TR-I-0124

Efficient Disjunctive Unification in a Bottom-Up Shift-Reduce Parser

David Carter

1989 . 11

ABSTRACT

This report describes two novel techniques which, when applied together, in practice significantly reduce the time required for unifying disjunctive feature structures. The first is a safe but fast method for discarding irrelevant disjunctions from newly-created structures. The second reduces the time behaviour of checking the consistency of a structure from exponential to polynomial in the number of disjunctions, except in cases that, it will be argued, should be very unusual in practical systems.

The techniques are implemented in Propane, an experimental Japanese analyser that uses the large, existing disjunctive Japanese and lexicon created for the Nadine system. Propane is a shift-reduce parser whose behaviour is guided by the preference in Japanese for left-branching structures.

The effectiveness of both the parsing and unification strategies is assessed. The results suggest that the parsing algorithm combines the expected efficiency with a promising degree of accuracy, while the time required for unification is much reduced from that of exponential algorithms.

Contents

1 Introduction

This report describes an experimental Japanese analyser called Propane, for **Prolog Parser** using the **Nadine** Grammar. Nadine (Kogure, 1989) is the analysis and translation component of SL-TRANS, the spoken language translation system under development at ATR Interpreting Telephony Research Laboratories. Propane was originally motivated by a desire to investigate the usefulness of a class of bottom-up shift-reduce strategies for the efficient parsing of Japanese sentences, and the implemented system includes such a parser. However, by far the larger part of the system represents an attempt to solve the problem of efficiently unifying the Nadine grammar's disjunctive feature structures for the purpose of building sentence constituents in a bottom-up fashion. Such a solution was a prerequisite for any investigation of parsing strategies using the Nadine grammar.

The general problem of unifying two disjunctive feature structures is non-polynomial in the number of disjunctions (Kasper, 1987). That is, barring revolutionary developments in the theory of algorithms, the problem is NP-complete, and the time taken to perform such a unification can, in general, at best be an exponentially increasing function of the number of disjunctions.

However, in writing large grammars of natural languages, it is often convenient to be able to specify constraints in terms of disjunctions. This seems especially to be the case for Japanese, because of its relatively free word order. In experimental or research systems, therefore, the trade-off between ease of rule-writing and speed of parsing may come down on the side of a certain amount of disjunction. This indeed is the case in Nadine, where considerable effort has been expended in creating a large grammar and lexicon of Japanese that both make widespread use of disjunctive structures.

It is therefore important to develop unification algorithms that can unify disjunctive feature structures in a reasonable time, despite the inherent NP-completeness of the task. By considering each term in the equation for unification time, $T(d) = \alpha e^{\beta d}$, several complementary types of approach can be identified.

Firstly, one can decrease α, the constant factor in the time equation, so that a very large number d of disjunctions will be required to make T unacceptably large. A wide range of strategies fall into this category, including the use of up-to-date hardware, efficiently implemented programming languages, and any kind of programming technique that speeds up unification without being directly relevant to treating disjunction. For example, Kogure (1989) discusses ways of eliminating unnecessary structure copying and of looking for unification failures as early as possible.

Secondly, one can decrease β so that T grows comparatively slowly with d. Approaches in this category will typically be treatments of the general problem of disjunctive feature structure unification, for example Kasper (1987).

Thirdly, one can try to keep d as low as possible. One way to do this is to discourage grammar writers from using disjunctions; but as we have already observed, this will probably make it more difficult to formalize the facts of the language. Another way to minimize d is to design the parsing algorithm so that, when constituents are created, any parts of them that are not necessary for further processing are discarded. If this is done, unacceptably large values of d may never be encountered by the unifier.

A fourth approach involves the controlled use of techniques that, while they do not always succeed, have the advantage of not being NP-complete. The NP-complete nature of the problem does not preclude the possibility that an algorithm can be defined that, "almost all the time", can unify disjunctive structures in better than exponential time. For a particular grammar and parsing algorithm, there may be a technique for which "almost all the time" means so often that the exceptions can effectively be ignored. The time equation can then be rewritten as $T(d) = \lambda\alpha e^{\beta d} + (1-\lambda)f(d)$, where λ is very close to zero and $f(d)$ will typically be a polynomial.

Since no natural language parser will ever be able to cope with anything close to 100% of the inputs it is given in real use, one can imagine attaching a timeout to a parser so that, if a sentence takes more than a certain time to process, it is deemed to be outside coverage. In that case, the small minority of sentences that require exponential unification time will hardly be noticeable beside all the others that are genuinely outside the coverage of the grammar. It might even be the case that, for a particular grammar and parser, exceptions could never occur, although, of course, the grammar formalism would allow rules leading to exceptions to be defined.

The unification method used in Propane is a combination of strategies of all four of the types outlined above; the four types provide the structure for most of this report. Section 2 describes the Propane parser, which was the original focus of the work. The description provides a context for the discussion of the four unification strategies, to each of which a section of this report is devoted. These four sections can be summarized as follows.

Firstly and most straightforwardly, section 3 discusses the attempt made to reduce α by writing the system in Prolog, rather than Lisp, because of its efficient built-in unification capability. Secondly, section 4 explains how β was kept to a minimum by adopting a principle of what will be called "minimal disturbance" in designing the representation for feature structures. Thirdly, d was kept low by pruning all disjunctions that could be shown not to be relevant to newly-created constituents; the need to carry out such pruning also influenced the design of the representation, as discussed in section 5. Fourthly, section 6 concerns the way the disjunctive unifier seeks to avoid NP-complete operations whenever possible. Practical results suggest that this avoidance is quite successful: no evidence of exponential behaviour has been detected with the mid-November 1989 version of the Nadine grammar. Sentence parsing times are reasonable, and more importantly, the time taken for each unification does not seem to grow unacceptably with the sizes of constituents or with the overall size of the parse tree being constructed. Details of time measurement are given in section 7.

2 The Parser

2.1 The Basic Mechanism

It has often been observed that Japanese is a head-final, strongly left-branching language. This means that modifiers always attach to a head on their right, and that there is a preference for attachment to the nearest such head that obeys the constraints that syntax, semantics and pragmatics place on possible combinations. This preference is so strong as to suggest a parsing algorithm that first constructs analyses that obey it, backtracking and producing analyses with different bracketings only if the initial analysis or analyses are judged unacceptable by some outside process.

A promising technique for implementing the left-branching preference is that of shift-reduce parsing. Propane is based on this technique. In a shift-reduce parser, a stack of partial analyses is maintained at all times, together with a pointer into the input. At each stage, the decision can be made either to *reduce* the top few items on the stack by matching them with the right hand side of some grammar rule, or to *shift* an item from the input and push it onto the stack.

In general, a shift-reduce parser uses a table of parse states and possible actions that determine, at each stage, whether a shift or a reduction is appropriate, and in the latter case, what grammar rule should be used. However, when Japanese is formalized using a grammar in which every rule has exactly two right-hand-side elements – for example, the Nadine grammar – the left-branching preference corresponds to a strategy of reducing the top two categories on the stack whenever there is a grammar rule that allows them to be reduced, and shifting only when this cannot be done. No table is therefore required. Nadine's grammar rules include syntactic, semantic and pragmatic information, so that Propane's decision to reduce or not depends on the acceptability of the result at all three of these linguistic levels. Such a test takes advantage of the maximum amount of available information, and applies it in a fairly straightforward and efficient way.

Alternative lexical entries for words, and alternative grammar rules that can apply to the same pair of daughter categories, mean that each position on the parser's stack is in fact occupied not by a single category but by a list of categories (each of which, of course, contains a disjunctive structure that may have many realizations). The lengths of these lists do not grow significantly as parsing progresses, because just as the lexicon and the grammar can introduce alternatives, so the application of grammar rules can remove them. The attempt to reduce each of m possible head daughters with each of n possible non-head daughters typically results in far fewer than $m * n$ mother structures, because not every rule application succeeds.

Many of the sentences in the corpus of simulated conference office dialogues used in much of ATR's work can be analysed correctly and quite rapidly using this technique. For example, the bracketing corresponding to Propane's analysis of the following sentence, [1] meaning "When (or if) the representative is decided, I will let you know", exactly follows the left-branching preference:

[(<{[(dairinin ga) (kima ri)] mashi} tara> [oshirase (ita shi)]) masu]

The stages of processing are as follows:

1 *shift* dairinin ("representative")
2 *shift* ga (subject marker)
3 *reduce* dairinin *and* ga
4 *shift* kima (root of "kimar-", "decide")
5 *try and fail to reduce* (dairinin+ga) *and* kimari
6 *shift* ri (continuation of verb)
7 *reduce* kima *and* ri
8 *reduce* (dairinin+ga) *and* (kima+ri)
9 *shift* mashi (politeness marker)
10 *reduce* (dairinin...kimari) *and* mashi
11 *shift* tara (inflection meaning "if" or "when")

[1] In the text of this report, Japanese sentences will be written in Roman script using the Hepburn romanization system. However, the actual system input consists of the usual mixture of kanji (Chinese symbols with semantic content) and kana (syllabic symbols), with no spaces to indicate word boundaries.

12 *reduce* (dairinin...mashi) *and* tara
13 *shift* oshirase ("informing")
14 *try and fail to reduce* (dairinin...tara) *and* oshirase
15 *shift* ita (root of "itas-", humble verb for "do")
16 *try and fail to reduce* oshirase *and* ita
17 *shift* shi (continuation of verb)
18 *reduce* ita *and* shi
19 *reduce* oshirase *and* (ita+shi)
20 *reduce* (dairinin...tara) *and* (oshirase+(ita+shi))
21 *shift* masu (politeness marker)
22 *reduce* (dairinin...shi) *and* masu

At each stage, whenever a grammar rule allows a reduction to take place, it is performed.

2.2 Intelligent Backtracking

Of course, there are many cases where the correct analysis of a sentence does not entirely follow the left-branching preference. For example, using the Nadine grammar, the correct analysis of the sentence

(1) dareka ga watashi no kawari ni sanka suru koto wa dekimasu ka

("Can someone participate instead of me?") involves reducing "dareka ga" ("someone", subject) not onto "sanka suru" ("participate") but onto "dekimasu" ("can"), even though the former is allowed by the grammar.

In such cases, the algorithm outlined above will reach a state where all the input has been shifted onto the stack, and either the single item on the stack contains no acceptable sentence interpretations, or there are several items on the stack and the top two cannot be reduced. One possible behaviour for Propane on encountering this situation would simply be to backtrack, by Prolog goal failure, and undo the most recent reduction in favour of a shift. While such an algorithm is complete – it will eventually produce every interpretation allowed by the grammar – it is far from efficient in cases where backtracking is necessary. Moreover, if backtracking occurs, the first full analysis to be found will not necessary be the one with the fewest (or least serious) violations of the left-branching preference; it will merely be the one whose first violation occurs furthest to the right.

Two kinds of improvement to the algorithm are therefore required. The first is the standard and conceptually straightforward technique of introducing a well-formed substring table so that the same portion of input is never analysed twice in the same way. The second, and more interesting, modification is to introduce some form of intelligence in backtracking, so that the reduction that is undone is not necessarily the most recent one but is, rather, the one that seems in some way to be the most promising candidate for replacement. For example, in Japanese, if a noun phrase is immediately followed by a postposition, the latter virtually always governs the former; but if a postpositional phrase is followed by a verb form, the preference is rather less strong, even when the grammar allows the reduction. Therefore, when the first attempt to parse (1) fails, the system should consider undoing the reduction of "dareka ga" onto "sanka suru", but should be much more reluctant to alter that of "dareka" onto "ga".

In deciding what reductions to undo, the system should consider not only the reductions themselves, but also, if possible, the places where reductions narrowly failed to be made. For example, the reduction of "dareka ga ... sanka suru" onto "koto" fails only narrowly; there is a grammar rule that allows verbs to be reduced onto nouns, but the application of that rule in this instance fails unification. The occurrence of a narrow failure may, as in this case, be due to the dependent (non-head daughter) item not having enough free slots, which in turn is due to the erroneous reduction of "dareka ga" onto "kuru". Thus, in general, a reduction that is closely followed by a narrow failure to reduce should be regarded as a candidate for undoing.

There may even, with longer sentences, be occasions where an attempt to parse according to the reduce-over-shift preference should be abandoned, or at least suspended, even before the end of the sentence is reached. If the stack becomes more than a few items deep, it is likely that something has gone wrong; successful parses seem in practice seldom to involve the creation of stacks more than three or at most four items deep.

The implemented Propane parser does not, because of a problem with Symbolics Prolog, perform any backtracking at the level of choosing between shifts and reductions. Initially, the parser was implemented essentially in the following way:

```
parse(Stack,Result) :-
  parsing_is_successful(Stack,Result),!.
parse(Stack,Result) :-
  shift_or_reduce(Stack,NewStack),
  parse(NewStack,Result).
```

However, an attempt to parse a long sentence by consecutive goal satisfaction always led eventually to a stack overflow, which even the insertion of a cut after the call to `shift_or_reduce` seemed not to prevent. For this reason, and because the assertion of partial results in the database will in any case eventually be necessary, the style of the implementation was altered to the following:

```
parse(Result) :-
  retrieve_stack_from_database(Stack),
  parse_with_stack(Stack,Result).

parse_with_stack(Stack,Result) :-
  parsing_is_successful(Stack,Result),!.
parse_with_stack(Stack,Result) :-
  shift_or_reduce(Stack,NewStack),!,
  assert_stack_in_database(Newstack),
  fail.
```

This means that backtracking can not easily be implemented in the straightforward style; nevertheless, as we observed above, such "blind" backtracking is in any case undesirable.

2.3 Lattices of Word Candidates

One problem that the above discussion has glossed over is that sometimes, because of the absence of word separation in Japanese text, it will be possible to shift items of different lengths from the input. At stage 11 in our first example, the system is in fact able to shift "ta", the past tense inflection, as well as, or instead of, "tara", the "if/when" inflection.

If the algorithm is not more carefully defined, the concept of a shift operation will become blurred, because different edges at the top of the stack will extend to different points.

Two approaches are possible here. The simpler one, which is adopted in Propane, is to allow such differences to persist as long as possible. Thus in fact at stage 11, both "ta" and "tara" are shifted, and both are successfully reduced with the second (dependent) item in the stack. Before the next shift, however, Propane "prunes" the edges that constitute the top of the stack, removing all but the longest. This corresponds to the assumption that there is a preference for longer strings of characters to correspond to lexical items where possible, but that this preference should be overturned when a shorter string, but not a longer one, allows a reduction, with what precedes it. Propane's pruning operation in fact allows backtracking, so that if a parse fails, the system may investigate the effects of keeping the shorter string at the expense of the longer one.

The second approach is to allow edges of different lengths to continue to exist, and to shift at all points that represent the right-hand extremes of such edges. An attempted reduction will then, of course, have to check that the two edges in question share the appropriate end points.

While the first approach seems to be adequate for parsing sentences in the text corpus, it is quite possible that, were the system to be applied to lattices of word candidates resulting from speech recognition, the second strategy would be superior. Only experimentation would decide the issue.

2.4 Modifications to the Basic Preference

As a first step towards a more sophisticated control strategy, one exception was introduced to the basic immediate preference for reductions over shifts. Sometimes, an examination of the parts of speech in the grammar and those of the constituents in the stack shows the following situation: a reduction is possible, but if it is performed, the next shift cannot itself be followed by a reduction, whereas if a shift is performed next, two reductions may be possible. That is, there two alternatives: reduce now and then be forced to shift twice, or shift now and, unless unification failure prevents it, be able to reduce twice. In such situations, the parser chooses the second option. This often allows sentences to be parsed which would not otherwise be, and does not seem to prevent the parsing of any sentences. The most common cases which are made to work by this exception involve the word "desu" ("am/is/are"), represented in the lexicon by separate entries for "de" and "su". For example, in "nan desu ka" ("what is (it)?"), it would be possible to reduce "nan" ("what") and "de" ("with"), but the result could not then be reduced with "su". However, if "su" is shifted, it can be reduced with "de" and the result reduced with "nan".

An analogous exception might also be appropriate for the preference of longer edges over shorter before a shift takes place. Occasionally, keeping the shorter edge would, at some not-too-distant point in the future, allow more reductions and hence a shorter stack. For example (and this appears to be the only example for the part of the corpus examined), sentences involving the constructions "...no desu" and "...no deshō" (whose semantic values correspond to nothing very specific in English) currently fail to parse, because "no de" (which can mean "because", or more literally "because of [de] the fact that [no]") has its own lexical entry. Thus an attempt to parse "mōshikomitai no desu" ("(I) want to apply") results in "mōshikomitai" being reduced with both "no" and "no de". Before the next shift, the edge for "mōshikomitai no" is incorrectly pruned. It would be preferable to look ahead and see that “mōshikomitai no de” cannot reduce with “su”, whereas “mōshikomitai no” will allow a reduction with “desu” (which itself will be parsed a constituent because of the heuristic described above). However, because of its greater complexity and uncertain reliability, this heuristic is not currently implemented.

The issue of how such alterations to the basic control strategy should be combined with intelligent backtracking is a complex one. Clearly, it is better, if possible, to avoid an incorrect path rather than backtrack out of it; but equally, it is highly unlikely that heuristics of the kind discussed here will always obviate the need for backtracking. Further insight on the issue will depend on further experience with the way that the various modifications reflect, or fail to reflect, the realities of the language.

2.5 Practical Results

Propane was applied to the 100 sentences [2] in dialogues one to five of the conference office corpus. The breakdown of results was as follows.

Dialogue	Sentences	Parsed	(a)	(b)	(c)
d1	20	17	1	2	0
d2	21	14	3	4	0
d3	16	11	2	3	0
d4	21	18	1	1	1
d5	22	15	3	2	2
Totals	100	75	10	12	3

Thus 75% of the sentences received one or more analyses. No attempt was made to check thoroughly the validity of these because of the present author’s limited familiarity with Japanese and the Nadine grammar. However, none of those inspected seemed to be obviously wrong.

The 25 failures, listed in appendix A, have been divided into three categories. The ten sentences in category (a) are those for which a reduction was incorrectly performed when a shift was required. In eight of these cases, it seemed possible that the problem could be avoided by modifying the grammar to constrain the “sentence levels” (Gunji, 1988) at which different relationships can be established. Thus, for example, in the sentence

Annaisho ni mo ka i te i masu ga

(segmented in the way corresponding to the Nadine lexicon, and meaning “(I)’m writing to the enquiry office too (but...)”), the phrase “annaisho ni mo” (“to the enquiry office too”) was reduced onto the verb component “kai” (“write”), which prevented the subsequent reduction of that phrase onto “te” (“-ing”). In the other two cases, intelligent backtracking seemed unavoidable.

Category (b) consists of the 12 sentences where, before a shift, a shorter edge was incorrectly discarded in favour of a longer one. As already observed, in every case this was due to the phrase “no de” being defined as a single lexical item, which caused problems with the phrases “no desu” and “no deshō”. It appears that these cases could be cured either by implementing the exception described in 2.2 to the “prefer long edges” heuristic, or by altering the treatment of “no de” in the lexicon.

[2]Excluding one that contained a lexical item missing an entry.

Category (c) consists of three sentences that failed for other reasons. In two of the cases, the main Nadine parser also failed, so that the problem is to do with the grammar or lexicon and not anything intrinsic to Propane. The other case is unexplained.

Thus in summary, it seems that, with the above-mentioned modification to the code, between 90% and 98% of sentences in the corpus could expect to get a parse, depending on the treatment of modifier levels in the grammar. Further work would be needed to establish the correctness and preferredness of the readings Propane finds. It would also be worthwhile to investigate the usefulness of this whole technique in parsing lattices of word candidates output by a speech recognizer.

3 Prolog and the Nadine Grammar

The programming language chosen for the work described here was Prolog running on a Symbolics workstation. The choice of a Symbolics was dictated by the limited availability of workstations running Prolog and able to handle the same kanji code as that used in Nadine. Symbolics Prolog has many quantitative limitations built into it that suggest it was never intended for the development of large-scale, serious programs. Although it proved possible to work around most of these limitations, doing so proved quite time-consuming.

3.1 Prolog Unification

Prolog was chosen partly because of the author's familiarity with it, and partly in order to allow some kind of comparison with the existing Lisp based Nadine unifier and parser. It may well be that a Prolog based system will be quicker and easier to write, because of the efficiency and ease of use of its built-in unifier, even though the unification required for Nadine goes beyond Prolog unification in the two following ways. Firstly, there is the obvious disjunction problem, discussed in the other sections of this report. Secondly, there is the fact that Nadine data structures are often circular. If circular structures are handed directly to a Prolog unifier, an infinite loop will result, or at best the unification will fail.

This meant that a special non-disjunctive unifier had to be written which, while using the underlying Prolog unifier to keep track of variable bindings, explicitly maintained a list of pairs of structures that had been provisionally associated with one another pending the successful unification of their components. If one of these structures was encountered as one of its own subparts, unification would continue after substitution of that structure by the one with which it had been paired. This, together with checks for token identity at each stage, ensured that circular structures could be unified correctly and quite fast (although not, of course, as fast as non-circular ones).

It is worth noting that a token identity check is not possible within the standard Prolog language; Propane therefore makes an external call to Lisp to perform it. A facility for such external calls, to Lisp or another appropriate language, would be needed for any port of Propane to a more robust Prolog.

One promising technique for making the most of the speed of Prolog unification is that of transforming or compiling feature structures specified as name-value pairs into structures in which each feature has its own predetermined position. This is done, for example, in the SRI Core Language Engine (CLE; Alshawi et al, 1988). However, the technique relies on the set of possible features for a particular node being specified, or at least being easily derivable from the grammar in question. This is the case in the CLE, but not in Nadine; the set of features seems to be open-ended, and there is no explicit definition of what features can co-occur in legal structures. Typically, Nadine structures are "sparse" in the sense that, at a given level, only a few of all the features that exist in the grammar will actually occur. This means that a positional representation would always contain large numbers of redundant variables.

Thus a feature list is represented in this system as a Prolog list that ends with a variable. When one feature structure is unified with another that specifies values for features not mentioned in the first, the tail variable of the first is (partly) instantiated to include these extra features. This process is quite efficient because it does not involve any structure copying.

The net result of all this is that the operation of unifying two non-disjunctive feature structures, although not as fast as straight Prolog unification, is fast enough that it does not seem to represent one of the major uses of computation time in the system.

Nevertheless, it should be stressed that there is nothing to prevent the work described here being reimplemented in another symbol processing language. Some loss of speed might result, but this might be more than offset by factors of compatibility with hardware or with other software.

3.2 Converting the Grammar

On an implementation level, the difference between Prolog and Lisp syntax means that some preprocessing of the grammar is needed to put it into Prolog format. Because of time constraints, this conversion is at present done by Unix shell scripts followed by a small amount of hand editing. However, it would not be difficult to write a fully accurate preprocessor in either Prolog or Lisp. The changes are mainly a matter of conforming to Prolog's syntax for brackets, and of quoting objects that would not otherwise be legal Prolog atom names. Nadine variables are converted directly into Prolog variables.

After preprocessing, each template, grammar rule and lexical entry is converted into a Prolog clause that is asserted in the database. [3] A rule or lexical entry can then be invoked simply by calling the relevant predicate, which instantiates its arguments to the appropriate disjunctive feature structure and other necessary structures (for details of which see section 5). Although the construction of Prolog clauses in this way is non-trivial, the clauses are made to correspond as directly as possible to the format of the originals from which they derive. This is a deliberate policy, the reasons for which are given in the next section.

[3]In fact, because of a very low Symbolics Prolog limit on the number of variables and predicate calls allowed in a single clause, several linked clauses must often be asserted.

4 Grammar Conversion by Minimal Disturbance

4.1 The Argument for Minimal Disturbance

One of the keys to processing and unifying disjunctive feature structures efficiently is, I believe, to preserve as far as possible their logical (and/or) structuring. This is what is meant by "minimal disturbance". Any attempt to transform one structuring into another, for example conjunctive or disjunctive normal form, can be computationally inefficient, and may result in a much larger data object. Also, the structuring chosen by the grammar writer may have been selected because it allows the efficient application of grammatical and other constraints that can, in the right circumstances, rule out large parts of the and/or tree. If this structuring is destroyed, the eventual result may be logically equivalent, but it may only be achieved after much computation, and it may slow down later unifications.

To take an obvious example, suppose that the grammar originally contains a disjunctive structure of the form

```
(:or (:and c1 r1) (:and c2 r2))
```

where `c1` and `c2` are fairly trivial constraints and `r1` and `r2` are larger structures. If this is unified with a structure `s1` that matches `c1` but not `c2`, and if the logical structure is followed from left to right, then no attempt will be made to unify `s1` with `r2`. However, if the above structure is transformed into the logically equivalent

```
(:and (:or c2 (:and c1 r1)) (:or r2 (:and c1 r1))
```

or into a number of other such structures, `s1` will be unified with `r2` even in cases where `c2` does not match `r2`. The point is not that the second and/or structure is more complex than the first, but that the fact that `c1` is a "condition" on the "result" `r1` is not stated explicitly and so risks being lost in any process of normalisation.

Nevertheless, although the and/or structuring of the objects returned by grammar predicates corresponds to that specified by the grammar writer, the control structure of those predicates is a linear (deterministic) sequence of calls to other predicates, each of which is expected to succeed. Each of these calls contributes something to the overall disjunctive structure that will be returned. It is expected that grammar predicates will be called without any constraint having been imposed on the form of this structure. The reason for this is that disjunctive feature structures, once constructed, will in general be copied and unified with several different alternatives, and such copying is quicker and simpler than calling the predicate again. It would be possible to make grammar rule predicates non-deterministic, in order to allow them to be called with partly instantiated results and thereby avoid following impossible control paths. However, this would introduce a far more serious risk of inefficiency. In the case where one predicate called several nondeterministic ones, it could happen that the first result returned by the first such predicate was inconsistent with the partial instantiation of the overall result, but that this inconsistency would not manifest itself until several subsequent predicates had succeeded. Since Prolog backtracking is blind, all these intervening predicates would be called repeatedly until the original, offending one backtracked. The moving of all nondeterminacy from the program into the data allows inconsistency to be handled far more intelligently, as will be described in detail in section 6.

4.2 An Example

The preceding discussion may be clarified by an example. The template defined in the original grammar by

```
(DEFFSTEMP SC-SL-2-1 (%COMP1 %COMP2)
  (<!M !SYNSC !FIRST> == %COMP1)
  (:OR
    ((<!M !SYNSC !REST> == (:LIST %COMP2))
     (<!M !SYNSL>       == !EMPTY-DLIST))
    ((<!M !SYNSC !REST> == !EMPTY-LIST)
     (<!M !SYNSL>       == (:DLIST %COMP2)))) )
```

is turned into the Prolog clause (slightly simplified, and with variables given more informative names):

```
constraint_template(sc_sl_2_1,[Comp1,Comp2],struc(Def,Indef)) :-
  path([mthr,syn,subcat,first],Def,Comp1),
  path([mthr,syn,subcat,rest],Opt1,Screst1),
  create_list(Screst1,[Comp2],[]),
  path([mthr,syn,slash],Opt1,Synsl1),
  constraint_template(empty_dlist,[],struc(Synsl1,[])),
  path([mthr,syn,subcat,rest],Opt2,Screst2),
  constraint_template(empty_list,[],struc(Screst2,[])),
  path([mthr,syn,slash],Opt2,Synsl2),
  path([in],Synsl2,Synsl2in),
  path([out],Synsl2,Synsl2out),
  create_list(Synsl2in,[Comp2|Synsl2out],Synsl2out),
  collapse_ranges([range([],[struc(Opt1,[]),struc(Opt2,[])])],Def,Indef),
  !.
```

This predicate will be called, by grammar rules and lexical entries, with the variables Comp1 and Comp2 possibly instantiated. Its effect will be to instantiate Def and Indef, the definite and indefinite parts of the resulting feature structure. Each of the twelve deterministic calls to other predicates should succeed. The cut at the end of the definition ensures that there will be no attempt made to backtrack; such attempts should, in any case, never succeed.

In constructing the definition, the calls to the path templates M, SYNSC, FIRST, etc., have been expanded out and explicit feature paths put in their place. The predicate path(Path,Fs,Value) instantiates the end of the feature path Path in Fs to Value. create_list corresponds to the :LIST and :DLIST constructions, and itself makes a number of path calls.

The first path call in the definition acts directly on Def, the result variable, because in the original template it is not embedded in any :OR operator. Thereafter, each branch of the :OR causes a different variable, Opt1 or Opt2 to be used in place of Def. These variables, with their instantiations, are assembled into the indefinite part Indef at the end of the call by the run-time predicate collapse_ranges. This process of assembly involves consistency checks between the Def structure and each member of the Indef list, but, in accordance with the policy of minimal disturbance, does not attempt wholesale alteration of the and/or structure. Were the templates empty_list and empty_dlist to return complex disjunctive structures (in fact they do not), the relationships between their parts would not be altered.

The disjunctive structure struc(Def,Indef) resulting from the call can be (disjunctively) unified with other structures that may match either, both or neither of its branches. It is printed by the system (again with variable names altered) as

```
[*S* [mthr [syn [subcat [first Comp1]]]]
     [*R* [*S* [mthr [syn [slash [in X] [out X]]
                          [subcat [rest [first Comp2] [rest []]]]]]]
          [*S* [mthr [syn [slash [in [first Comp2] [rest Y]] [out Y]]
                          [subcat [rest []]]]]]]]
```

where *S* stands for (disjunctive) structure and *R*, for "range" or indefinite component, denotes a list of alternative constraints, exactly one of which must be selected for a given realization. An *S* structure may have any number of ranges; here, the outer one has two, and the inner ones both have none. These inner structures derive from the variables Opt1 and Opt2 respectively in the predicate definition.

4.3 A Further Issue

Through the use of alternative variables for disjuncts, e.g. Opt1 and Opt2 for Def in the example above, no interference should arise between structures corresponding to different disjuncts. However, such interference is in principle possible when a Nadine variable such as Comp2 above (as opposed to variables like Synsl1 and Screst2 created during the conversion process) occurs in more than one disjunct. This could occur if such a variable is further instantiated, rather than just used to build other structures, in one or more of the disjuncts. In the case where a variable occurs only in several different disjuncts and not outside the :OR expression in question, it is possible simply to rename the variable differently for each disjunct to prevent interference. However, when the variable also occurs outside the :OR, renaming alone can lead to incorrect results, and some additional constraint is in principle required to associated the renamed version of the variable with its original. For implementation reasons, this is not quite trivial in the current version of the system, but it seems that cases where a variable occurs both inside and outside a set of disjuncts and is altered in one of them are rare in the Nadine grammar if not nonexistent.

5 Pruning Irrelevant Disjuncts

If parsing is to be efficient, it is important that disjunctions that are irrelevant to a newly-created constituent – that is, disjunctions whose values never affect the realizations of the constituent – are discarded whenever possible. Otherwise, the number of disjunctions in a constituent will be proportional to the number of lexical entries and grammar rules used to construct it, and the time taken to unify two constituents will increase at least as fast as that number and probably rather faster.

With a trivial alteration (changing the M path template to have the value [mthr] instead of the empty list, so that the mother component of a non-disjunctive feature structure can be easily identified) the Nadine grammar would, at first sight, appear to allow bottom-up construction of a mother constituent from a rule and two daughters to be followed by the required pruning. Every indefinite part of a disjunctive structure specifies that the alternatives it contains are to be interpreted as the value at the end of a certain feature path. If, after rule application, disjuncts that refer only to daughter positions and not to mother ones are discarded, the desired effect would seem to have been achieved.

The problem, however, is that the same substructure frequently appears in different parts of a feature structure. When a grammar rule identifies part of the mother structure with part of a daughter one, then any disjunctions involving the latter must be preserved.

Some means must therefore be found of keeping track of what pieces of structure are shared, or in other words, what pairs of feature paths lead to the same values. If this is done, a disjunction that explicitly involves only daughter constituents can safely be discarded if no feature path through the mother leads to it or to any of its components.

Of course, the set of feature paths that share a value depends on what realization (choices of disjuncts) is made for a disjunctive structure. It is not even simply the case that each position contributes its own set of common paths; two structures that are some distance apart in the and/or tree can cause two paths to have the same value by, for example, placing the same variable in two different positions. Thus to decide infallibly whether a given disjunct should or should not be discarded, one would need to cycle through every possible realization of the whole structure, a process that is exponential in the number of disjuncts and therefore unacceptable. This rules out, for our purposes, a representation similar to that of Eisele and Dörre (1988), in which equivalences between different parts of a structure are denoted by explicit pointers from one part to the other. Eisele and Dörre's pointers can occur inside a disjunction, and the values at the positions to which they refer can also be affected by disjunction.

The alternative adopted was, therefore, one that errs on the side of caution, in the sense that it never throws away a disjunct that should be kept, but does sometimes keep a disjunct that should be thrown away. The result of the latter kind of error is not to give incorrect results but merely to encumber the representation with some irrelevant information.

Each disjunctive structure returned by a lexicon or grammar predicate, therefore, is assigned a set of "path groups", which each correspond either to a variable that appears more than once in the original Nadine definition, or to a token-identity equation (with the Nadine "`==`" construct) between two or more items representing different positions in the feature structure. To some extent, a path group is analogous to a set of Eisele and Dörre pointers that all point to the same position. However, the crucial point is that no record is kept of which position in the and/or tree each path comes from. This means two things. Firstly, the point of the whole exercise is that when deciding whether to throw away a disjunction referring to a particular position in a daughter structure, Propane can check the (unique, disjunction-independent) set of path groups, and if no possible equivalence with part of the mother structure is found, the disjunction can safely be pruned. The price we pay for this disjunction-independence is that the pathgroups can specify spurious equivalences. It is possible for two paths to be associated when they arise from two different, incompatible disjuncts, or to remain associated after the disjunct(s) from which they arose have been eliminated through later unification. Thus the occurrence of the variable `COMP2` in the above example in the positions `[mthr,syn,subcat,rest,first]` and `[mthr,syn,slash,in,first]` will cause these paths to be associated, even though they arise from alternative disjuncts and so never actually share the same value in any one realization. However, since path groups are used only for deciding what disjunctions to discard, and not as part of the feature structure representation itself, a spurious path group can only result in some inefficiency, and not in an incorrect result.

This technique is thus a compromise between, on the one hand, carrying out possibly exhaustive computation to achieve a perfect result, and on the other hand not discarding anything at all. It maintains the non-exponential character of the algorithm at the cost of some slight unnecessary processing at a later stage. In practice, the cost involved seems quite acceptable, in that the number of disjuncts in a constituent does not increase greatly with its height in the parse tree.

Kasper (1989) describes an approach to disjunctive unification involving transformation of information to a classification-based (hierarchical) knowledge representation language. He states that "The process of classification also keeps track of dependencies between different objects, eliminating the need for checking consistency between components of a description that have no features in common. In effect, an index is incrementally constructed from features to descriptions that contain them." It might be that this technique could be used not only to reduce consistency checking, but also to decide whether to prune a disjunction. However, in the absence of a more detailed description, it is difficult to judge whether this scheme is (or could be made) equivalent to Propane's, or whether to use it for pruning would require exponential expansion of the kind we are trying to avoid.

Another consequence of keeping irrelevant disjuncts is that if, at the end of the parse, the set of all full realizations of a disjunctive feature structure is exhaustively enumerated, then the same realization may be encountered repeatedly. However, experience suggests that for the current Nadine grammar and corpus, this is not a problem. The average number of realizations (identical or different) per parse, of the 75 sentences parsed from the sample dialogues, was exactly two, and only one sentence received more than six realizations.

The pruning operation in fact resulted in, on average, a 20% decrease in the number of disjunctions in a newly created mother constituent. In more detail, the average, over all reductions performed in processing dialogues d1 to d5, of the number of disjunctions in a rule was 2.81, in a non-head daughter 1.95, and in a head daughter 2.10, giving a total of 6.86. Unification resulted in a decrease to 3.02, while pruning achieved a figure of 2.42. Probably for this reason, the number of disjunctions in a new mother constituent only barely shows a positive correlation to the size, in constituents, of the subtree that it dominates and from which it has been built. [4] The data is shown in figure 1 below. On the other hand, if pruning were not performed, each constituent could be expected to add its quota of irrelevant disjuncts to every other constituent that dominated it. Despite the relatively modest figure of a 20% decrease over one reduction, the cumulative effect of such decreases over a whole parse is probably quite significant.

[4] In detail: the product-moment correlation coefficient for the relationship between subtree size and number of disjunctions, for 406 observations, is 0.0864. The threshold for this coefficient for statistical significance at the 5% level, for the hypothesis that the number of disjunctions increases with subtree size, is about 0.082.

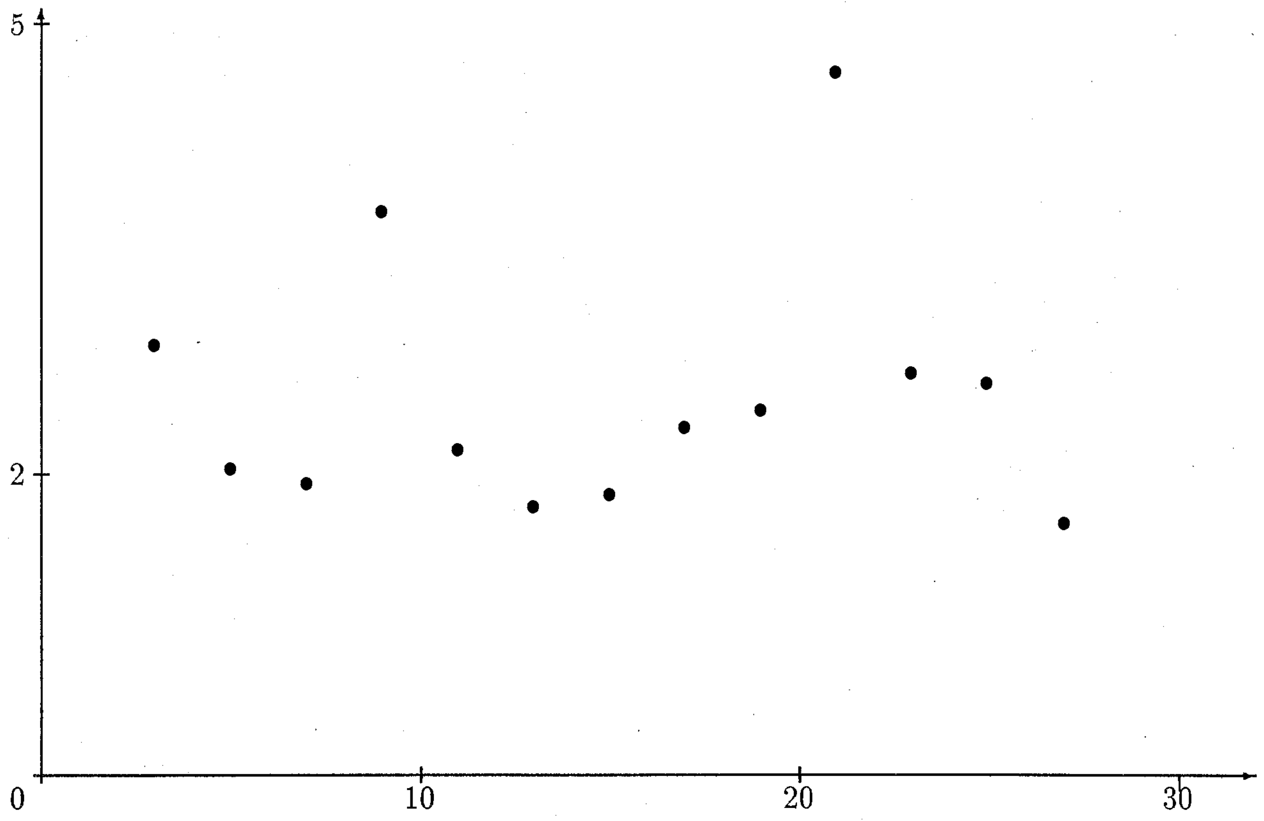

Figure 1: Mean disjunctions per node as a function of size of tree dominated

The "size of tree dominated" is the number of tree nodes dominated by the node in question, including itself. Because all grammar rules specify exactly two daughters, the number of nodes dominated is always an odd number. Data for terminal nodes (tree size one) and for tree sizes with less than three instances are not shown.

6 Pairwise Consistency Checking

When performing a reduction in the shift-reduce algorithm described in section 2, it is important to verify that the disjunctive feature structure constructed does in fact have at least one realization. If it does not, then the reduction should not be performed, and the course of the parse will be different. In any case, a representation that does not distinguish between realizable and unrealizable structures is seriously flawed in its own right.

However, finding a full realization of a disjunctive structure is a potentially exponential process. For example, if a structure has m indefinite parts each of which contains n disjuncts, and if none of the disjuncts of the last indefinite part are consistent with any realization of the rest of the structure, then the situation could arise where each of the n^m possible combinations of disjunct choices is tried and rejected.

There appears to be no general solution to this problem other than potentially exponential backtracking, whether intelligent or otherwise. The technique used in the Nadine parser is that of Kasper (1987), which, for every set of n disjunctions at the same node, involves checking the consistency first of pairs of disjuncts, then of triples, and so on up to n-tuples for full consistency. At each stage, any member of a disjunct that does not take part in any consistent tuple is eliminated. If all the members of a disjunct are eliminated, the mother node of that disjunct is eliminated too; if the root node of the whole feature structure is eliminated, unification fails. This technique has the advantage that the pruning of nodes at stage k will make stage $k+1$ more efficient. Nevertheless, since n can sometimes be quite large, this exhaustive process be time-consuming, and indeed in the limit will take exponential time.

However, it can be surmised that the vast majority of unrealizable feature structures that will be created in the use of a practical natural language grammar will be not only unrealizable, but also "pairwise unrealizable", in the sense that they will fail at the first stage of the above consistency check, for $k = 2$.

The reason we can expect most unrealizable structures also to be pairwise unrealizable is that most commonly, unrealizability will result from the contents of two nodes in the tree being incompatible, through assigning non-unifiable values to the same position in a feature structure. Exceptions can certainly occur, when for example one node assigns the value `v1` at position `p1`, another assigns an incompatible value `v2` at `p2`, and a third identifies the values at `p1` and `p2` with each other but does not otherwise constrain them. However, the hypothesis is that it is fairly unlikely, in a large and/or tree (which is the case where exponentiality would be harmful) that there would be a non-pairwise inconsistency but no pairwise inconsistency.

Following this hypothesis, when the Propane unifier has created a structure, it checks and prunes it first for pairwise consistency, and if this succeeds, risks trying for a single full realization (one choice at each disjunct) straight away. Thus it differs from Kasper's algorithm in two ways: no exhaustive k-wise checks are made for $k > 2$, and when a full check is made, only one success is required, avoiding an exhaustive search through all combinations of nodes. Of course, if the structure is pairwise realizable but not fully realizable, the search for a single success will take exponential time; but, according to the hypothesis, such occurrences, on structures with enough disjuncts for exponential time to be unacceptably long, should be extremely rare.

The effectiveness of this strategy can only be judged by observing its behaviour in practice. To date, no instances have been observed of a check for full realizability taking an inordinately long time after pairwise consistency checking and pruning have succeeded. Thus it can be tentatively concluded that, with the current version of the Nadine grammar and bottom-up parsing, the risk is worth taking: that is, full realizability is virtually always possible, in reasonable time, for a partially realizable structure. Of course, this does not alter the fact that in general, i.e. for an arbitrary input and for an arbitrary grammar written in the Nadine formalism, the unification algorithm, like Kasper's, is exponential in behaviour. In the limit, an exponential term in the formula for the time behaviour of an algorithm will dominate, however small its associated constant factor.

Unlike Nadine's unifier, Propane's strategy has the property that when a structure survives consistency checking, not every member of every disjunct in it can necessarily participate in a full realization; that is, ideally, it should have been pruned. However, this property is only undesirable to the extent that, at the end of the parse, it makes any exhaustive search for full realizations inefficient through excessive backtracking. Again, in practice, this seems not to be a problem; exhaustive full realization is extremely quick compared to parsing itself.

An analysis of Propane's processing of its corpus, the results of which are shown in figure 2 below, reveals quite wide variation in the relationship between the total number of disjunctions in a rule application (in both daughters and the rule) and the time taken to perform the unification. However, although, unsurprisingly, unification time increases with the number of disjunctions, it appears from inspection to be perhaps linear with a small parabolic component, and not exponential. This is, in fact, what an analysis of the algorithm predicts. The linear component derives from the check of each disjunct separately against the definite part, while the parabolic component derives from the pairwise check. The relatively small size of the latter may imply that the majority of disjunct eliminations happen in the first phase, not the second.

7 Observed Parsing Times

The absence of any known exponential process (other than the final phase of unification, which appears never to take very long) in Propane's parsing and unification algorithms gives grounds for expecting that in practice, the time taken to parse a sentence of n lexical items should be polynomial in n. Because of the pruning of irrelevant disjunctions, the value of n should be fairly small, leading to a significant speed advantage. Observed results suggest that such an advantage does exist, but are not sufficiently detailed to allow the verification of the exact time behaviour of the system.

Figure 3 shows the relationship between the number of node dominated by a newly-created mother constituent, and the time taken to create the mother. The relationship would appear from inspection to be just worse than linear. Since, as we have already observed, the number of disjunctions in a constituent does not increase with the number of nodes dominated, the effect is presumably due to a large average size of the definite part of each constituent. This behaviour is to be expected, since compositional semantic interpretation means that semantic information is seldom or never thrown away; thus the size of the definite part may be roughly proportional to the number of nodes dominated. Non-disjunctive unification is a task of order $n\ log\ n$ in the size n of the inputs (Kasper, 1987), and the relationship in figure 3 seems to agree well with this.

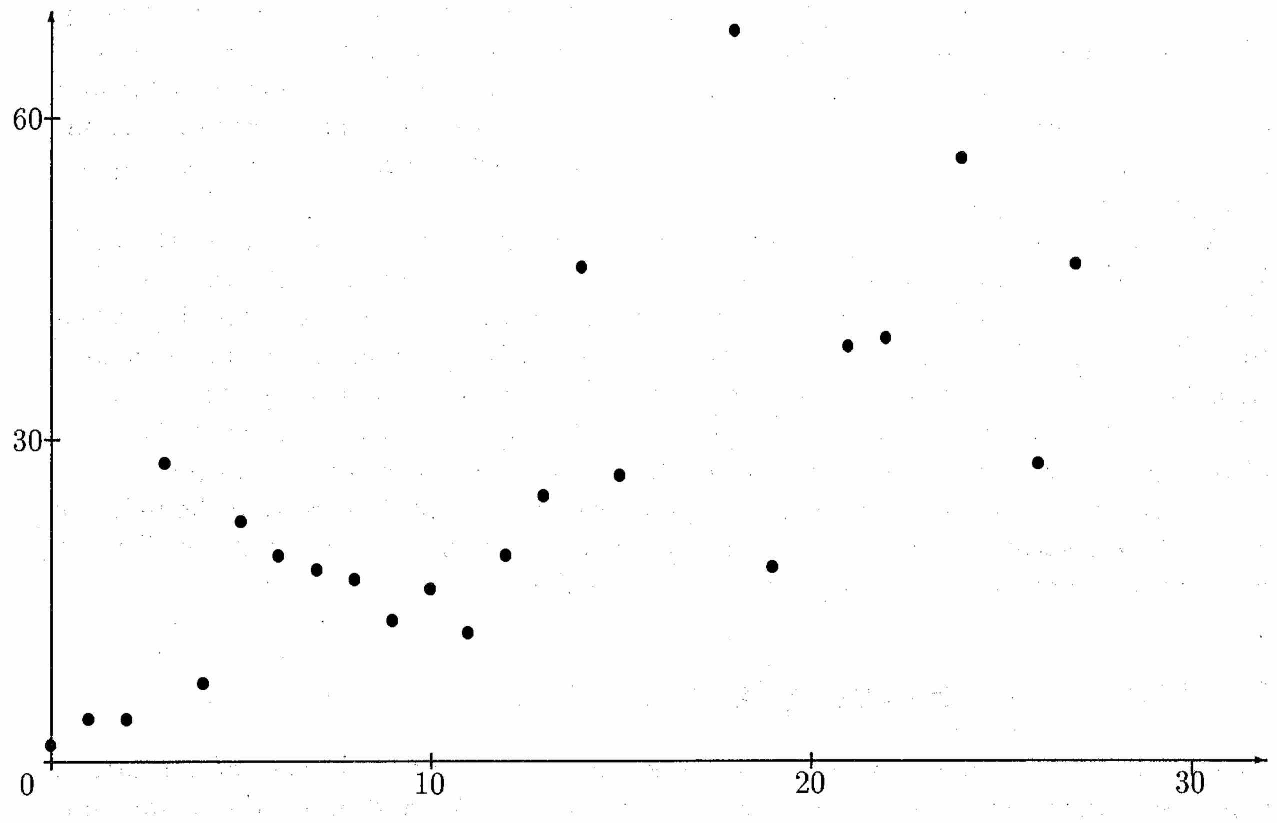

Figure 2: Mean time (seconds) taken for reduction as a function of input disjunctions

Input disjunctions are the total of those in both daughters and the rule. Times include all phases of reduction including final assertion in the Prolog database. Data points with less than three instances are not shown.

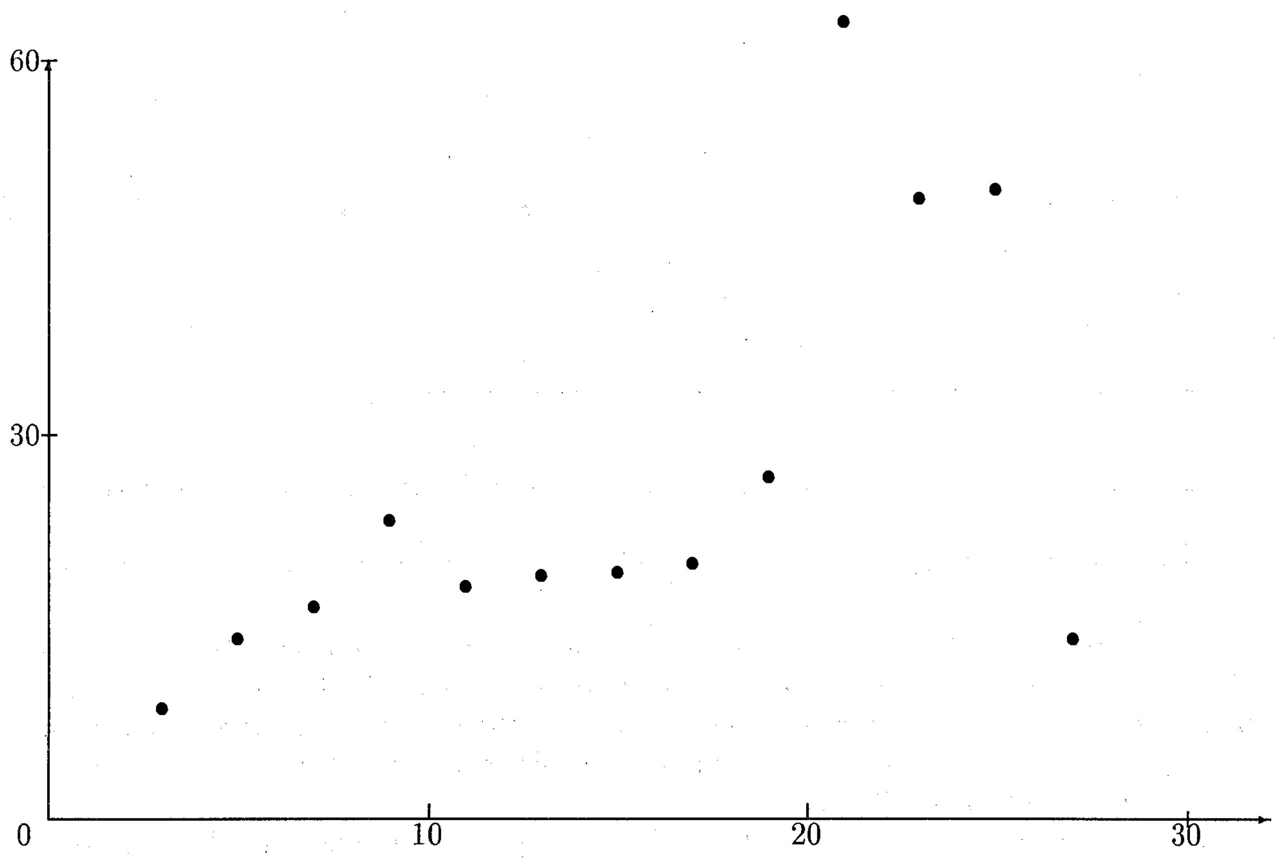

Figure 3: Mean time (secs) taken for reduction as a function of size of tree dominated

"Size of tree dominated" is the number of nodes including the current one, as in figure 1. Again, means of less than three instances are not shown.

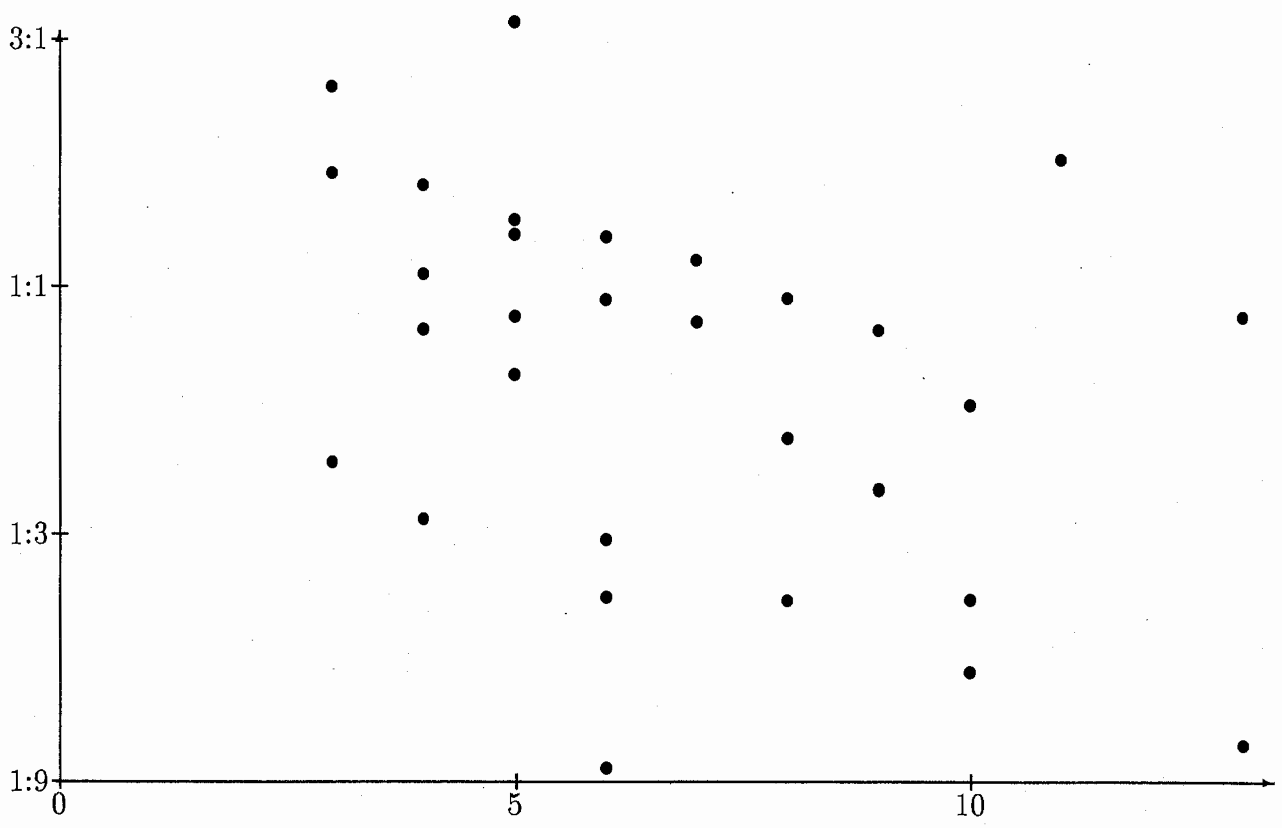

Figure 4: Ratio of parsing times (Propane:Nadine, log scale) as a function of sentence size

Sentences consisting of only one lexical item, and hence only trivial parsing, are not represented.

Furthermore, as the total time taken to parse a sentence increases, the proportion of that time taken in asserting newly-created constituents in the Prolog database also increases. Thus database assertion time grows faster than the time for parsing itself. In an improved version of Propane, or using another Prolog, it might be possible to make constituent storage more efficient.

Figure 4 shows the relationship, for each of the sentences in dialogues d1 to d5 successfully parsed by both Nadine and Propane, both running on a Symbolics, between the elapsed time taken by each to complete the first parse. The value shown is the ratio of the Propane time to the Nadine time, with the scale being logarithmic.

Some caution is needed in interpreting this figure. Firstly, Symbolics elapsed time is notoriously subject to variation according to memory usage and other ephemeral factors. Secondly, the scale on the vertical axis of the graph is in a sense irrelevant, as is, to some extent, the slope of whatever line or curve one might want to draw through the points. This is because many of the factors affecting the time behaviour of each system are different; the most obvious differences are those of programming language and parsing algorithm.

Nevertheless, as sentence length grows, Propane tends to perform progressively faster in a statistically significant way. [5] In particular, the attempt to parse two fairly long

[5]In detail: the product-moment correlation coefficient for the relationship between the number of lexical items in the sentence and the logarithm of the ratio of parsing times, for each of the 31 sentences of length greater than one parsed by both systems, is -0.40. This is easily statistically significant at the

sentences in the corpus with Nadine had to be aborted because of the time it was taking, but both these sentences received a parse, albeit after over ten minutes, from Propane. Had Nadine not been aborted in these cases, two more data points would be present in the lower right part of the graph.

The progressive speed advantage of Propane is not due only to the fact that it goes "straight" to the parses it finds, without exploring any alternative bracketings of the sentence. Nadine is also, through numerical scoring, sensitive to the left branching preference, which guides it to explore, and presumably to find, preferred parses first. In any case, on a logarithmic scale, the increasing effect of such polynomial differences should become less marked as the sentence length increases, whereas the difference seems rather to gather pace, or at least maintain it, even on this scale.

Such vague remarks about the nature of Propane's speed advantage are admittedly unsatisfying and should, given time, be backed up by the acquisition of more data and by further statistical analyses. Nevertheless, there are, as already mentioned, a priori grounds for expecting any parser adopting the unification techniques described here to have an increasingly marked advantage, and the data presented here show a statistically significant trend in that direction.

In the general case, the time behaviour of a bottom-up parser using the techniques described in this paper will depend on several factors. Firstly, of course, if the essential hypothesis about pairwise consistency does not hold for the particular grammar and the sentences in question, exponential time can be expected. If the hypothesis holds, as it appears to for the Nadine grammar, then the behaviour depends on the way in which both the number of disjunctions and the size of the non-disjunctive part of a constituent increase with the number of constituents it dominates. If disjunction pruning means that the average mother constituent contains no more disjunctions than the average daughter (and this only just, if it all, fails to be the case for Propane), then the processing of disjunctions will not alter the order of the parsing algorithm. Otherwise, the number of disjunctions will contain a component (very small in Propane's case) proportional to the number of constituents dominated, which, except for pathological grammars where as sentence length increases, the average number of daughters per mother tends to zero, means that disjunction will at worst contribute a factor of N^2 for sentence length N; for example, a parsing algorithm that would run in time N^3 on atomic categories will run in time N^5. Compositional semantic interpretation will probably mean in the limit that the size of the non-disjunctive part of a constituent will also be proportional to the number of constituents dominated. Unification time here is order $nlogn$ in the sizes n of the input structures, so that, for example, an order N^3 algorithm will become order N^4logN in non-disjunctive unification time (although, of course, this will be swamped by the N^5 term for disjunctions if the latter applies). In practice, the use of the left-branching preference should allow the parsing algorithm (without unification) to operate in rather better than N^3 time.

In conclusion, the numerical data presented in this and earlier sections, taken together, give reasonable grounds for hoping that the techniques described in this report for disjunctive unification will be helpful in increasing the efficiency of a variety of language processing systems that must perform such unification. The shift-reduce technique described here would also seem to be a promising starting point for efficient parsing

5% level, and its negative sign indicates that the correlation is in the direction of Propane performing better for longer sentences.

algorithms for Japanese. Taken together, and assuming, of course, that the essential hypothesis about pairwise consistency continues to hold, allowing us in practice to avoid the consequences of NP completeness, the techniques should allow sentences to be parsed in an acceptably low-order polynomial time, as compared to an exponential time for other approaches involving disjunctive unification.

8 References

Alshawi, H., *et al* (1988) *Interim Report on the Core Language Engine*, Report 005, SRI International Cambridge Research Centre, U.K.

Eisele, A., and Dörre, J. (1988) "Unification of Disjunctive Feature Descriptions", *Proceedings of the 26th Annual Meeting of the Association for Computational Linguistics.*

Gunji, T. (1989) "Syntactic Sketch 88: Japanese". In: *Syntax: an International Handbook of Contemporary Research*, de Gruyter.

Kasper, R. (1987) "A Unification Method for Disjunctive Feature Descriptions", *Proceedings of the 25th Annual Meeting of the Association for Computational Linguistics.*

Kasper, R. (1989) "Unification and Classification: An Experiment in Information-Based Parsing", *Proceedings of the International Workshop on Parsing Technologies*, Carnegie-Mellon University.

Kogure, K. (1989) "Parsing Spoken Sentences based on HPSG", *Proceedings of the International Workshop on Parsing Technologies*, Carnegie-Mellon University.

A Sentences Not Parsed by Propane

[a] Failures due to Propane reducing when it should shift. In each case, A + B + C means A and B were reduced, when C should have been shifted and then B and C reduced. Other parts of the sentence, if any, are shown in parentheses.

Cases that might be prevented by the application of Gunji's treatment of "sentence levels":

それでは + 登録用紙をお送り致し + ます
失礼ですが + お名前とご住所をお願いし + ます
案内書にも + 書い + て (いますが)
来月 + お申込みになり + ます (と四万円です)
参加料には + 予稿集代と歓迎会費が含まれ + て (います)
今回の会議は + 通訳電話に関連する広範な研究分野を含んで + い (ます)
言語学や心理学を専攻する方にも + 参加し + て (頂く予定です)
参加料は + 銀行振り込み + です

Cases that seem to require intelligent backtracking:

ベル研の + ジム + ワイベル (です)
(では) 誰かが + 私の代わりに参加する + こと (はできますか)

[b] Failures due throwing away the shorter edge before a shift instead of the longer one. Note that all these involve ので.

会議に申し込みたい + ので + すが
どのような手続きをすればよろしい + ので + しょうか
電話番号もお聞きしたい + ので + すが
ちょっとお願いがある + ので + すが
参加を取り消したい + ので + すが
会議の参加料について教えて頂きたい + ので + すが
わたしは情報処理学会の会員な + ので + すが
参加料の割引はない + ので + すか
参加料はどのようにお支払いしたらよい + ので + すか
会議に論文を発表したいと思っている + ので + すが
わたしは日本語が全然分からない + ので + すが
発表が日本語で行われる場合英語への同時通訳はある + ので + すか

[3] Other failures.

Sentences on which Nadine also fails:

私は会議に申込みをした者です
登録料を払い戻して頂けますか

An unexplained failure:

持っていません

B How to Load and Use Propane at ATR

From a Symbolics Prolog window, type

```
?- :Set Stack Size Control 700000
?- compile('LM13:>carter>srp>loadsrp.prolog').
?- loadsrp.
```

Answer `P` to any requests for confirmation, and ignore warnings about undefined functions. The Propane code is now loaded. To load the Nadine grammar and lexicon, type:

```
?- define_templates.
?- define_grammar.
?- define_lexicon.
```

Each of these commands takes several minutes to complete. When they have done so, you can enter Propane in various ways. The predicate `parse(String)` can be called with `String` instantiated to a Prolog string containing a sentence (in kanji/kana form) to be parsed. To parse a sentence from a sample dialogue, call `parse_s(Id)` where `Id` should be a sentence identifier such as `d1_10`. To parse a whole dialogue, type `parse_d(Id)`, where `Id` might be `dialogue_1`, and to parse everything in dialogues one to five, type `parse_main_dialogues`. (A transcript of a run of this predicate is in `LM13:>carter>srp>scriptfile1to5.prolog`).

Since the system is an experimental one, a fair amount of diagnostic, timing and other information is displayed as sentences are processed. If a parse is successful, the semantic and pragmatic parts of the final realization(s) are printed. Modification of the code (in `LM13:>carter>srp>*.prolog`) according to the comments associated with it should allow the amount and type of output provided to be altered.

The process of adapting a new version of the Nadine grammar to Prolog form is, for lack of time, not fully automated. Most of the work is done by a Unix shell script, but some hand editing is necessary. The procedure is as follows.

(1) The command `/usr1/carter/bin/lisp2prol` will convert the file that is its first argument to something closely approximating a Prolog-readable form, and output it on the standard output. The command should be called once on each of the files to be converted; a standard script for this is in `/usr1/carter/bin/lisp2prolall`.

(2) Copy the results of the conversion to a Symbolics machine, using the Symbolics command `Copy Kanji File` on each file.

(3) "Debug" the results for Prolog readability. The file `LM13:>carter>srp>testread.prolog` contains a predicate `testread(File)` which will read terms successively until a failure or the end of file. Each term successfully read is asserted as the value of the predicate `lterm`. Thus when `testread` fails, abort the call and type `lterm(X)`. The term after `X` in the file in question is the unreadable one, which should be hand-edited. Typically failures are caused by terms not ending with a period and by commas immediately following opening brackets.

(4) Inspect the code of the predicates `define_templates`, `define_grammar` and `define_lexicon` in `LM13:>carter>srp>convtop.prolog` to see what input files are expected, and either copy the new files into place or create your own versions of those predicates accessing different files.

(5) Call the three "`define_`" predicates as usual. If templates are redefined, the grammar, on which they depend, also becomes undefined automatically. However, lexical entries are converted on demand at parse time, so a change to the template file does not require the lexicon to be reloaded.