TSNLP

Test Suites for Natural Language Processing

User Manual

Volume 2

Core Test Suite Technology

Stephan Oepen, Frederik Fouvry
Klaus Netter, Tom Fettig, Fred Oberhauser

Deutsches Forschungszentrum für Künstliche Intelligenz GmbH

Preface

The TSNLP User Manual summarizes the results of the multi-national project TSNLP (*Test Suites for Natural Language Processing*) funded by the European Commission (DG13) as part of the Language Engineering programme (as research grant LRE-62-089). TSNLP had a duration of 29 months and started in December 1993.

The project produced a methodology and tools for test suite construction in addition to substantial test suites in three languages — English, French, and German (over 4500 distinct items in each language). The test material covers central syntactic phenomena in the three languages and aims to be relevant to any developer or user of systems with grammar components who wish to test, benchmark, or evaluate those systems. To allow for flexible access to the test suites they are mounted on a database.

The project consortium consisted of the following centres and principal researchers:[1]

- University of Essex, UK (coordinator):
  Doug Arnold, Lorna Balkan, Frederik Fouvry
- Istituto Dalle Molle per gli Studii Semantici e Cognitivi (ISSCO), Switzerland:
  Dominique Estival, Kirsten Falkedal, Sabine Lehmann, Hervé Compagnion
- Aerospatiale, France:
  Eva Dauphin, Veronika Lux, Sylvie Regnier-Prost
- Deutsches Forschungszentrum für Künstliche Intelligenz GmbH, Germany:
  Klaus Netter, Stephan Oepen, Hans Uszkoreit

Other contributors included Evangelia Kordoni, Siety Meijer, and Martin Rondell (University of Essex) and Judith Baur, Tom Fettig, Judith Klein, and Fred Oberhauser (DFKI).

The User Manual is in three volumes:

- Volume 1: Background, Methodology, Customisation and Testing
  Lorna Balkan, Frederik Fouvry, Sylvie Regnier-Prost (editors);
- Volume 2: Core Test Suite Technology
  Stephan Oepen, Frederik Fouvry, Klaus Netter, Tom Fettig, Fred Oberhauser;
- Volume 2b: Test Suite Technology
  Frederik Fouvry (editor);

[1] Full postal addresses and phone numbers for contact persons of the consortium members are:

CL|MT Group	ISSCO	Aerospatiale France	DFKI GmbH
University of Essex	Université de Genève	Common Research Center	CL Department
Wivenhoe Park	54, route des Acacias	12, rue Pasteur BP 76	Stuhlsatzenhausweg 3
UK Colchester CO4 3SQ	CH 1227 Genève	F 92152 Suresnes Cedex	D 66123 Saarbrücken
+44 - 1206 - 87 20 86	+41 - 22 - 705 7870	+33 - 1 - 4697 3061	+49 - 681 - 302 52 82

- Volume 3: Test Data Documentation
  Sabine Lehmann, Dominique Estival, Kirsten Falkedal, Hervé Compagnion, Lorna Balkan, Frederik Fouvry, Judith Baur, Judith Klein.

Volume 1 contains a background chapter in which some of the factors which have influenced the design of the project are sketched. The methodology chapter gives a step by step account of how one can go about writing core data, that is, data that cover central phenomena of a language and that are intended to be applicable to a wide range of applications. The customisation chapter describes how the core data can be customized to a particular application (and sketches how it could be customized to a particular domain or text type). The chapter on testing gives an example of how the test suite can be applied to a real life evaluation scenario.

Volume 2 contains a description of the annotation scheme on which the data was constructed, the construction tool tsct used to create the data, the database tsdb on which the data is mounted, and the automated import and consistency checking procedure from tsct to tsdb.

Volume 2b contains a description of and documentation on the automatic test suite generation tool (AutoTSG) and the lexical replacement tool.

Volume 3 contains the detailed documentation that accompanies the data, and which is intended to make the data more accessible to users. It also contains the category and function labels used in the English, French, and German test data with examples for each language, in addition to the vocabulary list used for the test data by the three languages.

TSNLP reports and this User Manual are available from the coordinator at the following address:

Lorna Balkan
CL|MT Group
Department of Language and Linguistics
University of Essex
Wivenhoe Park
UK Colchester, CO4 3SQ
Fax: +44 1206 872085
email: `balka@essex.ac.uk`

Additionally, all TSNLP deliverables, data, and software can be accessed through the internet from the following address:

`http://tsnlp.dfki.uni-sb.de/tsnlp/`

or via the ftp site:

`anonymous@tsnlp.dfki.uni-sb.de/tsnlp/`

Contents

1 Overview **1**

2 TSNLP **Annotation Schema** **3**

3 The Test Suite Construction Tool (tsct) **7**
3.1 Motivation 7
3.2 Implementation 8
3.3 Navigation in tsct 9
3.4 Test Item Window 10
3.5 Phenomenon Window 11
3.6 Test Set Window 12
3.7 User & Application Profile Window 13
3.8 A Typical Test Data Construction Scenario 13
3.9 The tsct Data File Format 14

4 The Test Suite Database (tsdb) **17**
4.1 Motivation 17
4.2 The TSNLP Database Schema 17
4.3 A Few Open Questions 20
4.4 Query and Retrieval: Some Examples 21
4.5 Implementing tsdb_1: Building It Ourselves 22
4.6 tsdb_1: Operation and Usage 23

5 Import and Consistency Checking (import) **29**
5.1 import: Operation and Usage 29

6 Test Suite Utilities (tsu) **33**

A Contents of the Current tsct **Distribution** **i**

B Contents of the Current tsdb **Distribution** **iii**

C tsdb_1 **Functional Interface:** `tsdb.h` **v**

D The Import Procedure **xiii**

Chapter 1

Overview

Because the test data construction proper as well as the customization and application of a multi-purpose test suite to a specific NLP system or domain are laborious, cost-intensive and error-prone tasks, TSNLP put strong emphasis on supplying suitable special-purpose technology to facilitate both the development as well as the usage of the TSNLP test data. The TSNLP *core technology* designed to support both developers of additional or new test data and users who plan to apply the TSNLP data to a token system or domain. It comprises the following software packages:

- the test suite construction tool tsct, an intelligent form-based editor for TSNLP test data (see section 3);
- the test suite database tsdb, a simplified and fine-tuned relational database management system (see section 4);
- the import and consistency checking procedure import that converts data from the tsct to tsdb format and ensures basic wellformedness and consistency; and
- the test suite utilities tsu, a set of utility scripts that have evolved throughout the project and provides some useful functionality (e.g. computing a list of vocabulary used sorted by frequency).

Since both the test suite construction tool and the test suite database crucially build on the TSNLP annotation schema (the formal specification of properties and values used in classifying and organizing TSNLP test data), the abstract annotation schema is reviewed in section 2 and then related to its implementations in tsct and tsdb.

In general, all technology building put strong emphasis on the following key desiderata, in order to allow for the wide dissemination and acceptance of TSNLP results as a pre-standard reference:

- **usability**: to allow for the application of the methodology, technology and test data developed in TSNLP to a wide variety of diagnosis and evaluation purposes of very different applications by developers or users with varied backgrounds (e.g. academic vs. industrial);
- **suitability**: to meet the specific necessities of storing and maintaining natural language test data (e.g. in string processing) and to provide maximally flexible interfaces;

- **extensibility**: to enable and encourage users of the TSNLP database to add test data and annotations according to their needs without changes to the underlying database; and
- **portability and simplicity**: to make the results of TSNLP available on several different hard- and software platforms free of royalties and easy to use.

In both test data construction and database design issues TSNLP could build on experiences gained in the DiTo project ([Nerbonne *et al.* 1993]) previously carried out at DFKI. The DiTo effort produced some 1500 systematically constructed and annotated sentences for four syntactic phenomena of German; the test data were organized into a simple relational database that was implemented by means of standard Un∗x text processing tools (viz. `sort(1)`, `uniq(1)` and `join(1)`) combined with a query engine (an `awk(1)` script) and a query processor (realized in `yacc(1)` and `lex(1)`).

The annotation schema and parts of the data developed in DiTo (for German at least) served as a starting point for work done in TSNLP. At the same time, however, the DiTo project revealed two important problems that prompted the TSNLP consortium to pursue more elaborate approaches:

(i) Data construction in DiTo was directly carried out in the internal format defined by the database: the test data authors had to know and understand some of the database internals and use a standard text editor to produce the appropriate format; the method, obviously, turned out to be inefficient, laborious and error prone;

(ii) although `awk(1)` as the query engine provided for very high-level string manipulation facilities, with growing database sizes processing efficiency became a problem since `awk(1)` is an interpreted language; additionally, even though the Un∗x text processing tools deployed in DiTo are (at least) available for the Microsoft DOS universe as well, the hybrid combination of scripts with `awk(1)` and `yacc(1)` programs imposed a severe limitation on portability and simplicity.

In the TSNLP approach to test suite technology both limitations have been addressed: (i) test data authors are now supported by an appropriate editor tool and (ii) the database implementation is entirely in ANSI C, thus supporting a wide variety of platforms. Moreover, in order to facilitate a wide distribution, the software design primarily puts emphasis on engineering rather than on scientific aspects such that simplicity, robustness and documentation of the software play a very important role.

Chapter 2

TSNLP Annotation Schema

Based on a survey of existing test suites ([(principal author) *et al.* 1994]) and the types of annotations (if any) found, TSNLP has designed a detailed annotation schema which neither presupposes some token linguistic theory, nor a particular evaluation type nor a limited range of suitable applications. The TSNLP annotation schema was first developed in [Estival *et al.* 1994] and has since — throughout the project lifetime, the implementation of the construction tool and test suite database, and the parallel construction of substantial test suite fragments (documented in volume 3 of this user manual: [Lehmann *et al.* 1996]) — been elaborated, further specified, and slightly revised.

This section reviews the specification of the annotation schema on an abstract level and includes examples for the central parts. The more technical and detailed presentation of the individual properties (attributes), admissible values and intended interpretations is given in section 3 in conjunction with the discussion of the test suite construction tool.

Test data and annotations in TSNLP test suites, basically, are organized at four distinct representational levels:

- **Test data core:** The core of the data collection are the individual *test items* (sentences, phrases et al.) together with all general, categorial, and structural information that is independent of phenomenon, application, and user parameters. Besides the actual string of words, annotations at this level include (i) bookkeeping records (date, author, origin, and item id), (ii) the item format, length, category, and wellformedness code, (iii) the (morpho-)syntactic categories and string positions of lexical and — where uncontroversial — phrasal constituents, and (iv) an abstract dependency structure. Encoding a dependency or functor-argument graph rather than a phrase structure tree avoids imposing a specific constituent structure but still can be mapped onto one.

- **Phenomenon related data:** Based on a hierarchical classification of *phenomena* (e.g. verb valency being a subtype to general complementation) each phenomenon description is identified by its phenomenon identifier, supertype(s), interaction with other phenomena, and presupposition of such. Moreover, the set of (syntactic) parameters that is relevant in the analysis of a token phenomenon (e.g. the number and type of complements in the case of verb valency) is determined. Individual test items can be assigned to one or several phenomena and annotated according to the respective parameters.

- **Test sets:** As an optional descriptive level in between the test item and phenomenon levels, pairs or sets of grammatical and ill-formed items can be grouped into *test*

Test Item		
item id: *24033*	author: *dfki-klein*	date: *jan-94*
register: *formal*	format: *none*	origin: *invented*
difficulty: *1*	wellformedness: *1*	category: *S*
input: *Der Manager sieht den Präsidenten*		length: *5*
comment:		

position	instance	category	function	domain
0:2	*Der Manager*	*NP_nom-sg*	*subj*	*2:3*
2:3	*sieht*	*V*	*func*	*0:5*
3:5	*den Präsidenten*	*NP_acc-sg*	*obj*	*2:3*

Phenomenon		
phenomenon id: *24*	author: *dfki-klein*	date: *jan-94*
name: *C_Complementation*		
supertypes: *Complementation*		
presupposition: *C_Agreement, NP_Agreement*		
restrictions: *neutral*	interactions: *none*	purpose: *test*
comment:		

Figure 2.1: Sample instance of the TSNLP annotation schema for one test item (viz. the German sentence *Der Manager sieht den Präsidenten* ('the manager sees the president.')): annotations are given in tabular form for the *test item*, *analysis*, and *phenomenon* levels.

sets. Thus, it can be encoded how, for example, test items that originate from the systematic variation of a single phenomenon parameter relate to each other.

- **User and application parameters:** Information that will typically correlate with different evaluation and application types in the use of a test suite is factored from the remainder of the data into *user & application profiles*. Currently, there are foreseen (i) a centrality measure on a per phenomenon, test set, or even test item basis (e.g. user x may in general consider phenomenon y to be central but test set z to be marginal within y) and (ii) judgements of relevance with respect to a specific application. To match the results obtained from a token NLP system (a parser, say) against a test suite, a formal or informal output specification can be added at the profile level (figure 4.3 gives an example).

In addition to the parts of the annotation schema that follow a more or less formal specification, there is room for textual comments at the various levels in the above hierarchy to accommodate information that cannot or need not be formalized.

Figure 2.1 gives an example of a test item and a phenomenon description. The following is an example for the use of phenomenon-specific parameters for the C_Complementation (verbal government) phenomenon: two parameters — viz. (i) the total number of (obligatory) complements; and (ii) a summary of the actual valency frame[1] — encode the arity

[1]Obviously, both of the C_Complementation parameters are redundant with respect to the information that is already encoded in the phenomenon-independent *analysis* part of the *test item* annotations. However, making the valency information explicit as two separate per phenomenon parameters allows explaining the source of ungrammaticality in ill-formed test items quite straightforwardly. Additionally, the parameters (at least intuitively) correspond to the procedures that can be applied to grammatical test

and selectional restrictions of the matrix verb. For the sentence *Der Manager sieht den Präsidenten.* ('the manager sees the president.'), for example, the parameter values are:

number of complements: *2*

valency frame: ⟨ *subj (NP_nom)* , *obj (NP_acc)* ⟩

The corrsponding ill-formed test items derived from this example

**Der Manager sieht.*

**Der Manager sieht [dem Präsidenten]*$_{dat}$.

are classified according to the respective parameter violated and linked to the positive example into a *test set*.

items in order to derive ill-formed examples: in the case of verbal government ungrammatical items either result from the deletion of complements or from a violation of the governed properties.

Chapter 3

The Test Suite Construction Tool (tsct)

3.1 Motivation

Very generally, the motivation for the development of a Test Suite Construction Tool (tsct) particular to the TSNLP project has already been sketched: the basic idea in building tsct was to facilitate and guide the production of systematically and consistently annotated test data in three languages by several test data authors at four geographically distant sites. More specifically, the objectives of tsct are:

- to provide a mask-based graphical editor (i.e. an input form fitted to the annotation schema) for TSNLP test data;
- to promote the systematic production of test data by tailoring tsct to the TSNLP annotation schema developed in work package 2.2 (see [Estival *et al.* 1994]);[1]
- to partly automate the writing of test data through the reuse and adaptation of previously constructed data;
- to fit the tool to the needs of the test data author by providing specialized editing functionality (i.e. in string manipulation and the on-line computation of dependent field values);
- to control the test data construction through the form-based input mask to reduce erratic variation and ensure a basic degree of formal consistency; and
- to allow for convenient browsing and navigation in TSNLP test data files, thus facilitating the exchange and comparison of data among different test data authors and sites.

Besides, tsct provided the consortium with a uniform and well-defined format of storing the TSNLP test data (see section 3.9) even before the scheduled start of the database design work package. The machine-readable tsct data format greatly simplifies the automated import procedure into the TSNLP database, such that the labour that was originally allocated for this task could be transferred into data construction and consistency checking.

[1] Although several of the attribute names and part of the grouping have changed inbetween the work package 2.2 report, the tsct and, in turn, the tsdb implementations respectively, it should become obvious how the individual parts relate to each other by simple comparison of the three representational stages.

3.2 Implementation

The `tsct` implementation is based on the ANSI C and X11 industrial standards; hence, the tool is efficient, highly portable and can be made free software. To minimize the development time and to provide for easy adaptation and extension, `tsct` was built on top of the Widget Creation Library (Wcl) available from the public domain (e.g. through the Usenet group `comp.sources.x` and most major `ftp(1)` servers). Following is a quotation from the Wcl 2.7 distribution:

> Wcl is the Widget Creation Library. It allows all Look and Feel aspects for an application to be specified in Xrm resource files, leaving your application code to consist of a simple main() and callback procedures. Wcl greatly reduces the learning curve for developing applications with Widget based user interfaces, yet does not restrict the developers to platforms, widget sets, nor development languages.
>
> Wcl is widget set independent, but Xt Intrinsics dependent. Wcl can be used on Xt Intrinsics from X11R3, Motif 1.0 (X11R3.5 as with SCO ODT), X11R4, X11R5, and X11R6. Some capabilities are not available on Xt Intrinsic versions before X11R5.
>
> Wcl is portable: the distribution has been built and tested on platforms with SVR3, SVR4, SunOS, and other varieties of UNIX. Some embriotic support for VMS is also provided. Systems with dynamic linking (SunOS and SVR4) which have dlopen() and dlsym() can also invoke dynamically bound callbacks and actions from shared libraries.
>
> Support for Athena, Cornell, Motif, and OpenLook widgets are provided with campanion libraries (widget set registration functions) and resource interpreters with many example resource files.

— Accordingly, `tsct` comes as a single executable file (the resource interpreter) together with four separate resource definition files corresponding to the organization of the TSNLP annotation schema into four basic windows (see sections 3.4 to 3.7). The command line synopsis is:

`tsct -rf` *resource file*
   `-file` *data file*
   [`-intelligent`]
   [`-offset` *n*]
   [*generic X11 options*]

where

- *resource file* is one of the Wcl resource description files corresponding to the four `tsct` windows (i.e. one of `item.ari`, `phenomenon.ari`, `set.ari` or `profile.ari`; see appendix A);
- *data file* is a data file in `tsct` format (see section 3.9); if the file already exists it is opened in appending mode and the original version is preserved as a backup copy (file name with '~' suffix);
- `-intelligent` is only applicable to the *Test Item* window and enables the automatic computation of the *Length* and *Instance* (in the embedded *Analysis* table) fields (see section 3.4);
- n is an integer setting the initial browsing position to the n^{th} record in *data file*; and
- *generic X11 options* is a set of standard X11 options such as `-title`, `-geometry` or `-font` (see `X(1)` for details).

The current version of tsct (viz. 1.4) is stable and has been heavily used by all consortium members since April 1995; although it is not envisaged that further development of tsct will become necessary for TSNLP, bug reports and suggestions for improvement are welcome by electronic mail to `tsct@cl.dfki.uni-sb.de`. Currently, tsct binaries are available for Sun and HP workstations as well as for Intel x86 based machines running the Linux operating system. Versions for most any Un*x flavour can be made available on demand.

3.3 Navigation in tsct

Each of the four tsct windows (see sections 3.4 to 3.7) displays the contents of one data file one record at a time. To navigate through the list of data records (e.g. individual test items in the case of the *Test Item* window) there are Previous and Next buttons moving the current browsing position through the data file backwards and forwards respectively. Clicking the left, middle or right mouse buttons on Previous or Next moves by one, five or ten data records respectively. To indicate the current browsing position in the list of records, the upper right corner of each of the windows displays the active record and the total number of records (in the format 'Data Set # m (of n)' where m is current and n total). Since the *Test Item* and *Phenomenon* windows have embedded *Analysis* and *Parameters* parts that in turn are list (or table) valued, for these there are additional movement buttons labelled prev and next.

Navigation through the individual input fields is by mouse or cursor or tabulator key movement; within any of the text input fields line editing facilities are similar to the GNU Emacs key bindings.

In addition to the global navigation means each of the tsct windows has the following buttons:

- Reset — to distinguish between fields that typically remain unchanged (or static) throughout one tsct session (e.g. the *author* and *date* information) and those that vary for each individual data record each of the four windows has a solid horizontal line delimiting the former set of fields (the upper part of the window) from the latter (the lower and typically larger part of the window); the Reset button deletes all field values of the current record except for those from above the delimiter line (i.e. all non-static fields are reset);

- Delete removes the current data record from the list of records;

- New creates a new data record as a copy of the current record (by duplicating all its information) and appends it to the list of records;

- New + Reset is similar in effect to the sequence of New and Reset: it creates a new but empty (except for the static fields) data record;

- Save writes the current list of records to disk (i.e. into the *data file* specified at startup);

- Quit, finally, allows to exit from tsct without saving the current state; if, however, the active list of data records and the *data file* differ, a popup dialogue will request confirmation of the quitting first.

3.4 Test Item Window

Figure 3.1 shows the layout of the *Test Item* window of `tsct` that accommodates the *core data* of any test suite, viz. the individual test items (sentences, phrases *et al.*) together with all general, categorial, and structural information that is independent of phenomenon, application, and user parameters. Following is a summary of the individual fieds, value restrictions and — where appropriate — interpretations (see [Estival *et al.* 1994] and [Estival *et al.* 1995] for further details).

author *string*: creator of this record;

date *date* in tsdb_1 format (allowing underspecification): date of creation;

origin *string*: origin of the test item (for TSNLP core test data typically '`invented`');

register *string*: stylistic register (for TSNLP core test data typically '`format`');

format *string*: test item format (for TSNLP core test data typically '`none`');

difficulty {`0` | `1`}: test item difficulty (`0`: standard; `1`: trap);

item id *integer*: unique identifier;

input *string*: actual surface string for test item

wf `[0 .. 3]`: grammaticality code (`0`: illformed; `1`: wellformed; `2`: argueable or dubious; `3`: ungrammatical with respect to parameter settings — see [Estival *et al.* 1995] for details)

length *integer*: length of *input* field not counting punctuation marks (automatically computed in `tsct`);

comment *string*: free text

position *integer*:*integer*: zero-based substring position (discontinuous substrings can be denoted by a comma-separated lists of integer pairs, e.g. '`0:2,3:5`');

instance *string*: substring in *input* corresponding to *position* (automatically computed in `tsct`);

category *string*: category label

function *string*: function label

function *integer*:*integer*: functional domain (for functors) or position of governing functor (for non-functors);

tag *string*: (optional) tag label;

comment *string*: free text.

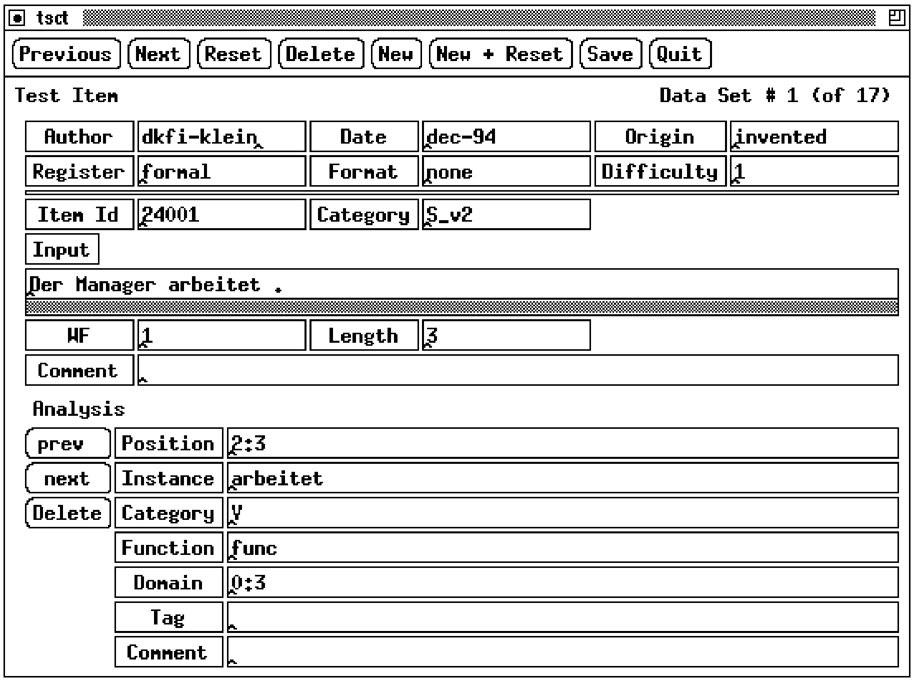


Figure 3.1: The *Test Item* window of tsct.

3.5 Phenomenon Window

The tsct *Phenomenon Window* (see figure 3.2) serves a dual purpose: firstly, it is the home for the descriptions of individual phenomena, and, secondly, it allows one to associate test items (and their parameters) with phenomena; in this sense, it holds the vast majority of *phenomenon related data.* Following is a summary of the individual fieds, value restrictions and — where appropriate — interpretations (see [Estival *et al.* 1994] and [Estival *et al.* 1995] for further details).

author *string*: creator of this record;

date *date* in tsdb_1 format (allowing underspecification): date of creation;

name *string*: phenomenon name;

supertypes *string*: comma-separated list of supertypes

phenomenon id *integer*: unique identifier

presupposition *string*: comma-separated list of presupposed phenomena;

interaction *string*: comma-separated list of interacting phenomena;

purpose *string*: purpose of phenomenon (for TSNLP core test data typically '`test`')

restrictions *string*: restrictions on phenomenon (for TSNLP core test data typically '`neutral`')

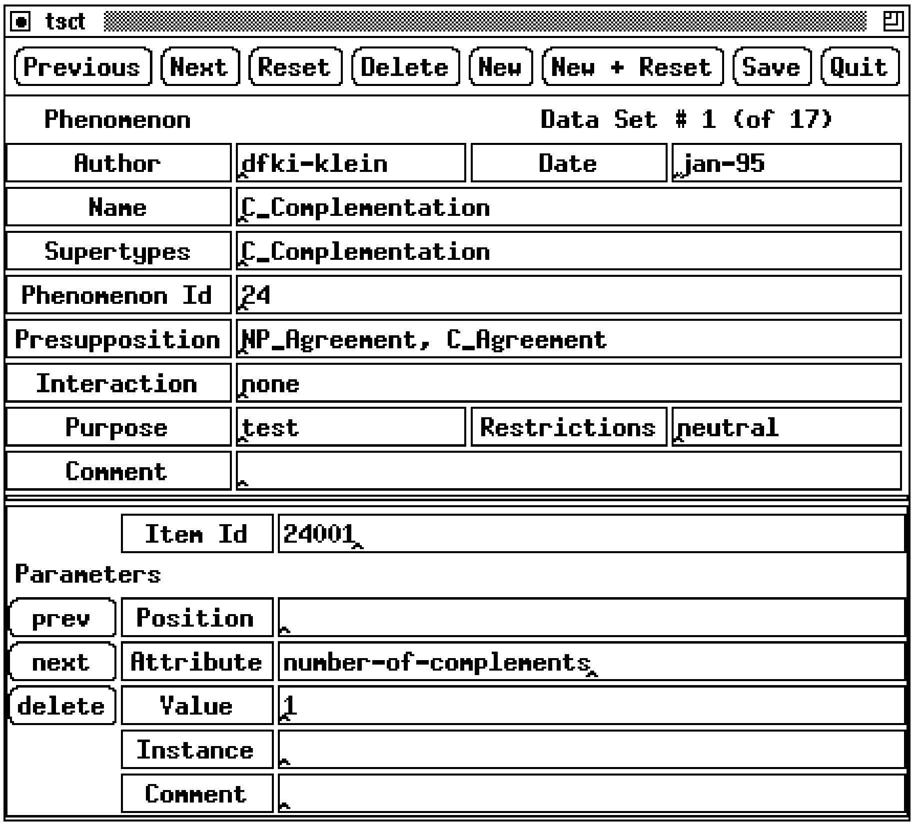


Figure 3.2: The *Phenomenon* window of tsct.

comment *string*: free text

item id *integer*: identifier of associated item;

position *integer*:*integer* (see above);

attribute *string*: phenomenon-specific parameter name;

value *string*: value for parameter *attribute*;

instance *string*: substring in *input* corresponding to *position*;

commend *string*: free text.

3.6 Test Set Window

As an optional descriptive device inbetween the test item and phenomenon levels pairs or sets of grammatical and ill-formed items can be grouped into so-called *test sets* (see figure 3.3). Thus, it can be encoded how for example test items that originate from the systematic variation of a single phenomenon parameter relate to each other. Following is a summary of the individual fieds, value restrictions and — where appropriate — interpretations (see [Estival *et al.* 1994] and [Estival *et al.* 1995] for further details).

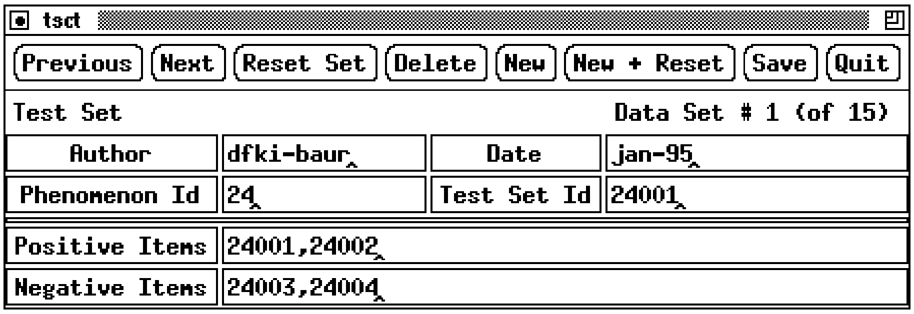


Figure 3.3: The *Test Set* window of tsct.

author *string*: creator of this record;

date *date* in tsdb_1 format (allowing underspecification): date of creation;

phenomenon id *integer* (see above);

test set id *integer*: unique identifier

positive items *integer*[,*integer*]*: comma-separated list of item identifiers

negative items *integer*[,*integer*]*: comma-separated list of item identifiers

3.7 User & Application Profile Window

Information that will typically correlate with different evaluation and application types in the use of a test suite is factored from the remainder of the data into *user & application profiles* (see figure 3.4). Currently, we have foreseen (i) a centrality measure on a per phenomenon, test set or even test item basis (e.g. user x may in general consider phenomenon y to be central but test set z to be marginal within y) and (ii) judgements of relevance with respect to a specific application. To match the results obtained from some NLP system (a parser, say) against a test suite, users can add informal or formal output specifications at the profile level. As these are dependent on the purpose of the use of the test suite, there is no template provided (figure 4.3 gives an excerpt from the DFKI user & application profile as an example).

In general, the user & application profiles are considered to be a hook into the TSNLP database that developers or end users with only a very basic knowledge of relational database technology shall use to extend the database schema and add user- or application-specific data without touching the core inventory of test items and annotations. Accordingly, data in this part of the database will not necessarily be input through tsct — which typically will be too restricted in its static number of fields — but possibly imported from different sources (e.g. output specifications for a specific application might be produced semi- or fully automatically).

3.8 A Typical Test Data Construction Scenario

The following algorithm (sequence of actions) has been suggested for the test data construction using tsct; and even though probably procedures or shortcuts very different to

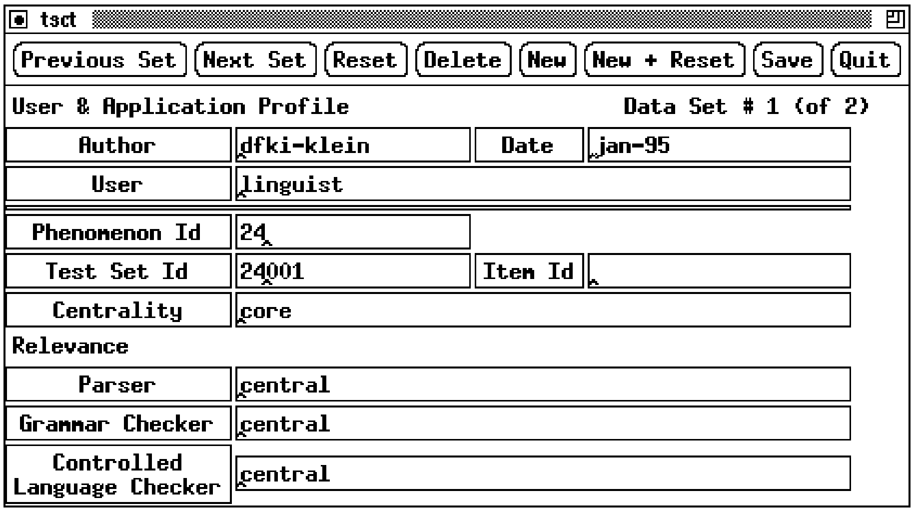


Figure 3.4: The *User & Application Profile* window of tsct.

these may turn out to be applicable depending on the individual preferences of some test data author or the specific characteristics of some phenomenon,[2] the algorithm proposed can be read as a reasonable (but naturally informal) procedure to follow (the instructions to save frequently have meanwhile become less important as tsct has matured and become more stable).

(1) Fill in a particular *Phenomenon* description and identify the parameters that play a role in testing and systematic variation;

(2) instantiate the *Test Item* form and save it;

(3) paste the id from step (2) into the *Phenomenon* window; add any relevant *Parameters* information and save it;

(4) iterate through steps (2) and (3);

(5) organize the test items from steps (2) into test sets; fill in the *Test Set* window accordingly;

(6) iterate through steps (2) to (5);

(7) add the classification for the current phenomenon to the *User & Application Profile* window;

(8) continue with step (1).

3.9 The tsct Data File Format

Because the design and implementation of the TSNLP database was scheduled only towards the end of the project, the tsct data file format had to serve several different purposes (viz.

[2]In the German coordination data, for example, the core strings and some of the annotations already result from systematic permutations and deletions with respect to an initial (or base) sentence that were computed off-line; in this setup, tsct is then used to browse the resulting data and add missing annotations.

data exchange and test item retrieval during the test data construction and testing phases) throughout the project. To meet the various requirements

- tsct organizes the test data into human-readable ASCII files that can be manipulated using ordinary text processing tools (e.g. standard editors and Un∗x commands or even Microsoft Word macro processing); and

- at the same time the use of tsct enforces the data to be stored in a well-defined machine-readable exchange format.

See section 5 for the import procedure that transforms tsct data files into the final TSNLP database format.

Chapter 4

The Test Suite Database (tsdb)

4.1 Motivation

Since the main objectives for the test data storage, maintenance and retrieval in TSNLP are *usability*, *suitability* and *extensibility* (see section 1), special emphasis in the database design is put on the aspect of flexibility and interfaces (to both people and programs); additionally, as technological requirements of developers or users from an academic vs. industrial background may be different, from a technical perspective *portability* and *simplicity* of the database were included into the list of desiderata.

Firstly, in order to produce a tool that is well tailored to the purpose of storing and retrieving TSNLP test data, highly flexible in its interfaces and portable and free of royalties at the same time, a small and simple relational database engine (which we presently call tsdb_1 using the subscript to distinguish between the two parallel implementations of tsdb; see below) has been implemented in plain ANSI C. Since the expected size of the database as well as major parts of the database schema are known in advance, for this implementation it seemdd plausible to impose a few restrictions on the full generality of relational database management systems, thus allowing to fine-tune and simplify both the program and especially the query language.

Secondly, though outside of TSNLP proper, a parallel implementation of the test suite database (called tsdb_2) has been carried out at DFKI experimenting with a commercial database product. Because commercially available database systems have different assets (and typically many more features) than the home-grown implementation tsdb_1, the dual realizations of the TSNLP database carried out in parallel nicely complement each other and provide valuable feedback and comparison; see [Oepen *et al.* 1996] for an overview of tsdb_2 and how to obtain it.

Thirdly, in an extension to the original TSNLP contract, a similar strategy (building on the Microsoft Access database package for MS Windows) has been carried out at the University of Essex.

4.2 The TSNLP Database Schema

While the tsct data format was primarily motivated by data construction considerations (and possibly the requirements imposed by its consortium-internal use before the actual database implementation), the design goals for the abstract TSNLP data model (and analogously for the tsdb database schema) are somewhat different.

Using the construction tool and its reuse and duplicate facilities, the resulting data

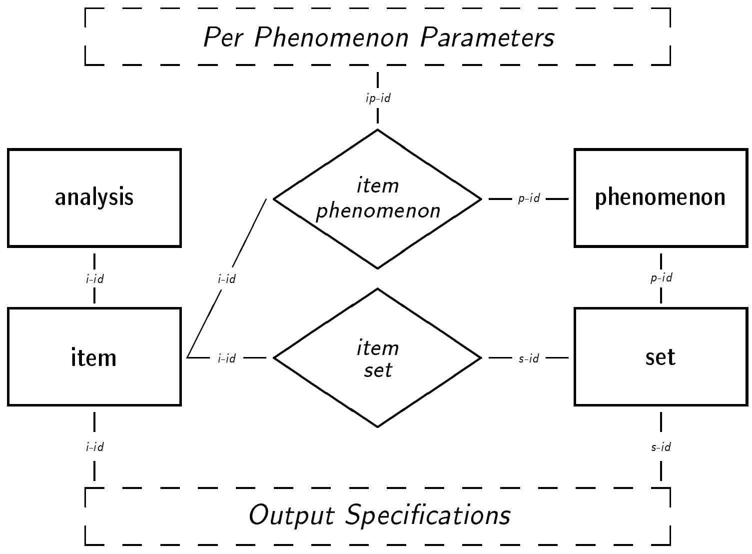


Figure 4.1: Graph representation of the TSNLP database schema; see figure 4.2 for details on the individual relations.

files are highly redundant in that static fields (see section 3.3 for the distinction of static vs. reset fields) are stored with each individual data record; looking at the *Phenomenon* window (in figure 3.2) of tsct where about half of the fields are static, it is obvious that the rough classification into four basic relations serves very well for data input purposes but falls short from a data storage and maintenance perspective.

Accordingly, the four-valued classification corresponding to the tsct windows has to be refined and cast into more specific relations to allow for the elimination of redundancy. In addition to eliminating redundant information, the database schema has to more clearly

(i) factor basic entity (or object) types from relationship types (for a semi-formal introduction to the entity-relationship conception of databases see [Elmasri and Navathe 1989]); and

(ii) make the distinction explicit in its postulation of relations and index attributes used to relate entity types to each other in order to allow for variable logical views on the data.

Another important issue in the refinement of the data model is the representation of list- or multi-valued attributes which usually are not allowed in plain relational databases. Hence, the tsct representation of test sets — typically relating a comma-separated group of positive test items (or rather their identifiers) to a set of negative items — has to be transformed into multiple data records that relate the individual items to one and the same test set (viz. its identifier again); but since the lists of positive and negative items may not match up pairwise and in order to avoid empty fields, the polarity (positive or negative) for each of the items is cast as an attribute of its own.[1] As an example for

[1]The reason to introduce a separate *polarity* attribute in addition to the already existing *i-wf* judgement on grammaticality of the test items themselves, is to still allow for test sets for which well-formed items can serve as negative examples or vice versa. In testing a controlled language checker, for example, it might

Basic Entity Types

item
- i-id *:integer :key*
- i-origin *:string*
- i-register *:string*
- i-format *:string*
- i-difficulty *:integer*
- i-category *:string*
- i-input *:string*
- i-wf *:integer*
- i-length *:integer*
- i-comment *:string*
- i-author *:string*
- i-date *:string*

analysis
- i-id *:integer :key*
- a-position *:position*
- a-instance *:string*
- a-category *:string*
- a-function *:string*
- a-domain *:string*
- a-tag *:string*
- a-comment *:string*

phenomenon
- p-id *:integer :key*
- p-name *:string*
- p-supertypes *:string*
- p-presupposition *:string*
- p-interaction *:string*
- p-purpose *:string*
- p-restrictions *:string*
- p-comment *:string*
- p-author *:string*
- p-date *:string*

parameter
- ip-id *:integer :key*
- position *:string*
- attribute *:string*
- value *:string*
- instance *:string*
- comment *:string*

set
- s-id *:integer :key*
- p-id *:integer :key*
- s-author *:string*
- s-date *:string*

Relationship Types

item-phenomenon
- ip-id *:integer :key*
- i-id *:integer :key*
- p-id *:integer :key*
- ip-author *:string*
- ip-date *:string*

item-set
- i-id *:integer :key*
- s-id *:integer :key*
- polarity *:integer*

Figure 4.2: TSNLP database schema. The *item* and *analysis* relations correspond to the tsct *Test Item* window; the *phenomenon* and *item-phenomenon* relations to the *Phenomenon* and the *set* and *item-set* relations to the *Test Set* windows respectively. Relationship types are postulated for $m \times n$ relations only; $n \times 1$ relationships rather are represented as additional index attributes of the entity types (e.g. the *i-id* attribute of the *analysis* relation).

i-id	system	version	readings	first	total	user	date	error
11001	page	4711	1	0.9	2.0	oe	23-sep-1995	
11006	page	4711	2	3.6	8.8	oe	23-sep-1995	
11178	page	4711	-1			oe	23-sep-1995	bus error

i-id	nll (normalized meaning representation)
11001	direct(theme:ˆ(∃ ?x ?y addressee(instance:?x) work(instance:?y agent:?x)))
11006	direct(theme:ˆ(∃ ?x ?y addressee(instance:?x) work(instance:?y agent:?x)))
11006	ask(theme:ˆ(∃ ?x ?y addressee(instance:?x) work(instance:?y agent:?x)))

Figure 4.3: Information added in customizing the German test suite for testing the DFKI HPSG system: the DFKI *user & application profile* records processing measures (e.g. the grammar version and date of testing, the number of readings obtained, the processing times for finding the first reading and overall total, and the type of processing error where applicable) for each of the test items; rather than deploying bracketed phrase structure trees as an output specification, a normalized semantic representation serves as a reference point to judge the quality of analysis and to identify spurious ambiguities.

this transformation consider the following records from the **set** and **item-set** relations respectively which are the tsdb equivalent of the test set in figure 3.3:

s-id	*p-id*	*info-id*
24001	24	4711

i-id	*s-id*	*polarity*
24001	24001	1
24002	24001	1
24003	24001	0
24004	24001	0

Figures 4.1 and 4.2 present the core of the database schema as it has been been used throughout TSNLP. While the annotation schema is considered stable for the *test item*, *phenomenon*, and *test set* levels, naturally, *user & application profiles* will vary with a particular type of application and domain that TSNLP is applied to. Figure 4.3 gives an example of the DFKI *user & application profile* that has been developed to facilitate continuous progress evaluation and regression testing for a large-scale grammar development project.

4.3 A Few Open Questions

The tsdb database schema — unlike the tsct data model which basically followed very pragmatic data construction considerations — is determined by formal and data organization criteria; therefore, there cannot be a one-to-one mapping between the two. Accordingly, the import procedure into tsdb has to incorporate a few non-trivial transformations and poses some questions which mostly deal with the formal nature of the representation.

- In the *Analysis* part of the *Test Item* window the pair of attributes *function* and *domain* is overloaded in the description of functor – argument dependencies: whereas

be desirable to construct test sets containing grammatical items that still are assigned negative polarity simply because they are not part of the controlled target language (e.g. based on a distinction between active vs. passive voice).

functor elements use *domain* to indicate the substring in which they act as the top-level functor, arguments, in turn, have the position of their functor as the value of *domain*. As the two interpretations of the attribute *a-domain* are rather different, it would have been an option to recast the dependency structure in a separate relation. Yet, there is no loss of information in the current encoding as long as there is agreement on its intended interpretation.

- In order to be maximally general in the description of phenomenon-specific parameters, the tsct *Phenomenon* window has the fields *attribute* and *value* which, strictly speaking, serve as a meta-level description of per phenomenon relations. Again, the current database schema decides to preserve the tsct encoding in the database; however, an obvious disadvantage of this method is that it cannot impose any restrictions on the number and type of parameters associated with some token phenomenon.
- The *User & Application Profile* of tsct basically can only serve as an initial approximation of the information that shall go into this part of the database (e.g. it has no fields to allow the input of application-specific output specifications since these typically will have variable formats).

In general, in the extensive TSNLP data construction and testing so far none of the issues listed above has lead to problems; thus, the database schema as it has now been in use for a while seems to be sufficiently well-behaved for the task at hand.

4.4 Query and Retrieval: Some Examples

For the TSNLP test data the basic purpose of the relational database retrieval facilities are:

- to dynamically construct test suite instances according to an arbitrary selection from the data; and
- to employ the query engine to compute appropriate virtual (join) relations that contain everything relevant for a particular task but at the same time hide away all internal, bookkeeping, or irrelevant data.

— Following are a few retrieval examples formulated in the simplified SQL-like query language that is interpreted by the home-grown implementation tsdb_1 (see section 4.5):

- find all grammatical verb-second sentences relevant to the phenomenon of clausal agreement:

```
retrieve i-input
  where i-wf = 1 && i-category = "S" &&
        p-name = "C_agreement"
```

- collect all grammatical sentences with pronominal subjects together with their identifiers and categories:

```
retrieve i-id i-input i-category
  where i-wf = 1 &&
        a-function = "subj" && a-category ~ "PRON"
```

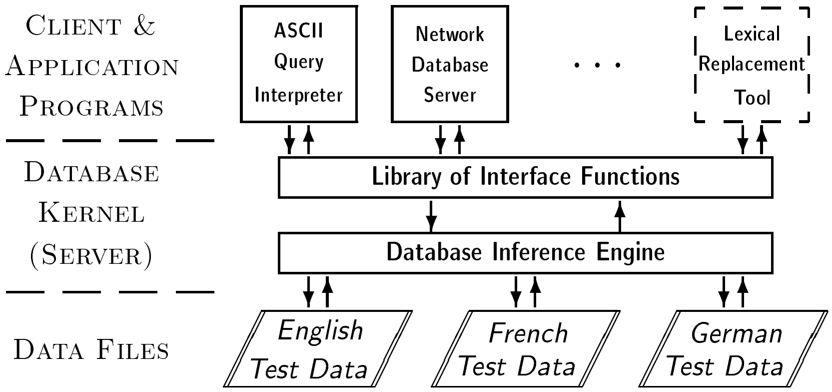


Figure 4.4: Rough sketch of the modular tsdb_1 design.

4.5 Implementing tsdb_1: Building It Ourselves

The motivation for the dual database implementation has already been given in section 4.1. And although there was no intention to compete with commercial database products in terms of efficiency and generality, there are good reasons speaking in favour of the proprietary implementation of tsdb_1 within TSNLP:

+ the database software and query language can be fine-tuned to the test suite extraction task; because the focus of natural language test data retrieval is on string manipulation, the tsdb_1 implementation was enriched with regular expression matching (a concept that is typically unavailable in standard relational database products);

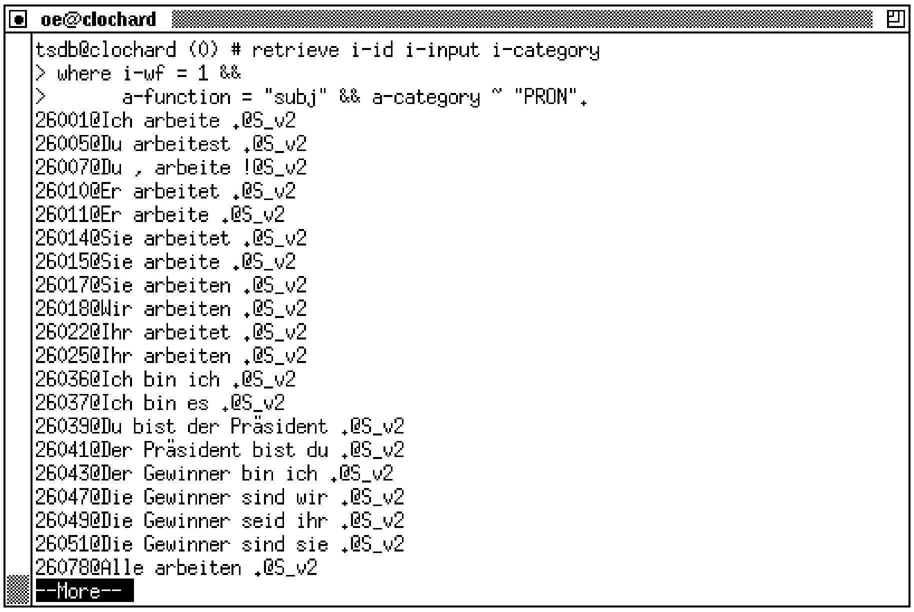
+ both data *and* software can be made available to interested parties royalty-free;

+ the database can be given a flexible and well-documented interface allowing to connect it to arbitrary (client) application within or outside of TSNLP; the applicability of the interface has been exemplified in several cases within TSNLP (see below) and is currently exploited in integrating tsdb_1 into the ALEP (Advanced Linguistic Engineering Platform); and

+ the data format can be designed to be transparent (similar to the tsct ASCII files) and is thus accessible to standard Un∗x text processing utilities.

Figure 4.4 gives an overview of the tsdb_1 architecture: the database kernel consists of the inference engine and a separate layer of interface functions that are organized into a library of C functions.

Thus, any application that is capable of interfacing with a library of external functions (e.g. programs that can be statically linked with the library or Common-Lisp or Prolog applications by use of the foreign function interface) can be given full bidirectional access to the database. Within TSNLP two of the client applications were developed, viz. an ASCII-based query interpreter that allows to retrieve data in a subset of the SQL query language (figure 4.5 shows a screen dump from a retrieval session using tsdb_1) and a network database server allowing remote (though read-only) access to the TSNLP test data over a socket-based network.



```
oe@clochard
tsdb@clochard (0) # retrieve i-id i-input i-category
> where i-wf = 1 &&
>       a-function = "subj" && a-category ~ "PRON".
26001@Ich arbeite .@S_v2
26005@Du arbeitest .@S_v2
26007@Du , arbeite !@S_v2
26010@Er arbeitet .@S_v2
26011@Er arbeite .@S_v2
26014@Sie arbeitet .@S_v2
26015@Sie arbeite .@S_v2
26017@Sie arbeiten .@S_v2
26018@Wir arbeiten .@S_v2
26022@Ihr arbeitet .@S_v2
26025@Ihr arbeiten .@S_v2
26036@Ich bin ich .@S_v2
26037@Ich bin es .@S_v2
26039@Du bist der Präsident .@S_v2
26041@Der Präsident bist du .@S_v2
26043@Der Gewinner bin ich .@S_v2
26047@Die Gewinner sind wir .@S_v2
26049@Die Gewinner seid ihr .@S_v2
26051@Die Gewinner sind sie .@S_v2
26078@Alle arbeiten .@S_v2
--More--
```

Figure 4.5: Screen dump of a tsdb_1 session: query results are displayed through a pager (`more(1)` in this case) and written into ASCII files. The tsdb_1 ASCII query interpreter (topmost three lines) has command line editing and relation and attribute name completion facilities.

The tsdb_1 query language provides for the standard numeric and string comparison operations plus regular expression matching and the boolean connectives (viz. conjunction, disjunctio,n and negation of conditions).

Other applications to connect to the TSNLP database could include the TSNLP lexical replacement tool (which currently reads and writes data files in the tsct format) and possibly a graphical browser for test data derived from tsct itself.

In addition to the work done on tsdb clients, DFKI has deployed the functional tsdb_1 interface in connecting the database to the DFKI PAGE (Platform for Advance Grammar Engineering) that during the testing phase was used to test for the adequacy of TSNLP data and annotations. For this application the bidirectional interface allowed for automated retrieve, process ,and compare cycles in grammar development and diagnosis (see [Regnier-Prost *et al.* 1995]).

The implementation of tsdb_1 is in ANSI C and supports (among others) Un∗x workstations as well as Macintosh and Intel x86-based personal computers. The interface specification is as a C header file to be included by client programs (see appendix C).

4.6 tsdb_1: Operation and Usage

The tsdb_1 ASCII standalone query processor is highly parameterizable through command line options and environment variables.

Command Line Options All command line options are optional, can be given in arbitray order and be abbreviated (as long as the prefix remains unambiguous); the command line synopsis is the following:

```
tsdb -server[=host]
     -client
     -port=n
     -home=base directory
     -relations-file=relations file
     -data-path=data directory
     -result-path=result directory
     -result-prefix=prefix
     -max-results[=m]
     -debug-file=debug file
     -pager[={off | command}]
     -query=query
     -usage
     -help
     -version
```

where the individual options mean

- `-server`: go into server (daemon) mode or — if *host* is given — connect to tsdb_1 server on *host*;
- `-client`: go into client mode connection to local server;
- `-port`: establish server socket at port n or — in client mode — connect to server at port n (default is 4711);
- `-home`: locate all database files relative to *base directory*;
- `-relations-file`: locate database schema in *relations file* (absolute or relative file name);
- `-data-path`: locate tsdb_1 data files in *data directory* (absolute or relative directory name);
- `-result-path`: store query results into *result directory* (absolute or relative directory name);
- `-result-prefix`: generate query result file names from *prefix* (full file name is *result directory* + *prefix* + result number (if $m > 1$);
- `-max-results`: disable query result storage or — with numerical argument — set maximal number of query result files generated to m;
- `-debug-file`: log debug information to *debug file* (absolute or relative file name);
- `-pager`: disable interactive browsing of data through pager (default is platform-dependent; typically `more(1)`) or — with string argument use *command* as pager;
- `-query`: run in batch mode processing *query*, writing query result to the standard output and returning the result code as exit status;
- `-usage` or `-help`: print summary on command line synopsis;
- `-version`: print current tsdb_1 version, revision, and copyright.

The `-server`, `-client`, and `-port` options control the database network server mode. Note that `-client` (i.e. using the tsdb_1 standalone query processor as the frontend talking to a tsdb_1 network server) is not yet implemented; however, using `telnet(1)` (or similar socket-based clients) to connect to the appropriate port allows to communicate with a tsdb_1 process in daemon mode.

The `-home`, `-relations-file`, and `-data-path` control how the tsdb_1 database schema and database files are located; the default values are '`.`' (for `-home`), '`etc/relations`' (for `-relations-file`), and '`german/`' (for `-data-path`) respectively such that the tsdb_1 standalone query processor can be run without further customization from the directory resulting from unpacking the distribution. Both, absolute and relative file and path names are valid values for the `-home`, `-relations-file`, and `-data-path` options where relative file or path names are expanded relative to the current `-home` value or — in the case of the `-home` option itself — to the current working directory.

The `-result-path`, `-result-prefix`, and `-max-results` options control the generation of query result storage files; default values are '`/tmp/`' (for `-result-path`), '`tsdb.query.`*user*`.`' (for `-result-prefix`), and '`20`' (for `-max-results`) respectively such that, for examples, for a user name '`oe`' tsdb_1 will generate up to twenty files in '`/tmp`' using the names '`tsdb.query.oe.1`' to '`tsdb.query.oe.20`' where version 1 always corresponds to the most recent query result (i.e. for existing query result storage files the version number is incremented and the file renamed such that the file with a version number equal to `-max-results` is overwritten). Used without an argument or a value of 0, the `-max-results` option disables the tsdb_1 query result storage facility.

The `-debug-file` option allows to set the file used by tsdb_1 to log debug information (assuming the '`-DDEBUG`' flag was set at compile time); the default value is '`/tmp/tsdb.debug.`*user*' and there should rarely be a reason to it (possibly to '`/dev/null`'). In server (daemon) mode, tsdb_1 will use the port address rather than a user name as the suffix in generating the `-debug-file` name.

Using the `-pager` option allows to set the pager program used in interactive tsdb_1 mode to display query results; the default value (one of `more(1)`, `less(1)`, or `page(1)`) depends on the platform used to run tsdb_1. When using the `-pager` option, the pager command either has to be in the shell search `PATH` or specified as an absolute file name; a value of '`null`', '`off`', or no value at all disables the tsdb_1 pager facility.

The `-query` options runs tsdb_1 in batch (rather than interactive) mode and exits after processing the `-query` argument. Note that whitespace and special characters (e.g. '`&`', '|', '`>`' '`"`' et al.) have to be escaped from the shell; usually, a pair of single quotes ('`'`') surrounding the `-query` argument should be sufficient.

Environment Variables The majority of the tsdb_1 variables controlled by command line options can be set through the shell environment as well. Following is a list of environment variables relevant to tsdb_1; the interpretation and admissible values are similar to the corresponding command line options:

- `TSDB_HOME`
- `TSDB_RELATIONS_FILE`
- `TSDB_DATA_PATH`
- `TSDB_RESULT_PATH`
- `TSDB_RESULT_PREFIX`

- TSDB_MAX_RESULTS
- TSDB_PAGER
- PAGER

Thus, evaluating (e.g. from the shell initialization file '.bashrc')

```
export TSDB_DATA_PATH=french
```

or (for old-fashioned csh(1) users from '.cshrc')

```
setenv TSDB_HOME /home/tsdb
```

changes the default for the data directory to (a subdirectory) 'french' or, respectively, announces '/home/tsdb' as the home (root directory) of the tsdb database such that the tsdb_1 executable can then be used from arbitrary directories.

Besides the -home command line option and the TSDB_HOME environment variable there is another way to determine the tsdb_1 root directory: at startup the database will check for the existence of a tsdb pseudo user account and — if available — use its home directory as the tsdb_1 root directory; the default user names checked are 'tsdb' and 'TSDB' but these could be changed at compile time through the '-DTSDB_PSEUDO_USER' compiler flag.

Additionally, running tsdb_1 under the name 'tsdbd' (through linking or copying the binary file) puts the database into server (daemon) mode (similar to the -server option).

tsql Query Language Following is the tsql syntax in extended Backus-Naur form (EBNF):

- ⟨query⟩ → { ⟨info⟩ | ⟨set⟩ | ⟨retrieve⟩ | ⟨insert⟩ }
- ⟨info⟩ → 'info'
  { 'all' | 'relations'
  | ⟨relation⟩
  | ⟨tsdb constant⟩ | ⟨tsdb variable⟩ }
- ⟨set⟩ → 'set' ⟨tsdb variable⟩ { ⟨integer⟩ | ⟨string⟩ | ':on' | ':off' }
- ⟨retrieve⟩ → { 'retrieve' | 'select' }
  { ⟨attribute⟩+ | '*' }
  ['from' { ⟨relation⟩+ | ⟨integer⟩ }]
  ['where' ⟨condition⟩]
  ['report' ⟨format string⟩]
- ⟨insert⟩ → 'insert' 'into'
  ⟨relation⟩ [⟨attribute⟩+]
  'values' { ⟨integer⟩ | ⟨string⟩ | ⟨date⟩ }+
- ⟨tsdb constant⟩ → { 'home' | 'tsdb_home'
  | 'relations-file' | 'tsdb_relations_file'
  | 'data-path' | 'tsdb_data_path' }

- ⟨tsdb variable⟩ → { ‘`result-path`’ | ‘`tsdb_result_path`’
  | ‘`result-prefix`’ | ‘`tsdb_result_prefix`’
  | ‘`max-results`’ | ‘`tsdb_max_results`’
  | ‘`uniquely-project`’ | ‘`tsdb_uniquely_project`’ }
- ⟨relation⟩ → ⟨identifier⟩
- ⟨attribute⟩ → ⟨identifier⟩
- ⟨condition⟩ → { ⟨attribute⟩ {‘=’ | ‘! =’ | ‘∼’ | ‘! ∼’ } ⟨string⟩
  | ⟨attribute⟩ {‘=’ | ‘! =’ | ‘<’ | ‘>’ | ‘<=’ | ‘>=’ } ⟨integer⟩
  | ⟨attribute⟩ {‘=’ | ‘! =’ | ‘<’ | ‘>’ | ‘<=’ | ‘>=’ } ⟨date⟩
  | ⟨condition⟩ { ‘`&`’ | ‘`&&`’ | ‘`and`’ } ⟨condition⟩
  | ⟨condition⟩ { ‘|’ | ‘||’ | ‘`not`’ } ⟨condition⟩
  | { ‘`!`’ | ‘`not`’ } ⟨condition⟩
  | ‘`(`’ ⟨condition⟩ ‘`)`’
- ⟨integer⟩ → [{ ‘+’ | ‘−’ }] ⟨digit⟩+
- ⟨digit⟩ → { ‘0’ | ‘1’ | ... | ‘9’ }
- ⟨string⟩ → { ‘`"`’ ⟨*any character except* ‘`"`’⟩ ‘`"`’
  | ‘`‘`’ ⟨*any character except* ‘`‘`’⟩ ‘`’`’ }
- ⟨date⟩ → { [⟨day⟩ ‘−’] ⟨month⟩ ‘−’ ⟨year⟩ [[‘`(`’] ⟨time⟩ [‘`)`’]]
  | ‘`:today`’
  | ‘`:now`’ }
- ⟨day⟩ → [⟨digit⟩] ⟨digit⟩
- ⟨month⟩ → [⟨digit⟩] ⟨digit⟩
- ⟨year⟩ → [⟨digit⟩ ⟨digit⟩] ⟨digit⟩ ⟨digit⟩
- ⟨time⟩ → ⟨digit⟩ ⟨digit⟩ ‘`:`’ ⟨digit⟩ ⟨digit⟩ [⟨digit⟩ ⟨digit⟩]
- ⟨identifier⟩ → ⟨character⟩ { ⟨character⟩ | ⟨digit⟩ | ‘−’ | ‘_’ }+
- ⟨character⟩ → { ‘`A`’ | ‘`B`’ | ... | ‘`Z`’ | ‘`a`’ | ‘`b`’ | ... | ‘`z`’ }

Relations File Format The tsdb_1 database schema file is plain ASCII: relations are separated by one or more empty lines (similar to paragraphs in TEX). The first line of a paragraph is the relation name followed by a colon (‘`:`’); the remaining lines of a paragraph define attributes (one per line) for this relation. Each attribute must have a datatype (one of ‘`:integer`’, ‘`:string`’, and ‘`:date`’); additionally, the token ‘`:key`’ can be used to designate one or more attributes as keys that serve for building complex (join) relations.

Data File Format The data file format is similar to tsct: plain ASCII files encoding one record per line with fields separated by ‘`@`’ characters (the default value of ‘`-DTSDB_FS`’ when compiling from source).

Command Line Editing and History tsdb$_1$ builds on the GNU (remember: GNU is not Un∗x) readline and history libraries for the interactive command line editing and history facilities (similar to e.g. GNU `bash(1)` and `gdb(1)`). The relevant parts of the GNU documentation are included with the tsdb$_1$ distribution (see appendix B) as both '`.dvi`' and '`.ps`' files.

Regular Expression Matching tsdb$_1$ uses the GNU regular expression library to implement the match ('∼') operator on strings; hence, the tsdb$_1$ regular expression syntax is mostly like in GNU Emacs. The GNU regular expression documentation (available in both '`.dvi`' and '`.ps`' format) is included with the tsdb$_1$ distribution (see appendix B).

Chapter 5

Import and Consistency Checking (import)

The import procedure from tsct into tsdb is implemented in the `awk(1)` programming language; it reads in tsct data files sequentially, distributes the individual data records and fields over the appropriate tsdb relations, provides for the necessary transformations and links between relations and writes the resulting data into files again. Besides, the import procedure is the appropriate time and place to normalize and test for the basic logical consistency of the data regarding properties that are local to, say, one data record or input file at a time. Normalization and consistency checking include:

- capitalization and inter-word and inter-punctuation spacing normalization;
- elimination of empty fields;
- checking of data types (for numerical fields and attributes with a limited set of admissible values);
- removal of duplicates (possibly originating from spelling variations in attribute values);
- inspection of *i-input* vs. *i-length* and *a-position* vs. *a-instance* mismatches; and
- generation of new unique identifiers where necessary (e.g. for the *ip-id* attribute).

The output file format of the import procedure is immediately suitable for the home-grown tsdb_1 implementation (see section 4.5): one ASCII file per relation with one record per line and variable-width fields delimited by a special field separator character. At the same time, the version of tsdb based on Microsoft FoxPro can directly read these files and convert them into its internal binary format. For further details on the tsct to tsdb import procedure see appendix D.

5.1 import: Operation and Usage

For the conversion of the tsct data files into tsdb data files, a `gawk(1)` program has been developed; `gawk(1)` offers the necessary facilities for processing files in the form of records and fields.

Prerequisites To run import the standard Un*x tools `gawk(1)` and `bash(1)`[1] as well as the tssi tool are required. `tssi(1)` (test suite string intelligence) is part of the tsu bundle (see section 6) and is used to check and possibly correct the *position* and *length* fields in the *analysis* relation. Basic knowledge of `gawk(1)` may be necessary when the adaptation of the scripts becomes desirable.

The script does not install itself; the user will need to make some changes to be able to run the conversion script. The two changes which will be needed for (nearly) every site, are

- the first line of the file '`import`', containing the location of the `gawk(1)` executable, e.g. '`#!/usr/gnu/bin/gawk`'. This will be different for different sites; you can find out about the `gawk(1)` location for most sites using the shell command '`which gawk`' (for old-fashioned `csh(1)` users) or '`type -path gawk`' (in `bash(1)`).
- the path to the `tssi(1)` executable; either `tssi(1)` has to be in the shell search path (the value of the '`PATH`' environment variable) or it has to be added to the setting of '`PATH`' in the top-level `bash(1)` script '`import`'.

Synopsis The program consists of a number of files. The main program is the `bash(1)` script '`import`'. This calls other files for initialization, checking the input file consistency, and finally converting the various tsct files into tsdb format.

While running, the script displays on the standard output an explanation of what it is doing. When the output directory already exists, this is signalled to the user as well.

The command line synopsis for import is as follows

`import` [*target directory*] tsct *basename*

where tsct *basename* is assumed to be the basename (i.e. prefix) of a set of tsct data files (with suffixes '`.items`', '`.phenomena`', and '`.sets`') and *target directory* is a tsdb database.

If *target directory* exists and contains tsdb data, the database is augmented incrementally. In particular, import uses the existing data in checking for identifier uniqueness; thus, it is impossible to import the very same tsct data into one tsdb database twice. The tsdb *target directory* is created by import if necessary. If *target directory* is ommitted, its name is derived from the tsct *basename* parameter following the Un*x `basename(1)` conventions; e.g. the two calls

- `import c_complementation tsct/c_complementation`
- `import tsct/c_complementation`

both have the same effect of reading the files

- '`c_complementation.items`',
- '`c_complementation.phenomena`', and
- '`c_complementation.sets`'

[1]Current versions of `gawk(1)` and `bash(1)` are preferable, as earlier versions caused some problems. Both `gawk(1)` and `bash(1)` are free software developed and maintained by the Free Software Foundation (remember: GNU is not Unix) and can be obtained from numerous anonymous `ftp(1)` sites around the world.

from the 'tsct' subdirectory and writing data to the tsdb database in the 'c_complementation' subdirectory (which is created if necessary). Running import without any arguments gives a brief summary of its command line synopsis.

The script reads the tsct files and converts them into tsdb tables. Before doing this, all fields that contain values from a certain domain, are checked for validity of syntax and on correctness of the values. In case a record does not comply with these requirements, an error message (or a warning message) is generated and written into a corresponding log file. These files are called 'error_log' and 'warning_log' respectively and created in *target directory*.

The difference between error messages and warning messages is that the former stop the script after the checks are finished (i.e. in case of an error no data is imported into *target directory*) while the latter allow the import procedure to continue. These files contain the name of the file containing the erroneous data, the record number, a diagnostic message and the field(s) where the error occurred. The tsu utilities 'get' and 'remove' can then be used inspect the incriminated data and — where desirable — remove individual records or fields.

There are also checks on the number of fields, and on co-occurrence of values within one record (e.g. if *instance* in *parameter* is filled in, then also *position* should be filled in, and vice versa) and on the uniqueness of identifiers. If the user wants to relax the scripts, then that needs to be done in the program; however, there is some provision for disabling a number of TSNLP-specific checks by setting the variable 'lazy' in the top-level file 'common.awk'.

Chapter 6

Test Suite Utilities (tsu)

The tsu package is a set of scripts (mostly in `awk(1)` and `sh(1)`) that have evolved throughout the project and add some useful functionality. Most of the tsu scripts make crucial use of the fact that both tsct and tsdb data files are organized as plain ASCII and thus accessible to standard Un∗x text utilities.

The overall motivation to make these scripts available as a separate package is that

- tools in tsu greatly facilitate the construction and maintenance of systematic and consistent test data;
- the collection of simple scripts encourages the extension and adaptation; and that
- tsu shall serve as a basic repository of tools to be drawn upon by future test suite developers and users.

Among others, tsu includes scripts to

- perform string lengths and substring computations following word counting conventions used in the tsct '`-intelligent`' mode;
- count test items according to wellformedness code;
- extract sorted list and frequency of vocabulary;
- extract or remove individual records or fields;
- generate unique consecutive identifiers; and to
- automate generation of inital tsct *phenomenon* and *parameters* files.

Each of the scripts has a header of comments explaining the purpose of the particular script and how to use it.

Appendix A

Contents of the Current tsct Distribution

Following is a listing of the contents of the current distribution of tsct version 1.4 (as of February 1996):

```
oe@clarinet (~) 52 $ tar ztf tsct.tar.z
tsct-1.4/
tsct-1.4/tsct
tsct-1.4/item.ari
tsct-1.4/phenomenon.ari
tsct-1.4/profile.ari
tsct-1.4/set.ari
tsct-1.4/README
```

The distribution is in `gzip(1)`ed `tar(1)` format and available from the TSNLP `ftp(1)` repository at `anonymous@tsnlp.dfki.uni-sb.de:/tsnlp/tsct/`.

Appendix B

Contents of the Current tsdb Distribution

Following is a listing of the contents of the current distribution of tsdb version 0.2 (as of February 1996):

```
oe@clarinet (~) 52 $ tar ztf tsdb.sunos.tar.z
tsdb-0.2/etc/
tsdb-0.2/etc/relations
tsdb-0.2/README
tsdb-0.2/german/
tsdb-0.2/french/
tsdb-0.2/english/
tsdb-0.2/doc/
tsdb-0.2/doc/regex.ps
tsdb-0.2/doc/rltech.texi
tsdb-0.2/doc/history.texi
tsdb-0.2/doc/hstech.texi
tsdb-0.2/doc/hsuser.texi
tsdb-0.2/doc/regex.texi
tsdb-0.2/doc/history.dvi
tsdb-0.2/doc/history.ps
tsdb-0.2/doc/rluser.texi
tsdb-0.2/doc/readline.texi
tsdb-0.2/doc/readline.dvi
tsdb-0.2/doc/readline.ps
tsdb-0.2/doc/regex.dvi
tsdb-0.2/tsdb
tsdb-0.2/libtsdb.a
tsdb-0.2/tsdb.h
```

The distribution is in `gzip(1)`ed `tar(1)` format and available from the TSNLP `ftp(1)` repository at `anonymous@tsnlp.dfki.uni-sb.de:/tsnlp/tsdb/`.

Appendix C

tsdb$_1$ Functional Interface: `tsdb.h`

Following is part of the ANSCI C header file '`tsdb.h`' included with the tsdb$_1$ distribtution detailing the functional interface to the tsdb interface library (see section 4.5).

```
/*****************************************************************************\
|*        file: tsdb.h
|*      module: TSDB global definitions and prototypes
|*     version: 0.2 -- 13-jan-96 (experimental)
|*  written by: andrew p. white, tom fettig & oe, dfki saarbruecken
|* last update: 2-apr-96
|*  updated by: oe, dfki saarbruecken
|*****************************************************************************|
|*
\*****************************************************************************/


#if defined(unix)
#  if defined(sun) && defined(__svr4__)
#    define SOLARIS
#  elif defined(sun) && !defined(__svr4__)
#    define SUNOS
#  elif defined(linux)
#    define LINUX
#  endif
#endif

#define TSDB_VERSION "0.2"

#define TSDB_DEFAULT_STREAM stdout
#define TSDB_ERROR_STREAM stderr

#define TSDB_SERVER_MODE 1
#define TSDB_CLIENT_MODE 2
#define TSDB_QUIT 4
#define TSDB_UNIQUELY_PROJECT 8

#define TSDB_UNKNOWN_TYPE 0
#define TSDB_INTEGER 1
#define TSDB_IDENTIFIER 2
#define TSDB_STRING 3
#define TSDB_DATE 4
#define TSDB_POSITION 5
#define TSDB_CONNECTIVE 6
#define TSDB_OPERATOR 7
#define TSDB_DESCRIPTOR 8
```

```
#define TSDB_VALUE_INCOMPATIBLE 0
#define TSDB_EQUAL 1
#define TSDB_NOT_EQUAL 2
#define TSDB_LESS_THAN 3
#define TSDB_LESS_OR_EQUAL_THAN 4
#define TSDB_GREATER_THAN 5
#define TSDB_GREATER_OR_EQUAL_THAN 6
#define TSDB_MATCH 7
#define TSDB_NOT_MATCH 8
#define TSDB_IMATCH 9
#define TSDB_NOT_IMATCH 10

#define TSDB_NOT 0
#define TSDB_AND 1
#define TSDB_OR 2
#define TSDB_NOT_NOT 3
#define TSDB_BRACE 4
#define TSDB_NONE 5

#define TSDB_NEW 0
#define TSDB_CHANGED 1
#define TSDB_UNCHANGED 2

#define TSDB_START_TIMER 0
#define TSDB_MAX_TIMERS 20

#define TSDB_SERVER_OPTION 0
#define TSDB_CLIENT_OPTION 1
#define TSDB_PORT_OPTION 2
#define TSDB_HOME_OPTION 3
#define TSDB_RELATIONS_FILE_OPTION 4
#define TSDB_DATA_PATH_OPTION 5
#define TSDB_RESULT_PATH_OPTION 6
#define TSDB_RESULT_PREFIX_OPTION 7
#define TSDB_MAX_RESULTS_OPTION 8
#define TSDB_DEBUG_FILE_OPTION 9
#define TSDB_PAGER_OPTION 10
#define TSDB_QUERY_OPTION 11
#define TSDB_USAGE_OPTION 12
#define TSDB_VERSION_OPTION 13
#define TSDB_HISTORY_OPTION 14
#define TSDB_UNIQUELY_PROJECT_OPTION 15
#define TSDB_COMPRESS_OPTION 16
#define TSDB_UNCOMPRESS_OPTION 17
#define TSDB_SUFFIX_OPTION 18

#ifndef TSDB_PSEUDO_USER
#  define TSDB_PSEUDO_USER "TSDB@tsdb"
#endif

#ifndef TSDB_DIRECTORY_DELIMITER
#  define TSDB_DIRECTORY_DELIMITER "/"
#endif

#ifndef TSDB_HOME
#  define TSDB_HOME "."
#endif

#ifndef TSDB_RELATIONS_FILE
```

```
#  define TSDB_RELATIONS_FILE "etc/relations"
#endif

#ifndef TSDB_DATA_PATH
#  define TSDB_DATA_PATH "german/"
#endif

#ifndef TSDB_RESULT_PATH
#  define TSDB_RESULT_PATH "/tmp/"
#endif

#ifndef TSDB_RESULT_PREFIX
#  define TSDB_RESULT_PREFIX "tsdb.query."
#endif

#ifndef TSDB_MAX_RESULTS
#  define TSDB_MAX_RESULTS 20
#endif

#ifndef TSDB_HISTORY_SIZE
#  define TSDB_HISTORY_SIZE 20
#endif

#ifndef TSDB_TEMPORARY_FILE
#  define TSDB_TEMPORARY_FILE "tsdb.tmp"
#endif

#ifndef TSDB_PAGER
#  define TSDB_PAGER "more"
#endif

#ifdef DEBUG
#  ifndef TSDB_DEBUG_FILE
#    define TSDB_DEBUG_FILE "/tmp/tsdb.debug"
#  endif
#endif

#ifndef TSDB_FS
#  define TSDB_FS '@'
#endif

#ifndef TSDB_BACKUP_SUFFIX
#  define TSDB_BACKUP_SUFFIX "~"
#endif

#ifndef TSDB_DEFAULT_VALUE
#  define TSDB_DEFAULT_VALUE ""
#endif

#ifndef TSDB_COMPRESS
#  define TSDB_COMPRESS "gzip -c -f"
#endif

#ifndef TSDB_UNCOMPRESS
#  define TSDB_UNCOMPRESS "gzip -c -f -d"
#endif

#ifndef TSDB_SUFFIX
#  define TSDB_SUFFIX ".gz"
#endif
```

```
#ifndef TSDB_SERVER_PORT
#  define TSDB_SERVER_PORT 4711
#endif

#ifndef TSDB_SERVER_QUEUE_LENGTH
#  define TSDB_SERVER_QUEUE_LENGTH 5
#endif

typedef struct tsdb_field {
  char *name;
  BYTE type;
  BOOL key;
} Tsdb_field;

typedef struct tsdb_value {
  BYTE type;
  union {
    int integer;
    char *identifier;
    char *string;
    char *date;
    char *position;
    BYTE connective;
    BYTE operator;
    int *descriptor;
  } value;
} Tsdb_value;

typedef struct tsdb_node {
  struct tsdb_node *left;
  Tsdb_value *node;
  struct tsdb_node *right;
} Tsdb_node;

typedef struct tsdb_relation {
  char *name;
  int n_fields;
  char **fields;
  BYTE *types;
  int n_keys;
  int *keys;
  BYTE *total;
  BYTE status;
} Tsdb_relation;

typedef struct tsdb_tuple {
  int n_fields;
  Tsdb_value **fields;
} Tsdb_tuple;

typedef struct tsdb_key_list {
  struct tsdb_value *key;
  int n_tuples;
  struct tsdb_tuple **tuples;
  struct tsdb_key_list *next;
} Tsdb_key_list;

typedef struct tsdb_selection {
  int n_relations;
```

```
  struct tsdb_relation **relations;
  int n_key_lists;
  struct tsdb_key_list **key_lists;
  int length;
} Tsdb_selection;

typedef struct tsdb_history {
  int command;
  char *query;
  struct tsdb_selection *result;
} Tsdb_history;

typedef struct tsdb {
  BYTE status;

  Tsdb_relation **relations;
  Tsdb_selection **data;

  char *input;

  char *home;
  char *relations_file;
  char *data_path;
  char *result_path;
  char *result_prefix;
  int max_results;

  char *server;
  int port;
  char *pager;
  char *query;
#ifdef DEBUG
  char *debug_file;
#endif

#ifdef COMPRESSED_DATA
  char *compress;
  char *uncompress;
  char *suffix;
#endif

  int command;
  Tsdb_history **history;
  int history_size;

  char *translate_table;
} Tsdb;

#if !defined(TSDB_C)
  extern Tsdb tsdb;

  extern FILE *tsdb_default_stream;
  extern FILE *tsdb_error_stream;
#ifdef DEBUG
  extern FILE *tsdb_debug_stream;
#endif

  extern char tsdb_version[];
  extern char tsdb_revision[];
  extern char tsdb_revision_date[];
```

```
#endif

int tsdb_parse(char *);
BOOL tsdb_verify_selection(Tsdb_selection *);
char *tsdb_pseudo_user();
float tsdb_timer(BYTE);

Tsdb_value *tsdb_integer(int);
Tsdb_value *tsdb_identifier(char *);
Tsdb_value *tsdb_string(char *);
Tsdb_value *tsdb_date(char *);
Tsdb_value *tsdb_position(char *);
Tsdb_value *tsdb_connective(BYTE);
Tsdb_value *tsdb_operator(BYTE);
Tsdb_value *tsdb_descriptor(int r,int f);
Tsdb_value **tsdb_singleton_value_array(Tsdb_value *);
Tsdb_value **tsdb_value_array_append(Tsdb_value **, Tsdb_value *);
Tsdb_field **tsdb_singleton_field_array(Tsdb_field *);
Tsdb_field **tsdb_field_array_append(Tsdb_field **, Tsdb_field *);

BYTE tsdb_value_compare(Tsdb_value *, Tsdb_value *);
BYTE tsdb_tuple_compare(Tsdb_tuple *, Tsdb_tuple *);
BOOL tsdb_verify_tuple(Tsdb_node *, Tsdb_tuple **);
BOOL tsdb_tuple_equal(Tsdb_tuple *, Tsdb_tuple *);
BYTE tsdb_value_match(Tsdb_value *, Tsdb_value *, char,void *);

FILE *tsdb_open_pager();
FILE *tsdb_open_debug();
void tsdb_close_debug(FILE *);

BOOL tsdb_print_value(Tsdb_value *, FILE *);
char *tsdb_sprint_value(Tsdb_value *);
char *tsdb_sprint_key_list(Tsdb_key_list *, int *, int *, int);
void tsdb_print_array(Tsdb_value **, FILE *);
void tsdb_print_relation(Tsdb_relation *, FILE *);
void tsdb_print_node(Tsdb_node *, FILE *);
void tsdb_print_tuple(Tsdb_tuple *, FILE *);
void tsdb_print_key_list(Tsdb_key_list *, FILE *);
void tsdb_print_join_path(Tsdb_relation **, FILE *);
void tsdb_print_projection(char **, int, char *, FILE *);
void tsdb_print_selection(Tsdb_selection *, FILE *);
int tsdb_uniq_projection(char **, int);

void tsdb_save_changes(void);

BOOL tsdb_are_joinable(Tsdb_relation *, Tsdb_relation *);
BOOL tsdb_is_attribute(Tsdb_value *);
BOOL tsdb_are_attributes(Tsdb_value **, Tsdb_relation *);
BOOL tsdb_is_relation(Tsdb_value *);
BOOL tsdb_relations_are_equal(Tsdb_relation *, Tsdb_relation *);
BOOL tsdb_initialize(void);
void tsdb_parse_environment(void);
BOOL tsdb_satisfies_condition(Tsdb_tuple *, Tsdb_node *, Tsdb_relation *);

char **tsdb_all_attribute_names(void);
char **tsdb_all_relation_names(void);
BOOL tsdb_attribute_in_relation(Tsdb_relation *, char *);
BOOL tsdb_attribute_in_selection(Tsdb_selection *, char *);
int tsdb_relation_in_selection(Tsdb_selection *, char *);
void tsdb_info(Tsdb_value **);
```

```
void tsdb_set(Tsdb_value *, Tsdb_value *);
int tsdb_drop_table(Tsdb_value *);
int tsdb_create_table(Tsdb_value *, Tsdb_field **);
int tsdb_alter_table(Tsdb_value *, Tsdb_field **);
int tsdb_insert(Tsdb_value *, Tsdb_value **, Tsdb_value **);
int tsdb_delete(Tsdb_value *, Tsdb_node *);
int tsdb_update(Tsdb_value *, Tsdb_node *);

void tsdb_project(Tsdb_selection *, Tsdb_value **, char *, FILE *);
FILE *tsdb_find_relations_file(char *);
FILE *tsdb_find_data_file(char *, char *);
FILE *tsdb_open_result();
char *tsdb_expand_directory(char *, char *);
char *tsdb_user(void);
char *tsdb_canonical_date(char *);
int *tsdb_parse_date(char *);
Tsdb_relation **tsdb_all_relations(void);
int tsdb_n_relations(void);
int tsdb_n_attributes(void);
Tsdb_relation *tsdb_copy_relation(Tsdb_relation *);

Tsdb_relation *tsdb_create_relation(void);
Tsdb_selection *tsdb_join(Tsdb_selection *, Tsdb_selection *);
Tsdb_selection *tsdb_select(Tsdb_selection *, Tsdb_node **, BYTE);
Tsdb_selection *tsdb_merge(Tsdb_selection *, Tsdb_selection *);
Tsdb_relation **tsdb_join_path(Tsdb_relation **, Tsdb_relation **);
Tsdb_selection *tsdb_retrieve(Tsdb_value **, Tsdb_value **,
                              Tsdb_node *, char *);

Tsdb_history *tsdb_get_history(int);
void tsdb_add_to_history(Tsdb_selection*);
void tsdb_set_history_size(int);
int tsdb_init_history(Tsdb*);

int tsdb_server_initialize(void);
void tsdb_server(void);
void tsdb_server_child(int);
int tsdb_socket_write(int, char *, int);
int tsdb_socket_readline(int, char *, int);
int tsdb_client(void);
```

Appendix D

The Import Procedure

Following is a listing of the contents of the current distribution of the import procedure import:

```
0 oe@clarinet (~) 137 $ tar ztf import.tar.z
import-2.1/
import-2.1/tsct2tsdb.bash
import-2.1/phenos-to-tsdb.gawk
import-2.1/README
import-2.1/items-to-tsdb.gawk
import-2.1/sets-to-tsdb.gawk
import-2.1/extract-embedded.gawk
import-2.1/common.gawk
```

The distribution is in `gzip(1)`ed `tar(1)` format and available from the TSNLP `ftp(1)` repository at `anonymous@tsnlp.dfki.uni-sb.de:/tsnlp/import/`.

Bibliography

[Elmasri and Navathe 1989] Ramez Elmasri and Shamkant B. Navathe. *Fundamentals of Database Systems.* Redwood City, California (The Benjamin/Cummings Publishing Company), 1989.

[Estival *et al.* 1994] Dominique Estival, Kirsten Falkedal, Sabine Lehmann (principal authors), Lorna Balkan, Siety Meijer, Doug Arnold, Sylvie Regnier-Prost, Eva Dauphin, Klaus Netter, and Stephan Oepen. Test Suite Design — Annotation Scheme. Report to LRE 62-089 D-WP2.2, University of Essex, UK, 1994.

[Estival *et al.* 1995] Dominique Estival, Kirsten Falkedal, Sabine Lehmann, Hervé Compagnion (principal authors), Lorna Balkan, Doug Arnold, Frederik Fouvry, Eva Dauphin, Sylvie Regnier-Prost, Véronika Lux, Judith Klein, Judith Baur, Klaus Netter, and Stephan Oepen. The Construction of Test Material. Report to LRE 62-089 D-WP3.1, University of Essex, UK, 1995.

[Lehmann *et al.* 1996] Sabine Lehmann, Dominique Estival, Kirsten Falkedal, Hervé Compagnion, Lorna Balkan, Frederik Fouvry, Judith Baur, and Judith Klein. TSNLP User Manual. Volume 3: Test Data Documentation. Technical report, Istituto Dalle Molle per gli Studii Semantici e Cognitivi (ISSCO) Geneva, Switzerland, 1996.

[Nerbonne *et al.* 1993] John Nerbonne, Klaus Netter, Kader Diagne, Ludwig Dickmann, and Judith Klein. A Diagnostic Tool for German Syntax. *Machine Translation*, 8 : 85–107, 1993.

[Oepen *et al.* 1996] Stephan Oepen, Klaus Netter, and Judith Klein. TSNLP — Test Suites for Natural Language Processing. In: John Nerbonne (editor), *Linguistic Databases*, CSLI Lecture Notes. Center for the Study of Language and Information, 1996. forthcoming.

[(principal author) *et al.* 1994] Dominique Estival (principal author), Kirsten Falkedal, Lorna Balkan, Siety Meijer, Sylvie Regnier-Prost, Klaus Netter, and Stephan Oepen. Survey of Existing Test Suites. Report to LRE 62-089 D-WP1, University of Essex, UK, 1994.

[Regnier-Prost *et al.* 1995] Sylvie Regnier-Prost, Eva Dauphin, Veronika Lux, Lorna Balkan, Frederik Fouvry, Kirsten Falkedal, Stephan Oepen (principal authors), Doug Arnold, Judith Klein, Klaus Netter, Dominique Estival, Kirsten Falkedal, and Sabine Lehmann. Testing and Customisation of Test Items. Report to LRE 62-089 D-WP4, University of Essex, UK, 1995.