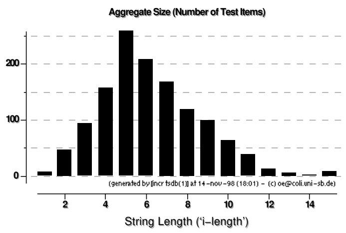
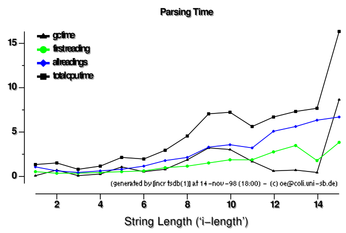


Phenomenon	total items #	positive items #	word string Ø	lexical items Ø	parser analyses Ø	total results #	overall coverage %
S_Types	235	180	6.53	28.03	1.86	125	69.4
C_Agreement	68	49	5.94	17.86	1.45	42	85.7
C_Complementation	179	108	5.52	20.27	2.97	98	90.7
C_Diathesis–Active	89	72	7.56	16.83	2.83	24	33.3
C_Diathesis–Passive	35	27	6.19	33.33	3.50	24	88.9
C_Tense–Aspect–Modality	83	79	5.77	23.34	1.89	70	88.6
C_Negation	58	44	5.30	17.45	1.52	40	90.9
C_Coordination	79	55	7.96	27.45	3.03	40	72.7
C_Modification	174	121	7.36	25.78	1.77	71	58.7
NP_Agreement	46	37	4.89	18.24	1.24	29	78.4
NP_Modification	83	71	7.03	24.87	1.61	57	80.3
NP_Coordination	55	28	6.07	14.68	1.32	22	78.6
Total	**1184**	**871**	**6.48**	**23.32**	**2.08**	**642**	**73.7**

(generated by [incr tsdb(1)] at 14–nov–98 (17:58) – (c) oe@coli.uni–sb.de)



[incr tsdb()]

Competence and Performance Laboratory

User & Reference Manual

Stephan Oepen

Computational Linguistics — Saarland University

Preface

> [...] we view the discovery of parsing strategies as a largely experimental process of incremental optimization. [Erbach (1991)]

> [...] the study and optimisation of unification-based parsing must rely on empirical data until complexity theory can more accurately predict the practical behaviour of such parsers. [...] It seems likely that implementational decisions and optimisations based on subtle properties of specific grammars can [...] be more important than worst-case complexity. [Carroll (1994)]

Contemporary lexicalized constraint-based grammars (e.g. within the HPSG framework) with wide grammatical and lexical coverage exhibit immense conceptual and computational complexity; as the grammatical framework aims to eliminate redundancy and factor out generalizations, the interaction of lexicon and phrase structure apparatus can be subtle and make it hard to predict how even modest changes to the grammar affect system behaviour. Additionally, in a distributed grammar engineering setup (i.e. for a project where several people or even sites contribute to a single grammatical resource) it becomes necessary to assess the impact of individual contributions, regularly evaluate the quality of the overall grammar, and compare it to previous versions.

Besides concise coverage (i.e. competence) judgments, in most application scenarios efficiency and resource consumption play an increasingly important role; hence, processing components typically provide a (potentially) large inventory of control parameters and preference settings. When tuning the analysis component to improve system performance, grammar writers often rely on introspection, knowledge of the grammar, and personal experience; yet, without systematic profiling and performance analysis, processor optimization amounts to guessing parameter settings and constant experimentation.

This user manual documents [incr tsdb()], an integrated package for diagnostics, evaluation, and benchmarking in practical grammar and system engineering. The software implements an approach to grammar development and system optimization that builds on precise empirical data and systematic experimentation as suggested by Erbach (1991) and Carroll (1994). [incr tsdb()] has been integrated with several contemporary HPSG development systems; the methodology and tools were designed for sufficient flexibility and generality to facilitate interfacing and adaption to other platforms. The [incr tsdb()] package is made available to the general public (see section 2.2 for details) in the hope that it may be useful to grammar and system developers and ultimately help in the study and comparison of salient grammar or processor properties across platforms. Developers are strongly encouraged to evaluate the package for connecting it to their systems; section 6.1 outlines various interface and daptation strategies. For advice and support please contact

Stephan Oepen
Computational Linguistics
Saarland University
Postfach 151140
66041 Saarbrücken (Germany)
`oe@coli.uni-sb.de`
(+49 681) - 302 41 76

The research and implementation of [incr tsdb()] has been carried out in close collaboration between Saarland University and CSLI Stanford over a period of several years. The

author is greatly indebted to Daniel P. Flickinger (CSLI) for the ever outstanding cooperation, continuous inspiration and support, and the friendship that has evolved from the enterprise. Numerous colleagues at the two institutions and their scientific vicinities have contributed to the approach through invaluable discussions and productive criticism. To name only a few, the feedback provided by John Carroll, Anne Copestake, Marius Groenendijk, Tibor Kiss, Sabine Lehmann, Rob Malouf, John Nerbonne, and Hans Uszkoreit has greatly influenced the current results.

The current [incr tsdb()] distribution contains code developed by Tom Fettig (co-developer of the tsdb database) and Oliver Plähn (co-developer of the table and graph widgets deployed in the [incr tsdb()] podium); many thanks for these contributions. Part of the research underlying the [incr tsdb()] package was funded by the German National Science Foundation (DFG) within the Special Research Divison 378 (*Resource-Adaptive Cognitive Processes*) project B4 (PERFORM), by the Commision of the European Union through the TSNLP project (LRE-62-089), by Anite Systems, Luxembourg (through a subcontract on integration with ALEP), and by the German Federal Ministry of Education, Science, Research, and Technology (BMB+F) in the framework of the Verb-Mobil project (FKZ:01IV7024). Additional funding was supplied by CSLI and DFKI Saarbrücken through travel support to the author.

Contents

1 Overview **1**
1.1 The Name of the Game: [incr tsdb()] 1
1.2 Structure of the Document 1

2 Installation and Startup **3**
2.1 Distribution Policy — Copyleft 3
2.2 Obtaining [incr tsdb()] 4
2.3 Installation & Registration 4
2.4 Loading and (Re)Starting [incr tsdb()] 5
2.5 Troubleshooting: Something Went Wrong 6

3 Profiling Terminology **7**

4 Sample Session **9**
4.1 First Time Preparation 9
4.2 Creating a Test Suite Instance 10
4.3 Browsing the Data 11
4.4 Obtaining a Competence and Performance Profile 13
4.5 Profile Analysis 14
4.6 Comparison to Earlier Test Runs 21
4.7 Zoom: In-Detail Profile Comparison and Analysis 25
4.8 Some Useful Switches 29
4.9 Recommendations for Future Experimentation 30

5 Reference Manual **31**
5.1 [incr tsdb()] Architecture 31
5.2 Contents of [incr tsdb()] Profiles 31
5.3 Storage and Reconstruction of Derivations 31
5.4 The Menu Structure 31
5.5 Visualization and Analysis of Profiles 31
5.6 Comparison among Profiles 31
5.7 Data Selection and Aggregation 31
5.8 TSQL syntax 31
5.9 Importing Data 31
5.10 Customization: '~/.podiumrc' and '~/.tsdbrc' 31
5.11 Options and Parameters 31
5.12 [incr tsdb()] Command-Line Interface 31
5.13 tsdb Database Format 31

6 Application Program Interface **33**
6.1 Connecting [incr tsdb()] to Another Processor 33
6.2 Parallel Virtual Machine 34
6.3 ANSI C Clients 36
6.4 Common-Lisp Clients 39
6.5 Using [incr tsdb()] Distributed Mode 39
6.6 Debugging [incr tsdb()] Distributed Mode 41

A Contents of the [incr tsdb()] Distribution **i**

References **iii**

1 Overview

This manual documents [incr tsdb()], an integrated package for diagnostics, evaluation, and benchmarking in practical grammar and system engineering. [incr tsdb()] builds on the following components and modules

- test and reference data stored with annotations in a structured database; annotations can range from minimal information (unique test item identifier, item origin, length et al.) to fine-grained linguistic classifications (e.g. regarding grammaticality and linguistic phenomena presented in an item) as represented by the TSNLP test suites (Lehmann et al. (1996));
- tools to browse the available data, identify suitable subsets and feed them through the analysis component of a processing system like LKB (Copestake (1992)), PAGE (Uszkoreit et al. (1994)), and others;
- the ability to gather a multitude of precise and fine-grained system performance measures (like the number of readings obtained per test item, various time and memory usage metrics, ambiguity and non-determinism parameters, and salient properties of the result structures) and store them as a *competence and performance profile*;
- graphical facilities to inspect the resulting profiles, analyze system competence (i.e. grammatical coverage and overgeneration) and performance at highly variable granularities, aggregate, correlate, and visualize the data, and compare profiles obtained from previous grammar or system versions or other platforms.

1.1 The Name of the Game: [incr tsdb()]

As the [incr tsdb()] package has some (partly historic) internal structure (see figure 5.1 for a sketch of the system architecture), so has its name. The data and profiles are stored in tsdb, a simple and portable relational database system (Oepen et al. (1997)) that grew out of the TSNLP project; the interfaces to LKB and PAGE and the bulk of the profiling and analysis routines are implemented in Common-Lisp: here, `tsdb()` is the central function call in the interface to the processors; finally, the graphical user interface and visualization components build on the Tk widget library (Ousterhout (1994)) and the Tcl command interpreter where '[incr]' is the Tcl equivalent of the C increment operator '++'.

In short, [incr tsdb()] is a hybrid collection of individual bits and pieces, and so is the name of the package; fortunately, there is a unique, simple, and universal pronounciation for it

tee ess dee bee plus plus[1]

Possibly, some people will find the name ugly, obscure, unpronounceable, meaningless, or really really geeky ... alas, it is the way it is. We hope they will like the software better.

1.2 Structure of the Document

The following sections describe (i) how to obtain, install, and start [incr tsdb()], (ii) present the core functionality by walking trough a sample session, and (iii) detail the facilities of the [incr tsdb()] user interface as a reference manual for experienced users.

[1]Many thanks to Rob Malouf of CSLI Stanford for the transliteration and detailed comments on earlier working titles for the [incr tsdb()] package deal.

2 Installation and Startup

The following sections detail the (i) conditions under which [incr tsdb()] is made available, (ii) how to obtain and (iii) install (and register) the package, and (iv) the basic loading and start-up commands for LKB and PAGE. Distribution details are likely to change in the near future when the package and up-to-date information become available for InterNet download.

2.1 Distribution Policy — Copyleft

Aiming for a commonly-accepted (pre-standard) diagnostic and evaluation methodology and technology, wide dissemination and assessment of the [incr tsdb()] package is most desirable. Thus, the software (in source code) and data are made available to the general public, free of royalties, for academic or other non-commercial use, including deployment in corporate environments. Though there is no principled obstacle to license commercial use of the package, prior consultation with the author and a written license agreement will be required.

All copyright and intellectual property rights remain with the author. To provide feedback on the distribution of the package, [incr tsdb()] users are encouraged to register (see below) and relay comments, bug reports or suggestion for improvement to the contact address given in the section PREFACE above. While unregistered, the package is fully functional but, *after several minutes of continuous use, a log message is generated and sent to a central protocol server* (at Saarbrücken University). This notification message only contains information about the version of [incr tsdb()] used and the name of the machine it is run on.[2] Registering the software will effectively suppress all message generation for all future uses.

Users of [incr tsdb()] are welcome to incorporate parts or all of the data and tools into applications or programs of their own and to modify, copy, and further distribute these as long as this license is preserved in its original form. It is strongly encouraged to contact the author for notifications of changes or extensions to the package and to aim for integration with the standard release and public distribution for any substantial additions made.

The author explicitly declines any warranty or liability for [incr tsdb()]. In particular, please note that:

> BECAUSE THE PROGRAM IS LICENSED FREE OF CHARGE, THERE IS NO WARRANTY FOR THE PROGRAM, TO THE EXTENT PERMITTED BY APPLICABLE LAW. EXCEPT WHEN OTHERWISE STATED IN WRITING THE COPYRIGHT HOLDERS AND/OR OTHER PARTIES PROVIDE THE PROGRAM AS IS WITHOUT WARRANTY OF ANY KIND, EITHER EXPRESSED OR IMPLIED, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE. THE ENTIRE RISK AS TO THE QUALITY AND PERFORMANCE OF THE PROGRAM IS WITH YOU.
>
> SHOULD THE PROGRAM PROVE DEFECTIVE, YOU ASSUME THE COST OF ALL NECESSARY SERVICING, REPAIR OR CORRECTION. IN NO EVENT, UNLESS REQUIRED BY APPLICABLE LAW OR AGREED TO IN WRITING, WILL ANY

[2] However, please note that because of the transmission protocol used over the InterNet, additional information becomes available to the registration server, viz. the time of [incr tsdb()] usage (when the message is generated) and the account used (the originator of the protocol message).

2.2 Obtaining [incr tsdb()]

As of January 1999, the [incr tsdb()] package is not yet available for public download over the InterNet. However, the entire package (all source code and data) is bundled with current distributions of the LKB and PAGE grammar engineering platforms.[3] Hence, in more recent installations of either system, [incr tsdb()] should be readily available.

While there is no InterNet address for free download of [incr tsdb()], the author will make the package available to interested parties upon request. See the PREFACE section earlier in this document for the contact address.

2.3 Installation & Registration

As of June 1999, [incr tsdb()] is only available for the Franz Allegro Common-Lisp (4.3 or 5.0) versions of LKB and PAGE and will only work on the Solaris (Sparc), Linux (x86), and OSF (DEC Alpha) platforms. Assuming a functional LKB or PAGE installation, no additional work should be required for the [incr tsdb()] package. Depending on which distribution was obtained (source or ready-to-run binary), it may be necessary to compile [incr tsdb()] source files at first-time use; the regular load and start-up procedure (section 2.4 below), however, will automatically compile files where necessary.

As part of the installation or at a later time (e.g. once you found the package to be useful for your purposes), it is recommended that you register with the author. Registration is not intended to restrict distribution or application of the software, to charge users a license fee, or turn a profit from the sale of a large customer database. The main and only purpose of voluntary registration is to provide feedback to the author and to allow (occasional) relaying of information to the actual users; unless otherwise requested, registered users will be added to the [incr tsdb()] maling list. The maling list is used by the author to announce major version upgrades as they become available; it is very low traffic.

Registration proceeds as follows

(i) make yourself known to the author: send email to the contact address containing information about the user name(s) and machine name(s) that you want to register; if you want to register for an entire InterNet domain (i.e. a wildcard machine name) or a group of users (i.e. all accounts for some host or domain), please give an estimate of the expected size of either set;

[3] For access to and download instructions for LKB and PAGE see

- `http://www-csli.stanford.edu/ aac/lkb.html` and
- `http://www.dfki.de/lt/systems/page/`

respectively.

(ii) receive one or multiple [incr tsdb()] license keys for your site by email;

(iii) add the license key(s), in exactly the same format as they were received, to the file '`Keys`' in the '`src/tsdb`' subdirectory of your installation; e.g.

```
*       *.coli.uni-sb.de    OE1234567890
stefan  *.dfki.uni-sb.de    OE1234567890
uc      *.dfki.uni-sb.de    OE1234567890
*       eo.stanford.edu     OE1234567890
*       eoan.stanford.edu   OE1234567890
malouf  gerund              OE1234567890
aac     anstac              OE1234567890
```

alternatively, e.g. if you have no write access to the [incr tsdb()] source tree at your site, you can set the license key from the file '`~/.podiumrc`' (see section 5.10) in your home directory; e.g.

```
set globals(copyleft,key) OE1234567890;
```

On start-up, the main [incr tsdb()] interaction window will display a copyright message that reflects the status of the current copy; it will either display a message like

— Registered Copy [OE1234567890] —

confirming the registration and the license key currently in use, or notify the user about the pending protocol message that will be generated after several minutes of continuous use (see section 2.1 above).

2.4 Loading and (Re)Starting [incr tsdb()]

For source distributions, once LKB or PAGE have been successfully compiled and loaded, evaluate the following form at the Common-Lisp prompt:[4]

```
(load-system "tsdb")
```

which should load the [incr tsdb()] Common-Lisp code (from the '`src/tsdb/lisp`' subdirectory of the source tree) into the current environment; if necessary, the system will suggest to (re)compile some or all of the source files first and ask for confirmation.[5] Once loading has completed, the [incr tsdb()] main interaction window — called the [incr tsdb()]

[4] If you find yourself using the [incr tsdb()] package regularly, you can in fact use the same (single) command to load both the host platform and the [incr tsdb()] code; thus, `(load-system "tsdb")` will automatically trigger the `(load-system "lkb")` (or equivalent for PAGE) that is usually required to load the host platform from a source distribution.

[5] Should the compilation and loading of the [incr tsdb()] Common-Lisp code fail, this usually indicates a problem in the installation of the underlying host platform (i.e. LKB or PAGE); see the appropriate documentation (if available) and check section 2.5 below.

podium (see figure 1 for a screen shot) — will be displayed. From here on continue through the sample session sketched in section 4 below.

With a binary distribution, on the other hand, the [incr tsdb()] Lisp code should already be included in the ready-to-run image used to start either LKB or PAGE; no additional loading will be required. To create the [incr tsdb()] podium window, evaluate the following Common-Lisp form:

```
(tsdb:tsdb :podium)
```

The same command can be used to shutdown a running [incr tsdb()] instance and create a fresh interaction window (e.g. to recover from a system error or a misbehaving [incr tsdb()] status; please remember to submit problem reports where unexpected behaviour occurs).

2.5 Troubleshooting: Something Went Wrong

3 Profiling Terminology

This section introduces some basic terminology that will be used throughout the discussion of the [incr tsdb()] approach.

Host Platform Earlier versions of [incr tsdb()] were always embedded into Lisp-based grammar development environments (viz. the LKB or PAGE systems); in this setup, the underlying grammar development and processing environment (sharing the same Lisp universe with [incr tsdb()]) is referred to as the *host platform*. Given the recent addition of a generic C application program interface (see section 6), the [incr tsdb()] package can now be run as a stand-alone application that communicates with client processes through a distributed inter-process communication mechanism (section 6.2). As of May 1999, however, the embedded setup still is the default solution for Lisp-based host platforms.

Test Item The basis elements used in testing and diagnosis are called *test items*; all [incr tsdb()] test data — classical test suites and corpora extracts alike — are structured as a sequence of test items, typically sentences or other types of phrases. A test item, typically, is composed of a string, used as input to the processor, together with linguistic and non-linguistic information associated with each test item (so-called annotations). The minimal annotations per test item are (i) a unique test item identifier, (ii) the item length (in words), and (iii) a grammaticality (or wellformedness) code. Other types of annotations, e.g. in the TSNLP test suites, can include the syntactic categories of test items, the ssociation with one or more syntactic phenomena, and a (sometimes partial) description of tectogrammatical structure. When importing test items from ASCII files (see section 5.9), [incr tsdb()] will automatically generate this minimal level of item annotation.

Test Suite Traditionally, the term *test suite* is used for hand-crafted sets of test data; within the [incr tsdb()] context this notion is generalized somewhat to include both manually-constructed data sets (typically containing systematically chosen grammatical as well as ungrammatical test items and aiming to present distinct phenomena in isolation or controlled interaction) and sequences of test items extracted from actual text corpora (typically excluding negative test items but demonstrating a richer combination of phenomena interaction and ambiguity). [incr tsdb()] makes a (technical) test suite subdivision according to processing state: see the discussion of test suite skeletons vs. instances below.

Test Suite Skeleton Analogous to the material vs. data distinction assumed in many experimental paradigms, [incr tsdb()] reserves the term *test suite skeleton* to refer to pure collections of test material, i.e. sets of test items and associated annotations, that have *not* (yet) been enriched with processing results (or data, in this respect). Test suite skeletons are stored as partial databases that contain all-empty relations for those parts of the database schema (see section 5.2 for details) that are used for application-specific parameters obtained from a test run (see below). Test suite skeletons are read-only databases that (without profound [incr tsdb()] knowledge) cannot be modified.

Test Suite Instance When preparing for a test run (see below), one of the available test suite skeletons is selected and subsequently instantiated to yield a new *test suite* *instance*. The [incr tsdb()] podium body (see section 4) displays the list of available test suite instances (sometimes called the current working set) together with size and status information. Right after the creation of a new test suite instance, the database contains all information copied from the test suite skeleton that was instantiated and is then available to store new data obtained by processing the test material. To simplify data organization and result analysis, [incr tsdb()] typically assumes a one to one correspondence between test runs and test suite instances; when processing an already existing test suite instance, by default all non-skeleton data will be deleted.

Test Run The process of batch processing a set of test items, obtaining competence and performance parameters for the application system used, and storing these results into the active test suite instance is referred to as a *test run*. Each test run is described in terms of the environment used for processing (like the application system and grammar versions employed, size of grammar and lexicon, current user and machine, start and end time, and others) and can have a descriptive comment associated with it. The completion of a test run fills in the system-specific sections in a test suite instance (i.e. the '*run*', '*parse*', and '*result*' relations).

(Compentence & Performance) Profile Complete test suite instances that have been enriched with competence and performance information for a token processing system (and grammar version) are frequently described as *competence & performance profiles*. While, technically, profiles are just test suite instances, i.e. databases in the current working set, the specialized term emphasizes the diagnostic nature of the data: within the [incr tsdb()] approach competence & performance profiles are the fundamental building blocks for the in-depth, empirical analysis and comparison of various processing systems and strategies. Each [incr tsdb()] profile is a rich, accurate, and structured snapshot of relevant competence and performance properties of a token processor and grammar version.

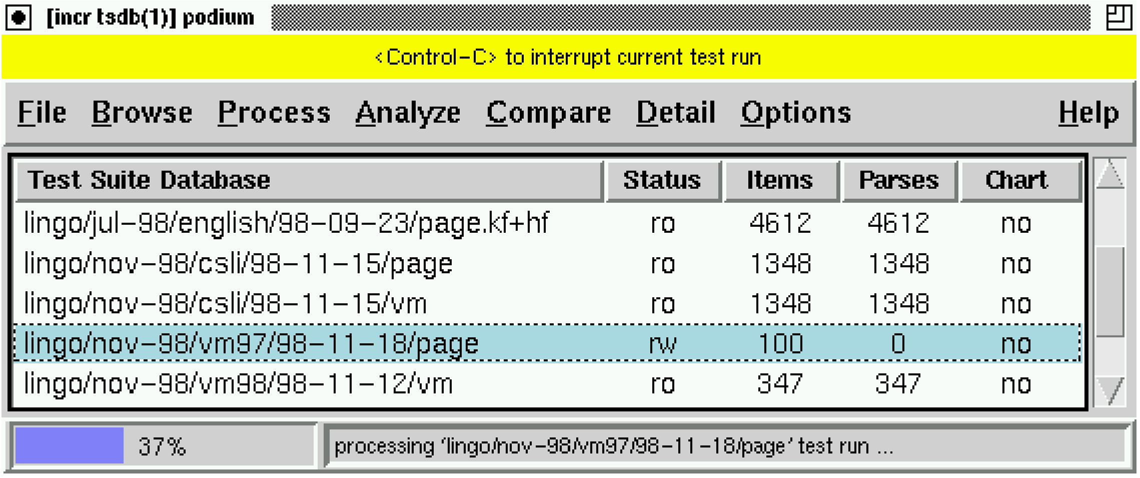


Figure 1: Screenshot of [incr tsdb()] podium: (i) the horizontal top area displays context-dependent (balloon-type) help; (ii) the menu structure reflects the prototypical profiling sequence: browsing the available data, processing and analysis of individual profiles, and comparison and in-detail study among profiles; (iii) the podium body lists all currently available profiles and their key properties; (iv) the progress meter gives an estimate of the remaining work to complete the current task; finally, (v) the lower left minibuffer serves for status messages and parameter input; when idle, the progress meter displays a digital clock.

4 Sample Session

Following is a detailed, step-by-step discussion of a sample session; [incr tsdb()] is used to (i) create a new test suite instance (see section 3 for some key terminology), (ii) inspect the available test data, (iii) batch process a set of test items, (iv) inspect the resulting profile, and (v) compare it to results obtained from a previous grammar version.

The examples assume a running [incr tsdb()] podium (in a state similar to what is displayed in figure 1) and a processor (either LKB or PAGE) with a suitable grammar loaded. The individual steps (in the sequence given) can be taken as a guided tour of the machinery; all data sets used in the presentation are included with the distribution, so that the sample tables and graphs can be reproduced and serve as a basis for individual experimentation.

4.1 First Time Preparation

Unless the user running [incr tsdb()] is, at the same time, the owner of the source tree (for the host platform and [incr tsdb()]), the package requires a dedicated directory that is used to store test suite instances and profiles. The default installation comes with a central profile repository that contains a few example data sets for common reference; but the central directory will typically not allow write access (i.e. creation of new directories and files) by non-privileged users. Thus, as soon as an [incr tsdb()] user wants to create a new test suite instance and obtain a current profile — as will be demonstrated in this sample session — she has to designate a user-writable directory (e.g. a subdirectory '`tsdb`' in the user home directory) for profile storage. After creation of the directory (e.g. using

Un∗x `mkdir(1)`), [incr tsdb()] has to be informed of the location: the 'Options – Database Root' command pops up a directory input dialogue in the podium minibuffer, e.g.

```
database root: ~/tsdb/
```

The minibuffer input dialogue provides `emacs(1)`-style context-sensitive completion — directory completion in this case — using the [Tab] key: hitting [Tab] (once) completes the current input as long as there is an unambiguous common prefix for the current set of alternatives or displays the list of choices in the podium body; directories are completed including a trailing '/' (see above) to simplify the validation of the value entered. [Return] completes the directory input and makes [incr tsdb()] search the specified location for existing profiles; the status message

```
obtaining tsdb(1) database list ...
```

is displayed in the minibuffer while the file system is inspected. Obviously, for a newly created directory the list of available profiles will at first be empty.

4.2 Creating a Test Suite Instance

The 'File – Create' menu (actually, the menu cascade that pops up when the 'Create' entry is selected from the 'File' menu) displays the current set of available test suite skeletons, their names and size in test items. Assuming a default installation with English skeletons activated, the list should be something like

Aged VerbMobil Data ('vm')	96 items
CSLI (LinGO) Test Suite ('csli')	1348 items
Development Test Suite ('toy')	26 items
TSNLP Test Suite ('english')	4612 items
VerbMobil 97 ('vm97')	100 items
VerbMobil 97 (Partials) ('vm97p')	252 items
VerbMobil 98 ('vm98')	347 items

Selecting one of the 'File – Create' entries, the 'CSLI (LinGO) Test Suite' say, pops up an input dialogue in the minibuffer

```
create: lingo/nov-98/csli/98-11-20/page
```

that prompts for the name for the new test suite instance to be created. The name suggested by [incr tsdb()] typically will contain the following path components (section 5.10 shows how to customize the system suggestion)

$\langle grammar\rangle$/$\langle version\rangle$/$\langle skeleton\rangle$/$\langle date\rangle$/$\langle processor\rangle$

where ⟨*grammar*⟩ and ⟨*version*⟩ (if applicable) are taken from the value of the (Common-Lisp) variable `*grammar-version*`;[6] ⟨*skeleton*⟩ is the short name of the test suite skeleton used (the name given in parenthesis in the 'File – Create' list); and ⟨*processor*⟩ identifies the current host platform (e.g. LKB or PAGE).

The minibuffer input dialogue allows `emacs(1)`-style editing of the name suggested by [incr tsdb()]: Control-G aborts the current task, Return completes the input and starts the creation of a new test suite instance. After a few seconds, the new name is inserted into the list of available test suite instances in the podium body (sorted by lexicographic order); the status for a fresh test suite instance is '`rw`' (read-write), the number of items as in the skeleton used (i.e. 1348 for the current example), and the number of parses 0 for an empty profile. The new test suite instance is selected (made active) after creation. The current selection is indicated by highlighting the complete entry in the podium list; there can be at most one active test suite instance at any given time.

4.3 Browsing the Data

Before starting a time-consuming test run, it can be desirable to inspect the avaiable test items from the active test suite instance.

The 'Browse – Vocabulary' command displays a sorted list of vocabulary (i.e. word forms) used in the test items together with the frequencies of occurence. The printed output goes to the window that was used to start the [incr tsdb()] podium (i.e. the LKB or PAGE Lisp listener).

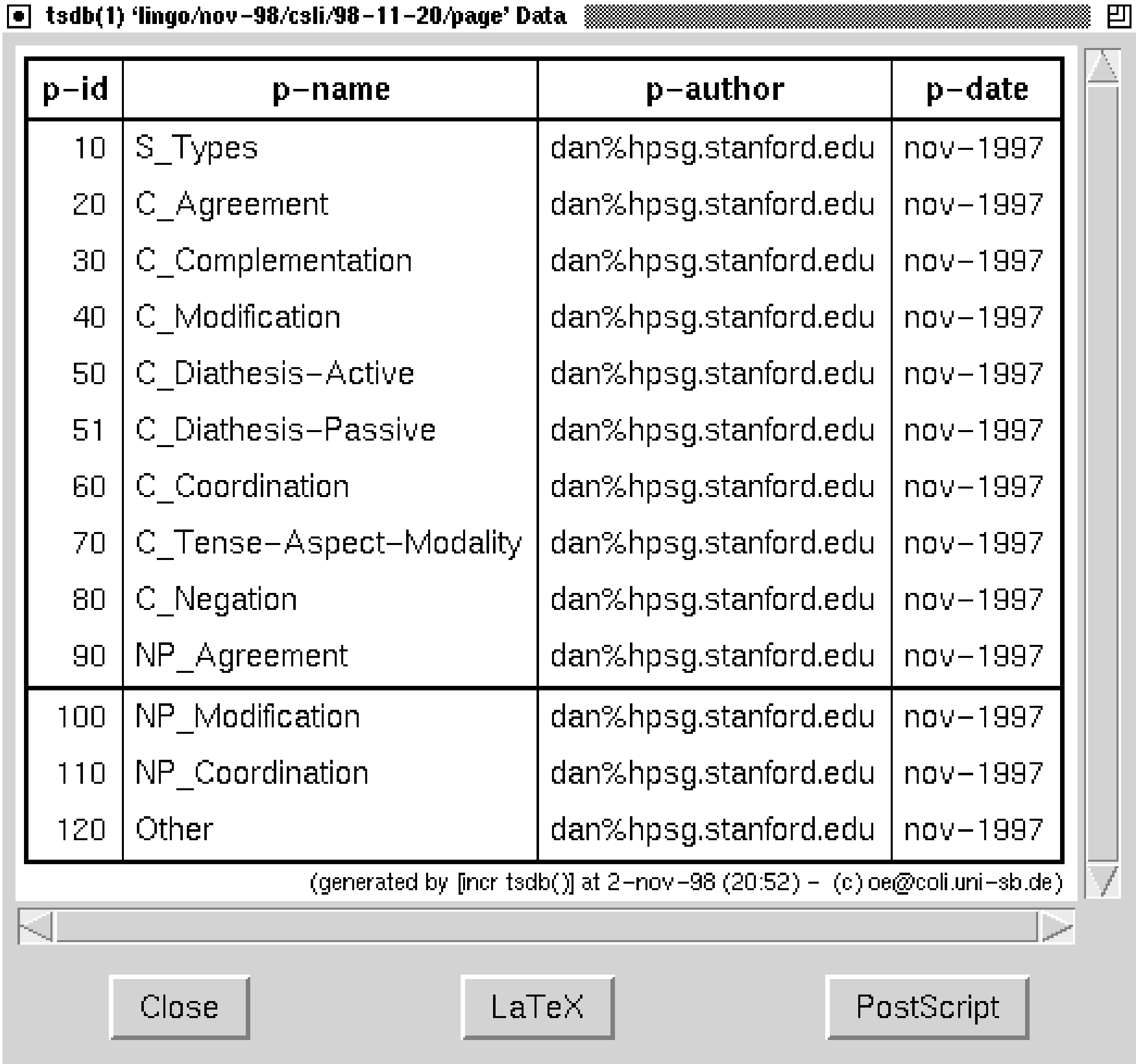
The 'Browse – Test Items' and 'Browse – Phenomena' menus can be used to view (part of) the raw data as a table presenting a selection of database fields (attributes) for all or a subset of the records from the active test suite instance. Since computing the table layout and geometry for larger databases can take substantially more time than it should (i.e. from a few seconds to around one minute on an average cpu), it is in general wise to restrict the selection of the data to what is actually needed.

A common way to select subsets of test items is by means of a classification of gross syntactic phenomena. Both the CSLI and TSNLP test suites deploy the classification scheme developed in the TSNLP project (see Lehmann et al. 1996 for details); 'Browse – Phenomena' on the CSLI data set yields the display shown in figure 2. The 'Browse – Test Items' menu, in turn, allows a selection from all available test items by phenomenon: 'Browse – Test Items – C_Complementation', for example, displays all test items, their unique identifiers, actual input string, wellformedness code (where 1 is grammatical and 0 ungrammatical), and root category as follows:

[6] The value of `*grammar-version*` typically is determined from the matrix load file for a token grammar; for the November 1998 version of the LinGO ERG, for example, the files '`script`' (LKB version) and '`english.tdl`' (PAGE version) contain the statement

```
(setf *grammar-version* "LinGO (nov-98)")
```

that results in values for ⟨*grammar*⟩ and ⟨*version*⟩ as used in the example.



p-id	p-name	p-author	p-date
10	S_Types	dan%hpsg.stanford.edu	nov-1997
20	C_Agreement	dan%hpsg.stanford.edu	nov-1997
30	C_Complementation	dan%hpsg.stanford.edu	nov-1997
40	C_Modification	dan%hpsg.stanford.edu	nov-1997
50	C_Diathesis-Active	dan%hpsg.stanford.edu	nov-1997
51	C_Diathesis-Passive	dan%hpsg.stanford.edu	nov-1997
60	C_Coordination	dan%hpsg.stanford.edu	nov-1997
70	C_Tense-Aspect-Modality	dan%hpsg.stanford.edu	nov-1997
80	C_Negation	dan%hpsg.stanford.edu	nov-1997
90	NP_Agreement	dan%hpsg.stanford.edu	nov-1997
100	NP_Modification	dan%hpsg.stanford.edu	nov-1997
110	NP_Coordination	dan%hpsg.stanford.edu	nov-1997
120	Other	dan%hpsg.stanford.edu	nov-1997

Figure 2: Display resulting from the 'Browse – Phenomena' command on an instance of the CSLI test suite: the list of core syntactic phenomena — ranging from sentence types over complementation, agreement, and modification phenomena to negation and coordination — is used in classifying test items according to the phenomena they present.

i-id	i-input	i-wf	i-category
1	Abrams works .	1	S
2	Abrams hired Browne .	1	S
3	Abrams showed the office to Browne .	1	S
4	Abrams showed Browne the office .	1	S
5	Abrams bet Browne five dollars that Chiang hired Devito .	1	S
6	Abrams became competent .	1	S
7	Abrams became a manager .	1	S
8	Abrams became in the office .	0	S
9	Abrams became working .	0	S
10	Abrams is interviewing an applicant .	1	S
⋮	⋮	⋮	⋮

Other means to browse the data selectively and identify meaningful subsets are the 'Browse – Custom Query' and 'Options – TSQL Condition' menues; they will be presented by example in section 4.7 below.

From the 'Browse – Test Items' table (and all similar tables that contain the *i-id* and *i-input* fields) it is possible to feed individual test items to the processor (e.g. to verify the parser well-functioning) by double-clicking any mouse button on the *i-input* field. Interactive processing will *not* disable the trace and result displays or write new data to the currently active profile (see below).

4.4 Obtaining a Competence and Performance Profile

While most of the [incr tsdb()] functionality is independent of the underlying platform (and, in fact, can be loaded and used without either LKB or PAGE), processing a set of test items and obtaining a new competence and performance profile, obviously, presupposes that the host platform is configured and fully operational (i.e. can parse sentences interactively and produces the expected result).

As for browsing the data (see above), it may seem desirable to save processing time by restricting a test run to a subset of the test items available in a token test suite instance. However, since all profiles are stored as a database and made available for future reference, it is typically desirable to obtain complete profiles that contain information for parsing all test items. Thus, all instances of the same test suite skeleton will always be mutually comparable to each other.

Yet, under certain conditions one may want to restrict the parser to a non-exhaustive search strategy by means of the 'Options – Switches – Exhaustive Search' switch. For PAGE at least, non-exhaustive parsing means that the parser stops searching for solutions when the first reading is found;[7] for test data where the grammar assigns at least one reading to most test items (i.e. it achieves a good coverage rate), the non-exhaustive mode will result in a significant reduction of procesing time. At the same time, again, obtaining a profile through non-exhaustive parsing limits interpretation and comparison of the data (e.g. there is no information on global ambiguity and the overall processing times are skewed); hence, non-exhaustive profiles should be marked explicitly, for example by the name chosen for the test suite instance and the descriptive comment associated with the test run.

[7] The LKB parser (as of November 1998) uses a mostly breadth-first parsing strategy that allows no meaningful distinction between exhaustive and non-exhaustive parsing. Hence, in the LKB system the times reported for finding the first reading and finding all readings should always be the same.

Assuming the default [incr tsdb()] settings and '`lingo/nov-98/csli/98-11-20/page`' (from above) as the active test suite instance, the command 'Process – All Items' will

(i) prompt for a descriptive comment (the purpose of this test run, say, or an unusual aspect of the current setup) in the minibuffer; this comment and some additional information about the current condition of the host platform will be stored together with the actual parsing results into the resulting profile;

(ii) delete (purge) all existing parses (none for the present example) from the active test suite instance (but see the documentation of 'Options – Switches – Overwrite Test Run' in section 5.4);

(iii) make the processor load the vocabulary required for parsing the test data (see 'Options – Switches – Autoload Vocabulary') and print a lexicographically sorted listing of the number of lexical items retrieved and successful lexical rule applications per input word (see figure 3);

(iv) put the processor into batch parsing mode (all graphical display for parsing results is disabled) and install the selected garbage collection strategy (see 'Options – Switches – Enable Tenuring' and others);

(v) feed the test data, one item at at a time, through the parser, gather a large number of processing metrics from the host platform (see figure 3 and section 5.2 for details), and store the results into the profile database.

While processing vocabulary and test items, the progress meter indicates the percentage of work already completed (as shown in figure 1 above); upon completion of the test run, the listing of test suite instances is updated to reflect the change in the number of parses in the active profile. The database that now contains both the test data from the test suite skeleton plus the overall test run information and individual processing results for each test item constitutes a new competence and performance profile, a large pool of structured information that is now ready for inspection.

Despite all reservations to profiles obtained from partial test runs expressed earlier, other commands from the 'Process' menu allow a selection of a subset of the available data to be processed (viz. the 'Positive Items', 'Negative Items', and 'TSQL condition' commands, of which the first two have the obvious effect while the latter prompts for a TSQL condition to be used in constraining the input data; see section 5.8 for the precise syntax used). 'Process – Vocabulary', finally, makes the host platform load the necessary vocabulary without triggering an actual test run.

4.5 Profile Analysis

The [incr tsdb()] podium offers a number of pre-configured views on a data set. Additional configuration options allow the flexible adjustment of the analysis perspective and granularity; additionally, [incr tsdb()] provides tools to visualize processing results on a per-item basis. All analysis commands use the currently active test suite instance (the one highlighted in the podium body) as the base data set; the selection can be changed by (single) clicking the left mouse button on another profile.

The most common queries to a profile are implemented through the 'Coverage', 'Overgeneration', and 'Performance' commands in the 'Analyze' menu. Either of the three commands will select the neccesary data from the selected test suite instance, compute the

```
retrieve(): found 1348 items.
merge-with-output-specifications(): found 0 output specifications.

  !            |   23 reference(s) | [0 + 0] lexical entrie(s);
  (            |    6 reference(s) | [0 + 0] lexical entrie(s);
  )            |    6 reference(s) | [0 + 0] lexical entrie(s);

  ...

  would        |   28 reference(s) | [1 + 4] lexical entrie(s);
  years        |    1 reference(s) | [1 + 0] lexical entrie(s);
  you          |   23 reference(s) | [1 + 0] lexical entrie(s);
  yourselves   |    2 reference(s) | [1 + 0] lexical entrie(s);

largest-run-id(): largest ‘run-id’ is 0.
retrieve(): found 1348 items.
merge-with-output-specifications(): found 0 output specifications.
create-cache(): tsdb(1) write cache in ‘/tmp/.tsdb.cache.oe.60890’.
install-gc-strategy(): disabling tenure; global garbage collection ... done.
largest-parse-id(): largest ‘parse-id’ (for ‘run-id’ 1) is 0.
(1) ‘Abrams works .’ [0] --- 1 (1.1|0.4:1.0 s) <5:38> (13.8M) [0].
(2) ‘Abrams hired Browne .’ [0] --- 1 (0.8|0.3:0.6 s) <5:63> (6.4M) [0].
(3) ‘Abrams showed the office to Browne .’ [0] --- 1 (5.1:0.5|2.2:3.9 s) <24:218> (45.1M) [0].
(4) ‘Abrams showed Browne the office .’ [0] --- 1 (1.8|0.5:1.4 s) <16:177> (8.3M) [0].

...

(1347) ‘The person to know is Kim .’ [0] --- 5 (2.7|0.8:2.2 s) <18:267> (10.4M) [0].
(1348) ‘*The person whether to know is Kim .’ [0] --- (2.5|2.0 s) <20:202> (9.5M) [0].
flush-cache(): tsdb(1) cache for ‘lingo/nov-98/csli/98-11-20/pageq’ flushed.
```

Figure 3: Excerpt of printout produced from the ‘Process – All Items’ command. After loading and expansion of the necessary lexical entries, the parser log format aims to give a compact summary of some of the information gathered; besides the item identifier, input string and an upper limit for chart edges (given in square brackets) if available, the information following the triple dash is: the number of readings obtained, the time used to find the first reading and overall (exhaustive search) processing time (in parentheses) the number of lexical items involved and total number of edges in the chart (angle brackets), the amount of memory used, and the number of (global) garbage collections while parsing (square brackets).

Phenomenon	total items #	positive items #	word string Ø	lexical items Ø	parser analyses Ø	total results #	overall coverage %
S_Types	235	180	6.53	17.92	2.50	126	70.0
C_Agreement	68	49	5.94	12.78	1.67	42	85.7
C_Complementation	179	108	5.52	14.27	2.27	100	92.6
C_Diathesis–Active	89	72	7.56	17.90	2.72	25	34.7
C_Diathesis–Passive	35	27	6.19	21.11	3.04	23	85.2
C_Tense–Aspect–Modality	83	79	5.77	13.94	2.29	72	91.1
C_Negation	58	44	5.30	9.80	2.10	39	88.6
C_Coordination	79	55	7.96	18.47	3.42	40	72.7
C_Modification	174	121	7.36	16.04	2.82	89	73.6
NP_Agreement	46	37	4.89	10.46	1.62	29	78.4
NP_Modification	83	71	7.03	15.54	2.47	62	87.3
NP_Coordination	55	28	6.07	11.29	3.42	24	85.7
Total	**1184**	**871**	**6.48**	**15.55**	**2.48**	**671**	**77.0**

(generated by [incr tsdb()] at 26–nov–98 (19:03) – (c) oe@coli.uni–sb.de)

Figure 4: Coverage Profile for the LinGO ERG on an instance of the CSLI test suite. Columns are (from left to right): TSNLP phenomenon name, total number of test items, number of grammatical test items, average test item length, average number of lexical entries per test item, average number of readings per test item, total number of test items successfully parsed, and percentage of grammatical items parsed; comparing columns 4 and 5 provides a measure of lexical ambiguity, while column 6 indicates syntactic ambiguity; for example, passive test items exhibit significant lexical ambiguity because of multiple lexical entries for the copula and the passive participle; the latter also contributes to the higher measure of syntactic ambiguity for passives.

requested information, and present it in a new window. Most of the [incr tsdb()] analysis windows have several export buttons (e.g. labeled (PostScript) and (LaTeX)) that allow the output of the current view into a file of the requested format.

The coverage and performance views are presented in figures 4 and 5, as PostScript and LaTeX output respectively (see the table captions for a description of the individual columns); the overgeneration view is the mirror image of the coverage summary in that it uses the test items marked as ungrammatical (instead of the grammatical test items) as the base set.

In general, a condensed summary of a profile as presented by the 'Analyze' menu often already presents salient properties of a token test run and points the developer to further (typically more focused and in-depth) analysis. While the coverage and overgeneration views mostly aim to summarize properties of the grammar used (i.e. competence and ambiguity measures), the performance summary has its focus on system behaviour, viz. on resource consumption and parser efficiency. Yet, it may still be desirable for a grammar

'lingo/nov-98/csli/98-11-20/page'									
Aggregate	**items** ♯	**etasks** ϕ	**filter** %	**edges** ϕ	**first** ϕ (s)	**total** ϕ (s)	**tcpu** ϕ (s)	**tgc** ϕ (s)	**space** ϕ (kb)
$20 \leq$ *i-length* < 25	3	32482	60.0	2520	7.60	24.21	40.40	14.13	97425
$15 \leq$ *i-length* < 20	10	21992	52.5	1449	6.60	14.63	20.29	3.83	64918
$10 \leq$ *i-length* < 15	121	8483	52.4	591	2.22	5.46	7.06	0.60	24778
$5 \leq$ *i-length* < 10	853	3194	57.9	279	0.93	2.21	3.04	0.32	11511
$0 \leq$ *i-length* < 5	303	1209	61.0	116	0.30	0.86	1.41	0.27	5099
Total	**1290**	**3437**	**56.8**	**284**	**0.96**	**2.34**	**3.26**	**0.40**	**11863**

(generated by [incr tsdb(1)] at 26-nov-1998 (19:06 h) – (c) oe@coli.uni-sb.de)

Figure 5: Performance Profile for the LinGO ERG on an instance of the CSLI test suite. Columns are (from left to right): the aggregation criterion (e.g. test items with a length of 20 to 25 words for the top row), the number of test items per aggregate, the average number of parser actions executed, the average percentage of potential parser actions filtered (i.e. not executed), the average number of edges built (active plus passive, where applicable), the average times to find the first reading and all readings (in seconds), average total cpu time used and share of garbage collection (gc) time, and the average amount of memory allocated (in kbytes) while parsing; the aggregation scheme chosen in the example indicates a strong correlation between the input sentence length and the amount of work done in the parser (e.g. when quantified by the number of tasks executed or the total time and space requirements) — as should be expected.

writer to inspect the performance profiles regularly, as, for example, an underspecified rule may well create large numbers of spurious edges (local ambiguity) without any noticeable increase in the number of readings obtained (global ambiguity). Similarly, system developers that take a token grammar (and its properties) as given could still use the competence profiles to gauge ambiguity in dependency to the average input length, for example, as they are tuning the system to reduce lexical ambiguity (by use of a suitable filter, say).

For all three (tabular) 'Analyze' summaries, the aggregation scheme can be adjusted dynamically. While aggregation by phenomenon classification (if available) is chosen by default, the 'Options – Aggregate By' menu provides a choice of aggregate computation by several relevant properties of the data set, such as, for example, the item grammaticality or string length, the degree of lexical or phrasal ambiguity, parsing complexity metrics, or time or space requirements. In addition, the 'Options – Aggregation Parameters' dialogue allows the adjustment of the following parameters:

- aggregate size the width of each aggregate interval, i.e. the number of units along the current aggregation dimension that fall into a single class (default value is 1);
- aggregate threshold the minimum number of elements per aggregate: aggregates that have fewer members (sparse data) will be supressed (default 1);
- lower bound the lower aggregation boundary used: data points with a value (along the current dimension) below this limit will be ignored (default 0);
- upper bound the upper aggregation boundary used: data points above this limit will be ignored; the upper bound parameter can be left empty (unset) to represent infinity (i.e. no upper bound, the default).

The 'Aggregation Parameters' dialogue pops up in the minibuffer as follows

size 5 threshold 1 lower 0 upper

Tab and Shift-Tab move between input fields; the Up and Down arrow keys increment and decrement values, respectively; see the balloon help for additional key bindings. Aggregation by string length with an aggregate size of 5 was used to create the performance profile given in figure 5.

Besides the tabular views, the 'Analyze' menu also allows the graphical presentation of individual profiles. Again, there is a large degree of flexibility, controlled by the following switches:

- 'Analyze – Graph By' is a cascaded menu that selects the x axis for the graph to be drawn, i.e. the dimension along which data points are plotted; the choice is very similar to the 'Options – Aggregation Parameters' seen before and (as of January 1999)[8] includes string and lexical length, the numbers of readings, parser tasks, edges et al. — finally, graphing by test item identifier yields singleton aggregates that, at least for numerically ordered test sets, give a visual clue on the distribution (and homogenity) of some parameter(s) throughout a test run;

- 'Analyze – Graph Values' is another cascaded menu that selects the value(s) plotted along the y axis; the current choice includes four major groups, viz. parser actions, parser times, overall times, and chart edges built; each of the groups contains several attributes (e.g. time for first vs. all readings) that can be selected or deselected individually; as the 'Graph Values' menu (like a few others) often requires several selections at a time, it allows the use of the middle mouse button (instead of the default left mouse button) to (de)activate entries and leave the menu visible; selection of another group or attribute from another group deactivates all incompatible selections;

- 'Analyze – Graph Parameters' finally, pops up an input dialogue similar to the aggregation parameters shown above; the four input fields have the same names and meanings for graph computation as for aggregation and table layout.

To see how the graphing component works, the default values for all the switches will initially be sufficient; 'Analyze – Show Chart' produces the barchart given in figure 6 (top), viz. a distribution of aggregate sizes along the *i-length* (input string length) dimension. Clearly, the test set (the CSLI test suite in this example) only contains a very small number of test items longer than twelve words, above 15 we find only occasional data points. Hence, it seems desirable to restrict the graph view to test items below that upper bound: the 'Analyze – Graph Parameters' dialogue allows the adjustment of that parameter.

Figure 7 shows the output of the 'Analyze – Show Graph' command for the (default) 'Parsing Times' group: the x axis range has been limited and additional attributes (*tcpu* and *tgc* from the compatible 'Overall Times' group) were added to the list of graph values (see section 5.2 for the precise semantics of the individual fields). Finally, the bottom of figure 6 presents the distribution of parser tasks by item identifier (i.e. the substantial

[8] In fact, a larger number of attributes from a competence and performance profile could be used as the aggregation or graphing (x axis) dimension. The author explicitly welcomes comments and suggestions for extension.

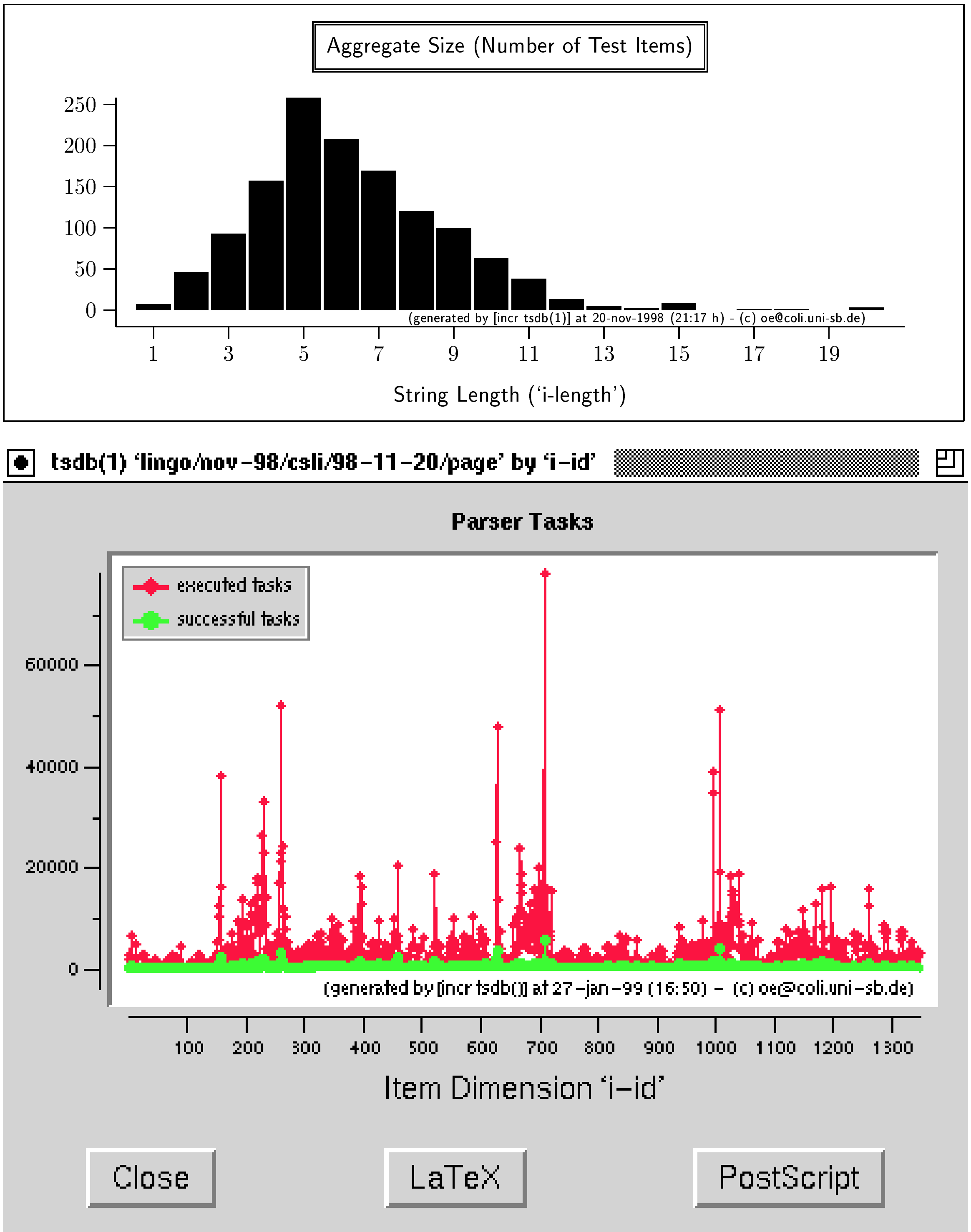


Figure 6: Barchart distribution of aggregate size by string length (top) and graphical distribution of parser tasks throughout test run (bottom); the graphs suggest that the test set is very sparsely populated with test items longer than twelve words and that parsing complexity differs immensely across the test set.

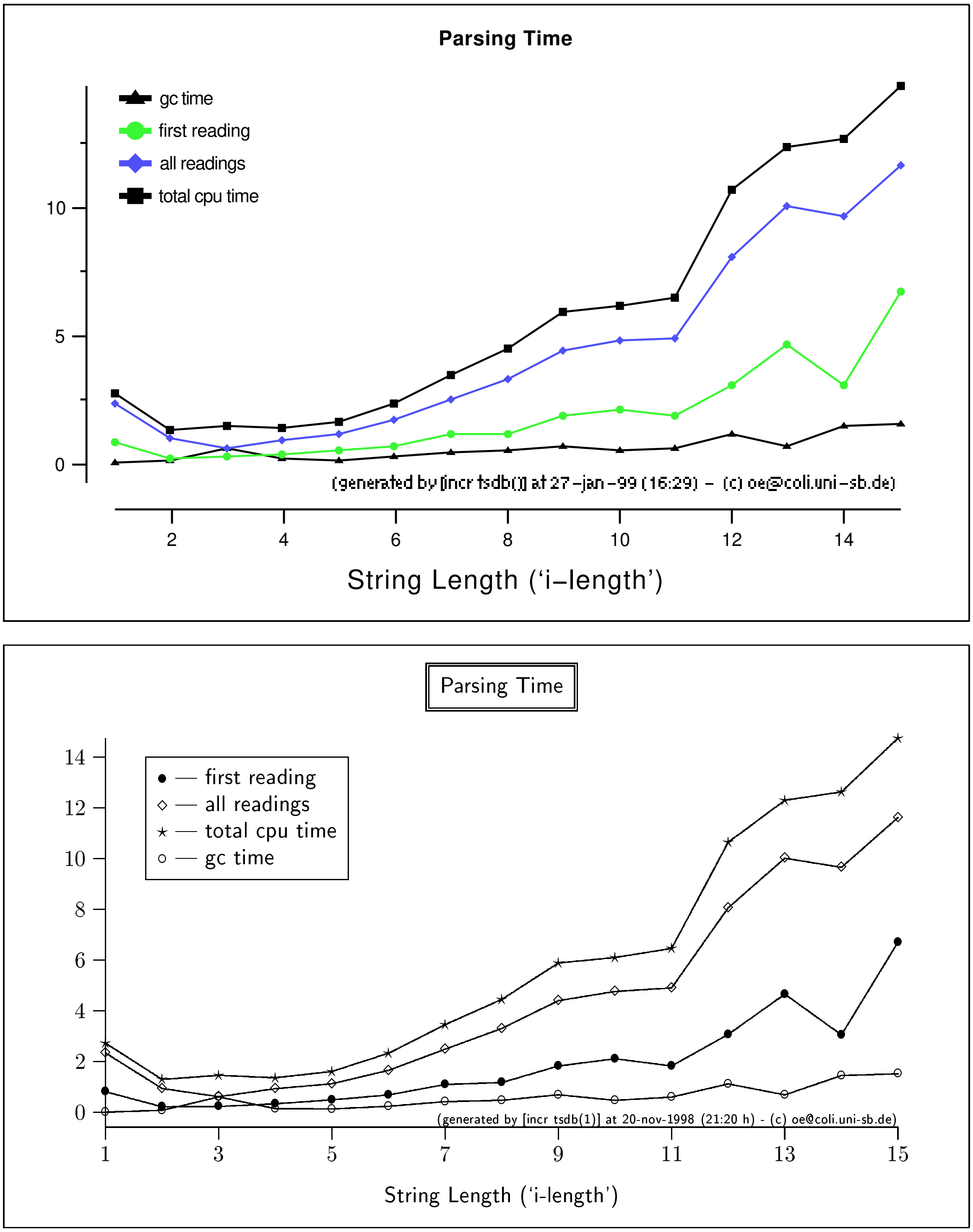


Figure 7: Graphical view of various parsing time metrics in [incr tsdb()] (encapsulated) PostScript (top) and PICTEX (bottom) formats; both formats are fully scalable in size; as of January 1999, only the PostScript output mode allows logarithmic scaling of axes, though.

variation of parsing complexity throughout the test run) obtained from changes to both 'Graph By' and 'Graph Values'; additionally, note that any upper limit imposed on x axis values (15 for string length in the above examples) has to be reset in 'Graph Parameters'.

4.6 Comparison to Earlier Test Runs

As errors are diagnosed and corrected, or as the grammar is modified to extend coverage to additional phenomena, it is often necessary to see how the current version of the grammar or system compares to a previous instance. This evaluation of progress in grammar and system development is facilitated by the commands from the 'Compare' menu, allowing developers to construct summary reports that concisely contrast salient characteristics for two test suite instances.

To obtain a contrastive summary, the two test suite instances to be compared have to be selected out of the current working set. In addition to the active profile (selection with left mouse button), the (second) profile that will be used as the source (or base) for comparison (i.e. the one that is compared to) can be selected either (i) by means of the 'Compare – Source Database' menu or (ii) by clicking the middle mouse button on an entry in the podium body. As a metaphor for progress evaluation (e.g. while working towards an improved version of the grammar or system), the source data set is often referred to as the '(g)old standard' reference to which the current (new) results are compared; accordingly, the source for comparison is highlighted in gold in the podium body. Once developers commit themselves to a new grammar or system version (instance), the newer data can then serve as a gold standard for future experimentation.

One such view of progress, given in Figure 8, shows how grammatical competence (of the LinGO ERG) changed over the course of about one year: as before aggregated according to (coarse-grained) linguistic phenomena, the competence progress profile shows where the analysis of a particular phenomenon has improved and where not.

The grammatical coverage has increased significantly for many phenomena; where coverage has dropped slightly, overgeneration and the number of analyses generated were reduced at the same time. In general, the comparison reassures the grammar engineer that the work on constraining the grammar more rigidly towards the intended analyses (since the same grammar is recently deployed for generation purposes) is going the right direction (as should be expected). Oepen and Flickinger (1998) present a more detailed discussion of the interpretation of individual and constrastive profiles and the use of multiple test sets.

For grammars that can be processed on multiple platforms[9] the comparison of competence can serve an additional purpose. When comparing platforms (or debugging), it is essential to obtain a precise account of how the same (version of a) grammar behaves on either system; such comparison typically greatly helps in the understanding of (formal or practical) differences between platforms.

The complementary comparison of performance profiles tyically proofs useful in system optimization (i.e. tuning it to improved run-time behaviour) and, again, contrastive comparison between platforms. While it remains to be seen whether and which insights can be obtained from comparing across grammar *and* systems simultaneously, clearly the comparative table in figure 9 (top) — contrasting salient performance characteristics of LKB

[9]As of January 1999, the LinGO ERG loads into both LKB and PAGE; at least two other systems, viz. the abstract machines developed at Tokyo University and DFKI Saarbrücken, are expected to use it in the near future

'lingo/oct-97/csli/97-11-26/page' vs. 'lingo/nov-98/csli/98-11-20/page'								
	October 1997				November 1998			
Phenomenon	**lexical** ϕ	**parser** ϕ	**in** %	**out** %	**lexical** ϕ	**parser** ϕ	**in** %	**out** %
S_Types	3.57	2.53	68.3	16.4	2.74	2.49	70.0	18.2
C_Agreement	2.73	1.76	81.6	57.9	2.12	1.57	85.7	57.9
C_Complementation	3.05	2.76	91.7	25.4	2.55	2.33	92.6	31.0
C_Diathesis-Active	3.12	2.70	34.7	47.1	2.35	2.67	34.7	47.1
C_Diathesis-Passive	4.21	3.64	88.9	50.0	3.31	2.96	85.2	37.5
C_Tense-Aspect-Modality	3.19	3.38	82.3	100.0	2.42	2.25	91.1	100.0
C_Negation	2.93	2.63	93.2	0.0	1.79	2.10	88.6	0.0
C_Coordination	2.87	3.36	78.2	29.2	2.32	3.34	72.7	29.2
C_Modification	2.98	2.69	66.1	35.8	2.16	2.74	73.6	43.4
NP_Agreement	3.47	2.44	81.1	44.4	2.15	1.61	78.4	44.4
NP_Modification	2.78	2.33	80.3	8.3	2.17	2.44	87.3	8.3
NP_Coordination	2.17	3.42	89.3	66.7	1.90	4.17	85.7	40.7
Total	**3.12**	**2.75**	**74.9**	**32.9**	**2.38**	**2.50**	**77.0**	**33.2**

(generated by [incr tsdb()] at 28-jan-1999 (15:46 h) – (c) oe@coli.uni-sb.de)

Figure 8: Competence Progress Profile comparing the October 1997 and November 1998 versions of the LinGO ERG: the columns repeat salient properties from the individual competence profiles, viz. the lexical and syntactic ambiguity measures and the overall coverage ('*in*') and overgeneration ('*out*') percentages.

and PAGE on the same grammar — points to several relevant similarities and differences between the two systems:

- both systems indicate a strong mutual dependency between the number of parser tasks executed (i.e. calls to the unifier) and the time and space consumption; thus, overall system performance crucially depends on unifier throughput (as should be expected);

- analysing wellformed input and enumerating all readings is significantly more cost-intensive than rejecting ungrammatical input; for both systems, all three parsing complexity measures given uniformly show that processing illformed test items requires about two thirds (67 – 71%) of the resources used when dealing with grammatical input;

- in total, LKB processes sentences somewhat faster; while executing almost the same number of parser tasks (only about 5% less than PAGE) it requires close to 30% less processing time and less than half of the space.

However, when comparing the additional information in the bottom of figure 9, viz. the total results from the individual performance profiles, more striking differences manifest themselves

- because (in exhaustive search mode, at least) LKB uses a breadth-first parser, the time to find a first reading is essentially the same as the overall processing time; though it may seem puzzling that the *first* value is actually higher than the *total* time, this is due to the fact that timing information for individual readings is only available for items that were successfully analyzed; thus, the average is computed

'lingo/nov-98/csli/98-11-20/page' vs. 'lingo/nov-98/csli/99-01-29/lkb'									
	PAGE			**LKB**			**reduction**		
Aggregate	**tasks** ϕ	**time** ϕ (s)	**space** ϕ (kb)	**tasks** ϕ	**time** ϕ (s)	**space** ϕ (kb)	**tasks** %	**time** %	**space** %
i-wf = 1	3758	3.58	13043	3587	2.56	5863	4.6	28.5	55.0
i-wf = 0	2655	2.47	8983	2421	1.73	4054	8.8	30.1	54.9
Total	**3437**	**3.26**	**11863**	**3249**	**2.32**	**5338**	**5.5**	**28.9**	**55.0**

Platform	**items** $\sharp$	**etasks** ϕ	**filter** %	**edges** ϕ	**first** ϕ (s)	**total** ϕ (s)	**tcpu** ϕ (s)	**tgc** ϕ (s)	**space** ϕ (kb)
PAGE	**1290**	**3437**	**56.8**	**284**	**0.96**	**2.34**	**3.26**	**0.40**	**11863**
LKB	**1293**	**3249**	**87.9**	**200**	**2.47**	**2.15**	**2.32**	**0.17**	**4653**

(generated by [incr tsdb()] at 30-jan-1999 (15:06 h) – (c) oe@coli.uni-sb.de)

Figure 9: Performance comparison between LKB and PAGE on the November 1998 version of the LinGO ERG and the CSLI test suite; here, the contrastive summary (top) is aggregated according to item grammaticality (wellformedness) as annotated in the input data (where '*i-wf = 0*' means grammatical) to highlight the corresponding difference in parsing complexity; again the individual columns repeat salient properties from the individual performance profiles and (in the rightmost block) indicate how they relate to each other (the space reduction of 55%, for example, means that LKB on average requires less than half of the memory allocated in PAGE — a reduction of 90% would correspond to a factor of 10, i.e. one order of magnitude); the bottom table repeats the total numbers from the individual profiles (compare to figure 5) to further illuminate the cross-platform comparison.

over a subset of the test items and should be compared to the value for '*i-wf = 1*' in the table above (2.56 seconds);[10]

- as PAGE uses an active chart parser while the LKB parser is purely passive, the number of edges recorded cannot be compared straightforwardly (280 for PAGE is the sum of active *and* passive edges); since on average about 40% of the total edges are active, this suggests, that PAGE actually builds significantly fewer (passive) edges;

- the observation on edges is supported from a comparison of the substantially different filter rates obtained in the parser:[11] for PAGE the 3437 tasks that were effectively executed manifest about 43% of the total number of parser actions created (i.e. the inverse of the filter efficiency rate), while in LKB the very similar number of executed tasks (3249) presents a much smaller fraction of the overall number of tasks created in the parser (viz. only about 12%); hence, LKB has postulated more than three times as many parser actions as has PAGE;

- while part of the projected difference in postulated tasks can be explained by the passive nature of the LKB parser — it has to recompute rule applications that in PAGE would be represented as an active edge (at the cost of creating edges that may never yield a passive edge because successive daughters cannot be instantiated) — the vast mismatch in the numbers and the larger inventory of passive edges created (as noted above) point to another difference in parsing strategy: in contrast to the LKB parser, PAGE deploys a bidirectional head-driven parsing strategy; since about half of the grammar rules have fairly unrestricted leftmost (or rightmost) daughters that are only constrained once another daughter (often but not always the linguistic head) has been instantiated, the unidirectional LKB parser executes those rules many more times and thus gives rise to the observed proliferation of parser actions;

- summing up, the in total superior performance of LKB is most likely an effect of better unifier throughput (especially noticeable in the 55% difference in memory consumption) that overcompensates a less efficient parsing strategy; as there is no principled obstacle to combining the two approaches (or even modules), the performance profiling suggests there is room for a significant improvement in processing efficiency.

From this very detailed (and maybe sometimes longish) example of performance profile analysis and comparison, at least two conclusions should become clear: (i) the in-depth study and correlation of several parameters can greatly help in the identification of relevant system properties and guide developers to ways of tuning a system; and (ii) the contrastive

[10] Strictly speaking, the grammaticality annotation does not necessarily imply that an item can actually be parsed. An aggregate on the basis of '*readings* $\geq$ *1*' (at least one analysis was obtained) would therefore be the correct reference set; see section 4.7 below on how to restrict the data set accordingly.

[11] Both systems deploy a mechanism to avoid parser actions (i.e. expensive unification) that can be predicted to fail: while (in the November 1998 version) LKB maintains vectors of feature values embedded at paths that are known to have the highests failure potential, and validates compatibility of those vectors before doing the actual unification, PAGE uses a table of rule incompatibility information compiled (off-line) from the input grammar; the average filter rate quoted in the performance profile is the percentage of parser tasks that were postulated but filtered prior to execution. The LKB on-line quick check achieves significantly better filter rates at slightly higher cost (restricted type unification vs. table lookup); recently (January 1999), the two systems have been synchronized to both use a pipeline of table lookup plus subsequent quick check and now obtain almost identical filter rates.

view of two performance profiles given in the top of figure 9 reveals relevant information but cannot substitute for the more detailed individual profile summaries — both have their virtues in their own right.

4.7 Zoom: In-Detail Profile Comparison and Analysis

Although all discussion of profile analysis and comparison so far was based on summaries of aggregated data (exploring some of the available variety in aggregation schemes), the underlying database concept allows the inspection of results at a finer degree of granularity. The 'Detail' and 'Options' menus (the latter when used in conjunction with other commands) provide several ways to zoom in and obtain in-depth views on the data, of which the following paragraphs present three by example.

Firstly, the commands in the 'Detail' menu implement the comparison between two profiles on a per-item basis. The approach is in many respects similar to the Un∗x text utility `diff(1)` but scaled up for structured data sets; the following set of switches allows the customization to user needs:

- 'Source Database' is the same as in the 'Compare' menu; allows the selection of the source data set to which the active profile will be compared; alternatively, use the middle mouse button in podium body;
- 'Phenomena' can be used to restrict the comparison to a subset of the linguistic phenomena (assuming phenomena information is available in the profiles); the menu allows multiple selections (use the middle mouse button to toggle individual entries and keep the menu on display) that will be interpreted as a disjunctive condition when selecting data (see example below); the default is to use the full data set (i.e. all phenomena);
- 'Decoration' allows to choose properties of test items (i.e. attributes from the annotations in the test suite skeleton) that will be presented in the display as decoration for the actual profile data; while the item identifier is always included, the choice of additional attributes (as of January 1999) is limited to the actual item string (*i-input*), the grammaticality code (*i-wf*), and the root category for the item (*i-category*); *i-input* is enabled by default (again, 'Decoration' is a multi-selection menu);
- 'Intersection' finally, is the central parameter in per-item comparison: this (multi-selection) menu requires that at least one attribute from the profile data (i.e. information specific to individual test suite instances) is selected to be used for comparison; only items that differ[12] in the attribute(s) chosen are included in the display and show the conflicting values from both profiles; items that are only included in of the data sets or differ in one of the decorating fields are printed separately with empty intersection values for one profile;

Figure 10 presents an example that goes back to the competence comparison done earlier for two versions of the LinGO ERG (see figure 8). As the 'C_Diathesis-Passive'

[12] As of January 1999, the comparison on values is by equality check only; therefore, the choice of intersection attributes is comparatively small (there is little point in comparing time or space metrics, as even the very same configuration of system and grammar may result in minor differences for these fields; let alone semantic formulae). Again, it is expected to enlarge the range of comparison functions in the near future; thus, comments on user requirements are especially welcome.

'lingo/oct-97/csli/97-11-26/page' vs. 'lingo/nov-98/csli/98-11-20/page'				
i-id	**i-wf**	**i-input**	**(g)old readings**	**new readings**
232	1	Abrams knew it to be true that Browne hired Chiang .	2	1
234	1	Abrams made Browne hire Chiang .	1	0
258	1	Abrams was known to be interviewing Browne .	8	6
259	1	Abrams was known by Chiang to be interviewing Browne .	17	12
260	1	Abrams was known to be interviewing Browne .	8	6
261	1	There was known to be a bookcase in Browne's office .	6	1
262	1	It was known to be time for an interview .	20	10
263	1	It was known to be true that Abrams hired Browne .	3	1
267	1	Abrams was made to interview Browne .	2	4
268	0	Abrams was made interview Browne .	1	0
⋮	⋮	⋮	⋮	⋮

Figure 10: Profile comparison on a per-item basis (for the 'C_Diathesis-Passive' phenomenon that exhibits a mild loss of coverage, significant reduction in overgeneration, and overall elimination of ambiguity in figure 8); only items that conflict in at least one of the fields selected for comparison are included in this view; items # 234 and 268 present reduced coverage and overgeneration, respectively; the overall number of readings obtained has decreased for most of the examples.

phenomenon shows interesting differences at the aggregate level, the configuration in this sample view includes the *i-wf* field in the decoration, restricts the data to the phenomenon in question, and intersects on the number of readings obtained for the two versions.

Double-clicking (the left mouse button) on the *i-input* field in the detailed comparison table, as usual, makes the host platform process the test item interactively, i.e. with all debugging output enabled according to the current system configuration. Interactive processing is often useful, for example, to see what the actual analyses obtained from the system are. This, obviously, presupposes that the host platform has been loaded with the suitable grammar and is operational.

Secondly, the tools introduced already for browsing and analyzing the data can be used in combination with suitable restrictions on the data to select and visualize relevant subsets. Again on the granularity level of individual test items, for example, a grammar engineer is most likely to request a listing of input that constitutes inadequacies in grammatical competence (as observed on the aggregate level of linguistic phenomena, say). Assuming that the test suite was at least annotated with judgements on grammaticality, the following paraphrased queries to the database correspond to lack of coverage and overgeneration, respectively:

- lack of coverage list test items (plus relevant properties) that are annotated as grammatical but failed to parse;
- overgeneration list test items (plus relevant properties) that are tagged ill-formed but accepted by the parser (i.e. were assigned at least one analysis).

Although a demonstration of the [incr tsdb()] query language is included towards the end of this sample session, luckily, the user interface provides a few common selectional conditions as hard-wired choices in the 'Options' menu. To realize the overgeneration query, for example, select 'Illformed' plus 'Analyzed' from the 'Options – TSQL Condition' menu (e.g. using the middle mouse button in activating entries to take advantage of the multi-selection feature) and execute any one command from 'Browse – Test Items' or 'Browse – Parses' to display the subset of items from the current profile that satisfy the condition. Note that, in contrast to the selection of phenomena seen before (e.g. in the 'Detail' menu), the set of TSQL conditions chosen is applied conjunctively, i.e. only test items that meet all active conditions are selected.[13] Once more, double-clicking on the *i-input* field triggers interactive processing to ease debugging.

Arbitrary TSQL conditions (going beyond the hard-wired choices in the user interface) can be composed by means of the 'Options – New Condition' command, are then dynamiclly added to the 'TSQL Condition' list, and can be applied conjunctively with other conditions. Yet, the composition of custom TSQL conditions presupposes a detailed knowledge of the underlying database organization (see 'Browse – Database Schema', the examples below, and section 5.2) as only experienced [incr tsdb()] users command it; to ease experimentation, the condition input dialogue that pops up in the minibuffer supplies context-sensitive completion of attribute names (pressing Tab once completes unambiguous prefixes, twice displays the list of available completions) and a query history that can be traversed using the Up and Down arrow keys. [incr tsdb()] windows created while a selectional condition is in effect show the TSQL representation of the condition used as part of the window title (in square brackets); always remember that summary views on a profile may vary substantially for different subsets of the data.

Finally, the 'Browse – Custom Query' command allows the input of full TSQL clauses consisting of

(i) a selection of attributes from the data set to project and display ('select' clause),

(ii) (optionally) one or more relations to use in the selection ('from'),[14] and

(iii) (optionally) a — possibly complex — condition imposed on the selection ('where').

To complete the guided tour of the [incr tsdb()] package, the query concept is introduced by example presently but not discussed in any formal detail.

Selecting 'Custom Query' from the 'Browse' menu pops up three consecutive input dialogues corresponding to the query clauses (i) – (iii); optional clauses (i.e. 'from' and 'where') can be left empty and skipped using the Return key; again, Tab provides context-sensitive completion (on attribute names or relations, as appropriate) and Up and Down navigate in the history of prior input. To give a simple example, the following query is equivalent to the 'Browse – Phenomena' command (listing the identifiers, names, authors, and dates of construction of all phenomena encoded in the data set) and (on our sample instance of the CSLI test suite) yields the display shown in figure 2 above:

[13] This conjunction includes the choice made by phenomenon, if any, in the 'Browse' submenus; therefore, the browse commands 'All Test Items' or 'All Parses', respectively, now display all entries that match the specified condition.

[14] Since the tsdb query processor — simplifying greatly in contrast to regular SQL — can infer the set of relations required to satisfy a query from the set of attributes used, it is usually not necessary to specify the 'from' clause. Only where attributes are shared between relations and the corresponding values (potentially) differ, the specification of the relation(s) requested will be desirable; besides, in some cases (complex join operations) the 'from' clause can be used to optimize the query processing.

```
select p-id p-name p-author p-date
```
[15]

As a second (slightly more rewarding) sample query, let us return to figure 7 (correlating input length and parsing time): in the segment up to a sentence length of, say, ten words (where aggregate sizes should be large enough and not sparse), the graph exhibits noticeable peaks for *i-length* values of three and nine words. To see which individual test items in these two aggregates require overproportional processing times, the query

```
select i-length i-input i-id tcpu tgc
```

```
where (i-length = 3 && tcpu >= 100) || (i-length = 9 && tcpu >= 200)
```

can be executed to extract the following table (reproduced partially):

i-length	**i-id**	**i-input**	**tcpu**	**tgc**
3	86	There is programmers .	187	87
3	795	Browne merely doesn't work .	420	188
3	366	Does there be a bookcase in Browne's office ?	487	0
⋮	⋮	⋮	⋮	⋮
9	452	Abrams does not know who was hired by Browne .	1034	180
9	394	Does Abrams work for Browne or work for Chiang ?	1206	173
⋮	⋮	⋮	⋮	⋮

Again, only the aggregated view in combination with the inspection of data at a finer-grained level of granularity reveals properties of the data set that would very likely be missed otherwise,[16] viz. that

- the *i-length* values for several test items are incorrect (in the test suite skeleton already); apparently apostrophes in the *i-input* field break the word counting during test data import into [incr tsdb()]; and that

- test items of nine word length are ideosyncratically rich in their use of auxiliaries and coordination, two phenomena that largely contribute to lexical and global ambiguity and, accordingly, parsing complexity; similar ideosyncracies are typical for smaller collections of systematically constructed test data (test suites in the traditional sense) and can often be avoided when aggregating by lexical ambiguity (i.e. the *words* field) rather than string length.

The last example demonstrates how the [incr tsdb()] approach can be used in tuning a system for a particular application (and domain): in the VerbMobil project[17] the PAGE

[15] Remember that, to execute the query, the optional 'from' and 'where' clauses have to be skipped; thus, input the attribute names into the 'select' window (experiment with the completion facility) and then press $\boxed{\text{Return}}$ three times.

[16] This argumentation has strong empirical support: the erroneous *i-length* annotations in the CSLI test suite skeleton slipped through unnoticed for more than a year, while the data set was in heavy use and frequently inspected at other levels of granularity. Obviously, this finding of the presentation will be corrected in the data distributed with [incr tsdb()] such that it shall not be reproducable in releases starting February 1999.

[17] VerbMobil is a German research and development project working on a machine translation prototype for spoken (face-to-face) dialogues in the domain of appointment scheduling.

platform is adapted for linguistic analysis of speech recogniser output. As the analysis component in this application can only be granted a limited time slice to compute (at least) one analysis, an upper limit on the number of parser actions that can be executed is imposed (as of November 1998, the limit chosen is 4000 tasks). By means of the following [incr tsdb()] query, typically applied to a profile obtained from processing test data from the target domain (with no limit imposed):

```
select i-id i-input r-etasks time tcpu
```

```
where result-id = 0 && r-etasks > 4000
```

developers can inspect the set of test items (from the training set) that require more than 4000 tasks for the first reading (‘*result-id = 0*’) already and, hence, will be lost when imposing the specified limit.

4.8 Some Useful Switches

Besides the ‘Database Root’ command from the ‘Options’ menu (used in section 4.1 above to adjust the profile repository) the following facilities may be useful at some point in individual experimentation with the machinery:

- ‘Skeleton Root’ prompts for a new directory containing test suite skeletons and the updates the choice in the ‘File – Create’ menu; while the default [incr tsdb()] distribution ships with English skeletons enabled, this command can be used to activate the German or French TSNLP test suites, some German VerbMobil corpus data, and additional German material (made available by Stefan Müller of DFKI Saarbrücken) that all are included with the distribution; skeletons are organized into language-specific groups, such that by choosing, say, ‘`src/tsdb/skeletons/deutsch`’ (relative to the root directory of the [incr tsdb()] source tree at your site) German data becomes available;

- ‘Update’ reinitializes part or all of the [incr tsdb()] podium state; while relevant changes (e.g. to the database root) usually cause an automatic update, the ‘Update – Skeleton List’ or ‘Update – Database List’ commands can be used to force reloading the necessary information and adjustment of podium status;

- ‘Switches – Exhaustive Search’ toggles the exhaustive search switch of the LKB and PAGE processors (on by default); deactivating exhaustive search makes the parser return when a first analysis was found which can speed up processing a test run significantly; however, remember that non-exhaustive profiles only contain partial information and should not be compared to complete data sets;

- ‘Switches – Overwrite Test Run’ makes [incr tsdb()] overwrite existing test run information (if any) in the current profile or preserve it and append to the profile; when enabled (the default), [incr tsdb()] performs the equivalent of the ‘File – Purge’ operation before the start of a new test run; cumulative profiles are rarely useful, since they require the use of TSQL conditions to distinguish between the various test runs represented in the data set;

- 'Switches – Autoload Vocabulary' finally, toggles the autoloading of vocabulary prior to starting a new test run (on by default); disabling this switch can save time when the necessary vocabulary is known to be loaded (e.g. from a previous test run) or when it is desirable to include lexical access time in the accounting.

4.9 Recommendations for Future Experimentation

5 Reference Manual

5.1 [incr tsdb()] Architecture

5.2 Contents of [incr tsdb()] Profiles

5.3 Storage and Reconstruction of Derivations

5.4 The Menu Structure

5.5 Visualization and Analysis of Profiles

5.6 Comparison among Profiles

5.7 Data Selection and Aggregation

5.8 TSQL syntax

5.9 Importing Data

5.10 Customization: '~/.podiumrc' and '~/.tsdbrc'

5.11 Options and Parameters

5.12 [incr tsdb()] Command-Line Interface

5.13 tsdb Database Format

6 Application Program Interface

Initially, the [incr tsdb()] package was developed as an extension to grammar development platforms (viz. the PAGE and LKB systems) implemented in Common-Lisp; therefore, earlier versions of the profiler were loaded into the same Lisp universe as the grammar development system (called the host platform; see section 3). This setup — sometimes referred to as *integrated* or *embedded mode* — allows [incr tsdb()] to call a set of Lisp functions to interact with the host platform directly.

Though [incr tsdb()] embedded mode still is the default setup for LKB and PAGE, it is not suited to connect the profiler to non-Lisp systems like CHiC and LiLFeS (both implemented in ANSI C). To simplify integration with additional platforms, the [incr tsdb()] *distributed mode* was devised: building on a clean (and simple) ANSI C application program interface, processing systems run as clients to a a stand-alone [incr tsdb()] server process and communicate by means of a general interprocess protocol. The Parallel Virtual Machine (PVM) model (see below) was chosen for interprocess communication in distributed mode; this not only allows users to run [incr tsdb()] on one host while the processing system itself can reside on a different machine (i.e. distribution across a local network or the InterNet), at the same time it facilitates parallelization of test runs: a single [incr tsdb()] instance can communicate with multiple processors (typically on multiple machines) and distribute processing among the clients. To deal with typical robustness issues in distribution (e.g. network or host failure), the [incr tsdb()] distributed mode monitors client status and reschedules tasks when clients become unavailable.

Parallelization of test run processing in LKB and PAGE (or potentially other Lisp-based systems) is achieved by virtue of a Lisp binding for the [incr tsdb()] application program interface. Hence, arbitrary processing systems can connect to the [incr tsdb()] distributed mode as long as they provide a functional interface that obeys the C calling conventions (possibly on the basis of foreign function facilities in Lisp or Prolog systems). The following sections summarize the steps required in adapting a new processing system to [incr tsdb()] distributed mode. The application program interface continues to evolve with the integration of additional processors; developers are therefore encouraged to seek assistance and feedback from the [incr tsdb()] contact address (see the PREFACE section above) — especially if their site is based in a sunny, dry, and urban part of this planet (or has direct beach access).

6.1 Connecting [incr tsdb()] to Another Processor

Integrating the [incr tsdb()] profiler with a new processor (typically a grammar-based parsing system) requires two basic steps:

- **interface setup** a set of interface functions as specified in the [incr tsdb()] application program interface (see below) has to be provided; then, the processor can be linked with the [incr tsdb()] side of the interface (called the client library) and, on request, go into client mode; this mode blocks the client application until a processing request is received from the [incr tsdb()] controller: the application program interface then executes the corresponding client function and relays the result returned by the processor to [incr tsdb()];

- **parameter identification and adaption** though the [incr tsdb()] data model aims to record system competence and performance in generalized (non-system-specific) terms, typically not all parameters foreseen in [incr tsdb()] profiles (see section 5.2) will be applicable (or available) for all processors; likewise, applications may want to record additional system-specific information (or request application-specific formats). It can sometimes require more effort to extract the relevant information from the processor (say, if the system did not record processing statistics already) than to establish the interface proper.

From previous integration experience (viz. with the LKB and CHiC systems) it seems most practical to aim for stepwise and iterative integration and adaptation. Since only very few of the competence and performance parameters (the profile contents) are strictly required for basic [incr tsdb()] functionality,[18] it is recommended to set up and validate the basic interface functionality before filling in the bulk of system parameters.

Abstractly, the [incr tsdb()] application program interface is comprised of four functions that correspond to the following tasks on the client side:

- **test run initialization** obtain information about the current processing environment (i.e. parameters in the '*run*' relation; see section 5.2 and the examples below); additionally, where appropriate, the client can be initialized and prepared for batch processing;

- **test item processing** given a single test item at each call, process the item and return system parameters (mostly stored in the '*parse*' and '*result*' relations) obtained while processing;

- **test run completion** complementary to initialization; (may) reset client to regular processing mode or complete client-side accounting; as of June 1999, no additional information is returned to [incr tsdb()];

- **tree reconstruction** given a derivation tree (see section 5.3), attempt to reconstruct the corresponding phrase structure tree (i.e. replay the derivation) and return information about (the nature of) unification failure when applicable.[19]

The sections on ANSI C and Common-Lisp clients below (6.3 and 6.4, respectively) detail the language-specific instantiations of the interface functions.

6.2 Parallel Virtual Machine

As noted above, the Parallel Virtual Machine (Geist et al. 1994) message-passing model is used for interprocess communication. PVM establishes a virtual machine (a set of cpus) from a collection of networked computers; a user-level PVM daemon on each physical node (e.g. a workstation or compute server) establishes a transparent message-passing layer that provides PVM applications with a uniform view of the virtual machine. Using PVM primitives, [incr tsdb()] can create client processes (i.e. [incr tsdb()]-aware application systems) on arbitray nodes in the virtual machine (or let PVM take the distribution decisions), transmit processing requests to available clients, and collect processing results (competence and performance parameters).

[18] Although, naturally, profile analysis and comparison may be severely restricted on partial data. Assuming a processor that does not fill in any of the system-specific parameters, test run processing will still be possible; however, the bulk of [incr tsdb()] analysis functionality will be non-functional.

[19] As of June 1999, however, reconstruction mode is not fully implemented on the [incr tsdb()] side of the application program interface; while integrating with the LiLFeS application, this is expected to change really soon™, though.

The [incr tsdb()] distribution includes PVM binaries for common platforms; these binaries were (mildly) customized, mostly to simplify PVM startup. There should be no principled obstacle, however, to using existing PVM installations where available. That the PVM daemons run as user-level processes means that no system-level installation or configuration support is required; at the same time, it obliges all users who want to deploy [incr tsdb()] in distributed mode to establish their personal PVM environment. Fortunately, setting things up is very simple and mostly a once-only task; the virtual machine, once established, is fully independent from [incr tsdb()] and, typically, remains available until explicitly (by user request) or implicitly (system `reboot(8)`) terminated. The following paragraphs summarize the necessary user action to make the PVM environment operational; for background information see the PVM `man(1)` pages distributed with [incr tsdb()] and Geist et al. 1994).

User-Level Configuration A PVM virtual machine is composed of one or more physical hosts connected to a common network.[20] Typically, individual users will have a set of machines available to them; the user-level PVM configuration file '`~/.pvm_hosts`' can be used to describe a group of hosts accesible to PVM that shall be joined into a virtual machine:

```
#
# list machines accessible to PVM; option fields are (see pvmd(8))
#
#   - dx: path to 'pvmd3' executable (on remote host);
#   - ep: colon-separated PATH used by pvmd(8) to locate executables;
#   - wd: working directory for remote pvmd(8);
#   - ip: alternate (or normalized) name to use in host lookup;
#
&teej.is.s.u-tokyo.ac.jp dx=/home/users/oe/src/itsdb/bin/osf/pvmd3 wd=/tmp
&eo.stanford.edu dx=/user/oe/src/itsdb/bin/solaris/pvmd3 wd=/tmp
&eoan.stanford.edu dx=/user/oe/src/itsdb/bin/solaris/pvmd3 wd=/tmp
top.coli.uni-sb.de dx=/proj/perform/itsdb/bin/solaris/pvmd3 wd=/tmp
cpio.coli.uni-sb.de dx=/proj/perform/itsdb/bin/linux/pvmd3 wd=/tmp
perl.coli.uni-sb.de dx=/proj/perform/itsdb/bin/linux/pvmd3 wd=/tmp
limit.dfki.uni-sb.de dx=/proj/perform/itsdb/bin/solaris/pvmd3 wd=/tmp
let.dfki.uni-sb.de dx=/proj/perform/itsdb/bin/solaris/pvmd3 wd=/tmp
```

The '`.pvm_hosts`' file contains one host per line together with optional information about the location of the `pvmd(8)` executable on the target host, the working directory and search '`PATH`' environment variable to be used, and other configuration options (see the example file and the PVM documentation). A leading '`&`' character can be used to indicate that a particular machine is not to be activated by default; still, the '`.pvm_hosts`' entry makes configuration data for that host available to PVM such that it can be added dynamically on user request.

[20] The amount of data transferred between [incr tsdb()] server and clients is modest but can be non-trivial; hence, (assuming a high-performance client) network throughput in non-local networks (or a highly congested ethernet) may become an issue. Still, [incr tsdb()] distributed mode can make it feasible to utilize remote cpus, typically in addition to local resources; even successful test runs across the Atlantic Ocean have already been reported.

Starting and Stopping Once the user-level PVM configuration has been completed, the virtual machine is created by starting the `pvmd(8)` daemon on one (arbitrary) node of the configuration. On startup, `pvmd(8)` will consult the '`.pvm_hosts`' file and attempt to start PVM daemons on all remote nodes automatically. Since remote `pvmd(8)` startup is achieved using the `rsh(1)` protocol, users have to make sure that the host used to start PVM has `rsh(1)` access to all nodes in the virtual machine (e.g. using the system-wide `hosts.equiv(5)` or the user-specific `rhosts(5)` mechanisms).[21]

The PVM daemon writes protocol messages into the file '`/tmp/.pvmd.debug.`*user*' (where '*user*' obviously is the active account name) that should be consulted in case of problems. Status information on PVM can be obtained using the `pvm(1)` shell that connects to an existing virtual machine and supplies a set of user commands to query PVM status (especially the `pvm(1)` 'conf' and 'ps' commands). The `pvm(1)` command 'halt' can be used to shutdown the virtual machine (on all active nodes).[22] However, it is often more convenient to leave an existing virtual machine alive even when it is not in active use; PVM daemons on the individual nodes can continue to run for weeks or months without user interaction. Only after explicit ('halt' or 'kill' in `pvm(1)`) or implicit (node failure) daemon termination users should have to restart the virtual machine.

6.3 ANSI C Clients

The [incr tsdb()] distribution contains a function library '`libitsdb.a`' or '`libitsdb.so`' (precompiled libraries are available for common [incr tsdb()] platforms; see appendix A for the exact content of the current distribution and the location of the library files) that clients should use to connect to the ANSI C application program interface. Prototypes for the functions used by the application to establish an [incr tsdb()] (client-side) binding are supplied in the header file '`itsdb.h`'. In [incr tsdb()] mode, basically, a client has to:

(i) supply the four interface functions (according to the function prototypes given below) such that they can be called following the C calling conventions (thus, the functions need not be implemented in C as long as they obey the argument passing and return value specification);

(ii) register these functions with the [incr tsdb()] side of the application program interface (i.e. make the entry points to the client-side functions known as function pointers) and notify the [incr tsdb()] server of the availability of the new client;

(iii) go into client mode: call the `slave()` function (supplied in '`itsdb.a`') that blocks the application until a task request is received, executes the requested task (i.e. calls back into the client using one of the interface functions), and relays the information returned to the [incr tsdb()] server.

The `slave()` function runs in an infinite loop and only returns to the client when the server requests client termination (which is caused implicitly when the server becomes

[21] To validate that these conditions for PVM startup are met, it may be helpful to verify that a command like '`rsh perl.coli.uni-sb.de date`' (substituting the name of an actual remote node, naturally) can be completed successfully. For improved system security it may be desirable to designate a single (trusted) machine in the network as the host that is used to start PVM; then, other nodes in the virtual machine only have to allow `rsh(1)` access to this one machine.

[22] Please note that the [incr tsdb()] process becomes part of the virtual machine the first time the application program interface is used; thus, killing off all PVM processes may terminate the [incr tsdb()] session, too.

unavailable) or an error in PVM communcation is encountered; the client should then terminate gracefully. Typically, a command-line option should be used to turn the processor into [incr tsdb()] client mode (which will then be supplied by the [incr tsdb()] server in process creation) that, in turn, makes the client execute a code fragment like:

```
#include "itsdb.h"
  [...]
  if(!capi_register(create_run,
                    process_item,
                    (int (*)(char *))NULL,
                    complete_test_run)) {
    slave();
  } /* if */
  exit(0);
```

The interface functions supplied by the client are instantiated as follows in the ANSI C application program interface:[23]

- int create_run(char *data, int run_id, char *comment, int interactivep, char *custom);

  with parameters:[24]

  → data name of the test suite instance to be processed;

  → run_id identifier for this test run;

  → comment descriptive comment supplied by the user;

  → interactivep flag indicating whether the test run is to be processed in batch mode (regular [incr tsdb()] mode) or interactively (see below);

  → custom an application-specific string that was (optionally) supplied in the client definition; when supplied, the client may take an appropriate action on this parameter (e.g. (re)load a script file or similar);

  since the interface functions need to return more information to [incr tsdb()] than just a simple function return value, while executing the client side of the interface (e.g. the `create_run()` function) the standard output stream[25] is redirected to the [incr tsdb()] server: the client is expected to write system parameters in a Lisp-like syntax (i.e. as bracketed pairs consisting of the parameter name and the corresponding

[23] Since client functions are registered by entry point (i.e. as function pointers), names for the interface functions can be assigned arbitrarily in the ANSI C application program interface. The function names used in the startup example above and the prototypes below are just one candidate choice of a consistent naming scheme.

[24] The interface functions often have more parameters than would be strictly required to process the requested task; the additional information is supplied to the client to facilitate client-side accounting or the display of status messages, where available. Obviously, these surplus parameters can be safely ignored.

[25] To avoid misinterpretation of output generated by the client, it is essential that — while executing a function from the [incr tsdb()] application program interface — nothing except system parameters is written to standard output. Developers preparing a client for [incr tsdb()] adaptation should make sure that additional (e.g. status or debugging output) is printed to the standard error stream instead: this stream is captured at the PVM layer, relayed to the server, and written to a protocol file without further interpretation.

value) to this stream;[26] following is an example of output generated by a recent LKB version when run in [incr tsdb()] distributed mode:

```
← (:platform . "Allegro CL (5.0.beta [Linux/X86] (1/1/90 0:56))")
← (:application . "LKB (version ‘$Date: 1999/05/14 04:10:18 $’)")
← (:grammar . "LinGO (may-99)")
← (:avms . 3144) (:sorts . 0) (:templates . 11)
← (:lexicon . 588) (:lrules . 27) (:rules . 36)
```

- int process_item(int i_id, char *i_input, int parse_id, int edges,
  int exhaustivep, int derivationp, int interactivep);

  with parameters:

  → i_id identifier for this test item;

  → i_input actual string for this item;

  → parse_id identifier for this process request;

  → edges upper limit for the number of edges (successful rule applications) to be built by the parser (‘-1’ for no limit);

  → exhaustivep flag requesting exhaustive (complete) search: ‘1’ by default; negative numbers encode an upper limit on the number of analyses to compute where ‘0’ is interpreted equivalent to ‘-1’;

  → derivationp flag asking the client to return not only the derivations for the analyses that were found but to append the complete list of all passive edges built to the ‘:results’ field (see section 5.3 and the example output below);

  → interactivep similar to ‘interactivep’ parameter in create_run(): [incr tsdb()] users can request interactive processing of individual items (e.g. by double-clicking on the *i-input* field in most analysis tables; see section 4); while processing in interactive mode the client should activate all available debugging tools, like chart and result structure displays, for example;

  again, sample output produced by the LKB:

```
← (:others . 367136) (:symbols . 0) (:conses . 158480)
← (:first . 60) (:total . 60)
← (:treal . 93) (:tcpu . 90) (:tgc . 0)
← (:words . 5) (:l-stasks . 2)
← (:p-ftasks . 825) (:p-etasks . 55) (:p-stasks . 31)
← (:pedges . 14) (:rpedges . 4)
```

[26] Though this mode of communication enables the client to relay processing results to the [incr tsdb()] server without much effort (i.e. using the regular printing routines available in all programming environments), it is, at the same time, not especially robust: since the client output is forwarded to the [incr tsdb()] server without further verification or format validation, failure to obey the surface format required in the interface can easily break the server-side of the interface (often, resulting in mysterious system malfunctioning). Therefore, it is expected to add an alternative mode of return parameter passing (using structured C objects) in the near future.

```
← (:readings . 1)
← (:results
    ((:result-id . 0) (:time . 60)
     (:r-redges . 4) (:size . 147)
     (:derivation . "(\"root_cl\" 0 2
                       (\"subjh\" 0 2
                        (\"no-affix_infl_rule\" 0 1
                         (\"abrams\" 0 1 (\"abrams\" 0 1)))
                        (\"third_sg_fin_verb_infl_rule\" 1 2
                         (\"work_v1\" 1 2 (\"work\" 1 2)))))")))
← (:error . "")
```

- int complete_run(int run_id, char *custom);

  with parameters:

  → run_id name of the test suite instance to be processed;

  → custom similar to 'custom' parameter in `create_run()`: an optional piece of system-specific information supplied in the client definition;

  as of June 1999, `complete_run()` does not return any information to [incr tsdb()].

6.4 Common-Lisp Clients

6.5 Using [incr tsdb()] Distributed Mode

As of June 1999, there is no explicit support for [incr tsdb()] distributed mode in the graphical user interface; although the regular test run functionally will transparently work on top of a set of client processors connected in distributed mode, the configuration and creation of client processes still has to be achived manually (i.e. on the Common-Lisp side). The discussion of user interaction with [incr tsdb()] will build on the following concepts:

- **cpu** the term *cpu* is used to refer to the specification of client processors: each cpu is usually described in terms of a host (node in the PVM virtual machine that is used to run the client), the command to start the client (i.e. a binary executed on the remote machine), optional startup options, and one or more class identifier(s) used to refer to individual cpus or cpu groups; [incr tsdb()] cpu descriptions can include additional information like the 'CUSTOM' data passed to the client on test run creation and completion (see above);

- **client** a new [incr tsdb()] *client* (or client task) is created each time a cpu is activated (or initialized); activating a cpu here means to request (from the PVM daemon responsible for the node in question) that the command associated with the cpu be executed; after process creation the client itself is responsible for registration with the [incr tsdb()] server (typically through execution of the '`slave()`' function presented in sections 6.3 and 6.4 above) a client process on some node in the virtual machine that the [incr tsdb()] can communicate with by virtue of the application program interface

```
(setf *pvm-cpus*
  (list
   (make-cpu
    :host "let.dfki.uni-sb.de" :class '(:chic :chic@let)
    :spawn "/project/cl/chic/bin/chic"
    :options '("-tsdb"))
   (make-cpu
    :host "let.dfki.uni-sb.de" :class '(:chic :chic@let)
    :spawn "/project/cl/chic/bin/chic"
    :options '("-tsdb"))
   (make-cpu
    :host "limit.dfki.uni-sb.de" :class '(:chic :chic@limit)
    :spawn "/project/cl/chic/bin/chic"
    :options '("-tsdb"))
   (make-cpu
    :host "top.coli.uni-sb.de" :class '(:chic :chic@coli)
    :spawn "/project/cl/chic/bin/chic"
    :options '("-tsdb"))
   (make-cpu
    :host "top.coli.uni-sb.de" :class :lkb
    :spawn "/proj/perform/nacl/bin/acl"
    :options '("-L" "/proj/perform/lkb/startup")
    :create "/proj/perform/lingo/jun-99/lkb/script")))
```

Figure 11: Sample definition of [incr tsdb()] cpus (taken from a user '.tsdbrc' file): the class names chosen — at least in some cases — reflect the client type as well as the node used to run the client.

Usually, users will have a set of [incr tsdb()] cpus to choose from; when preparing for a test run, a selection from the set of available cpus is made to create clients as needed. The per-user configuration file '~/.tsdbrc' (see section 5.10) can be used to enumerate a list of cpus (similar to the PVM node listing in the '~/.pvm_hosts' file).

6.6 Debugging [incr tsdb()] Distributed Mode

A Contents of the [incr tsdb()] Distribution

References

Carroll, John. 1994. Relating Complexity to Practical Performance in Parsing with Wide-Coverage Unification Grammars. In *Proceedings of the 31st Meeting of the ACL*, 287 – 294. Las Cruces, New Mexico.

Copestake, Ann. 1992. The ACQUILEX LKB. Representation Issues in Semi-Automatic Acquisition of Large Lexicons. In *Proceedings of ANLP 1992*, 88 – 96. Trento, Italy.

Erbach, Karl Gregor. 1991. An Environment for Experimenting with Parsing Strategies. In *Proceedings of IJCAI 1991*, ed. John Mylopoulos and Ray Reiter. San Mateo, California. Morgan Kaufmann Publishers.

Geist, Al, Adam Bequelin, Jack Dongarra, Weicheng Jiang Robert Manchek, and Vaidy Sunderam (ed.). 1994. *PVM — Parallel Virtual Machine. A Users' Guide and Tutorial for Networked Parallel Computing.* Scientific and Engineering Computation. Cambridge, Massachusetts: The MIT Press.

Lehmann, Sabine, Stephan Oepen, Sylvie Regnier-Prost, Klaus Netter, Veronika Lux, Judith Klein, Kirsten Falkedal, Frederik Fouvry, Dominique Estival, Eva Dauphin, Hervé Compagnion, Judith Baur, Lorna Balkan, and Doug Arnold. 1996. TSNLP — Test Suites for Natural Language Processing. In *Proceedings of COLING 1996*, 711 – 716. Kopenhagen, Dänemark.

Oepen, Stephan, and Daniel P. Flickinger. 1998. Towards Systematic Grammar Profiling. Test Suite Technology Ten Years After. *Journal of Computer Speech and Language* 12 # 4 (Special Issue on Evaluation):411 – 436.

Oepen, Stephan, Klaus Netter, and Judith Klein. 1997. TSNLP — Test Suites for Natural Language Processing. In *Linguistic Databases*, ed. John Nerbonne. CSLI Lecture Notes. Center for the Study of Language and Information.

Ousterhout, John K. 1994. *Tcl and the Tk Toolkit.* Reading, MA: Addison-Wesley Publishing Company.

Uszkoreit, Hans, Rolf Backofen, Stephan Busemann, Abdel Kader Diagne, Elizabeth A. Hinkelman, Walter Kasper, Bernd Kiefer, Hans-Ulrich Krieger, Klaus Netter, Günter Neumann, Stephan Oepen, and Stephen P. Spackman. 1994. DISCO — An HPSG-based NLP System and its Application for Appointment Scheduling. In *Proceedings COLING 1994*. Kyoto.