COMPOSE-REDUCE PARSING

Henry S. Thompson[1]
Mike Dixon[2]
John Lamping[2]
1: Human Communication Research Centre
University of Edinburgh
2 Buccleuch Place
Edinburgh EH8 9LW
SCOTLAND
2: Xerox Palo Alto Research Center
3333 Coyote Hill Road
Palo Alto, CA 94304

ABSTRACT

Two new parsing algorithms for context-free phrase structure grammars are presented which perform a bounded amount of processing per word per analysis path, independently of sentence length. They are thus capable of parsing in real-time in a parallel implementation which forks processors in response to non-deterministic choice points.

0. INTRODUCTION

The work reported here grew out of our attempt to improve on the $O(n^2)$ performance of the SIMD parallel parser described in (Thompson 1991). Rather than start with a commitment to a specific SIMD architecture, as that work had, we agreed that the best place to start was with a more abstract architecture-independent consideration of the CF-PSG parsing problem—given arbitrary resources, what algorithms could one envisage which could recognise and/or parse atomic category phrase-structure grammars in $O(n)$? In the end, two quite different approaches emerged. One took as its starting point non-deterministic shift-reduce parsing, and sought to achieve linear (indeed real-time) complexity by performing a constant-time step per word of the input. The other took as its starting point tabular parsing (Earley, CKY), and sought to achieve linear complexity by performing a constant-time step for the identification/construction of constituents of each length from 0 to n. The latter route has been widely canvassed, although to our knowledge has not yet been implemented—see (Nijholt 1989, 90) for extensive references. The former route, whereby real-time parsing is achieved by processor forking at non-deterministic choice points in an extended shift-reduce parser, is to our knowledge new. In this paper we present outlines of two such parsers, which we call *compose-reduce* parsers.

I. COMPOSE-REDUCE PARSING

Why couldn't a simple breadth-first chart parser achieve linear performance on an appropriate parallel system? If you provided enough processors to immediately process all agenda entries as they were created, would not this give the desired result? No, because the processing of a single word might require many serialised steps. Consider processing the word "park" in the sentence "The people who ran in the park got wet." Given a simple traditional sort of grammar, that word completes an NP, which in turn completes a PP, which in turn completes a VP, which in turn completes an S, which in turn completes a REL, which in turn completes an NP. The construction/recognition of these constituents is necessarily serialised, so regardless of the number of processors available a constant-time step is impossible. (Note that this only precludes a real-time parse by this route, but not necessarily a linear one.) In the shift-reduce approach to parsing, all this means is that for non-linear grammars, a single shift step may be followed by many reduce steps. This in turn suggested the beginnings of a way out, based on categorial grammar, namely that multiple reduces can be avoided if composition is allowed. To return to our example above, in a simple shift-reduce parser we would have had all the words preceding the word "park" in the stack. When it was shifted in, there would follow six reduce steps. If alternatively following a shift step one was allowed (non-deterministically) a compose step, this could be reduced (!) to a single reduce step. Restricting ourselves to a simpler example, consider just "run in the park" as a VP, given rules

VP → v PP

NP → d n

PP → p NP.

With a composition step allowed, the parse would then proceed as follows:

Shift *run* as a v

Shift *in* as a p

Compose v and p to give

[$_{VP}$ v [PP p • NP]]

where I use a combination of bracketed strings and the 'dotted rule' notation to indicate the result of composition. The categorial equivalent would have been to notate V as VP/PP, P as PP/NP, and the result of the composition as therefore VP/NP.

Shift *the* as d

Compose the dotted VP with d to give

[$_{VP}$ v [PP p [$_{NP}$ d • n]]]

Shift *park* as n

Reduce the dotted VP with n to give the complete result.

Although a number of details remained to be worked out, this simple move of allowing composition was the enabling step to achieving O(n) parsing. Parallelism would arise by forking processors at each non-deterministic choice point, following the general model of Dixon's earlier work on parallelising the ATMS (Dixon & de Kleer 1988).

Simply allowing composition is not in itself sufficient to achieve O(n) performance. Some means of guaranteeing that each step is constant time must still be provided. Here we found two different ways forward.

II. The First Compose-Reduce Parser—CR-I

In this parser there is no stack. We have simply a current structure, which corresponds to the top node of the stack in a normal shift-reduce parser. This is achieved by extending the appeal to composition to include a form of left-embedded raising, which will be discussed further below. Special attention is also required to handle left-recursive rules.

II.1 The Basic Parsing Algorithm

The constant-time parsing step is given below (slightly simplified, in that empty productions and some unit productions are not handled). In this algorithm schema, and in subsequent discussion, the annotation "ND" will be used in situations where a number of alternatives are (or may be) described. The meaning is that these alternatives are to be pursued non-deterministically.

```
Algorithm CR-I
  1  Shift the next word;
  2  ND look it up in the lexicon;
  3  ND close the resulting category wrt the unit productions;
  4a ND reduce the resulting category with the current structure
     or
  4b ND raise* the resulting category wrt the non-unary rules in the grammar for
     which it is a left corner, and compose the result with the current structure.
  If reduction ever completes a category which is marked as the left corner of one
     or more left-recursive rules or rule sequences, ND raise* in place wrt those
     rules (sequences), and propagate the marking.
```

Some of these ND steps may at various points produce complete structures. If the input is exhausted, then those structures are parses, or not, depending on whether or not they have reached the distinguished symbol. If the input is not exhausted, it is of course the incomplete structures, the results of composition or raising, which are carried forward to the next step.

The operation referred to above as "raise*" is more than simple raising, as was involved in the simple example in section IV. In order to allow for all possible compositions to take place all possible left-embedded raising must be pursued. Consider the following grammar fragment:

```
S → NP VP
VP → v NP CMP
CMP → that S
NP → propn
NP → d n
```

and the utterance "Kim told Robin that the child likes Kim".

If we ignore all the ND incorrect paths, the current structure after the "that" has been processed is

$[_S\ [_{NP}\ [_{propn}\ \text{Kim}]]$
$\quad [_{VP}\ [_v\ \text{told}]$
$\qquad [_{NP}\ [_{propn}\ \text{Robin}]]$
$\qquad [\text{CMP that} \bullet \text{S}]]]$

In order for the next word, "the", to be correctly processed, it must be raised all the way to S, namely we must have

$[_S\ [_{NP}\ [_d\ \text{the}] \bullet \text{n}]\ \text{VP}]]$

to compose with the current structure. What this means is that for every entry in the normal bottom-up reachability table pairing a left corner with a top category, we need a set of dotted structures, corresponding to all the ways the grammar can get from that left corner to that top category. It is these structures which are ND made available in step 4b of the parsing step algorithm CR-I above.

II.2 Handling Left Recursion

Now this in itself is not sufficient to handle left recursive structures, since by definition there could be an arbitrary number of left-embeddings of a left-recursive structure. The final note in the description of algorithm CR-I above is designed to handle this. Glossing over some subtleties, left-recursion is handled by marking some of the structures introduced in step 3b, and ND raising in place if the marked structure is ever completed by reduction in the course of a parse. Consider the sentence "Robin likes the child 's dog." We add the following two rules to the grammar:

```
D → art
D → NP 's
```

thereby transforming D from a pre-terminal to a non-terminal. When we shift "the", we will raise to *inter alia*

$$[_{NP}\ [_{D}\ [_{art}\ \text{the}]]\ \bullet\ n]^{r}$$

with the NP marked for potential re-raising. This structure will be composed with the then current structure to produce

$$[_{S}\ [_{NP}\ [_{propn}\ \text{Robin}]]\ [_{VP}\ [_{v}\ \text{likes}]\ [_{NP}\ (\text{as above})]^{r}]]$$

After reduction with "child", we will have

$$[_{S}\ [_{NP}\ [_{propn}\ \text{Robin}]]\ [_{VP}\ [_{v}\ \text{likes}]\ [_{NP}\ [_{D}\ [_{art}\ \text{the}]]\ [_{n}\ \text{child}]]^{r}]]$$

The last reduction will have completed the marked NP introduced above, so we ND left-recursively raise in place, giving

$$[_{S}\ [_{NP}\ [_{propn}\ \text{Robin}]]\ [_{VP}\ [_{v}\ \text{likes}]\ [_{NP}\ [_{D}\ [_{NP}\ \text{the child}]\ \bullet\ \text{'s}]\ n]^{r}]]$$

which will then take us through the rest of the sentence.

One final detail needs to be cleared up. Although directly left-recursive rules, such as e.g. NP → NP PP, are correctly dealt with by the above mechanism, indirectly left-recursive sets of rules, such as the one exemplified above, require one additional subtlety. Care must be taken not to introduce the potential for spurious ambiguity. We will introduce the full details in the next section.

II.3 Nature of the required tables

Steps 3 and 4b of CR-I require tables of partial structures: Closures of unit productions up from pre-terminals, for step 3; left-reachable raisings up from (unit production closures of) pre-terminals, for step 4b. In this section we discuss the creation of the necessary tables, in particular Raise*, against the background of a simple exemplary grammar, given below as Table 1.

We have grouped the rules according to type—two kinds of unit productions (from pre-terminals or non-terminals), two kinds of left recursive rules (direct and indirect) and the remainder.

vanilla	$unit_1$	$unit_2$	lr_d	lr_i
S → NP VP	NP → propn	NP → CMP	NP → NP PP	NP → D n
VP → v NP	D → art		VP → VP PP	D → NP 's
CMP → cmp S				
PP → prep NP				

Table 1. Exemplary grammar in groups by rule type

Cl*	[$_{NP}$ propn]^{1,2} [$_{D}$ art]^{4} [$_{NP}$ CMP]^{1,2}
LR$_{dir}$	1: [$_{NP}$ NP PP] 3: [$_{VP}$ VP PP]
LR$_{indir}$2	2: [$_{NP}$ [$_{D}$ NP 's] n] 4: [$_{D}$ [$_{NP}$ D n]^{1} 's]
Rs*	[$_{CMP}$ cmp S], [$_{NP}$ [$_{CMP}$ cmp S]]^{1,2},
	[$_{D}$ [$_{NP}$ [$_{CMP}$ cmp S]]^{1,2} 's],
	[$_{S}$ [$_{NP}$ [$_{CMP}$ cmp S]]^{1,2} VP]
	[$_{PP}$ prep NP]
	[$_{VP}$ v NP]^{3}
	[$_{NP}$ D n]^{1,2}, [$_{S}$ [$_{NP}$ D n]^{1,2} VP]
	[$_{D}$ NP^{1} 's]^{4}, [$_{S}$ NP^{1,2} VP]

Table 2. Partial structures for CR-I

Ras*	[$_{NP}$ $\underline{[_{NP}\ \text{propn}]}$ • PP]^{1,2}, [$_{NP}$ [$_{D}$ $\underline{[_{NP}\ \text{propn}]}$ • 's] n]^{1,2}
	[$_{D}$ [$_{NP}$ $\underline{[_{D}\ \text{art}]}$ • n]^{1} 's]^{4}
	[$_{CMP}$ cmp • S], [$_{NP}$ [$_{CMP}$ cmp • S]]^{1,2},
	[$_{D}$ [$_{NP}$ [$_{CMP}$ cmp • S]]^{1,2} 's],
	[$_{S}$ [$_{NP}$ [$_{CMP}$ cmp • S]]^{1,2} VP]
	[$_{PP}$ prep • NP]
	[$_{VP}$ v • NP]^{3}
	[$_{NP}$ $\underline{[_{D}\ \text{art}]}$ • n]^{1,2}, [$_{S}$ [$_{NP}$ $\underline{[_{D}\ \text{art}]}$ • n]^{1,2} VP]
	[$_{D}$ $\underline{[_{NP}\ \text{propn}]}^{1}$ • 's]^{4}, [$_{S}$ $\underline{[_{NP}\ \text{propn}]}^{1,2}$ • VP]

Table 3. Projecting non-terminal left daughters

As a first step towards computing the table which step 4b above would use, we can pre-compute the partial structures given above in Table 2.

Cl* contains all backbone fragments constructable from the unit productions, and is already essentially what we require for step 3 of the algorithm. LR$_{dir}$ contains all directly left-recursive structures. LR$_{indir}$2 contains all indirectly left-recursive structures involving exactly two rules, and there might be LR$_{indir}$3, 4,... as well. Rs* contains all non-recursive tree fragments constructable from left-embedding of binary or greater rules and non-terminal unit productions. The superscripts denote loci where left-recursion may be appropriate, and identify the relevant structures.

In order to get the full Raise* table needed for step 4b, first we need to project the non-terminal left daughters of rules such as [$_{S}$ NP^{1,2} VP] down to terminal left daughters. We achieve this by substituting terminal entries from Cl* wherever we can in LR$_{dir}$, LR$_{indir}$2 and Rs* to give us Table 3 from Table 2 (new embeddings are underlined).

Left recursion has one remaining problem for us. Algorithm CR-I only checks for annotations and ND raises in place after a reduction completes a constituent. But in the last line of Ras* above there are unit constituents

[$_{NP}$ [$_{NP}$ propn] • PP]^{1,2}, [$_{NP}$ [$_{D}$ [$_{NP}$ propn] • 's] n]^{1,2}
[$_{D}$ [$_{NP}$ [$_{D}$ art] • n]^{1} 's]^{4}
[$_{CMP}$ cmp • S], [$_{NP}$ [$_{CMP}$ cmp • S]]^{1,2},
[$_{D}$ [$_{NP}$ [$_{CMP}$ cmp • S]]^{1,2} 's],
[$_{S}$ [$_{NP}$ [$_{CMP}$ cmp • S]]^{1,2} VP]
[$_{PP}$ prep • NP]
[$_{VP}$ v • NP]^{3}
[$_{NP}$ [$_{D}$ art] • n]^{1,2}, [$_{S}$ [$_{NP}$ [$_{D}$ art] • n]^{1,2} VP]
[$_{D}$ [$_{NP}$ propn] • 's]^{4}, [$_{D}$ [$_{NP}$ [$_{NP}$ propn] • PP]^{1} 's]^{4}
[$_{S}$ [$_{NP}$ propn] • VP], [$_{S}$ [$_{NP}$ [$_{NP}$ propn] • PP]^{1,2} VP],
[$_{S}$ [$_{NP}$ [$_{D}$ [$_{NP}$ propn] • 's] n]^{1,2} VP]

Table 4. Final form of the structure table Raise*

with annotations. Being already complete, they will not ever be completed, and consequently the annotations will never be checked. So we pre-compute the desired result, augmenting the above list with expansions of those units via the indicated left recursions. This gives us the final version of Raise*, now shown with dots included, in Table 4.

This table is now suited to its role in the algorithm. Every entry has a lexical left daughter, all annotated constituents are incomplete, and all unit productions are factored in. It is interesting to note that with these tree fragments, taken together with the terminal entries in Cl*, as the initial trees and LRdir, LRindir2, etc. as the auxiliary trees we have a Tree Adjoining Grammar (Joshi 1985) which is strongly equivalent to the CF-PSG we started with. We might call it the left-lexical TAG for that CF-PSG, after Schabes et al. (1988). Note further that if a TAG parser respected the annotations as restricting adjunction, no spuriously ambiguous parses would be produced.

Indeed it was via this relationship with TAGs that the details were worked out of how the annotations are distributed, not presented here to conserve space.

II.4 Implementation and Efficiency

Only a serial pseudo-parallel implementation has been written. Because of the high degree of precomputation of structure, this version even though serialised runs quite efficiently. There is very little computation at each step, as it is straight-forward to double index the Raise* table so that only structures which will compose with the current structure are retrieved.

The price one pays for this efficiency, whether in serial or parallel versions, is that only left-common structure is shared. Right-common structure, as for instance in PP attachment ambiguity, is not shared between analysis paths. This causes no difficulties for the parallel approach in one sense, in that it does not compromise the real-time performance of the parser. Indeed, it is precisely because no recombination is attempted that the basic parsing step is constant time. But it does mean that if the CF-PSG being parsed is the first half of a two step process, in which additional constraints are solved in the second pass, then the duplication of structure will give rise to duplication of effort. Any parallel parser which adopts the strategy of forking at non-deterministic choice points will suffer from this weakness, including CR-II below.

III. The Second Compose-Reduce Parser—CR-II

Our second approach to compose-reduce parsing differs from the first in retaining a stack, having a more complex basic parsing step, while requiring far less pre-processing of the grammar. In particular, no special treatment is required for left-recursive rules. Nevertheless, the basic step is still constant time, and despite the stack there is no potential processing 'balloon' at the end of the input.

III.1 The Basic Parsing Algorithm

```
Algorithm CR-II
  1  Shift the next word;
  2  ND look it up in the lexicon;
  3  ND close the resulting category wrt the unit productions;
  4  ND reduce the resulting category with the top of the
     stack—if results are complete and there is input remaining, pop the stack;
  5a ND raise the results of (2), (3) and, where complete, (4) and
  5b ND either push the result onto the stack
     or
  5c ND compose the result with the top of the stack, replacing it.
```

This is not an easy algorithm to understand. In the next section we present a number of different ways of motivating it, together with an illustrative example.

III.2 CR-II Explained

Let us first consider how CR-II will operate on purely left-branching and purely right-branching structures. In each case we will consider the sequence of algorithm steps along the non-deterministically correct path, ignoring the others. We will also restrict ourselves to considering binary branching rules, as pre-terminal unit productions are handled entirely by step 3 of the algorithm, and non-terminal unit productions must be factored into the grammar. On the other hand, interior daughters of non-binary nodes are all handled by step 4 without changing the depth of the stack.

III.2.1 Left-branching analysis

For a purely left-branching structure, the first word will be processed by steps 1, 2, 5a and 5b, producing a stack with one entry which we can schematise as in Figure 1, where filled circles are processed nodes and unfilled ones are waiting.

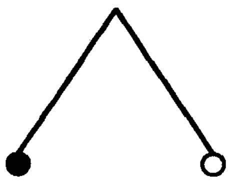

Figure 1.

All subsequent words except the last will be processed by steps 4, 5a and 5b (here and subsequently we will not mention steps 1 and 2, which occur for all words), effectively replacing the previous sole entry in the stack with the one given in Figure 2.

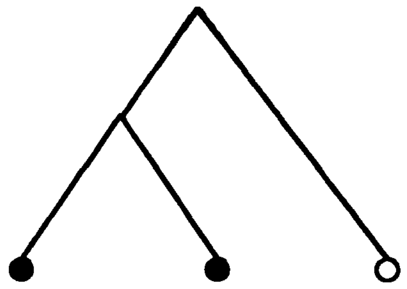

Figure 2.

It should be evident that the cycle of steps 4, 5a and 5b constructs a left-branching structure of increasing depth as the sole stack entry, with one right daughter, of the top node, waiting to be filled. The last input word of course is simply processed by step 4 and, as there is no further input, left on the stack as the final result. The complete sequence of steps for any left-branching analysis is thus raise—reduce&raise*—reduce. An ordinary shift-reduce or left-corner parser would go through the same sequence of steps.

III.2.2 Right-branching analysis

The first word of a purely right-branching structure is analysed exactly as for a left-branching one, that is, with 5a and 5b, with results as in Figure 1 (repeated here as Figure 3):

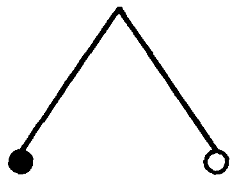

Figure 3.

Subsequent words, except the last, are processed via steps 5a and 5c, with the result remaining as the sole stack entry, as in Figure 4.

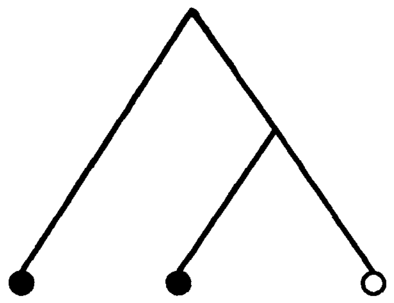

Figure 4.

Again it should be evident that cycling steps 5a and 5c will construct a right-branching structure of increasing depth as the sole stack entry, with one right daughter, of the most embedded node, waiting to be filled. Again, the last input word will be processed by step 4. The complete sequence of steps for any right-branching analysis is thus raise—raise&compose*—reduce. A categorial grammar parser with a compose-first strategy would go through an isomorphic sequence of steps.

III.2.3 Mixed Left- and Right-branching Analysis

All the steps in algorithm CR-II have now been illustrated, but we have yet to see the stack grow beyond one entry. This will occur in where an individual word, as opposed to a completed complex constituent, is processed by steps 5a and 5b, that is, where steps 5a and 5b apply other than to the results of step 4.

Consider for instance the sentence "the child believes that the dog likes biscuits." With a grammar which I trust will be obvious, we would arrive at the structure shown in Figure 5 after processing "the child believes that", having done raise—reduce&raise—raise&compose—raise&compose, that is, a bit of left-branching analysis, followed by a bit of right-branching analysis.

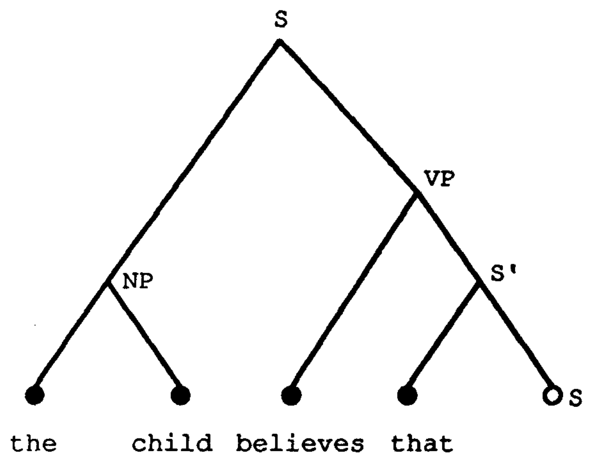

Figure 5.

Clearly there is nothing to be done with "the" which will allow immediate integration with this. The ND correct path applies steps 5a and 5b, raise&push, giving a stack as shown in Figure 6:

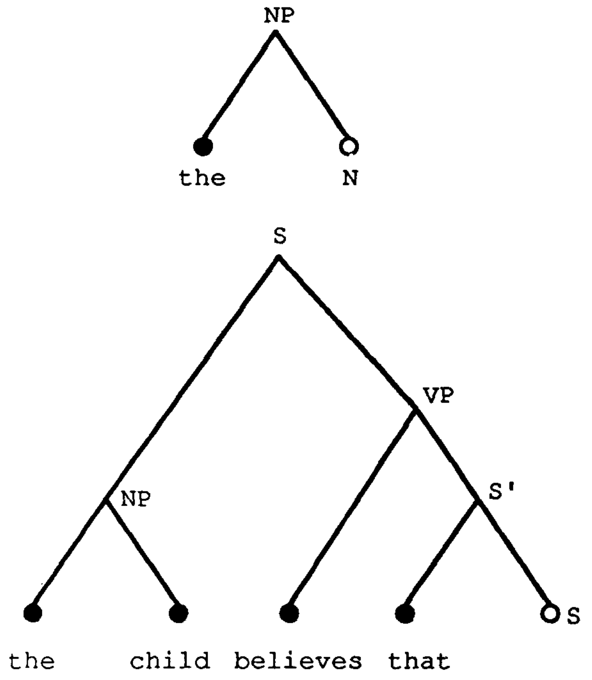

Figure 6.

We can then apply steps 4, 5a and 5c, reduce&raise&compose, to "dog", with the result shown in Figure 7. This puts uss back on the standard right-branching path for the rest of the sentence.

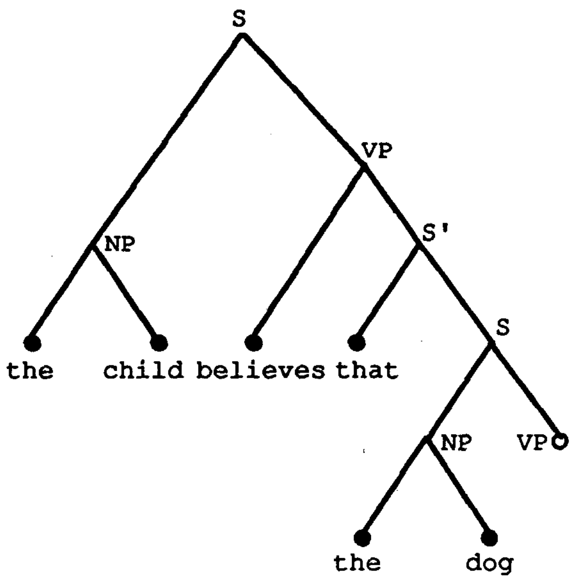

Figure 7.

III.3 An Alternative View of CR-II

Returning to a question raised earlier, we can now see how a chart parser could be modified in order to run in real-time given enough processors to empty the agenda as fast as it is filled. We can reproduce the processing of CR-II within the active chart parsing framework by two modifications to the fundamental rule (see e.g. Gazdar and Mellish 1989 or Thompson and Ritchie 1984 for a tutorial introduction to active chart parsing). First we restrict its normal operation, in which an active and an inactive edge are combined, to apply only in the case of pre-terminal inactive edges. This corresponds to the fact that in CR-II step 4, the reduction step, applies only to pre-terminal categories (continuing to ignore unit productions). Secondly we allow the fundamental rule to combine two active edges, provided the category to be produced by one is what is required by the other. This effects composition. If we now run our chart parser left-to-right, left-corner and breadth-first, it will duplicate CR-II.

The maximum number of edges along a given analysis path which can be introduced by the processing of a single word is now at most four, corresponding to steps 2, 4, 5a and 5c of CR-II—the pre-terminal itself, a constituent completed by it, an active edge containing that constituent as left daughter, created by left-corner rule invocation, and a further active edge combining that one with one to its left. This in turn means that there is a fixed limit to the amount of processing required for each word.

III.4 Implementation and Efficiency

Although clearly not benefiting from as much pre-computation of structure as CR-I, CR-II is also quite efficient. Two modifications can be added to improve efficiency—a reachability filter on step 5b, and a shaper test (Kuno 1965), also on 5b. For the latter, we need simply keep a count of the number of open nodes on the stack (equal to the number of stack entries if all rules are binary), and ensure that this number never exceeds the number of words remaining in the input, as each entry will require a number of words equal to the number of its open nodes to pop it off the stack. This test actually cuts down the number of non-deterministic paths quite dramatically, as the ND optionality of step 5b means that quite deep stacks would otherwise be pursued along some search paths. Again this reduction in search space is of limited significance in a true parallel implementation, but in the serial simulation it makes a big difference.

Note also that no attention has been paid to unit productions, which we pre-compute as in CR-I. Furthermore, neither CR-I nor CR-II address empty productions, whose effect would also need to be pre-computed.

IV. Conclusions

Aside from the intrinsic interest in the abstract of real-time parsablility, is there any practical significance to these results. Two drawbacks, one already referred to, certainly restrict their significance. One is that the restriction to atomic category CF-PSGs is crucial—the fact that the comparison between a rule element and a node label is atomic and constant time is fundamental. Any move to features or other annotations would put an end to real-time processing. This fact gives added weight to the problem mentioned above in section II.4, that only left-common analysis results are shared between alternatives. Thus if one finesses the atomic category problem by using a parser such as those described here only as the first pass of a two pass system, one is only putting off the payment of the complexity price to the second pass, in the absence to date of any linear time solution to the constraint satisfaction problem. On this basis, one would clearly prefer a parallel CKY/Earley algorithm, which does share all common substructure, to the parsers presented here.

Nevertheless, there is one class of applications where the left-to-right real-time behaviour of these algorithms may be of practical benefit, namely in speech recognition. Present day systems require on-line availability of syntactic and domain-semantic constraint to limit the search space at lower levels of the system. Hitherto this has meant these constraints must be brought to bear during recognition as some form of regular grammar, either explicitly constructed as such or compiled into. The parsers presented here offer the alternative of parallel application of genuinely context-free grammars directly, with the potential added benefit that, with sufficient processor width, quite high degrees of local ambiguity can be tolerated, such as would arise if (a finite subset of) a feature-based grammar were expanded out into atomic category form.

ACKNOWLEDGEMENTS

The work reported here was carried out while the first author was a visitor to the Embedded Computation and Natural Language Theory and Technology groups of the Systems Science Laboratory at the Xerox Palo Alto Research Center. These groups provided both the intellectual and material resources required to support our work, for which our thanks.

REFERENCES

Dixon, Mike and de Kleer, Johan. 1988. "Massively Parallel Assumption-based Truth Maintenance". In *Proceedings of the AAAI-88 National Conference on Artificial Intelligence*, also reprinted in *Proceedings of the Second International Workshop on Non-Monotonic Reasoning.*

Gazdar, Gerald and Mellish, Chris. 1989. *Natural Language Processing in LISP*. Addison-Wesley, Wokingham, England (sic).

Joshi, Aravind K. 1985. "How Much Context-Sensitivity is Necessary for Characterizing Structural Descriptions—Tree Adjoining Grammars". In Dowty, D., Karttunen, L., and Zwicky, A. eds, *Natural Language Processing—Theoretical Computational and Psychological Perspectives.* Cambridge University Press, New York.

Kuno, Susumo. 1965. "The predictive analyzer and a path elimination technique", *Communications of the ACM*, **8**, 687-698.

Nijholt, Anton. 1989. "Parallel parsing strategies in natural language processing". In Tomita, M. ed, *Proceedings of the International Workshop on Parsing Technologies*, 240-253, Carnegie-Mellon University, Pittsburgh.

Nijholt, Anton. 1990. *The CYK-Approach to Serial and Parallel Parsing*. Memoranda Informatica 90-13, faculteit der informatica, Universiteit Twente, Netherlands.

Shabes, Yves, Abeillé, Anne and Joshi, Aravind K. 1988. "Parsing Strategies with 'Lexicalized' Grammars: Application to Tree Adjoining Grammars". In *Proceedings of the 12th International Conference on Computational Linguistics*, 82-93.

Thompson, Henry S. 1991. "Parallel Parsers for Context-Free Grammars—Two Actual Implementations Compared". To appear in Adriaens, G. and Hahn, U. eds, *Parallel Models of Natural Language Computation*, Ablex, Norword NJ.

Thompson, Henry S. and Ritchie, Graeme D. 1984. "Techniques for Parsing Natural Language: Two Examples". In Eisenstadt, M., and O'Shea, T., editors, *Artificial Intelligence: Tools, Techniques, and Applications*. Harper and Row, London. Also DAI Research Paper 183, Dept. of Artificial Intelligence, Univ. of Edinburgh.