Web tools for introductory computational linguistics

Dafydd Gibbon
Fakultät für Ling. & Lit.
Pf. 100131, D–33501 Bielefeld
gibbon@spectrum.uni-bielefeld.de

Julie Carson–Berndsen
Department of Computer Science
Belfield, Dublin 4, Ireland
Julie.Berndsen@ucd.ie

Abstract

We introduce a notion of *training methodology space* (TM space) for specifying training methodologies in the different disciplines and teaching traditions associated with computational linguistics and the human language technologies, and pin our approach to the concept of *operational model*; we also discuss different general levels of interactivity. A number of operational models are introduced, with web interfaces for lexical databases, DFSA matrices, finite–state phonotactics development, and DATR lexica.

1 Why *tools* for CL training?

In computational linguistics, a number of teaching topics and traditions meet; for example:

- formal mathematical training,
- linguistic argumentation using sources of independent evidence,
- theory development and testing with empirical models,
- corpus processing with tagging and statistical classification.

Correspondingly, teachers' expectations and teaching styles vary widely, and, likewise, students' expectations and accustomed styles of learning are very varied. Teaching methods and philosophies fluctuate, too, between more behaviouristic styles which are more characteristic of practical subjects, and the more rationalistic styles of traditional mathematics training; none, needless to say, covers the special needs of all subjects.

Without specifying the dimensions in detail, let us call this complex field *training method space (TM space)*. The term *training* is chosen because it is neutral between *teaching* and *learning*, and implies the intensive acquisition of both theoretical and practical abilities. Let us assume, based on the variations outlined above, that we will need to navigate this space in sophisticated ways, but as easily as possible. What could be at the centre of TM space? As the centre of TM space, let us postulate a *model–based* training method, with the following properties:

1. The models in TM space are both formal, and with operational, empirical interpretations.
2. The empirical interpretations of models in TM space are in general operational models implemented in software.
3. The models in TM space may be understod by different users from several different perspectives: from the point of view of the mathematician, the programmer, the software user etc., like 'real life programmes'.
4. Typical lingware and software models are grammars, lexica, annotated corpora, operationalised procedures, parsers, compilers; more traditional models are graphs, slides, blackboards, three–dimensional block or ball constructions, calculators.

Why should operational models, in the sense outlined here, be at the centre of TM–

space? There are several facets to the answer: First, the use of operational models permits *practice* without succumbing to the naivetiés of stimulus–response models. Second, this notion of model is *integrative*, that is, they are on the one hand *mathematical*, in that they are structures which are involved in the interpretation of theories, and at the same time they are *empirical*, in representing chunks of the world, and *operational*, in that they map temporal sequences of states on to real time sequences. But, third, working with operational models is more *fun*. Ask our kids.

This paper describes and motivates a range of such models: for arithmetic, for manipulating databases, for experimenting with finite state devices, for writing phonological (or, analogously, orthographic) descriptions, for developing sophisticated inheritance lexica.

2 What kind of *interactivity*?

The second kind of question to be asked is: Why the Web? Interactive training tools are not limited to the Web; they have been distributed on floppy disk for over two decades, and on CD–ROM for over a decade. In phonetics, interactive operational models have a long history: audio I/O, visualisations as oscillogrammes, spectrogrammes, pitch traces and so on, have been leading models for multi–media in teacher training and speech therapy education since the 1970s. So why the Web? The answers are relatively straightforward:

- The Web makes software easy to distribute.
- The Web is both a distributed user platform and a distributed archive.
- New forms of cooperative distance learning become possible.
- Each software version is instantly available.
- The browser client software used for accessing the Web is (all but) universal (modulo minor implementation differences) in many ways: platform independent, exists in every office and many homes, ...

The tools describe here embody three different approaches to the *dependence* of students on teachers with regard to the provision of materials:

1. Server–side applications, realised with standard CGI scripts: practically unlimited functionality, with arbitrary programming facilities in the background, but with inaccessible source code.
2. Compiled client–side applications, realised with Java: practically unlimited functionality, particularly with respect to graphical user interfaces, typically with inaccessible source code.
3. Interpreted client–side applications, realised with JavaScript: limited functionality with respect to graphical user interfaces, functionality limited to text manipulation and manipulation of HTML attributes (including CGI preprocessing), typically with immediately accessible code.

From the formal point of view, these programming environments are equally suitable. From the (professional) programming point of view, the object oriented programming style of Java is often the preferred, homogeneous environment, though it is hard to relate it to other styles. CGI provides an interface for arbitrary programming languages, and scripting languages are highly relevant to linguistic tasks, particularly modern varieties such as `perl`, with respect to corpus tagging and lexicon processing, or *Tcl* to the visualisation of formal models or speech transformations. JavaScript is a pure client–side application, and has a number of practical advantages which outweigh many of its limitations: JavaScript is interpreted, not compiled, and the code is immediately available for inspection by the user; despite its simplicity, it permits arbitrarily complex textual and numerical manipulation and basic window management; like other scripting languages, Javascript is not designed for modular programme development or library deployment, but is best restricted to small applications of the kind used in introductory work.

There is another issue of interactivity at a very general level: in software development, perhaps less in the professional environment than in the training of non–professionals to understand what is 'going on under the bonnet', or to produce small custom applications: the open software, shared code philosophy. In the world 'outside' salaries are obviously dependent on measurable product output, and intellectual property right (IPR) regulations for shareware, licences and purchase are there to enable people to make a legitimate living from software development, given the prevailing structures of our society.

As far as teaching is concerned, the debate mainly affects programmes with medium functionality such as basic speech editors or morphological analysers, often commercial, with products which can be produced in principle on a 'hidden' budget by a small group of advanced computer science or engineering students (hence the problem). Obviously, it is easy for those with in stable educational institutions to insist that software is common property; indeed it may be said to be their duty to provide such software, particularly in the small and medium functionality range.

Finally, it is essential to consider design issues for interactive teaching systems, an area which has a long history in teaching methodology, going back to the programmed learning and language lab concepts of the 1960s, and is very controversial (and beyond the scope of the present paper). We suggest that the discussion can be generalised via the notion of TM space introduced above to conventional software engineering considerations: *requirements specification* (e.g. specification of location in TM space by topic, course and student type), *system design* (e.g. control structures, navigation, windowing, partition of material, use of graphics, audio etc.), *implementation* (e.g. server–side vs. client side), *verification* (e.g. 'subjective', by users; 'objective', in course context).

Only a small amount of literature is available on teaching tools; however, cf. (HH1999) for speech applications, and the following for applications in phonetics and phonology (CB1998a), (CBG1999), English linguistics (CBG1997), and multimedia communication (G1997). The following sections will discuss a number of practical model–based applications: a basic database environment; an interpreter for deterministic finite automata; a development environment for phonotactic and orthographic processing; a testbed and scratchpad for introducing the DATR lexical representation language. The languages used are JavaScript (JS) for client–side applications, and Prolog (P) or C (C) for server–side CGI applications.

3 Database query interface generator

Database methodology is an essential part of computational linguistic training; traditionally, UNIX ASCII databases have been at the core of many NLP lexical databases, though large scale applications require a professional DBMS. The example shown in Figure 1 shows a distinctive feature matrix (Jakobson and Halle consonant matrix) as a database relation, with a query designed to access phonological 'natural classes'; any lexical database relation can be implemented, of course. In this JavaScript application with on–the–fly query interface generation the following functionality is provided:

1. Input and query of single database relations.
2. Frame structure, with a control frame and a display/interaction frame which is allocated to on-the-fly or pre-stored database information.
3. The control frame permits selection of:
   (a) a file containing a pre-compiled database in JavaScript notation,
   (b) on–the–fly generation of a query interface from the first record of the database, which contains the names of the fields/attributes/columns,
   (c) on–the–fly generation of tabular representation of the database,
   (d) input of databases in tabular form.
4. Query interface with selection of arbitrary conjunctions of query attributes and values, and output attributes.

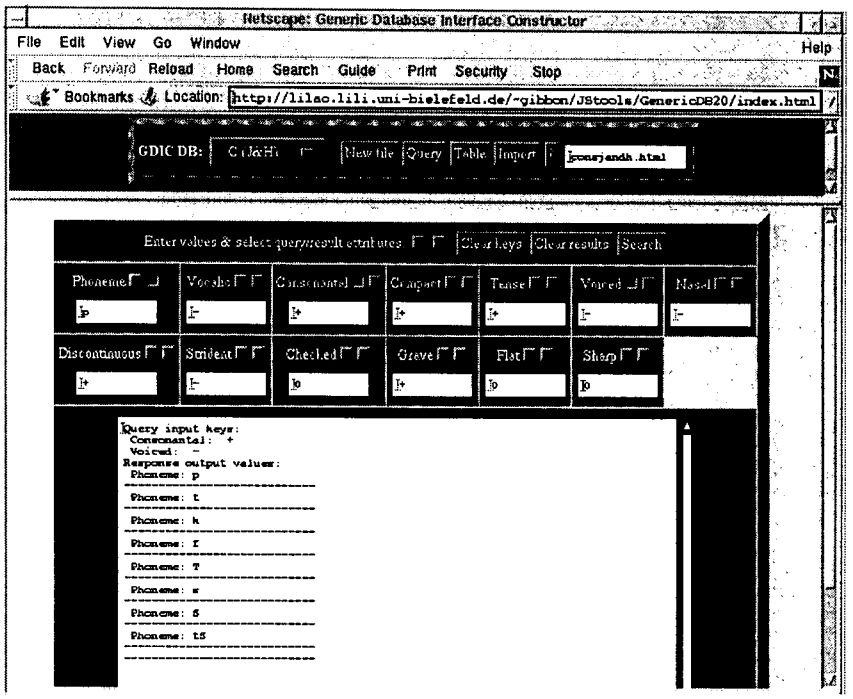

Figure 1: Database interface generator (JavaScript).

5. Compilation of database into a JavaScript data structure: a one–dimensional array, with a presentation parameter for construction of the on–the–fly query interface.

Typical applications include basic dictionaries, simple multillingual dictionaries, representation of feature structures as a database relation with selection of natural classes by means of an appropriate conjunction of query attributes and values.

Tasks range from user–oriented activities such as the construction of 'flat' databases, or of feature matrices, to the analysis of the code, and the addition of further input modalities. Advanced tasks include the analysis of the code, addition of character coding conventions, addition of further database features.

4 DFSA interpreter

There are many contexts in computational linguistics, natural language processing and spoken language technology in which devices based on finite state automata are used; for example, tokenisation, morphological analysis and lemmatisation, shallow parsing, syllable parsing, prosodic modelling, plain and hidden markov models. A standard component of courses in these disciplines is concerned with formal languages and automata theory. The basic form of finite state automaton is the *deterministic finite state automaton (DFSA)*, whose vocabulary is epsilon–free and which has no more than one transition with a given label from any state. There are several equivalent representation conventions for DFSAs, such as a full transition matrix ($Vocabulary \times StateSet$) with target states as entries; or sparse matrix representation as a relation, i.e. a set of triples constituting a subset of the Cartesian product $StateSet \times Stateset \times Vocabulary$; or transition network representation.

The interface currently uses the full matrix representation, and permits the entry of arbitrary automata into a fixed size matrix. The example shown illustrates the language a^*b, but symbols consisting of an arbitrary number of characters, e.g. natural language examples, may be used. A state–sequence trace, and detailed online help explanations, as well as task suggestions of varying degrees of difficulty are

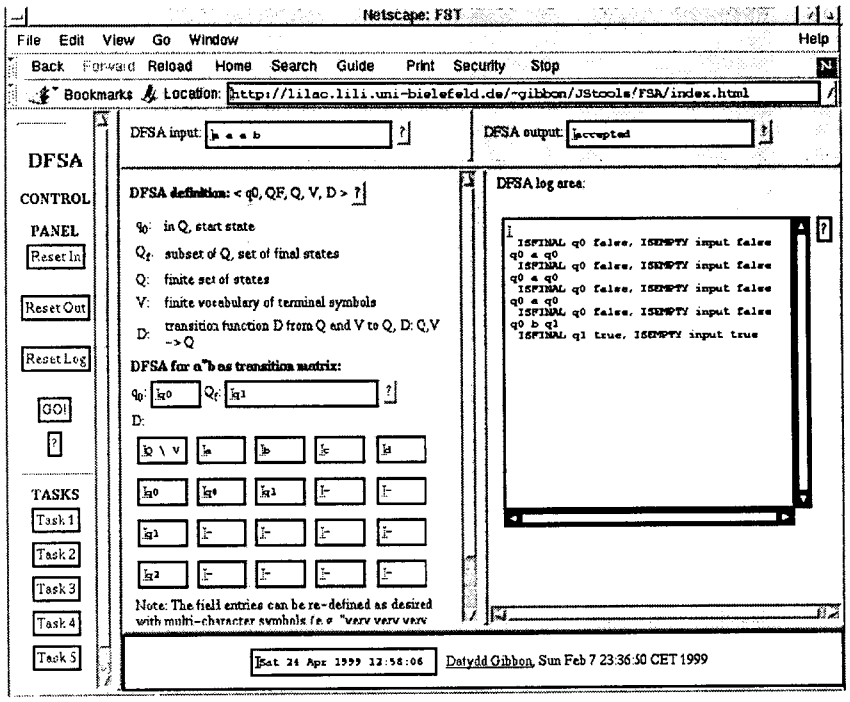

Figure 2: Deterministic finite state automaton (DFSA) workpad (JavaScript).

provided.

5 Phonological NDFSA development environment

Phonology and prosody were the first areas in which finite state technologies were shown to be linguistically adequate and computationally efficient, in the 1970s; a close second was morphological alternations in the 1980s (CB1998a). The goal of this application is to demonstrate the use of finite state techniques in computational linguistics, particularly in the area of phonotactic and allophonic description for computational phonology. However, the tool does not contain any level specific information and can therefore be used also to demonstrate finite state descriptions of orthographic information. In this CGI application, implemented in Prolog (see (CBG1999)), the following functionality is provided:

1. Display/Alter/Add to the current (non–deterministic) FSA descriptions.
2. Generate the combinations described in the FSA.
3. Compare two FSA descriptions.
4. Parse a string using an FSA description.

Typical applications of this tool include descriptions of phonological well-formedness of syllable models for various languages. Tasks range from testing and evaluating to parsing phonological (or orthographic) FSA descriptions. More advanced tasks include extension of the current toolbox functionality to cater for feature-based descriptions.

6 Zdatr testbed and scratchpad

DATR is a well–known theoretically well–founded and practically oriented lexicon representation language. It also has a high ratio of implementations to applications, and, until relatively recently, a low degree of standardisation between implementations. In order to create a platform independent demonstration and practice environment, a CGI interface was created. The engine was intended to be a Sicstus Prolog application; Sicstus turned out to be non–CGI–compatible at the time (1995), so a UNIX shell version of DATR (`mud`, minimal UNIX DATR) was implemented using a combination of UNIX text stream processing tools, mainly `awk`. This

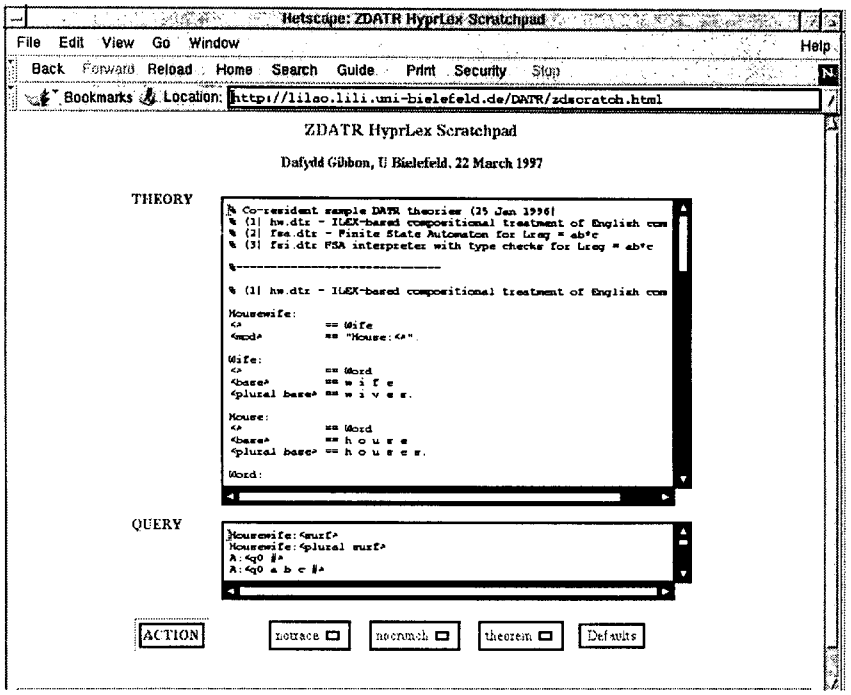

Figure 3: Zdatr scratchpad (CGI, UNIX shell, C).

was later replaced by *Zdatr V1.n*, and will shortly be replaced by *Zdatr V2.0* (implemented by Grigoriy Strokin, Moscow State University). The *Zdatr* software is widely used in the teaching of lexicography, lexicology, and lexicon theory (Ginpress).

Two interfaces are driven by Zdatr: The *testbed* which permits interactions with previously defined and integrated DATR theories (CBG1999), and the *scratchpad* (shown in Figure 3), with which queries can be written and tested. The *scratchpad* permits the entry of short theories and queries, and the *testbed* has the following functionality:

1. viewing of DATR theories;
2. selection of manual, declared (`#show` and `#hide`) and pre–listed queries;
3. selection of output properties (trace, atom spacing).
4. a function for automatically integrating new theories sent by email (not documented for external use).

Sample DATR theories which are available include a detailed model of compositionality in English compound nouns (Gibbon), an application for developing alternative feature or autosegmental phonological representations (Carson–Berndsen), and a number of algorithm illustrations (bubble sort, shift register machine) for more theoretical purposes.

7 Outlook

Tools like those introduced here are not ubiquitous, and there are many areas of computational linguistics, in particular formal training in computing and training in linguistic argumentation, which require intensive face–to–face teaching. Our tools are restricted to 'island' applications where we consider them to be most effective. For many students (and teachers), such tools provide an additional level of motivation because of their easy accessibility, portability, and the absence of installation problems, and can be used with different levels of student accomplishment, from the relatively casual user in a foreign language or speech therapy context, to the more advanced linguistic programmer in courses on database or automata theory or software development.

For reasonably small scale applications,

we favour client–side tools where possible. JavaScript is suitable in many cases, provided that minor browser incompatibilities are handled. The database application, for example, still provided very fast access when evaluated with a 2000 record, 10 attributes per record database. JavaScript has a number of disadvantages (no mouse–graphics interaction, no library concept), but being an interpreted language is very suitable for introducing an 'open source code' policy in teaching. In contrast to CGI applications, where query and result transfer time can be considerable, client–side JavaScript (or Java) applications have a bandwidth dependent once–off download time for databases and scripts (or compiled applets), but query and result transfer time are negligeable.

The applications presented here are fully integrated (with references to related applications at other commercial and educational institutions, e.g. parsers, morphology programmes, speech synthesis demonstrations) into the teaching programme. Obvious areas where further development is possible and desirable are:

- Automatic tool interface generation based more explicitly on general principles of training methodology, e.g. with a more explicit account of TM space and with more systematic control, help, error detection, query and result panel design.
- Automatic test generation for tool (and student) validation.
- Further tools for formal language, parsing and automata theoretic applications.
- Extension of database tool to include more database functionality.

We plan to extend our repertoire of applications in these directions, and will integrate more applications from other institutions when they become available.

References

Carson–Berndsen, J. 1998a. Computational Autosegmental Phonology in Pronunciation Teaching. In: Jager S; J. Nerbonne & A. van Essen (eds.) *Language Teaching and Language Technology*, Swets & Zeitlinger, Lisse.

Carson–Berndsen, J. 1998b. *Time Map Phonology: Finite State Methods and Event Logics in Speech Recognition.* Kluwer Acadmic Press, Dordrecht.

Carson-Berndsen J. & D. Gibbon 1997. Interactive English, 2nd Bielefeld Multimedia Day Demo, "`coral.lili.uni-bielefeld.de/MuMeT2`", Universität Bielefeld, November 1997.

Carson-Berndsen J. & D. Gibbon 1999. Interactive Phonetics, Virtually! In: V. Hazan & M. Holland, eds., *Method and Tool Innovations for Speech Science Education. Proceedings of the MATISSE Workshop*, University College, London, 16–17 April 1999, pp. 17–20.

Gibbon, D. 1997. *Phonetics and Multimedia Communication*, Lecture Notes, "`coral.lili.uni-bielefeld.de/Classes/Winter97`", Universität Bielefeld.

Gibbon, D. in press. Computational lexicography. In: van Eynde, F. & D. Gibbon: *Lexicon Development for Speech and Language Processing.* Kluwer Academic Press, Dordrecht.

Hazan, V. & M. Holland, eds. 1999. *Method and Tool Innovations for Speech Science Education. Proceedings of the MATISSE Workshop*, University College, London, 16–17 April 1999.