Final Report
of the
2005 Language Engineering Workshop
on

Statistical Machine Translation by Parsing

Andrea Burbank, Marine Carpuat, Stephen Clark, Markus Dreyer,
Pamela Fox, Declan Groves, Keith Hall, Mary Hearne, I. Dan Melamed,
Yihai Shen, Andy Way, Ben Wellington, and Dekai Wu

Johns Hopkins University
Center for Speech and Language Processing

`http://www.clsp.jhu.edu/ws2005/groups/statistical/`

November 12, 2005

Abstract

Designers of SMT system have begun to experiment with tree-structured translation models. Unfortunately, SMT systems driven by such models are even more difficult to build than the already complicated WFST-based systems. The purpose of our workshop was to lower the barriers to entry into research involving such SMT systems. Our goals were inspired by the successful 1999 MT workshop, which had a similar purpose. Specifically, we wanted to follow that precedent to

1. build a publicly available toolkit for experimenting with tree-structured translation models;
2. build a tool for visualizing the predictions of such models;
3. demonstrate the feasibility of SMT with tree-structured models by running baseline experiments on large datasets; and
4. demonstrate that it is easy to retarget the toolkit to new language pairs.

We also wanted to preemptively address some of the criticisms of the toolkit released in 1999 by

5. making our toolkit turn-key, i.e. not requiring any additional software for end-to-end operation;
6. providing extensive documentation; and
7. providing integrated prototype systems, to save the toolkit's users most of the effort of system integration.

We have largely achieved these goals. Our toolkit is downloadable from
`http://www.clsp.jhu.edu/ws2005/groups/statistical/GenPar.html` .
The visualization tool is available separately from
`http://www.clsp.jhu.edu/ws2005/groups/statistical/mtv.html` .

We also ran some pilot experiments. Based on these experiments, we can make the following claims:

- Our models can handle language pairs with divergent syntax, such as English and Arabic.
- It is straightforward to improve our joint translation models by using target language models, the same way it is done in the noisy-channel paradigm.
- Our software scales to hundreds of thousands of sentence pairs, without much optimization, on commodity hardware.
- A rudimentary system for SMT by Parsing can achieve the same level of accuracy as a rudimentary WFST-based system with the same target language model.
- The GenPar toolkit makes it easy to run tightly controlled experiments, and to integrate additional external software.

Acknowledgments, in roughly chronological order

- Josh Rosenblum, Noah Smith, and Svetlana Stenchikova, for the genesis of the "common" containers
- Wei Wang, for initial work on GenPar, and for the first draft of Chapter 3
- Kristo Kirov and Thomas Tornsey-Weir, for initial work on MTV
- Laura Graham and Sue Porterfield, for help with logistics
- Fred Jelinek, for funding us
- Victoria Fossum, for help with simplifying data structures
- Chris Pike, for help implementing the LearnableGrammar interfaces
- Ali Argyle, for help with organizing the 2nd planning meeting
- Eiwe Lingefors, for systems support at JHU
- Mona Diab and Nizar Habash, for help with Arabic tools
- Dan Bikel, for help with his parser, and for giving us a special license
- Abhishek Arun and Frank Keller, for help with the French treebank
- Charles Schafer, for help with running GIZA++ baselines

This material is based upon work supported by the U.S. National Science Foundation under grants numbered 0121285 and 0238406. Any opinions, findings, conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of the National Science Foundation.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

Since approximately 2001, the state of the art in machine translation has been defined by statistical machine translation (SMT) systems. Today's best SMT systems are weighted finite-state transducers (WFSTs) of the "phrase"-based variety, meaning that they memorize the translations of word n-grams, rather than just single words. Translating strings of multiple words as a unit is beneficial in two ways. First, the translations of individual words are more likely to be correct when they are translated together with the context of the words around them. Second, phrases can capture local variations in word order, making the decoder's job easier. Despite these attractive properties, SMT based on WFSTs is inherently limited in its scientific and engineering potential.

Major scientific advances often come from deep intuitions about the relationship of models to the phenomena being modeled. WFST-based translation models run counter to our intuitions about how expressions in different languages are related. In the short term, SMT research based on WFSTs may be a necessary stepping stone. In the long term, the price of implausible models is reduced insight, and therefore slower progress.

From an engineering point of view, modeling translational equivalence using WFSTs is like approximating a high-order polynomial with line segments. Given enough parameters, the approximation can be arbitrarily good. In practice, the number of parameters that can be reliably estimated is limited either by the amount of available training data or by the available computing resources. Suitable training data will always be limited for most of the world's languages. On the other hand, for resource-rich language pairs where the available training data is practically infinite, the limiting factor is the number of model parameters that fit into our computers' memories. Either way, the relatively low expressive power of WFSTs limits the quality of SMT systems.

To advance the state of the art, SMT system designers have begun to experiment with tree-structured translation models [e.g., Wu, 1995a,b,c, Alshawi, 1996, Yamada and Knight, 2002, Gildea, 2003, Hearne and Way, 2003, Chiang, 2005, Hearne, 2005]. Tree-structured translation models have the potential to encode more information using fewer parameters. For example, suppose we wish to express the preference for adjectives to follow nouns in language L1 and to precede them in language L2. A model that knows about parts of speech needs only one parameter to record this binary preference. Some finite-state translation models can encode parts of speech and other word classes [Och et al., 1999]. However, they cannot encode the preferred relative order of noun *phrases* and adjectival *phrases*, because this kind of knowledge involves parameters over recursive structures. To encode such knowledge, a model must be at least tree-structured. For example, a syntax-directed transduction grammar (SDTG) [Aho and Ullman, 1969] needs only one

parameter to know that an English noun phrase of the form (Det AdjP N) such as "the green and blue shirt" translates to Arabic in the order (Det N AdjP). A well-known principle of machine learning is that, everything else being equal, models with fewer parameters are more likely to make accurate predictions on previously unseen data.

Several authors have added tree-structured models to systems that were primarily based on WFSTs [Eng et al., 2003, Koehn et al., 2003]. Such a system can be easier to build, especially given pre-existing software for WFST-based SMT. However, such a system cannot reach the potential efficiency of a tree-structured translation model, because it is still saddled with the large number of parameters required by the underlying WFSTs. Although such hybrid systems are improving all the time, one cannot help but wonder how much faster they would improve if they were to shed their historical baggage. To realize the full potential of tree-structured models, an SMT system must use them as the primary models in every stage of its operation, including training, application to new inputs, and evaluation. Switching to a less efficient model at any stage can result in an explosion in the number of parameters necessary to encode the same information. If the resulting model no longer fits in memory, then the system is forced to lose information, and thus also accuracy.[1] Even when memory is not an issue, the increased number of parameters risks an increase in generalization error.

The above considerations, among others, are motivating SMT researchers to build systems whose every process is driven primarily by tree-structured models. Unfortunately, such systems are even more difficult to build than the already complicated WFST-based systems. The current economics of research in this area are very similar to what they were for the SMT approach overall in 1999. Here is what Al-Onaizan et al. [1999, p.2] wrote at that time:

> "Common software tools ... are not generally available. It requires a great deal of work to build the necessary software infrastructure for experimentation in this area. Such infrastructure is available, for example, in the speech recognition community. It is possible for an individual researcher to
>
> 1. come up with a new idea ...
> 2. do some initial studies ...
> 3. test the new idea [in] the context of an end-to-end speech recognition task ...
> 4. revise the idea
> 5. and compare the final results to previous work.
>
> It is far more difficult to do this sort of thing in machine translation – in fact, it is seldom done. In practice a huge proportion of the work would need to be spent on large-scale MT software development, rather than on inventing and revising the new idea itself. Of course, evaluating machine translation is harder than evaluating speech recognition word-error rate, and this complicated the picture substantially. However, the lack of tools and data sets remains the biggest barrier to entry in the field."

Al-Onaizan et al. [1999] were writing about SMT in general, to explain their main motivation for the 1999 JHU summer workshop on SMT. It is remarkable that their exact words can be used to describe today's barriers to research in tree-structured SMT. These barriers are particularly

[1] An alternative is to swap the model out to secondary storage, slowing down the system by several orders of magnitude.

high for smaller research groups, who lack the human resources "to build the necessary software infrastructure." Our estimate is that, out of the hundreds of research groups in the world who are interested in this area of research, fewer than a dozen can afford that investment.

To remedy the situation in 1999, Al-Onaizan et al. [1999]'s primary goal was to release a software toolkit for SMT. By all accounts, this effort was spectacularly successful. The EGYPT toolkit and its descendants opened the doors to legions of new SMT researchers, and catalyzed the dominance of the statistical approach. The toolkit was popular not only because it commoditized SMT software, but also because it was demonstrably easy to retarget to new language pairs, and included a tool for visualization. EGYPT was not perfect, however. The most common complaints were that it (a) did not include a decoder, (b) was poorly documented, and as a result (c) was difficult to integrate with other software.

The goals for our 2005 workshop were inspired by the successes and criticisms of the 1999 workshop. Specifically, we wanted to follow in the footsteps of Al-Onaizan et al. [1999] to

1. build a publicly available toolkit for experimenting with tree-structured translation models;
2. build a tool for visualizing the predictions of such models;
3. demonstrate the feasibility of SMT with tree-structured models by running baseline experiments on large datasets; and
4. demonstrate that it is easy to port the toolkit to new language pairs.

We also wanted to improve on EGYPT by

5. making our toolkit turn-key, i.e. not requiring any additional software for end-to-end operation;
6. providing extensive documentation; and
7. providing complete integrated prototypes, to save the toolkit's users most of the effort of system integration.

We have largely achieved these goals. Our toolkit is downloadable from

`http://www.clsp.jhu.edu/ws2005/groups/statistical/GenPar.html` .

The visualization tool is available separately from

`http://www.clsp.jhu.edu/ws2005/groups/statistical/mtv.html` .

Chapter 2 of this report is a user guide to the GenPar toolkit. As detailed in that chapter, the toolkit includes end-to-end SMT systems for 3 language pairs, all of which use the same programs. Thus we have satisfied goals 4 and 7 above. These systems are rudimentary, but they are complete, including modules for data preparation, training, testing, and evaluation. Chapter 3 documents GenPar's software design. The GenPar source code also includes extensive in-line documentation, formatted for the Doxygen automatic documentation generator. Thus, our toolkit has the three levels of documentation that are standard in quality-controlled industrial software projects: user documentation, design documentation, and code documentation. Chapter 4 is a user guide for the multitree visualization (MTV) tool. Chapter 5 describes our baseline experiments. **N.B.: Chapters 2, 3, and 4 pertain to GenPar-1.1.0 and MTV-1.0.2. Later software releases will contain documentation that is more up to date.** The appendices contain our file formats and a glossary of terms used in this report.

In addition to releasing the above-mentioned software, we ran some baseline experiments. These were only baselines: We did not have time to compare the many possible configurations of our toolkit, let alone to improve it. Even so, these baselines enable us to make the following claims:

- Our models can handle language pairs with divergent syntax, such as English and Arabic. Of course, they can also handle languages from the same family, such as English and French.
- It is straightforward to improve our joint translation models with target language models, the same way as it is done in the noisy-channel paradigm.
- Our software scales to hundreds of thousands of sentence on ordinary hardware. This scale was reached without extensive optimization. So, we are confident that further engineering will allow GenPar to process the same kind and size of data that is used to build today's best SMT systems.
- A rudimentary system for SMT by Parsing can achieve the same level of accuracy as a rudimentary WFST-based system with the same language model. As a point of comparison, the baseline experiments of Al-Onaizan et al. [1999] also gaged the accuracy of their initial system at a level comparable to an off-the-shelf representative of the preceding MT paradigm.
- The GenPar toolkit makes it easy to run tightly controlled experiments, and to integrate additional external software.

Chapter 2

GenPar User Guide

N.B.: This chapter pertains to GenPar-1.1.0. Later releases of the toolkit will contain documentation that is more up to date.

2.1 Introduction

The best place for new users to start familiarizing themselves with GenPar is in GenPar's *sandbox*. This holds both for users who intend to modify GenPar's core components and for those who don't. The sandbox is one of the main features of the GenPar toolkit. The sandbox comprises several prototype end-to-end systems for statistical machine translation by parsing. Each prototype contains the software modules and configuration files necessary for training, testing, and evaluation with a particular language pair. The sandbox has three purposes: education, validation, and infatuation:

- The sandbox serves an educational purpose. A new user can run some data through the different stages of system training, application, and evaluation, to get a sense of how these pieces fit together in the 'Translation by Parsing' architecture. For this purpose, we also recommend the Multitree Viewer (MTV) visualization tool that was developed alongside GenPar. See `http://www.clsp.jhu.edu/ws2005/groups/statistical/mtv.html` for more information.

- The sandbox functions as a validation suite for the software. Each prototype in the sandbox contains sample data, along with the expected output for that data as it is run through each of the prototype modules. When a change is made in the code, and the sandbox is rerun, it will report any differences between the output and the expected output. It's a good idea to use the sandbox in this manner regularly during software development, to help catch bugs.

- Eventually, users become infatuated with the GenPar toolkit, and they want to use it for everything. At that point, the sandbox can serve as the blueprint for a larger system. One of its prototypes can be copied to a new location on disk, and used on new data sets large and small.

With the GenPar toolkit, we have provided prototypes for three different language pairs: Arabic-English, French-English and English-English. Each of these prototypes conforms to the overall sandbox description given in Section 2.2, which gives an overview of all of the modules in the

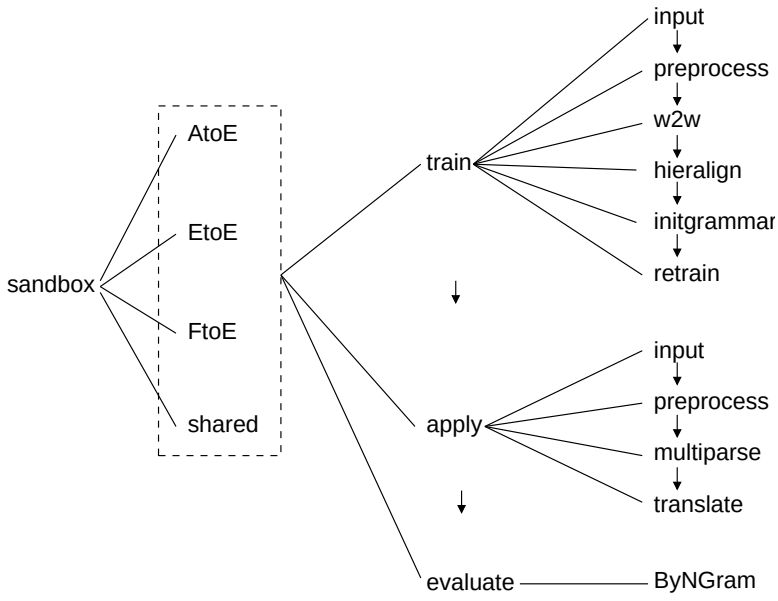

Figure 2.1: Sandbox directory structure and default order of execution.

sandbox, their directory structure, and the flow of data when all of the modules of the sandbox are run in the default order. Section 2.3 explains the 3 main ways that you can use the sandbox, corresponding to its 3 purposes above. In Section 2.4 we discuss sandbox configuration files.

Important: The code distributed with GenPar is sufficient to run all but the `preprocess` modules. The default configuration of the sandbox is to use previously preprocessed data sets that were shipped with the software, so it is sufficient for educational and development purposes. However, the preprocessing modules are necessary for using the system on new data sets. Additional language-specific software is necessary in order to run these additional modules. The INSTALL-ext file describes how to obtain and install the additional software.

2.2 Sandbox Contents

The `sandbox` directory contains four subdirectories. Three of them are prototypes for specific language pairs. The fourth subdiretory, called `shared`, stores common elements of the three prototypes. To reduce code duplication and maintenance effort, many of the files controlling the operation of the prototypes are just symbolic links to files in `shared`. Each prototype conforms to the sandbox directory structure illustrated in Figure 2.1. Broadly speaking, the `train` directory of each prototype is responsible for extracting a translation model from a set of sentence pairs, the `apply` directory uses that model to perform multiparsing or translation, and the `evaluate` directory judges the output of `apply` on some objective creteria.

The `train`, `apply`, and `evaluate` directories contain 6, 4, and 1 subdirectories, respectively. Except for the `input` subdirectories, each subdirectory corresponds to a prototype **module**. The arrows in Figure 2.1 show the default order of module execution. Figure 2.2 shows how the various kinds of data in each prototype are manipulated by each module: a solid arrow from `x` to `y` indicates

that x provides y, a dotted arrow from y to x indicates that y is used by x and a dashed double arrow between x and y indicates that x changes y.

The tasks carried out by each train module are as follows:

- train/input does not run any code. Files L1.text and L2.text contain the source and target training sentences respectively. It is assumed that the text on the n^{th} line of L1.text is translationally equivalent to the text on the n^{th} line of L2.text. Generally, the text on each line should correspond to a single sentence, but this is not a strict requirement.

- train/preprocess is responsible for all the monolingual preprocessing necessary for the data in train/input. Usually, this includes tokenization. It may or may not include monolingual parsing. The tokenized sentence pairs are placed in file output.snt. The source language parse trees, if any, are placed in file L1.tb and the target language parse trees in L2.tb.

- train/w2w induces a word-to-word translation model over the tokenized source and target language sentence pairs in preprocess/output.snt.[1] The translation model is written to file model_final. From this model, the file links is generated. This file contains word alignments for every sentence pair in the training set.

- train/hieralign takes as input the source and target sentence pairs in preprocess/output.snt, the source parse trees in preprocess/L1.tb, the target parse trees in preprocess/L2.tb and the links file generated by train/w2w. From these, it runs code to generate a set of multitrees – hierarchically aligned tree pairs – written to the file tb.out.

- train/initgrammar executes the code responsible for extracting a Weighted Multitext Grammar (WMTG) from hieralign/tb.out; this grammar is written to file pmtg.mle.

- train/retrain refines the grammar created by train/initgrammar, writing this refined grammar to pmtg_final.

The tasks carried out by each apply module are as follows:

- apply/input does not run any code. Files L1.text and L2.text contain the source and target test sentences respectively. The target test sentences are necessary if you wish to multiparse rather than translate. They are also used as references by the evaluate modules.

- apply/preprocess is responsible for tokenizing the source language strings and the target language strings. The tokenized sentence pairs are placed in file output.snt and the tokenized source sentences are placed in file output.1D.snt.

- apply/multiparse takes as input the file preprocess/output.snt, which contains pairs of sentences, and the grammar in file train/retrain/pmtg_final. It executes code to assign a multitree to each of the test sentence pairs; these multitrees are output to file tb.out.

- apply/translate takes as input the file preprocess/output.1D.snt, which contains the source test sentences, and the grammar in

[1] Currently, the GenPar toolkit includes two software systems for this purpose: the GIZA++ system by Franz Och, and our homegrown 'w2w' system. Each system is used in at least one prototype.

file `train/retrain/pmtg_final`. It executes code to assign a multitree to each of those sentences, where each multitree expresses the target-language string in its leaves. The multitrees are written to file `tb.out`, the pairs of sentences expressed by the multitrees are written to file `translations.snt` and the target sentences expressed by the multitrees are written to file `translations.txt`.

At the time of writing, the sandbox includes only one evaluation procedure, which is based on n-gram matching with a reference translation. `evaluate/ByNgram` takes the output translations from `apply/translate/translations.txt` and the reference translations in `apply/input/L2.text` and passes them to the GTM tool for MT evaluation. The scores assigned are written to files `evaluate/ByNGram/Fmeasure1` and `evaluate/ByNGram/Fmeasure2`, which correspond to GTM runs with exponents 1 and 2. See `http://nlp.cs.nyu.edu/GTM` for more information.

2.3 Using the sandbox

Once you are in the `sandbox` directory, there are four main[2] commands you can execute to run the system.

- Typing `make run` causes all of the modules of the sandbox to run in sequence.
- Typing `make valid` also causes all modules to run. In addition, it compares the actual output and expected output for each module, using the `*.expected` files in each module, and reports an error if there is a mismatch at any point.
- If you have already run the sandbox, and you want to run it again from scratch, you can return it to its original state with the `make clean` command.
- There is also a `make expected` command. If you type `make expected` in the sandbox, it will overwrite all the `*.expected` files to match your current output, setting the new standard for validation. This means all future output will be compared to your current output. This command is usually used after changing the code. It should be used with extreme caution and only after a thorough investigation of what the differences are between the current output and the expected files.

Typing any of these commands from the top-level `sandbox` directory will cause each of the three prototypes contained in that directory to run — these are directories `AtoE`, `EtoE` and `FtoE` as shown in Figure 2.1. However, by moving into any one of these three prototype directories and typing a `make` command, just that particular prototype will run. Similarly, any module of a prototype can be run individually over a particular dataset by simply going to the directory for that prototype and module and executing a `make` command.

2.4 Configuration Files

Three prototype modules – `train/hieralign`, `apply/multiparse` and `apply/translate` – execute the program `gp`. However, these modules perform different tasks because they each specify a distinct set of configurations according to which `gp` is run. Each prototype module directory contains a

[2]There are a couple of other utility commands that we didn't get around to documenting — details in the Makefiles.

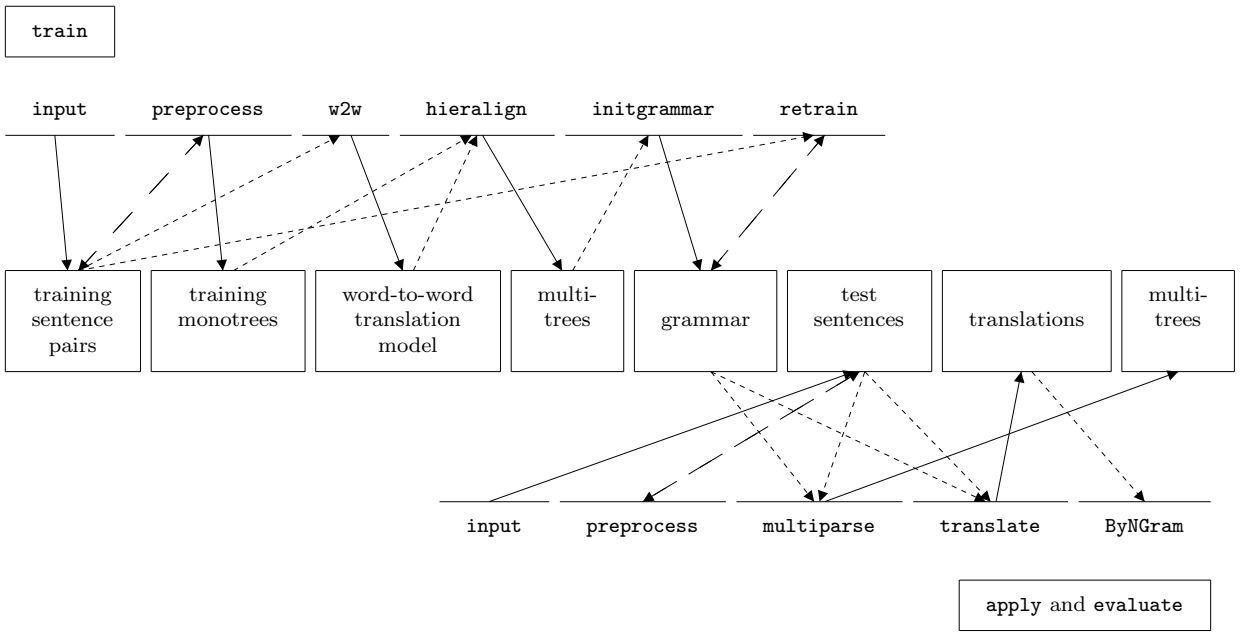

Figure 2.2: Prototype data flow. A solid arrow from `x` to `y` indicates that `x` provides `y`, a dotted arrow from `y` to `x` indicates that `y` is used by `x` and a dashed double arrow between `x` and `y` indicates that `x` changes `y`.

subdirectory called `config` in which the files specifying the required configuration are stored. In this section we outline some of the configuration options for `gp`, many of which are also used by other GenPar executables such as `viterbiTrain`. However, note that the configuration files for `gp` and the other GenPar programs were designed to be very flexible, and even their relation to each other can be changed without recompiling.

Figure 2.3 shows the `gp` configuration file hierarchy that is used in the prototypes distributed with the toolkit. This arrangement of configuration files happens to be almost the same for all invocations of `gp` in the sandbox, but this need not be the case in general. Each box in the hierarchy represents a config file, the name of which is given at the top of the box. The remainder of the box gives the options to be set for that file. Options given in bold take the name of another config file as their value. Each module using `gp` has a main config file – the topmost file in the hierarchy – from which all the other config files are specified. The only optional config file shown in figure 2.3 is `SubGrammar.config`, which is used in the default configuration of `hieralign` modules only.

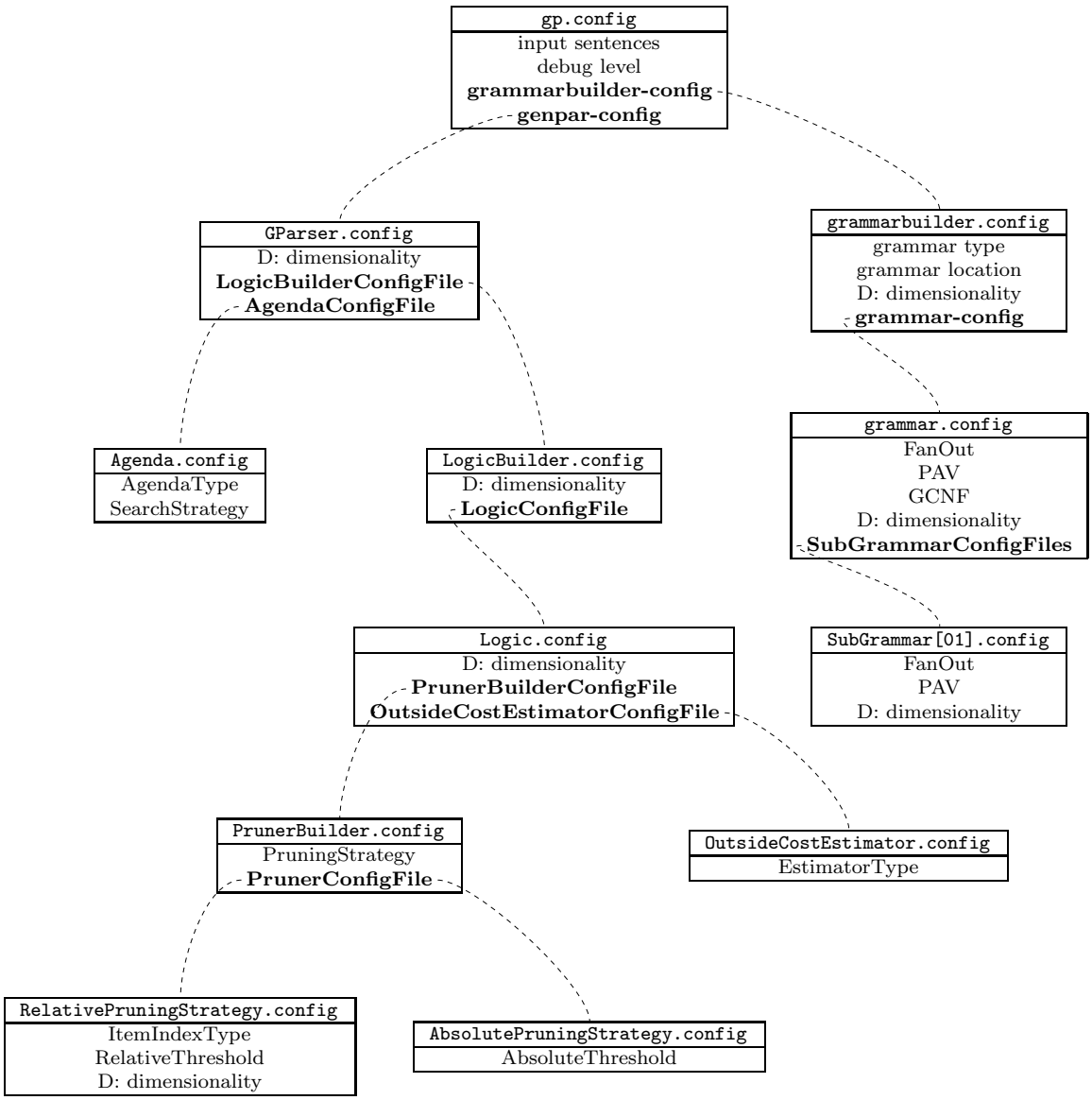

Figure 2.3: gp configuration file hierarchy. Each box in the hierarchy represents a config file, the name of which is given at the top of the box. The remainder of the box gives the options to be set for that file. Options given in bold take the name of another config file as their value.

Chapter 3

GenPar Design

N.B.: This chapter pertains to GenPar-1.1.0. Later releases of the toolkit will contain documentation that is more up to date.

3.1 Introduction

We present an object-oriented design of a software toolkit for generalized parsing. The design is based on the architecture laid out by Melamed and Wang [2005]. It is helpful but not mandatory to understand the architecture before attempting to understand the design. The design has two goals:

1. *flexibility*: The toolkit should support many parser variants, and should be easily configurable into one of them at run time.

2. *extensibility*: It should be easy to add new features and new functionality.

To satisfy the above requirements, we decompose the parser into different components and let the Parser class act as a "central controller". Each type of component is defined by an object class family. The parser refers to each component via a pointer[1] to the abstract type of the component and the concrete component type is allocated during the parser's construction. A better decomposition of the parser helps the parser to support more variations and to be easily configured on demand. It also makes the parser more easily extensible. To support a new component, one can merely add a new class to define the new component.

The parser implements the generic parsing algorithm in Figure 3.1. Almost all parsing algorithms ever published can be viewed as special cases of this parsing algorithm. The input of the algorithm is a logic, a grammar, a search strategy, a semiring, a termination condition, and a tuple of sentences to parse. The output is a semiring value, such as a Viterbi parse tree.[2] The main loop proceeds as follows. If the agenda is empty, indicating that the parsing has just started, the logic generates the first batch of inferences to initialize the agenda. These inferences are based on the logic's axioms.[3] In subsequent iterations of the main loop, the parser first pops an item from the agenda called the "trigger", and then passes that item to the logic. Taking this trigger, the

[1] Some of the components are references, in the C++ sense, but we call them pointers for simplicity.
[2] Under other semirings, we might return a parse forest, or just a score.
[3] Such terms are defined in the glossary.

Input: `tuple of sentences`, `Grammar`, `Logic`, `Comparator`, `TerminationCondition`

```
1: Agenda(Comparator).clear()
2: Item e = NULL;
3: repeat
4:   if not Agenda.empty() then
5:     Item e ← Agenda.pop()
6:   set⟨Item⟩ I ← Logic.expand(e, Grammar)
7:   for Item i ∈ I do
8:     Agenda.push(i)
9: until TerminationCondition is achieved
```

Output: a `semiring` value, such as a `parse tree`

Figure 3.1: Generic parsing algorithm, a special case of the abstract parsing algorithm presented by Melamed and Wang [2005].

logic decides whether the item should be pruned with the help of a pruning strategy. If the item survives, new items are generated according to the constraints of the grammar. These items are then pushed onto the agenda. If the parsing goal is achieved, the loop terminates.

To support the generic parsing algorithm, a parser can be decomposed into the following major abstract components.

- **Logic**: a set of inference type signatures, a set of item type signatures and a set of axioms.
- **Grammar** : an evaluator of parse structures.
- **Semiring** : specifying what type of values the parser computes, and the relevant operators for computing them.[4]
- **Search strategy**: how the inferences on the agenda are scheduled.
- **Termination Condition**: a function that decides when to stop parsing.

There can be many varieties of each component, and each of the components has its own subcomponents. For example, the Logic generates Items for the Chart. Each of these components and subcomponents are described in the following sections. Table 3.1 shows the main classes involved in the parser.

The relationship between the parser and its components is one of aggregation. The idea is that a Parser contains the five main components via a pointer to the abstract base class of each component type. The implementation of the parser is realized by calling abstract operations in component base classes. Figure 3.2 shows how different abstract components make up a Parser. The types of the concrete components are specified via the Parser constructor's arguments.[5] In the figure, there is a member pointer to Logic which will refer to the concrete BottomUpLogic or to some other Logic at runtime, depending on the config files. In addition, there are pointers to other abstract base classes.

[4]Only the Viterbi-derivation semiring is currently implemented.

[5]We pass the configuration information in a config file – see Section 3.4.

class	functionality
Parser	central controller; implements the main parsing algorithm
Logic	the logic (axioms and inference templates) used in the deduction
Grammar	an evaluator of parse structures
Chart	container of items already derived
Agenda	contains new chart items that have not been expanded upon
ParsingStats	specifies what type of values to compute and how to compute them
Item	represents a partial parse
Inference	inference rule in the logic
HeadMTree	headed multitree, used to represent parse trees
HeadProduction	production rule (specific to WMTGs)
SentenceTuple	sentence for which we are going to get its parse
Comparator	compares two items in Agenda
ParseStats	holds diagnostic info from the parse
GrammarIndex	for efficiency of querying production scores
OutsideCostEstimator	for computing outside costs of items

Table 3.1: Prominent classes and class families in the GenPar design

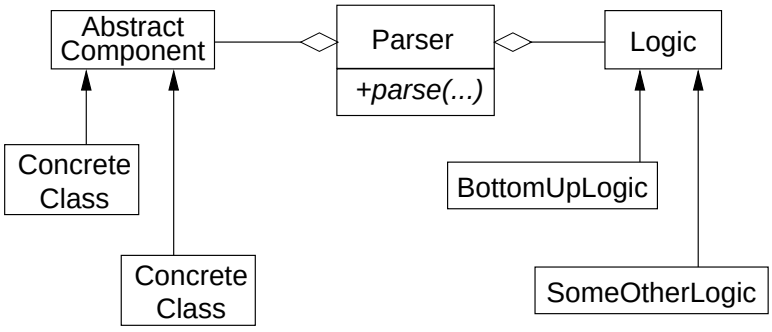

Figure 3.2: Class Parser is the central component of the architecture.

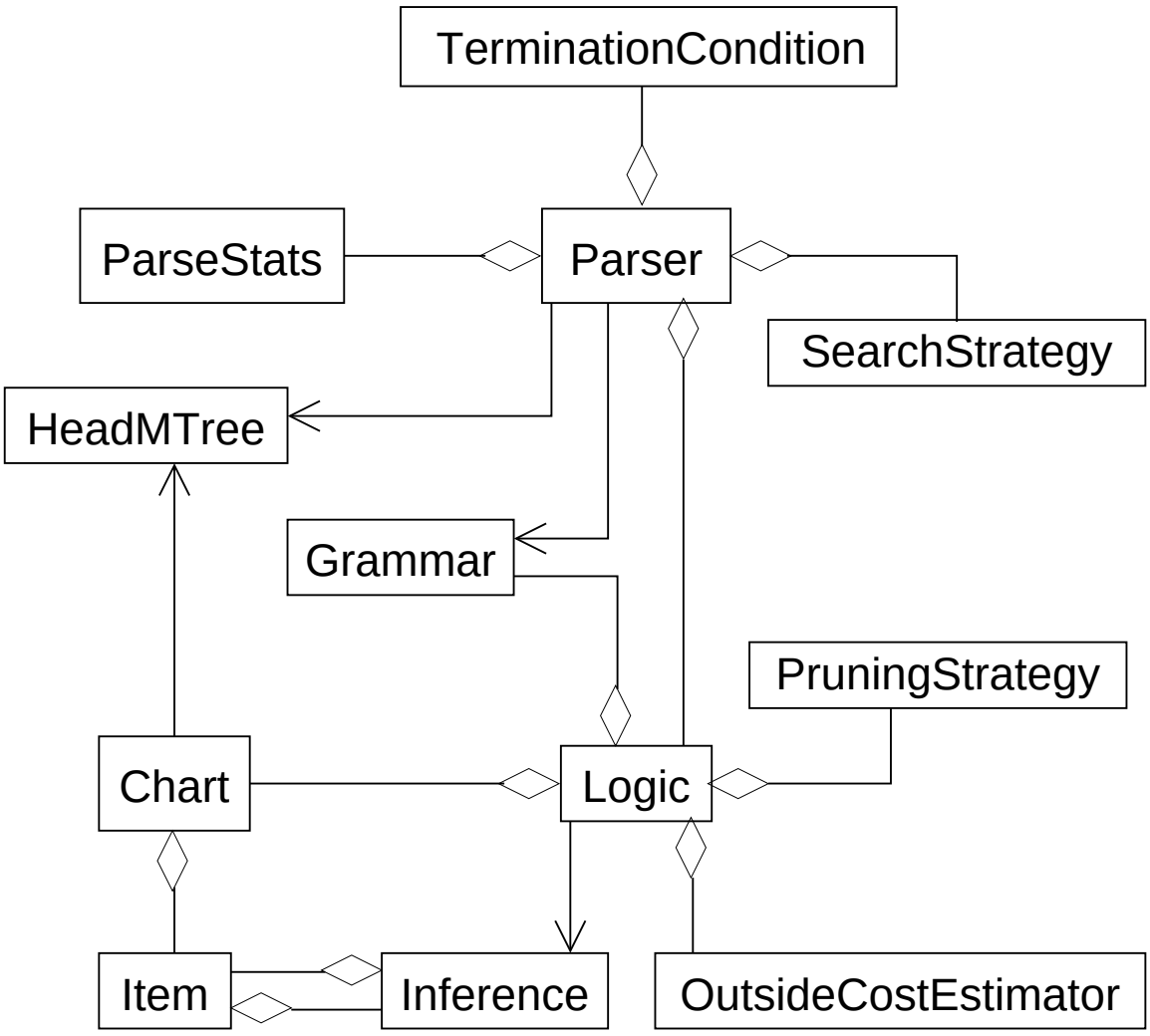

Figure 3.3: Top-level design of the generalized parser

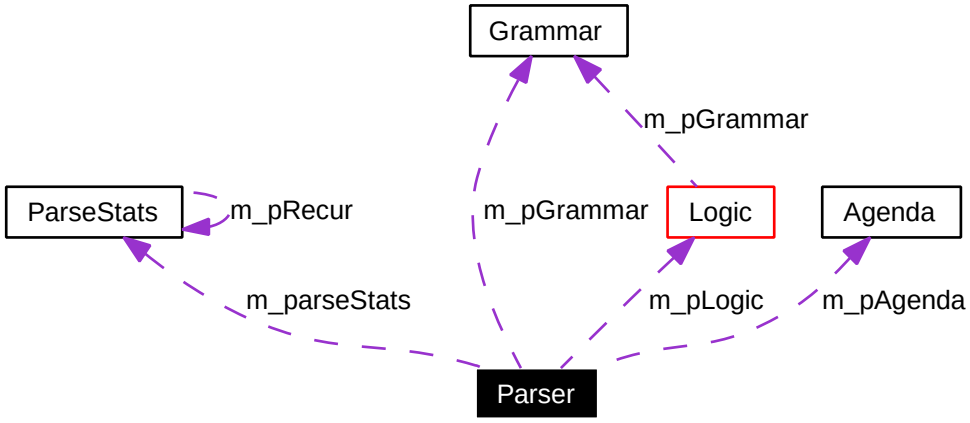

Figure 3.4: Parser Collaboration Diagram

The UML diagram in Figure 3.3 shows the top-level design of the parser. At the center of the diagram is the Parser, and around the Parser are its components. Among the four major parser components, the **grammar** is realized by the Grammar class family. The **logic** corresponds to the Logic class family. The **semiring** is fixed and currently corresponds to no classes. The Viterbi-derivation semiring is currently hardwired into the code, though this should not be difficult to generalize, if it's ever deemed necessary. Finally, the **search strategy** is implemented by the SearchStrategy class family.

3.2 Classes

3.2.1 Parser

The class Parser is the core of the generalized parser. It is a "mediator" of different parser components [Gamma et al., 1994]. A Parser contains pointers to several different objects: A Logic which creates inferences, a Grammar which evaluates the partial parse structures, and an Agenda of items which have not been expanded upon. A Parser also keeps track of the diagnostic info of the parse in ParseStats. The collaboration diagram for this class is shown in Figure 3.4. The Parser runs the generic chart parsing algorithm, as seen in Figure 3.1, and references each of these components when needed during this algorithm.

3.2.2 Grammar

A Grammar is, essentially, an evaluator of parse structures. It decides what parse structures can be inferred and assigns scores to different allowable structures. While traditional grammars in formal language theory have "productions," grammars in the generalized parser do not necessarily need productions, as long as the grammar has some way of assigning scores[6] to partial parses.

The Grammar class family is shown in Figure 3.5. The base class is called Grammar. Since the base should be sufficiently general to encompass all imaginable grammars, it contains very little beyond sets of terminals and nonterminals, and some utility methods. Most of the grammar devel-

[6]These scores are semiring values. This means that the scores can be boolean.

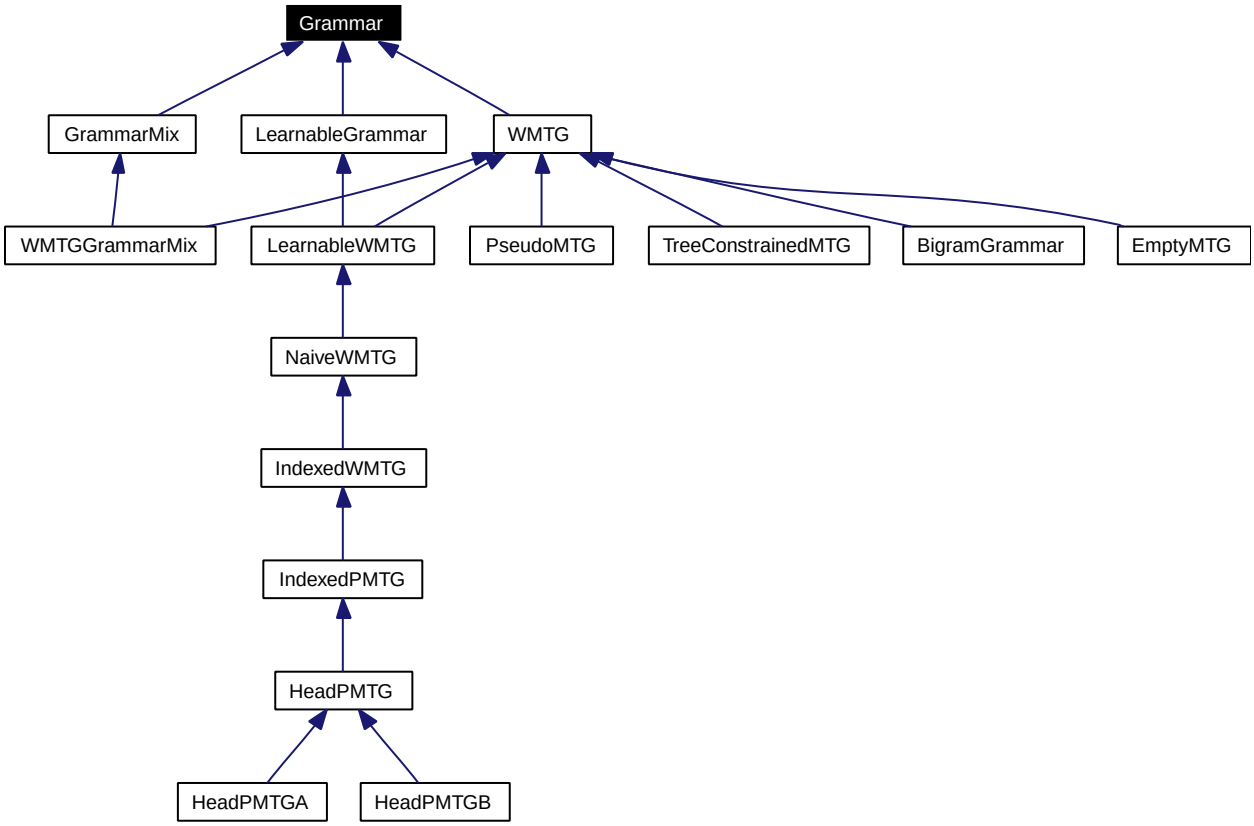

Figure 3.5: Grammar class family

opment to date has focused on (Generalized) Multitext Grammars [Melamed et al., 2004]. WMTG is an abstract class for Weighted Multitext Grammars. Its signature includes methods for querying the standard components of an MTG with weights, as well as for conversion into a Generalized Chomsky Normal Form (GCNF). It also supports query methods that are necessary for bottom-up parsing. The GrammarMix class is an abstract base class for grammar mixtures. Its concrete subclass WMTGGrammarMix is currently used to mix in a target language model for translation, by way of a BigramGrammar. The other direct subclass of Grammar, LearnableGrammar, includes methods required for parameter estimation. The longest chain of WMTG subclasses is the most frequently used. NaiveWMTG is a concrete class that includes a Histogram of HeadProductions. Such a simple structure cannot be queried efficiently, so the IndexedWMTG subclass adds a production rule index. The IndexedPMTG class adds a normalization method to make the grammar probabilistic.

Statistics over production rules are too crude to generalize well to unseen data. To improve generalization, we have designed generative processes that decompose the generation of each production rule into a series of smaller events. The generative processes are stochastic, and they can assign a probability to any parse structure. We therefore call them Probabilistic Multitext Grammars (PMTGs). All the currently implemented PMTGs include information about head-children, so the class that serves as the abstract interface to such grammars is called HeadPMTG. It presents

just a couple of methods for manipulating the events of an arbitrary generative process. The specific PMTGs that are currently implemented are HeadPMTGA and HeadPMTGB. They differ only in whether the precedence arrays in each component of an HeadProduction are generated independently of each other.

A subset of the subclasses of WMTG have no production rules to constrain what antecedent items can compose. At the moment, this set of classes is used for hierarchical alignment. The first grammar in this set, PseudoMTG, is a grammar which is made up of one or more sub-grammars, each applying constraints for only one component at a time. PseudoMTG takes the constraints imposed in each component and decides what the combined constraints are over all components. It also contains a word-to-word model. The implemented sub-grammars that can be used in PseudoMTG are TreeConstrainedMTG, which encodes a parse tree as a constraint, and EmptyMTG, which simply composes any items given to it because it is virtually constraintless. PseudoMTGs offer the flexibility to decide what constraints you want to apply in each dimension.[7] This means you can apply syntactic constraints from a monoparser on all, some, or none of the dimensions of the parser. For example, to apply a tree constraint in only one component, you might perform alignment using a PseudoMTG, with a TreeConstrainedMTG in one component and an EmptyMTG in the other.

3.2.3 Logic

A Logic can be seen as a factory of items. Each concrete Logic subclass is characterized by a different set of inference types. For example, the inference types involved in traditional bottom-up parsing are Compose and Scan; those involved in Earley parsing[8] are Scan, Predict, and Complete.

Figure 3.6 shows the Logic class hierarchy. Four concrete Logic types are currently implemented in the Logic class family. BottomUpTranslationLogic corresponds to LogicCT in [Melamed and Wang, 2005], which uses Scan, Load and Compose inferences. It is a subclass of BottomUpLogic, presented in that paper as LogicC. FasterBottomUpTranslationLogic is a variant that uses a word-to-word translation model to constrain Load inferences. The CKYTranslator is described in Section 3.2.16.

Figure 3.7 shows the collaboration diagram of Logic. The main member variables in class Logic are a Chart, an OutsideCostEstimator and a PruningStrategy. The Chart is in Logic because, formally speaking, Charts are just a bookkeeping device for efficiency, and no other parser component needs to know about them. The OutsideCostEstimator is currently unused but will eventually be moved up to be a component of Parser, rather than of Logic. The PruningStrategy is in Logic because pruning strategies can be viewed as side conditions of inferences, and the Logic is responsible for creating new Items by firing Inferences. A ParsingGoal is often part of the Parser's termination condition. This part of it is implemented in a class contained in the Logic. The logic also contains a pointer to the Grammar. New logics can be added by providing any necessary new types of items, inferences, grammars, and charts. However, most new logics will be able to reuse many of the classes built for previous logics, as we have done with our translation logics.

[7] "Dimension" is the same as "component," and refers to one of the texts in a multitext, one of the components of a multiparsing logic, one of the languages generated by a transduction grammar, etc.. In the latter case, we intend the formal language theory sense of "language" — a transduction grammar that generates pairs of strings that are both in the same *natural* language is still considered to range over two *formal* languages. This ambiguity of the word "language" is why we prefer "dimension" or "component." See Melamed and Wang [2005] for details.

[8] not implemented yet

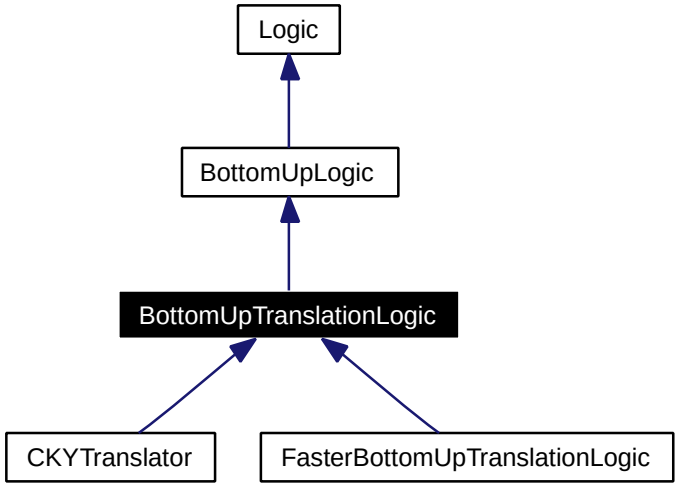

Figure 3.6: The Logic class hierarchy.

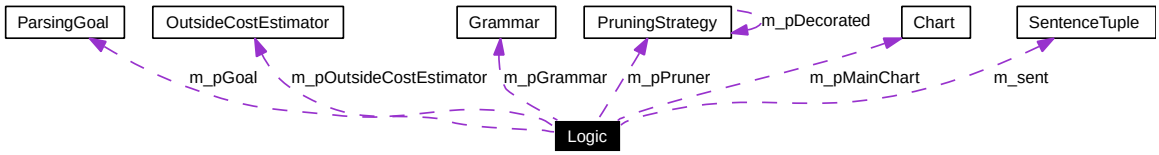

Figure 3.7: The Logic collaboration diagram.

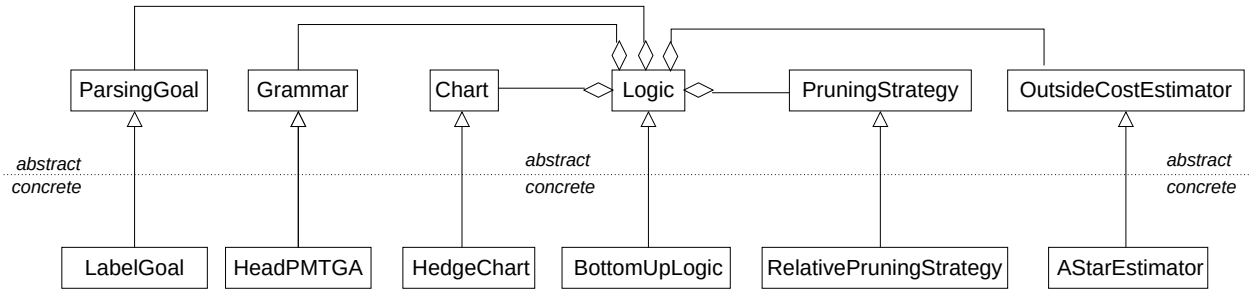

Figure 3.8: The Abstract classes with some example instantiations.

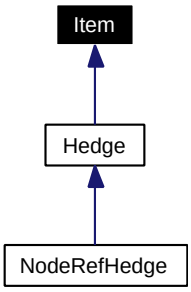

Figure 3.9: Item class family

While Figure 3.7 shows the collaboration diagram for the Logic abstract base class, Figure 3.8 shows a typical implementation of the Logic. Everything above the dotted line are abstract base classes, and everything below the line are concrete classes; the Logic acts as a mediator of all these parts. Note that it is the Logic's responsibility to construct the concrete Chart, OutsideCostEstimator, PruningStrategy and ParsingGoal. The Logic constructor takes in parameters indicating which concrete classes to instantiate.

The key method provided by Logic is `expand`(trigger), which generates a batch of new items:

```
expand(Item trigger){
    #Get "sibling" item from trigger
    Item item2 = chart.matchItem(trigger);
    #Get "parent" from trigger and sibling
    set ⟨ Item ⟩ I = grammar.getPossibleCons(trigger,item2);
    return I;
}
```

In the implementation of `expand`, a NULL pointer to the trigger indicates the beginning of parsing. In this case, `expand` will generate the first batch of inferences from the axioms, so that the agenda can be initialized by them. For example, if the logic is BottomUpTranslationLogic, the first iteration creates Scan and Load inferences.

3.2.4 Item

Different logics make use of different types of items. There is currently only one immediate subclass of Item called Hedge. Hedges corresponds to what is called an edge in 1D CKY parsing; the name "hedge" is short for hyper-edge. A Hedge is composed of a nonterminal link, a d-span vector, and heir information. In addition, all items maintain a pointer to the inference that they were the consequent of, which is necessary for reconstructing a parse. A subclass of Hedge, NodeRefHedge, also contains a pointer to a node in a PositionTree, used during hierarchical alignment. Figure 3.9 shows this class family.

3.2.5 Chart

The Chart is a container that holds items that have already been inferred. The current Chart class family is shown in Figure 3.10. The abstract base class Chart is just an interface. It does not even

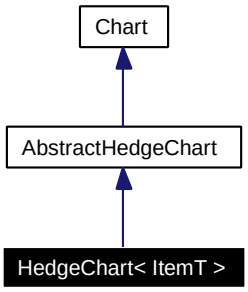

Figure 3.10: Chart class family

inference	antecedents	consequent	relevant logic(s)
Scan	input word	Hedge	BottomUpLogic and subclasses
Load	none	Hedge	BottomUpTranslationLogic and subclasses
Compose	2 Hedges	Hedge	BottomUpLogic and subclasses

Table 3.2: Inferences

know the type of items that will be stored. AbstractHedgeChart has methods that are specific to Hedges, but only the concrete HedgeChart specifies data structures.

When a trigger item is introduced to the logic, the chart is consulted to see which items satisfy the linear precedence constraints for composition. Inferences of different logics might derive different kinds of items, requiring different types of containers for the items. For example, BottomUpTranslationLogic builds Hedges, so it needs a HedgeChart to store its items. HedgeChart contains Hedges or their subclasses like NodeRefHedges.

All Charts provide an `insert` method which takes an Item to be inserted in the chart, and an `exportParses` method which returns a vector of parse trees with their scores. If a full parse was not found, the chart returns a "maximum cover" which is as few parse trees as possible that cover the entire span.

3.2.6 Inference

An Inference deduces an item based on items that have already been derived. The currently implemented inference types are listed in Table 3.2. A Scan or Load has only the axioms as antecedents. Compose inferences are made from two "adjacent" Hedges, in addition to the usual grammar term.

3.2.7 PruningStrategy

Our parser supports both relative pruning and absolute pruning. Figure 3.11 shows the family of pruning strategies. Constructing the relative pruning strategy requires specifying the pruning factor, which decides the size of the pruning beam. The absolute pruning strategy involves two scores: one serves as the threshold used to prune inferences whose consequent f-cost is larger, the other serves as the increment from the previous pruning threshold. This is useful for multi-pass parsing. It is also possible to use a composite of both pruning strategies, with thresholds set for

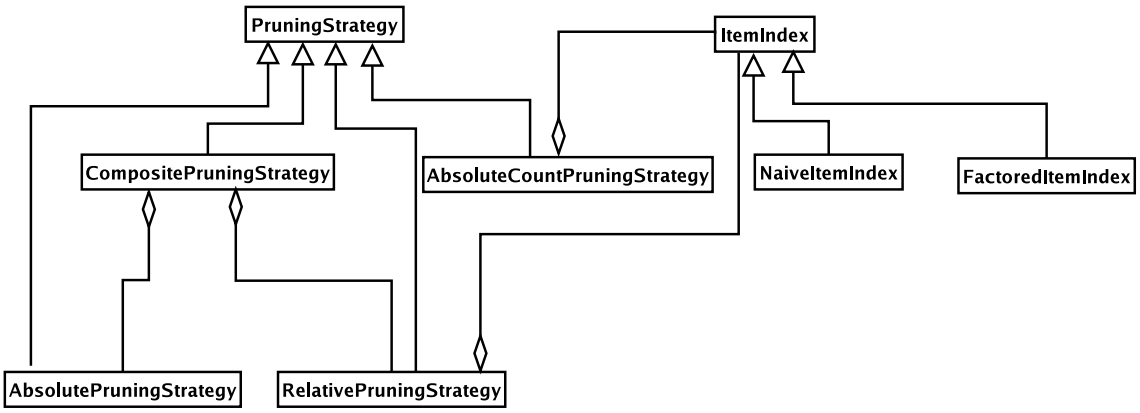

Figure 3.11: PruningStrategy class family

each. The absolute count pruning strategy sets a maximum number of inferences which can exist over a given span of the input.

As shown in Figure 3.11, the RelativePruningStrategy class contains an index of items. This index maps from item spans to items. Items located by the same key belong to the same beam, the size of which is decided by the pruning factor stored in RelativePruningStrategy. The AbsolutePruningStrategy does not require such an item index, because it is able to decide whether to prune an item only by the cost of the item and the absolute threshold stored inside AbsolutePruningStrategy. The composite strategy uses an item index for relative pruning and an absolute threshold for absolute pruning. The AbsoluteCountPruningStrategy uses an index of items like that used in RelativePruningStrategy. It then prunes all but the k lowest cost inferences over a given span.

3.2.8 OutsideCostEstimator

This class family realizes different estimators for the outside costs of items proposed by the logic. The base class OutsideCostEstimator specifies the public interface of this class family. We currently are not using OutsideCostEstimators, but they will be needed for optimization purposes, and so some of the needed classes exist.

3.2.9 ParsingGoal

The ParsingGoal determines when the parsing should terminate. The ParsingGoal class family is designed as a flexible utility for this purpose. For example, the parsing could continue until a specific item is derived, or until any item with a given label is derived, or until some set of items is derived. Other goals might be a span, a threshold, a time limit, or the composition of them. Figure 3.12 shows the ParsingGoal class family. In this family, all goals excluding ConjunctCompositeParsingGoal and DisjunctCompositeParsingGoal are primitive parsing goals. The ConjunctCompositeParsingGoal is used to compose primitive or composite goals into a larger goal. Only when all the subgoals are reached will the parsing terminate. DisjunctCompositeParsingGoal also composes subgoals, but parsing terminates if any subgoal is reached. Based on this class family, we can specify a large variety of parsing goals.

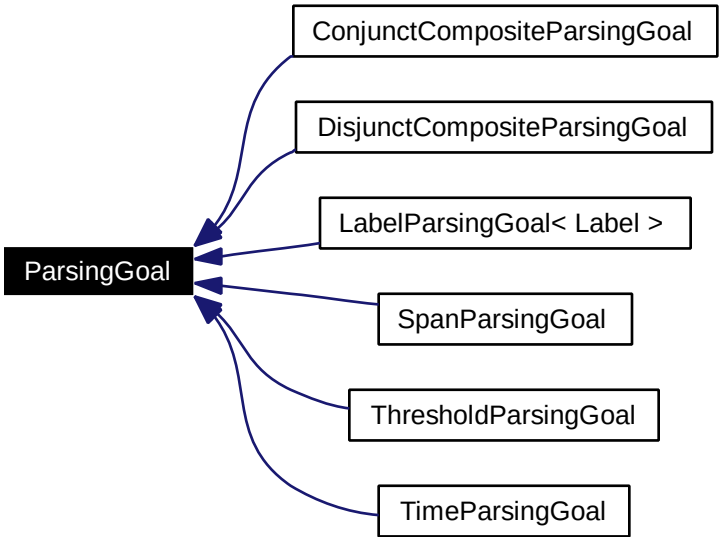

Figure 3.12: Parsing goal class family.

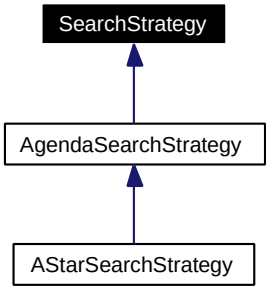

Figure 3.13: SearchStrategy class family.

The ParsingGoal class family provides two main methods: `add` and `achieved`. The former adds a subgoal into the current parsing goal, and returns an ID for the just added goal. The latter checks whether the goal has been achieved by the item that has just been derived.

3.2.10 SearchStrategy

The SearchStrategy governs the order the Parser processes Items. The base SearchStrategy class is abstract. There are two concrete implementations. AgendaSearchStrategy uses an Agenda and Comparator to schedule items. AStarSearchStrategy uses an OutsideCostEstimator to do best first parsing. This is implemented using the Agenda with a BestFirstComparator.

A BestFirstComparator compares two items based on the scores (i.e., f-cost); a SmallestSize-FirstComparator compares two items according to their spans, as in ordinary CKY parsing. Agenda is implemented using a priority queue templatized over a concrete Comparator class.

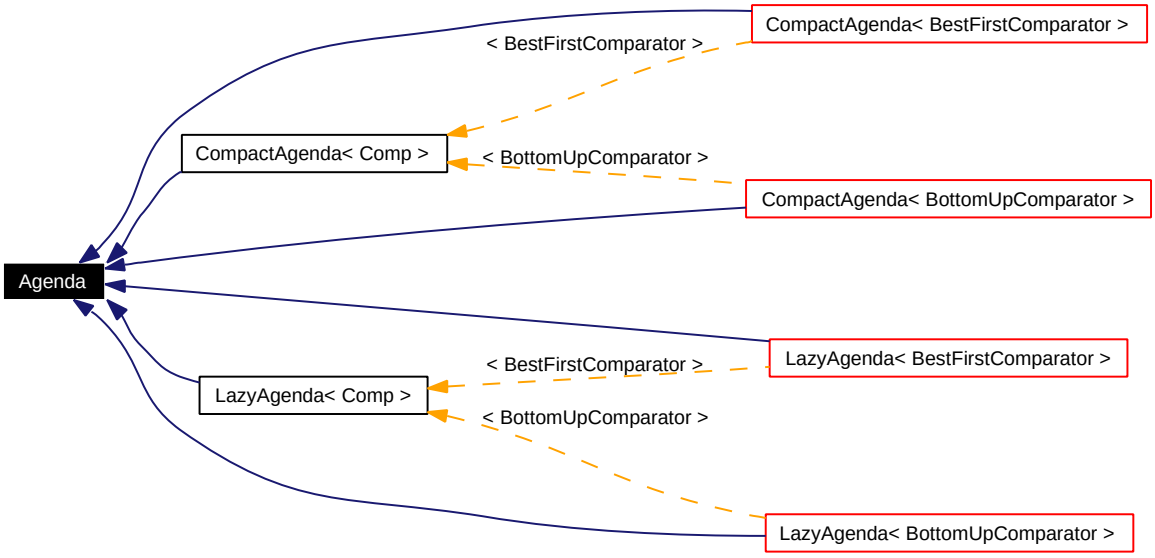

Figure 3.14: Agenda class family.

The difference between the LazyAgenda and the CompactAgenda is that the latter never duplicates items of the same signature. However, this space efficiency comes at the price of some time efficiency – the time necessary to detect and remove duplicates. Both LazyAgenda and CompactAgenda are template classes with Comparator type being their template parameter.

3.2.11 The Atomic Units: Terminal and SynCat

Terminals and SynCats are both just numbers. They represent the atomic units in the parser that are grouped together to make the links. A Terminal represents a "word" in the input language, and every Terminal is associated with its corresponding word in the terminal vocabulary. SynCats represents "syntactic categories", which are also indexed in a dictionary. Both of these types are implemented using typedefs.

3.2.12 Nonterminals

Nonterminals can be seen as the smallest unit in a single dimension that is being rewritten. In a non-lexicalized grammar, a Nonterminal contains simply a SynCat, but in some grammars, nonterminals have more information than just the syntactic category. For example, the code-base uses LexNTs, which are Lexicalized Nonterminals. LexNTs are a subclass of Nonterminal, and so they inherit a syntactic category, but they also have a Terminal member, which represents the lexical head of the Nonterminal. There is also a subclass of LexNT called LexNTWithExtremumTerminals, used for grammar mixtures that combine a WMTG with a target language model.

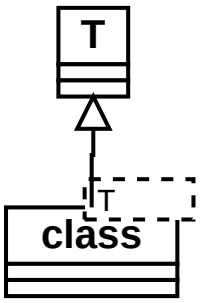

Figure 3.15: Static decorator pattern

3.2.13 NTLinks and TLinks

Terminals and Nonterminals are grouped into links called TLinks (terminal links) and NTLinks (nonterminal links). Terminal links are a D-dimensional vector of Terminals, one in each component. NTLinks are a D-dimensional vector of Nonterminal pointers, which allow for polymorphism. Many components of the code don't need to know whether a Nonterminal pointer is lexicalized or not, (i.e. whether it is pointing to a Nonterminal or a LexNT) and so it just passes the Nonterminal pointer along to the next part of the code without dereferencing the pointer. If at any time the code needs to know what kind of Nonterminal pointer it has, it can simply dereference and get the correct type using static or dynamic casting.

Care needs to be taken when using NTLinks since they are vectors of pointers. When first initialized with just a size parameter, the pointers in an NTLink are not yet initialized, since it is unclear which kind of Nonterminal pointer the user wants to put into the NTLink. The NTLink copy constructor and operator= copy the values of the Nonterminals pointed to, not the pointers, using the `clone()` operation on each member nonterminal. `clone()` is a virtual method which, when called on a object, returns a new instance of that object.

3.2.14 Parse Trees

The two main classes used to represent parse trees are HeadMTree (headed multitree) and PositionTree. Either kind of parse tree is a tree container composed of tree nodes, each node containing such features as the Precedence Array Vector (PAV), the HeirVector, and the constituent label assigned to the node. There are quite a few types of features for the parse tree to maintain. To avoid a potentially huge parse tree class that maintains all the features, we prefer the parse tree be a composition of several simpler parse trees, each manipulating only one type of feature. We use a design technique knows as the "static decorator pattern" to meet the requirement. A static decorator is a template class, and at the same time, derives from the template type (see Figure 3.15). It is intended to decorate other objects of the template type by adding more methods to or overriding the methods in the decorated object being inherited. Both the decorating and decorated classes share the same data, and there is no extra data inside the decorator. All methods of both the decorating and decorated classes operate only on the shared data. The decorator is initialized by copying the pointer to the data inside the decorated object. The reason we call it "static" is that the decoration is achieved through template and inheritance, and thus is a "compile time struc-

ture". To apply the static pattern to the design of the parse tree, we first design some simple trees, such as a PAVTree that maintains only the PAV information, and an HeirTree that maintains only the heir information. Then a more complex tree can be obtained according to the static pattern by templatizing the PAVTree over the HeirTree, and the HeirTree over a `boost::tuple<RHSElem, PAV, Vector<void*> >`. (RHSElem is either an NTLink or TLink. See Section 3.2.15 for more information.) Compared to the standard decorator pattern [Gamma et al., 1994], the static decorator pattern doesn't require the methods of each class (base or subclass) to be exactly the same, saving much code if each class has a large number of methods.

It is worth mentioning how to implement the methods involving several features (e.g., PAV and label) in a simple tree. In a simple tree, we don't know the type of the base class (it is only a template parameter), neither could we know the types that tuple is templatized over (they are also template parameters). But most methods involving multiple features need to know the types of the base class and data, and must make sure they are of certain type(s). In this case, we add a **test** to the very beginning of the method implementation. A test consists of some casting statements, which checks the types of the base class and data type at compile time.

There are 3 simple trees that make up a HeadMTree, namely, SpanTree, HeirTree, and the generic tree. All the general methods should be added to the generic tree classes. Each simple tree has such defined types as "value_type" standing for the data type contained and "_Base" for the type of the base class.

A PositionTree is implemented in much the same way as a HeadMTree. However, instead of containing PAVs in each node, it contains Spans. Note that PAVs are directly obtainable from these spans. PositionTrees are used as the constraining tree in TreeConstrainedMTG.

The file formats used for multitrees are described in the MTV documentation.

3.2.15 HeadProduction

The HeadProduction class family represents the production rules of MTGs. It consists of an LHS, which is an NTLink, an HeirVector, which keeps track of which child is the heir in each dimension, a PAV, which gives the linear ordering of the children in each component and a RHS, which is a dequeue of RHSElems. RHSElems are a boost variant of NTLinks and TLinks. This means that they can either contain a NTLink or TLink. The = operator is used on a variant to set it to any of its associated types. Later, when you want to obtain the link, you simply check which type it is first. See boost.org for more information on the syntax of boost variants.

Since NTLinks and TLinks are numerous, we use variants to avoid an unneeded common superclass that uses extra memory.

3.2.16 CKY Translator

There are a small number of classes designed specifically for translation using the CKY parsing algorithm: CKYTranslator, CKYHedgeChart, CKYCell and CKYEquivalence. This implementation of the parsing algorithm is a special case of the general GenPar algorithm in that it does not really use its Agenda and has an alternative data structure for the chart.

The onePassParse function in the Parser class is used in the same way, except that the complete CKY algorithm is called via the expand function in the CKYTranslator class, which is a subclass of BottomUpTranslationLogic. The set of items returned to the parser is empty, so nothing gets pushed onto the Agenda, and the parser terminates after the first iteration.

CKYHedgeChart implements the chart, which is simply an array with an indexing function which identifies a particular cell given a start position and length.

CKYEquivalence determines whether an item is already in the chart for dynamic programming purposes, given a start position and length (which identifies a cell) and the item. The nonterminal label of the item is used as part of the key. For head-lexicalized models the nonterminal label includes the head(s).

Currently the Viterbi algorithm is hard-coded in the insert function in the CKYEquivalence class. However, one of the motivations for developing this set of classes was to allow a complete chart to be built, so that inside-outside scores could be computed, perhaps leading to a new translation metric. Building the complete chart would require all equivalent items to be stored, say as a linked list, rather than simply the highest scoring item.

There is currently only a simple pruning strategy implemented, which only allows an item to be inserted if the item's Viterbi-max probability is within some factor of the highest probability item for the cell. The BEAM parameter is hard-coded in the CKYTranslation class; ideally this should be read from one of the existing pruning configuration files.

This set of classes has been tested only on a small number of sentences in the English-English sandbox.

3.3 Serialization

Serialization has been implemented using the boost serialization library to enable convenient i/o of large objects like grammars. To be serializable, all classes must contain a serialize function templatized on archive type. If a class inherits from another, it must explicitly serialize the base class as well using `ar & boost::serialization::base_object<BaseClass>(*this);` The base class serialization should not be called by an explicit typecast, as this skips some essential steps in associating derived and base classes.

The serialize function simply archives all member variables and, if applicable, recurses through the base class(es) to serialize those member variables as well. Files must #include <boost/serialization/serialization.hpp>.

Example:

```
template<class Archive>
void serialize(Archive & ar, unsigned int version) {
   ar & boost::serialization::base_object<_base>(*this);
   ar & m_depHeirDistribs; }
```

If the member variables include multiple ways of accessing the same object, the object itself must be archived before the pointers to it (so that only one copy of the relevant object is created upon restoration).

Abstract base classes must #include <boost/serialization/is_abstract.hpp> and use the macro BOOST_IS_ABSTRACT(ClassName).

All standard library containers are automatically serialized using boost, but classes derived from them must #include the relevant serialization file, e.g. <boost/serialization/vector.hpp> for the Vector class.

Repeated serialization and restoration will not produce exactly the same archive contents every time, but the restored objects will be identical.

When a derived class object is referred to via a pointer to a base class, a few extra steps must be taken to ensure that the object is recognized and restored correctly. The relevant output and input archives must explicitly register the derived type, which instantiates an object of that type at compile time so that a subsequently archived base class pointer "recognizes" the existence of the derived class. Because the derived object is instantiated, the derived classes and all the classes from which it inherits must contain default constructors. If the derived object is templatized, the specific classes on which it is instantiated in the grammar must be registered. For example:

```
std::ofstream ofs(filename.c_str());
boost::archive::binary_oarchive oa(ofs);
oa.register_type<HeadPMTGA>();
oa << oldGrammar;
```

Function pointers cannot be directly serialized. They are instead mapped one-to-one to strings and restored to pointers on loading. The mapping is created using NameAddressMap, and any new function pointers used in new classes must be added to the mapping manually. Files instantiating the function pointers must then use the pointers created by the mapping - for an example, refer to HeadPMTGA.cpp and LazyCD.hpp.

For more information on serialization using the boost library, refer to `http://boost.org/libs/serialization/doc/index.html`.

3.4 Data Encapsulation via Nested Configuration Files

We generally define the behavior of class objects via the arguments of the class constructor. However, our design has many classes composed of other classes and so the number of parameters being passed around can get large. To solve this problem, we turn to configuration files to construct class objects instead of using constructor arguments.

Any class with configurable options has a separate configuration file. The class itself defines which option names and types it knows about and reads the option values from the configuration file whose name is typically passed as the only constructor argument. The options reader (from the boost::program_options library, see below) accepts only options that the class specifically defines. This way we ensure data encapsulation: Just as classes themselves are small encapsulated units containing only the data and behavior they need locally, so is every configuration file a single unit containing only the values that the specific class needs.

Figure 3.16 shows the config files used by the `gp` program and their relationships. Each node represents a config file, lines connecting nodes represent the composition (containment) relation. The relationships between config files are almost isomorphic to that in the overall design of the generalized parser (figure 3.1). The `gp` program takes as parameters the names of the config files of the Parser and the Grammar, as well as the options specifying, for example, the sentence tuple file. The Parser config file not only contains the values of member variables of the Parser class itself, but also the names of the config files of class Agenda and class Logic. The Logic configuration file then contains the config file names of OutsideCostEstimator and PruningStrategy.

Each config file is of the following format:

```
\# The Config File for Agenda class family.
\# the type of agenda
AgendaType = LAZY
\# The search strategy used by the agenda
```

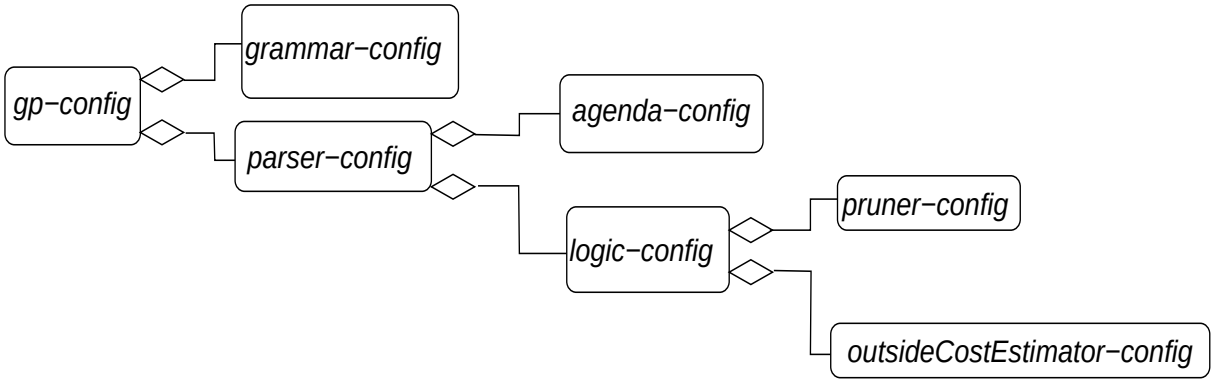

Figure 3.16: The config files of the Generalized Parser.

```
SearchStrategy = BESTFIRST
```

Comments starts with symbol "#". An option name and its value are separated by symbol "=".

The option names and types that a class defines are inherited to its subclasses. For example, the class NaiveWMTG defines an option `FanOut`, and so its subclasses HeadPMTG, HeadPMTGA, HeadPMTGB know about and accept such an option in their respective configuration files.

Suppose you implement MySubClass, a subclass of BaseClass. Your class has to take a config file name as a constructor argument and pass it on to the base class in the member initialization list:

```
MySubClass(const string& configFile) : BaseClass(configFile){
  // ...
}
```

The base class will read the config file and provide the values through its member functions. If it does not provide the values you need you will have to parse the config file yourself, which means you need a boost::program_options::options_description object that contains the names and types of the options expected in the config file. Fortunately, the base class defines those options descriptions for you. You just have to construct the options_description object and pass it to the base class in the initialization list. GenPar classes take a smart pointer (boost::shared_ptr) to an options_description as an optional constructor argument. After the base class is initialized your options_descriptions object will automatically be filled with the expected options, e.g. the string option `TreebankGrammar`. You are ready to parse the config file:

```
shared_ptr<options_description> optionsDescription;
MySubClass(const string& configFile) : BaseClass(configFile,
                                                 optionsDescription){
  string optTreebankGrammar;
  variables_map vm;
  ifstream ifs(configFile.c_str());
  store(parse_config_file(ifs, *optionsDescription), vm);
  notify(vm);
  if(vm.count("TreebankGrammar"))
```

```
    optTreebankGrammar = vm["TreebankGrammar"].as<string>();
  // ...
}
```

This way you can get all the options the user has specified in the config file. But this might still not be enough. Now you want to add your own options description. You want the config file to also contain an integer option `MaxValue`. Remember that the same config file will be read by the base class and by MySubClass. Therefore, they both have to know that such an option might occur. You have to add `MaxValue` before you construct the base class:

```
shared_ptr<options_description> optionsDescription;
MySubClass(const string& configFile)
  : BaseClass(configFile,
              addOptionsDescriptionsTo(optionsDescription)){
  int optMaxValue;
  variables_map vm;
  ifstream ifs(configFile.c_str());
  store(parse_config_file(ifs, *optionsDescription), vm);
  notify(vm);
  if(vm.count("MaxValue"))
    optMaxValue = vm["MaxValue"].as<int>();
  // ...
}
shared_ptr<options_description>
MySubClass::addOptionDescriptionsTo(shared_ptr<options_description> optionsDesc){
  optionsDesc->add_options()
    ("MaxValue", value<int>()->default_value("99999"), "Defines the max value");
  return optionsDesc;
}
```

That is all you have to know if you want to write your subclass with its own options handling in GenPar. If you want others to be able to again subclass your class you must make its constructor take a smart pointer to an options_description as optional argument:

```
  MySubClass(const string& configFile,
     shared_ptr<options_description> priorOptions =
       shared_ptr<options_description>(new options_description()));
```

This will make sure that others can also pass in an optional options_description when they construct your class as their base class, the same way we did it with BaseClass in the above example.

See NaiveWMTG.cpp and IndexedWMTG.cpp for good examples of config file handling in our code base. For more information on the boost::program_options library see
`http://www.boost.org/doc/html/program_options.html`.

3.4.1 "Builder" classes

The "builder" classes [Gamma et al., 1994], also known as "virtual constructors," are responsible for instantiating concrete classes based on the config files that they receive. These builder classes include the AgendaBuilder, GrammarBuilder, LogicBuilder, OutsideCostEstimatorBuilder and PrunerBuilder. For example, the GrammarBuilder is given a config file that contains what type of concrete Grammar needs to be instantiated by the Parser, for example a HeadPMTGA. The GrammarBuilder parses the file, and instantiates this concrete Grammar and then passes it back to the Parser. The Parser then maintains a pointer to this Grammar and whenever it is accessed, it is in fact the concrete class being accessed. This means that the Parser need not know what kind of Grammar it is pointing to. Another benefit is that the Grammar construction code, which is quite involved, need not be implemented more than once. All of the builder classes follow the singleton design pattern.

3.5 Examples of implemented applications

We describe three programs that have been implemented using the GenPar classes. There are many others in the `GenPar/tools/` directory of the toolkit.

3.5.1 The program `gp`

`gp` acts as a front end to the Parser class. The front end's main job is to handle configuration and I/O.

To use the Parser class, we first allocate the necessary components of the parser. Next, we allocate the parser object, passing the references (pointers) to the components into the parser.

The following statement allocates a HeadPMTGA object:

```
Grammar* grammar = new HeadPMTGA(...)
```

We omitted the arguments of the HeadPMTGA constructor. Readers can refer to the Doxygen-generated system documentation for details. After the grammar is allocated, we define the parser object.

```
Parser* parser = new Parser(grammar, ...)
```

The `grammar` object has been passed into the parser as a constructor argument. The constructor arguments also specify the types of other components, which will be allocated at construction time. Using the parser object, we now can parse a sentence tuple.

`gp` can perform **hierarchical alignment**, ordinary **multiparsing** and **translation**. Alignment uses a different grammar class than multiparsing and translation. The only difference between translation and multiparsing is the number of input sentences. If the dimensionality of the input sentence tuple is less than that of the grammar, it is translation. If it is equal, we are doing hierarchical alignment or ordinary multiparsing.

3.5.2 The grammar initializer: `trees2grammar`

To create a weighted MTG instance from a multitreebank, we read in the multitrees from the treebank, and increment them into the grammar object. First we instantiate the NaiveWMTG class.

```
NaiveWMTG mtg(...);
```

Next, we read one tree at a time and break it down into the grammar using the `increment` function. `ist` is an input stream.

```
HeadMTree mtree;
ist>> mtree;
mtg.increment(mtree);
```

We do this for all trees in the treebank. After we have incremented all the trees into the NaiveWMTG, we can create a more sophisticated PMTG and increment these new productions into it:

```
PMTG model(...);
model.increment(mtg);
```

The HeadPMTGA decomposes each production into smaller units under some Markovian independence assumptions. Lastly, we dump the model to standard out.

```
cout << model;
```

At this point, this program has created a HeadPMTGA from a multitreebank.

3.5.3 The re-estimation program: `viterbiTrain`

The re-estimation program performs (a simplistic approximation to) the EM algorithm, using the classes illustrated in Figure 3.17. It first builds an empty WMTG, and then instantiates the GPEM class.

```
WMTG* initialGrammar = GrammarBuilder::Instance()->build(...);
GPEM em(emConfigFile);
```

Next it sets up a file to read the input sentence tuples.

```
iFile data(sntFileName);
```

Lastly, it calls GPEM.run() which runs the EM algorithm.

```
em.run(data, initialGrammar, grammarConfigFile);
```

The `run` function in GPEM basically alternates between calling the same functions as `gp` to parse the input sentence and calling the same functions as `trees2grammar` to build a new grammar.

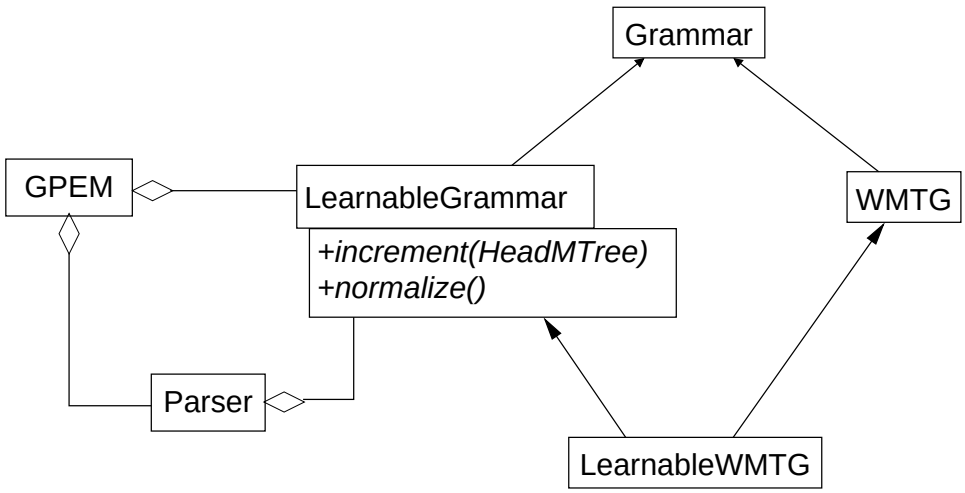

Figure 3.17: Classes used by the `viterbiTrain` program.

`viterbiTrain` is not intended to be a serious tool for machine learning, but merely a demonstration of how GenPar can serve as the inference engine for grammar induction. On the other hand, given a good inference engine, machine learning need not be complicated.

Chapter 4

MTV User Guide

N.B.: This chapter pertains to MTV-1.0.2. Later releases of the tool will contain documentation that is more up to date.

4.1 Introduction

The Multitree Viewer is an application originally created to visualize output of the GenPar toolkit. It can now be used by any MT developer to visualize their multitree output and optionally, their PMTG grammar output. Users who are running this with GenPar should read the GenPar user guide before reading this guide for help on locating the input files.

4.2 Compiling MTV

To compile MTV, download package directory to a system with at least Java 1.5 installed. In a Windows-based Visual Java program, create a project with the package directory and compile all files. On Unix/Linux, first make sure that your JAVAHOME environment variable is set to indicate where your java executables live. Then type `make` in the package directory.

Users who are interested in developing MTV and want to read the javadoc should type `make doc` in the package root. This will put the javadoc in doc/design/.

4.3 Running MTV

To run MTV on a Unix/Linux system, execute the `runit` command from any directory. To run MTV in a Windows-based Visual Java program, specify `Mtv.java` as the main class and run the project.

MTV comes with sample input files to enable sample compile/runs. Users wanting to use their own input files should refer to the next section. Sample Files provided:

- examples.tree: DO NOT delete/move this file. MTV will use it as default tree file if the user-requested file is not found. It has several interesting parses so it's suggested the user check these out on first run.

- english.tree: This is early English/English output from GenPar. The trees are messy, but the file is included to enable sample use of a grammar file.
- english.grammar: The accompanying grammar file to english.tree
- symbol.converter: A short converter file that should be used with examples.tree
- *.tvcb: A sample vocabulary file for terminal vocabulary
- *.ntvcb: A sample vocabulary file for nonterminal vocabulary

NOTE: No examples of integerized input is provided as GenPar has stopped producing it. Users are welcome to try their own.

MTV loads in all the information from the input files at the beginning, so it needs enough heap space to do this. Experiments on a Solaris system have shown that the necessary amount of heap space (in MB) can be found by multiplying the number of input sentences by 0.15.

For example, 5000 sentences will need about 750MB of heap space, to be safe. If not enough MB are available on the user system for their input, its suggested the input be segmented into smaller files.

4.4 Configuration

All user configuration files and settings must be specified in `mtv.config` in the MTV package root. After specifying all configuration settings, run MTV again to see visualization of input.

NOTE: Absolute paths must be specified with a "/" in front. A relative path is assumed relative to MTV package root on Windows, or directory from which MTV was run on Unix systems (runit command may be run from outside package root).

BaseDirectory On Unix systems, MTV will automatically set this to the directory from which the program was run. It may need to be specified for some Windows systems. This variable is **only** used for locating the images used in the GUI.

DataDirectory Users who have all their data files in one directory (such as the provided data directory in the MTV package) can specify that directory here. Users with data files in various locations should leave this blank.

MTV will concatenate `DataDirectory`, if it exists, with the filenames below to locate files. For each of the file variables, users can specify filename, relative path, or absolute path.

GrammarFile A description of that file format can be found in Appendix B. If the user specifies an integerized file here, then the user must also specify vocabulary files for each of the input dimensions below so MTV can display words to the user.

Users who simply want to see multitrees visualized with no probability information can choose to leave this blank.

TreeFile A description of that file format can be found in Appendix B. If user specifies an integerized file here, user must also specify vocabulary files for each of the input dimensions below so MTV can display words to the user.

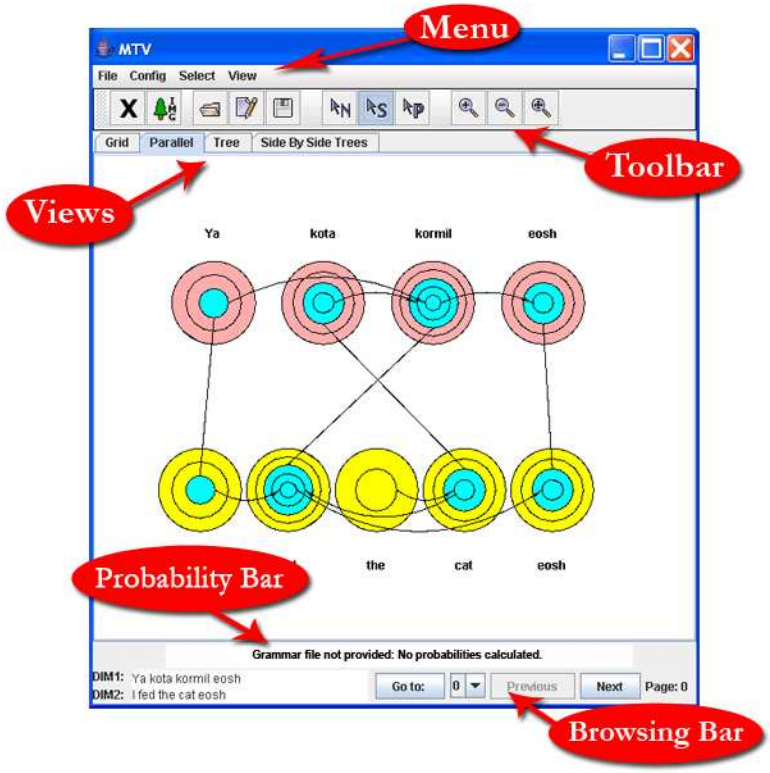

Figure 4.1: An annotated screenshot of the MTV User Interface.

dim[#]Converter A description of this file format can be found in the appendix to this document. Users who are inputting a dimension with transliterated characters can specify an ASCII to Unicode mapping in a converter file here. Converter files can be specified for 0, 1, or both dimensions. Leave blank if none needed.

dim[#](NT)VocabFile Users who are inputting integerized input must specify terminal and nonterminal vocabulary files here for both dimensions with a mapping for all integers used.

VocabFormat Users who are inputting integerized input should specify `1` here. Users inputting deintegerized input should specify `2` here.

4.5 User Interface

As a visualization program, MTV is designed to have a user-friendly GUI. Hopefully, it succeeded.

4.5.1 Menu/ToolBar

The four menus in the menu bar, **File**, **Config**, **Select**, **View**, are duplicated in the iconic toolbar below it.

File The **Save to Image** command allows the user to specify a filename and directory to save a "screenshot" of the current view as a JPEG file. The image will be the same pixel size as the view, so users should resize as desired. The **Quit** command allows the user to quit MTV.

Config The configuration file can be modified inside or outside of MTV, as preferred by the user. MTV will always look at mtv.config for its settings when loading up, but once loaded, MTV can load settings from any file.

The **Load** command allows the user to choose a file to load settings from. This will replace all the currently displayed trees with the trees from the loaded file. The **Edit** command provides a graphical form for the user to change the current settings. This will **not** save the new settings to any file, but will reload trees and grammar in MTV as specified. The **Save** command writes the current config settings to a user specified file.

Select These commands allow the user to specify their "selection"(rollover) mode. This affects the displayed probabilities and highlighted nodes. In **Node** selection mode, only the currently selected node will be highlighted. In **Span** selection mode, the currently selected node and all its descendants will be highlighted. In **Parent** selection mode, both the currently selected node and its parent will be highlighted.

View The **Zoom In** command will zoom in 10% each time clicked. The **Zoom Out** command will zoom out 10% each time clicked. The **Zoom Fit** command will attempt to zoom so that all nodes are visible on the screen. For Parallel and Tree views, this also centers the nodes.

4.5.2 Probability Status Bar

The probability status bar displays the currently calculated probabilities, if a grammar file is provided, and depending on selection mode.

In **node** mode, only the probability of the production containing that node (i.e. the selected node and its child) is calculated.

In **span** mode, the probability of the selected nodes and all productions below it are multiplied together.

In **parent** mode no probabilities are calculated.

The probability is displayed in terms of the events in the grammar file (see the glossary). Probability ranges from 0 to 1.

4.5.3 Parse Browsing

This bar will help in browsing through the parse trees loaded.

The **Previous** button can be used for going to the previous parse tree, and the **Next** button can be used for going to the next parse tree. The parse number is displayed next to **Page:** . Remember the parse number of interesting parses you'd like to be able to jump to, as you can select it in the combo box and click the **Go to** button to jump to that parse.

This bar also displays the dimension 1 sentence next to **DIM 1:**, and the dimension 2 sentence next to **DIM 2:** .

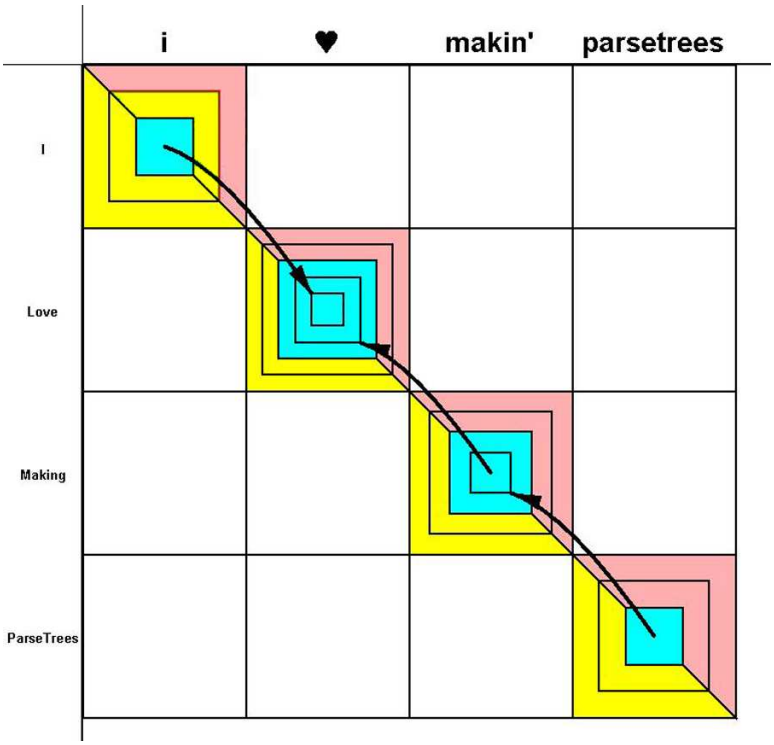

Figure 4.2: A typical grid view

4.5.4 Views

MTV features four views (**Grid**, **Parallel**, **Tree**, **Side-by-Side Tree**) that display all the information contained in the multitree output in varying representations. It's suggested that users explore the views and decide which view conveys the information in the way that makes the most sense to them. Everyone's got their favorite! In all four views, pink represents a 1-dimensional node that is null in dimension 2, yellow represents a 1-dimensional node that is null in dimension 1, and blue represents a 2-dimensional node (one that is non-null in both dimensions).

Grid View

This view displays a grid of 2-dimensional information, like a birds-eye view.

By convention, dimension 1 information is represented on the horizontal axis, and dimension 2 information is represented on the vertical axis.

A 2-dimensional node is positioned according to the position of its two dimensions' lexical heads in their respective sentences. A 1-dimensional node is positioned according to the position of its non-null dimensions' lexical head along with the position of the lexical head of the nearest 2-dimensional ancestor.

A fully linked node (see glossary) is represented as a square, and all other nodes are represented as triangles.

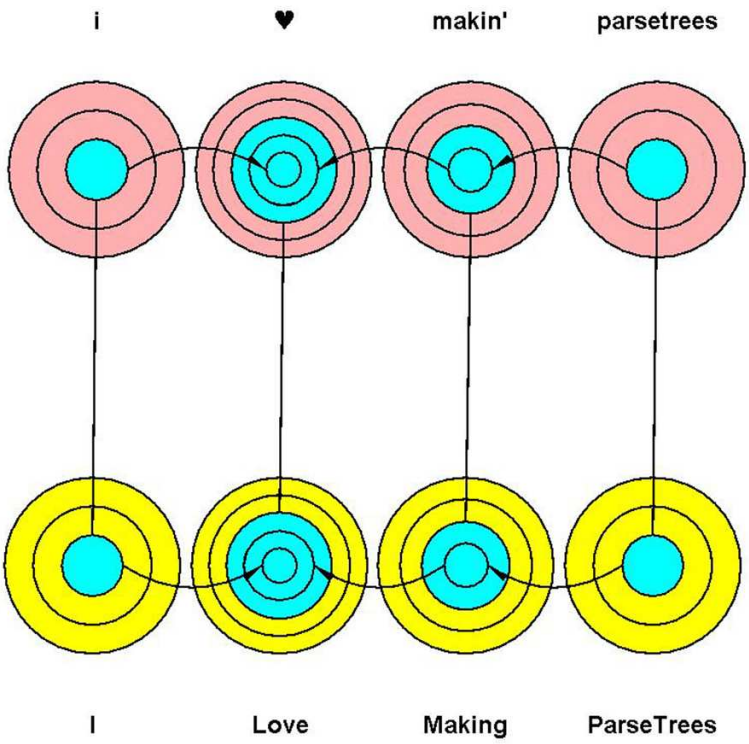

Figure 4.3: A typical parallel view

Nested triangles represent a solely 1-dimensional heir chain, while nested squares represent a 2-dimensional heir chain - the innermost square represents the highest node which received the lexical head of a lowest link in both dimensions.

Arrows indicate dependency in at least one dimension. The user can know which dimension the child node is dependent on only by looking at where on the parent it landed. If it lands on a corner, it is dependent in both dimensions; if it lands on the upper or right edge, it is dependent in dimension 1; and if it lands on the lower or left edge, it is dependent in dimension 2.

Selecting the node in parent mode will also aid the user in figuring out dependency, as the arrow will be highlighted only if the dependent child is selected.

Parallel View

The parallel view displays two rows of 1-dimensional information with lines between indicating word alignment.

By convention, dimension 1 information is conveyed by the top sentence and row of circles, and dimension 2 information is conveyed by the bottom row.

Each of the biggest circles represents a terminal in that dimension, and these biggest circles are ordered according to their position in the sentence. The word above a biggest circle is the terminal. Thus, the sentence in each dimension can be read off from left-to-right.

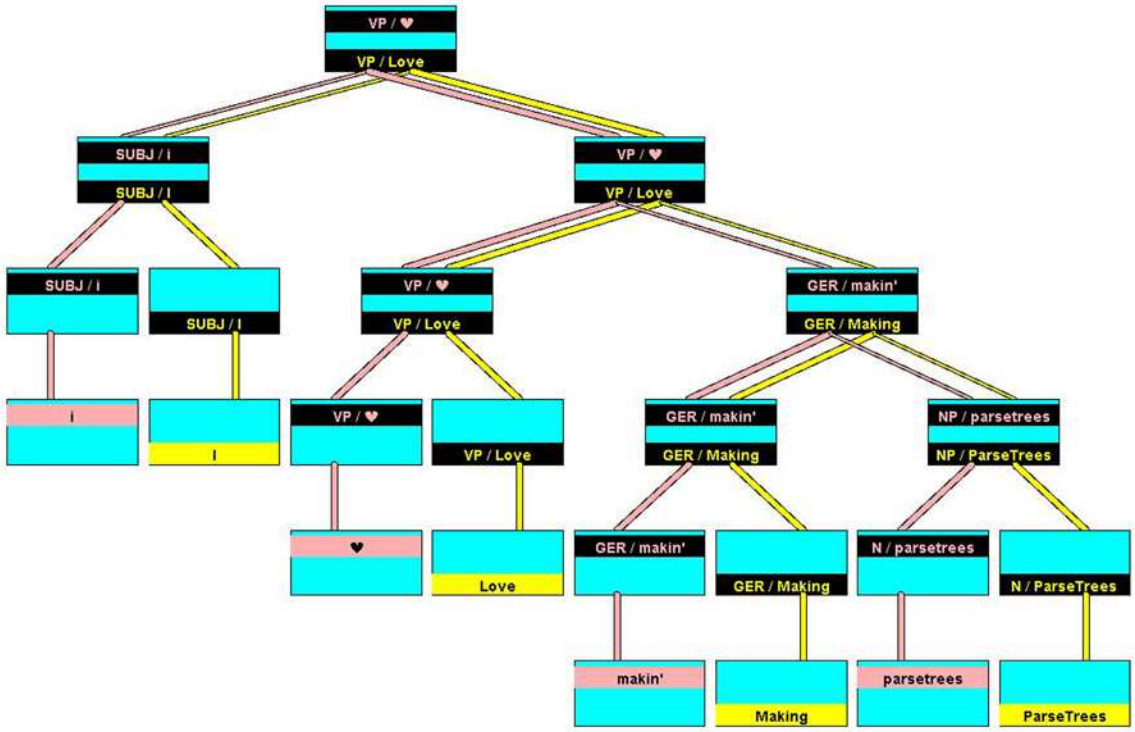

Figure 4.4: A typical tree view

The nested circles represent the 1-dimensional heir chain of the terminal; the innermost circle represents the highest node which received that lexical head (regardless of what happened in the other dimension).

The curved arrows above the top circles and below the bottom circles represent 1-dimensional dependencies. In a full parse, there will always be an arrow from each innermost circle to a nested circle elsewhere (unless user selects the parent-less ROOT). In a partial parse situation, the user can expect to see far fewer dependency arrows and nested circles as there is not as much parent/child information in the parse.

The straight lines between circles indicate a lowest link (see glossary).

Tree View

The root is the highest node in the tree, if it is a full parse. Nonterminal nodes are designated by colored text on black backgrounds, terminal nodes are designated by black text on colored backgrounds.

The nonterminal category and lexical head information for each language is displayed on the node. By convention, dimension 1 information is on the top row and dimension 2 information is on the bottom.

The lines between nodes indicate a parent/child relationship. A thin line indicates a child is dependent on its parent; a thick lines indicates the child is the heir of the lexical head.

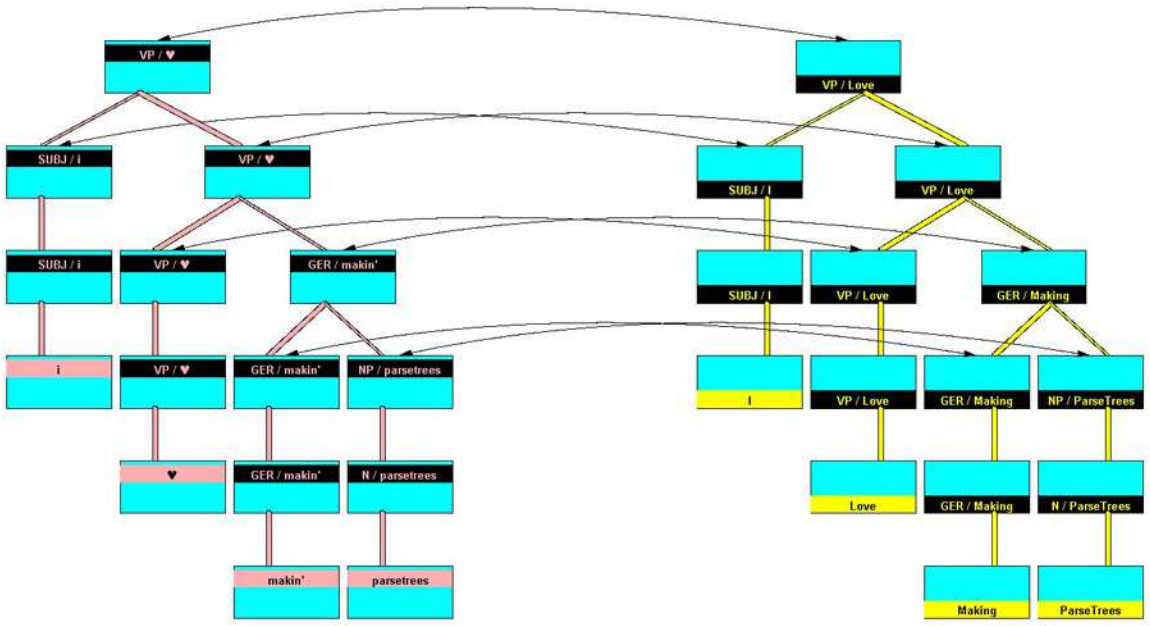

Figure 4.5: A typical side by side tree view

The absence of one of the rows indicates that dimension is null for that word (this is always the case with terminals and preterminals when the tree is generated by a GMTG in GCNF).

Because of the possibility of differing PAVs in the multiple dimensions, the ordering of the multidimensional nodes is not significant. They are simply output in the order they are generated.

Side by Side Tree View

This view displays the multitree separated into two 1D trees with links between them.

The root is the highest node in the tree, if it is a full parse. Nonterminal nodes are designated by colored text on black backgrounds, terminal nodes are designated by black text on colored backgrounds. The nonterminal category and lexical head information for each language is displayed on the node.

The lines between nodes indicate a parent/child relationship. A thin line indicates a child is dependent on its parent; a thick lines indicates the child is the heir of the lexical head.

This view is different from the Tree view in the following ways:

- A node will be drawn in a dimension's parse tree only if it is non-null in that dimension. Because we are now drawing one dimension's parse tree at a time, there is only one precedence array per node and therefore MTV can order the nodes according to the precedence array, so the ordering of the nodes is significant.

- The sentence can be read off by reading the terminals from left to right.

Chapter 5

Pilot Experiments

The vast majority of the workshop was devoted to software engineering. The experiments described in this section were executed in great haste during the closing days of the workshop. After the workshop, we discovered (and fixed) several serious bugs in our software. Therefore, **the results in this section are NOT indicative of GenPar's potential**, even with relatively primitive machine learning techniques. If we assume that bug fixes can only improve speed and accuracy, then the results that we report here can serve as very loose lower bounds. Even with these loose lower bounds, we can make a few claims:

- Our baseline models produce translations of a quality comparable to baseline finite-state models.
- Our software scales to hundreds of thousands of sentence pairs, without much optimization, on commodity hardware.
- Our models can handle language pairs with widely divergent syntax.
- It is straightforward to improve our joint translation models with target language models, the same way it is done in the noisy-channel paradigm.

Our main measure of translation quality was the F-measure, as computed by the General Text Matcher software [Turian et al., 2003]. This measure has an "exponent" parameter that controls the relative importance placed on longer matching n-grams. Setting this parameter to 1 eliminates all importance for longer n-grams, and reduces the measure to an F-measure over a bag of words. An exponent of 2 gives longer matching n-grams quadratic gain and, conversely, quadratic loss for unmatched n-grams. In the results tables that follow, FMS-1 stands for the F-measure with exponent 1, and likewise for FMS-2. Turian et al. [2003] showed that even with the exponent set to 1, the F-measure correlates with human judgments more highly than the BLEU score or the NIST score. We report the latter two scores because this has become standard practice in the MT literature. However, we are not aware of any situation in which these measures would be more informative than the more principled and less biased F-measure.

5.1 Baseline Model

Our baseline experiments used a simplistic generative model, inspired by the lexicalized head-driven models of Collins [1997]. In practice, because of the sparse data problem, our model will have poor

estimates for most of the probabilities that we need. To make better use of the training data, we break down the PMTG productions into more fine-grained elements, and assume some statistical independence among these elements. In particular, we shall assume that the nonterminal links on the RHS of a given production are rewritten in a depth-first order. We shall then assume that, given the LHS, the RHS of each production is generated in stages, as follows:

1. Generate a valency v for the production, from a multinomial distribution over non-negative integers in a small finite range. In the HeadPMTGA grammar that we used, which is binarized, the valency is either 0, for terminal productions, or 2, for nonterminal productions.

2. **IF** $v == 0$
   THEN it's a terminal production, so copy out the lexical heads of the LHS to the RHS and quit;
   ELSE it's a nonterminal production, so continue.

3. Generate the HeirRoleVector for the production, which determines which child is the lexical heir in each component.

4. Generate the heir nonterminals of the LHS, which inherit the lexical heads. (I.e. only the syntactic category labels are chosen stochastically, so the probability of a nonterminal lexicalized with a different head is zero.)

5. Generate the remaining (non-heir) nonterminal links as dependents of the LHS. (This includes the lexical head and syntactic category).

6. Generate a D-dimensional PAV for the dependents, where D is the dimensionality of the LHS. The PAV must be in normal form, so that the order of the dependents in the originally generated list is not disturbed in at least the first dimension.

In principle, each of the decisions in the above stochastic process can be conditioned on the entire derivation history up to that decision point. In practice, we must make some independence assumptions to reduce the effects of sparse data. As a starting point, we will assume that

- The valency of each production depends only on the LHS.
- The HeirRoleVector depends only on the LHS and the valency.
- The syntactic categories of the heir portions of a link on the RHS depend only on the LHS.
- Each of the dependent portions of a nonterminal links depend only on the LHS of the production and, if applicable, any heir portion already generated in that nonterminal link in the prior step.
- Each dimension of the PAV depends only on that dimension of the RHS and that dimension of the LHS.

  Under the above assumptions, we can compute the probability of MTG productions as follows.

  - Let h_n, a_n and b_n be terminals.
  - Let $L[h_n]$ be a nonterminal on the LHS.

– Let $A[a_n]$ and $B[b_n]$ be nonterminals on the RHS.
– Let $\mathcal{P}$ be the PAV.
– Let $\mathcal{P}_i$ be the i'th dimension of the PAV.
– Let $\mathcal{H}$ be the HeirRoleVector.

Then, the probability of a **terminal production** is:

$$\Pr\begin{pmatrix} L[h_1] \\ L[h_2] \end{pmatrix} \Rightarrow \begin{matrix} h_1 \\ h_2 \end{matrix}\Big) = \Pr\left(v == 0 | \begin{matrix} L[h_1] \\ L[h_2] \end{matrix}\right) \tag{5.1}$$

And the probability[1] of a **nonterminal production** is:

$$\begin{aligned}
\Pr\begin{pmatrix} L[h_1] \\ L[h_2] \end{pmatrix} \Rightarrow \mathcal{P}, \mathcal{H}, \begin{matrix} A[a_1] & B[b_1] \\ A[a_2] & B[b_2] \end{matrix}\Big) = \\
&\Pr\left(v == 2 | \begin{matrix} L[h_1] \\ L[h_2] \end{matrix}\right) \\
&* \Pr\left(\mathcal{H} | \begin{matrix} L[h_1] \\ L[h_2] \end{matrix}\right) \\
&* \begin{cases} \Pr\begin{pmatrix} A \\ A \end{pmatrix} | \mathcal{H}, \begin{matrix} L[h_1] \\ L[h_2] \end{matrix}\Big) \text{ if } \mathcal{H} = \begin{matrix} 1 \\ 1 \end{matrix} \\ \Pr\begin{pmatrix} A \\ \end{pmatrix} | \mathcal{H}, \begin{matrix} L[h_1] \\ L[h_2] \end{matrix}\Big) * \Pr\begin{pmatrix} \\ B \end{pmatrix} | \mathcal{H}, \begin{matrix} L[h_1] \\ L[h_2] \end{matrix}\Big) \quad \text{if } \mathcal{H} = \begin{matrix} 1 \\ 2 \end{matrix} \end{cases} \\
&* \begin{cases} \Pr\begin{pmatrix} B[b_1] \\ B[b_2] \end{pmatrix} | \mathcal{H}, \begin{matrix} L[h_1] \\ L[h_2] \end{matrix}\Big) & \text{if } \mathcal{H} = \begin{matrix} 1 \\ 1 \end{matrix} \\ \Pr\begin{pmatrix} B[b_1] \\ \end{pmatrix} | \mathcal{H}, \begin{matrix} L[h_1] \\ L[h_2], B[b_2] \end{matrix}\Big) * \Pr\begin{pmatrix} \\ A[a_2] \end{pmatrix} | \mathcal{H}, \begin{matrix} L[h_1], A[a_1] \\ L[h_2], \end{matrix}\Big) & \text{if } \mathcal{H} = \begin{matrix} 1 \\ 2 \end{matrix} \end{cases} \\
&* \sum_{i=1,2} \Pr\left(\mathcal{P}_i | L[h_i], A[a_i], B[b_i]\right)
\end{aligned}$$

The above generative process and independence assumptions are encoded in GenPar's HeadPMTGA. There are many other ways to stochasticize an MTG. Each makes a different set of independence assumptions. In the maximum likelihood paradigm, fewer independence assumptions usually lead to more computationally expensive parameter estimation. More independence assumptions lead to less predictive power.

5.2 Arabic-to-English

For the Arabic-to-English experiments, since we had access to syntactic parsers for both Arabic and English, we used parse trees for both sides of the input in the hierarchical alignment process. The preprocessing decisions made for Arabic originate in the task of parsing. In order to parse the Arabic data, we required a syntactically annotated corpus. We follow the procedure described by Mona Diab for parsing Modern Standard Arabic (MSA) [Diab, 2005].

[1] We show the probability when $\mathcal{H} = \begin{matrix} 1 \\ 1 \end{matrix}$ or $\begin{matrix} 1 \\ 2 \end{matrix}$. A similar approach is used when $\mathcal{H} = \begin{matrix} 2 \\ 1 \end{matrix}$ or $\begin{matrix} 2 \\ 2 \end{matrix}$

Arabic is a morphologically rich language which encodes many syntactic functions in inflectional affixes. For example, definiteness, which is expressed in English by the articles *the* and *a*, is expressed in Arabic by a prefix that appears only when a word is definite. Similarly, prepositions are often agglutinated with the words to which they attach. This agglutination is often fusional, meaning that the modified word is transformed to allow for the composite of the word and the preposition. Languages of this type must be dealt with differently than languages like English in order to perform syntactic analysis. In the following sections we describe the Arabic preprocessing stages that address these linguistic features.

5.2.1 Arabic Preprocessing

In this section we describe the tools used for preparing the Arabic side of a parallel Arabic-to-English corpus for use with GenPar. The preprocessing objective is to produce syntactically annotated Arabic sentences (Arabic parse trees). We used Mona Diab's SVM-based Arabic morphological analysis toolkit (ASVMT) to perform tokenization, lemmatization (described shortly), and part-of-speech (POS) tagging [Diab et al., 2004]. Additionally, these tools generate base-phrase labeled bracketings; however, our Arabic parser was unable to make use of these.

A brief description of each of the preprocessing stages follows:

Tokenization Tokenization of Arabic involves breaking apart syntactic units (those that resulted from inflectional morphological agglutination). The version of the ASVMT used attempts to make the same tokenization decisions that were made in the construction of the Penn Arabic Treebank [Maamouri et al., 2004, LDC, 2005]. During the workshop, Mona Diab provided a version of the Penn Arabic Treebank that was modified to be consistent with additional tokenization decisions made by the ASVMT tokenizer.

Lemmatization When an Arabic function word (e.g., a preposition) combines with another word, the word forms may change (e.g., the final letter of a word may change to accommodate the affixed preposition). During tokenization, the words are split into their components; however, the stem may have been altered in the original agglutination. Lemmatization, as it's called in the ASVMT, is the process of recovering the original unmodified version of the stem words.

POS tagging The ASVMT produces POS tags according to the reduced tag set for the Penn Arabic Treebank. The reduced set is consistent with the POS tags in the modified Treebank.

Parsing Dan Bikel's parser is used along with the Arabic parsing configuration file [Bikel, 2004]. The parser was trained on the training section of the Penn Arabic Treebank (parts 1, 2, and 3). Mona Diab provided a split of the data which is to be released as the standard training, development, and test sets for the treebank. We used the parser with POS tags generated by the ASVMT tagger.

English Tokenization As with the Arabic data, raw English text is tokenized prior to training or evaluation. We use the standard English preprocessing scripts as provided by the University of Pennsylvania for the Penn Wall Street Journal Treebank.

English POS tagging and Parsing The English side of the parallel corpus was parsed using Dan Bikel's parser. Standard training data is used, specifically the Penn Wall Street Journal Treebank sections 2-21. We used the Ratnaparkhi POS tagger [Ratnaparkhi, 1997] as a preprocessing stage prior to parsing.

5.2.2 Word Alignments

The GenPar hierarchical alignment algorithm is constrained by a word-alignment model. In the Arabic-English experimental framework, we used GIZA++ to generate the alignment model. We chose to use a simplified version of the bidirectional alignment technique that was used for French-English (described above). An outline of the process is as follows:

1. Generate GIZA++ alignments for Arabic-to-English and English-to-Arabic.

2. For each sentence, take the union (or intersection) of these alignments. The resulting set is the set of training alignments.

3. Compute a joint probability distribution over the alignments for the entire training set. We use the Maximum likelihood estimate as computed by the relative frequency estimator:

$$\frac{c(w_{arabic}, w_{english})}{\sum_{w_{arabic}, w_{english}} c(w_{arabic}, w_{english})}$$

   This results in a bilexical probabilistic dictionary.

4. For each sentence pair in the training set, extract all word-pairs (candidate alignments) for which there is a non-zero probability in the bilexical dictionary. Add the word-pairs with their probabilities to the candidate set of alignments. These are used as constraints on the tree-alignment algorithm.

In the current implementation, we provide for generating the word-alignment constraints via a unidirectional alignment, the union of a bidirectional alignment, or the intersection of a bidirectional alignment. In the experiments explored during the workshop period, we used the union method to estimate the probabilistic dictionary.

5.2.3 Arabic Data

Both the Arabic and English parsers used have been trained on news text: the English parser is trained on the standard Penn Treebank training set, and the Arabic parser is trained on the Penn Arabic Treebank. In the following experiments, we train the word-alignment models and GenPar on the Arabic-to-English parallel news corpus (LDC catalog number: LDC2004T18). We restrict the training data to contain only one-to-one sentence alignments. Furthermore, we exclude sentence pairs for which one of the preprocessing stages had failed (POS tagging, parsing, etc.). This results in approximately 43K sentence pairs from which we selected subsets in order to explore the performance and accuracy of the GenPar toolkit.

The NIST 2003 machine translation evaluation set (MTeval03) is used as the evaluation dataset for our experiments. The results reported in the next section are for tokenized versions of the hypothesized translations and reference translations.

5.2.4 Experiments

In this section we report the results for a baseline GenPar system with a small dataset. The word-to-word constraining model is trained on the full corpus of 43K sentence pairs. Due to the relatively unconstrained model, the grammar induced from pairs of parse trees becomes large. Since these

experiments were run before we fixed numerous memory leaks, we limited our experiments to a subset of the training corpus. We report results for a subset of 7k sentence-pairs. This data set was additionally filtered by removing sentences with more than 8 commas; we do this in order to exclude *flat* structures from the training trees.

In the first set of experiments, we explore various levels of translation-time pruning for a model trained on 7K sentence-pairs.

Translation pruning	Off	Medium	Heavy
FMS-1	0.1720	0.1720	0.1700
FMS-2	0.0831	0.0831	0.0820
BLEU	0.0151	0.0151	0.0149
NIST	1.7606	1.7606	1.7680

In the second set of experiments we explore the same translation-time pruning thresholds for a model trained on the same 7K dataset, but with training-time pruning:

Translation pruning	Off	Medium	Heavy
FMS-1	0.1751	0.1751	0.1749
FMS-2	0.0829	0.0829	0.0825
BLEU	0.0089	0.0089	0.0088
NIST	1.6555	1.6555	1.6737

As expected, the accuracy of the model decreased when pruning is applied during training; however the purpose of pruning is to reduce the computational effort exerted during training and translation. Currently, we are utilizing a relatively crude pruning strategy which may explain the loss in accuracy. In fact, the results indicate that in order to increase training speed without loss in accuracy, we must develop more accurate pruning criteria.

These preliminary Arabic-to-English translation results suggest that GenPar is capable of learning from syntactic trees for which there is a large amount of structural divergence.

5.3 French-to-English

Our French-to-English experiments were carried out on the EuroParl corpus [Koehn, 2005]. We automatically parsed the English section of the corpus using Dan Bikel's parser [Bikel, 2004]. The resulting data was then split into training sets consisting of 5K, 10K, 50K and 100K randomly selected English–French sentence pairs, with the English side containing parsed sentences. In addition we held out a number of development sets from the original corpus for each training set. The size of the development data was set at 10% of the size of the testing data (e.g. 500 sentences were in the development set for the 5K training set). The development sets and training sets were disjoint.

Word alignment was performed using the *refined* word alignment method as described by Tiedemann [2004] (following the method described by Och and Ney [2003], or Koehn et al. [2003] for a more recent implementation). Word alignment is performed in both source-target and target-source directions. These unidirectional alignments are then combined and the intersection is taken, producing a set of highly confident word alignments. The intersection set can then be extended iteratively by adding adjacent alignments present within the union of the unidirectional alignments.

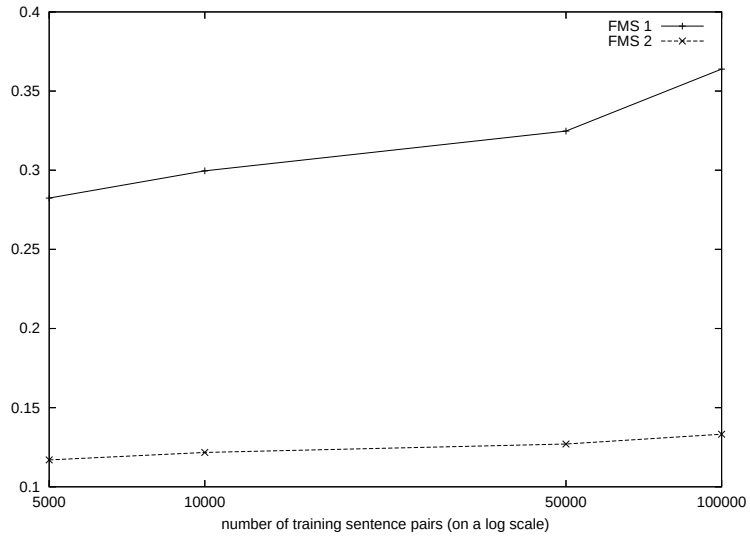

Figure 5.1: Learning curve for French-to-English experiments

This process is continued until no new adjacent alignments can be added. In a final step, alignments are added to the set that occur in the union, where both the source and target words are unaligned.

The parsed English sentences, their un-parsed French translations, and the corresponding word alignment sets were submitted to the hierarchical alignment process. We did not have a French parser, so for the French-to-English experiments the hierarchical alignment process was constrained by the word-to-word links and by English parses only. The resulting multitreebank was fed into a Viterbi training process for HeadPMTGA (described in previous section). During training, a best-first search strategy was used, with relative pruning set at -ln(0.2). This pruning is quite harsh but was set initially in order to facilitate the actual performance of experiments, due to the fact that at earlier stages of development the training was much less efficient and far from optimal. In order to maintain consistency this pruning threshold was maintained across all training set sizes. During translation no pruning was used. For each training set, translation was performed on the corresponding development set. The results are in Table 5.1 and Figure 5.3.

Table 5.1: French-to-English translation baselines. Training set size is in terms of number of training sentence pairs.

training set size	FMS-1	FMS-2	BLEU	NIST
5K	0.2824	0.1170	0.0409	2.6369
10K	0.2996	0.1217	0.0499	2.8515
50K	0.3247	0.12703	0.0624	3.0895
100K	0.3639	0.13321	0.0779	3.3484

5.4 Target language models

A close look at the bilingual parsing process revealed that the competing translations for particular input spans differed widely in fluency. There were two competing items on the agenda, for example, covering the French NP *la politique régionale*. The English output dimension of one item was *policy Regional*, another item contained *regional policy*. As it happened, the bilingual grammar assigned the first item a higher score although the second one contained the better English translation.

Incorporating a target language model can alleviate this problem of non-fluent translations. We ran some basic experiments, mixing a bilingual grammar (HeadPMTGA) with a bigram grammar.[2] The scores were calculated using a simple factorization:

$$Pr\left(\begin{array}{ccc} S & & NP \quad VP \\ S[I\ see\ you] & \rightarrow & NP[I] \quad VP[see\ you] \end{array}\right) = Pr\left(\begin{array}{ccc} S & & NP \quad VP \\ S & \rightarrow & NP \quad VP \end{array}\right) \cdot Pr(I\ see\ you) \quad (5.2)$$

For the bilingual grammar, we used the models described in Section 5.2.4. We trained three different English bigram models on 10, 15, and 20 million words of the English Gigaword corpus.

Table 5.2 and 5.3 show the results of tests on a set of 100 Arabic sentences, with different pruning thresholds. Throughout all experiments, mixing in a bigram model greatly improves the translation accuracy. The trend is: the better the bigram model, the better the translation results. Similar conclusions hold for experiments with French (Table 5.4 and 5.5). Here, we used the trained models described in 5.3, again with a smaller test set, i.e. the first 100 sentences.

The best results seem to reach those of other rudimentary SMT systems: We trained GIZA++ alignments and the CMU LM toolkit on the same 7k Arabic-English sentence pairs as described above. The results are comparable: FMS1: 0.2310, FMS2: 0.1024. However, the CMU toolkit was just trained with the default parameters.

These experiments show that it is possible to integrate a target language model into the bilingual parser. However, the amount of data used to train and test the model is very small. Furthermore, the factorized model is deficient. Members of our team are planning to run further experiments with integrated target language models on more data, using linear interpolation or log-linear models.

	No pruning	Medium pruning	Heavy pruning
No bigrams	0.1735	0.1735	0.1735
10M	0.2149	0.2149	0.2117
15M	**0.2337**	0.2295	0.2295
20M	0.2305	0.2305	0.2303

Table 5.2: FMS1 scores: Arabic-to-English with different bigram models. Adding a bigram model greatly improves the translation accuracy. Medium pruning means using a beam size of 1/10,000, heavy pruning means a beam size of 1/3.

[2] See Melamed [2004] for details of grammar mixtures.

	No pruning	Medium pruning	Heavy pruning
No bigrams	0.0837	0.0837	0.0837
10M	0.0967	0.0967	0.0934
15M	**0.1008**	0.0983	0.0983
20M	**0.1008**	**0.1008**	0.0987

Table 5.3: FMS2 scores: Arabic-to-English with different bigram models.

	Medium pruning	Heavy pruning
No bigrams	0.3140	0.2408
10M	0.3375	0.2751
15M	0.3355	0.2724
20M	**0.3417**	0.2853

Table 5.4: FMS1 scores: French to English with different bigram models. Heavy pruning (beam size: 1/3) degraded the French-English results. Still, the bigram models improved the results for both pruning experiments. Running the French experiments without pruning was not possible, due to memory limitations.

5.5 Late-Breaking Result

Just before this report was published, in November 2005, we decided to try an experiment with a configuration that was even more simplistic than the ones we used previously. We used an IndexedWMTG instead of the HeadPMTGA. The IndexedWMTG just memorizes whole production rules and does not permit the inference of any rule that was not seen in training data. We used 100K training sentence pairs in French and English, and tested on 50 previously unseen sentence pairs, without a target language model. The training and test sets were not exactly the same as the ones described in Section 5.3, so the results are not directly comparable. In the hierarchical alignment stage of the experiment, we tried two kinds of word-to-word translation models. The first model was induced by the *refined* method described in Section 5.3. The second model was induced by the very simple Method A of Melamed [2000]. Table 5.6 shows the results.

These results are based on a very small test set, so they are suggestive rather than conclusive. However, we were surprised that such a simple word-to-word translation model did not degrade accuracy, relative to a state-of-the-art model. We conjecture that word-to-word models developed for WFST-based SMT might not be good starting points for systems that are primarily driven by

	Medium pruning	Heavy pruning
No bigrams	0.1115	0.1035
10M	0.1200	0.1110
15M	0.1205	0.1105
20M	**0.1217**	0.1121

Table 5.5: FMS2 scores: French to English with different bigram models.

Table 5.6: French-to-English results starting from different word-to-word models.

word-to-word model	FMS-1	FMS-2
"refined" [Tiedemann, 2004]	0.41	0.21
Method A [Melamed, 2000]	0.43	0.20

tree-structured models. It is also worth noting that *only* the word-to-word model was varied in this experiment, and the rest of the system was *exactly* the same, which demonstrates how easy it is to run controlled experiments with the GenPar toolkit.

Chapter 6

Conclusion

It was a very hectic summer, and we are now a long way from where we started. Yet there is still a long way to go. We have laid the foundations, and now everyone can build on them. After the 1999 workshop, it took our research community several years to establish the superiority of the statistical approach. Likewise, we expect the full impact of our workshop to become apparent within the next couple of years. Workshop team members at at least three universities are continuing to work towards that goal, and new researchers from other institutions have already joined the effort. New collaborators are always welcome! To join the fun, see `http://nlp.cs.nyu.edu/GenPar/join.html`.

Appendix A

Glossary

This glossary contains terms relating to the bilingual case. All terms starting with *bi*, e.g. *biparsing*, generalize to the multilingual case by replacing *bi* with *multi*.

- **antecedent**: either an item or a production used to infer another item in an **inference rule**
- **axiom**: Goodman [1999] refers to productions or items which cannot be consequents as **axioms**
- **bitext**: see **parallel text**
- **component text / component**: one of the texts in a tuple of parallel texts; analogously, the part of a production rule or inference rule that pertains to one of the components of the multitext being generated or inferred
- **Compose inference**: an inference which combines items using a nonterminal production in a CKY-style logic
- **consequent**: an item created through use of an **inference rule**
- **dimension**: same as **component**
- **dimensionality**: number of components in a parallel text (This term generalizes to the dimensionality of a grammar etc.)
- ***D*-GMTG**: a **GMTG** which generates D-tuples of strings
- **generalized Chomsky normal form** (for **GMTGs**): a generalization of Chomsky normal for the GMTG case.
- **fully linked node**: a node (of a multitree) that is a lowest link or has an heir that is a lowest link
- **lowest link**: a multitree node that is the lowest node where some pair of nonterminals are linked
- **heir**: A constituent which is the head among its siblings.

- **heir vector**: a vector of heirs, usually one per dimension
- **inactive production component** (in a **GMTG**): whenever a component of a production on the LHS is (), the RHS of the component must also be (); in this case the production component is referred to as *inactive*
- **inference rule**: a rule schema showing how items can be deduced from existing items and productions; an inference rule consists of a set of **antecedents** and an inferred **consequent** (an instance of an inference rule schema is also referred to as an inference rule)
- **link** (also **NTLink**): links express the translational equivalence between nonterminals in different components of a production (there doesn't appear to be a specific term for this in the original papers; Lewis and Stearns [1968] talk about a "correspondence" between nonterminals; Aho and Ullman [1972] talk about an "association")
- **logic**: using a logic to describe a parsing algorithm is motivated and explained by Shieber et al. [1995] and Goodman [1999]. The logic describes how items can combine, abstracting away from any values being computed by the parsing algorithm (typically using some **semiring**) and the order in which items combine (except of course that any **antecedents** must be built before the **consequent** in an **inference rule**).
- **generalized multitext grammar (GMTG)**: grammar formalism used to generate tuples of strings (see relevant papers for formal definition)
- **multitree**: a D-tuple of trees plus a hierarchical alignment between all pairs
- **parallel text**: texts that are translations of each other

  item **precedence array**: a data structure that describes the relative order and contiguity of a set of nonterminals in one component of a production rule or inference rule
- **precedence array vector**: a vector of precedence arrays, usually one per component
- **production** (in a **GMTG**): a production rule consisting of a D-tuple of nonterminals on the LHS, and a D-tuple of sequences of terminals and indexed nonterminals on the RHS; indexes express translational equivalence between nonterminals on the LHS and RHS.
- **rank** of a production: the number of nonterminal links on the RHS
- **scan inference**: an inference which deals specifically with terminal productions
- **search strategy**: logic-based descriptions of parsing algorithms are (deliberately) nondeterministic; a search strategy determines the order in which the parse space is explored.
- **semiring**: an algebraic object which can be used to describe and unify many of the values typically calculated during parsing, e.g. Viterbi-best parse, parse forest, inside scores [see Goodman, 1999].
- **split heir**: an RHS of a production rule where the heir is not in the same link in all components

- **syntax-directed transduction grammar (SDTG)**: the original transduction grammar formalism defined by Lewis and Stearns [1968]
- **transduction**: a tuple of strings
- **transduction grammar**: a grammar that generates tuples of strings
- **translation**: mathematically, simply any mapping from one set of strings to another
- **valency**: number of children on RHS of production
- **Viterbi-derivation semiring**: a semiring which computes the highest-probability derivation

Appendix B

File Formats

MTV was originally designed to be bundled with the GenPar toolkit, so it expects input in the same format as the GenPar output. The format is similar to "standard" 1-dimensional tree and grammar output but with extensions to handle the multiple dimensions.
NOTE: Spaces are necessary between all symbols in the input, as is shown in the examples.

B.1 General Formatting Conventions

In all cases, dimension 1 information is given first and dimension 2 information is given second, and a `|` separates the two dimensions' info.

tree level: new nesting level is indicated by `(`, end of level indicated by `)`

nonterminal node: `( { NP [ Inauguration ] } | { NP [ for ] } )`
NT category on outside of `[ ]` , lexical head inside.

terminal node: `( also | null )`
Word in each dimension - one is always `null` .

PAV: `[ 1 2 | 2 1 ]`
Size of PAV corresponds to number of children under node, and numbers refer to nth child (1 is first), arrangement of numbers describes order of children.

Span: `[ 1 2 | ]`
Since span is used only for terminals in these files, one dimensions' span is always empty. Sentence span starts at 0. First word spans from 0 to 1.

heir marker: `* 0 * 1`
This indicates what dimensions this node is the heir in. `0` refers to dimension 1, `1` refers to dimension 2.

B.2 PMTG Grammar File

See example file EtoE.grammar in the MTV data directory The grammar reader expects to start reading in relevant information after encountering the first </SCORESET> tag in the file. After

that it expects to read in 8 scoresets of PMTG events separated by <SCORESET>,</SCORESET> tags. Each line should begin with an integer representing the number of times that particular event was seen, a tab, and then the event. The event scoresets must be in the following order: valency events, heir vector events, 2-dimensional heir events, dependency events, heir events in dimension 1, heir events in dimension 2, PAV events in dimension 1, and PAV events in dimension 2. The format for each particular event is explained with an example below:

valency event: LHS followed by integer representing valency.
`{ PP [ near ] } | { PP [ Near ] } 2`

heir vector event: LHS followed by valency, heir vector (first element indicates position of heir child in first dimension, second element indicates position of heir child in second dimension)
`{ PP [ at ] } | { PP+PP [ at ] } 2 1 2`

heir event: LHS followed by RHS heir (nonterminals only)
`{ NP [ Inauguration ] } | { NP [ for ] } { NNP } | { PP+<PP+PP> }`

dependency event: LHS followed by dependent RHS
`{ NP [ city ] } | { NP [ city ] } { PP [ of ] } | { PP [ of ] }`

1-d heir event: LHS followed by RHS with "heirmark" designating heir portion
`{ PP [ for ] } | { PP+PP [ for ] } { NP [ boy ] } | { PP+PP [ heirmark ] }`

PAV event: LHS followed by PAV
`{ NNP [ Free ] } { NNP [ Zone ] } [ 1 2 ]`

B.3 Multitree File

See example file EtoE.tree in the MTV data directory.

The MTV tree reader expects to see one parsetree per line. This parse may consist of one complete parse tree or many partial parse trees, separated by tabs. Each nonterminal node must start with a 2-dimensional PAV and each terminal node must end with a 2-dimensional span vector. All heir nodes must also begin with an heir marker. Because MTV must have a default node to align unlinked terminal nodes in partial parse trees to in Grid View, the terminal in the last position of the sentence should always be "eosh" (end of sentence head). The span in each sentence is assumed to begin at position 0. Though MTV can read deintegerized input, it maps every unique word/label to an integer internally. There are several keywords that MTV has pre-mapped to integers and always expects them to be represented by that integer in integerized input. Nonterminal keywords:

"NULL" : 0

"ROOT" : 1

"EOSH" : 2

"DUM" : 3

"UNK" : 4

"UNKPT" : 5

“UNDEF” : 6

“HEIRMARK” : 7

Terminal keywords:

“null” : 0

“root” : 1

“eosh” : 2

“dummy” : 3

“unk” : 4

“unkprtm” : 5

“undef” : 6

“heirmark” : 7

B.4 Vocabulary File

See example files L1.tvcb, L1.ntvcb, L2.tvcb, L2.ntvcb in data directory.
Vocabulary files must begin and end with <VOCAB>, </VOCAB> tags, respectively. The files must have one mapping per line. The first column indicates the integerization of the word/label in the second column. Columns are separated by tabs.

B.5 MTV Converter File

See example file arabic.converter in data directory.
The converter file must have one mapping per line, with at least 2 columns. The first column indicates the character that is to be replaced, and the second column indicates the Unicode that should replace that character. Anything after those columns will be ignored, so it can be left blank or commented for human readers. Columns are separated by tabs.

Bibliography

A. Aho and J. Ullman. Syntax directed translations and the pushdown assembler. *Journal of Computer and System Sciences*, 3:37–56, February 1969.

A. Aho and J. Ullman. *The Theory of Parsing, Translation and Compiling, Vol. 1.* Prentice-Hall, Englewood Cliffs, N.J., 1972.

Yaser Al-Onaizan, Jan Curin, Michael Jahr, Kevin Knight, John Lafferty, Dan Melamed, Franz-Josef Och, David Purdy, Noah A. Smith, and David Yarowsky. Statistical machine translation: Final report, JHU workshop 1999. Technical report, The Center for Language and Speech Processing, The Johns Hopkins University, www.clsp.jhu.edu/ws99/projects/mt/final_report, 1999.

Hiyan Alshawi. Head automata and bilingual tiling: Translation with minimal representations. In *Proceedings of the 34th Annual Meeting of the Association for Computational Linguistics (ACL)*, pages 167–176, Santa Cruz, USA, 1996.

Dan Bikel. A distributional analysis of a lexicalized statistical parsing model. In *Proceedings of the 9th Conference on Empirical Methods in Natural Language Processing (EMNLP)*, Barcelona, Spain, 2004.

David Chiang. A hierarchical phrase-based model for statistical machine translation. In *ACL05*, 2005.

Michael Collins. Three generative, lexicalized models for statistical parsing. In *Proceedings of the 35th Annual Meeting of the Association for Computational Linguistics (ACL)*, pages 16–23, Madrid, Spain, July 1997.

Mona Diab. *Documentation for the Arabic SVM Toolkit*, 2005. `http://www.cs.columbia.edu/ mdiab/`.

Mona Diab, Kadri Hacioglu, and Daniel Jurafsky. Automatic tagging of arabic text: From raw text to base phrase chunks. In *Proceedings of HLT-NAACL 2004*, 2004.

Katherine Eng, Alex Fraser, Daniel Gildea, Viren Jain, Zhen Jin, Shankar Kumar, Sanjeev Khudanpur, Franz Och, Dragomir Radev, Anoop Sarkar, Libin Shen, David Smith, and Kenji Yamada. Final report, syntax for statistical MT group, JHU workshop 2003. Technical report, The Center for Language and Speech Processing, The Johns Hopkins University, 2003.

Erich Gamma, Richard Helm, John Vlissides, and Ralph Johnson. *Design Patterns: Elements of Reusable Object Oriented Software.* Addison Wesley Longman, Inc., 1994.

Daniel Gildea. Loosely tree-based alignment for machine translation. In *Proceedings of the 40th Annual Meeting of the Association for Computational Linguistics (ACL)*, Sapporo, Japan, July 2003.

Joshua Goodman. Semiring parsing. *Computational Linguistics*, 25(4):573–605, 1999.

Mary Hearne. *Data-Oriented Models of Parsing and Translation*. PhD thesis, Dublin City University, Dublin, Ireland, 2005.

Mary Hearne and Andy Way. Seeing the wood for the trees: Data-oriented translation. In *Proceedings of Machine Translation Summit IX*, 2003.

Philipp Koehn. Europarl: A parallel corpus for statistical machine translation. In *Proceedings of MT Summit X*, Phuket, Thailand, 2005. International Association for Machine Translation.

Philipp Koehn, Franz Josef Och, and Daniel Marcu. Statistical phrase-based translation. In *Proceedings of the Human Language Technology Conference and the North American Association for Computational Linguistics (HLT-NAACL)*, pages 127–133, Edmonton, Canada, 2003.

LDC. Penn arabic treebank, 2005. Catalog numbers: LDC2005T02, LDC2004T02, and LDC2005T20.

P. M. Lewis and R. E. Stearns. Syntax-directed transduction. *Journal of the Association for Computing Machinery*, 15(3):465–488, 1968.

Mohamed Maamouri, Ann Bies, Tim Buckwalter, and Wigdan Mekki. The penn arabic treebank: Building a large-scale annotated arabic corpus. In *NEMLAR International Conference on Arabic Language Resources and Tools*, pages 102–109, Cairo, Egypt, 2004.

I. Dan Melamed. Models of translational equivalence among words. *Computational Linguistics*, 26 (2):221–249, June 2000.

I. Dan Melamed. Algorithms for syntax-aware statistical machine translation. In *Proceedings of the 10th Conference on Theoretical and Methodological Issues in Machine Translation (TMI)*, Baltimore, MD, 2004.

I. Dan Melamed, Giorgio Satta, and Benjamin Wellington. Generalized multitext grammars. In *Proceedings of the 42nd Annual Meeting of the Association for Computational Linguistics (ACL)*, Barcelona, Spain, 2004.

I. Dan Melamed and Wei Wang. Statistical machine translation by generalized parsing. Technical Report 05-001, Proteus Project, New York University, 2005. `http://nlp.cs.nyu.edu/pubs/`.

Franz Josef Och and Hermann Ney. A systematic comparison of various statistical alignment models. *Computational Linguistics*, 29(1), 2003.

Franz Josef Och, Christoph Tillmann, and Hermann Ney. Improved alignment models for statistical machine translation. In *Proceedings of the 4th Conference on Empirical Methods in Natural Language Processing (EMNLP)*, pages 20–28, College Park, Maryland, 1999.

Adwait Ratnaparkhi. A linear observed time statistical parser based on maximum entropy models. In *Proceedings of the 2nd Conference on Empirical Methods in Natural Language Processing (EMNLP)*, Providence, Rhode Island, 1997.

S. Shieber, Y. Schabes, and F. Pereira. Principles and implementation of deductive parsing. *Journal of Logic Programming*, 24(1&2):3–36, 1995.

Jörg Tiedemann. Word to word alignment strategies. In *Proceedings of the 20th International Conference on Computational Linguistics (COLING) 2004*, pages 212–218, Switzerland, August 2004.

Joseph P. Turian, Luke Shen, and I. Dan Melamed. Evaluation of machine translation and its evaluation. In Elliott Macklovitch, editor, *Proceedings of MT Summit IX*, New Orleans, September 2003. International Association for Machine Translation.

Dekai Wu. An algorithm for simultaneously bracketing parallel texts by aligning words. In *Proceedings of the 33rd Annual Meeting of the Association for Computational Linguistics (ACL)*, Cambridge, Massachusetts, June 1995a.

Dekai Wu. Stochastic inversion transduction grammars, with application to segmentation, bracketing, and alignment of parallel corpora. In *Proceedings of the 14th International Joint Conference on Artificial Intelligence (IJCAI)*, Montreal, Canada, 1995b.

Dekai Wu. Trainable coarse bilingual grammars for parallel text bracketing. In *Proceedings of the 3rd ACL Workshop on Very Large Corpora (WVLC)*, Cambridge, Massachusetts, 1995c.

Kenji Yamada and Kevin Knight. A decoder for syntax-based statistical MT. In *Proceedings of the 40th Annual Meeting of the Association for Computational Linguistics (ACL)*, pages 303–310, Philadelphia, July 2002.