ACL-05

# Workshop on Software

## Proceedings of the Workshop

30 June 2005
University of Michigan
Ann Arbor, Michigan, USA

Production and Manufacturing by
*Omnipress Inc.*
*Post Office Box 7214*
*Madison, WI 53707-7214*

# Introduction

Welcome to the ACL Workshop on Software, the first of its kind. It is intended as a venue for discussing and comparing the implementation of software and algorithms used in Natural Language and Speech Processing. The goal is to bring together researchers, software developers, teachers, and students with a common interest in the implementation of NLP applications, and to allow useful implementation techniques and “tricks of the trade” to be discussed in detail and disseminated widely.

We received 13 submissions, of which 8 were selected for presentation and inclusion in the proceedings, after a careful review process. Because the number of reviewers exceeded the number of submissions, each submission received more than four reviews on average, while the workload per reviewer was less than three papers on average. This being the workshop on software, the initial assignment of reviews was performed algorithmically using the Ford–Fulkerson max-flow algorithm, while taking into account individual reviewer preferences. The reviewers did an admirable job dealing with a diverse set of submissions, for which they deserve the thanks of the community.

The papers presented in this workshop deal with many different aspects of NLP software: Carpenter describes a scalable implementation of high-order character language models; Clegg & Shepherd take three existing parsers that were trained on business news text and perform a comparative evaluation on a corpus of biomedical journal papers; Cohen-Sygal & Wintner have implemented a compiler which translates between the description languages of two different finite state toolboxes; Foster has designed a generation module for a dialogue system which can ship out text without having to wait for the planning phase to finish; Koller & Thater describe the intelligent design of increasingly powerful constraint solvers; Newman proposes a uniform formalism for representing the output of parsers for easy inspection and comparison; Trón, Gyepesi, Halácsy, Kornai, Németh & Varga have implemented a generic library for analyzing orthographic words; and White discusses the design of a generation component which flexibly incorporates language models in a syntactic surface realizer.

The workshop proceedings are being made available in electronic form only. Not only does this save costs, but it also allows the distribution of additional software and resources that could not be included in printed proceedings. A number of authors have included the software described in their papers directly on the proceedings CD. As always, the latest versions of the included software can be found on the Internet or by contacting the individual authors.

I would like to thank the reviewers and authors once again for their hard work and look forward to an exciting workshop.

Martin Jansche
Columbia University, New York

# Table of Contents

# Workshop Program

**Thursday, June 30, 2005**

| | |
|---|---|
| 9:25–9:30 | Opening Remarks |
| | **Session 1: Parsing** |
| 9:30–10:00 | *TextTree Construction for Parser and Treebank Development*<br>Paula S. Newman |
| 10:00–10:30 | *Evaluating and Integrating Treebank Parsers on a Biomedical Corpus*<br>Andrew B. Clegg and Adrian J. Shepherd |
| 10:30–11:00 | Break |
| | **Session 2: Generation and Semantics** |
| 11:00–11:30 | *Interleaved Preparation and Output in the COMIC Fission Module*<br>Mary Ellen Foster |
| 11:30–12:00 | *Designing an Extensible API for Integrating Language Modeling and Realization*<br>Michael White |
| 12:00–12:30 | *The Evolution of Dominance Constraint Solvers*<br>Alexander Koller and Stefan Thater |
| 12:30–2:00 | Lunch |
| | **Session 3: Morphology and Language Modeling** |
| 2:00–2:30 | *Hunmorph: Open Source Word Analysis*<br>Viktor Trón, György Gyepesi, Péter Halácsy, András Kornai, László Németh and Dániel Varga |
| 2:30–3:00 | *Scaling High-Order Character Language Models to Gigabytes*<br>Bob Carpenter |
| 3:00–3:30 | *XFST2FSA: Comparing Two Finite-State Toolboxes*<br>Yael Cohen-Sygal and Shuly Wintner |

# TextTree Construction for Parser and Treebank Development

**Paula S. Newman**
newmanp@acm.org

## Abstract

TextTrees, introduced in (Newman, 2005), are skeletal representations formed by systematically converting parser output trees into unlabeled indented strings with minimal bracketing. Files of TextTrees can be read rapidly to evaluate the results of parsing long documents, and are easily edited to allow limited-cost treebank development. This paper reviews the TextTree concept, and then describes the implementation of the almost parser- and grammar-independent TextTree generator, as well as auxiliary methods for producing parser review files and inputs to bracket scoring tools. The results of some limited experiments in TextTree usage are also provided.

## 1 Introduction

The TextTree representation was developed to support a limited-resource effort to build a new hybrid English parser[1]. When the parser reached significant apparent coverage, in terms of numbers of sentences receiving some parse, the need arose to quickly assess the quality of the parses produced, for purposes of detecting coverage gaps, refining analyses, and measurement. But this was hampered by the use of a detailed parser output representation.

The two most common parser-output displays of constituent structure are: (a) multi-line labeled and bracketed strings, with indentation indicating dominance, and (b) 2-dimensional trees. While these displays are indispensable in grammar development, they cannot be scanned quickly. Labels and brackets interfere with reading. And, although relatively flat 2D node + edge trees for short sentences can be grasped at a glance, for long sentences this property must be compromised.

In contrast, for languages with a relatively fixed word order, and a tendency to post-modification, TextTrees capture the dependencies found by a parser in a natural, almost self-explanatory way. For example:

```
They
must have
    a clear delineation
        of
            [roles,
             missions,
             and
             authority].
```

Indented elements are usually right-hand post-modifiers or subordinates of the lexical head of the nearest preceding less-indented line. Brackets are generally used only to delimit coordinations (by [...]), nested clauses (by {…}), and identified multi-words (by |…|).

Reading a TextTree for a correct parse is similar to reading ordinary text, but reading a TextTree for an incorrect parse is jarring. For example, the following TextTree for a 33-word sentence exposes several errors made by the hybrid parser:

```
But
    by |September 2001|,
the executive branch
    of
        [the U.S. government,
         the Congress,
             the news media,
         and
         the American public]
had received
   clear warning
       that
          {Islamist terrorists
           meant
              to kill
                  Americans
                     in high numbers}.
```

[1] The hybrid combines the chunker part of the fast, robust XIP parser (Aït-Mokhtar et al., 2002) with an ATN-style parser operating primarily on the chunks.

*Proceedings of the ACL 2005 Workshop on Software*, pages 1–13,
Ann Arbor, June 2005. © 2005 Association for Computational Linguistics

TextTrees can be embedded in bulk parser output files with arbitrary surrounding information. Figure 1 shows an excerpt from such a file, containing the TextTree-form results of parsing the roughly 500-sentence "Executive Summary" of the 9/11 Commission Report (2004) by the hybrid parser, with more detailed results for each sentence accessible via links. (Note: the links in Figure 1 are greyed to indicate that they are not operational in the illustration.)

Such files can also be edited efficiently to produce limited-function treebanks, because the needed modifications are easy to identify, labels are unnecessary, and little attention to bracketing is required. Edited and unedited TextTree files can then be mapped into files containing fully bracketed strings (although bracketed differently than the original parse trees), and compared by bracket scoring methods derived from Black et al (1991).

Section 2 below examines the problems presented by detailed parse trees for late-stage parser development activities in more detail. Section 3 describes the inputs to and outputs from the TextTree generator, and Section 4 the generator implementation. Section 5 discusses the use of the TextTree generator in producing TextTree files for parser output review and TextTreebank construction, and the use of TextTreebanks in parser measurement. The results of some limited experiments in TextTree file use are provided in Section 6. Section 7 discusses related work and Section 8 explores some directions for further exploitation of TextTrees.

6 (3) We have come together with a unity of purpose because our nation demands it. best more chunks

```
    We
    have come together
        with a unity
            of purpose
        because
            {our nation
             demands
                 it}.
```

7 (15) September 11 , 2001 , was a day of unprecedented shock and suffering in the history of the United States . best more chunks

```
    |September 11 , 2001|,
    was
        a day
            of
                [unprecedented shock
                 and
                 suffering]
                     in the history
                         of the United States.
```

8 (1) The nation was unprepared . best more chunks

```
    The nation
    was
        unprepared.
```

Figure 1. A TextTree file excerpt

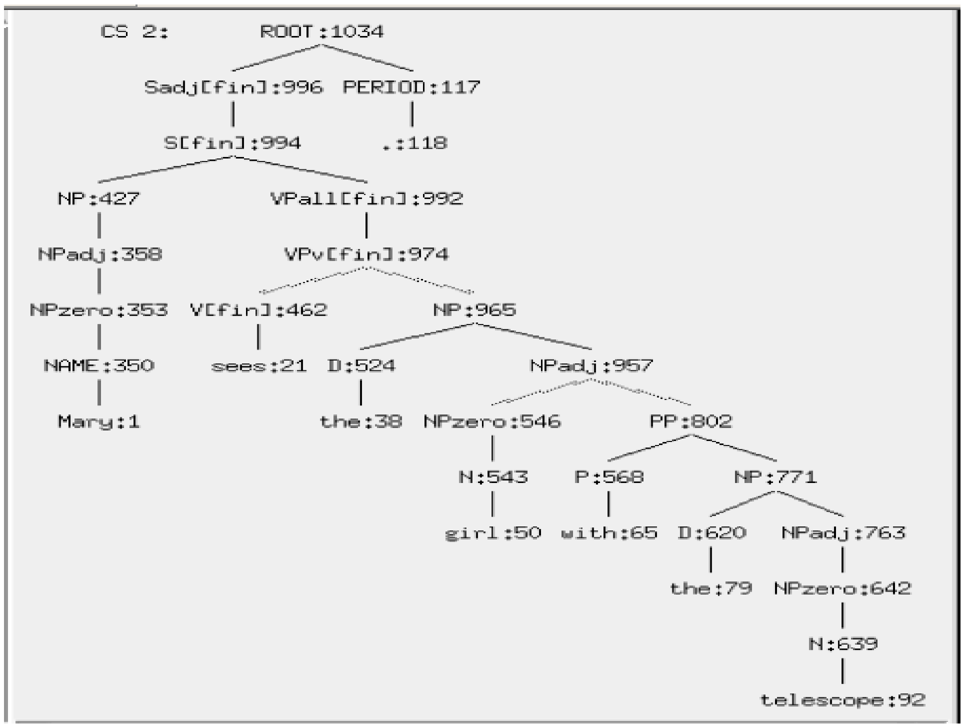


Figure 2. An LFG c-structure

## 2 Problems of Detailed Parse Trees

This section examines the readability problems posed by conventional parse tree representations, and their implications for parser development activities.

As noted above, parse trees are usually displayed using either 2-dimensional trees or fully bracketed strings. Two dimensional trees are intended to provide a clear understanding of structure. Yet because of the level of detail involved in many grammars, and the problem of dealing with long sentences, they often fail in this regard. Figure 2 illustrates an LFG c-structure, reproduced from King et al. (2000), for the 7 word sentence "Mary sees the girl with the telescope." Similar structures would be obtained from parsers using other popular grammatical paradigms. The amount of detail created by the deep category nesting makes it difficult to grasp the structure by casual inspection, and the tree in this case is actually wider than the sentence.

Grammar-specific transformations have been used in the LKB system to simplify displays (Oepen et al., 2002). But there are no truly satisfactory approaches for dealing with the problem of tree width, that is, for presenting 2D trees so as to provide an "at-a-glance" appreciation of structure for sentences longer than 25 words, which are very prevalent.[2] The methods currently in use or suggested either obscure the structure or require additional actions by the reader. Allowing trees to be as wide as the sentences requires horizontal scrolling. Collapsing some subtrees into single nodes (wrapping represented token sequences under those nodes) requires repeated expansions to view the entire parse. Using more of the display space by overlapping subtrees vertically interferes with comprehension because it obscures dominance and sequence. In Figure 2, for example, the period at the end of the sequence is the second constituent at the top. For a longer sentence, a coordinated constituent might be placed here as well. Such unpredictable arrangements impede reading, because the reader does not know where to look

[2] Casual records of parser results for many English non-fiction documents suggest an average of about 20 words per sentence, with a standard deviation of about 11.

```
S (NP-SBJ Stokely)
  (VP says
     (SBAR  0
        (S (NP-SBJ stores)
           (VP revive
               (NP (NP specials)
                   (PP like
                       (NP (NP three cans)
                           (PP of
                              (NP peas))
                           (PP for
                              (NP 99 cents )))))))))
```

Figure 3. A Penn Treebank Tree

The other conventional parse tree representation is as a fully-bracketed string, usually including category labels and, for display purposes, using indentation to indicate dominance. Figure 3 shows such a tree, drawn from a Penn Treebank annotation guide (Bies et al., 1995), shown with somewhat narrower indentation than the original. Even though the tree is in the relatively flat form developed for use by annotators, the brackets, labels, and depth of nesting combine to prohibit rapid scanning.

However, it should be noted that this format is the source of the TextTree concept. Because by eliminating labels, null elements, and most brackets, and further flattening, the more readable TextTree form emerges:

```
Stokely
says
    {stores
     revive
         specials
             like three cans
                 of peas
                 for 99 cents}
```

## 2.1 Implications

The conventional parse tree representations discussed above can be a bottleneck in parser development, most importantly in checking parser accuracy with respect to current focus areas and in extending coverage to new genres or domains. For these purposes, one would like to review the results of analyzing collections of relatively long (100+ sentence) documents. But unless this can be done quickly, a parser developer or grammar writer is tempted to rely on statistics with respect to the number of sentences given a parse, and declare victory if a high rate of parsing is reported.

Another approach to assessing parser quality is to rely on treebank-based bracket scoring measures, if treebanks are available in the particular genre. This can also can be a pitfall, as bracket scores tend to be unconsciously interpreted as measures of full-sentence parse correctness, which they are not.

On the other hand, treebank-based measurements can play a useful secondary role in evaluating parser development progress and in comparing parsers. But if insufficient treebanked material is available for the relevant domains and/or genres, a custom treebank must be developed. This process generally consists of two phases: (a) a bootstrapping phase in which an existing parser is applied to the corpus to produce either a single "best" tree, or multiple alternative trees, for each sentence, and then (b) a second phase in which annotators approve or edit a given parse tree, select among alternative trees, or manually create a full parse tree when no parse exists. All of the second phase alternatives are difficult given conventional parse-tree representations. For example, experiments by Marcus et al. (1993) associated with the Penn Treebank indicated that for the first annotation pass "At an average rate of 750 words per hour, a team of five part-time annotators …", i.e., a bit more than a page of this text per hour. Aids to selecting among alternative 2D trees can be given in the form of differentiating features (Oepen et al., 2002), but their effectiveness in helping to select among large trees differing in attachment choices is not clear.

Another activity impeded by conventional parse tree representations is regression testing. As a grammar increases in size, it is advisable to frequently re-apply the grammar to a large test corpus to determine whether recent changes have had negative effects on accuracy. While the existence of differences in output can be detected by automatic means, one would like to assess the differences by quickly comparing the divergent parses, which is difficult to do using detailed parse displays for long sentences.

Finally, an activity that is rarely discussed but is becoming increasingly important is providing comprehensible parser demonstrations. A syntactic parser is not an end-in-itself, but a building block in larger NLP research and development efforts. The criteria for selecting a parser for a project include both the kind of information provided by the parser and parser accuracy. However, current parser output representations are not geared to allowing potential users to quickly assess accuracy with respect to document types of interest.

## 3 The TextTree Generator: Externals

TextTrees are generated by an essentially grammar-independent processor from a combination of

(a) parser output trees that have been put into a standard form that retains the grammar-specific category labels and constituents, but whose details conform to the requirements of a parser-independent tool interface, and,

(b) a set of grammar-specific <category, directive> pairs, e.g., "<ROOT, Align>". Each pair specifies a default formatting for constituents in the category, specifically whether their sub-constituents are to be aligned vertically, or concatenated relative to a marked point, with subsequent children indented. These defaults are used in the absence of overriding factors such as coordination.

The directives used are very simple ones, because the simpler the formatting rules, the more likely it is that outputs can be accurately checked for correctness, and edited conveniently

It is assumed that the parser output either includes conventional parse trees, or that such trees can be derived from the output. This assumption covers many grammatical approaches, such as CG, HPSG, LFG, and TAG.

The logical configuration implied by these inputs is shown in Figure 4. It includes a parser-specific adaptor to convert parser-output trees into a standard form. The adaptor need not be very large; for the hybrid parser it consists of about 75 lines of Java code.

The subsections below discuss the standard ParseTree form, the directives that have been identified to date and their formatting effects, and the treatment of coordination.

### 3.1 Standard ParseTrees

A standard ParseTree consists of recursively

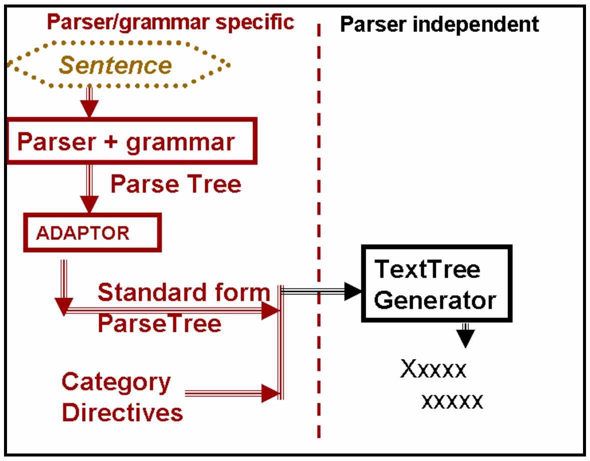


Figure 4: Logical Configuration.

nested subtrees, each representing a constituent C, and each indicating:

- A grammar-specific category label for C.
- Whether C should be considered the head of its immediately containing constituent P for formatting purposes. Generally, this is the case if C is, or dominates, the lexical head of P, but might differ from that to obtain some desired formatting effects. As heads are identified by most parsers, this marker can usually be set by parser-specific rules embedded in the adaptor.
- Whether C is a *coord_dom*, that is, whether it immediately dominates the participants in a coordination.
- If C is a leaf, the associated token, and
- (optionally) whether C is a multi-word.

### 3.2 Formatting Directives

To generate TextTrees for a particular grammar, a <category, directive> pair is provided for each category that will appear in a ParseTree for the grammar. The directive specifies how to format the children of constituents in the category in the absence of lengthy coordinations.

The definitions of the directives make use of one additional locator for a constituent, its *real_head*. While we generally want the directives to format constituents relative to their lexical heads, in some grammars, those heads are deeply nested. For example, a tree for an NP might be generated by rules such as:

NPz => DET <u>NPy</u>
NPy => ADJ* <u>NPx</u>
NPx => <u>NOUN</u> PP*

where each of the underscored terms are *heads*, but the *real_head* of NPz is the head NOUN of NPx. More generally, the *real_head* of a constituent C is the first constituent found by following the *head*-marked descendants of C downward until either (a) a leaf or headless constituent is reached, or (b) a post-modified *head*-marked constituent is found.

Using this definition, the current formatting directives are as follows:

**Align:** Align specifies that all children of the constituent are vertically aligned. Thus, for example, a directive <ROOT, Align>, would cause a constituent generated by a rule "ROOT => NP, VP" to be formatted as:

```
formatted NP
formatted VP
```

**ConcatHead:** ConcatHead concatenates the tokens of a constituent (with separating blanks) up to and including its *real_head* (as defined above), and indents and aligns the post-modifiers of the *real_head*. For example, given the directive "<NP, ConcatHead>", a constituent produced by the rules:

```
NPz => PreMod* NPx
NPx => NOUN PostMods*
```

would be formatted as:

```
All-words-in-PreMod* NOUN
        Formatted PostMod1
        Formatted PostMod2
```

**ConcatCompHead**: ConcatCompHead concatenates everything up to and including the *real_head*, and concatenates with that the results of a ConcatHead of the first post-modifier of the *real_head* (if any).

This directive is motivated by rules such as "PP => P NP", where the desired formatting groups the head with words up to and including the lexical head of the single complement, e.g.,

```
of the man
    in the moon
```

**ConcatSimpleComp**: ConcatSimpleComp concatenates material up to and including the *real_head* and, if first post-modifying constituent is a simple token, concatenates that as well, and then aligns and indents any further post-modifiers of the *real_head*. It thus formats noun phrases for languages that routinely use simple adjective post-modifiers. For example (Sp.):

```
La casa blanca
      que ...
```

**ConcatPreHead**: This directive concatenates material, if any, before the *real_head*, and then if such material exists, increases the indent level. It then aligns the following component. The directive is intended for formatting clauses that begin with subordinating conjunctions, complementizers, or relative pronouns, where the grammar has identified the following sentential component as the head, but it is more readable to indent that head under the initial constituent. In practice, in such cases it is easier to just alter the *head* marker within the adaptor.

### 3.3 Treatment of Coordination

The directives listed in the previous subsection are defaults that specify the handling of categories in the absence of coordination. A set of coordinated constituents are always indicated by surrounding square brackets ([ ]). If the coordination occurs within a requested concatenation, then if the width of coordination is less than a predesignated size, the non-token constituents of the coordination are bracketed, as in:

```
[{Old men} and women]
    in the park.
```

However, if the width of the coordination exceeds that size, the concatenation is converted to an alignment to avoid line wrap, for example,

```
He
gave
    sizeable donations
        to
            [the church,
             the school,
             and
             the new museum
                 of art.]
```

## 4 TextTree Generator: Implementation

TextTrees are produced in two steps. First, a ParseTree is processed to form one or more InternalTextTrees (ITTs), which are then mapped into an external TextTree. Most of the work is accomplished in the first step; the use of a second step allows the first to focus on logical structure, independent of many details of indentation, linebreaks, and punctuation.

We begin by describing ITTs and their relationship to external TextTrees, to motivate the description of ITT formation. We then describe the mapping from ParseTrees to ITTs.

### 4.1 Transforming ITTs to Strings

Figure 5 shows a simple ITT and its associated external TextTree. The node labels of an ITT are called the "headparts" of their associated subtrees. Headparts may be null, simple strings, or references to other ITTs.

If the headpart of a subtree is null, like that of the outermost subtree of Figure 5, the external TextTrees for its children are aligned vertically at the current level of indentation. Also, if the subtree is not outermost, the aligned sequence is bracketed.

However, if the headpart of a subtree is a simple string, as in the other subtrees of Figure 5 ITT, that string is printed at the current level of indentation, and the external TextTrees for its children, if any, are aligned vertically at the next level of indentation.

The headpart of a subtree may also reference another ITT, as illustrated in Figure 6. Such a reference headpart signals the bracketed inclusion of the referenced tree, which has a null headpart, at the current indentation level. This permits the entire referenced tree to be post-modified. The brackets used depend on a feature associated with the referenced tree, and are either [ ], if the referenced tree represents a coordination, or { } otherwise.

Pseudo-code for the ITT2String function that produces external TextTrees from ITTs is shown in Figure 7. The code omits the treatment of non-bracketing punctuation; in general, care is taken to prefix or suffix punctuation to tokens appropriately.

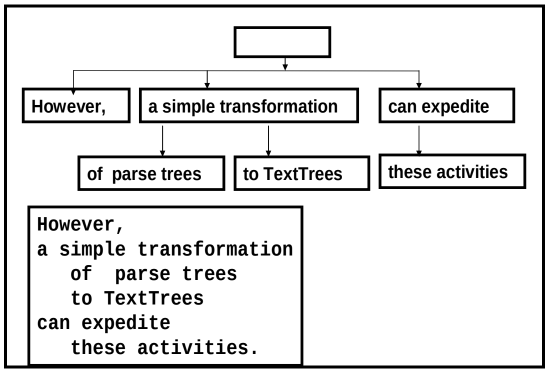
### 4.2 Transforming ParseTrees to ITTs

To transform ParseTrees into ITTs, subtrees are processed recursively top-down using the function BuildITT of Figure 8. Its arguments are an input subtree *p*, and a directive *override*, which may be null. It returns an ITT for *p*.

BuildITT sets the operational directive *d* as either the *override* argument, if that is non-null, or to the default directive for the category of *p*.

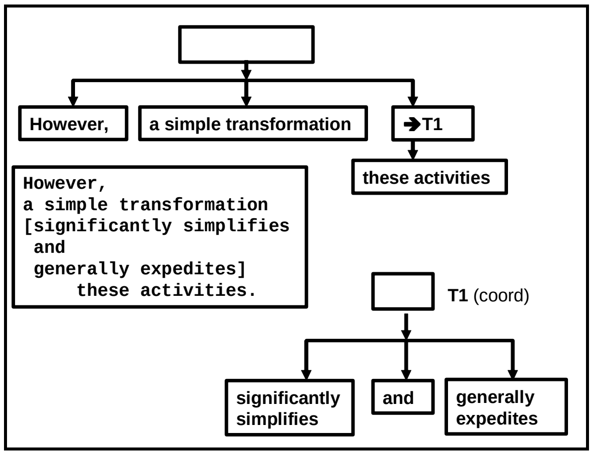
Then, if *d* is "Align", the result ITT has a null headpart and, if *p* is a *coord_dom*, the isCoord property of the ITT is set. The children of the result ITT are the ITTs of *p*'s children.

However, if *d* specifies that an initial sequence of *p* is to be concatenated, BuildITT uses the recursive function Catenate (Figure 9) to obtain the initial sequence if possible.



```
However,
a simple transformation
   of  parse trees
   to TextTrees
can expedite
   these activities.
```

Figure 5: Simple InternalTextTree.



```
However,
a simple transformation
[significantly simplifies
 and
 generally expedites]
     these activities.
```

Figure 6: ITT with long coordination

```
Function ITT2String(ITT s, String indent)
                        returns String
// indent is a string of blanks, eol is end-of-line
Set ls to null
If s has a null headpart, set nextIndent to indent
    + 1 blank  (adds space for [ or { bracket )
Otherwise  set nextIndent to indent + N blanks,
    where N is the constant indent increment
If s has a headpart reference
    set nextIndent to nextIndent + 1 blank

If s has children, set ls to the lines produced
    by ITT2String (ci, nextIndent)
        for each child ci of s

If s has a headpart string hs
    Return the concatenation of
        indent, hs, eol, ls
Else if s has a headpart reference to an ITT s2
    Return the concatenation of
        ITT2String(s2, indent), ls
Else  (s has a null headpart )
    Remove initial & trailing whitespace from ls
    If s is a coordination, set pfx to [ and sfx to ]
    Else set pfx to {and sfx to }
    Return the concatenation of
        nextIndent – 1 blank, pfx, ls, sfx, eol
```
Figure 7. The ITT2String Function

```
Function Catenate (ParseTree p,
    ParseTree r)  returns <Code, String>
1. Set result = null, code = incomplete.
   If p is a leaf, result = the token of p.
2. for each child ci of p,
   while code ≠ complete:
     If ci = r, set code = complete.
     Set <ccode, cstring> to result of
         Catenate(ci, r)
     If (ccode = failed) return <failed, null>
     If p is a coord_dom & cstring not 1 word
         // indicate coordinated within concat
         Suffix "{cstring}" to result.
     Otherwise suffix "cstring" to result
     If ccode = complete, set code = complete
3. Finally,
     If p = r, set code = complete.
     if p is a coord_dom and
        the length of  result  > LONG_CONST,
        return <failed, null>.
   Else if p is a coord_dom
        Return <code, "[result]">
   Else if p is a multi-word,
        return <code, "|result|">
   Otherwise return <code, result>
```
Figure 9. The Catenate Function

```
Function BuildITT(ParseTree p,  Directive override)  returns ITT
Set  d to override if non-null, otherwise to the default category directive for p.
1. If p is a leaf or a multiword:
   -  return an ITT whose headline concatenates the tokens spanned by p, and which has no children.
      If p is a multiword, bracket the headline by |.
2. Else if  d is Align
      return an ITT with a null headpart, and children built by invoking BuildITT(ci, null) for each
      child ci of p. If p is a coord_dom, indicate that the ITT isCoord.
3. Otherwise concatenate:
a) Find the nested subtree s that contains the rightmost element r to be concatenated according to d.
b)  Set the pair <ccode, cstring> to the result of Catenate (p, r).
c) If ccode ≠ "failed",   return ITT with headpart = cstring, and children formed by BuildITT(ci, null)
   for each child ci of s after r.  .
d) Otherwise return a directive-dependent tree aligning the contained coordination, for example:
   For ConcatHead:  i. Let p'  be like p but without the right siblings of the real_head of  p
                    ii. Return an ITT with headpart referencing the  results of BuildITT (p', Align)
                        and with children obtained from BuildITT(ci, null)
                            for each right sibling ci of the  real_head of p
   For ConcatCompHead:  Return an ITT obtained by invoking BuildITT(p, ConcatHead)
```
Figure 8. The BuildITT Function

Catenate takes two arguments: a ParseTree *p* whose initial sequence is to be concatenated, and a ParseTree *r*, beyond which the concatenation is to stop, based on the particular directive involved. It returns a pair <*Code, String*>. The *Code* indicates if the concatenation succeeded, failed, or is incomplete. If the concatenation succeeded, BuildITT creates an ITT with a headpart string containing the concatenated sequence, and children consisting of ITTs of the right-hand siblings of *r*.

Complex aspects of BuildITT and Catenate relate to coordinations within to-be-concatenated extents. The desired effect is to include short coordinations within the concatenation, while bracketing its boundaries and non-leaf components, e.g. " [{Old men} and women]", but aligning the elements of longer coordinations.

So if Catenate (in step 3) determines that the string resulting from a coordination is very long, it directly or indirectly returns an indicator to BuildITT that the concatenation failed. BuildITT (step 3d) then returns an ITT structured so that the sub-constituents that were to be concatenated are eventually shown as aligned, using different methods dependent on the directive *d*.

For example, if *d* is ConcatHead or ConcatSimpleComp, the result ITT contains:

a) a headpart reference to an ITT built by BuildITT(*p*',Align), where p' is like *p* but without the right-hand siblings of its *real_head*, and
b) children consisting of the ITTs for those right-hand siblings, if any.

## 5 TextTree Files and TextTreebanks

Previous sections focused on the production of individual TextTrees by the TextTree generator. This section considers some uses of the generator and auxiliary methods within parser development.

A particularly useful approach for producing parser output review material using the generator is sketched in Figure 10. In that approach, the best parses for a document are converted to standard ParseTrees expressed as XML entities and written to a file of such entities, interspersed with arbitrary other information. A separate, parser-independent process that includes the TextTree generator then creates a TextTree file by substituting TextTrees for the ParseTree entities.

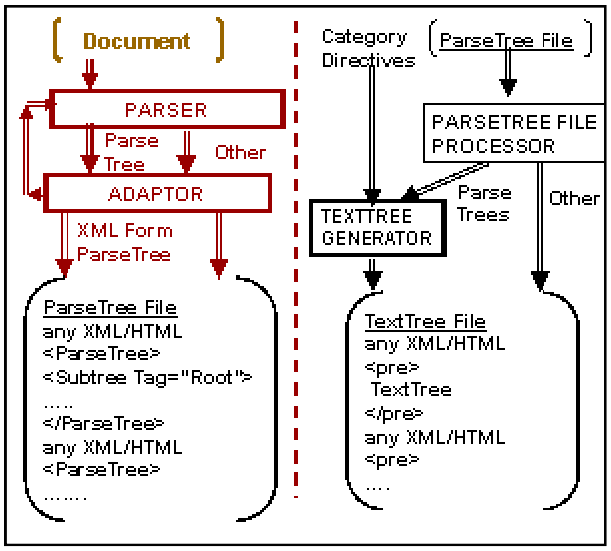


Figure 10. TextTree file creation

Such a process may be used to create the HTML TextTree file of Figure 1, which is a standard output form for the hybrid parser. The TextTrees are surrounded by HTML <pre> and </pre> tags to maintain the spacing and linebreaks produced by the generator. The interspersed information in this case consists of the sentence text and links to detailed displays of the best parse found, other parses with high preference scores, as well as the initial chunks.

Reviewing parse results for a document then consists of reading the TextTree file and, depending on circumstances, either simply noting or classifying the errors found for later debugging, or investigating them further via the links to the detailed displays.

### 5.1 TextTreebanks

Whatever the limitations (Carroll et al., 1998) of the various treebank-based bracket scoring measures derived from the Parseval approach of Black et al. (1991), they can be useful in monitoring parser development progress and in comparing the capabilities of different parsers, at least if there are large differences in scores.

But, as noted earlier, obtaining a fully labeled treebank for a specific domain or genre is generally a very labor-intensive process. A potentially less costly alternative is to create informal treebanks consisting of TextTree files corrected by manual editing.

Both corrected and uncorrected TextTree files can be converted to files consisting of fully-bracketed strings by a simple script that considers only the contained TextTrees. The script brackets the TextTrees so as to retain the explicit brackets, and to add brackets around each subtree, i.e., around each sequence of a line and the lines, if any, indented beneath it, directly or indirectly.

For example, a full bracketing of the TextTree of Figure 5 would be:

```
{However,}
{a simple transformation
          {of parse trees}
          {to text trees}}
{can expedite {these activities}}
```

The actual bracketed strings produced by the script are ones acceptable as input to the EVALB bracket scoring program (Sekine and Collins, 1997) with all brackets expressed as parentheses, and brackets added around words (apparently required but subsequently discarded by the program). Also, most punctuation is removed, to avoid spurious token differences. Then bracketed files deriving from noncorrected and corrected TextTree files can be submited to EVALB to obtain a bracket score.

Lest this approach be dismissed as overly sketchy, we note that the resulting brackets are similar to those resulting from a proposal by Ringger et al. (2004) for neutralizing differences between parser outputs and treebanks by bracketing maximal head projections, plus some additional mechanisms to further minimize brackets.

### 5.2 Preventing Bracketing Errors

To avoid bracketing errors resulting from imprecise spacing in manually edited trees, the TextTree indentations used are relatively wide. With indentations of five spaces, it is likely that an imprecisely positioned line will be placed closer to the desired level of indentation, so that an intelligent guess can be made as to the intent. For example, in:

```
Line a
     Line b
          Line c
 Line e??
```

the misplaced Line e begins at a point closer to the beginning of Line a than Line b. It is then reasonable to guess that Line e is sibling to Line a.

## 6 Experiments

This section describes two limited experiments to assess the efficiency of reviewing parser outputs for accuracy using TextTrees. One of the experiments also measures the efficiency of TextTreebank creation

### 6.1 First Experiment

The document used in the first experiment was the roughly 500-sentence "Executive Summary" of the 9/11 Commission Report (2004). After parsing by the hybrid parser, the expected TextTree file, excerpted in Figure 1, was created, reproducing each sentence and, for parsed sentences, the TextTree string for the best parse obtained, and a links to the detailed two-dimensional tree representation.

Of the 503 sentences, averaging 20 words in length, 93% received a parse. However, reviewing the TextTree file revealed that at least 191 of the 470 parsed sentences were not parsed correctly, indicating an actual parser accuracy for the document of at most 55%.

Reviewing the TextTrees required 92 minutes, giving a review rate of 6170 words per hour, including checking detailed parses for sentences where errors might have lain in the TextTree formatting.

That review rate can be compared to the results of Marcus et al. (1993) for post-annotation proofreading of relatively flat, indented, but fully labelled and bracketed trees. Those results indicated that:

> "... experienced annotators can proofread previously corrected material at very high speeds. A parsed subcorpus …was recently proofread at an average speed of approximately 4,000 words per annotator per hour. At this rate…, annotators are able to find and correct gross errors in parsing, but do not have time to check, for example, whether they agree with all prepositional phrase attachments."

While the two tasks are not exactly comparable, if we assume that little or no editing was required in proofreading, the ballpark improvement of 50% is encouraging.

### 6.2 Second Experiment

For the second experiment, we used the CB01 file[3] of the Brown Corpus (Francis and Kucera, 1964), and reviewed both the TextTree file and then, separately, the detailed 2D parse trees also produced.

While the parser reported that 91 of the 103 sentences, or 88%, received a parse, the review of the TextTree file determined that at most only 50 sentences, or 48.5%, received a fully correct parse. The review of the detailed parse trees revealed three additional errors.

The comparison of review times was less decisive in this experiment, with the rate for the TextTree review being 5733 words per hour, and that for the detailed 2D representation 4410 words per hour.

However, there were non-quantifiable differences in the reviews. One difference was that the TextTree review was a fairly relaxed exercise, while the review of the 2D representations was done with a conscious attempt at speed, and was quite stressful—not something one would like to repeat. Another difference was that scanning the TextTree file provided a far better cumulative sense of the kinds of problems presented by the document/genre, which might be further exploited by a more interactive format (e.g., using HTML forms) allowing users to classify erroneous parses by error type.

The experiment was then extended to check the extent to which TextTree files could efficiently edited for purposes of limited-function treebank creation.

For this purpose, to minimize typing when a sentence had no complete parse, the TextTree file included the list of chunks identified by the XIP parser. A similar strategy could be used with parsers that, when no complete parse is found, return an unrelated sequence of adjacent constituent parses. This is done by some statistical and finite-state-based parsers, as well as by parsers employing the "fitted parse" method of Jensen et al. (1983) or the "fragment parse" method described by Riezler et al. (2002).

[3] With part-of-speech tags removed

Editing the TextTrees for the 104 sentences, with an average sentence length of 21 words, required 83 minutes, giving a rate of 1518 words per hour. This might be compared with the average of 750 words per hour for the initial annotation of parses in the Penn Treebank experiment (Marcus et al., 1993) mentioned earlier.

After the TextTreebank was created, the bracketing script described in section 5.1 was applied both to the original TextTree file and to the TextTreebank, and the results were submitted to the EVALB program, which reported a bracketing recall of 71%, a bracketing precision of 84%, and an average crossing bracket count of 1.15.[4] Two sentences were not processed because of token mismatches. As expected, these scores were much higher than the percentage of sentences correctly parsed.

## 7 Related Work

Most natural language parsers include some provision for displaying their outputs, including parse tree representations, and/or other material, such as head-dependent relationships, feature structures, etc. These displays are generally intended for deep review of parse results, and thus attempt to maximize information content

Work on reducing review effort usually takes place in the context of developing treebanks by selection among, and/or manual correction of, parser outputs. In this area, the most relevant work may be the experiments of Marcus et al. (1993) using bracketed, indented trees. They found that annotator efficiency required eliminating many detailed brackets and category labels from the parser outputs presented. Other approaches rest, in whole or in part, on selecting among alternative two dimensional parse trees, such as the distinguishing-feature-augmented approach of Oepen et al. (2002), discussed in Section 2. As discussed in that section, however, two-dimensional tree displays are problematical for large trees, and it is not clear to what extent distinguishing feature information can expedite selecting among attachment choices.

[4] Sentences not receiving complete parses were submitted to EVALB without any brackets, contributing zero counts to the total # of correct constituents recalled.

Other related work deals with reducing measured differences between parser outputs and treebanks due solely to grammar style. As discussed in section 5, the bracketed material used in treebank-based measurement by Ringger et al. (2004) is similar to the bracketed material that would result from systematically bracketing TextTrees.

Finally, work by Walker (2001), intended not for parser/grammar development, but to facilitate reading and improve retention, produces text formats that bear some similarity to TextTrees, but are more closely attuned to spoken phrasing than syntactic form. The method uses complex segmentation and indentation strategies generally based on a combination of punctuation and closed-class words.

## 8 Directions for Further Development

We have described an implemented method for presenting parser outputs permitting fast visual inspection of the results of parsing long documents, as well as efficient editing to create informal treebanks. Although, because of their flattened, skeletal nature, TextTrees can hide some parser errors, we strongly believe, based on extended usage, that the convenience of reviewing TextTree files can contribute significantly to parser development efforts.

However, TextTrees are best suited to languages tending to a fixed word order and post-modification. To improve results for languages and language aspects that do not fall into this category, TextTrees might be augmented with highlighting and color to indicate syntactic functions. One way this could be done in a parser- and grammar-independent way is by adding a string-valued representation feature to the standard ParseTrees, and an additional set of directives to map the feature values to representation alternatives. For example, a subtree might be annotated with the feature Rep = "subject", and the additional directives might include <"subject", "blue">. Experimentation is needed to determine the usability of this approach.

Another topic to explore is the use of TextTrees in the creation of corpora annotated by deeper syntactic or semantic information. Because such information is generally expressed in forms that are even less readable than parse trees, a useful bootstrapping practice is to allow annotators to approve, or select among, parser output trees connected with the deeper information (King et al., 2003). TextTrees might be used to facilitate this process, with annotators either (a) interactively selecting among alternative TextTrees or, because there may be many alternatives, (b) editing a TextTree file containing at most one parse for each sentence (possibly chosen arbitrarily) and using the result for offline selection. Also, a parser used in the bootstrapping might refer to bracketed TextTreebanks to avoid pruning away elements of correct parses at intermediate points in parsing.

A third direction for further work is in extending the TextTree approach to deal with outputs of dependency-based parsers that do not produce constituent trees. While this should be a natural extension, an alternative system of features and directives would seem to be needed.

## Acknowledgements

The author is indebted to Tracy Holloway King, John Sowa, and the anonymous reviewers for their very helpful comments and suggestions on earlier versions of this paper.

## References

Salah Aït-Mokhtar, Jean-Pierre Chanod, and Claude Roux. 2002. Robustness beyond shallowness: incremental deep parsing, *Natural Language Engineering* 8:121-144 Cambridge University Press.

Ann Bies, Mark Ferguson, Karen Katz, and Robert MacIntyre. 1995. Bracketing Guidelines for the Treebank II Style Penn Treebank Project.

E. Black, S. Abney, D. Flickinger, C. Gdaniec, R. Grishman, P. Harrison, D. Hindle, R. Ingria, F. Jelinek, J. Klavans, M. Liberman, M. Marcus, S. Roukos, B Santorini, and T. Strzalkowski. 1991. A procedure for quantitatively comparing the syntactic coverage of english grammars. In *Proc 1991 DARPA Speech and Natural Language Workshop*. 306-311.

John Carroll, Ted Briscoe, and Antonio SanFillipo. 1998. Parser Evaluation: a Survey and a New Proposal. In Rubio et al., editors, *Proc First*

*International Conference on Language Resources and Evaluation*, Granada, Spain, 447-454.

W. Nelson Francis and Henry Kucera. 1964, 1971, 1979. Brown Corpus Manual. Available at http://helmer.aksis.uib.no/icame/brown/bcm.html Corpus available at http://nltk.sourceforge.net

Karen Jensen, George Heidorn, Lance A. Miller, Yael Ravin. 1983. Parse fitting and prose fixing: getting a hold on ill-formedness. *Computational Linguistics* 9(3-4):147-160.

Tracy H. King, Stefanie Dipper, Anette Frank, Jonas Kuhn, and John Maxwell. 2000. Ambiguity management in grammar writing. In Erhard Hinrichs, Detmar Meurers, and Shuly Wintner, editors, *Proc ESSLLI Workshop on Linguistic Theory and Grammar Implementation*, 5-19, Birmingham, UK.

Tracy H. King, Richard Crouch, Stefan Riezler, Mary Dalrymple, and Ronald M. Kaplan. 2003. The PARC 700 Dependency Bank, *Proc. 4th International Workshop on Linguistically Interpreted Corpora, at the 10th Conf of the European Chapter of the ACL*, Budapest.

Mitchell P. Marcus, Mary Ann Marcinkiewicz, and Beatrice Santorini. 1993. Building a Large Annotated Corpus of English: The Penn Treebank, *Computational Linguistics* 19(2):313-330.

Paula S. Newman. 2005. Abstract: TextTrees in Grammar Development. *Workshop on Grammar Engineering of the 2005 Scandinavian Conference on Linguistics (SCL).* Available at http://www.hf.ntnu.no/scl/grammar_engineering

9/11 Commission. 2004. Final Report of the National Commission on Terrorist Attacks upon the United States, Executive Summary. Available at http://www.gpoaccess.gov/911

Stephan Oepen, Dan Flickinger, Kristina Toutanova, and Christopher Manning. 2002. LinGO Redwoods: A Rich and Dynamic Treebank for HPSG, *Proc Workshop on Treebanks and Linguistic Theories (TLT02),* Sozopol, Bulgaria

Stephan Riezler, Tracy H. King, Ronald M. Kaplan, Richard Crouch, John T. Maxwell, and Mark Johnson. 2002. Parsing the Wall Street Journal using a Lexical-Functional Grammar and Discriminative Estimation Techniques. *Proc 40th Annual Meeting of the Assoc. for Computational Linguistics* (ACL), Philadelphia, 271-278.

Eric K. Ringger, Robert C. Moore, Lucy Vanderwende, Hisam Suzuki and Eugene Charniak. 2004. Using the Penn Treebank to Evaluate Non-Treebank Parsers, *Proc 2004 Language Resources and Evaluation Conference (LREC)*, Lisbon, Portugal.

Satoshi Sekine and Michael Collins. 1997. EvalB. Available at http://nlp.cs.nyu.edu/evalb

Randall C. Walker. 2001. Method and Apparatus for Displaying Text Based Upon Attributes Found Within the Text, U.S. Patent 6279017.

# Evaluating and Integrating Treebank Parsers on a Biomedical Corpus

**Andrew B. Clegg and Adrian J. Shepherd**
School of Crystallography
Birkbeck College
University of London
London WC1E 7HX, UK
{a.clegg,a.shepherd}@mail.cryst.bbk.ac.uk

## Abstract

It is not clear *a priori* how well parsers trained on the Penn Treebank will parse significantly different corpora without retraining. We carried out a competitive evaluation of three leading treebank parsers on an annotated corpus from the human molecular biology domain, and on an extract from the Penn Treebank for comparison, performing a detailed analysis of the kinds of errors each parser made, along with a quantitative comparison of syntax usage between the two corpora. Our results suggest that these tools are becoming somewhat over-specialised on their training domain at the expense of portability, but also indicate that some of the errors encountered are of doubtful importance for information extraction tasks.

Furthermore, our inital experiments with unsupervised parse combination techniques showed that integrating the output of several parsers can ameliorate some of the performance problems they encounter on unfamiliar text, providing accuracy and coverage improvements, and a novel measure of trustworthiness.

Supplementary materials are available at http://textmining.cryst.bbk.ac.uk/acl05/.

## 1 Introduction

The availability of large-scale syntactically-annotated corpora in general, and the Penn Treebank[1] (PTB; Marcus et al., 1994) in particular, has enabled the field of stochastic parsing to advance rapidly over the course of the last 10-15 years. However, the newspaper English which makes up the bulk of the PTB is only one of many distinct genres of writing in the Anglophone world, and certainly not the only domain where potential natural-language processing (NLP) applications exist that would benefit from robust and reliable syntactic analysis. Due to the massive glut of published literature, the biomedical sciences in general, and molecular biology in particular, constitute one such domain, and indeed much attention has been focused recently on NLP in this area (Shatkay and Feldman, 2003; Cohen and Hunter, 2004).

Unfortunately, annotated corpora of a large enough size to retrain stochastic parsers on do not exist in this domain, and are unlikely to for some time. This is partially due to the same differences of vocabulary and usage that set biomedical English apart from the *Wall Street Journal* in the first place; these differences necessitate the input of both biological and linguistic knowledge on biological corpus annotation projects (Kulick et al., 2004), and thus require a wider variety of annotator skills than general-English projects. For example, *5′* (pronounced "five-prime") is an adjective in molecular biology, but *p53* is a noun; *amino acid*

[1] http://www.cis.upenn.edu/~treebank/

is an adjective-noun sequence[2] but *cadmium chloride* is a pair of nouns. These tagging decisions would be hard to make correctly without biological background knowledge, as would the prepositional phrase attachment decisions in Figure 1.

Although it is intuitively apparent that there are differences between newspaper English and biomedical English, and that these differences are quantifiable enough for biomedical writing to be characterised as a sublanguage of English (Friedman et al., 2002), the performance of conventionally-trained parsers on data from this domain is to a large extent an open question. Nonetheless, papers have begun to appear which employ treebank parsers on biomedical text, essentially untested (Xiao et al., 2005). Recently, however, the GENIA project (Kim et al., 2003) and the Mining the Bibliome project (Kulick et al., 2004) have begun producing small draft corpora of biomedical journal paper abstracts with PTB-style syntactic bracketing, as well as named-entity and part-of-speech (POS) tags. These are not currently on a scale appropriate for retraining parsers (compare the ∼50,000 words in the GENIA Treebank to the ∼1,000,000 in the PTB; but see also Section 7.2) but can provide a sound basis for empirical performance evaluation and analysis. A collection of methods for performing such an analysis, along with several interesting results and an investigation into techniques for narrowing the performance gap, is presented here.

### 1.1 Motivation

We undertook this project with the intention of addressing several questions. Firstly, in order to deploy existing parsing technologies in a bioinformatics setting, the biomedical NLP community needs a comprehensive assessment of performance – which parser(s) to choose, what accuracy each should be expected to achieve etc., along with information about the different situations in which each parser can be expected to perform well or poorly. Secondly, assuming there is a performance deficit, can any simple steps be taken to mitigate it? Thirdly, what engineering issues arise from the idiosyncracies of biomedical text?

The differences discovered in the behaviour of each parser, either between domains or between different software versions on the same domain, will also be of interest to those in the computational linguistics community who are involved in parser design. These values will give a comparative index of the flexibility of each parsing model on being presented with out-of-domain data, and may help parser developers to detect signs of overtraining or, analogously, ‘over-design’ for one narrow genre of English. It is hoped that our findings can assist those better equipped than ourselves in properly investigating these phenomena, and that our analysis of the problems encountered can shed new light on the thorny problem of parser evaluation.

Finally, several questions arise from the use of multiple parsers on the same corpus that are of both theoretical and practical interest. Does agreement between several parsers indicate that a sentence has been parsed correctly, or do they tend to make the same mistakes? How best can the output of an ensemble of parsers be integrated, in order to boost performance above that of the best single member? And what additional information can be gleaned from comparing the opinions of several parsers that can help make sense of unfamiliar text?

## 2 Evaluation methodologies

We initially chose to rate the parsers in our assessment by several different means which can be grouped into two broad classes: constituent- and lineage-based. While Sampson and Babarczy (2003) showed that there is a limited degree of correlation between the per-sentence scores assigned by the two methods, they are independent enough that a fuller picture of parser competence can be built up by combining them and thus sidestepping the drawbacks of either approach. However, overall performance scores designed for competitively evaluating parsers do not provide much insight into the aetiology of errors and anomalies, so we developed a third approach based on production rules that enabled us to mine the megabytes of syntactic data for enlightening results more ef-

[2] According to some annotators at least; others tag *amino* as a noun, although one would not speak of **an amino*, **some amino* or **several aminos*.

a. *[ This protein ] [ binds the DNA [ by the TATA box [ on its minor groove. ]$_2$ ]$_1$ ]*

b. *[ This protein ] [ binds the DNA [ by the TATA box ]$_1$ [ at its C-terminal domain. ]$_2$ ]*

Figure 1: These two sentences are biologically clear but syntactically ambiguous. Only the knowledge that the C-terminal domain is part of a protein, whereas the TATA box and minor groove are parts of DNA, allows a human to interpret them correctly, by attaching the prepositional phrases 1 and 2 at the right level.

fectively. All the Perl scoring routines we wrote are available from our website.

### 2.1 Constituent-based assessment

Most evaluations of parser performance are based upon three primary measures: labelled constituent precision and recall, and number of crossing brackets per sentence. Calculation of these scores for each sentence is straightforward. Each constituent in a candidate parse is treated as a tuple $\langle lbound, LABEL, rbound \rangle$, where *lbound* and *rbound* are the indices of the first and last words covered by the constituent. Precision is the proportion of candidate constituents that are correct and is calculated as follows:

$$P = \frac{\text{\# true positives}}{\text{\# true positives + \# false positives}}$$

Recall is the proportion of constituents from the gold standard that are in the candidate parse:

$$R = \frac{\text{\# true positives}}{\text{\# true positives + \# false negatives}}$$

The crossing brackets score is reached by counting the number of constituents in the candidate parse that overlap with at least one constituent in the gold standard, in such a way that one is not a subsequence of the other.

Although this scoring system is in wide use, it is not without its drawbacks. Most obviously, it gives no credit for partial matches, for example when a constituent in one parse covers most of the same words as the other but is truncated or extended at one or both ends. Indeed, one can imagine situations where a long constituent is truncated at one end and extended at the other compared to the gold standard; this would incur a penalty under each of the above metrics even though some or even most of the words in the constituent were correctly categorised. One can of course suggest modifications for these measures designed to account for particular situations like these, although not without losing some of their elegance. The same is true for label mismatches, where a constituent's boundaries are correct but its category is wrong.

More fundamentally, it could be argued that by taking as it were horizontal slices through the syntax tree, these measures lose important information about the ability of a parser to recreate the gross grammatical structure of a sentence. The height of a given constituent in the tree, and the details of its ancestors and descendants, are not directly taken into account, and it is surely the case that these broader phenomena are at least as important as the extents of individual constituents in affecting meaning. However, constituent-based measures are not without specific advantages too. These include the ease with which they can be broken down into scores per label to give an impression of a parser's performance on particular kinds of constituent, and the straightforward message they deliver about whether a badly-performing parser is tending to over-generate (low precision), under-generate (low recall) or mis-generate (high crossing brackets).

### 2.2 Lineage-based assessment

In contrast to this horizontal-slice philosophy, Sampson and Babarczy (2003) advocate a vertical view of the syntax tree. By walking up the tree structure from the immediate parent of a given word until the top node is reached, and adding each label encountered to the end of a list, a 'lineage' representing the word's ancestry can be retrieved. Boundary symbols are inserted into this lineage before the highest constituent that begins on the word, and after the highest constituent that ends on the word, if such conditions apply; this allows potential ambiguities to be avoided, so that the tree as a whole has one and only one corresponding set of 'lineage strings' (see Figure 2).

Using dynamic programming, a Levenshtein edit distance can be calculated between each word's lineage strings in the candidate parse and the gold standard, by determining the smallest number of symbol insertions, deletions and substitutions required to transform one of the strings into the other. The leaf-ancestor (LA) metric, a similarity score ranging between 0 (total parse failure) and 1 (exact match), is then calculated by taking into account the lengths of the two lineages:

$$LA = 1 - \frac{dist(lineage_1, lineage_2)}{len(lineage_1) + len(lineage_2)}$$

The per-word score can then be averaged over a sentence or a whole corpus in order to arrive at an overall performance indicator. Besides avoiding some of the limitations of constituent-based evaluation discussed above, one major advantage of this approach is that it can provide a word-by-word measure of parser performance, and thus draw attention easily to those regions of a sentence which have proved problematic (see Section 6.2 for an example). The algorithm can be made more sensitive to near-matches between phrasal categories by tuning the cost incurred for a substitution between similar labels, e.g. those for 'singular noun' and 'proper noun', rather than adhering to the uniform edit cost dictated by the standard Levenshtein scheme. In order to avoid over-complicating this study, however, we chose to keep the standard penalty of 1 for each insertion, deletion or substitution.

One drawback to leaf-ancestor evaluation is that although it scores each word (sentence, corpus) between 0 and 1, and these scores are presented here as percentages for readability, it is misleading to think of them as percentages of correctness in the same way that one would regard constituent precision and recall. Indeed, the very fact that it results in a single score means that it reveals less at first glance about the broad classes of errors that a parser is making than precision, recall and crossing brackets do. Another possible objection is that since an error high in the tree will affect many words, the system implicitly gives most weight to the correct determination of those features of a sentence which are furthest from being directly observable. One might argue, however, that since a high-level attachment error can grossly perturb the structure of the tree and thus the interpretation of the sentence, this is a perfectly valid approach; it is certainly complementary to the uniform scoring scheme described in the previous section, where every mistake is weighted identically.

### 2.3 Production-based assessment

In order to properly characterise the kinds of errors that occurred in each parse, and to help elucidate the differences between multiple corpora and between each parser's behaviour on each corpus, we developed an additional scoring process based on production rules. A production rule is a syntactic operation that maps from a parent constituent in a syntax tree to a list of daughter constituents and/or POS tags, of the general form:

$$LABEL_p \rightarrow LABEL_1 \ldots LABEL_n$$

For example, the rule that maps from the topmost constituent in Figure 2 to its daughters would be `S` $\rightarrow$ `NP VP`. A production is the application of a production rule at a particular location in the sentence, and can be expressed as:

$$LABEL_p(lbound, rbound) \rightarrow LABEL_1 \ldots LABEL_n$$

Production precision and recall can be calculated as in a normal labelled constituent-based assessment, except that a proposed production is a true positive if and only if there exists a production in the gold standard with the same parent label and boundaries, and the same daughter labels in the same order. (The respective widths of the daughter constituents, where applicable, are not taken into account, only their labels and order; any errors of width in the daughters are detected when they are tested as parents themselves.)

Furthermore, as an aid to the detection and analysis of systematic errors, we developed a heuristic

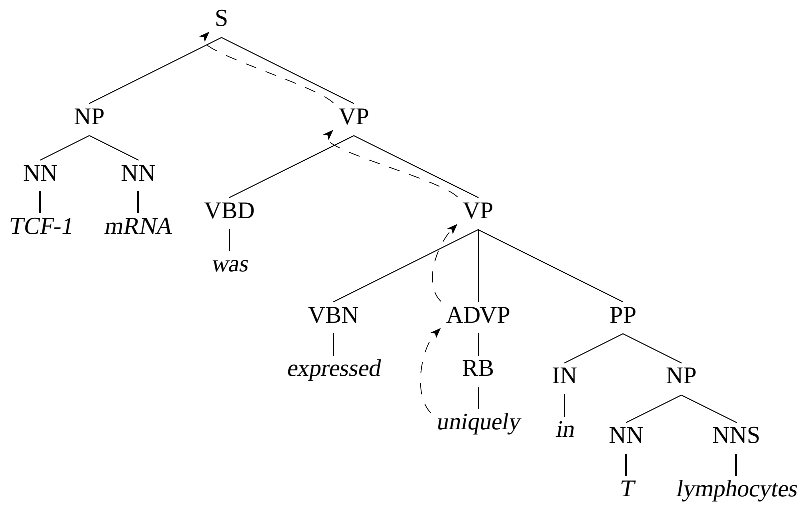


Figure 2: Skipping the POS tag, the lineage string for *uniquely* is: `[ ADVP ] VP VP S` . The left and right boundary markers record the fact that the `ADVP` constituent both starts and ends with this word.

for finding the closest-matching candidate productions $PROD_{c1} \ldots PROD_{cm}$ in a parse, in each case where a production $PROD_g$ in the gold standard is not exactly matched in the parse.

1. First, the heuristic looks for productions with correct boundaries and parent labels, but incorrect daughters. The corresponding production rules are returned.

2. Failing that, it looks for productions with correct boundaries and daughters, preserving the order of the daughters, but with incorrect parent labels. The corresponding production rules are returned.

3. Failing that, it looks for productions with correct boundaries but incorrect parent labels and daughters. The corresponding production rules are returned.

4. Failing that, it looks for all extensions and truncations of the production (boundary modifications such that there is at least one word from $PROD_g$ still covered) with correct parent and daughter labels and daughter order, keeping only those that are closest in width to $PROD_g$ (minimum number of extensions and truncations). The meta-rules `EXT_ALLMATCH` and/or `TRUNC_ALLMATCH` as appropriate are returned.

5. Failing that, it looks for all extensions and truncations of the production where the parent label is correct but the daughters are incorrect, keeping only those that are closest in width to $PROD_g$. The meta-rules `EXT_PARENTMATCH` and/or `TRUNC_PARENTMATCH` are returned.

6. If no matches are found in any of these classes, a null result is returned.

Note that in some cases, $m$ production rules of the same class may be returned, for example when the closest matches in the parse are two productions with the correct parent label, one of which is one word longer than $PROD_g$, and one of which is one word shorter. It is also conceivable that multiple productions with the same parent or same daughters could occupy the same location in the sentence without branching, although it seems unlikely that this would occur apart from in pathologically bad parses. In any ambiguous cases, no attempt is made to decided which is the ‘real’ closest match; all $m$ matches are returned, but they are downweighted so that each counts as $1/m$ of an error when error frequencies are calculated. In no circumstances are matches from different classes returned.

The design of this procedure reflects our requirements for a tool to facilitate the diagnosis and summarisation of parse errors. We wanted to be able to answer questions like "given that parser A has a low recall for `NP` $\rightarrow$ `NN NN` productions, what syntactic structures is it generating in their place? Why might this be so? And what effect might these errors have on the interpretation of the sentence?" Accordingly, as the heuristic casts the net further and further to find the closest match for a production $PROD_g$, the classes to which it assigns errors become broader and broader. Any match at stages 1–3 is not simply recorded as a substitution error, but a substitution for a *particular* incorrect production rule. However, matches at stages 4 and 5 do not make a distinction between different magnitudes of truncation and extension, and at stage 5 the information about the daughters of incorrect productions is discarded. This allowed us to identify broad trends in the data even where the correspondences between the gold standard and the parses were weak, yet nonetheless recover detailed substitution information akin to confusion matrices where possible.

Similar principles guided the decision not to consider extensions and truncations with different parent labels as potential loose matches, in order to avoid uninformative matches to productions elsewhere in the syntax tree. In practice, the matches returned by the heuristic accounted for almost all of the significant systematic errors suffered by the parsers (see Section 6) – null matches were infrequent enough in general that their presence in larger numbers on certain production rules was itself useful from an explanatory point of view.

### 2.4 Alternative approaches

Several other proposed solutions to the evaluation problem exist, and it is an ongoing and continually challenging field of research. Suggested protocols based on grammatical or dependency relations (Crouch et al., 2002), head projection (Ringger et al., 2004), alternative edit distance metrics (Roark, 2002) and various other schemes have been suggested. Many of these alternative methodologies, however, suffer from one or more disadvantages, such as specificity to one particular grammatical formalism (e.g. head-driven phrase structure grammar) or one class of parser (e.g. partial parsers), or a requirement for a specific manually-prepared evaluation corpus in a non-treebank format. In addition, none of them deliver the richness of information supplied by production-based assessment, particularly in combination with the other methods outlined above.

## 3 Comparing the corpora

The gold standard data for our experiments was drawn from the GENIA Treebank[3], a beta-stage corpus of 200 abstracts drawn randomly from the MEDLINE database[4] with the search terms "human", "blood cell" and "transcription factor". These abstracts have been annotated with POS tags, named entity classes and boundaries[5], and syntax trees which broadly follow the conventions of the PTB. Some manual editing was required to correct annotation errors and remove sentences with uncorrectable errors, leaving 1757 sentences (45406 tokens) in the gold standard. All errors were reported to the GENIA group.

For comparison purposes, we used the standard set-aside test set from the PTB, section 23. This consists of 56684 words in 2416 sentences.

To gain insight into the differences between the two corpora, we ran several tests of the grammatical composition of each. For consistency with the parser evaluation results, we stripped the following punctuation tokens from the corpora before gathering these statistics: period, comma, semicolon, colon, and double-quotes (whether they were expressed as a single double-quotes character, or pairs of opening or closing single-quotes). We also removed any super-syntactic information such as grammatical function suffixes, pruned any tree branches that did not contain textual terminals (e.g. traces), and deleted any duplicated constituents – that is, constituents with only one daughter that has the same label.

### 3.1 Sentence length and complexity

Having performed these pre-processing steps, we counted the distributions of sentence lengths (in

[3]`http://www-tsujii.is.s.u-tokyo.ac.jp/GENIA/topics/Corpus/GTB.html`

[4]`http://www.pubmed.org/`

[5]The named entity annotations are supplied in a separate file which was discarded.

words) and sentence complexities, using the number of constituents, not counting POS tags, as a simple measure of complexity – although of course one can imagine various other ways to gauge complexity (mean tree depth, maximum tree depth, constituents per word etc.). The results are shown in Figure 3, and reveal an unexpected level of correlation. Apart from a few sparse instances at the right-hand tails of the two GENIA distributions, and a single-constituent spike on the PTB complexity distribution (due to one-phrase headings like *STOCK REPORT.*), the two corpora have broadly similar distributions of word count and constituent count. The PTB has slightly more mass on the short end of the length scale, but GENIA does not have a corresponding number of longer sentences. This ran contrary to our initial intuition that newspaper English would tend to be composed predominantly of shorter and simpler sentences than biological English.

### 3.2 Constituent and production rule usage

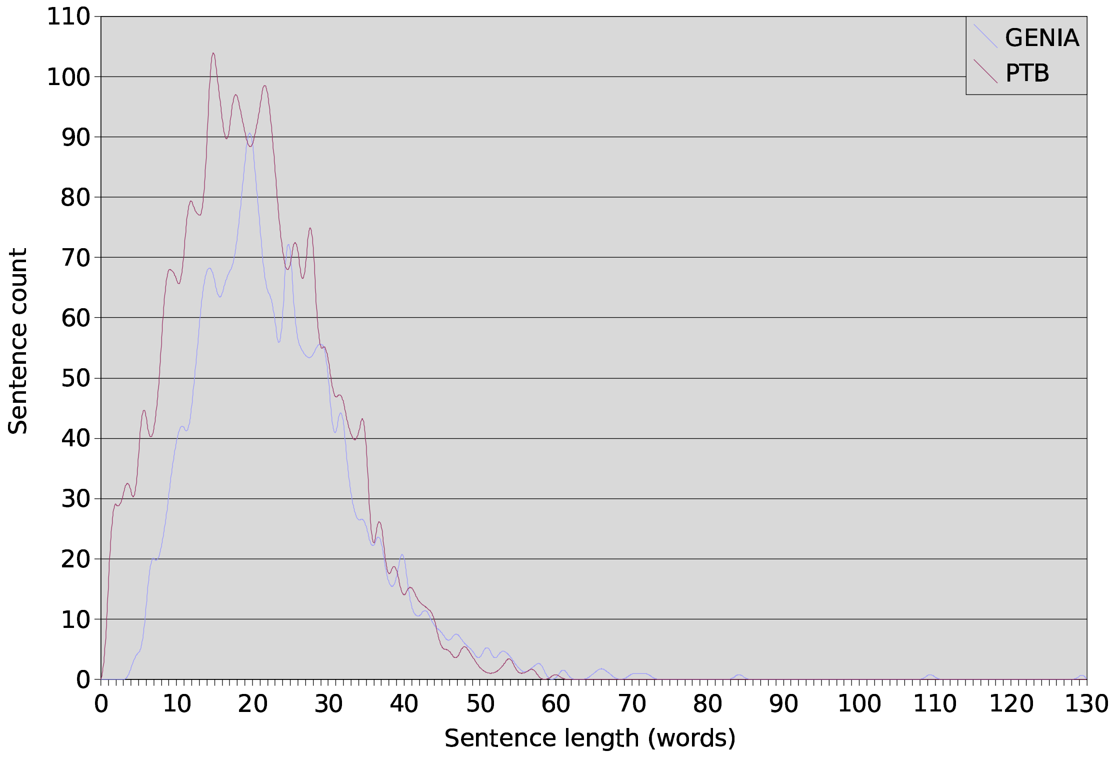
Next, we counted the frequency with which each constituent label appears in each corpus. The results are shown in Figure 4. The distributions are reasonably similar between the two corpora, with the most obvious difference being that GENIA uses noun phrases more often, by just over six percentage points. This may reflect the fact that much of the text in GENIA describes interactions between multiple biological entities at the molecular and cellular levels; conjunction phrases are three times as frequent in GENIA too, although this is not obvious from the chart as the numbers are so low in each corpus.

One surprising result is revealed by looking at Table 1 which shows production rule usage across the corpora. Although GENIA uses slightly more productions per sentence on average, it uses marginally fewer *distinct production rules* per sentence, and considerably fewer overall – 62% of the number of rules used in the PTB, despite being 73% of the size in sentences. These figures, along with the significantly different rankings and frequencies of the actual rules themselves (Table 2), demonstrate that there are important syntactic differences between the corpora, despite the similarities in length, complexity and constituent usage. Such differences are invisible to conventional constituent-based analysis.

The comparative lack of syntactic diversity in GENIA may seem counter-intuitive, since biological language seems at first glance dense and difficult. However, it must be remembered that the text in GENIA consists only of abstracts, which are tailored to the purpose of communicating a few salient points in a short passage, and tend to be composed in a somewhat formulaic manner. They are written in a very restricted register, compared to the range of registers that may be present in one issue of a newspaper – news articles, lifestyle features, opinion pieces, financial reports and letters will be delivered in very different voices. Also, some of the apparent complexity of biomedical texts is illusory, stemming from the unfamiliar vocabulary, and furthermore, a distinction must be made between syntactic and semantic complexity. Consider a phrase like *iron-sulphur cluster assembly transcription factor*, the name of a family of DNA-binding proteins, which is a semantically-complex concept expressed in a syntactically-simple form – essentially just a series of nouns.

## 4 Evaluating the parsers

The parsers chosen for this evaluation were those described originally in Collins (1999), Charniak (1999) and Bikel (2002). These were selected because they are up-to-date (having last been updated in 2002, 2003 and 2004 respectively), highly regarded by the computational linguistics community, and importantly, free to use and modify for academic research. Since part of our motivation was to detect signs of over-specialisation on the PTB, we assessed the current (0.9.9) and previous (0.9.8) versions of the Bikel parser individually. The current version was invoked with the new `bikel.properties` settings file, which enables parameter pruning (Bikel, 2004), whereas the previous version used the original `collins.properties` settings which were designed to emulate the Collins parser model 2 (see below). The same approach was attempted with the Charniak parser, but the latest version (released February 2005) suffered from fatal errors



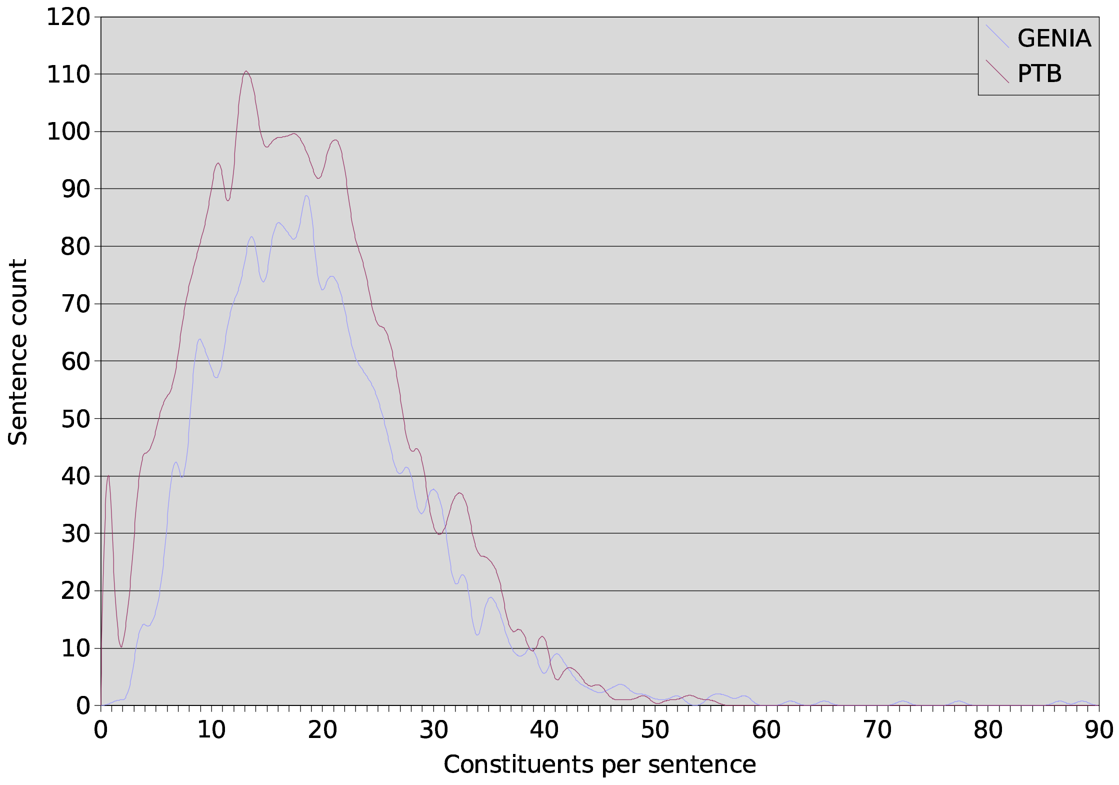


Figure 3: Sentence length and complexity distributions, GENIA vs. PTB.

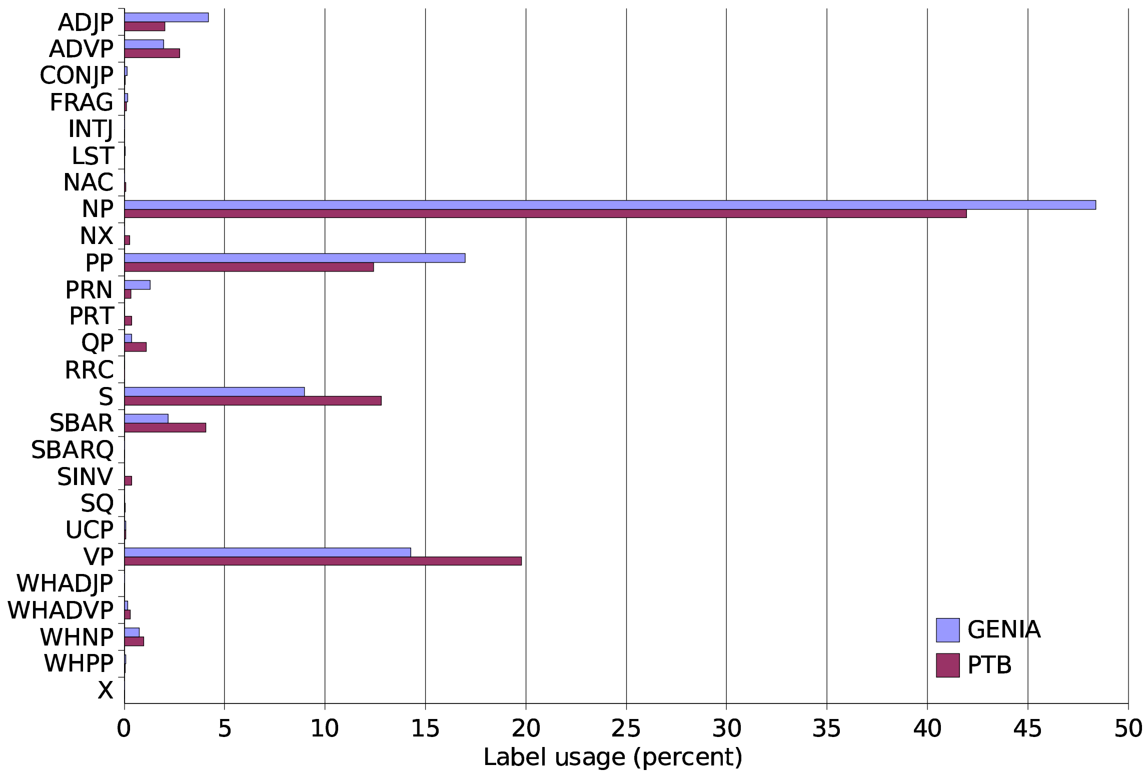


Figure 4: Constituent usages, GENIA vs. PTB.

| | GENIA | PTB |
|---|---|---|
| Num. productions used in corpus | 78831 (44.87/sent.) | 96694 (40.02/sent.) |
| Num. distinct production rules | 1364 (5.47/sentence) | 2184 (5.55/sent.) |

Table 1: Production and production rule usage in the two corpora.

on GENIA which could not be diagnosed in time for publication. Earlier versions of the Collins parser are not available; however, the distribution comes with three language models of increasing sophistication which were treated initially as distinct parsers.

Tweaking of parser options was kept to a minimum, aside from trivial changes to allow for unexpectedly long words, long or complex sentences (e.g. default memory/time limits), and differing standards of tokenisation and punctuation, although a considerable degree of pre- and post-processing by Perl scripts was also necessary to bring these into line. More detailed tuning would have massively increased the number of variables under consideration, given the number of compile-time constants and run-time parameters available to the programs; furthermore, it is probably safe to assume that each author distributes his software with an optimal or near-optimal configuration, at least for in-domain data.

## 4.1 Part-of-speech tagging

The Collins parser requires pre-tagged input, and although the Bikel parser can take untagged input, the author recommends the use of a dedicated POS tagger. For this reason, we pre-processed GENIA with MedPost (Smith et al., 2004), a specialised biomedical POS tagger that was developed and trained on MEDLINE abstracts. The supplied gold-standard POS tags were discarded as using them would not provide a realistic ap-

| Top-25 production rules in GENIA (left) and PTB (right) | | | | |
|---|---|---|---|---|
| Freq. | Rule | Rank | Freq. | Rule |
| 6.36 | PP → IN NP | 1 | 4.80 | PP → IN NP |
| 3.32 | NP → NN | 2 | 2.95 | S → NP VP |
| 3.13 | NP → NP PP | 3 | 2.26 | NP → NP PP |
| 2.17 | S → NP VP | 4 | 2.15 | TOP → S |
| 2.03 | TOP → S | 5 | 1.86 | NP → DT NN |
| 1.23 | NP → DT NN | 6 | 1.43 | S → VP |
| 0.89 | NP → NN NN | 7 | 1.08 | NP → PRP |
| 0.85 | NP → NP CC NP | 8 | 0.92 | ADVP → RB |
| 0.82 | S → VP | 9 | 0.91 | NP → NNP |
| 0.76 | VP → VBN PP | 10 | 0.83 | NP → NNS |
| 0.74 | ADVP → RB | 11 | 0.81 | VP → TO VP |
| 0.70 | NP → DT JJ NN | 12 | 0.78 | NP → NN |
| 0.66 | NP → NNS | 13 | 0.74 | NP → NNP NNP |
| 0.58 | NP → JJ NNS | 14 | 0.63 | SBAR → IN S |
| 0.53 | NP → JJ NN | 15 | 0.62 | NP → DT JJ NN |
| 0.51 | SBAR → IN S | 16 | 0.60 | NP → NP NP |
| 0.51 | PP → TO NP | 17 | 0.57 | SBAR → S |
| 0.49 | NP → DT NN NN | 18 | 0.50 | VP → VB NP |
| 0.48 | NP → NP PRN | 19 | 0.48 | NP → NP SBAR |
| 0.48 | ADJP → JJ | 20 | 0.47 | VP → MD VP |
| 0.47 | NP → NP PP PP | 21 | 0.46 | NP → JJ NNS |
| 0.47 | PRN → ( NP ) | 22 | 0.41 | SBAR → WHNP S |
| 0.45 | NP → NN NNS | 23 | 0.40 | PP → TO NP |
| 0.44 | NP → NP VP | 24 | 0.33 | VP → VBD SBAR |
| 0.40 | VP → VBD VP | 25 | 0.32 | NP → NP CC NP |

Table 2: The most common production rules in the two corpora, in order, with the frequency of occurrence of each. Notice that several rules are much more common in one corpus than the other, such as VP → TO VP, which is the 11th most common rule in the PTB but doesn't make it into GENIA's list.

proximation of the kinds of scenario where parsing software would be deployed on unseen text. MedPost was found to tag GENIA with 93% accuracy.

Likewise, although the Charniak parser assigns POS tags itself and was developed and trained without exposure to a biological vocabulary, it was allowed to compete on its own terms against the other two parsers each in conjunction with MedPost. Although this may seem slightly unfair, to do otherwise would not reflect real-life usage scenarios. The parser tagged GENIA with an accuracy of 85%.

The PTB extract used was included preprocessed with the MXPOST tagger (Ratnaparkhi, 1996) as part of the Collins parser distribution; the supplied tagging scored 97% accuracy. The Charniak parser re-tagged this corpus with 96% accuracy.

## 4.2 Initial performance comparison

Having parsed each corpus with each parser, the output was post-processed into a standardised XML format. The same pruning operations performed on the original corpora (see Section 3) were repeated where necessary. TOP nodes (S1 nodes in the case of the Charniak parser) were removed from all files as these remain constant across every sentence. NAC and NX labels were replaced by NP labels in the parses of GENIA as the GENIA annotators use NP labels where these would occur. We then performed lineage- and

| | Raw scores on GENIA (1757 sentences) | | | | | |
|---|---|---|---|---|---|---|
| Parser | LA score | Precision | Recall | F-measure | % perfect | % failure |
| Bikel 0.9.8 | 91.12 | 81.33 | 77.43 | 79.33 | 14.29 | 0.06 |
| Bikel 0.9.9 | 65.30 | 81.68 | 55.75 | 66.27 | 11.21 | 25.04 |
| Charniak | 89.91 | 77.12 | 76.05 | 76.58 | 12.81 | 0.00 |
| Collins 1 | 88.74 | 79.06 | 73.87 | 76.38 | 13.15 | 0.68 |
| Collins 2 | 87.85 | 81.30 | 74.49 | 77.75 | 14.00 | 1.42 |
| Collins 3 | 86.33 | 81.57 | 73.28 | 77.20 | 14.00 | 2.28 |
| | Raw scores on PTB (2416 sentences) | | | | | |
| Parser | LA score | Precision | Recall | F-measure | % perfect | % failure |
| Bikel 0.9.8 | 94.45 | 88.09 | 88.13 | 88.11 | 33.44 | 0.04 |
| Bikel 0.9.9 | 80.11 | 88.03 | 74.61 | 80.76 | 29.80 | 12.75 |
| Charniak | 94.36 | 88.09 | 88.28 | 88.18 | 35.06 | 0.00 |
| Collins 1 | 94.17 | 86.80 | 86.70 | 86.75 | 31.13 | 0.00 |
| Collins 2 | 94.36 | 87.29 | 87.20 | 87.24 | 33.49 | 0.04 |
| Collins 3 | 94.25 | 87.28 | 87.10 | 87.19 | 33.11 | 0.08 |

Table 3: Initial performance comparison.

constituent-based scoring runs using our own Perl scripts.

The results of this experiment are summarised in Table 3, showing both the scores on both GENIA and the PTB. The LA score given is the mean of the leaf-ancestor scores for all the words in the corpus, and the precision and recall scores are taken over the entire set of constituents in the corpus. Initially, these measures were calculated per sentence, and then averaged across each corpus, but the presence of pathologically short sentences such as *Energy.* gives an unrepresentative boost to per-sentence averages. (Interestingly, many published papers do not make clear whether the results they present are per-sentence averages or corpus-wide scores.)

'Mean X' is simply the average number of crossing brackets per sentence. 'F-measure' (van Rijsbergen, 1979) is the harmonic mean of precision and recall; it is a balanced score that penalises algorithms which favour one to the detriment of the other, and is calculated as follows:

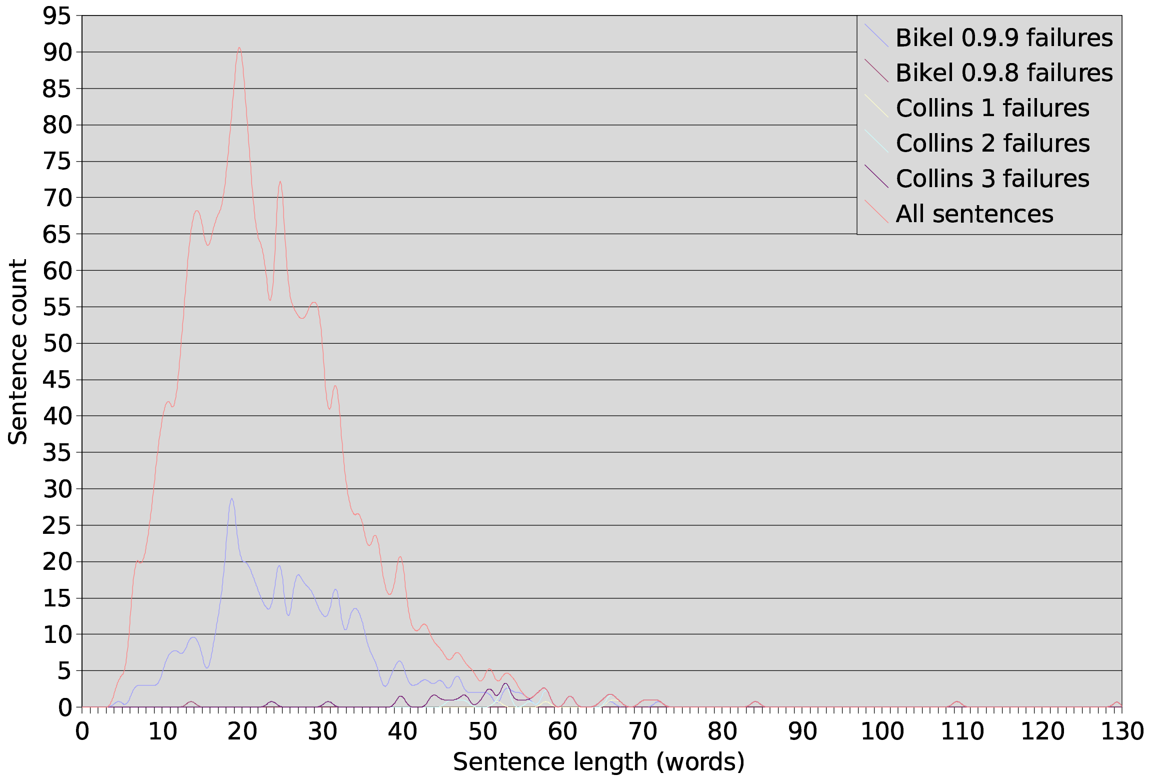
$$F = \frac{2 \times P \times R}{P + R}$$

### 4.3 Parse failures

Since most of the parsers suffered from a considerable number of parse failures in GENIA – sentences where no parse could be obtained – Table 4 shows recalculated scores based on evaluation of successfully-parsed sentences only. Conflating the performance drops caused by poorly parsed sentences with those caused by total failures gives an inaccurate picture of parser behaviour. In order to determine if there was any pattern to these failures, we plotted the number of parse failures for each parser against sentence length and sentence complexity (see Figure 5). These charts revealed some interesting trends.

There is a known problem with the Collins models 2 and 3 failing on two sentences in the PTB section 23 due to complexity, but this problem is exacerbated in GENIA, with even the simpler model 1 failing on a number of sentences, one of which was only 24 words long plus punctuation.

Overall, however, the failures do tend to cluster around the right-hand tails of the sentence length and constituent count distributions. Discounting such sentences, the three models do show a consistent monotonic increase in precision, recall and LA score from the simplest to the most complex, accompanied by a decrease in the number of crossing brackets per sentence. Interestingly, these intervals are much more pronounced on GENIA than on the PTB, where the performance seems to level off between models 2 and 3. Difficult sentences aside, then, it appears that the advanced fea-



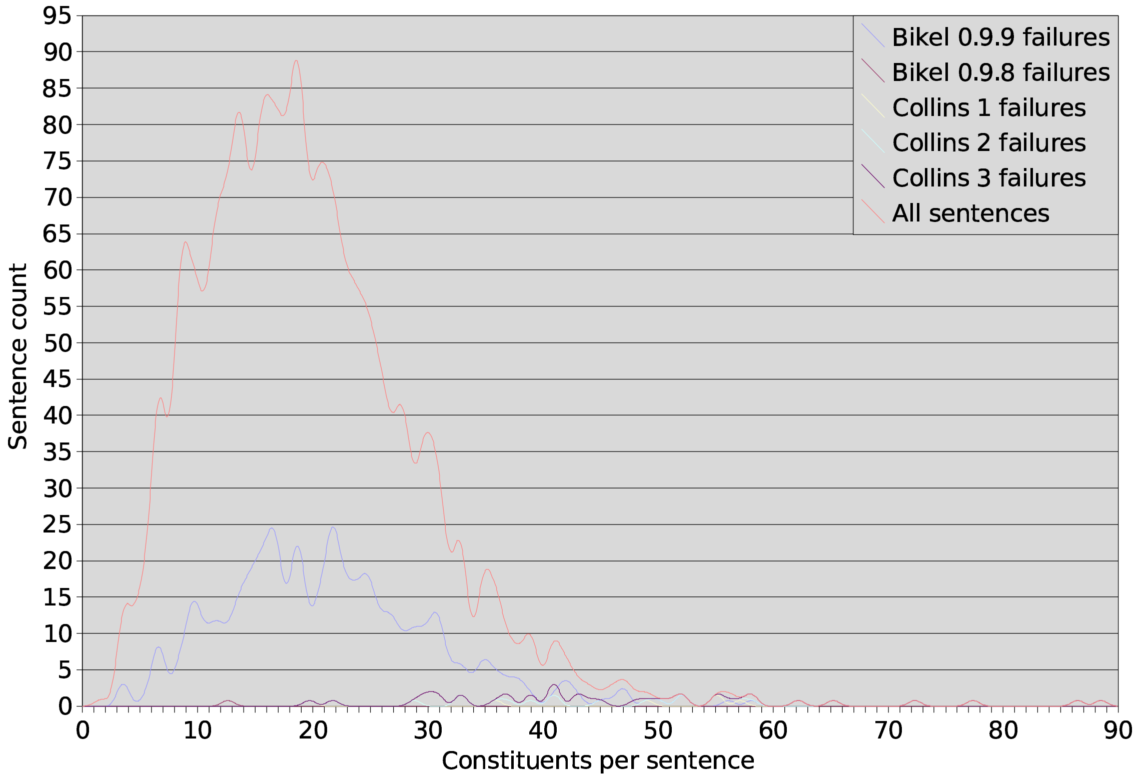


Figure 5: Parse failures on GENIA vs. sentence length and complexity for each parser.

| | Scores on GENIA, successfully-parsed sentences only | | | | | |
|---|---|---|---|---|---|---|
| Parser | LA score | Precision | Recall | F-measure | Mean X | # parsed |
| Bikel 0.9.8 | 91.15 | 81.33 | 77.46 | 79.35 | 2.06 | 1756 |
| Bikel 0.9.9 | 91.17 | 81.68 | 77.04 | 79.29 | 1.89 | 1317 |
| Charniak | 89.91 | 77.12 | 76.05 | 76.58 | 2.42 | 1757 |
| Collins 1 | 90.53 | 79.06 | 75.35 | 77.16 | 2.29 | 1745 |
| Collins 2 | 91.21 | 81.30 | 77.24 | 79.22 | 2.01 | 1732 |
| Collins 3 | 91.32 | 81.57 | 77.42 | 79.44 | 1.95 | 1717 |
| | Scores on PTB, successfully-parsed sentences only | | | | | |
| Parser | LA score | Precision | Recall | F-measure | Mean X | # parsed |
| Bikel 0.9.8 | 94.53 | 88.09 | 88.20 | 88.15 | 1.07 | 2415 |
| Bikel 0.9.9 | 94.52 | 88.03 | 88.07 | 88.05 | 1.04 | 2108 |
| Charniak | 94.36 | 88.09 | 88.28 | 88.18 | 1.08 | 2416 |
| Collins 1 | 94.17 | 86.80 | 86.70 | 86.75 | 1.23 | 2416 |
| Collins 2 | 94.45 | 87.29 | 87.28 | 87.28 | 1.19 | 2415 |
| Collins 3 | 94.44 | 87.28 | 87.27 | 87.28 | 1.18 | 2414 |

Table 4: Performance scores, discounting all parse failures. Scores for the Charniak parser, and Collins model 1 on the PTB, are shown again for comparison, although they did not fail on any sentences.

tures of models 2 and 3 are actually more valuable on this unfamiliar corpus than on the original development domain – provided that they do not trip the parser up completely.

While Bikel 0.9.8's failures are relatively few and tend to occur more often in longer and more complex sentences, like those of the Collins models, the distributions in Figure 5 for Bikel 0.9.9 follow the shapes of the distributions remarkably accurately. In other words, the length or complexity of a sentence does not seem to be a major influence on the ability of Bikel 0.9.9 to parse it. Undoubtedly, there is something more subtle in the composition of these sentences that confuses Bikel's updated algorithm, although we could not discern any pattern by eye. Perhaps this problem could be diagnosed by monitoring the parser in a Java debugger or modifying it to produce more verbose output, but such an examination is beyond the scope of this work.

Although version 0.9.9 fails on far fewer sentences in the PTB than in GENIA, it still suffers from two orders of magnitude more failures than any other parser on the same corpus. These results suggest that the author's claim that parameter pruning results in "no loss of accuracy" (Bikel, 2004) can only be taken seriously when the test set has been cleaned of all unparseable sentences; this impression is reinforced by the fact that the precision and recall scores reported by the author agree quite closely with our results on the PTB once the parse failures have been removed.

## 5 Combining the parsers

Given the poorer results of these parsers on GENIA than on the PTB, and the comparative lack of annotated data in this domain, it is important to consider ways in which performance can be enhanced without recourse to supervised training methods. Various experimental techniques exist for reducing or eliminating the need for labelled training data, particularly in the presence of several diverse parsers (or more generally, classifiers). These include active learning (Osborne and Baldridge, 2004), bagging and boosting (Henderson, 1999) and co-training (Steedman et al., 2003). In addition to these 'knowledge-poor' techniques, one can easily imagine domain-specific 'knowledge-rich' techniques that employ existing biological data sources and NLP methods in order to select, modify, or constrain parses (see Section 7.2). For this preliminary investigation, however, we concentrated on knowledge-poor methods originating in work on parsing the

PTB which could exploit the availability of multiple parsers whilst requiring no time-consuming retraining processes or integration with external resources. Perl implementations of the algorithms discussed below can be downloaded from our website.

### 5.1 Fallback cascades

In the Collins parser instructions, the author suggests stacking the three models in decreasing order of sophistication ($3 \rightarrow 2 \rightarrow 1$), and for each sentence, falling back to the next less sophisticated model each time a more sophisticated one fails to obtain a parse. The principle behind this is that the more complex a model is, the more often it will fail, but the better the results will be when it does return a parse. We implemented this system for the Collins models, and also for the Bikel parser, starting with version 0.9.9 and falling back to 0.9.8 on failure. Since the Charniak parser did not suffer any failures, we added it to each of these cascades as a last-resort level, to fill any remaining gaps.

As expected, the results for each cascade (Table 5) were comparable to their component parsers' scores on successfully-parsed sentences (Table 4), except with 100% coverage of the corpus. In each of the following parser integration methods, we used these fallback cascades to represent the Bikel and Collins parsers, rather than any of their individual parser models. The Bikel cascade was used as a baseline against which to test the results of each method for statistical significance, using a two-tailed dependent t-test over paired scores.

### 5.2 Constituent voting

Henderson (1999) reports good results when using a simple parse integration method called constituent voting, where a hybrid parse is produced by taking votes from all the parsers in an ensemble. Essentially, all the constituents proposed by the parsers are pooled, and each one is added to the hybrid parse if more than half of the parsers in the ensemble agree on it. The assumption behind this concept is that the mistakes made by the parsers are reasonably independently distributed – if different kinds of errors beset the parsers, and at different times, then a majority vote will tend to converge on the correct set of constituents.

We implemented a three-way majority vote ensemble between the Collins and Bikel cascades and the Charniak parser; the results are shown in Table 6. The most notable gain was in precision, as one would hope from an algorithm designed to screen out minority parsing decisions, but the scores also illustrate an interesting phenomenon. Although the ensemble took the lead on all the constituent-based performance indicators, it performed poorly on LA score. This demonstrates an important point about parser scoring metrics – that an algorithm designed to boost one measure of quality can do so without necessarily raising performance according to a different yardstick.

Part of the reason for this discrepancy may be a quirk of the constituent voting algorithm that constituent-based precision and recall scores gloss over. The trees it produces are not guaranteed to be well-formed under the grammars of any of the members of the ensemble; if, for example, the parsers cannot reach consensus about the exact boundaries of a verb phrase, a sentence without a VP constituent will be produced, leading to some unusual attachments at a higher level. Unlike the constituent-based approach, LA scoring tends to favour parses that are accurate at the upper levels of the tree, so an increase in precision and recall without a corresponding increase in LA score would be consistent with this kind of oddity.

### 5.3 Parse selection

An alternative approach to integrating the outputs of a parser ensemble is whole parse selection on a per-sentence basis, which has the potential added advantage over hybridisation methods like constituent voting that the gaps in trees described above cannot occur. The most obvious way to guess the best candidate parse for a sentence is to assume that the true parse lies close to the centroid of all the candidates in parse space, and then, using some similarity or distance measure between the candidates, pick the candidate that is most similar to (or least distant from) all the other parses.

We implemented three parse switchers, two based on constituent overlap, and one based on lineage similarity between pairs of parses. Similarity and distance switching (Henderson, 1999) take the

| | Ensemble scores on GENIA, all sentences parsed successfully | | | | | |
|---|---|---|---|---|---|---|
| Ensemble | LA score | Precision | Recall | F-measure | Mean X | % perfect |
| Collins-Charniak fallback | 90.74 | 80.51 | 76.44 | 78.42 | 2.16 | 14.00 |
| Bikel-Charniak fallback | 91.08 | 81.31 | 76.96 | 79.08 | 2.05 | 14.11 |

Table 5: Ensemble scores on GENIA for Collins(3, 2, 1)→Charniak and Bikel(0.9.9, 0.9.8)→Charniak fallback cascades.

| | Ensemble scores on GENIA, all sentences parsed successfully | | | | | |
|---|---|---|---|---|---|---|
| Algorithm | LA score | Precision | Recall | F-measure | Mean X | % perfect |
| Majority vote ensemble | 90.21 | 83.41 | 77.50 | 80.35 | 1.71 | 14.68 |

Table 6: Ensemble scores on GENIA for parse combination by majority constituent voting.

number of constituents that each parse has in common, and the number that are proposed by either one but not both parsers, as measures of similarity and distance respectively. Levenshtein switching, a novel method, uses the sentence-mean LA scores between parses as the similarity measure. In all methods, the parse with the maximum total pairwise similarity (minimum total pairwise distance) to the set of rival parses for a sentence is chosen. In no case were POS tags taken into account when calculating similarity, as they would have made the Collins and Bikel parsers artificially similar.

The results of these experiments are shown in Table 7. All three methods achieved comparable improvements overall, with the similarity and distance switching routines favouring recall and precision respectively (both differences significant at $p < 0.0001$). Note however that the winning LA score for Levenshtein switching is not a statistically significant improvement over the other switching methods.

## 6 Error analysis

All of the parser integration methods discussed above make the assumption that the parsers in an ensemble will suffer from independently-distributed errors, to a greater or lesser extent. Simple fallback cascades rely on their individual members failing on different sentences, but the more sophisticated methods in Section 5.2 and Section 5.3 are all ultimately based on the principle that agreement between parsers indicates convergence on the true parse. Although the performance gains we achieved with such methods are statistically significant, they are nonetheless somewhat unimpressive compared to the 30% reduction of recall errors and 6% reduction of precision errors reported for the best ensemble techniques in Henderson and Brill (1999) on the PTB.

This led us to suspect that the parsers in the ensembles were making similar kinds of errors on GENIA, perhaps not across the board, but certainly often enough that consensus methods pick incorrect constituents, and centroid methods converge on incorrect parses, with a significant frequency. To investigate this phenomenon, and more generally to tease apart the reasons for each parser's performance drop on GENIA, we measured the precision and recall for each parser on each production rule over GENIA and the PTB. We then gathered the 25 most common production rules in GENIA and compared the scores achieved by each parser on each rule to the same rule in PTB, thus drawing attention to parser-specific issues and more widespread systematic errors. We also collected closest-match data on each missed production in GENIA, for each parser, and calculated substitution frequencies for each production rule. This enabled us to identify both the sources of performance problems, and to a certain extent their causes and connotations. These data tables have been omitted for space reasons, since the discussion below covers the important lessons learnt from them, but they are available as supplementary materials on our website.

| | Ensemble scores on GENIA, all sentences parsed successfully | | | | | |
|---|---|---|---|---|---|---|
| Algorithm | LA score | Precision | Recall | F-measure | Mean X | % perfect |
| Similarity switching | 91.34 | 81.73 | 78.01 | 79.83 | 1.97 | 14.85 |
| Distance switching | 91.35 | 82.10 | 77.72 | 79.85 | 1.92 | 15.08 |
| Levenshtein switching | 91.39 | 81.83 | 77.51 | 79.61 | 1.95 | 14.74 |

Table 7: Ensemble scores on GENIA for parse selection by three centroid-distance algorithms

## 6.1 Bikel parser errors

Despite the similar overall LA score and F-measure for the two versions on parseable sentences only, there are signs that the differences between them run deeper than failure rates. The newer version's higher precision and lower crossing brackets per sentences, along with lower recall, indicates that it is generating slightly more conservatively than the older version, on GENIA at least; these scores are much closer on the PTB. Also, the production rule scores show one unexpected phenomenon – the older version is actually considerably better at labelling noun phrases of the form `( NP )` as parenthetical expressions in GENIA ($F = 81.07$) than in PTB ($F = 61.29$), as are the Collins and Charniak parsers, while the newer version is much worse at this task in GENIA ($F = 42.36$). On closer inspection, however, 84% of the occurrences of `PRN` $\rightarrow$ `( NP )` mislabelled by the newer version are instead marked as `PRN` $\rightarrow$ `( NN )` productions of the same width – in other words, an intermediate `NP` constituent covering just a single noun has in these cases been removed, and the noun 'promoted' to a direct daughter of the `PRN` constituent. Although this demonstrates a difference in the modelling of noun phrases between the two versions, it is unlikely that such a difference would alter the meaning of a sentence. Furthermore, it must be noted that `PRN` $\rightarrow$ `( NP )` is much more common in GENIA than in PTB, so the improvements achieved by the other parsers may be partially accidental; even if they simply assumed that every phrase in parentheses is a noun phrase, they would do better on this production in GENIA as a result.

## 6.2 Charniak parser errors

The Charniak parser goes from state-of-the-art on PTB to comparatively poor on GENIA. It ranks lowest in both LA score and F-measure when only successfully parsed sentences are taken into account, and still only achieves mediocre performance when the other parsers' scores cover failed sentences too, despite not failing on any sentences itself. This discrepancy can be explained by a lack of biomedical vocabulary available to its built-in POS tagger. Although it tags GENIA with an accuracy of 85% across all word classes, it achieves only 63% on the `NN` (singular/mass noun) class. This is the most numerous single class in GENIA, and that which many domain-specific single-word terms and components of multi-word phrases belong to.

The knock-on syntactic effects of this disability can be traced in the leaf-ancestor metrics, where the parser scores an impressive mean of 91.50 for correctly-tagged words, compared to just 80.71 for incorrectly-tagged words. A similar effect can be seen in the statistics for productions with `NN` tags on the right hand side. `NP` $\rightarrow$ `NN` and `NP` $\rightarrow$ `NN NN` are identified with respective recalls of only 33% and 19% in GENIA, for example, as opposed to 90% and 82% in the PTB. More than 40% of mislabelled `NP` $\rightarrow$ `NN` productions in GENIA were identified instead as `NP` $\rightarrow$ `NNP` (proper noun) or `NP` $\rightarrow$ `NNS` (plural noun) productions by the parser, and the implications of these mistakes for information extraction tasks do not seem great, especially since the majority of single-word noun phrases of particular interest in this domain are likely to be genes, proteins etc. that can be tagged independently by a dedicated named-entity recognizer. The story is different for mislabelled `NP` $\rightarrow$ `NN NN` productions, where 29% are mistaken for `NP` $\rightarrow$ `JJ NN` productions, a substitution that one can imagine causing greater semantic confusion. On the other hand, the Charniak parser goes from being the worst at identifying ADVP $\rightarrow$ RB productions (single-word adverb phrases) on the PTB ($F = 87.68$) to being the best at this task on GENIA ($F = 87.06$).

Accuracy issues notwithstanding, Charniak's is still the most robust of all the parsers, failing on none of the supplied sentences in either corpus. This may reflect a strategy of 'making-do' when an exact parse cannot be derived; it deployed more general-purposes FRAG (fragment) and X (unknown/unparseable) constituents combined than any other parser, and even a handful of INTJ (interjection) phrases that no other parser used in GENIA. Of course, such productions are not always correct – there are actually no interjections in GENIA – but from an information extraction point of view, a rough parse may be better than no parse at all, especially if the inclusion of such inexact labels can be reflected in a reduced level of confidence or trustworthiness for the sentence.

## 6.3 Collins parser errors

We noted in Section 4.2 that the Collins models achieved successively better performance on GENIA once parse failures were discounted. Considering individual production rules, however, the trends are not so clear-cut. There are rules that do follow this overall pattern, such as S $\rightarrow$ NP VP, which model 3 actually assigns more effectively on GENIA ($F = 88.98$) than it does on the PTB ($F = 88.79$). However, there are several common productions where model 3's accuracy degrades more than model 2's, most of which begin with NP $\rightarrow$ .... Most of these are found in the PTB more effectively by model 3 than model 2, which suggests over-fitting; these specific increases in performance have apparently come at the expense of portability. Note, however, the caveat regarding noun phrases below.

## 6.4 Common trends

In addition to these parser-specific observations, there are various phenomena that are common to all or most of the parsers. Due to slight differences in the annotation of co-ordinated structures between the GENIA and PTB guidelines, the correct generation of noun-phrase conjunctions (NP $\rightarrow$ NP CC NP) proved much harder on GENIA, with all parsers having problems with identification of the boundaries of the conjunction in the text, and often with correct labelling of the constituents involved too. Cases where each NP is a single word were handled relatively well, with the essentially equivalent NN CC NN construction often being proposed instead, but more complex cases caused widespread difficulty.

More surprisingly, the labelling of single-word noun and adjective phrases (NP $\rightarrow$ NN and AJDP $\rightarrow$ JJ), both of which are significantly more frequent in GENIA, seemed challenging across the board. The most commonly-occurring error involved subsumption by a wider constituent with the same label, apart from the errors of vocabulary for the Charniak parser as described above. However, for correctly-tagged adjectives and nouns, there are many situations where this will not make any difference to the sense of a sentence. For example, in a production like NP $\rightarrow$ DT JJ NN, the adjective still has its modificatory effect on the noun without needing to be placed within an ADJP phrase of its own, and the noun is entirely capable of acting as the head of the phrase without being nested within an NP sub-phrase.

Similar effects occurred frequently with longer phrases containing nouns, such as NP $\rightarrow$ DT NN or NP $\rightarrow$ NN NN, where the most common errors were also subsumptions by wider noun phrases. Although the GENIA annotators warn that "when noun phrase consists with sequence of nouns *[sic]*, the internal structure is not neccessarily shown,"[6] which must account for some of the noun handling problems, such subsumptions suggest that the opposite may be occurring too – that there are cases where the parsers are failing to generate internal structure within noun phrases.

Prepositions were involved in many of the problematic cases, both on the left-hand and right-hand sides of productions, and understandably so. The disambiguation of prepositional attachment is a continuing problem in parser design, and methods that take into account lexical dependencies between head words will be less effective when the words in question are out-of-domain and thus un-

[6] http://www-tsujii.is.s.u-tokyo.ac.jp/~genia/topics/Corpus/manual-for-bracketing.html

seen in training. The production `NP` → `NP PP PP` is a good example. The most common error for all parsers was to produce `NP` → `NP PP` at the same span of words in the sentence, indicating that one of the prepositional phrases is frequently attached at the wrong level. The second most common error was to substitute a shorter `NP`, suggesting that one or both of the `PP`s were excluded. Such errors are potentially of more serious semantic importance than differences of opinion about how to mark up the internal structure of noun-only phrases.

## 7 Discussion

Although the performance gains achieved by our parser integration methods are statistically significant, and illustrative of some important points about parser behaviour and syntactic evaluation methodologies, it is doubtful that the results are good enough to justify deploying these techniques on large amounts of text, at least in their current form. The small increases in accuracy are probably outweighed by the additional computational costs. The fallback cascades provided the same protection from parse failure, with better performance than the widest-coverage parser alone, and in a production system most sentences would only need to be parsed by the first parser in the cascade.

However, the parser integration idea as a whole is not without its merits; we determined that an oracle picking the best sentences on GENIA would achieve an LA score of 93.56, so there is still room for improvement if the algorithms can be made smarter. Although our analysis of the parsers' mistakes on GENIA indicated that the ideal of independently-distributed errors which underpins these integration methods does not hold true, the very fact that we can analyse their behaviour patterns in such detail suggests that a sufficiently well-designed ensemble could in principle learn the circumstances under which each parser could be trusted on a given corpus.

Furthermore, there are additional ways in which an ensemble might assist with practical NLP issues. While analysing the data from the parse selection algorithms, we discovered that the centroid distances for the *winning* parses, scaled by sentence size where necessary, correlate fairly well ($|r| \approx 0.5$) with those parses' true LA scores and F-measures (see Section 7.2). We developed a similar measurement for constituent voting based on the level of consensus among the ensemble members – the number of constituents winning a majority vote divided by the number of distinct candidate constituents – which showed a comparable degree of correlation. This provides an answer to our initial question about the extent to which parser agreement signals parse correctness. Presumably the major limiting factor on these correlations is the presence of widespread systematic errors like those described in Section 6.4.

### 7.1 Engineering issues

As noted previously, the Charniak parser takes raw text and performs tokenisation and POS tagging internally. While this may seem like an advantageous convenience, in practice it is the source of considerable extra work, besides being the cause of avoidable parse errors. The tokenisation standards encoded by Charniak did not match those assumed by either the GENIA corpus, or indeed the PTB extract, although problems were much more widespread in the GENIA. Words containing embedded punctuation were frequently split into multiple tokens, so these word-internal symbols had to be converted into textual placeholders before parsing and converted back afterwards. The somewhat idiosyncratic conventions of the GENIA corpus did not help (*differentiation/activation* being tagged as one token for example) but the fact that similar issues occurred on the newspaper corpus (e.g. with *US$* or *81-year-old*) suggests that making assumptions about the 'correct' way to tokenise text is a bad policy in any domain.

Even when working on in-domain data, it seems like a bad design decision to assume the parser will be able to match the performance of a state-of-the-art POS tagger on unseen text. The Bikel parser can operate in either mode, which is a much more flexible policy. In all fairness, however, it would probably be fairly trivial for an interested C++ developer to bypass the Charniak parser's tokeniser and tagger and recompile it.

A different kind of engineering issue is that of computation time. Parsing is a slow process in any case, and ensemble methods compound this problem. However, parsing is a canonically easy task to perform in parallel, since (at this level of understanding at least) each sentence has no dependencies on the previous, so even the parse integration step can be split across multiple pipelines. We intend to run pilot studies on the scalability of parallel distributed parsing techniques, both on an IBM Blade Center cluster running Beowulf Linux, and a heterogeneous network of Windows PCs in their spare hours, in order to determine the feasibility and comparative attractiveness of each approach.

### 7.2 Future work

We mentioned above that there is a significant correlation between the distance of a winning parse from the centroid and its accuracy compared to the gold standard. In an IE or other text mining scenario, one could use these values as estimators of trustworthiness for each parse – empirical measures indicating how reliable to consider it. We would like to explore this idea further, as it can provide an extra level of understanding which is missing from many IE techniques, and could be easily employed in the ranking of extracted 'facts' or the resolution of contradictions. Since LA scores can be calculated per word or per any arbitrary region of the sentence, the potential even for rating the trustworthiness of different clauses or relationships individually cannot be ignored.

We have been experimenting with training a neural network to pick the best parse for a sentence based only on the pairwise Levenshtein distances between the candidate parses, in the hope that it can learn what decision to make based on patterns of agreement between the parsers, rather than just picking the parse which is most similar to all the others as the current methods do. So far, however, it has been unable to exceed the performance of the unsupervised methods, which suggests that additional features may need to be considered.

A complementary approach to parse integration would be to merge the PTB with an annotated biomedical corpus and retrain a parser from scratch. This proposition appears more attractive in the light of the production rule frequency differences between GENIA and the PTB, and will become more effective as the GENIA treebank grows and the Mining the Bibliome project begins posting official releases. In the meantime, we have begun investigating the potential for using biological named-entity and ontological-class information to help rule out unlikely parses, for example in cases where an entity name is bisected by a constituent boundary.

## Acknowledgements

This work was funded by the Biotechnology and Biological Sciences Research Council and AstraZeneca PLC. We would like to thank the reviewers for their insights and suggestions, and Mark McLauchlan for his help with prepositions.

## References

Daniel M. Bikel. 2002. Design of a multi-lingual, parallel-processing statistical parsing engine. In *Proceedings of the Human Language Technology Conference 2002 (HLT2002)*. San Diego.

Daniel M. Bikel. 2004. A distributional analysis of a lexicalized statistical parsing model. In *Proceedings of the 2004 Conference on Empirical Methods in Natural Language Processing (EMNLP 2004)*. ACL, Barcelona.

Eugene Charniak. 1999. A maximum-entropy-inspired parser. Technical report, Brown University.

Kevin B. Cohen and Lawrence Hunter. 2004. Natural language processing and systems biology. In Werner Dubitzky and Francisco Azuaje, editors, *Artificial Intelligence Methods and Tools for Systems Biology*. Springer Verlag, Heidelberg.

Michael Collins. 1999. *Head-Driven Statistical Models for Natural Language Parsing*. PhD, University of Pennsylvania.

Richard Crouch, Ronald Kaplan, Tracy King, and Stefan Riezler. 2002. A comparison of evaluation metrics for a broad coverage parser. In *Proceedings of the LREC-2002 workshop "Beyond PARSEVAL: Towards Improved Evaluation Measures for Parsing Systems"*. Las Palmas, Spain.

Carol Friedman, Pauline Kra, and Andrey Rzhetsky. 2002. Two biomedical sublanguages: a description based on the theories of Zellig Harris. *Journal of Biomedical Informatics*, 35(4):222–235.

John C. Henderson. 1999. *Exploiting Diversity for Natural Language Parsing*. PhD, Johns Hopkins University.

John C. Henderson and Eric Brill. 1999. Exploiting diversity in natural language processing: Combining parsers. In *Proceedings of the Fourth Conference on Empirical Methods in Natural Language Processing (EMNLP99)*. University of Maryland.

Jin-Dong Kim, Tomoko Ohta, Yuka Tateisi, and Jun'ichi Tsujii. 2003. GENIA corpus – a semantically annotated corpus for bio-textmining. *Bioinformatics*, 19(Suppl. 1):i180–i182.

Seth Kulick, Ann Bies, Mark Liberman, Mark Mandel, Ryan McDonald, Martha Palmer, Andrew Schein, Lyle Ungar, Scott Winters, and Pete White. 2004. Integrated annotation for biomedical information extraction. In Lynette Hirschman and James Pustejovsky, editors, *HLT-NAACL 2004 Workshop: BioLINK 2004, Linking Biological Literature, Ontologies and Databases*, pages 61–68. Association for Computational Linguistics, Boston, Massachusetts, USA.

Mitchell P. Marcus, Beatrice Santorini, and Mary Ann Marcinkiewicz. 1994. Building a large annotated corpus of English: The Penn Treebank. *Computational Linguistics*, 19(2):313–330.

Miles Osborne and Jason Baldridge. 2004. Ensemble-based active learning for parse selection. In Daniel Marcu Susan Dumais and Salim Roukos, editors, *HLT-NAACL 2004: Main Proceedings*, pages 89–96. Association for Computational Linguistics, Boston, Massachusetts, USA.

Adwait Ratnaparkhi. 1996. A maximum entropy part-of-speech tagger. In *Proceedings of the Empirical Methods in Natural Language Processing Conference*. University of Pennsylvania.

Eric Ringger, Robert C. Moore, Eugene Charniak, Lucy Vanderwende, and Hisami Suzuki. 2004. Using the Penn Treebank to evaluate non-treebank parsers. In *Proceedings of 4th International Conference on Language Resource and Evaluation (LREC2004)*, volume IV. ELDA.

Brian Roark. 2002. Evaluating parser accuracy using edit distance. In *Proceedings of the LREC-2002 workshop "Beyond PARSEVAL: Towards Improved Evaluation Measures for Parsing Systems"*. Las Palmas, Spain.

Geoffrey Sampson and Anna Babarczy. 2003. A test of the leaf-ancestor metric for parse accuracy. *Journal of Natural Language Engineering*, 9(4):365–380.

Hagit Shatkay and Ronen Feldman. 2003. Mining the biomedical literature in the genomic era: An overview. *Journal of Computational Biology*, 10(6):821–856.

Larry H. Smith, Thomas Rindflesch, and W. John Wilbur. 2004. MedPost: a part-of-speech tagger for biomedical text. *Bioinformatics*, 20(14):2320–2321.

Mark Steedman, Steven Baker, Stephen Clark, Jay Crim, Julia Hockenmaier, Rebecca Hwa, Miles Osborne, Paul Ruhlen, and Anoop Sarkar. 2003. CLSP WS-02 final report: Semi-supervised training for statistical parsing. Technical report, Johns Hopkins University.

Cornelis J. van Rijsbergen. 1979. *Information Retrieval, 2nd edition*. Butterworths, London.

Juan Xiao, Jian Su, GuoDong Zhou, and ChewLim Tan. 2005. Protein-protein interaction: A supervised learning approach. In *Proceedings of the First International Symposium on Semantic Mining in Biomedicine*. Hinxton, UK.

# Interleaved Preparation and Output in the COMIC Fission Module

**Mary Ellen Foster**
Institute for Communicating and Collaborative Systems
School of Informatics, University of Edinburgh
2 Buccleuch Place, Edinburgh EH8 9LW United Kingdom
M.E.Foster@ed.ac.uk

## Abstract

We give a technical description of the fission module of the COMIC multimodal dialogue system, which both plans the multimodal content of the system turns and controls the execution of those plans. We emphasise the parts of the implementation that allow the system to begin producing output as soon as possible by preparing and outputting the content in parallel. We also demonstrate how the module was designed to ensure robustness and configurability, and describe how the module has performed successfully as part of the overall system. Finally, we discuss how the techniques used in this module can be applied to other similar dialogue systems.

## 1 Introduction

In a multimodal dialogue system, even minor delays in processing at each stage can add up to produce a system that produces an overall sluggish impression. It is therefore critical that the output system avoid as much as possible adding any delays of its own to the sequence; there should be as little time as possible between the dialogue manager's selection of the content of the next turn and the start of that turn's output. When the output incorporates temporal modalities such as speech, it is possible to take advantage of this by planning later parts of the turn even as the earlier parts are being played. This means that the initial parts of the output can be produced more quickly, and any delay in preparing the later parts is partly or entirely eliminated. The net effect is that the overall perceived delay in the output is much shorter than if the whole turn had been prepared before any output was produced.

In this paper, we give a technical description of the output system of the COMIC multimodal dialogue system, which is designed to allow exactly this interleaving of preparation and output. The paper is arranged as follows. In Section 2, we begin with a general overview of multimodal dialogue systems, concentrating on the design decisions that affect how output is specified and produced. In Section 3, we then describe the COMIC multimodal dialogue system and show how it addresses each of the relevant design decisions. Next, in Section 4, we describe how the segments of an output plan are represented in COMIC, and how those segments are prepared and executed in parallel. In Section 5, we discuss two aspects of the module implementation that are relevant to its role within the overall COMIC system: the techniques that were used to ensure the robustness of the fission module, and how it can be configured to support a variety of requirements. In Section 6, we then assess the practical impact of the parallel processing on the overall system responsiveness, and show that the output speed has a perceptible effect on the overall user experiences with the system. Finally, in Section 7, we outline the aspects of the COMIC output system that are applicable to similar systems.

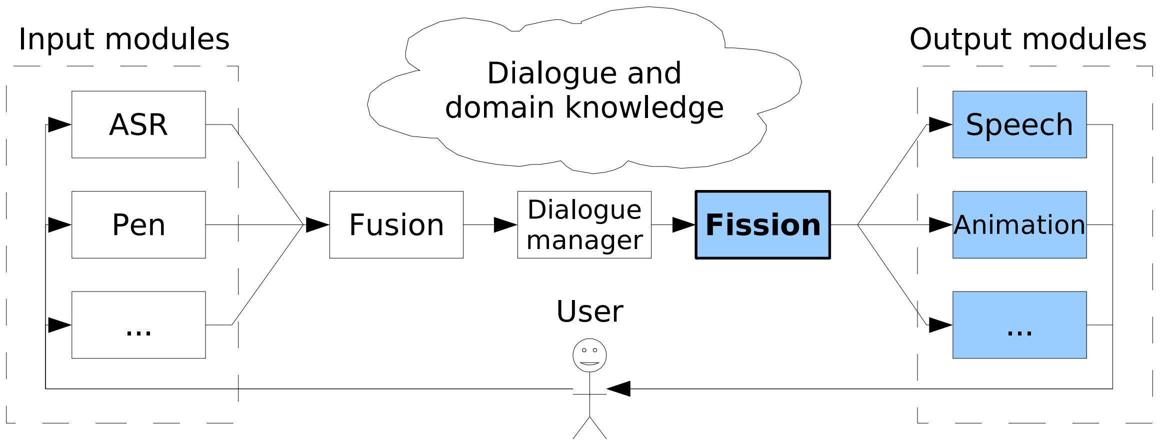


Figure 1: High-level architecture of a typical multimodal dialogue system

## 2 Output in Multimodal Dialogue Systems

Most multimodal dialogue systems use the basic high-level architecture shown in Figure 1. Input from the user is analysed by one or more input-processing modules, each of which deals with an individual input channel; depending on the application, the input channels may include speech recognition, pen-gesture or handwriting recognition, or information from visual sensors, for example. The messages from the various sources are then combined by a fusion module, which resolves any cross-modal references and produces a combined representation of the user input. This combined representation is sent to the dialogue manager, which uses a set of domain and dialogue knowledge sources to process the user input, interact with the underlying application if necessary, and specify the content to be output by the system in response. The output specification is sent to the fission module, which creates a presentation to meet the specification, using a combination of the available output channels. Again, depending on the application, a variety of output channels may be used; typical channels are synthesised speech, on-screen displays, or behaviour specifications for an animated agent or a robot.

This general structure is typical across multimodal dialogue systems; however, there are a number of design decisions that must be made when implementing a specific system. As this paper concentrates on the output components highlighted in Figure 1, we will discuss the design decisions that have a particular impact on those parts of the dialogue system: the domain of the application, the output modalities, the turn-taking protocol, and the division of labour among the modules. We will use as examples the the WITAS (Lemon et al., 2002), MATCH (Walker et al., 2002), and SmartKom (Wahlster, 2005) systems.

The domain of the system and the interactions that it is intended to support both have an influence on the type of output that is to be generated. Many systems are designed primarily to support information exploration and presentation, and concentrate on effectively communicating the necessary information to the user. SmartKom and MATCH both fall into this category: SmartKom deals with movie and television listings, while MATCH works in the domain of restaurant recommendations. In a system such as WITAS, which incorporates real-time control of a robot helicopter, very different output must be generated to communicate the current state and goals of the robot to the user.

The choice of output modalities also affects the output system—different combinations of modalities require different types of temporal and spatial coordination, and different methods of allocating the content across the channels. Most multimodal dialogue systems use synthesised speech as an output modality, often in combination with lip-synch and other behaviours of an animated agent (e.g., MATCH, SmartKom). Various types of visual output are also often employed, including interactive maps (MATCH, WITAS), textual information presentations (SmartKom, MATCH), or images from visual sensors (WITAS). Some systems also dynamically adapt the output channels based on changing constraints; for example, SmartKom chooses a spoken presentation over a visual one in an eyes-busy situation.

Another factor that has an effect on the design of the output components is the turn-taking protocol selected by the system. Some systems—such as WITAS—support *barge-in* (Ström and Seneff, 2000); that is, the user may interrupt the system output at any time. Allowing the user to interrupt can permit a more intuitive interaction with the system; however, supporting barge-in creates many technical complications. For example, it is crucial that the output system be prepared to stop at any point, and that any parts of the system that track the dialogue history be made aware of how much of the intended content was actually produced. For simplicity, many systems—including SmartKom—instead use half-duplex turn-taking: when the system is producing output, the input modules are not active. This sort of system is technically more straightforward to implement, but requires that the user be given very clear signals as to when the system is and is not paying attention to their input. MATCH uses a click-to-talk interface, where the user presses a button on the interface to indicate that they want to speak; it is not clear whether the system supports barge-in.

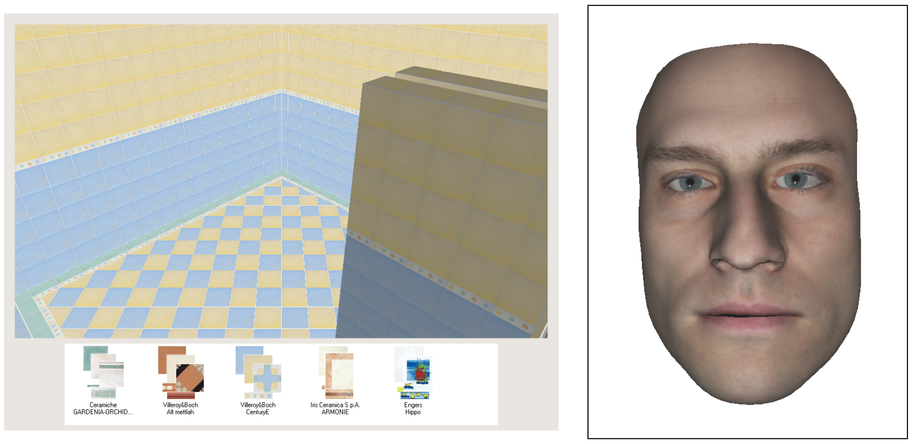
The division of labour across the modules also differs among implemented systems. First of all, not all systems actually incorporate a separate component that could be labelled *fission*: for example, in WITAS, the dialogue manager itself also addresses the tasks of presentation planning and coordination. The components of the typical natural language generation "pipeline" (Reiter and Dale, 2000) may be split across the modules in a variety of ways. When it comes to content selection, for instance, in MATCH the dialogue manager specifies the content at a high level, while the text planner selects and structures the actual facts to include in the presentation; in WITAS, on the other hand, the specific content is selected by the dialogue manager. The tasks of text planning and sentence planning may be addressed by various combinations of the fission module and any text-generation modules involved—SmartKom creates the text in a separate generation module, while in MATCH text and sentence planning is more tightly integrated with content selection.

Coordination across multiple output channels is also implemented in various ways. If the only presentation modality is an animated agent, in many cases the generated text is sent directly to the agent, which then communicates privately with the speech synthesiser to ensure synchronisation. This "visual text-to-speech" configuration is the default behaviour of the Greta (de Rosis et al., 2003) and RUTH (DeCarlo et al., 2004) animated presentation agents, for instance. However, if the behaviour of the agent must be coordinated with other forms of output, it is necessary that the behaviour of all synchronised modules be coordinated centrally. How this is accomplished in practice depends on the capabilities of selected speech synthesiser that is used. In SmartKom, for example, the presentation planner pre-synthesises the speech and uses the schedule returned by the synthesiser to create the full multimodal schedule; in MATCH, on the other hand, the speech synthesiser sends progress messages as it plays its output, which are used to control the output in the other modalities at run time.

## 3 The COMIC Dialogue System

COMIC[1] (COnversational Multimodal Interaction with Computers) is an EU IST 5th framework project combining fundamental research on human-human dialogues with advanced technology development for multimodal conversational systems. The COMIC multimodal dialogue system adds a dialogue interface to a CAD-like application used in sales situations to help clients redesign their bathrooms. The input to COMIC consists of speech, pen gestures, and handwriting; turn-taking is strictly half-duplex, with no barge-in or click-to-talk. The output combines the following modalities:

[1] `http://www.hcrc.ed.ac.uk/comic/`


Ceramiche GARDENIA-ORCHID...
Villeroy&Boch Alt mettlah
Villeroy&Boch CenturyE
Iris Ceramica S.p.A. ARMONIE
Engers Hippo

"*[Nod]* Okay. *[Choose design] [Look at screen]* THIS design *[circling gesture]* is CLASSIC. It uses tiles from VILLEROY AND BOCH'S CENTURY ESPRIT series. There are FLORAL MOTIFS and GEOMETRIC SHAPES on the DECORATIVE tiles."

Figure 2: COMIC interface and sample output

- Synthesised speech, created using the OpenCCG surface realiser (White, 2005a;b) and synthesised by a custom Festival 2 voice (Clark et al., 2004) with support for APML prosodic markup (de Carolis et al., 2004).
- Facial expressions and gaze shifts of a talking head (Breidt et al., 2003).
- Direct commands to the design application.
- Deictic gestures at objects on the application screen, using a simulated mouse pointer.

Figure 2 shows the COMIC interface and a typical output turn, including commands for all modalities; the small capitals indicate pitch accents in the speech, with corresponding facial emphasis.

The specifications from the COMIC dialogue manager are high-level and modality-independent; for example, the specification of the output shown in Figure 2 would indicate that system should show a particular set of tiles on the screen, and should give a detailed description of those tiles. When the fission module receives input from the dialogue manager, it selects and structures multimodal content to create an output plan, using a combination of scripted and dynamically-generated output segments. The fission module addresses the tasks of low-level content selection, text planning, and sentence planning; surface realisation of the sentence plans is done by the OpenCCG realiser. The fission module also controls the output of the planned presentation by sending appropriate messages to the output modules including the text realiser, speech synthesiser, talking head, and bathroom-design GUI. Coordination across the modalities is implemented using a technique similar to that used in SmartKom: the synthesised speech is prepared in advance, and the timing information from the synthesiser is used to create the schedule for the other modalities.

The plan for an output turn in COMIC is represented in a tree structure; for example, Figure 3 shows part of the plan for the output in Figure 2. A plan tree like this is created from the top down, with the children created left-to-right at each level, and is executed in the same order. The planning and execution processes for a turn are started together and run in parallel, which makes it possible to begin producing output as soon as possible and to continue planning while output is active. In the following section, we describe the set of classes and algorithms that make this interleaved preparation and execution possible.

The COMIC fission module is implemented in a combination of Java and XSLT. The current module consists of 18 000 lines of Java code in 88 source files, and just over 9000 lines of XSLT templates. In the diagrams and algorithm descriptions that follow, some non-essential details are omitted for simplicity.

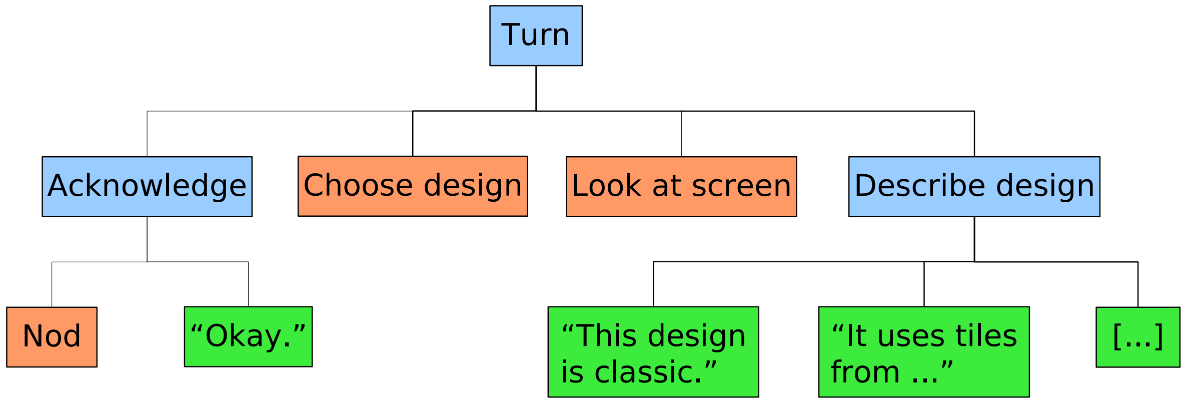


Figure 3: Output plan

## 4 Representing an Output Plan

Each node in a output-plan tree such as that shown in Figure 3 is represented by an instance of the Segment class. The structure of this abstract class is shown in Figure 4; the fields and methods defined in this class control the preparation and output of the corresponding segment of the plan tree, and allow preparation and output to proceed in parallel.

Each Segment instance stores a reference to its parent in the tree, and defines the following three methods:

- `plan()` Begins preparing the output.
- `execute()` Produces the prepared output.
- `reportDone()` Indicates to the Segment's parent that its output has been completed.

`plan()` and `execute()` are abstract methods of the Segment class; the concrete implementations of these methods on the subclasses of Segment are described later in this section. Each Segment also has the following Boolean flags that control its processing; all are initially false.

- `ready` This flag is set internally once the Segment has finished all of its preparation and is ready to be output.
- `skip` This flag is set internally if the Segment encounters a problem during its planning, and indicates that the Segment should be skipped when the time comes to produce output.

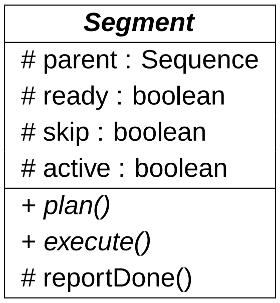


Figure 4: Structure of the Segment class

- `active` This flag is set externally by the Segment's parent, and indicates that this Segment should produce its output as soon as it is ready.

The activity diagram in Figure 5 shows how these flags and methods are used during the preparation and output of a Segment. Note that a Segment may send asynchronous queries to other modules as part of its planning. When such a query is sent, the Segment sets its internal state and exits its `plan()` method; when the response is received, preparation continues from the last state reached. Since planning and execution proceed in parallel across the tree, and the planning process may be interrupted to wait for responses from other modules, the `ready` and `active` flags may be set in either order on a particular Segment. Once both of these flags have been set, the `execute()` method is called automatically. If both `skip` and `active` are set, the Segment instead automatically calls `reportDone()` without ever ex-

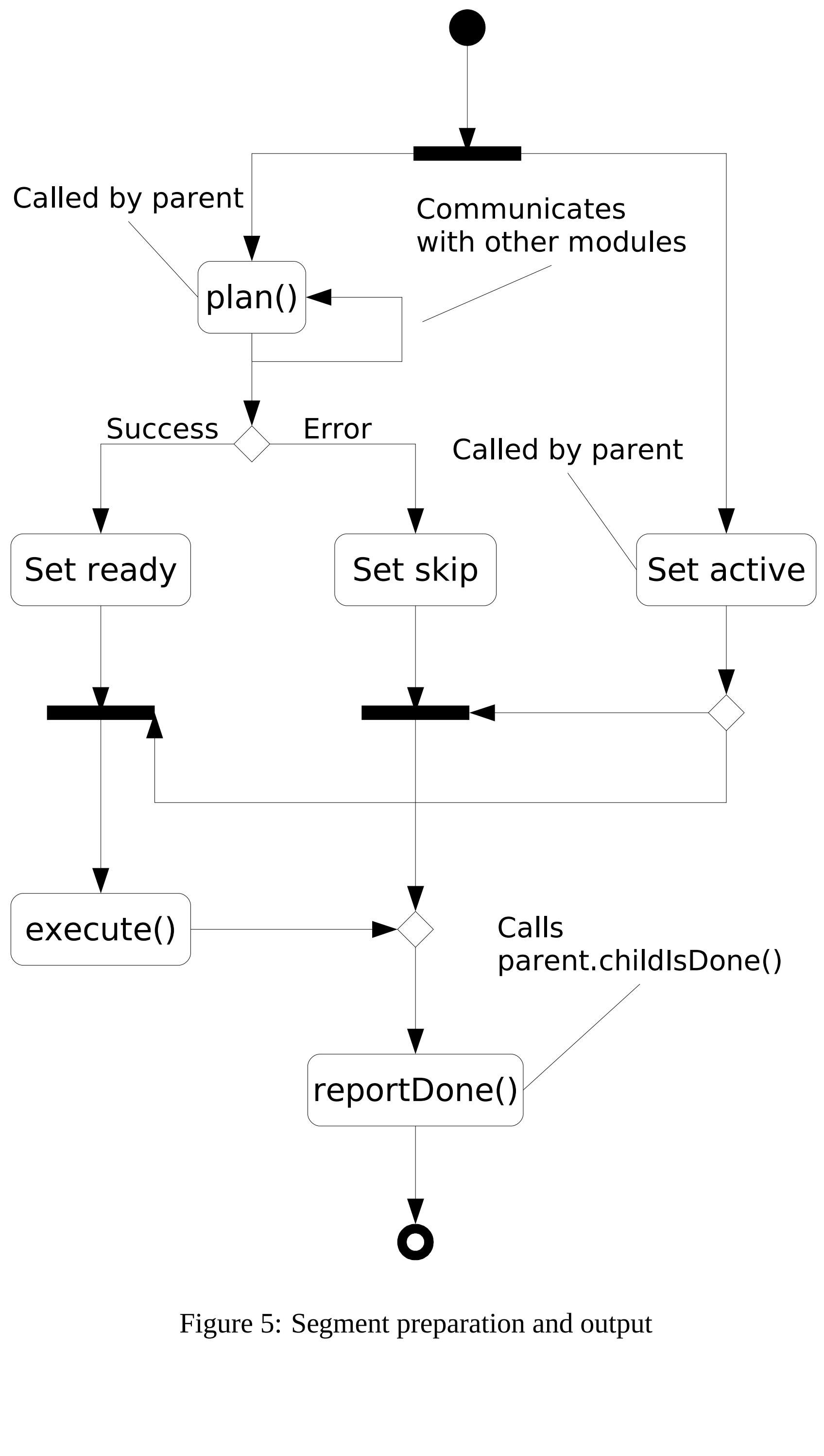


Figure 5: Segment preparation and output

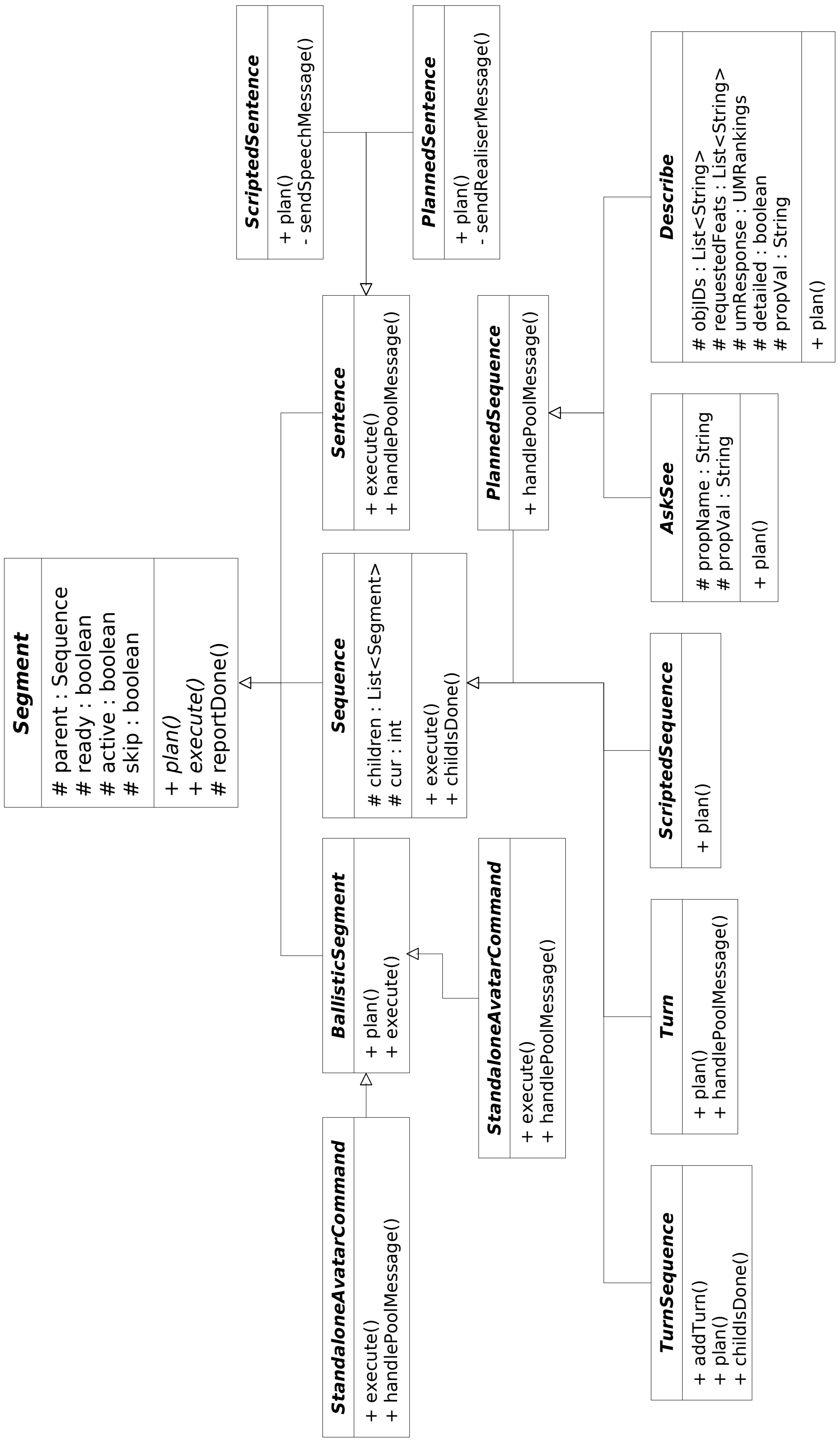


Figure 6: Segment class hierarchy

ecuting; this allows Segments with errors to be skipped without affecting the output of the rest of the turn.

The full class hierarchy under Segment is shown in Figure 6. There are three main top-level subclasses of Segment, which differ primarily based on how they implement `execute()`:

**Sequence** An ordered sequence of Segments. It is executed by activating each child in turn.

**BallisticSegment** A single command whose duration is determined by the module producing the output. It is executed by sending a message to the appropriate module and waiting for that module to report back that it has finished.

**Sentence** A single sentence, incorporating coordinated output in all modalities. Its schedule is computed in advance, as part of the planning process; it is executed by sending a "go" command to the appropriate output modules.

In the remainder of this section, we discuss each of these classes and its subclasses in more detail.

## 4.1 Sequence

All internal nodes in a presentation-plan tree (coloured blue in Figure 3) are instances of some type of Sequence. A Sequence stores a list of child Segments, which it plans and activates in order, along with a pointer to the currently active Segment. Figure 7 shows the pseudocode for the main methods of a typical Sequence.

Note that a Sequence calls sets its `ready` flag as soon as all of its necessary child Segments have been created, and only then begins calling `plan()` on them. This allows the Sequence's `execute()` method to be called as soon as possible, which is critical to allowing the fission module to begin producing output from the tree before the full tree has been created.

When `execute()` is called on a Sequence, it calls `activate()` on the first child in its list. All subsequent children are activated by calls to the `childIsDone()` method, which is called by each child as part of its `reportDone()` method after its execution is completed. Note that this ensures that the children of a Sequence will always be executed in the proper order, even if they are prepared out of order. Once all of the Sequence's children have reported that they are done, the Sequence itself calls `reportDone()`.

```
public void plan() {
    // Create child Segments

    cur = 0;
    ready = true;

    for( Segment seg: children ) {
        seg.plan();
    }
}

public void execute() {
    children.get( 0 ).activate();
}

public void childIsDone() {
    cur++;
    if( cur >= children.size() ) {
        reportDone();
    } else {
        children.get( cur ).activate();
    }
}
```

Figure 7: Pseudocode for Sequence methods

The main subclasses of Sequence, and their relevant features, are as follows:

**TurnSequence** The singleton class that is the parent of all Turns. It is always active, and new children can be added to its list at any time.

**Turn** Corresponds to a single message from the dialogue manager; the root of the output plan in Figure 3 is a Turn. Its `plan()` implementation creates a Segment corresponding to each dialogue act from the dialogue manager; in some cases, the Turn adds additional children not directly specified by the DAM, such as the verbal acknowledgement and the gaze shift in Figure 3.

**ScriptedSequence** A sequence of canned output segments stored as an XSLT template. A ScriptedSequence is used anywhere in the dialogue where dynamically-generated content is not necessary; for example, instructions to the user and acknowledgements such as the leftmost subtree in Figure 3 are stored as ScriptedSequences.

**PlannedSequence** In contrast to a ScriptedSequence, a PlannedSequence creates its children dynamically depending on the dialogue context. The principal type of PlannedSequence is a description of one or more tile designs, such as that shown in Figure 2. To create the content of such a description, the fission module uses information from the system ontology, the dialogue history, and the model of user preferences to select and structure the facts about the selected design and to create the sequence of sentences to realise that content. This process is described in detail in (Foster and White, 2004; 2005).

### 4.2 BallisticSegment

A BallisticSegment is a single command for a single output module, where the output module is allowed to choose the duration at execution time. In Figure 3, the orange *Nod*, *Choose design*, and *Look at screen* nodes are examples of BallisticSegments. In its `plan()` method, a BallisticSegment transforms its input specification into an appropriate message for the target output module. When `execute()` is called, the BallisticSegment sends the transformed command to the output module and waits for that module to report back that it is done; it calls `reportDone()` when it receives that acknowledgement.

### 4.3 Sentence

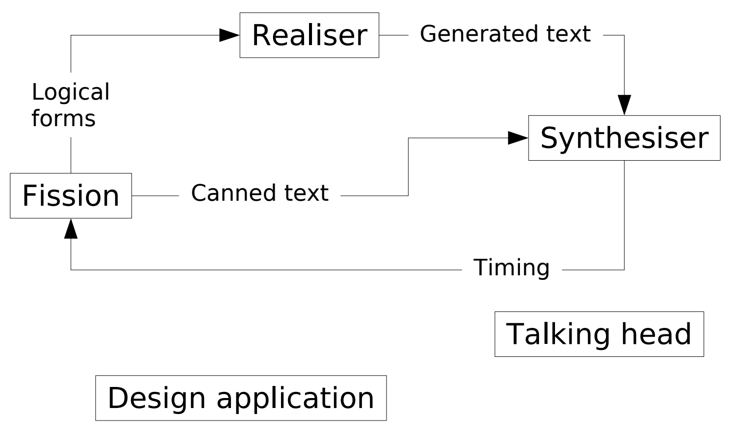
The Sentence class represents a single sentence, combining synthesised speech, lip-synch commands for the talking head, and possible coordinated behaviours on the other multimodal channels. The timing of a sentence is based on the timing of the synthesised speech; all multimodal behaviours are scheduled to coincide with particular words in the text. Unlike a BallisticSegment, which allows the output module to determine the duration at execution time, a Sentence must prepare its schedule in advance to ensure that output is coordinated across all of the channels. In Figure 3, all of the green leaf nodes containing text are instances of Sentence.

There are two types of Sentences: ScriptedSentences and PlannedSentences. A ScriptedSentence is generally created as part of a ScriptedSequence, and is based on pre-written text that is sent directly to the speech synthesiser, along with any necessary multimodal behaviours. A PlannedSentence forms part of a PlannedSequence, and is based on logical forms for the OpenCCG realiser (White, 2005a;b). The logical forms may contain multiple possibilities for both the text and the multimodal behaviours; the OpenCCG realiser uses statistical language models to make a final choice of the actual content of the sentence.

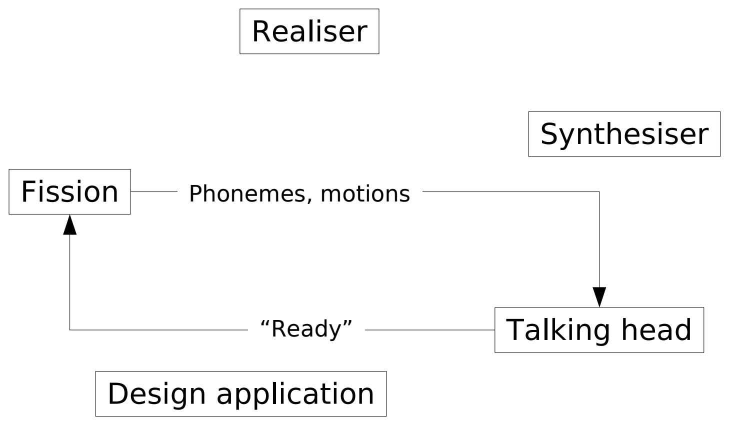
The first step in preparing either type of Sentence is to send the text to the speech synthesiser (Figure 8(a)). For a ScriptedSentence, the canned text is sent directly to the speech synthesiser; for a PlannedSentence, the logical forms are sent to the realiser, which then creates the text and sends it to the synthesiser. In either case, the speech-synthesiser input also includes marks at all points where multimodal output is intended. The speech synthesiser prepares and stores the waveform based on the input text, and returns timing information for the words and phonemes, along with the timing of any multimodal coordination marks.

The fission module uses the returned timing information to create the final schedule for all modalities. It then sends the animation schedule (lip-synch commands, along with any coordinated expression or gaze behaviours) to the talking-head module so that it can prepare its animation in advance (Figure 8(b)). Once the talking-head module has prepared the animation for a turn, it returns a “ready” message. The design application does not need its schedule in advance, so once the response is received from the talking head, the Sentence has finished its preparation and is able to set its `ready` flag.

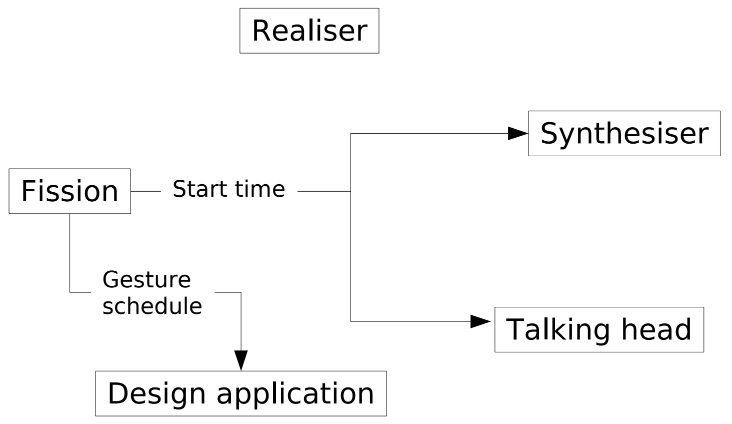
When a Sentence is executed by its parent, it selects a desired start time slightly in the future and sends two messages, as shown in Figure 8(c). First, it sends a “go” message with the selected starting time to the speech-synthesis and talking-head modules; these modules then play the prepared output for that turn at the given time. The Sentence also sends the concrete schedule for any coordinated gesture commands to the bathroom-design application at this point. After sending its messages, the Sentence waits until the scheduled duration has elapsed, and then calls `reportDone()`.



(a) Preparing the speech



(b) Preparing the animation



(c) Producing the output

Figure 8: Planning and executing a Sentence

## 5 Robustness and Configurability

In the preceding section, we gave a description of the data structures and methods that are used when preparing and executing and output plan. In this section, we describe two other aspects of the module that are important to its functioning as part of the overall dialogue system: its ability to detect and deal with errors in its processing, and the various configurations in which it can be run.

### 5.1 Error Detection and Recovery

Since barge-in is not implemented in COMIC, the fission module plays an important role in turn-taking for the whole COMIC system: it is the module that informs the input components when the system output is finished, so that they are able to process the next user input. The fission module therefore incorporates several measures to ensure that it is able to detect and recover from unexpected events during its processing, so that the dialogue is able to continue even if there are errors in some parts of the output.

Most input from external modules is validated against XML schemas to ensure that it is well-formed, and any messages that fail to validate are not processed further. As well, all queries to external modules are sent with configurable time-outs, and any Segment that is expecting a response to a query is also prepared to deal with a time-out.

If a problem occurs while preparing any Segment for output—either due to an error in internal processing, or because of an issue with some external module—that Segment immediately sets its `skip` flag and stops the preparation process. As described in Section 4, any Segments with this flag set are then skipped at execution time. This ensures that processing is able to continue as much as possible despite the errors, and that the fission module is still able to produce output from the parts of an output plan unaffected by the problems and to perform its necessary turn-taking functions.

### 5.2 Configurability

The COMIC fission module can be run in several different configurations, to meet a variety of evaluation, demonstration, and development situations. The fission module can be configured not to wait for "ready" and "done" responses from either or both of the talking-head and design-application modules; the fission module simply proceeds with the rest of its processing as if the required response had been received. This allows the whole COMIC system to be run without those output modules enabled. This is useful during development of other parts of the system, and for running demos and evaluation experiments where not all of the output channels are used. The module also has a number of other configuration options to control factors such as query time-outs and the method of selecting multimodal coarticulations.

As well, the fission module has the ability to generate multiple alternative versions of a single turn, using different user models, dialogue-history settings, or multimodal planning techniques; this is useful both as a testing tool and as part of a system demonstration. The module can also store all of the generated output to a script, and to play back the scripted output at a later time using a subset of the full system. This allows alternative versions of the system output to be directly compared in user evaluation studies such as (Foster, 2004; Foster and White, 2005).

## 6 Output Speed

In the final version of the COMIC system, the average time[2] that the speech synthesiser takes to prepare the waveform for a sentence is 1.9 seconds, while the average synthesised length of a sentence is 2.7 seconds. This means that, on average, each sentence takes long enough to play that the next sentence is ready as soon as it is needed; and even when this is not the case, the delay between sentences is still greatly reduced by the parallel planning process.

The importance of beginning output as soon as possible was demonstrated by a user evaluation of an interim version of COMIC (White et al., 2005). Subjects in that study used the full COMIC system in one of two configurations: an "expressive" condition, where the talking head used all of the expressions it was capable of, or a "zombie" condition where all of the behaviours of the head were disabled except for lip-synch. One effect of this

[2] On a Pentium 4 1.6GHz computer.

difference was that the system gave a consistently earlier response in the expressive condition—a facial response was produced an average of 1.4 seconds after the dialogue-manager message, while spoken input did not begin for nearly 4 seconds. Although that version of the system was very slow, the subjects in the expressive condition were significantly less likely to mention the overall slowness than the subjects in the zombie condition.

After this interim evaluation, effort was put into further reducing the delay in the final system. For example, we now store the waveforms for acknowledgements and other frequently-used texts pre-synthesised in the speech module instead of sending them to Festival, and other internal processing bottlenecks were eliminated. Using the same computers as the interim evaluation, the fission delay for initial output is under 0.5 seconds in the final system.

## 7 Conclusions

The COMIC fission module is able to prepare and control the output of multimodal turns. It prepares and executes its plans in parallel, which allows it to begin producing output as soon as possible and to continue with preparing later parts of the presentation while executing earlier parts. It is able to produce output coordinated and synchronised across multiple modalities, to detect and recover from a variety of errors during its processing, and to be run in a number of different configurations to support testing, demonstrations, and evaluation experiments. The parallel planning process is able to make a significant reduction in the time taken to produce output, which has a perceptible effect on user satisfaction with the overall system.

Some aspects of the fission module are specific to the design of the COMIC dialogue system; for example, the module performs content-selection and sentence-planning tasks that in other systems might be addressed by a dialogue manager or text-generation module. Also, aspects of the communication with the output modules are tailored to the particular modules involved: the fission module makes use of features of the OpenCCG realiser to help choose the content of many of its turns, and the implementation of the design application is obviously COMIC-specific.

However, the general technique of interleaving preparation and execution, using the time while the system is playing earlier parts of a turn to prepare the later parts, is easily applicable to any system that produces temporal output, as long as the same module is responsible for preparing and executing the output. There is nothing COMIC-specific about the design of the Segment class or its immediate sub-classes.

As well, the method of coordinating distributed multimodal behaviour with the speech timing (Section 4.3) is a general one. Although the current implementation relies on the output modules to respect the schedules that they are given—with no adaptation at run time—in practice the coordination in COMIC has been generally successful, providing that three conditions are met. First, the selected starting time must be far enough in the future that it can be received and processed by each module in time. Second, the clocks on all computers involved in running the system must be synchronised precisely. Finally, the processing load on each computer must be low enough that timer events do not get delayed or pre-empted.

Since the COMIC system does not support barge-in, the current fission module always produces the full presentation that is planned, barring processing errors. However, since the module produces its output incrementally, it would be straightforward to extend the processing to allow execution to be interrupted after any Segment, and to know how much of the planned output was actually produced.

## Acknowledgements

Many thanks to Peter Poller, Tilman Becker, and especially Michael White for helpful advice on and discussions about the implementation of the COMIC fission module, and to the anonymous reviewers for their comments on the initial version of this paper. This work was supported by the COMIC project (IST-2001-32311).

## References

M. Breidt, C. Wallraven, D.W. Cunningham, and H.H. Bülthoff. 2003. Facial animation based on 3d scans and motion capture. In Neill Campbell, editor, *SIGGRAPH 03 Sketches & Applications*. ACM Press.

R.A.J. Clark, K. Richmond, and S. King. 2004. Festival 2 – build your own general purpose unit selection speech synthesiser. In *Proceedings, 5th ISCA workshop on speech synthesis*.

B. de Carolis, C. Pelachaud, I. Poggi, and M. Steedman. 2004. APML, a mark-up language for believable behaviour generation. In H. Prendinger, editor, *Life-like Characters, Tools, Affective Functions and Applications*, pages 65–85. Springer.

F. de Rosis, C. Pelachaud, I. Poggi, V. Carofiglio, and B. De Carolis. 2003. From Greta's mind to her face: modelling the dynamics of affective states in a conversational embodied agent. *International Journal of Human-Computer Studies*, 59(1–2):81–118.

D. DeCarlo, M. Stone, C. Revilla, and J.J. Venditti. 2004. Specifying and animating facial signals for discourse in embodied conversational agents. *Computer Animation and Virtual Worlds*, 15(1):27–38.

M.E. Foster. 2004. User evaluation of generated deictic gestures in the T24 demonstrator. Public deliverable 6.5, COMIC project.

M.E. Foster and M. White. 2004. Techniques for text planning with XSLT. In *Proceedings, NLPXML-2004*.

M.E. Foster and M. White. 2005. Assessing the impact of adaptive generation in the COMIC multimodal dialogue system. In *Proceedings, IJCAI 2005 Workshop on Knowledge and Reasoning in Practical Dialogue systems*.

O. Lemon, A. Gruenstein, and S. Peters. 2002. Collaborative activities and multi-tasking in dialogue systems. *Traitement Automatique des Langues (TAL)*, 43(2):131–154.

E Reiter and R Dale. 2000. *Building Natural Language Generation Systems*. Cambridge University Press.

N. Ström and S. Seneff. 2000. Intelligent barge-in in conversational systems. In *Proceedings, ICSLP-2000*, volume 2, pages 652–655.

W. Wahlster, editor. 2005. *SmartKom: Foundations of Multimodal Dialogue Systems*. Springer. In press.

M.A. Walker, S. Whittaker, A. Stent, P. Maloor, J.D. Moore, M. Johnston, and G. Vasireddy. 2002. Speech-plans: Generating evaluative responses in spoken dialogue. In *Proceedings, INLG 2002*.

M. White. 2005a. Designing an extensible API for integrating language modeling and realization. In *Proceedings, ACL 2005 Workshop on Software*.

M. White. 2005b. Efficient realization of coordinate structures in Combinatory Categorial Grammar. *Research on Language and Computation*. To appear.

M. White, M.E. Foster, J. Oberlander, and A. Brown. 2005. Using facial feedback to enhance turn-taking in a multimodal dialogue system. In *Proceedings, HCI International 2005 Thematic Session on Universal Access in Human-Computer Interaction*.

# Designing an Extensible API for Integrating Language Modeling and Realization

**Michael White**
School of Informatics
University of Edinburgh
Edinburgh EH8 9LW, UK
`http://www.iccs.informatics.ed.ac.uk/~mwhite/`

## Abstract

We present an extensible API for integrating language modeling and realization, describing its design and efficient implementation in the OpenCCG surface realizer. With OpenCCG, language models may be used to select realizations with preferred word orders, promote alignment with a conversational partner, avoid repetitive language use, and increase the speed of the best-first anytime search. The API enables a variety of n-gram models to be easily combined and used in conjunction with appropriate edge pruning strategies. The n-gram models may be of any order, operate in reverse ("right-to-left"), and selectively replace certain words with their semantic classes. Factored language models with generalized backoff may also be employed, over words represented as bundles of factors such as form, pitch accent, stem, part of speech, supertag, and semantic class.

## 1 Introduction

The OpenCCG[1] realizer (White and Baldridge, 2003; White, 2004a; White, 2004c) is an open source surface realizer for Steedman's (2000a; 2000b) Combinatory Categorial Grammar (CCG). It is designed to be the first practical, reusable realizer for CCG, and includes implementations of CCG's unique accounts of coordination and information structure–based prosody.

Like other surface realizers, the OpenCCG realizer takes as input a logical form specifying the propositional meaning of a sentence, and returns one or more surface strings that express this meaning according to the lexicon and grammar. A distinguishing feature of OpenCCG is that it implements a hybrid symbolic-statistical chart realization algorithm that combines (1) a theoretically grounded approach to syntax and semantic composition, with (2) the use of integrated language models for making choices among the options left open by the grammar (thereby reducing the need for hand-crafted rules). In contrast, previous chart realizers (Kay, 1996; Shemtov, 1997; Carroll et al., 1999; Moore, 2002) have not included a statistical component, while previous statistical realizers (Knight and Hatzivassiloglou, 1995; Langkilde, 2000; Bangalore and Rambow, 2000; Langkilde-Geary, 2002; Oh and Rudnicky, 2002; Ratnaparkhi, 2002) have employed less general approaches to semantic representation and composition, and have not typically made use of fine-grained logical forms that include specifications of such information structural notions as theme, rheme and focus.

In this paper, we present OpenCCG's extensible API (application programming interface) for integrating language modeling and realization, describing its design and efficient implementation in Java. With OpenCCG, language models may be used to select realizations with preferred word orders (White, 2004c), promote alignment with a conversational partner (Brockmann et al., 2005), and avoid repetitive language use. In addition,

[1]`http://openccg.sourceforge.net`

by integrating language model scoring into the search, it also becomes possible to use more accurate models to improve realization times, when the realizer is run in anytime mode (White, 2004b).

To allow language models to be combined in flexible ways—as well as to enable research on how to best combine language modeling and realization—OpenCCG's design includes interfaces that allow user-defined functions to be used for scoring partial realizations and for pruning low-scoring ones during the search. The design also includes classes for supporting a range of language models and typical ways of combining them. As we shall see, experience to date indicates that the benefits of employing a highly generalized approach to scoring and pruning can be enjoyed with little or no loss of performance.

The rest of this paper is organized as follows. Section 2 gives an overview of the realizer architecture, highlighting the role of the interfaces for plugging in custom scoring and pruning functions, and illustrating how n-gram scoring affects accuracy and speed. Sections 3 and 4 present OpenCCG's classes for defining scoring and pruning functions, respectively, giving examples of their usage. Finally, Section 5 summarizes the design and concludes with a discussion of future work.

## 2 Realizer Overview

The UML class diagram in Figure 1 shows the high-level architecture of the OpenCCG realizer; sample Java code for using the realizer appears in Figure 2. A realizer instance is constructed with a reference to a CCG grammar (which supports both parsing and realization). The grammar's lexicon has methods for looking up lexical items via their surface forms (for parsing), or via the principal predicates or relations in their semantics (for realization). A grammar also has a set of hierarchically organized atomic types, which can serve as the values of features in the syntactic categories, or as ontological sorts for the discourse referents in the logical forms (LFs).

Lexical lookup yields lexical signs. A sign pairs a list of words with a category, which itself pairs a syntactic category with a logical form. Lexical signs are combined into derived signs using the rules in the grammar's rule group. Derived signs maintain a derivation history, and their word lists share structure with the word lists of their input signs.

As mentioned in the introduction, for generality, the realizer makes use of a configurable sign scorer and pruning strategy. A sign scorer implements a function that returns a number between 0 and 1 for an input sign. For example, a standard trigram language model can be used to implement a sign scorer, by returning the probability of a sign's words as its score. A pruning strategy implements a method for determining which edges to prune during the realizer's search. The input to the method is a ranked list of edges for signs that have equivalent categories (but different words); grouping edges in this way ensures that pruning cannot "break" the realizer, i.e. prevent it from finding some grammatical derivation when one exists. By default, an N-best pruning strategy is employed, which keeps the N highest scoring input edges, pruning the rest (where N is determined by the current preference settings).

The realization algorithm is implemented by the `realize` method. As in the chart realizers cited earlier, the algorithm makes use of a chart and an agenda to perform a bottom-up dynamic programming search for signs whose LFs completely cover the elementary predications in the input logical form. See Figure 9 (Section 3.1) for a realization trace; the algorithm's details and a worked example appear in (White, 2004a; White, 2004c). The `realize` method returns the edge for the best realization of the input LF, as determined by the sign scorer. After a realization request, the N-best complete edges—or more generally, all the edges for complete realizations that survived pruning—are also available from the chart.

The search for complete realizations proceeds in one of two modes, anytime and two-stage (packing/unpacking). In the anytime mode, a best-first search is performed with a configurable time limit (which may be a limit on how long to look for a better realization, after the first complete one is found). With this mode, the scores assigned by the sign scorer determine the order of the edges on the agenda, and thus have an impact on realization speed. In the two-stage mode, a packed forest

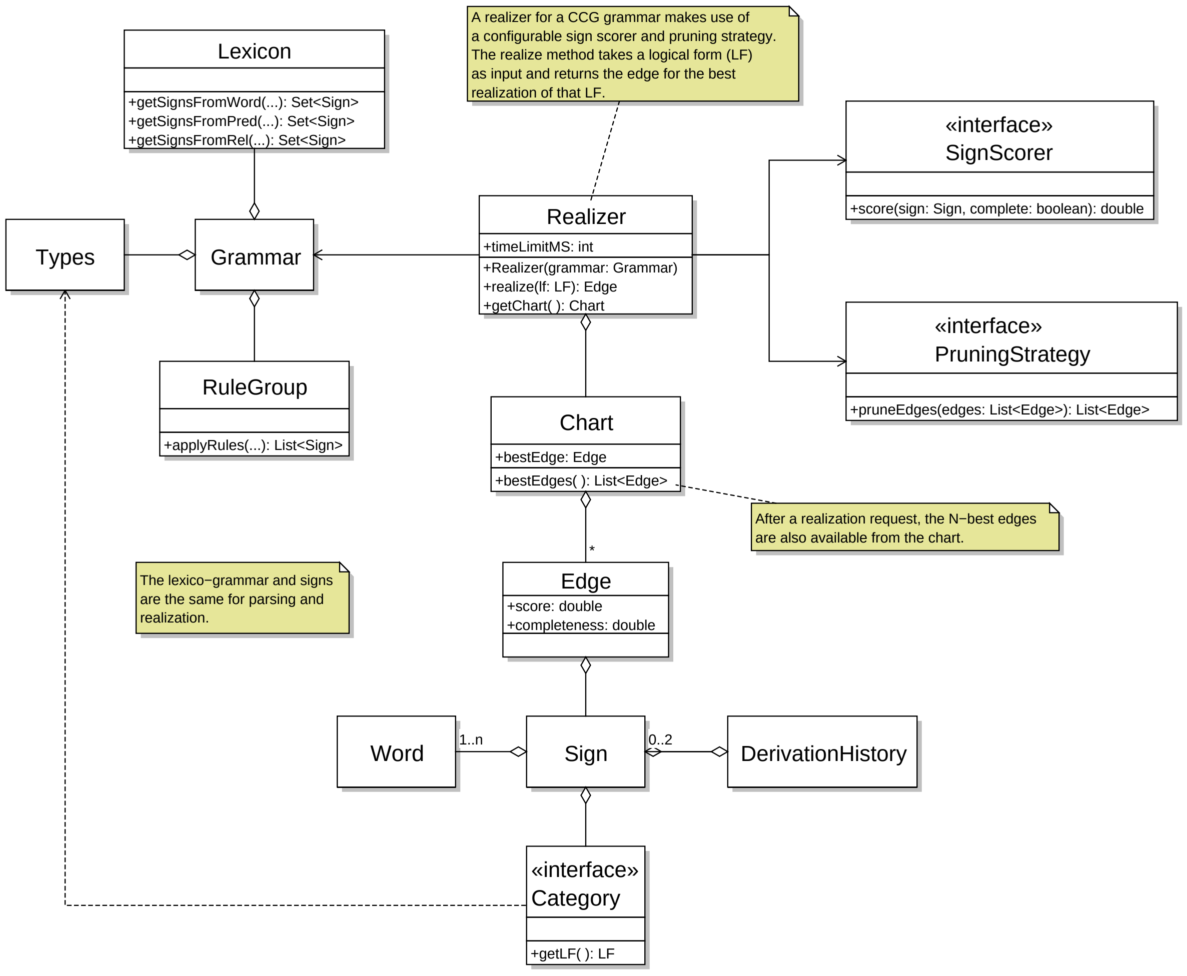

A realizer for a CCG grammar makes use of a configurable sign scorer and pruning strategy. The realize method takes a logical form (LF) as input and returns the edge for the best realization of that LF.
Lexicon
+getSignsFromWord(...): Set<Sign>
+getSignsFromPred(...): Set<Sign>
+getSignsFromRel(...): Set<Sign>
«interface» SignScorer
+score(sign: Sign, complete: boolean): double
Types
Grammar
Realizer
+timeLimitMS: int
+Realizer(grammar: Grammar)
+realize(lf: LF): Edge
+getChart( ): Chart
«interface» PruningStrategy
+pruneEdges(edges: List<Edge>): List<Edge>
RuleGroup
+applyRules(...): List<Sign>
Chart
+bestEdge: Edge
+bestEdges( ): List<Edge>
After a realization request, the N−best edges are also available from the chart.
The lexico−grammar and signs are the same for parsing and realization.
*
Edge
+score: double
+completeness: double
Word
1..n
Sign
0..2
DerivationHistory
«interface» Category
+getLF( ): LF

Figure 1: High-level architecture of the OpenCCG realizer

```
// load grammar, instantiate realizer
URL grammarURL = ...;
Grammar grammar = new Grammar(grammarURL);
Realizer realizer = new Realizer(grammar);

// configure realizer with trigram backoff model
// and 10-best pruning strategy
realizer.signScorer = new StandardNgramModel(3, "lm.3bo");
realizer.pruningStrategy = new NBestPruningStrategy(10);

// ... then, for each request:

// get LF from input XML
Document inputDoc = ...;
LF lf = realizer.getLfFromDoc(inputDoc);

// realize LF and get output words in XML
Edge bestEdge = realizer.realize(lf);
Document outputDoc = bestEdge.sign.getWordsInXml();

// return output
... outputDoc ...;
```

Figure 2: Example realizer usage

of all possible realizations is created in the first stage; then in the second stage, the packed representation is unpacked in bottom-up fashion, with scores assigned to the edge for each sign as it is unpacked, much as in (Langkilde, 2000). In both modes, the pruning strategy is invoked to determine whether to keep or prune newly constructed edges. For single-best output, the anytime mode can provide signficant time savings by cutting off the search early; see (White, 2004c) for discussion. For N-best output—especially when a complete search (up to the edges that survive the pruning strategy) is desirable—the two-stage mode can be more efficient.

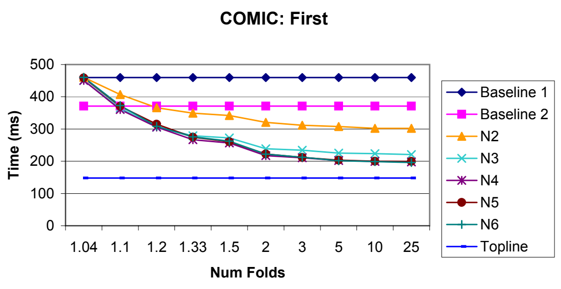
To illustrate how n-gram scoring can guide the best-first anytime search towards preferred realizations and reduce realization times, we reproduce in Table 1 and Figures 3 through 5 the cross-validation tests reported in (White, 2004b). In these tests, we measured the realizer's accuracy and speed, under a variety of configurations, on the regression test suites for two small but linguistically rich grammars: the English grammar for the COMIC[2] dialogue system—the core of which is shared with the FLIGHTS system (Moore et al., 2004)—and the Worldcup grammar discussed in (Baldridge, 2002). Table 1 gives the sizes of the test suites. Using these two test suites, we timed how long it took on a 2.2 GHz Linux PC to realize each logical form under each realizer configuration. To measure accuracy, we counted the number of times the best scoring realization exactly matched the target, and also computed a modified version of the Bleu n-gram precision metric (Papineni et al., 2001) employed in machine translation evaluation, using 1- to 4-grams, with the longer n-grams given more weight (cf. Section 3.4). To rank candidate realizations, we used standard n-gram backoff models of orders 2 through 6, with semantic class replacement, as described in Section 3.1. For smoothing, we used Ristad's natural discounting (Ristad, 1995), a parameter-free method that seems to work well with relatively small amounts of data.

To gauge how the amount of training data affects performance, we ran cross-validation tests with increasing numbers of folds, with 25 as the maximum number of folds. We also compared the realization results using the n-gram scorers with two baselines and one topline (oracle method). The first baseline assigns all strings a uniform score of zero, and adds new edges to the end of the agenda, corresponding to breadth-first search. The

[2] `http://www.hcrc.ed.ac.uk/comic/`

|  | *LF/target pairs* | *Unique up to SC* | *Length* |  |  | *Input nodes* |  |  |
|---|---|---|---|---|---|---|---|---|
|  |  |  | *Mean* | *Min* | *Max* | *Mean* | *Min* | *Max* |
| **COMIC** | 549 | 219 | 13.1 | 6 | 34 | 8.4 | 2 | 20 |
| **Worldcup** | 276 | 138 | 9.2 | 4 | 18 | 6.8 | 3 | 13 |

Table 1: Test suite sizes.



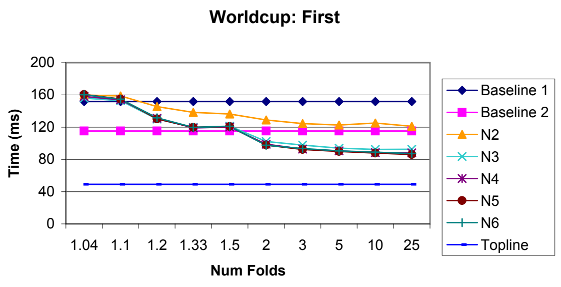
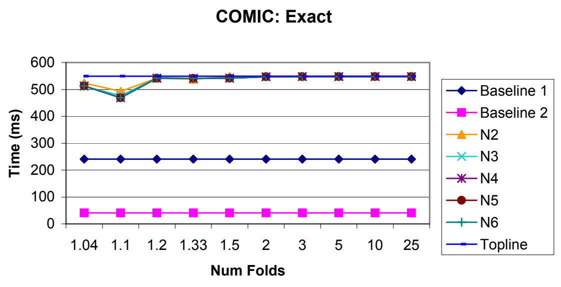


Figure 3: Mean time (in ms.) until first realization is found using n-grams of different orders and Ristad's natural discounting (N), for cross-validation tests with increasing numbers of folds.



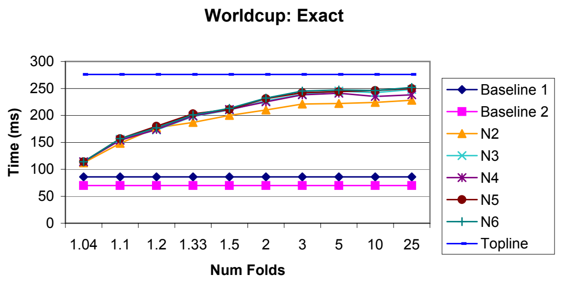
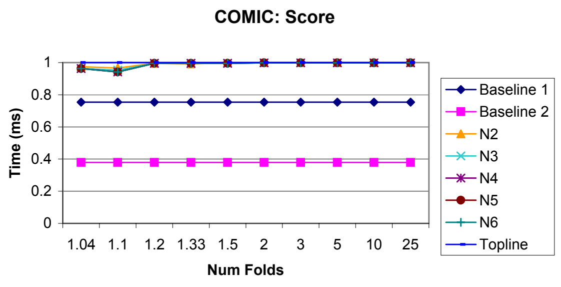


Figure 4: Number of realizations exactly matching target using n-grams of different orders.



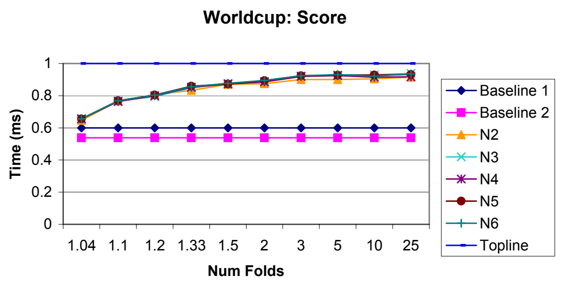


Figure 5: Modified BLEU scores using n-grams of different orders.

second baseline uses the same scorer, but adds new edges at the front of the agenda, corresponding to depth-first search. The topline uses the modified Bleu score, computing n-gram precision against just the target string. With this setup, Figures 3-5 show how initial realization times decrease and accuracy increases when longer n-grams are employed. Figure 3 shows that trigrams offer a substantial speedup over bigrams, while n-grams of orders 4-6 offer a small further improvement. Figures 4 and 5 show that with the COMIC test suite, all n-gram orders work well, while with the Worldcup test suite, n-grams of orders 3-6 offer some improvement over bigrams.

To conclude this section, we note that together with OpenCCG's other efficiency methods, n-gram scoring has helped to achieve realization times adequate for interactive use in both the COMIC and FLIGHTS dialogue systems, along with very high quality. Estimates indicate that n-gram scoring typically accounts for only 2-5% of the time until the best realization is found, while it can more than double realization speed by accurately guiding the best-first anytime search. This experience suggests that more complex scoring models can more than pay for themselves, efficiency-wise, if they yield significantly more accurate preference orders on edges.

## 3 Classes for Scoring Signs

The classes for implementing sign scorers appear in Figure 6. In the diagram, classes for n-gram scoring appear towards the bottom, while classes for combining scorers appear on the left, and the class for avoiding repetition appears on the right.

### 3.1 Standard N-gram Models

The `StandardNgramModel` class can load standard n-gram backoff models for scoring, as shown earlier in Figure 2. Such models can be constructed with the SRILM toolkit (Stolcke, 2002), which we have found to be very useful; in principle, other toolkits could be used instead, as long as their output could be converted into the same file formats. Since the SRILM toolkit has more restrictive licensing conditions than those of OpenCCG's LGPL license, OpenCCG includes its own classes for scoring with n-gram models, in order to avoid any necessary runtime dependencies on the SRILM toolkit.

The n-gram tables are efficiently stored in a trie data structure (as in the SRILM toolkit), thereby avoiding any arbitrary limit on the n-gram order. To save memory and speed up equality tests, each string is interned (replaced with a canonical instance) at load time, which accomplishes the same purpose as replacing the strings with integers, but without the need to maintain a separate mapping from integers back to strings. For better generalization, certain words may be dynamically replaced with the names of their semantic classes when looking up n-gram probabilities. Words are assigned to semantic classes in the lexicon, and the semantic classes to use in this way may be configured at the grammar level. Note that (Oh and Rudnicky, 2002) and (Ratnaparkhi, 2002) make similar use of semantic classes in n-gram scoring, by deferring the instantiation of classes (such as *departure city*) until the end of the generation process; our approach accomplishes the same goal in a slightly more flexible way, in that it also allows the specific word to be examined by other scoring models, if desired.

As discussed in (White, 2004c), with dialogue systems like COMIC n-gram models can do an excellent job of placing underconstrained adjectival and adverbial modifiers—as well as boundary tones—without resorting to the more complex methods investigated for adjective ordering in (Shaw and Hatzivassiloglou, 1999; Malouf, 2000). For instance, in examples like those in (1), they correctly select the preferred positions for *here* and *also* (as well as for the boundary tones), with respect to the verbal head and sister dependents:

(1) a. Here$_{L+H*}$ LH% we have a design in the classic$_{H*}$ style LL% .

b. This$_{L+H*}$ design LH% here$_{L+H*}$ LH% is also$_{H*}$ classic LL% .

We have also found that it can be useful to use reverse (or "right-to-left") models, as they can help to place adverbs like *though*, as in (2):

(2) The tiles are also$_{H*}$ from the Jazz$_{H*}$ series though LL% .

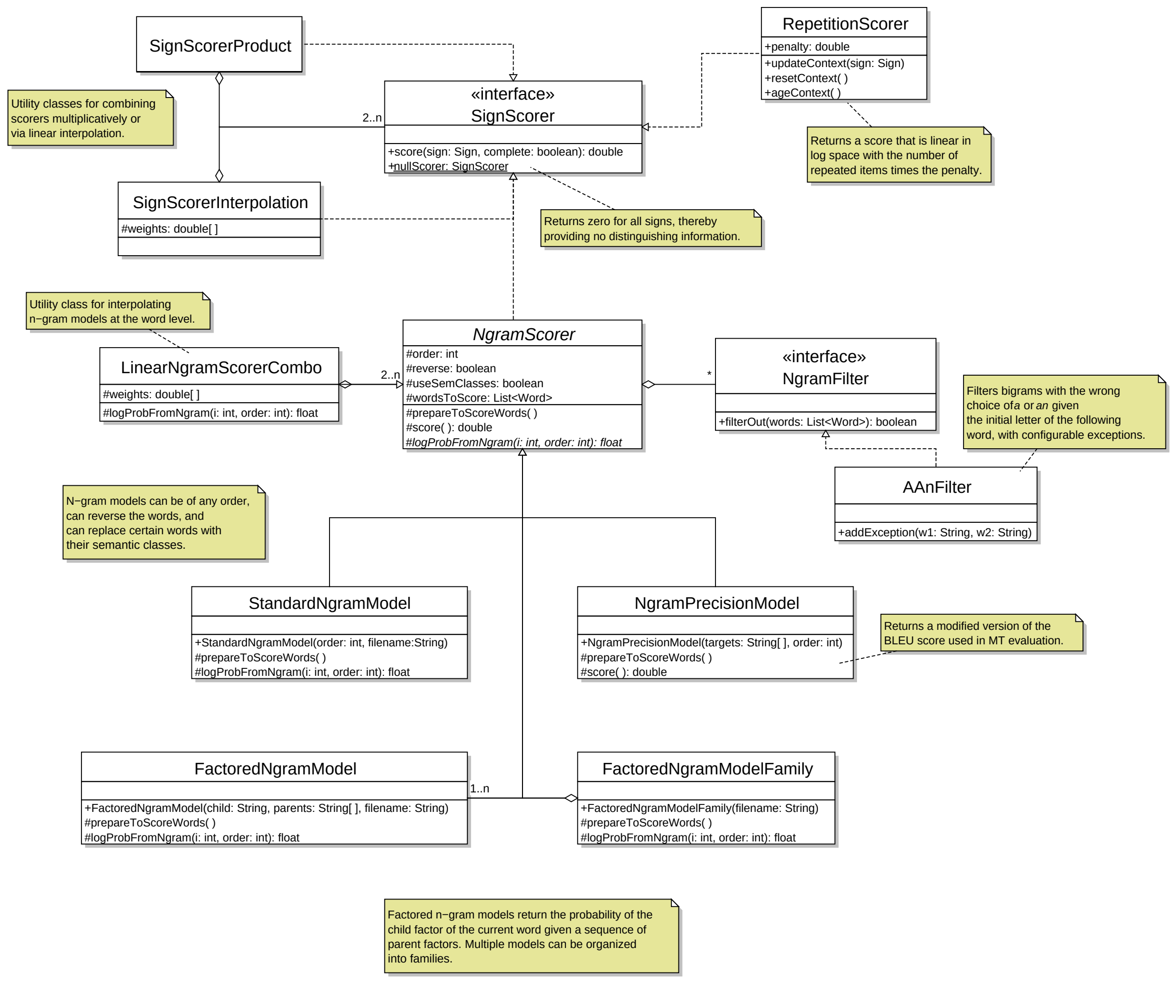

SignScorerProduct
Utility classes for combining scorers multiplicatively or via linear interpolation.
«interface» SignScorer
+score(sign: Sign, complete: boolean): double
+nullScorer: SignScorer
2..n
RepetitionScorer
+penalty: double
+updateContext(sign: Sign)
+resetContext( )
+ageContext( )
Returns a score that is linear in log space with the number of repeated items times the penalty.
SignScorerInterpolation
#weights: double[ ]
Returns zero for all signs, thereby providing no distinguishing information.
Utility class for interpolating n−gram models at the word level.
LinearNgramScorerCombo
#weights: double[ ]
#logProbFromNgram(i: int, order: int): float
2..n
NgramScorer
#order: int
#reverse: boolean
#useSemClasses: boolean
#wordsToScore: List<Word>
#prepareToScoreWords( )
#score( ): double
#logProbFromNgram(i: int, order: int): float
*
«interface» NgramFilter
+filterOut(words: List<Word>): boolean
Filters bigrams with the wrong choice of a or an given the initial letter of the following word, with configurable exceptions.
AAnFilter
+addException(w1: String, w2: String)
N−gram models can be of any order, can reverse the words, and can replace certain words with their semantic classes.
StandardNgramModel
+StandardNgramModel(order: int, filename:String)
#prepareToScoreWords( )
#logProbFromNgram(i: int, order: int): float
NgramPrecisionModel
+NgramPrecisionModel(targets: String[ ], order: int)
#prepareToScoreWords( )
#score( ): double
Returns a modified version of the BLEU score used in MT evaluation.
FactoredNgramModel
+FactoredNgramModel(child: String, parents: String[ ], filename: String)
#prepareToScoreWords( )
#logProbFromNgram(i: int, order: int): float
1..n
FactoredNgramModelFamily
+FactoredNgramModelFamily(filename: String)
#prepareToScoreWords( )
#logProbFromNgram(i: int, order: int): float
Factored n−gram models return the probability of the child factor of the current word given a sequence of parent factors. Multiple models can be organized into families.

Figure 6: Classes for scoring signs

In principle, the forward and reverse probabilities should be the same—as they are both derived via the chain rule from the same joint probability of the words in the sequence—but we have found that with sparse data the estimates can differ substantially. In particular, since *though* typically appears at the end of a variety of clauses, its right context is much more predictable than its left context, and thus reverse models yield more accurate estimates of its likelihood of appearing clause-finally. To illustrate, Figures 7 and 8 show the forward and reverse trigram probabilities for two competing realizations of (2) in a 2-fold cross-validation test (i.e. with models trained on the half of the test suite not including this example). With the forward trigram model, since *though* has not been observed following *series*, and since *series* is a frequently occurring word, the penalty for backing off to the unigram probability for *though* is high, and thus the probability is quite low. The medial placement (following *also*$_{H*}$) also yields a low probability, but not as low as the clause-final one, and thus the forward model ends up preferring the medial placement, which is quite awkward. By contrast, the reverse model yields a very clear preference for the clause-final position of *though*, and for this reason interpolating the forward and reverse models (see Section 3.3) also yields the desired preference order.

Figure 9 shows a trace of realizing (2) with such an interpolated model. In the trace, the interpolated model is loaded by the class `MyEvenScorer`. The input LF appears at the top. It is flattened into a list of elementary predications, so that coverage of these predications can be tracked using bit vectors. The LF chunks ensure that the subtrees under `h1` and `s1` are realized as independent subproblems; cf. (White, 2004a) for discussion. The edges produced by lexical lookup and instantiation appear next, under the heading `Initial Edges`, with only the edges for *also*$_{H*}$ and *though* shown in the figure. For each edge, the coverage percentage and score (here a probability) appear first, followed by the word(s) and the coverage vector, then the syntactic category (with features suppressed), and finally any active LF chunks. The edges added to the chart appear (unsorted) under the heading `All Edges`. As this trace shows, in the best-first search, high probability phrases such as *the tiles are also*$_{H*}$ can be added to the chart before low-frequency words such as *though* have even left the agenda. The first complete realization, corresponding to (2), also turns out to be the best one here. As noted in the figure, complete realizations are scored with sentence delimiters, which—by changing the contexts of the initial and final words—can result in a complete realization having a higher probability than its input partial realizations (see next section for discussion). One way to achieve more monotonic scores—and thus more efficient search, in principle—could be to include sentence delimiters in the grammar; we leave this question for future work.

## 3.2 N-gram Scorers

The `StandardNgramModel` class is implemented as a subclass of the base class `NgramScorer`. All `NgramScorer` instances may have any number of `NgramFilter` instances, whose `filterOut` methods are invoked prior to n-gram scoring; if any of these methods return true, a score of zero is immediately returned. The `AAnFilter` provides one concrete implementation of the `NgramFilter` interface, and returns true if it finds a bigram consisting of *a* followed by a vowel-inital word, or *an* followed by a consonant-initial word, subject to a configurable set of exceptions that can be culled from bigram counts. We have found that such n-gram filters can be more efficient, and more reliable, than relying on n-gram scores alone; in particular, with *a/an*, since the unigram probability for *a* tends to be much higher than that of *an*, with unseen words beginning with a vowel, there may not be a clear preference for the bigram beginning with *an*.

The base class `NgramScorer` implements the bulk of the `score` method, using an abstract `logProbFromNgram` method for subclass-specific calculation of the log probabilities (with backoff) for individual n-grams. The `score` method also invokes the `prepareToScoreWords` method, in order to allow for subclass-specific pre-processing of the words in the given sign. With `StandardNgramModel`, this method is used to extract the word forms or semantic classes into a list of strings to score. It also appends any pitch accents to the

```
the tiles are also_H* from the SERIES_H* series though LL% .
    p( the | <s> )  = [2gram] 0.0999418 [ -1.00025 ]
    p( tiles | the ...)  = [3gram] 0.781102 [ -0.107292 ]
    p( are | tiles ...)  = [3gram] 0.484184 [ -0.31499 ]
    p( also_H* | are ...)  = [3gram] 0.255259 [ -0.593018 ]
    p( from | also_H* ...)  = [3gram] 0.0649038 [ -1.18773 ]
    p( the | from ...)  = [3gram] 0.5 [ -0.30103 ]
    p( SERIES_H* | the ...)  = [3gram] 0.713421 [ -0.146654 ]
    p( series | SERIES_H* ...)  = [3gram] 0.486827 [ -0.312626 ]
    p( though | series ...)  = [1gram] 1.58885e-06 [ -5.79892 ]
    p( LL% | though ...)  = [2gram] 0.416667 [ -0.380211 ]
    p( . | LL% ...)  = [3gram] 0.75 [ -0.124939 ]
    p( </s> | . ...)  = [3gram] 0.999977 [ -1.00831e-05 ]
1 sentences, 11 words, 0 OOVs
0 zeroprobs, logprob= -10.2677 ppl= 7.17198 ppl1= 8.57876

the tiles are also_H* though from the SERIES_H* series LL% .
    p( the | <s> )  = [2gram] 0.0999418 [ -1.00025 ]
    p( tiles | the ...)  = [3gram] 0.781102 [ -0.107292 ]
    p( are | tiles ...)  = [3gram] 0.484184 [ -0.31499 ]
    p( also_H* | are ...)  = [3gram] 0.255259 [ -0.593018 ]
    p( though | also_H* ...)  = [1gram] 1.11549e-05 [ -4.95254 ]
    p( from | though ...)  = [1gram] 0.00805451 [ -2.09396 ]
    p( the | from ...)  = [2gram] 0.509864 [ -0.292545 ]
    p( SERIES_H* | the ...)  = [3gram] 0.713421 [ -0.146654 ]
    p( series | SERIES_H* ...)  = [3gram] 0.486827 [ -0.312626 ]
    p( LL% | series ...)  = [3gram] 0.997543 [ -0.00106838 ]
    p( . | LL% ...)  = [3gram] 0.733867 [ -0.134383 ]
    p( </s> | . ...)  = [3gram] 0.999977 [ -1.00831e-05 ]
1 sentences, 11 words, 0 OOVs
0 zeroprobs, logprob= -9.94934 ppl= 6.74701 ppl1= 8.02574
```

Figure 7: Forward probabilities for two placements of *though* (COMIC test suite, 2-fold cross validation)

```
the tiles are also_H* from the SERIES_H* series though LL% .
    p( . | <s> )  = [2gram] 0.842366 [ -0.0744994 ]
    p( LL% | . ...)  = [3gram] 0.99653 [ -0.00150975 ]
    p( though | LL% ...)  = [3gram] 0.00677446 [ -2.16913 ]
    p( series | though ...)  = [1gram] 0.00410806 [ -2.38636 ]
    p( SERIES_H* | series ...)  = [2gram] 0.733867 [ -0.134383 ]
    p( the | SERIES_H* ...)  = [3gram] 0.744485 [ -0.128144 ]
    p( from | the ...)  = [3gram] 0.765013 [ -0.116331 ]
    p( also_H* | from ...)  = [3gram] 0.0216188 [ -1.66517 ]
    p( are | also_H* ...)  = [3gram] 0.5 [ -0.30103 ]
    p( tiles | are ...)  = [3gram] 0.432079 [ -0.364437 ]
    p( the | tiles ...)  = [3gram] 0.9462 [ -0.0240173 ]
    p( </s> | the ...)  = [3gram] 0.618626 [ -0.208572 ]
1 sentences, 11 words, 0 OOVs
0 zeroprobs, logprob= -7.57358 ppl= 4.27692 ppl1= 4.88098

the tiles are also_H* though from the SERIES_H* series LL% .
    p( . | <s> )  = [2gram] 0.842366 [ -0.0744994 ]
    p( LL% | . ...)  = [3gram] 0.99653 [ -0.00150975 ]
    p( series | LL% ...)  = [3gram] 0.0948425 [ -1.023 ]
    p( SERIES_H* | series ...)  = [3gram] 0.733867 [ -0.134383 ]
    p( the | SERIES_H* ...)  = [3gram] 0.744485 [ -0.128144 ]
    p( from | the ...)  = [3gram] 0.765013 [ -0.116331 ]
    p( though | from ...)  = [1gram] 3.50735e-08 [ -7.45502 ]
    p( also_H* | though ...)  = [1gram] 0.00784775 [ -2.10525 ]
    p( are | also_H* ...)  = [2gram] 0.2291 [ -0.639975 ]
    p( tiles | are ...)  = [3gram] 0.432079 [ -0.364437 ]
    p( the | tiles ...)  = [3gram] 0.9462 [ -0.0240173 ]
    p( </s> | the ...)  = [3gram] 0.618626 [ -0.208572 ]
1 sentences, 11 words, 0 OOVs
0 zeroprobs, logprob= -12.2751 ppl= 10.5421 ppl1= 13.0594
```

Figure 8: Reverse probabilities for two placements of *though* (COMIC test suite, 2-fold cross validation)

```
Input LF:
@b1:state(be ^ <info>rh ^ <mood>dcl ^ <tense>pres ^ <owner>s ^ <kon>- ^
          <Arg>(t1:phys-obj ^ tile ^ <det>the ^ <num>pl ^ <info>rh ^ <owner>s ^ <kon>-) ^
          <Prop>(h1:proposition ^ has-rel ^ <info>rh ^ <owner>s ^ <kon>- ^
                 <Of>t1:phys-obj ^
                 <Source>(s1:abstraction ^ series ^ <det>the ^ <num>sg ^ <info>rh ^ <owner>s ^ <kon>- ^
                          <HasProp>(j1:series ^ Jazz ^ <kon>+ ^ <info>rh ^ <owner>s))) ^
          <HasProp>(a1:proposition ^ also ^ <kon>+ ^ <info>rh ^ <owner>s) ^
          <HasProp>(t2:proposition ^ though ^ <info>rh ^ <owner>s ^ <kon>-))

Instantiating scorer from class: MyEvenScorer

Preds:
ep[0]:  @a1:proposition(also)
ep[1]:  @a1:proposition(<info>rh)
ep[2]:  @a1:proposition(<kon>+)
ep[3]:  @a1:proposition(<owner>s)
ep[4]:  @b1:state(be)
ep[5]:  @b1:state(<info>rh)
...

LF chunks:
chunk[0]:  {14, 15, 16, 17, 18, 19, 20, 21, 22, 23, 24, 25, 26, 27, 28, 29, 30}
chunk[1]:  {20, 21, 22, 23, 24, 25, 26, 27, 28, 29, 30}

Initial Edges:
{0.12} [0.011] also_H* {0, 1, 2, 3, 12} :- s\.s
{0.12} [0.011] also_H* {0, 1, 2, 3, 12} :- s\np/^(s\np)
{0.12} [0.011] also_H* {0, 1, 2, 3, 12} :- s/^s
...
{0.12} [6E-4] though {13, 37, 38, 39, 40} :- s\np/^(s\np)
{0.12} [6E-4] though {13, 37, 38, 39, 40} :- s\.s
{0.12} [6E-4] though {13, 37, 38, 39, 40} :- s/^s
...

Uninstantiated Semantically Null Edges:
{0.00} [0.073] LL% {} :- s$1\*(s$1)
{0.00} [0.011] L {} :- s$1\*(s$1)

All Edges:
{0.02} [0.059] . {7} :- sent\*s
{0.02} [0.059] . {7} :- sent\*(s\np)
{0.02} [0.052] the {25} :- np/^n  < 0 1 >
{0.02} [0.052] the {32} :- np/^n
{0.12} [0.032] tiles {31, 33, 34, 35, 36} :- n
{0.02} [0.018] from {19} :- n\n/<np  < 0 >
{0.15} [0.018] from {14, 15, 16, 17, 18, 19} :- s\!np/<np  < 0 >
{0.17} [0.017] is {4, 5, 6, 8, 9, 10, 11} :- s\np/(s\!np)
...
{0.15} [0.009] the tiles {31, 32, 33, 34, 35, 36} :- np
{0.15} [0.009] the tiles {31, 32, 33, 34, 35, 36} :- s/@i(s\@inp)
{0.15} [0.009] the tiles {31, 32, 33, 34, 35, 36} :- s$1\@i(s$1/@inp)
...
{0.44} [0.001] the tiles are also_H* {0, 1, 2, 3, 4, 5, 6, 8, 9, 10, 11, 12, 31, 32, 33, 34, 35, 36} :- s/(s\!np)
...
{0.12} [6E-4] though {13, 37, 38, 39, 40} :- s\np/^(s\np)
{0.12} [6E-4] though {13, 37, 38, 39, 40} :- s\.s
{0.12} [6E-4] though {13, 37, 38, 39, 40} :- s/^s
...
{0.85} [1.32E-5] the tiles are also_H* from the Jazz_H* series {...} :- s
{0.85} [1.32E-5] the tiles are also_H* from the Jazz_H* series LL% {...} :- s
...
{0.24} [2.64E-6] also_H* though {0, 1, 2, 3, 12, 13, 37, 38, 39, 40} :- s\np/^(s\np)
{0.24} [2.64E-6] also_H* though {0, 1, 2, 3, 12, 13, 37, 38, 39, 40} :- s\.s
...
{0.88} [5.44E-8] the tiles are from the Jazz_H* series though LL% . {...} :- sent
...
{0.85} [4.85E-8] the tiles are from the Jazz_H* series also_H* {...} :- s
...
{0.88} [3.14E-9] the tiles also_H* are from the Jazz_H* series LL% . {...} :- sent
...
{0.98} [2.96E-9] the tiles are also_H* from the Jazz_H* series though {...} :- s
...
{0.98} [1.51E-9] the tiles are also_H* from the Jazz_H* series though LL% {...} :- s
{1.00} [1.34E-8] the tiles are also_H* from the Jazz_H* series though LL% . {...} :- sent

     ****** first complete realization; scored with <s> and </s> tags ******

...
{0.24} [1.44E-9] also_H* though L {...} :- s\np/^(s\np)
...
{0.56} [2E-10] though the tiles are also_H* LL% {...} :- s/(s\!np)
...

Complete Edges (sorted):
{1.00} [1.34E-8] the tiles are also_H* from the Jazz_H* series though LL% . {...} :- sent
{1.00} [1.33E-8] the tiles are also_H* from the Jazz_H* series LL% though LL% . {...} :- sent
...
```

Figure 9: Realizer trace for example (2) with interpolated model

word forms or semantic classes, effectively treating them as integral parts of the words.

Since the realizer builds up partial realizations bottom-up rather than left-to-right, it only adds start of sentence (and end of sentence) tags with complete realizations. As a consequence, the words with less than a full $n-1$ words of history are scored with appropriate sub-models. For example, the first word of a phrase is scored with a unigram sub-model, without imposing backoff penalties.

Another consequence of bottom-up realization is that both the left- and right-contexts may change when forming new signs from a given input sign. Consequently, it is often not possible (even in principle) to use the score of an input sign directly in computing the score of a new result sign. If one could make assumptions about how the score of an input sign has been computed—e.g., by a bigram model—one could determine the score of the result sign from the scores of the input signs together with an adjustment for the word(s) whose context has changed. However, our general approach to sign scoring precludes making such assumptions. Nevertheless, it is still possible to improve the efficiency of n-gram scoring by caching the log probability of a sign's words, and then looking up that log probability when the sign is used as the first input sign in creating a new combined sign—thus retaining the same left context—and only recomputing the log probabilities for the words of any input signs past the first one. (With reverse models, the sign must be the last sign in the combination.) In principle, the derivation history could be consulted further to narrow down the words whose n-gram probabilities must be recomputed to the minimum possible, though `NgramScorer` only implements a single-step lookup at present.[3] Finally, note that a Java `WeakHashMap` is used to implement the cache, in order to avoid an undesirable buildup of entries across realization requests.

## 3.3 Interpolation

Scoring models may be linearly interpolated in two ways. Sign scorers of any variety may be combined using the `SignScorerInterpolation` class. For example, Figure 10 shows how forward and reverse n-gram models may be interpolated.

With n-gram models of the same direction, it is also possible to linearly interpolate models at the word level, using the `LinearNgramScorerCombo` class. Word-level interpolation makes it easier to use cache models created with maximum likelihood estimation, as word-level interpolation with a base model avoids problems with zero probabilities in the cache model. As discussed in (Brockmann et al., 2005), cache models can be used to promote alignment with a conversational partner, by constructing a cache model from the bigrams in the partner's previous turn, and interpolating it with a base model.[4] Figure 11 shows one way to create such an interpolated model.

## 3.4 N-gram Precision Models

The `NgramPrecisionModel` subclass of `NgramScorer` computes a modified version of the Bleu score used in MT evaluation (Papineni et al., 2001). Its constructor takes as input an array of target strings—from which it extracts the n-gram sequences to use in computing the n-gram precision score—and the desired order. Unlike with the Bleu score, rank order centroid weights (rather than the geometric mean) are used to combine scores of different orders, which avoids problems with scoring partial realizations which have no n-gram matches of the target order. For simplicity, the score also does not include the Bleu score's bells and whistles to make cheating on length difficult.

We have found n-gram precision models to be very useful for regression testing the grammar, as an n-gram precision model created just from the target string nearly always leads the realizer to choose that exact string as its preferred realization. Such models can also be useful for evaluating the success of different scoring models in a cross-validation setup, though with high quality output, manual inspection is usually necessary to determine the importance of any differences between

[3]Informal experiments indicate that caching log probabilities in this way can yield an overall reduction in best-first realization times of 2-3% on average.

[4]At present, such cache models must be constructed with a call to the SRILM toolkit; it would not be difficult to add OpenCCG support for constructing them though, since these models do not require smoothing.

```
// configure realizer with 4-gram forward and reverse backoff models,
// interpolated with equal weight
NgramScorer forwardModel = new StandardNgramModel(4, "lm.4bo");
NgramScorer reverseModel = new StandardNgramModel(4, "lm-r.4bo");
reverseModel.setReverse(true);
realizer.signScorer = new SignScorerInterpolation(
    new SignScorer[] { forwardModel, reverseModel }
);
```

Figure 10: Example interpolated n-gram model

```
// configure realizer with 4-gram backoff base model,
// interpolated at the word level with a bigram maximum-likelihood
// cache model, with more weight given to the base model
NgramScorer baseModel = new StandardNgramModel(4, "lm.4bo");
NgramScorer cacheModel = new StandardNgramModel(2, "lm-cache.mle");
realizer.signScorer = new LinearNgramScorerCombo(
    new SignScorer[] { baseModel, cacheModel },
    new double[] { 0.6, 0.4 }
);
```

Figure 11: Example word-level interpolation of a cache model

the preferred realization and the target string.

### 3.5 Factored Language Models

A factored language model (Bilmes and Kirchhoff, 2003) is a new kind of language model that treats words as bundles of factors. To support scoring with such models, OpenCCG represents words as objects with a surface form, pitch accent, stem, part of speech, supertag, and semantic class. Words may also have any number of further attributes, such as associated gesture classes, in order to handle in a general way elements like pitch accents that are "coarticulated" with words.

To represent words efficiently, and to speed up equality tests, all attribute values are interned, and the `Word` objects themselves are interned via a factory method. Note that in Java, it is straightforward to intern objects other than strings by employing a `WeakHashMap` to map from an object key to a weak reference to itself as the canonical instance. (Using a weak reference avoids accumulating interned objects that would otherwise be garbage collected.)

With the SRILM toolkit, factored language models can be constructed that support *generalized parallel backoff*: that is, backoff order is not restricted to just dropping the most temporally distant word first, but rather may be specified as a path through the set of contextual parent variables; additionally, parallel backoff paths may be specified, with the possibility of combining these paths dynamically in various ways. In OpenCCG, the `FactoredNgramModel` class supports scoring with factored language models that employ generalized backoff, though parallel backoff is not yet supported, as it remains somewhat unclear whether the added complexity of parallel backoff is worth the implementation effort. Typically, several related factored language models are specified in a single file and loaded by a `FactoredNgramModelFamily`, which can multiplicatively score models for different child variables, and include different sub-models for the same child variable.

To illustrate, let us consider a simplified version of the factored language model family used in the COMIC realizer. This model computes the probability of the current word given the preceding ones according to the formula shown in (3), where a word consists of the factors word (W), pitch accent (A), gesture class (GC), and gesture instance (GI), plus the other standard factors which the model ignores:

$$(3)\quad \begin{aligned} &P(\langle W,A,GC,GI\rangle \,|\, \langle W,A,GC,GI\rangle_{-1} \ldots) \approx \\ &\quad P(W \,|\, W_{-1}W_{-2}A_{-1}A_{-2}) \times \\ &\quad P(GC \,|\, W) \times \\ &\quad P(GI \,|\, GC) \end{aligned}$$

In (3), the probability of the current word is approximated by the probability of the current word form given the preceding two word forms and preceding two pitch accents, multiplied by the probability of the current gesture class given the current word form, and by the probability of the current gesture instance given the current gesture class. Note that in the COMIC grammar, the choice of pitch accent is entirely rule governed, so the current pitch accent is not scored separately in the model. However, the preceding pitch accents are taken into account in predicting the current word form, as perplexity experiments have suggested that they do provide additional information beyond that provided by the previous word forms.

The specification file for this model appears in Figure 12. The format of the file is a restricted form of the files used by the SRILM toolkit to build factored language models. The file specifies four models, where the first, third and fourth models correspond to those in (3). With the first model, since the previous words are typically more informative than the previous pitch accents, the backoff order specifies that the most distant accent, `A(-2)`, should be dropped first, followed by the previous accent, `A(-1)`, then the most distant word, `W(-2)`, and finally the previous word, `W(-1)`. The second model is considered a sub-model of the first—since it likewise predicts the current word—to be used when there is only one word of context available (i.e. with bigrams). Note that when scoring a bigram, the second model will take the previous pitch accent into account, whereas the first model would not. For documentation of the file format as it is used in the SRILM toolkit, see (Kirchhoff et al., 2002).

Like `StandardNgramModel`, the `FactoredNgramModel` class stores its n-gram tables in a trie data structure, except that it stores an interned factor key (i.e. a factor name and value pair, or just a string, in the case of the word form) at each node, rather than a simple string. During scoring, the `logProbFromNgram` method determines the log probability (with backoff) of a given n-gram by extracting the appropriate sequence of factor keys, and using them to compute the log probability as with standard n-gram models. The `FactoredNgramModelFamily` class computes log probabilities by delegating to its component factored n-gram models (choosing appropriate sub-models, when appropriate) and summing the results.

### 3.6 Avoiding Repetition

While cache models appear to be a promising avenue to promote lexical and syntactic alignment with a conversational partner, a different mechanism appears to be called for to avoid "self-alignment"—that is, to avoid the repetitive use of words and phrases. As a means to experiment with avoiding repetition, OpenCCG includes the `RepetitionScorer` class. This class makes use of a configurable penalty plus a set of methods for dynamically managing the context. It returns a score of $10^{-c_r \times p}$, where $c_r$ is the count of repeated items, and $p$ is the penalty. Note that this formula returns 1 if there are no repeated items, and returns a score that is linear in log space with the number of repeated items otherwise.

A repetition scorer can be combined multiplicatively with an n-gram model, in order to discount realizations that repeat items from the recent context. Figure 13 shows such a combination, together with the operations for updating the context. By default, open class stems are the considered the relevant items over which to count repetitions, though this behavior can be specialized by subclassing `RepetitionScorer` and overriding the `updateItems` method. Note that in counting repetitions, full counts are given to items in the previous words or recent context, while fractional counts are given to older items; the exact details may likewise be changed in a subclass, by overriding the `repeatedItems` method.

## 4 Pruning Strategies

The classes for defining edge pruning strategies appear in Figure 14. As mentioned in Section 2, an N-best pruning strategy is employed by default, where N is determined by the current preference settings. It is also possible to define custom strategies. To support the definition of a certain kind

```
## Simplified COMIC realizer FLM spec file

## Trigram Word model based on previous words and accents, dropping accents first,
##   with bigram sub-model;
## Unigram Gesture Class model based on current word; and
## Unigram Gesture Instance model based on current gesture class

4

## 3gram with A
W : 4 W(-1) W(-2) A(-1) A(-2) w_w1w2a1a2.count w_w1w2a1a2.lm 5
  W1,W2,A1,A2  A2 ndiscount gtmin 1
  W1,W2,A1  A1 ndiscount gtmin 1
  W1,W2  W2 ndiscount gtmin 1
  W1  W1 ndiscount gtmin 1
  0   0  ndiscount gtmin 1

## bigram with A
W : 2 W(-1) A(-1) w_w1a1.count w_w1a1.lm 3
  W1,A1  A1  ndiscount gtmin 1
  W1  W1  ndiscount gtmin 1
  0   0   ndiscount gtmin 1

## Gesture class depends on current word
GC : 1 W(0) gc_w0.count gc_w0.lm 2
  W0  W0 ndiscount gtmin 1
  0   0  ndiscount gtmin 1

## Gesture instance depends only on class
GI : 1 GC(0) gi_gc0.count gi_gc0.lm 2
  GC0  GC0 ndiscount gtmin 1
  0 0
```

Figure 12: Example factored language model family specification

```
// set up n-gram scorer and repetition scorer
String lmfile = "ngrams/combined.flm";
boolean semClasses = true;
NgramScorer ngramScorer = new FactoredNgramModelFamily(lmfile, semClasses);
ngramScorer.addFilter(new AAnFilter());
RepetitionScorer repetitionScorer = new RepetitionScorer();

// combine n-gram scorer with repetition scorer
realizer.signScorer = new SignScorerProduct(
    new SignScorer[] { ngramScorer, repetitionScorer }
);

// ... then, after each realization request,
Edge bestEdge = realizer.realize(lf);

// ... update repetition context for next realization:
repetitionScorer.ageContext();
repetitionScorer.updateContext(bestEdge.getSign());
```

Figure 13: Example combination of an n-gram scorer and a repetition scorer

of custom strategy, the abstract class `Diversity-PruningStrategy` provides an N-best pruning strategy that promotes diversity in the edges that are kept, according to the equivalence relation established by the abstract `notCompellingly-Different` method. In particular, in order to determine which edges to keep, a diversity pruning strategy clusters the edges into a ranked list of equivalence classes, which are sequentially sampled until the limit N is reached. If the `single-BestPerGroup` flag is set, then a maximum of one edge per equivalence class is retained.

As an example, the COMIC realizer's diversity pruning strategy appears in Figure 15. The idea behind this strategy is to avoid having the N-best lists become full of signs whose words differ only in the exact gesture instance associated with one or more of the words. With this strategy, if two signs differ in just this way, the edge for the lower-scoring sign will be considered "not compellingly different" and pruned from the N-best list, making way for other edges whose signs exhibit more interesting differences.

OpenCCG also provides a concrete subclass of `DiversityPruningStrategy` named `Ngram-DiversityPruningStrategy`, which generalizes the approach to pruning described in (Langkilde, 2000). With this class, two signs are considered not compellingly different if they share the same $n-1$ initial and final words, where $n$ is the n-gram order. When one is interested in single-best output, an n-gram diversity pruning strategy can increase efficiency while guaranteeing no loss in quality—as long as the reduction in the search space outweighs the extra time necessary to check for the same initial and final words—since any words in between an input sign's $n-1$ initial and final ones cannot affect the n-gram score of a new sign formed from the input sign. However, when N-best outputs are desired, or when repetition scoring is employed, it is less clear whether it makes sense to use an n-gram diversity pruning strategy; for this reason, a simple N-best strategy remains the default option.

## 5 Conclusions and Future Work

In this paper, we have presented OpenCCG's extensible API for efficiently integrating language modeling and realization, in order to select realizations with preferred word orders, promote alignment with a conversational partner, avoid repetitive language use, and increase the speed of the best-first anytime search. As we have shown, the design enables a variety of n-gram models to be easily combined and used in conjunction with appropriate edge pruning strategies. The n-gram models may be of any order, operate in reverse ("right-to-left"), and selectively replace certain words with their semantic classes. Factored language models with generalized backoff may also be employed, over words represented as bundles of factors such as form, pitch accent, stem, part of speech, supertag, and semantic class.

In future work, we plan to further explore how to best employ factored language models; in particular, inspired by (Bangalore and Rambow, 2000), we plan to examine whether factored language models using supertags can provide an effective way to combine syntactic and lexical probabilities. We also plan to implement the capability to use `one-of` alternations in the input logical forms (Foster and White, 2004), in order to more efficiently defer lexical choice decisions to the language models.

## Acknowledgements

Thanks to Jason Baldridge, Carsten Brockmann, Mary Ellen Foster, Philipp Koehn, Geert-Jan Kruijff, Johanna Moore, Jon Oberlander, Miles Osborne, Mark Steedman, Sebastian Varges and the anonymous reviewers for helpful discussion.

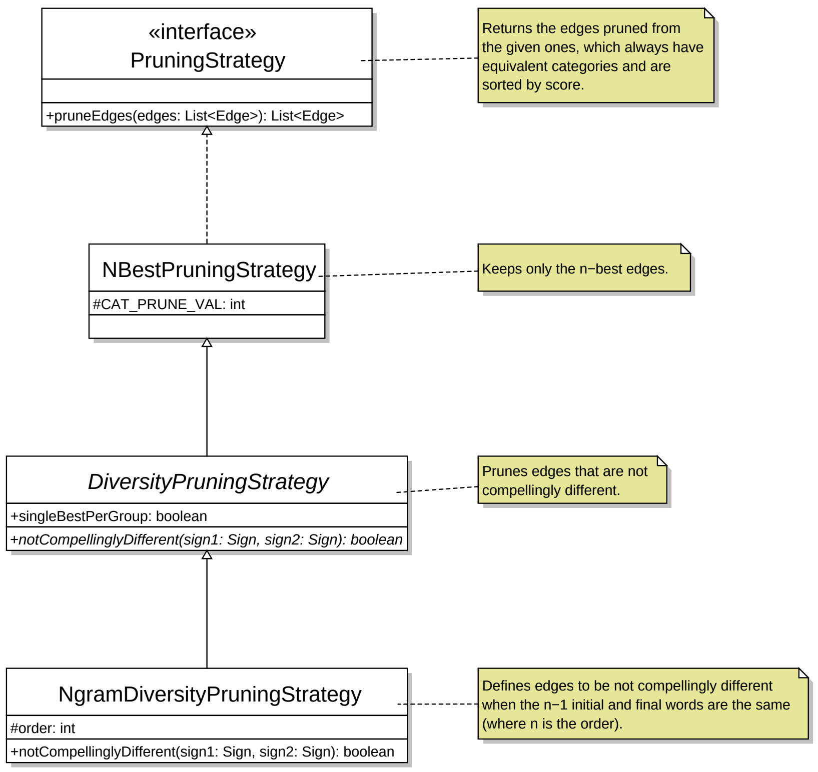

«interface»
PruningStrategy
+pruneEdges(edges: List<Edge>): List<Edge>
Returns the edges pruned from the given ones, which always have equivalent categories and are sorted by score.
NBestPruningStrategy
#CAT_PRUNE_VAL: int
Keeps only the n−best edges.
DiversityPruningStrategy
+singleBestPerGroup: boolean
+notCompellinglyDifferent(sign1: Sign, sign2: Sign): boolean
Prunes edges that are not compellingly different.
NgramDiversityPruningStrategy
#order: int
+notCompellinglyDifferent(sign1: Sign, sign2: Sign): boolean
Defines edges to be not compellingly different when the n−1 initial and final words are the same (where n is the order).

Figure 14: Classes for defining pruning strategies

```
// configure realizer with gesture diversity pruner
realizer.pruningStrategy = new DiversityPruningStrategy() {
    /**
     * Returns true iff the given signs are not compellingly different;
     * in particular, returns true iff the words differ only in their
     * gesture instances. */
    public boolean notCompellinglyDifferent(Sign sign1, Sign sign2) {
        List words1 = sign1.getWords(); List words2 = sign2.getWords();
        if (words1.size() != words2.size()) return false;
        for (int i = 0; i < words1.size(); i++) {
            Word w1 = (Word) words1.get(i); Word w2 = (Word) words2.get(i);
            if (w1 == w2) continue;
            if (w1.getForm() != w2.getForm()) return false;
            if (w1.getPitchAccent() != w2.getPitchAccent()) return false;
            if (w1.getVal("GC") != w2.getVal("GC")) return false;
            // nb: assuming that they differ in the val of GI at this point
        }
        return true;
    }
};
```

Figure 15: Example diversity pruning strategy

## References

Jason Baldridge. 2002. *Lexically Specified Derivational Control in Combinatory Categorial Grammar*. Ph.D. thesis, School of Informatics, University of Edinburgh.

Srinivas Bangalore and Owen Rambow. 2000. Exploiting a probabilistic hierarchical model for generation. In *Proc. COLING-00*.

Jeff Bilmes and Katrin Kirchhoff. 2003. Factored language models and general parallelized backoff. In *Proc. HLT-03*.

Carsten Brockmann, Amy Isard, Jon Oberlander, and Michael White. 2005. Variable alignment in affective dialogue. In *Proc. UM-05 Workshop on Affective Dialogue Systems*. To appear.

John Carroll, Ann Copestake, Dan Flickinger, and Victor Poznański. 1999. An efficient chart generator for (semi-) lexicalist grammars. In *Proc. EWNLG-99*.

Mary Ellen Foster and Michael White. 2004. Techniques for Text Planning with XSLT. In *Proc. 4th NLPXML Workshop*.

Martin Kay. 1996. Chart generation. In *Proc. ACL-96*.

Katrin Kirchhoff, Jeff Bilmes, Sourin Das, Nicolae Duta, Melissa Egan, Gang Ji, Feng He, John Henderson, Daben Liu, Mohamed Noamany, Pat Schone, Richard Schwartz, and Dimitra Vergyri. 2002. Novel Approaches to Arabic Speech Recognition: Report from the 2002 Johns-Hopkins Summer Workshop.

Kevin Knight and Vasileios Hatzivassiloglou. 1995. Two-level, many-paths generation. In *Proc. ACL-95*.

Irene Langkilde-Geary. 2002. An empirical verification of coverage and correctness for a general-purpose sentence generator. In *Proc. INLG-02*.

Irene Langkilde. 2000. Forest-based statistical sentence generation. In *Proc. NAACL-00*.

Robert Malouf. 2000. The order of prenominal adjectives in natural language generation. In *Proc. ACL-00*.

Johanna Moore, Mary Ellen Foster, Oliver Lemon, and Michael White. 2004. Generating tailored, comparative descriptions in spoken dialogue. In *Proc. FLAIRS-04*.

Robert C. Moore. 2002. A complete, efficient sentence-realization algorithm for unification grammar. In *Proc. INLG-02*.

Alice H. Oh and Alexander I. Rudnicky. 2002. Stochastic natural language generation for spoken dialog systems. *Computer, Speech & Language*, 16(3/4):387–407.

Kishore Papineni, Salim Roukos, Todd Ward, and Wei-Jing Zhu. 2001. Bleu: a Method for Automatic Evaluation of Machine Translation. Technical Report RC22176, IBM.

Adwait Ratnaparkhi. 2002. Trainable approaches to surface natural language generation and their application to conversational dialog systems. *Computer, Speech & Language*, 16(3/4):435–455.

Eric S. Ristad. 1995. A Natural Law of Succession. Technical Report CS-TR-495-95, Princeton Univ.

James Shaw and Vasileios Hatzivassiloglou. 1999. Ordering among premodifiers. In *Proc. ACL-99*.

Hadar Shemtov. 1997. *Ambiguity Management in Natural Language Generation*. Ph.D. thesis, Stanford University.

Mark Steedman. 2000a. Information structure and the syntax-phonology interface. *Linguistic Inquiry*, 31(4):649–689.

Mark Steedman. 2000b. *The Syntactic Process*. MIT Press.

Andreas Stolcke. 2002. SRILM — An extensible language modeling toolkit. In *Proc. ICSLP-02*.

Michael White and Jason Baldridge. 2003. Adapting Chart Realization to CCG. In *Proc. EWNLG-03*.

Michael White. 2004a. Efficient Realization of Coordinate Structures in Combinatory Categorial Grammar. *Research on Language and Computation*. To appear.

Michael White. 2004b. Experiments with multimodal output in human-machine interaction. IST Project COMIC Public Deliverable 7.4.

Michael White. 2004c. Reining in CCG Chart Realization. In *Proc. INLG-04*.

# The evolution of dominance constraint solvers

**Alexander Koller** and **Stefan Thater**
Dept. of Computational Linguistics
Saarland University, Saarbrücken, Germany
{koller,stth}@coli.uni-sb.de

## Abstract

We describe the evolution of solvers for dominance constraints, a formalism used in underspecified semantics, and present a new graph-based solver using charts. An evaluation on real-world data shows that each solver (including the new one) is significantly faster than its predecessors. We believe that our strategy of successively tailoring a powerful formalism to the actual inputs is more generally applicable.

## 1 Introduction

In many areas of computational linguistics, there is a tension between a need for powerful formalisms and the desire for efficient processing. Expressive formalisms are useful because they allow us to specify linguistic facts at the right level of abstraction, and in a way that supports the creation and maintenance of large language resources. On the other hand, by choosing a more powerful formalism, we typically run the risk that our processing tasks (say, parsing or inference) can no longer be performed efficiently.

One way to address this tension is to switch to simpler formalisms. This makes processing more efficient, but sacrifices the benefits of expressive formalisms in terms of modelling. Another common strategy is to simply use the powerful formalisms anyway. This sometimes works pretty well in practice, but a system built in this way cannot give any runtime guarantees, and may become slow for certain inputs unpredictably.

In this paper, we advocate a third option: Use a general, powerful formalism, analyse what makes it complex and what inputs actually occur in practice, and then find a restricted fragment of the formalism that supports all practical inputs and can be processed efficiently. We demonstrate this approach by describing the evolution of solvers for *dominance constraints* (Egg et al., 2001), a certain formalism used for the underspecified description of scope ambiguities in computational semantics. General dominance constraints have an NP-complete satisfiability problem, but *normal* dominance constraints, which subsume all constraints that are used in practice, have linear-time satisfiability and can be solved extremely efficiently.

We describe a sequence of four solvers, ranging from a purely logic-based saturation algorithm (Koller et al., 1998) over a solver based on constraint programming (Duchier and Niehren, 2000) to efficient solvers based on graph algorithms (Bodirsky et al., 2004). The first three solvers have been described in the literature before, but we also present a new variant of the graph solver that uses caching to obtain a considerable speedup. Finally we present a new evaluation that compares all four solvers with each other and with a different underspecification solver from the LKB grammar development system (Copestake and Flickinger, 2000).

The paper is structured as follows. We will first sketch the problem that our algorithms solve (Section 2). Then we present the solvers (Section 3) and conclude with the evaluation (Section 4).

## 2 The Problem

The problem we use to illustrate the progress towards efficient solvers is that of enumerating all

*Proceedings of the ACL 2005 Workshop on Software*, pages 65–76,
Ann Arbor, June 2005. © 2005 Association for Computational Linguistics

readings of an *underspecified description*. Underspecification is a technique for dealing with the combinatorial problems associated with quantifier scope ambiguities, certain semantic ambiguities that occur in sentences such as the following:

(1) Every student reads a book.

This sentence has two different readings. Reading (2) expresses that each student reads a possibly different book, while reading (3) claims that there is a single book which is read by every student.

(2) $\forall x.\mathsf{student}(x) \to (\exists y.\mathsf{book}(y) \wedge \mathsf{read}(x,y))$

(3) $\exists y.\mathsf{book}(y) \wedge (\forall x.\mathsf{student}(x) \to \mathsf{read}(x,y))$

The number of readings can grow exponentially in the number of quantifiers and other scope-bearing operators occuring in the sentence. A particularly extreme example is the following sentence from the Rondane Treebank, which the English Resource Grammar (Copestake and Flickinger, 2000) claims to have about 2.4 trillion readings.

(4) Myrdal is the mountain terminus of the Flåm rail line (or Flåmsbana) which makes its way down the lovely Flåm Valley (Flåmsdalen) to its sea-level terminus at Flåm.
(Rondane 650)

Of course, this huge number of readings results not only from genuine meaning differences, but from the (quite reasonable) decision of the ERG developers to uniformly treat all noun phrases, including proper names and definites, as quantifiers. But a system that builds upon such a grammar still has to deal with these readings in some way.

The key idea of underspecification is now to not enumerate all these semantic readings from a syntactic analysis during or after parsing, but to derive from the syntactic analysis a single, compact *underspecified description*. The individual readings can be *enumerated* from the description if they are needed, and this enumeration process should be efficient; but it is also possible to eliminate readings that are infelicitous given knowledge about the world or the context on the level of underspecified descriptions.

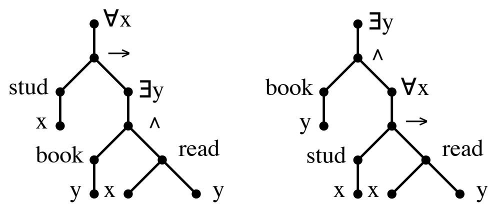
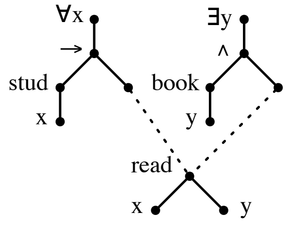


Figure 1: Trees for the readings (2) and (3).



$X_1 : \forall x(X_2) \wedge$
$X_2 : \to(X_3, X_4) \wedge$
$X_5 : \mathit{stud}(X_6) \wedge$
$X_6 : x \wedge$
$\ldots$
$X_4 \lhd^* X_7 \wedge$
$X_7 : \mathit{read}(X_8, X_9) \wedge$
$X_8 : x \wedge X_9 : y$

Figure 2: A dominance constraint (right) and its graphical representation (left); the solutions of the constraint are the two trees in Fig. 1.

**Dominance constraints.** The particular underspecification formalism whose enumeration problem we consider in this paper is the formalism of *dominance constraints* (Egg et al., 2001). The basic idea behind using dominance constraints in underspecification is that the semantic representations (2) and (3) can be considered as *trees* (see Fig. 1). Then a set of semantic representations can be characterised as the set of models of a formula in the following language:

$$\varphi ::= X{:}f(X_1,\ldots,X_n) \mid X \lhd^* Y \mid X \neq Y \mid \varphi \wedge \varphi$$

The *labelling atom* $X{:}f(X_1,\ldots,X_n)$ expresses that the node in the tree which is denoted by the variable $X$ has the label $f$, and its children are denoted by the variables $X_1$ to $X_n$. *Dominance atoms* $X \lhd^* Y$ say that there is a path (of length 0 or more) from the node denoted by $X$ to the node denoted by $Y$; and *inequality atoms* $X \neq Y$ require that $X$ and $Y$ denote different nodes.

Dominance constraints $\varphi$ can be drawn informally as graphs, as shown in Fig. 2. Each node of the graph stands for a variable; node labels and solid edges stand for labelling atoms; and the dotted edges represent dominance atoms. The constraint represented by the drawing in Fig. 2 is *satisfied* by both trees shown in Fig. 1. Thus we can use it as an underspecified description representing these two readings.

The two obvious processing problems connected to dominance constraints are *satisfiability* (is there a model that satisfies the constraint?) and *enumeration* (compute all models of a constraint). Because every satisfiable dominance constraint technically has an infinite number of models, the algorithms below solve the enumeration problem by computing *solved forms* of the constraint, which are finite characterisations of infinite model sets.

## 3 The Solvers

We present four different solvers for dominance constraints. As we go along, we analyse what makes dominance constraint solving hard, and what characterises the constraints that occur in practice.

### 3.1 A saturation algorithm

The first dominance constraint solver (Koller et al., 1998; Duchier and Niehren, 2000) is an algorithm that operates directly on the constraint as a logical formula. It is a *saturation algorithm*, which successively enriches the constraint using *saturation rules*. The algorithm terminates if it either derives a contradiction (marked by the special atom **false**), or if no rule can contribute any new atoms. In the first case, it claims that the constraint is unsatisfiable; in the second case, it reports the end result of the computation as a *solved form* and claims that it is satisfiable.

The saturation rules in the solver try to match their preconditions to the constraint, and if they do match, add their conclusions to the constraint. For example, the following rules express that dominance is a transitive relation, and that trees have no cycles:

$$\begin{array}{lcl} X \triangleleft^* Y \wedge Y \triangleleft^* Z & \rightarrow & X \triangleleft^* Z \\ X{:}f(\ldots,Y,\ldots) \wedge Y \triangleleft^* X & \rightarrow & \textbf{false} \end{array}$$

Some rules have disjunctive right-hand sides; if they are applicable, they perform a case distinction and add one of the disjuncts. One example is the *Choice Rule*, which looks as follows:

$$X \triangleleft^* Z \wedge Y \triangleleft^* Z \;\rightarrow\; X \triangleleft^* Y \vee Y \triangleleft^* X$$

This rule checks for the presence of two variables $X$ and $Y$ that are known to both dominate the same variable $Z$. Because models must be trees, this means that $X$ and $Y$ must dominate each other in some order; but we can't know yet whether it is $X$ or $Y$ that dominates the other one. Hence the solver tries both choices. This makes it possible to derive multiple solved forms (one for each reading of the sentence), such as the two different trees in Fig. 1.

It can be shown that a dominance constraint is satisfiable iff it is not possible to derive **false** from it using the rules in the algorithm. In addition, every model of the original constraint satisfies exactly one solved form. So the saturation algorithm can indeed be used to solve dominance constraints. However, even checking satisfiability takes nondeterministic polynomial time. Because all choices in the distribution rule applications have to be checked, a deterministic program will take exponential time to check satisfiability in the worst case.

Indeed, satisfiability of dominance constraints is an NP-complete problem (Koller et al., 1998), and hence it is likely that any solver for dominance constraints will take exponential worst-case runtime. At first sight, it seems that we have fallen into the expressivity trap: We have a formalism that allows us to model scope underspecification very cleanly, but actually computing with this formalism is expensive.

### 3.2 Reduction to Set Constraints

In reaction to this NP-completeness result, Duchier and Niehren (2000) applied techniques from *constraint programming* to the problem in order to get a more efficient solver. Constraint programming (Apt, 2003) is a standard approach to solving NP-complete combinatorial problems. In this paradigm, a problem is modelled as a formula in a logical constraint language. The program searches for values for the variables in the formula that satisfy the formula. In order to reduce the size of the search space, it performs cheap deterministic inferences that exclude some values of the variables *(propagation)*, and only after propagation can supply no further information it performs a non-deterministic case distinction *(distribution)*.

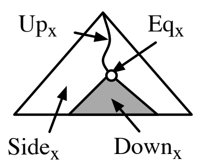


Figure 3: The four node sets

Duchier and Niehren solved dominance constraints by encoding them as *finite set constraints*. Finite set constraints (Müller and Müller, 1997) are formulas that talk about relations between (terms that denote) finite sets of integers, such as inclusion $X \subseteq Y$ or equality $X = Y$. Efficient solvers for set constraints are available, e.g. as part of the Mozart/Oz programming system (Oz Development Team, 2004).

**Reduction to set constraints.** The basic idea underlying the reduction is that a tree can be represented by specifying for each node *v* of this tree which nodes are dominated by *v*, which ones dominate *v*, which ones are equal to *v* (i.e. just *v* itself), and which ones are “disjoint” from *v* (Fig. 3). These four node sets are a partition of the nodes in the tree.

Now the solver introduces for each variable $X$ in a dominance constraint φ four variables $Eq_X$, $Up_X$, $Down_X$, $Side_X$ for the sets of node variables that denote nodes in the respective region of the tree, relative to $X$. The atoms in φ are translated into constraints on these variables. For instance, a dominance atom $X \triangleleft^* Y$ is translated into

$$Up_X \subseteq Up_Y \wedge Down_Y \subseteq Down_X \wedge Side_X \subseteq Side_Y$$

This constraint encodes that all variables whose denotation dominates the denotation of $X$ ($Up_X$) must also dominate the denotation of $Y$ ($Up_Y$), and the analogous statements for the dominated and disjoint variables.

In addition, the constraint program contains various redundant constraints that improve propagation. Now the search for solutions consists in finding satisfying assignments to the set variables. The result is a search tree as shown in Fig. 4: The blue circles represent case distinctions, whereas each green diamond represents a solution of the set constraint (and therefore, a solved form of the dominance constraint). Interestingly, all leaves of the search tree in Fig. 4 are solution nodes; the search never runs into inconsistent constraints. This seems to happen systematically when solving any constraints that come from underspecification.

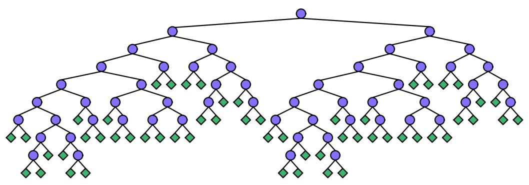


Figure 4: Search tree for constraint 42 from the Rondane Treebank.

## 3.3 A graph-based solver

This behaviour of the set-constraint solver is extremely surprising: The key characteristic of an NP-complete problem is that the search tree must necessarily contain failed nodes on some inputs. The fact that the solver never runs into failure is a strong indication that there is a fragment of dominance constraints that contains all constraints that are used in practice, and that the solver automatically exploits this fragment. This begs the question: What is this fragment, and can we develop even faster solvers that are specialised to it?

One such fragment is the fragment of *normal* dominance constraints (Althaus et al., 2003). The most important restriction that a normal dominance constraint φ must satisfy is that it is *overlap-free*: Whenever φ contains two labelling atoms $X{:}f(\ldots)$ and $Y{:}g(\ldots)$ (where $f$ and $g$ may be equal), it must also contain an inequality atom $X \neq Y$. As a consequence, no two labelled variables in a normal constraint may be mapped to the same node. This is acceptable or even desirable in underspecification: We are not interested in solutions of the constraint in Fig. 2 in which the quantifier representations overlap. On the other hand, the NP-completeness proof in (Koller et al., 1998) is no longer applicable to overlap-free constraints. Hence normal dominance constraints are a fragment that is sufficient from a modelling perspective, and possibly admits polynomial-time solvers.

Indeed, it can be shown that the satisfiability problem of normal dominance constraints can be

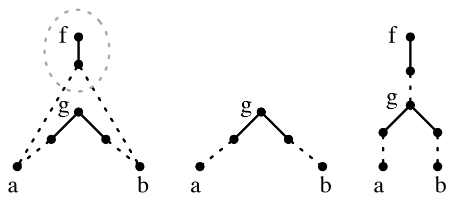


Figure 5: An example computation of the graph solver.

decided in linear time (Thiel, 2004), and the linear algorithm can be used to enumerate $N$ solved forms of a constraint of size $n$ in time $O(n^2N)$. We now present the simpler $O(n^2N)$ enumeration algorithm by Bodirsky et al. (2004).[1] Note that $N$ may still be exponential in $n$.

**Dominance Graphs.** The crucial insight underlying the fast solvers for normal dominance constraints is that such constraints can be seen as *dominance graphs*, and can be processed using graph algorithms. Dominance graphs are directed graphs with two kinds of edges: *tree edges* and *dominance edges*. The graph without the dominance edges must be a forest; the trees of this forest are called the *fragments* of the graph. In addition, the dominance edges must go from *holes* (i.e., unlabelled leaves) of fragments to *roots* of other fragments. For instance, we can view the graph in Fig. 2, which we introduced as an informal notation for a dominance constraint, directly as a dominance graph with three fragments and two (dotted) dominance edges.

A dominance graph $G$ which is a forest is called *in solved form*. We say that $G'$ is a *solved form of* a graph $G$ iff $G'$ is in solved form, $G$ and $G'$ contain the same tree edges, and the reachability relation of $G'$ extends that of $G$. Using this definition, it is possible to define a mapping between normal dominance constraints and dominance graphs such that the solved forms of the graph can serve as solved forms of the constraint – i.e., we can reduce constraint solving to graph solving.

By way of example, consider Fig. 5. The dominance graph on the left is not in solved form, because it contains nodes with more than one incoming dominance edge. By contrast, the other two dominance graphs are in solved form. Because the graph on the right has the same tree edges as the one on the left and extends its reachability relation, it is also a solved form of the left-hand graph.

[1]The original paper defines the algorithm for *weakly* normal dominance constraints, a slight generalisation.

```
GRAPH-SOLVER($G'$)
1  **if** $G'$ is already in solved form
2     **then return** G'
3  $free \leftarrow$ FREE-FRAGMENTS($G'$)
4  **if** $free = \emptyset$
5     **then fail**
6  **choose** $F \in free$
7  $G_1, \ldots, G_k \leftarrow$ WCCS($G' - F$)
8  **for** each $G_i \in G_1, \ldots, G_k$
9  **do** $S_i \leftarrow$ GRAPH-SOLVER($G_i$)
10 $S \leftarrow$ Attach $S_1, \ldots, S_k$ under $F$
11 **return** $S$
```

Figure 6: The graph solver.

**The algorithm.** The graph-based enumeration algorithm is a recursive procedure that successively splits a dominance graph into smaller parts, solves them recursively, and combines them into complete solved forms. In each step, the algorithm identifies the *free fragments* of the dominance (sub-)graph. A fragment is free if it has no incoming dominance edges, and all of its holes are in different biconnected components of the undirected version of the dominance graph. It can be shown (Bodirsky et al., 2004) that if a graph $G$ has any solved form and $F$ is a free fragment of $G$, then $G$ has a solved form in which $F$ is at the root.

The exact algorithm is shown in Fig. 6. It computes the free fragments of a sub-dominance graph $G'$ in line 3. Then it chooses one of the free fragments, removes it from the graph, and calls itself recursively on the weakly connected components $G_1, \ldots, G_k$ of the resulting graph. Each recursive call will compute a solved form $S_i$ of the connected component $G_i$. Now for each $G_i$ there is exactly one hole $h_i$ of $F$ that is connected to some node in $G_i$ by a dominance edge. We can obtain a solved form for $G'$ by combining $F$ and all the $S_i$ with dominance edges from $h_i$ to the root of $S_i$ for each $i$.

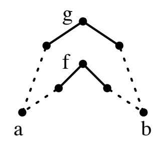


Figure 7: An unsolvable dominance graph.

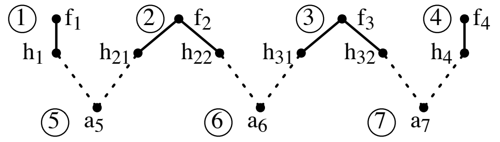


Figure 8: The chain of length 4.

| | |
|---|---|
| $\{1,2,3,4,5,6,7\}$ : | $\langle 1, h_1 \mapsto \{2,3,4,5,6,7\}\rangle$ |
| | $\langle 2, h_{21} \mapsto \{1,5\}, h_{22} \mapsto \{3,4,6,7\}\rangle$ |
| | $\langle 3, h_{31} \mapsto \{1,2,5,6\}, h_{32} \mapsto \{4,7\}\rangle$ |
| | $\langle 4, h_4 \mapsto \{1,2,3,5,6,7\}\rangle$ |
| $\{2,3,4,5,6,7\}$ : | $\langle 2, h_{21} \mapsto \{5\}, h_{22} \mapsto \{3,4,6,7\}\rangle$ |
| | $\langle 3, h_{31} \mapsto \{2,5,6\}, h_{32} \mapsto \{4,7\}\rangle$ |
| | $\langle 4, h_4 \mapsto \{\mathbf{2},\mathbf{3},\mathbf{5},\mathbf{6},\mathbf{7}\}\rangle$ |
| $\{1,2,3,5,6,7\}$ : | $\langle 1, h_1 \mapsto \{\mathbf{2},\mathbf{3},\mathbf{5},\mathbf{6},\mathbf{7}\}\rangle$ |
| | $\langle 2, h_{21} \mapsto \{1,5\}, h_{22} \mapsto \{3,6,7\}\rangle$ |
| | $\langle 3, h_{31} \mapsto \{1,2,5,6\}, h_{32} \mapsto \{7\}\rangle$ |
| $\{\mathbf{2},\mathbf{3},\mathbf{5},\mathbf{6},\mathbf{7}\}$ : | $\langle 2, h_{21} \mapsto \{5\}, h_{22} \mapsto \{3,6,7\}\rangle$ |
| | $\langle 3, h_{31} \mapsto \{2,5,6\}, h_{32} \mapsto \{7\}\rangle$ |
| $\cdots$ | $\cdots$ |

Figure 9: A part of the chart computed for the constraint in Fig. 8.

The algorithm is written as a nondeterministic procedure which makes a nondeterministic choice in line 6, and can fail in line 5. We can turn it into a deterministic algorithm by considering the nondeterministic choices as case distinctions in a search tree, as in Fig. 4. However, if the input graph $G$ is solvable, we know that every single leaf of the search tree must correspond to a (different) solved form, because for every free fragment that can be chosen in line 6, there is a solved form that has this fragment as its root. Conversely, if $G$ is unsolvable, every single branch of the search tree will run into failure, because it would claim the existence of a solved form otherwise. So the algorithm decides solvability in polynomial time.

An example computation of GRAPH-SOLVER is shown in Fig. 5. The input graph is shown on the left. It contains exactly one free fragment $F$; this is the fragment whose root is labelled with $f$. (The single-node fragments both have incoming dominance edges, and the two holes of the fragment with label $g$ are in the same biconnected component.) So the algorithm removes $F$ from the graph, resulting in the graph in the middle. This graph is in solved form (it is a tree), so we are finished. Finally the algorithm builds a solved form for the whole graph by plugging the solved form in the middle into the single hole of $F$; the result is shown on the right. By contrast, the graph in Fig. 7 has no solved forms. The solver will recognise this immediately, because none of the fragments is free (they either have incoming dominance edges, or their holes are biconnected).

### 3.4 A graph solver with charts

The graph solver is a great step forward towards efficient constraint solving, and towards an understanding of why (normal) dominance constraints can be solved efficiently. But it wastes time when it is called multiple times for the same subgraph, because it will solve it anew each time. In solving, for instance, the graph shown in Fig. 8, it will solve the subgraph consisting of the fragments $\{2,3,5,6,7\}$ twice, because it can pick the fragments 1 and 4 in either order.

We will now present a previously unpublished optimisation for the solver that uses caching to alleviate this problem. The data structure we use for caching (we call it "chart" below because of its obvious parallels to charts in parsing) assigns each subgraph of the original graph a set of *splits*. Splits encode the splittings of the graph into weakly connected components that take place when a free fragment is removed. Formally, a split for the subgraph $G'$ consists of a reference to a fragment $F$ that is free in $G'$ and a partial function that maps some nodes of $F$ to subgraphs of $G'$. A split is determined uniquely by $G'$ and $F$.

Consider, by way of example, Fig. 9, which displays a part of the chart that we want to compute for the constraint in Fig. 8. In the entire graph $G$ (represented by the set $\{1,\dots,7\}$ of fragments), the fragments 1, 2, 3, and 4 are free. As a consequence, the chart contains a split for each of these four fragments. If we remove fragment 1 from $G$, we end up with a weakly connected graph $G_1$ containing the fragments $\{2,\dots,7\}$. There is a dom-

```
GRAPH-SOLVER-CHART($G'$)
if there is an entry for $G'$ in the chart
   then return true
$free \leftarrow$ FREE-FRAGMENTS($G'$)
if $free = \emptyset$
   then return false
if $G'$ contains only one fragment
   then return true

for each $F \in free$
do $split \leftarrow$ SPLIT($G', F$)
    for each $S \in$ WCCS($G' - F$)
    do if GRAPH-SOLVER-CHART($S$) = false
          then return false
    add $(G', split)$ to the chart
return true
```

Figure 10: The graph solver with charts

inance edge from the hole $h_1$ into $G_1$, so once we have a solved form of $G_1$, we will have to plug it into $h_1$ to get a solved form of $G$; therefore $G_1$ is assigned to $h_1$ in the split. On the other hand, if we remove fragment 2 from $G$, $G$ is split into two weakly connected components $\{1,5\}$ and $\{3,4,6,7\}$, whose solved forms must be plugged into $h_{21}$ and $h_{22}$ respectively.

We can compute a chart like this using the algorithm shown in Fig. 10. This recursive algorithm gets some subgraph $G'$ of the original graph $G$ as its first argument. It returns **true** if $G'$ is solvable, and **false** if it isn't. If an entry for its argument $G'$ was already computed and recorded in the chart, the procedure returns immediately. Otherwise, it computes the free fragments of $G'$. If there are no free fragments, $G$ was unsolvable, and thus the algorithm returns **false**; on the other hand, if $G'$ only contains one fragment, it is solved and we can immediately return **true**.

If none of these special cases apply, the algorithm iterates over all free fragments $F$ of $G'$ and computes the (unique) split that places $F$ at the root of the solved forms. If all weakly connected components represented in the split are solvable, it records the split as valid for $G'$, and returns **true**.

If the algorithm returns with value **true**, the chart will be filled with splits for all subgraphs of $G$ that the GRAPH-SOLVER algorithm would have visited. It is also guaranteed that every split in the chart is used in a solved form of the graph. Extracting the actual solved forms from the chart is straightforward, and can be done essentially like for parse charts of context-free grammar.

**Runtime analysis.** The chart computed by the chart solver for a dominance graph with $n$ nodes and $m$ edges can grow to at most $O(n \cdot \text{wcsg}(G))$ entries, where $\text{wcsg}(G)$ is the number of weakly connected subgraphs of $G$: All subgraphs for which GRAPH-SOLVER-CHART is called are weakly connected, and for each such subgraph there can be at most $n$ different splits. Because a recursive call returns immediately if its argument is already present in the chart, this means that at most $O(n \cdot \text{wcsg}(G))$ calls spend more than the expected constant time that it takes to look up $G'$ in the chart. Each of these calls needs time $O(m+n)$, the cost of computing the free fragments.

As a consequence, the total time that GRAPH-SOLVER-CHART takes to fill the chart is $O(n(n+m)\text{wcsg}(G))$. Applied to a dominance *constraint* with $k$ atoms, the runtime is $O(k^2\text{wcsg}(G))$. On the other hand, if $G$ has $N$ solved forms, it takes time $O(N)$ to extract these solved forms from the chart. This is a significant improvement over the $O(n(n+m)N)$ time that GRAPH-SOLVER takes to enumerate all solved forms. A particularly dramatic case is that of *chains* – graphs with a zig-zag shape of $n$ upper and $n-1$ lower fragments such as in Fig. 8, which occur frequently as part of underspecified descriptions. A chain has only $O(n^2)$ weakly connected subgraphs and $O(n)$ edges, so the chart can be filled in time $O(n^4)$, despite the fact that the chain has $\frac{1}{n+1}\binom{2n}{n}$ solved forms (this is the $n$-th Catalan number, which grows faster than $n!$). The worst case for the chart size is shown in Fig. 11. If such a graph has $n$ upper fragments, it has $O(2^n)$ weakly connected subgraphs, so the chart-filling phase takes time $O(n^2 2^n)$. But this is still dominated by the $N = n!$ solved forms that this graph has.

## 4 Evaluation

We conclude this paper with a comparative runtime evaluation of the presented dominance con-

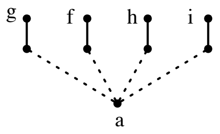


Figure 11: A worst-case graph for the chart solver.

| | | constraints | max. solved forms |
|---|---|---|---|
| Rondane | | 961 | ? |
| Nets | | 879 | $2.4 \cdot 10^{12}$ |
| Nets $< 10^6$ solved forms | | 852 | 997920 |
| Solver | | solvable | max. solved forms |
| Saturation | (§3.1) | 757 | 10030 |
| Set constraints | (§3.2) | 841 | 557472 |
| Graph | (§3.3) | 850 | 768254 |
| Chart | (§3.4) | 852 | 997920 |
| LKB | | 682 | 17760 |
| All | | 682 | 7742 |

Figure 12: Sizes of the data sets.

straint solvers. To put the results into context, we also compare the runtimes with a solver for Minimal Recursion Semantics (MRS) (Copestake et al., 2004), a different formalism for scope underspecification.

**Resources.** As our test set we use constraints extracted from the Rondane treebank, which is distributed as part of the English Resource Grammar (Copestake and Flickinger, 2000). The treebank contains syntactic annotations for sentences from the tourism domain such as (4) above, together with corresponding semantic representations.

The semantics is represented using MRS descriptions, which we convert into normal dominance constraints using the translation specified by Niehren and Thater (2003). The translation is restricted to MRS constraints having certain structural properties (called *nets*). The treebank contains 961 MRS constrains, 879 of which are nets.

For the runtime evaluation, we restricted the test set to the 852 nets with less than one million solved forms. The distribution of these constraints over the different constraint sizes (i.e. number of fragments) is shown in Fig. 15. We solved them using implementations of the presented dominance constraint solvers, as well as with the MRS solver in the LKB system (Copestake and Flickinger, 2000).

**Runtimes.** As Fig. 12 shows, the chart solver is the only solver that could solve all constraints in the test set; all other solvers ran into memory limitations on some inputs.[2] The increased complexity of constraints that each solver can handle (given as the maximum number of solved forms of a solvable constraint) is a first indication that the repeated analysis and improvement of dominance constraint solvers described earlier was successful.

Fig. 13 displays the result of the runtime comparison, taking into account only those 682 constraints that all solvers could solve. For each constraint size (counted in number of fragments), the graph shows the mean quotient of the time to enumerate all solved forms by the number of solved forms, averaged over all constraints of this size. Note that the vertical axis is logarithmic, and that the runtimes of the LKB and the chart solver for constraints up to size 6 are too small for accurate measurement.

The figure shows that each new generation of dominance constraint solvers improves the performance by an order of magnitude. Another difference is in the slopes of the graphs. While the saturation solver takes increasingly more time per solved form as the constraint grows, the set constraint and graph solvers remain mostly constant for larger constraints, and the line for the chart solver even goes down. This demonstrates an improved management of the combinatorial explosion. It is also interesting that the line of the set-constraint solver is almost parallel to that of the graph solver, which means that the solver really does exploit a polynomial fragment on real-world data.

The LKB solver performs very well for smaller constraints (which make up about half of the data set): Except for the chart algorithm introduced in this paper, it outperforms all other solvers. For larger constraints, however, the LKB solver gets very slow. What isn't visible in this graph is that the LKB solver also exhibits a dramatically higher variation in runtimes for constraints of the same size, compared to the dominance solvers. We believe this is because the LKB solver has been optimised by hand to deal with certain classes of in-

[2]On a 1.2 GHz PC with 2 GB memory.

puts, but at its core is still an uncontrolled exponential algorithm.

We should note that the chart-based solver is implemented in C++, while the other dominance solvers are implemented in Oz, and the MRS solver is implemented in Common Lisp. This accounts for some constant factor in the runtime, but shouldn't affect the differences in slope and variability.

**Effect of the chart.** Because the chart solver is especially efficient if the chart remains small, we have compared how the number of solved forms and the chart size (i.e. number of splits) grow with the constraint size (Fig. 14). The graph shows that the chart size grows much more slowly than the number of solved forms, which supports our intuition that the runtime of the chart solver is asymptotically less than that of the graph solver by a significant margin. The chart for the most ambiguous sentence in the treebank (sentence (4) above) contains 74.960 splits. It can be computed in less than ten seconds. By comparison, enumerating all solved forms of the constraint would take about a year on a modern PC. Even determining the number of solved forms of this constraint is only possible based on the chart.

## 5 Conclusion

In this paper we described the evolution of solvers for dominance constraints, a logical formalism used for the underspecified processing of scope ambiguities. We also presented a new solver, which caches the intermediate results of a graph solver in a chart. An empirical evaluation shows that each solver is significantly faster than the previous one, and that the new chart-based solver is the fastest underspecification solver available today. It is available online at `http://utool.sourceforge.net`.

Each new solver was based on an analysis of the main sources of inefficiency in the previous solver, as well as an increasingly good understanding of the input data. The main breakthrough was the realisation that normal dominance constraints have polynomial satisfiability and can be solved using graph algorithms. We believe that this strategy of starting with a clean, powerful formalism and then successively searching for a fragment that contains all practically relevant inputs and excludes the pathologically hard cases is applicable to other problems in computational linguistics as well.

However, it is clear that the concept of "all practically relevant inputs" is a moving target. In this paper, we have equated it with "all inputs that can be generated by a specific large-scale grammar", but new grammars or different linguistic theories may generate underspecified descriptions that no longer fall into the efficient fragments. In our case, it is hard to imagine what dominance constraint used in scope underspecification wouldn't be normal, and we have strong intuitions that all useful constraints must be nets, but it is definitely an interesting question how our algorithms could be adapted to, say, the alternative scope theory advocated by Joshi et al. (2003).

An immediate line of future research is to explore uses of the chart data structure that go beyond pure caching. The general aim of underspecification is not to simply enumerate all readings of a sentence, but to use the underspecified description as a platform on which readings that are theoretically possible, but infelicitous in the actual context, can be eliminated. The chart may prove to be an interesting platform for such operations, which combines advantages of the underspecified description (size) and the readings themselves (explicitness).

**Acknowledgements.** The work has been funded by the DFG in the Collaborative Research Centre 378 *Ressource-Adaptive Cognitive Processes*, project MI 2 (CHORUS).

We would like to thank Joachim Niehren and Denys Duchier for the extremely fruitful collaboration on dominance constraint solving, Ann Copestake and Dan Flickinger for helpful discussions about the ERG and the LKB solver, and our reviewers for their comments. The primary implementors of the various earlier constraint solvers were Katrin Erk and Sebastian Padó (§3.1), Denys Duchier (§3.2), and Sebastian Miele (§3.3).

## References

Ernst Althaus, Denys Duchier, Alexander Koller, Kurt Mehlhorn, Joachim Niehren, and Sven Thiel. 2003. An efficient graph algorithm for dominance constraints. *Journal of Algorithms*, 48:194–219.

Krzysztof R. Apt. 2003. *Principles of Constraint Programming*. Cambridge University Press.

Manuel Bodirsky, Denys Duchier, Joachim Niehren, and Sebastian Miele. 2004. An efficient algorithm for weakly normal dominance constraints. In *ACM-SIAM Symposium on Discrete Algorithms*. The ACM Press.

Ann Copestake and Dan Flickinger. 2000. An open-source grammar development environment and broad-coverage english grammar using HPSG. In *Conference on Language Resources and Evaluation*. The LKB system is available at `http://www.delph-in.net/lkb/`.

Ann Copestake, Dan Flickinger, Carl Pollard, and Ivan Sag. 2004. Minimal recursion semantics: An introduction. *Journal of Language and Computation*. To appear.

Denys Duchier and Joachim Niehren. 2000. Dominance constraints with set operators. In *Proceedings of the First International Conference on Computational Logic*, number 1861 in Lecture Notes in Computer Science, pages 326–341. Springer-Verlag, Berlin.

Markus Egg, Alexander Koller, and Joachim Niehren. 2001. The Constraint Language for Lambda Structures. *Logic, Language, and Information*, 10:457–485.

Aravind Joshi, Laura Kallmeyer, and Maribel Romero. 2003. Flexible composition in LTAG, quantifier scope and inverse linking. In Harry Bunt, Ielka van der Sluis, and Roser Morante, editors, *Proceedings of the Fifth International Workshop on Computational Semantics*, pages 179–194, Tilburg.

Alexander Koller, Joachim Niehren, and Ralf Treinen. 1998. Dominance constraints: Algorithms and complexity. In *Proceedings of LACL*, pages 106–125. Appeared in 2001 as volume 2014 of LNAI, Springer Verlag.

Tobias Müller and Martin Müller. 1997. Finite set constraints in Oz. In François Bry, Burkhard Freitag, and Dietmar Seipel, editors, *13. Workshop Logische Programmierung*, pages 104–115, Technische Universität München.

Joachim Niehren and Stefan Thater. 2003. Bridging the gap between underspecification formalisms: Minimal recursion semantics as dominance constraints. In *Proceedings of the 41st Annual Meeting of the Association for Computational Linguistics*.

Oz Development Team. 2004. The Mozart Programming System. Web pages. `http://www.mozart-oz.org`.

Sven Thiel. 2004. *Efficient Algorithms for Constraint Propagation and for Processing Tree Descriptions*. Ph.D. thesis, Department of Computer Science, Saarland University.

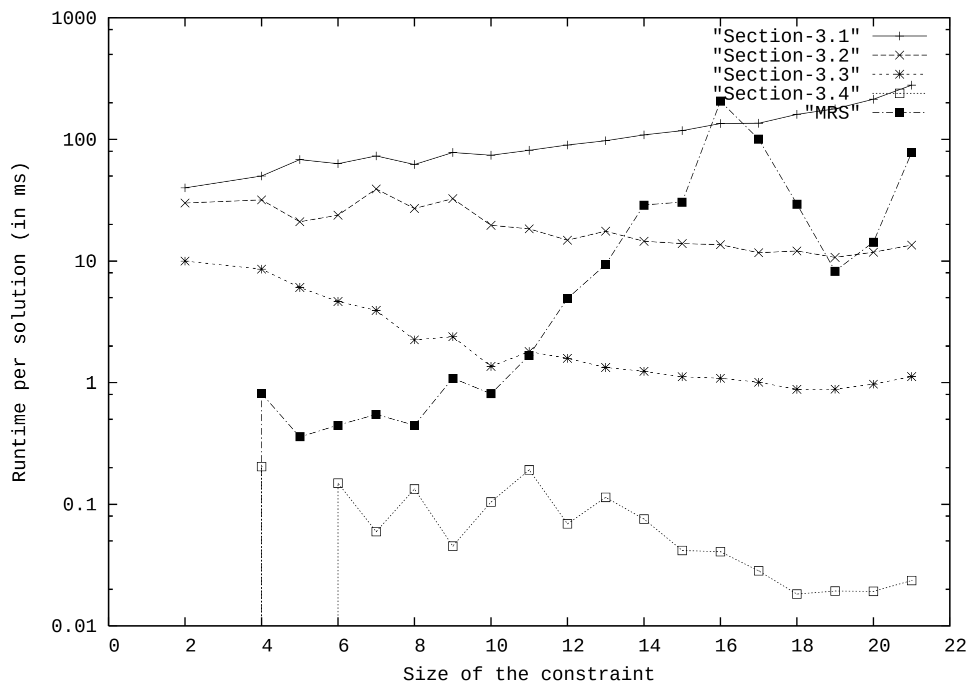


Figure 13: Average runtimes per solved form, for each constraint size (number of fragments).

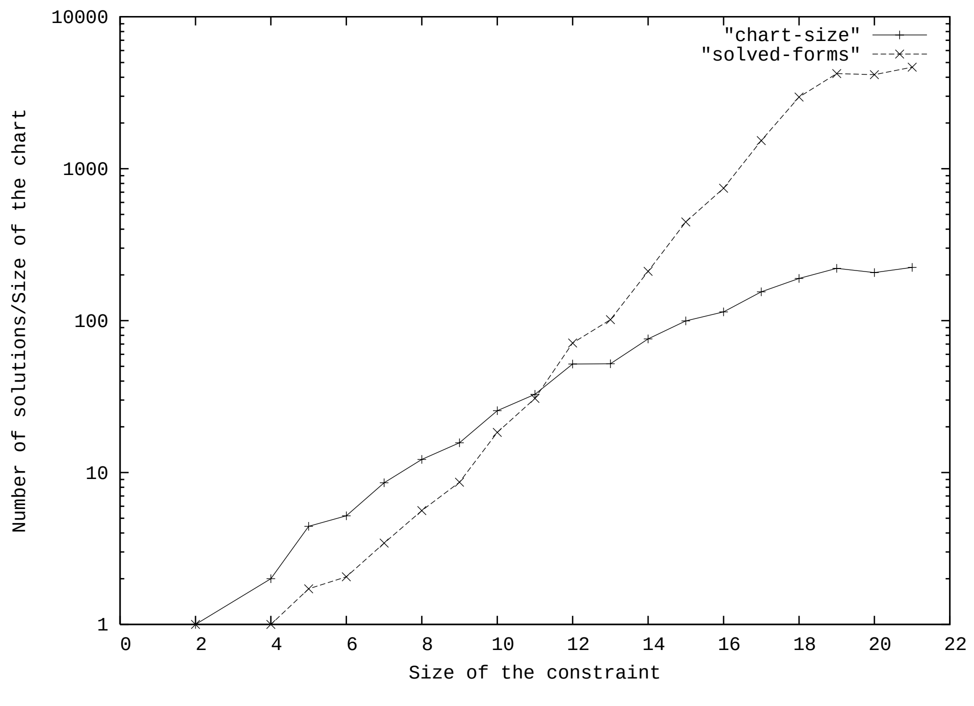


Figure 14: Average size of the chart compared to the average number of solved forms, for each constraint size. Notice that the measurements are based upon the same set of constraints as in Fig. 13, which contains very few constraints of size 20 or more.

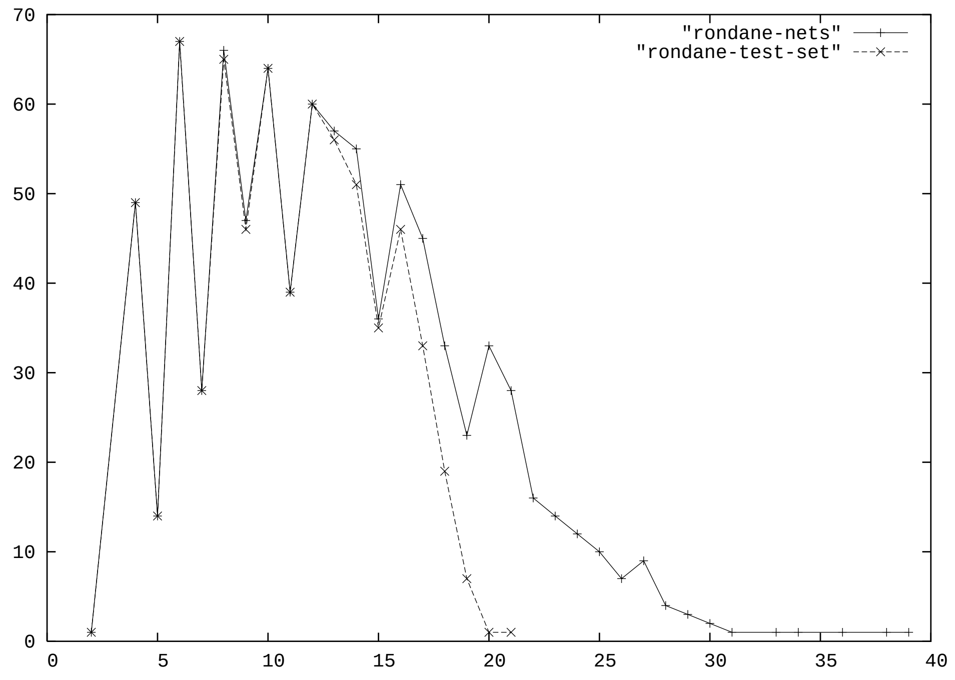


Figure 15: Distribution of the constraints in Rondane over the different constraint sizes. The solid line indicates the 852 nets with less than one million solved forms; the dashed line indicates the 682 constraints that all solvers could solve.

# Hunmorph: open source word analysis

**Viktor Trón**
IGK, U of Edinburgh
2 Buccleuch Place
EH8 9LW Edinburgh
v.tron@ed.ac.uk

**György Gyepesi**
K-PRO Ltd.
H-2092 Budakeszi
Villám u. 6.
ggyepesi@kpro.hu

**Péter Halácsy**
Centre of Media Research and Education
Stoczek u. 2
H-1111 Budapest
hp@mokk.bme.hu

**András Kornai**
MetaCarta Inc.
350 Massachusetts Avenue
Cambridge MA 02139
andras@kornai.com

**László Németh**
CMRE
Stoczek u. 2
H-1111 Budapest
nemeth@mokk.bme.hu

**Dániel Varga**
CMRE
Stoczek u. 2
H-1111 Budapest
daniel@mokk.bme.hu

## Abstract

Common tasks involving orthographic words include spellchecking, stemming, morphological analysis, and morphological synthesis. To enable significant reuse of the language-specific resources across all such tasks, we have extended the functionality of the open source spellchecker `MySpell`, yielding a generic word analysis library, the runtime layer of the `hunmorph` toolkit. We added an offline resource management component, `hunlex`, which complements the efficiency of our runtime layer with a high-level description language and a configurable precompiler.

## 0 Introduction

Word-level analysis and synthesis problems range from strict recognition and approximate matching to full morphological analysis and generation. Our technology is predicated on the observation that all of these problems are, when viewed algorithmically, very similar: the central problem is to dynamically analyze complex structures derived from some lexicon of base forms. Viewing word analysis routines as a unified problem means sharing the same codebase for a wider range of tasks, a design goal carried out by finding the parameters which optimize each of the analysis modes independently of the language-specific resources.

The C/C++ runtime layer of our toolkit, called `hunmorph`, was developed by extending the codebase of `MySpell`, a reimplementation of the well-known `Ispell` spellchecker. Our technology, like the `Ispell` family of spellcheckers it descends from, enforces a strict separation between the language-specific resources (known as `dictionary` and `affix` files), and the runtime environment, which is independent of the target natural language.

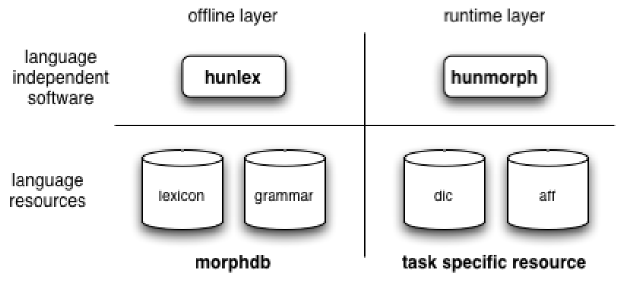


Figure 1: Architecture

Compiling accurate wide coverage machine-readable dictionaries and coding the morphology of a language can be an extremely labor-intensive task, so the benefit expected from reusing the language-specific input database across tasks can hardly be overestimated. To facilitate this resource sharing and to enable systematic task-dependent optimizations from a central lexical knowledge base, we designed and implemented a powerful offline layer we call `hunlex`. `Hunlex` offers an easy to use general framework for describing the lexicon and morphology of any language. Using this description it can generate the language-specific `aff/dic` resources, optimized for the task at hand. The architecture of our toolkit is depicted in Figure 1. Our toolkit is released under a permissive LGPL-style license and can be freely downloaded from `mokk.bme.hu/resources/hunmorph`.

The rest of this paper is organized as follows. Section 1 is about the runtime layer of our toolkit. We discuss the algorithmic extensions and implementational enhancements in the C/C++ runtime layer over `MySpell`, and also describe the newly created Java port `jmorph`. Section 2 gives an overview of the offline layer `hunlex`. In Section 3 we consider the free open source software alternatives and offer our conclusions.

## 1 The runtime layer

Our development is a prime example of code reuse, which gives open source software development most of its power. Our codebase is a direct descendant of `MySpell`, a thread-safe C++ spell-checking library by Kevin Hendricks, which descends from `Ispell` Peterson (1980), which in turn goes back to Ralph Gorin's `spell` (1971), making it probably the oldest piece of linguistic software that is still in active use and development (see `fmg-www.cs.ucla.edu/fmg-members/geoff/ispell.html`).

The key operation supported by this codebase is *affix stripping*. Affix rules are specified in a static resource (the `aff` file) by a sequence of conditions, an append string, and a strip string: for example, in the rule forming the plural of *body* the strip string would be *y*, and the affix string would be *ies*. The rules are reverse applied to complex input wordforms: after the append string is stripped and the edge conditions are checked, a pseudo-stem is hypothesized by appending the strip string to the stem which is then looked up in the base dictionary (which is the other static resource, called the `dic` file).

Lexical entries (base forms) are all associated with sets of *affix flags,* and affix flags in turn are associated to sets of affix rules. If the hypothesized base is found in the dictionary after the reverse application of an affix rule, the algorithm checks whether its flags contain the one that the affix rule is assigned to. This is a straight table-driven approach, where affix flags can be interpreted directly as lexical features that license entire subparts of morphological paradigms. To pick applicable affix rules efficiently, `MySpell` uses a fast indexing technique to check affixation conditions.

In theory, affix-rules should only specify genuine prefixes and suffixes to be stripped before lexical lookup. But in practice, for languages with rich morphology, the affix stripping mechanism is (ab)used to strip complex clusters of affix morphs in a single step. For instance, in Hungarian, due to productive combinations of derivational and inflectional affixation, a single nominal base can yield up to a million word forms. To treat all these combinations as affix clusters, legacy ispell resources for Hungarian required so many combined affix rule entries that its resource file sizes were not manageable.

To solve this problem we extended the affix stripping technique to a multistep method: after stripping an affix cluster in step $i$, the resulting pseudo-stem can be stripped of affix clusters in step $i+1$. Restrictions of rule application are checked with the help of flags associated to affixes analogously to lexical entries: this only required a minor modification of the data structure coding affix entries and a recursive call for affix stripping. By cross-checking flags of prefixes on the suffix (as opposed to the stem only), simultaneous prefixation and suffixation can be made interdependent, extending the functionality to describe circumfixes like German participle *ge+t*, or Hungarian superlative *leg+bb*, and in general provide the correct handling of prefix-suffix dependencies like English *undrinkable* (cf. **undrink*), see Németh et al. (2004) for more details.

Due to productive compounding in a lot of languages, proper handling of composite bases is a feature indispensable for achieving wide coverage. `Ispell` incorporates the possibility of specifying lexical restrictions on compounding implemented as switches in the base dictionary. However, the algorithm allows any affixed form of the bases that has the relevant switch to be a potential member of a compound, which proves not to be restrictive enough. We have improved on this by the introduction of position-sensitive compounding. This means that lexical features can specify whether a base or affix can occur as leftmost, rightmost or middle constituent in compounds and whether they can *only* appear in compounds. Since these features can also be specified on affixes, this provides a welcome solution to a number of residual problems hitherto problematic for open-source spellcheckers. In some Germanic languages, 'fogemorphemes', morphemes which serve linking compound constituents can now be handled easily by allowing position specific compound licensing on the foge-affixes. Another important example is the German common noun: although it is capitalized in isolation, lowercase variants should be accepted when the noun is a compound constituent. By handling lowercasing as a prefix with the compound flag enabled, this phenomenon can be handled in the resource file without resort to language specific knowledge hard-wired in the code-base.

## 1.1 From spellchecking to morphological analysis

We now turn to the extensions of the `MySpell` algorithm that were required to equip `hunmorph` with stemming and morphological analysis functionality. The core engine was extended with an optional output handling interface that can process arbitrary string tags associated with the affix-rules read from the resources. Once this is done, simply outputting the stem found at the stage of dictionary lookup already yields a stemmer. In multistep affix stripping, registering output information associated with the rules that apply renders the system capable of morphological analysis or other word annotation tasks. Thus the *processing of output tags* becomes a mode-dependent parameter that can be:

- switched off (spell-checking)
- turned on only for tag lookup in the dictionary (simple stemming)
- turned on fully to register tags with all rule-applications (morphological analysis)

The single most important algorithmic aspect that distinguishes the recognition task from analysis is the handling of ambiguous structures. In the original `MySpell` design, identical bases are conflated and once their switch-set licensing affixes are merged, there is no way to tell them apart. The correct handling of homonyms is crucial for morphological analysis, since base ambiguities can sometimes be resolved by the affixes. Interestingly, our improvement made it possible to rule out homonymous bases with incorrect simultaneous prefixing and suffixing such as English *out+number+'s*. Earlier these could be handled only by lexical pregeneration of relevant forms or duplication of affixes.

Most importantly, ambiguity arises in relation to the number of analyses output by the system. While with spell-checking the algorithm can terminate after the first analysis found, performing an exhaustive search for all alternative analyses is a reasonable requirement in morphological analysis mode as well as in some stemming tasks. Thus the *exploration of the search space* also becomes an active parameter in our enhanced implementation of the algorithm:

- search until the first correct analysis
- search restricted multiple analyses (e.g., disabling compounds)
- search all alternative analyses

Search until the first analysis is a functionality for recognizers used for spell-checking and stemming for accelerated document indexing. Preemption of potential compound analyses by existing lexical bases serves as a general way of filtering out spurious ambiguities when a reduction is required in the space of alternative analyses. In these cases, frequent compounds which trick the analyzer can be precompiled to the lexicon. Finally, there is a possibility to give back a full set of possible analyses. This output then can be passed to a tagger that disambiguates among the candidate analyses. Parameters can be used that guide the search (such as 'do lexical lookup first at all stages' or 'strip the shortest affix first'), which yield candidate rankings without the use of numerical weights or statistics. These rankings can be used as disambiguation heuristics based on a general idea of blocking (e.g., *Times* would block an analysis of *time+s*). All further parametrization is managed offline by the resource compiler layer, see Section 2.

### 1.2 Reimplementing the runtime layer

In our efforts to gear up the `MySpell` codebase to a fully functional word analysis library we successfully identified various resource-related, algorithmic and implementational bottlenecks of the affix-rule based technology. With these lessons learned, a new project has been launched in order to provide an even more flexible and efficient open source runtime layer. A principled object-oriented refactorization of the same basic algorithm described above has already been implemented in Java. This port, called `jmorph` also uses the `aff/dic` resource formats.

In `jmorph`, various algorithmic options guiding the search (shortest/longest matching affix) can be controlled for each individual rule. The implementation keeps track of affix and compound matches checking conditions only once for a given substring and caching partial results. As a consequence, it ends up being measurably faster than the C++ implementation with the same resources.

The main loop of `jmorph` is driven by configuring *consumers*, i.e., objects which monitor the recursive step that is running. For example the analysis of the form *beszédesek* 'talkative.PLUR' begins by inspecting the global configuration of the analysis: this initial consumer specifies how many analyses, and what kind, need to be found. In Step 1, the initial consumer finds the rule that strips *ek* with stem *beszédes,* builds a consumer that can apply this rule to the output of the analysis returned by the next consumer, and launches the next step with this consumer and stem. In Step 2, this consumer finds the rule stripping *es* with stem *beszéd*, which is found in the lexicon. *beszéd* is not just a string, it is a complete lexical object which lists the rules that can apply to it and all the homonyms. The consumer creates a new analysis that reflects that *beszédes* is formed from *beszéd* by suffixing *es* (a suffix object), and passes this back to its parent consumer, which verifies whether the *ek* suffixation rule is applicable. If not, the Step 1 consumer requests further analyses from the Step 2 consumer. If, however, the answer is positive, the Step 1 consumer returns its analysis to the Step 0 (initial) consumer, which decides whether further analyses are needed.

In terms of functionality, there are a number of differences between the Java and the C++ variants. `jmorph` records the full parse tree of rule applications. By offering various ways of serializing this data structure, it allows for more structured information in the outputs than would be possible by simple concatenation of the tag chunks associated with the rules. Class-based restrictions on compounding is implemented and will eventually supersede the overgeneralizing position-based restrictions that the C++ variant and our resources currently use.

Two major additional features of `jmorph` are its capability of morphological synthesis as well as acting as a guesser (hypothesizing lemmas). Synthesis is implemented by forward application of affix rules starting with the base. Rules have to be indexed by their tag chunks for the search, so synthesis introduces the non-trivial problem of chunking the input tag string. This is currently implemented by plug-ins for individual tag systems, however, this should ideally be precompiled offline since the space of possible tags is limited.

## 2 Resource development and offline precompilation

Due to the backward compatibility of the runtime layer with `MySpell`-style resources, our software can be used as a spellchecker and simplistic stemmer for some 50 languages for which `MySpell` resources are available, see `lingucomponent.openoffice.org/spell_dic.html`.

For languages with complex morphology, compiling and maintaining these resources is a painful undertaking. Without using a unified framework for morphological description and a principled method of precompilation, resource developers for highly agglutinative languages like Hungarian (see `magyarispell.sourceforge.net`) have to resort to a maze of scripts to maintain and precompile `aff` and `dic` files. This problem is intolerably magnified once morphological tags or additional lexicographic information are to be entered in order to provide resources for the analysis routines of our runtime layer.

The offline layer of our toolkit seeks to remedy this by offering a high-level description language in which grammar developers can specify rule-based morphologies and lexicons (somewhat in the spirit of `lexc` Beesley and Karttunen (2003), the frontend to Xerox's Finite State Toolkit). This promises rapid resource development which can then be used in various tasks. Once primary resources are created, `hunlex`, the offline precompiler can generate `aff` and `dic` resources optimized for the runtime layer based on various compile-time configurations.

Figure 2 illustrates the description language with a fragment of English morphology describing plural formation. Individual rules are separated by commas. The syntax of the rule descriptions organized around the notion of *information blocks*. Blocks are introduced by keywords (like `IF:`) and allow the encoding of various properties of a rule (or a lexical entry), among others specifying affixation (`+es`), substitution, character truncation before affixation (`CLIP: 1`), regular expression matches (`MATCH: [^o]o`), positive and negative lexical feature conditions on application (`IF: f-v_altern`), feature inheritance, output (continuation) references (`OUT: PL_POSS`), output tags (`TAG: "[PLUR]"`).

One can specify the rules that can be applied to the output of a rule and also one can specify application conditions on the input to the rule. These two possibilities allow for many different styles of morphological description: one based on input feature constraints, one based on continuation classes (paradigm indexes), and any combination between these two extremes. On top of this, regular expression matches on the input can also be used as conditions on rule application.

Affixation rules "grouped together" here under `PLUR` can be thought of as allomorphic rules of the plural morpheme. Practically, this allows information about the morpheme shared among variants (e.g., morphological tag, recursion level, some output information) to be abstracted in a *preamble* which then serves as a default for the individual rules. Most importantly, the grouping of rules

```
PL
        TAG: "[PLUR]"
        OUT: PL_POSS
# house -> houses
, +s   MATCH: [^shoxy]    IF: regular
# kiss -> kisses
, +es  MATCH: [^c]s       IF: regular
# ...
# ethics
, +    MATCH: cs          IF: regular
# body -> bodies <C> is a regexp macro
, +ies MATCH: <C>y CLIP:1 IF: regular
# zloty -> zlotys
, +s   MATCH: <C>y       IF: y-ys
# macro -> macros
, +s   MATCH: [^o]o      IF: regular
# potato -> potatoes
, +es  MATCH: [^o]o      IF: o-oes
# wife -> wives
, +ves MATCH: fe CLIP: 2 IF: f-ves
# leaf -> leaves
, +ves MATCH: f  CLIP: 1 IF: f-ves
;
```

Figure 2: `hunlex` grammar fragment

into morphemes serves to index those rules which can be referenced in output conditions, For example, in the above the plural morpheme specifies that the plural possessive rules can be applied to its output (`OUT: PL_POSS`). This design makes it possible to handle some morphosyntactic dimensions (part of speech) very cleanly separated from the conditions regulating the choice of allomorphs, since the latter can be taken care of by input feature checking and pattern matching conditions of rules. The lexicon has the same syntax as the grammar only that morphemes stand for lemmas and variant rules within the morpheme correspond to stem allomorphs.

Rules with zero affix morph can be used as *filters* that decorate their inputs with features based on their orthographic shape or other features present. This architecture enables one to let only exceptions specify certain features in the lexicon while regular words left unspecified are assigned a default feature by the filters (see `PL_FILTER` in

```
REGEXP: C [bcdfgklmnprstvwxyz];

DEFINE: N
  OUT: SG PL_FILTER
  TAG: NOUN
;

PL_FILTER
  OUT:
       PL
  FILTER:
       f-ves
       y-ys
       o-oes
       regular
, DEFAULT:
       regular
;
```

Figure 3: Macros and filters in `hunlex`

Figure 3) potentially conditioned the same way as any rule application. Feature inheritance is fully supported, that is, filters for particular dimensions of features (such as the plural filter in Figure 3) can be written as independent units. This design makes it possible to engineer sophisticated filter chains decorating lexical items with various features relevant for their morphological behavior. With this at hand, extending the lexicon with a regular lexeme just boils down to specifying its base and part of speech. On the other hand, independent sets of filter rules make feature assignments transparent and maintainable.

In order to support concise and maintainable grammars, the description language also allows (potentially recursive) macros to abbreviate arbitrary sets of blocks or regular expressions, illustrated in Figure 3.

The resource compiler `hunlex` is a standalone program written in OCaml which comes with a command-line as well as a Makefile as toplevel control interface. The internal workings of `hunlex` are as follows.

As the morphological grammar is parsed by the precompiler, rule objects are created. A block is read and parsed into functions which each transform the 'affix-rule' data-structure by enriching its internal representation according to the semantic content of the block. At the end of each unit, the empty rule is passed to the composition of block functions to result in a specific rule. Thanks to OCaml's flexibility of function abstraction and composition, this design makes it easy to implement macros of arbitrary blocks directly as functions. When the grammar is parsed, rules are arranged in a directed (possibly cyclic) graph with edges representing possible rule applications as given by the output specifications.

Precompilation proceeds by performing a recursive closure on this graph starting from lexical nodes. Rules are indexed by 'levels' and contiguous rule-nodes that are on the same level are merged along the edges if constraints on rule application (feature and match conditions, etc.) are satisfied. These precompiled affix-clusters and complex lexical items are to be placed in the `aff` and `dic` file, respectively.

Instead of affix merging, closure between rules *a* and *b* on different levels causes the affix clusters in the closure of *b* to be registered as rules in a hash and their indexes recorded on *a*. After the entire lexicon is read, these index sets registered on rules are considered. The affix cluster rules to be output into the affix file are arranged into maximal subsets such that if two output affix cluster rules *a* and *b* are in the same set, then every item or affix to which *a* can be applied, *b* can also be applied. These sets of affix clusters correspond to partial paradigms which each full paradigm either includes or is disjoint with. The resulting sets of output rules are assigned to a flag and items referencing them will specify the appropriate combination of flags in the output `dic` and `aff` file. Since equivalent affix cluster rules are conflated, the compiled resources are always optimal in the following three ways.

First, the affix file is *redundancy free*: no two affix rules have the same form. With hand-coded affix files this can almost never be guaranteed since one is always inclined to group affix rules by linguistically motivated paradigms thereby possibly duplicating entries. A redundancy-free set of affix rules will enhance performance by minimizing the search space for affixes. Note that conflation of identical rules by the runtime layer is not possible without reindexing the flags which would be very computationally intensive if done at runtime.

Second, given the redundancy free affix-set, maximizing homogeneous rulesets assigned to a flag *minimizes the number of flags* used. Since the internal representation of flags depends on their number, this has the practical advantage of reducing memory requirements for the runtime layer.

Third, identity of output affix rules is calculated relative to mode and configuration settings, therefore *identical morphs* with different morphological tags *will be conflated* for recognizers (spellchecking) where ambiguity is irrelevant, while for analysis it can be kept apart. This is impossible to achieve without a precompilation stage. Note that finite state transducer-based systems perform essentially the same type of optimizations, eliminating symbol redundancy when two symbols behave the same in every rule, and eliminating state redundancy when two states have the exact same continuations.

Though the bulk of the knowledge used by spellcheckers, by stemmers, and by morphological analysis and generation tools is shared (how affixes combine with stems, what words allow compounding), the ideal resources for these various tasks differ to some extent. Spellcheckers are meant to help one to conform to orthographic norms and therefore should be error sensitive, stemmers and morphological analyzers are expected to be more robust and error tolerant especially towards common violations of standard use. Although this seems at first to justify the individual efforts one has to invest in tailoring one's resources to the task at hand, most of the resource specifics are systematic, and therefore allow for automatic fine-tuning from a central knowledge base. Configuration within `hunlex` allows the specification of various features, among others:

- selection of registers and degree of normativity based on usage qualifiers in the database (allows for boosting robustness for analysis or stick to normativity for synthesis and spellchecking)
- flexible selection of output information: choice of tagset for different encodings, support for sense indexes
- arbitrary selection of morphemes
- setting levels of morphemes (grouping of morphs that are precompiled as a cluster to be stripped with one rule application by the runtime layer)
- fine-tuning which morphemes are stripped during stemming
- arbitrary selection of morphophonological features that are to be observed or ignored (allows for enhancing robustness by e.g., tolerating non-standard regularizations)

The input description language allows for arbitrary attributes (ones encoding part of speech, origin, register, etc.) to be specified in the description. Since any set of attributes can be selected to be compiled into the runtime resources, it takes no more than precompiling the central database with the appropriate configuration for the runtime analyzer to be used as an arbitrary word annotation tool, e.g., style annotator or part of speech tagger. We also provide an implementation of a feature-tree based tag language which we successfully used for the description of Hungarian morphology.

If the resources are created for some filtering task, say, extracting (possibly inflected) proper nouns in a text, resource optimization described above can save considerable amounts of time compared to full analysis followed by post-processing. While the relevant portion of the dictionary might be easily filtered therefore speeding up lookup, tailoring a corresponding redundancy-free affix file would be a hopeless enterprise without the precompiler.

As we mentioned, our offline layer can be configured to cluster any or no sets of affixes together on various levels, and therefore resources can be optimized for either memory use (affix by affix stripping) or speed (generally toward one level stripping). This is a major advantage given potential applications as diverse as spellchecking on the word processor of an old 386 at one end, and industrial scale stemming on terabytes of web content for IR at the other.

In sum, our offline layer allows for the principled maintenance of a central resource, saving the redundant effort that would otherwise have to be invested in encoding very similar knowledge in a task-specific manner for each word level analysis task.

## 3 Conclusion

The importance of word level analysis can hardly be questioned: spellcheckers reach the extremely wide audience of all word processor users, stemmers are used in a variety of areas ranging from information retrieval to statistical machine translation, and for non-isolating languages morphological analysis is the initial phase of every natural language processing pipeline.

Over the past decades, two closely intertwined methods emerged to handle word analysis tasks, affix stripping and finite state transducers (FSTs). Since both technologies can provide industrial strength solutions for most tasks, when it comes to choice of actual software and its practical use, the differences that have the greatest impact are not lodged in the algorithmic core. Rather, two other factors play a role: the ease with which one can integrate the software into applications and the infrastructure offered to translate the knowledge of the grammarian to efficient and maintainable computational blocks.

To be sure, in an end-to-end machine learning paradigm, the mundane differences between how the systems interact with the human grammarians would not matter. But as long as the grammars are written and maintained by humans, an offline framework providing a high-level language to specify morphologies and supporting configurable precompilation that allows for resource sharing across word-analysis tasks addresses a major bottleneck in resource creation and management.

The Xerox Finite State Toolkit provides comprehensive high-level support for morphology and lexicon development (Beesley and Karttunen, 2003). These descriptions are compiled into minimal deterministic FST-s, which give excellent runtime performance and can also be extended to error-tolerant analysis for spellchecking Oflazer (1996). Nonetheless, XFST is not free software, and as long as the work is not driven by academic curiosity alone, the LGPL-style license of our toolkit, explicitly permitting reuse for commercial purposes as well, can already decide the choice.

There are other free open source analyzer technologies, either stand-alone analyzers such as the Stuttgart Finite State Toolkit (SFST, available only under the GPL, see `www.ims.uni-stuttgart.de/projekte/gramotron/SOFTWARE/SFST.html`, Smid et al. (2004)) or as part of a powerful integrated NLP platform such as Intex/NooJ (freely available for academic research to individuals affiliated with a university only, see `intex.univ-fcomte.fr`; a clone called Unitex is available under LGPL, see `www-igm.univ-mlv.fr/~unitex`.) Unfortunately, NooJ has its limitations when it comes to implementing complex morphologies (Vajda et al., 2004) and SFST provides no high-level offline component for grammar description and configurable resource creation.

We believe that the liberal license policy and the powerful offline layer contributed equally to the huge interest that our project generated, in spite of its relative novelty. `MySpell` was not just our choice: it is also the spell-checking library incorporated into OpenOffice.org, a free open-source office suite with an ever wider circle of users. The Hungarian build of OpenOffice is already running our C++ runtime library, but OpenOffice is now considering to completely replace `MySpell` with our code. This would open up the possibility of introducing morphological analysis capabilities in the program, which in turn could serve as the first step towards enhanced grammar checking and hyphenation.

Though in-depth grammars and lexica are available for nearly as many languages in FST-based frameworks (InXight Corporation's LinguistX platform supports 31 languages), very little of this material is available for grammar hacking or open source dictionary development. In addition to permissive license and easy to integrate infrastructure, the fact that the `hunmorph` routines are backward compatible with already existing and freely available spellchecking resources for some 50 languages goes a long way toward explaining its rapid spread.

For Hungarian, `hunlex` already serves as the development framework for the MORPHDB project which merges three independently developed lexical databases by critically unifying their contents and supplying it with a comprehensive morphological grammar. It also provided a framework for our English morphology project that used the XTAG morphological database for English (see `ftp.cis.upenn.edu/pub/xtag/morph-1.5`, Karp et al. (1992)). A project describing the morphology of the Beás dialect of Romani with `hunlex` is also under way.

The `hunlex` resource precompiler is not architecturally bound to the `aff/dic` format used by our toolkit, and we are investigating the possibility of generating FST resources with it. This would decouple the offline layer of our toolkit from the details of the runtime technology, and would be an important step towards a unified open source solution for method-independent resource development for word analysis software.

## Acknowledgements

The development of `hunmorph` and `hunlex` is financially supported by the Hungarian Ministry of Informatics and Telecommunications and by MATÁV Telecom Co., and is led by the Centre for Media Research and Education at the Budapest University of Technology. We would like to thank the anonymous reviewers of the ACL Software Workshop for their valuable comments on this paper.

## Availability

Due to the confligting needs of Unix, Windows, and MacOs users, the packaging/build environment for our software has not yet been finalized. However, a version of our tools, and some of the major language resources that have been created using these tools, are available at `mokk.bme.hu/resouces`.

## References

Kenneth R. Beesley and Lauri Karttunen. 2003. *Finite State Morphology*. CSLI Publications.

Daniel Karp, Yves Schabes, Martin Zaidel, and Dania Egedi. 1992. A freely available wide coverage morphological analyzer for english. In *Proceedings of the 14th International Conference on Computational Linguistics (COLING-92) Nantes, France.*

László Németh, Viktor Trón, Péter Halácsy, András Kornai, András Rung, and István Szakadát. 2004. Leveraging the open-source ispell codebase for minority language analysis. In *Proceedings of SALTMIL 2004*. European Language Resources Association. URL `http://www.lrec-conf.org/lrec2004`.

Kemal Oflazer. 1996. Error-tolerant finite-state recognition with applications to morphological analysis and spelling correction. *Computational Linguistics*, 22(1):73–89.

James Lyle Peterson. 1980. *Computer programs for spelling correction: an experiment in program design*, volume 96 of *Lecture Notes in Computer Science*. Springer.

Helmut Smid, Arne Fitschen, and Ulrich Heid. 2004. SMOR: A German computational morphology covering derivation, composition, and inflection. In *Proceedings of the IVth International Conference on Language Resources and Evaluation (LREC 2004)*, pages 1263–1266.

Péter Vajda, Viktor Nagy, and Emília Dancsecs. 2004. A Ragozási szótártól a NooJ morfológiai moduljáig [from a morphological dictionary to a morphological module for NooJ]. In *2nd Hungarian Computational Linguistics Conference*, pages 183–190.

# Scaling High-Order Character Language Models to Gigabytes

**Bob Carpenter**
Alias-i, Inc.
181 North 11th St., #401, Brooklyn, NY 11211
carp@colloquial.com

## Abstract

We describe the implementation steps required to scale high-order character language models to gigabytes of training data without pruning. Our online models build character-level PAT trie structures on the fly using heavily data-unfolded implementations of an mutable daughter maps with a long integer count interface. Terminal nodes are shared. Character 8-gram training runs at 200,000 characters per second and allows online tuning of hyperparameters. Our compiled models precompute all probability estimates for observed n-grams and all interpolation parameters, along with suffix pointers to speedup context computations from proportional to n-gram length to a constant. The result is compiled models that are larger than the training models, but execute at 2 million characters per second on a desktop PC. Cross-entropy on held-out data shows these models to be state of the art in terms of performance.

## 1 Introduction

Character *n*-gram language models have been applied to just about every problem amenable to statistical language modeling. The implementation we describe here has been integrated as the source model in a general noisy-channel decoder (with applications to spelling correction, tokenization and case normalization) and the class models for statistical classification (with applications including spam filtering, topic categorization, sentiment analysis and word-sense disambiguation). In addition to these human language tasks, *n*-grams are also popular as estimators for entropy-based compression and source models for cryptography. (Teahan, 2000) and (Peng, 2003) contain excellent overviews of character-level models and their application from a compression and HMM perspective, respectively.

Our hypothesis was that language-model smoothing would behave very much like the classifiers explored in (Banko and Brill, 2001), in that more data trumps better estimation technique. We managed to show that the better of the interpolation models used in (Chen and Goodman, 1996), namely Dirichlet smoothing with or without update exclusion, Witten-Bell smoothing with or without update exclusion, and absolute discounting with update exclusion converged for 8-grams after 1 billion characters to cross entropies of 1.43+/-0.01. The absolute discounting with update exclusion is what Chen and Goodman refer to as the Kneser-Ney method, and it was the clear winner in their evaluation. They only tested non-parametric Witten-Bell with a suboptimal hyperparameter setting (1.0, just as in Witten and Bell's original implementation). After a billion characters, roughly 95 percent of the characters were being estimated from their highest-order (7) context. The two best models, parametric Witten-Bell and absolute discounting with update exclusion (aka Kneser-Ney), were even closer in cross-entropy, and depending on the precise sample (we kept rolling samples as described below), and after a million or so characters, the differences even at the higher variance 12-grams were typically in the +/-0.01 range. With a roughly 2.0 bit/character deviation, a 10,000 character sample, which is the size we used, leads to a $2\sigma$ (95.45%) confidence interval of +/-0.02, and the conclusion that the differences between these systems was insignificant.

Unlike in the token-based setting, we are not optimistic about the possibility of improving these results dramatically by clustering character contexts. The lower-order models are very well trained with existing quantities of data and do a good job of this kind of smoothing. We do believe that training hyperparameters for different model orders independently might improve cross-entropy fractionally; we found that training them hierarchically, as in (Samuelsson, 1996), actually increased cross-entropy. We believe this is a direct correlate of the effectiveness of update exclusion; the lower-order models do not need to be the best possible models of those orders, but need to provide good estimates when heavily weighted, as in smoothing. The global optimization allows a single setting to balance these attributes, but optimizing each dimension individually should do even better. But with the number of estimates taking place at the highest possible orders, we do not believe the amount of smoothing will have that large an impact overall.

These experiments had a practical goal — we needed to choose a language modeling implementation for LingPipe and we didn't want to take the standard Swiss Army Knife approach because most of our users are not interested in running experiments on language modeling, but rather using language models in applications such as information retrieval, classification, or clustering. These applications have actually been shown to perform better on the basis of character language models than token models ((Peng, 2003)). In addition, character-level models require no decisions about tokenization, token normalization and subtoken modeling (as in (Klein et al., 2003)).

We chose to include the Witten-Bell method in our language modeling API because it is derived from full corpus counts, which we also use for collocation and relative frequency statistics within and across corpora, and thus the overall implementation effort was simpler. For just language modeling, an update exclusion implementation of Kneser-Ney is no more complicated than Witten-Bell.

In this paper, we describe the implementation details behind storing the model counts, how we sample the training character stream to provide low-cost, online leave-one-out style hyperparameter estimation, and how we compile the models and evaluate them over text inputs to achieve linear performance that is nearly independent of *n*-gram length. We also describe some of the design patterns used at the interface level for training and execution. As far as we know, the online leave-one-out analysis is novel, though there are epoch-based precursors in the compression literature.

As far as we know, no one has built a character language model implementation that will come close to the one presented here in terms of scalability. This is largely because they have not been designed for the task rather than any fundamental limitation. In fact, we take the main contribution of this paper to be a presentation of simple data sharing and data unfolding techniques that would also apply to token-level language models. Before starting our presentation, we'll review some of the limitations of existing systems. For a start, none of the systems of which we are aware can scale to 64-bit values for counts, which is necessary for the size models we are considering without pruning or count scaling. It's simply easier to find 4 billion instances of a character than of a token. In fact, the compression models typically use 16 bits for storing counts and then just scale downward when necessary, thus not even trying to store a full set of counts for even modest corpora. The standard implementations of character models in the compression literature represent ordinary trie nodes as arrays, which is hugely wasteful for large sparse implementations; they represent PAT-trie nodes as pointers into the original text plus counts, which works well for long *n*-gram lengths (32) over small data sets (1 MB) but does not scale well for reasonable *n*-gram lengths (8-12) over larger data sets (100MB-1GB). The standard token-level language models used to restrict attention to 64K tokens and thus require 16-bit token representatives per node just as our character-based approach; with the advent of large vocabulary speech recognition, they now typically use 32-bits per node just to represent the token. Arrays of daughter nodes and lack of sharing of low-count terminal nodes were the biggest space hogs in our experiments, and as far as we know, none of the standard approaches take the immutable data unfolding approach we adopt to eliminate this overhead. Thus we would like to stress again that existing character-level compression and token-level language modeling systems were simply not designed for handling large character-level models.

We would also like to point out that the standard finite state machine implementations of language models do not save any space over the trie-based implementations, typically only approximate smoothing using backoff rather than interpolation, and further suffer from a huge space explosion when determinized. The main advantage of finite state approaches is at the interface level in that they work well with hand-written constraints and can interface on either side of a given modeling problem. For instance, typical language models implemented as trivial finite state transducers interface neatly with triphone acoustic models on the one side and with syntactic grammars on the other. When placed in that context, the constraints from the grammar can often create an overall win in space after composition.

## 2 Online Character Language Models

For generality, we use the 16-bit subset of unicode as provided by Java 1.4.2 to represent characters. This presents an additional scaling problem compared to ASCII or Latin1, which fit in 7 and 8 bits.

Formally, if $\mathsf{Char}$ is a set of characters, a *language model* is defined to be a mapping $P$ from the set $\mathsf{Char}^*$ of character sequences into non-negative real numbers. A *process* language model is normalized over sequences of length $n$: $\sum_{X \in \mathsf{Char}^*, |X|=n} P(X) = 1.0$. We also implement bounded language models which normalize over all sequences, but their implementation is close enough to the process models that we do not discuss them further here. The basic interfaces are provided in Figure 1 (with names shortened to preserve space). Note that the process and sequence distribution is represented through marker interfaces, whereas the cross-cutting dynamic language models support training and compilation, as well as the estimation inherited from the language model interface.

```
interface LM {
    double log2Prob(char[] cs,
        int start, int end);
}
interface ProcessLM extends LM {
}
interface SequenceLM extends LM {
}
interface DynamicLM extends LM {
    double train(char[] cs,
        int start, int end);
    void compile(ObjectOutput out)
        throws IOException;
}
```

Figure 1: Language Model Interface

We now turn to the statistics behind character-level langauge models. The chain rule factors $P(x_0, \ldots, x_{k-1}) = \prod_{i<k} P(x_i | x_0, \ldots, x_{i-1})$. An $n$-gram language model estimates a character using only the last $n-1$ symbols, $\hat{P}(x_k | x_0, \ldots, x_{k-1}) = \hat{P}(x_k | x_{k-n+1}, \ldots, x_{k-1})$; we follow convention in denoting generic estimators by $\hat{P}$.

The maximum likelihood estimator for $n$-grams is derived from frequency counts for sequence $X$ and symbol $c$, $P_{\mathrm{ML}}(c|X) = \mathsf{count}(Xc)/\mathsf{extCount}(X)$, where $\mathsf{count}(X)$ is the number of times the sequence $X$ was observed in the training data and $\mathsf{extCount}(X)$ is the number of single-symbol extensions of $X$ observed: $\mathsf{extCount}(X) = \sum_{c \in \mathsf{Char}} \mathsf{count}(Xc)$. When training over one or more short samples, the disparity between $\mathsf{count}(X)$ and $\mathsf{extCount}(X)$ can be large: for *abracadabra*, $\mathsf{count}(a) = 5$, $\mathsf{count}(bra) = 2$, $\mathsf{extCount}(a) = 4$, and $\mathsf{extCount}(bra) = 1$.

We actually provide two implementations of language models as part of LingPipe. For language models as random processes, there is no padding. They correspond to normalizing over sequences of a given length in that the sum of probabilities for character sequences of length $k$ will sum to 1.0. With a model that inserts begin-of-sequence and end-of-sequence characters and estimates only the end-of-sequence character, normalization is over all strings. Statistically, these are very different models. In practice, they are only going to be distinguishable if the boundaries are very significant and the total string length is relatively small. For instance, they are not going to make much difference in estimating probabilities of abstracts of 1000 characters, even though the start and ends are significant (e.g. capitals versus punctuation being preferred at beginning and end of abstracts) because cross-entropy will be dominated by the other 1000 characters. On the other hand, for modeling words, for instance as a smoothing step for token-level part-of-speech, named-entity or language models, the begin/end of a word will be significant, representing capitalization, prefixes and suffixes in a language. In fact, this latter motivation is why we provide padded models. It is straightforward to implement the padded models on top of the process models, which is why we discuss the process models here. But note that we do not pad all the way to maximum *n*-gram length, as that would bias the begin/end statistics for short words.

We use linear interpolation to form a mixture model of all orders of maximum likelihood estimates down to the uniform estimate $P_U(c) = 1/|\mathsf{Char}|$. The interpolation ratio $\lambda(dX)$ ranges between 0 and 1 depending on the context $dX$.

$$\begin{aligned}\hat{P}(c|dX) &= \lambda(dX)P_{\mathrm{ML}}(c|dX) \\ &+ (1-\lambda(dX))\hat{P}(c|X) \\ \hat{P}(c) &= \lambda()P_{\mathrm{ML}}(c) \\ &+ (1-\lambda())(1/|\mathsf{Char}|)\end{aligned}$$

The Witten-Bell estimator computed the interpolation parameter $\lambda(X)$ using only overall training counts. The best performing model that we evaluated is parameterized Witten-Bell interpolation with hyperparameter $K$, for which the interpolation ratio is defined to be:

$$\lambda(X) = \frac{\mathsf{extCount}(X)}{\mathsf{extCount}(X) + K \cdot \mathsf{numExts}(X)}$$

We take $\mathsf{numExts}(X) = |\{c|\mathsf{count}(Xc) > 0\}|$ to be the number of different symbols observed following the sequence $X$ in the training data. The original Witten-Bell estimator set $K = 1$. We optimize the hyperparameter $K$ online (see the next section).

## 3 Online Models and Hyperparameter Estimation

A language model is *online* if it can be estimated from symbols as they arrive. An advantage of online models is that they are easy to use for adaptation to

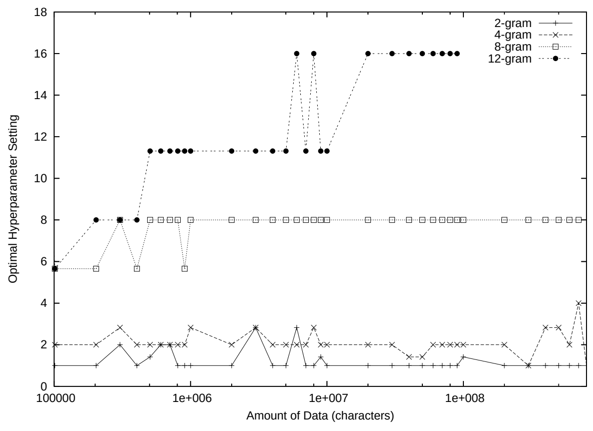


Figure 2: Optimal Hyperparameter Settings for Witten-Bell

documents or styles, hence their inclusion in commercial dictation packages such as DragonDictate and ViaVoice. Another advantage is that they are easy to integrate into tag-a-little/learn-a-little systems such as MITRE's Alembic Workbench.

With online models, we are able to estimate hyperparameters using an online form of leave-one-out analysis (Ney et al., 1995). This can be performed in a number of ways as long as the model efficiently estimates likelihoods given a set of hyperparameter settings. We opted for the simplest technique we could muster to find the right settings. This was made easier because we only have a single hyperparameter whose behavior is fairly flat around the optimal setting and because the optimal setting didn't change quickly with increasing data. The optimal settings are shown in Figure 2. Also note that the optimal value is rarely at 1 except for very low-order *n*-grams. To save the complexity of maintaining an interval around the best estimate do do true hill climbing, we simply kept rolling averages of values logarithmically spaced from 1/4 to 32. We also implemented a training method that kept track of the last 10,000 character estimates (made before the characters were used for training, of course). We used a circular queue for this data structure because its size is fixed and it allowed a constant time insert of the last recorded value. We used one circular queue for each hyperparameter setting, thus storing around 5MB or so worth of samples. These samples can be used to provide an estimate of the best hyperparameter at any given point in the algorithm's execution. We used this explicit method rather than the much less costly rolling average method so that results would be easier to report. We actually believe just keeping a rolling average of measured cross-entropies on online held-out samples is sufficient.

We also sampled the character stream rather than estimating each character before training. With a gigabyte of characters, we only needed to sample 1 in 100,000 characters to find enough data for estimates. At this rate, online hyperparameter estimate did not measurably affect training time, which was dominated by simply constructing the trie.

We only estimated a single hyperparameter rather than one for each order to avoid having to solve a multivariate estimation problem; although we can collect the data online, we would either have to implement an EM-like solution or spend a lot time per estimate iterating to find optimal parameters. This may be worthwhile for cases where less data is available. As the training data increased, the sensitivity to training parameters decreased. Counterintuitively, we found that recursively estimating each order from low to high, as implemented in (Samuelsson, 1996), actually increased entropy considerably. Clearly the estimator is using the fact that lower-order estimates should not necessarily be optimal for use on their own. This is a running theme of the discounting methods of smoothing such as absolute discounting or Kneser-Ney.

Rather than computing each estimate for hyperparameter and *n*-gram length separately, we first gather the counts for each suffix and each context and the number of outcomes for that context. This is the expensive step, as it require looking up counts in the trie structure. Extension counts require a loop over all the daughters of a context node in the trie because we did not have enough space to store them on nodes. With all of these counts, the *n*-gram etimates for each *n* and each hyperparameter setting can be computed from shortest to longest, with the lower order estimates contributing the smoothed estimate for the next higher order.

## 4 Substring Counters

Our *n*-gram language models derive estimates from counts of substrings of length *n* or less in the training corpus. Our counter implementation was the trickiest component to scale as it essentially holds the statistics derived from the training data. It contains statistics sufficient to implement all of the estimators defined above. The only non-trivial case is Kneser-Ney, which is typically implemented using the technique known in the compression literature as "update exclusion" (Moffat, 1990). Under update exclusion, if a count "abc" is updated and the context "ab" was known, then counts for "a" and "ab" are excluded from the update process. We actually compute these counts from the total counts by noting that the update exclusion count is equal to the number of unique characters found following a shorter context. That is, the count for "ab" for smoothing is equal to the number of characters "x" such that "xab" has a non-zero count, because these are the situations in which the count of "ab" is not excluded. This is not an efficient way to implement update exclusion, but merely an expedient so we could share implementations for experimental purposes. Straight update exclusion is actually more efficient to implement than full counts, but we wanted the full set of character substring counts for other purposes, as well as language modeling.

```
interface Node {
    Node increment(char[] cs,
                   int start, int end);
    long count(char[] cs,
               int start, int end);
    long extCount(char[] cs,
                  int start, int end);
    int numExts(char[] cs,
                int start, int end);
    Node prune(long minCount);
}
```

Figure 3: Trie Node Interface

Our implementation relies heavily on a data unfolded object-oriented implementation of Patricia tries. Unlike the standard suffix tree algorithms for constructing this trie for all substrings as in (Cleary and Teahan, 1997), we limit the length and make copies of characters rather than pointing back into the original source. This is more space efficient than the suffix-tree approach for our data set sizes and *n*-gram lengths.

The basic node interface is as shown in Figure 3. Note that the interface is in terms of long integer values. This was necessary to avoid integer overflow in our root count when data size exceeded 2 GB and our 1-gram counts when data sizes exceeded 5 or 6GB. A widely used alternative used for compression is to just scale all the counts by dividing by two (and typically pruning those that go to zero); this allows PPM to use 8-bit counters at the cost of arithmetic precision ((Moffat, 1990)). We eschew pruning because we also use the counts to find significant collocations. Although most collocation and significance statistics are not affected by global scaling, cross-entropy suffers tremendously if scaling is done globally rather than only on the nodes that need it.

Next note that the interface is defined in terms of indexed character slices. This obviates a huge amount of otherwise unnecessary object creation and garbage collection. It is simply not efficient enough, even with the newer generational garbage collectors, to create strings or even lighter character sequences where needed on the heap; slice indices can be maintained in local variables.

The `increment` method increments the count for each prefix of the specified character slice. The `count` method returns the count of a given character sequence, `extensionCount` the count of all one-character extensions, `numExtensions` the number of extensions. The `extensions` method returns all the observed extensions of a character sequence, which is useful for enumerating over all the nodes in the trie.

Global pruning is implemented, but was not necessary for our scalability experiments. It *is* necessary for compilation; we could not compile models nearly as large as those kept online. Just the size of the floating point numbers (two per node for estimate and interpolation) lead to 8 bytes per node. In just about every study every undertaken, including our informal ones, unpruned models have outperformed pruned ones. Unfortunately, applications will typically not have a gigabyte of memory available for models. The best performing models for a given size are those trained on as much data available and pruned to the specified size. Our pruning is simply a minimum count approach, because the other methods have not been shown to improve much on this baseline.

Finally, note that both the increment and prune methods return nodes themselves. This is to support the key implementation technique for scalability – replacing immutable objects during increments. Rather than having a fixed mutable node representation, nodes can return results that are essentially replacements for themselves. For instance, there is an implementation of `Node` that provides a count as a byte (8 bits) and a single daughter. If that class gets incremented above the byte range, it returns a node with a short-based counter (16 bits) and a daughter that's the result of incrementing the daughter. If the class gets incremented for a different daughter path, then it returns a two-daughter implementation. Of course, both of these can happen, with a new daughter that pushes counts beyond the byte range. This strategy may be familiar to readers with experience in Prolog (O'Keefe, 1990) or Lisp (Norvig, 1991), where many standard algorithms are implemented this way.

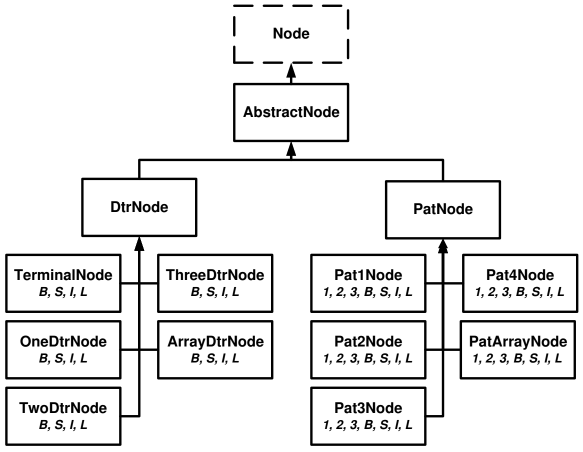

Node
AbstractNode
DtrNode
PatNode
TerminalNode B, S, I, L
ThreeDtrNode B, S, I, L
OneDtrNode B, S, I, L
ArrayDtrNode B, S, I, L
TwoDtrNode B, S, I, L
Pat1Node 1, 2, 3, B, S, I, L
Pat4Node 1, 2, 3, B, S, I, L
Pat2Node 1, 2, 3, B, S, I, L
PatArrayNode 1, 2, 3, B, S, I, L
Pat3Node 1, 2, 3, B, S, I, L

Figure 4: Unfolded Trie Classes

A diagram of the implementations of `Node` is provided in Figure 4. At the top of the diagram is the `Node` interface itself. The other boxes all represent abstract classes, with the top class, `AbstractNode`, forming an abstract adapter for most of the utility methods in `Node` (which were not listed in the interface).

The abstract subclass `DtrNode` is used for nodes with zero or more daughters. It requires its extensions to return parallel arrays of daughters and characters and counts from which it implements all the update methods at a generic level.

```
abstract class TwoDtrNode
    extends DtrNode {

    final char mC1; final Node mDtr1
    final char mC2; final node mDtr2;

    TwoDtrNode(char c1, Node dtr1,
                      char c2, Node dtr2,
        mC1 = c1; mDtr1 = dtr1;
        mC2 = c2; mDtr2 = dtr2;
    }

    Node getDtr(char c) {
        return c == mC1
            ? mDaughter1
            : ( c == mC2
                ? mDaughter2
                : null );
    }

    [] chars() {
        return new char[] { mC1, mC2 };
    }

    Node[] dtrs() {
        return new Node[] { mDaughter1,
                             mDaughter2 };
    }

    int numDtrs() { return 2; }
}
```

Figure 5: Two Daughter Node Implementation

The subclass `TerminalNode` is used for nodes with no daughters. Its implementation is particularly simple because the extension count, the number of extensions and the count for any non-empty sequence starting at this node are zero. The nodes with non-empty daughters are not much more complex. For instance, the two-daughter node abstract class is shown in Figure 5.

All daughter nodes come with four concrete implementations, based on the size of storage allocated for counts: `byte` (8 bits), `short` (16 bits), `int` (32 bits), or `long` (64 bits). The space savings from only allocating bytes or shorts is huge. These concrete implementations do nothing more than return their own counts as long values. For instance, the `short` implementation of three-daughter nodes is shown in Figure 6. Note that because these nodes are not public, the factory can be guaranteed to only call the constructor with a count that can be cast to a short value and stored.

Increments are performed by the superclass

```
final class ThreeDtrNodeShort
    extends ThreeDtrNode {

    final short mCount;

    ThreeDtrNodeShort(char c1, Node dtr1,
                      char c2, Node dtr2,
                      char c3, Node dtr3,
                      long count) {
        super(c1,dtr1,c2,dtr2,c3,dtr3);
        mCount = (short) count;
    }

    long count() { return mCount; }
}
```

Figure 6: Three Daughter Short Node

and will call constructors of the appropriate size. The `increment` method as defined in `AbstractDtrNode` is given in Figure 7. This method increments all the suffixes of a string.

The first line just increments the local node if the array slice is empty; this involves taking its characters, its daughters and calling the factory with one plus its count to generate a new node. This generates a new immutable node. If the first character in the slice is an existing daughter, then the daughter is incremented and the result is used to increment the entire node. Note the assignment to `dtrs[k]` after the increment; this is to deal with the immutability. The majority of the code is just dealing with the case where a new daughter needs to be inserted. Of special note here is the factory instance called on the remaining slice; this will create a PAT node. This appears prohibitively expensive, but we refactored to this approach from a binary-tree based method with almost no noticeable hit in speed; most of the arrays stabilize after very few characters and the resizings of big arrays later on is quite rare. We even replaced the root node implementation which was formerly a map because it was not providing a measurable speed boost.

Once the daughter characters and daughters are marshalled, the factory calls the appropriate constructor based on the number of the character and daughters. The factory then just calls the appropriately sized constructor as shown in Figure 8.

Unlike other nodes, low count terminal nodes are stored in an array and reused. Thus if the result of an increment is within the cache bound, the stored

```
Node increment(char[] cs,
               int start, int end) {
  // empty slice; incr this node
  if (start == end)
    return NodeFactory
           .createNode(chars(),dtrs(),
                       count()+1l);
  char[] dtrCs = chars();
  // search for dtr
  int k = Arrays.binarySearch(dtrCs,
                              cs[start]);
  Node[] dtrs = dtrs();
  if (k >= 0) {  // found dtr
    dtrs[k] = dtrs[k]
              .increment(cs,start+1,end);
    return NodeFactory
           .createNode(dtrCs,dtrs,
                       count()+1l);
  }
  // insert new dtr
  char[] newCs = new char[dtrs.length+1];
  Node[] newDtrs = new Node[dtrs.length+1];
  int i = 0;
  for (; i < dtrs.length
         && dtrCs[i] < cs[start];
       ++i) {
    newCs[i] = dtrCs[i];
    newDtrs[i] = dtrs[i];
  }
  newCs[i] = cs[start];
  newDtrs[i] = NodeFactory
              .createNode(cs,start+1,
                          end,1);
  for (; i < dtrCs.length; ++i) {
      newCs[i+1] = dtrCs[i];
      newDtrs[i+1] = dtrs[i];
  }
  return NodeFactory
         .createNode(newCs,newDtrs,
                     count()+1l);
}
```

Figure 7: Increment in `AbstractDtrNode`

version is returned. Because terminal nodes are immutable, this does not cause problems with consistency. In practice, terminal nodes are far and away the most common type of node, and the greatest saving in space came from carefully coding terminal nodes.

The abstract class `PatNode` implements a so-called "Patricia" trie node, which has a single chain of descendants each of which has the same count. There are four fixed-length implementations for the one, two, three and four daughter case. For these implementations, the daughter characters are stored in member variables. For the array implementation, `PatArrayNode`, the daughter chain is stored

```
static Node createNode(char c, Node dtr,
                       long n) {
  if (n <= Byte.MAX_VALUE)
    return new OneDtrNodeByte(c,dtr,n);
  if (n <= Short.MAX_VALUE)
    return new OneDtrNodeShort(c,dtr,n);
  if (n <= Integer.MAX_VALUE)
    return new OneDtrNodeInt(c,dtr,n);
  return new OneDtrNodeLong(c,dtr,n);
}
```

Figure 8: One Daughter Factory Method

as an array. Like the generic daughter nodes, PAT nodes contain implementations for byte, short, int and long counters. They also contain constant implementations for one, two and three counts. We found in profiling that the majority of PAT nodes had counts below four. By providing constant implementations, no memory at all is used for the counts (other than a single static component per class). Pat nodes themselves are actually more common that regular daughter nodes in high-order character tries, because most long contexts are deterministic. As *n*-gram order increases, so does the proportion of PAT nodes. Implementing increments for PAT nodes is only done once in the abstract class `PatNode`. Each PAT node implementation supplied an array in a standardized interface to the implementations in `PatNode`. That array is created as needed and only lives long enough to carry out the required increment or lookup. Java's new generational garbage collector is fairly efficient at dealing with garbage collection for short-lived objects such as the trie nodes.

## 5 Compilation

Our online models are tuned primarily for scalability, and secondarily for speed of substring counts. Even the simplest model, Witten-Bell, requires for each context length that exists, summing over extension counts and doing arithmetic including several divisions and multiplications per order a logarithm at the end. Thus straightforward estimation from models is unsuitable for static, high throughput applications. Instead, models may be compiled to a less compact but more efficient static representation.

We number trie nodes breadth-first in unicode order beginning from the root and use this indexing for four parallel arrays following (Whittaker and Raj, 2001). The main difference is that we have not

|  |  | char | int | float | float | int |
|---|---|---|---|---|---|---|
| Idx | Ctx | C | Suf | $\log P$ | $log(1\text{-}\lambda)$ | Dtr |
| 0 | n/a | n/a | n/a | n/a | -0.63 | 1 |
| 1 |  | a | 0 | -2.60 | -0.41 | 6 |
| 2 |  | b | 0 | -3.89 | -0.58 | 9 |
| 3 |  | c | 0 | -4.84 | -0.32 | 10 |
| 4 |  | d | 0 | -4.84 | -0.32 | 11 |
| 5 |  | r | 0 | -3.89 | -0.58 | 12 |
| 6 | a | b | 2 | -2.51 | -0.58 | 13 |
| 7 | a | c | 3 | -3.49 | -0.32 | 14 |
| 8 | a | d | 4 | -3.49 | -0.32 | 15 |
| 9 | b | r | 5 | -1.40 | -0.58 | 16 |
| 10 | c | a | 1 | -1.59 | -0.32 | 17 |
| 11 | d | a | 1 | -1.59 | -0.32 | 18 |
| 12 | r | a | 1 | -1.17 | -0.32 | 19 |
| 13 | ab | r | 9 | -0.77 | n/a | n/a |
| 14 | ac | a | 10 | -1.10 | n/a | n/a |
| 15 | ad | a | 11 | -1.10 | n/a | n/a |
| 16 | br | a | 12 | -0.67 | n/a | n/a |
| 17 | ca | d | 8 | -1.88 | n/a | n/a |
| 18 | da | b | 6 | -1.55 | n/a | n/a |
| 19 | ra | c | 7 | -1.88 | n/a | n/a |

Figure 9: Compiled Representation of 3-grams for "abracadabra"

coded to a fixed *n*-gram length, costing us a bit of space in general, and also that we included context suffix pointers, costing us more space but saving lookups for all suffixes during smoothing.

The arrays are (1) the character leading to the node, (2) the log estimate of the last character in the path of characters leading to this node given the previous characters in the path, (3) the log of one minus the interpolation parameter for the context represented by the full path of characters leading to this node, (4) the index of the first daughter of the node, and (5) index of the suffix of this node. Note that the daughters of a given node will be contiguous and in unicode order given the breadth-first nature of the indexing, ranging from the daughter index of the node to the daughter index of the next node.

We show the full set of parallel arrays for trigram counts for the string "abracadabra" in Figure 9. The first column is for the array index, and is not explicitly represented. The second column, labeled "Ctx", is the context, and this is also not explicitly represented. The remaining columns are explicitly represented. The third column is for the character. The fourth column is an integer backoff suffix pointer; for instance, in the row with index 13, the context is "ab", and the character is "r", meaning it represents "abr" in the trie. The suffix index is 9, which is for "br", the suffix of "abr". The fifth and sixth columns are 32-bit floating point estimates, the fifth of $\log_2 P(r|ab)$, and the sixth is empty because there is no context for "abr", just an outcome. The value of $\log_2(1-\lambda(ab))$ is found in the row indexed 6, and equal to -0.58. The seventh and final column is the integer index of the first daughter of a given node. The value of the daughter pointer for the following node provides an upper bound. For instance, in the row index 1 for the string "a", the daughter index is 6, and the next row's daughter index is 9, thus the daughters of "a" fall between 6 and 8 inclusively — these are "ab", "ac" and "ad" respectively. Note that the daughter characters are always in alphabetical order, allowing for a binary search for daughters.

For *n*-gram estimators, we need to compute $\log P(c_n|c_0\cdots c_{n-1})$. We start with the longest sequence $c_k,\ldots,c_{n-1}$ that exists in the trie. If binary search finds the outcome $c_n$ among the daughters of this node, we return its log probability estimate; this happens in over 90 percent of estimates with reasonably sized training sets. If the outcome character is not found, we continue with shorter and shorter contexts, adding log interpolation values from the context nodes until we find the result or reach the uniform estimate at the root, at which point we add its estimate and return it. For instance, the estimate of $\log_2 P(r|ab) = -0.77$ can be read directly off the row indexed 13 in Figure 9. But $\log_2 P(a|ab) = -0.58+\log_2 P(a|b) = -0.58+-0.58+\log_2 P(a) = -0.58+-0.58+-2.60$, requiring two interpolation steps.

For implementation purposes, it is significant that we keep track of where we backed off from. The row for "a", where the final estimate was made, will be the starting point for lookup next time. This is the main property of the fast string algorithms — we know that the context "ba" does not exist, so we do not need to go back to the root and start our search all over again at the next character. The result is a linear bound on lookup time because each backoff of *n* characters guarantees at least *n* steps to get back to the same context length, thus there can't be more backoff steps than characters input. The main bottleneck in run time is memory bandwidth due to cache misses.

The log estimates can be compressed using as much precision as needed (Whittaker and Raj, 2001), or even reduced to integral values and integer arithmetic used for computing log estimates. We use floats and full characters for simplicity and speed.

## 6 Corpora and Parsers

Our first corpus is 7 billion characters from the *New York Times* section of the Linguistic Data Consortium's Gigaword corpus. Only the body of documents of type story were used. Paragraphs indicated by XML markup were begun with a single tab character. All newlines were converted to single whitepspaces, and all other data was left unmodified. The data is problematic in at least two ways. First, the document set includes repeats of earlier documents. Language models provide a good way of filtering these repeated documents out, but we did not do so for our measurements because there were few enough of them that it made little difference and we wanted to simplify other comparative evaluations. Second, the document set includes numerical list data with formatting such as stock market reports. The *Times* data uses 87 ASCII characters.

Our second corpus is the 5 billion characters drawn from abstracts in the United States' National Library of Medicine's 2004 MEDLINE baseline citation set. Abstract truncation markers were removed. MEDLINE uses a larger character set of 161 characters, primarily extending ASCII with diacritics on names and Greek letters.

By comparison, (Banko and Brill, 2001) used one billion tokens for a disambiguation task, (Brown et al., 1991) used 583 million tokens for a language model task, and (Chen and Goodman, 1996) cleverly sampled from 250 million tokens to evaluate higher-order models by only training on sequences used in the held-out and test sets.

Our implementation is based a generic text parser and text handler interface, much like a simplified version of XML's SAX content handler and XML parser. A text parser is implemented for the various data sets, including decompressing their zipped and gzipped forms and parsing their XML, SGML and tokenized form. A handler is then implemented that adds data to the online models and polls the model for results intermittently for generating graphs.

## 7 Results

We used a 1.4GB Java heap (unfortunately, the maximum allowable with Java on 32-bit Intel hardware without taking drastic measures), which allowed us to train 6-grams on up to 7 billion characters with room to spare. Roughly, 8-grams ran out of memory at 1 billion characters, 12 grams at 100 million characters, and 32 grams at 10 million characters. We did not experiment with pruning for this paper, though our API supports both thresholded and pdivisive scaling pruning. Training the counters depends heavily on the length of *n*-gram, with 5-grams training at 431,000 characters per second, 8-grams at 204,000 char/s, 12-grams at 88,000 char/s and 32-grams at 46,000 char/s, including online hyperparameter estimation (using a $2000 PC running Windows XP and Sun's 1.4.2 JDK, with a 3.0GHz Pentium 4, 2GB of ECC memory at 800MHz, and two 10K SATA drives in RAID 0).

Our primary results are displayed in Figure 11 and Figure 10, which plot sample cross-entropy rates against amount of text used to build the models for various *n*-gram lengths. Sample cross entropy is simply the average log (base 2) probability estimate per character. All entropies are reported for the best hyperparameter settings through online leave-one-out estimation for parameterized Witten-Bell smoothing. Each data point in the plot uses the average entropy rate over a sample size of up to 10,000 for MEDLINE and 100,000 for the *Times*, with the samples being drawn evenly over the data arriving since the last plot point. For instance, the point plotted at 200,000 characters for MEDLINE uses a sample of every 10th character between character 100,000 and 200,000 whereas the sample at 2,000,000,000 characters uses every 100,000th character between characters 1,000,000,000 and 2,000,000,000.

Like the Tipster data used by (Chen and Goodman, 1996), the immediately noticeable feature of the plots is the jaggedness early on, including some

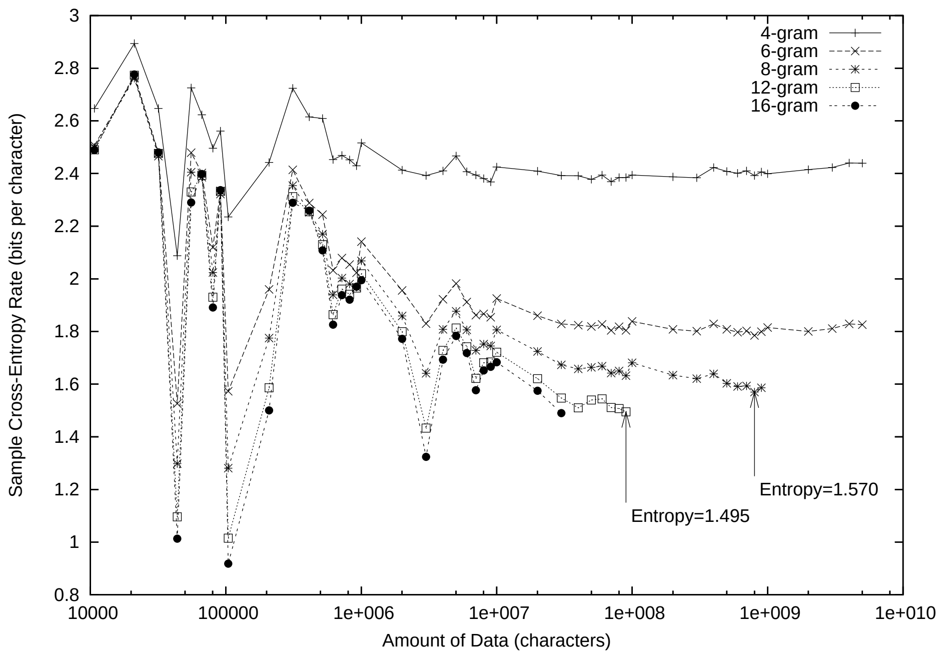


Figure 10: *NY Times* Sample Cross-Entropy Rates

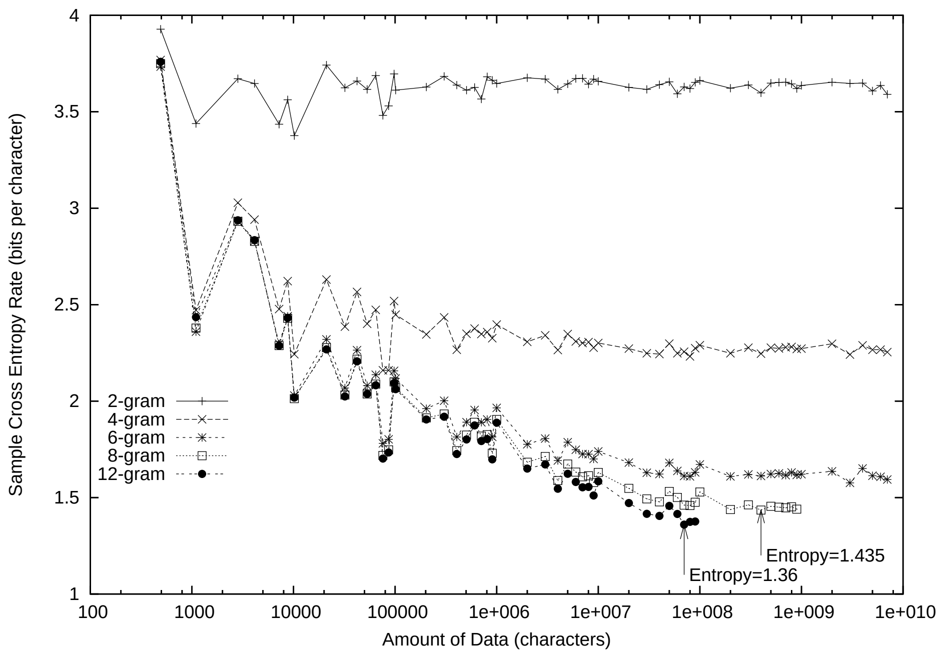


Figure 11: MEDLINE Sample Cross-Entropy Rates

ridiculously low cross-entropy rates reported for the *Times* data. This is largely due to low training data count, high *n*-gram models being very good at matching repeated passages coupled with the fact that a 2000 word article repeated out of 10,000 sample characters provides quite a cross-entropy reduction. For later data points, samples are sparser and thus less subject to variance.

For applications other than cross-entropy bake-offs, 5-grams to 8-grams seem to provide the right

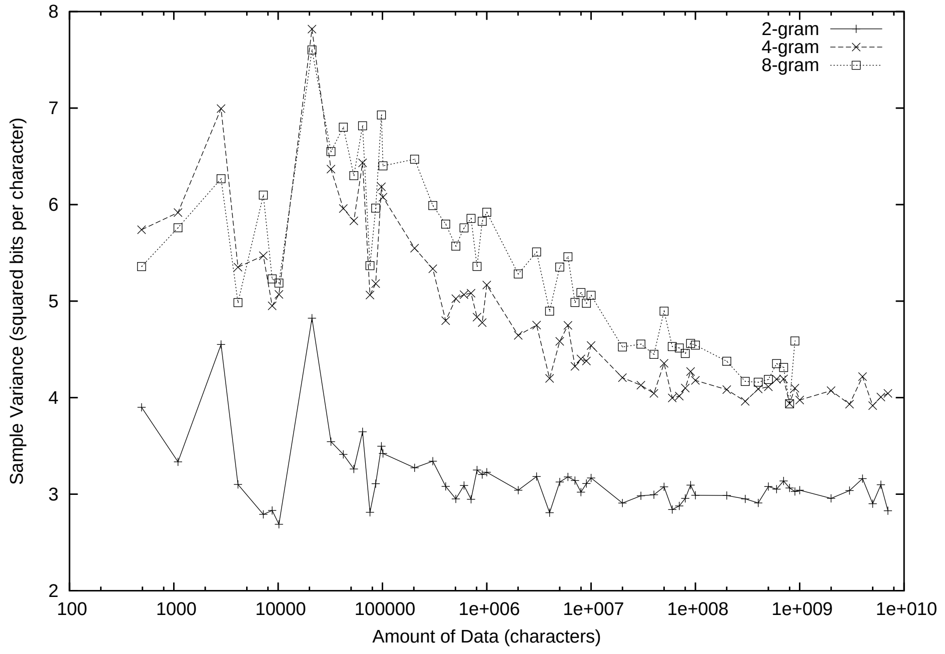


Figure 12: MEDLINE Sample Variances

compromise between accuracy and efficiency.

We were surprised that MEDLINE had lower *n*-gram entropy bounds than the *Times*, especially given the occurrence of duplication within the *Times* data (MEDLINE does not contain duplicates in the baseline). The best MEDLINE operating point is indicated in the figure, with a sample cross-entropy rate of 1.36 for 12-grams trained on 100 million characters of data; 8-gram entropy is 1.435 at nearly 1 billion characters. The best performance for the *Times* corpus was also for 12-grams at 100 million characters, but the sample cross-entropy was 1.49; with 8-gram sample cross-entropy as low as 1.570 at 1 billion characters. Although MEDLINE may be full of jargon and mixed-case alphanumeric acronyms, the way in which they are used is highly predictable given enough training data. Data in the *Times* such as five and six digit stock reports, sports scores, etc., seem to provide a challenge.

The per-character sample variances for 2-grams, 4-grams and 8-grams for MEDLINE are given in Figure 12. We did not plot results for higher-order *n*-grams, as their variance was almost identical to that of 8-grams. Standard error is the square root of variance, or about 2.0 in the range of interest. With 10,000 samples, variance should be 4/10,000, with standard error the square root of this, or 0.02. This is in line with measurement variances found at the tail end of the plots, but not at the beginnings.

Most interestingly, it turned out that smoothing method did not matter once *n*-grams were large, thus bringing the results of (Banko and Brill, 2001) to bear on those of (Chen and Goodman, 1996). The comparison for 12-grams and then for the tail of more data for 8-grams in Figures 13 and 14. Figure 14 shows the smoothing methods for 8-grams on an order of magnitude more data.

## Conclusions

We have shown that it is possible to use object oriented techniques to scale language model counts to very high levels without pruning on relatively modest hardware. Even more space could be saved by unfolding characters to bytes (especially for token models). Different smoothing models tend to converge to each other after gigabytes of data, making smoothing much less critical.

Full source with unit tests, javadoc, and applications is available from the LingPipe web site:

`http://www.alias-i.com/lingpipe`

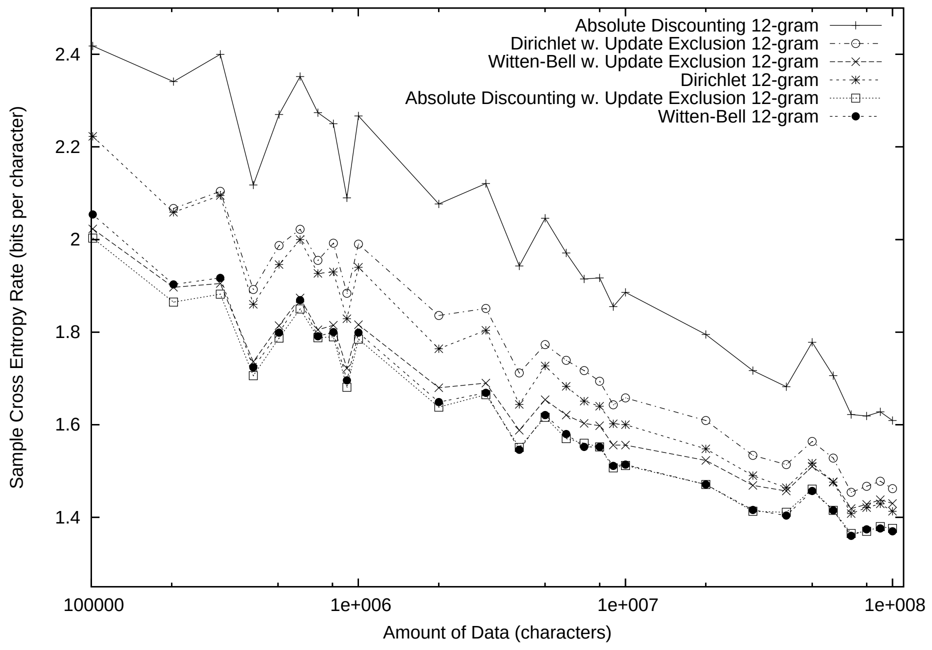


Figure 13: Comparison of Smoothing for 12-grams

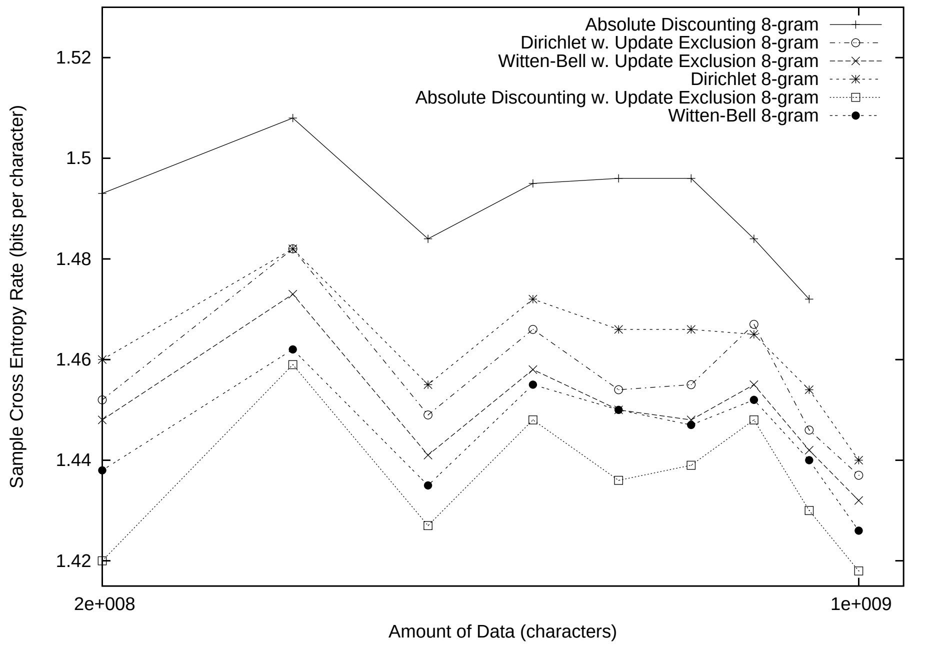


Figure 14: Comparison of Smoothing for 8-grams

## References

Michele Banko and Eric Brill. 2001. Scaling to very very large corpora for natural language disambiguation. In *Proceedings of the 39th Meeting of the ACL*.

Eric Brill and Robert C. Moore. 2000. An improved error model for noisy channel spelling correction. In *Proceedings of the 38th Annual Meeting of the ACL*.

Peter F. Brown, Stephen Della Pietra, Vincent J. Della Pietra, and Robert L. Mercer. 1991. Word-sense disambiguation using statistical methods. pages 264–270.

Stanley F. Chen and Joshua Goodman. 1996. An empirical study of smoothing techniques for language modeling. In *Proceedings of the 34th Annual Meeting of the ACL*, pages 310–318.

John G. Cleary and William J. Teahan. 1997. Unbounded length contexts for PPM. *The Computer Journal*, 40(2/3):67–??

Thomas M. Cover and Joy A. Thomas. 1991. *Elements of Information Theory*. John Wiley.

Frederick Jelinek and Robert L. Mercer. 1980. Interpolated estimation of Markov source parameters from sparse data. In *Proceedings of the Workshop on Pattern Recognition in Practice*. North-Holland.

Dan Klein, Joseph Smarr, Huy Nguyen, and Christopher D. Manning. 2003. Named entity recognition with character-level models. In *Proceedings the 7th ConNLL*, pages 180–183.

Reinhard Kneser and Hermann Ney. 1995. Improved backing off for n-gram language modeling. In *Proceedings of ICASSP*, pages 181–184.

Kevin Knight and Vasileios Hatzivassiloglou. 1995. Two-level, many-paths generation. In *Proceedings of the 33rd Annual Meeting of the ACL*.

Lucian Vlad Lita, Abe Ittycheriah, Salim Roukos, and Nanda Kambhatla. 2003. tRuEcasIng. In *Proceedings of the 41st Annual Meeting of the ACL*, pages 152–159.

David J. C. MacKay and Linda C. Peto. 1995. A hierarchical Dirichlet language model. *Natural Language Engineering*, 1(3):1–19.

Alistair Moffat. 1990. Implementing the PPM data compression scheme. *IEEE Transactions on Communications*, 38:1917–1921.

Hermann Ney, U. Essen, and Reinhard Kneser. 1995. On the estimation of 'small' probabilities by leaving-one-out. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 17:1202–1212.

Peter Norvig. 1991. *Paradigms of Artificial Intelligence Programming: Case Studies in Common Lisp*. Morgan Kaufmann.

Richard O'Keefe. 1990. *The Craft of Prolog*. MIT Press.

Fuchun Peng. 2003. *Building Probabilistic Models for Language Independent Text Classification*. Ph.D. thesis.

Gerasimos Potamianos and Frederick Jelinek. 1998. A study of n-gram and decision tree letter language modeling methods. *Speech Communication*, 24(3):171–192.

Christer Samuelsson. 1996. Handling sparse data by successive abstraction. In *Proceedings of COLING-96*, Copenhagen.

William J. Teahan and John G. Cleary. 1996. The entropy of english using PPM-based models. In *Data Compression Conference*, pages 53–62.

William J. Teahan. 2000. Text classification and segmentation using minimum cross-entropy. In *Proceeding of RIAO 2000*.

Edward Whittaker and Bhiksha Raj. 2001. Quantization-based language model compression. In *Proceedings of Eurospeech 2001*, pages 33–36.

ChengXiang Zhai and John Lafferty. 2004. A study of smoothing methods for language models applied to information retrieval. *ACM Transactions on Information Systems*, 2(2):179–214.

# XFST2FSA: Comparing Two Finite-State Toolboxes

**Yael Cohen-Sygal**
Department of Computer Science
University of Haifa
yaelc@cs.haifa.ac.il

**Shuly Wintner**
Department of Computer Science
University of Haifa
shuly@cs.haifa.ac.il

**Abstract**

This paper introduces *xfst2fsa*, a compiler which translates grammars expressed in the syntax of the XFST finite-state toolbox to grammars in the language of the FSA Utilities package. Compilation to FSA facilitates the use of grammars developed with the proprietary XFST toolbox on a publicly available platform. The paper describes the non-trivial issues of the compilation process, highlighting several shortcomings of some published algorithms, especially where replace rules are concerned. The compiler augments FSA with most of the operators supported by XFST. Furthermore, it provides a means for comparing the two systems on comparable grammars. The paper presents the results of such a comparison.

## 1 Introduction

Finite-state technology is widely considered to be the appropriate means for describing the phonological and morphological phenomena of natural languages since the pioneering works of Koskenniemi (1983) and Kaplan and Kay (1994). Finite state technology has some important advantages, making it most appealing for implementing natural language morphology. One can find it very hard, almost impossible, to build the full automaton or transducer describing some morphological phenomena. This difficulty arises from the fact that there are a great number of morpho-phonological processes combining together to create the full language. However, it is usually very easy to build a finite state machine to describe a specific morphological phenomenon. The closure properties of regular languages make it most convenient to implement each phenomenon independently and combine them together. Moreover, finite state techniques have the advantage of being efficient in their time and space complexity, as the membership problem is solvable in time linear in the length of the input. Furthermore, there are known algorithms for minimizing and determinizing automata and some restricted kinds of transducers.

Several finite state toolboxes (software packages) provide extended regular expression description languages and compilers of the expressions to finite state devices, automata and transducers (Karttunen et al., 1996; Beesley and Karttunen, 2003; Mohri, 1996; van Noord and Gerdemann, 2001a; van Noord and Gerdemann, 2001b). Such toolboxes typically include a language for extended regular expressions and a compiler from regular expressions to finite-state devices (automata and transducers). These toolboxes also include efficient implementations of several standard algorithms on finite state machines, such as union, intersection, minimization, determinization etc. More importantly, they also implement special operators that are useful for linguistic description, such as replacement (Kaplan and Kay, 1994; Mohri and Sproat, 1996; Karttunen, 1997; Gerdemann and van Noord, 1999) or predicates over alphabet symbols (van Noord and Gerdemann, 2001a; van Noord and Gerdemann, 2001b), and even operators for particular linguistic theories such as Optimality Theory (Karttunen, 1998; Gerdemann and van Noord, 2000). Unfortunately, there are no standards for the syntax of extended regular expression languages and switching from one toolbox to another is a non-trivial task.

This work focuses on two toolboxes, XFST (Beesley and Karttunen, 2003) and FSA Utilities (van Noord, 2000). Both are powerful tools for specifying and manipulating finite state machines (acceptors and transducers) using extended regular expression languages. In addition to the standard operators, XFST also provides advanced operators such as replacement, markup, and restriction (Karttunen, 1995; Karttunen, 1996; Karttunen, 1997; Karttunen and Kempe, 1995), and advanced methods such as compile-replace and flag-diacritics. FSA, on the other hand, supports weighted finite state machines and provides visualization of finite state networks. In addition, FSA is built over Prolog, allowing the additional usage of Prolog predicates. A first significant difference between the two packages is the wide variety of operators that XFST provides in comparison to FSA. However, FSA has the clear advantage of being a free, open source package, whereas XFST is proprietary.

This paper describes *xfst2fsa*, a compiler which translates XFST grammars to grammars in the language of the FSA Utilities package.[1] There is a strong parallelism between the languages, but certain constructs are harder to translate and require more innovation. In particular, all the replace operators that XFST provides do not exist in FSA and had to be re-implemented. In this work we relate only to the core of the finite state calculus – naïve automata and transducers. We do not deal with extended features such as the weighted networks of FSA or with advanced methods such as Prolog capabilities in FSA and compile replace and flag diacritics in XFST.

The contribution of this work is manifold. Our main motivation is to facilitate the use of grammars developed with XFST on publicly available systems. Furthermore, this work gives a closer insight into the theoretical algorithms which XFST is based on. We show that the algorithms published in the literature are incomplete and require refinement in order to be correct for all inputs. Moreover, our compiler enriches FSA with implementations of several replace rules, thereby scaling up the system and improving its expressivity. Finally, this work offers an investigation of two similar, but different systems: the compiler facilitates a comparison of the two systems on very similar benchmarks.

## 2 The xfst2fsa compiler

Compilation of a given XFST grammar into an equivalent FSA code is done in two main stages: first, the XFST grammar is parsed, and a tree representing its syntax is created. Then, by traversing the tree, the equivalent FSA grammar is generated.

In order to parse XFST grammars, a specification of the XFST syntax is required. Unfortunately, we were unable to obtain a formal specification and we resorted to reconstructing it from available documentation (Beesley and Karttunen, 2003).This turned out to be a minor inconvenience; a more major obstacle was the semantics of XFST expressions, especially those involving advanced operators such as replace rules, markup and restriction. We exemplify in this section some of these issues.

XFST operators can be divided into three groups with respect to their FSA equivalence: those which have an equivalent operator in FSA, those which do not but can be easily constructed from basic FSA operators, and those whose constructions is more complicated. In what follows we refer to operators of the first two groups as basic operators. Figure 1 displays a comparison table of some basic operators in XFST and FSA.[2] For example, consider the XFST operator `$?A` ("contains at most one"). This operator is not provided by FSA but can be constructed as `{$A - ignore([A,A],?*),?* - $A}`. It is now provided by FSA in our publicly available package of extended FSA operators. The same holds for XFST operators such as `A./.B` (internally ignore), `$.A` (contains one) etc. As another example consider the XFST operator `^` (n-ary concatenation). It does not have an equivalent operator in FSA, but it can be simply constructed in FSA by explicitly expressing the concatenation as many times as needed. Thus, the XFST regular expression `[a*|b^3]` is translated into the equivalent FSA regular expression `{a*,[b,b,b]}`. Similar techniques are used for other basic XFST operators such as `A^>n` (more

[1]The system and the source code are available for download from http://cl.haifa.ac.il/projects/fstfsa/index.shtml

[2]The complete list of XFST operators and the way they where translated into FSA can be found in Appendix A.

than *n* concatenations of *A*), `A^{n,k}` (*n* to *k* concatenations of A) etc.

Another minor issue is the different operator precedence in XFST and FSA. This problem was solved by bracketing each translated operator in XFST with '( )' to force the correct precedence.

Special care is needed in order to deal with XFST operators of the third group, e.g., all the replace, markup and restriction rules in XFST. These rules do not have any equivalents in FSA, and hence the only way to use them in FSA is to implement them from scratch. This was done using the existing documentation (Karttunen, 1995; Karttunen, 1996; Karttunen, 1997; Karttunen and Kempe, 1995) on the construction of these operators from the basic ones. However, not all the operators are fully documented and in some cases some innovation was needed. As an example, consider the XFST operator `A@<-B` (obligatory, lower to upper, left to right, longest match replacement). To the best of our knowledge, this operator is not documented. However, by Karttunen (1995), the operator `A<-B` (obligatory, lower to upper replacement ) is defined as `[B->A].i` (where `B->A` is the obligatory, upper to lower replacement of the language B by the language A). We then concluded that `A@<-B` is constructed as `[B@->A].i` (where `[B@->A]` is the obligatory, upper to lower, left to right, longest match replacement of the language B by the language A) and from Karttunen (1996) the construction of the operator `B@->A` is known.

For some of the documented operators, we found that the published algorithms are erroneous in some special cases. Consider the replace operator `A->B || L _ R` (conditional replacement of the language A by the language B, in the context of L on the left and R on the right side, where both contexts are on the upper side). In Karttunen (1997; 1995), a detailed description of the construction of this operator is given. In addition, Karttunen (1997) discusses some boundary cases, such as the case in which the language *A* contains the empty string. We discovered that there are some cases which are not discussed as boundary ones in Karttunen (1997) and for which the standard algorithm in Karttunen (1997; 1995) does not produce the expected result by the definition of the operator denotation. One such case is a rule of the form `A->B || _ ?`, where *A* and *B* are some regular expressions denoting languages. This rule states that any member of the language A on the upper side is replaced by all members of the language B on the lower side when the upper side member is not followed by the end of the string on which the rule operates. For example, the rule `a->b || _ ?` is expected to generate the automaton of Figure 2. However, a direct implementation of the algorithms of Karttunen (1997; 1995) always yields a network accepting the empty language, independently of the way *A* and *B* are defined. Other ambiguous cases are discussed in Vaillette (2004).

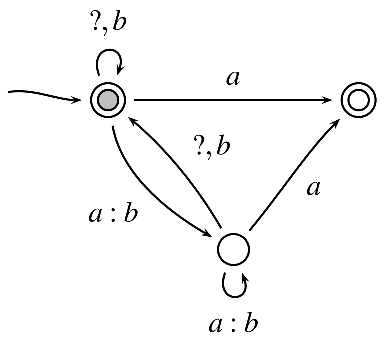


Figure 2: Desired interpretation of the regular expression $a- > b \,||\, \_?$

Furthermore, in some cases XFST produces networks that are somewhat different from the ones in the literature: the relations (as sets) are equal but the resulting automata (as graphs) are not isomorphic. For example, consider the replace rule `a->b || c _ d`. This expression is compiled by XFST to the automaton shown in Figure 3. Implementing this rule from basic operators as described in Karttunen (1997; 1995), results in the automaton of Figure 4. Observe that in some cases multiple accepting paths are obtained. This is probably a result of adding $\epsilon$-self-loops in order to deal correctly with $\epsilon$ symbols, following Pereira and Riley (1997); the multiple paths can then be removed using filters. We assume that the same solution is adopted by XFST. This solution requires direct access to the underlying network, and cannot be applied at the level of the regular expression language. Therefore, we did not utilize it in our implementation of replace rules.

To validate the construction of the compiler, one would ideally want to check that the obtained FSA

| XFST syntax | FSA syntax | Meaning |
|---|---|---|
| `A*` | `A*` | Kleene star |
| `A \| B` | `{A,B}` | union |
| `A & B` | `A & B` | intersection |
| `A - B` | `A - B` | A minus B |
| `A/B` | `ignore(A,B)` | A ignoring B |
| `$A` | `$A` | containment |
| `$?A` | `{$A - ignore([A,A],?*) , ?* - $A}` | maximum one containment |
| `A B` | `[A,B]` | concatenation |
| `A^n` | does not exist | n-ary concatenation |
| `A.x.B` | `A x B` | crossproduct |
| `A.o.B` | `A o B` | composition |
| `(A)` | `A^` | optionality |
| `[ ]` | `( )` | precedence |
| `R.i` | `invert(R)` or `inverse(R)` | regular relation inverse |

Figure 1: A comparison table of some simple classic operators in XFST and FSA

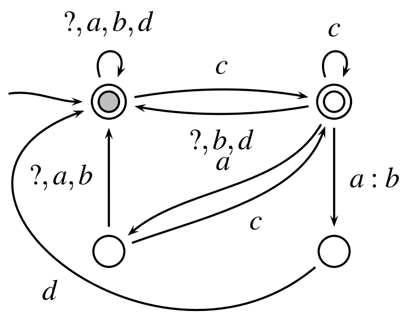


Figure 3: Automaton created from the regular expression $a- > b \mid\mid c \_ d$ by XFST

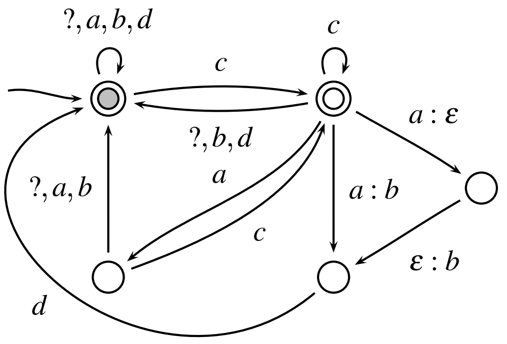


Figure 4: Automaton created from the regular expression $a- > b \mid\mid c \_ d$ by the published algorithm

networks are equivalent to the XFST ones from which they were generated. Unfortunately, this is only possible for very small networks, since XFST does not allow to print its networks, when they are significantly large. We could only test XFST networks and their FSA images over test strings to validate the identity of the outputs. In addition to checking each operator by itself for several instances, we tested the compiler on a more comprehensive code, namely HAMSAH (Yona and Wintner, 2005), which was designed and implemented using XFST. We successfully converted the entire network into FSA with the compiler. Exhaustive tests produced the same outputs for both networks.

## 3 Comparison of XFST and FSA

A byproduct of the compiler is a full implementation, in FSA, of a vast majority of XFST's operators.[3] In addition to the contribution to FSA users, this also facilitates an effective comparison between the two toolboxes on similar benchmarks. We now describe the results of such a comparison, focusing on usability and performance.

### 3.1 Display of networks

FSA displays networks in two possible formats: as text, by listing the network states and transitions,

[3]We implemented in FSA all the operators of XFST, except parallel conditional replace rules and some direct replacement and markup rules.

and through a graphical user interface. The GUI that FSA employs is user friendly, allows many kinds of manipulations of the networks and significantly helps to the understanding of the networks, especially when they are small. The viewing parameters can be scaled by the user, thus improving the visualization possibilities. Moreover, networks can be saved in many different formats including binary (for fast loading and saving), text (allowing inspection of the network without the necessity to use FSA) and postscript (for printing). FSA also enables generation of C, C++, Java and Prolog code, implementing analysis with a network.

XFST, on the other hand, prints its networks only in text format, and even this is supported for small networks only. Networks in XFST can be saved in binary format only, thus requiring the usage of XFST in order to inspect the network. With respect to visual display and ease of use, therefore, FSA has clear benefits over XFST.

### 3.2 Performance

A true comparison of the two systems should compare two different grammars, each designed specifically for one of the two toolboxes, yielding the same comprehensive network. However, as such grammars are not available, we compared the two toolboxes using a grammar designed and implemented in XFST and its conversion into FSA. Again we used HAMSAH (Yona and Wintner, 2005) for this purpose. The Hebrew morphological analyzer is a large XFST grammar: the complete network consists of approximately 2 million states and 2.2 million arcs. We also inspected two subnetworks: the Hebrew adjectives network (approximately 100,000 states and 120,000 arcs) and the Hebrew nouns network (approximately 700,000 states and 950,000 arcs). Each of the networks was created by composing a series of rules over a large-scale lexicon. Since Hebrew morphology is non-trivial, the final network is created by composing many intermediate complex regular expressions (including many replace rules and compositions). The grammars were compiled and executed on a 64-bit computer with 16Gb of memory. The table in Figure 5 shows the differences in compilation and analysis times and memory requirements between the two toolboxes. XFST performed immeasurably better than FSA. In particular, we were unable to use the complete FSA network for analysis, compared to analyzing 70 words per second with the full network in XFST. Another issue that should be noticed is the difference in memory requirements.

## 4 Conclusions

We presented a compiler which translates XFST grammars to grammars in the language of the FSA Utilities package. This work allows a closer look into two of the most popular finite state toolboxes. Although FSA has the advantage of being a publicly available software, we discovered that it does not scale up as well as XFST. However, for the non-expert user or for teaching purposes, where more modest networks are manipulated, FSA seems to be more friendly, especially with regard to graphical representation of networks. With our new implementation of replace rules in FSA, it seems that for such uses FSA is better. However, for larger systems and when time performance is an issue, XFST is preferable.

This work can be extended in several directions. Not all XFST operators are implemented in FSA. Some for lack of documentation and some simply require more time for implementation. Thus, further work can be done to construct more operators (see footnote 3). We believe that replace rules still hide boundary cases which require special treatment. More work is needed in order to locate such cases. Furthermore, other finite state toolboxes exist (Mohri, 1996) which present different operators. Extending the compiler to convert XFST grammars into those formalisms will provide opportunities for better comparison of different finite-state toolboxes. On a different course, an fsa2xfst compiler can be constructed. Such a compiler will enable a reverse performance comparison, i.e. using a larger network for FSA and making it operational in XFST. Notice that in contrast to the xfst2fsa direction, this course is rather trivial: FSA allows the user to save its networks in a readable format (listing the network states and arcs). Although XFST is not capable of reading any format but its own, Kleene (1954) presents a simple algorithm for generating from a given FSA a regular expression denoting it. Using this algorithm, an XFST regular expression denoting the network

|  |  | FSA |  | XFST |  |
|---|---|---|---|---|---|
|  |  | Time | Memory | Time | Memory |
| Compilation | complete network | 13h 43m | ≈11G | 27m 41s | ≈3G |
|  | nouns network | 2h 29m |  | 11m 4s |  |
|  | adjectives network | 14m 56s |  | 8m 21s |  |
| Analysis | complete network, 350 words | not possible |  | 5s |  |
|  | nouns network, 120 nouns | 1h 50m |  | 0.17s |  |
|  | adjectives network, 50 adjectives | 2m 34s |  | 0.17s |  |

Figure 5: Times and memory requirements

can be generated. The only disadvantage of such an approach is that the resulting XFST expression will be most cumbersome.

## Acknowledgments

This research was supported by The Israel Science Foundation (grant no. 136/01). We are grateful to Gertjan van Noord and Shlomo Yona for their help.

## References

Kenneth R. Beesley and Lauri Karttunen. 2003. *Finite-State Morphology*. CSLI Publications.

Dale Gerdemann and Gertjan van Noord. 1999. Transducers from rewrite rules with backreferences. In *Ninth Conference of the European Chapter of the Association for Computational Linguistics*, pages 126–133, Bergen Norway.

Dale Gerdemann and Gertjan van Noord. 2000. Approximation and exactness in finite state optimality theory. In *Proceedings of Sigphon Workshop on Finite State Phonology (invited paper)*, pages 34–45, Luxembourg. http://xxx.lanl.gov/ps/cs.CL/0006038 or ROA-403-08100 at http://ruccs.rutgers.edu/roa.html.

Ronald M. Kaplan and Martin Kay. 1994. Regular models of phonological rule systems. *Computational Linguistics*, 20(3):331–378, September.

Lauri Karttunen and Andre Kempe. 1995. The parallel replacement operation in finite state calculus. Technical Report 1995-021, Rank Xerox research centre – Grenoble laboratory.

Lauri Karttunen, Jean-Pierre Chanod, Gregory Grefenstette, and Anne Schiller. 1996. Regular expressions for language engineering. *Natural Language Engineering*, 2(4):305–328.

Lauri Karttunen. 1995. The replace operator. In *Proceedings of the 33rd Annual Meeting of the Association for Computational Linguistics*, pages 16–24.

Lauri Karttunen. 1996. Directed replacement. In *Proceedings of ACL'96*, pages 108–115.

Lauri Karttunen. 1997. The replace operator. In Emmanuel Roche and Yves Schabes, editors, *Finite-State Language Processing*, Language, Speech and Communication, chapter 4, pages 117–147. MIT Press, Cambridge, MA.

Lauri Karttunen. 1998. The proper treatment of Optimality Theory in computational phonology. In *Finite-state methods in natural language processing*, pages 1–12, Ankara, June.

S. C. Kleene. 1954. Representation of events in nerve nets and finite automata. In C. E. Shannon and J. McCarthy, editors, *Automata Studies*, pages 3–42. Princeton University Press.

Kimmo Koskenniemi. 1983. *Two-Level Morphology: a General Computational Model for Word-Form Recognition and Production*. The Department of General Linguistics, University of Helsinki.

Mehryar Mohri and Richard Sproat. 1996. An efficient compiler for weighted rewrite rules. In *Proceedings of the 34th Annual Meeting of the Association for Computational Linguistics*, pages 231–238, Santa Cruz.

Mehryar Mohri. 1996. On some applications of finite-state automata theory to natural language processing. *Natural Language Engineering*, 2(1):61–80.

Fernando Pereira and Michael Riley. 1997. Speech recognition by composition of weighted finite automata. In Emmanuel Roche and Yves Schabes, editors, *Finite-State Language Processing*, pages 431–453. MIT Press, Cambridge.

Nathan Vaillette. 2004. *Logical Specification of Finite-State Transductions for Natural Language Processing*. Ph.D. thesis, Ohio State University.

Gertjan van Noord and Dale Gerdemann. 2001a. An extendible regular expression compiler for finite-state approaches in natural language processing. In O. Boldt and H. Jürgensen, editors, *Automata Implementation*, number 2214 in Lecture Notes in Computer Science. Springer.

Gertjan van Noord and Dale Gerdemann. 2001b. Finite state transducers with predicates and identity. *Grammars*, 4(3):263–286.

Gertjan van Noord, 2000. *FSA6 Reference Manual*.

Shlomo Yona and Shuly Wintner. 2005. A finite-state morphological grammar of hebrew. In *Proceedings of the ACL-2005 Workshop on Computational Approaches to Semitic Languages*.

## Appendix

## A A comparison table of XFST and FSA operators

### A.1 Symbols

| XFST syntax | FSA syntax | Meaning |
|---|---|---|
| $a$ | $a$ | single symbol a |
| %$*$ or “*” | escape(*) or ’*’ | escape literal symbol |
| $abc$ | $abc$ | multi-character symbol |
| ? | ? | any symbol |
| 0 or [ ] or $''\,''$ | [ ] | epsilon symbol, the empty string |
| $\{abcd\}$ | $[a,b,c,d]$ | single character brace |

### A.2 Basic operators

| XFST syntax | FSA syntax | Meaning |
|---|---|---|
| $A*$ | $A*$ | Kleene star |
| $A+$ | $A+$ | iteration (Kleene plus) |
| $A \mid B$ | $\{A,B\}$ | union |
| $A \,\&\, B$ | $A \,\&\, B$ | intersection |
| $A - B$ | $A - B$ | A minus B |
| $\backslash A$ | $[\sim A] \,\&\, ?$ | term complement |
| $\sim A$ | $\sim A$ | complement |
| $A/B$ | $ignore(A,B)$ | A ignoring B |
| $A./.B$ | $ignore(A,B) - \{[B,?*],[?*,B]\}$ | A ignoring internally B |
| $\$A$ | $\$A$ | containment |
| $\$.A$ | $\$A - ignore([A,A],?*)$ | one containment |
| $\$?A$ | $\{\$A - ignore([A,A],?*), ?* - \$A\}$ | maximum one containment |
| $A\ B$ | $[A,B]$ | concatenation |
| $A\hat{}n$ | | n-ary concatenation |
| $A\hat{}\{n,k\}$ | | n to k concatenations of A |
| $A\hat{} > n$ | | more than n concatenations of A |
| $A\hat{} < n$ | | less than n concatenations of A |
| $A.x.B$ | $A\ x\ B$ | crossproduct |
| $A.o.B$ | $A\ o\ B$ | composition |
| $(A)$ | A^ | optionality |
| $a : b$ | $a : b$ | symbol pair |
| [ ] | ( ) | order control |
| $R.P.Q$ | $\{R, (domain(Q) - domain(R))\ o\ Q\}$ | upper-side priority union |
| $R.p.Q$ | $\{R, Q\ o\ (range(Q) - range(R))\}$ | lower-side priority union |
| $R.-u.Q$ | $(domain(R) - domain(Q))\ o\ R$ | upper-side minus |
| $R.-l.Q$ | $R\ o\ (range(R) - range(Q))$ | lower-side minus |
| $A < B$ | $\sim \$[B,A]$ | A before B |
| $A > B$ | $\sim \$[A,B]$ | A after B |
| $A.r$ | $reverse(A)$ | reverse |
| $R.u$ or $R.1$ | $domain(R)$ | upper language of the regular relation R |
| $R.l$ or $R.2$ | $range(R)$ | lower language of the regular relation R |
| $R.i$ | $invert(R)$ or $inverse(R)$ | regular relation inverse |

### A.3 Restriction

XFST restriction rules are not provided by FSA, nor did we implement them. Therefore, we only present their syntax in XFST.

- $A => L\_R$
- $A => L_1\_R_1, L_2\_R_2, \ldots, L_n\_R_n$

### A.4 Replacement

XFST replace rules do not exist in FSA. We present implementation of most of them in FSA, based on (Karttunen, 1995; Karttunen, 1996; Karttunen, 1997; Karttunen and Kempe, 1995). XFST replace rules can be divided into 4 groups:

1. Unconditional replace rules (one rule with no context).
2. Unconditional parallel replace rules (several rules with no context that are performed at the same time).
3. Conditional replace rules (one rule and one condition).
4. Conditional parallel replace rules (several rules and/or several contexts).

#### A.4.1 $->$ (obligatory, upper to lower replacement)

- XFST syntax: $A -> B$

  Meaning: Unconditional replacement of the language A by the language B.

  Construction: [[NO_ A [$A$ .$x$. $B$] ]$*$ NO_ A] where NO_ A abbreviates $\sim$ \$[ $A - [\,]$ ].

- XFST syntax: $A_1 -> B_1, \ldots, A_n -> B_n$

  Meaning: Unconditional parallel replacement of the language $A_1$ by the language $B_1$ and the language $A_2$ by the language $B_2$ ... and the language $A_n$ by the language $B_n$.

  Construction: $[\,[N\,R\,]*\,N\,]$ where N denotes the language of strings that do not contain any $A_i$:

$$N =\sim \$[\;[A_1 \mid \ldots \mid A_n] - [\,]\;]$$

  and R stands for the relation that pairs every $A_i$ with the corresponding $B_i$:

$$R = [\;[A_1 \;.x. B_1] \mid \ldots \mid [A_n \;.x. B_n]\;]$$

  .

- XFST syntax: $A -> B \,||\, L\,\_R$

  Meaning: Conditional replacement of the language A by the language B. This is like the relation $A -> B$ except that any instance of A in the upper string corresponds to an instance of B in the lower string only when it is preceded by an instance of L and followed by an instance of R. Other instances of A in the upper string remain unchanged. A, B, L, and R must all denote simple languages, not relations. The slant of the double bars indicates whether the precede/follow constraints refer to the instance of A in the upper string or to its image in the lower string. In the || version, both contexts refer to the upper string.

  Construction: *InsertBrackets .o. ConstrainBrackets .o. LeftContext .o. RightContext .o. Replace .o. RemoveBrackets* where:

– Let $<$ and $>$ be two symbols not in $\Sigma$. The escape character % is used since $<$ and $>$ are saved symbols in XFST.

– $InsertBrackets = [\,[\,] <- \% < \mid \% >\,]$
InserBrackets eliminates from the upper side language all context markers that appear on the lower side.

– $ConstrainBrackets = [\sim \$[\% < \ \% >]\,]$
ConstrainBrackets denotes the language consisting of strings that do not contain <> anywhere.

– $LeftContext = [\sim[\sim[...LEFT]\ [<...]\,]\ \&\ \sim[\,[...LEFT]\ \ \sim[<...]\,]\,]$
LeftContext denotes the language in which any instance of < is immediately preceded by LEFT and every LEFT is immediately followed by <, ignoring irrelevant brackets. $[...LEFT]$ denotes $[\,[\,?*\ L/[\% < \mid \% >]\,] - [?*\ \% <]\,]$, the language of all strings ending in $L$, ignoring all brackets except for a final <. $[<...]$ denotes $[\% < /\% > ?*]$, the language of strings beginning with <, ignoring the other bracket.

– $RightContext = [\sim[\,[...>]\ \ \sim[RIGHT...]\,]\ \&\ \sim[\sim[...>]\ [RIGHT...]\,]\,]$
RightContext denotes the language in which any instance of > is immediately followed by RIGHT and any RIGHT is immediately preceded by >, ignoring irrelevant brackets. $[RIGHT...]$ denotes $[\,[\,R/[\% < \mid \% >]\ \ ?*\,] - [\% < ?*]\,]$, the language of all strings beginning with $R$, ignoring all brackets except for an initial >. $[...>]$ denotes $[?*\ \% > /\% <]$, the language of strings ending with >, ignoring the other bracket.

– The definition of Replace divides into three cases:

1. If A does not contain the empty string (epsilon) then

$$Replace = [\% <\ \ A/[\% < \mid \% >]\ \ \% >\ ->\ \% <\ \ B/[\% < \mid \% >]\ \ \% >]$$

This is the unconditional replacement of $<A>$ by $<B>$, ignoring irrelevant brackets.

2. If A is the empty string (i.e., $A = \varepsilon$) then

$$Replace = [\% >\ \ \% <\ ->\ \% <\ \ B*\ \ \% >]$$

3. If A contains the empty string but is not equal to it (i.e., contains other strings too) then

$$Replace =$$

$$[\% < A/[\% <\mid \% >]\,\% >\ ->\ \% < B/[\% <\mid \% >]\,\% >]\ ,\ \% >\ \% <\ ->\ \% < B*\ \% >$$

That is, the first two cases are performed in parallel.

– $RemoveBrackets = [\ \% < \mid \% >\ ->\ [\,]\,]$. RemoveBrackets denotes the relation that maps the strings of the upper language to the same strings without any context markers.

• XFST syntax: $A->B\ //\ L\ \_\ R$

Meaning: Conditional replacement of the language A by the language B. This is like the relation $A->B\ ||\ L\ \_\ R$ except that the // variant requires the left context on the lower side of the replacement and the right context on the upper side.

Construction: *InsertBrackets .o. ConstrainBrackets .o. RightContext .o. Replace .o. LeftContext .o. RemoveBrackets* where InsertBrackets, ConstrainBrackets, RightContext, Replace, LeftContext and RemoveBrackets are the same as above.

- XFST syntax: $A- > B \backslash\backslash L \_ R$

  Meaning: Conditional replacement of the language A by the language B. This is like the relation $A- > B \, || \, L \_ R$ except that the \\ variant requires the left context on the upper side of the replacement and the right context on the lower side.

  Construction: *InsertBrackets .o. ConstrainBrackets .o. LeftContext .o. Replace .o. RightContext .o. RemoveBrackets* where InsertBrackets, ConstrainBrackets, RightContext, Replace, LeftContext and RemoveBrackets are the same as above.

- XFST syntax: $A- > B \backslash/ L \_ R$

  Meaning: Conditional replacement of the language A by the language B. This is like the relation $A- > B \, || \, L \_ R$ except that in the \/ variant, both contexts refer to the lower string.

  Construction: *InsertBrackets .o. ConstrainBrackets .o. Replace .o. LeftContext .o. RightContext .o. RemoveBrackets* where InsertBrackets, ConstrainBrackets, RightContext, Replace, LeftContext and RemoveBrackets are the same as above.

The rest of the obligatory upper to lower replace rules are conditional parallel replace rules that where not implemented. An example of such a rule is $A_{11}- > B_{11}, \ldots, A_{1n}- > B_{1n} \, || \, L_{11} \_ R_{11}, \ldots, L_{1m} \_ R_{1m}$.

### A.4.2 $(- >)$ (optional, upper to lower replacement)

- XFST syntax: $A(- >)B$

  Construction: $[\, [\, ?* \; [A \; .x. \; B] \,]* \; ?* \,]$.

- XFST syntax: $A_1(- >)B_1, \ldots, A_n(- >)B_n$

  Construction: $[\, [\, ?* \; R \,]* \; ?* \,]$ where R stands for the relation that pairs every $A_i$ with the corresponding $B_i$: $R = [\, [A_1 \; .x. B_1] \mid \ldots \mid [A_n \; .x. B_n] \,]$.

- XFST syntax: $A(- >)B \, || \, L \_ R, A(- >)B \, // \, L \_ R, A(- >)B \, \backslash\backslash \, L \_ R, A(- >)B \, \backslash/ \, L \_ R$

  Construction: The same as the construction for the corresponding rules with the operator $- >$ with the difference that in the *Replace* stage, for each one of the three cases the obligatory upper to lower replace operator $- >$ is replaced by the optional upper to lower replace operator $(- >)$.

The rest of the optional upper to lower replace rules are conditional parallel replace rules and were not implemented.

### A.4.3 $< -$ (obligatory, lower to upper replacement)

- XFST syntax: $A < -B$

  Construction: $[B- > A].i$

- XFST syntax: $A_1 < -B_1, \ldots, A_n < -B_n$

  Construction: $[\, B_1- > A_1, \ldots, B_n- > A_n \,].i$

- XFST syntax: $A < -B \, || \, L \_ R, A < -B \, // \, L \_ R, A < -B \, \backslash\backslash \, L \_ R, A < -B \, \backslash/ \, L \_ R$

  Construction: The same as the construction for the corresponding rules with the operator $- >$ with the difference that in the *Replace* stage, for each one of the three cases the obligatory upper to lower replace operator $- >$ is replaced by the operator $< -$.

The rest of the obligatory lower to upper replace rules are conditional parallel replace rules and were not implemented.

### A.4.4 $(<-)$ (optional, lower to upper replacement)

- XFST syntax: $A(<-)B$

  Construction: $[B(->)A].i$

- XFST syntax: $A_1(<-)B_1,\ldots,A_n(<-)B_n$

  Construction: $[\,B_1(->)A_1,\ldots,B_n(->)A_n\,].i$

- XFST syntax: $A(<-)B\ ||\ L\ \_R, A(<-)B\ //\ L\ \_R, A(<-)B\ \backslash\backslash\ L\ \_R, A(<-)B\ \backslash/\ L\ \_R$

  Construction: The same as the construction for the corresponding rules with the operator $->$ with the difference that in the *Replace* stage, for each one of the three cases the obligatory upper to lower replace operator $->$ is replaced by the operator $(<-)$.

The rest of the optional lower to upper replace rules are conditional parallel replace rules and were not implemented.

### A.4.5 $<->$ (obligatory, upper to lower, lower to upper replacement)

- XFST syntax: $A<->B$

  Construction: Let @ be a character not in $\Sigma$. We use the escape character % to precede @, since @ is a reserved character in XFST. Thus, $A<->B$ is defined as

$$\sim\$[\%@]$$

$$.o.$$

$$A->\ \%@$$

$$.o.$$

$$\%@\ <-B$$

$$.o.$$

$$\sim\$[\%@]$$

- XFST syntax: $A_1<->B_1,\ldots,A_n<->B_n$

  Construction: Let $@1,\ldots,@n$ be characters not in $\Sigma$. We use the escape character % to precede each $@i$, since @ is a reserved character in XFST. Thus, $A<->B$ is defined as

$$\sim\$[\%@1\ |\ \ldots\ |\ \%@n]$$

$$.o.$$

$$A_1->\ \%@1\ ,\ \ldots\ ,\ A_n->\ \%@n$$

$$.o.$$

$$\%@1\ <-B_1\ ,\ \ldots\ ,\ \%@n\ <-B_n$$

$$.o.$$

$$\sim\$[\%@1\ |\ \ldots\ |\ \%@n]$$

The rest of the obligatory upper to lower and lower to upper replace rules are conditional replace rules and they were not implemented.

### A.4.6 $(<->)$ (optional, upper to lower, lower to upper replacement)

- XFST syntax: $A(<->)B$

  Construction: Let @ be a character not in $\Sigma$. We use the escape character % to precede @, since @ is a reserved character in XFST. Thus, $A(<->)B$ is defined as

$$\sim \$[\%@]$$

$$.o.$$

$$A\ (->)\ \%@$$

$$.o.$$

$$\%@\ (<-)\ B$$

$$.o.$$

$$\sim \$[\%@]$$

The rest of the optional upper to lower and lower to upper replace rules are conditional and parallel replace rules and they were not implemented.

### A.4.7 $@->$ (obligatory, upper to lower, left to right, longest match replacement)

The following operations are the same as in section A.4.1, except that instead of $->$ occurs $@->$. As $->$ represented an obligatory upper to lower replacement, $@->$ represents an obligatory upper to lower left to right longest match replacement. Instances of the language A on the upper side of the relation are replaced selectively. The selection factors each upper language string uniquely into A and non-A substrings. The factoring is based on a left-to-right and longest match regimen. The starting locations are selected from left-to-right. If there are overlapping instances of A starting at the same location, only the longest one is replaced by all strings of B.

- XFST syntax: $A@->B$

  Construction: $InitialMatch\ .o.\ LeftToRight\ .o.\ LongestMatch\ .o.\ Replacement$

  Where:

  - Let ˆ, <, > be characters not in $\Sigma$. We use the escape character % to precede them since they are reserved characters in XFST.
  - $InitialMatch =$

$$\sim \$[\ \%\hat{}\ |\ \%<\ |\ \%>\ ]$$

$$.o.$$

$$[.\,.] -> \ \%\hat{}\ ||\ \_A$$

  where $[.\,.] -> \ LOWER\ ||\ LEFT\_RIGHT$ is a version of empty string replacement that allows only one application between any LEFT and RIGHT. The construction for $[.\,.] -> LOWER\ ||\ LEFT\_RIGHT$ is the same as for $UPPER -> \ LOWER\ ||\ LEFT\_RIGHT$ except that $Replace = [\% > \ \% < \ -> \ \% < \ LOWER\,\% >]$.
  - $LeftToRight =$

$$[\sim \$[\%\hat{}]\ [\%\hat{}:\% < \ UPPER'\ \ 0:\% >\ ]\ ]* \ \sim \$[\%\hat{}]$$

$$.o.$$

$$\%\hat{} -> \ [\,]$$

  where $UPPER' = [\,A/[\%\hat{}] - [?*\ \%\hat{}]\,]$

- $LongestMatch = \sim \$[\ \% <\ [\ UPPER'' \ \&\ \$[\% >]\ ]\ ]$
  where $UPPER'' =\ A/[\% <\ |\ \% >]\ -\ [\ ?*\ [\ \% <\ |\ \% >]\ ]$
- $Replacement =\ \% < \sim \$[\% >]\ \% >\ -\ >\ B$

The rest of the obligatory upper to lower left to right longest match replace rules are conditional and parallel replace rules and they were not implemented.

The following XFST operators were not implemented:

- @ > (obligatory, upper to lower, left to right, shortest match replacement)
- − > @ (obligatory, upper to lower, right to left, longest match replacement)
- > @ right (obligatory, upper to lower, right to left, shortest match replacement)

## A.5 Markup

Markup rules take an input string and mark it by inserting some strings before and after it. XFST markup rules do not exist in FSA. We present the implementation of most of them in FSA, based on Karttunen (1996).

- XFST syntax: $A- > L \ldots R$

  Meaning: Markup. Instances of the language A on the upper side of the relation are selected for markup. Each selected A string is marked by inserting all strings of L to its left and all strings of R to its right. The selected A strings themselves remain unchanged, along with the non-A segments.

  Construction: $A\ -\ >\ L\,A\,R$

- XFST syntax: $A@- > L \ldots R$

  Meaning: Directed markup. Instances of the language A on the upper side of the relation are selected for markup under left-to-right, longest match regimen. Thus, the starting locations are selected from left-to-right. If there are overlapping instances of A starting at the same location, only the longest one is replaced by all strings of B. Each selected A string is marked by inserting all strings of R to its left and all strings of S to its right. The selected A strings themselves remain unchanged, along with the non-A segments.

  Construction: $InitialMatch\ .o.\ LeftToRight\ .o.\ LongestMatch\ .o.\ Insrtion$

  Where:

  - Let ˆ, <, > be characters not in Σ. We use the escape character % to precede them since they are reserved characters in XFST.
  - $InitialMatch =$

    $$\sim \$[\ \%\hat{}\ |\ \% <\ |\ \% >\ ]$$

    $$.o.$$

    $$[.\ .]\ -\ >\ \ \%\hat{}\ ||\ \_A$$

    where $[.\ .]\ -\ >\ \ LOWER\ ||\ LEFT\_RIGHT$ is a version of empty string replacement that allows only one application between any LEFT and RIGHT. The construction for $[.\ .]\ -\ >$ $LOWER\ ||\ LEFT\_RIGHT$ is the same as for $UPPER\ -\ >\ \ LOWER\ ||\ LEFT\_RIGHT$ except that $Replace = [\% >\ \ \% <\ \ -\ >\ \ \% <\ \ LOWER\ \% >]$.

- *LeftToRight* =

$$[\sim \$[\%\hat{}]\,[\%\hat{} : \% < \;\; UPPER' \;\; 0 : \% >]\,]* \;\sim \$[\%\hat{}]$$

$$.o.$$

$$\%\hat{} - > \; [\,]$$

  where $UPPER' = [\,A/[\%\hat{}] - [?* \;\%\hat{}]\,]$
- $LongestMatch = \sim \$[\;\% < \;[\,UPPER'' \;\&\; \$[\% >]\,]\,]$
  where $UPPER'' = \; A/[\% < \; | \; \% >] \; - \; [\; ?* \;[\; \% < \; | \; \% >]\,]$
- $Insrtion = \;\; \% < \; - > \; L \;, \; \% > \; - > \; R$

The rest of the markup rules were not implemented since we could not obtain any documentation of their constructions. These operators are:

- XFST syntax: $A@ > L...R$

  Meaning: Directed markup. Instances of the language A on the upper side of the relation are selected for markup under left-to-right, shortest match regimen. Thus, the starting locations are selected from left-to-right. If there are overlapping instances of A starting at the same location, only the shortest one is replaced by all strings of B. Each selected A string is marked by inserting all strings of R to its left and all strings of S to its right. The selected A strings themselves remain unchanged, along with the non-A segments.

- XFST syntax: $A- > @L...R$

  Meaning: Directed markup. Instances of the language A on the upper side of the relation are selected for markup under right-to-left, longest match regimen. Thus, the starting locations are selected from right-to-left. If there are overlapping instances of A starting at the same location, only the longest one is replaced by all strings of B. Each selected A string is marked by inserting all strings of R to its left and all strings of S to its right. The selected A strings themselves remain unchanged, along with the non-A segments.

- XFST syntax: $A > @L...R$

  Meaning: Directed markup. Instances of the language A on the upper side of the relation are selected for markup under right-to-left, shortest match regimen. Thus, the starting locations are selected from right-to-left. If there are overlapping instances of A starting at the same location, only the shortest one is replaced by all strings of B. Each selected A string is marked by inserting all strings of R to its left and all strings of S to its right. The selected A strings themselves remain unchanged, along with the non-A segments.

### A.6 Boundary symbol for restriction and replacement

In the restriction, $=>$, and conditional replacement, $->, (->), <-, (<-), <->, (<->), @->, @>$ $, -> @, > @$ expressions we can use a special boundary marker, .#., to refer to the beginning or to the end of a string. In the left context, the boundary marker signals the beginning of the string; in the right context it means the end of the string.

Construction: We do not deal with all the cases where the boundary symbol .#. can be used. We only deal with boundary cases contexts that are in one of the following forms (*LeftContext* and *RightContext* are assumed not to contain .#.):

- .#. *LeftContext* _*RightContext*

- $[.\#.\ \mathit{LeftContext}]\ \_\ \mathit{RightContext}$
- $[.\#.]\ \mathit{LeftContext}\ \_\ \mathit{RightContext}$
- $[[.\#.]\ \mathit{LeftContext}]\ \_\ \mathit{RightContext}$
- $\mathit{LeftContext}\ \_\ \mathit{RightContext}\ .\#.$
- $\mathit{LeftContext}\ \_\ [\mathit{RightContext}\ .\#.]$
- $\mathit{LeftContext}\ \_\ \mathit{RightContext}\ [.\#.]$
- $\mathit{LeftContext}\ \_\ [\mathit{RightContext}\ [.\#.]]$
- $.\#.\ \mathit{LeftContext}\ \_\ \mathit{RightContext}\ .\#.$
- $[.\#.\ \mathit{LeftContext}]\ \_\ \mathit{RightContext}\ .\#.$
- $[.\#.]\ \mathit{LeftContext}\ \_\ \mathit{RightContext}\ .\#.$
- $[[.\#.]\ \mathit{LeftContext}]\ \_\ \mathit{RightContext}\ .\#.$
- $.\#.\ \mathit{LeftContext}\ \_\ [\mathit{RightContext}\ .\#.]$
- $[.\#.\ \mathit{LeftContext}]\ \_\ [\mathit{RightContext}\ .\#.]$
- $[.\#.]\ \mathit{LeftContext}\ \_\ [\mathit{RightContext}\ .\#.]$
- $[[.\#.]\ \mathit{LeftContext}]\ \_\ [\mathit{RightContext}\ .\#.]$
- $.\#.\ \mathit{LeftContext}\ \_\ \mathit{RightContext}\ [.\#.]$
- $[.\#.\ \mathit{LeftContext}]\ \_\ \mathit{RightContext}\ [.\#.]$
- $[.\#.]\ \mathit{LeftContext}\ \_\ \mathit{RightContext}\ [.\#.]$
- $[[.\#.]\ \mathit{LeftContext}]\ \_\ \mathit{RightContext}\ [.\#.]$
- $.\#.\ \mathit{LeftContext}\ \_\ [\mathit{RightContext}\ [.\#.]]$
- $[.\#.\ \mathit{LeftContext}]\ \_\ [\mathit{RightContext}\ [.\#.]]$
- $[.\#.]\ \mathit{LeftContext}\ \_\ [\mathit{RightContext}\ [.\#.]]$
- $[[.\#.]\ \mathit{LeftContext}]\ \_\ [\mathit{RightContext}\ [.\#.]]$

As we do not deal with restriction rules we need to deal with boundary cases only in replace rules. The replace rules were constructed from six stages: InsertBrackets, ConstrainBrackets, LeftContext, RightContext, Replace and RemoveBrackets. In boundary cases where the left context is in the beginning of a string, only the LeftContext stage is changed. The LeftContext stage was defined as

$$\mathit{LeftContext} = [\ \sim [\ \sim [...\mathit{LEFT}]\ \ [<...]\ ]\ \&\ \sim [\ [...\mathit{LEFT}]\ \ \sim [<...]\ ]\ ]$$

where $[...\mathit{LEFT}]$ denoted

$$[\ [\ ?*\ L/[\% <\ |\ \% >]\ ] - [?*\ \% <]\ ]$$

and $[< ...]$ denoted

$$[\% < /\% > ?*]$$

The definition of LeftContext is not changed but the definition of $[...LEFT]$ is changed into

$$[\,[\,L/[\% < \ | \ \% >]\,] - [?* \ \% <]\,]$$

In boundary cases where the right context is at the end of a string, only the RightContext stage is changed. The RightContext stage was defined as

$$RightContext = [\, \sim [\,[...>] \ \ \sim [RIGHT...]\,] \ \& \ \sim [\, \sim [...>] \ [RIGHT...]\,]\,]$$

where $[RIGHT...]$ denoted

$$[\,[\,R/[\% < \ | \ \% >] \ \ ?*\,] - [\% < ?*]\,]$$

and $[...>]$ denoted

$$[?* \ \% > /\% <]$$

The definition of RightContext is not changed but the definition of $[RIGHT...]$ is changed into

$$[\,[\,R/[\% < \ | \ \% >]\,] - [\% < ?*]\,]$$

In boundary cases where both the right context and the left context are at the end and in the beginning of a string respectively, both the RightContext and the LeftContext stages are changed as described above. The idea behind these changes is that the context part of replacement expression can be actually seen as $?* \ LEFT \ \_ \ RIGHT \ ?*$ and by simply eliminating one of the $?*$ in one of the ends we can relate to a boundary case. The definitions that were changed above did exactly that: eliminated the appropriate $?*$ for each case. More complicated cases, for example $a-> b \,||\, [.\#. \mid a] \ \_$ should be dealt by conditional parallel replacement. For example, $a-> b \,||\, [.\#. \mid a] \ \_$, should be interpreted as

$$a-> b \,||\, [.\#.] \ \_ \ ,, \ a-> b \,||\, [a] \ \_$$

Since we do not deal with conditional parallel replacement, we cannot deal with these cases.

## A.7 Order of Precedence

### A.7.1 XFST

The following list defines the order of precedence of all XFST operators. Operators of same precedence are evaluated from left to right, except the prefix operators (~ \ $ $? $.) that are evaluated from right to left. The list begins with the operators of highest precedence, i.e., with the most tightly binding ones. Operators of same precedence are on the same line.

```
:
~ \ $ $? $.
+ * ^ .1 .2 .u .l .i .r
/
concatenation
> <
| & -
=> -> (->) <- (<-) <-> (<->) @-> @> ->@ >@
.x. .o.
```

### A.7.2 FSA

The following list defines the order of precedence in FSA:

: /
..
+ * ^
& −
*o x xx*
! #

## A.8 Advanced techniques

Both XFST and FSA have advanced techniques that do no exist in the other toolbox. For XFST these techniques include Compile-Replace and Flag-Diacritics; for FSA these techniques include predicates and weighted networks.

# Author Index