TSNLP

Test Suites for Natural Language Processing

User Manual

Volume 2b

Test Suite Technology

University of Essex, UK
ISSCO, Université de Genève, Switzerland
Aerospatiale, France
DFKI, Germany

Preface

The TSNLP user manual summarises the results of the EU-funded project TSNLP (LRE-62-089, Test Suites for Natural Language Processing), which started in December 1993 and had a duration of 29 months. The project produced a methodology and tools for test suite construction in addition to substantial test suites in three languages - English, French and German (over 4000 items in each language). The test material covers central syntactic phenomena in the three languages and aims to be relevant to any developer or user of systems with grammar components who wish to test, benchmark or evaluate those systems. To allow for flexible access to the test suites they are mounted on a database.

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

- University of Essex, UK (Coordinators):[1]
  Doug Arnold, Lorna Balkan, Frederik Fouvry
- ISSCO, Switzerland:[2]
  Dominique Estival, Kirsten Falkedal, Sabine Lehmann, Hervé Compagnion
- Aerospatiale, France:[3]
  Eva Dauphin, Veronika Lux, Sylvie Regnier-Prost
- DFKI, Germany:[4]
  Klaus Netter, Stephan Oepen, Hans Uszkoreit

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

The user manual is in three volumes:

- Volume 1:
  Background, methodology, customisation and testing
  Editors: Lorna Balkan, Frederik Fouvry, Sylvie Regnier-Prost

[1] University of Essex, Wivenhoe Park, Colchester C04 3SQ, UK
[2] ISSCO, Université de Genève, 54, route des Acacias, CH 1227 Genève
[3] Aerospatiale, Centre Commun de recherches Louis-Blériot, Département Information Documentation, 12, rue Pasteur BP 76, 92152 Suresnes, Cédex, France
[4] DFKI, Stuhlsatzenhausweg 3, D 66123 Saarbrücken, Germany

- Volume 2
  Test suite technology
  Stephan Oepen, Frederik Fouvry, Klaus Netter, Tom Fettig, Fred Oberhauser, John Gregory.
- Volume 2b
  Test Suite Technology
  Editor: Frederik Fouvry
- Volume 3
  Test Data documentation
  Sabine Lehmann, Dominique Estival, Kirsten Falkedal, Hervé Compagnion, Lorna Balkan, Frederik Fouvry, Judith Baur, Judith Klein.

Volume 1 contains a Background chapter in which some of the factors which have influenced the design of the project are sketched.

The Methodology chapter gives a step by step account of how one can go about writing core data, that is, data that cover central phenomena of a language and that are intended to be applicable to a wide range of applications.

The Customisation chapter describes how the core data can be customised to a particular application (and sketches how it could be customised to a particular domain or text type).

The chapter on Testing gives an example of how the test suite can be applied to a real life evaluation scenario.

Volume 2 contains a description of the annotation scheme on which the data was constructed, the construction tool `tsct` used to create the data, the database `tsdb` on which the data is mounted.

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

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

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

L. Balkan,
CL/MT Group,
Department of Language and Linguistics,
University of Essex,
Wivenhoe Park,
Colchester,
CO4 3SQ
UK

Fax
+44 1206 872085

email
`balka@essex.ac.uk`

TSNLP deliverables, data and software can be accessed at the following address:

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

or via the ftp site:

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

for those who have ftp but not world-web access.

Overview

This volume of the TSNLP user manual describes the tools developed during the project, to automate the process of test item construction.

The volume contains three chapters:

Chapter 1 describes the Auto-TSG: the automatic test suite generation tool.

Chapter 2 describes the graphical user interface (GUI) to the test suite generation tool. This is also the user documentation for the GUI.

Chapter 3 describes the lexical replacement tool for changing the vocabulary of test suites.

Chapters 1 and 3, and this introduction are written by Doug Arnold, part 2 is written by Martin Rondell. The revisions[5] are made by Frederik Fouvry. As regards the Prolog/Lisp code, the code for the engine is due to Dave Moffat (who did the original implementation), Doug Arnold, and Martin Rondell. The GUI code is by Martin Rondell, apart from the code dealing with the viewing of trees. The Lisp code for the lexical replacement tool is due to Doug Arnold and Frederik Fouvry.

[5]This volume is a slightly revised and updated version of [3].

Contents

Chapter 1

Automatic Test Suite Generation: Engine

1.1 Introduction

The overall aim of this task was the development of a grammar based test-suite generation tool. We have called this tool *Auto-TSG*.

The Auto-TSG system is a Prolog program for automatically generating a test suite: a collection of test items (e.g. sentences) which can be used for evaluating Natural Language Processing (NLP) systems. The system has a relatively sophisticated (graphical) user interface which allows the user to (*inter alia*) create, view, and store test suite items. This interface is described in a separate part of the report (chapter 2). Here we describe the 'engine' of the system, the part actually responsible for interpreting a grammar to produce a test suite.

The system requires a *grammar*, which must be written by the user, and which defines (generates) a suite of test items—essentially, sentences or other expressions, with their surface structure (as implied by the grammar), and collections of annotations (again, as specified by the grammar). This grammar is written in a slightly extended context free notation. This notation is described in section 1.4. An example of such a grammar which produces a non-trivial test suite is given as Appendix B.

Given a grammar like the following, the system will produce a database of sentences such as those in figure 1.1, with their representations and certain annotations. The interface programs allow such a database to be saved, stored, etc. and the individual items displayed, inspected, and edited. The engine described here is responsible for creating the test suite items apart from the database identifier (the integer in the first field of the record), which is supplied by the interface programs.

```
s --->  np, vp, add_note(declarative).
np ---> [it].
vp ---> v, np ; [].
```

```
    v --->  [fired], add_note(past).
    v --->  [fires], add_note(pres).



% 1 % declarative past % it fired it
   % [s,[np,[it]],[vp,[v,[fired]],[np,[it]]]] %DATE:11/25/94
% 2 % declarative pres % it fires it
   % [s,[np,[it]],[vp,[v,[fires]],[np,[it]]]] %DATE:11/25/94
% 3 % declarative past % it fired
   % [s,[np,[it]],[vp,[v,[fired]]]] % DATE:11/25/94
% 4 % declarative pres % it fires
   % [s,[np,[it]],[vp,[v,[fires]]]] % DATE:11/25/94
```

Figure 1.1: Sample Database

The system is not intended for the linguistically or computationally naive, but the level of sophistication required to use it is not great. It should be easily usable by anyone who has some basic idea about Prolog, and who can write a phrase structure grammar.

The system is written in SICStus Prolog (2.1), the GUI uses the graphics management facilities which are part of SICStus library, and so requires X Windows.

This document is structured as follows. Section 1.2 motivates, and describes the background to, the idea of automatically creating test suites. Section 1.3 describes what the TS that the Auto-TSG system produces actually looks like. Section 1.4 describes the rule notation that the user employs to define test suite items: section 1.4.1 describes the rule syntax; section 1.4.2 describes how the rules are interpreted (*i.e.* what they mean). An example grammar which generates a non-trivial test suite is given as Appendix B. Section 1.4.2 describes the code: it is relatively simple, and transparent, and can be easily modified. No doubt the system could be used in many different ways, but it is developed with a particular use in mind, and seems to be appropriate in a particular context. This is discussed in section 1.6. Section 1.7 gives some conclusions and pointers to further work.

We use the following non-standard abbreviations:

TS	Test Suite
TSI	Test Suite Instance (*i.e.* an element in the test suite: e.g. a sentence with associated information and annotations).
TSG	Test Suite Grammar

1.2 Motivation and History

The idea of automatically generating TSs was first proposed by Arnold *et al.* ([2]), where the following points were noted:

- Writing TSs is quite difficult: it requires a certain level of linguistic expertise, and even so there are some things (such as the creation of ungrammatical examples) which are problematic.
- It is easy to make mistakes, for example including ungrammatical examples (which are not marked as such), or by missing out examples that are required if a TS is to be systematic.
- Despite its difficulty, much of the process of creating TSs by hand is very repetitive and boring.

The repetitive nature of the task, and the need for systematicity suggested it could (and should) be automated.

Of course, the difficulty of the task makes it unlikely that the process can be automated *completely*. However, the fact that TS writers must anyway have some linguistic sophistication means that they should be able to specify at least some of the knowledge that is required in a form that can be used automatically. So the possibilities of substantial automation looked good.

In fact, Arnold *et al.* (in [2]) compared the TS they produced automatically with one produced independently by hand (and used extensively in practice). They found that the automatically produced TS was better:

- The hand-produced TS was less systematic (it missed out some grammatical examples that should have been included);
- Despite having been checked (and used in practice), the hand-produced TS included some *ungrammatical* examples that should not have been included. These were not in the automatically produced TS.

The following are among the other potential advantages of the approach:

- It is very easy to customize or tailor an automatically created TS, at least as far as lexical coverage is concerned – one simply changes the lexicon, and re-runs the system.
- The TS grammar acts as a compact specification of a potentially very large (even infinite) TS, but is much easier to store and transmit.
- The grammar from which the TS is produced serves as a kind of self-documentation for the TS, showing relatively transparently and systematically what sorts of structures are tested for.

Of course, the Auto-TSG system and a grammar together constitute a form of NLP system, so one is proposing to use an NLP system to produce data to evaluate other NLP systems. At first glance, this idea has a certain air of paradox. After all, if one can get one NLP system right, one should be able to get the other one right, and perhaps the effort that goes into producing the TSG would be better spent in developing and refining the NLP system that one has set out to evaluate.

```
% 4
% declarative
  np
  np
% it fired it
% [s,[np,[it]],[vp(finite),[v(finite),[fired]],[np,[it]]]]
% DATE: 11/25/94
```

Figure 1.2: Example TSI

In fact, the idea of automatically generating TSs only to make sense if the following conditions are met:

1. Writing the TS Grammar must be *much* easier than writing the kind of NLP system one is testing.
2. It must be possible to design and produce a TS automatically from scratch in less time than it takes by hand (*i.e.* including writing and debugging the grammar, and editing the result).

In our (admittedly limited) experience, these conditions can be, and the approach is promising. Still, it should be admitted that despite some positive experience with automatic TS generation, it is not clear how sensible or widely applicable the approach is.

1.3 Output: the form of TSIs

Given a TSG, Auto-TSG produces a database of test suite instances TSIs. The GUI allows the individual TSIs and the database as a whole to be manipulated in various ways. The format of these TSIs is as follows.

ItemId	(an integer)
Annotations	(a collection of annotations)
Sentence	(a sentence, or other expression)
Structure	(a structural representation of the sentence)
DateStamp	(information about creation time, etc.)

The ItemId is supplied by the programs that load and save the TS (which are part of the GUI) and is not seen by the engine itself. The percent character(`%`) is used as a field separator. Figure 1.3 is an example: the sentence is *it fired it*, with annotations `declarative`, `np`, and `np` (*i.e.* there are two NPs). The representation is as in figure 1.3; it was created on 25th November.

Internally, the database is represented as a collection of facts of the form:

```
tsi([Notes,String,Structure,Date])
```

For example:

```
        s
   _____|_____
   |         |
   np    vp(finite)
   |      ____|_____
   it     |         |
     v(finite)    np
          |         |
        fired      it
```

Figure 1.3: Tree structure for *it fired it*

```
tsi([[declarative,np,np],
     [it,fired,it],
     [s,[np,[it]],[vp(finite),[v(finite),[fired]],[np,[it]]]],
     ['DATE: 11/25/94']
     ]).
```

This format is clearly something arbitrary, and restricted (e.g. annotations must be Prolog atoms), and something that a user might wish to customize. We have tried to design things so that this is not too difficult.

1.4 Grammar Rule Notation

The process of automatically writing a TS begins with writing a grammar (the TSG) which can be loaded by the Auto-TSG system and run. This section describes the grammar rule syntax, and how it is interpreted in Prolog.

1.4.1 Rule Syntax

The Auto-TSG rule syntax is very close to that of standard context free grammars, and the standard Definite Clause Grammar notation found in Prolog. It differs from the latter in the way it is interpreted, however (see below).

As a first approximation, rules may be of the following basic forms, where each C_i is a Category, and $TERMINALS$ is a Prolog *list* of zero or more terminals:

1. $C_0 \rightarrow C_1, C_2, \ldots, C_n$.
2. $C_0 \rightarrow C_1; C_2; \ldots, C_n$.
3. $C_0 \rightarrow TERMINALS$.

The first form is a conjunction of categories – the left-hand-side (Lhs) category (C_0) is assumed to expand as the sequence of categories on the right-hand-side (Rhs). The second form is a disjunction: the Lhs expands as any one of the Rhs categories. The third form introduces terminals (*i.e.* categories which cannot be rewritten).

For example:

```
s     ---> np, vp, pp.
np    ---> art, n.
art   ---> det ; quant.
det   ---> [the].
quant ---> [most].
v ---> [run,up].
```

These types of rule can be mixed, so the following is possible:

```
np ---> ( det ; art ), n.
np ---> ( [the] ; [most] ), n.
```

The first means that an `np` can consist of (be expanded as) either a `det` followed by an `n`, or an `art` followed by an `n`. The second means that an `np` can consist of *the* followed by an `n`, or *most* followed by an `n`.

More precisely: a rule consists of a Lhs and a Rhs, separated by a 3-dash arrow (‘`--->`’), and followed by a full-stop (‘.’). The Lhs can be any Prolog term.

The Rhs is either a single Category, or a sequence of Categories, separated by commas or semi-colons. A Category is one of the following:

1. A Prolog list (e.g. `[man]`, `[kick,the,bucket]`), which is taken to be a terminal or sequence of terminals;

2. A term of the form `add_note(N)`. N is interpreted as an *annotation*, and added to a list of annotations associated with with the Lhs category. For example, the following would add the annotation “`declarative`” to sentences produced with this rule.

   ```
   s ---> np, vp(finite), add_note('declarative').
   ```

3. A term of the form `remove_note(N)`. N is interpreted as an *annotation*, and removed from the list of annotations associated with with the Lhs category.

4. Any other kind of Prolog Term (for example `np`, `np(sing)`, or `np([num: sing, per: 3])`, etc.), which is taken to be a non-terminal.

5. Something enclosed in brace brackets, which is taken to be a Prolog goal, to be evaluated in the normal way. For example, the first of the following examples will expand an `s` as an `np` followed by a `vp`, with a call to the SICStus built-in commands `statistics/2` and `write/2` to write out some timing information on the user's screen in between. The second uses `write/1` to write messages indicating what stage has been reached on the standard output.

```
s ---> np,
       {statistics(runtime,[_,T]),
        write(user_output,T)},
       vp.
s ---> np, {write('done np')},
       vp, {write('done vp')}.
```

More sensible examples using this will be found in appendix B.

1.4.2 Interpretation

The notation differs from the standard DCG notation in a number of ways:

- Trivially, the arrow used is different (the DCG arrow consists of two dashes, the Auto-TSG one consists of three).
- Auto-TSG rules are not compiled into Prolog clauses, but used as written.
- The Auto-TSG interpreter takes care of building a surface structure or parse tree (a Prolog list, where the mother of a node is the first element, and the remaining elements are the daughters).
- The rules are applied in a 'breadth first' manner that makes left recursion unproblematic.
- Rhs elements of the form `add_note(N)` and `remove_note(N)` are interpreted specially.

In general, this means that DCG techniques, such as 'gap threading' or recasting iteration as recursion (as in the following example) work without problems.[1]

```
pps ---> pp.
pps ---> pp pps.
```

The interpreter begins with a category (the goal category – the category of TSI that is to be created: the user specifies this via the GUI), and an indication of the 'size' of item that is to be produced (again, specified via the GUI). TSIs containing alternative realizations of that category are produced by backtracking.

The size is stated in terms of the length of branches. For example, the tree in figure 1.4.2 has a maximum depth of 5 (this being the depth from S to *baby*).

Intuitively, the idea is that a category is expanded into a sequence of non-terminal categories and terminals according to the rules (which are applied in textual order): each non-terminal category is further expanded, until all that

[1] We have not implemented a Kleene star ('*') to allow iteration of categories: it does not seem necessary for test suite generation – one is not usually interested in arbitrary numbers of constituents in TSs.

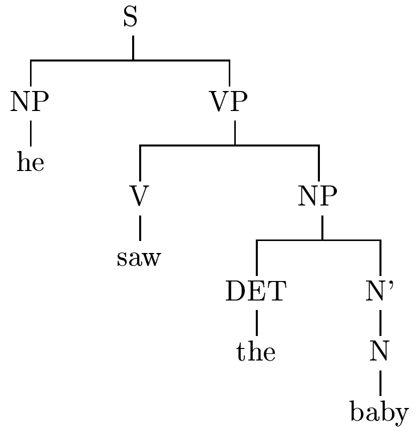

Figure 1.4: Example Tree

remains is a list of terminals – the string. At the same time: (i) a representation, reflecting the rule application, is generated (this is a Prolog list); (ii) a list of annotations is produced and maintained; and (iii) a notion of the current 'depth' is maintained, and compared with the maximum allowed: when the maximum allowed depth is reached, not further expansions of a category are allowed (and the system backtracks for more alternatives at this, and lower depths of embedding).

The choice of rule that is used to expand a non-terminal category involves matching/unification, as usual in Prolog. Thus, a non-terminal like `vp(finite)` can be expanded by the first of the following rules, but not the second, third or fourth:

```
vp(F) ---> ...
vp(non_fin) ---> ...
vp ---> ...
vp(F,Tense) ---> ...
```

Aside: There is one exception to this. The 'goal category' is interpreted 'flexibly' in the following way: suppose the goal category is `s`. First, all possible expressions of category `s/0` are sought, then all expressions of category `s/1`, `s/2`, etc. In effect, setting the goal category to `s` and generating is like setting the goal category to each of the following in turn, generating after each one:

```
s(_)
s(_,_)
s(_,_,_)
s(_,_,_,_)
```

This is achieved by forcing backtracking on a predicate `addargs(Functor, FunArgs)`, were Functor is the goal category specified by the user, and FunArgs is the one actually used by the system:

```
addargs(Functor,FunArgs) :-
```

```
   FunArgs = Functor
;  FunArgs =.. [Functor,_]
;  FunArgs =.. [Functor,_,_]
;  FunArgs =.. [Functor,_,_,_]
;  FunArgs =.. [Functor,_,_,_,_].
```

End of Aside.

As in standard Prolog, categories (*i.e.* goals) are expanded (*i.e.* solved) left to right, and depth first. However, as noted, unlike standard Prolog, the Auto-TSG interpreter maintains a depth measure to ensure that all structures of a certain size (*i.e.* depth) are produced before any of a greater depth. This means that features of normal Prolog that are undesirable in the context of test suite generation are avoided:

- Left recursive rules do not lead to looping. For example, the following grammar rules are unproblematic:

  ```
  np ---> np_possessive, n.
  np_possessive ---> np, ['''s'].
  ```

  (intended to produce expressions like *this boy's father*, *this boy's father's sister*, *this boy's father's sister's husband*, etc: `'''s'` means 'an atom consisting of an apostrophe and the letter `s`').

- right recursive rules to not lead to tedious repetition as chronological backtracking leads to further expansions of the rightmost daughter always being sought. Cf. with the following grammar, the standard Prolog interpretation will produce *the park near the park*, *the park near the park near the park*, *the park near the park near the park near the park*, etc, but never *a park near the park.*

  ```
  np --> det, n.
  np --> det, n, pps.
  det ---> [the].
  det ---> [a].
  n ---> [park].
  p ---> [near].
  pps --> [].
  pps ---> pp, pps.
  ```

This is achieved by the use of a depth counter (`max_depth`). This starts at the maximum depth of tree currently being created. It is decremented each time a rule applies, so applying a rule like the following at `max_depth` of 5 involves applying the rules for `np`, and `vp` at a `max_depth` of 4. When `max_depth` reaches zero, no rules can apply. (The user may also specify a minimum depth – TSIs whose trees have no branch that exceeds this are simply discarded).

```
s ---> np, vp.
```

The following are the relevant pieces of source code. We include it here (a) because we expect it to be helpful to potential users of the system (it is very simple and straightforward); and (b) because it makes clear that there is ample scope for customization.

The basic predicate is called (somewhat counter-intuitively) `parse/10`:

- Argument 1 is a category;
- Argument 2 (`Structure`) is a representation (a Prolog list).
- Argument 3 is the maximum depth of tree required.
- Arguments 4 (`Deep`) and 5 (`Deepest`) are an indication of the current depth, and the greatest depth so far reached.
- Arguments 6 and 7 (`S0`, `S`) represent the string (sentence) before and after this category is produced.
- Arguments 8 (`Star`) is a slot that could be used to indicate if an example is grammatical or not – it essentially just a distinguished annotation, and plays no real role in this version of the system.
- Argument 9 and 10 are lists of annotations (respectively, the annotations before this category is processed, and afterwards)

It is defined as follows:

- The first case is to deal with non-terminals: if `NT` is a non-terminal, we change the depth counters, choose an appropriate rule, and proceed with the Rhs of that:

  ```
  parse(NT,Structure,Max,Deep,Deepest,S0,S,Star,In,Out) :-
     Max > 0,               % Check max depth not reached
     Max2 is Max - 1,       % Decrement max depth counter
     Deep2 is Deep + 1,     % Increment depth counter
     grammar((NT--->Body)), % Choose a rule
     parse(Body,SubBodies,Max2,Deep2,Deepest,S0,S,Star,In,Out),
     combine(NT,SubBodies,Structure).
  ```

  `grammar/1` simply calls its argument, *i.e.* recovers a relevant rule, and `combine/3` simply builds up a structure.

- Dealing with conjunctions and disjunctions is straightforward:

  ```
  parse((X,Xs),Structure,As,D,Deepest,S0,S,Star,In,Out) :-
          parse(X,XBody,As,D,D1,S0,S1,Star,In,Mid),
          parse(Xs,XBodies,As,D,D2,S1,S,Star,Mid,Out),
          combine(XBody,XBodies,Structure),
          ( (D1 > D2,           % Pick greatest depth.
  ```

```
            Deepest = D1 );
           Deepest = D2 ).
```

```
parse((X;Y),Structure,As,D,Deepest,S0,S,Star,In,Out) :-
        parse(X,Structure,As,D,Deepest,S0,S,Star,In,Out) ;
        parse(Y,Structure,As,D,Deepest,S0,S,Star,In,Out).
```

- Terminals are simply added to the terminal string:

```
parse(Terminals,Terminals,_,D,D,S0,S,_,Notes,Notes) :-
        append(Terminals,S,S0).
```

- arbitrary Prolog goals are simply 'called':

```
parse({Trapdoor},[],_,D,D,S,S,_,Notes,Notes) :-
        (call(grammar(Trapdoor))).
```

- `add_note` and `remove_note` add, or delete items from the list of annotations.

```
parse(add_note(Note),[],_,D,D,S,S,_,Ns,[N|Ns]).

parse(remove_note(Note),[],_,D,D,S,S,_,NsIn,NsOut) :-
        delete(NotesIn,Note,NotesOut).
```

  Note that the use of `delete/3` here means that *all* instances of the annotation `Note` are removed from the list.

The top level predicate is the following:

```
tsg_engine(Cat,Arrows,MinDepth) :-
   addargs(Cat,CatArgs),
   parse(CatArgs,Structure,Arrows,0,Deepest,Sentence,[],Star,[],Notes),
   Deepest >= MinDepth,
   create_tsi([Star,Notes,Sentence,Structure]).
```

Where `create_tsi` essentially just adds some simple documentation (creation date), and asserts a term of the following form, providing that this does not duplicate an existing TSI:

```
tsi([Notes,String,Structure,Date])
```

1.5 Filter

We originally thought that some method for *filtering* the output of a TSG would be useful and important. A *filter* in this sense is a process that checks the output of the TSG and performs various operations on it, e.g. adding information ("this example is ungrammatical") to a TSI, or perhaps removing the TSI completely. In particular, we thought that there might be things which might be quite difficult to do with a TSG, but which might be relatively easy with a filter of some kind. For example, filtering apparatus might allow a user to change the output of the grammar without understanding the grammar itself at all (but only the output of the grammar).

For this reason, we originally envisaged, and implemented a filtering mechanism which allowed the user to control what happened to TSIs, depending on their shape of the representations they contained (in particular, users could specify whether they were marked as ungrammatical or not, and whether they were preserved or discarded). In this implementation, the 'shape' of a representation was simply a matter of what it would unify with. Thus, specifying a filter involved writing down Prolog terms which would unify with representations, and specifying an action to be taken (e.g. to discard the representation). Thus, (roughly) the following would throw out (cause to be discarded) VPs with an initial V, where the V was separated from the NP by another (exactly one) item:

```
filter([vp,[v,|_],_,[np|_]|_],discard).
```

However, on reflection, this approach seems to us to be wrong:

- Writing the sorts of Prolog term required is quite tricky, and hence unlikely to be of much practical use.
- In any case, it provides too weak a method for selecting TSIs (e.g. one might want to use regular expressions over representations, or look for patterns of annotation).
- We contemplated writing more sophisticated tools (e.g. an integrated regular expression interpreter in Prolog), but the (considerable) effort involved did not seem worthwhile (or compatible with the time scale of the work package).

Instead, we now think that 'filtering', if needed, should not be carried out inside the Auto-TSG, but should be performed with ordinary (e.g. Unix) tools on the external database, and ultimately by hand (*i.e.* with an editor).

Thus, though we have included a filtering mechanism in this prototype, it is *very* primitive, and is mainly intended as a hook onto which more sophisticated apparatus could be attached.

1.6 Methodology/Usage

The Auto-TSG has been developed with a particular kind of use in mind.

You are developing some kind of NLP system. The issue of test data should be of concern from the very beginning: assuming the system is in some way 'rule based', then test data for each construction should be produced as the construction is described (as the rules are written). Of course, this data can be produced by hand, or by your NLP system itself. However, there are several reasons why you might not want to do this:

- you may want the test data before you have fully implemented your description;
- you will want negative (*i.e.* ungrammatical) examples, which you do not expect your system to produce;
- your system may not have a generation capacity.

Instead, you use a system like the Auto-TSG. You use it from the very beginning to produce the very simplest sentences (*John saw Mary*, etc.) Writing a TS grammar is very simple, and takes almost no time, because:

- the constructions you are describing are very simple,
- you already understand the constructions you are describing
- the rule notation is very simple,
- you know Prolog,
- you do not care about efficiency (the TS grammar will only be run once a week, or once a month, etc),
- you do not care about elegance or understandability of the grammar (it is not going to be used for anything else, after all),
- you do not care about all the overgeneration that is usually a problem – this is all useful (negative) test data – and anyway, it is easy to go through the TS database that is produced by hand and remove the really silly examples.

As you proceed to more and more complex constructions, most of these points remain valid.

The basic idea is that you will probably run the Auto-TSG at relatively long intervals (days or weeks), depending on your development and test cycle. (This is one reason to give it a very friendly user interface – the user cannot be expected to be very familiar with the system).

Of course, the TS produced in this way does not have everything you want of a TS. You want your TSIs punctuated (*i.e.* with capitals, commas, full stops, etc.) or annotated in some particular way; you want some TSIs marked as ungrammatical, and you may want to associate various representations with them (e.g. semantic representations, translations, etc.). Now one idea would be to complicate the TSG to do this, e.g. have the TSG itself construct the semantic representation. We have provided some simple annotation adding facilities, but in general, using the TSG for any more than very simple annotations seems us to be a bad idea. There is absolutely no reason to think that building (e.g.) semantic representations should be any easier with the Auto-TSG rules than it is with any other notation (specifically, the notation of your system). Instead, you should use the Auto-TSG to produce a basic data set, which is then augmented by hand with whatever other information you may wish. When you expand your TS grammar, and rerun the Auto-TSG, you will, of course generally produce this basic data set again – pointlessly, since you already have it, with extra annotations and information. This repeated data can simply be discarded.

Of course, there are other ways the system (*i.e.* Auto-TSG) can be used. For example, rather than holding TSs as actual databases, one could simply hold the TSG, and generate the TS afresh each time it is needed. For example, rather than querying an existing database for all NPs, one could simply have the Auto-TSG construct NPs from scratch. However, this is not the use we have in mind, mainly because the we think the amount of information that can be automatically and reliably associated with TSIs is not very large (little more than a representation of surface syntactic structure, and some annotations).

1.7 Developments and Open Questions

Though the system is delivered as a working prototype, there are a number of unanswered questions:

- Before a TSI is added to the database, a check is made that this does not duplicate an existing TSI. Currently, this treats all parts of a TSI equally (apart from the unique identifier, which is not produced by the engine at all). Thus, e.g. TSIs which are identical apart from their creation date are treated as different). This is probably too restrictive a notion of identity. However, it is not clear what notion should be used.

- The role and specification of the filtering apparatus is unclear.

- The system is produced with a particular use in mind. It is not clear what the optimal functionality is for this use (e.g. should the system provide more scope for user defined annotations; should the system itself provide other annotations, e.g. about language). Are there other sorts of rule type that would be generally useful?

- Finally, it should be admitted that the jury is still out on the whole idea of automatically creating test suites. Though our experience is encouraging, it is still not really clear that writing a grammar to produce a basic TS (which is then edited by hand) is really easier than producing one completely by hand.

The creation of the test suites in the project,[2] provided an opportunity to use and evaluate the tools. We reached the following conclusions:

- Auto-TSG certainly improves the systematicity of the test suite, and it proved very useful for full coverage of closed class items.
- It would have been better if Auto-TSG had been developed in closer relation with the annotation scheme and the test suite database tsdb (see [4]): it would allow the test suite writer to generate a certain number of annotations along with the items. This would save effort later on in the construction process.
- The tool is useful for creating big test suites, or small test suites that one wants to extend later on. It is too time consuming for very small test suites (e.g. less than 30 items). But when one uses the Auto-TSG, it will almost certainly generate a test suite of over 50 items, because of the idea of generalisation and systematicity it is based on.

[2]The construction of the test suites followed the development of the tools described in this volume.

Chapter 2

Graphical User Interface to Auto-TSG

2.1 Introduction

The Automatic Test Suite Generation Package (Auto-TSG) is a graphical user interface to the Test Suite Generator (TSG).[1] The Auto-TSG was designed to aid the user in generating and viewing Test Suite Instances (TSIs) via the TSG. It was programmed in SICStus Prolog and runs under the X Windowing System both of which are available on a wide variety of platforms.

The package was designed to be as modular as possible so that the user is able to modify parts of the package. For example the TSG supplied may be replaced by the users own TSG, as well as modifying the grammar and filters that are included.

Although more or less a prototype system the Auto-TSG should meet the needs of users wishing to build large test suite packages.

Please send comments, suggestions etc to `rondm@essex.ac.uk`.

2.2 Software Requirements

Both the X Windowing System and SICStus prolog 2.1.8 are required to run the Auto-TSG. The former is freely available from MIT[2] and the latter freely available to the academic community.[3] The Prolog versions higher than 2.1.9 use a different graphical manager, not compatible with the one in which the Auto-TSG interface is written. The TSG Engine code can be reused.

[1] The TSG is described in chapter 1.

[2] As well as a variety of archive sites

[3] SICStus was developed at the Swedish Institute of Computer Science, PO BOX 1263, S-164 28 KISTA, Sweden.

2.3 Obtaining the Auto-TSG

The Auto-TSG is freely available from the CL/MT Research Group's anonymous ftp server, `clftp.essex.ac.uk` (155.245.89.4) in the file `/tsnlp/`.

Both compressed and uncompressed versions of the archived Auto-TSG package reside in that directory. After downloading the package it can be installed anywhere on the system. Simply move the Auto-TSG package to the desired location and unarchive it with the following command:

```
$ gunzip -c auto\_tsg.tar.gz | tar xf -
```

or

```
$ uncompress -c auto\_tsg.tar.Z | tar xf -
```

2.4 Starting the Auto-TSG

To start the Auto-TSG first change directory to the directory where the Auto-TSG has been installed. Then start SICStus prolog and load the `auto_tsg.pl` file which should be located at the top level directory.

```
$ sicstus
SICStus 2.1 #8: Fri Apr 23 16:15:42 BST 1993
| ?- compile(auto_tsg).
{compiling ....
 .....
yes
| ?-
```

The `auto_tsg.pl` file will load all the necessary files to run the Auto-TSG including a sample grammar.

Once the Auto-TSG has loaded successfully you can start the graphical user interface with the following prolog goal:

```
| ?- gui.
```

The main Auto-TSG window should now appear.

2.5 The Main Auto-TSG Window

The Main Auto-TSG window see figure 2.1 is divided into five separate areas:

- **The Message Window**, which is the top section of the window.

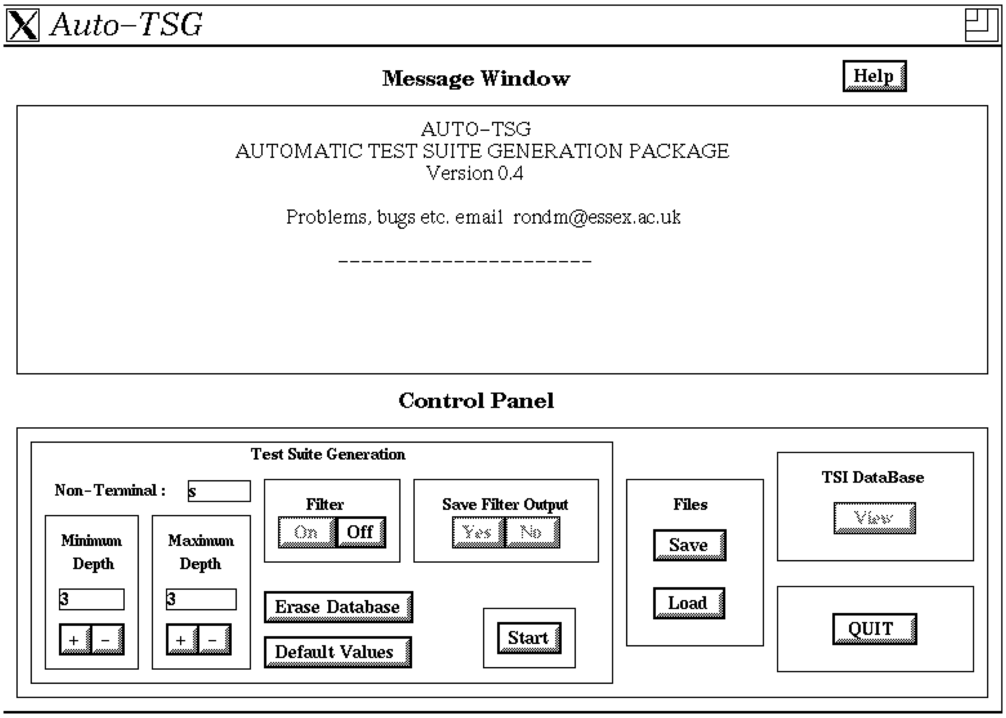

Figure 2.1: The Main Auto-TSG Window

- **The Test Suite Generation buttons**, located at the bottom left of the window.
- **The File buttons**, located near the centre of the bottom of the window.
- **The TSI Database button**, located near the bottom left of the window.
- **The QUIT button**, located at the bottom left of the window.
- **The HELP button**, located at the top right of the window.

Each of these areas is described in more detail below:

2.5.1 The Message Window

The Auto-TSG displays the majority of it's messages to the user through this window. It initially contains the title and version number of the current Auto-TSG program.

During the course of using the Auto-TSG it's very likely that a large number of messages will be displayed, although only approximately ten lines are displayed at once. However, it is possible for the user, if they wish, to review previously displayed messages. The user should press and hold the right-hand mouse button in the Message Window. A small hand icon will appear allowing the user to drag the displayed text back and forth.

2.5.2 The Test Suite Generation Buttons

Using these buttons the user can start the TSG as well as set various parameters which are passed to the TSG controlling the type of TSIs produced. Initially the default settings are show, however these settings can be changed and each setting is described in more detail below.

The Non-Terminal Setting

Towards the right of the 'Non-Terminal' title is a small box which initially contains an 's'. This is the non-terminal category that is passed to the TSG and indicates what category of test suite item the TSG should produce. The user can select any non-terminal by positioning the mouse pointer over the box and clicking the left-hand mouse button. A vertical line should appear after the last character in the box notifying the user that they can now modify the box's contents by typing the characters at the keyboard. By pressing the delete key the previous contents can be erased. Once a non-terminal has been selected, press the return button and the vertical line should disappear. Unfortunately no checks are yet made to make sure the entered characters are actually a valid non-terminal.

Minimum and Maximum Depth Setting

Initially both the minimum and maximum depth settings are set to three. The user can modify these in two separate ways. Firstly by entering the number directly in the input box (see previous subsection) or by clicking the mouse pointer on the '+' or '-' buttons. The '+' button will increment the current depth setting by one and the '-' button will decrement the current depth by one.

Note that the maximum depth setting cannot be lower than the minimum depth setting, nor can either depth by lower than one. Any depth that contradicts this will be disallowed and an appropriate message will be displayed in the message window. If an invalid setting is entered into the input box then the previous valid setting is automatically retained.

Erase Database Button

This button erases all the TSIs that so far been generated. An appropriate message is displayed in the message window.

Default Values Button

This button sets the non-terminal and depth settings to their default values. An appropriate message is displayed in the message window.

Filter Buttons

Initially no filtering of the generated TSIs take place, this is reflected by the 'Off' button being highlighted whereas the 'On' button is grey in appearance. To turn the filtering mechanism on click on the 'Off' button, the 'Off' button will now appear grey while the 'On' button is highlighted. To turn the filtering mechanism off click on the 'On' button.

The filtering mechanism itself is described in a later section.

Save Filter Output Buttons

These buttons are only activated (or highlighted) when the filtering mechanism has been turned on (see previous subsection). The default setting is that the filter output isn't saved, this is reflected in that the 'No' button is highlighted whereas the 'Yes' button is grey. To change this and to save the filter output click on the 'No' button, the 'Yes' button will now be highlighted and output now saved.

Start Button

To actual start the TSG and produce TSIs click on the 'Start' button. All the settings will be passed to the TSG which will now start to generate TSIs. Appropriate messages will be displayed in the message window informing the user of this and how many TSIs have been generated.

Note that depending on the settings and the TSG being used this process may take quite some time.

2.5.3 The File Buttons

There are two file buttons one to save files and one to load files. Depending on which button is pressed a pop-up window appears whereby the user selects an appropriate file and it's type. See section of loading and saving files.

2.5.4 The TSI DataBase Button

By clicking on this button a pop-up window appears in which the user can modify and view the TSIs that have been generated by the TSG. Initially this window is grey in appearance reflecting the fact that no TSIs have been generated. See section on viewing generated TSIs.

2.5.5 The HELP Button

If the user wishes to review the these instructions from within the Auto-TSG then the user should simply chick on this button. A pop-up window will appear showing this section of the user manual.

2.5.6 The QUIT Button

If the user wishes to quit the the Auto-TSG then simply click on this button. Another pop-up window appears to confirm that the user wishes to quit. If the answer is 'yes' the user is returned to a Prolog prompt.

2.6 Viewing the Generated TSIs

To view the TSIs generated by the TSG click on the 'View' button in the Main Auto-TSG window. Please note that the view will remain grey, that is inactive, until TSIs exist in the internal database either through generation or by loading previously generated TSIs.

A large window should appear, see figure 2.2, entitled TSI Database. This window is divided into five different areas, these areas are:

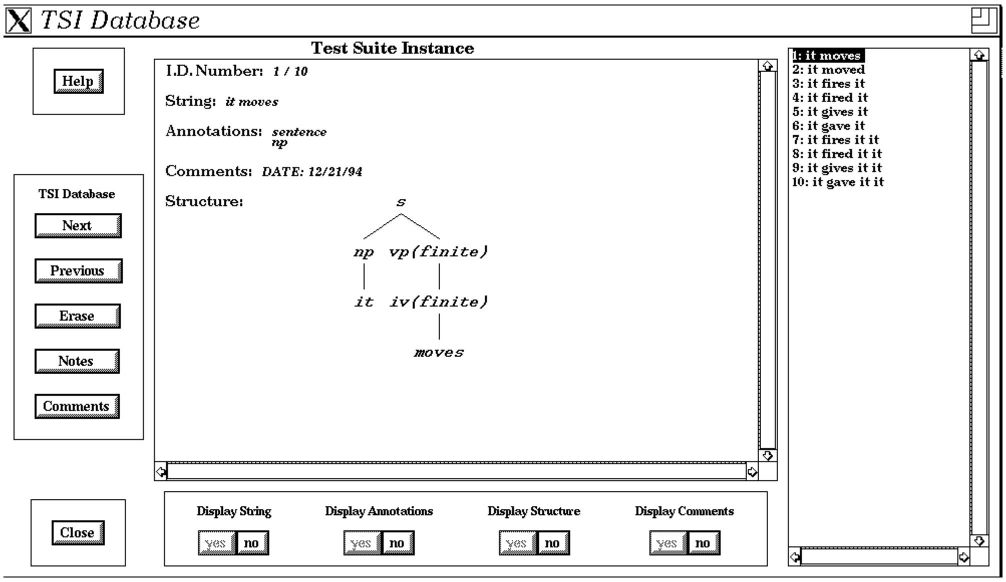

Figure 2.2: The View Window

- **Test Suite Instance Window**, located in the middle of the window.
- **The Overview Window**, located on the right-hand side.
- **The TSI Database manipulation buttons**, located on the left-hand side.
- **The TSI Information Selector buttons**, located beneath the TSI Window.
- **The Close Button**, located at the bottom left of the window.
- **The Help Button**, located at the top left of the window.

Each of these areas is described in more detail below:

2.6.1 The Test Suite Instance Window

The information generated on each TSI is displayed in this window. This information includes:

- **The ID Number**, each TSI generated has unique identification number.
- **String**, the string that was generated.
- **Annotations**, annotations generated from the grammar.
- **Comments**, comments about the TSI which currently contains the date when the TSI was generated.
- **Structure**, the tree structure of the string.

It's possible to display only certain parts of the TSI's information by setting certain flags. These flags are set via from the TSI Information Selector buttons (see later subsection). By default only the ID number, string and annotations are displayed.

Note that ID number is followed by the number of TSIs current in the database, so for example an ID Number displayed in the TSI Window such as 1/10 would mean that the currently displayed TSI's unique id number is 1 and there are 10 TSIs in total in the database.

2.6.2 The Overview Window

Within this window is a list of all the TSIs in the database represented by the TSIs id number followed by it's string. The TSI currently being show in the TSI window is highlighted. To view a different TSI simply click the left mouse button anywhere on the line and that TSI will be displayed. Scroll bars exist to allow you to move quickly through large TSI databases.

2.6.3 The TSI Database manipulation buttons

These buttons allow the user to manipulate the TSIs in the database. There are currently five of these buttons:

- **Next**, displays the next TSI in the database.
- **Previous**, displays the previous TSI in the database.
- **Erase**, will erase the currently selected TSI from the database.
- **Notes**, allows the user to add or modify the TSIs annotations.
- **Comments**, allows the user to add or modify the TSIs comments.

Both the 'Notes' and 'Comments' buttons generate a pop-up window in which the current TSIs annotations or comments will be displayed. This window is a simple text editor that will allow the user to add or modify the TSIs information.

2.6.4 The TSI Information Selector buttons

These buttons determine which parts of the TSIs are actually displayed in the TSI window. The user can simply click on the currently highlighted box to select or deselect part of the TSI.

2.6.5 The Help Button

If the user wishes to review the these instructions from within the Auto-TSG then the user should simply chick on this button. A pop-up window will appear showing this section of the user manual.

2.6.6 The Close Button

This button closes the TSI Database pop-up window and returns the user to the TSI Main Window.

2.7 Loading/Saving Files

2.7.1 Loading Files

To load a previously saved TSI database, a new grammar, or new filters the user should click on the 'Load' button in the main Auto-TSG window.

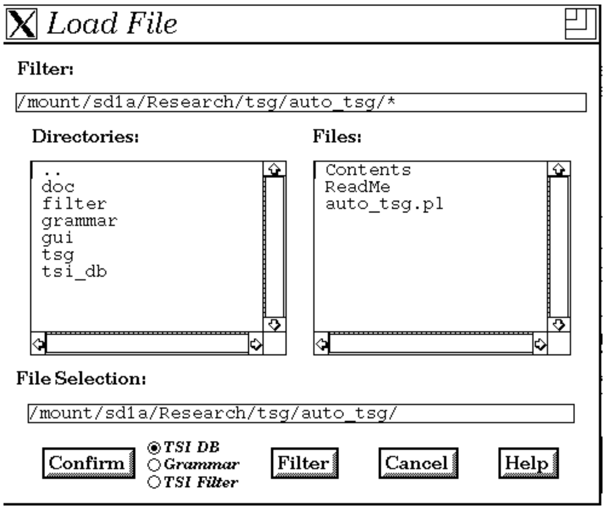

Figure 2.3: The *Load File* Window

A pop-up window should appear entitled 'Load File' (figure 2.3). The window allows the user to select files in different directories on the disk. This pop-up window is divided into four areas:

- **Filter**, located at the top of the window.

- **Directories and Files**, located in the center of the display.
- **File Selection**, located below the 'Directories and Files' windows.
- **Load File Buttons**, located at the bottom of the display.

Each area is described in more detail below.

Filename Filter

The Filename Filter input box initially contains the current working directory followed by the default filter '`*`'. The filename filter allows the user to specify which files in the current directory should actually be displayed. The '`*`' matches all possible files in the current directory. By altering the filter the user is able to display only the files matching a certain pattern, e.g. `*.pl` would display only files with a `.pl` extension.

The filter is only applied if the user presses return after entering the filter in the input box or clicking on the Filter button itself.

Directories and Files

Initially these two windows will display all the directories and files in the current working directory. By clicking on a file in the 'Files Window' that particular file can be selected and is placed in the File Selection input box. Alternatively, clicking on directories in the 'Directories Window' will change the current working directory and both windows and input boxes will be updated.

File Selection

The File Selection box contains the name of the file the user has chosen to be loaded when either the return key is pressed in the Input Box or the Confirm button is pressed. If an inappropriate file has been chosen, *i.e.* a directory, then an appropriate message will be displayed above the Input Box.

Load File Buttons

At the bottom of the screen are four separate buttons:

- **Confirm Button**, selects the file currently found in the File Selection Input Box.
- **File Type**, the type of the file that the user wishes to load is selected here by clicking on the appropriate file type.
- **Filter Button**, selects the filter currently found in the Filter Selection Input Box.

- **Cancel Button**, cancels the load request and removes the pop-up window returning the user the the main Auto-TSG window.

- **Help Button**, generates a pop-up window displaying this section of the user manual.

2.7.2 Saving Files

To save a TSI database the user should click on the 'Save' button in the main Auto-TSG window.

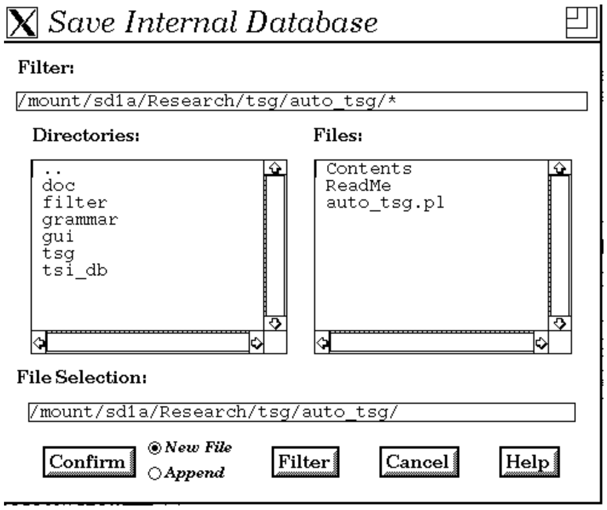

Figure 2.4: The *Save Internal Database* Window

A pop-up window should appear entitled 'Save Internal Database' (figure 2.4). The window allows the user to select files in different directories on the disk. This pop-up window is divided into four areas:

- **Filter**, located at the top of the window.

- **Directories and Files**, located in the center of the display.

- **File Selection**, located below the 'Directories and Files' windows.

- **Load File Buttons**, located at the bottom of the display.

The only difference between the actions of these areas and those in the 'Load File' window is that there are two file types:

- **New File**, overwrites or creates a new file.

- **Append**, appends the TSI database to the contents of the selected file.

2.8 The Filter Mechanism

At present only a very simply filter mechanism has been included and needs to be expanded. Currently if a atom is found within a specified part of the TSI then that TSI is either removed or copied to a file. For example if the user wishes to filter out all TSI's string information that contain a particular word *i.e.* moves then the user should create and load a filter of the following form:

```
filters(string,moves).
```

The filter is able to match against any part of the TSI by selecting the appropriate key:

- **string**, matches against the TSI's string.
- **notes**, matches against the TSI's annotations.
- **tree**, matches against the TSI's tree structure.
- **comments**, matches against the TSI's comments.

Chapter 3

Lexical Replacement tool

3.1 Introduction

The general topic of this part of the project was the design of a tool intended to make easier the customization/tailoring of Test Suites (TSs), by facilitating changes to the lexical coverage of a TS. In particular, such a tool is intended to make it easier to *extend* the lexical coverage of a TS, by creating new test suite items (TSIs) which are like existing ones, except in the lexical items they contain. Section 3.2 gives some general discussion of what this entails.

It was our initial intention to simply build such a tool. However, it was obvious that this work should be closely related to the tsct developed by DFKI, and if possible actually integrated with it. Though this was not possible (for reasons of timing, *inter alia*), we decided that we should nevertheless produce a tool that was compatible with the DFKI tool at the level of input and output, so that they could be used in conjunction. However, since a closer integration is desirable, we have also tried to make sure that what we have produced constitutes a realistic, if informal, runnable specification of what such a tool should do.

In the event, the approach we have taken is to define an emacs 'mode' which provides the basic functionality we think is required. This is described in section 3.4, which also serves as the 'user manual' for the tool as it exists.

3.2 General Ideas

The user has a TS which is designed to test NLP systems in a particular area: Telecommunication, say. The user wants to change the lexical coverage to test a system that deals with some other subject area, agricultural texts, for example. Alternatively, the user may want to extend the lexical coverage—to include additional TSIs. This second case is just a generalization of the first, where both the new TSIs and the old ones they were based on are preserved.

It is clear what this should involve, principle. Assume for simplicity that a TSI is simply a pair $< S, R >$, where S is a sentence and R a "representation" of some kind, for example a tree representation (or a set of the same), a semantic formula, a collection of annotations stating which phenomena S exemplifies, and comments (perhaps including documentation about the source of the original sentence, if it was naturally occurring), or in the simplest case just a mark indicating whether the sentence is grammatical or not. Of course, in general, a TSI will actually be an n-tuple $< \ldots, S, r_1, r_2, \ldots, r_n >$, where $r_1, r_2, \ldots, r_n$ are the separate parts of this "representation", and the "sentence" may be any kind of expression, and need not be the first element. However, this simplification will not matter here.

The general process for creating new TSIs would be:

1. a TSI $< S, R >$ is chosen;
2. a word (or phrase) W in S is selected to substitute for;
3. a suitable item (W') is found which can be substituted for W;
4. S' is produced from S, using W';
5. the "representation" (R') that corresponds to S' is created;
6. the new item $< S', R' >$ is checked and added to the TS;
7. the change is generalized across the whole TS (*i.e.* W' is substituted for W in every TSI, which appropriate updating of the associated representation).

A significant subset of the operations required are those provided by a normal text editor—in particular, the substitution operation, and the generalization across the whole database are reminiscent of local and global replace commands. Several of these step involve things that can only reasonably be done by the user. Of the others, the final step does not seen to involve anything new—it is just a generalization or iteration of early steps. The tricky steps are steps 3, 4, and 5: finding the word to substitute (W'); creating the new sentence (S'); and creating the new representation (R').

Taking these in reverse order, the problem of creating the new representation may involve all sorts of complication, or may be very simple (e.g. $loves(j, m)$ becomes $hates(j, m)$ as one substitutes *hates* for *loves*). It is difficult to see how it could be automated *in general*, though it is easy enough to see that something can be done in simple cases.

Similarly, as regards creating the new sentence, once a new word W' has been chosen and manipulated so it has the right form, it should normally be a relatively simple matter to create S' from S by substitution of W by W'—the kind of thing that an ordinary word processor allows. Of course, there may also be changes required in S (e.g. some verbs require particular case marking on their objects—if this case marking property is regarded as negotiable, then W and W' may differ in it, and it may be necessary to change S to allow for it). Nevertheless, the central problem seems to be that of getting W' itself.

As regards this task, we should begin by distinguishing what one might call *contextual* properties of items. These are properties an item has purely because of where it appears. Hence they are properties that any replacement item should inherit. For example, in the following, *stinks* is third person singular because of its subject, any verb we choose in its place should also be third singular.

(1) Everything stinks.

Among the other properties we should distinguish are those that are *fixed*, and those that are *negotiable*: the fixed properties are also ones we want replacements to inherit (for example, if we want to make only grammatical TSIs, we must make sure that we only choose verbs that can be used intransitively in place of *stinks*). The negotiable ones are the ones that we do not care about: they define a 'space' within which alternatives are possible. For example, we may not care about the tense of replacements, so (2) is an acceptable alternative to (1).

(2) Everything stank.

Obviously, there must be some negotiable properties, otherwise only the only possible substitution for W is W itself.

Choosing the item to substitute (W') involves the following:

- finding the properties of W
- deciding which properties of W are contextual, which negotiable, and which fixed
- finding another word with the same fixed properties
- manipulating it so it fits the context (*i.e.* has the right contextual properties, e.g. has the right case and agreement marking)

In general, finding the properties of an item requires either that the properties of the sentence it appears in are already known (*i.e.* that it is already parsed, as in a tree bank), or that they are found out (*i.e.* it is parsed then and there, or the user is consulted).

Deciding what properties are negotiable is something only the user can do.

Finding other words with the same fixed properties requires access to some linguistic knowledge source (e.g. a dictionary or thesaurus, or the user herself).

Manipulating the word so it fits the context requires some kind of morphological generator (or interaction with the user).

The following gives an interesting view on how one might allow users to define the negotiable properties in a very straightforward way. In essence, it is a method by which the negotiable properties of an item can be inferred. The idea comes from work on 'authoring tools' for writing language drills in Computer Aided Language teaching ([1]).

The user is presented with (or chooses) a particular TSI, and a particular word (W) within that item, and then provides an alternative word W' whose whose substitution would produce another acceptable TSI. The idea is that from this the system should be able to infer the range of possible substitutions by considering the properties that are common to W and W', and from this produce appropriate TSIs. To take a rather silly example, suppose the TSI is

(3) Sam ate a herring

the user focuses on *herring*, and indicates that an appropriate substitution would be *flounder*. Both herrings and flounders are kinds of edible unsmoked sea fish—the system might be able to infer that any other sort of edible unsmoked sea fish is a reasonable substitution, and so produce:

(4) Sam ate a flounder

(5) Sam ate a cod

(6) Sam at a plaice

etc.

If the user specified *kipper* as an acceptable substitution, the system might be able to infer that *any* kind of sea fish (smoked or not) would be possible. If the user specified *lobster*, the possibilities might be generalized to all sea creatures. Of course, this is only possible if the system has access to the relevant knowledge about herrings (etc). In this case, this might be included in a thesaurus of some kind.

However, the approach can be generalized to other properties. For example, if the original word chosen, and the replacement suggested are both singular count noun, or both verbs with a particular subcategorization code, a system might be able to infer that any such items are possible, and generate alternatives on the basis of this.

In principle, all this requires is either an appropriately structured knowledge source (e.g. a hierarchically structured thesaurus or dictionary—the task is then to find the node that dominates both the original item and its suggested replacement, and then to enumerate all the other items that are dominated by that node), or a knowledge source that allows 'two-way' access (e.g. from words to their properties, and from properties to the associated words). In practice, such knowledge sources are not as easily available as one might think, and it is much easier to leave most of this job to the user. So the idea remains an interesting and appealing possibility at present.

3.3 Designing a Lexical Replacement Tool

It seems desirable that a tool of this kind have the following properties:

Integration It should be integrated into a general tool for manipulating TSs. For reasons described at the start of the document, we have not attempted this.

Flexibility (1) It should not be closely tied to any particular TS, or database format; at least, it should be easily customizable for different formats.

Flexibility (2) The tool should be flexible as regards the knowledge sources it is to be used with (e.g. the particular dictionaries, thesauri, and languages).

Interactivity Many of the tasks required are rather sophisticated, and not obviously automatable, and the user is in any case the final arbiter of what should and should not be done: a good deal of interaction will be required.

Tool Box Approach (1) Building the tool should not duplicate work done elsewhere in NLP (a morphological analyzer/generator, and lexicon is required – these should not be built specially).

Tool Box Approach (2) Since much of the functionality required is that of a text editor, the tool should probably be based on one.

A considerable amount of time (given the shortness of this part of the work package) was spent in investigating morphological analyzers/generators and lexica with a view to integrating them with this tool (at least as options). However, it was clear that (a) no single tool is remotely adequate, even to a reasonable sub-set of the task (for example, no morphological processor seems equally able to analyze and generate, or interface generally with other tools); and in any case, no single tool would be generally useful for the different languages. In any case, it was clear that one could manage without such tools providing one relies on the user to supply the expertise, and that even placing this burden on the user one could produce a usable and useful tool.

Given this, and constraints on time, our plan was to implement an emacs 'mode' which would provide the basic functionality of interaction and textual manipulation. Tasks that cannot comfortably be performed inside emacs (or which can be more comfortably performed outside emacs) would be performed outside emacs. In particular, the main task of the emacs mode would be to deal with interaction with the user. Access to external knowledge sources would be provided by external calls from emacs, but in general and by default, the user would be expected to provide most of the knowledge, and would be expected/allowed to confirm system generated choices at every point.

The functionality and operation of this tool is described in the following section.

3.4 Test Suite Mode: Description and User Manual

Here we describe the structure operation of the emacs *Test Suite Mode* that we have implemented. This is a collection of emacs functions/commands (with names beginning with `tsm-`) which facilitate modification of the lexical content of test suite. The description is also intended to serve as a rather crude user manual. In addition, on-line documentation for functions and variables is provided in the usual way for emacs, and the code is fairly readable (we hope). The user is expected to be familiar with the use of emacs for ordinary editing tasks, and in particular for 'search and replace' type tasks (cf. the emacs command `query-replace`). The user is also expected to have at least some idea about setting emacs variables, and about shell scripts, though the mode can be used at a basic level without this knowledge.

It should be stressed that our aim in writing this was simply to to make the job of test suite construction *eas***ier** than it would otherwise be (*i.e.* we do not claim this tools makes any part of the task *easy*).

3.4.1 Overview

The basic idea is that the user will have a test suite consisting of sentences such as:

1. He left
2. She left
3. It left
4. They left

which he/she wants either to change into a test suite where *left* is replaced by some other verb, such as *arrived*, or to expand into a test suite which contains sentences with *arrived* as well. The user may then want to further amend or expand the test suite by replacing *He*, *She*, etc. with names.

The user can choose which items to replace, and can choose the replacements. However, a simple interface is provided which will allow programs external to emacs to suggest replacements. An example of such an external program might be *Wordnet*, or other thesaurus program; a corpus program, or an on-line dictionary. Test Suite mode assumes that the external program name is a single word (*i.e.* a Unix command without options—see below), and makes rather slight assumptions about the form in which this external program returns its results.

As regards the results, what is expected is just collection of words, each of which may be considered as a potential replacement (a program that returns other things will still work, but may ask the user some relatively silly questions). These words are displayed in a special buffer, and for each such word, the user is asked whether he/she want to use the word as a replacement. If no words are chosen (or, equivalently, if the external program did not return any potential replacements), the user is asked to supply one.

Thus, suppose the user is looking at a buffer containing at TS like that above, in Test Suite mode. If he/she gives the emacs command `tsm-lex-replace`. The following will happen:

- The user is asked what word he/she wants to change (a guess is made, and the user is asked to confirm it: if he/she answers "no"), he/she is asked to specify the word. Suppose the user chooses *left*
- An external (Unix) program will be executed with *left* as its argument. Which program this is depends on an emacs variable which the user can set. Suppose, however, the program is a shell script that returns a list of words with similar meaning on its standard output (e.g. vacate, renounce, resign, abandon, give up).
- These words will be displayed in an emacs buffer, and for each one, the user will be asked "Do you want to use X as a replacement to *left*?"
- For each word that the user answers "yes", and for each test suite item (TSI) emacs performs a search and replace operation, replacing *left* by the chosen word.
- This search and replace is *interactive* in the normal way, so the user has the choice of rejecting a substitution or editing the result (which is necessary in this case, since the proposed replacements are in the wrong form).
- If a change is made, the user is asked whether the original TSI should be discarded, or whether both the original and the new TSIs should be kept.

There are just a few of complications:

- For compatibility with the DFKI test suite construction tool, test suites are assumed to be stored in a special form; they are displayed to the user in a more readable version of this form.
- The user can specify an emacs 'filter' which will be applied to the word that is originally chosen for replacement. Such a filter might be an emacs function that removes punctuation, for example. By default, no such filter is used.
- The external command must be a single word (e.g. it cannot be an Unix command with command line options). It should take the word to be substituted as its argument. The effect of this is that the user must be prepared to construct a shell script that takes care of all options, etc. For example, suppose you would like to use `wn` (which is the core program in *Wordnet*) to find antonyms of the verb *left*. The normal command you would give for this would be something like:

  ```
  wn left -antsv
  ```

  To use this with Test Suite mode, you must define a sell script that looks as follows:

```
#!\bin\sh
wn $1 -antsv
```

If this script is in an executable file called (say) `antsv`, then setting the emacs variable that defines the external command (it is called `tslr-command`) to `anstv` will have the desired effect.

This may seem a limitation, but we do not think so (at least at this stage). The idea is that the user develops and tests the programs he/she wants to use to access the external knowledge base outside emacs, and only later uses it in connection with a Test Suite. The problem with a more general approach is that it is very difficult to predict the form in which external programs will expect their arguments (notice, e.g. that `wn` violates the Unix norm by taking its obligatory argument before it optional 'flags'. Other external commands may expect their arguments to come in a file). Moreover, the output of external programs is even less predictable, and the user's script should take care of this too (e.g. `wn` gives various bits of information along side the actual words:

```
2 of 10 senses of leave

Sense 4
leave, take leave
       => arrive, come, reach a destination

Sense 10
leave, leave unchanged, leave unaltered
       => change, undergo a change, become different
       => change, alter
```

This other information should be discarded. It is asking too much of a tool like Test Suite mode to do this for arbitrary programs.

3.4.2 Starting up—Loading

To begin with, the relevant code must be loaded. Assuming it is contained in a file called `tsm.el`, this can be achieved by the emacs command `load-file` (e.g. by typing `ESC-x` or `META-x`, followed by "`load-file`", followed by "`tsm.el`").

3.4.3 Basic Commands/Functions

The following commands then become available, though the ordinary user will only be interested in the first two:[1]

`ts-mode` set up the mode

`tsm-lex-replace` the main function

`tsm-get-word-to-replace` choose a word W to replace

`tsm-get-potential-replacements` choose words Ws to replace it with

`tsm-query-replace-one-item` interactively replace one word by another, asking whether to replace the original test suite item with the new one, or to keep both

`tsm-query-replace-buffer` Do the same over the whole buffer

`tsm-do-replacements` Do the same, but for a whole list of candidate replacements

`tsm-save-buffer` save a test suite buffer

`tsm-save-and-kill` save a test suite buffer and kill it

`tsm-save-and-kill-all` save and kill all test suite buffers

`tsm-write-file` write a test suite out to a file

`tsm-format-buffer` make a test suite file readable by humans

`tsm-unformat-buffer` make it readable by machine

`tsm-find-file` find and load a test suite (making it readable by humans)

Some keys are redefined so that there is less chance of the user messing up the test suite format so the file becomes unusable by other programs (see below).

3.4.4 Test Suite Format

Externally, test suites are held in a form like figure 3.1, inherited from the DFKI test suite construction tool (each TSI occupies a single line, with fields separated by "`@`"). We will call this format "database format". Database format is not very readable by humans. Thus, we use a slightly different format inside Emacs, as in figure 3.2. We will call this *tsm-format*.

```
Frederik Fouvry@1/12/94@invented@formal@none@1@1@sentence@It went .@1@
3@@1:1@It@np@arg-subj@2:2@@@2:2@went@vp@functor@1:2@@@3:3@.@punct@@@@
```

Figure 3.1: Database Format

Both formats are customizable by setting the values of the following variables:

[1]The following emacs functions are also provided, but are simply utilities used by others, and not relevant to the user:

`tsm-get-current-word`

`tsm-get-word-list`

`tsm-delete-backward-char`

```
Frederik Fouvry
@1/12/94
@invented
@formal
@none
@1
@1
@sentence
@It went .
@1
@3
@
@1:1
@It
@np
@arg-subj
@2:2
@
@
@2:2
@went
@vp
@functor
@1:2
@
@
@3:3
@.
@punct
@
@
@
@
```

Figure 3.2: Test Suite Mode (`tsm-`) Format

`tsm-DBFS` The value of the field-separator in the database. It is assumed to be "`@`" by default.

`tsm-UFS` The value of the field-separator as shown to the user (*i.e.* in tsm-format). Its default value is "`@`" preceded by a carriage return (so each field is shown on a new line).

`tsm-DBRS` The value of the record-separator in the database. Its default value is "`\n`" (*i.e.* carriage return, a line ending).

`tsm-URS` The value of the record-separator as shown to the user. Its default value is "`\n`" (*i.e.* two carriage returns, a blank line, in effect).

Two functions exist for formatting and unformatting buffers:

`tsm-format-buffer` Changes the format of current test suite buffer so it is more easily readable, *i.e.* puts a buffer in database format into emacs/tsm format.

`tsm-unformat-buffer` Does the reverse job, *i.e.* restores a buffer to internal database form.

Both operate on whole buffers, and it is possible that versions which work on single TSIs, or regions might be useful as well. Both are relatively crude. A variable `tsm-buffer-formatted` is used to record whether a buffer has been formatted or not.

Both functions are basically very simple: they just make two passes over the buffer: (i) changing `tsm-DBRS` into `tsm-URS`, and (ii) changing `tsm-DBFS` into `tsm-UFS` (or vice versa).

3.4.5 Loading and Saving TS files

The fact that test-suite format and database format are different means that special commands are required for loading and saving TS files.

Saving TS Files

There are three functions for saving files

`tsm-save-buffer` Saves the current test suite buffer. This is the basic function. It is responsible for saving the buffer without its ts-mode formatting. It does this in the simplest and most obvious way—by removing the formatting, saving the file in the conventional way, and then restoring the format.

`tsm-save-and-kill` Saves the current buffer and kills it.

`tsm-save-and-kill-all` Saves and kills all test suite buffers.

There is also a function `tsm-write-file`, which allows a buffer to be saved to a file (like ordinary emacs `write-file`, on which it is based). This prompts for the name of the file in which to save the current buffer.

Loading TS Files

Test suite files are loaded by means of the function `tsm-find-file`. This loads the file in the normal way, put the buffer into `ts-mode`, and formats it.

3.4.6 The Mode Command (ts-mode)

The function `ts-mode` puts a buffer into *test suite mode*. It is mainly responsible for :

```
^X^s   tsm-save-buffer
^X^w   tsm-write-file
^X^f   tsm-find-file
"@"    forward-char
\n     next-line
^M     next-line
^J     next-line
\177   tsm-delete-backward-char
```

Figure 3.3: Test Suite Mode Keymap

- setting the mode name;
- installing the special keymap;
- setting various standard emacs variables such as `sentence-end`, `paragraph- start` and `page-delimiter` in a way that is useful for test suites. In particular, the `page-delimiter` is set so that each TSI occupies a separate page. This is useful, since it makes it easy to focus on just one TSI at a time when performing the lexical replacements.

The keymap redefines keys as in figure 3.3. The main effect is to make the keys normally used for loading and saving files invoke the special test suite mode commands, and to disable keys that would insert record or field separators. The delete key is bound to a command that deletes anything except these separators. (Of course, there are still plenty of ways the user can mess up a test suite.)

3.4.7 Main Functions

The central function of the package is `tsm-lex-replace`. This calls two functions in turn:

`tsm-get-word-to-replace` This allows the user to choose the word he/she wants to replace. The default is to replace the word under the point. The user is asked to confirm or reject this choice, and if it is rejected, the user is prompted to provide the word he/she wants.

`tsm-get-potential-replacements` Constructs a list of potential replacements for the chosen word. It does this by running the command that is the value of the variable `tslr-command` with the chosen word as an argument. Whatever results are returned are displayed in a buffer entitled `*Replacements*`, and the user is presented with them a word at a time, each time with the option of accepting or rejecting the item. If no items are chosen (or indeed, if none were ever proposed by the external program), the user is asked to supply one.

`tsm-do-replacements` This takes the word that has been chosen to be replaced, and the list of one or more replacements. For each replacement, it performs a query-replace on the whole buffer, using the function `tsm-query-replace-buffer`.

`tsm-query-replace-buffer` This steps through the test suite buffer, one item at a time (other TSIs actually become invisible), calling the following function. Some statistics are maintained about number of items created, changed, and left unaffected, and these are reported at the end.

`tsm-query-replace-one-item` This looks at a single item at a time (with other TSI actually being invisible), and does the following:

- saves a copy of the current item
- performs a query replace of the word chosen for replacement by the word currently being used to replace it. This is done using the standard emacs `query-replace` (see below)
- if a change has been made to the current item, the user is asked if he/she wants to keep both the original, and the changed item (*i.e.* whether he/she is adding items to the test suite, or just changing the ones that are there).

The fact that standard emacs `query-replace` is used mean, first, that the user is expected to be familiar with this. Second, it means that the user has a considerable amount of control over what happens:

- the user is able to do various things as each replacement is suggested, as indicated by the following message (this message is printed in response to a "?" at the appropriate point):

```
Type Space or 'y' to replace one match, Delete or 'n' to skip
    to next,
ESC or 'q' to exit, Period to replace one match and exit,
Comma to replace but not move point immediately,
C-r to enter recursive edit (ESC C-c to get out again),
C-w to delete match and recursive edit,
C-l to clear the screen, redisplay, and offer same replacement
    again,
! to replace all remaining matches with no more questions,
^ to move point back to previous match.
```

- search and replacement are sensitive to the emacs variables `case-fold-search` and `case-replace`, which have whatever values the user normally has.

3.5 Conclusion

As it stands, the Lexical Replacement tool we have described and implemented has a number of limitations and drawbacks. There are also a number of open questions about its design and use, which are mainly the result of it not having been used "in anger" yet.

Two limitations seem to us unproblematic, given its immediate intended use: namely, the fact that it is built on to of, and so requires emacs, and the fact that it assumes a Unix environment.

For reasons described at the beginning, the tool is not integrated in a more general environment for handling Test Suites. It is, on the other hand, very flexible: it makes almost no commitments about the kinds of external knowledge source it can be used with. It is perhaps too liberal here: a clearer commitment to particular knowledge sources (and programs to access them) might make it easier to use in practice.

One question which is quite unclear is what the right nesting of the various loops is, and what the right points for interaction are. As regards the former point, the current state is that the user chooses some replacement items, and each TSI is processed with this replacement in mind, and only after all TSIs have been looked at is another replacment item considered. Perhaps users would find it easier to look at all potential replacements with respect to one TSI, and only then go on to the next TSI. As regards the second point, for example, as things stand, when the external program returns a list of replacements, the user is invited to make a choice, but is not allowed to edit the items (only accept or reject them). Perhaps it would be useful if there was also an option to edit at this point.

3.6 Hindsight remarks on the lexical replacement tool

After using the tool, tsdb and tsct, we have the benefit of hindsight. We thought it therefore useful to add the following remarks.

The usefulness of the tool depends to a large extent on the size and quality of the dictionary which it consults. tsdb itself provides a dictionary interface which can be exploited by the lexical replacement tool to extend the test suite.[2]

The loop nesting, referred to in the previous section, is currently demanding for the user. It would make life easier if all replacements were done at once, followed by a dialogue with the user to decide what items to keep. It would also be better to take only the sentences and make the replacements in there, not throughout the entire record. The adaptation of records to reflect the changes in the string would be made during the process of writing the changed/added records. This results in the following procedure:[3]

1. indicate the word to be replaced;

[2] It will however be clear to the reader that there is a limitation to this method (*i.e.* using the test suite as a dictionary): you can only reuse words and constituents that are already in the test suite.

[3] The steps between square brackets are to be performed by the program only.

2. [look up replacements (using an appropriate lookup script—this can be changed or adapted in test-suite mode);]

3. [make *all* replacements;]

4. interact with the user to decide on items (or rather input string) to keep. Possibly a choice of ordering principle should be offered: order the items on the basis of the input string or on the basis of the (replaced and replacing) words that were involved in the replacement;

5. [write the new test suite file.]

Appendix A

Shell Scripts

The simple shell script `lookup` is useful for testing Lexical substitution in Test Suite mode. It gives the kind of output one might expect from a thesaurus. Given a single word as argument, it looks down its left column (LeftC) to find something that matches the word, and then prints the word in the right column (RightC):

```
% lookup rain
hailed
snowed
drizzled
sleeted
```

```
#!/bin/sh
# syntax: lookup Word
# Given a single word Word in the command line, it looks it up
# in the left hand list of words and
# returns the associated items in the right hand list
nawk '{ if ( $1 ~ Pattern ){ print $2 } }' Pattern=$1 <<!
     #LeftC RightC
     Monday Tuesday
     Monday Wednesday
     Monday Thursday
     Monday Friday
     Monday Saturday
     Monday Sunday
     rained hailed
     rained snowed
     rained drizzled
     rained sleeted
     comes goes
     walked strolled
     walked sauntered
     he she
```

```
    he it
!
```

The following are scripts for accessing Wordnet `wn`. The first recovers nominal antonyms for single argument, the second recovers nominal synonyms.

```
#!/bin/sh
wn $1 -antsn
```

```
#!/bin/sh
wn $1 -synsn
```

And for this script, you need `tsdb(1)`. It uses the test suite present as dictionary.

```
#! /bin/sh

for el in $(tsdb -query=\"select a-category
                           where a-instance == \"$1\"")
do
    tsdb -query="select a-instance
                 where a-category == \"$el\""
done
```

Appendix B

Example Grammar for Auto-TSG

```
% Grammar and lexicon intended to generate all permutations
% of 1, 2, and 3-place predicates, plus some stuff
% for to-deletion,
% plus passives, progressives, modality and negation.
% It will be simple to add number features to these rules.

s ---> np, vp(finite).

% NP Rules: only one NP at present
np ---> [it].

%% VP Rules

vp(Form) ---> iv(Form).
vp(Form) ---> v(Form), np.
vp(Form) ---> v(Form), pp1.
vp(Form) ---> v(Form), np, pp1.
vp(Form) ---> v(Form), np, np.     %%To-deletion
vp(Form) ---> v(Form), pp1, pp2.   %%HACK
vp(Form) ---> v(Form), pp2, pp1.   %%HACK
vp(Form) ---> aux(Form/Required), vp(Required).

%%Negative equivalents

vp(Form) ---> iv(Form), negp.
vp(Form) ---> v(Form), negp, np.
vp(Form) ---> v(Form), negp, pp1.
vp(Form) ---> v(Form), negp, np, pp1.
vp(Form) ---> v(Form), negp, np, np.    %%To-deletion
vp(Form) ---> aux(Form/Required), negp, vp(Required).

%%Main verbs

iv(Form) ---> [IV], {iv(IV, Form)}.
```

```
iv(moves, finite).
iv(moved, finite).
iv(move, nonfinite).
iv(moving, pres_part).
iv(moved, past_part).

v(Form) ---> [V], {v(V, Form)}.
v(fires, finite).
v(fired, finite).
v(fire, nonfinite).
v(firing, pres_part).
v(fired, past_part).

v(gives, finite).
v(gave, finite).
v(give, nonfinite).
v(giving, pres_part).
v(given, past_part).

%%Auxiliaries

aux(Form) ---> [Aux], {aux(Aux,Form)}.

%% i.e. "/" as in Categorial grammar == "requires"

aux(can, finite/nonfinite).
aux(could, finite/nonfinite).
aux(may, finite/nonfinite).
aux(might, finite/nonfinite).
aux(shall, finite/nonfinite).
aux(should, finite/nonfinite).
aux(will, finite/nonfinite).
aux(would, finite/nonfinite).
aux(have, nonfinite/past_part).
aux(has, finite/past_part).
aux(had, finite/past_part).
aux(is, finite/pres_part).
aux(was, finite/pres_part).
aux(is, finite/past_part).             %%Passive
aux(was, finite/past_part).            %%Passive
aux(been, past_part/pres_part).        %%Passive
aux(being, pres_part/past_part).       %%Passive
aux(be, nonfinite/pres_part).
aux(does, finite/nonfinite).

negp ---> neg.
neg ---> [not].

pp1 ---> [to], np.             %%HACK:don’t need ‘to’ in lexicon
pp2 ---> [by], np.             %%HACK: don’t need ‘by’ in lexicon;
                               %%would have interfered with active PP rules
```

Appendix C

Example Database for Auto-TSG

The following is part of an example database produced by the grammar given in Appendix B.

```
% 1
% sentence
  np
% it moves
% [s,[np,[it]],[vp(finite),[iv(finite),[moves]]]]
% DATE: 11/25/94

% 2
% sentence
  np
% it moved
% [s,[np,[it]],[vp(finite),[iv(finite),[moved]]]]
% DATE: 11/25/94

% 3
% sentence
  np
  np
% it fires it
% [s,[np,[it]],[vp(finite),[v(finite),[fires]],[np,[it]]]]
% DATE: 11/25/94

% 4
% sentence
  np
  np
% it fired it
% [s,[np,[it]],[vp(finite),[v(finite),[fired]],[np,[it]]]]
% DATE: 11/25/94

% 5
% sentence
```

```
  np
  np
% it gives it
% [s,[np,[it]],[vp(finite),[v(finite),[gives]],[np,[it]]]]
% DATE: 11/25/94

% 6
% sentence
  np
  np
% it gave it
% [s,[np,[it]],[vp(finite),[v(finite),[gave]],[np,[it]]]]
% DATE: 11/25/94

% 7
% sentence
  np
  np
  np
% it fires it it
% [s,[np,[it]],[vp(finite),[v(finite),[fires]],[np,[it]],[np,[it]]]]
% DATE: 11/25/94

% 8
% sentence
  np
  np
  np
% it fired it it
% [s,[np,[it]],[vp(finite),[v(finite),[fired]],[np,[it]],[np,[it]]]]
% DATE: 11/25/94

% 9
% sentence
  np
  np
  np
% it gives it it
% [s,[np,[it]],[vp(finite),[v(finite),[gives]],[np,[it]],[np,[it]]]]
% DATE: 11/25/94

% 10
% sentence
  np
  np
  np
% it gave it it
% [s,[np,[it]],[vp(finite),[v(finite),[gave]],[np,[it]],[np,[it]]]]
% DATE: 11/25/94
```

Bibliography

[1] Atsuko Shioya, *Example-based Drill Generation for Japanese CALL*, MSc Dissertation, CCL, Dept. of Languages and Linguistics, UMIST, 1993.

[2] D.J. Arnold – Dave Moffat – Louisa Sadler *et al.* (1993), *Automatic Test Suite Generation*, in: *Machine Translation*, 8, pp.29–38.

[3] Doug Arnold – Martin Rondell – Frederik Fouvry. *Design and implementation of test suite tools. Report to LRE 62-089 D-WP5.1*, University of Essex, UK, December 1994.

[4] TSNLP. University of Essex, ISSCO, Aerospatiale, DFKI. *User Manual. Volume 2: Test Suite Technology*, forthcoming 1996.