High-Performance Multi-Pass Unification Parsing

Paul Wesley Placeway

May 14, 2002

CMU-LTI-02-172

Language Technologies Institute
School of Computer Science
Carnegie Mellon University
Pittsburgh, PA 15213

Submitted in partial fulfillment of the requirements
for the degree of Doctor of Philosophy.

Thesis Committee:
Eric Nyberg, Chair
Jaime Carbonell
Alon Lavie
Robert Bobrow, BBN Technologies

This research was supported in part by Carnegie Mellon University

Keywords: Parsing, Unification, Ambiguity

For Mary, Mom and Dad.

Abstract

Parsing natural language is an attempt to discover some structure in a text (or textual representation) generated by a person. This structure can be put to a variety of uses, including machine translation, grammar conformance checking, and determination of prosody in text-to-speech tasks.

Recent theories of Syntax use Unification to better describe the intricacies of natural language [137]. For parsing systems, unification techniques have been either added to a context-free base system [152, 40, 4, 23], or replaced the context-free base entirely [118, 135, 45] (possibly putting it back later [136]). The seemingly small step of adding unification has opened a Pandora's Box of computational complexity, increasing the difficulty of the problem from polynomial [48] to somewhere between NP-complete and intractable, depending on the details of the unification system and how it was added [10]. Worse, unification on a context-free base parser can break the packing technique used to address the problem of ambiguity, leading to exponential blow-ups of the parser's performance in both space and time in practice.

I propose the use of a multi-pass strategy to avoid these problems in practice. I describe a parser which combines the use of shallow, simple value unification with some approximation techniques in order to find a covering packed parse-forest. This parse-forest is then searched for a single-best fully-unifying value; the scoring system which drives the heuristic search encodes linguistically-based disambiguation preferences.

The resulting two-pass parser is compared to an ordinary single-pass parser in the context of a heavy-weight knowledge-based machine translation system. The two-pass parser is shown to be competitive with the single-pass parser on average data, both in terms of time and space. It is also shown to be able to avoid a common class of ambiguity blow-up that the single-pass parser is subject to. These results indicate that the multi-pass technique, interleaving *some* of the unification equations in the parse, is the superior approach for heavy-weight unification parsing.

Acknowledgements

I would like to thank the many people without whom this work would not have been possible:

- My advisor, Eric Nyberg, for asking the critical question "Why do these sentences take so long?", many technical and philosophical discussions, and for help in turning my writing into English.
- The members of my committee, Jaime Carbonell, Alon Lavie, and Robert "Rusty" Bobrow, for guidance in setting the technical direction of this work, many useful discussions, and their patience.
- Robert Moore, for technical advise related to high-performance context-free parsing.
- Kathy Baker, for supporting the KANT grammar, and for helpful technical discussions.
- Krzysztof Czuba, for providing the Broadcast News grammar and test set, and many helpful technical discussions.
- David Svoboda, for help in organizing the Catalyst 10,000 sentence test corpus.
- Robert Igo, for organizing the FOATS regression test corpus.
- The many other members of the KANT team.
- Finally, my wife Mary Placeway, for her love and support throughout the adventure of graduate school.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

In working with the KANT machine translation system, we discovered a problem: the vast majority of input sentences parsed very quickly, but a small minority required extremely large amounts of time to parse. These outliers often require several hundred times the amount of time to parse an average sentence. The discovery of these outliers begged the questions "why is this happening?" and "what can we do about it?" The answer involved the handling of values in the unification system, and bad interactions between these unification values and the ambiguity inherent in some sentences.

This work describes the detailed engineering of a new parsing system, designed to avoid the problem which creates this symptom. We will examine the problem in detail, create several novel techniques designed to work around aspects of the problem, and apply existing techniques in order to solve our problem. Careful engineering will be stressed throughout: the decision to adopt a technique or choose between competing techniques is made on the basis of empirical evaluation whenever possible.

1.1 Introduction

As we put natural language parsers to more elaborate and more realistic tasks, we keep changing the problem that the parser must solve [49, 40, 68]. Sometimes this is a simple change, but quite often it only *seems* to be simple [10].

It is tempting to treat parsing as a separable problem, and address it as a task in and of itself. This separation is useful because it allows one to examine a problem in the absence of distractions. On the other hand, a major disadvantage of this separation is that it can lead a researcher to overlook

certain aspects of the whole problem that they actually face, and cause them to mistakenly focus on details of lesser importance.

The theme we shall be exploring is how natural-language parsing has become more complex in practice, how this added complexity has resulted in a much more difficult computational task than one might expect at first glance, and what can be done to avoid some of the common problems that this added complexity aggravates. This will be investigated principally in the context of a large, practical natural-language processing system [89, 110, 111].

Natural language parsing is often introduced simply enough with context-free parsing [3, 48, 34]. Beginning with his description of formal classes of languages, Chomsky argued that the context-free languages are not sufficient to capture the subtleties of human language grammar [25, p. 120], citing mostly examples of tense and number as problematic. One more recent method of addressing these deficiencies led to the introduction of unification parsing [3, 137]. While the idea of adding unification to a context-free parser may at first seem simple [152], it turns out not to be. The main theoretical draw-back to unification parsing is that it can vary in difficulty from NP-complete (hard) [10] to Turing-complete (impossible) [137, 10]. For unification parsers in practice the situation is not so bad, but we will see below that a unification parser can suffer from exponential blow-ups of space and time requirements which will lead to unacceptable performance in practice.

To work around the difficulty, I propose the use of multi-pass parsing, whereby the solution space is narrowed down by applying inexpensive operations. Only after the space has been narrowed, are expensive operations carried out and a final parse found.

By delaying these expensive operations until they are actually known to be needed, the parser will avoid degenerate performance in the face of structurally ambiguous inputs, a commonly-occurring phenomenon in natural language which leads to bad performance in practice. The multi-pass technique will also save time compared to the common one-pass method. Finally, this multi-pass technique will also allow the efficient use of linguistically-motivated disambiguation techniques, which allow for the discovery of not just any resulting parse, but a *desirable* parse as the final result.

The culmination of this approach is a natural-language parser that is good from a linguistic standpoint (theoretically as well as practically), good from a computational standpoint (again both theoretically and practically), and good from a software engineering standpoint.

1.1.1 Statement of Thesis

My claim is that it is possible to build a parser that fulfils all of these desires. In particular, I claim that:

> By combining automatically-derived, simple-valued, interleaved partial unification in a high-performance parser with a post-parse search for a single complete unification value, we can avoid the complexity problems of ambiguity, while preserving the speed and grammar maintainability advantages of a simple bottom-up parser.

In the remainder of this document, we will examine a system that implements all of the elements of this claim, and experimental evidence that supports it. The description of the system is wide-ranging, working from theoretical concerns down though the particulars of the implementation of the techniques. We will also see how this work is related to, but different from, the past efforts of other researchers.

1.1.2 Summary of Contributions

In the process of developing this system, we have extended the state of the art in parsing in several ways. Our contributions include:

- Evidence that interleaved unification is generally preferable to fully-delayed unification evaluation.
- Evidence that delayed unification is preferable over single-pass interleaved unification.
- A novel technique to automatically produce a shallow unification grammar from a complex-valued grammar.
- The use of approximation of Shallow Interleaved Unification Values to gain efficiency in packing interleaved unification values.
- Various techniques in Unification Optimization.
- A novel, optimal algorithm to search a parse forest for a single, best value.
- Advances to Context Free parsing. These include the structuring of the grammar into a tree and the modified chart algorithm which uses this grammar structure, as well as the integration of various filtering techniques with this algorithm.

- Complete descriptions of all major portions of the implementation.
- Empirical investigation of many engineering trade-offs in the development of this parsing system.

1.1.3 Motivation: Why parsing is useful?

Parsing is about discovering a structure in an input, based on external information known about the elements of the input and their order. Generally, the external information consists of a lexicon, which is a list of input tokens (*e.g.* words), and a grammar, which describes which structures may be built from, and implied by, sequences of tokens.

For the general task of processing natural language, it is useful to discover structure in order to better process the input. So, for instance, one can do a mediocre job of translating a text from one language to another by translating each word independently [49, pp. 175–189], or a decent job considering the words in the context of other particular words [19, 14]. But one way of performing a high-quality translation is to parse the input, to discover some internal structure, and then use this structural analysis to better translate the input [40, 89, 131].

1.1.4 As part of Knowledge-based translation

Parsing is often the first major step in a machine translation system. A machine translation system attempts to translate input (often in the form of text) in some source language into some target language, while preserving as much as possible of the meaning (both explicit and implicit) of the input.

In Knowledge-based Machine Translation, the input is first parsed to discover its syntactic structure, referred to as its constituent structure or *C-structure*. This then is used to create a more abstract representation (such as a *feature structure* or *F-structure*). The F-structure can be turned into an even more abstract representation: an *Interlingua* representation that explicitly expresses everything that could be implicitly represented in any of the source or target languages. The Interlingua representation then can be turned into an F-structure appropriate to the target language. Finally, this target F-structure can be used to generate an appropriate target C-structure, and then output string [40, 89].

In this process, the combined steps of parsing the input to discover the F-structures and disambiguating, or choosing amongst multiple F-structures, is very important and one of the slowest parts of the process. One must remember that the disambiguation step is also part of this process — if we were to consider parsing alone, and not worry about disambiguation, we might be led to a solution

that very quickly produces a representation of all possible F-structures but does not disambiguate, or choose amongst them [114]. I will argue below that this outlook is insufficient because we have not chosen amongst these alternatives, and thus not actually completed the parsing task required by a machine translation system.

Once an input has been parsed and disambiguated, the remaining steps of the machine translation system are fairly rapid [49, 40]. Therefore it is desirable for a practical machine translation system to include a high-performance parser as a component.

1.1.5 Checking conformance to a restricted language

Experience with the KANT system [87, 111] has shown that it can be helpful to restrict the input language of a machine translation system. Doing so allows a gain in accuracy and coverage within the limited domain. But the trade-off is that a documentation writer must manage to write within the restricted language.

Since checking conformance is something that computers can do much better than people, it is useful to provide an automatic way to check for such conformance. Of course, the most straightforward way to do this is to separate out the parser and grammar from the beginning of the machine translation system, package them separately as a checking system, and provide this checker in a software package that is convenient for the writer to use [87].

The performance of this conformance checker could be improved upon further by noting that a complete parse and F-structure analysis may not be required — all that is really needed is a "yes/no" indication for each input sentence, along with helpful information on what problem the parser had for the "no" responses. Because of this, finding all the F-structures of all possible parse trees is simply too much work.

1.1.6 Discovering prosody in text-to-speech

Yet another use for a parser is to improve on the prosody of text-to-speech systems. It is useful to chose the correct homograph, for example choosing correct pronunciation for *record* in these examples:

```
Please record these results.
Put another record on the phonograph.
```

In this case, the former is a verb, whereas the latter is a noun. (There are other cases for which the analysis must go beyond a simple part-of-speech analysis, for instance the two nouns *bow*: the

bow of a ship vs. a *bow* used to shoot arrows [35].)

In the case of *record*, the primary difference is in which syllable is stressed. However, this is not the only way in which stress is important in English. Stress is also used to mark questions as distinct from statements, as well as mark main clauses as differing from subordinating clauses [66].

In order to produce higher quality speech output, a syntactic analysis of the text input can be quite helpful [77]. Like any other problem of engineering, there can be a trade-off between the level of detail required for a particular text-to-speech task and the system resources available to this part of the larger system. Up to a point, if we can produce a more accurate analysis with fewer resources, the overall text-to-speech process can be done with higher quality, or fewer resources, or both.

1.2 Dissertation Overview

Before getting into the details, it is helpful to see where we should be headed. Thus, a brief overview of this document will be helpful to see how all the pieces fit into the whole.

- In the next chapter, we illustrate several instances of how parsing fits into the larger scheme of natural language processing, and argue that increased performance remains a worthy goal in natural language processing. Section 2.1 briefly reviews some aspects of parsing, unification, linguistic theories of grammar, and results in theoretical computer science that have particular bearing on this work. Section 2.2 briefly reviews several practical applications for natural language parsers.

- Chapter 3 examines several options for implementing a unification parser that exist in the literature, with emphasis on the parsers which use the sorts of unification grammars we are ultimately interested in.. It then examines ways in which unification parsers can experience blow-ups in space and time, despite efforts to prevent these same situations in the context of strictly context-free parsing. The situations that lead to these blow-ups are demonstrated to be realistic in an extant system.

- Chapter 4 examines how much of the unification process could be delayed until *after* finding a parse forest which covers the set of all valid parses.

- The next four chapters examine the pieces which we will use to work around and avoid these problems, as well as achieve a truly high-performance parsing system:

- Chapter 5 examines our goals and requirements for a high-performance unification parser. It then surveys some options in the general architecture of a parsing system, and examines why

one of these architectural options can address the complaints we have previously exposed. Finally, it examines the actual architectural design of our system, and examines the components from which our system is built at high level.

The next three chapters take an extremely detailed look at the three major components of the system.

- Chapter 6 introduces an improvement to context-free parsing. The *tree-structured grammar* is an improvement to the structure of a chart parser that eliminates redundant processing. *Left-corner* filtering is a technique that greatly reduces ultimately unproductive processing in a chart parser. This section describes how to use these two techniques together, and describes a system that implements this combination of techniques, as well as others necessary for the construction of a modern high-performance parser. It concludes with a set of experiments demonstrating the effectiveness of this combination of techniques.
- Chapter 7 reviews our particular pseudo-unification framework, and examines how pseudo-unification can be considered to be a "little (programming) language" [13, ch. 9]. Section 7.2 discusses a set of modifications which were made to the pseudo-unification system for this work. In Section 7.3, the techniques used to compile this little language into executable code are discussed, and the pseudo-unification optimizer is introduced.

  These optimization techniques are then extended in order to support the implementation of the shallow unification techniques introduced Chapters 4 and 5. These shallow unification techniques are shown to preserve much of the effectiveness of the *pruning* effect of interleaved unification while avoiding the space blow-ups we discovered in Chapter 3.

  Finally, a number of apparent optimizations to the unification which did not actually improve performance are discussed.
- In order to take advantage of a parse forest which covers a superset of all valid parses, we must find the final unification value for a single valid parse tree. Chapter 8 reviews prior work in post-parse search and in sentence disambiguation, and then introduces our technique for performing a post-parse search to simultaneously solve the problems of finding a fully-unifying result, and performing disambiguation in a linguistically-motivated way.

  Section 8.2 introduces the Iterative-Deepening Greedy Best-First search which is used to find a best-scoring, complete unification value. This search is described in detail, and shown to be optimal by means of a comparison in function to A* search [31]. A

cost calculator is necessary to effectively drive this search. Section 8.3 describes the reference knowledge-based disambiguation cost calculator.

A number of empirical tests are used to clearly see the benefits and trade-offs of the many design decisions made in these chapters. The conditions of these development tests are discussed in Section 5.4.

- In Chapter 9, the system is evaluated on independent test corpora in order to fairly evaluate its performance. The effectiveness of the full system is examined using an independent test corpus of real-world data. The system is shown to be competitive in the average case.

  Finally, the generality of this set of techniques is examined using a test grammar and corpus which have not been specially adapted to this system.

  Chapter 10 reviews the work as a whole, and conclusions are presented. Contributions of this work to the state of the art are examined, and several directions for future work are examined.

Chapter 2

Background

> "Those who cannot remember the past are condemned to repeat it."
>
> — George Santayana, *Life of Reason, Reason in Common Sense* [130]

2.1 General Background

Before examining the technical details of unification parsing, we should first define a few terms and problems related to parsing. This will make a number of technical issues in the following text much more clear.

2.1.1 Preliminaries

Generation and acceptance Formally, we can either look at a formal language from a generative standpoint, where the grammar is used to *output* a string, or from an acceptance standpoint, where the string is considered *input* to a process that will answer `yes` if the input is 'in' the grammar or `no` if it is not. These are actually the flip-sides of the same metaphoric coin; even though we will be mostly concentrating on treating the string as input, we will occasionally look at the problem in the other direction. This is not simply an abstraction for the purpose of explanation; the final stage of the Knowledge-Based Machine Translation system described above actually implements the output process [40], [49, pp. 131–145].

Recognition The recognition problem is the "yes/no" version of the problem:

Given some grammar and lexicon G, and some input string I, is I in the language described by G? Could G have generated I, or equivalently does G accept I?

This is conveniently simple: either *I* parses, or it doesn't. It does not matter that there may be multiple ways to generate *I*, not even if there an infinite number of ways.

Parsing The parsing problem is not only to answer the above "yes/no" question, but also to come up with some number of sets of derivations (often expressed as trees) that could produce *I*. We will see that in some cases, the problem can be made easier by only requiring *one* example derivation; but in other cases this does substantially change the problem.

Given G and I, find not only if G accepts I, but if so what set of productions would cause G to generate I? Or what set of sets?

Words We have not been clear about the *acceptance* or *generation* means, nor about the structure of *I*, nor that of *G*.

We will say that an *input string* is an ordered set of *tokens*, which we will generally refer to as *words*. A *word* is considered to be an atomic unit. A *lexicon* is a relation between *words* and *features* that these words can entail. We will generally assume that there are a finite number of possible *words* and *features*.

2.1.2 Families of grammars

Chomsky defined a family of classes of grammars, each in larger than the next [25, 48]. By larger, we mean a subset inclusion relation: language family *F'* is as large as *F* if and only if every language *L* in *F* can also be generated by *F'*. *F'* is (strictly) larger than *F* if there is some language *L'* that is in (can be generated by) *F'*, but cannot be generated by *F*.

We will not give proofs of separation of these families of languages. The characterizations of these families are derived from Hopcroft & Ullman [48]; proofs of the separation of families, as well as a wealth of other information about the formal properties of languages can be found in this reference.

The following three language families are particularly relevant to this work:

- *Context-Free*
- *Context-Sensitive*

- *Unrestricted*

Context-Free Languages and Push-Down Automata

Hopcroft & Ullman's description of context-free languages is remarkably clear. First they present an informal definition:

> A *context-free grammar* is a finite set of variables (also called *nonterminals* or *syntactic categories*) each of which represents a language. The languages represented by the variables are described recursively in terms of each other and primitive symbols called *terminals*. The rules relating the variables are called *productions*. A typical production states that the language associated with a given variable contains strings that are formed by concatenating strings from the language of certain other variables, possibly along with some terminals.
> [48, ch. 4, p. 77]

And then, after some examples, they formalize it:

> Now we shall formalize the intuitive notions introduced in the previous section. A *context-free grammar* (*CFG* or just *grammar*) is denoted $G = (V, T, P, S)$, where V and T are finite sets of *variables* and *terminals*, respectively. We assume that V and T are disjoint. P is a finite set of productions; each production is of the form $A \to \alpha$, where A is a variable and α is a string of symbols from $(V \cup T)*$. Finally, S is a special variable called the *start symbol*.
> [48, ch. 4, p. 79]

In the next chapter of this book, Hopcroft & Ullman note that, from a computational standpoint, context-free language recognition is computationally equivalent to a nondeterministic push-down automaton [48, pp. 107–119].

Context Sensitive languages

The *context-sensitive languages* can be characterized in a manner similar to the context-free language formalization above, except that productions can take the general form $\alpha \to \beta$, where α or β are both arbitrary strings of terminals and non-terminals, but with the restriction that the length of β must be at least as long as α [48, p. 223].

Context Sensitive Language recognition is computationally equivalent to Linear Bounded Automaton (Linear Bounded Turing Machine) (with the exception that the LBA can accept the empty string ϵ, but this string cannot be generated by a CSL) [48, pp. 225–226]. The recognition problem for context sensitive grammars is also PSPACE-complete [48, pp. 346–347]. (Also, Quantified Boolean Formula satisfiability ("QBF") is PSPACE-complete [48, pp. 343–346].)

Unrestricted languages and Turing Machines

The *unrestricted languages* can be characterized in a similar manner, with productions taking the form $\alpha -> \beta$, but *without* restrictions on the lengths of either α or β [48, p. 220].

Unrestricted, or type-0, language recognition is Turing Complete [48, pp. 220–223], hence the unrestricted languages are exactly the recursively enumerable ones [48, p. 150].

2.1.3 Context-Free Parsing

Early work in Computational Linguistics and Automatic Program Translation (*i.e.* computer language compilers) centered on using context-free grammars to discover the structure some input [64, 42, 163, 34, 2].

The Context-Free family of grammars is a fairly descriptive

For example, if we have the terminal symbols *Det* (for *determiner*, such as "the" or "a") and *N* (for *noun*, such as "car" or "man"), we can write a rule to describe the structure of such noun phrases as "a man" or "the car" with the rule:

$$NP \quad \leftarrow \quad Det \quad N$$

If we also had the terminal *V* to describe verbs such as "drove", and the punctuation symbol **.**, we can create rules to describe entire sentences such as "A man drove the car.":

$$\begin{array}{lcll} S & \leftarrow & NP & VP \quad \text{"."} \\ NP & \leftarrow & Det & N \\ VP & \leftarrow & V & NP \end{array}$$

This collection of rules is known as a grammar. By repeatedly applying rules of the grammar to the input, and to the results of other rules, we can find whether the input is described by the grammar, and if so what structure can be assigned to it.

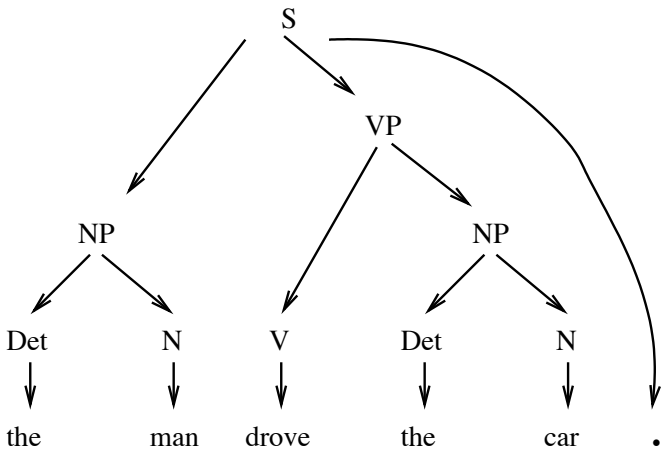


Figure 2.1: Example context-free parse tree

For example, if we applied these rules to the input "A man drove the car.", we would discover a structure such as shown in Figure 2.1.

This repeated application of rules in order to discover the structure of an input is known as *parsing*.

2.1.4 Unification Parsing

Although much of the pioneering work in Computational Linguistics centered on the parsing of Context-Free Languages (*e.g.* [42, 163, 34]), it has also long been recognized that the Context-Free formalism is insufficient for adequately describing the grammar of human languages [25].

Syntactic agreement

Syntax is more than just nouns and verbs. Although the mathematical formalization of grammar is a modern notion, scholarly descriptions of languages have long included features such as number and gender.

According to McArthur [86], Dionysius Thrax, of Alexandria, presented *He grammátike tékhne* circa 100 B.C.E. In this work, Thrax presented a grammar for Greek, containing lists of inflections as well as some elements of syntax (according to McArthur, "through the presentation of imprecise examples" [86]).

In that spirit, we will begin our discussion of features using an imprecise example. We have said that a noun phrase can be made from a determiner and a noun:

$$NP \quad \leftarrow \quad Det \quad N$$

Now consider the two sentences:

```
 Argyle saw a man.
*Argyle saw a men.
```

In English, there are several types of agreement: gender, number, and time (tense) for example. In the above example, the first sentence is considered grammatical, but the second is not, because the determiner *a* is of singular number, whereas the noun *men* is plural.

Strictly CF not preferred for writing grammars In order to maintain a context free grammar, we could give more precise categories to the words, and duplicate the above rule using this finer description [3, p. 83]:

$$\begin{array}{lcll} NP & \leftarrow & Det_{sing} & N_{sing} \\ NP & \leftarrow & Det_{plural} & N_{plural} \end{array}$$

It should be fairly clear that this is not such a good solution in the general case. We would have to duplicate large portions of our grammar for *each* feature we wished to account for, leading to an exponential blow-up in the number of rules in the grammar. From a software engineering outlook, the grammar itself is a piece of a system that must be maintained. So the original source grammar, as maintained by a person, must not only have good specificity and coverage, but it must also be transparent — the underlying intent of the grammar writer must be clear. So we would really like to write only one rule, but say that the number feature of the *Det* must match the number feature of the *N*.

This is a fairly small example, yet when taken to the extreme of a grammar designed to parse a realistic fraction of a language, the number of features can get inconveniently large. For example, in their description of a Revised GPSG grammar for English, Barton, Berwick, and Ristad point out that for verb agreement, they use a total of 19 3-valued ($+$, $-$, *unspecified*) features, which would create out to over 10^{10} rules if expanded into separate context-free rules as above, and that their full binary coordination schema corresponds to over 10^{64} context-free rules [10, pp. 293–4].

Semantic disambiguation Features are not only useful for enforcing agreement, they can also be used for some types of disambiguation. Consider the two ambiguous sentences:

```
Argyle saw the man with the telescope.
Argyle saw the man with the bagpipe.
```

The first sentence clearly has two ambiguous readings: either *Argyle* was using *the telescope* to help him observe, or *the man* had *the telescope*. The second sentence does not have an analogous first reading for broadly semantic reasons: one cannot use a bagpipe to help one to see better.

We could simply allow ambiguities such as this to pass through the parser intact. This is really a cheat in the context of many larger systems, however, since we would have to deal with the ambiguity eventually, and it very well might be more difficult to resolve later.

Alternately, we could attach some very simple semantic information to the words, using this to express the notion that a *telescope* can be used to help one *see*, but a *bagpipe* does not have this aspect. But in order to correctly reject the erroneous sense of using the *bagpipe* to *see*, we would have to apply some operation to this extra information in the grammar rules.

Feature-based transform useful for some more powerful systems

There is another fundamental problem with strictly context-free parses: they may not be especially useful.

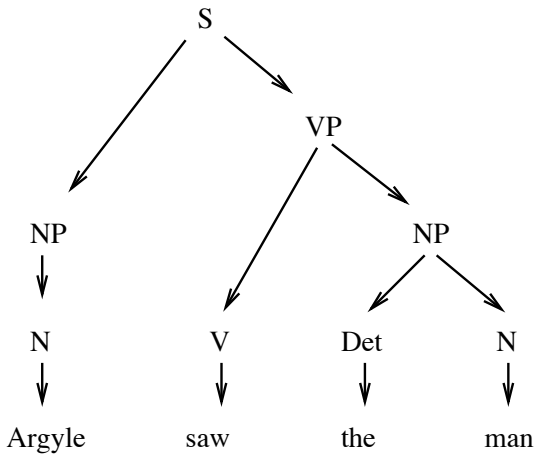


Figure 2.2: Example context-free parse tree

It is fine to say that we have something like the tree shown in Figure 2.2, but in many cases, it might be much more useful to know that a proper noun *Argyle* is the subject of the sentence, that the sentence is in the past tense, and that the object of the sentence is a noun, *man*, modified by a definite determiner.

It is true that we can figure all this out based on the parse tree that we discovered, but that begs the question: *how* is this information to be discovered? This is exactly the problem we are actually examining.

What we really want is a formalism which allows the grammar writer to express these aspects of syntax in a principled way. Unification has been found to be an effective solution to , either alone [39] or as an adjunct to a context-free grammar [56] has been found to be a good formalism

About Unification

Unification can be viewed as a simple generalization of pattern matching. Consider having lists in which some values could be variables, rather than strictly constants. For this scenario, Allen has a good definition of unification (when referring to nested lists of variables and values):

> Two lists are said to **unify** if there is a set of bindings for their variables such that, if you replace the variables with their bindings, the two lists are identical.
> [3, p. 599]

We can also relax the above definition somewhat to say that a **pattern** will unify with a list if there is some set of bindings of values to variables such that the pattern does not conflict with the list. Note that this is even more general than a subset operation. In this way of looking at unification, the lists are sets of name-value bindings, and the question could be rephrased as "Do these sets of bindings agree, or do they clash?"

So, for example, we might say that the name-value list $((AB))$ could unify with the pattern $((C = D))$, giving the result $((AB)(CD))$. But the list $((AB))$ could not unify with the pattern $((A = C))$.

We can generalize this somewhat more. In the above example, we have considered only a flat set of names and value bindings. I will refer to this as **simple-valued unification**.

One can also imagine generalizing the values that could be bound to names as more than simple constants, but instead as whole sets of subordinate name-value bindings. Thus we could have a name-value set such as:

```
((foo bar)
 (baz ((a b) (c d))))
```

For this, the pattern will need to be enhanced as well, in order to have a sufficiently precise way of referring to names and values. This is because a name is only bound with respect to its context

(with the top-level being generally unnamed). Thus rather than saying *name = value*, we will say *path = path*.

So, for example, the pattern `((a = c))` would unify with the name-value set:

```
((foo bar)
 (baz ((a b) (c d))))
```

Giving as a result:

```
((a c)
 (foo bar)
 (baz ((a b) (c d))))
```

But the pattern `((baz a) = c)` would not, since in the context for the bindings of `baz`, `a` has a value of `b`.

Unification of this sort can build large and complex trees, which we will see is both useful and troublesome. I will refer to this sort of unification as **tree-valued unification**.

Unification in Theories of Syntax

There are several overall view-points one can take when developing a theory of syntax. From the stand-point of the computational linguist, all view-points are not equally good. Quite often, the theoretical linguist will concentrate on generative theories. In these, the starting point is some abstract structure representing a thought, and the ending point is an ordered string of words or equivalent symbols.

The main problem with this viewpoint is that the computational linguist often finds him- or her-self working in the other direction: given an ordered set of words, what is the implied structure?

So called 'structuralist' theories of grammar can be of significant help in this. From the outlook of generation, structuralist theories assume that the words will be eventually produced in the final order only, rather than *e.g.* being created and then rearranged. Due to straight-forwardness of this outlook, the process can often be easily reversed, which is exactly what we need for parsing.

When working with a grammar, it is often convenient to write rules that have two parts: a rule in some strict grammar formalism, such as a context-free production, and a set of unification patterns. The values that these patterns will match against are the values that the immediately subordinate children of the rule might have. In this scenario, a node in the grammar tree will take a value that

is the result of applying the generating rule's patterns to the set of values of the nodes that are immediate children. Finally, to initialize the process, terminals such as words will have defined sets of values themselves. If the base rule succeeds, but the unification fails, then the overall rule will have failed as well.

Harrison and Ellison point out that the context-free production can be viewed as 'syntactic sugar' for certain types of unification equations [45]. Briefly, consider the rule:

$$S \quad \leftarrow \quad NP \quad VP$$

If there is a *category* variable in the unification system, and the unification system allows for the over-writing or replacement of unification values, then the effect of this context-free rule may be easily simulated by the three equations:

$$(x1\ \mathit{category}) = np$$
$$(x2\ \mathit{category}) = vp$$
$$(x0\ \mathit{category}) := s$$

(Where x0 refers to the parent constituent, x1 and x2 refer to the child constituents, and *:=* is an over-write operation.)

One can imagine using this sort of unification to build grammars more interesting than that of the simple grammar formalism that is the basis, or **spine** of the grammar. I will briefly look at two such formalisms: Generalized Phrase Structure Grammar, or GPSG [39], and Lexical Functional Grammar, or LFG [56].

GPSG - simple features only, with a multiple inheritance system.

GPSG[39] can be seen as one extreme in the continuum of unification-based structuralist theories. GPSG does not try to do anything more than allow valid phrases and sentences, and disallow invalid combinations. GPSG attempts, whenever possible, to accomplish this using only simple-valued features. In order to accomplish this, GPSG uses a system of multiple-inheritance to create generalizations of patterns in the unification system [137].

GPSG was originally designed as a general theory of syntax, rather than as "a programming language specialized for writing grammars" [23, p. 20]. As such, it tries to make predictions about the universe of all possible human syntaxes. One such prediction was that it should be possible to describe (almost all) languages using a strictly context-free formalism. Thus GPSG was designed

so that the rules of a grammar could be expanded into a strictly context-free set, albeit with an exponential blow-up in the number of rules. Barton, Berwick, and Ristad point out that no one would actually want to do such an expansion. In their description of a Revised GPSG grammar for English, they estimate that their binary coordination schema corresponds to over 10^{64} context-free rules, and conclude that "Any recognition procedure that explicitly calculates or uses the set of admissible local trees can only result in a slower recognition time than one that does not." [10, pp. 293–4]

GPSG is attractive for its simplicity. For example, in order to claim that for a well-formed sentence, the person (*e.g.* first, second, third) aspect of the subject NP must match that of the VP, one need only define a *PER* feature, make sure that it is set appropriately for the nouns and verbs, and insist that the *PER* feature match in the appropriate rule. Using a notation somewhat less obscure than GPSG, this rule would say:

$$\begin{aligned} S \quad \leftarrow \quad & NP \qquad VP \\ & \langle \textit{NP PER} \rangle = \langle \textit{VP PER} \rangle \end{aligned}$$

Although GPSG was designed as a theory of syntax, rather than a computational syntax formalism, it has inspired several computational formalisms, including HPSG [118] and the syntax formalism used in the Core Language System [4].

LFG - tree-based features

LFG[56] is another extreme. Unlike GPSG, LFG tries to associate phrases with their grammatical role in an overall clause, as a way of mapping from the more abstract space of grammatical functions into the less abstract string of words.

Like several other types of syntactic theories, LFG provides for a set of high-level descriptions, and a way to map these down to the usual low-level form: a sequence of words. In LFG, there are actually two closely related high-level descriptions. Constituent structures, or *C-structures*, are medium-level grammatical structures of the usual form, with nouns, noun phrases, verbs, verb phrases, etc. Figure 2.3 shows an example of such a structure.

These simple structures are augmented by decorating them with functional structures, or *F-structures*. These values are interpreted according to *grammatical functions* associated with the context-free rules that dictate the form of the C-structures.

For example, in the LFG rule below, one might say that a verb phrase can be built from a verb phrase and a noun phrase, with the subordinate verb phrase forming the phrasal head, and the

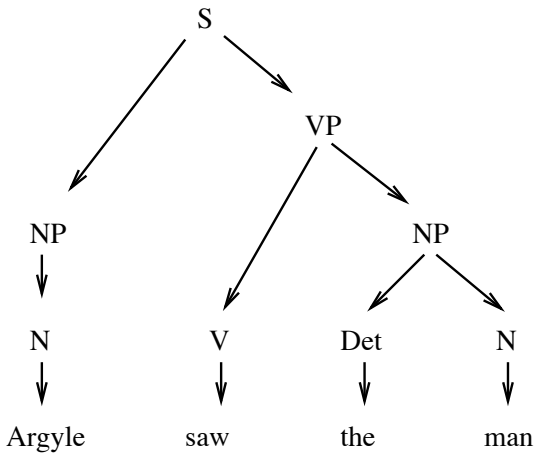


Figure 2.3: Example constituent structure

subordinate noun phrase filling the role of direct object, or *obj*. The latter requires that the head verb phrase can allow an *obj* slot to be filled, and that there is not already something in this slot.

$$\begin{array}{lcll} VP & \leftarrow & VP & NP \\ & & \uparrow{=}\downarrow & (\uparrow \mathit{obj}) = \downarrow \end{array}$$

These rules can be used to *generate* by repeatedly applied to an abstract grammatical role structure to create a well-formed string of words, or to *parse* by working in the reverse order.

LFG-style grammars have the advantage that they can be very convenient for the computational linguist who is interested in doing more that just parsing. The abstract representation can be just the thing for tasks such as machine translation [40, 4].

The down side is that a parser using an LFG-style grammar must build and maintain these complex tree structures. As a result, the unification process can be slow in some cases, and in the worst case the process could be computationally intractable [10, 137].

2.1.5 Unification grammars are computationally powerful

In [56, pp. 268–272], Kaplan and Bresnan show that LFG is more powerful than a simple context-free grammar by presenting two examples, one to accept all strings of the form $d^n b^n c^n$, the other to accept all strings of the form $\omega\omega$. This hints at the computational power of LFG; we shall see that it can be very strong indeed.

In order to ensure computability, Kaplan and Bresnan restrict their theory in various ways in order to ensure the computational tractability of their system. Notable in these restrictions is a limit

that no unproductive loops are allowed in the context-free backbone of the grammar. (For example, the set of rules $X \leftarrow Y$, $Y \leftarrow Z$, and $Z \leftarrow X$ would not be allowed [56, p. 264].) In order to see why this was done, let us consider a system much like LFG, but without these restrictions.

According to Shieber [137], PATR-II [135] is both simple and unrestricted. It can, however, contain tree-structured unification values, rather than a simple binary vector. Shieber notes that because of this, PATR-II "can characterize any recursively enumerable language" [137, p. 66], which is to say that it is Turing Complete [48]. (The proof for this is fairly straight-forward. One simply uses the nesting ability of the unification values to form two stacks and a current symbol, and implements the Turing Machine "move left" and "move right" actions by manipulating these unification values. This is built, however, using loops of rules that are "unproductive" according to the LFG restrictions [56, p. 264].)

Avoiding Turing equivalence is a worthy goal — it ensures that we *can* actually parse. On the other hand, this is not a sufficiently tight computational restriction for real-world problems. Not only would we like to be able to compute the parses, but ideally we would like to know that we can do so in polynomial time.

In their book on natural language and computational complexity, Barton, Berwick, and Ristad [10] examine a number of theories of grammar in great detail, with an eye toward the computational difficulty of each theory of grammar.

The most restricted grammar that they examine are what they call *Agreement Grammars*. An Agreement Grammar is a context-free grammar where each nonterminal consists of a set of feature-value pairs, where both features and their values are chosen from a set of predefined finite sets. Further, a production may only be applied if the left-hand-side nonterminal can share a set of feature-value pairs with each of the right-hand-side nonterminals. In other words, if the feature-value sets unify. As such, this grammar could be considered to be the simplest version of a unification grammar, and in fact the authors note that they designed the AG definition to be a completely stripped-down version of the core of GPSG [10, pp. 89–90].

Barton, Berwick, and Ristad use the simple AG definition to show that even such a simple task as agreement is NP-Complete [10, pp. 89–96] by showing that they can solve AG in NP time, by guessing a solution and verifying that it is correct in polynomial time, and that they can reduce 3SAT — the satisfiability problem restricted to conjunctive normal form, and with a limit of three terms per disjunctive clause, *e.g.* $(a \vee b \vee c) \wedge (\neg a \vee d \vee e)$ — a standard NP-Complete problem [132, pp. 633-639]. to the AG parsing problem, thus AG is also NP-Complete. The upshot of this is that the parsing problem with unification may be very, very hard. In the worst case, no (deterministic)

polynomial-time solution is known.

Kaplan and Bresnan point this out in [56] as well. They write: "The problem of deciding whether or not a given f-description is satisfiable is equivalent to the decision problem of the quantifier-free theory of equality. ... (Oppen & Nelson, 1977)" [56, p. 190].

Barton, Berwick, and Ristad confirm this by presenting a novel proof that LFG recognition is NP-hard [10, pp. 107–13]. This is not too surprising, since LFG can implement simple feature unification such as Barton, Berwick, and Ristad's AG formalism, which is NP-complete (see above in Section 2.1.5).

2.1.6 Parsing as Constraint Satisfaction

We can return to the two fundamental questions in parsing. We saw that the *recognition* question "Does grammar G parse input I?" is equivalent to asking "Is I a member of the language L(G) encoded by the grammar G?". This in turn is equivalent to asking "Are there a set of constraints P(G) consistent with I?". Likewise, the *parsing question* "*How* does G parse I?" can be seen to be equivalent to asking "What are a set of constraints P(G) that prove that I is a member of L(G)?"

This seemingly simple viewpoint is actually somewhat subtle. By taking this view, we can make several important observations:

- Parsing is a constraint satisfaction problem [85, 22, 96].
- Context-free and Unification constraints can be encoded in first-order logic (or preferably a subset of first-order logic) [22, 96].
- The constraints can be re-ordered according to the mathematics of FOL.
- The mathematics of the logic in use allow for both conjunctions (*AND*s) and disjunctions (*AND*s) to be *short-circuited*: If some clause within a conjunction is false, we know that the whole conjunction must be false without evaluating any remaining terms. Similarly, if any term within a disjunction is true, the whole disjunction is true (without evaluating any remaining terms).

From this in turn we can observe the following: within a family of grammars, all methods which speed up parsing are one of the following:

1. A method to calculate individual constraints.

2. An algorithm to calculate some set of constraints with special computational properties.

3. A way to re-arrange the constraints so that redundant calculation of constraints are avoided.

4. A way to re-arrange the constraints so that Inexpensive constraints are applied *prior to* the expensive constraints so that the failure of an inexpensive constraint allows the parser to avoid calculating the expensive constraints entirely.

5. The introduction of additional constraints, which are derived from, and less expensive than the original constraints, in order to avoid calculating expensive constraints as above.

We can also characterize most robust parsing techniques as a weakening of these constraints in certain ways. For example, the ability to 'skip' words could be viewed as a relaxation of the constraint that all the words must be used in consecutive order.

2.2 Parsing Applications

2.2.1 Machine Translation Systems

As introduced in Chapter 1 above, this parsing system is engineered to be a part of a larger system — in this case, the KANT system: a large-scale knowledge-based machine translation system. One concrete goal of this work is to produce a parser that is largely compatible with the existing one used in the KANT system, while being resistant to certain failure modes that the current parser suffers from. In order to understand the particular design decisions that have directed the development of this parser, we should review the context in which it is intended to operate.

Machine translation is the process of translating text in one human language into a different human language though an automatic process. There are several major successful approaches to attacking this problem; the KANT system uses a knowledge-based approach.

In a knowledge-based machine translation system, the system attempts to build an explicit higher-level representation of the source text (sentences or sets of sentences), which is then translated down to the target language. This is in contrast to example-based or statistical approaches, in which any abstraction away from the source text is a substantially implicit effect [49].

The KANT system

KANT architecture The KANT system [20, 89, 110, 111] attempts to take the knowledge-based approach to useful extreme: the source text from any one of a number of input languages is first translated into an *Interlingua* representation, which can then be translated to any one of a number of target languages [20, 89]. This is shown conceptually in Figure 2.4.

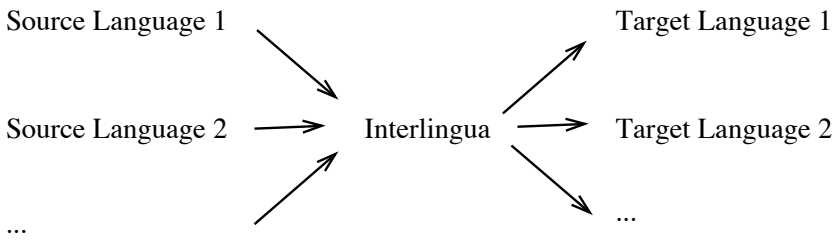


Figure 2.4: Multiple-source, multiple-target Interlingua Translation

The primary advantage to this sort of over-all structure is that m input languages and n output languages can be supported by a system with some $O(m+n)$ major components, whereas a system that requires each translation path to be implemented as a completely separate piece would require $O(m \cdot n)$ major components. Of course, maintaining compatibility amongst the m input modules requires extra work, but over-all this is a good way to structure such a system. [20]

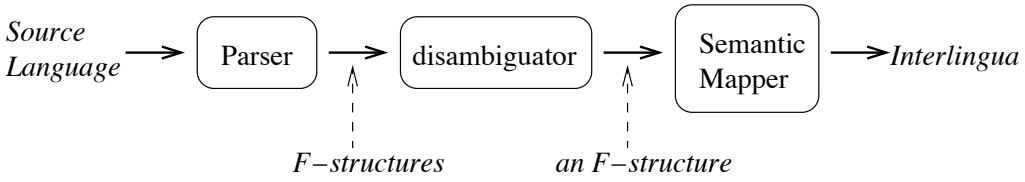


Figure 2.5: Block Structure of Source-to-Interlingua Conversion

The process of translating from a single source language to the Interlingua is broken down into a set of pieces, shown conceptually in Figure 2.5. In this figure, the parser translates the input string into a set of F-structures. The disambiguator then selects a best single F-structure for this parse. Finally, the semantic mapper translates the (still partially source-language dependent) F-structure into a (language independent) interlingua representation.

Ambiguity a problem for machine translation Ambiguity of any sort is a large problem for a machine translation system. The reason for this should be fairly clear: what is ambiguous in one language is rarely ambiguous in the same way in another. As a result, a single ambiguous input sentence may have several different correct translations into a given output language. As a

result, there is a strong desire to find a single "right" interpretation for any input, so that this single interpretation can be translated to a single "right" translation[1].

Obviously there are two general ways to attack the problem of ambiguity:

- Select from amongst a set of ambiguities.
- Avoid having ambiguity.

While the latter may sound trite at first, this is a perfectly acceptable approach for some situations. In particular, a limited-domain machine translation system narrows down the space of the source language to a manageable subset. Much of the point of limiting the domain is to exclude ambiguities that have nothing to do with the domain that one is translating from [62, 20]. For example, even though 'grease the wheels', meaning to bribe, is a common phrase in colloquial American English, it is unlikely to be used in this metaphoric sense in a manual on the maintenance of heavy equipment.

That said, limiting the domain of a machine translation system can help, but does not solve the ambiguity problem. Only some machine translations are used in a situation where the input domain and/or language *can* be limited. Still, limiting the domain when and where appropriate can substantially reduce ambiguity problems [8].

A class of ambiguities that causes trouble for a parser There are several classes of ambiguities that can cause trouble for a machine translation system.

- *Lexical ambiguity.* This ambiguity is caused by a single orthographic representation of a word having more than one meaning. For example, the word 'grease' could either mean the verb which means 'to lubricate', or it could mean the noun, a type of heavy, thick lubricant. This is an ambiguity in the grammatical category of the word.

  A more subtle form of lexical ambiguity is an ambiguity in the detailed meaning of a word. For instance, 'wheel' could refer to the part of a truck that rolls against the ground, or the control device used to steer the vehicle. While these are both nouns, these are in fact distinct and non-interchangeable parts. In some circumstances, it is important for a machine translation system to be able to distinguish between choices such as these.

[1]Obviously the problem of translating a single interpretation into a single "right" output is fraught with many of the same problems of disambiguation that picking a single "right" interpretation is.

- *Non-compositional phrase ambiguity* (idiomatic ambiguity). In this form of ambiguity, a phrase of words has a different meaning as a group than the ordinary composition of the meanings of the words would indicate. For example, in the Cantonese dialect of Chinese, to 'fry the squid' means to be fired from one's job [161].

  Idiomatic phrases such as these cause trouble both for students of a non-native language, as well as for machine translation systems, for much the same reasons: there are many of these idioms, and the clue to whether the speaker or author is using the idiomatic meaning or the compositional meaning is often only subtly indicated by the context.

  Because the idiomatic meaning of a phrase is often quite different than the composite meaning of those same words, an incorrect guess can lead to a rather spectacular (and often amusing) translation failure, either from a machine, or a human translator. A particularly famous example of this type of ambiguity is the (June 26, 1963) John F. Kennedy said "Ich bin ein Berliner." [52] meaning 'I am from/of Berlin.' It is widely believe that it also could be understood to mean 'I am a jelly doughnut,' though this is disputed [80].

- *Syntactic attachment ambiguity.* Attachment ambiguity is an effect of the grammar. This form of ambiguity comes from a choice between two or more different ways of analyzing some input. For example, Modern English prepositional phrases can be the source of syntactic ambiguity. The prepositional phrase is formed out of a preposition and a noun phrase, and can attach to both a verb phrase, as well as a noun phrase. This allows the prepositional phrase to attach to the noun phrase of another prepositional phrase, which in turn can lead to attachment ambiguity when two or more prepositional phrases appear together. An example would be the sentence:

  `Argyle saw the man on the hill with the telescope.` [47]

  This sentence has four distinct readings:

  1. Argyle could have been on the hill and used the telescope to see the man.
  2. The man could have been on the hill, but Argyle used the telescope to see him.
  3. Argyle could have been on the hill; the telescope was also on the same hill but Argyle didn't use it.
  4. The man could have been on the hill, along with the telescope.

Although ambiguities of any form are a problem for an entire machine translation system, individual localized ambiguities are not particularly a problem for the parser. Localized ambiguities can become a problem, however, because the *addition* of each local ambiguity can increase the number

of global interpretations *multiplicatively*. Thus the total number of global interpretations can be exponential in the number of local ambiguities.

It is easiest to see this happen with syntactic attachment ambiguity. We will see in Section 3.3.1 phenomena such as such as prepositional phrase attachment ambiguity lead to possibly very bad behavior in the process of parsing such as an exponential blow-up in the space and time requirements of the parser. These unfortunate behaviors are not limited to any particular type of ambiguity — any set of local ambiguities, each of which cause a multiplication of global interpretations, can lead to exponential blow-ups in the parser if they are not carefully controlled.

In Section 3.3.2, we will investigate this phenomenon, and examine how it can be fixed for a purely context-free parser. Then in Section 3.3.3 we will investigate how such a context-free parser can become broken once again by the addition of a unification system, and that (as of this writing) no guaranteed polynomial time solution to this problem is known to exist.

The remaining chapters of this work describe a set of techniques which, when used together, can allow the system to avoid this potential pit-fall for the sorts of parsing problems we have encountered in practice in a machine translation system.

Interactive checking One key element that allows the KANT system to work very well on restricted domains is an effective method of checking to make sure that the human-authored input language abides by the required restrictions [20, 89, 8]. This check must be effective, but must not be too time-consuming for the author [55].

The simplest way to implement this checking is to run the full parser on the authors' source document, and check to make sure that a valid result is produced. However, this is more work than the problem of conformance checking actually requires.

In the particular case of interactive conformance checking, we could simply run some minimal amount of the parser — not enough to actually produce results that are useful for the rest of the machine translation system, but simply *just* enough to check that a sentence either conforms or doesn't conform to the input language that is specified by the grammar.

Conceptually, we can extract out the first part of the pipeline shown above in Figure 2.5, and adapt it to the needs of interactive checking. This is shown in Figure 2.6.

Ideally, this checker should be integrated into an authoring tool, so that the writer can write naturally and easily, but still receive indications warning them when what they have written fails to conform to the restricted language that is required by the machine translation system.

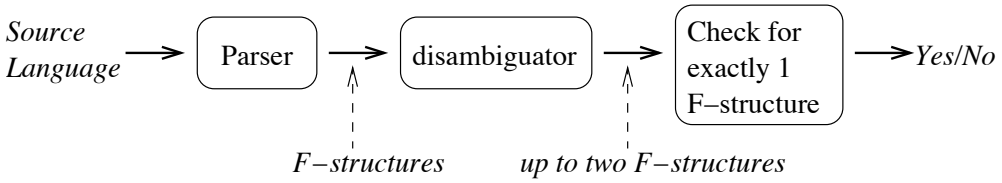


Figure 2.6: Block Structure of Source-Language Conformance Checker

One important guideline in the design of good human-computer interfaces is: *The system should* not *surprise the user* [5, 109, 7].

The source-checking component should produce clear indications warning the author about parts of their text that does not conform to the restricted language required by the machine translation system. On the other hand, the display of these indications should not force the author to deal with them as they are discovered to the exclusion of the normal task, for otherwise the author might loose track of the flow of the thoughts they are trying to express (which is, after all, their primary task).

So, phrasing the design in terms of current standard user interface design ideas, the lack-of-conformance warnings should not be modal — they should not force the user to deal with a single problem before being presented with the next problem [5, pp. I-28–I-29]. Baecker and Buxton note that Larry Tesler argues particularly strongly against the use of modes in user interfaces [7, p. 429].

In addition, the system should not surprise the user with checking that is overly variable in time either [46, pp. 29–32][51]. Ronald Baecker makes this very clear, in his 1980 paper on graphical user interfaces. In a list of guidelines entitled "some characteristics of good interactive techniques", he includes:

> 3. The feedback occurs predictably. Unpredictable response is even worse than predictably slow response, leading to frustration, tension, and anxiety.
> [6]

If the system normally checks text at a rate of several sentences per second, it should not "bog down" and spend the larger part of a minute checking some particularly bad sentence, and it should absolutely not do this without some indication to the user that it is still working on something [5, 46]. Unfortunate behavior of this sort in the KANT system several years ago [114] supplied the seed from which this work has sprouted. Our goal, then, is to have a system that is not only fairly quick at parsing sentences, but does not bog down or blow up on *any* input.

This design principle can be seen to be a special case of generally good engineering practice. Butler Lampson gives the following hint in "Hints for Computer System Design":

In allocating resources, strive to avoid disaster rather than to attain an optimum. [67], [13, p. 67])

Chapter 3

Unification Parsing

> **DEAR GRAMMAR LADY:** I don't know about you, but in my student days, mastering English grammar seemed about as easy as learning the art of black magic. Well, I was apparently onto something. The word "grammar" entered medieval English as "grammarye," via Scotland. The Scots got it from the French word, "grimoire," which meant a collection of magic spells. The connection was made between grammar and magic because most people then were illiterate, so any linguistic smarty-pants were metaphorically seen as dabbling in sorcery. (Source: "Forgotten English.") I wish they had retained grimoire. It sounds more like the way I felt about the subject.
>
> **COMMENT:** I'm sure a lot of people feel the same. Thanks for the information.
>
> *— The Grammar Lady, The Pittsburgh Post-Gazette, Friday, November 3, 2000, p. C-11*

3.1 Unification Parsing

As we noted in Chapter 2, one general framework that has gained acceptance with theoretical linguists involves adding additional information to each of the context-free grammar rules which describe the valid parse structures which can be represented. [137, 39, 56]. Most of these formalisms fall under the general category of unification grammars [137].

One such a formalism is Kaplan and Bresnan's Lexical Functional Grammar [56] discussed in Section 2.1.4 above. One of the nice features of LFG is that it tries to associate structures such as phrases with the grammatical roles of the structures. So, for example, the value associated with the top-level (sentential) node of a declarative transitive sentence would include not only information inherited from the head verb, but also explicit slots for the subject and object information.

Marking out the grammatical roles in such a manner gives us a nice, somewhat abstract point

from which to generate text, or to which we can parse text. For example, if we are translating from a language such as English, which marks roles such as the subject and object by their position in a clause, into a language such as Spanish, which marks these roles morphologically, it is convenient to know that the *subject* of the input language is a female, third person, singular pronoun. And even though we might wish to perform the translation by transforming this abstract structure into an even more abstract one, it can be easier to make the representation of the input more abstract in a set of steps, rather than all at once [40]. So we might like to implement something like LFG in our system.

To implement this on a computer, we need a somewhat simpler syntax than using all those arrows. What we shall do is number the children on the right-hand-side of each rule, left to right. The leftmost child we shall call `x1`, the next, `x2`, and so forth. We will call the value of the current production `x0`. Thus the above rule can be written as:

```
(VP <== (VP NP)
    ((x0 = x1)
     ((x0 obj) = x2)))
```

If one considers unification to be the process which decides what information to associate with each node of the (context-free) parse tree, and that the parse operates in a bottom-up fashion, then the set of unification rules can be considered a function which will either produce the new value to be stored at the parent node (`x0`), or some sort of indication that the unification failed.

This is the representation our system uses internally. It was described in detail in Goodman and Nirenburg [40].

3.1.1 Pure Unification Parsing

In some systems, the feature-theoretic portion has been carefully integrated with the parsing process, to the point that the chart algorithm was modified to support a pure unification system [136, 45]. These systems concentrated on correct execution, or at least correct execution in some cases; issues of efficiency were not considered as such.

Shieber's unification-chart with restriction

In [136], Shieber examined pure unification parsing – *i.e.* parsing using unification equations *without* a context free spine. Taking this approach to the parsing problem is motivated by theories of grammar such as GPSG[39], LFG[56], and PATR-II[135] (also see below). GPSG, for example, treats

the traditional category of a rule as the union of a set of features. Other formalisms such as LFG have values that are more complex; some such as PATR-II [135] have essentially no restrictions.

Shieber addressed this problem by adapting the chart parsing algorithm to parsing in a strictly unification environment such as PATR-II. His motivation in this was, in part, that gluing a unification formalism to a context-free spine violated the spirit of unification formalisms (in the abstract), and could cause the parser to ignore information that could help guide the parse. On the other hand, simply allowing the category symbols of the grammar rules to come from an infinite domain, such as the domain of tree-valued unification values, may cause parsing algorithms such as left-corner [23] and LR preprocessing [48, 151] to fail [23, 136].

So rather than hard-wiring a dependence on the context-free spine into the parser, Shieber made the parser depend on an abstract subset of the space of unification values, which he called the *restrictor*. This could be set on a per-grammar basis in order to better tune the parser for a particular unification grammar.

Consider the PATR-II rule:

$$\begin{aligned} S \quad \rightarrow \quad & NP \qquad VP : \\ & < NP\ agreement > = < VP\ agreement > \end{aligned}$$

Now suppose there is a *cat* slot. This rule can be re-written as:

$$\begin{aligned} X_0 \quad \rightarrow \quad & X_1 \qquad X_2 : \\ & < X_0\ cat > = S \\ & < X_1\ cat > = NP \\ & < X_2\ cat > = VP \\ & < X_1\ agreement > = < X_2\ agreement > \end{aligned}$$

There is, however, no particular reason to "hardwire" only the *cat* slot into the parser by using it alone as the category of a context-free parser. GPSG uses the entire unification value (the whole set) as the category.

Shieber noted that one need not wire in a particular feature, nor need one use the entire unification value set, but instead one could define an abstract subset of the slots as the "interesting" ones, the *restrictor*, and define this subset separately for each grammar.

So if we were to set the restrictor to $< cat >$, we would get exactly the same result as binding our grammar to the context-free spine. But we could also for example set the restrictor to the set $< cat, agreement >$.

Obviously in theory, while a unification grammar can be bound to a context-free spine, it doesn't have to. Shieber showed that this could be made true in practice as well.

3.1.2 Unification Parsing on Context Free Spine

Unfortunately, there has been somewhat of a disconnect between the theoretical linguists and the computational linguists. In many systems, feature-theoretic portions have been added into a context free parser essentially as an after-thought [3], with little consideration of the ramifications of the additions.

GLR (Tomita)

In his thesis, Tomita doesn't address a feature-theoretic (or unification) system for his parser at all. His thesis work was strictly on the development of an algorithm for Context-Free parsing.

He does touch on the issue in a later paper [152]. The development of the GLR system saw the addition of a simple *pseudo*-unification framework [153] (see also Section 3.2 below). In [152], Tomita notes that local ambiguity packing for augmented context-free (*i.e.* unification-valued) grammars is not as easy as local ambiguity packing for purely context-free ones. He goes on to say that the solution he took was to have the GLR parser "unpack" packed nodes whenever the unification values needed to be checked. He then argues that this unpacking is not such a big deal in practice; that most of the nodes resulting in packed ambiguities will not need to be unpacked in practice.

Carroll's chart parser

Carroll looked at the problem of parsing with unification, in the environment of a large, high-coverage grammar. To this end, he developed a new unification algorithm, better suited to the grammar formalism he was using. He then investigated both bottom-up chart parsing with ambiguity packing, and non-deterministic LR parsing as an improvement over Tomita's GLR. Finally, he investigated probabilistic parsing in this environment.

In Carroll's system, the 'category' of a production is the unification value. He shows how to use the feature system to automatically build a context-free "spine" of rules from the unification equations; the generated context free rule will be subsumed by the unification equation, but can be checked quickly and used to build the LR tables used by his nondeterministic LR parser.

The grammar Carroll used was closely modeled on the GPSG [39] theory of grammar. The

emphasis of GPSG is on simplicity: with only one exception, GPSG unification values are a set of simple-valued features. And the exception is also a set of simple-valued features. This differs from the complex tree-based values of *e.g.* LFG [56].

These simple values promote local ambiguity packing. Carroll writes that "the correct test for packing is category subsumption." [23, p. 57],[4, p. 142]. This decision is closely coupled to the simple structure of Carroll's unification grammar. We will see that while good for Carroll's system, it will not suffice for ours.

Alshawi, *et al.*'s system: the Core Language Engine

The Core Language Engine [4] (CLE) is a remarkably "clean" system for processing language. The processing of language is separated into a set of distinct stages, each substantially separate from another [4, pp. 1–9]. It uses unification-valued categories for both the syntactic and semantic components. Unlike some other systems (including the one presented in this work), the syntactic values are quite different from, and independent of, the semantic values. Syntactic values are almost entirely simple-valued, and compiled down to binary features. The exception to this are the list-like features used to implement *gap-threading* method of handling elision and other "transformation-like" phenomena.

The semantic values are Quasi-Logical Forms, which are modeled somewhat on LFG [4, p. 43], but also resemble the Discourse Representation Structures of Kamp and Reyle [54]. The building and processing of these structures is done strictly after all syntactic processing has finished. As a result, the semantics can eliminate some possible interpretations, but cannot directly influence the syntax.

As a result of the simplicity and clear separation of the syntax and semantics, Robert Moore and Hiyan Alshawi could devise a clever solution to some of the problems of parsing with unification. In their system, the structure of the syntactic unification system, and the structure of the parser itself are quite closely related. The parser is a left-corner parser (see above), operating on the unification categories of the syntax. It employs packing on the categories. Because the semantic processing is delayed until after the syntactic parse, this packing accomplishes semantic as well as syntactic packing.

One way the authors demonstrated the utility of this system was to create a Swedish to English machine translation system. This MT system could be said to be an interlingua-based translator, using Quasi-Logical Forms as the interlingua.

Such a carefully thought-out feature theoretic system, with its associated parser, can avoid many

of the pit-falls of ambiguity [4, pp. 142–143]. Unfortunately, this system is not general in the sense that one cannot just lift the parsing module and use it for an entirely different theory of syntax and semantics. The parsing methodology is strongly linked into the overall system architecture, and purpose-built for the particular theories of morphology, syntax, and semantics used in this system.

The forced separation of syntactic constraint and semantic interpretation of the Core Language Engine could be considered to be a desirable goal from a systems design stand-point: the separation can prevent needless complications in the programming of rules of grammar that have to do with the mixing of two possibly distinct and different effects.

On the other hand, this separation can cause trouble for the grammar maintainer, because the joint effects and interactions of two separate sets of rules must be considered while debugging the system. If an input which is believed to be valid does not produce the intended result, the incorrect operation may be due to a failure of the syntactic system, or a failure in the semantic interpretation, or a possibly subtle interaction between the two subsystems. We will examine this problem in detail in Section 3.3.4.

As a matter of the philosophy of language, it is not in fact clear that it is in fact better to separate "syntax" and "semantics", nor is it clear that the separation happens along the lines that the Core Language Engine uses. The under-pinnings of Lexical-Functional Grammar, on which the KANT system is largely based [89, 110, 111], argue counter to this explicit separation. So while the Core Language Engine is based on a reasonable theory of language, it is based on but one of many competing theories of language, none of which have been shown to be so overwhelmingly advantageous as to become universally favored. In short, considerations of theories do not provide any clear reason for us to discard the linguistic-theoretical basis of our system and adopt a different one.

Furthermore, discarding the grammars that we have at the present time would imply not only reengineering several grammars, but also reengineering the F-structure to Interlingua translation stage [40, 89, 87], which might amount to *many* person-years of effort. While the prospects of completely re-engineering such a system may be academically acceptable to some, the possible gains may not be sufficiently compelling to be worth the cost of the re-engineering effort. Since part of the academic interest in *e.g.* KANT *is* that it is a large, practical, working system, we must look for a better answer than a complete re-write of large portions of it. So due to these entirely practical reasons, we cannot just adopt (or re-engineer) the Core Language System.

However, there are several interesting ideas presented by Alshawi, *et al.* ; these ideas will substantially influence the design of our new parser.

Maxwell and Kaplan

In the unification systems we are interested in, the conditions for creating a valid constituent r are two-fold: not only must the children have the correct categories, but the F-structures of the children must unify according to the equations that are also part of r. It is somewhat convenient to implement a parser by having it check for the unification condition immediately after discovering the valid (Context-Free) constituent, and only creating the constituent when the associated unification succeeds. We will refer to this as *interleaved* unification.

Maxwell and Kaplan [85] suggest that there is no particular reason why we must evaluate these equations in an interleaved fashion — one could evaluate the context-free part of the grammar, and only when it is complete go back and evaluate the unifications for the portions of the parse that are part of some complete analysis of the input.

Their approach is quite interesting. They argue that a potential problem with interleaved unification is that the parser might spend an exponential amount of time working on distinct unification results. (We will discuss why this is in detail in section 3.3.4.) By delaying the unification until later, they are able to avoid much of this problem.

Rather than simply evaluating the constraints within the system of the parser, they collect the entire set of constraints, optimize their order to eliminate redundancy, and then hand the complete set to a dedicated constraint solver, which produces a final F-structure (if any can be found). This extraction and joint optimization of constraints has the potential to greatly reduce the amount of work that is put into evaluating unifications, since the only unification constraints that are considered are known to be part of a potential complete solution.

In general, the results and conclusions of any study are dependent on the overall conditions under which that study was developed and the tests run. In some cases it is possible to prove generality, in others the strong possibility of generality can be demonstrated empirically, and in some, no claims of generality are made. Good software practice dictates that one not blindly adopt techniques advocated by others, regardless of the relative strengths and weaknesses of their studies or any claims of generality, but rather experimentally investigate possibly good ideas in one's own system to see if they are appropriate.

There are three fundamental problems with Maxwell and Kaplan's study. First, their study suffers from having an empirical evaluation on far too small a test set — 20 test sentences [85, p. 586] — to be considered by any means definitive, even for the implicit conditions of their own system. Thus, while their idea is an interesting one, the researcher should be doubly careful about examining the viability of this technique in the context of his or her own system.

Second, as Maxwell and Kaplan point out [85, pp. 575–6], there are possible advantages to being able to prune out derivations that succeed for the context-free rules but fail to unify, before they are used to create other useless derivations. By delaying this check until after the context-free parse is complete, their approach could lead to the production of a quite 'bushy' covering forest, containing considerable ambiguity that would not have been created had we used interleaved unification.

And finally, when faced with several different possible solutions, the constraint solver of their system seems to pick a single satisfying F-structure solution, without regard to syntactic structure or any other common disambiguating technique. In other words, it appears to pick *a* parse, rather than trying to pick *the best* parse. Any syntactic disambiguation would have to be somehow coded into the constraints (with the effect of weakening the purity of the syntax), and other disambiguation techniques such as statistical disambiguation would require modification of the constraint solver itself.

While we cannot blindly adopt their technique, we cannot simply dismiss it either. Rather, we will study it empirically, and develop a compromise that combines the best features of interleaved and delayed unification.

LCFlex (Rosé and Lavie)

LCFlex is a new parsing system proposed by Rosé and Lavie [125], which addresses several deficiencies in GLR* [68], as well as providing additional capabilities. LCFlex is designed to be a high-performance robust parser for unification formalisms similar to LFG [56]. It is based on a left-corner chart parser [34, 58, 3, 107], but allows several degrees of robustness.

Like GLR*, the primary method of robustness that LCFlex applies is the ability to skip over input words that the parser cannot fit into an over-all parse.

More interesting for this work, however, is a modification to the unification system of LCFlex that allows the parser to be flexible in matching certain unification features. This is best illustrated in an example from [125]: Suppose that the grammar in use had a rule saying that a sentence can be made from a noun phrase and a verb phrase, but only if these two have the same agreement feature:

```
(S <== (NP VP)
    (((x1 agr) = (x2 agr))
     ...))
```

LCFlex allows for a set of features to be specified as *flexible*. The LCFlex parser will try to

evaluate all equations that involve flexible features, but it will not require these particular equations to succeed.

Rosé and Lavie show, as an example, that if we were to mark the `agr` feature as flexible then we could successfully parse the following sentence even though its subject and verb agreement features would not match:

> `robust parsing present interesting computational challenges`
> [125]

This feature flexibility is implemented through the use of a wild-card feature that is guaranteed to match any other value. In this work, we will make extensive use of this wild-carding idea, though with the primary goal of faster parsing, rather than more robust parsing.

An additional interesting feature of LCFlex is that it is *substantially* faster than GLR* for robust parsing. According to Rosé, this is in large part due to the simplicity of modifying the chart algorithm to deal with the extra work of robust parsing [124].

3.2 About Pseudo-Unification

Pseudo-Unification is a variant of Unification which was originally proposed by Tomita and Knight [153] for use in a bottom-up parsing environment. If one considers the general unification equation:

$$path1 = path2 \tag{3.1}$$

In normal (full) unification, evaluating this equation will either have the side-effect of causing both *path1* and *path2* to have the same value, or to *fail* when (prior to evaluation) *path1* and *path2* already have separate (incompatible) values.

In contrast, in a Pseudo-Unification system, evaluating equation 3.1 will result in a modification to *path1*, but *no change in path2* (even if Full Unification would have caused this value to change).

Further, when a complete block of equations are associated with a rule production, we consider this block of equations to specify how to calculate the unification value associated with the parent node, based on the unification values of the child nodes, *without modifying the values associated with the child nodes*.

This contrast between Pseudo-Unification and Full Unification is most clear in the handling of references to structures, or whole sets of values. An example of such a reference is the LFG equation $\uparrow obj) = \downarrow$ typically associated with binding an argument NP to a transitive verb. In a full unification system, evaluating such an equation is typically accomplished by setting *obj* to be a reference (or in the terminology of computer languages, a pointer) to the sub-tree structure corresponding to whatever the $\downarrow$ is associated with.

More generally, in a full unification system, equations such as these specify (two-way) links in a graph of associations. Once a link is made, any other operation which effects the value of one side of the such a link will effect the other side of the link as well. Ensuring that all of the previously specified associations in such a structure are compatible with a new modification involves substantial amounts of effort [153, 63].

In contrast, a Pseudo-Unification system will set the *obj* slot to a copy of the unification value associated with the $\downarrow$ child. (In more sophisticated implementations of Pseudo-Unification, including ours, a copy-on-write mechanism [162, 153] is implemented so that the assignment is initially simply a pointer to the original value, but appropriate parts of the structure are duplicated should there be a change to the structure in the future. See Tomita & Knight's technical report [153] for a good description of this technique.)

In terms of computer language theory, we can consider Pseudo-Unification to be a *call by value* system [2, sec. 2.11, pp. 59–63], [99, sec. 5.5, pp. 116–119]. In Section 3.3.7, we will see that the call-by-value nature of pseudo-unification is both its greatest strength as well as its greatest weakness.

3.3 Parsing and Ambiguity

We have examined context-free parsing, and hinted at combining context-free parsing with unification. Before we can just do this, we must examine unification in some detail.

3.3.1 Ambiguity is a problem for context-free parsing

Maxwell and Kaplan [85] note that it is well known that syntactic ambiguity can lead to an exponential blow-ups in the number of distinct parses of an input. Although these are reasonably well known, for the sake of clarity we will review these arguments here.

Recall our in our definitions above that *recognition* is asking if an input is within some language,

whereas *parsing* is the process of finding a set of derivations that match that input. In the above descriptions, it has been assumed that parsing is about as difficult as recognition: naïvely, one could just assume that a parser simply works like a recognizer, but keeps track of which productions were used at each stage of the processing.

Unfortunately, it isn't that simple. Grammatical ambiguity can cause a problem here. If the parser keeps track of all possible parses, rather than just one parse, then when presented with an ambiguous input the parser could use space exponential in the input size to keep track of all of the possible parses.

If, on the other hand, the parser keeps track of only a single parse, then this parser will no be suitable for unification parsing (below) because the parser may "throw out" the only legal parse by retaining a parse that is syntactically good but will no unify due to *e.g.* agreement problems.

A simple ambiguous Context Free grammar

Claim: A context-free language can exhibit a number of derivations exponential in the size of the input.

Consider the context free grammar:

$$\begin{array}{lcll} S & \leftarrow & X & \\ X & \leftarrow & a & \\ X & \leftarrow & X & X \end{array}$$

For this grammar, we can show the derivations concisely with parenthesis. For the input "a a", there is obviously only one derivation:

```
(a a)
```

However, for "a a a", we can attach the third "a" on either side:

```
(a (a a))
((a a) a)
```

And for "a a a a", not only can the fourth "a" be attached to either end of any of the results for "a a a" above, but we can also regroup the whole set:

```
a a a a
```

```
(a (a (a a)))
(a ((a a) a))

((a (a a)) a)
(((a a) a) a)

((a a) (a a))
```

Claim: There exist context-free grammars, and strings described by these grammars for which the number of legal derivations of the string is exponentially proportional to the length of the string.

Consider the above grammar. By induction on the length of the output. For the base case, consider that the output "a" has exactly one derivation: $S \leftarrow X, X \leftarrow a$.

Now for an output formed of n "a's" has at least twice as many derivations as one formed of n-1 "a's": one set by joining the new "a" at the beginning of any from the previous set, and one by joining the new "a" at the end.

So there are at least $2(n-1)$ derivations for a string of n "a's". (Actually, the number of derivations in the above example is exactly equal to the number of ways of parenthesizing the string of N letters. The number of ways to do this is exactly Catalan$(n) = \frac{1}{n}\binom{2n-2}{n-1}$, which is in O(exp(n)) [158, pp. 116–118].)

A slightly more realistic example A problem with the previous example is bit contrived, and at first glance is unlike grammars that we might wish to write to describe natural language.

Billot and Lang [15] present an example derived from ordinary English which shows this same behavior as the above example. This example is also used by Moore and Alshawi [4], as well as Oepen and Carroll [115].

Consider the following grammar:

$$
\begin{array}{lcll}
S & \leftarrow & NP & VP \\
NP & \leftarrow & n & \\
NP & \leftarrow & NP & PP \\
VP & \leftarrow & v & \\
VP & \leftarrow & VP & PP \\
PP & \leftarrow & p & NP
\end{array}
$$

(where n, p, and v are input symbols)

Also consider an output of the form:

```
n v p n p n p n p n p n
```

Moore and Alshawi's example of such a pattern is the sentence:

$$\textit{Kim saw a cat } (\textit{in the hotel}^n).$$

Given n repetitions of the sequence "p n", an ordinary parser, without any type of packing, will have a number of derivations equal to Catalan(n), which is in $O(\exp(n))$.

The point of this is that in any context free parsing algorithm, in order to avoid completely blowing up and using enormous amounts of time and space when processing some inputs with certain grammars, the algorithm must either throw away some ambiguous derivations or store the set of possible derivations in a more compact form.

3.3.2 Context free parsing with packing

Clearly the parser must do something more clever than to store each distinct derivation independently. Younger [163] explicitly speaks to the problem of ambiguity, and showed how his solution addresses this problem. Tomita [151] found a good explicit solution to this problem, which More & Alshawi [4] and Carroll [23] then showed was more widely applicable than to only GLR.

Younger Younger, in his fundamental paper on context-free parsing, directly confronts the problem of ambiguity and the possible consequences it can have:

> A *parse* of a sentence in a language $L(G)$ is a description of how the sentence is generated by G; this description is generally in the form of a generation tree. There is for an ambiguous sentence more than one generation tree. In fact, there exist grammars G for which some sentences n the language generated by G have associated with them a number of distinct generation trees which is an exponential function of the sentence length.

He continues:

> Even if it were feasible to exhibit all the generation trees of a highly ambiguous sentence, that set of trees would not convey a great deal of information to the analyst. What is needed is a more compact form which summarizes the grammatical structure.
> [163, p. 199]

Younger then defines a "parsing matrix" P, shows that this matrix can contain a description of all possible context-free parses of a sentence, exhibits an algorithm to derive P from the "recognition matrix" R, and finally argues that this procedure (like the procedure to find R) runs in $O(N^3)$ time.

P is nothing more than an n-by-n-by-q matrix of bits (n is the length of the input sentence; q is the number of terminal + non-terminal symbols), where $P(j, i, c)$ is true whenever there exists a constituent c spanning from word j to word $j + i$.

Clearly, whenever an ambiguous analysis is discovered, the appropriate bits for all of the daughter analyses should (and indeed are) set. Thus multiple ambiguous parses are implicitly stored in this structure.

GLR (Tomita) Tomita, for his parser, proposed the idea of using a *packed forest* to explicitly keep track of ambiguities. The general idea is to keep ambiguities local, by grouping ambiguous alternatives together as a set, and then referring to the set of them as a single unit when using them in further, superior derivations. This allows the GLR parser to deal with local ambiguity locally, rather than having to store N completely separate trees.

Billot and Lang Billot and Lang [15] investigate the problem of ambiguity in Context-free parsing, and show that for every input sentence of every context-free grammar, there exists an output graph representing the union of all parses for that sentence of size $O(n^3)$. Billot and Lang structure their output graph somewhat differently than usual packed forest; the only difference is that they allow partial sharing between sets of children of a production. They point out that this difference in structure is critical to achieving the $O(n^3)$ size bound for any C-F grammar (in contrast to Chomsky Normal Form grammars). They also include a description of how to transform their output graph into a more standard packed forest.

Carroll's chart parser Carroll, in his thesis [23], showed that the packed forest representation works to address the ambiguity problem for chart parsers in practice, just as it had for GLR. (This is not surprising in light of Billot and Lang's work [15].)

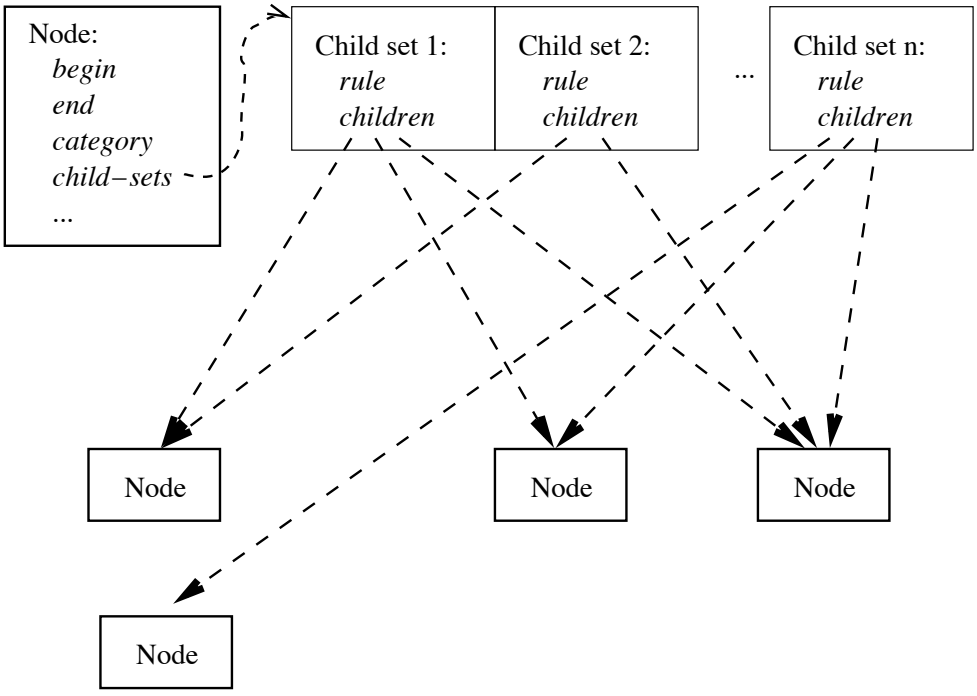


Figure 3.1: An example shared packed node.

A packing example Figure 3.1 shows an example packed forest node, corresponding to a packed set of inactive arcs. Each inactive arc is a representation of the rule that produced the arc, the set of children of the arc, and the category of the parent. In order to pack inactive arcs together, each packed forest node contains a *set* of sets of children; one set per inactive arc.

Given this structure, we can demonstrate packing in a parser with an example:

```
1       2     3     4     5    6     7      8      9    10         11
 Argyle   saw   the   boy   on   the   hill   with   the   telescope
  n       n,v   det    n    p    det    n      p     det     n
```

This sentence has five possible parses, which we can represent schematically as:

```
[1] Argyle (((saw the boy) on the hill) with the telescope)
[2] Argyle (saw (the boy on the hill)) with the telescope
[3] Argyle saw ((the boy on the hill) with the telescope)
[4] Argyle saw (the boy (on the hill with the telescope))
[5] Argyle ((saw the boy) (on the hill with the telescope))
```

Without packing, we could get a chart like Figure 3.2. In this figure, we can see many entries

with the same symbols and covering the same span of words.

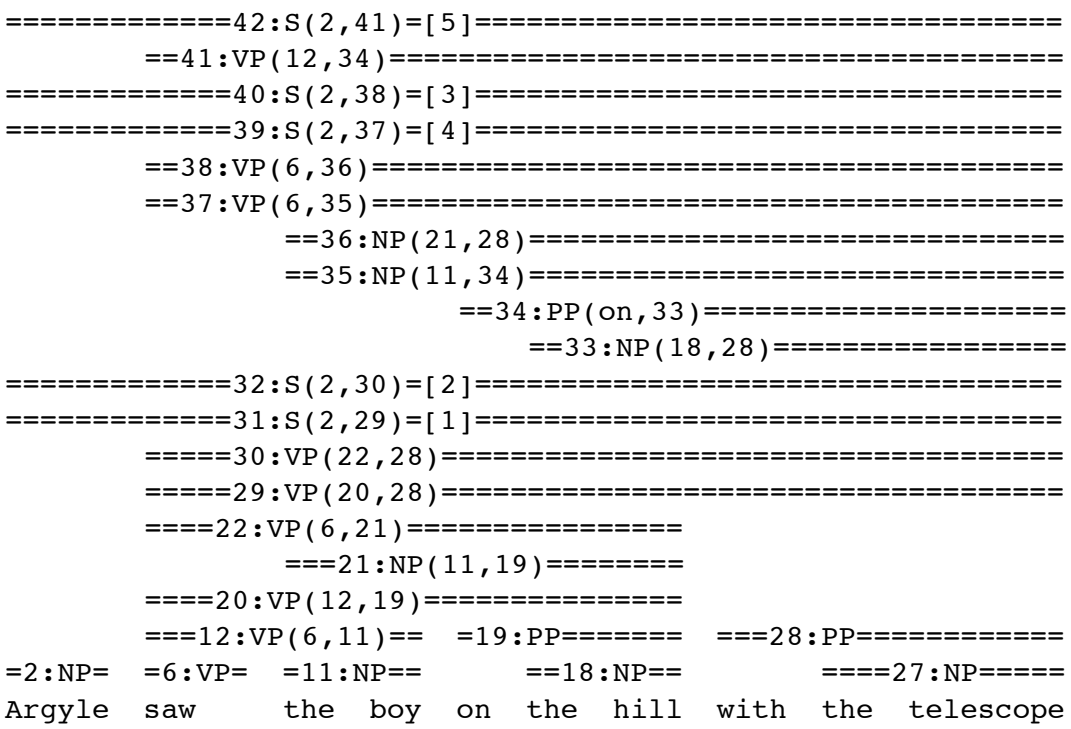


Figure 3.2: Example chart without packing. (Figure after Allen [3, Fig. 6.17; p. 178].)

Without packing, we get some sets of ambiguous sub-parses. In this example, because nodes 29 and 30 have the same span, we are forced to also generate nodes 31 and 32.

With packing, the chart will be smaller, as shown in Figure 3.3. We have avoided producing ambiguous sets of sub-parses such as nodes 29 and 30 (hence also avoiding the duplicates 31 and 32). Although we have saved only 5 entries in this example, we have also avoided propagating ambiguities outward, and hence saved some 10% of the required structure. We will see that in longer cases, the time and space saved can be substantial.

Worst-case Theoretic Bounds on the size of a Packed Forest. In this formulation of a packed forest, there will be at most one node for every category in the grammar, for each possible span (*start, end*). Therefore there will be no more than $|S| \cdot n^2$ top-level nodes, where $|S|$ refers to the number of symbols in the grammar, and n refers to the number of input words.

The expansion is not so nice for the sets containing children. In fact, there may be one set of

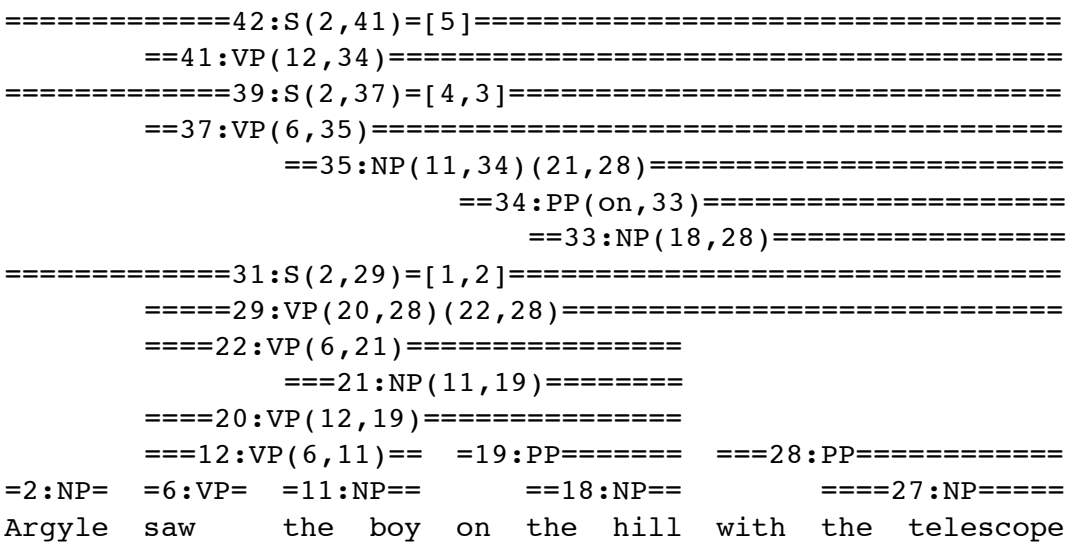


Figure 3.3: Example chart with packing. For clarity of comparison, redundant nodes have been deleted without re-numbering the remaining ones. (Figure after Allen [3, Fig. 6.18; p. 179].)

children corresponding to each individual rule in the grammar. Therefore, there may be as many as $|G| \cdot |S| \cdot n^2$ sets of children, where $|G|$ refers to the number of rules in the grammar.

Finally, the links representing the children must be accounted for. Noting that each link runs from a set of children to some node, we can derive a loose upper bound by simply multiplying these two sizes together. So there must be no more than $(|S| \cdot n^2) \cdot (|G| \cdot |S| \cdot n^2) = |G| \cdot |S|^2 \cdot n^4$ links in the packed forest.

3.3.3 Context free parsing with packing and unification

It is tempting to assume that one can just simply integrate a unification (or feature-theoretic) system into a context-free parser by adding the additional requirement that a production can be accepted only if the corresponding right hand side is found *and* that the unification of the grammar rule and the features from the right hand side elements succeeds [3]. Unfortunately it is not quite so simple.

Simple implementation may negate advantages of packing for complex feature systems.

GLR1 Tomita, in his thesis [151], doesn't address this at all. His thesis is devoted to strictly context-free parsing. So even though the implementation of his parser that this work will be com-

pared to includes a unification system (in fact, the *same* unification system that this work uses), the ramifications of unification are not addressed in his thesis. As a result, Tomita's test for packing was category (*i.e.* rule left-hand-side identity) equality.

GLR2 In his later paper on GLR adapted to "augmented-context-free grammars" [152], (*i.e.* unification grammars), Tomita writes that unification can be added to GLR, though this addition complicates local ambiguity packing somewhat. He goes on to write that the solution is to unpack locally packed ambiguities whenever their unification value is accessed.

This is essentially the same as combining the unification values together with a disjunction (described in detail below). The advantage of this is that it *partially* addresses the problems of ambiguities interacting badly with complex unification values.

This has two problems. First, while the strictly context-free version of GLR can avoid an exponential blow-up in memory when faced with multiple ambiguities, the unification grammar writer can build a grammar that requires ambiguous nodes to be unpacked often. And when they are unpacked, they may have to be *fully* unpacked, leading to a possible exponential blow-up in storage for the unification values. Second, since the parse is done in one pass, this implicit packing must be exact and complete.

Moore and Alshawi In *The Core Language Engine*, Moore and Alshawi note the problem of packing constituents within a unification system:

> In order to handle the more general case of CLE categories that contain arbitrary terms as feature values, more complex management of the records implementing packing is necessary. In the general case, in order to check that two or more analysis records can be 'packed' under a single constituent record, it is no longer sufficient to test that their mother categories are identical. Instead, we must check whether the category in the constituent record subsumes the mother categories in each of the analysis records in question.
> [4, ch. 7, p. 145]

After an example, they continue:

> There may, in addition, be other analyses for the same segment of text, where the categories in the constituent records are unifiable, but neither subsumes the other, so

packing does not occur.
[4, ch. 7, p. 146]

They also exhibit an example showing that in their system, by using this packing technique, the number of packed nodes (or complete arcs) grows more slowly than either the number of unpacked nodes, or even the number of parses [4, table 7.1, p. 146]. They do not, however, analyze the worst-case consequences of their solution to packing.

In the introduction and system overview chapter, Alshawi and Moore note that the extension of Tomita's packing to unification structures is original to their work. [4, ch. 1, p. 7]

Carroll At first glance, Carroll [23] confronts exactly the issue of packing with unification. In his work, the unification system is tightly integrated into both Earley-style and LR-style parsers.

Carroll was working at somewhat of an advantage, in that his unification system was in some sense a best-case system. His parser was designed to parse grammars written in a unification framework closely modeled on GPSG [39, 23]. In this system, a category is not an atomic value, but rather a feature structure [23, p. 57]. This feature structure is extremely flat, however, being almost a simple vector of atomic values. As a result, Carroll, citing Moore & Alshawi, then can claim that "the correct test for packing is category subsumption." [23, p. 57],[4, p. 145].

I have every reason to believe that for Carroll's system, just as in Moore and Alshawi's, that subsumption was the solution to packing was indeed the correct analysis of the situation. Since the unification values are so simple, it is quite possible to get, for example, multiple phrasal analyses that either have the same unification value for each phrase, or for one to subsume the other as far as restrictions go. As a result, these phrases can be clustered together easily.

Unfortunately, this test for clustering does not apply to our situation; our parser is designed to use an LFG-inspired unification system [20, 89, 110, 111]. In this environment, the unification values are not simply representations of the restrictions imposed by a phrase on an immediately dominating rule, but instead are more abstract representations of the entire phrase. [40]

Suppose that instead of a simple feature system of *e.g.* GPSG, using only values such as *count* and *gender*, we use the much richer F-structure system of LFG. One of the important aspects of an F-structure analysis is that it captures not only simple agreement information, but also the essence of the *structure* of the input.

(Note that I will *not* be arguing for or against the particular choices of F-structure in the following examples. These are an example of what is used by KANT system, as part of the machine

translation process, for a particular domain. Thus for the purposes of this parser, the production of these structures can be considered a functional requirement.)

The problem should now be clear: two ambiguous productions will result in two *distinct* F-structures. Consider the following example pair[1]:

```
"saw the man on the hill", with "on the hill" modifying "man":

((NP-SUBJECT +) (FORM FINITE) (TENSE PAST)
 (MODIFIED +) (CONTRACTED -) (EXPL-SUBJ -) (VALENCY SUBJECT+OBJECT)
 (IMPERSONAL -) (CAT V) (SEM *A-SEE) (ROOT "see") (ORTHO "saw")
 (NONT WORD)
 (OBJECT
    ((SEM *O-MAN) (CAT N) (CASE ACCUSATIVE) (TIME-ADJUNCTS -)
     (TIME -) (NUMBER SG) (DEFINITE +) (UNIT -) (PRONONOMINAL -)
     (NUMERAL -) (Q -) (EXPL -) (PERSON THIRD) (ROOT "man")
     (NONT WORD)
     (MODIFIER
        ((TIME -) (ROOT "on") (NP-RESTRICTED -) (CAT P) (SEM *K-ON)
         (NONT WORD)
         (OBJECT
            ((CAT N) (UNIT -) (TIME -) (INTERROGATIVE -)
             (EXPL -) (CASE ACCUSATIVE) (COUNT +) (NUMBER SG)
             (DEFINITE +) (PRONONOMINAL -) (NUMERAL -) (Q -)
             (PERSON THIRD) (ROOT "hill") (SEM *O-HILL)
             (NONT WORD)
             (DET
                ((ROOT "the") (DEMONSTRATIVE -) (CAT DET)
                 (DEFINITE +) (SEM *DET-THE) (NONT WORD)))))))
     (COUNT (*OR* + -))
     (DET
        ((ROOT "the") (DEMONSTRATIVE -) (CAT DET) (DEFINITE +)
         (SEM *DET-THE) (NONT WORD))))))
```

```
"saw the man on the hill", with "on the hill" modifying "saw":

((NP-SUBJECT +) (FORM FINITE) (TENSE PAST)
 (MODIFIED +) (CONTRACTED -) (EXPL-SUBJ -) (VALENCY SUBJECT+OBJECT)
 (IMPERSONAL -) (CAT V) (SEM *A-SEE) (ROOT "see") (ORTHO "saw")
 (NONT WORD)
 (OBJECT
    (( *O-MAN) (CAT N) (CASE ACCUSATIVE) (TIME-ADJUNCTS -)
     (TIME -) (NUMBER ) (DEFINITE +) (UNIT -) (PRONONOMINAL -)
     (NUMERAL -) (Q -) (EXPL -) (PERSON THIRD) (ROOT "man")
```

[1] I am indebted to Krzysztof Czuba for the grammar used in producing these examples [28, 29].

```
    (NONT WORD)
    (COUNT (*OR* + -))
    (DET
       ((ROOT "the") (DEMONSTRATIVE -) (CAT DET) (DEFINITE +)
        ( *DET-THE) ( WORD)))))
(ADJUNCTS
 ((TIME -) (ROOT "on") (NP-RESTRICTED -) (CAT P) ( *K-ON)
  (NONT WORD)
  (OBJECT
   ((CAT N) (UNIT -) (TIME -) (INTERROGATIVE -)
    (EXPL -) (CASE ACCUSATIVE) (COUNT +) (NUMBER SG)
    (DEFINITE +) (PRONONOMINAL -) (NUMERAL -) (Q -)
    (PERSON THIRD) (ROOT "hill") (SEM *O-HILL)
    (NONT WORD)
    (DET
     ((ROOT "the") (DEMONSTRATIVE -) (CAT DET)
      (DEFINITE +) (SEM *DET-THE) (NONT WORD)))))))))
```

These F-structures are two representations of a perfectly normal verb phrase; and either could be attached to a subject to form a complete sentence. Thus it would be best if they could be clustered together. Yet, they cannot be clustered on the basis of subsumption, since neither subsumes the other. This situation should not be a surprise; as Shieber points out, "Subsumption is only a partial order — that is, not every two feature structures are in a subsumption relation with each other." [137, p. 16].

As a result of this difference in F-structures, these two ambiguous productions can no longer be clustered via subsumption. And neither can any productions that use these as children. Such a system is now in trouble: ambiguity can now lead to an exponential blow-up in the storage used by successful productions once again.

To see this, consider the ambiguous readings for the following examples:

- `Argyle saw the man on the hill with the telescope.`
- `Kellogg broadens its reach into the food industry on wall street.`
- `Grease flows through the bearing seal.`
- `Accuracy is dependent on the amount of fuel that is burned in relation to the volume of flow.`

It is important to emphasize here that simply sharing sub-structure in the F-structure will not get us off the hook: the feature values are different for the different ambiguous readings, so otherwise

clusterable productions are not clustered.

Is this a problem in practice?

This example is clear enough, but it is appropriate to question how often this situation actually arises. Unfortunately, it happens rather often.

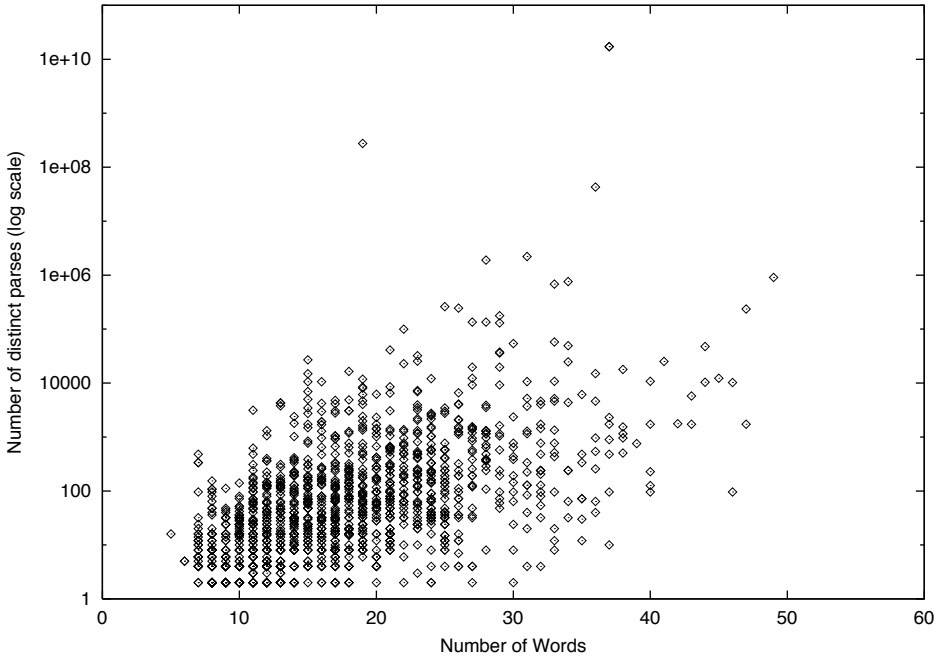


Figure 3.4: Number of words vs. Number of ambiguous analyses

Figure 3.4 shows a scatter-plot of the number of words vs. the number of analyses (or readings) of each sentence, for 1447 test sentences chosen because they each exhibit some sort of ambiguity. (For more information on these test sentences, see Section 5.4 below.) We can see from the figure that there can be a remarkably large number of ambiguities in some sentences; below we will see that the number of readings can be exponentially proportional to the number of phrases in the sentence.

Can we simply pack with disjunction?

As noted above, the LFG formalism allows disjunction within the unification framework, whereas GPSG/HPSG formalisms do not. This feature suggests a possible direction for avoiding the blow-up in unification space: simply pack the unification structures by wrapping local ambiguities together in disjunctions.

Of course, in order to avoid a blow-up, these disjunctively packed structures must *never ever* be unpacked. This should seem an obvious point, but was is in fact a problem that had to be partially corrected in the process of developing the test system, with some remaining cases still possibly remaining yet extremely difficult to pin-point and correct.

Miyao This seems an obvious point, and one on which we shall expand substantially, it can still be subtle enough that it is missed by some. In [92], Miyao proposes a packing system that he claims is substantially smaller than a simple disjunctive packing, or an enumeration of non-packed feature structures. His system can be substantially smaller than disjunctive packing in an absolute sense by the use of a clever data structure to store the disjunctions. In this structure, the over-all feature structure is represented as a tree of (possibly nested) slot-value pairs, with points of difference between individual choices indicated with a variable, and a *dependency function* — an enumeration of possible bindings of sub-trees to variables.

Unfortunately, a close examination of his proposal reveals that it can be only a constant factor smaller than an enumeration of non-packed feature structures, and larger than simple disjunctive packing. The key observation that leads to this conclusion is to note that the *dependency function* structure requires a separate listing of bindings for *each distinct overall structure*. As we have shown above, this will lead to an exponentially large number of listings of bindings. So while his method may be some constant factor more efficient, it will still suffer from the sort of exponential "blow-up" that we are trying to avoid.

In the next section, we will see first that this problem of ambiguity cannot really be avoided altogether, and second that an unpacking such as this must lead to an exponential blow-up of the unification data structures — in the particular case of [92], the structure containing the *dependency functions*. The unfortunate conclusion is that Miyao's proposal must be substantially modified to avoid this unpacking, and thus blow-up, before it can be effectively adopted.

3.3.4 Ambiguity inherently causes disjunction

The problem we have been hinting at above is this: grammars are inherently disjunctive [137, p. 42], so structural ambiguity causes disjunction.

Recall that structural ambiguity is a nondeterministic choice between multiple ways of applying the grammar rules. What this means is that a structurally ambiguous input could be parsed using one set of rules (applied in a particular, ordered, way) **or** a different set of rules **or** ...

Further, as we have discussed above, a set of rules can be viewed as a proof that an input is in the

language described by the grammar. For languages that can be described by a set of rules (in some arbitrary form) such as we are dealing with in this work, this proof is of the form 'rule x applies **and** rule y applies **and**...'

This means that to encode a *set* of parses, we need to encode a *disjunction* over individual parses. At a casual first glance, this may not seem to be much of a problem, but in fact, it is. As we have seen above, we can easily end up with an exponential number of parses in extreme cases of ambiguity *in real languages*. If we do nothing to encode this ambiguity, we will end up having to spend an exponential (in the input length) amount of space to store the parses, and spend an exponential amount of time building this structure. As a matter of good engineering, we do *not* want to do this.

But because the ambiguity encodes a *choice* over sub-analyses, efficiently encoding the ambiguity involves encoding this choice in a more efficient way. We must either directly encode the disjunctive choice over sub-analyses, or we must use some different, more obscure encoding. In any case, it involves somehow encoding disjunctions buried in the middle of a set of ordered conjunctions.

It is important to emphasize here that this is true for unification values as well as for *e.g.* context-free rules. As we have seen above, if we just ignore this, we may very well experience a blow-up in the unification space even though we have guarded against this in the context-free portion parser.

The conclusion is that for any sort of data in the parser (such as the unification values), if we have ambiguity that is reflected in this type of data, and we have effective packing that prevents against exponential blow-ups due to this ambiguity, then we must somehow encode disjunctions within that data.

3.3.5 Solution to ambiguity Packing is not Subsumption

Several researchers have suggested that "the solution to the packing problem is subsumption," including Kay [72], [4, p. 142] as well as Carroll [23, p. 57]. Subsumption is a type of generalization. Suppose we had two feature structures a and b. If a and b have no features that are in direct conflict (*i.e.* do not share any features with different values for each respective one), then the subsumption of a and b is whichever of a or b has the least specific set of features. [137, pp. 14–16].

Suppose that we have two unification values a and b:

$$a_1 = \left\{ \begin{array}{l} (\textit{feature1} +) \\ (\textit{feature2} +) \end{array} \right\} \qquad a_2 = \left\{ (\textit{feature1} +) \right\}$$

The result of applying the subsumption operator on a and b would then be:

$$\text{subsume}(a_1, a_2) = \left\{ (\textit{feature1} +) \right\}$$

This is because a_2 does not specify *feature2*, so leaving it unspecified results in the more general unification value.

Note, however, that it is possible to have a pair of feature structures such neither can subsume the other. [137, p. 16]. This case can arise when both feature structures share a feature, but this feature has a different value in each of the two feature structures (that is, the feature-values are in conflict). In this case, we could not use subsumption at all, and would be forced to keep to keep these structures separate. This may not be a rare occurrence — indeed, this situation is not at all rare in our system.

Packing with subsumption may give incorrect results

The first problem with trying to use subsumption to pack is that it is often undefined. In fact, as we have seen above, for grammars such as we are examining it is *usually* undefined. The second problem is that it doesn't do what we need it to do even when it is defined.

Consider what happens if we have a region of local ambiguity A, and try to cluster two different sub-analyses of this region a_1 and a_2. As above, for analysis a_1, we have (*feature1* $+$) and (*feature2* $+$), but for analysis a_2, we have only (*feature1* $+$) leaving *feature2* unspecified.

If we pack these two features structures together, using subsumption we will compute the packed value as $a = \text{subsume}(a_1, a_2) = a_2$ in this case. Thus for the packed node, *feature1* will be specified as $+$, but *feature2* will be unspecified.

Now consider what happens when the resulting unification value is subjected to the filtering condition $\textit{feature2} = -$. What *should* happen is that analysis d should succeed and be included, but analysis a should fail, since it has *(feature2 +)*.

But what does happen is that because we clustered together a with d, both sub-analyses will be considered valid derivations, even though a should have been filtered out. Obviously in cases like this it would be better to *not* pack at all, in order to assure correct results.

One can, of course, verify the validity of the derivations by unifying back down the trees, but doing so entails extra work, and in the case of full (non-pseudo) unification, may itself imply an explosion of instances of unification values for the lower-level structures (one for each *combination*

of higher-level constructs).

And as we have seen above, simply not packing is *not* a good answer to this problem: if this happens often enough, we may very well experience an exponential blow-up in the space of unification values. As a result, we cannot really consider subsumption to be a "solution."

Packing with a commonality operator

Suppose we wanted to define a new operator, common, a generalization of subsumption, to be used to pack feature structures. We will define $\mathrm{common}(x, y)$ as most specific set of features which do not conflict with either x or y. Thus when x subsumes y ($\mathrm{subsume}(x, y) = x$), then $\mathrm{common}(x, y) = \mathrm{subsume}(x, y) = x$, but when $\mathrm{subsume}(x, y)$ is undefined, then letting $z = \mathrm{common}(x, y)$, $\mathrm{subsume}(x, z) = z$ and $\mathrm{subsume}(y, z) = z$.

The advantage of the common operator is that it will always work (*i.e.* is always defined). The problem with common is that it still loses information.

Consider what happens if we have a region of local ambiguity A, and try to cluster two different sub-analyses of this region a_1 and a_2. In analysis a_1, we have *(feature1 +)* , but *feature2* unspecified. In a_2, we have *(feature2 +)* , but *feature1* unspecified.

Now consider what happens when we try to compute $\mathrm{common}(a, a_2)$. *feature1* is not specified in a_2, so it will have to be left unspecified. But *feature2* is not specified in a_1, so it too will be left unspecified.

Finally, consider what happens when the resulting unification value is subjected to the filtering condition $feature2 = -$. What *should* happen is that analysis a_1 should succeed and be included, but analysis a_2 should fail, since it has *(feature2 +)*.

Because we clustered together a_1 with a_2, both sub-analyses will be considered valid, even though one isn't. If we have a non-pure unification system such as pseudo-unification such that we require features to be set to values *without* changing the feature structure under test, then *neither* of these derivations are valid. In this case, we would like the unification on the packed node to fail, rather than succeed.

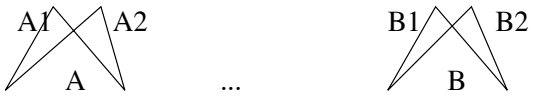
The fundamental problem with subsumption, or even a generalization of subsumption, is that using it to pack involves *throwing out information*. We have just seen that throwing out information during packing is not going to get us what we want, thus neither proper subsumption, nor a generalization of it can be a solution to our problem.

3.3.6 Problems with Packing in Disjunctions

Suppose, rather than packing feature structures together by throwing out information, we wanted to keep the information intact. One reasonable option is to pack the feature structures together by keeping track of optionality in a disjunction.

One problem with packing unification equations using disjunctions is that the unification values may have to be "unwound" in order to correctly calculate the result of later unifications.

Suppose that we have two separate regions of local ambiguity A and B. Further suppose that for region A, one choice has feature *(f1 +)* and the other has *(f1 -)*, and likewise region B has an ambiguity, one with *(f2 +)* and the other with *(f2 -)*.



Now suppose that somewhere upward in the parse tree, these structures are combined, via one of two different rules, both creating a structure for some category C, such as shown in Figure 3.5. One of these rules r contains the unification equation *(f1) = (f2)*, and the other rule r' includes *(f1) != (f2)*.

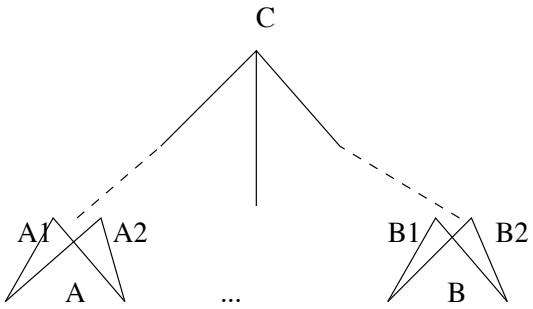


Figure 3.5: Multiple Ambiguity Unification Example

In order to correctly evaluate these unifications, in a bottom-up fashion, the subordinate unification F-structures will have to be unwound with respect to *f1* and *f2*. This will have the effect of globalizing the disjunction — that is, promoting the disjunction up to the top-level. Only then can each disjunctive term be evaluated, with some succeeding for r, and others failing. (By this particular construction, the terms that fail for r will succeed for r' and vice-versa.)

Finally, since r and r' have the same category and cover the same over-all region, they will be packed together into a single disjunction.

At this point, we now have a single F-structure with four top-level disjunctions:

- *(f1 -) (f2 -)*
- *(f1 +) (f2 -)*
- *(f1 -) (f2 +)*
- *(f1 +) (f2 +)*

This sequence can be repeated recursively, leading to an exponential blow-up of the unification space, despite our efforts to pack. The only way out of this situation is to attempt to repack the F-structures. Repacking them will avoid this particular case of exponential blow-up *in space*, but the repacking operation itself is a relatively slow operation, and will add substantially to the overhead of the parser. And, as we have seen above, repacking may not avoid an exponential blow-up in time, since an exact evaluation of unification expressions such as these require either an unpacking of the packed expressions, or a search over the packed space which is effectively the same thing. (An explicit unpacking could be part of a breadth-first search, whereas a search over the space could be part of a depth-first search. In either case, pruning combinations is a heuristic process at best [30].)

We must avoid the problem We have seen that any technique which attempts to discover *all* valid F-structures simultaneously is subject to a blow-up in space. What we would really like to do is avoid this problem altogether. In the above discussion, we have been assuming that we must perform the unification as we parse. This is not in fact the case; in the remainder of this work, we will examine what we must do in order to use this to our advantage.

We will next examine high-performance context-free parsing and the implementation of the optimizing pseudo-unification compiler. We will then examine how to achieve much of the *pruning* effect of the unification while avoiding the space blow-ups we have just seen. In order to take advantage of this pruning, we introduce a technique to simultaneously solve for a *single* F-structure while performing disambiguation in a linguistically motivated way. Finally, we put these pieces together and show that they are effective in avoiding this problem.

3.3.7 Pseudo-unification and Disjunction

Figure 3.6 shows an example of an interesting situation where node packing and unification can interact. In this example, node X is a packed node, with two choices. The rules which created these nodes are included for clarity; one choice includes the unification rule

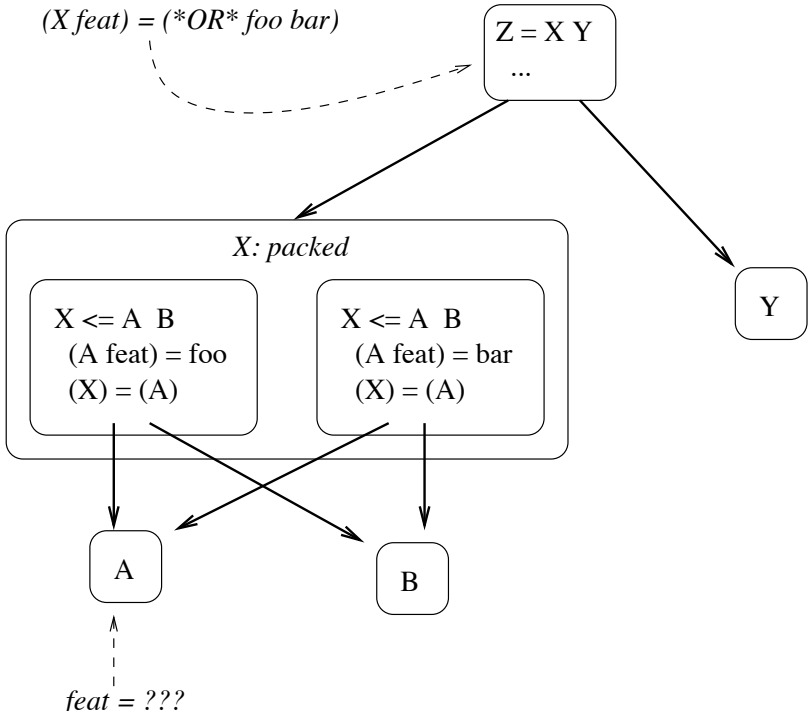


Figure 3.6: A unification example: a partial tree including a packed node X

In a full unification system, the central question for this example is: what should the value of *feat* be for node A? There are really two possibilities which make sense:

1. We could have one instance of node A, with value *feat = (*OR* foo bar)*.

   This has the advantage of keeping a single instance of node A, modifying its value to contain the disjunction as we grow the parse forest.

   On the other hand, this has the disadvantage that the value associated with A becomes incoherent: In theory both instances of node X should share the same structure. But if they do so, the value unified into *feat* from the first instance of X could leak into the second instance of X, which is obviously an error. And the mechanisms to prevent this would prevent the actual sharing of structure. So we must reject this option.

2. We could have two instances of node A, one with value *feat = foo* and one with *feat = bar*.

   This solution overcomes the computational problems of the other options, but comes with the unfortunate side-effect of potentially causing a lot of extra work. In the general case, whenever a node is created, *the entire sub-forest of each of its children may have to be duplicated*. This will certainly be true for the second and later instances of the use of a node (and hence the second and later sets of children in a packed node as well).

This implies a huge amount of work. In addition, it may be subject to the kind of exponential blow-up which we described in Sections 3.3.3 and 3.3.4. Duplicating nodes is essentially equivalent to using only top-level disjunctions to express ambiguity in the unification values, so if there is any chance that there could be a cross-product-induced multiplication of ambiguities from the grammar, using this technique is sure to trigger this behavior.

3. We could have one instance of node A, with a (possibly hidden) disjunctive sets of values, one with *feat = foo* and one with *feat = bar*.

   Implementing this approach will require some potentially difficult record-keeping. Node A should have access to every distinct version of its structure, each instance of node X should only have access to the version of A's structure that it itself changed (and not the one its sibling changed). Finally, node Z should have access to both of the versions of A's structure which the two X modes modified.

   Given such an implementation, this approach has the advantage of keeping the values distinct in the right sorts of ways, and as a result calculating the unification values correctly.

As we have noted above, the example shown in Figure 3.6 is not a problem for Pseudo-Unification because for a bottom-up parser, applying the unification equations in (either instance of) X will not change the value associated with A. It will, however, change the value associated with each of the Xs which are passed on to the application of Z's equations.

We are not implying that using pseudo-unification will totally avoid ambiguity problems; it will not. However, if one is only using the unification value at the top-most level in the end, it simply does not make sense to spend a lot of extra effort keeping the sub-tree values correct when we will never use these values.

Chapter 4

Delayed Unification Parsing

> Never put off till tomorrow what you can avoid altogether.
>
> — *modern proverb*

One of the main problems we have confronted in Chapter 3 was the cost of adding unification to a parser, and potential for a run-time blow-up caused by unification. Perhaps the most straight-forward ways to add unification is as an interleaved process [152]. Although an interleaved solution is straight-forward, there is no *a priori* reason why this is the best way to include unification in a parser [85, 115, 9]. We found that for this system it was advantageous to calculate some sort of unification values in an interleaved manner, but to delay the calculation of the entire final unification value (or values) until after the actual parse. In this chapter, we consider several strategies in the literature, and then begin to develop the strategy used in this system. Chapters 5 through 8 will then describe the implementation of this strategy.

4.1 Delaying Unification Until After Parsing

The method we have been examining is *interleaved* unification. Recall that the conditions for creating a valid constituent r are two-fold: not only must the children have the correct categories, but the F-structures of the children must unify according to the equations that are also part of r. For the system above, we check to see that r unifies immediately after discovering that r can be created according to the context-free spine of the grammar.

Although it is somewhat convenient to implement a parser in this way, there is no mathematical reason why we must do so. We next consider the other extreme, as suggested by Maxwell and

Kaplan [85]: delaying the unification until after the entire context-free parse is complete.

4.1.1 Interleaved unification Versus Delayed unification

We conducted an experiment to examine the effectiveness of delaying unification. For the test case, we tried parsing with a context-free (only) grammar and delaying the unification until after we have a context free backbone in-hand. All possible unification values were discovered in a recursive-descent search of this context-free backbone. This configuration was compared against the fully interleaved unification system presented above.

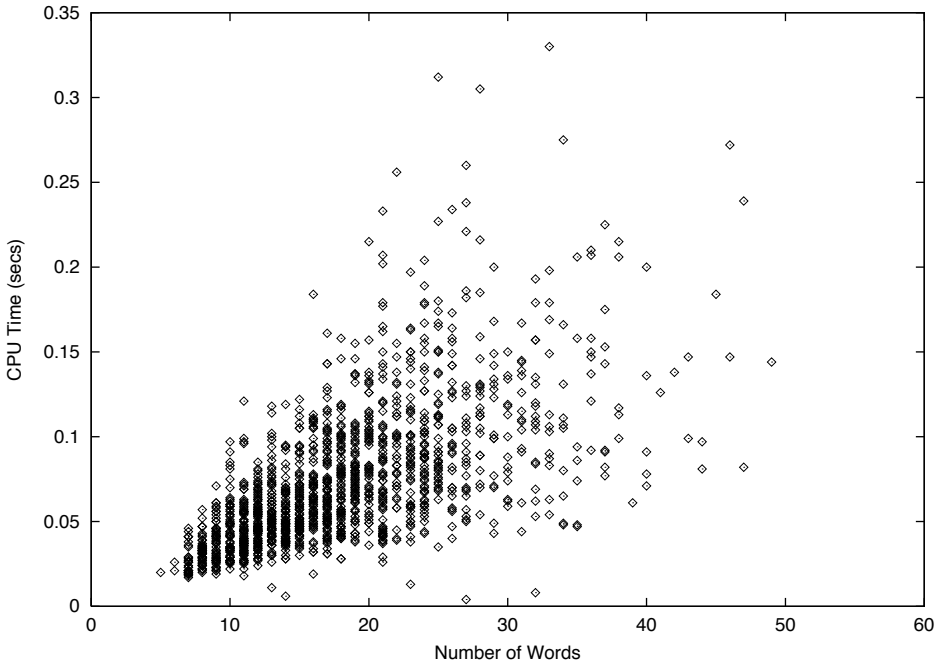


Figure 4.1: Number of Words vs. Runtime Used: Delayed Unification on a CF-spine

Figure 4.1 shows a plot of the combined time to parse and then search for all unification values for each sentence. Once again, there are three outliers at or above 4 seconds of CPU time, and these aren't too bad.

Each of interleaved- and delayed-unification has an advantage. For delayed unification, the advantage is that one need not calculate unifications for partial parses that cannot participate in a total context-free analysis of the sentence. On the other hand, the advantage for interleaved unification is that many partial parses that are valid in a context-free sense, but cannot be unified, will be immediately removed from consideration, saving considerable resources.

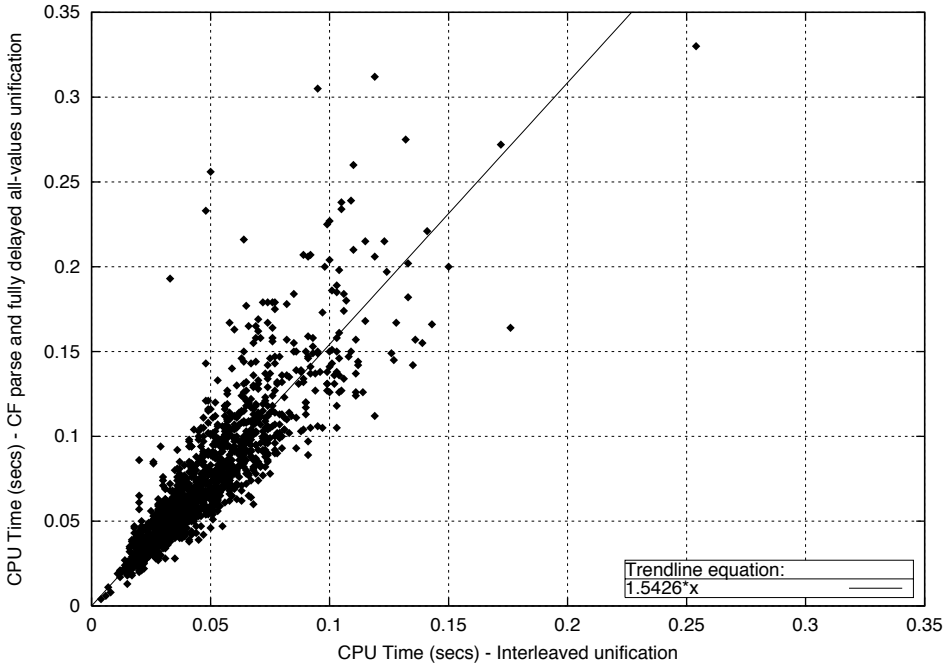


Figure 4.2: Runtime Used: Interleaved vs. Fully-Delayed Unification

Discussion According to Maxwell and Kaplan [85], the advantage in speed should be decidedly in favor of delayed unification. Yet the results shown in Figure 4.2 are counter to this: interleaved unification has a slight speed advantage over delayed unification — a total of 1228.75 CPU seconds spent to parse all utterances on this test corpus for the interleaved unification configuration versus 1392.10 CPU seconds to parse and search for all complete unifications for the fully-delayed configuration. This is a significant difference: a single-tailed matched-pair T-test gives result at p = 1.29787E-78. Also, although most sentences take about the same amount of time to parse in the interleaved configuration as they do in the fully-delayed configuration, there are a few sentences that take decidedly more time in the delayed-unification situation.

We will see below that there is another large disadvantage to delaying all of the unification until later: if we try to integrate a disambiguation system into the post-parse search, and search not for all valid F-structures, but for a single best F-structure, the context-free spine may not provide sufficient constraint for the search, and the search may spend inordinate amounts of time on unfruitful derivations. In other words, it will blow up by getting "garden pathed".

4.2 Delaying *Some* Unification Until After Parsing

Rather than either doing *all* of the unifications interleaved, or delaying *all* of them until later, what if we interleave *some* of the unifications, leaving the remainder for after the main parse?

The Core Language Engine Moore and Alshawi [4] take such an approach, separating the unification values into two separate parts: in the parsing phase, the unification value is formed solely by a union of simple feature values, which are compiled into a binary feature vector. By tightly integrating the parser and the syntactic feature system, Moore and Alshawi are often able to cluster locally ambiguous nodes. Within such a feature system it *could* be possible to avoid exposing the semantic differences of *e.g.* PP attachment, thus allowing local ambiguities of this sort to be clustered together easily.

As we have argued above in section 3.1.2, we cannot simply adopt the Core Language Engine as-is, however. It is based on a substantially different theory of syntax than ours system is, so adopting it as-is would involve substantial amounts of reengineering effort.

We have argued above in Section 2.2.1 that ambiguity is a problem for machine translation systems, and that we generally have to resolve ambiguities as part of the translation process. In addition to having some sort of disambiguator, one of our goals is to use a disambiguation technique that is both linguistically and computationally motivated.

Unfortunately, the Core Language Engine, as presented in Moore and Alshawi [4], does not include such a component; the problems of ambiguity are touched on [4, p. 141], but no solution other than packing is presented. Therefore integrating the Core Language Engine parser into the KANT machine translation system would require us to create that piece of the puzzle.

So, rather than adopting the Core Language Engine itself, we shall examine it carefully, and adopt several ideas from it, and integrate these into our parser. Our goal for unification processing is to allow the system to transform the problem so that we can gain the advantages in processing that the Core Language Engine exhibits, while changing our (working) source grammars to the least reasonable extent.

4.2.1 Negative Restriction

In Section 3.1, we examined Shieber's *restriction* technique, which allows one to adapt a pure unification grammar for use with a context-free parsing algorithm. The essence of Shieber's technique is to specify which unification features should be used to create a context-free spine grammar [136].

Harrison and Ellison's extension of restriction

One unfortunate problem that Shieber discovered is that there are some unification grammars for which the predictor step of his modified algorithm will not terminate. Harrison and Ellison [45] examined this problem in detail, exhibiting two grammars for which Shieber's algorithm with restriction will not terminate. They then examine why this is, and examine some possible changes to the algorithm to address the difficulty.

They conclude that in some cases, this non-termination problem is inherent in the nature of the grammars, but that grammars exhibiting this problem fall outside the formal class *off-line parsable* in the sense described by Kaplan and Bresnan in [56].

To work around this problem, Harrison and Ellison introduced the concept of *negative restriction*: rather than to specify which features *are* interesting, which they describe as "positive restriction", they specify which features are *not* interesting.

Trujillo

The FIRST and FOLLOW relations are used in calculating both Left-Corner and L-R parsing tables [1, pp. 188-189] (see also section 6.4 below).

Trujillo [156] extended the calculation of these relations from context-free grammars to feature-theoretic ones. Without the use of some sort of equivalence-relation, it is impossible to calculate these relations in the general case, because the number of possible nonterminals in a feature-theoretic grammar can be infinite. Even when finite, the use of these relations would either help the efficiency of a parser very little, or be so time-consuming to compute during the parse as to be no aid. Trujillo found that by using negative restriction to establish a set of equivalence classes, it was possible to calculate and use these relations productively.

Oepen and Carroll

Establishing and using equivalence classes as done by Trujillo [156] is very similar to the problem of packing. Contemporaneous with early parts of our work, Oepen and Carroll [115] investigated the problem of ambiguity packing, and the use of negative restriction to aid packing, in the context of an HPSG system.

They introduce a new linear-time bidirectional algorithm for determining subsumption, and use this to improve the ambiguity packing portion of their parser.

They also examine negative restriction, and investigate its use to improve ambiguity packing during the parse phase. They found that: "Restricting compositional semantics but preserving attributes that participate in selection and agreement results in minimal chart size and parsing time." [115]

They show that by employing these two techniques together, they are able to get good performance on a the PP-attachment case reported by Billot and Lang [15] and Moore and Alshawi [4]. Oepen and Carroll's parse times are approximately cubic in the number of PPs, though their unpacking (*i.e.* post-parse resolution) times are proportional to the number of readings, which is exponential in the number of PPs. (See section 3.3.1 below.)

In this work we expand on these results, both in packing and in unpacking of the final unification values.

4.3 The Two Purposes of Interleaved Unification

Rich and Knight point out that in general, a parser performs two kinds of operations: matching (of constraints) and building structure [121, p. 395]. In the abstract, there is no particular reason why we would want to choose the context-free constraints to perform early, while delaying the unification constraints until later.

Suppose, like the Core Language Engine [4], that we had a parser with *two* distinct unification systems: one with a simple-valued unification system, to be used only to filter the parse forest as it is being built, in order to avoid building structure that will not be ultimately useful; the second with a full-featured unification system to compute the eventual F-structures that we are interested in getting in the end. The general idea is to use the first parse as a *filter* to reduce the search space, and only when this space is some reasonable size look at full (or 'semantic') abstract unification structures [4, 85].

4.3.1 'Cheating' in the interleaved unification

Further suppose that we will throw out (*i.e.* ignore) all unification values calculated during the first pass, using only the structure of the parse forest to guide our post-parse calculation of unification values. Such a framework is very powerful: *it will allow us do almost anything at all during the initial parse,* so long as we do not exclude any structures that could potentially lead to a correct parse.

This allows us to 'cheat' by approximating, in the earlier pass, anything that would be expensive to calculate exactly. The preservation of partial unification results, as is done in parsers employing restriction such as Oepen and Carroll's [115], assumes that partial unification values are expensive to calculate.

In Section 5.4, in the next chapter, we will examine experimental results which show that this recalculation is *not* expensive – that for average data, complete recalculation of the unification values requires only as much as 14% of the time required to perform the back-bone parse.

Thus the basic engineering trade-off is time spent calculating unification values in the second pass versus time spent in the parse due to packing or lack thereof. Since we found that complete recalculation of unification values in the post-parse phase is not expensive, we can consider methods to improve packing which do not exactly preserve the unification values calculated during the parse.

Approximation in the packing of unification values Since we are doing a fresh unification in the second pass, we do not need to maintain strictly accurate F-structures in the initial parse. We could, however, maintain F-structures that *include* every structure that would be part of a correct parse. In other words, we can over-generate, but cannot under-generate F-structures.

Consider, for example, two F-structures: `((a b) (c d))` and `((a x) (c y))`. The strictly correct way to pack this disjunction would be:

```
(*or* ((a b) (c d))
      ((a x) (c y)))
```

But since we will do another pass to compute the correct F-structure, we can approximate this particular case as:

```
((a (*or* b x))
 (c (*or* d y)))
```

In general, this packing technique generates the Conjunctive Normal Form equation which represents a superset of the original equation by removing any inter-slot dependencies, and then pushing the disjunctions down to the leaf (individual slot) level of the unification value tree.

The advantage of such a packing is that it can allow items to be packed more compactly; possibly avoiding the exponential blow-up of the exact disjunction. The disadvantage is that such a disjunction will allow previously disallowed elements, such as `((a b) (c y))`. Obviously there is a time/space trade-off here, but we do get to make the trade-off.

This technique closely corresponds to modifying Miyao's proposal [92] by removing the *dependency function* entirely, and simply noting the packing points (represented above with an `*or*`, and not worrying about the fact that we will likely over generate the actual disjunctive possibilities.

This is a permissible approximation, because the F-structures will be completely generated anew in the second pass, so these approximately packed F-structures will be thrown out. Their only purpose is to ensure that further structures built on them won't be grossly over-generative.

Wild-carding feature values Sometimes it would be useful, in the first pass, to be able to skip over parts of the unification equations. Fortunately, the feature-based flexibility of the unification system of Rosé and Lavie's LCFlex system [125] suggests a solution: it is possible for us to introduce a wild-card to stand in place of part of an F-structure. This wild-card will have the effect of causing any unification equation that reaches this part of the F-structure (either as a value, or as part of a path) to succeed, returning an appropriately wild-carded F-structure result.

In LCFlex [125], this wild-card is used to allow certain features to be optional, rather than required parts of the unification. But this use is to enhance robustness; we have somewhat different goals. One of the other things that we can use this for is to impose a limit on the length of a number of values in a disjunction.

Suppose that we are going to pack together the three F-structures `((a b) (c d))`, `((a x) (c y))`, and `((a y) (c y))`. Further suppose that we are applying the approximate disjunction technique described above, and we have imposed a length limit: no disjunction my contain more than two terms.

After packing the first two F-structures, using the approximate packing technique we have just discussed, we will have:

```
((a (*or* b x))
 (c (*or* d y)))
```

But when we try to pack in the third F-structure, we will exceed our disjunction length limit for the feature `a`. So instead we will replace this disjunction with the wild-card `*WILD*`. The result will be:

```
((a *WILD*)
 (c (*or* d y)))
```

The results of unifying with a `*WILD*` are slightly different than unifying with a normal value. For example, A `*WILD*` unified with anything should return a `*WILD*`; asking if `*WILD*` is defined should be true; Asking if `*WILD*` is undefined should also be true.

Using these wild-cards to impose a length limit can eliminate the possibility of exponential blow-up in the face of ambiguity, but will cause the F-structures to be substantially over-generated.

Shallow Unification Another limitation we could impose is to limit the depth we will search F-structures [56], or even limit the depth of the F-structures themselves. We have seen above that these deep F-structures contribute to exponential blow-ups. In order to help prevent this, we might wish to limit our first-pass unification to be only a single level deep.

For example, consider the rule:

$$\begin{aligned} NP \quad \rightarrow \quad & NP \qquad PP : \\ & \langle X_1 \textit{ can-mod} \rangle = + \\ & \langle X_0 \rangle = \langle X_1 \rangle \\ & \langle X_0 \textit{ mod} \rangle = \langle X_2 \rangle \end{aligned}$$

We know that the unification equation $\langle X_1 \textit{can-mod} \rangle = +$ is a shallow operation — it looks at the F-structure of the right-hand-side NP to see that *can-mod* is either set to a $+$, or can be set to that now.

The equation $\langle X_0 \rangle = \langle X_1 \rangle$ identifies X_1 as the lexical head of the rule. Evaluating this equation will produce no more deep structure in X_0 than X_1 has already, so we will assume that it is safe to keep as-is. Also, were we to wild-card these sort of equations, we would render the actions of the unification equations of other rules effectively moot, and effectively relax the grammar to little more constrained than Context-Free. So we will keep rules such as this one.

Finally, we can tell that the equation $\langle X_0 \textit{ mod} \rangle = \langle X_2 \rangle$ *does* produce deep-structure, since it puts the entirety of the right-hand-side PP F-structure into the slot *mod*.

If we weren't too worried about this last feature being used to control the size of the parse, we could re-write this particular equation to be something like: $\langle X_0 \textit{ mod} \rangle = \textit{wild}$ By doing this, any future reference into the left-hand-side's *mod* feature will succeed, returning *wild*.

We could do all of this entirely at parse time (*i.e.* run time), replacing deep structures with wild-cards as they are built. But since we can tell in advance which unification equations are going to produce or need deep structures and which won't, there is no reason to wait until run-time to do this.

If we were to apply this knowledge ahead of parse time, and apply it to our example rule above, we could re-write the unification equations as:

$$\langle X_1 \textit{ can-mod} \rangle = +$$
$$\langle X_0 \rangle = \langle X_1 \rangle$$
$$\langle X_0 \textit{ mod} \rangle = \texttt{*WILD*}$$

4.3.2 Our unification approach

From our full unification grammar, we will make *two sets* of unification equations: from each full set of equations associated with a rule we generate a set of equations which produce and support only these shallow unification operations outlined above.

We will interleave the shallow unification system within the context-free parser, in order to help prune the parse forest as it is being built. After a covering parse forest has been found, we will use search and disambiguation techniques to find the result of unifying a single, preferred, parse tree. The next four chapters cover how this is done.

Chapter 5

Overview of the Approach

> Patient: "Doctor, doctor, it hurts when I do this."
> Physician: "So don't do that."
>
> — *Old Vaudeville joke*

In Chapter 3, we investigated unification and ambiguity, and how the latter can adversely effect implementations of the former. In this chapter we will review the goals for the proposed system, and see how these goals and several high-level design decisions lead to a working system.

Our main goals for this parsing system are three-fold:

- We wish to parse quickly in the average case.
- We wish to parse quickly in certain common problematic cases, avoiding bad behaviors which lead to extreme slow-downs or outright failures.
- And we wish to make the system easy to work with. We wish to largely maintain compatibility with previous systems.

Ultimately what we want is a good tool that hits the balance between being sufficiently general that it can be used for many useful problems, yet sufficiently specialized that it can take advantage of the particulars of the structure of the particular problem it is used on. Our parsing system is designed with this in mind: we have seen that structural ambiguity can lead to major computational problems in a unification parser. We deliberately took advantage of the context of the larger system in which this parser is designed to operate — a knowledge-based machine translation system — in order to carefully avoid these problems.

In this chapter, we present a high-level view of our system. We start by summarizing the high-level design principles from which our design is derived, and then examine each of these principles in detail. We will see that each of these design principles is a departure from some 'standard' way of attacking the unification parsing problem, and why each are necessary.

Next we will review a set of detailed requirements for the parsing system itself, examining the motivation behind each of these requirements.

Finally, we examine the architecture of our actual implementation. This implementation jointly follows the design principles and detailed requirements which we have surveyed. This overview is in preparation for the three following chapters, which will present important aspects of the implementation in detail.

5.1 Conceptual Design

In this section, we consider the interrelationship of unification and the context-free parser, concentrating on *when* the unification constraints should be applied during the over-all processing of the input. Conceptually, we can summarize our design criteria in five the design principles:

1. Don't try to do the parse all in one shot.
2. Don't keep the unification values from the parse phase.
3. Don't follow the grammar precisely early in the process.
4. Don't try to find *all* possible final unification values.
5. Don't pick just *any* single unification value; pick a good one.

An explanation of these design criteria follows.

5.1.1 Don't try to do the parse all in one shot.

The single-pass context free parsers employing a chart data structure (or something closely related) employ the *dynamic programming* design pattern [78]: a total solution is found by first creating and then combining small sub-solutions until the one or more full-size solutions have been discovered.

For the context-free case this is an effective solution; a good implementation will have $O(n^3)$ run-time behavior in the general case. Unfortunately, as we have seen above in Section 3.3.3, the

addition of complex-valued unification features will defeat the fundamental assumptions of the chart algorithm which allow for an efficient solution. The result is a breadth-first search which is subject to worst-case computational time requirements which are exponential in the length of the input, with realistic inputs triggering this bad computational behavior.

The dynamic-programming breadth-first design pattern is not the only one we can employ. As we have seen above in Section 3.1.2, other solution patterns have been investigated by many researchers, including Moore and Alshawi [4], Maxwell and Kaplan [85], and Oepen and Carroll [115]. The strategy common to all of these approaches is to first narrow down a very large space of potential solutions to a much smaller space which contains the solutions (as well as possibly containing non-solutions). Once the space has been narrowed, a potentially more expensive algorithm is used to discover one or more of the actual solutions.

The other interesting feature shared by all of these approaches is that the discovery of a complete unification value is delayed until the *end* of the process. Delaying the full unification avoids performing unification calculations which will not ultimately be useful [85], as well as improving the ability to pack together sub-solutions [115] and thus partially avoid the problems described above in Section 3.3.

We will adopt this strategy, and improve upon the previous solutions through the use of additional techniques.

5.1.2 Don't keep the unification values from the parse phase.

As we just mentioned above, not trying to parse in a single pass allows for considerable flexibility in the way we process the grammar. We can gain further flexibility by *not* attempting to preserve any unification values from the first pass, but instead throwing these values out and recalculating them from scratch in the second pass.

For a parser using this general structure, the only thing which we must calculate in the first pass is the packed parse forest structure. This structure is then used to guide the second pass in finding the ultimate unification value results.

At first glance this would seem to be a foolish move, since it may imply a duplication of effort in the parse and post-parse passes, however it also allows for the calculation of completely different unification operations and values in the two passes. This structure was used by Moore and Alshawi in the Core Language Engine [4]; syntactic values were used in the parse, and then semantic values were calculated from the resulting parse trees.

Thus the principle advantage is that we can do *anything we want to* in the parse pass, so long as we create a packed parse forest which describes every parse that would be valid according to the original grammar. That is, the packed parse forest must describe a superset of all possible parses.

Adopting this design decision is based on two observations. First that recalculating the unification values can be fast in comparison to calculating the parse forest. In Table 8.1 of Chapter 8 we will see that the effort required to recalculate all of the possible unification values from scratch only accounts for some 13% of the total run-time of a two-pass parser running on natural data, with the other 87% going to the parse pass.

The second observation is that one key to high performance in the parse phase is efficient packing [151, 115]. From the relative speed of the post-parse phase which we have observed, we can reason that aiding packing in the parse phase can be more important than preserving unification values for the post-parse phase, and that a reasonable engineering trade-off can be made along these lines.

5.1.3 Don't follow the grammar precisely early in the process.

Since we have decided not to preserve any unification values calculated in the first pass along with the parse forest, we can take liberties with the grammar used in this pass.

We can certainly selectively leave out unification equations which are computationally expensive — this is one of the goals in using negative restriction [115].

Like Oepen and Carroll [115], we have found that restricting the first pass so that unification equations which enforce simple-valued constraints such as agreement should be preserved in the first pass, but that equations used to create complex structures as part of a more abstract ('semantic') analysis are best left to the post-parse processing.

We can also modify the parse-phase unification values in more elaborate ways. As we have mentioned above, aiding packing in the parse pass is extremely important for achieving high performance. Since we are not re-using the parse-pass unification values beyond the first pass, there is no reason why we must preserve exact mathematical equivalence while packing together locally ambiguous productions, so long as we do nothing which will eliminate parses which would otherwise be legal. Unlike Oepen and Carroll's exact packing technique [115], we will *always* be able to pack locally ambiguous nodes together using such a technique.

5.1.4 Don't try to find all possible final unification values.

We neither need nor want to calculate *all* possible unification values in the post-parse phase. As we have seen above in Section 3.3, the main problem with calculating all possible unification values is that there could be an exponentially-large number of them [85, 115]. Such exponential blow-ups cause bad performance of exactly the sort which we wish to avoid.

When used as part of a machine translation system, a common next step after finding all possible unification values is to *disambiguate* — to select one single parse.

Rather than find all possible values, and then select one of them, we could simply find one single best unification value. With this realization, the post-parse processing can be seen as a *search* for one or more unification values, rather than a simple 'unpacking' [115].

We adopted this approach. We created a disambiguating post-parse search, and show that this search is effective at avoiding some of the ambiguity problems which lead to exponential run-time blow-ups, without slowing the parser in the average case.

5.1.5 Don't pick just *any* single unification value; pick a good one.

The final observation is that we don't want just *any* single unification value — we want the *best* value.

The notion of 'best' here is a very important one. Consider the sentence:

> I saw the man with the telescope.

There are two readings for this sentence: either *I* used *the telescope* as a tool to see *the man*, or *the man* has *the telescope*. Since both readings are legal, we can see that a preference for one interpretation over another is not strictly part of the syntax. On the other hand, the first reading is commonly understood to be the preferred one.

In contrast, consider the sentence:

> I saw the man on the hill.

There are two possible readings for this sentence: either the speaker (*"I"*) or *the man* is on the hill. To a native English speaker, the latter reading is considerably preferred over the latter.

If the parser was designed to solve for any arbitrary complete unification value [84], the user might very well get the less-preferred reading instead of the more-preferred one. Furthermore, the

choice of which we got might not be stable in the face of changes to the grammar: some seemingly independent change to the grammar might make the system change which reading was first found for no readily apparent reason.

Instability such as we have described above makes for a very hard-to-maintain system. Large-scale natural language systems which use a natural language parser as a component, such as Machine Translation systems, are notoriously difficult to maintain [129]. Any effect which would tend to make this harder should be avoided.

So rather than picking just any single reading of ambiguous inputs, we designed the system to select a *best* unification value according to a heuristic which attempts to capture the notion of human preference which we have demonstrated above.

5.2 System Requirements

Now that we have examined a number of design principles which motivated the implementation of our system, we can survey the requirements for this particular parsing system.

What we need is a good tool that hits the balance between being sufficiently general that it can be used for many useful problems, and sufficiently specialized that it can take advantage of the particulars of the structure of the particular problem it is used on.

- **The parser need not be fully general.** The parser should not be fully general in the sense of parsing every unification formalism — as we are not proposing to switch to a radically different formalism (*e.g.* HPSG), there is no reason to worry about compatibility with such a formalism. On the other hand, we should not produce a system which is so specialized to some single target grammar that it is not general at all; the parser should be able to handle a reasonably large class of related formalisms.

  This is not an exactly radical proposal as the parsers of several other systems (including The Core Language Engine [4] and LCFlex [125] are closely tailored to the particular grammar formalism they are using).

- **Should parse LFG-style grammars.** This said, our particular grammar formalism is closely modeled on LFG: a context-free spine, with tree-structured unification values attached to the complete arcs (*i.e.* partial productions).

- **Should take advantage of current techniques in context-free parsing.** Since the grammar formalism we adopt is based on a context-free spine, the first-pass parser should take

advantage of recent advances in context-free parsing. These should include:

- A tree-structured grammar [42, 116].
- Packing of ambiguous complete arcs [163, 151, 23, 18, 4, 125, 94].
- Left-corner constraints [42, 127, 119, 41, 58, 83, 134].
- Look-ahead constraints [32, 2, 160, 1, 102, 94].
- Optimal agenda ordering [76].
- End-point restriction [23].

In addition, we also need to support unification which is interleaved within the application of Context-Free rules:

- Interleaved Unification [152, 68, 125].
- Precompiled unification [20].
- Optimization of unification equations.
- Disjunctive packing of Unification values [152, 4, 92].

- **Unification should allow for disjunction in both unification values and equations.** As we have seen above in Section 3.3, in order to avoid some common sorts of blow-ups, it is critical for the parser to both pack ambiguous nodes together, and also pack the unification values attached to the nodes. It is also convenient to allow disjunctions in the unification equations of the grammar, so we will support this as well.

- **Allow flexibility in the application of unification equations.** Above, we have seen how other researchers have benefited from applying multi-pass unification techniques, using separate unification equations for each pass. These have been both explicitly represented in the grammar and semantic rules [4] and implicitly as a mechanical process applied to the grammar [115].

  We feel that the question of whether single or multi-pass unification is better is not a philosophical question, but rather one best answered empirically. In order to perform experiments which examine these questions, we must parameterize the implementation so that it supports multiple solution techniques.

  The first (parsing) pass must support both one-pass parsing and the parse tree search for multi-pass parsing. This is to say that we support both:

- Full exact unification in the first pass.
- Shallow unification with approximate packing in the first pass.

The second (post-parse) pass must support two general search types for full, exact unification values:

- best-first search (or k-best search) with integrated disambiguation.
- breadth-first all-values search.

- **Separate best-first search from the measure of 'best'.**

  In order to support additional flexibility in design and configuration, we chose to separate the mechanics of the search away from the measure used to select which trees are better to search first. This allows us to modify and configure each of these components separately while lowering the chance that by modifying one we break the other.

- **Linguistically-motivated disambiguation.**

  Since we are *searching* for a single best result, we can improve on the usefulness of our system by not simply searching for just *any* complete unification value. We argued above in Section 5.1.5 that some types of structural ambiguities are not considered highly ambiguous by a native speaker. Ideally, we wish our system to faithfully reproduce this type of human preference.

  A number of researchers have considered the problem of disambiguation from a linguistic standpoint, including Kimball [60], Whittemore and Ferrara [159] and Mitamura *et al.* [90]. In particular, Mitamura *et al.* showed that a small number of heuristics will select the naïvely correct parse in most cases. As a proof-of-concept, we will implement a disambiguator which implements many of the heuristics used by Mitamura *et al.* [90].

- **Shallow 1st-pass unification automatically derived from full grammar.**

  Grammar authors (*i.e.* users of this parser) should not be expected to go so far out of their way as to write completely separate unification rules for the various phases of unification — they should be able to write one set of unification rules, and have the 'right thing' happen. So we will calculate the shallow-valued unification equations *automatically* from the full unification equations that the user has supplied.

We next examine the over-all architecture of our system. This is a high-level look at the system; important implementation details are left to the three subsequent chapters.

5.3 System Architecture

Cohesion and *coupling* are two criteria of good system design: loosely related components should be loosely coupled so that internal changes to one component generally do not effect any other component, and each component should implement a small number of very closely related (*i.e.* cohesive) functionalities [120, ch. 13–14].

We applied these principles to the overall architecture of our system. We have split the over-all parsing system into two phases of processing comprised of a total of five major components:

- The preprocessing is comprised of three parts:
  1. Lexical and phrasal information storage.
  2. Context-free grammar preprocessing.
  3. Unification filtering, optimization, and compilation.
- Parsing is a three-step process:
  1. Lexical and phrasal information retrieval.
  2. Parsing (Context-Free / Interleaved Unification).
  3. Post-parse search and disambiguation.

These components are kept largely separate, so that each component may be substantially modified without affecting the correct operation of the other components (the lexical and phrasal handling is actually one module).

The weak coupling between the various components, particularly the unification compilation, parsing, and post-parse search, allow considerable flexibility in the configuration of the system, which in turn allows us the freedom to easily isolate and evaluate different configuration choices. The form of the interfaces which allow this will be discussed in the sub-sections below.

Figure 5.1 contains a sketch of the highest-level view of the system. Before the actual parsing system can be run, several collections of information must be processed; most notably the grammar, from which several relations are discovered, as well as the unification equations, which are extensively processed.

We next examine this preprocessing in more detail.

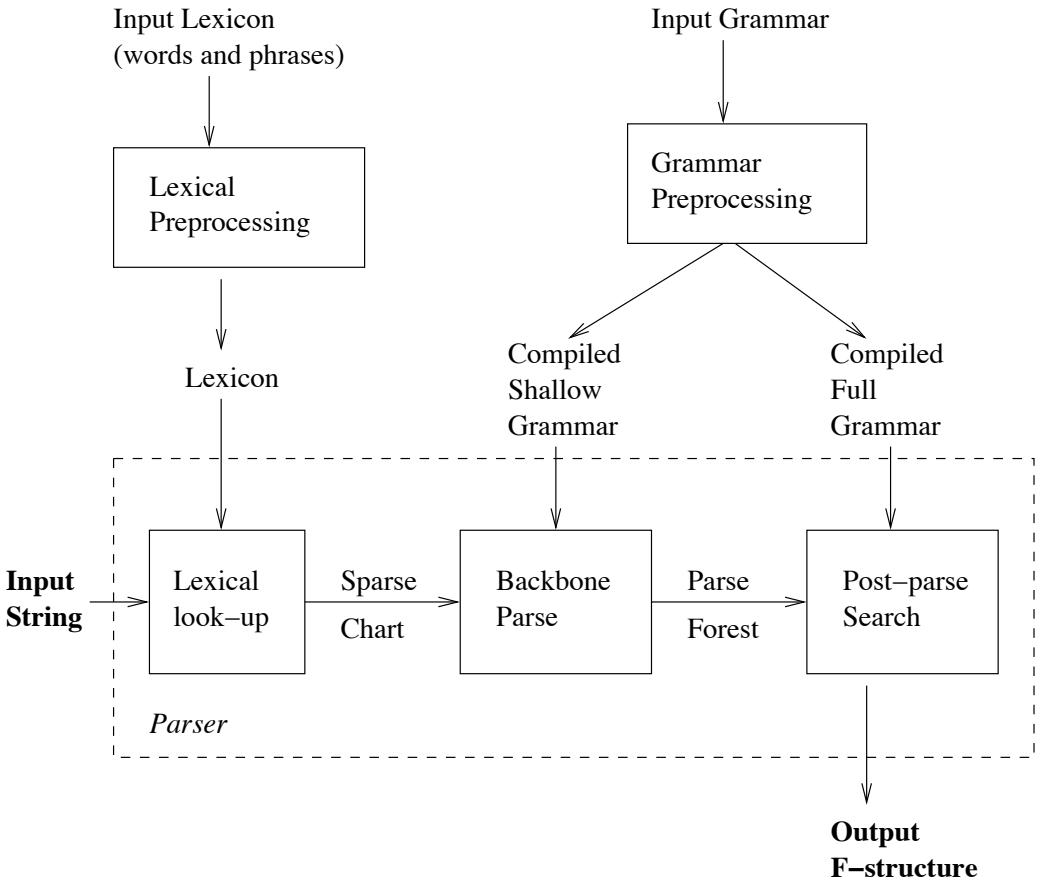


Figure 5.1: High-level View of System

5.3.1 Preprocessing

There are three independent pieces of preprocessing which must be done in order to support the run-time system: lexical and phrasal storage, preprocessing of the context-free portion of the grammar, and filtering and optimization of the unification portion of the grammar, followed by compilation of the unification equations into a form which is very rapid to evaluate.

The lexical storage is best described along with lexical look-up, in Section 5.3.2 below.

Figure 5.2 is a block diagram of the major pieces of the grammar preprocessor. This preprocessing can be divided into two categories: Context-Free grammar preprocessing, and Unification preprocessing.

This preprocessing is not particularly expensive. On the test machine (a Macintosh PowerBook G3 466; see Section 5.4 below), it takes a total of 67.2 seconds of time to entirely preprocess, compile, and load the KANT grammar.

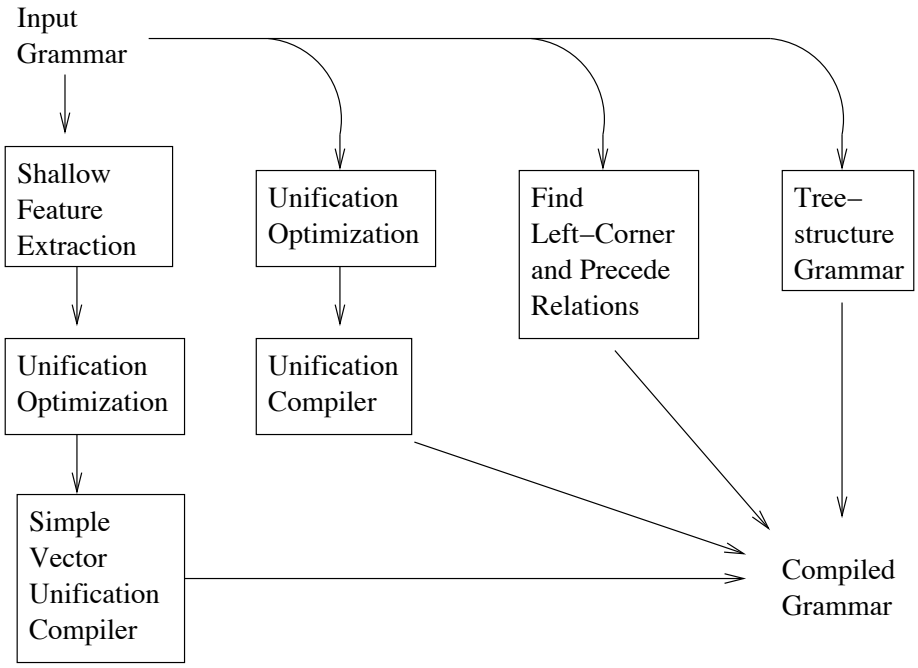


Figure 5.2: Grammar Preprocessing

Grammar preprocessing takes 22.3 seconds of time. Of this time, 14.5 seconds are required to calculate the various filtering relations, and 7.8 seconds are required to convert the unification equations into LISP code. 36.7 seconds of time are spent compiling the LISP functions derived from the unification equations into machine code.

Finally, it takes 8.2 seconds to load the compiled grammar. 4.0 seconds of this time are used to convert the context-free backbone into a *trie* data structure [132, pp. 248–252] and decorate that trie with left-corner relation information, and the remaining 4.2 seconds are used by the LISP system in loading the functions and data derived from the grammar.

Context-Free Grammar Preprocessing

There are four pieces of preprocessing which must be done to a grammar before the run-time parser can use it. Two of these concentrate on the context-free backbone of the grammar:

- The left-corner, precede (look-ahead), and partial-ordering relations are discovered from the structure of the context-free backbone of the grammar.
- The context-free backbone is reduced to a tree structure.

These preprocessing steps are discussed in detail in Chapter 6.

Unification Preprocessing

The other two major pieces of preprocessing of the grammar concern the unification portions of the grammar rules. This is a bottom-up unification system, so the unification equations in a grammar rule specify how the unification values for a parent are to be computed from the unification values of its children.

From these full unification equations, we also calculate a set of rules which support only a shallow subset of the full unification values. So in the preprocessing:

- The shallow features for the first-pass unifier are extracted, optimized, and compiled.
- The full grammar unifications for the second pass are optimized and compiled.

The *Restriction* technique, as originally proposed by Shieber [136] and extended as "*Negative Restriction*" by Harrison and Ellison [45], is a good tool for this unification preprocessing. The idea behind Negative Restriction is to describe a set of unification slots for which calculation should be delayed during parsing of an input. A unification equation using one of these slots will *not* be evaluated at the same time as the unrestricted unification equations; instead this equation will be evaluated only after the unrestricted equations for all of rules participating in any entire parse tree have been evaluated.

Contemporaneous with parts of this work, Oepen and Carroll [115] reported that it was most helpful to include unifications which enforce syntactic constraints such as agreement, but to restrict compositional semantic unifications until after all syntactic restrictions have been met.

Oepen and Carroll were working with an HPSG [118] grammar. Our grammar formalism is considerably different than theirs; the differences in grammar authoring conventions give rise to the need for slightly different handling of restrictions.

This is best demonstrated with a brief example, of a structure which occurs fairly often in the KANT grammar (see Section 5.4.1 below). We might have a grammar rule containing:

```
(X0 obj) = X2
```

We might also have, in some parent grammar rule:

```
(X2 obj) = *UNDEFINED*
```

While we could simply eliminate both of these equations, doing so would lose some precision in the first-pass grammar. These two rules illustrate a fairly common practice in our largest grammar.

Syntacticians often prefer a notion of minimum expressibility, leading to a minimization of several countable aspects of the grammar, notably including the number of distinct unification slots used. Because the `obj` feature can serve double duty, both forming part of the compositional semantics, as well as existentially as a syntactic constraint, removing this feature entirely will cause our first-pass grammar to over-generate unnecessarily.

For the first pass, counter to what many linguists might at first expect, we do not want constraints such as these to serve multiple uses, because we wish to remove the uses which are computationally expensive for us. So rather than simply remove both equations altogether, we will remove the structure-building aspect, but keep the (non-) existential check.

We do this through the introduction of a *wildcard* feature value [125], which causes unification equations which reference substructures of the wild-carded slot to succeed. Existential checks ending at this slot will succeed, and non-existential tests will fail. This behavior allows us to separate the existential-tests unification equations from the structure-building ones, and perform the former while deferring the latter.

Unification Optimizer In addition to simply modifying or removing unification statements from the grammar, as part of the process of creating the shallow first-pass grammar, we also have found it useful to optimize both the derived (first-pass) and original (second-pass) unification grammars.

The main optimization which we apply to the original unification grammar is *conditional OR unwinding*. This technique allows us to rewrite conditional ORs, which are expensive to correctly evaluate, into unconditional ORs, which are much less expensive.

As we have discussed above, applying restrictions to the original grammar in order to create the shallow unification subset grammar removes some unification statements and substantially modifies others. In order to cope with the results of applying these restrictions, the unification optimizer applies the following additional optimizations to the shallow grammar:

- empty-statement removal.
- single-term OR and EOR weakening.
- wild-carded parallel path removal in disjunctions.

These techniques will be explained in considerable detail in Chapter 7.

5.3.2 Run-time Processing

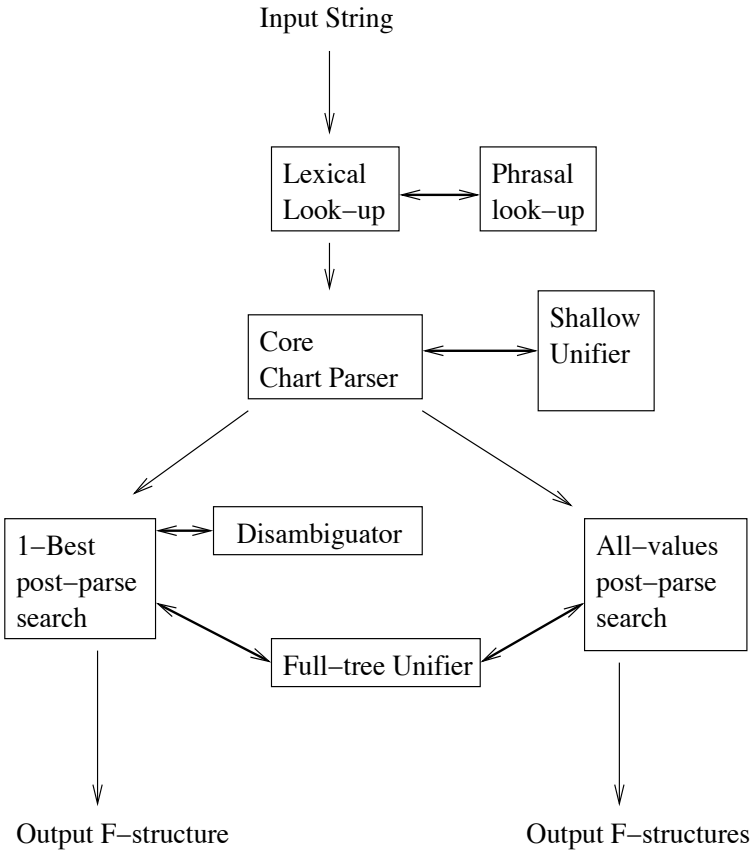


Figure 5.3: The Parser

The run-time processing uses the information distilled in the preprocessing phase to actually parse input strings, creating representations of a structural analysis of these inputs.

Figure 5.3 is a block diagram of the major pieces of the run-time system. This is a three-stage process; each of the stages performing a clear task. For experimental purposes, several performance characteristics are monitored separately for each stage of processing.

1. The input string is separated into "words", these words are looked up, and their lexical F-structures are saved. These words are also assembled into phrases, which are also looked up.

2. The chart parser is initialized with these lexical items, and a parse is performed to find a covering parse forest, using a context-free backbone with interleaved unification.

3. The resulting parse forest is then searched, and the full F-structure value or values are com-

puted. This can be done in one of two ways:

- 1-best F-structure search
- all-F-structures search

Both of these use the full-tree unifier. The all-values search is a simple depth-first traversal of the entire parse forest. The single value search is implemented as a recursive heuristic best-first search, in which the costs which drive the search are calculated via a disambiguation weighting system.

Lexical Look-up

Following the design of the KANTOO system [113], we separated the lexical and phrasal look-up from the main parse. Before actual parsing of an input, the input string is separated into words, and each of the words are looked up in order to retrieve all possible syntactic categories for these words along with their basic unification values (*i.e.* F-structures).

This look-up is performed for each entire input sentence prior to parsing that sentence. This is done so that the look-ahead information (described in Chapter 6) can be correctly calculated and used.

Lexical look-up for individual words is a straight-forward process. In the preprocessing phase, every F-structure and syntactic category corresponding to each unique sense of every word known to the system is associated with the word, so that all of the senses of a particular word can be retrieved when this word is encountered in the input to the parsing system. In addition, some fairly simple root, prefix, and suffix morphology is performed so that different tenses of words do not require completely separate entries.

In addition to lexical look-up for individual words, the system also performs a lexical look-up on phrasal, or composite entries [3, 110]. Phrases are stored and retrieved in a *trie* data structure [132, pp. 248–252], keyed on the root-form of each entry in sequence. Phrasal look-up is left-to-right breadth-first process: a number of positions within the phrase trie are kept; for each word in turn, the root-position is added to the set of positions (corresponding to the theory that a phrase could begin at this position), then an attempt is made to advance every trie position by the lexical root of the current word. Every trie-position which cannot be advanced is dropped; every one which does advance to another interior trie node is kept, and for every one which reaches a leaf of the trie (indicating that a phrase has been found), the phrase, syntactic category, and base unification value is retrieved. (Note that it is possible to reach both an interior node *and* a leaf in the trie, when one

phrase is a left-prefix of another. For example, "oil pressure signal" is both a valid phrase and a prefix of "oil pressure signal voltage".)

Because of the left-prefix compression of the trie, phrasal lookup is at most an $O(p \cdot n^2)$ process (for n words, where p is the maximum number of distinct phrases for some identical set of words), because at each position k there can be at most k trie positions to keep track of at any one time (corresponding to phrases starting at each of the $1 \ldots k$ starting positions).

In the recent ancestors to this system, the process of finding and assembling a phrasal entry was done from within the grammar itself, though the use of grammar call-outs. While expedient, this has two major problems. First, it simply reduces the coherence of the syntactic part of the system. Second, and much more damaging, it can basically eliminate any benefit from look-ahead filtering in the grammar (see below). Because in principle any word could end any phrase, and there are phrasal versions of virtually every lexical syntactic category, any lexical category could appear adjacent to any other lexical category. Thus, when in the middle of processing an input, knowing what the possible categories of the next adjacent item are is of no particular help in ruling out partial grammatical productions which could not be completed with the current input. The result is that including phrasal look-up in the grammar removes any left-corner and look-ahead constraints, by allowing any terminal category to appear anywhere.

The principle reason we chose to separate lexical and phrasal look-up from syntactic processing was in order to better take advantage of look-ahead filtering. After making this separation, the system design benefits of this architectural separation became clear.

The (First Pass) Backbone Parser

The main task of the backbone parser is to discover a *packed parse forest* representing a superset of the union of all possible individual parse trees.

The choice of methods employed by the backbone parser is largely independent from the rest of the system. So long as the parser takes in words, and creates a parse forest, the other parts of the over-all system will work without modification. That is to say, we can parse using any algorithm so long as we get a parse forest in the end.

Input The inputs to the backbone parser are a *grammar*, compiled into some form, and a sequence of *input words*. The grammar is simply a combination of the tree-structured context-free grammar rules, compiled functions implementing the collections of unification rules, and the tables representing the left-corner, look-ahead, and partial ordering relations. We assume that the grammar will not

change from run to run, but rather will be loaded once and used.

The sequence of input words is just that – the actual input. The main requirement is that the format of the input words matches that of the lexical and phrasal storage, so that they can be looked up.

Output There are two possible outputs from the backbone parser. If the parsing system is running in one-pass mode, then the expected output is a representation of the unification values associated with the top-most nodes in the parse tree, if the parse was successful.

If the parsing system is running in two-pass mode, then the backbone parser is expected to produce a packed parse forest, which is a representation of all possible parse trees which cover the input. This parse forest is then *searched* in the second pass to discover one or more final unification values.

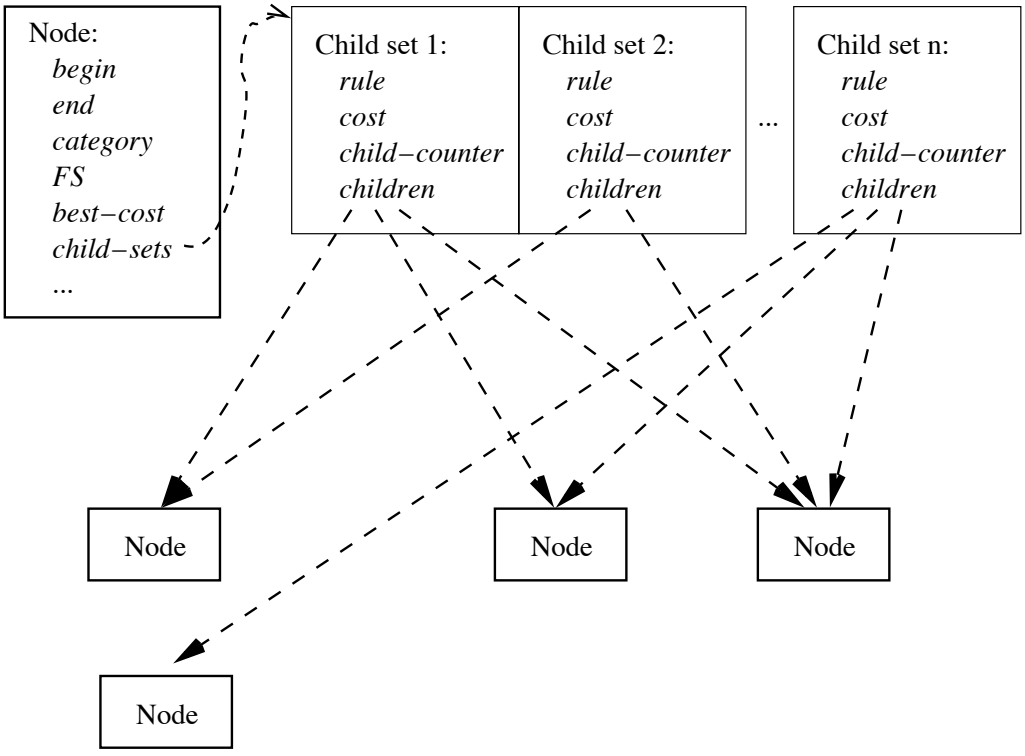


Figure 5.4: A shared packed node structure.

The Parse Forest A parse forest is simply a generalization of a parse tree [151, 15, 23]. In this system, the parse forest is composed of *nodes*; each node may either be a leaf, representing a lexical item, or an internal node. Internal nodes contain a set of one or more sets of children, each of these

sets describes an ordered sequence of child nodes. When a node has more than one set of children, it is said to be a packed node. Figure 5.4 schematically shows an example packed node. (This is a more complete version of Figure 3.1; note that this parse forest representation does not speak to the issue of associating unification values with pares tree nodes.)

Parser Technology One of our original goals is high performance. In order to reach that goal, we will use

- A Chart data structure. Moore [94] presented considerable evidence that a chart parser can be at least as fast as an L-R parser, and in many cases much more so due to the extremely large potential size of the L-R parse tables.
- Optimal agenda ordering. Lavie and Rosé [76] discovered that by carefully ordering the sequence in which chart parser agenda items are processed, one can optimally pack together these agenda items before they are used, avoiding many inefficiency problems (*e.g.* as found in [115]). In addition, optimal packing implies minimal handling, and thus improved performance.
- Interleaved Unification. As discovered by *e.g.* Lavie [68], Oepen and Carroll [115], and contrary to Maxwell and Kaplan [85], we have observed that interleaved unification can be used as an effective filter to prevent the creation of structures which are legal according to the Context-free backbone of the grammar, but which are ruled out by some unification constraint. We allow for interleaved unification, though it is not required.
- Packing of both nodes and f-structures. Packing of parse forest nodes has been shown to be necessary for achieving high-performance in quite a variety of systems [151, 4, 23]. In addition to the parse nodes, when employing interleaved unification it is also necessary to pack the unification values [92, 115]. We extend on previous techniques by allowing for both exact and *approximate* packing of the unification values.
- Tree-structured grammar. The common implementation of a chart parser makes inefficient use of the grammar by failing to take advantage of common prefixes in the right-hand-sides of the context-free rules [94, 42, 134, 103]. We arrange to reduce the grammar into a left-prefix-compressed tree, in the same way as Griffiths and Petrick did [42]. Our implementation differs, however, by using this data structure directly rather than as an intermediate step in the preprocessing of the grammar.

- Filtering. We use a variety of techniques to filter out, or avoid building, grammar productions which could not participate in a complete analysis of each particular input sentence. We use left-corner filtering [42, 127, 119, 58, 83] of the Cocke-Schwartz variant [41, 134], and employ a novel technique to use this filtering with the tree-structured grammar. We also use look-ahead restrictions [2, 160, 1, 102, 94], and combine these two together, applying a left-corner-of-look-ahead constraint as well. In addition, we also use an end-point restriction proposed by Carroll [23].

We have arranged for considerable flexibility in the implementation of the backbone parser, largely hiding the implementation details of this component from other components in the overall system. This allows us to modify the backbone parser component to take advantage of other context-free parsing techniques as they become available.

The Post-Parse Search

The role of the post-parse search is to search through a packed parse forest in order to discover one or more unification values corresponding to complete parses. The packed parse forest which was the output of the backbone parser forms the input to this component; the output of this component is this collection of unification values.

This system contains two different post-parse searching components, one simple and one considerably more complex. The simple post-parse search performs a complete depth-first search of the parse forest in order to calculate the set of all possible unification values. We will see that performing a two-pass parse using the all-values search is slightly faster than performing a one-pass parse, but that both suffer from the same type of exponential blow-up in the face of bad ambiguity.

In order to attempt to avoid the ambiguity problems, we use a best-single-value search. The main goal of this search is to find the *best* parse which *fully unifies*. For this search, *best* means the parse that is most preferred by the disambiguator. This is not as simple a process as one might first think because the backbone parser might give us a parse forest which describes a superset of all fully-unifying parses. In other words, the best parse tree described by the parse forest might not fully unify.

The second goal for the best-single-value search is that it should not search the entire parse forest unless absolutely necessary. This is because the parse forest can (and often does) describe an exponentially-large number of possible parses; thus searching though all of them will take an exponentially large amount of time. On the other hand, in a parse forest describing many possible parses, most of the parse forest will *not* take part in the best parse. Of course we wish to avoid

spending time calculating unification values for parts of the parse forest which we will not ultimately use.

We accomplish all of these goals though the use of two components. The search is performed using a novel recursively nested branch-and-bound best-first search algorithm, which is based on standard search techniques [128], but custom designed for the particular needs of searching a parse forest. We show that in operation this algorithm is equivalent to the A* search [128, 31], demonstrating that our search is also optimal.

This search uses a disambiguating cost calculator to drive its choice of which potential sub-tree is better, but is otherwise decoupled from this disambiguator. We demonstrate the viability of this approach by creating a disambiguating cost calculator which is linguistically motivated, basing its decisions on local structural assignment. We also sketch the requirements for a statistically-based cost calculator.

5.4 Evaluation during Development

Bentley [11, pp. 32–33] points out that it is virtually always important to measure the performance of a correctly working program before attempting to improve its performance, and that careful profiling should be used as a guide to improving performance.

A good test suite is a essential part of measurement. The test corpora are used to assure consistent results in profiling, allowing the researcher to objectively measure whether some seemingly good idea actually improves performance. In addition, the test corpora can be used to assure the correctness of the system [120, ch. 17–18].

We used constant testing to assure the correctness and effectiveness of each change to this system during its development. We describe the development testing conditions in the remainder of this section.

5.4.1 Development test conditions

The goals of the development testing corpora are three-fold: we wish to assure the correctness of the system, measure its effectiveness on natural data, and measure its effectiveness on particularly difficult "worst-case" conditions. Further, it is important to test and measure in a realistic, rather than "toy" system.

In the remainder of this section, we describe the grammar and test corpora which were primarily

used in the development of this system, and which form the input of the tests reported in this chapter as well as the following three.

The KANT grammar

The KANT machine translation system [20, 89, 110, 111] employs a substantial and heavy-weight grammar. The grammar contains 982 rules (958 unique context-free parts), using 544 distinct context-free symbols in the back-bone of the grammar. There are an average of 4.36 unification equations per rule. The grammar handles a substantial subset of American English, as well as special handling of SGML-based mark-up indicators.

Testing Data: The KANT Regression Suite

Jon Bentley relates an old industry adage in a column of rules-of-thumb for Computer Science: "If a program doesn't work, it doesn't matter how fast it runs; after all, the null program give a wrong answer in no time at all." [13, p. 66] It is generally the case that optimizations to a program cause the modified version to produce the same results as the old unmodified version[1]. In the development of this system, we strove to maintain substantial compatibility with the previous parser [20, 155] as well as correctness of execution.

In order to assure the correctness of the system while making various modifications, we used the KANT "FOATS" regression suite. This is a set of 3237 sentences which exercise virtually all of the KANT grammar rules, created to aid ongoing maintenance of CTE usage examples as described by Kamprath *et al*. [55]. No modification to this system was considered to be completed until the new system would correctly parse all of the sentences of the FOATS test corpus. We found this high threshold of testing was critical in discovering and correcting subtle flaws in our implementation, some of which substantially effected our performance test results. We are sure that without such a good regression test corpus, some number of these flaws would still exist.

Natural Data: The Catalyst Ambiguity Corpus

To gauge our development progress on realistic data, we selected sentences used for regression-testing and investigation of ambiguity-handling of the KANT system [89, 55].

The test set we used for these experiments is 1447 sentences taken from the corpus used by

[1] Some probabilistic modeling systems can be exceptions to this rule-of-thumb.

Mitamura *et al*. in their investigation of automatic disambiguation [90]. These are sentences from actual heavy equipment manuals. Each of these sentences contain some sort of structural ambiguity, and were chosen primarily to investigate the effects of ambiguity on parsing. The sentences range in length from 5 to 40 words (26133 words total), with an average length of just over 18 words per sentence.

Artificial Data: PP Attachment

In order to judge worst-case behavior, we also examined an interesting artificial example, constructed to force a prepositional-phrase attachment ambiguity. The following is a close variant of an example used by Moore and Alshawi [95], as well as Oepen and Carroll [115].

```
Grease flows through the bearing seal (in the engine)*.
```

For this sentence, it is interesting to examine the relationship between one of several measures of parser efficiency and the number of repetitions of the prepositional phrase "in the engine".

5.4.2 Development hardware

All of the tests described in this document were conducted on an Apple Macintosh (R) PowerBook (R) G3-series ("WallStreet"), with a 466 MHz G3 processor and 288 MB of main memory, running MacOS version 9.1. The parser was compiled and run using Macintosh Common Lisp version 4.2, with options `(speed 3) (safety 3) (space 0) (compilation-speed 0)`.

5.4.3 Run-time Performance and Optimization Priorities

In the following chapters, we will consider a variety of design choices, and use empirical measurements to guide our choices. But before we examine the details of the engineering of this system, we need to examine a high-level (if crude) profile of the system.

Table 5.1 summarizes the run-time performance of the system running in various configurations on a corpus of natural sentences. Although we can see several interesting things from this summary of results, one stands out from a high-level design point of view: the post-parse search takes *substantially* less time than the backbone parse when a shallow interleaved unification grammar is used in the backbone parser. In this latter condition, the post-parse search takes only 14% of the time required by the backbone parse.

Search type	lexical look-up (CPU secs.)	parse time	post-parse search time	Total time
Fully-Interleaved Unification	13.94	68.35	*n/a*	82.29
Context-Free parse with all-paths search	14.09	87.40	18.99	120.48
Context-Free parse with single-best search	14.11	87.36	56.30	157.77
Shallow Unification with all-paths search	13.81	58.90	**8.82**	81.53
Shallow Unification with single-best search	13.84	58.89	**8.45**	81.18

Table 5.1: Break-down of total run times for Interleaved Unification, Context-free (*i.e.* no unification) plus All-Paths Post-Parse Search, Context-free plus Single-best Post-Parse Search, Shallow Unification plus All-Paths Post-Parse Search, and Shallow Unification plus Single-best Post-Parse Search, on the Catalyst ambiguity corpus (see Section 5.4). Variations in lexical look-up times and parse times for identical parse configurations are due to sampling noise.

The implication of this is fairly clear: worrying about reducing time in the back-bone parser is (within limits) more important than reducing time in the post-parse search. Although there are potential techniques which could improve the post-parse search further, such improvements can be seen to be of strictly secondary importance in light of these measurements.

5.5 Summary

Our system implements all of the requirements presented above in Section 5.2. The context-free parser that is the core of the first-pass search supports many of the best techniques for increasing performance, including unification (both heavy- and light-weight), packing of nodes, left-corner and look-ahead constraints, a tree-structured grammar, and optimal agenda ordering.

Our system also supports shallow unification with approximate unification-value packing, and collects some information needed for the disambiguator. The second-pass search supports both best-first-search with disambiguation and breadth-first search for all possible unification values. The unification equations used in the first and second passes are independent, allowing for considerable flexibility in the first pass.

Among these features, the use of a tree-structured grammar, the unification optimizer, approximate unification packing, and the best-value post-parse-search algorithm are novel to this system. In addition, the realization that the unification values obtained in the first pass need not be used directly to calculate the final unification result, and that as a consequence we may make certain approximations in the first pass is also novel to this system.

The unification results of the this parsing system have been verified against other similar known-good systems in order to assure correct execution. This verification included successfully running the KANT "FOATS" standard regression tests, and hand-verifying that the results obtained were correct and complete.

The following three chapters describe the major components of the implementation in detail:

- Chapter 6 describes the context-free parser part of the system, including implementations of the tree grammar, left-corner and look-ahead filtering, and how these interact and must be harmonized.
- Chapter 7 describes the unification system, including the pseudo-unifier and its optimizing compiler, and the mechanisms used to calculate the shallow unification grammar from out of the original grammar, as well as the approximate packing of unification values.
- Chapter 8 describes the two variations on post-parse search, including a description of the nested branch-and-bound search, its proof of optimality, and a description of the disambiguation cost calculator.

Following these chapters, experimental tests of the system are examined and conclusions are drawn from these results. Finally, further directions for research are considered.

Chapter 6

Efficient Chart Parsing

> UNLESS someone like you
> cares a whole awful lot,
> nothing is going to get better.
> It's not.
>
> — *The Lorax* [148]

6.1 Motivation

As we have noted in Chapter 1, many large-scale natural language systems include the user of context-free-spine based unification parsers. The larger system for which this parser was designed to fit into works in just such a manner.

The KANT system [87, 111] originally used Tomita's "Augmented-Context-Free" GLR parser described in [152]. This parser was enhanced to include an interleaved pseudo-unification [153] system as part of the context-free parser.

As we have stated in Section 3.1.2, we wish to maintain substantial compatibility with this previous parser. That does not mean we need completely limit ourselves to interleaved unification, as Tomita's Augmented GLR used, but that choice is the subject of future chapters. In any case we know that a high-performance context-free parser will be an integral part of the overall parsing system; investigating how to construct one is the topic of this chapter.

Goal In general, what want in the context-free portion of our parser is a context-free parsing algorithm and implementation that runs quickly, while creating the least number of chart entries

(e.g. inactive edges) required.

The needs of interleaved unification also require that the parser be able to recover the particular child entries that allow for the creation of any new parent entry, before it has been fully created, so that the unification equations associated with the relevant rule can be evaluated against the unification values associated with the children. (While this is simply a particular sub-case of the requirement that we be able to recover the set of all valid parses, it is a design consideration when implementing an interleaved unification parser.)

6.1.1 Chapter Outline

In order to better understand the problem and what other researchers have done, the next section contains a brief review of some of the history of context free parsing, especially as applied to the processing of natural language. This section is mostly for the reader who is not already familiar with the details of NL processing.

Section 6.3 introduces the *Tree-Structured Grammar*. We will see that by compressing the grammar into a left-prefix tree, we can gain substantial parsing efficiency.

In Section 6.4 we consider left-corner and look-ahead filtering techniques, and their effect on the parser, and the special considerations that result from combining the tree-structured grammar with left-corner filtering.

In Section 6.5 we will briefly touch on interleaved unification (as it effects the context-free parser), as well as packing and an end-point restriction, and then discuss the algorithm for the combination of all of these features.

Finally, Section 6.6 contains a full set of results comparing the contributions on each feature upon the whole context-free parser. The best measured combination of features is then selected as the basis of the remainder of this work.

6.2 Prior Context-Free Parsers

The earliest work on automatic parsing concentrated on the Context Free class of languages. There are basically three major ways to recognize and parse a context free language: searching *top-down* from the starting symbol for a sequence that produces the input string, building partial parses of the input into a complete parse starting with the input string and working *bottom-up*, and some combination of the two, either adding top-down constraints to a bottom-up parser or vice-versa [3,

108, 104, 125, 48].

There is also a choice between interpreting the grammar directly [163, 34, 58] and compiling the input grammar by transforming it into a *pushdown automaton* and running the result on the input [48, 1, 151].

Top-down parsers

One straight-forward way to write a parser is to use information from the grammar to predict structure, and thus guide the process [3, pp. 47–53]. A top-down parser starts by predicting the start symbol, and then recursively expands this symbol in an attempt to discover a sequence of symbols that correspond to the input string. Viewed in this way, parsing is simply another search task — an abstraction that has allowed researchers in the field of Artificial Intelligence to solve many seemingly different problems.

There are two main problems to a strictly top-down approach. The first is that the search can often spend a lot of computational time proposing and then eliminating predictions that do not even closely match the input (*e.g.* the failures at steps 2 and 8 in the above example). The second is that if we give such a parser an ungrammatical input (according to the supplied grammar), the top-down parser will give us nothing useful at all. It is frequently useful to get *some* partial analysis even if a complete parse is not possible [68].

Bottom-up parsers

Cocke, Kasami, and Younger independently suggested a different approach [48, p. 145]. Rather than trying to use the grammar to predict what the input might look like, building from the top downward, they proposed a dynamic programming solution that starts with the input and builds a matrix of constituent occurrences in a bottom-up fashion. In Younger's solution, this "*recognition matrix*" of occurrences is then used to build a "*parsing matrix*" of constituents actually used in analyses that cover the complete input. This second step is performed in a top-down manner [163].

The major advantage to these algorithms is that they run in $O(n^3)$ time for any context-free grammar in *Chomsky normal form*[163, 25, 48]: every rule must either be composed of two daughter non-terminal constituents, *e.g.* $S \leftarrow AB$, or a single terminal constituent, *e.g.* $A \leftarrow 0$ [163, p. 192]. Younger showed that both the recognition step, where the "*recognition matrix*" of occurrences is built, and the parsing step, where the "*parsing matrix*" is built, run in $O(n^3)$.

The major disadvantage of these types of algorithms is that their fundamental data structures

are a matrix of binary occurrences. For an input of 50 words long and a grammar containing 500 symbols, such as we will use below, this algorithm requires two matrices of 1,250,000 bits each. While not impractical on current hardware, this is rather a lot of overhead. Also, most of the entries in these matrices will be zeros, so this may not be the most efficient way to deal with the data.

Chart parsers

In order to gain the best of both worlds, Earley devised a set of algorithms now commonly known as 'Chart parsers' [34]. This method uses a data structure S_i to hold both 'final' and non-final states, and an iteration over the states predicting new non-final states (arcs) to start, completing non-final states with final states to create new final states, and extending non-final states with final states to create new non-final states. Earley proved that for the recognition case, his algorithm has a time complexity of $O(n^3)$ [34, pp. 98–99].

Some twelve years after Earley, Kay [58] proposed a family of algorithms that are quite similar to those proposed by Earley. In Kay's formulation, the 'chart' structure is used to hold information about edges (states) that have been found (corresponding to Earley's S_i). When an edge can be extended, the new extension is placed into an 'agenda' structure, to be processed at an appropriate time. Kay's innovation is two-fold: first, placing work still to be done in an agenda turns out to have many advantages, notably the ability to control *which* pending item to work on next; we will see below that this is important. Also, Kay's proposal included a form of left-corner look-ahead, which is also important.

In the final analysis, it is fair to say that the modern 'Chart Parser' is really a combination of ideas from these two papers. The basic algorithm, described in terms of a *predictor*, *completer*, and *scanner* components came from Early [34, p. 97], whereas the names 'chart', 'agenda', and the left-corner look-ahead came from Kay [58].

The bottom-up approach solves the problem of creating something useful in the face of ungrammatical input. On the other hand, it still has some of the problems of spending time starting rules that cannot finish. These aren't as bad for bottom-up parsing as for top-down, however, since the only wasted time and space is in the creation of of the unfinishable active rule. This is much less expensive than in the top-down approach since the active rule can be allowed to languish, rather than being forced to explore the rule and discover its failure followed by enough back-tracking to continue from a useful point. Still, the expense of creating a useless active rule is non-trivial, and worth trying to minimize.

GLR (Tomita)

To solve the context-free parsing problem, Tomita took a different approach in creating the Generalized L-R parser [151]. GLR is basically a pseudo-nondeterministic implementation of a shift-reduce-style nondeterministic finite state automaton parser.

Chomsky, and separately Evey, showed that any context free language could be recognized by a push-down automaton [48, p. 123]; for details see also Aho and Ullman [2, p. 198]. A shift-reduce parser is basically an implementation of this. The context free grammar is compiled into a state-action table that instructs a push-down automaton how to process words (*i.e.* symbols) of input. The available actions are to:

- *shift* a new symbol onto the stack
- *reduce* a set of symbols currently on the stack according to a particular grammar rule
- *accept* the input (as being in the grammar).
- *reject* the input (as being outside of the grammar).

LR parsers were primarily developed to parse computer programming languages, where ambiguities are generally either flaws in language design or resolvable in advance of parsing by some general rule. There are two types of ambiguities that can appear:

- *shift*/*reduce* ambiguities. These are due to an ambiguous choice between two different rules, one of which can be reduced right away, the other of which contains some topmost part of the stack as a prefix to the right-hand-side of this rule. One way to trigger this condition is to have two rules in the grammar, the right-hand-side of one of which is a prefix of the other, for example:

$$\begin{array}{lcll} VP & \leftarrow & V & \\ VP & \leftarrow & V & NP \end{array}$$

  In this case, after discovering a *V*, we (might) have an ambiguous choice between shifting to the state that expects an *NP*, and reducing the *V* to a *VP*.

- *reduce*/*reduce* ambiguities. These are due to an ambiguous choice between two different rules that can both be reduced according to the current stack. One way to get into this condition is to have two different rules in the grammar with identical right-hand-sides:

$$\begin{array}{lcll} S & \leftarrow & V & NP \\ VP & \leftarrow & V & NP \end{array}$$

For these rules, any time a VP is recognized, followed by an NP, the parser will have an ambiguous choice between reducing an *S* and reducing a *VP*.

It would be an oversimplification to say that these different ambiguities result from particular types of ambiguous rules in the grammar, since fairly complex sets of rules can result in seemingly simple ambiguities [2, 48].

Shift-reduce parsers have the advantage that they can be very, very fast on unambiguous grammars. If a grammar G is a member of the class $LR(1)$, then by definition it can be compiled into an LR table, employing one symbol of look-ahead, without any of the conflicts described just above. When a grammar can be so compiled, a shift-reduce parser will run in linear time [2]. The possibility of applying this sort of speed to natural language provided strong motivation for Tomita to adapt this method.

The main obstacle to this is that the LR subset of the Context Free class of languages does not permit ambiguity in the grammar [2, p. 225]. The obvious problem is in the push-down automaton: if there are a choice of actions for some inputs, then the notion of *the* stack only makes sense in a nondeterministic sense. Thus Tomita's idea was to nondeterministically follow all choices in parallel, with a set of stacks. So when faced with, say, choosing between a shift and a reduce, the GLR parser splits the current theory into two separate ones: one to follow the shift choice, and one to follow the reduce.

It was not enough to simply duplicate a theory when splitting it, however, as this led to exponential run-time in the face of ambiguity [151, pp. 14–16]. To address this, Tomita developed the graph-structured stack: the set of stacks is represented by a directed acyclic graph, in which local ambiguities could be represented by local splits in the graph which rejoin at a later point.

Johnson [53] showed that there exist grammars for which Tomita's GLR algorithm can be made to require $O(\exp(n))$ time to run. Addressing this problem, Kipps [61] and Carroll [23, pp. 91–108] each showed that with some modifications, variants of the GLR algorithm can be shown to run in $O(n^3)$ time for all context free grammars. Kipps also demonstrated experimental evidence that his modifications lead to parser that is a faster than Tomita's, though one that requires more space [61, p. 54–56].

GLR* (Lavie)

For GLR*, Lavie extended Tomita's GLR algorithm to better handle spoken language by making it robust to ill-formed input [68]. Much of the robustness of GLR* comes from its ability to skip over words in the input, allowing it to avoid many of the problems in input data that is the output from a speech recognition system.

In order to make the search feasible, Lavie applied several techniques; may of which are interesting simply from a performance standpoint (irrespective of robustness in parsing). He applied a Beam Search, to prune out less likely productions in the middle of the parse. To make this work, he also applied a trainable distribution on ambiguous actions within the shift-reduce table.

He made further use of the distribution on actions by using it for statistical disambiguation. He showed that his method of disambiguation was considerably superior to a (simple) principled approach. This is a very intriguing idea; methods of adapting this approach to a chart parser could be well worth considering.

He noted that with unification grammars, it is important to detect local ambiguities before packing these ambiguities together. We will consider the subtlety of this point in some detail below.

The connection between Chart and LR parsers

In the above description of the GLR parser, we did not mention anything specific about how grammar rules were recorded when performing a *reduce* action. There are many ways one could imagine recording such information; the chart data structure is a natural choice. Kipps showed that the GLR technique can be made polynomial ($O(n^3)$) all the time with the addition of a dynamic data structure, which he calls `ANCESTORS`, which bears a remarkable similarity to a chart [61].

In Lavie's thesis [68], an examination of the example implementation of GLR* reveals a similar technique: when productions are completed, the results are recorded in a chart-like structure [68, pp. 144-166],[70].

The similarity between chart and LR parsers can be taken further. van Noord [107] noted the similarity of bottom-up chart and LR parsers.

Superiority of Chart parsers for this application

Contemporaneously with much of the work presented in this chapter, Moore [94] presented a set of experiments investigating the effects of a large number of left-corner, look-ahead, and factoring

optimizations on a set of grammars spanning three orders of magnitude in size.

Moore shows that while these techniques are all advances to context-free parsing, they are not fully independent, but instead interact with each other. Often they reinforce each other, but sometimes they work at cross-purposes, with a combination of particular techniques resulting in *slower* parsing than using either alone.

He also compared these variants of Chart parsers to the GLR algorithm [151], and found that while GLR could be faster in some cases, it was in fact slower in others, and in some cases the LR table was so large that it could not be adequately tested.

On the basis of early investigations of our own, we chose to base our parser on the Chart framework. Happily, Moore's results help confirm our choice as reasonable.

6.3 The Tree-Structured Grammar

Grammar structure optimizations

There are several ways to improve the efficiency of a context-free parser. Not only can one improve the efficiency of a parser by constraining the search according to left-child and adjacency information gathered from the grammar, one can also improve the efficiency of how the rules of the grammar are processed.

As Griffiths and Petrick [42], Shann [134], and Nederhof [103] have all observed, one basic shortcoming of simple table parsers are that they make inefficient use of the grammar whenever there are grammar rules with common prefixes. Aho and Ullman also point out that this can be a fatal flaw in recursive-descent parsers [2, pp. 180–182]. That the grammar is used inefficiently is as true for a simple chart parser [34, 58, 3, 107] as for other types of table-based parsers.

To see how to make better use of the structure of the grammar in our chart parser, we should first review the problem. The following explanation starts with a detailed expansion of the observation made by Griffiths and Petrick [42, p. 295].

Consider the following context-free rules:

$$
\begin{array}{lcllll}
S & \leftarrow & NP & VP & & (1) \\
NP & \leftarrow & NP & PP & & (2) \\
NP & \leftarrow & Det & N & & (3) \\
N & \leftarrow & N & Comma & Conj \quad N & (4) \\
N & \leftarrow & N & Comma & N & (5) \\
NP & \leftarrow & N & & & (6) \\
N & \leftarrow & N & Conj & N & (7) \\
\ldots & & & & &
\end{array}
$$

One of the interesting features of this set of rules is that there is a lot of redundancy in the prefixes of the right-hand-sides, and that this structure can be rearranged into a tree structure to eliminate this redundancy.

Unfortunately, a naïve implementation of a chart parser will not take advantage of this redundancy. Consider what would happen inside a "standard" chart parser using this grammar fragment. Suppose that the first input token was an N; then the parser would start active arcs for rules (4), (5), (6), and (7) above. Further suppose that the second input token was a Comma; then the parser would create active arcs to advance the active arcs from rules (4) and (5) that we created before.

In contrast, a shift-reduce parser [48, 151, 23, 107] will often use a grammar that has been optimized to eliminate this redundancy [48]. Since chart parsing and shift-reduce parsing are substantially similar [107], many techniques used in shift-reduce parsing can be applied to a chart parser.

Consider this grammar once again from the stand-point of a Push-Down Automaton (PDA) [48]. We will call reducing a sequence of symbols and producing a left-hand-side symbol a *reduction*, and we will call traversing a right-hand-side symbol a *shift*. (The obvious motivation for these particular terms is their use in the shift-reduce parser literature [48].)

$$
\begin{array}{llllcll}
NP & VP & & & \rightarrow & S & (1) \\
NP & PP & & & \rightarrow & NP & (2) \\
Det & N & & & \rightarrow & NP & (3) \\
N & Comma & Conj & N & \rightarrow & N & (4) \\
N & Comma & N & & \rightarrow & N & (5) \\
N & Conj & N & & \rightarrow & N & (7) \\
N & & & & \rightarrow & NP & (6) \\
\ldots & & & & & &
\end{array}
$$

Most shift-reduce parsers include a step that combines together multiple equivalent shift or

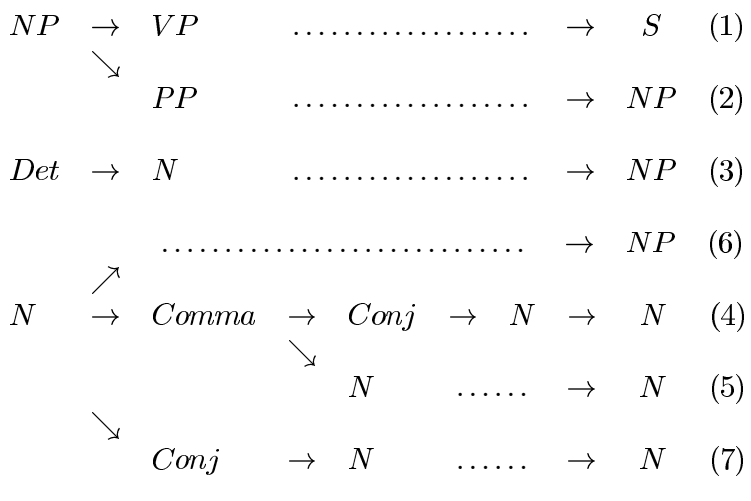


Figure 6.1: An example Tree-structured Grammar

---

reduce rules that appear in parallel. Consider what happens when we combine the left sides of the right-hand-sides (or shifts) of our example grammar. We will use a downward- (upward-) diagonal arrow $\searrow$ ($\nearrow$) to indicate that the RHS up to this point comes from the previous (next) rule. The result of this re-drawing is shown in Figure 6.1.

Griffiths and Petrick refer to the structure shown in Figure 6.1 as a "linked-list (tree) representation" of a set of rules [42, p. 295].

Obviously fewer symbols appear in this grammar. Its structure is also quite indicative of how it will be used — active arcs will indicate places on the grammar tree, rather than (rule, offset) tuples.

6.3.1 Building a Tree-Structured Grammar

Properly, the grammar shown in Figure 6.1 is structured into a *trie* data structure [132, pp. 248–252]. Each node in the trie contains a `symbol`, and two sets: the `shifts` and the `reductions`. The shifts are a set of (pointers to) trie nodes; the reductions are a set of rules (possibly containing other useful information such as unification equations).

We build the grammar trie by "tree-ifying," or left-prefix compressing, the first rule in the grammar by producing a list of trie nodes (linked through the `shifts` field), with a final node containing an entry in the `reductions` field pointing to the original rule. The algorithm used for this com-

```
tree-ify (rule, rhs)
    new-t = new node
    new-t.symbol = first(rhs)
    if length(rhs) == 1
        new-t.reductions = rule
    else
        new-t.shifts = tree-ify (rule, rest(rhs))
    return new-t

add-to-tree (rule, rhs, t-node)
    if lhs is empty
      push rule onto t-node.reductions
    else if (fist(rhs) == n.symbol) for some n in t-node.shifts
      recursively call add-to-tree (rule, rest(rhs), n)
    else push tree-ify (rule, rhs) onto t-node.shifts

build-tree (rule-list)
    tree = new node
    tree.shifts = tree-ify (first(rule-list), first(rule-list).rhs)
    foreach r in rest(rule-list)
        add-to-tree (r, r.rhs, tree)
    return tree
```

Figure 6.2: Method to Left-Prefix-Compress a Grammar

---

pression is outlined in Figure 6.2. [1]

6.3.2 Using a Tree-Structured Grammar

Once the grammar tree has been built, applying it in the chart-parser is quite straightforward. The algorithm presented below is, at its essence, a chart parser. The principle difference between the algorithm below and one presented in *e.g.* Allen [3, p. 55] is that in the classic formulation of the chart algorithm, for each extendible active arc, *a* new active arc is added, and for each completable active arc, *a* new constituent is added to the agenda. In contrast, when using the tree-grammar *several* new active arcs and *several* new constituents may be created from a single extendible active arc.

The other notable difference is in the representation of the children of an active arc. We represent the children of an active arc with a *reversed-order* linked list in the `traceback` field of the arc. Because one active arc could spawn several arcs in turn *representing different rules*, the addition of a new child on a successfully extended arc must not disturb the children of any sister arcs. As noted

[1] The pseudo-code in this paper is in a vaguely C/C++/Java-like notation. An element of a structure is denoted by `structure.element`; lists are assumed to have a linked list structure, accessed by the operators first() and rest().

by Carroll [23, p. 55], an efficient way to handle this is to simply build the child set as an up-tree [27], and then trace-back the list to build the final set of children should the arc successfully finish.

With this in mind, the basic tree-structured grammar chart parser algorithm shown in Figure 6.3 is quite straight-forward.

Implementation note The root of the grammar tree contains only shift actions (assuming that there are no ϵ-reductions), and these are searched in the `rules-started-by` and `single-rules-completed-by` functions. The speed of the parser can be substantially improved by instead putting these initial grammar-tree entries in a table so that these two functions are unit-time operations.

6.3.3 Previous Approaches

While the details of our implementation of the tree-structured grammar, as well as the combination of this optimization and left-corner filtering (described below in Section 6.4) are unique, the general idea of removing this redundancy in the grammar is by no means new.

Griffiths and Petrick Griffiths and Petrick seem to be the first to describe this redundancy and take advantage of it. In their 1965 paper [42], they refer to the structure shown in Figure 6.1 as a "linked-list (tree) representation" of a set of rules [42, p. 295]. Even though they describe this tree structured grammar quite clearly, they chose not to implement a parser directly based on this structure, but instead used the structure to generate a transformed set of rules. Though this seems slightly odd, it is worth pointing out that Griffiths and Petrick's paper predates Early's parser [34] by some five years, and Kay's chart parser [58] by fifteen. The direct implementation that we will pursue uses a formulation of the chart parser, thus not all necessary pieces were in place at the right time.

Shann (1991) More recently, Shann [134] also exploited the tree structure, albeit partially, by compacting together the common left parts of the right-hand-sides of rules. Unfortunately, he chose to do so only for rules that share common left-hand-sides. So in Figure 6.1 above, only rules (4), (5), and (7) would be combined together; (1) could not share prefixes with (2), nor could (6) be combined with the (4,5,7) set. In Section 6.4, we will see below that we need not have such a restriction.

Nederhof (1994) Nederhof [103] suggests that one can fully take advantage of this redundancy, and exhibits several algorithm variants that do so. For strictly context-free parsing, these algorithms are extremely good, removing many redundancies from the processing. All of these algorithms are table parsing algorithms. As such, they are much more closely related to GLR and other compiled table parsing algorithms than to the class of chart parsers (to the extent that chart parsers are different than table parsers, at least).

Unfortunately, however, these optimizations partially loose track of which particular rule created a particular derivation. ('Partially', in the sense that one must do some work to recover the original rule.) So they are still incomplete for our purposes. Fortunately we have seen that this extra bit of book-keeping can be quite simple, due to our changing the base formulation of the algorithms from being table-based to being chart-based.

Moore (2000) Contemporaneously with the work presented in this chapter, Moore [94] presented a set of experiments investigating the effects of a large number of left-corner, look-ahead, and factoring optimizations on a set of grammars spanning three orders of magnitude in size. He compares and contrasts a set of parsers including the improvements proposed by Griffiths & Petrick [42], Shann [134], and Nederhof [102], as well as several others.

For the optimizations to the grammar, Moore follows Shann and Nederhof, transforming the grammar itself. This is perfectly fine for a strictly context-free parser, but we wish to *easily* accommodate sets of unification equations as well. It would involve too much extra work and book-keeping to re-write the grammar for the context-free parse, and then figure out what the original structure was in order to correctly apply unification rules.

6.4 Left-Corner and Look-ahead Filtering

Although the search for a complete parse can be performed in a purely top-down or bottom-up manner, these searches can be improved upon by combining top-down and bottom-up information together, and also by taking advantage of information slightly beyond the span of a single constituent.

We can improve on the efficiency of the parser by restricting the edges that are added to the chart according to restrictions discovered from the structure of the grammar. The two constraints we use are *left-corner* and *look-ahead* constraints.

6.4.1 Left-Corner Constraint

Robert Moore has described the origins of left-corner parsing as "murky" [94]. He notes that the origins of left-corner parsing go back to Griffiths and Petrick [42] and Rosenkrantz and Lewis [127], with additional work by Pratt [119], Kay [58], and Matsumoto [83], among others. A particularly accessible description of this process is presented by Aho, Sethi, & Ullman [1, pp. 188-189].

The idea behind *left-corner* parsing is to start with a bottom-up parser, but to restrict the search in such a way that the parser only searches for and builds constituents that can take part in a parse that covers the entire input. In other words, to add top-down restrictions onto the bottom-up parse.

In order to perform the actual parse, the parser starts by proclaiming that it is interested in building a top-level (*i.e.* S) constituent spanning the entire input. From then on, the parser should not create a constituent (or even start a rule that would create a constituent [93]) unless this constituent can be used in the construction of some larger constituent that is in the process of being built.

This is generally accomplished with the aid of the *left-corner relation*: A is said to be 'in the left-corner' of B either if there exists a rule of the form $B \leftarrow A\beta$ (where β could either be empty or not), or if there is some C such that A is in the left-corner of C and C is in the left-corner of B (*i.e.* the transitive closure of the relation).

This restriction can either be explicitly checked, or partially implicitly encoded into the implementation of an interpretive (*e.g.* chart) parser [108, 104, 125, 94], or compiled into the state-space of an LR (table-driven) parser [48, 1].

In the naïve chart implementation, this check is quite straight-forward. Whenever either a singleton (unit-length) rule is about to be evaluated, or a multiple-constituent rule is about to be started, at starting position s, check to see if the rule's LHS category c is a member of the union of the downward-left-corner sets of all current active arcs which start at s. It is, of course, convenient to precalculate the downward-left-corner relation (known in Aho, Sethi & Ullman as FIRST [1]) during the grammar compilation stage.

Cocke-Schwartz filtering is a variant of the left-corner restriction. In Cocke-Schwartz filtering, the left-corner sets are expanded as they are added to the prediction sets, so that the actual filtering check becomes a simple set-membership operation. Again according to Moore [94], this type of restriction first appeared in Graham, Harrison, and Ruzzo [41], and was used by Shann [134]. (According to Graham, Harrison, and Ruzzo [41, p.456], their restrictor is somewhat more specialized than the Cocke-Schwartz restrictor in that it applies to whole rules.)

"Ordinary" Left-Corner filtering There are two basic ways of implementing a left-corner check. Since we wish to use the best technique, and did not have an *a priori* preference, we implemented both techniques and consider each in turn here. The following description is due to Moore [94].

The essence of the "ordinary" left-corner constraint [94, 42, 127, 1] is to collect together the "next-needed" categories of all the active arcs for each position i, and then before adding an inactive edge to the chart, make sure that some active arc is "looking for it" by checking to make sure that the left-corner-set of one of these active arcs includes the category of the new inactive edge.

We can restate this somewhat more formally:

- Let P_i be the set of all categories for which there is a rule or an active arc starting at i. (Moore calls this the "*prediction set*" [94].)
- On initialization, P_1 is set to S (the grammar start symbol).
- When an arc (or active edge) is added to the chart at $a =< i, \text{cat} >$, set $P_i = P_i \cup \text{cat}$.
- An entry (or inactive edge) $e =< i, \text{cat} >$ is added to the agenda only if $\exists l \in P_i$ s.t. cat $\in$ LC(l).

In our implementation, P is an array of bit-vectors (see below), and the final test is actually implemented as the intersection test $\{P_i \cap \text{UPWARD-LC(cat)}\} \neq \emptyset$. (Just as LC(cat) is the inclusive transitive closure of the first-child relationship, UPWARD-LC(cat) is the inclusive transitive closure of the *parent* relationship.)

Cocke-Schwartz filtering In contrast to ordinary left-corner filtering, the essence of Cocke-Schwartz filtering [41, 134, 94] is that we union together the left-corner *sets* when developing the prediction set, and the filtering test is a (quick) single set-membership test.

More formally, this can be summarized as:

- Let P_i be the union of all left corner sets of categories, for all categories for which there is a rule or an active arc starting at i.
- On initialization, P_1 is set to LC(S) (the grammar start symbol).
- When an arc (or active edge) is added to the chart at $< i, \text{cat} >$, set $P_i = P_i \cup \text{LC(cat)}$.
- An entry (or inactive edge) e with category c is added to the agenda starting at i only if: $c \in P_i$.

Filtering type	*features*	total num. nodes	total num. arcs	parse time (CPU secs.)
Ordinary Left-Corner	LC at leaves only	1,014,451	2,607,842	171.49
	full LC in tree	984,083	2,264,255	265.61
Cocke-Schwartz LC	LC at leaves only	1,014,451	2,607,842	**77.45**
	full LC in tree	984,083	2,264,255	**77.96**

Table 6.1: Comparison of parse times for Ordinary Left-Corner versus Cocke-Schwartz Filtering using the Tree-Structured Grammar

Implementation notes We found that in our implementation of each of these filtering techniques, it is *very* important for the constraint-checking functions to be implemented efficiently. This is particularly true for the left-corner-of-look-ahead constraint (described below). Since arc creation and handling is relatively cheap, any check used to reduce the number of arcs must be less expensive still, or the over-all run-time will go up rather than down.

Our initial implementation used an unsorted-list representation of sets, but this slowed down the parser considerably. We now represent these sets as packed bit-vectors, so that set-membership is a unit-time operation, and set-union, set-intersection, and querying for set-emptiness are all vectorizable bit-parallel operations which are quick in practice [12, p. 146], [78, pp. 257–259], [99, pp. 759—760].

This particular implementation choice is probably quite implementation-language dependent, however, and should be reconsidered for a truly optimal implementation.

Comparison For completeness, we implemented both methods in order to compare the one against the other. Table 6.1 summarizes this comparison. (For test conditions, see Section 5.4 above.) In it we can see that Cocke-Schwartz filtering outperformed the "ordinary" left-corner filtering by almost a factor of two in speed. In Section 6.6 we will consider this choice as well as several others in more detail.

The results shown in Table 6.1 are not surprising when one considers how the filtering is implemented in detail. As Graham, Harrison, and Ruzzo observed [41, p. 456], "ordinary" left-corner filtering requires the least computation to generate the prediction set, and the most computation to test against it, whereas the reverse is true for their method as well as for Cocke-Schwartz filtering. In particular, for "ordinary" left-corner filtering an element must be added to a set for every arc (active edge), but an iteration over a sparse set must be performed to evaluate the restriction before adding a chart entry (inactive edge). On the other hand, for Cocke-Schwartz filtering, a union-of-sets operation must be performed for each arc with a unique next-needed symbol, but this allows

the restriction test to be an inexpensive set-membership operation.

In either case, the whole-set operation is going to be more expensive than the single-item operation, so the choice of where to put the expensive operation should follow from the observation of whether creating an arc (active edge) happens more often than creating a chart entry (inactive edge).

We will see below that with the tree-structured grammar, we create many more inactive edges than active edges, so of course we should chose Cocke-Schwartz filtering since it does more work for each active edge.

Integrating Left-corner constraints into the Grammar Tree There is one potential problem with structuring the grammar into a tree: several techniques, notably including the left-corner constraint proposed by Graham, Harrison, and Ruzzo [41, p. 456], involve evaluating predicates involving the left-hand-sides of proposed or active grammar rules.

In the naïve implementation of the grammar, evaluating these predicates is straight-forward. Taking the example of the left-corner constraint, before "starting" a rule, one can ask whether the rule will be used if it can be completed. This question can be formalized as asking whether the LHS category of the rule c is a member of the (downward) left-corner set for the current position (at offset s). Since we have the rule immediately to hand, computing this is straight-forward.

The situation becomes more complicated when the grammar is structured into a tree. The notion of "the" category that can be derived from a rule becomes the *set* of categories that can be derived from some point in the tree. This somewhat complicates our calculations.

For Cocke-Schwartz filtering, we can re-formulate the left-corner constraint to ask if there is a *non-empty intersection* between the set of all categories that could be completed from the point in the tree corresponding to the category of the current entry and the left-corner set for the current position.

For Ordinary left-corner filtering, we must instead ask if there is a non-empty intersection between any of the left-corner sets of the categories that could be completed from the current point in the tree, and the set of categories of active arcs starting at the current position.

The set of completable categories can be recursively defined as the union of:

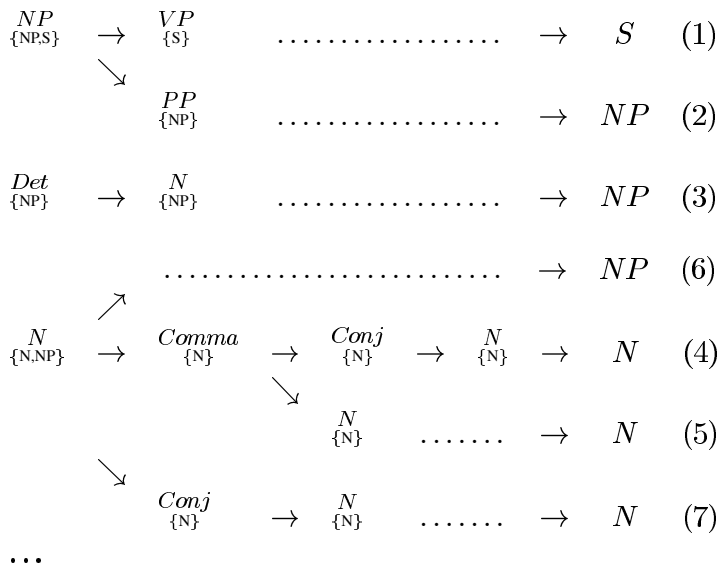
- The categories of all rules in the reduce list of the current node, and
- The union of the sets of completable categories for each node on the shift list of the current node.

Filtering type	*features*	total num. nodes	total num. arcs	parse time (CPU secs.)
Ordinary Left-Corner	LC at leaves only	1,014,451	2,607,842	**171.49**
	full LC in tree	**984,083**	**2,264,255**	265.61
Cocke-Schwartz LC	LC at leaves only	1,014,451	2,607,842	77.45
	full LC in tree	**984,083**	**2,264,255**	77.96

Table 6.2: Comparison of parse times for the Tree-Structured Grammar with left-corner in the middle of the grammar tree ("full LC in tree") versus left-corner only at end (only when creating nodes)

This definition invites the following extension to the left-corner restriction: As one proceeds down a path in the tree, the set of categories that can be derived from the current node in the tree shrinks. With this in mind, one can decorate *each* tree node with the set of all nodes that are completable from that point in the tree, and require the left-corner non-empty intersection constraint to be true for every new active arc. When this extension is applied to the Cocke-Schwartz restriction, the resulting computation can be considered an extension of the restriction proposed by Graham, Harrison, and Ruzzo [41].

If we decorate the tree-structured grammar example we used above, the result is this:



For the actual parser, we compute these sets according to the above definition, decorating the tree nodes with the resulting sets.

Table 6.2 briefly summarizes a comparison of using these sets of reductions within the tree to filter arc (active edge) production versus performing the left-corner constraint only when nodes (inactive edges) are to be created.

From Table 6.2, we can see that when using Cocke-Schwartz filtering, this filtering on the set of reductions reduces the number of both arcs and nodes created. Unfortunately, because this reduction is slight, the parse time is not improved.

On the other hand, for ordinary left-corner filtering, filtering on the reduction set introduces more work than it avoids. So while fewer arcs and nodes are created, the result is a parser that is 35% slower than not using this filtering.

This is quite interesting, because it shows that reducing the number of chart entries is not always the better choice. In this case, we pay a 35% speed penalty in order to reduce the number of chart entries by $3\frac{1}{4}\%$. If we were forced to adopt ordinary left-corner filtering for some other reason, unification would have to be *substantially* slower than chart entry creation (and the associated extra work that having these extra entries would induce) in order to be a net benefit in run-time performance.

Fortunately, we do not have to make that choice. Since the parser using Cocke-Schwartz filtering is over twice as fast as using the parser with ordinary left-corner filtering, we will adopt the Cocke-Schwartz filtering. In the full results in Section 6.6, we will see that in the context of the full system, including the features which we will discuss below, the left-corner filtering on the possible reduction-set is a slight net benefit, so in the end we will use this feature as well.

6.4.2 Look-ahead Constraint

Another restriction that can be placed on a parser involves adding look-ahead. The essence of look-ahead is to only add constituents (or start rules) that can be combined with the token or tokens that come immediately after the position currently under consideration. According to Moore [94], framing this filtering in this way was proposed by Wirén [160], though the use of look-ahead was described as part of building an LALR(1) parsing table by Aho and Ullman [2, pp. 219–224, 236–241], who in turn credit DeRemer [32] with inventing the LALR parser-construction algorithm [2, p. 243]. DeRemer in turn indicates that the look-ahead technique came from Knuth [64], though the separate calculation of the look-ahead set was novel (to DeRemer) [32, p. 459].

This constraint is also accomplished with the aid of a simple relation, which encodes whether some constituent A can be adjacent to another constituent B. If the parser is about to complete constituent A using word w_i, then we can ask whether word w_{i+1} can immediately follow A. If not, then we should not add A to our set of constituents, because it cannot be used in any analysis.

The guiding principle of look-ahead is this: don't create a constituent that cannot possibly be followed by the next word. We should only create a constituent of category c ending at e if c is

Filtering type	*features*	total num. nodes	total num. arcs	parse time (CPU secs.)
No left-corner	No look-ahead	1,979,651	3,719,086	135.66
	Look-ahead	**1,754,684**	**3,498,758**	**128.18**
Ordinary Left-Corner	No look-ahead	1,014,451	2,607,842	171.49
	Look-ahead	**928,007**	**2,463,204**	**159.74**
Cocke-Schwartz LC	No look-ahead	984,083	2,264,255	77.96
	Look-ahead	**900,674**	**2,126,952**	78.06

Table 6.3: Comparison of parse times for the Tree-Structured Grammar (using full Cocke-Schwartz Filtering) with and without look-ahead. (Ordinary Left-Corner is without LC-of-LHS-set filtering, whereas Cocke-Schwartz Left-Corner filtering uses this feature.)

a member of the set of categories that can occur to the immediate left of the category of the next word (at $e+1$). Since this relation depends only on the grammar, it can be calculated in advance of actually parsing, during a grammar compilation phase of processing.

Our implementation We implement the look-ahead by precalculating a PRECEDE relation: if $b \in$ FOLLOW(a), then $a \in$ PRECEDE(b). Then, while processing the input word at e, we union together all the PRECEDE sets for each basic category of the next word ($e+1$). Finally, we allow the creation of c only if c is a member of this union. Our actual implementation of the calculation of the FIRST and FOLLOW relations follows the presentation in Aho, Sethi, & Ullman [1, pp. 188-189].

The chart-parser implementation of this restriction is straight-forward. Just before starting a new input word, the PRECEDE set of each of the possible categories for the next word after that (and any phrases that can start with the second word) are added to the look-ahead set.

Then, before proposing a new chart entry (inactive edge), the look-ahead set is consulted to see if the new category could precede anything that the next word could begin by checking to see that it is a member of the current look-ahead set.

From Table 6.3, we can see that adding look-ahead constraints to our parser reduces the number of chart nodes (inactive edges) that are built by 9% (for the Cocke-Schwartz filtering case). For ordinary left-corner filtering, it also speeds up the parser, but for Cocke-Schwartz filtering it is about a wash; there is no improvement in run-time. We will see in the next section that look-ahead is a necessary condition for an additional filtering option that does substantially decrease structure building and improves the run-time performance as a result.

Filtering type	*features*	total num. nodes	total num. arcs	parse time (CPU secs.)
Ordinary Left-Corner	no LC-of-LA	928,007	2,463,204	159.74
	LC-of-LA	928,007	**711,296**	159.70
Cocke-Schwartz LC	no LC-of-LA	900,674	2,126,952	78.06
	LC-of-LA	900,662	**576,622**	**75.32**

Table 6.4: Comparison of parse times for the Tree-Structured Grammar, using left-corner and look-ahead constraints, with and without the left-corner-of-look-ahead [102] constraint.

6.4.3 Left-corner of Look-ahead

This combines together the look-ahead and left-corner restrictions described above. In essence: don't create an active arc which requires a constituent that cannot be started by the next word. Only create active arc a that would next require category $a.c$ if $a.c$ could be started by the next word.

Nederhof [102] proposed, as an additional optimization to look-ahead filtering, precomputing the (upward) left-corner relation of the next input token, and then filtering by performing a simple set-membership test against this left-corner set. Our implementation follows this proposal.

We implement this by precalculating an UPWARD-LC (or FIRST-PARENT) relation, which is an inclusive transitive closure of the *parent-of-leftmost* relation: a is a member of UPWARD-LC(b) if b could be a left-most descendant of a. Then, while processing the input word at e, we union together the UPWARD-LC sets for each basic category of the next word starting at $e+1$. Finally, we allow the creation of an active arc $a =< (e+1), \text{cat} >$ only if cat is a member of the union of UPWARD-LC sets for $e+1$.

Implementation notes When integrating left-corner constraints into the grammar tree, we noticed that the set of completable categories is quite sparse compared to the left-corner set. Rather than performing a simple set-intersection and set-non-empty operation, it turned out to be much faster to ask if the intersection between these two sets was non-empty by performing a sparse-set to dense-set intersection operation, short-circuiting when any part of the intersection was discovered. For the actual implementation, we found it best to simply decorate the tree with a list of completable categories, and to perform the restriction check by looking down this list for an element that was also in the left-corner set.

From Table 6.4, we can see that adding the left-corner-of-look-ahead constraint to our parser greatly reduces the number of arcs (active edges) that the parser needs to build — a reduction by a factor of over $3\frac{1}{2}$ times. Even though making a new arc is a relatively cheap operation, this results in a slight improvement in the run-time performance.

6.5 Other Parser Features

There are a considerable number of features that can be added to the basic algorithm above to turn it into what could be considered a 'modern' parser. In addition to the left-corner and look-ahead search restrictions, these features include unification, packing, and an end-point search restriction.

Unification We will note here that this parser was developed to support "augmented context-free" unification grammar [152] using a Lexical Functional Grammar [56] inspired formalism. Supporting unification is the reason we could not adopt methods such as those used by Nederhof [103] or Moore [94], which modify the grammar so substantially that rediscovering the constituents of a complete rule is difficult.

Following Tomita [151] and Carroll [23], we apply packing of locally ambiguous constituents. As Briscoe & Carroll [18] and Alshawi, *et al.*[4] noted, packing can be used to save considerable amounts of duplicated work, but when working within a unification grammar framework both the constituent structures *and* the unification feature structures must be packed in order to see any actual gains.

Packing In the algorithm presented below, packing is hidden within the `agenda-add` operation. Since an entry actually entered in the chart may have been used to complete other rules, and since its unification value will have been used in this case, we cannot simply pack into constituents already in the chart unless we then recalculate the unification values for any other constituent for which the chart entry in question is a descendant [70]. Implementing this adds a lot of extra book-keeping, so we simply chose to pack only into the agenda (since we know that agenda items have not been used and thus may be packed together safely) [76].

Optimal agenda ordering Lavie and Rosé [76] examine an interesting aspect of chart parser implementation: agenda ordering. In a chart parser, as new constituents are discovered, they are placed on an *agenda* of items which require further attention. These are then used to start, extend, and finish other partial constituents.

What Lavie and Rosé discovered was that in a chart parser with interleaved unification such as LCFlex [125], once the unification value attached to a node N has been used, other ambiguous nodes N' cannot be packed into N without recalculating the unification results for all parents of N. Figuring out which unification results to recalculate in inconvenient, and it is often better to simply leave the node unpacked. Unfortunately, leaving it unpacked will lead to redundant work later in

the parse.

The best way to handle this would be to find an ordering on the agenda that allows us to know that once we start to use node N, we will have already packed N' into N. Such an order would be an optimal order in which to process the agenda.

Lavie and Rosé [76] discovered that such an ordering exists, and that it can be precalculated, based solely on the structure of the grammar.

We adopt this technique as-is, calculating the partial ordering at the same time that we calculate the left-corner and look-ahead relations. The ordering of the actual agenda is then compartmentalized into the `agenda-add` function.

End-point restrictions The end-point restriction on starting rules is from Carroll [23, p. 55]. Carroll's actual restriction was on the minimum number of tokens needed to complete the new rule. Unfortunately, because the tree-grammar can encode several rules of different lengths together in one starting arc, we could only apply this restriction in this more limited form. (We actually tried annotating the tree with an indication of minimum length needed to complete any node, but this didn't save any appreciable run-time over and above this simple version.)

6.5.1 Complete algorithm

Figure 6.4 shows the main loop of the full tree-structured grammar chart parser, including unification and the restrictions discussed above. The actual implementation of the parser follows this structure exactly. We used object-oriented design to hide techniques such as packing and agenda ordering in the implementation of the agenda; table insertion and look-up details are in the implementations of the chart and (separate) active arcs. In doing so, we can clearly separate the core algorithm from these other (important) additional techniques.

For an efficient implementation of this algorithm, there are several data structures that must be implemented in an efficient way.

- The agenda must be implemented in a way that supports quick addition and removal. This is because every entry in the chart must first be added and then later removed from the agenda.

  (We shall note here that searching down a list, such as the LISP `ASSOC` function, is *not* a unit-time operation, and therefore not to be considered 'quick'.)

- Active arcs should be stored in such a way that a new active arc can be added quickly.

- Active arcs should also be stored in such a way that the `arcs-continued-by` operation is quick — preferably a unit-time operation such as a table- or hash-lookup.

6.6 Context-free Parsing Results

To demonstrate the effectiveness of the various techniques described above, we used the test corpus of natural KANT data described in Section 5.4 above.

As a reminder, this corpus consists of 1447 sentences, with an average length of just over 18 words per sentence, taken from the corpus used by Mitamura *et al*. in their investigation of automatic disambiguation [90]. These sentences are applied to the KANT (982-rule) unification grammar.

All of the results shown in tables 6.5 and 6.6 were collected using the same basic parser. For the "standard" entries, the parser was loaded with a grammar *without* applying the prefix-compression described above, whereas the "tree" entries do use the fully compressed tree-structured grammar. Both configurations were run with various constraints turned on or off. "LC-at-end" refers to left-corner restriction applied just prior to computing the unification function and (if successful) creating a chart entry; "LA" to look-ahead; and "LC-of-LA" refers to the left-corner-of-look-ahead as outlined above. "LC-in-rules" refers to applying the left-corner restriction prior to *starting* a rule, whereas "LC-in-tree" refers to applying the left-corner set-intersection-non-empty restriction in the grammar tree (as described above).

6.6.1 Discussion of results

As we stated in the beginning of this chapter, what we are looking for in the context-free portion of our parser is the least run-time, while creating the smallest number of chart entries (i.e. inactive arcs).

Table 6.5 lists some comparative results of running the parser on only the context-free spine of the test grammar. Comparing the numbers of arcs generated by the naïve chart versus the tree-structured chart that using the grammar tree does indeed save a substantial number of arc creations. Without any left-corner or look-ahead restrictions, when compared to the naïve implementation, the tree-structured grammar reduced the number of active arcs created by 22%. When employing full left-corner and look-ahead constraints [108, 125, 48] on the parser, the tree-grammar gave a 33% reduction in the number of active arcs, and a run-time improvement of 6.6%.

Table 6.6 lists comparative numbers for running the parser with fully-interleaved unification.

Grammar type	LC type	*LC of LHS set*	Look-ahead	LC-of-LA	total num. nodes	total num. arcs	Parse time (CPU secs)
Standard (linear)	none				2,006,968	4,554,782	144.70
			X		1,777,092	4,213,835	138.09
	Ordinary Left-Corner				1,016,736	3,164,755	194.23
		X			986,091	2,784,207	308.36
			X		930,071	2,945,009	175.60
		X	X		902,497	2,574,894	281.99
			X	X	930,071	924,532	173.42
		X	X	X	902,485	794,128	200.77
	Cocke-Schwartz				1,016,736	3,164,755	84.32
		X			986,091	2,784,207	84.48
			X		930,071	2,945,009	85.10
		X	X		902,497	2,574,894	85.63
			X	X	930,071	892,131	81.82
		X	X	X	902,485	767,651	80.32
Tree-structured	none				1,979,651	3,719,086	135.66
			X		1,754,684	3,498,758	128.18
	Ordinary Left-Corner				1,014,451	2,607,842	171.49
		X			984,083	2,264,255	265.61
			X		928,007	2,463,204	159.74
		X	X		900,674	2,126,952	247.61
			X	X	928,007	711,296	159.70
		X	X	X	900,662	601,253	180.10
	Cocke-Schwartz				1,014,451	2,607,842	77.45
		X			984,083	2,264,255	77.96
			X		928,007	2,463,204	78.19
		X	X		900,674	2,126,952	78.06
			X	X	928,007	680,800	76.90
		X	**X**	**X**	**900,662**	**576,622**	**75.32**

Table 6.5: Parse times for Context-Free (only) parse.
LC type refers to the type of left-corner restriction used (none, Ordinary, or Cocke-Schwartz).
LC of LHS set refers to the integration of Left-hand-side category sets into the grammar tree as described in Section 6.4.1.
Look-ahead indicates that this constraint has been used.
LC-of-LA refers to the restriction of active arcs to be in the upward-left-corner of the next symbol as described in Section 6.4.3.
The best combination of features for this configuration is shown in bold face.

Grammar type	LC type	*LC of LHS set*	Look-ahead	LC-of-LA	total num. nodes	total num. arcs	parse time (CPU secs)
Standard (linear)	none				893,483	2,046,123	108.44
			X		803,341	1,909,494	106.37
	Ordinary Left-Corner				515,282	1,514,648	152.82
		X			498,264	1,294,209	207.53
			X		484,163	1,424,560	136.34
		X	X		468,309	1,208,840	187.70
			X	X	484,162	496,948	132.89
		X	X	X	468,297	419,017	148.76
	Cocke-Schwartz				515,282	1,514,648	73.74
		X			498,264	1,294,209	72.63
			X		484,163	1,424,560	74.49
		X	X		468,309	1,208,840	73.15
			X	X	484,162	480,208	70.89
		X	X	X	468,297	405,262	69.46
Tree-structured	none				876,266	1,640,324	99.72
			X		789,083	1,554,241	96.82
	Ordinary Left-Corner				513,632	1,231,939	131.27
		X			496,787	1,037,666	175.11
			X		482,663	1,173,499	123.19
		X	X		466,957	982,738	165.60
			X	X	482,662	367,302	118.18
		X	X	X	466,945	305,543	129.05
	Cocke-Schwartz				513,632	1,231,939	67.89
		X			496,787	1,037,666	68.17
			X		482,663	1,173,499	67.77
		X	X		466,957	982,738	67.48
			X	X	482,662	352,446	64.77
		X	**X**	**X**	**466,945**	**293,610**	**63.10**

Table 6.6: Parse times for parse with interleaved unification.
LC type refers to the type of left-corner restriction used (none, Ordinary, or Cocke-Schwartz).
LC of LHS set refers to the integration of Left-hand-side category sets into the grammar tree as described in Section 6.4.1.
Look-ahead indicates that this constraint has been used.
LC-of-LA refers to the restriction of active arcs to be in the upward-left-corner of the next symbol as described in Section 6.4.3.
The best combination of features for this configuration is shown in bold face.

(See Sections 5.4 and 7.4.3 for details.) As in the context-free case, use of the tree-structured grammar reduces arc creation by 25% when using no restrictions, and 38% using full left-corner and look-ahead restrictions. This latter configuration is also 10% faster than using the naïve grammar structure.

It is also interesting to note the huge decrease in the number of active arcs created when applying the left-corner-of-look-ahead restriction in the creation of active arcs when compared to applying the left-corner and look-ahead restrictions only to the creation of the chart entries. This is clearly a win in terms of reducing memory use, though the resulting speed improvement is extremely modest.

On this corpus, left-corner constraints alone are clearly superior to look-ahead constraints alone. Further, the combination of left-corner and look-ahead constraints are only a modest win over left-corner alone until the left-corner-of-look-ahead restriction is applied as well.

Conclusion Over all, using the tree-structured grammar substantially reduces the number of arcs created in in this chart parser, and cooperates well with the combined left-corner and look-ahead restrictions.

We can also see that we need not only a good parsing algorithm, but that the implementation of that algorithm must be good as well. This is especially true in the case of filtering techniques, where the cost of applying some filter might be substantially greater than the savings that result from applying that filter. In these cases, making an appropriate choice may require an empirical comparison of several alternate approaches. This is shown most clearly in the results for the "ordinary" Left-corner constraint, where not using this feature is faster than using it, even though the amount of structure that is created has been reduced.

Fortunately, by carefully implementing several options and empirically comparing their effects, we can see that by the adoption of the tree-structured grammar, full Cocke-Schwartz left-corner filtering, and full look-ahead constraints, we realize an improvement of some 67% in the average run-time of our fully-interleaved unification parser, while almost halving the number of chart entries that are created.

The topic of the next chapter is how the unification portion of our parser works.

```
basic_tree_parse (words):

    loop:
        if (agenda empty)
            if (no more words)
                break
            else
                Add next word

        e = remove next entry from agenda

        add e to chart.

        foreach r in rules-started-by (e.constituent) do:
            new-arc = make-arc (e.end, r, traceback = (item) )
            arcs-add (new-arc)

        foreach rule in single-rules-completed-by (e.constituent) do:
            new = make-entry (e.start, e.end, rule.LHS,
                              children = list(list(e)))
            agenda-add (new)

        foreach arc in arcs-continued-by (e.start, e.constituent) do:
            foreach node in arc.tnode.shiftlist
                new-arc = make-arc (e.end, node,
                                    traceback = (cons(e, arc.traceback))
                arc-add (new-arc)

            foreach rule in arc.tnode.reducelist
                let new-children = reverse (cons(e, arc.traceback))
                new = make-entry (first(new-children).start,
                                  e.end, rule.LHS,
                                  children = list (new-children))

                agenda-add (new)

    end loop

end Basic_tree_parse
```

Figure 6.3: The Basic Tree-Structured Grammar Chart Parser

```
Parse (words):
    Add words to look-ahead (words)
    loop:
        if (agenda empty)
            if (no more words)  break loop
            else
                Add next word

        e = pop next entry off of agenda;  add e to chart.

        unless (e.end == final-position)
            foreach r in rules-started-by (e.constituent) do:
                if (lc-of-look-ahead-licenses (r.LHS)
                    && lc-node-set-licenses (r, e.start))
                    new-arc = make-arc (e.end, r, traceback = list(e) )
                    arcs-add (new-arc)

        foreach r in single-rules-completed-by (e.constituent) do:
            if (look-ahead-licenses (r.LHS) && LC-licenses (r.LHS, e.start))
                fs = rule.unify ( (e.fs) )
                if (fs)
                    new = make-entry (e.start, e.end, r.LHS, fs,
                                      children = list(list(e)))
                    agenda-add (new)

        foreach arc in arcs-continued-by (e.start, e.constituent) do:
            foreach node in arc.tnode.shiftlist
                if (lc-of-look-ahead-licenses (rule.LHS, e.end)
                    && lc-node-set-licenses (arc.tnode, arc.start))
                    new-arc = make-arc (e.end, node,
                                        traceback = cons(e, arc.traceback))
                    arc-add (new-arc)

            foreach rule in arc.tnode.reducelist
                let new-children = reverse (cons(e, arc.traceback))
                    child-fs = get-FSs-of (new-children)
                if (look-ahead-licenses (rule.LHS)
                    && LC-licenses (rule.LHS, first(new-children).start))
                    fs = rule.unify ( child-fs )
                    if (fs)
                        new = make-entry (first(new-children).start,
                                          e.end, rule.LHS, fs,
                                          children = list (new-children))
                        agenda-add (new)
    end loop
```

Figure 6.4: Tree-Structured Grammar Chart Parser with constraints and Interleaved Unification

Chapter 7

Pseudo-Unification: Implementation and Optimization

"Don't work harder, work *smarter*."

— The Pointy-Haired Boss,
from the comic strip *Dilbert*.

For our system, we can make an observation about the problem of unification values: for parsing, we don't really care about what the value at a subordinate node (such as A in Figure 3.6 of Chapter 3) because *we will only use the top-level unification value of a parse.*

With this in mind, it makes perfect sense to use a unification framework which does not bother calculating values which we will not use anyway. As we noted in Section 3.2, Pseudo-unification [153] is such a framework. In this chapter, we survey the pseudo-unification framework which we actually use, and investigate several methods which we used to improve the performance of the unification system.

We then investigate how to transform a grammar in order to create a *shallow* unification system – one which entirely avoids building the sorts of deep unification values which we found to be so bothersome in Chapter 5. Finally, we investigate techniques to further improve the run-time performance of the shallow unification system.

7.1 Introduction to Pseudo-Unification

The pseudo-unifier for our system is a descendant of the one designed by Tomita and Knight [153] and described in detail in the GENKIT version 3.2 User's Manual [154, pp. 10–13]. We will describe the changes to this unifier which we found to be useful, but first we must briefly review the syntax and semantics of the unifier.

7.1.1 On Interpreting Pseudo-Unification

First, let us consider the syntax of this unifier. We can describe this with a small number of meta-syntax rules:

- A grammar rule has associated with it a list of equations.
- An equation can be either a simple equation or a special form.
- Ordinary equations are made up of *paths* and *atoms*. A path is a list of slots, starting with a top-level variable.
- A top-level variable is either a reference to the grammar node's value (`X0`), or one of the children (`X1`...`X`n, where `X1` refers to the leftmost child, `X2` the second from the left, etc.).
- The special forms include both ordinary and conditional disjunctions. Each of these offer a choice over a list of lists-of-equations, with somewhat differing semantics as to how the individual lists-of-equations are handled (see Section 7.3.1 below).

Values

Unlike full unification, this unifier operates on *tree* structured values. The fundamental data structure is a labeled tree, where every node in the tree (both internal- and leaf-nodes) has a label with the exception of the root. The children of an internal node of the tree are labeled, but unordered. The two values:

```
(FOO A)          and     (BAR X)
(BAR X)                  (FOO A)
```

are considered to be equal. Thus the two are unifiable together.

These trees are a generalization of a set of *slot-value* associations: the value of a slot-value pair can be another set of slot-value pairs. Slots are then the names of internal nodes of a tree. A *path* is

a sequence of slot names, starting at the (unnamed) root. The trees need not be strictly a collection of values, or even of any particular depth. For the unification value:

```
(FOO ((FOO-VALUE A) (SUB-FOO FOOD)))
(BAR X)
```

`(FOO FOO-VALUE)` is a path.

Data Disjunctions. In addition to a single value (or entire sub-tree), a slot can also contain a disjunction over values. For example:

```
(FOO (*OR* A B C))
```

Although this example shows a disjunction at the leaves of the tree, disjunctions are also possible at interior nodes of the tree:

```
(FOO (*OR* ((FOO-VALUE A) (SUB-FOO FOOD))
           ((FOO-VALUE B) (SUB-FOO FOOL))
           ((FOO-VALUE C) )
     ))
(BAR X)
```

Data value complements. A value can also be a complement over one or more sets of values (or even sub-trees), for example:

```
(FOO (*NOT* B C D))
```

Similar to data disjunctions, a complement is considered to describe an open-ended set, containing any value except for the ones named.

Top-Level Variables. If we consider the entire environment of values to be a single tree, then the top-level variables are simply the labels on the first-level sub trees. We will see below in Section 7.1.1 that this is a convenient data representation. Since we could end up with *several* sets of values of top-level variables, we can conveniently represent these with a data disjunction at the top-most level.

In the bottom-up pseudo-unifier that we use, `X1...X`n are initialized to contain the unification values associated with the child nodes, and the parent variable `X0` is initialized to contain no value. After the unification equations have been evaluated, the set of values bound to the `X0` top-level variable are retrieved, and associated with the node.

In a top-down pseudo-unifier, the parent variable `X0` should be initialized with the appropriate value, and the child nodes should be initialized to be empty. After unification equations have been evaluated, the resulting values for each of the child variables `X1...X`n should be extracted and associated with the appropriate child nodes.

Ordinary Equations.

There are two basic types of unification equations:

- *l-path = value* (where value is either a single value or a disjunctive set of values)
- *l-path = r-path*

***l-path = value* equations.** The *l-path = value* equations are handled in a very straight-forward way: if the value will unify with the path, the equation succeeds, otherwise it fails. More precisely, the tree is traversed, one element of *l-path* at a time, until final slot has been found. Then the value stored in this slot is compared to *value*. If the two values are equal (without regard to ordering), the unification succeeds; if these values are different, then the unification fails.

***l-path = r-path* equations.** *L-Path = r-path* equations are handled in a straight-forward way: *r-path* is first traversed to find the value stored in the final slot of this path. Then this value is unified with *l-path* as above.

Side effects in Ordinary Equations. The evaluation of unification values becomes more interesting when some portion of *l-path* does not exist. For ordinary unifications, indicated with the `=` operator as above, if some portion of the path does not exist, it is created in the left-hand-side tree.

There is also a 'checking' unification operator, `=c`, which insists on finding some value: if some portion of *l-path* does not exist, this operator simply fails.

Unlike full unification, no side effects can take place in the right-hand-side (*r-path*) of *l-path = r-path* equations. As a result, the two equations

```
(X1 FOO) = (X2 FOO)
```

and

```
(X2 FOO) = (X1 FOO)
```

are *not* equivalent in this unification system, because they do not produce the same side effects.

Data Disjunctions in Ordinary Equations. When either *value* or the value stored in the final slot of *path* contains a disjunction over a set of values, both the standard and checking unification equations perform a set-intersection operation to determine if there are some number of values which are equal within these two sets. If so, then the slot at the end of *path* is modified to reflect this set intersection. It is important to note that this modification also happens for the 'checking' unification operator, `=c`.

When the (unnamed) tree of values contains a data disjunction in an interior node, the unifier loops over each sub-tree within the disjunction, recursively attempting to perform the unification with each in turn. Some of these may succeed, whereas others may fail. In this situation, the failing sets of sub-trees are removed. In this situation, the over-all unification is considered to fail only if every sub-tree unification fails.

Finally, note that the top-level tree of values may contain a disjunction over *sets* of top-level variables. This situation is perfectly legal, necessary for correct operation, and notably different than the early version of the pseudo-unifier illustrated by Carbonell and Tomita [20, 153].

Data value complements in Ordinary Equations. It is possible for either of *value* or the value stored in the final slot of *l-path* to contains a complement of one or more sets of values, for example:

```
(*NOT* B C D)
```

A complement describes an open-ended set. These unify with single values and disjunctions in the way one would naïvely expect; *e.g.* unifying `(*NOT* B C D)` with `(*OR* A B)` is `A`.

Unifying two `*NOT*`s together results in a `*NOT*` of union of the complemented sets. So the result of unifying together `(*NOT* B C D)` and `(*NOT* A B X)` is `(*NOT* A B C D X)`.

Lists of Equations

The result of unifying a list of equations is the result of applying each of the equations in order. If any equation should fail, then the entire list of equations is considered to have failed. Because of this, lists of equations can be considered conjunctive.

In many situations, notably including disjunctions (below), we must create a copy of the current environment (*i.e*. the current values of all top-level variables) every time we start to evaluate a list of equations, so that if some equation in the list fails we can restore the old values. This is because each individual unification equation might modify the current values, but we cannot accept a modified set of values until every unification in the list succeeds.

Disjunctions of Lists of Equations

There are two types of disjunctions over lists of equations: ordinary disjunctions and conditional disjunctions. The ordinary disjunction has the form:

(`*OR*` *list-of-equations-1 list-of-equations-2 list-of-equations-3 ...*)

The result of an ordinary disjunction special form is a data disjunction containing the results of applying each of the lists of equations in turn to the current values.

In some sense, these can be considered to be simply "syntactic sugar" on the grammar rules themselves. Any nested list of equations associated with a single rule can be re-written into an equivalent Disjunctive Normal Form [126, pp. 584–586] — a single top-level `*OR*` with some number of subordinate lists of equations (see also [144, sec. 15.4, pp. 461–472]). This can then be replaced by a number of individual grammar rules, one for each subordinate list of equations, containing this list, and with the same Context-Free part as the parent rule.

For example, the rule:

```
X <== Y
  ((*OR*
     (((X1 FOO) = A)
      ((X0 FOO-FEAT) = +))
     (((X1 FOO) = *UNDEFINED*)
      ((X0 FOO-FEAT) = -)) )
   (*OR*
     (((X1 BAR) = A)
      ((X0 BAR-FEAT) = +))
     (((X1 BAR) = *UNDEFINED*)
      ((X0 BAR-FEAT) = -)) )
   )
```

can be expanded into:

```
X <== Y
  ((*OR*
     (((X1 FOO) = A)
      ((X0 FOO-FEAT) = +)
      ((X1 BAR) = A)
      ((X0 BAR-FEAT) = +))

     (((X1 FOO) = A)
      ((X0 FOO-FEAT) = +)
      ((X1 BAR) = *UNDEFINED*)
      ((X0 BAR-FEAT) = -))

     (((X1 FOO) = *UNDEFINED*)
      ((X0 FOO-FEAT) = -)
      ((X1 BAR) = A)
      ((X0 BAR-FEAT) = +))

     (((X1 FOO) = *UNDEFINED*)
      ((X0 FOO-FEAT) = -)
      ((X1 BAR) = *UNDEFINED*)
      ((X0 BAR-FEAT) = -))
   )
```

and finally expanded into the four rules:

```
X <== Y
  (((X1 FOO) = A)
   ((X0 FOO-FEAT) = +)
   ((X1 BAR) = A)
   ((X0 BAR-FEAT) = +))

X <== Y
  (((X1 FOO) = A)
   ((X0 FOO-FEAT) = +)
   ((X1 BAR) = *UNDEFINED*)
   ((X0 BAR-FEAT) = -))

X <== Y
  (((X1 FOO) = *UNDEFINED*)
   ((X0 FOO-FEAT) = -)
   ((X1 BAR) = A)
   ((X0 BAR-FEAT) = +))

X <== Y
  (((X1 FOO) = *UNDEFINED*)
   ((X0 FOO-FEAT) = -)
   ((X1 BAR) = *UNDEFINED*)
   ((X0 BAR-FEAT) = -))
```

For ordinary disjunctions, such an expansion should result in a weakly-equivalent set of equations. If each rule in this collection is applied to some set of input unification value, and the output values are packed together in a disjunction, resulting unification value should be weakly equal to the result of applying the same input to the original nested list of equations. By weakly equivalent, we mean that if each are expanded into Disjunctive Normal Form with the removal of any duplicate terms, the expanded unification values will be equal.

It is important to note that since the expansion into Disjunctive Normal Form results in a number of clauses exponentially proportional to the number of independent disjunctions in the original rule [126, pp. 580], such an expansion would create a similarly exponential expansion in the number of rules.

The reverse of this transformation could be used as a grammar optimization, though we did not explore this possibility.

Conditional disjunctions The conditional disjunction special form has a similar form:

`(*EOR*` *list-of-equations-1 list-of-equations-2 list-of-equations-3* `...)`

For ordinary (non-disjunctive) unification values, the conditional disjunctive special form is evaluated in a manner similar to the ordinary one: each list of equations is evaluated as a group. Unlike the ordinary disjunction, as soon as one of these lists of equations succeeds, evaluation is terminated and the side-effects of the successful list-of-equations are preserved. This structure is commonly called a *conditional OR* because in an ordinary programming language, such a structure can be equivalently re-written as [44, sec. 7.6.2, p. 181], [99, sec. 8.7.1, pp. 236–244]:

if *list-of-equations-1*
else if *list-of-equations-2*
else if *list-of-equations-3*
...
else *list-of-equations-N*

The GENKIT version 3.2 User's Manual [154, pp. 10–13], mistakenly describes these as "exclusive ORs". They are not exclusive, however, since it may be possible for several of the lists of equations to successfully evaluate, whereas an exclusive disjunction would require that *exactly one* of these lists should succeed.

The conditional OR structure is somewhat problematic to apply to unification values which themselves contain disjunctions. The naïvely correct behavior for evaluating a conditional OR structure in the context of a set of unification values containing disjunctions is to expand the disjunctions to the top-most level (*i.e.* into disjunctive normal form), and then iterate over the terms in this set, collecting the results of applying the conditional OR structure to each term in a disjunction. Section 7.3.1 below will consider an alternative method for evaluating these structures.

Inter-slot Dependencies Both ordinary and conditional disjunctions can create a situation in which several slots become semantically dependent on each other. Consider the rules:

```
((*OR*
   (((X1 FOO) = A)
    ((X1 BAR) = +))
   (((X1 FOO) = *UNDEFINED*)
    ((X1 BAR) = -))
   (((X1 FOO) = *DEFINED*)) )
 )
```

It is *not* semantically correct to simply represent the result of this special form by assigning the value `(*OR* + -)` to `(X1 BAR)` because `(X1 BAR)` only has the value `-` *when (X1 FOO) is undefined*.

To see this, suppose that we start `(X1 FOO)` with the value `(*OR* A B)`. After evaluating the above disjunction, there are two different values for `X1`:

1. `(X1 FOO) = A, (X1 BAR) = +`
2. `(X1 FOO) = *OR* A B, (X1 BAR)` undefined.

The same effects which create these inter-*slot* dependencies within a unification structure also create inter-*variable* dependencies. Consider the implications of changing the example above to:

```
((*OR*
   (((X1 FOO) = A)
    ((X0 BAR) = +))
   (((X1 FOO) = *UNDEFINED*)
    ((X0 BAR) = -))
   (((X1 FOO) = *DEFINED*))
 ))
```

If we were to evaluate this disjunction in an environment with `(X1 FOO) = (*OR* A B)` as above, now `(X0 BAR)` will depend on and be related to `(X1 FOO)`.

As we mentioned in Section 7.1.1, it is convenient to represent the entire environment of unification variables as a single tree. Doing so allows us to represent multiple sets of interdependent sets of variables as a top-level disjunction over trees, each of which contains a single set of variable bindings.

(*N. b.:* The tree unifier in this system is a direct descendant of the one described by Carbonell and Tomita [20, 153]. We did not change this particular aspect of the pseudo-unification system, though it *is* different from the description given by Carbonell and Tomita; we assume that this change was done by some prior engineer.)

7.2 Modifications to the Pseudo-Unifier

In this section, we will highlight the modifications which we found to be important to make to the pseudo-unifier. These modifications support the optimizations described in Section 7.3 and the negative restriction and approximate packing described in Section 7.4 below.

7.2.1 'Gray-Box' Adaptation

Because we agree with some of the design choices but strongly disagree with others (*e.g.* LISP call-outs), we will make no attempt to justify the particular choices of the semantics of the pseudo-unifier other than to note that our grammars were written to use this particular set of semantics. Since compatibility with earlier systems is a design goal of this system, we attempted to only extend, rather than modify, the semantics of the pseudo-unifier. (In the terminology of Software Engineering, this is a 'gray-box wrapping' extension of the previous version [120, pp. 731–2].)

Our feeling is that a radical change to the semantics of the pseudo-unifier (notably in removing some types of unification equations) would require a large re-engineering effort of existing grammars. While possibly worthy for other reasons, such an effort is orthogonal to the goals of this work. So the pseudo-unifier was predominately extended rather than being drastically reorganized.

7.2.2 Handling of Data Disjunctions

In the previous implementation of the pseudo-unifier, disjunctions in the data were 'unpacked' – expanded so that the disjunctions appear only at the top-most level (*i.e.* disjunctive normal form). A comment in the code says that it does this so that the unifier could be implemented without worrying about disjunction during unification [155].

Of course, if there is any possibility of a blow-up in the unification space, expanding the values to disjunctive normal form will insure that such a blow-up will happen. Because of this, we found that we could save a considerable amount of run-time by examining how all unification values involving disjunctions were evaluated, and endeavoring to avoid expanding these values into Disjunctive Normal Form unless there was no other way to correctly evaluate the unification equations.

Section 7.3.1 below discusses an optimization for conditional ORs which avoids data unpacking. In this section, we see that avoiding unpacking the data disjunctions can give an average speed-up of a factor of 18 on ambiguous natural data, and that most of this speed-up is due to avoiding disjunctive blow-ups in several outlier sentences. We also examine this disjunctive blow-up behavior in detail, and see that simply by avoiding unpacking the data disjunctions we can almost double the number of consecutive prepositional phrases that an all-values configuration of the parser can handle before facing a computational blow-up.

7.2.3 Explicit No-Value Values

Normally, the absence of a value in a slot is indicated by the absence of the slot. Disjunctions in data values complicate this somewhat, because we may very well wish to express a choice between a slot being assigned some value and that slot having no value whatsoever. This is particularly useful for packing unification values together as part of local ambiguity packing during the parse pass (see Section 6.5 above), though it can also occur as the result of the evaluation of a disjunction over a list of equations (see Section 7.1.1 above).

For example, the unification value:

`(FOO (*OR* BAR` *no-value*`))`

describes the situation where either slot `FOO` holds value `BAR` or is unassigned.

In order to handle this case smoothly, we included an explicit *no-value* value. These are treated exactly as the absence of a value, except that they can be conveniently combined in disjunctions.

7.2.4 Wild-Carded Values

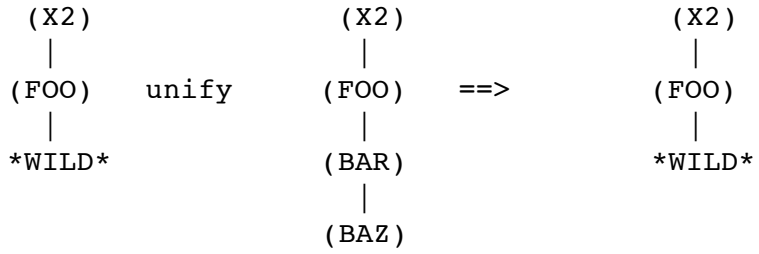
We found it useful in some circumstances to skip over the evaluation of some parts of the unification equations. The feature-based flexibility of the unification system of Rosé and Lavie's LCFlex system [125] suggested a technique. Rosé and Lavie introduced a wild-card value. This wild-card is used to allow certain feature slots to be optional, rather than required parts of the unification in order to enhance the robustness of their unification grammar.

For our system, we found that the wild-carding mechanism served several different goals equally well. This mechanism is primarily used to support a form of negative restriction [136, 45, 115]. It is also used to support the (unsuccessful) approximate packing length limitation described below in Section 7.5.2.

Implementation. We introduced a wild-card value `*WILD*` to stand in place of part of an F-structure. This wild-card has the effect of causing most types of unification equations that reach this part of the F-structure (either as a value, or as part of a path) to succeed, returning an F-structure containing the wild-card.

This is best illustrated with an example. Suppose that we have some F-structure including an element `X2`, which has a feature slot `FOO`, and that `(X2 FOO)` currently has the value `*WILD*`.

Applying the operation `(X2 FOO BAR) = BAZ` will succeed, leaving the `FOO` slot of `X2` with `*WILD*`. This is because our pseudo-unifier is a tree unifier [152, 154], and the leaf node `*WILD*` successfully unifies with any sub-tree giving itself as the result. Graphically, this example appears thus:



There are three exceptions to this behavior. The negative-existential condition will fail if applied *at a leaf node*, and path-removal equations and over-write equations (see below) will modify the `*WILD*` at a leaf node.

Continuing our example, the equation:

```
(X2 FOO) = *UNDEFINED*
```

Will fail for the above example. Similarly, the equation:

```
(X2 FOO) = *REMOVE*
```

will successfully remove the `FOO` slot from `X2`, even though it contained a `*WILD*`.

On the other hand, if that removal equation was a request for the removal of a sub-slot of the wild-carded slot:

```
(X2 FOO BAR) = *REMOVE*
```

It would not change `X2`, because even though the `BAR` sub-slot should be removed, we must continue to "pretend" that other sub-slots of `FOO` exist. For similar reasons, the equation:

```
(X2 FOO BAR) = *UNDEFINED*
```

will succeed.

Wild-carding and Restriction. We chose to adopt this particular behavior because the KANT grammar primarily used by this parser contains many rule sequences which include (and are thus conditioned on) rules such as:

```
(X2 obj) = *UNDEFINED*
```

Rules are written in this way because complex-valued features, such as the `obj` feature in this example, are often used to hold both semantic and existential information.

The assignment of a complex value to features such as these cause problems with packing, but that some value was assigned is also useful information in the grammar. What we wish to do is remove the problematic cases (*i.e.* the complex feature assignment), but keep the (easy-to-pack) existential behavior.

A straight-forward implementation of negative restriction, such as used by Oepen and Carroll [115], would be accomplished by simply removing all references to the restricted features from the unification equations used in the first-pass grammar. This would have the effect of removing both the problematic semantic information as well as the beneficial existential information.

In order to retain the existential information, we chose to define the semantics of the wild-card value such that a `*WILD*` value will succeed when tested for existence (*i.e.* unified with `*DEFINED*`), but will fail when tested for non-existence (*i.e.* unified with `*UNDEFINED*`). Then rather than removing all references to features which are assigned complex values (as in straight-forward negative restriction), we replace the complex assignment with an assignment of `*WILD*`.

This modification in behavior has an extremely small effect on the performance of our system. When tested on the Catalyst ambiguity corpus (see Section 5.4), causing a terminal `*WILD*` value to fail when tested for non-existence (rather than succeed) reduced the number of nodes created by only 0.25%. This in turn reduced the number of nodes searched in the second pass by 1.5% for the all-values post-parse search, and only 0.9% for the single-value search. Differences in the run-times due to this modification were not significant.

7.2.5 Complements of Unifications

In order to support the conditional OR unwinding optimization described below in Section 7.3.1, we added logical complements for as many unification operations as possible, including every basic equation.

Ordinary Unification Equations

The right way to consider these are as set-complement operators. Suppose we have a set of unification values A, some operator $\star$, and some value V. One can think of applying $A \star V$ as splitting the set into two parts, those for which $A \star V$ succeeds, and those for which $A \star V$ fails.

```
----------------------
|(A * V) |  not(A * V)|
----------------------
```

What we want in a complement operator $A \bar{\star} V = \overline{A \star V}$ is exactly the set of all elements of A for which $A \star V$ fails.

More concretely, suppose we have some F-structure including an element `X`, which has a feature slot `FOO`, and that `(X FOO)` currently has the value `(*OR* A B C)`. Applying the operation `(X FOO) = B` will result in `(X FOO)` having only the value `B`.

Applying the complement operation `(X FOO) /= B` will result in `(X FOO)` having the value `(*OR* A C)` since neither A nor C would have unified with B.

The no-value case is slightly more subtle. If `X` currently does not have a `BAR` slot, then applying the operation `(X BAR) = B` will result in the `(X BAR)` slot being created with value `B`. Because of this, if instead the complement operation `(X BAR) /= B` were to be applied, the result should be a failure (because the uncomplemented operation would have succeeded).

Combining these cases, if `X` also has a slot `BAZ`, which currently has the value `(*OR* A B C` *`no-value`*`)`, applying `(X BAZ) = B` will result in `(X FOO)` having only the value `B`. This is because the actual result will be the four values `(*OR* *FAIL* B *FAIL* B)`; the failures are removed, leaving `(*OR* B B)`; finally duplicates are removed resulting in the value `B`. Similarly, applying the complement operation `(X BAZ) /= B` will result in `(X BAZ)` having the value `(*OR* A C)`, as above.

Of course, since disjunctions may occur at any level of the tree, if instead of one slot `BAZ` with four choices of value we had four choices of values for `X`, each with a different single value for the `BAZ` slot, the result of applying the operators as above would be semantically equivalent.

Complements of Disjunctions and Conjunctive Lists

Both ordinary and conditional ORs are complemented according to DeMorgan's Law[144, pp. 8–10], [126, p. 46]:

$$\neg(a \vee b) = \neg(a) \wedge \neg(b)$$

In our pseudo-unification system, a conjunction is simply a list of equations (which are assumed to be applied in order). So the complement of a disjunction is a list of the complements of each of the items in the original disjunction. Similarly, the complement of a list of equations is an ordinary (non-conditional) disjunction of the complements of the equations in the list.

Therefore the complement of a disjunction of sets of lists of equations is a list of the complements of the lists of equations (which themselves become a disjunction of complemented equations). For example, the complement of:

```
((*OR*
   (((X1 FOO) = *UNDEFINED*)
    ((X0 FOO-DEFNED) = *UNDEFINED*))
   (((X1 FOO) = BAR)
    ((X0 FOO-DEFNED) = +)
    ((X0 FOO-SPECIAL) = +))
   (((X1 FOO) = QUUX)
    ((X0 FOO-DEFNED) = +)
    ((X0 FOO-SPECIAL) = +))
 ))
```

is:

```
(
  (*OR* (((X1 FOO) = *DEFINED*))
        (((X0 FOO-DEFNED) = *DEFINED*)))
  (*OR* (((X1 FOO) /= BAR))
        (((X0 FOO-DEFNED) /= +))
        (((X0 FOO-SPECIAL) /= +)))
  (*OR* (((X1 FOO) = QUUX))
        (((X0 FOO-DEFNED) = +))
        (((X0 FOO-SPECIAL) = +)))
)
```

Exceptions

This pseudo-unification system contains some operators for which no complement can be calculated; notably LISP call-outs. The system simply indicates that it is not possible to complement these statements. By implication, it is also not possible to find the complement of a sequence of statements containing a non-complementable statement, nor is it possible to find the complement of a disjunction containing such a sequence of statements. This is of particular importance for the unwinding of conditional ORs (see Section 7.3.1 below).

7.2.6 Explicit over-write value equation

The pseudo-unifier described in the GENKIT version 3.2 User's Manual [154, pp. 10–13] had a combination of explicit assignment (*i.e.* over-write) and LISP call-out function:

```
(X0 FOO) <= atom
(X0 BAR) <= (LISP-FUNCTION ...)
```

We found it useful to separate these functionalities by introducing a new **:=** operator, which performs an explicit over-write function.

```
path := <atom or path>
```

In a bottom-up environment, this has exactly the same semantics as the (GENKIT unifier) sequence:

```
path = *REMOVE*
path = <atom or path>
```

As such, this addition is mostly 'syntactic sugar', but it did aid the implementation of the compiler optimizer slightly.

7.3 Compilation and Optimization of Pseudo-Unification

We can simply interpret the unification statements as-is. However, we are using an implementation language which allows us to easily create and manipulate executable fragments of code

(LISP [143]). Because we can create fragments of code, we can compile the sets of unification equations into executable functions as part of the preprocessing of the grammar [97, 153, 98]. This technique is fairly common, appearing in many other systems, including DyPar II–IV [21, 97], GLR [152, 153], the KANT system [20, pp. 86–88], [110], GLR* [68], LCFlex [125], and the Core Language Engine [4].

	Search type	parse time	post-parse search time	Parse + Search time
Interpreted Unification	Fully-Interleaved Unification	316.89	*n/a*	316.89
	Shallow Unification with all-paths search	267.77	37.76	305.53
	Shallow Unification with single-best search	267.85	25.24	293.09
Compiled Unification	Fully-Interleaved Unification	68.35	*n/a*	68.35
	Shallow Unification with all-paths search	58.90	8.82	67.72
	Shallow Unification with single-best search	58.89	8.45	67.34

Table 7.1: Comparison of directly interpreting unification equations with compiling sets of equations into LISP. Results are total run time for Interleaved Unification, Shallow Unification plus All-Paths Post-Parse Search, and Shallow Unification plus Single-best Post-Parse Search, on the Catalyst ambiguity corpus (see Section 5.4). Variations in lexical look-up times and parse times for identical parse configurations are due to sampling noise.

Table 7.1 summarizes the run-time performance of our system, comparing programmatically interpreting the unification equations with translating them into LISP functions which are then compiled and loaded into the system.

It is not surprising that it is better to compile the unification equations into a more efficient form. What is surprising is the magnitude of the performance improvement. In every case, compiling the unification equations into machine code results in a speed-up of a factor of 4.5.

Clearly treating the unification equations as a "little (programming) language" [13, ch. 9] is useful. We continued this thought by investigating optimizations of the source language [1, 99]: prior to compiling the unification equations into some faster-executing form, we attempt to optimize the equation equations themselves. (This was done for the equations used in the test shown in Table 7.1.)

The possibilities for optimization are fairly broad. In general, any technique that has been developed for optimizing code in computer languages (such as loop invariant movement or strength reduction [1, 99]) may be applicable to unification optimization. (Obviously we do not have loops

as such, but it can be useful to move invariants out of disjunctions, which is a very similar problem.)

In the remainder of this section, we discuss several such optimizations in detail.

7.3.1 Unwinding of Conditional ORs

The conditional OR special form of our unifier causes special computational problems in the face of disjunctions in the unification values. Recall from Section 7.1.1 that the naïve method for evaluating conditional ORs involves turning the current values into disjunctive normal form by 'unwinding' any data disjunctions to the top-most level, and then applying the conditional OR to each of these terms in turn.

There is another option for correctly interpreting the conditional OR structure [99, sec. 8.7.1, pp. 236–244]. We can re-write the conditional form:

if A else B

using a normal (non-conditional) OR, along with the complement of a term, as:

A or ((not A) and B)

Of course, in order to successfully apply this sort of transformation, we must be able to compute the complement of the first term.

This transformation is fairly simple to express in a recursive form for conditional ORs of arbitrary length, following the logical form above. The compiler attempts to perform this transformation. If it is possible to calculate the complement of all but the last of the terms of the conditional OR (as in Section 7.2.5 above), then the compiler substitutes the structure above to the first term, and calls itself recursively for the remainder of the terms.

To see the need for this optimization, we ran two simple experiments.

The results shown in Table 7.2 show the effectiveness of the Conditional OR Unwinding optimization on natural data. It is of particular note that 19 of the test sentences failed to parse by "timing out" — requiring more than 10 CPU-seconds of run-time to parse. (In fact, the failure of the first of these sentences in the prior parser prompted this entire investigation.)

Using Data Disjunction Unpacking to evaluate conditional ORs still results in a parser that is an average of 12 times slower than one using Conditional OR Unwinding for the fully interleaved parse. For the fully-interleaved conditions of this comparison, 190 CPU-seconds of this slowness is

	Search type	Number of Sents Parsed	Parse Time	Search Time	Total time
Data Disjunction Unpacking	Fully-Interleaved Unification	1404	849.89	*n/a*	849.89
	Shallow Unification with all-paths search	1408	170.91	442.47	613.38
	Shallow Unification with single-best search	1423	169.82	39.31	209.13
EOR Unwinding	Fully-Interleaved Unification	1423	68.35	*n/a*	68.35
	Shallow Unification with all-paths search	1423	58.90	8.82	67.72
	Shallow Unification with single-best search	1423	58.89	8.45	67.34

Table 7.2: Comparison of unification data disjunction unpacking with unwinding conditional ORs in the grammar preprocessing. Results are total run time for Interleaved Unification, Shallow Unification plus All-Paths Post-Parse Search, and Shallow Unification plus Single-best Post-Parse Search, on the Catalyst ambiguity corpus (see Section 5.4). All sentences were run with a 10-CPU-second time-out.

due to the 19 outlier sentences that time-out before successfully parsing. However, even neglecting the sentences which actually time out, Data Disjunction Unpacking is still over 9 times slower than Conditional OR Unwinding.

From these results it is clear that we must avoid unpacking disjunctions in the data. The only configuration in which Data Disjunction Unpacking is competitive is the shallow first-pass parse followed by single-best unification value search configuration. And this case is a red herring, because Conditional OR unwinding *was* applied to the shallow first-pass unification equations, and the single-best search avoids creating the data disjunctions which must be unpacked.

To see what is happening in these outliers, we ran a simple experiment using the PP-attachment case (see Section 7.4.3 below).

Figure 7.1 demonstrates why conditional OR unwinding is superior to data disjunction unpacking of the unification values. For this input sentence, the data disjunction unpacking technique considerably extends the point-of-failure – Using Data Disjunction Unpacking we were unable to process 7 PPs due to running out of memory after several minutes of run-time. Using conditional OR unwinding extends the failure point out to 11 PPs, with 10 PPs taking 6.581 CPU-seconds.

Figure 7.2 demonstrates why we see a run-time blow-up much earlier when using Data Disjunction Unpacking than when using the conditional OR unpacking technique: unpacking the data (even when limited to only the slots used in the body of the conditional OR) causes the unification system to create much larger unification structures, which in turn cause the system to make many, many

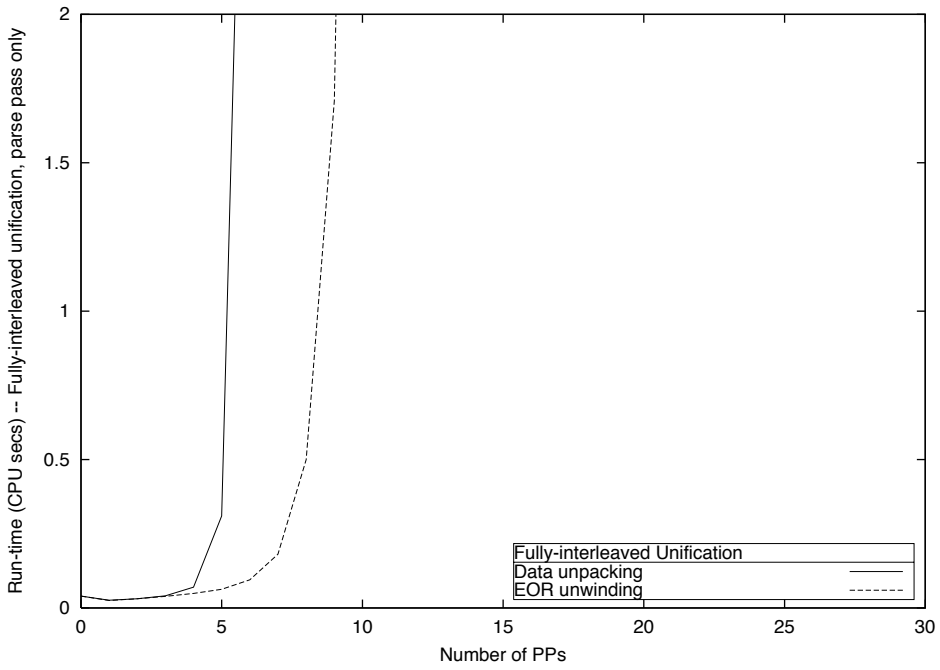


Figure 7.1: Graph of the number of prepositional phrases vs. the net run-time (in CPU seconds) for a parse using Fully-interleaved Unification, comparing unification data disjunction unpacking to unwinding conditional ORs in the grammar preprocessing.

more calls to the low-level unification routines.

7.3.2 Disjunction Flattening

The problem with the simple recursive implementation of Conditional OR Unwinding of Section 7.3.1 is that it leaves a piece of sub-optimal structure. The ordinary disjunctions generated by the recursive application of the conditional OR unwinder are right-trees such as:

```
A or (B or (C or (D or ...)))
```

In the prefix notation of the unification system, this is expressed as:

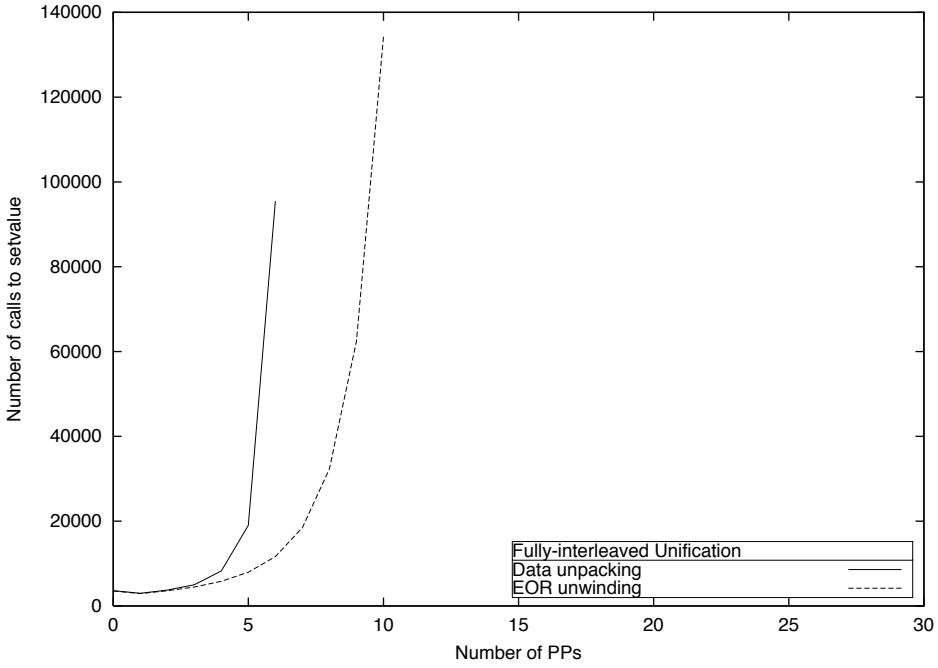


Figure 7.2: Graph of number of prepositional phrases vs. the number of calls to the `setvalue` function for a parse using Fully-interleaved Unification, comparing unification data disjunction unpacking to unwinding the conditional ORs.

```
(*OR* A
    (*OR* B
        (*OR* C
              (*OR* D
                    ...))))
```

Because the compiler is still relatively unsophisticated, and introduces a new unification value result context for every equation disjunction (as well as a new value context for every term of the disjunction), we found that we could eliminate some needless duplication of effort by flattening out these OR structures. Continuing our example above, the result of this flattening would be:

```
(*OR* A
      B
      C
      D
      ...)
```

7.3.3 Multiple-Value Strength Reduction

Our initial effort at final testing of the system using independent test data revealed a source of blow-up in the unification space not due to structural ambiguity. In the test, two sentences exhibited unusually bad run-time behavior. The worst exhibiting this behavior was:

```
The possible sources of positive voltage on connector
<code>J1</code> are contacts <code>J1</code>, <code>J4</code>,
<code>J5</code>, <code>J6</code>, <code>J10</code>,
<code>J11</code>, <code>J12</code>, <code>J13</code>,
<code>J14</code>, <code>J15</code>, <code>J16</code>,
<code>J17</code>, and <code>J19</code>.
```

Both of these outlier sentences exercise a particular feature of the pseudo-unification system: multiple value push. The unification system allows one to specify that a slot can take on multiple values. For example, consider the rules:

```
(<np> <== (<np&> <comma?-coord-conj> <np>)
    (...
     (x0 = x1)
     ((x0 conj) = x2)
     ((x0 member) > x3)))

(<np&> <== (<np&> *comma* <np>)
    (...
     (x0 = x1)
     ((x0 member) > x3)))

(<np&> <== (<np>)
    (...
     (x0 member) > x1))
```

where the `<comma?-coord-conj>` rule specifies an optional comma followed by one of several conjunctions such as *and* or *or*.

These rules specify that the `member` slot of the head will take the value of the last child (the `<np>`) simultaneously with any previous value it might have. These values are not unified together, but instead are stored in this slot together as an unordered set.

The original intent of the previous authors of this unification system was that when a list such as:

foo, bar, and baz

the unifier should create a structure of the form:

```
(member (*MULTIPLE* ((ROOT "foo") ...)
                    ((ROOT "bar") ...)
                    ((ROOT "baz") ...)))
```

These multiple-value sets cause problems because they interact poorly with disjunctions in the current unifier. Suppose that there is rule in the grammar which is superior to the above <np> rule, and that this superior rule checks some element in the member slot. To correctly evaluate this check, the unification system will have to check that the equation is true for each of the multiple values.

Recall that for rule disjunctions (*OR* or *EOR*), the unification system could produce a different value for each clause of the disjunction. If, in this superior rule, the checking equation is contained within a rule disjunction, we could end up with a data value such as:

```
(member (*OR* (*MULTIPLE* (...) (...))
              (*MULTIPLE* (...) (...))
              ...
```

In the actual grammar rules, rather than having a rule production which is a parent of the <np> rule, the above rules themselves contain disjunctions. This interaction between rule disjunctions and multiple values causes an explosion of nested (*OR* (*MULTIPLE* ...)) structures. So for a three element list such as:

foo, bar, and baz

the unifier will create a structure of the form:

```
(member (*OR* (*MULTIPLE* (...) (...))
              (*MULTIPLE* (...) (...))))
```

For a four element list (*"foo, bar, baz, and foob"*), the structure will have an additional level:

```
(member (*OR* (*MULTIPLE* (*OR* (*MULTIPLE* (...) (...))
                                (*MULTIPLE* (...) (...))
                          (*OR* (*MULTIPLE* (...) (...))
                                (*MULTIPLE* (...) (...))
              (*MULTIPLE* (*OR* (*MULTIPLE* (...) (...))
                                (*MULTIPLE* (...) (...))
                          (*OR* (*MULTIPLE* (...) (...))
                                (*MULTIPLE* (...) (...)))))))))))
```

For every additional item in the list, and additional level of tree is created. The predictable result is an exponential explosion in the unification space.

Initial Solution

In order to control blow-ups of unification values of this sort, we investigated changing the representation of multiple-value unification values.

Rather than represent the multiple values as composite entry using `*MULTIPLE*`, we changed the representation to be a linked-list of unification structures. We created a `PREVIOUS` slot, and used the unification optimizer to replace all calls of the form:

```
((x0 SLOT) > xn)
```

With the sequence of unification equations:

```
(*or* (((x0 SLOT) = *DEFINED*)
   ((x0 previous-SLOT) = (x0 SLOT))
   ((x0 SLOT) = *REMOVE*)
   ((x0 SLOT previous) = (x0 previous-SLOT))
   ((x0 previous-SLOT) = *REMOVE*))
  (((x0 SLOT) = *UNDEFINED*)))
((x0 SLOT) = xn)
```

(where `SLOT` refers to some slot name, and `xn` refers to some top-level variable name.)

The effect of this new representation is that the for the initial assignment of a value to some slot, it is simply assigned. However for each additional assignment, the previous value is saved under the `PREVIOUS` slot.

This change in representation cooperates much more with the handling of disjunctions. A test run using this change in representation reduced the run-time of these two outliers from in the 2–6 second range to under 0.2 seconds. Further, with this modification to the grammar, we were able to parse the same basic sentence with triple the number of conjoined terms in only slightly more than triple the total run-time.

7.4 Shallow Pseudo-Unification as a First-Pass Filter

Recall from the system overview (Chapter 5) that for the first-pass parse, we do not need a complete or even exact unification value, but instead only want to perform enough unification to provide an effective filter for the bottom-up parser. In Chapter 8, we will discuss how we calculate an exact, preferred unification value from this over-covering parse forest.

Also, we must avoid the problems of unification-space "blow-up" noted in Chapter 3. When a complex-valued unification grammar creates a unique representation for each distinct parse tree, a system using this grammar which calculates all possible unification values for each input will suffer an exponential blow-up in run-time when given a highly structurally ambiguous input.

We wanted a unification system which avoids blowing up because of having a unique representation for every distinct parse tree. The most straight-forward way to avoid the blow-up problem is to make a grammar in which the unification values which can be efficiently packed together.

In order to fulfil this, we adopted a variant of the restriction technique proposed by Shieber [136] and Harrison and Ellison [45]: we took a subset of the full unification grammar. That is, we used a number of techniques to transform the unification equations of the full grammar into a *shallow grammar*: a unification grammar which uses only simple values or disjunctive sets of simple values.

We will next discuss our variant of negative restriction and how this is used to produce the shallow unification grammar. After this we will describe a set of non-exact optimizations that are applied to the shallow unification grammar. We then discuss the approximate packing technique which allows us to ensure that unification values will always pack efficiently. Finally, we examine the effectiveness of this approach experimentally.

7.4.1 Wild-carding deep structure assignments

The key to developing a collection of grammar rules which allow only for shallow unification is to re-write the grammar rules which either create or use deep unification structures so that they do not

do so.

We apply five types of modifications to the original (possibly deep) unification equations in order to form the shallow equations:

- For unification of one of the parent's slots with an entire child structure, the child structure is replaced by a wild-card. This is the primary way in which we eliminate deep structures — by failing to create them.

  For example, the equation:

  ```
  (X0 OBJ) = X2
  ```

  would be replaced by the equation:

  ```
  (X0 OBJ) = *WILD*
  ```

  After this replacement, any long-path unifications with this particular `OBJ` slot will succeed because of the semantics of unifying with a wild-card value.

- Long unifications or assignments to the parent structure (X0) are shortened with a wild-card assignment. This is because the deep structure under test will not exist.

  For example,

  ```
  (X0 FOO BAR) = BAZ
  ```

  becomes

  ```
  (X0 FOO) = *WILD*
  ```

- Checking unification of any long left-hand-side path is removed.

  This is a special optimization on the above case. We could just shorten a checking unification as we do above for normal unification and explicit assignment. But unlike those two equations, checking unifications are defined to have no side-effects. So rather than include an equation which will always succeed, we will simply remove it as redundant (but see Section 7.4.2 on equation removal below).

- Unification with a long right-hand-side path is wild-carded.

  Suppose we have the equation:

```
(X0 FOO) = (X2 BAR BAZ)
```

Because the right-hand-side of this equation is a reference to a long path, we can safely assume that the primary slot `BAR` of this path will already have been wild-carded (as above). Therefore we know that the value we will retrieve when this is evaluated will always be `*WILD*`. Because of this, we can simply save the work of the look-up, and substitute in the wild-card as a compile-time constant:

```
(X0 FOO) = *WILD*
```

- All multiple-value pushes

  For the grammars which we have examined, multiple-value push equations such as:

  ```
  (X0 MOD) > X1
  ```

  are often structure-building, and in any case create the sorts of distinct values which cause us packing problems and thus lead to unification-space blow-ups.

  We replace all such push equations with a wild-card unification against the first two elements of the path:

  ```
  (X0 MOD) = *WILD*
  ```

As we noted in Section 4.3.1, we do *not* remove grammatical head assignments such as:

```
X0 = X1
```

since these do not induce unification-space blow-ups by themselves, and wild-carding them would render the unifications essentially moot as a whole — essentially relaxing the grammar to simply its context-free spine as a result.

7.4.2 Pseudo-Optimizations for Shallow Unification

We could simply apply the deep-structure-removal transformation described above and compile the result, but we can be slightly more clever than this. Because we know that we will not be using the unification values from the shallow unification grammar beyond the first pass, but will instead recalculate our final unification value from scratch, we can safely make further modifications to the shallow unification grammar. We will now examine a number of optimizations and *pseudo-optimizations*: modifications to the grammar which improve the run-time behavior of the system but change the resulting unification values.

Useless Unification Killing

By finding information about the way unifications are used in the entire grammar, we can identify some special equations for removal or other optimization. In order to accomplish this, we perform two passes on the grammar.

Information Collection Before we can optimize the unification equations, we first collect information about them. We are primarily interested in identifying the range of values which each unification slot can potentially take.

Recall from Section 7.1.1 that there are two fundamental types of unification equations:

- *(path feature) = value*

  We use these to calculate *feature-value* pairs. If, for instance, we see an equation such as `(X1 FOO) = +`, we know that `+` is one of the possible values which slot `FOO` could take.

- *(l-path feature1) = (r-path feature2)*

  We refer to these as *feature-feature* unifications. Should we encounter an equation `(X0 BAR) = (X1 FOO)`, we know that slot `BAR` could take on any value which slot `FOO` could take, and vice-versa.

In the first pass, we apply the conditional-OR optimization described above in Section 7.3.1, then prefilter the sets of unification equations to remove deep-structure references as above, and finally walk through the resulting shallow unification equations, collecting up the ranges of values, and the feature-feature assignments which are encountered.

After we have collected up this information for the entire grammar, we take the symmetric transitive closure of the union of unification slot ranges according to the feature-feature relation. So for example if we had encountered the unification equations:

```
(X1 FOO) = +
(X0 BAR) = (X1 FOO)
(X2 BAR) = -
```

We could conclude that both of the slots `FOO` and `BAR` can take the values: {`+`, `+`, *undefined*}.

Useless Unification Removal Once this information has been collected, the transformed, filtered grammar is again traversed in order to actually produce the final compiled unification values.

Any unification equations which involve a slot which is only unified with the value `*WILD*` or is removed (*i.e.* set to *undefined*) is simply deleted from the shallow-unification grammar, because we can conclude that this unification slot is never usefully tested within the confines of the grammar. (See Section 7.4.2 on equation removal below).

Similarly, for any equation which involves a slot which is only ever unified with the value `*WILD*`, or tested for existence, non-existence, or is removed (set to *undefined*), the assignment to `*WILD*` can be changed to any single (globally consistent) value. This optimization is of value because we know that this slot takes only two values, and thus could potentially be compiled into a very space-efficient form.

OR Strength Reduction

We identified several optimizations to perform on disjunctions of unification equations.

Empty disjunction removed. Disjunctions (both normal `*OR*` and conditional `*EOR*`) that contain *no* statements are simply removed. (But see Section 7.4.2 below.)

Single-term disjunction reduced to single term. For disjunctions with only a single list of statements, that list of statements is substituted in place of the disjunction. For example, the disjunction:

```
... ((*OR* (statement1 statement2) )) ...
```

becomes the sequence:

```
... statement1 statement2 ...
```

Wild-carded parallel path removal in disjunctions. When the wild-card removal described in Section 7.4.1 is applied to the KANT grammar [55], an interesting structure is generated. For example:

```
((*OR*
   (((X0 FOO) = BAR))
   (((X0 FOO) = QUUX))
   (((X0 FOO) = *WILD*))))
```

The interesting feature of this structure is that it contains a disjunction over a set of single statements, all of which contain the same left-hand-side, and one of which contains an assignment to *WILD*.

Since an assignment to *WILD* will successfully unify with anything, we can safely replace this entire disjunction with the single statement:

```
((X0 FOO) = *WILD*)
```

The exception to this is that if one of the statements contains the non-existential unification *UNDEFINED*, it cannot be removed. So the structure:

```
((*OR*
   (((X0 FOO) = *UNDEFINED*))
   (((X0 FOO) = BAR))
   (((X0 FOO) = QUUX))
   (((X0 FOO) = *WILD*))))
```

could only be optimized into:

```
((*OR*
   (((X0 FOO) = *UNDEFINED*))
   (((X0 FOO) = *WILD*))))
```

This optimization is actually performed in two steps: first all statements other than the one containing the *WILD* (and any containing *UNDEFINED*) are removed, and then the single-term disjunction optimization (above) applies.

Unification Equation Removal

Above we have seen two different ways in which unification equations can be removed. When the equations in question are part of a simple top-level list, this removal can simply proceed.

The conditional-ORs optimization (described in Section 7.3.1) creates both a problem and an opportunity for further optimization.

Complements of empty statements An empty statement — the result of removing one or more equations — can be considered to be a statement which always succeeds with no side-effects. Seen in this light, the converse of an empty statement is obviously an unconditional failure.

```
not(*SUCCEED*) --> *FAIL*
not(*FAIL*) --> *SUCCEED*
```

This is fine as-is, but we must then correctly handle this unconditional failure statement.

Failures in lists of equations Taking the complement of a disjunction containing a removed set of equations will give us a conjunctive list of equations including a failure statement (thanks to DeMorgan's Law). Since a list of equations will only produce side-effects if the equations in that list all succeed, we can replace such a list with a single failure:

```
(... *FAIL* ...)
```

becomes simply:

```
*FAIL*
```

Ordinary OR optimization Just as we can delete empty (or unconditional success) statements from a conjunctive list of equations, we can also remove unconditional failure statements from a disjunction:

```
(*OR* A B *FAIL* C)
```

becomes:

```
(*OR* A B C)
```

If we were creating an optimizing compiler for a programming language which was free of side-effects, we could also simply substitute a succeed statement in place of a disjunction containing a succeed statement. Unification (even pseudo-unification) is not this simple: because statements can have side-effects, we must properly evaluate the terms in a disjunction.

On the other hand, accounting for these same side-effects causes us to keep the side-effect-free success statements in a disjunction (or at least keep one of them). This is because these success

statements have the effect of allowing the disjunction to succeed without changing the unification environment. For example, the disjunction:

```
(*OR* A B *SUCCEED* C ...)
```

becomes:

```
(*OR* A B C ... (do nothing) )
```

Conditional OR optimization Like the ordinary disjunction, we can simply remove unconditional failures from a conditional-OR:

```
(*EOR* A B *FAIL* C)
```

becomes:

```
(*EOR* A B C)
```

Because of the conditional nature of a conditional-OR, there is an additional optimization which can be performed for unconditional success statements. Any success causes the conditional-OR to "short circuit" and stop evaluating its terms. Thus the unconditional success statement will always cause the conditional-OR to "short-circuit" when it is evaluated; any disjunctive clauses after this unconditional success will always be ignored. So these "dead" clauses can be deleted:

```
(*EOR* A B *SUCCEED* C ...)
```

becomes:

```
(*EOR* A B *SUCCEED*)
```

Approximated Unification Packing after Unification Function Evaluation

In Section 4.3.1, we discussed a method to approximately pack unification data disjunctions by ignoring inter-slot dependencies and pushing the disjunctions down to the leaf level of the unification value tree. This technique is helpful for assembling a single unification value from nodes which are being packed together in the parser as part of the local ambiguity packing.

Because both ordinary- and conditional-OR statements can cause a proliferation of disjoined unification values, we found that it was also helpful to apply this approximate packing technique to the final results of applying the shallow unification equations of a grammar rule to its children.

We also investigated adding this approximate re-packing everywhere, but this did not help — see Section 7.5.1 below.

Removal of the Implicit Interdependency of Top-Level Unification Variables

As we discussed in Section 7.1.1 above, an interdependency of the individual top-level variable values on each other can develop in the exact unification system.

We also found that we could ignore this interdependency at the top-most level in the first-pass. This is useful because there are many grammar rules which contain only a single list of a very small number of unification equations. An example of such a rule is one of the rules for handling noun phrases with number words:

```
(<NP> <==> (<Num> <N>)
       (((x2 numword) =c +)
        (x0 = x2)
        ((x0 card) = x1)))
```

If we just assume that there are no subtle interdependencies amongst the top-level unification variables, then we can simply store the values of these top-level variables in *e.g.* LISP variables. By doing this, many simple equations become very simple indeed; for example, the lexical-head setting rule:

```
(X0 = X1)
```

can be compiled down to the LISP statement:

```
(setq X0 X1)
```

which in turn will be compiled down into very simple machine code: a single pointer assignment.

This represents much less work than the exact unifier perform, since the exact unifier must iterate over all sets of trees representing associated variables. Even when there is only one set of values, executing `(X0 = X1)` involves at least two, one-link tree traversals and a `CONS` or two.

7.4.3 Effectiveness of Shallow Approximate Unification

We have examined several ways to 'cheat' in order to reduce the work of the first-pass unification system while attempting to remain close to level of pruning which the full (*i.e.* non-shallow) unification equations give us.

We will use three different measures to gauge the effectiveness of these techniques. Of course the primary criteria is the total amount of time spent parsing and searching the sentences. Any technique which saves some time in the first-pass parse, but which expands the size and structure of the parse forest in such a way that the post-parse search must spend drastically larger amounts of time searching for a result is not a net win, and thus is not preferable.

Minimizing over-all run-time also depends on the second pass search being fast (as well as effective). We will examine how to perform this search in considerable detail in Section 8; for this chapter we will simply take the best single-value and all-values searches as constants, and examine how to best prepare a parse forest for them.

If a first-pass unification does not overly slow the second-pass search, then we can consider two other criteria: the amount of work which is done by the first-pass unification system, and the size of the resulting parse forests. Our ideals are for the amount of work which goes into the unification to be low, and the size of the parse forests to be barely larger than those produced using the full (*i.e.* not shallow) interleaved unification.

Size of parse forests

There are actually two different ways we can measure the size of a parse forest: we can measure the total number of nodes used to represent the forest, and we can calculate the total number of parse trees represented by the forest.

The total number of nodes used to represent the forest tends to under-represent the amount of work which the post-parse search will need to perform. Although the all-values post-parse search performs a simple depth-first traversal, and thus visits every interior node twice (once while descending, once while ascending), the amount of unification work performed is *not* linear in the number of nodes visited.

(As we noted in Section 2.1.5, if the amount of work was *always* linear, or even polynomially proportional, to the number of nodes in a packed parse forest, we would have a constructive proof that $\mathcal{P} = \mathcal{NP}$. Alas, while our search techniques are effective, they are not *that* effective.)

For the single-value search, in order to eliminate an interior node from the search, the entire sub-

tree represented by this node must be searched: every possible combination of unification values from all of the children must be considered before the node can be judged to be hopeless and permanently removed from the search.

On the other hand, because both searches calculate and re-use partial values, neither has to attempt to unify every single parse tree from scratch. So the number of trees represented by the parse forest over-represents the amount of work which the search will need to perform.

We will cautiously use the total number of nodes measure with these considerations firmly in mind.

Artificial Data We will first examine an artificial example which was constructed to force a prepositional-phrase attachment ambiguity. The following is a close variant of an example used by Moore and Alshawi [95], as well as Oepen and Carroll [115].

```
Grease flows through the bearing seal (in the engine)*.
```

For this sentence, we examined the relationship between one of several measures of parser efficiency and the number of repetitions of the phrase "in the engine", contrasting Full interleaved Unification, Shallow Unification, and strictly Context-Free parsing.

Figure 7.3 compares the number of prepositional phrases in the input against the size of the parse forest which was created in the process of parsing, for each of Full Interleaved Unification, Shallow Interleaved Unification, and Context-free (*i.e.* no unification).

For this particular family of sentences, parsed using the KANT grammar, full interleaved unification parsing and shallow unification produce the same sized parse trees up to a limit of ten PPs. We were only able to measure full interleaved unification for up to ten PPs, whereas we were able to measure shallow unification and strict context-free parsing well beyond the limit of thirty displayed in this graph. The reason for this is a blow-up in overall run-time, which we will examine below in Section 7.4.3.

The unfiltered context-free grammar produces an increase in the size of parse forest which increases very slightly with the number of prepositional phrases in the sentence. This increase is due to an ambiguity in the base sentence (without any additional prepositional phrases) which is prevented from occurring by the filtering action of both types of unification.

Real Data We can also make these same comparisons on real data. For this we chose the Catalyst Ambiguity corpus (described in Section 5.4). Briefly, these are 1447 natural sentences from heavy

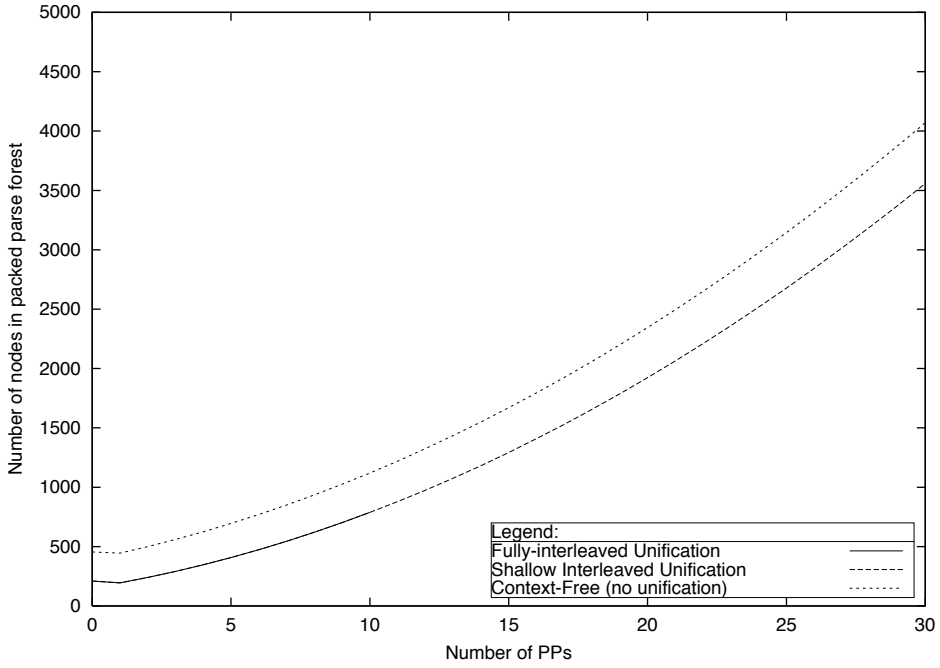


Figure 7.3: Graph of the number of prepositional phrases vs. the number of nodes created in the parse tree: Full Interleaved Unification, Shallow Interleaved Unification, and Context-free (*i.e.* no unification).

equipment manuals; these sentences were chosen because each has some form of ambiguity.

Table 7.3 compares the total number of nodes created in running the Catalyst ambiguity corpus for full interleaved unification parsing, with the parse passes of both shallow unification parsing and context-free parsing (without any unification).

This table shows two very interesting things:

- Full interleaved unification prevents the creation of about half of the nodes which would be created according to the context-free backbone of this grammar.
- Shallow unification is almost as good at filtering as full interleaved unification, allowing only a small number of nodes to be created which full interleaved unification would have filtered out.

Figure 7.4 shows a scatter-graph of the number of nodes created in the parse forest for each sentence in our test-set. This graph compares full interleaved unification with both shallow unification and a context-free (*i.e.* no unification) parse. We can see two interesting things in this graph. First, full interleaved unification substantially prunes the parse for virtually every test sentence. Sec-

Search type	total num. nodes	percentage compared to Interleaved U.
Interleaved Unification	466,945	
Shallow Unification	467,919	100.2%
Context-Free (No Unification)	906,573	194.0%

Table 7.3: Comparison of number of nodes created for Interleaved Unification, Shallow Unification (parse only), and Context-Free (no unification; parse only), on the Catalyst Ambiguity corpus (see Section 5.4).

ond, shallow interleaved unification is doing *almost* as good a job of pruning as fully interleaved unification. (These per-sentence results serve to confirm the over-all results shown in Table 7.3.)

Unification effort

The size of the parse forests is a rather remote measure: we are more interested in measuring (and reducing) the amount of work which the system performs more directly.

We found that a good way measure the total amount of effort required to complete the unification on an input was to measure the number of calls to a low-level routine in the unifier. The routine we chose to measure is called `setvalue`. This function takes an input tree, an operation to apply to the tree, and a value (possibly expressed as a path of slots) to use with that operation, and returns a tree which is the result of that operation. `setvalue` is implemented recursively, so that if the value is a path (rather than an end-value), `setvalue` will search the tree for the first slot element specified in the value, and if this is found recursively call itself on the sub-tree and the remainder of the value.

We estimate the total amount of unification effort by counting the number of times `setvalue` has been called. In general, the fewer number of times that `setvalue` was called, the lower the amount of unification work, and thus the faster the parse will run.

Artificial Data Again we will first consider results obtained from our artificial ambiguity case. Here we only consider Full and Shallow interleaved unification, since Context-Free parsing does not make any use of unification, and thus does not make use of this function.

Figure 7.5 begins to demonstrate why there is a problem with full interleaved unification. For this input sentence, full interleaved unification suffers from a steep increase in the number of calls to `setvalue` as we approach ten repetitions of the PP, but shallow unification does not suffer from this problem.

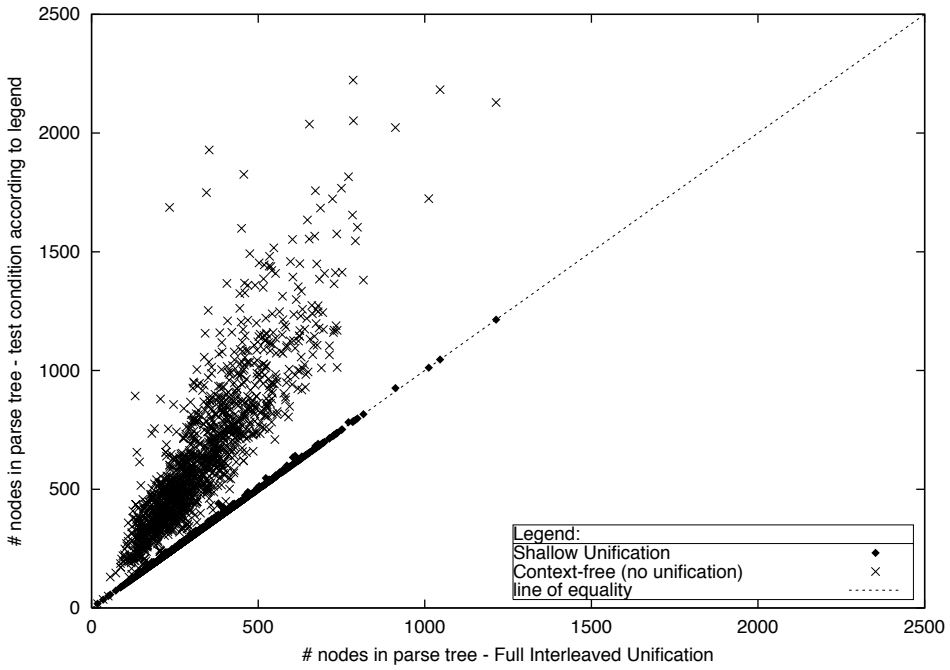


Figure 7.4: Per-datum scatter-graph of number of nodes created during parse: Shallow Interleaved Unification vs. Full Interleaved Unification, and Context-free (*i.e.* no unification) vs. Full Interleaved Unification.

Real Data We have seen how the shallow unification technique does not blow up for a particularly bad case, but we might ask how this technique performs on real inputs.

Table 7.4 lists the total amount of unification work performed for processing the Catalyst ambiguity corpus. These results are encouraging, because the shallow unification performs only two-thirds of the amount of unification work that full interleaved unification requires, while pruning almost as effectively (as shown in Figure 7.4)

Figure 7.6 shows a scatter-graph of the number of calls to `setvalue`, comparing the full interleaved unification parse technique to the parse portion of a shallow interleaved parse. This graph shows that by this measure, the shallow unification technique saves a substantial amount of unification effort in the parse pass.

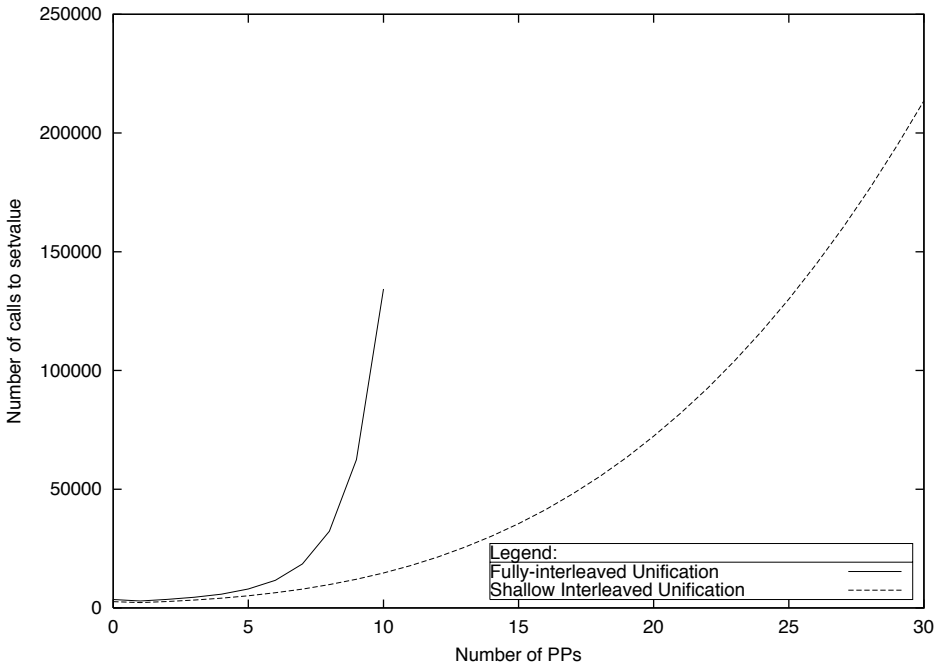


Figure 7.5: Graph of number of prepositional phrases vs. the number of calls to the `setvalue` function for Full Interleaved Unification and Shallow Interleaved Unification.

Search type	Total calls to `setvalue` in first pass	percentage compared to Interleaved U.
Full Interleaved Unification	8,150,477	
Shallow Unification	5,346,790	65.6%
Context-Free (No Unification)	*n/a*	0.0%

Table 7.4: Comparison of number of calls to `setvalue` for Interleaved Unification, Shallow Unification (parse only), and Context-Free (no unification; parse only), on the Catalyst ambiguity corpus(see Section 5.4).

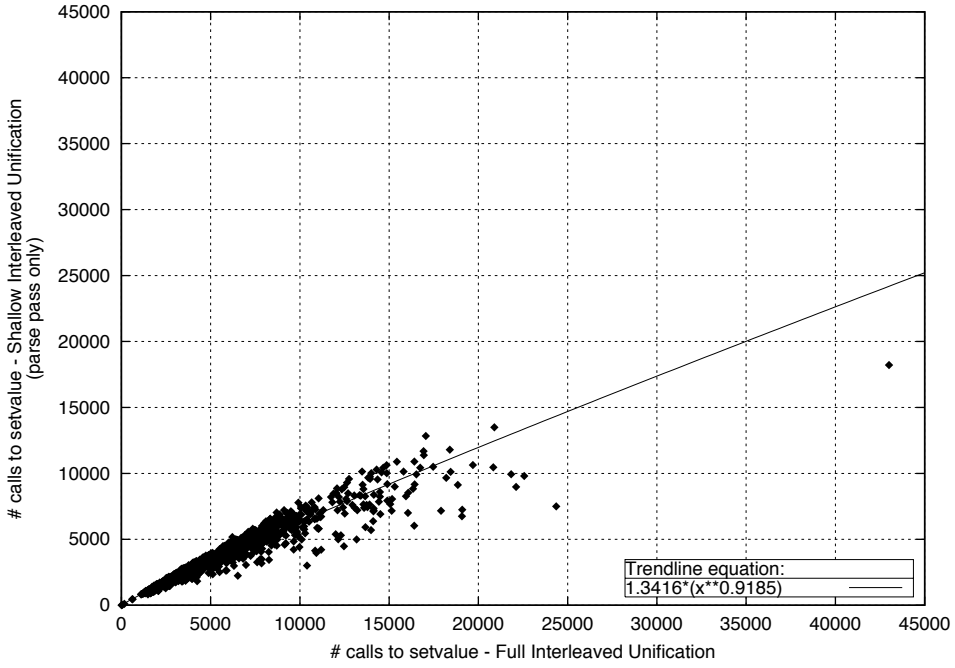


Figure 7.6: Per-datum scatter-graph of number of calls to the `setvalue` function: Shallow Interleaved Unification (parse-pass only) vs. Full Interleaved Unification. The trend line, $y = 1.3416x^{0.9185}$, fits with an R^2 confidence of 0.95.

Net Run-time

The measures which we have discussed above are fine and useful, but our principle goal is to reduce the amount of *time* required to fully process the data. In order to improve in this way, we must also measure it directly.

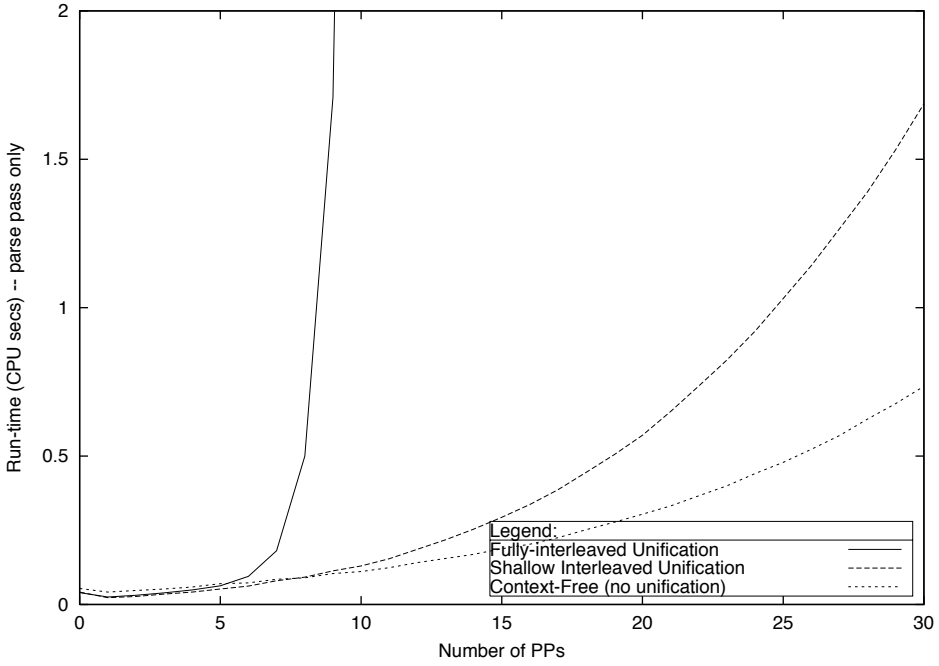


Figure 7.7: Graph of the number of prepositional phrases vs. the net run-time (in CPU seconds) of the parse pass *only*, comparing Full Interleaved Unification, Shallow Interleaved Unification, and Context-free (*i.e.* no unification).

Artificial Data In Figures 7.7 and 7.8, we can see one way in which problems develop because of ambiguity. The run-time for all parsing variations starts under 55 CPU-milliseconds for no additional PPs, and has only risen to 100 milliseconds for six PPs. After this point, full interleaved unification suffers from a sharp increase in run-time, reaching 1.7 seconds (1700 milliseconds) for 9 PPs and 6.6 seconds for 10 PPs.

In contrast, neither shallow interleaved parsing nor strictly context-free parsing suffer from this blow-up in run-time for the parse pass – context-free parsing requires only 0.74 seconds of run-time for the thirty PP case, and shallow interleaved parsing requires 1.7 seconds.

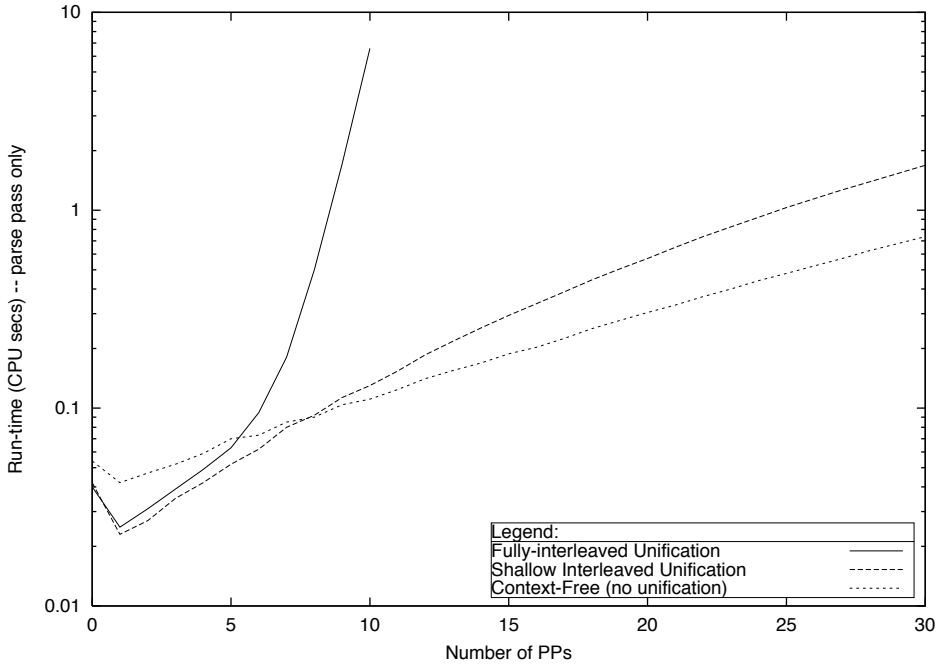


Figure 7.8: Graph of the number of prepositional phrases vs. the net run-time: same data as Figure 7.7, but with run-time plotted on a logarithmic scale.

Real Data After considering an artificial worst-case, we must consider real (*i.e.* natural, rather than created) inputs.

Table 7.5 compares the total run-times for three equivalent cases: full interleaved unification, shallow unification parse with a post-parse search for all possible F-structures, and context-free parse with an all-values post-parse search. For these data, the average run-times of full interleaved unification and shallow unification with the post-parse search are approximately equivalent, but starting with a strictly context-free parse is substantially slower than either interleaved unification approach.

Figure 7.9 shows a per-sentence scatter-graph of the CPU time required to parse (but not search), comparing full interleaved unification with shallow unification. This graph shows that compared to fully interleaved unification, the shallow unification technique usually saves around 23% ($1/0.8116$) of the over-all run-time of the parse pass. (Table 7.5 shows that the average savings is 16% of the over-all run-time).

Search type	parse time (CPU secs.)	post-search time	Total time
Full Interleaved Unification	68.35	*n/a*	68.35
Shallow Unification with all-paths search	58.90	8.82	67.72
Context-Free (No Unification) with all-paths search	87.40	18.99	106.39

Table 7.5: Comparison of parse times for Interleaved Unification versus Shallow Unification plus All-Paths Post-Parse Search, on the Catalyst ambiguity corpus (see Section 5.4). The 1% difference in total run-time is not considered significant.

Conclusions

From these results, we can conclude that the shallow unification technique achieves the goal we have set for it: this technique avoids the unification blow-up problems of full interleaved unification for our most common case of ambiguity, while substantially preserving the pruning effect of using full interleaved unification. We can also see that this technique does not substantially increase the run-time cost of the parse phase of processing — in fact, it saves a bit of parse-pass run time. Since this technique is saving time by doing less actual work, we will also have to efficiently complete the work of finding a parse.

But before we move on to see how we will attack this latter problem in the next chapter, we will briefly review two potential optimizations to shallow unification which did not turn out to be fruitful.

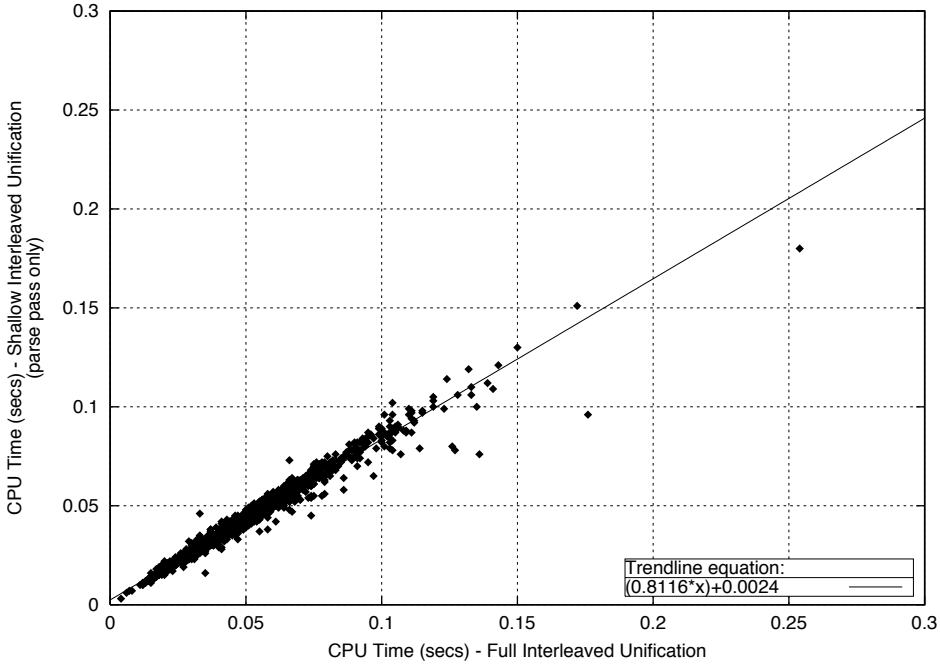


Figure 7.9: Per-datum scatter-graph total run-time (in CPU seconds) of the parse-pass for Shallow Interleaved Unification (parse-pass only) compared to Full Interleaved Unification. The trend line, $y = 0.8116x + 0.0024$, fits with an R^2 confidence of 0.97.

7.5 Optimizations That Did Not Help

Like any research project, there are some ideas which seem promising when first proposed, but on further investigation do not turn out to be fruitful. In this section we consider three such ideas.

7.5.1 Approximated unification packing in disjunctions

In Section 7.1.1 above, we discussed how disjunctive statements cause data disjunctions.

We attempted to push the approximate packing technique to its logical conclusion by applying this technique to the end result of every equation disjunction (both ordinary and conditional).

The choice of where and when to re-pack data disjunctions is a particular example of a classic trade-off: on the one hand, re-packing helps prevent unification-space blow-ups by converting the unification values to a smaller representation. On the other hand, re-packing takes some time to perform, and it is quite possible to do it more often than necessary for optimal performance.

As we noted above in Section 7.4.2, we found that it was beneficial to re-pack the results of applying an entire grammar rules worth of unification equations. On the other hand, we also found that adding an explicit re-packing step after the application of every (equation) disjunction (*i.e.* `*OR*` or `*EOR*`) statement was actually more re-packing effort than necessary.

7.5.2 Length limits in approximate packing

Examination of the behavior of approximate packing suggested another possible optimization. In some circumstances, the number of clauses in an approximate disjunction becomes very large. These long disjunctions tend to to slow the evaluation of unification equations with paths running through them because the unifier must loop over the disjunction, attempting the unification operation for each clause in turn.

We can easily impose a limit on the length of these disjunctions by simply replacing any long disjunctive value with a wild-card.

For example, suppose that we are going to pack together the three F-structures `((A B) (C D))`, `((A X) (C Y))`, and `((A Y) (C Y))`. Further suppose that we are applying the approximate disjunction technique described above, and we have imposed a length limit: no disjunction my contain more than two terms.

After packing the first two F-structures, we will have:

```
((A (*OR* B X))
 (C (*OR* D Y)))
```

But when we try to pack in the third F-structure, we will exceed our disjunction length limit for the feature `A`. So instead we will replace this disjunction with the wild-card `*WILD*`. The result will be:

```
((A *WILD*)
 (C (*OR* D Y)))
```

Using these wild-cards to impose a length limit can eliminate the possibility of exponential blow-up in the face of ambiguity, but will cause the F-structures to be over-generated even more than the approximate packing technique.

Unfortunately, this length limitation did not improve the performance of our system. A test of this technique on the Catalyst ambiguity corpus, using a shallow packing length limit of 2, showed no measurable change in run-time. Imposing the limit increased the total number of calls to `set-value` by 0.28%, virtually all of this increase was in the first pass. The over-generation effect caused a slight (0.2%) increase in the number of nodes created, which in turn increased number of nodes searched in the second pass by 0.7

7.5.3 Vector Unifier (is not faster)

In this system, the shallow unification is done with *almost* the same tree-structured unifier that the complex-valued unification uses. (The main difference is in the binding of top-level variables, which destroys the implicit interdependency of these top-level variables as described in Section 7.4.2 above.)

However, unlike the complex-valued unification which will be used in the post-parse-search (Chapter 8), we know that the shallow unification values are just that: shallow. Also, from the precompilation analysis described above in Section 7.4.2, we know that many of the unification slots take a bounded range of values.

We want the shallow unifier to be as fast as possible. Because of these considerations, we investigated the use of a radically different data structure for the shallow first-pass unifier: a vector unifier.

In a vector unifier, where each value of a slot can be one (or more) of a small range of values, the data is represented as a vector of bits. Each possible value of every slot is assigned a unique

offset in the vector of bits; a 1 bit indicates that the appropriate slot has the given value set, whereas a 0 bit indicates that this slot does not have this value. The offsets of the bits are calculated during compile time [12, p. 146], [78, pp. 257–259], [99, pp. 759—760].

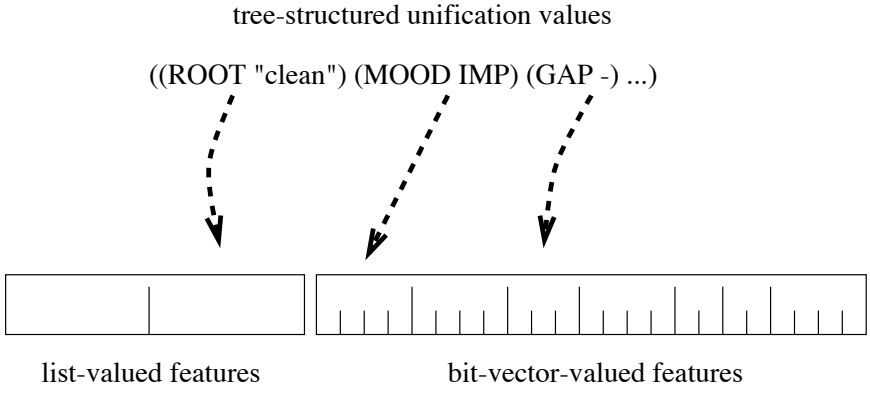


Figure 7.10: Unification Tree-Structured Value to Bit-Vector Mapping

Figure 7.10 shows an example of mapping between values in a tree-structured unification value and values in a structure containing both fixed, list-valued slots and bit-vectors.

Unfortunately, we found this technique to cost more time than it saves. Recall from Section 7.1.1 above that we must often make a copy of the current environment before beginning to evaluate a new list of equations, and that we must make several copies of the current values when interpreting a disjunction special form (one for each subordinate list of equations, plus one for the over-all results).

Since our tree-structured unifier is implemented using a copy-on-modification heuristic, initializing these new environments in the tree unifier is a single pointer assignment. Even when we are storing the top-level variables individually, as described in Section 7.4.2 above, we need only make a single pointer assignment for each variable. And since these refer to the parent and each of the child nodes of the application of a rule, there are only small number of them.

Initializing a bit-vector is considerably more work – since the bit vector contains only values, we cannot directly keep the copy-on-write behavior (at least, not without storing a separate reference count [78, pp. 345–346] or some similar mechanism), so we must copy the entire vector. (Even if we did keep a reference count, we would have to copy the *entire* vector the first time we actually modified an individual value, so the use of reference counting is unlikely to improve this situation.)

We found that the time saved in references and unification-value tree traversal was more than used up by value copying: thus this technique used more time than it saved.

Chapter 8

Post-parse Search

> Search well and be wise, nor believe that self-willed pride will ever be better than good counsel.
>
> — Aeschylus (525456 B.C.), *Prometheus Bound* [26, quote no. 3105]

8.1 Introduction

We have seen above in Chapter 3 that attempting to perform a whole parse, with interleaved full unification, can easily lead to an exponential blow-up of the parser. In this chapter, we investigate how to avoid this problem by stepping around it: rather than performing the unification interleaved with the Context-Free parse, we will delay performing the unification until after the parse is complete.

We will see that by stepping around the problems of interleaved unification, we can achieve run-time performance comparable to the interleaved system in the average case, while avoiding the exponential blow-ups that adversely affect the interleaved system. We will also see that the process of computing a single, preferred, unification result is not nearly as simple as one might first imagine it to be.

Before we can examine results, we should better understand what the post-parse search must accomplish, and how it does so.

Requirements and Goals The goal of the post-parse search is to find the complete unification value of a single best parse that fully unifies, where the judgement of 'best' is made according to a well-motivated disambiguator. The input to the post-parse search is a parse forest that is a superset of all possible fully-unifying parse trees; the output is the unification value derived from a single,

complete, best parse tree.

The requirement for full unification is due to the superset nature of the input — the input is guaranteed to contain all of the fully-unifying parses, but it is not guaranteed to contain *only* these fully-unifying parses. (Allowing for this over-coverage is key to achieving acceptable performance in the main parse.)

We also wish to perform this search *without* searching the entire parse forest [81, pp. 9–10]. This is because the parse forest may contain an exponentially large number of parses [151][81, pp. 53–55]. When parsing with a unification grammar, performing this search requires us to perform unifications. Recall from Chapter 3 that this exponential number of parses may produce a similar number of unifications. Of course we wish to avoid calculating all of these unification values; instead we wish to keep the amount of work expended in the search as small as possible so as to do a minimal amount of work.

8.1.1 Previous Approaches

Several previous systems have attacked the problem of disambiguation — finding a single preferred parse tree or value — as part of their operation. Unfortunately, from our point of view, these approaches have been deficient ways:

1. Several systems employ an interleaved parse, and then perform a separate disambiguation step. Examples include KANT and KANTOO [87, 111]. The primary deficiency of this approach is that the all possible unification values must be computed. As we saw in Chapter 3, this is likely to involve an exponential blow-up in the storage of the unification values. In addition, the KANT disambiguator used to expand any disjunction in the final unification values to the top-most level, *guaranteeing* an exponential blow-up under ambiguity.

2. The Core Language Engine [4] uses a separate syntactic and all-parses semantic phase, and disambiguates after the semantic phase is complete. Again, the primary deficiency of this approach is that in order to disambiguate we must first calculate all possible unification values, leading to an exponential blow-up.

3. Maxwell and Kaplan's system [84] performs a context-free parse, and then passes the entire set of unification equations to a constraint solving system. This does result in the efficient computation of a single result, but in case of ambiguity does not choose between the ambiguities in any separately controllable way.

4. Several systems have used weighted context-free grammars [149] to control disambiguation. These include Slocum's METAL machine translation system [140], which (according to Thiel [149, p. 139]) traces its use of weights back to Robinson's DIAGRAM parser [122], as well as Charniak's PARAGRAM parser [24]. Unfortunately, none of these systems use unification-based grammars – they are all weighted context-free grammar parsers, and therefore are insufficient for our needs.

For this system, we wish to have it both ways: we want to efficiently find a single unification result, and we want that result to be from a parse that is selected in a linguistically-motivated way by a disambiguator.

8.1.2 Method of Attack

The general approach we shall adopt is to separate the search for a *best parse* which *fully unifies* from the disambiguation measure that determines which structures are *better* than which. By separating out the search itself from the measure of goodness which drives the disambiguation, we are able to separately prove that the search algorithm is correct, implement, and debug the search component, while still being able to substantially modify the measure of goodness.

We also wish the search to be efficient, in the sense that if there are multiple parse trees which will fully unify, the entire forest of all possible parse trees need not be traversed in the second pass in order to discover a single best fully-unifying parse tree. Since, as we have seen in Section 3.3.1, in the worst case there may be an exponentially-large number of individual parses (relative to the input length), it could take a proportionally long amount of time to search though all possible parses for the best over-all parse. Obviously we wish to avoid doing this if at all possible.[1]

This is the main reason why we chose not to frame this as a dynamic programming problem. The general approach to solving this sort of optimization problem though dynamic programming is to divide the potential solution into k classes, and then traverse the entire tree in a bottom-up manner, keeping the best answers for the k situations separately. Finally, the over-all best solution is chosen from amongst the final k answers [146]. The advantage of this approach is that it is often linear in the total size of the tree (dependent, of course, on the details of the per-node calculations). The disadvantage is that it requires not only the evaluation of the entire tree, but possibly multiple repetitions of similar per-node operations for each of the different k classes. This is much more work than we would prefer to do to solve our problem.

[1] As we have argued in Chapter 2, our pseudo-unification system, and hence our parser, is quite capable of solving $\mathcal{NP}$-complete problems. It is quite possible to give our parser which requires an exponentially-long search.

In order to achieve acceptable performance, we adapted the general best-first search [128, pp. 92–101] technique to our particular needs. In general we wish our disambiguator to be separated from the search, and the search to treat the disambiguator measure of goodness as a 'black box,' without any internal knowledge of how this measure is calculated, but only a few general requirements on what can be examined in the search.

There are three general requirements that the goodness measure must obey in order for the search to be efficient, and a fourth which greatly simplifies the construction of the search. In designing the search we will assume that:

- The goodness value of a sub-tree is an additive combination of the goodness values of the children and a local *node attachment cost* which is based on other information that is local to the root node of the sub-tree. The partial value of a given node is not effected by the partial value, or any other information, derived from any *sister* nodes.
- The goodness measure is calculated in a bottom-up manner. The goodness of leaf nodes is some predefined constant; the over-all goodness of the root of a full tree is simply the goodness value calculated for the root node of the tree.
- The goodness measure must be monotonic increasing in value: the goodness value of a node must be greater than the goodness value of any of that node's children.
- (For the full branch-and-bound search:) The cost function is composed, in part, of the *sum* of the costs of the children.

This combination of requirements on the measure of goodness allows us to adapt the well-known Best-First Branch-and-Bound Search [128, p. 116],[106, sec. II.4.2, pp. 354–367] to our particular needs in order to perform the search in a relatively efficient manner.

8.2 The Search Component

As stated above, we have developed an Iterative-Deepening Greedy Best-First search that is specialized to find the unification value of the best (according to the disambiguator) tree that fully unifies, while attempting to avoid calculating all combinations of unification possibilities.

8.2.1 Best-First Search

Best-first searches are generally described by talking about searching for a best single leaf (terminal node) within a tree covering all possible leaves. This over-loading of terminology will be somewhat confusing for us, as we are actually searching amongst parse trees within a parse forest, but as this is the standard terminology for these searches, we will stick with it for the moment.

According to Russell and Norvig [128, pp. 70–73], we can characterize searches in a nicely abstract way:

Suppose we have some problem in which the search for a complete solution can be characterized as a tree, where each node represents a partial solution, and an arc between nodes indicates that the partial solution represented by the first node can be extended to become the somewhat more complete partial solution represented by the second node.

In addition to this, our problem must meet several other conditions:

- We must be able to identify some single starting state, which we will refer to as the `initial_state(problem)`.
- We can identify when the partial solution represented by some node is a complete solution. We do this by arranging for the predicate function `is_goal` to return *true* when applied to a complete solution, and *false* when applied to an incomplete solution.
- There must exist a cost function `costfn` which maps from partial or complete solutions to numbers, such that if A is a better solution than B, then $\texttt{costfn}(A) < \texttt{costfn}(B)$.

  Note that this implies an important requirement on the estimation of partial solutions: For every goal state A which is reachable from some partial search state $\tilde{A}$, $\texttt{costfn}(\tilde{A}) \leq \texttt{costfn}(A)$. The quality of the estimate will effect the search: the search will be better when the estimate is closer to equality.
- We can create a function `expand_state`, which given a node will return a set of all nodes which are one step further away from the root of the search tree (*i.e.* closer to a goal state).

When all of these conditions are met, we can formulate a search which repeatedly expands the best node so far, until it reaches a goal state. Figure 8.1 shows pseudo-code for an example best-first-search formulated according to these conditions.

It is helpful to also define two functions[128, pp. 92–97]:

```
best-first-search (problem, costfn) {
  PriorityQueue PQ;

  root = initial_state(problem);
  push root, costfn(root) onto PQ;

  loop {
    if empty(PQ) then return FAIL;

    node = pop(PQ);

    if is_goal(node) {
      return node;
    } else {
      newnodes = expand_state(node);
      foreach newnode in newnodes {
        push newnode, costfn(newnode) onto pq;
      }
    }
  }
}
```

Figure 8.1: best-first-search, after Russell & Norvig [128, figs. 3.10, 4.1].

- $g(n)$ is the (actual) cost of the path from the root of the search tree to node n.
- $h(n)$ is the estimated cost of the cheapest path from n to a goal node. We will require that when n is a goal node, $h(n) = 0$.

If we use $g(n)$ as our cost function for `best-first-search`, then we have uniform-cost search. At the other extreme, if we use $h(n)$ as our cost function, we have a greedy search.

A* search If we can combine the two above functions together, using $f(n) = g(n) + h(n)$ as our search cost function, this gives us **A* search**. $f(n)$ is an estimate of the cost for the best path going through n. So by using $f(n)$ as our cost function, we are searching by repeatedly expanding a node along the estimated best path to a goal.

One crucial property of A* search is that it is an *optimally efficient* search: no other search is guaranteed to expand fewer nodes than A* [128, p. 99], [31]. As a result, A* is very widely used to solve searching problems.

The need for nested searches In order to find a true goal state, it is necessary to unify the complete search tree, though it is not necessary to unify the complete tree all at once. If we calculate this unification result piece-wise, we will want to save and possibly re-use the partial unification results. This is because without this re-use of partial solutions, we are sure to perform too many unifications when we don't find the best parse on the very first try.

We can take advantage of these partial solutions by structuring our search to follow the natural structure of the problem. Below we will show that a convenient way to do this for our problem is to perform a recursive top-down search on the parse forest itself.

Unfortunately, the heuristic searches presented above (notably including A* search) have a draw-back: they are not designed to be recursively nested — it is difficult to nest one search inside another search such that any optimality criteria (*i.e.* in A* search) still hold. At first glance, this seems conceptually easy: the outer search could simply consider using an inner nested search to implement the `expand_state()` procedure.

This is an simplistic answer. In order for such a nested search to be as efficient as a single A* search (as measured by the number of nodes expanded), the outer search must provide the inner search with a `no-worse-than` stopping criteria value, so that the inner search will just stop searching if the best partial search node (the one at the front of the priority queue) is worse than this value. The outer search can then use this criteria to prevent the inner search from pursuing a result which is worse than the outer search's second-best search possibility.

Figure 8.2 shows a conceptualized example of such a case. If the expansion from A to B is done by an inferior search, this search must also be given the cost value of `no-worse-than` $= C$ as stopping value, so that the inner search from A to B (the $\overline{AB}$ search) does not bother expending a lot of work coming up with a result which is ultimately more expensive than the *estimated* cost to C.

In addition, the inner search must be continuable from such a stopping point. Still considering Figure 8.2, suppose that the $\overline{AB}$ search was stopped because no solution could be as good as the estimated cost of getting to C. Further suppose that the $\overline{AC}$ search returned an actual cost value for C which was *higher* than the initial estimate. In this case, the $\overline{AB}$ search must be continued using this new, higher, actual cost as the revised stopping criteria, because there could still be an $\overline{AB}$ path resulting in B being cheaper: $g(B) < g(C)$.

We initially questioned whether adding this cost bounding to the search would be worth the effort (both in terms of computational cost and difficulty in implementation), and eventually tried the search both without and with the cost bounding. Empirical results on this choice are quite

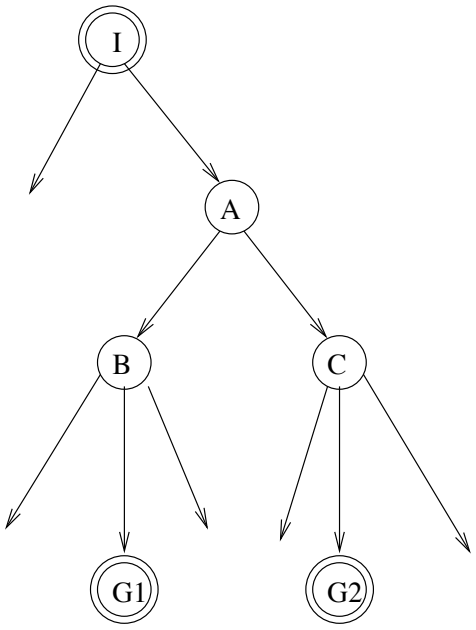


Figure 8.2: An example partial search tree

interesting

The complete best-first branch-and-bound search is somewhat complicated. In order to present our technique clearly, we will first present the problem of searching a packed parse forest abstractly in Section 8.2.2. Next, in Section 8.2.3, we will apply abstract search to the problem of finding all possible unification values. In Section 8.2.4 we present the elements of our best-first search, and describe the pseudo-code for the branching-only version of the search. Finally, we discuss the changes needed to implement full a branch-and-bound search in Section 8.2.5, along with pseudo-code for this revised search.

8.2.2 Searching a parse forest

We can next turn our attention to the details of searching a parse forest. Recall that a parse forest is a collection of nodes and links between nodes. Each node represents an instance of a production in the grammar: an individual node will have one or more ordered sets of pointers to children nodes; each of these ordered sets of pointers to children corresponds to one particular instance of a set of nodes fulfilling the requirements to be the right-hand-sides of some rule, for which the left-hand-side is represented by the parent node. A node is said to be *packed* if it contains more than one of these sets of children [151, 23]. A schematic example of a packed node is shown in Figure 8.3.

A complete, top-down, depth-first search of a parse forest is quite straight-forward. Figure 8.4

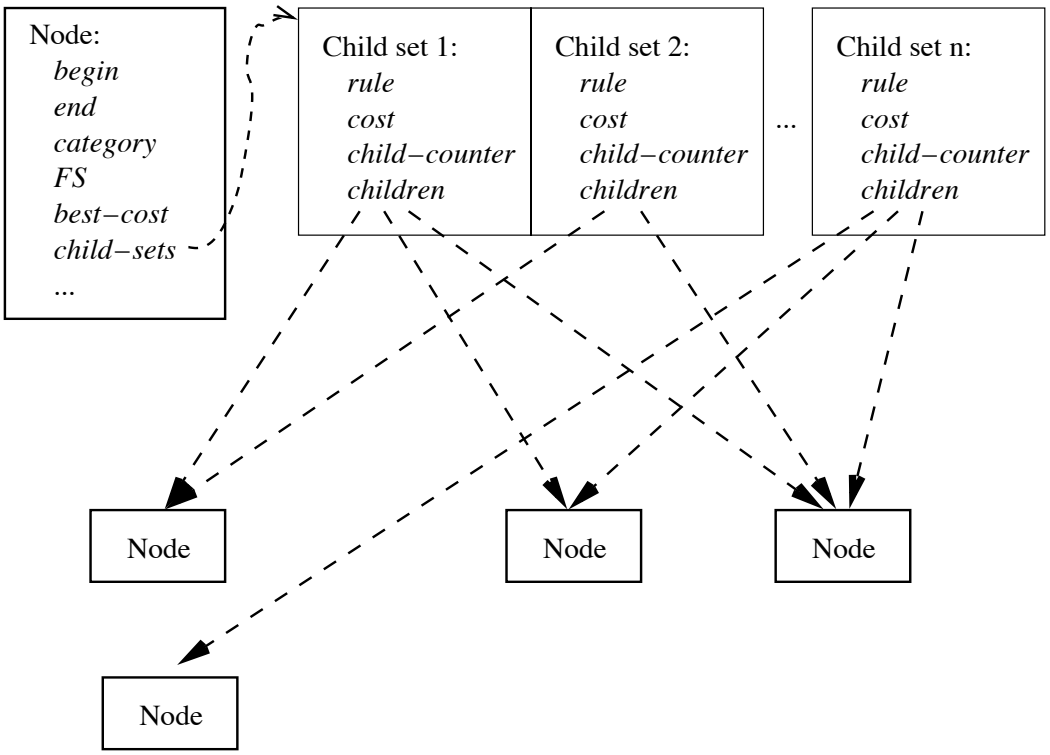


Figure 8.3: An example node structure.

presents an abstract depth-first-search of a parse forest. This pseudo-code assumes that there is a single top-level node to begin searching at; should there be multiple top-level nodes (representing multiple ambiguous parses) one can simply form a super-node, which contains a set of (size-one) sets of pointers to each of the actual top-level nodes as children. This super-node would then become the new top-level node to search.

```
dfs(Node n) {
  do anything needed before descending into children.
  foreach child-set in n.children-sets {
    do anything needed with the set of children before searching them.
    foreach c in child-set {
      if (is_leaf(c))
        do something with the leaf node.
      else
        dfs(c);
    }
    do anything needed with the set of children after searching them.
  }
  do anything needed after descending into children.
}
```

Figure 8.4: Pseudo-code to perform a depth-first search of a parse forest

8.2.3 An All-Paths Search of a Parse Forest

```
all-FSs(Node n) {
  let retval be empty.

  foreach child-set in n.children-sets {
    do anything needed with the set of children
       before searching them.

    new child-fss[length(child-set)];

    foreach i in length(child-set) {
      c = child-set[i];

      if (is_leaf(c))
        child-fss[i] = get-child-FS(c);
      else
        child-fss[i] = all-FSs(c);
    }

    let this-child-fs = unify(node.rule, child-fss);

    if (succeeded(this-child-fs))
      retval = pack-fs(retval, this-child-fs);
  }
  return retval
}
```

Figure 8.5: Pseudo-code to find all FSs of a parse forest via depth-first search

Figure 8.5 adapts the abstract search from Figure 8.4 to the task of finding all of the F-Structures of a packed parse forest in a manner very similar to the technique described by Carroll for unpacking unification values [23, p. 59].

Table 8.1 compares fully-interleaved unification as described in chapters 3 and 7 with Shallow Unification, as described in Sections 4.3.1 and 7.4, in concert with our implementation of the all-paths search shown in Figure 8.5.

From these results, we can see that this combination results in 8% fewer calls to `setvalue`. Because this function implements one of the basic operations of the unification package, we have found that the number of calls to it is a good measure of the total amount of unification work which system has had to perform [114].

We can also see that the amount of time required by the interleaved unification technique to

Search type	total num. nodes	total num. arcs	total calls to `setvalue`
Interleaved Unification	466,945	293,610	8,150,477
Shallow Unification with all-paths search	467,919	293,849	**7,536,736**

Search type	parse time (CPU secs.)	post-search time	Total time
Interleaved Unification	68.35	*n/a*	68.35
Shallow Unification with all-paths search	58.90	8.82	67.72

Table 8.1: Comparison of parse times for Interleaved Unification versus Shallow Unification plus All-Paths Post-Parse Search, on the Catalyst ambiguity corpus (see Section 5.4). The difference in total run-time is not significant.

parse is comparable with the total amount of time required to parse with shallow unification, and then search the full parse forest for the set of all unification values.

All-Paths blows up under extreme ambiguity We are interested in improving the behavior of the parser under the stress of extreme ambiguity. Unfortunately, performing an all-paths post-parse search is no better under extreme ambiguity than fully-interleaved unification is.

We show this by constructing artificial data which exhibits extreme PP-attachment ambiguity by starting with a plausible sentence, and adding zero or more repetitions of a plausible prepositional phrase:

Grease flows through the bearing seal $(\textit{in the engine})^n$.

(The outputs for several initial exemplars were checked to ensure that there was indeed a PP-attachment ambiguity for the unification grammar.)

These test sentences were then used as inputs to several configurations of our system to investigate the run-time behavior of the system.

The graph in Figure 8.6 shows the run-time used both for an all-choices single-pass parse, and for two different two-pass parses; one of which first finds the forest of all context-free parses and then calculates all F-structures for that forest, the other first calculating the back-bone forest using the shallow interleaved unification techniques described in Sections 4.3.1 and 7.4.

For all of these runs, the system is getting somewhat bogged down calculating the unification values for the different possible ways to resolve the PP attachment ambiguity. The two-pass system

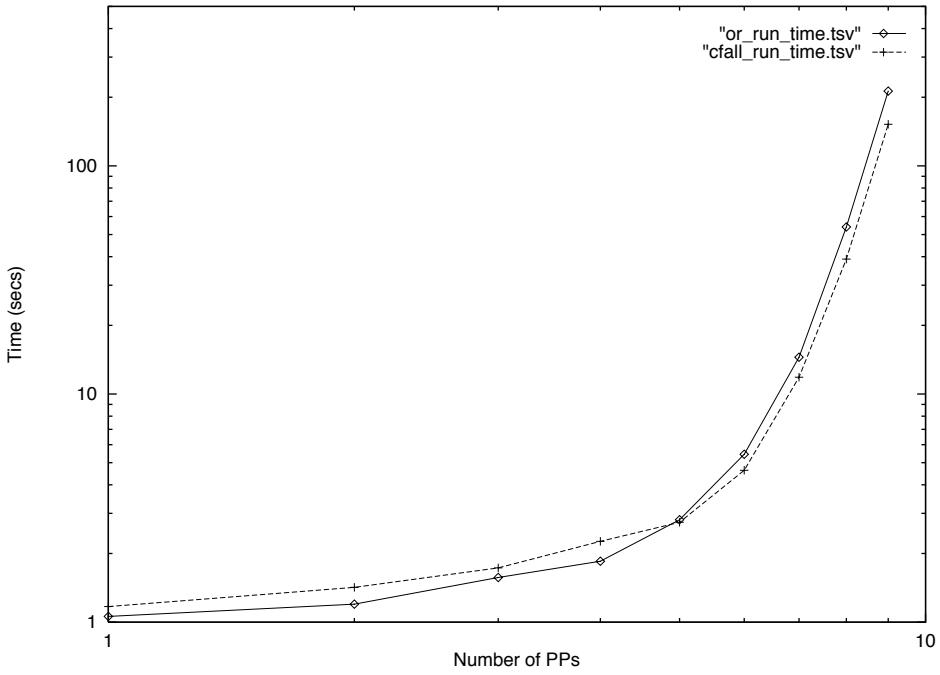


Figure 8.6: Runtime Used for: Single-pass OR-packing (“or”) and 2 pass: C-F 1st pass with all-values 2nd pass (“cf-all”)

is behaving somewhat differently than the single-pass one, using *slightly* less run-time to calculate the eventual results. On the other hand, we can also see that the all-values second-pass parse technique has run-time characteristics similar to the interleaved unification approach, and that both are suffering from a blow-up in required run-time under increasing ambiguity.

8.2.4 A Backtracking Greedy search for a best parse

Our original aim was not to search the parse forest for a packed representation of all valid F-structures, but instead to search for *the best* complete F-structure according to our disambiguation cost calculator. In order to avoid unnecessary computations, we use the disambiguation costs and cost estimates to guide the search. Since not every combination of children will unify together, this search turns out to be non-trivial. We now turn to the task of designing this particular search.

In Section 8.2.3, we saw that it is convenient to search the parse forest by directly searching through the nodes of the forest; it is particularly evident from Figure 8.5 that unification is convenient to do in this sort of traversal. On the other hand, in Section 8.2.1, we have seen that the customary way to frame a best-first search is with single search queue. What we want is a search which allows us to traverse the parse tree in a convenient fashion, while still allowing for a search

that is relatively efficient (both in terms of amount of traversal done and number of search states expanded).

Since we believe that unification is the most expensive part of this process, we will create a search which traverses nodes of the parse forest. We will search for the best F-structure of a node by (in part) recursively searching the node's children for their best F-structure. We will accomplish this by putting enough information into the data structure of a node that we can perform this search.

The basic search It is extremely convenient to frame the search in terms of some basic routines used to control the search. Let us start by defining a pair of routines:

- `get-first-fs` gets the best F-structure and cost of a node. Should there be *no* best FS, then `get-first-fs` will return an indication of failure.
- `get-next-fs` gets the next F-structure and cost: the best-valued FS and cost that is no better than the one we have most recently asked for. Should there be no "next" FS, then `get-next-fs` will return a failure.

Suppose we have called `get-first-fs` and then `get-next-fs`. If we call `get-next-fs` again, we will get the third-best FS for this node. We will further stipulate that we can get *all* of the F-structures by calling `get-first-fs` once, and if it succeeds, then calling `get-next-fs` until this fails.

These will ultimately be convenient, once fully implemented, since we can get the top-n complete F-structures, in disambiguation cost goodness order, by calling `get-first-fs` once on the top-level node in the parse forest, and then calling `get-next-fs` on this node $n-1$ times.

Saving the disambiguation cost value In order to support the search for the best value efficiently, we will store a disambiguation cost value with each node. For packed nodes, those containing a choice over multiple sets of children, this per-node disambiguation cost value will hold the best of the disambiguation costs of the sets of children. In addition, each set of children will hold its own disambiguation cost.

We will also calculate these values in the first-pass parse, so that we can use the values as *estimates* of the final disambiguation costs. We will note here that since not all of the parses that result from the first-pass parse will fully unify, these disambiguation goodness values are *only* estimates. On the other hand, we will see below that they serve as fairly good estimates of the final values.

Memoization We will see below that it is fairly common to ask for the n-th-best F-structure for a given node repeatedly during a search of one of that node's parents. Since the search is for the best parse that fully unifies, unification is at the core of the process. Further, we have argued above that unification is a potentially expensive process. In order to improve the efficiency of our search, it is convenient to *memoize* the (FS, cost) result pairs that we calculate from searching a node, so that we can restore these values when asked without having to recompute them (recursively) from scratch. This will be useful for two reasons: first, the search at a given node may require looking at the first few F-structure choices for a given child, and second, we may end up re-using results for a node in a part of the search which has descended via a different path.

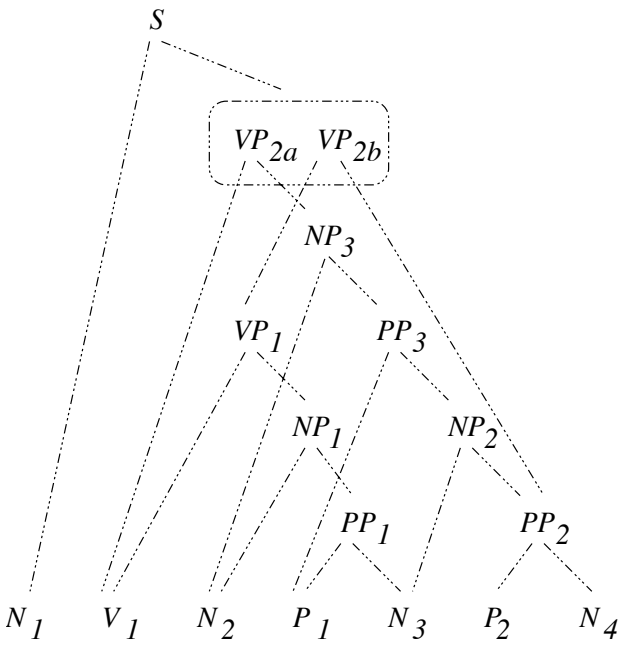


Figure 8.7: An example of multiple attachment

Figure 8.7 shows an example of the latter. In this example, the PP_2 node is a child of both NP_2 and VP_{2b}. If we had first tried the VP_{2b} attachment, but discovered that its rule couldn't unify VP_1 with PP_2, we might end up also trying the NP_2 attachment (by descending down the $VP_{2a} \longrightarrow NP_3 \longrightarrow PP_3 \longrightarrow NP_2$ path). In this case, since we had already calculated one or more F-structures for PP_2, we might be able to simply re-use these values. (This re-use is not universally doable, however — we investigate this question more fully in Section 10.3.2 below.)

The base cases `get-first-fs` and `get-next-fs` are easy to implement for leaf nodes: if

the leaf has only one FS, then `get-first-fs` will return it, and `get-next-fs` will return failure. And if the leaf has more than one FS, then `get-first-fs` will (deterministically) return one of them, the first call to `get-next-fs` will return the second, the second call to `get-next-fs` will return the third, and so forth.

Correctly implementing these functions for a *non-leaf* node is considerably more complicated. We will start with an overly simplistic case: suppose we only had one set of children, containing only one child. In this case, to get the best FS, we can simply call `get-first-fs` on the child node, and then try to unify the result with the current rule. We do not explicitly code for this case, but instead assure that the full search will have this behavior under these conditions.

Multiple children: expanding the search by dimension The search becomes much more difficult when a node has one set of multiple children. In this case, the principal question is:

- If we have more than one child, and the "best" combination of each doesn't unify, where do we search next? For which child should we ask for the next-best result?

Consider Figure 8.8. This figure shows a schematic for searching the space of two children, under the condition that there is no particular combination of these children which would merit a different node-attachment score.

The x axis denotes the relative disambiguation cost measure for the first child, the y axis is similarly so for the second child. Open points indicate combinations of children which do not successfully unify at this node; solid dots indicate combinations which do unify. The total cost of a given combination of children will be the sum of the costs of the children; thus we will take the Manhattan distance as the total cost. Our goal is to find the least-cost point, in this diagram marked as point e, while trying the smallest number of combinations of the children, and while recursively expanding the smallest number of choices of each child (*i.e.* asking for the second-best, third-best, fourth-best, etc. choice for a child when these are not in the end necessary).

For this search, we will satisfy the second goal, that of avoiding expanding the children, by expanding the search space in an as-needed way. We can think of this as a search itself: we will start with the best point (*i.e.* the best of each of the children). From then on, whenever we need to consider more space, we shall do so by expanding in the direction of the smallest estimated incremental cost.

Suppose that we have expanded child x though the i-th choice, and child y though the j-th choice, and we need to expand the search space. If we believe that the $(i+1, 0)$ combination will

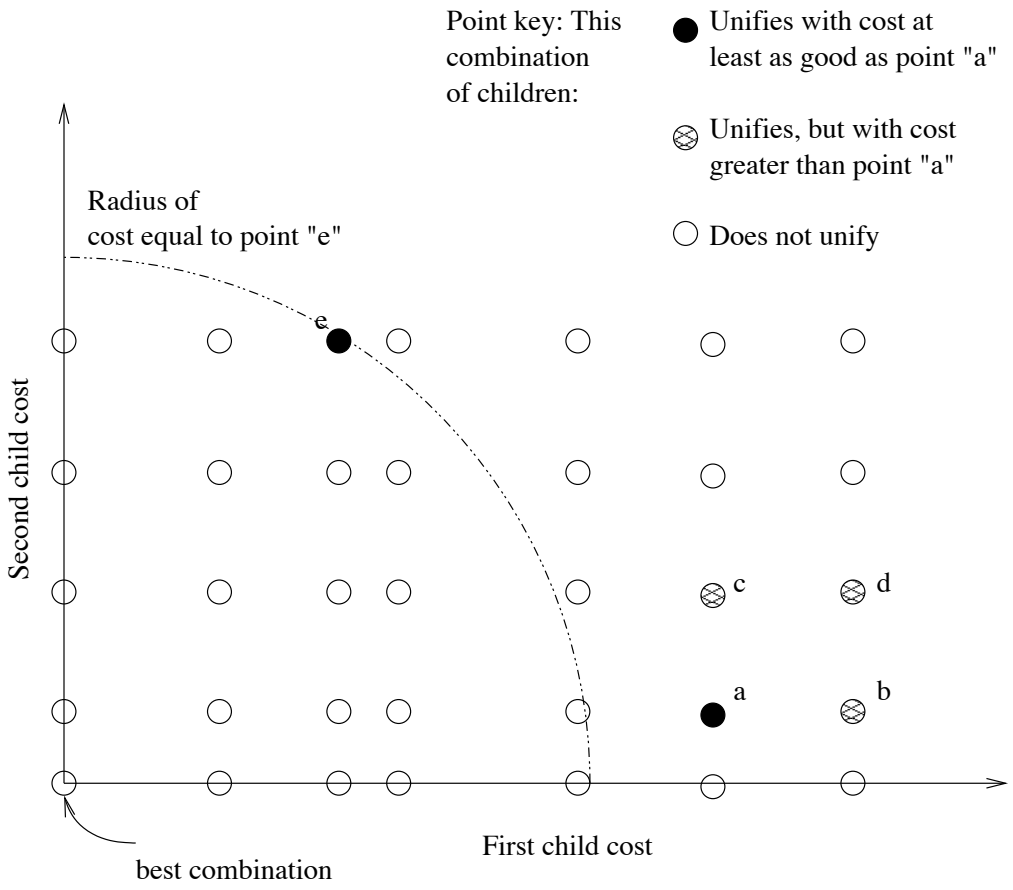


Figure 8.8: Example search space for two children

be lower-cost than the $(0, j+1)$ combination, we will expand the space by generating all the search points $(i+1, 0) \ldots (i+1, j)$.

An example of this technique is shown in Figure 8.9. In this figure, we show the expansion of the search space as a set of expanding rectangles. Of course, in the general case of k children, each dimensional expansion will entail generating all the points within the current bounds for a $(k-1)$-dimensional space. For example, if we had three children rather than two, with current limits h, i, j, and were expanding the search to cover the $i+1^{\text{th}}$ entry of the second child, we would generate all of the search points:

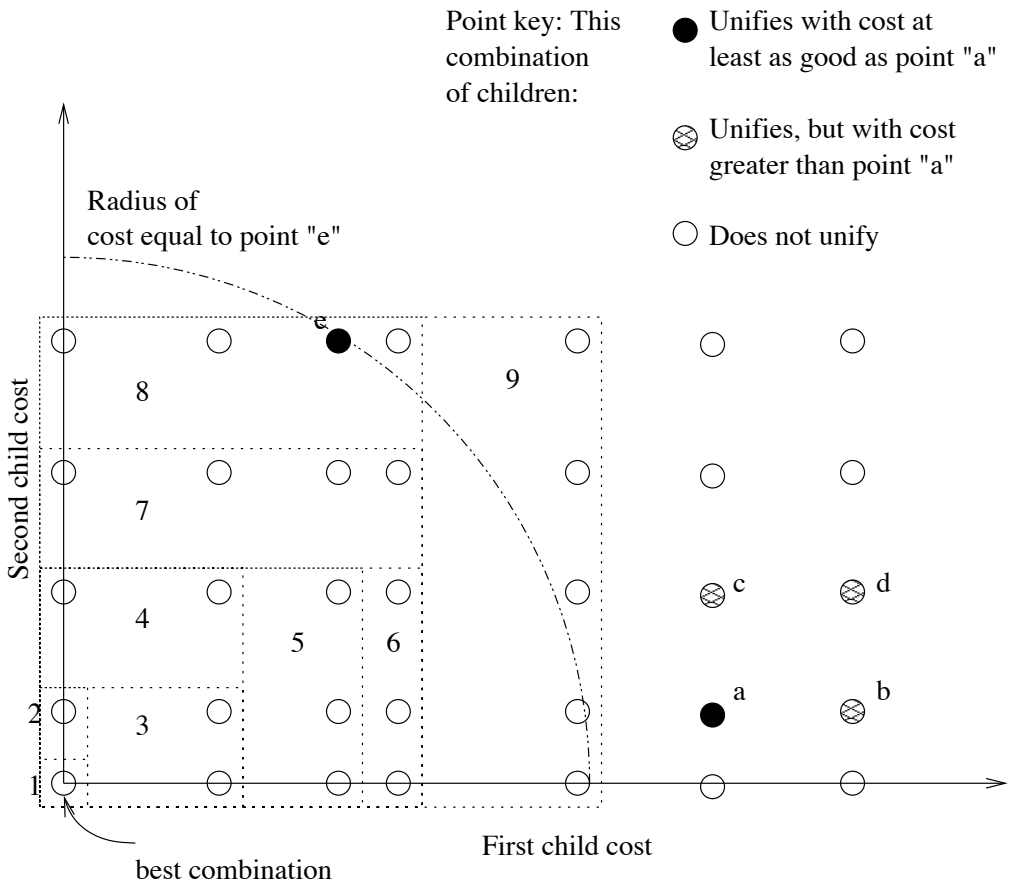


Figure 8.9: Example search space, showing order of dimension search. Note that in the general case of k children, each dimensional expansion will generate a $k-1$-dimensional slice.

$$\begin{array}{ccc} (0,i+1,0) & \ldots & (h,i+1,0) \\ (0,i+1,1) & \ldots & (h,i+1,1) \\ \vdots & \ddots & \vdots \\ (0,i+1,j) & \ldots & (h,i+1,j) \end{array}$$

One way of understanding this expansion is that the space of possible combinations of children forms a polyhedron in a k-dimensional space, and that the dimension expansion is actually selecting

a facet[2] of that polyhedron [106, ch. I.4, pp. 83–113].

In order to support expanding the search in this way, every child-set will have to have a set of the current maximums explored so far. We have already seen in Section 8.2.1 that the general solution to the question 'where do we search next?' is the use of a priority queue. And although it is tempting to assume that the elements on the queue must all be of the same type, there is no such limitation. In order to get the right behavior from our search, we will have to put *two* different types of objects on our search queue:

- Dimensions to expand as described just above
- Points (particular combinations of children) to search.

When a dimension is at the top of the queue, we will expand the search as above, placing the generated points back on the queue. When a point is on the queue, we will try that particular combination of children to see if it works.

The reason why we need only expand the search by dimensions and points is put must succinctly by Nemhauser and Wolsey in Proposition 3.4 of *Integer and Combinatorial Optimization*:

> **Proposition 3.4.** *Every inequality* $a^r x \leq b_r$ *for* $r \in M^{\leq}$ *that represents a face of* P *of dimension less than dim*$(P) - 1$ *is irrelevant to the description of* P.
> [106, p. 90]

The upshot of this is that for a search, we need only generate dimensions to expand (*i.e.* facets) and points of our solution.

Additional considerations There are three other factors which complicate the search somewhat:

- It is not true that the best combination of the best children will be the best solution to this node. Not even if they all unify. Due to the long-distance-relationship nature of disambiguation cost, the final full cost may be worse than predicted for this attachment.
- A combination which unifies may return a worse cost than we had estimated. It is possible that some child just never gets as good a cost as we estimate it should.

[2] A *facet* of a k-dimensional polyhedron is a$(k-1)$-dimensional affinely independent set of extreme points on the exterior of the polyhedron [106, pp. 88–90].

- If we have more than one sets of children, and the "best" combination of the best children doesn't unify, where do we search next?

Fortunately we can use the search queue to address all of these problems as well. In the case of both the first and second of these considerations, when we finally do try a point combination and discover that it unifies, we will also re-calculate the local part of the disambiguation cost. If this cost is greater than we anticipated, we can simply place the final F-structure onto the search queue, along with its actual cost. This will then come to the top of the queue only if it is the lowest-cost solution.

To address the third, we can add a fourth search type to represent the expansion of a new child-set in the search. When this type is encountered, we can try the best combination of the children of this child-set, and also set up the dimension searches for the child-set as well.

Thus, the way we will simultaneously address the considerations we mentioned above is to define a total of *four* different types of objects which can be placed on a search queue. To re-cap, these types are:

1. CHILD-SET: new child set to search
2. DIM: expand one dimension (one child) by one.
3. POINT: evaluate some particular point
4. FS: an already-evaluated FS

Given these four different types of search queue elements, the complete search is simply a matter of initializing the queue with a CHILD-SET item for each of the child-sets, and then processing according to the top of the queue until we find an F-structure. The only additional complication is that because we may be addressing points in different sets of children, every search item (except for the FS) must indicate which child-set it is part of. So, for example, the third-best choice for the first child, and the fourth-best choice for the second child of the second child-set would be the index $(1, 2, 3)$. (Recall that our counts are zero-based, so $(0, 0, 0)$ is the best of each of two children of the first child-set.)

Cost Estimation Finally, there is one remaining issue which we have not fully addressed: the estimation of the disambiguation cost, which drives the choice between the whether we will decide that some F-structure is good enough, or whether we need to keep searching.

As we noted above in Section 8.1.2, and will describe in more detail below, the full disambiguation cost for a given node is an additive combination of two factors: the costs of the children, and a node attachment cost.

For the children, there are three possible cases for estimating a cost:

- When we have never investigated some child, we are examining the best combination (*i.e.* the $(0, 0, \ldots, 0)$ point). In this case, we have an estimated value for this best combination of children and local node attachment from the first-pass, so we just use this value.
- We may have already calculated the n-th-best F-structure for a given child. If so, we already have also calculated the actual cost, so we just use it.
- Finally, when we have not already calculated the n-th-best FS, we know that we *have* calculated the $(n-1)$-st-best FS (due to the way the dimensions are expanded). We can safely use this as an estimate, since the cost of the n-th-best FS cannot be any lower. (It could, however, be the same, so this is the right choice of estimate to use.)

The other factor that goes into the full cost is the local node attachment cost. For that, we must be able to calculate what the most optimistic, or best possible, node attachment cost will be. This, then, is what is used for the estimation.

There are two other types we must estimate costs for: child-sets and dimensions. For an unexplored child-set, we have not explored any of its children, so as above we just use as our estimate the cost value we got for this child set from the first pass.

This leaves the estimate for extending some dimension by one. We will note here that since the cost function is additive, the most optimistic reasonable cost estimate is to combine the most optimistic local node attachment cost, the best cost for each of the other children, along with the estimate for the $(n+1)$-st cost of the child which corresponds to the dimension in question.

The Branching-Only Search Algorithm

We have discussed the search queue, the items which go onto the search queue, and the estimation of costs. We will now put them together into a complete algorithm. For ease and clarity, we will first consider a branching-only (*i.e.* without bound constraints), unoptimized version of the search.

There are two aspects to the correctness of our search algorithm which we will need to keep in mind: the search must be *complete:* if repeatedly called until no further results are available, it must

have tried every fruitful parse tree contained in the parse forest. In addition, we wish to be sure that the *order* of the search is correct: it must return the valid results in order of overall cost.

We will first re-cap the actions of the search functions at leaf nodes. Then, for the case of non-leaf nodes, we will consider the actions of each of the core functions `get-nth-fs`, `get-first-fs` and `get-next-fs`. Finally we will argue that this search is complete and gives results in the correct order.

Leaf nodes For the base case, a leaf node will only contain one single F-structure. The obviously correct action on a leaf is for `get-first-fs` to return this result, and `get-next-fs` to return failure. `get-nth-fs` will then return the result for index 0, and failure for any higher index.

get-nth-fs `(get-nth-fs n node)` has three possible actions:

- If the n^{th} result for `node` has already been computed and memoized, then it is set as the current value of `node`, and we return success.
- If we know that all of the possible results for `node` have been computed, but there are fewer than `n` of them, then we return failure.
- Otherwise:
  - if some earlier-than-n result of `node` has been computed, then we set the node to last computed result.
  - else we call `get-first-fs`; if it fails, then we return failure.

  then we call `get-next-fs` repeatedly until we either have computed the n^{th} result (in which case we return success), or `get-next-fs` fails, in which case we also return failure.

get-first-fs `get-first-fs` is only responsible for initializing or reinitializing the search of a node.

- If the node *has* been previously visited and has a memoized best result, the node is set to it and success is returned.
- Otherwise if the node has not been previously searched, it sorts the packed sets of children into estimated-cost order, and initializes the search priority queue, pushing a **CHILD-SET** search

item onto the search queue for each packed set of children, using the (first-pass) estimated cost of combination of children as the estimated cost.

Finally, `get-first-fs` calls `get-next-fs` to compute the best result, and returns whatever it returns.

get-next-fs Finally consider `get-next-fs`. `get-next-fs` assumes that the search priority queue has already been set up, and its job is to keep removing and evaluating items from off of the search queue until the next-best answer has been found or the queue becomes empty.

If the queue becomes empty before a result is found, then the node is declared to be *fully searched*. If no successful results were ever computed for this node (i.e. none have been memoized), then the node is declared *hopeless*.

When the queue is not yet empty, there is still work to be done for this node. Consider the actions of each of the types of items which could appear on the search queue:

1. **CHILD-SET.** The best values of each of the children are recursively calculated. If all such values can be retrieved, then unification of these values is attempted with the unification function associated with this child-set. If this unification succeeds, the resulting value is pushed onto the stack as an **FS** item.

   If *no* best value can be calculated for any one of the children, then we know that this child-set is fruitless, and no further processing is done.

   If all children have best results (whether or not this child-set actually unifies), then the current maximum counters are initialized to $(0, 0, \ldots, 0)$. Finally, for every child which could have multiple values, a **DIM** item is pushed onto the queue using the sum of the actual child costs and the best possible attachment cost for this child-set as the estimated cost.

2. **DIM.** A dim item contains a (child-set, child index) pair, which selects a particular child (index d) of that child-set. The current maximum count for that child of that child-set (m_d) is then incremented by one. If the corresponding child number d could have a result of m_d (*i.e.* either that child has not been fully searched or it does in fact have that many memoized results), then **POINTS** are generated for this child-set for every combination of children of this child-set from zero up to the current maxima, for all children other than number d:

$$\begin{aligned}\mathcal{P} = \{ \quad & \forall i \in [0 \ldots n-1], i \neq d \\ & \forall j_i \in [0 \ldots m_i] \\ & (j_1, j_2, \ldots, j_{d-1}, m_d, j_{d+1}, \ldots, j_n) \ \}\end{aligned}$$

Where d be the index of child to next increment, n be the number of children in this child-set, i index over children, and m_i be the current maximum for the i^{th} child, and $\mathcal{P}$ is the set of **POINT**s to add to the search queue.

Finally, if child d could possibly be incremented to index $(m_d + 1)$, then this **DIM** is placed back on the queue, with estimate cost

3. **POINT.** A point is simply evaluated: values for the respective i^{th} index (as stored in the **POINT** item) of each of the children are recursively calculated. If all such values can be retrieved, then unification of these values is attempted with the unification function associated with this child-set. If this unification succeeds, the resulting value is pushed onto the stack as an **FS** item.

4. **FS.** When an FS item comes to the top of the search stack, we know that this FS is a best next item, because its actual cost is at least as good as the optimistic estimated cost of any other possibility. So it is memoized and success is returned.

Correctness We must ensure that the search will cover all valid combinations of the children of a node. This is a proof by induction on the height of a node in the parse forest, starting with the leaves and working up. For ease and clarity, we will consider a version of the search *without* any optimizations first.

If one considers the combined action resulting from the evaluation of the **CHILD-SET**, **DIM**, and **POINT** search items, as well as the action of the m_i dimension counts, one can see that slices of points are generated in the manner depicted in Figure 8.9, and that every possible point (*i.e.* combination of choices over children) for all sets of children are considered. This demonstrates that our algorithm is complete: it will search the entire space.

Further, because the search queue is set to pop off items in a least-cost-first order, and because the estimates are optimistic, whenever we encounter an **FS** item on the queue, we know that no possible other choice of children of any other child-set could be better. Therefore `get-next-fs` will produce results that are ordered from best to worst.

Finally, since our search is trivially correct for the base-case of leaf nodes, and correct for any interior node (from immediate parents of the leaves on up), we can see that it is also correct for the top-most node of the entire parse forest.

Our branching-only search implementation

There is one obvious optimization that we found worth making to the algorithm presented just above in Section 8.2.4. While it is not the case that the best combination of children of the lowest-cost child-set will *always* result in the best FS, it is often the case that this is so. In order to avoid doing work that is potentially redundant, we will avoid setting up the priority queue until after we have tried to use this best combination. Figure 8.11 shows the final version of the pseudo-code for the branch-only search, which includes this first-point optimization.

```
get-nth-fs (n node) {
  if we have memoized FS number n
    set current FS and cost from the memoized values for this index
    return that FS
  else if (n == 0)
    return get-first-fs (node);
  else
    let k = the number of currently memoized FSs
    set current FS and cost to the last (number k-1) memoized values
    do (n-k) times {
      get-next-fs (node);
    }
    return current FS
}

get-first-fs (node) {
  if this node is marked hopeless
    return FAILURE.
  else if node is a leaf node
       or we have already computed some number of FSs
    then return the first (best) FS.
  else
    try best (0,0,...,0) combination of children for first set of children.
    If they unify,
      memoize cost and FS, and return them.
    else
      return (get-next-fs(node));
}

init_queue (PQ) {
  Initialize search PQ to have a CHILD-SET entry for each of the
  second thorugh last  child set.
    Use the search-cost from the first-pass as the estimated cost of each child.

  for each child of the first child-set
    If the child has multiple descendants
      push new DIM entry for this (child-set = 0, child) combination.
}
```

Figure 8.10: Pseudo-code for branching-only version of get-nth-fs, get-first-fs, and init-queue

```
get-next-fs (node) {
  if this node is marked hopeless, or fully searched, or is a leaf node
    then return FAILURE.

  if (PQ not initialized) init_queue(PQ);

  loop {
    if the PQ is empty
      if node has some memoized FSs
        mark node as fully searched
      else
        mark node as hopeless
      return FAILURE

    s = pop the top value off the PQ;
    if (s.type == CHILD-SET)
      try best (0,0,...,0) combination of children for this set of children.
      If they unify
        push the result on the PQ.
        for each child with multiple descendants
          push new DIM entry for this (child-set, child) combination.

    else if (s.type == DIM)
      re-estimate cost for this dimension.
      if new cost estimate is more than the next item on PQ
        push s back onto PQ with new cost estimate
      else
        increment cur_count(s.dim) by 1
        forall combinations of points from 0..cur_count(d), d != s.dim
          push a new POINT entry for this combination onto PQ
            with newly estimated cost for this POINT.
        if (child[s.dim] can be incremented past cur_count(s.dim))
          estimate cost for this dimension (including increment)
          push s back onto PQ with new cost estimate.

    else if (s.type == POINT)
      set children to s.dims combinations
      Unify node; if this works, push resulting FS onto PQ

    else (s.type == FS)
      memoize cost and FS, and return them.
  }
}
```

Figure 8.11: Pseudo-code for branching-only version of get-next-fs

8.2.5 Full branch-and-bound search

The search presented in Sections 8.2.4 and 8.2.4 is not as good as it could be, in that it performs more unifications than necessary. To see why, consider what happens when we have a node with two sets of children (A and B), and the estimated cost of child-set A is lower than the estimated child-set of B, but this estimate of A turns out to be overly optimistic (because some of the 'best' children of A don't successfully unify), and the *true* cost of A is higher than the true cost of B. The branching-only search will happily pursue a result from A, even when it turns out that this result may be more expensive than we could get from B.

What we really want to do is to search child-set A, but to (possibly temporarily) abandon this search if it becomes more expensive than the estimated cost for B. And to bound the components of the search not only by the next-best child-set, but more generally by the next-best item of the search queue.

As we hinted at in Section 8.2.1, the addition of cost bounding to our search is a non-trivial change. The principle reason for this is that in the branching search, there were only two possible results of trying out some particular sub-tree: either that particular combination worked, or it didn't. If it worked, then we remember the result and return it. If it didn't, then we go on to a different combination. In either case, we are done with the current combination.

Cost bounding adds a third possibility: this combination is too expensive to work within a given cost budget. This result is problematic, because we cannot just forget about our current combination — we will need to save it away, along with a *revised* cost estimate, because we might have to try it again later.

Branch-and-bound search algorithm

We have to make several adjustments to our algorithm to account for the possibility that the cost is too great, and the proof that the algorithm is correct becomes slightly more difficult. Rather than largely duplicate the algorithm as presented in Section 8.2.4, we simply highlight these differences:

- For both `get-nth-fs` and `get-first-fs`, if the saved value is more expensive than the cost-bound, then these functions return *too-expensive*.
- For `get-next-fs`, whenever the cost of the cheapest item on the search queue is greater than the cost-bound, we must (temporarily) stop the search and return *too-expensive*.
- We will assume that the cost function is composed, in part, of the *sum* of the costs of the

children.

- Finally, in `get-next-fs` the evaluation of **POINT** and **CHILD-SET** entries must also involve cost bounding. We must impose a tight bound on the allowable cost of recursive calls to `get-nth-fs` used in evaluating a set of children. To impose this bound, we first calculate a local-cost-bound as the minimum of either the cost-bound we were called with, or the cost estimate of the second item on the search queue.

  We then initialize a post-cost-est with the cost estimate that was associated with this **POINT** entry in the search queue. Finally, for each child $i \in 1 \ldots n$, we then bound each of the recursive calls to `get-nth-fs` with bound$_i$:

$$\begin{aligned} \text{bound}_i &= \text{lcoal-cost-bound} - \text{attach-cost-est} - \sum_{j \neq i} \text{estcost}_j \\ &= \text{local-cost-bound} - \text{point-cost-est} + \text{estcost}_i \end{aligned}$$

  If any of these recursive calls return too-expensive, the **POINT** or **CHILD-SET** is returned to the search queue. When we do this, we re-calculate the estimated cost for this item. If this **POINT** or **CHILD-SET** is still viable then we place the item back on the search queue with a new cost which is the maximum of this new estimate and local-cost-bound $+ 1$. (The reason for this is that the new estimate may be lower than local-cost-bound $+ 1$, but we know we were unable to evaluate this point for less than this amount.

- Finally, the top-level call to `get-first-fs` is given an extremely high (but finite) cost bound, one higher than the cost of any actual solution.

There are two keys to understanding the cost bounding. The first is the observation that the existence of an item on the search queue implies that we believe that we can find a solution with this cost.

By extension, a *second* entry on a search queue implies that an alternative exists, with this second entry's cost.

Because of the tight bound described above, as far as some particular instance of `get-next-fs` is concerned, a cost-bound from our parent implies that the parent (or possibly some n^{th}-grandparent) believes that it can find an alternate solution, with a fractional cost equivalent

Finally, these solutions may not actually be available at the estimated costs, so a search cannot ever be completely abandoned due to a cost over-run, but merely suspended in favor of a better-

looking prospect.

Branch-and-bound search implementation

Now that we have considered revisions to the search algorithm, we can examine the pseudo-code in detail. Figures 8.12, 8.13, and 8.14 show complete pseudo-code for the revised search.

```
get-nth-fs (n node) {
  if we have memoized FS number n
    set current FS and cost from the memoized values for this index
    return that FS
  else if (n == 0)
    return get-first-fs (node);
  else
    let k = the number of currently memoized FSs
    set current FS and cost to the last (number k-1) memoized values
    do (n-k) times {
      get-next-fs (node);
    }
    return current FS
}
```

Figure 8.12: Pseudo-code for branch-and-bound version of get-nth-fs

```
get-first-fs (node, cost_bound) {
  if node is marked hopeless
    return FAILURE.
  else if node is a leaf node
       or we have already computed some number of FSs
    if the old cost is <= cost_bound
      return the first (best) FS.
    else
      return TOO-EXPENSIVE
  else if PQ has been set up for node (but no results were found)
    return (get-next-fs (node, cost_bound))
  else
    try best (0,0,...,0) combination of children for first set of
    children bounded by cost_bound
    If they unify,
      memoize cost and FS, and return them.
    else if they were TOO-EXPENSIVE
      if node is a packed node (with more than one set of children)
        init_queue (PQ)
        push {CHILD-SET 0, cost_bound} onto PQ
        return (get-next-fs (n, cost_bound));
      else
        return TOO-EXPENSIVE
    else
      return (get-next-fs(n));
}

init_queue (PQ) {
  Initialize search PQ to have a CHILD-SET entry for each of the
  second thorugh last  child set.
    Use the search-cost from the first-pass as the estimated cost of each child.

  for each child of the first child-set
    If the child has multiple descendants
      push new DIM entry for this (child-set = 0, child) combination.
}
```

Figure 8.13: Pseudo-code for branch-and-bound versions of get-first-fs and init_queue

```
get-next-fs (node, cost_bound) {
  if node is marked hopeless, or fully searched, or is a leaf node
    then return FAILURE.
  if (PQ not initialized) init_queue(PQ);

  loop {
    if the PQ is empty
      if node has some memoized FSs
        mark node as fully searched
      else mark node as hopeless
      return FAILURE

    if the cost of the next item on PQ > cost_bound, return TOO-EXPENSIVE
    s = pop the top value off the PQ;
    local_cost_bound = min(cost_bound, cost of NEXT item on PQ);

    if (s.type == CHILD-SET)
      try best (0,0,...,0) combination of children for this set of
      children with bound local_cost_bound.
      If they unify
        push the result on the PQ.
        for each child with multiple descendants
          push new DIM entry for this (child-set, child) combination.
      else if TOO-EXPENSIVE
        re-estimate cost of this CHILD-SET
        push s back onto PQ with cost max(re-estimate, local_cost_bound).

    else if (s.type == DIM)
      re-estimate cost for this dimension.
      if new cost estimate is more than the next item on PQ or cost_bound
        push s back onto PQ with new cost estimate
      else
        increment cur_count(s.dim) by 1
        forall combinations of points from 0..cur_count(d), d != s.dim
          push a new POINT entry for this combination onto PQ
            with newly estimated cost for this POINT.
        if (child[s.dim] can be incremented past cur_count(s.dim))
          estimate cost for this dimension (including increment)
          push s back onto PQ with new cost estimate.

    else if (s.type == POINT)
      set children to s.dims combinations, with cost bound local_cost_bound.
      if TOO-EXPENSIVE
        re-estimate cost of point
        push s back onto PQ with cost max(re-estimate, local_cost_bound).
      Unify node; if this works, push resulting FS onto PQ

    else (s.type == FS)
      memoize cost and FS, and return them.
  }
}
```

Figure 8.14: Pseudo-code for branch-and-bound version of get-next-fs

Proofs of Correctness and Optimality

Now that we have considered the changes to the search to allow cost bounding, and examined the pseudo-code for our actual implementation, we can conclude our examination of the search component with arguments that our search is both correct and optimal.

Correctness As for the simple branching (only) search, the key to the correctness of the branch-and-bound search is that it is complete, in the sense that if we call `get-first-fs` on some given node, and then repeatedly call `get-next-fs` on that node until it fails, we will have considered every combination of children from all of the child-sets of that node. The main difference between the bound-only search and the branch-and-bound search is that the branch-and-bound search may delay looking at some points longer than the branch-only search will, due to the cost bounding, but will eventually evaluate every combination of children.

Since the handling of **DIM**s has not changed — successful processing of a **CHILD-SET** still results in the generation of the same **DIM**s, and these **DIM**s still generate all the same **POINTS**, we know that we will generate all of the **POINTS** we need to in order to search a node.

In the branch-and-bound search, there are only two things that happen to a **POINT** once it has been put onto the search stack:

1. The **POINT** is fully evaluated. This results in one of:
    - An FS which is memoized and returned (cost is as good as next item on stack.)
    - An FS which is put back on the stack (cost is worse than next item on stack.)
    - No FS. The only way for this to happen is for unification of the children to fail.
2. It is not fully evaluated, due to a cost over-run. In this case:
    - the POINT is put back on the stack with a strictly higher cost (as long as the cost is better than 'hopelessly bad'

So long as there is a finite limit on these costs, if a node is properly initialized (by calling `get-first-fs`), and then called for more results by repeated calling of get-next-fs until it fails (because of its search queue becoming empty), every possible **POINT** will have been evaluated for that node.

Finally, because the top-level call to `get-first-fs` is given a threshold which is higher than any actual solution, we will either find a solution or discover that no solution exists. As a result,

either some solution will be found, or the search is completed unsuccessfully, in which case the top-level node has no solutions.

A comparison of branch-and-bound search against A* search The proof of the optimality of our branch-and-bound search depends on a comparison between the action of our search and the action of an A* search with identical handling of the child-sets of a node. Before comparing the action of these two types of searches, we must sketch the operation of such an A* search.

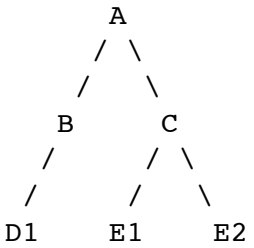


Figure 8.15: An example search tree

First consider Figure 8.15. For the purpose of our example, suppose that the nodes have the following initial cost estimate relationship: $E1 < E2$, $C < B$, $E1 < B$, $E2 > B$, and $E2 > D1$. Also suppose that state $E1$ won't unify.

Now consider an A* search applied to this search tree. This search would have a single priority queue for the search, with points encoding the intermediate stages of a search stored on the priority queue.

Now consider the actions of an A* search as it explores this search tree:

```
A* actions:
  stack starts as: (A)
  pop A:   (A has two child-sets: B & C)
    push B
    push C
          (stack now C, B)
  pop C    (C has two child-sets: E1 & E2)
    push E1
    push E2
          (stack now E1, B, E2)
  pop E1
  Discover E1 goal state, but doesn't work; keep searching
          (stack now B, E2)
  pop B
    push D1
          (stack now D1, E2)
  pop D1
```

```
  Discover D1 goal state; terminate
```

In comparison, consider the actions of our branch-and-bound search on this same search tree:

```
Branch-and-bound search actions:
  search (A, inf)
    push B, C on local stack
    pop C
    search (C, cost(B))
      push E1, E2 on local stack
      pop E1
      search (E1, min(cost(B), cost(E2)) = cost(B))
        return FS for E1
      C fails to unify with E1; drop search item
      E2 more expensive than cost(B); return TOO-EXPENSIVE
    new est cost(C) = max(cost(B)+epsilon, cost(E2)) = cost(E2)
    put (C, new_est = cost(E2)) back on heap
    pop B
    search (B, cost(C) = cost(E2))
      (node only has D1)
      search (D1, min(cost(E2), inf))
        return FS for D1
      B succeeds in unifying with D1
      return FS for B
    A succeeds in unifying with B
    return FS for A
```

Optimality We can see that for our example, an A* would consider the search nodes in the same order as our branch-and-bound search does. This is not a coincidence: if we were to create an A* search, with the same structure to the A* search tree as is implied in our branch-and-bound search, then our branch-and-bound search will search the same nodes in the same order as the A* would.

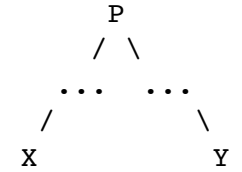


Figure 8.16: Example case for proof

We will show this though contradiction: let P be the top of a search subtree, which is the ancestor of two interior search nodes X and Y. (This situation is shown in Figure 8.16.) Now suppose that at some point in the search, the branch-and-bound search chooses to pursue X but the A* search

would have chosen Y. For the A* search to choose Y, The estimated cost of Y must be strictly less expensive than the estimated cost of X.

But for this to happen, either X's parent must have incorrectly continued a search when it should have stopped due to a cost over-run, or X's parent was called with an overly-high cost-bound. The former obviously should not happen, and a close examination of how **POINT**s are handled reveals that the latter cannot happen either (see Section 8.2.5). Since this situation cannot occur, our supposition is contradicted, and therefore we can conclude that in this case the branch-and-bound search will match an A* search in action.

Since the packing of nodes (with the implied choice over sets of children) is the only way in which structural ambiguity is expressed in the parse forest, it is trivial to see that we can use this case to inductively prove the fully general case, starting at the leaf-nodes of the search and working back toward the root.

So we can conclude that our branch-and-bound search will actually search the a parse forest in the same order as an A* search would. As we pointed out in Section 8.2.1, we know that A* is an optimally efficient search — given the same estimator function, no other search could examine a smaller number of nodes and still be correct in all cases. Because of the equivalence we have demonstrated, we can conclude that our branch-and-bound search is also optimal for this problem.

Actual Performance

Search type	total num. nodes searched	calls to `setvalue` during search	search time (CPU secs.)
All-paths	**906,218**	**5,456,111**	**18.99**
Branch-only	1,132,905	5,999,155	50.67
Branch-and-Bound	1,329,872	5,588,761	56.30

Table 8.2: Comparison of number of nodes searched and amount of unification work done for all-paths, (best-value) branch-only, and branch-and-bound searches, with Context-Free (only) first-pass, on the Catalyst ambiguity corpus (see Section 5.4).

Table 8.2 compares the search effectiveness of the branch-only and branch-and-bound searches when applied to a parse forest resulting from a context-free (only) first pass. We have seen in Table 7.3 of Chapter 7 that performing an interleaved unification as part of the parse cuts down the total number of nodes used (and hence the total amount of structure) by almost a factor of two. Failing to do this filtering substantially adds to the amount of work which a post-parse-search must perform.

This is particularly true of the single-best searches (as opposed to the all-values search). When a

node further up the tree will not unify at all, *every* combination of children will be attempted before this node is declared to be hopeless, which means that the search will spend a lot of time on this useless sub-tree. This effect accounts for the very long search times which we observe in Table 8.2.

Search type	total num. nodes searched	calls to `setvalue` during search	search time (CPU secs.)
All-paths	203,806	2,189,946	8.82
Branch-only	175,304	1,412,759	8.56
Branch-and-Bound	**174,767**	**1,365,715**	8.45

Table 8.3: Comparison of number of nodes searched and amount of unification work done for all-paths, (best-value) branch-only, and branch-and-bound searches, with shallow first-pass unification, on the Catalyst ambiguity corpus (see Section 5.4).

Table 8.3 compares these searches on the parse forests generated by a first pass parse which uses the shallow interleaved unification described in Sections 4.3.1 and 7.4. These results are much better — we can see that the shallow unification filtering is doing a very good job of removing the parts of the parse forest which the search would have wasted a lot of time fruitlessly searching.

We can clearly see that both the branch-only and branch-and-bound searches are avoiding examining all of the nodes of the parse forest, and both variants are performing substantially fewer unification operations (*i.e.* calls to `setvalue`) than the all-values search needs.

The bounding of the branch-and-bound only slightly reduces the number of unification operations performed over the branch-only search by some 3%. On the other hand, the run-time of the branch-and-bound search is not significantly better than the branch-only search. The extra calculation of the search bounds takes small additional amounts of time to perform, and the action of the bounding and the possible under-estimate of the cost of some sub-trees may some extra effort as well (see Section 8.2.5). All this extra effort is done in the time that is saved by performing 3% fewer unifications, so on these data there is no net improvement.

8.2.6 N-Best search

Because we have designed our Branch-and-bound search centered around the `get-first-fs` and `get-next-fs` functions, it is natural for us to ask how much additional effort is required to find the next-best several unification values, in comparison to the effort to find the best value.

Table 8.4 summarizes the additional amount of work required to extract the second through (up to) thirtieth-best unification value, using the full branch-and-bound search. From this table, we can see that the cost of extracting the *n*-best answers is relatively inexpensive. Finding the top three unification values is only twice as expensive as finding the best value, and finding up to thirty best

Maximum n-best Unification values	total num. nodes searched	calls to `setvalue` during search	search time (CPU secs.)	*time relative to 1-best*
1	174,767	1,382,824	8.33	
2	264,223	1,782,497	14.08	*1.69*
3	315,064	2,010,934	16.62	*2.00*
6	393,087	2,338,569	20.63	*2.48*
10	449,358	2,597,434	23.83	*2.86*
20	516,518	2,930,944	27.92	*3.35*
30	552,507	3,120,056	30.13	*3.62*

Table 8.4: Comparison of effort to calculate *n*-best results for a variety of values of *n* on the Catalyst ambiguity corpus. Work represented by number of nodes searched, amount of unification work done, and total search time.

requires less than four times the time to find the best answer.

The graph in Figure 8.17 presents the run-time results of Table 8.4 in graphical form. From this graph we can see that the rate of increase in required time to find the n-best values is substantially sub-linear in n.

This effect is due to three main factors. The primary factor that is that not every sentence in this test corpus *has* as many distinct readings as we are searching for in the limit. However, in this corpus every sentence does have at least two readings. The comparison of required effort to find the second-best ($n = 2$) and third-best ($n = 3$) unification values shows that the memoization technique implemented in the `get-nth-fs` function (Figure 8.12) is performing well.

Finally, the cost increase for $n = 2$ over $n = 1$, compared to $n = 3$ over $n = 2$, is due to the optimization in the implementations for `get-first-fs`: if the best combination of children unifies, then this answer is returned without setting up any of the search mechanism. The full search stack is only set up when the best combination of children does not unify, or when a second-best answer is required.

Dynamic Dimensional Analysis of Search

Recall from Section 8.2.4 above, where we introduced backtracking search, that in our search expansion we expand by *dimensions* and *points*. Also recall that expanding by a dimension entails increasing the current search limit for the dimension in question by one, and then generating all of the combinations of points to fill out this expansion.

Finally, recall that we said that expanding by a dimension for a child-set with k children involves generating all combinations of search points in a $k-1$-dimensional space. And that this is therefore

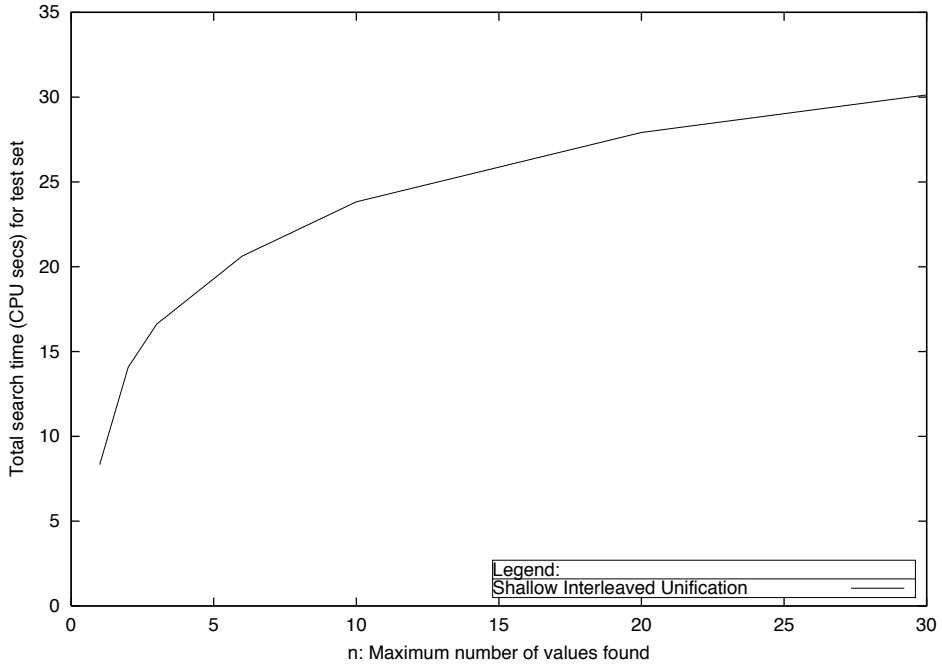


Figure 8.17: Maximum number of unification values to find versus search time (in CPU-seconds) for finding those n-best values.

exponential in the number of children.

A reasonable question to ask is: how often does this occur in practice. In particular, how often do we expand 3-or-more-dimensional children (*i.e.* have to generate a 2-or-more-dimensional slice of points) in practice?

Table 8.5 shows the static distribution of the number of non-terminal symbols in the right-hand-sides of rule in the KANT grammar. From this table, we can see that the vast majority of rules contain at most two non-terminals. Simply reasoning from this distribution can be misleading, however, because some of these rules may be used much more often than others, and we are mainly interested in the dynamic behavior of the system.

In order to see how the system behaves in practice, we instrumented the function which expands by a **DIM** and generates **POINT**s to count the number of children it is expanding over every time it is called. We then re-ran the n-best test as above on the Catalyst ambiguity corpus.

Table 8.6 presents a summary of these counts. **DIM**s with sets of children of length $1 \ldots 4$ were expanded. (Although there is a one rule with five non-terminal children in the KANT grammar, it was not ever used in the processing of this test corpus.)

From Table 8.6 we can see that the dynamic behavior of the system is very favorable with respect

number of non-terminals in right-hand-side	*total number of rules in grammar*
0	133
1	384
2	381
3	80
4	3
5	1

Table 8.5: Static distribution of the number of non-terminals in the right-hand-sides of the KANT grammar rules.

Maximum n-best	*number of children expanded by* 1	2	3	4
1	6645	2743	205	20
	69%	*28.5%*	*2.1%*	*0.2%*
2	58326	16097	1216	88
	77%	*21.2%*	*1.6%*	*0.1%*
3	125333	33446	2515	168
6	220226	58643	4271	256
10	337685	90425	6372	353
20	484300	130213	8904	460
30	647467	174575	11648	575

Table 8.6: A comparison of the number of dimensions in DIMENSION expansions for the branch-and-bound search, for a variety of n-best maxima. Percentages of the total number of counts are shown *in italics* for the single-best and 2-best cases. Tests were run on the Catalyst ambiguity corpus (see Section 5.4), using Shallow Interleaved Unification in the first pass parse.

to the number of children participating in dimension expansions. For the single-best case, we are expanding a node with a single child or two children most of the time — expansions of a space covering 3 or more children only accounts for 2.3% of the total.

When the search is expanded to look for the two best over-all unification results, the distribution becomes slightly more favorable; **DIM** expansions of 3 or more children only account of 1.7% of the total number. For $n \geq 3$, the ratios of expansion stay approximately proportional to the $n = 2$ case.

From these results, we can conclude that even though our search algorithm is exponential in the lengths of the rules, this is not a problem in practice.

8.3 Disambiguation Cost Calculator

As we have stated above, we opt to interleave the process of disambiguation into the process of calculating the final unification values, which is done after we have found a superset forest of possible parse trees. Since the search, as described above in Sections 8.2.4 and 8.2.5, handles the task of searching for the best parse, the problem of disambiguation is reduced to making sure that preferred parses are scored 'better' than less-preferred ones.

This isolation of function reduces the difficulty of creating the disambiguator, and allows the disambiguation technique to be modified without worrying about the details of the search. In this section, we examine the cost calculation function that drives the disambiguation process.

8.3.1 Algorithmic Requirements

In order to work with our search, there are two conditions that our cost calculator must meet:

1. The cost function must be estimable, and the estimate must never be greater than the actual cost value.
2. The cost function for a sub-tree must be dependent on (or conditioned by) only the contents of that sub-tree — it must not depend on the contents of any other sub-tree.

One important detail to keep in mind when creating such a disambiguation scoring system is that one does not either *select* or *rule out* parses; rather, one *penalizes* less-preferred parses more than more-preferred ones. Also, in order to maintain monotonicity one must never apply a bonus to some particular parse, but rather one makes sure to penalize all the other choices. A clear understanding of this is key to the successful implementation of a disambiguator in this system.

8.3.2 Useful disambiguators can be implemented within these criteria

In order to demonstrate that this is a reasonable framework for the implementation of disambiguators, we implemented a non-trivial knowledge-based disambiguator modeled on the disambiguator used in an earlier version of the KANT [87, 111, 90] system. Though the disambiguator described in [90] is similar in final result, the techniques used to reach this result are completely different.

For those more statistically inclined, we also sketch a method of incorporating a statistical disambiguator into this system.

An example knowledge-based disambiguator

One important class of disambiguators are those that are based on knowledge of the syntax of some particular language. An example knowledge-based disambiguator has been implemented in this system; the experimental results presented below include the use of this disambiguator.

The disambiguator is based on three heuristics proposed by Kimball [60] and Frazier [38]. Application of these heuristics to disambiguation in parsing was investigated by Whittemore and Ferrara [159], Hindle and Rooth [47], and Mitamura, *et al.* [90]. Here we list them in order of importance:

1. Certain prepositions, when applied to certain nouns, will prefer to attach to certain verbs over an attachment to a close-by noun [159, 47, 90]. In Figure 8.7, VP_{2b} could be selected over VP_{2a} if the triple (V_1, P_2, N_4) was a preferred attachment set.

2. Near attachments are preferred over far ones [60, 47, 90]. Again referring to Figure 8.7, VP_{2a} would be preferred over VP_{2b} by this criterion because the head-word of VP_{2a}, namely V_1, is 1 word away from the head-word (N_2) of the non-head child (NP_3). In contrast, the head-word of VP_{2a} (also V_1) is 4 words away from the head-word (P_2) of non-head child PP_2.

3. Shallow productions are preferred over deep ones [38, 47, 90]. In Figure 8.7, VP_{2b}, with a maximum depth of 4 ($VP_{2b} \rightarrow VP_1 \rightarrow NP_1 \rightarrow PP_1 \rightarrow N_3$), would be preferred by this criterion over VP_{2a} with a maximum depth of 5 ($VP_{2a} \rightarrow NP_3 \rightarrow PP_3 \rightarrow NP_2 \rightarrow PP_2 \rightarrow N_4$).

The system maintains a disambiguation-cost structure for every successful production within the parse forest. This record contains both the partial cost for a sub-tree, along with other information that is of use to the disambiguation scoring system.

The cost of a given production is equal to the sum of the costs of its children, plus a penalty for making an attachment. When any of the children nodes have a packed choice over sets of grandchildren, the *best* choice of children is used for the score or the parent. Similarly, if the parent contains a packed choice over sets of children, the score for the parent will be the best score for any of its children.

The penalty for the attachment is assessed according to the guidelines above:

- Of topmost importance, we have triples of Verb-Prep-Noun that prefer to associate together. The way we persuade this to happen is to penalize any attachment of a PP that matches one of these triples to anything other than a preferred verb.

We do this in the actual implementation by maintaining a set of tables. In the lexical look-up pass, we save away the `(SEM)` slots of the unification values of all words. Whenever we construct a PP, we look up these `SEM` values for both the Prep and the child object. (The unification optimizer, described in Section 7.4.2 above, is told that the (SEM) slot is used in order to prevent unifications of this slot from being optimized away.) We then look up this pair in a (hash) table of "interesting" verbs, and add the set (list) of verbs to part of the disambiguation-cost information.

Finally, when attaching anything, we look to see if there is something in the appropriate portion of the disambiguation-cost record. If so, we look to see if the (SEM) slot of the lexical head of this production appears in the list in the disambiguation-cost record. If so, it gets a small value (*e.g.* 0) added to the cost; if not, it gets a large value (*e.g.* 200). This value is at least an order of magnitude greater than the multiplier on crossing-cost below.

- At the middle level of importance, we prefer near attachments to far ones. We do this by keeping track of the word-position of the lexical heads of all rules. Then, when attaching elements of a production together, we add up the absolute value number of words between the lexical head and every non-head element. This sum is multiplied by a middle-level multiplier (*e.g.* 10) and added to the cost of the production.

- At the lowest level, we prefer shallow productions to deep ones. We do this by adding a small penalty (*e.g.* 1) for every production which is created.

Though this scoring system may seem complicated, it is actually quite straight-forward to implement and inexpensive. The core of the scoring component is under 300 lines of code, and calculating the score costs less than 2% of the total run-time of the system.

Fitting a statistical disambiguator into this framework

The frame-work for this disambiguator is amenable to statistical as well as knowledge-based techniques. As we have said in Section 8.1.2 above, it is quite useful to separate the calculation of the disambiguation cost from the search itself, and that we can use any cost function which obeys certain locality measurements outlined in Section 8.3. A final requirement is that the cost must be additive.

Although we implemented the reference disambiguation cost calculator in a knowledge-based way, there is no *a priori* reason why we must do so. We could just as easily have implemented the cost calculator in terms of likelihood.

In order to obey our locality constraints, we must frame the likelihood calculation in terms of the (informal) equation:

$$P(\text{node}) = P(\text{attachment}|\text{node}) \cdot \prod_{\text{child} \in \text{children(node)}} P(\text{child}) \tag{8.1}$$

We can modify equation 8.1 by taking negative log probabilities. If we define the function $L(x)$ as:

$$L(x) = -\log P(x) \tag{8.2}$$

Then substituting this into our probability estimate gives us the familiar form of our disambiguation cost:

$$L(\text{node}) = L(\text{attachment}|\text{node}) + \sum_{\text{child} \in \text{children(node)}} L(\text{child}) \tag{8.3}$$

It is possible that other probabilistic disambiguation methods could be used within this framework, so long as the ultimate cost function can be framed so as to meet the requirements given at the beginning of this section.

8.3.3 Sensitivity of Search to Choice of Cost Function

The results shown above in Tables 8.3 and 8.4) were run using the cost calculator described in Section 8.3.2. While these results are quite encouraging, we should question how dependent they are to the choice of cost calculator.

In order to understand how sensitive the search is to the choice of cost calculator, we evaluated the system using a variety of cost functions:

- *Full Knowledge-Based* — This is the cost function described above in Section 8.3.2.
- *Crossings* — For this cost function, the cost of any tree is equal to ten times the sum over the non-lexical-head children of the number of words between the lexical head each child and the lexical head of the tree, plus a penalty of 1 for each production in the tree. (This is exactly second and third terms of the Full Knowledge-Based cost function without the Verb-Prep-Noun triple PP attachment cost term.)

- *Size of Tree* — For this cost function, the cost of any tree is equal to the number of nodes in that tree.

- *Reverse Crossings* — This cost function is like the *Crossings* cost function, except that we select for the *greatest* number of crossings by subtracting the count of crossings from the number of children times the total span of words for the tree. As a result, this function will prefer trees with high numbers of crossings; completely the opposite action of the *Crossings* cost function.

Cost function	total num. nodes searched	calls to `setvalue` during search	search time (CPU secs.)
Full Knowledge-Based	174,767	1,365,715	8.45
Crossings	164,296	1,321,747	7.65
Size of Tree	164,256	1,344,632	6.85
Reverse Crossings	163,631	1,317,104	7.46

Table 8.7: Comparison of the search effort required for the branch-and-bound search using several cost functions, test running on the Catalyst ambiguity corpus.

Table 8.7 shows the results of these comparisons. Although the run-times, number of nodes visited, and number of calls to `setvalue` varies for these different cost functions, the variations are not that great.

The cost function effects the run-time in two different ways. Directly, some cost functions (such as the Full Knowledge-Based one) are more expensive to compute than others (such as the Size of Tree function, which is of essentially trivial cost). Indirectly, the cost function can drive the search to consider a greater number of choices before discovering the best one, which will also increase computation.

In fact, this comparison shows that the Full Knowledge-Based cost function is somewhat worse than the other functions both in terms of driving the search to discover an answer (a fully-unifying tree) quickly, and in the raw cost to compute the function itself.

Due to the non-monotonic nature of unification — that some combinations of child nodes will *not* successfully unify — it is important for the cost function to work in cooperation with the grammar, rather than at odds against it.

In terms of raw speed, the best case situation for the post-parse search is a cost function which guides the search to find a completely unifying tree on the first traverse of the parse forest down to the individual leaves and back up.

In contrast, the search will run slowly if the cost function which carefully guides the search

to attempt to combine children, each containing many alternate deep trees, in ways which do not ultimately unify. Such at-odds behavior would cause the search to waste substantial amounts of time generating all combinations of the children's sub-trees. In the worst case, such a cost function would require the search to examine all possible parse trees before finding one that fully unifies. Of course, since there are an exponentially large (in the input length) number of possible trees for some inputs, such a worst-case cost function would cause the search to spend exponentially large amounts of time looking for an answer.

Ultimately, however, we cannot judge a cost function to be "good" simply in terms of reducing the search time – a good cost function will select parses we prefer (based on some set of criteria possibly involving run-time or possibly not) over parses we do not want.

Chapter 9

Evaluation of the System

> "If you didn't test it, it doesn't work."
>
> — *Bob Colwell, Intel*

In the previous four chapters, we have examined this system in detail. We have discussed the high-level design of the multi-pass architecture in Chapter 5, the first-pass interleaved unification backbone parser in Chapter 6, the optimizing unification compiler and shallow grammar generator in Chapter 7, and the 1-best post-parse search with disambiguator in Chapter 8.

As part of this examination, we have investigated many of the aspects of this system in detail. We have discussed the ambiguity problem, and seen how the design of the system attempts to address the problems related to this phenomenon. We have seen the system successfully avoid these problems for both our natural and artificial development test corpora.

In this chapter, we will demonstrate some level of generality of our solution by testing the system with unseen data – test data which we have not previously run though the system, and for which we will not attempt to 'tune' the system. These tests will demonstrate that our system has not become accidentally "trained" to the development test conditions, but instead implements more general solution to the problems we have pointed out.

For this demonstration, we present two sets of experimental results. For the first evaluation, we examine the behavior of the system, running with the same grammar we used for development, on new data (*i.e.* data which we have not previously used). This evaluation will demonstrate that our techniques are not inadvertently 'trained' on the development data, but work in the target domain generally.

For the second evaluation, we use new test data on a new test grammar. This test examines the

behavior of our techniques much more generally, extending beyond the domain we have developed our techniques for into a different, and only partially related domain.

9.1 Experimental Methodology and Conditions

All of the tests described in this document were conducted on an Apple Macintosh (R) PowerBook (R) G3-series ("WallStreet"), updated with a PowerLogix BlueChip G3 CPU card. The test machine has a CPU clock rate 466 MHz, 1 MB of level-2 cache running at half-speed (i.e. 233 MHz), and 288 MB of main memory on a 64-bit-wide bus running at 66 MHz [79]. The test machine ran MacOS version 9.1.

The parser was compiled and run using Macintosh Common Lisp version 4.2, with options `(speed 3) (safety 3) (space 0) (compilation-speed 0)`.

9.2 Test on the Catalyst System

We wish to assure that the optimizations we have made are not specific to our development test set, the Catalyst ambiguity corpus. In order to determine whether these optimizations are effective for the KANT system in general, we tested our parser running with the KANT grammar on a previously-unseen corpus of sentences.

9.2.1 About the KANT grammars

For this test we used the same grammar as for the system development. The KANT grammar [20, 89, 110, 111] has a total of 982 rules, of which 19 are duplicates of the context-free parts with different unification rules. There are 544 distinct context-free symbols in the back-bone of the grammar. The grammar has an average of 4.36 unification equations per rule.

This grammar is designed to handle a substantial, though restricted, subset of American English, as well as SGML-based mark-up indicators.

9.2.2 About the Catalyst 10k corpus

As we mentioned in Chapter 1, the KANT system parser is used to check conformance to a controlled language [87, 111]. This system has been deployed at a manufacturer of heavy equipment for

over a decade now [55]. In order to help improve the performance of the system, the manufacturer collected a number of heavy equipment manuals in 1997.

For final testing, we selected 10,000 sentences at random from among these data. The chosen sentences range in length from 1 to 100 words, with an average length of 15.97 words per sentence, and a median length of 14 words. The test set data is a total of 977965 characters long, for an average of 97.8 characters per sentence.

Sample Test Sentences. The following ten sentences were chosen at random from out of this 10,000 sentence test set.

```
For 3-phase applications, the terminal blocks are labeled
<label>L1</label>, <label>L2</label>, and <label>L3</label>.

Maximum Inlet Manifold Air Temperature

Pressure tap for<?Pub Tag AddTerm> venturi throat pressure
<?Pub /AddTerm>(front of the valve body)

<pubref><pubtype>Special Publication</pubtype><media><formno>
JEHP1026</formno></media><pubtitle>Information and Requirements
Sheet</pubtitle></pubref>

Refer to <pubref><pubtype>Disassembly and Assembly</pubtype>
<media><formno>RENR1466</formno></media><ie-topic><pubtitle>
Vane Pump (Hydraulic and Steering) - Install</pubtitle>
<ieref controlno="i00995088"></ie-topic></pubref>.

Refer to <pubref><pubtype>Operation and Maintenance
Manual</pubtype><ie-topic><pubtitle>S&middot;O&middot;S Oil
Analysis</pubtitle></ie-topic></pubref> for information that
pertains to obtaining a sample of the engine oil.

Replace any O-ring seal that is worn or damaged.

<salesmdl>D9R </salesmdl>(all models)

The backup alarm remains disconnected.

The oscillation and delay of the weld bead allows the same
size of electrodes to apply wider, more uniform weld beads.
```

9.2.3 Results on the Catalyst corpus

Parse type	*Search type*	parse time	post-parse search time	Total time
Fully-Interleaved	*n/a*	348.72	*n/a*	348.72
Restricted	all-values	309.78	43.73	353.51
Shallow	single-best	309.41	44.21	353.62
Wild-carded	all-values	303.81	33.76	337.57
Shallow	single-best	303.03	35.05	338.08

Table 9.1: Test on the Catalyst 10,000-sentence evaluation corpus: break-down of total run times for Interleaved Unification, and both Restricted and Wild-carded shallow unification; each with all-values and single-best-value Post-Parse Search. Variations in parse times for identical parse configurations are due to sampling noise.

Parse type	*Search type*	*Number of nodes:*	
		Created	Searched
Fully-Interleaved	*n/a*	2,374,027	*n/a*
Restricted	all-values	2,660,450	941,775
Shallow	single-best	2,660,450	838,353
Wild-carded	all-values	2,388,946	796,385
Shallow	single-best	2,388,946	707,118

Table 9.2: Test on the Catalyst 10,000-sentence evaluation corpus: total number of nodes (*i.e.* inactive arcs) created in the parse, and searched.

Table 9.1 shows the run-time results for the Catalyst 10,000-sentence test corpus. These results compare performing a fully-interleaved unification parse with combining either a Restricted or Wild-Carded shallow unification parse with either an all-values or single-best-value post-parse search.

These results are consistent with those of Table 5.1 of Chapter 5: on average it is 3.1% faster to parse using the wild-carded shallow grammar and then search for either one or all complete unification values than to perform a single heavy-weight interleaved unification parse. This advantage is statistically significant, with P = 1.39794E-67 for a single-tailed, matched-pair T-test.

These results also show that there is a slight but statistically significant run-time advantage to using the wild-carded version of the shallow unification grammar over using straight negative restriction on this parser for this corpus. (For an explanation of these conditions, see Sections 5.3.1 and 7.2 above.)

Table 9.2 shows the number of nodes created in the parse pass, and searched in the post-parse search for these same five parser configurations, and Table 9.3 shows the number of calls to the

Parse type	*Search type*	*Number Calls to* `setvalue`:		
		parse	search	total
Fully-Interleaved	*n/a*	39,459,589	*n/a*	39,459,589
Restricted	all-values	16,046,039	11,366,785	27,412,824
Shallow	single-best	16,046,039	6,459,305	22,505,344
Wild-carded	all-values	26,714,294	7,537,036	34,251,330
Shallow	single-best	26,714,294	5,304,235	32,018,529

Table 9.3: Test on the Catalyst 10,000-sentence evaluation corpus: total number of calls to `setvalue` in parse and post-parse search.

`setvalue` function. These results are also consistent with our development test findings: the Wild-Carded shallow grammar requires more unification effort (here measured as calls to `setvalue`) than the Restricted shallow grammar, but the parse forests produced are smaller for the Wild-Carded grammar, resulting in significantly fewer parse forest nodes being searched to compute the final unification value or values.

Figure 9.1 shows a sentence-by-sentence comparison of the net CPU time requirements of two parser configurations. For this graph, the simple single-pass Full Interleaved Unification parser configuration is compared against the two-pass technique using Wild-carded Shallow Unification and a single-best post-parse search. Each point represents a single sentence; the CPU time required to parse using the single-pass Full Interleaved Unification configuration determines the horizontal offset, and the time required for the two-pass configuration determines the vertical offset.

This figure also includes a linear trend line, $y = 0.8797x + 0.0031$, with goodness-of-fit measure $R^2 = 0.9511$, indicating that this linear equation fits the data quite well. Points to the right and below the $x = y$ diagonal are sentences for which the one-pass parser is slower than the multi-pass single-result parser, whereas points to the left and above are sentences for which the one-pass parser was faster. As we have seen above in Table 9.1, this two-pass configuration is 3.1% faster than the one-pass configuration on average. This is a significant difference (P = 1.39794E-67 for a single-tailed, matched-pair T-test).

9.2.4 Conclusions from this test

The main conclusion we can draw from the results of this test is that shallow unification with post-parse search is competitive with straight-forward interleaved unification parsing. In fact, we have seen that it is very slightly faster on average.

We have also seen that for some outliers, the shallow parse plus single-value search strategy avoids the ambiguity problems, and even in a extreme outlier case where this strategy suffers from

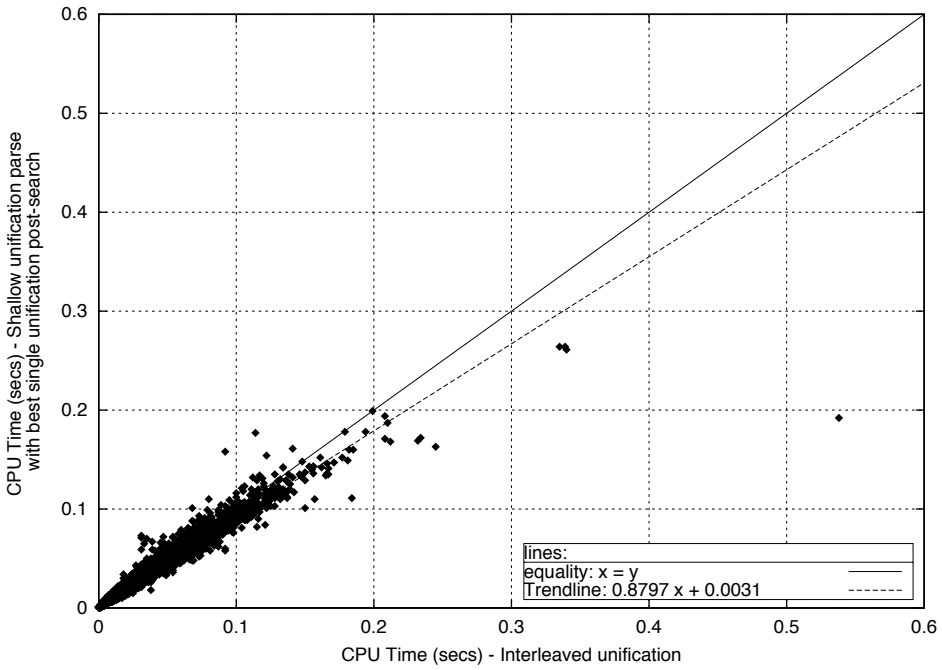


Figure 9.1: Complete test on the Catalyst 10,000-sentence evaluation corpus: Scatter-graph of parse time for full interleaved unification parse vs. wild-carded shallow unification parse plus single-best search times.

a blow-up as well, it is still much faster than doing a single fully-interleaved unification parse.

9.3 Tests on a Broadcast News grammar

Along with considerations of tuning to the test data, we might also ask to what extent our system is tuned specifically for the development grammar. In this section, we examine the behavior of the parser on a grammar other than the one we have developed for.

The domain for this test is the parsing of closed-captions from a business news broadcast [112]. This was chosen because it is a real, rather than created, domain, with quite a bit of richness in the language used.

For these next results, it is important to note that we have not attempted to correct, or even test the behavior of our system on this grammar prior to the running of these tests. The purpose of this particular evaluation is to determine how general our techniques are in practice, independent of the actual details of the grammar.

9.3.1 About the Broadcast News Grammar

The Broadcast News grammar, built by Krzysztof Czuba [29], contains 272 rules and approximately 3000 individual unification equations. This grammar is substantially smaller than the KANT grammar, though it is still non-trivial.

This grammar was also developed with different goals than the KANT grammar used throughout the development of this system. The KANT grammar was developed to conform to a carefully controlled subset of English [55]. In contrast, the Broadcast News grammar was designed for wide-coverage of unrestricted American English; as a result it is much more prone to ambiguity than the KANT grammar is.

9.3.2 Test sentences

The test set is 53 sentences, also provided by Krzysztof Czuba. These were collected via a closed-captioning reader from the "Market Wrap" cable television broadcast in 1997, and only slightly "cleaned-up" to remove obvious mistranscriptions. The sentences range in length from 2 through 38 words. Of these, 40 parse; 13 do not.

These test sentences are not particularly difficult, other than in length. Most generate only one or two distinct F-structures, though three of them generate over 90 F-structures.

The following example sentences are taken from out of this test set:

```
TOSCO sealed a deal to buy the west coast operations of UNOCAL
also known as "76 Products" company for about $1.4 billion.

TOSCO will become the nation's largest independent refinery.

UNOCAL says it will use some of the proceeds to pare down debt
and will consider a stock repurchase plan.
```

9.3.3 Results on Broadcast News transcriptions

Table 9.4 shows the run-time results for the Broadcast News 53-sentence test corpus. These results compare performing a single-pass fully-interleaved unification parse with four combinations of two-pass strategy. These four combinations are the result of combining either a Context-Free (only) parse or Wild-Carded shallow unification parse with either an all-values or single-best-value post-parse search.

Parse type	*Search type*	parse time	post-parse search time	Total time
Fully-Interleaved	*n/a*	2.11	*n/a*	2.11
Context-Free	all-values	9.37	1.09	10.46
	single-best	9.35	10.47	19.82
Wild-carded Shallow	all-values	1.94	0.38	**2.32
	single-best	1.93	0.39	**2.32

Table 9.4: Test on the Broadcast News 53-sentence evaluation corpus: break-down of total run times for Interleaved Unification, and both Restricted and Wild-carded shallow unification; each with all-values and single-best-value Post-Parse Search. ***Note that for shallow unification, one sentence incorrectly failed to parse*. This failure is examined in Section 9.3.5. Variations in parse times for identical parse configurations are due to sampling noise.

Parse type	*Search type*	*Number of nodes:*		*Number Calls to* `setvalue`:		
		Created	Searched	parse	search	total
Fully-Interleaved	*n/a*	7,057	*n/a*	456,117	*n/a*	456,117
Context-Free	all-values	25,837	103878	41,354	323,186	364,540
	single-best	25,837	259674	41,354	586,272	627,626
Wild-carded Shallow	all-values	8,968	4442	260,131	119,719	379,850
	single-best	8,968	4681	260,131	85,664	345,795

Table 9.5: Test on the Broadcast News 53-sentence evaluation corpus: break-down of: total number of nodes (*i.e*. inactive arcs) created in the parse, and searched; total number of calls to `setvalue` in parse and post-parse search.

It is important to note that in this test, 40 of the 53 sentences parse, and 13 correctly fail to parse in all configurations. However, one additional sentence fails to parse for both shallow unification configurations.

This sentence fails in the shallow first-pass parse itself. This failure demonstrates that the unification subsetting system is somewhat dependent on the grammar itself. We investigated this failure briefly, and determined that there is a subtle interaction between grammar features which the subsetting optimizer fails to account for, causing incorrect behavior for this grammar. We discuss this exceptional case in Section 9.3.5 below.

Aside from this failing sentence, the remaining results are encouraging. Without any effort to either tune the grammar to this parsing system or tune the optimizer to this grammar, we are getting acceptable results. Both of the shallow two-pass configurations are slower than the full interleaved unification configuration, but only by 10%, a small additional cost. The shallow unification grammar is creating parse forests which are 27% larger on average than those created by the Full Interleaved Unification configuration.

The Context-Free first-pass configurations show a stark contrast with the shallow unification system: both Context-Free first-pass configurations are worse than performing a Full Interleaved single-pass parse by at least a factor of ten. This slow-down is principally due to the substantially larger parse forests created by the Context-Free parse. These are larger than the parse forests created by the full interleaved unification parse by a factor of 3.7. Creating such large parse forests not only takes additional time in the first-pass parse itself, but also creates much more work for the second-pass search to do, resulting in slower times for each pass.

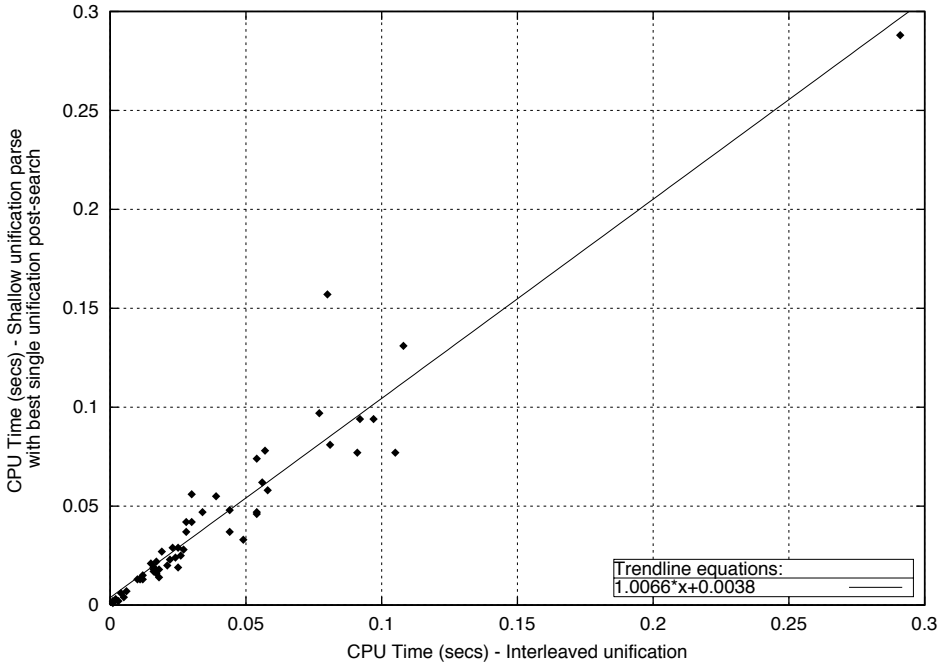


Figure 9.2: Test on the Broadcast News 53-sentence evaluation corpus: Scatter-graph of parse time for full interleaved unification parse vs. wild-carded shallow unification parse plus single-best search times.

Figure 9.2 illustrates a sentence-by-sentence comparison of the net CPU time requirements of two parser configurations. For this graph, the simple single-pass Full Interleaved Unification parser configuration is compared against the two-pass technique using Wild-carded Shallow Unification and a single-best post-parse search.

These data confirm our conclusions based on the over-all run-times; aside from the failure of one sentence, the shallow unification configuration is performing acceptably on a completely untuned system.

9.3.4 Test on Artificially Bad Data

In order to investigate the worst-case behavior of the system on the Broadcast News grammar, we once again employ the PP-attachment ambiguity test. For these data, the test was of the form:

$$\textit{That deal will be } (\textit{in stock}^n).$$

Where the prepositional phrase *in stock* is repeated one or more times.

Results on this test.

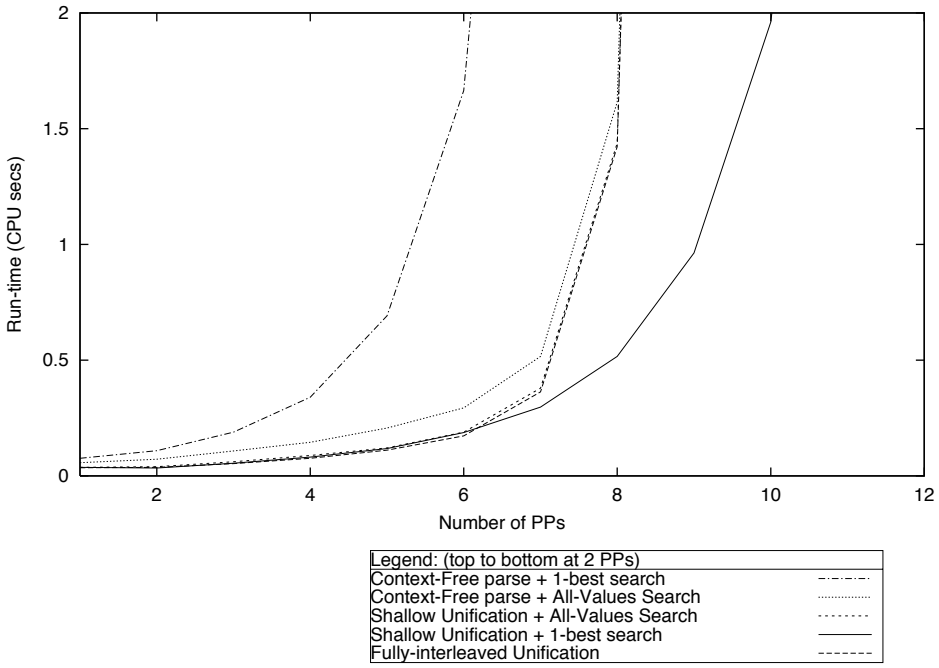


Figure 9.3: Net run-time requirements for the prepositional phrase attachment test on the Broadcast News grammar for a variety of conditions (see Legend above).

Figures 9.3 and 9.4 show the results of running the system with the Broadcast News grammar on the prepositional phrase attachment case, with run-times shown on linear and logarithmic scales respectively.

From Figure 9.3, we can see that the Context-Free parse followed by best single-value search configuration is clearly the worst, suffering a blow-up after 6 prepositional phrases.

The three all-values configurations: the single-pass Fully Interleaved Unification, and both

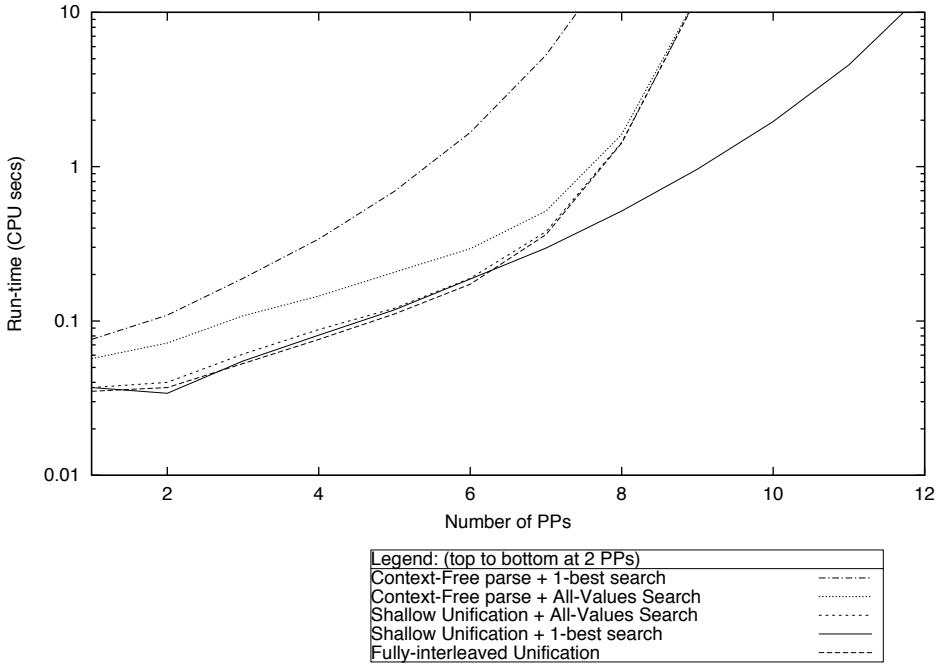


Figure 9.4: The same net run-time requirements for the PP-attachment test on Broadcast news as shown in Figure 9.3. In this graph, the run-time is plotted on a logarithmic scale.

Context-Free and Shallow Unification first-pass parses followed by an all-values search, suffer a blow-up at the same point. Beyond 8 prepositional phrases, none of these configurations exhibit acceptable run-time performance.

Finally, the shallow unification parse plus best single-value post-parse search blows up beyond 10 prepositional phrases. From Figure 9.4 we can see that this blow-up is more gentle than for the all-values configuration, and the "knee" is further out. For the 8 PP case, the shallow parse plus single-best search requires about a quarter of the run-time required by the single-pass full interleaved unification parse. That said, this configuration is still suffering from a blow-up in the unification space. While the single-value search helps reduce the effect of the blow-up for this grammar, it does not eliminate it entirely.

We investigated this blow-up in order to determine why it was occurring for the single-value search case. One of the effects we discovered was that the post-parse search was spending a lot of time trying out (partial) parse structures which were desirable from the stand-point of the disambiguator, but eliminated by the unification equations of the grammar. Worse, these structures were eliminated based tests for deep structure – exactly the kind of tests which we eliminate in the first pass. This caused the search component to spend a lot of time pursuing fruitless search paths.

A second reason for the blow-up was related to the multiple-value blow-up which occurred in outliers which we found in the Catalyst 10,000-sentence test (Section 9.2 above). Several rules in the Broadcast News grammar contain a sequence of disjunctions. Evaluation of each of these rule disjunctions may cause large numbers of data disjunctions to be created (see Section 7.1.1 above).

We found that it is beneficial to "re-pack" disjunctions, so as to remove redundant elements from them whenever they are created. For example, we would wish to replace the value:

```
(*OR* a b c a b)
```

with the value:

```
(*OR* a b c)
```

For the KANT system, we found that we could do an adequate job of eliminating redundancies by simply testing for equality by using the LISP `equal` predicate.

This `equal` test is exact for atomic values, and in the KANT grammar it is a sufficient test even for complex values. While it does not eliminate all of the redundancy in the complex values, we found that it does reduce most of this redundancy in practice.

Not so for the Broadcast News grammar. The evaluation of unification expressions tends to rearrange items in an unordered set. The evaluation of different clauses in a rule disjunction may order these rearrange the values of otherwise equivalent unification values in different ways. The sequences of disjunctions seem to create an explosion of mostly equivalent, but differently-ordered unification values. Because we are not using an exact test for the redundancy removal, these equivalent structures are not recognized as such in this grammar.

That said, adopting an exact unification equivalence test is not without cost. In our system, such a test would be substantially more computationally expensive than the LISP `equal` predicate, and the redundancy reduction is performed quite often for both of these grammars. Since more exact redundancy removal was not an issue for the KANT grammar, we have not fully investigated the computational impact of exact redundancy removal.

9.3.5 Outliers in the Broadcast News corpus

In Section 9.3, we observed two different kinds of exceptional behavior when our system was used with the Broadcast News corpus. There are two directions we could take to further improve the performance of our system on this domain.

The first option for improvement is to modify the grammar to be more compatible with the multi-pass parsing strategy. This is certainly possible, and even desirable due to the preliminary, proof-of-concept nature of the grammar. Modifications to make this grammar more amenable to the shallow parsing strategy would pay particular attention to the use of shallow versus deep features, with an understanding that deep structure tests are deferred to the second pass, and thus are expensive.

The second direction we could take would involve modifying this system to be more compatible with this grammar. These modifications would specifically include further work on the shallow unification calculator to better handle this grammar, with particular attention paid to the sentence which incorrectly fails to parse.

The longer-term improvement most likely to yield improved performance would be extending the search to use more top-down information. In investigating the unfavorable behavior of the prepositional-phrase attachment test, we found that the post-parse search was spending a lot of time trying out (partial) parse structures which were desirable from the stand-point of the disambiguator, but eliminated by the unification equations of the grammar. Worse, these structures were eliminated based on tests for deep structure – exactly the kind of tests which we eliminate in the first pass.

To take better account of these tests, we would have to include the use of more top-down information, both unification values and disambiguator cost information, in the post-parse search.

We found that including top-down constraints in the search was well beyond the scope of this work; the reasons for this are explained in Section 10.3.2, on future research directions, below.

9.3.6 Conclusions from the Broadcast News tests

Applying our techniques, without any grammar-specific specialization, to this domain is clearly extending the useful range of performance.

While our techniques do not simply avoid all possible blow-ups, they were not in fact designed to do so. Unification is a very powerful technique, and while it is possible to reduce the negative impact of the interaction of some unification features, it is not possible to eliminate them entirely.

9.4 Retrospective Examination

In the previous sections, we have seen that we have met our major goals for this system. Since much of the development effort of the system concentrated on performance, one might ask how much improvement we have made over the course of development.

In this section, we look at this question from three different angles. First, we will briefly compare this system to the GLR version 8-4 system of Tomita *et al.* [155]. After this internal comparison, we examine a summary of improvements which had the greatest impact on average-case run-time. Finally, we examine the performance of several worst-case sentences on a very early version of the system and on the latest version in order to see how well we have done at fixing the outlier problem for several known cases.

9.4.1 Comparison to GLR

Since this system was designed as a functional replacement for Tomita's (extended) GLR parser, a comparison between the two parsers is inevitable.

A direct comparison between these two parsers was complicated by the fact that we have not ported the complete GLR system to the Macintosh test environment. As a result, to compare the relative speeds of these two parsers, we will have to make a sequence of approximations.

The most recent test of GLR which we performed was in December of 1999. For this test, GLR version 8-4 [155] 5382.40 CPU-seconds to run the complete development test-set (see Section 5.4.1 above) on a Sun IPX workstation, with a 40 MHz processor, 64 MB of memory, running Lucid Common LISP 4.0.

In January of 2001, the fully interleaved configuration of an earlier version of this system took 1243.8 CPU-seconds to run on this same Sun workstation. The same configuration took 946.7 CPU-seconds on a Macintosh PowerBook 2400c with a 180 MHz PowerPC 603e processor.

Later, in July of 2001, a different previous version of this system took 513.35 CPU-seconds turn on the PowerBook 2400c, and also took 87.48 CPU-seconds on the Macintosh PowerBook G3 with a 466 MHz PowerPC processor (the final test platform).

On the basis of these numbers, we can derive an approximate comparison of the speeds of these two machines:

- Mac 603e-180 **1.31** times faster than Sun IPX.

- Mac G3-466 **5.87** times faster than Mac 603e-180.
- Therefore Mac G3-466 *approximately* **7.7** times faster than Sun IPX.

As a result, we can fairly safely assume that GLR v. 8-4 would require approximately 700 CPU-seconds to run on the Mac G3-466 ($5382.40 \div 7.7 = 698.1$).

As reported in Table 5.1 of Chapter 5, the fully interleaved configuration of the final system took a total of 82.29 CPU-seconds to parse the development test corpus.

Therefore, the fully-interleaved configuration of this system is approximately 8.5 times faster than GLR version 8-4.

9.4.2 Summary of Improvements

Table 9.6 shows the over-all cumulative effects of optimizations to the unification system. From this table, we can see that most of our improvements in average run-time performance were due to improvements in the handling of disjunctions in the unification system.

Performing this retrospective evaluation revealed an interesting bug in the original unifier. The original unifier from the GLR version 8-4 code [155] had a hidden dependency on the LISP compiler used to run it (Lucid Common LISP). When we back-ported this old unifier into our system, we discovered that many more sentences (61) in our development test-set failed due to computational blow-ups than originally occurred with the earlier GLR-based system.

Because the final system does not suffer any sort of blow-up for these sentences, we can observe a 4.2% greater coverage simply due to this effect.

Unifier type	*Search type*	*Number of Sents Parsed*	*Parse Time*	*Search Time*	*Total Time*
Original unifier:	Fully-Interleaved Unification	1363	1324.62	*n/a*	1324.62
Always Unpacking	Shallow Unification, all-paths search	1386	176.83	1973.20	2150.03
PUSH operator	Shallow Unification, single-best search	1422	175.84	68.06	243.90
	Fully-Interleaved Unification	1404	849.89	*n/a*	849.89
Conditional OR Data Unpacking	Shallow Unification, all-paths search	1408	170.91	442.47	613.38
PUSH operator	Shallow Unification, single-best search	1423	169.82	39.31	209.13
	Fully-Interleaved Unification	1423	68.35	*n/a*	68.35
No Data Unpacking	Shallow Unification, all-paths search	1423	58.90	8.82	67.72
PUSH operator	Shallow Unification, single-best search	1423	58.89	8.45	67.34
Final unifier:	Fully-Interleaved Unification	1423	68.79	*n/a*	68.79
No Data Unpacking	Shallow Unification, all-paths search	1423	58.88	8.93	67.81
no PUSH (precompilation)	**Shallow Unification, single-best search**	1423	58.97	8.51	67.48

Table 9.6: Comparison of four behaviors of unification system: the original behavior of the unification system from GLR [155] (updated only to correctly handle shallow unification features), Only performing unification data disjunction unpacking to handle conditional OR operators, with unwinding conditional ORs in the grammar preprocessing, and the final configuration: unwinding conditional ORs and PUSH operators in the grammar preprocessing.
Results are total run time for Interleaved Unification, Shallow Unification plus All-Paths Post-Parse Search, and Shallow Unification plus Single-best Post-Parse Search, on the Catalyst ambiguity development corpus (see Section 5.4). All sentences were run with a 10-CPU-second time-out.

9.4.3 Improvements for Selected Worst-Case Sentences.

We have presented a number of analyses of over-all performance of the system. In this section we analyze several particular test sentences in order to better see the improvements in the behavior of the system in detail.

For this detailed examination, we will look at a set of problematic sentences, and examine the behavior of the parser on these sentences. We will examine four versions of the unification system, concentrating on the handling of disjunctions and multiple-value PUSH commands because most of the extremely bad behavior of early versions of this system were due to poor handling of these two features of the unifier.

Disjunctions within unification values are typically generated either through the packing of ambiguous rules or the evaluation of a unification rule disjunction. As we argue in Sections 7.2 and 7.3.1 above, this unpacking is extremely dangerous. To see this in practice, we investigate the behaviors of these example sentences on four different unification system configurations:

- The original behavior of the unification system from GLR [155] (updated only to correctly handle shallow unification features). In the original KANT unification system, these data disjunctions would be "unpacked" before evaluating any unification rule that referenced a value within the scope of the disjunction.
- Only performing unification data disjunction unpacking to handle conditional OR operators. All other data disjunctions are handled by traversing the disjunctive trees in-place.
- Unwinding conditional ORs in the grammar preprocessing, so that no data disjunctions are unwound. No special handling of multiple-value (PUSH) unification equations.
- Final unification system: unwinding conditional ORs and re-writing of PUSH operators in the grammar preprocessing.

Moderate PP ambiguity

The first set of sentences we examine demonstrate a moderate level of Prepositional Phrase attachment ambiguity, and the dangers of unpacking data disjunctions:

```
A. There is a makeup valve <callout>22</callout> for each line of
   output oil.
```

B. `Bores in the block for all camshaft bearings except front bearing`

C. `Operate the engine in gear at low idle rpm for 5 minutes.`

These tree sentences exhibit a moderate level of ambiguity: sentences A and B each contain two PPs, and sentence C contains three. While these can be considered simple sentences, all are problematic for the original unifier.

Unifier type	*Search type*	*Sentence Number*	*Number of Parses*	*Num. Calls to* `setvalue`	*Total Parse + Search Time*
Original unifier	Fully-Interleaved	A	97	209,586	112.175
		B		*timeout*	*> 600*
		C		*timeout*	*> 600*
Conditional OR Data Unpacking	Fully-Interleaved	A	7	7,220	0.099
		B	16	10,524	0.209
		C	188	138,919	96.20
Final unifier	Fully-Interleaved	A	1	4,336	0.037
		B	4	5,665	0.041
		C	8	12,049	0.075
	Shallow Unification 1-best search	A	1	3,702	0.073
		B	1	4,649	0.073
		C	1	6,400	0.101

Table 9.7: Comparison of unification configurations on example sentences A,B,& C:
A. `There is a makeup valve <callout>22</callout> for each line of output oil.`
B. `Bores in the block for all camshaft bearings except front bearing`
C. `Operate the engine in gear at low idle rpm for 5 minutes.`
Results are for either Interleaved Unification, or Shallow Unification plus Single-best Post-Parse Search as indicated.

We can see from Table 9.7 that all three exhibit a blow-up for the original unifier.

Removing the unification unpacking operations for all but Conditional OR handling substantially helps, allowing sentences A and B to each parse in under a quarter second of CPU time. Sentence C remains problematic, however. Much of the CPU time goes into handling attachment ambiguities due to the final PP, and a bad interaction between the attachment ambiguity and a multiple-value assignment of the PP structures.

Removing all data unpacking from the unifier by precompiling out the Conditional ORs and the multiple-value pushes shows its worth for these sentences, allowing all of them to parse in a small

fraction of a second. Further, applying this precompilation substantially improves the parsing speed for each sentence.

Multiple-value Unification Problems

As we mentioned in Section 7.3.3, we discovered a problem related to multiple-value assignment in the unification system. Our next example sentence is one of the two test sentences which allowed us to discover this problem:

```
D. The possible sources of positive voltage on connector
   <code>J1</code> are contacts <code>J1</code>, <code>J4</code>,
   <code>J5</code>, <code>J6</code>, <code>J10</code>,
   <code>J11</code>, <code>J12</code>, <code>J13</code>,
   <code>J14</code>, <code>J15</code>, <code>J16</code>,
   <code>J17</code>, and <code>J19</code>.
```

In this sentence, there is no particular ambiguity problem. However, the unification system handles the long string of conjoined elements through the use of a multiple-value PUSH assignment.

Unifier type	*Search type*	*Number of Parses*	*Num. Calls to* `setvalue`	*Total Parse + Search Time*
Original unifier	Fully-Interleaved	9	40,774	0.836
Conditional OR Data Unpacking	Fully-Interleaved	9	39,061	0.273
EOR Precompilation	Fully-Interleaved	1	2,135,719	3.883
	Shallow Unification 1-best search	1	540,921	1.058
Final unifier	Fully-Interleaved	1	12,019	0.095
	Shallow Unification 1-best search	1	9,749	0.111

Table 9.8: Comparison of unification configurations on example sentence D:
`The possible sources of positive voltage on connector <code>J1</code> are contacts <code>J1</code>, <code>J4</code>, <code>J5</code>, <code>J6</code>, <code>J10</code>, <code>J11</code>, <code>J12</code>, <code>J13</code>, <code>J14</code>, <code>J15</code>, <code>J16</code>, <code>J17</code>, and <code>J19</code>.`
Results are for either Interleaved Unification, or Shallow Unification plus Single-best Post-Parse Search as indicated.

Table 9.8 shows an interesting facet of the development effort: fixing the disjunction unpacking caused a problem with multiple-value handling that did not previously exist.

When run with the original unifier, this sentence parses fairly quickly. Unpacking data disjunctions only for the handling of Conditional ORs shows a further benefit, giving a factor of three speed-up for sentence.

Unlike most of the other sentences, however, removing all unpacking by precompiling the Conditional ORs out of the grammar does not help, but substantially hurts the parser's performance on this sentence, resulting in a factor of over 14 slow-down of the parser.

Removal of the multiple-value PUSH equations, as described in Section 7.3.3, improves the system for this sentence once again. The combination of this technique and the precompilation of Conditional ORs improves the run-time and reduces the number of calls to `setvalue` by a substantial amount.

Extreme PP Ambiguity

Our final example demonstrates extreme prepositional phrase and subordinate clause attachment ambiguity:

```
E. A load sensing system provides a constant flow for a
   given directional control spool position by maintaining
   a constant pressure difference across the orifice that
   is made by the directional control spool.
```

This sentence contains a combination of prepositions *for, by,* and *across*, *that*, which can either be a preposition or a subordinating conjunction in the KANT grammar, and three complex noun phrases.

This sentence caused the original parser configuration to time out. In order to better understand the behavior of the parser, we also examined two simplified versions of this sentence. For each of these, we removed some portion of the end of the original sentence:

```
E1. A load sensing system provides a constant flow for a
    given directional control spool position by maintaining
    a constant pressure difference across the orifice.

E2. A load sensing system provides a constant flow for a
    given directional control spool position by maintaining
    a constant pressure difference.
```

Table 9.9 shows a summary of results for attempting to parse the original sentence, as well as the two simplified variations of it.

The original sentence proved to be too hard for the parser running with either the original unifier, or the unifier which only unpacked data disjunctions for handling of Conditional ORs. The behavior of the parser while running each of the two simplified sentences hints at the reason for this failure: the addition of the phrase *"across the orifice"* causes the parser to take over six times as long, and produces over 3.5 times as many top-level unification values for each of these two conditions. Close examination of these unification values revealed that they were largely redundant, but structured in such a way as to obscure this redundancy from the duplicate removal code in the parser.

In contrast, the parser running with Conditional OR precompilation shows a marked difference in behavior. The original sentence, which had required far too much time and memory to compute for the early configurations, now requires only a quarter CPU-second to parse, and the redundant unification values have been eliminated.

Finally, the addition of multiple-value PUSH removal further aids the parsing speed of this sentence when the system is set to perform a shallow parse followed by a 1-best post-parse search.

Conclusions from Example Sentences

All of these sentences highlight some particular aspect of improvement to the system by the removal of a problematic case. While each sentence was chosen in order to highlight some improvement, they were all taken from real data rather than being artificially constructed.

The results for these sentences highlight the over-all improvements shown in Table 9.6: coverage has been improved by 4.2%, parse times have been reduced by as much as a factor of 20, and most importantly, no known outliers remain.

Unifier type	*Search type*	*Sentence Number*	*Number of Parses*	*Num. Calls to* `setvalue`	*Total Parse + Search Time*
Original unifier	Fully-Interleaved	E2	198	83,140	6.321
		E1	771	187,582	42.315
		E		*timeout*	*> 600*
Conditional OR Data Unpacking	Fully-Interleaved	E2	306	85,099	6.329
		E1	1071	184,662	38.841
		E		*timeout*	*> 600*
EOR Precompilation	Fully-Interleaved	E2	4	17,039	0.124
		E1	8	20,027	0.141
		E	16	43,007	0.254
	Shallow Unification 1-best search	E2	1	11,228	0.121
		E1	1	12,928	0.134
		E	1	20,129	0.223
Final unifier	Fully-Interleaved	E2	4	17,191	0.121
		E1	8	20,179	0.139
		E	16	43,227	0.253
	Shallow Unification 1-best search	E2	1	11,232	0.146
		E1	1	12,932	0.160
		E	1	20,133	0.247

Table 9.9: Comparison of unification configurations on example sentence E, and two simplified variations of it:

E2. `A load sensing system provides a constant flow for a given directional control spool position by maintaining a constant pressure difference.`

E1. `A load sensing system provides a constant flow for a given directional control spool position by maintaining a constant pressure difference across the orifice.`

E. `A load sensing system provides a constant flow for a given directional control spool position by maintaining a constant pressure difference across the orifice that is made by the directional control spool.`

Results are for either Interleaved Unification, or Shallow Unification plus Single-best Post-Parse Search as indicated.

Chapter 10

Conclusions and Future Directions

> After long days of training the result begins to tell and the beast is faced round,
> A nature so wild and ungoverned is finally broken, he has become gentler;
> But the tender has not yet given him his full confidence,
> He still keeps his straw rope with which the ox is now tied to a tree.
>
> — *D.T. Suzuki, tr. Oxherding Picture number 4* [147]

10.1 Conclusion: Shallow Unification Preferable

In the experiments above, we have seen that combining the shallow unification technique with a post-parse search for a single best unification result results in a parser that is in the average case competitive with an ordinary single-pass parser (both in terms of time and space), and also can avoid a common class of ambiguity blow-up that the single-pass parser is subject to. From this, we can conclude that the multi-pass technique, interleaving *some* of the unification equations in the parse, is the superior approach.

10.2 Contributions

In this section we will summarize the contributions of this work to the state of the art Natural Language Processing.

For each contribution, we summarize the results from this study that are relevant to the conclusion, as well as any recent results of other in the literature. Finally, we try to assess the generality of each conclusion.

That said, we do not claim to be better able to perceive the hidden limitations to our study than any other researcher. Therefore, we expect that any researcher who might consider adopting our techniques will be as willing to judge these techniques experimentally as we have been willing to judge those of others.

10.2.1 Evidence that delayed unification is preferable over single-pass interleaved unification.

We demonstrated that performing a shallow interleaved unification parse, followed by some sort of search for unification values, can save unification effort and potentially time over performing an single-pass full interleaved unification parse.

This confirms the results of Maxwell and Kaplan [85] and Oepen and Carroll [115], who have found very similar results in their respective systems.

This is a very general result, as it suggests a general strategy for parsing.

10.2.2 Evidence that interleaved unification is generally preferable to fully-delayed unification evaluation

Maxwell and Kaplan [85] argued that it is preferable to delay unification processing until after a complete context-free parse forest has been found.

In contrast to this, Oepen and Carroll [115] found that it was preferable to delay the calculation of unification equations which create complex structures until after the main parse, but to calculate simple-value-producing unification equations in an interleaved fashion, as a filtering technique.

Our results confirm those of Oepen and Carroll. We found that the calculation of these simple-valued unification results can substantially reduce the amount of parse forest structure which must be searched in order to calculate a complete unification value. This interleaved unification calculation saves time both in the parse and in the post-parse search over using a simple Context-Free parse to calculate the parse forest.

This result is very general, and is potentially applicable to any unification-based system. This is due to the general filtering nature of Natural Language Unification Grammars. That our conclusion confirms that of Oepen and Carroll is therefore not a surprise.

10.2.3 Evidence that single value search is preferable over all-values search.

We also demonstrated that performing a shallow interleaved unification parse, followed by a best single-value search, is preferable over performing either a single-pass full interleaved unification parse, or a partially-unifying parse followed by a search for all possible unification values. We demonstrated that this two-pass strategy can avoid some problems which cause the single-pass parser to require resources exponentially proportional to the input.

This result is also very general, and applicable to many unification parsing systems. To summarize the arguments found in Chapters 3 and 4, any unification grammar which creates a unique representation of each distinct parse tree, running in a parser which finds all possible unification values for each input, is susceptible to an exponential resource blow-up in the face of ambiguous input.

10.2.4 A novel technique to automatically produce a shallow unification grammar from a complex-valued grammar.

We have also demonstrated that we can gain the advantages of a two-pass parsing strategy without requiring the grammar author to write two separate grammars. We do so by introducing a novel technique

While this technique is related to the Negative Restriction processing proposed by Oepen and Carroll [115], we extend this by automatically separating the existential testing and structure-building behaviors of unification equations, preserving the former in the parse pass while delaying the latter until the post-parse search. We have shown that this technique produces a significant improvement over simple negative restriction.

This technique is fairly general, since it is applicable to any complex-valued unification grammar. This technique has applicability both in the context of multi-pass parsing, and more generally in the interpretation and optimization of Natural Language Unification systems.

10.2.5 Approximation of Shallow Interleaved Unification Values.

We have demonstrated that it is preferable in our system to perform a parse pass using a shallow unification grammar, *throw out the first-pass unification results,* and calculate the final unification value or values from scratch. We demonstrated that because we are committed to completely recalculating the final unification value, we can make approximations to the first-pass unification values with impunity. We have investigated several approximations which reduce the cost of the parse-pass

unifications and improve the ability of the parser to effectively pack local ambiguities together.

Our use of approximation and complete recalculation of the final unification value is in contrast to the common usage of unification Restriction currently in the literature. Previously, both positive restriction [136] and Negative Restriction [45, 156, 115] have been applied to the grammar in order to calculate partial exact unification results. These results are then augmented with the remaining restricted unification equations to compute a complete unification value.

This technique is somewhat less general than those above, as it is specifically limited to multi-pass unification systems where all but the final unification values are discarded. That said, we have shown that the strategy of discarding unification values calculated early in the process is a good one in practice for multi-pass parsing systems. Once such a strategy has been adopted, the approximation of unification values in the earlier passes becomes applicable.

10.2.6 Techniques in Unification Optimization

One valid criticism of current work in Computational Linguistics in general, and parsing of natural language in particular, is that the field has become somewhat disconnected from the study and construction of compilers for computer languages.

We have demonstrated that many of the successful high-level optimization techniques commonly used in compilers [1] are adaptable to unification parsing. We do so by treating unification as a programming language, and optimize this language by reasoning about the effects of each of the unification statements.

Although the specific techniques we demonstrated are applicable to our specific pseudo-unification system, the general strategy of treating unification as a programming language is much more generally applicable. When such a strategy is adopted, finding ways to optimize that programming language naturally follows.

In addition, we also presented several *pseudo-optimizations* for the shallow grammar — operations which produce unification results that are similar to, but not exactly equal to the more expensive operations they replace. These are quite specific to our system, but naturally follow from the combination of strategies we chose to adopt. We have no doubt that similar optimizations could be found in other environments.

10.2.7 Single-value Post-Parse Search Algorithm

We have proposed a novel technique to simultaneously disambiguate and calculate a single unification value corresponding to a single preferred parse tree. In order to perform this calculation, we treat the problem as one of search, rather than of simple calculation.

We present a novel algorithm to perform this single-value post-parse search, and prove the optimality of our algorithm.

This technique is in contrast to the all-values calculations proposed by *e.g.* Maxwell and Kaplan [85], and Oepen and Carroll [115].

This algorithm is fairly general, and with some adaptation should be applicable to a wide variety of parsing tasks related to the calculation of a single parse tree.

This algorithm is not dependent on the use of a chart parser in the first pass, but only depends on the parser producing a packed parse forest as a result. The specific algorithm depends on the parse forest being of a particular form, but could be adapted to other forms of representation.

10.2.8 Advances to Context Free parsing

We have demonstrated that we can substantially reduce the amount of processing in a Chart Parser by structuring the grammar into a tree. To support this, we developed a novel variant of the Chart algorithm [34, 58] which efficiently uses this grammar structure. We have also shown how to integrate this tree-structured grammar with Left-Corner filtering, and that these two techniques work well combination.

Although this technique was developed specifically for the chart parsing algorithm, it is potentially applicable to other context-free parsing algorithms as well. (It is important to note, however, that this optimization is implicit in the optimization of states of an L-R parser [2], and thus is already done in *e.g.* GLR [151].)

10.2.9 A complete description of implementation.

The results of this work rest on an efficient parser implementation — all of the experimental results depend on the quality of this implementation. In order to promote the use of these techniques, we describe them fully, including detailed descriptions of the implementation of each piece.

10.2.10 Empirical investigation of many engineering trade-offs.

In many cases the superiority of some technique over a contrasting one is not clear. We have performed targeted empirical investigations of many such choices in order to make good engineering choices. These results have been presented throughout this work.

Although the specific results of these trade-offs are specific to our system, we hope that other researchers in Computational Linguistics will take similar care when implementing their systems. We therefore offer these detailed investigations as exemplars of Software Engineering practice as applied to Computational Linguistics.

10.3 Future Directions

There are a number of fairly straight-forward techniques that could be added to this system, but are additional improvements rather than being central to the design and structure of the system. Since these techniques are not critical to the demonstration that the over-all system can meet the goals of being simultaneously good according to the standards of Linguistics, Theoretical Computer Science, and Software Engineering, these techniques can be safely classified as beyond the scope of this work. Below are several such directions.

10.3.1 Dynamic Optimization of the Shallow Interleaved Unifier

We presented a method to statically reduce a complex-valued unification grammar to form a shallow unification grammar. We also presented several techniques to statically optimize this shallow unification grammar.

An obvious extension to the static optimization of the first-pass shallow unification grammar is to further optimize the grammar by using information about how the unification values are used dynamically.

It is extremely likely that some features will limit a set of parses more than other features. One could imagine determining which features are the "best" at doing this (by some measure). Armed with this additional information, one could both re-order the interleaved unification equations in order to be more likely to fail (and thus prune) early, and also to simply remove tests that are extremely unlikely to fail, or are in fact never tested against, as redundant.

It should be obvious that both of these techniques *could potentially* speed up the first pass. The work then is to figure out how to collect the data, how to form the measure (*e.g.* maximum

likelihood, maximum entropy, etc.), whether re-ordering in fact helps, and for what sort of cut-off is redundant elimination best.

Because the current system does not use a pure unification system (as we have discussed above in Section 3.2), the unification equations cannot be simply re-ordered without potentially changing the semantics of the unification equations taken as an ordered set. As a result, these suggested modifications are less trivial than they might seem at first glance.

10.3.2 Top-down constraints in the post-parse search

Unification constraints

There is an outstanding technical question involving the propagation of constraints (both unification constraints and disambiguation preferences): Should some constraints be propagated from the top, downward during the post-parse search?

For this system, all unification is done in a bottom-up manner: both in the interleaved unification of the first pass parser, which operates in a fundamentally bottom-up manner, and in the second pass search where the unification is done while returning out of the tree search, which makes it a bottom-up process as well.

There is good reason to believe that some portions of the second pass search could be avoided by percolating unification constraints down the search tree starting at the root (whole sentence) nodes. For example, consider the following three rules with unification equations:

$$
\begin{array}{lcll}
S1 & \leftarrow & R\ T & (1) \\
 & & (x_1\ \mathit{foo}) = + & \\
 & & x_0 = x_2 & \\
 & & & \\
T & \leftarrow & X\ Y & (2) \\
 & & x_0 = x_1 & \\
 & & (x_0\ \mathit{baz}) = x_2 & \\
 & & (x_0\ \mathit{foo}) = - & \\
 & & & \\
T & \leftarrow & X\ Z & (3) \\
 & & x_0 = x_1 & \\
 & & (x_0\ \mathit{baz}) = x_2 & \\
 & & (x_0\ \mathit{foo}) = + &
\end{array}
$$

If these are evaluated in a top-down manner, rule 1 will be evaluated first; this will have the side-effect of causing any T constituent which is attached to this rule as a child to have the unification

value *foo* set to the value "+". If rules 2 and 3 are attempted in order to fill this open non-terminal, rule 2 will fail because it requires *foo* to be set to "-", whereas rule 3 may succeed because this setting is compatible.

In a pure unification system, the propagation of unification constraints is easily accomplished, though we will note that this advantage is not in our view sufficient to counterbalance the computational problems which we discussed in Section 3.3.7.

If we had a pure pseudo-unification [153] system, we could consider this as well. This is true because the side-effects of assigning values could be reasonably controlled, and it should not matter whether the side-effect values are propagated up or down the tree of a given parse.

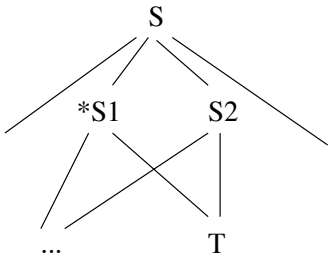


Figure 10.1: Propagation of Unification features in a Packed Forest

That said, the action of propagating values *down* the tree is not without cost. Consider Figure 10.1; suppose we have sub-tree T, which is used in analysis S1 and S2.

Suppose that both S1 and S2 will be successful, but that S1 side-effects the values of T in such a way that the top-level S unifications will rule this out. It is important in this case not to allow any side-effects of S1 to modify any values used in S2, since the existence of S1 should not have any effect on the unification values derived through S2.

Alternately, suppose that S2 will ultimately be successful, but S1 will not, and that search will examine S1 before S2. It is important that any top-down constraints in S1 *not* be allowed to effect the unification value in T — otherwise these values could have an erroneous effect on S2. In a top-down search, it is important to search T while descending from the S2 path, even if T (or some part of T) was ruled out while descending from S1. In the general case we can only save and re-use partial result values which are all calculated in the same direction. So if we are propagating results downward, and ultimately need to back-track up the tree, any results we might have calculated must be discarded in order to assure correct evaluation of these same unification values should we choose an alternate path down to the same lower level parts of the tree.

Because of the need to discard results from incompatible paths, downward propagation of unifi-

cation values may very well lead to *more* computation, rather than less. In our second scenario, the evaluation of T as a child of S1 would cause T to be left not fully evaluated, so a full descent would be required while descending from S2.

Without any top-down constraints, but instead calculating all values in a strictly bottom-up fashion, on the way "back up" the tree, T *would* be fully searched during the S1 search, and when we got to S2 we would not have to recompute any values in T, but could simply (and correctly) re-use a memoized value from our previous visit there.

Also, if the search is restricted to a fairly small superset of the actual valid parses, this downward propagation of unification values will only result in roughly twice as many unification operations as we actually need. Because we ultimately want the unification value from a top-level node as a result of calculating the unification values from the bottom up, all of this proposed top-down unification work only serves to better prune the search.

Whether this will ultimately improve the performance of the second pass search is an empirical question. Unfortunately, it is not a question we can readily answer. The grammar we are currently using includes call-outs to (LISP) functions. Many of these call-outs assume that the unification values are calculated in a strictly bottom-up fashion — in some cases, calling these call-outs with top-down values will cause the functions to fail because they use values which are calculated from further down the tree. As these values would not have been computed at the time these call-out are called, they will erroneously fail.

This is a research direction which we could not conveniently investigate within the bounds of this thesis. The unification framework that we use in this system is not pure — the side-effects of unifications do not propagate backwards. In the bottom-up-oriented unification case, when a unification function (a set of unification equations) is applied to the values associated with a set of child constituents, the calculated result is then associated with the parent constituent. In the process of evaluating this function, the children's values may be changed due to side-effects of the individual unification equations. These modified values are *not* propagated downward, but are instead discarded.

As a result, we must leave the question of whether top-down unification constraints are better or worse as open for future investigation.

Disambiguation preference constraints

The question of whether it is better or worse to impose top-down constraints on the disambiguation cost calculator mirrors this same question for unifications.

The problem is that for the search as described in Chapter 8, the disambiguation cost is not really separate from the unification value (*i.e.* F-structure). If a different set of constraints are propagated down to a node, any memoized values for that node must be thrown out and recalculated from scratch, potentially losing the advantage of memoization.

10.3.3 Automatic Promotion of shallow unification features into the Context-Free grammar spine.

One interesting suggestion of Maxwell and Kaplan [85] was to promote some simple feature values into the context-free spine itself, by subcategorizing the relevant sub-set of rules.

So, for example, if we discover that a VP attachment is often done when `foo` has value `+`, but not when it is `-`, then we might contemplate creating two complete sets of all the VP creation rules, one marked *e.g.* `VP_foo_+` and another marked `VP_foo_-`. If these duplications are promoted sufficiently far "down" the grammar, then we could potentially save a lot of time by entirely avoiding the evaluation of rules that specify the wrong setting of `foo`.

Maxwell and Kaplan propose this, but wrote that they ended up doing the splitting entirely by hand. Since we will be calculating which features of the shallow unification are least useful, by symmetry we will also be able to tell which are most useful.

One possible way to gain extra speed in parsing might be to promote these features into the context-free spine of the grammar, in a way similar to that proposed by Maxwell and Kaplan. If one parameterizes either the minimum utility needed for promotion, or the number of parameters to promote into the grammar, the result would be a nicely automatic method for optimizing the grammar.

10.3.4 Optimization of the Context-Free grammar.

One of the more ambitious classes of optimizations that could be performed involves recognizing and changing the structure of the context-free grammar for the first pass, and then programmatically compensating for this change during the second.

One such optimization is *PP-folding* [145]. Consider the context free grammar:

$$\begin{aligned} NP2 &\leftarrow NP2 \quad PP \\ NP2 &\leftarrow NP1 \end{aligned}$$

Suppose that we could recognize such a construction, and for the first pass substitute the following:

$$NP2 \quad \leftarrow \quad NP1 \quad (PP)*$$

(Where * is the Kleene-closure star [48], *i.e.* (PP)* refers to zero or more repetitions of PP.)

It should be evident that these two constructions generate (and thus recognize) the same language. It should also be evident that the latter does not suffer from the ambiguity problems of the former, because it implicitly gathers all the ambiguous attachments together.

Of course, for the purposes of the second pass search, whenever the second construction is encountered, the first construction would have to be simulated for the purposes of calculating the unification value. The reason that this is an ambitious optimization is that this substitution will interact with the post-parse search and disambiguation cost calculator described above in Chapter 8. This interaction may be subtle and difficult to resolve. Consider, for example, the following four rules:

$$\begin{array}{llll} VP & \leftarrow & V & NP2 \\ VP & \leftarrow & VP & PP \\ NP2 & \leftarrow & NP2 & PP \\ NP2 & \leftarrow & NP1 & \end{array}$$

It might be possible to programmatically combine these into:

$$VP \quad \leftarrow \quad V \quad NP1 \quad (PP)*$$

But simulating the original rules in the second pass may be difficult and involved. Since we have already attacked the problem of ambiguity in the chart parser itself, it is not clear that we would gain anything computationally by this transformation, and even if such an advantage does exist it is by no means clear that the extra trouble of simulating the original rules would be worth the gain.

Bibliography

[1] Alfred V. Aho, Ravi Sethi, and Jeffrey D. Ullman. *Compilers: Principles, Techniques, and Tools*. Addison-Wesley Series in Computer Science and Information Processing. Addison-Wesley, Reading, MA, 1985.

[2] Alfred V. Aho and Jeffrey D. Ullman. *Principles of Compiler Design*. Addison-Wesley Series in Computer Science and Information Processing. Addison-Wesley, Reading, MA, 1977.

[3] James Allen. *Natural Language Understanding*. Benjamin/Cummings, Redwood City, CA, second edition, 1995.

[4] Hiyan Alshawi, editor. *The Core Language Engine*. ACL-MIT Press Series in Natural Language Processing. MIT Press, Cambridge, MA, 1992.

[5] Apple Computer, Inc. *Inside Macintosh*, volume 1. Addison-Wesley, Reading, MA, 1985.

[6] Ronald Baecker. Towards an efficient characterization of graphical interaction. In Baecker and Buxton [7], pages 471–481.

[7] Ronald M. Baecker and William A. S. Buxton, editors. *Readings in Human-Computer Interaction: A Multidisciplinary Approach*. Morgan Kaufmann Publishers, Los Altos, CA, 1987.

[8] K. L. Baker, A. M. Franz, P. W. Jordan, T. Mitamura, and E. Nyberg. Coping with ambiguity in a large-scale machine translation system. In *Proc. 15th Intl. Conf. on Computational Linguistics (COLING-94)*, pages 90–94, Kyoto, August 1994. ACL.

[9] Kathryn L. Baker. *Delayed Evaluation of Linguistic Constraints*. Ph.D. thesis, Carnegie Mellon University, Language Technologies Institute, CMU, Pittsburgh PA, May 2002. CMU-LTI-02-173.

[10] G. E. Barton, R. C. Berwick, and E. S. Ristad. *Computational Complexity and Natural Language*. Computational Models of Cognition and Perception. MIT Press, Cambridge, MA, 1987.

[11] Jon Bentley. *Writing Efficient Programs*. Prentice-Hall, Englewood Cliffs, NJ, 1982.

[12] Jon Bentley. *Programming Pearls*. Addison-Wesley, Reading, MA, 1986.

[13] Jon Bentley. *More Programming Pearls*. Addison-Wesley, Reading, MA, 1988.

[14] A. Berger, S. Della Pietra, and V. Della Pietra. A Maximum Entropy Approach to Natural Language Processing. *Computational Linguistics*, 22(1):39–71, 1996.

[15] Sylvie Billot and Bernard Lang. The Structure of Shared Forests in Ambiguous Parsing. In *Proc. 27th Ann. Mtg. Assn. Comp. Ling.*, pages 143–151, Vancouver, June 1989. ACL.

[16] M. Bouckaert, A. Pirotte, and M. Snelling. Efficient Parsing Algorithms for General Context-free Grammars. *Inf. Sci.*, 8(1):1–26, Jan 1975.

[17] R. Brachman and H. Levesque. The Tractability of Subsumption in Frame-Based Description Languages. In *Proc. AAAI-84*, pages 34–37, Austin, TX, August 1984. Wm. Kaufmann, Inc.

[18] T. Briscoe and J. Carroll. Generalized Probabilistic LR Parsing of Natural Language (Corpora) with Unification-Based Grammars. *Computational Linguistics*, 19(1):25–59, 1993.

[19] Ralf D. Brown. Example-Based Machine Translation in the Pangloss System. In *Proc. 16th Intl. Conf. on Computational Linguistics (COLING-96)*, pages 169–174, Copenhagen, August 1996.

[20] J. G. Carbonell and M. Tomita. Knowledge-based machine translation, the CMU approach. In Sergei Nirenburg, editor, *Machine Translation: Theoretical and methodological issues*, chapter 5, pages 68–89. Cambridge University Press, Cambridge, England, 1987.

[21] Jaime G. Carbonell and Philip J. Hayes. Recovery Strategies for Parsing Extragrammatical Language. *Computational Linguistics*, 9(3–4):123–146, 1983.

[22] Bob Carpenter and Gerald Penn. ALE The Attribute Logic Engine (User's Guide). Technical report, Carnegie Mellon University, Lab for Comp. Ling., CMU, Pittsburgh PA, 1994. CMU-LCL-94-6.

[23] John Andrew Carroll. *Practical Unification-based Parsing of Natural Language*. Ph.D. thesis, University of Cambridge, Computer Laboratory, September 1993.

[24] Eugene Charniak. A Parser with Something for Everyone. In M. King, editor, *Parsing Natural Language*, pages 117–149. Academic Press, London – New York, 1983.

[25] Noam Chomsky. Three models for the description of language. *IRE Transactions P.G. on Info. Theory*, 2(3):113–124, 1956.

[26] The Columbia World of Quotations, 1996. available at: `http://www.bartleby.com/66/`.

[27] Thomas H. Cormen, Charles E. Leiserson, and Ronald L Rivest. *Introduction to Algorithms*. McGraw-Hill and MIT Press, Cambridge, MA, 1990.

[28] Krzysztof Czuba, 1998. personal communication.

[29] Krzysztof Czuba. Efficient parsing strategies for syntactic analysis of closed captions. In *Proc. 1st. NAACL Student Workshop*, Seattle, May 2000. ACL.

[30] Craig A. Damon. *Selective Enumeration*. Ph.D. thesis, Carnegie Mellon University, Computer Science, CMU, Pittsburgh PA, July 2000. CMU-CS-00-151.

[31] Rina Dechter and Judea Pearl. The Optimality of A*. In L. Kanal and V. Kumar, editors, *Search in Artificial Intelligence*, chapter 5, pages 166–199. Springer-Verlag, New York, 1988.

[32] F. L. DeRemer. Simple LR(k) grammars. *Comm. ACM*, 14(7):453–460, July 1971.

[33] Jochen Dörre and Andreas Eisele. Feature logic with disjunctive unification. In *Proc. 13th Intl. Conf. on Computational Linguistics (COLING-90)*, volume 2, pages 100–105, Helsinki, 1990.

[34] Jay Earley. An efficient contest-free parsing algorithm. *Comm. ACM*, 13(2):94–102, February 1970.

[35] Encyclopædia Britannica, Inc. Encyclopædia britannica online. Availiable at: `http://www.eb.com`.

[36] Alex Franz. TUT: A tree unification toolkit. Technical report, Carnegie Mellon University, Lang. Tech. Inst., CMU, Pittsburgh PA, 1992. CMU-CMT-92-131.

[37] Christopher W. Fraser, David R. Hanson, and Todd A. Proebsting. Engineering a simple, efficient code generator generator. *ACM Letters on Programming Languages and Systems*, 1(3):213–226, Sep 1992.

[38] Lyn Frazier. *On Comprehending Sentences: Syntactic Parsing Strategies*. Ph.D. thesis, University of Connecticut, 1978.

[39] G. Gazdar, E. Klein, G. Pullum, and I. Sag. *Generalized Phrase Structure Grammar*. Harvard U. Press, Cambridge, MA, 1985.

[40] Kenneth Goodman and Sergei Nirenburg, editors. *The KBMT Project: A Case Study in Knowledge-Based Machine Translation*. Morgan Kaufmann, San Mateo, CA, 1991.

[41] S. L. Graham, M. A. Harrison, and W. L. Ruzzo. An improved context-free recognizer. *ACM Trans. Programming Languages and Systems*, 2(3):415–462, 1980.

[42] T. V. Griffiths and S. R. Petrick. On the relative efficiencies of context-free grammar recognizers. *Comm. Assoc. Computing Mach.*, 8(5):289–300, 1965.

[43] Abelson H., G. J. Sussman, and J. Sussman. *Structure and Interpretation of Computer Programs*. The MIT EECS Series. MIT Press, Cambridge, MA, 1985.

[44] Samuel P. Harbison and Guy L. Steele Jr. *C: A Reference Manual*. Tartan Laboratories/Prentice Hall, New Jersey, 1984.

[45] S. P. Harrison and T. M. Ellison. Restriction and termination in parsing with feature-theoretic grammars. *Computational Linguistics*, 18(4):519–530, 1992.

[46] Paul Heckel. *The Elements of Friendly Software Design*. Warner Books, New York, 1984.

[47] Donald Hindle and Mats Rooth. Structural ambiguity and lexical relations. *Computational Linguistics*, 19(1):103–120, 1993.

[48] John E. Hopcroft and Jeffrey D. Ullman. *Introduction to Automata Theory, Languages, and Computation*. Addison-Wesley Series in Computer Science. Addison-Wesley, Reading, MA, 1979.

[49] W. John Hutchins and Harold L. Somers. *An Introduction to Machine Translation*. Academic Press, 1992.

[50] Ray Jackendoff. $\overline{X}$ *Syntax: A Study of Phrase Structure*. Linguistic Inquiry. MIT Press, Cambridge, MA, 1977.

[51] Bonnie John, 2001. personal communication.

[52] John F. Kennedy, *speaker*. Ich bin ein Berliner. Availiable at: `http://www.debateinfo.com/hall_of_fame/speeches/jfk2.html`, June 1963.

[53] Mark Johnson. The computational complexity of GLR parsing. In Masaru Tomita, editor, *Generalized LR Parsing*, chapter 3, pages 35–42. Kluwer, Dordrecht, 1991.

[54] Hans Kamp and Uwe Reyle. *From Discourse to Logic*. Studies in Linguistics and Philosophy. Kluwer, Dordrecht, student edition, 1993.

[55] C. Kamprath, A. Adolphson, T. Mitamura, and E. Nyberg. Controlled Language for Multilingual Document Production: Experience with Caterpillar Technical English. In *Proc. Second Int. Workshop on Controlled Language Applications (CLAW '98)*, 1998.

[56] Ronald M. Kaplan and Joan Bresnan. Lexical-functional grammar: A formal system for grammatical representation. In *The Mental Representation of Grammatical Relations*, chapter 4, pages 173–281. MIT Press, Cambridge, MA, 1982.

[57] Robert T. Kasper. A unification method for disjunctive feature descriptions. In *Proc. 25th Ann. Mtg. Assn. Comp. Ling.*, pages 235–242, Stanford, 1987. ACL.

[58] Martin Kay. Algorithm schemata and data structures in syntactic processing. In *Readings in Natural Language Processing*. Morgan Kaufmann, Los Altos, CA, 1986 (1980).

[59] B. Kiefer, H.-U. Krieger, J. Carroll, and R. Malouf. A bag of useful techniques for efficient and robust parsing. In *Proc. 37th Ann. Mtg. Assn. Comp. Ling.*, pages 473–480. ACL, 1999.

[60] J. P. Kimball. Seven principles of surface structure parsing in natural language. *Cognition*, 2(1):15–47, 1973.

[61] James R. Kipps. GLR parsing in time $O(n^3)$. In Masaru Tomita, editor, *Generalized LR Parsing*, chapter 4, pages 43–59. Kluwer, Dordrecht, 1991.

[62] R. Kittredge and J. Lehrberger, editors. *Sublanguage: Studies of Language in Restricted Semantic Domains*. deGruyter, Berlin – New York, 1982.

[63] Kevin Knight. Unification: A multidisciplinary survey. *ACM Computing Surveys*, 21(1), March 1989.

[64] D. E. Knuth. On the translation of languages from left to right. *Inf. Contr.*, 8(6):607–639, December 1965.

[65] Vipin Kumar and Laveen N. Kanal. The CDP: A Unifying Formulation for Heuristic Search, Dynamic Programming, and Branch-and-Bound. In L. Kanal and V. Kumar, editors, *Search in Artificial Intelligence*, chapter 1, pages 1–27. Springer-Verlag, New York, 1988.

[66] Peter Ladefoged. *A Course in Phonetics*. Harcourt Brace Jovanovich, New York, second edition, 1982.

[67] Butler Lampson. Hints for computer system design. *IEEE Software*, 1(1), January 1984.

[68] Alon Lavie. *GLR*: A Robust Grammar-Focused Parser for Spontaneously Spoken Language*. Ph.D. thesis, Carnegie Mellon University, Computer Science, CMU, Pittsburgh PA, May 1996. CMU-CS-96-126.

[69] Alon Lavie. GLR*: A robust parser for spontaneously spoken language. In *Proc. ESSLLI-96 Workshop on Robust Parsing*, 1996.

[70] Alon Lavie, 1998. personal communication.

[71] Alon Lavie, 1999. personal communication.

[72] Alon Lavie, February 2000. public comment at 6th Intl. Wkshp on Parsing Technologies.

[73] Alon Lavie, 2000. personal communication.

[74] Alon Lavie, 2001. personal communication.

[75] Alon Lavie and Carolyn P. Rosé. Towards optimal ambiguity packing in unification-augmented context-free parsers, 1999.

[76] Alon Lavie and Carolyn P. Rosé. Optimal Ambiguity Packing in Context-Free Parsers with Interleaved Unification. In *Proc. 6th Intl. Wkshp on Parsing Technologies*, pages 147–158, Trento, Italy, February 2000. ACL/SIGPARSE.

[77] Kevin Lenzo, 1997. personal communication.

[78] Harry R. Lewis and Larry Denenberg. *Data Structures and Their Algorithms*. Harper Collins, New York, 1991.

[79] Low End Mac. PowerBook G3 Series. Availiable at: `http://www.lowendmac.com/pb2/g3series.shtml`.

[80] soc.culture.german faq. Availiable at: `http://www.watzmann.net/scg/index.html`, March 2001.

[81] William A. Martin, Keneth W. Church, and Ramesh S. Patil. Preliminary analysis of a breadthfirst parsing algorithm: Theoretical and experimental results. Technical report, Massachusetts Institute of Technology, Lab. for Computer Science, MIT, Cambridge MA., June 1981. MIT/LCS/261.

[82] William A. Martin, Keneth W. Church, and Ramesh S. Patil. Preliminary analysis of a breadth-first parsing algorithm: Theoretical and experimental results. In Leonard Bolc, editor, *Natural Language Parsing Systems*, pages 267–327. Springer-Verlag, New York, 1987.

[83] Y. Matsumoto and *et al*. BUP: A Bottom-Up Parser Embedded in Prolog. *New Generation Computing*, 1:145–158, 1983.

[84] John T. Maxwell and Ronald M. Kaplan. An overview of disjunctive constraint satisfaction. In *Proc. (1st) Intl. Wkshp on Parsing Technologies*, pages 18–27, Pittsburgh, August 1989. Carnegie Mellon.

[85] John T. Maxwell and Ronald M. Kaplan. The interface between phrasal and functional constraints. *Computational Linguistics*, 19(4):571–589, 1993.

[86] Tom McArthur, editor. *The Oxford Companion to the English Language*. Oxford University Press, 1992. available at: `http://www.xrefer.com/entry/442255`.

[87] T. Mitamura and E. Nyberg. Controlled English for Knowledge-Based MT: Experience with the KANT System. In *Proc. TMI-95*, 1995.

[88] T. Mitamura, E. Nyberg, K. Baker, D. Svoboda, E. Torrejon, and M. Duggan. The KANTOO MT System: Controlled Language Checker and Knowledge Maintenance Tool. In *Proc. NAACL 2001 (Demonstration)*, 2001.

[89] T. Mitamura, E. Nyberg, and J. Carbonell. An efficient interlingua translation system for multi-lingual document production. In *Proc. 3rd Machine Translation Summit*, Washington D.C., 1991.

[90] T. Mitamura, E. Nyberg, E. Torrejon, and R. Igo. Multiple Strategies for Automatic Disambiguation in Technical Translation. In *Proc. TMI-99*, 1999.

[91] Teruko Mitamura, 2000. personal communication.

[92] Yusuke Miyao. Packing of feature structures for efficient unification of disjunctive feature structures. In *Proc. 37th Ann. Mtg. Assn. Comp. Ling.*, pages 579–584. ACL, 1999.

[93] Robert Moore, 2000. personal communication.

[94] Robert Moore. Improved left-corner chart parsing for large context-free grammars. In *Proc. 6th Intl. Wkshp on Parsing Technologies*, pages 171–182, Trento, Italy, February 2000. ACL/SIGPARSE.

[95] Robert Moore and Hiyan Alshawi. Syntactic and Semantic Processing. In Alshawi [4], pages 129–148.

[96] Frank Morawietz. Chart parsing as constraint propagation. In *Proc. 6th Intl. Wkshp on Parsing Technologies*, pages 315–316, Trento, Italy, February 2000. ACL/SIGPARSE.

[97] Stephen E. Morrisson. unpublished manuscript, 1983.

[98] Stephen E. Morrisson, 2002. personal communication.

[99] Steven S. Muchnick. *Advanced Compiler Design and Implementation*. Morgan Kaufmann, San Francisco, 1997.

[100] Mikio Nakano. Constraint projection: An efficient treatment of disjunctive feature descriptions. In *Proc. 29th Ann. Mtg. Assn. Comp. Ling.*, pages 307–314, Berkeley, 1991. ACL.

[101] M.-J. Nederhof and G. Satta. Efficient tabular lr parsing. In *Proc. 34th Ann. Mtg. Assn. Comp. Ling.*, pages 239–246, Santa Cruz, June 1996. ACL.

[102] Mark-Jan Nederhof. Generalized left-corner parsing. In *Proc. 6th Conf. European Assn. Comp. Ling.*, pages 305–314, Utrecht, The Netherlands, 1993. ACL.

[103] Mark-Jan Nederhof. An optimal tabular parsing algorithm. In *Proc. 32nd Ann. Mtg. Assn. Comp. Ling.*, pages 117–124, Las Cruces NM, 1994. ACL.

[104] Mark-Jan Nederhof and Janos J. Sarbo. Increasing the applicability of lr parsing. Technical report, Katholieke Universiteit Nijmegen, Dept. of Infomatics, Toernooiveld, 6525 ED Nijmegen, The NETHERLANDS, March 1993. NIJT-93-06.

[105] Greg Nelson and Derek C. Oppen. Fast decision procedures based on congruence closure. *J. Assn. Comp. Mach.*, 27(2):356–364, April 1980.

[106] George L. Nemhauser and Laurence A. Wolsey. *Integer and Combinatorial Optimization*. John Wiley & Sons, New York, 1988.

[107] G. van Noord, M-J. Nederhof, R. Koeling, and G. Bouma. Conventional Natural Language Processing in the NWO Priority Programme on Language and Speech Technology. Technical report, Rijksuniversiteit Groningen, Vakgroep Alfa-informatica & BCN, March 1996.

[108] Gertjan van Noord. An efficient implementation of the head-corner parser. *Computational Linguistics*, 23(3):425–456, March 1997.

[109] Donald A. Norman. *The Psychology of Everyday Things*. Basic Books, New York, 1988.

[110] E. Nyberg and T. Mitamura. The KANT System: Fast, Accurate, High-Quality Translation in Practical Domains. In *Proc. 14th Intl. Conf. on Computational Linguistics (COLING-92)*, 1992.

[111] E. Nyberg and T. Mitamura. Controlled Language and Knowledge-Based Machine Translation: Principles and Practice. In *Proc. First Int. Workshop on Controlled Language Applications (CLAW '96)*, 1996.

[112] E. Nyberg and T. Mitamura. A real-time mt system for translating broadcast captions. In *Proceedings of MT Summit VI*, 1997.

[113] E. Nyberg and T. Mitamura. The KANTOO Machine Translation Environment. In *Proc. AMTA 2000*, 2000.

[114] Eric Nyberg, 1998. personal communication.

[115] Stephan Oepen and John Carroll. Ambiguity Packing in Constraint-based Parsing — Practical Results. In *Proc. 1st. NAACL*, Seattle, May 2000. ACL.

[116] Paul W. Placeway. Tree-structured chart parsing. In *Proc. 6th Intl. Wkshp on Parsing Technologies*, pages 317–318, Trento, Italy, February 2000. ACL/SIGPARSE.

[117] Paul W. Placeway. Tree-structured chart parsing with left-corner and look-ahead constraints. Technical report, Carnegie Mellon University, Lang. Tech. Inst., Carnegie Mellon U., Pittsburgh, PA, 2000. CMU-LTI-00-161.

[118] Carl Pollard and Ivan A. Sag. *Head-Driven Phrase Structure Grammar*. Studies in Contemporary Linguistics. U. Chicago Press, Chicago, 1994.

[119] V. R. Pratt. Lingol – a progress report. In *Advance Papers of the Fourth International Joint Conference on Artificial Intelligence*, pages 422–428, Tbilisi, Georgia, USSR, 1975.

[120] Roger S. Pressman. *Software Engineering: A Practitioner's Approach*. McGraw-Hill, New York, 5th edition, 2001.

[121] Elaine Rich and Kevin Knight. *Artificial Intelligence*. McGraw-Hill, New York, etc., 2nd edition, 1991.

[122] J. J. Robinson. Diagram: A grammar for dialogues. *Comm. Assoc. Computing Mach.*, 35(1):27–47, Jan 1982.

[123] Carolyn P. Rosé. *Robust Interactive Dialogue Interpretation*. Ph.D. thesis, Carnegie Mellon University, School of Computer Science, CMU, Pittsburgh PA, 1998. CMU-CS-96-126.

[124] Carolyn P. Rosé, 1999. personal communication.

[125] Carolyn P. Rosé and Alon Lavie. LCFLEX: An efficient robust left-corner parser, 1999.

[126] Kenneth H. Rosen. *Discrete Mathematics and its Applications*. McGraw Hill, fourth edition, 1999.

[127] D. J. Rosenkrantz and P. M. Lewis. Deterministic left corner parsing. In *IEEE Conference Record of the 11th Annual Symposium on Switching and Automata Theory*, pages 139–152. IEEE, 1970.

[128] Stewart Russell and Peter Norvig. *Artificial Intelligence: a modern approach*. Prentice-Hall, New Jersey, 1995.

[129] Geoffrey Sampson. MT: A Nonconformist's View of the State of the Art. In Margaret King, editor, *Machine Translation Today*, pages 91–110. Edinburgh University Press, Edinburgh, 1987.

[130] The Santayana Edition. available at: `http://www.iupui.edu/~santedit`.

[131] H. Sawaf, K. Schütz, and H. Ney. On the use of grammar based language models for statistical machine translation. In *Proc. 6th Intl. Wkshp on Parsing Technologies*, pages 231–241, Trento, Italy, February 2000. ACL/SIGPARSE.

[132] Robert Sedgewick. *Algorithms in C*. Addison Wesley, 1990.

[133] Peter Sells. *Lectures on Contemporary Syntactic Theories*. CSLI Lecture Notes. CSLI, Stanford, 1985.

[134] Patrick Shann. Experiments with GLR and Chart Parsing. In Masaru Tomita, editor, *Generalized LR Parsing*, chapter 2, pages 17–34. Kluwer, Dordrecht, 1991.

[135] S. M. Shieber, H. Uszkoreit, F. C. N. Pereira, J. J. Robinson, and M. Tyson. The Formalism and Implementation of PATR-II. Technical report, SRI International, AI Center, SRI International, Menlo Park, CA, 1983.

[136] Stuart M. Shieber. Using restriction to extend parsing algorithms for complex-feature-based formalisms. In *Proc. 23rd Ann. Mtg. Assn. Comp. Ling.*, pages 145–152, Chicago, 1985. ACL.

[137] Stuart M. Shieber. *An Introduction to Unification-based Approaches to Grammar*. CSLI Lecture Notes. CSLI, Stanford, 1986.

[138] Daniel Sleator and Davy Temperley. Parsing english with a link grammar. Technical report, Carnegie Mellon University, Computer Science, CMU, Pittsburgh PA, 1991. CMU-cs-91-196.

[139] Daniel Sleator and Davy Temperley. Parsing english with a link grammar. In *Proc. 3rd Intl. Wkshp on Parsing Technologies*. ACL/SIGPARSE, August 1993.

[140] J. Slocum. METAL: The LRC Machine Translation System. Technical report, University of Texas, Linguistic Research Center, 1984. Working Paper LRC-84-2.

[141] Standard Performance Evaluation Corporation. SPEC CPU2000. `http://www.spec.org/cpu2000/`.

[142] Standard Performance Evaluation Corporation. SPEC CPU95 Benchmarks. `http://www.spec.org/osg/cpu95/`.

[143] Guy L. Steele Jr. *Common LISP the Language*. Digital Press, second edition, 1990.

[144] Thomas A. Sudkamp. *Languages and Machines*. Addison Wesley, second edition, 1997.

[145] Dean F. Sutherland, 1999. personal communication.

[146] Dean F. Sutherland, 2001. personal communication.

[147] Daisetz Teitaro Suzuki. *The Manual of Zen Buddhism*. The Eastern Buddhist Society, 1935. available at: `http://www.sacred-texts.com/bud/mzb/`.

[148] Dr. Seuss (Theodor Seuss Geisel). *The Lorax*. Random House, 1971.

[149] M. Thiel. Weighted parsing. In Leonard Bolc, editor, *Natural Language Parsing Systems*, pages 137–158. Springer-Verlag, New York, 1987.

[150] Hideto Tomabechi. Quasi-destructive graph unification. In *Proc. 29th Ann. Mtg. Assn. Comp. Ling.*, pages 315–322, Berkeley, 1991. ACL.

[151] Masaru Tomita. *Efficient Parsing for Natural Language*. Kluwer, Boston, 1986.

[152] Masaru Tomita. An efficient augmented-context-free parsing algorithm. *Computational Linguistics*, 13(1–2):31–46, Jan–Jun 1987.

[153] Masaru Tomita and Kevin Knight. Pseudo-unification and full-unification. Technical report, Carnegie Mellon University, Lang. Tech. Inst., CMU, Pittsburgh PA, 1988. CMU-CMT-88-MEMO.

[154] Masaru. Tomita and Eric H. Nyberg. Generation kit and transformation kit, version 3.2 user's manual. Technical report, Carnegie Mellon University, Lang. Tech. Inst., CMU, Pittsburgh PA, 1988. CMU-CMT-88-MEMO.

[155] Tomita, M., *et al*. The generalized lr parser/compiler/interpreter, rt version 8-4. unpublished source code, 1988.

[156] Arturo Trujillo. Computing FIRST and FOLLOW Functions for Feature-Theoretic Grammars. In *Proc. 15th Intl. Conf. on Computational Linguistics (COLING-94)*, pages 875–880, Kyoto, August 1994. ACL.

[157] Pascal Van Hentenryck. *Constraint satisfaction in logic programming*. MIT Press, Cambridge, MA, 1989.

[158] J. H. Van Lint and R. M. Wilson. *A Course in Combinatorics*. Cambridge University Press, Cambridge, 1992.

[159] Greg Whittemore and Kathleen Ferrara. Empirical study of predictive powers of simple attachment schemes for post-modifier prepositional phrases. In *Proc. 28th Ann. Mtg. Assn. Comp. Ling.*, pages 23–30, Pittsburgh, June 1990. ACL.

[160] M. Wirén. A comparison of rule-invocation strategies in context-free chart parsing. In *Proc. 3rd Conf. European Assn. Comp. Ling.*, pages 226–233, Copenhagen, Denmark, 1987. ACL.

[161] Theodore Ming-Tao Wong, 2001. personal communication.

[162] David A. Wroblewski. Nondestructive graph unification. In *Proc. 6th Nat. Conf. on AI (AAAI87)*, volume 2, pages 582–589, Seattle, Washington, July 1987. AAAI.

[163] D. H. Younger. Recognition and parsing of context-free languages in time n^3. *Information and Control*, 10(2):189–208, February 1967.