can also be conceptualized as an Object-Attribute-Value representation, where the predicate name is the Attribute, the first argument is the Object, and the third argument is the Value.

The actual representation in Tulip is a bit more complex than described above. A binary relationship is represented by a Prolog predicate which takes three arguments - the first argument is the name of the binary relationship and the remaining two arguments are the objects which have that relationship. The predicate name specifies the type of relationship which is being expressed.

For example, the binary relationship expressed in the sentence :

Joe is the father of Tom

would be represented by :

prop0(father,Tom,Joe)

Here, 'father' is the relationship and 'Tom' and 'Joe' are the objects involved in the relationship. The predicate name 'prop0' indicates that this is a 'property' relationship. A property relationship is one which can be expressed as a possessive relationship, eg. Tom's father is Joe. Tulip needs to know the relationship types so that it can generate appropriate responses to queries. For examples, Tulip knows that any property relationship of the form :

prop0(Rel,X,Y)

can be expressed as : X's Rel is Y.

Encoding relationship type information also allows Tulip to answer 'metalevel' queries. For example, the query :

What does Hank have?

is translated into :

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

where Rel represents a variable relationship. This finds the values of all property relationship which Hank has. If the binary relationships were expressed directly, this query would have the form :

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

which would be illegal in most versions of Prolog because variable predicate names are not allowed.

Know and Inferred Facts

Tulip distinguishes between explicitly know and inferred facts. Explicit facts are expressed by predicates which have a '0' appended to their names. Inferred facts, ie. rules, are represented by predicates which do NOT have a '0' appended to their names. To link up these two predicates a linking clause is used, eg. :

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

Besides these linking clauses, there may also be Prolog rules which establish relationships using inferencing. For example, when generating a description of an object, Tulip only displays the object's direct relationships.

Representation of Verbs

In Tulip, verbs are generally represented by an 'event'predicate. For example, the sentence :

Jack married Jill on Thursday.

is translated (in part) into :

event0(marry,skol(53),'Jack','Jill'),adjunct0(on,skol(53),date(89,3,30))

The first argument of event0 is the citation form of the verb, the second argument is a skolem constant which represents this particular event, the third argument is the subject of the verb, and the fourth argument is the object of the verb. The event skolem constant, skol(53), (the second argument of event0) is used by predicates which express extra information about the event, such as when and where the event occurred.

In this case :

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

indicates that the event occurred on Thursday.

REFERENCES

- [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 Extended Logic Grammar, 1989.
- [Ferguson 89b] Ferguson, Ronald, Sun TULIP Users Manual, 1989.
- [Pereira 83] Pereira, Fernando, Logic for Natural Language Analysis, Technical Note 275, SRI International, 1983.
- [Robb 88] Robb, Madeleine, Index of Work Completed by Madeleine Robb on Tulip Lexicons, Aug. 1988.

95324

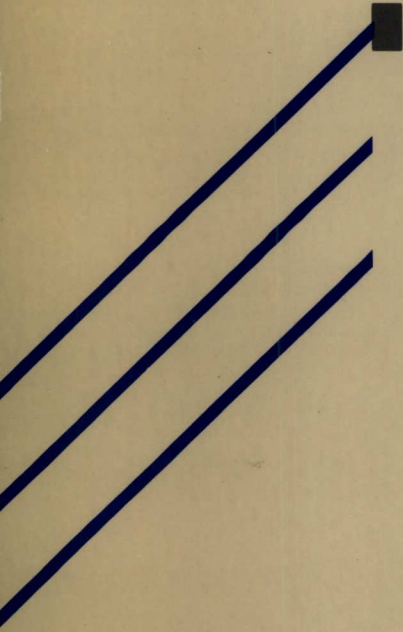
FERGUSON, RON

QUEEN QA 76.9 .N38 F476 1989
Ferguson, Ronald John, 1951-
The Tulip project

QA
76.9
N38
F476e
1989
c.3

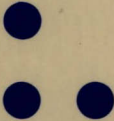
DATE DUE

[illegible]



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